FreeRTOS

A real time operating system for embedded systems

RESOURCE MANAGEMENT

Resource Management

Topics covered:

- When and why resource management and control is necessary.
- What a critical section is.
- What mutual exclusion means.
- What it means to suspend the scheduler.
- How to use a mutex.
- How to create and use a gatekeeper task.
- What priority inversion is, and how priority inheritance can reduce (but not remove) its impact.

Critical Sections and Suspending the Scheduler

- Basic critical sections are regions of code that are surrounded by calls to the macros taskENTER_CRITICAL() and taskEXIT_CRITICAL(), respectively. Critical sections are also known as critical regions.

- Critical sections implemented in this way are a very crude method of providing mutual exclusion. They work by disabling interrupts, either completely, or up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY, depending on the FreeRTOS port being used.

- Pre-emptive context switches can occur only from within an interrupt, so, as long as interrupts remain disabled, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.

- Basic critical sections must be kept very short, otherwise they will adversely affect interrupt response times.

Critical Sections

```
void vPrintString( const char *pcString )
{
   /* Write the string to stdout, using a critical section as a crude method of
      mutual exclusion. */
   taskENTER_CRITICAL();
   {
      printf( "%s", pcString );
      fflush( stdout );
   }
   taskEXIT_CRITICAL();
}
```

Suspending the Scheduler

- Critical sections can also be created by suspending the scheduler. Suspending the scheduler is sometimes also known as 'locking' the scheduler.

- Basic critical sections protect a region of code from access by other tasks and by interrupts, but a critical section implemented by suspending the scheduler only protects a region of code from access by other tasks, because interrupts remain enabled.

- A critical section that is too long to be implemented by simply disabling interrupts can, instead, be implemented by suspending the scheduler. However, interrupt activity while the scheduler is suspended can make resuming (or 'un-suspending') the scheduler a relatively long operation, so consideration must be given to which is the best method to use in each case.

Suspending the Scheduler

```
void vPrintString( const char *pcString )
{
   /* Write the string to stdout, suspending the scheduler as a method of
      mutual exclusion. */
   vTaskSuspendScheduler();
   {
      printf( "%s", pcString );
      fflush( stdout );
   }
   xTaskResumeScheduler();
}
```

API functions

vTaskSuspendAll()

- The scheduler is suspended by calling vTaskSuspendAll().
- Suspending the scheduler prevents a context switch from occurring, but leaves interrupts enabled.
- If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending, and is performed only when the scheduler is resumed (un-suspended).

xTaskResumeAll()

- The scheduler is resumed (un-suspended) by calling xTaskResumeAll().
- Return value:

Context switches that are requested while the scheduler is suspended are held pending and performed only as the scheduler is being resumed. If a pending context switch is performed before xTaskResumeAll() returns, then pdTRUE is returned. Otherwise pdFALSE is returned.

Mutex

- A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks.
- The word MUTEX originates from 'MUTual EXclusion'. configUSE_MUTEXES must be set to 1 in FreeRTOSConfig.h for mutexes to be available.
- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource. A task is not permitted to access the shared resource unless it holds the token.

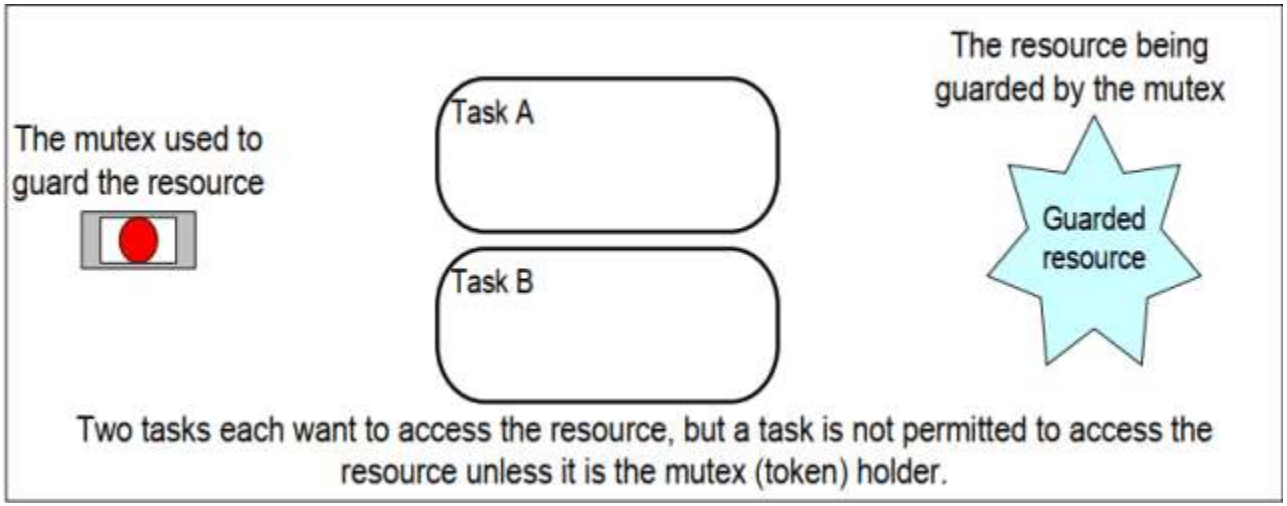
Mutex vs Binary Semaphore

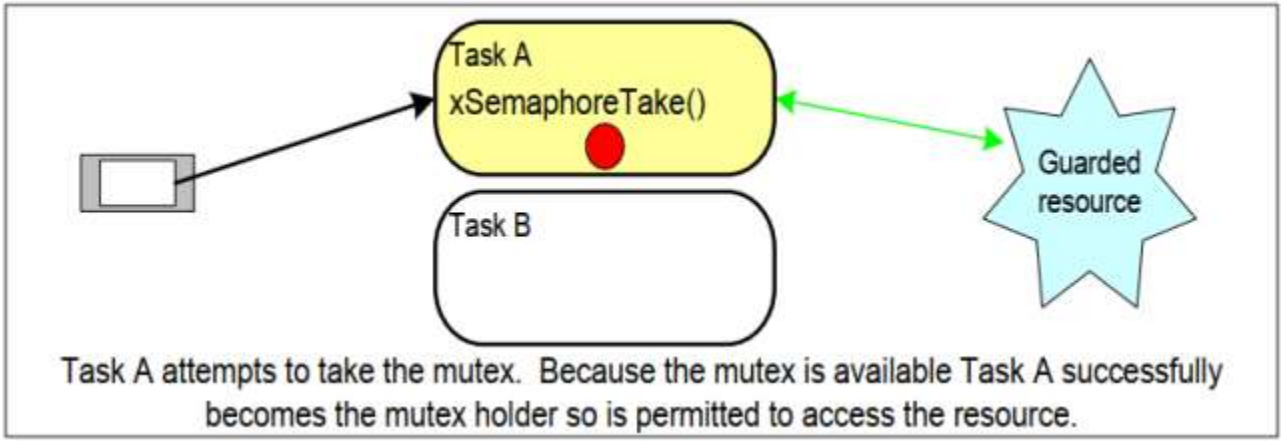
- Even though mutexes and binary semaphores share many characteristics, the scenario where a mutex is used for mutual exclusion) is completely different to that where a binary semaphore is used for synchronization.

The primary difference is what happens to the semaphore after it has been obtained:

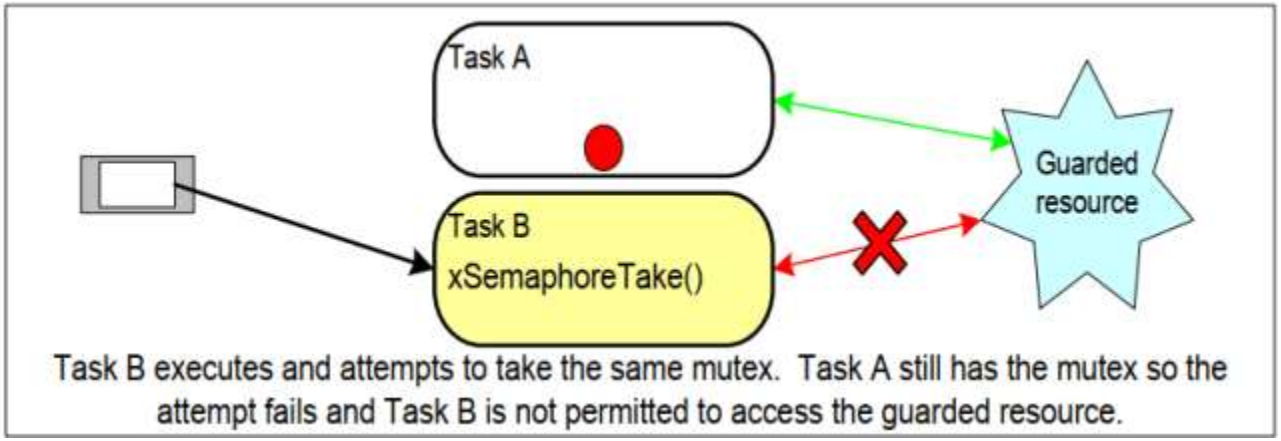
- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.

Mutual exclusion implemented using a mutex

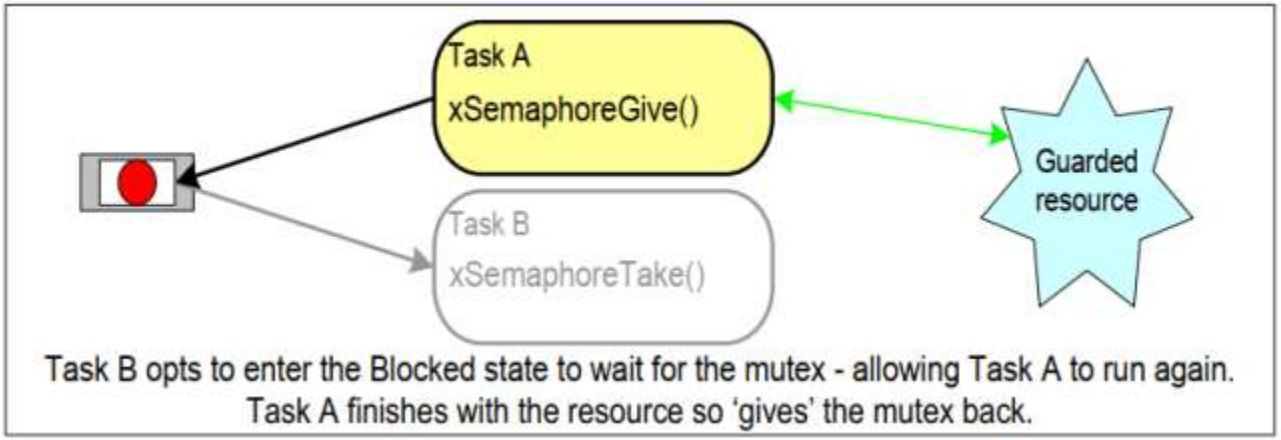




Mutual exclusion implemented using a mutex

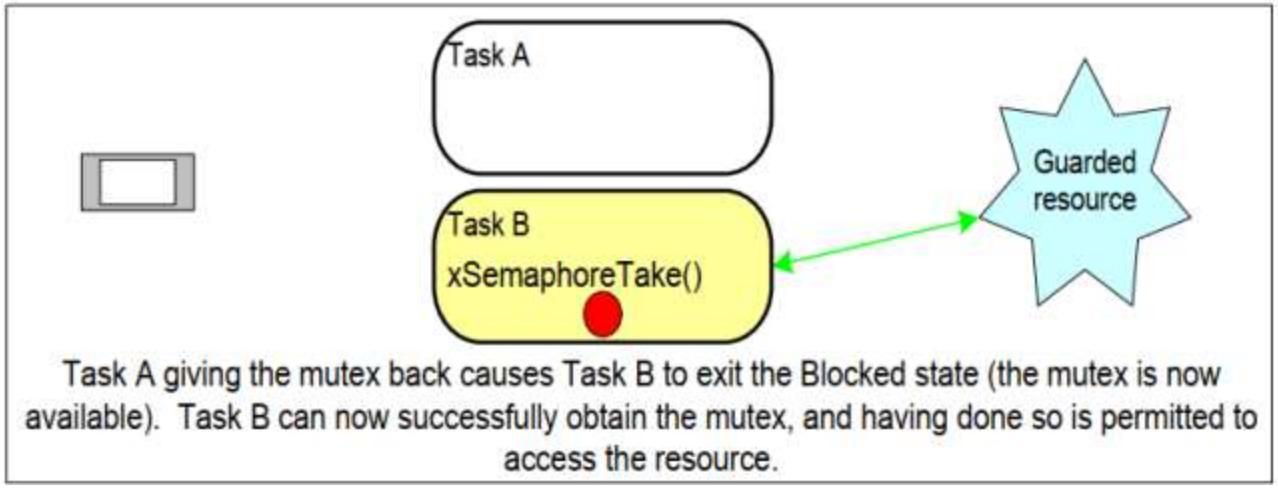


Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.

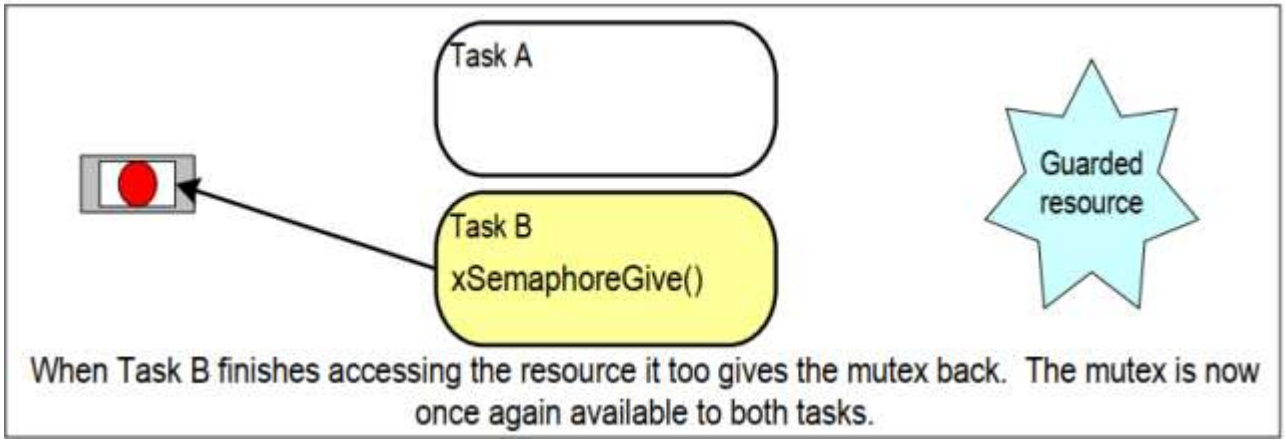


Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.

Mutual exclusion implemented using a mutex



Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

Create a Mutex

Before a mutex can be used, it must be created. Use the **`xSemaphoreCreateMutex()`** API function to create a mutex type semaphore.

```
xSemaphoreHandle_t myMutex;
      myMutex = xSemaphoreCreateMutex();
```

❖ Return value:

- If NULL is returned, then the mutex could not be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures.
- A non-NULL return value indicates that the mutex has been created successfully. The returned value should be stored as the handle to the created mutex.

‘Take’ a Semaphore

- Use **`xSemaphoreTake()`** to ‘take’ a semaphore

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t   xSemaphore,
                          TickType_t   xTicksToWait );
```

- `xSemaphore`
  - A handle to the semaphore being ‘taken’
- `xTicksToWait`
  - The maximum amount of time the task should remain in the Blocked state
  - Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely
- Return value
  - `pdTRUE` will only be returned if it was successful in obtaining the semaphore
  - `pdFALSE` if the semaphore was not available

‘Give’ a Semaphore

- Use **`xSemaphoreGive()`** (**`xSemaphoreGiveFromISR()`**) to ‘give’ a semaphore (when in an ISR)

```
BaseType_t    xSemaphoreGiveFromISR(
                    xSemaphoreHandle xSemaphore,
                    BaseType_t    *pxHigherPriorityTaskWoken );
```

- `xSemaphore`
  A handle to the semaphore being ‘given’
- `pxHigherPriorityTaskWoken`
  If the handler task has a higher priority than the currently executing task (the task that was interrupted), this value will be set to `pdTRUE`
- Return value
  `pdPASS` will only be returned if successful
  `pdFAIL` if a semaphore is already available and cannot be given

Example 15. Rewriting vPrintString() to use a semaphore

```
static void prvNewPrintString( const char *pcString )
{
   /* The mutex is created before the scheduler is started, so already exists
      by the time this task executes.

*/
   xSemaphoreTake( xMutex, portMAX_DELAY );
   {
      /* The following line will only execute once the mutex has been
         successfully obtained. Standard out can be accessed freely now as
         only one task can have the mutex at any one time. */
      Serial.print(pcString);
      Serial.flush();
      /* The mutex MUST be given back! */
   }
   xSemaphoreGive( xMutex );
}
```

prvPrintTask()

```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;
    const TickType_t xMaxBlockTimeTicks = 0x20;

    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time. */
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

```
void setup( void)
{
Serial.begin(9600);
xMutex = xSemaphoreCreateMutex();

  /* The tasks are going to use a pseudo random delay, seed the random number
  generator. */
  srand( 567 );
 if( xMutex != NULL )
  {
   xTaskCreate( prvPrintTask, "Print1", 200, (void*)"Task 1
**********************************************\r\n", 1, NULL );
    xTaskCreate( prvPrintTask, "Print2", 200, (void*)"Task 2 ----------------------------------
\r\n", 2, NULL );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
  }
for( ;; );
}
```

Example015 | Arduino IDE 2.3.2

File Edit Sketch Tools Help

Arduino Mega or Meg...

Example015.ino

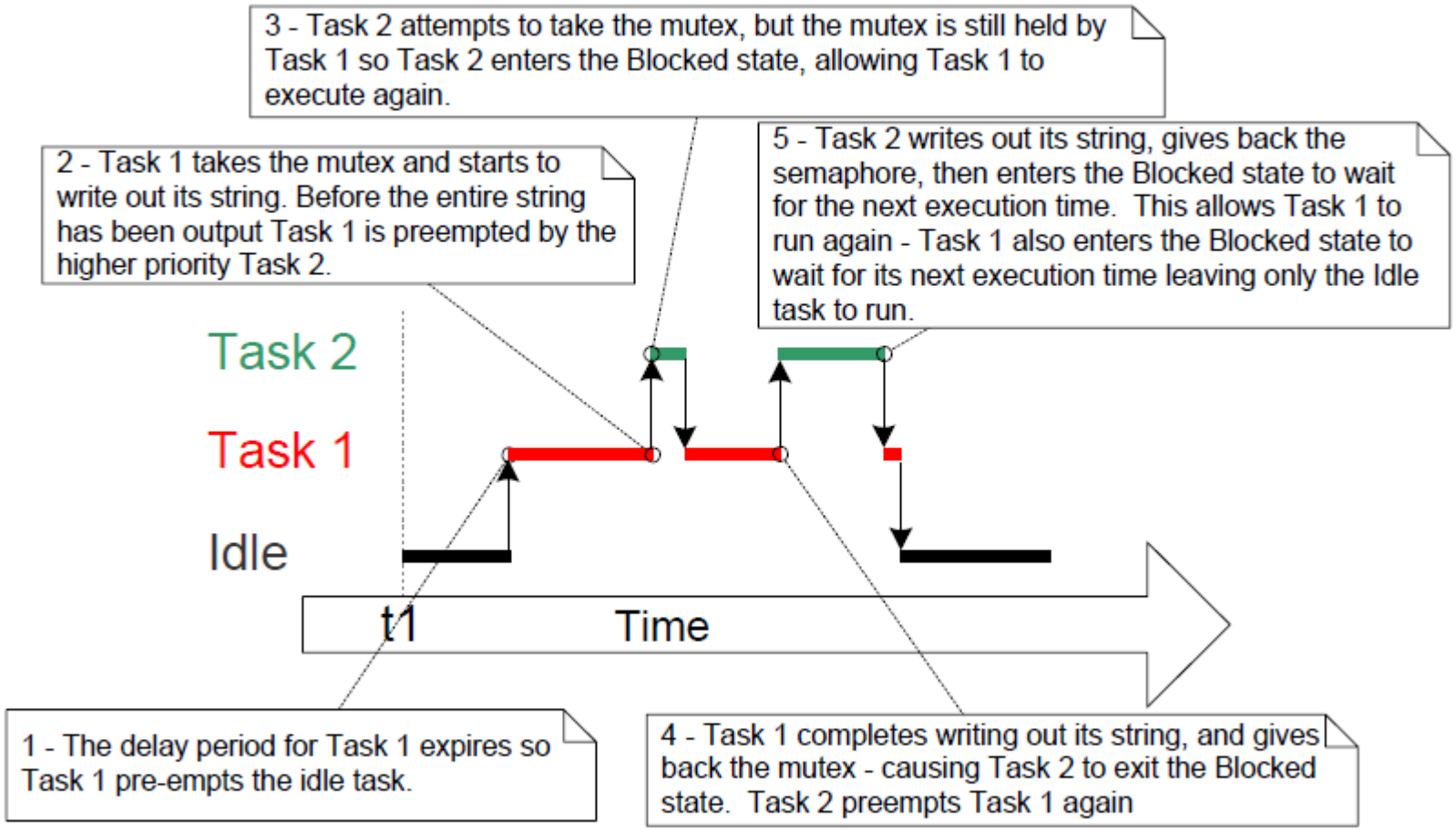
```
90  required type. */
```

Output Serial Monitor ×

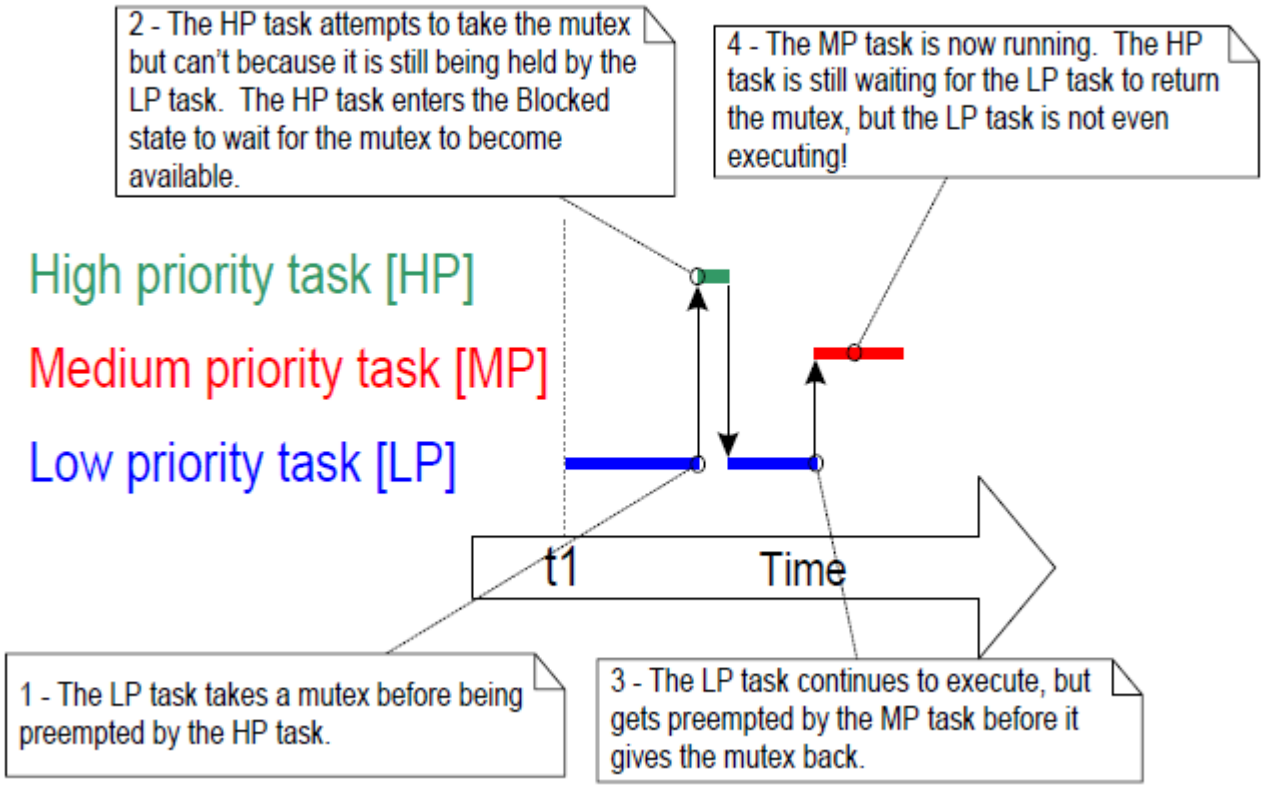
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM3')

```
12:30:51.148 -> Task 1 ********************************************
12:30:51.537 -> Task 2 --------------------------------------------
12:30:59.035 -> Task 2 --------------------------------------------
12:30:59.968 -> Task 1 ********************************************
12:31:01.675 -> Task 2 --------------------------------------------
12:31:06.500 -> Task 1 ********************************************
12:31:08.242 -> Task 2 --------------------------------------------
12:31:09.596 -> Task 1 ********************************************
12:31:10.563 -> Task 2 --------------------------------------------
12:31:15.071 -> Task 1 ********************************************
12:31:15.269 -> Task 2 --------------------------------------------
12:31:15.875 -> Task 1 ********************************************
12:31:16.519 -> Task 1 ********************************************
12:31:19.718 -> Task 2 --------------------------------------------
12:31:20.107 -> Task 1 ********************************************
12:31:21.014 -> Task 1 ********************************************
12:31:22.983 -> Task 1 ********************************************
```

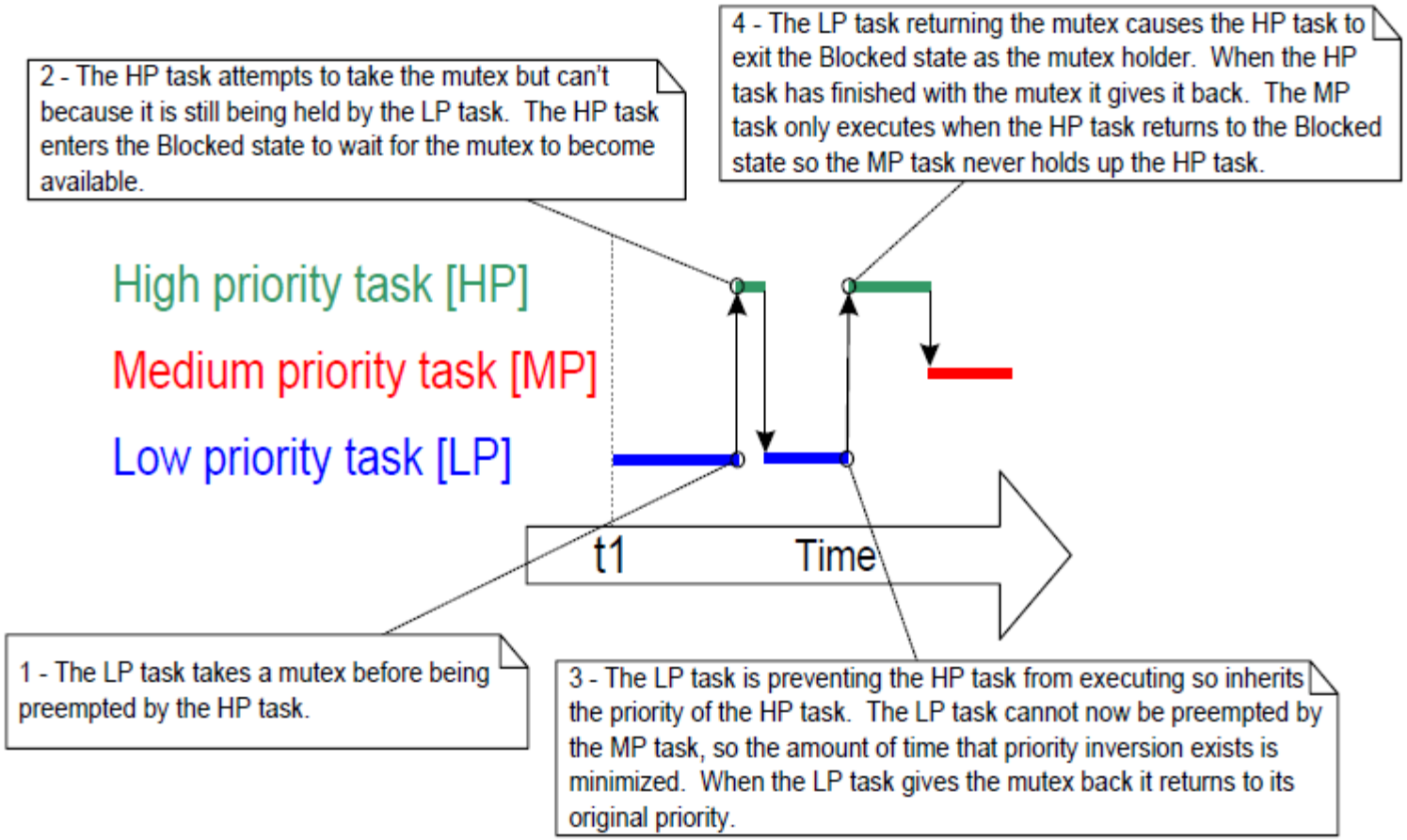
A possible sequence of execution



Priority Inversion



Priority Inheritance



Deadlock

Deadlock occurs when two tasks cannot proceed because they are both waiting for a resource that is held by the other. Consider the following scenario where Task A and Task B both need to acquire mutex X *and* mutex Y in order to perform an action:

- Task A executes and successfully takes mutex X.
- Task A is pre-empted by Task B.
- Task B successfully takes mutex Y before attempting to also take mutex X—but mutex X is held by Task A so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
- Task A continues executing. It attempts to take mutex Y—but mutex Y is held by Task B, so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released.

At the end of this scenario, Task A is waiting for a mutex held by Task B, and Task B is waiting for a mutex held by Task A. Deadlock has occurred because neither task can proceed.

Recursive Mutexes

It is also possible for a task to deadlock with itself. This will happen if a task attempts to take the same mutex more than once, without first returning the mutex. Consider the following scenario:

- A task successfully obtains a mutex.
- While holding the mutex, the task calls a library function.
- The implementation of the library function attempts to take the same mutex, and enters the Blocked state to wait for the mutex to become available.

At the end of this scenario the task is in the Blocked state to wait for the mutex to be returned, but the task is already the mutex holder. A deadlock has occurred because the task is in the Blocked state to wait for itself.

- This type of deadlock can be avoided by using a recursive mutex in place of a standard mutex. A recursive mutex can be 'taken' more than once by the same task, and will be returned only after one call to 'give' the recursive mutex has been executed for every preceding call to 'take' the recursive mutex.

Recursive mutexes

Standard mutexes and recursive mutexes are created and used in a similar way:

- Standard mutexes are created using xSemaphoreCreateMutex(). Recursive mutexes are created using xSemaphoreCreateRecursiveMutex(). The two API functions have the same prototype.

- Standard mutexes are 'taken' using xSemaphoreTake(). Recursive mutexes are 'taken' using xSemaphoreTakeRecursive(). The two API functions have the same prototype.

- Standard mutexes are 'given' using xSemaphoreGive(). Recursive mutexes are 'given' using xSemaphoreGiveRecursive(). The two API functions have the same prototype.

Gatekeeper Tasks

- Gatekeeper tasks provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.
- A gatekeeper task is a task that has sole ownership of a resource. Only the gatekeeper task is allowed to access the resource directly—any other task needing to access the resource can do so only indirectly by using the services of the gatekeeper.

Example 16

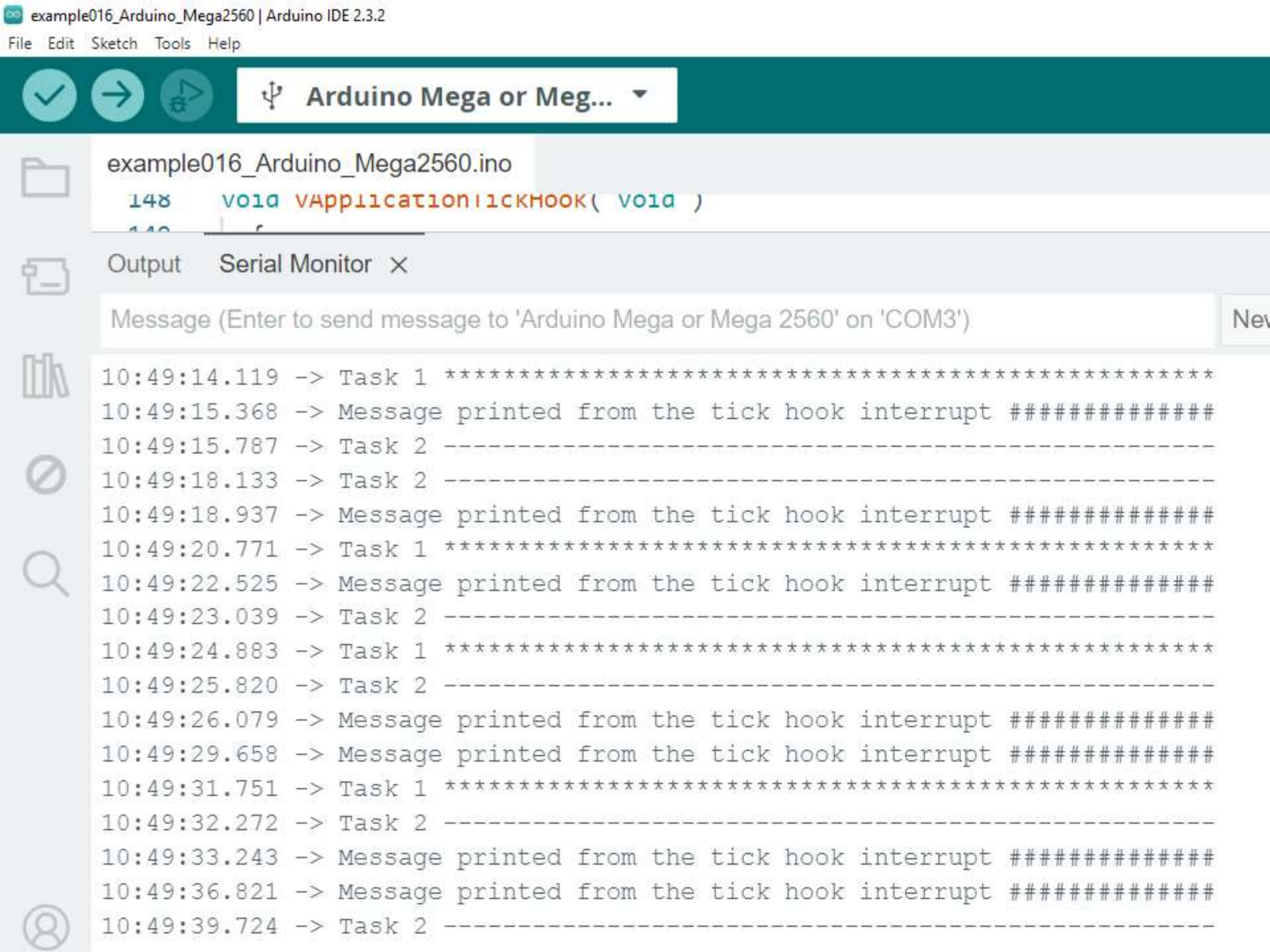
```
static void prvStdioGatekeeperTask( void *pvParameters )
{
   char *pcMessageToPrint;

for( ;; )
   {
      /* Wait for a message to arrive. An indefinite block time is specified
        so there is no need to check the return value – the function will
        only return when a message has been successfully received. */
      xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

      /* Output the received string. */
      Serial.print( pcMessageToPrint );
      Serial.flush( );

      /* Loop back to wait for the next message. */
   }
}
```

```
#include "Arduino_FreeRTOS.h"
#include "queue.h"

/* The task that sends messages to the stdio gatekeeper.  Two instances of this
task are created. */
static void prvPrintTask( void *pvParameters );

/* The gatekeeper task itself. */
static void prvStdioGatekeeperTask( void *pvParameters );

/* Define the strings that the tasks and interrupt will print out via the gatekeeper. */
static char *pcStringsToPrint[] =
{
  "Task 1 ****************************************************\r\n",
  "Task 2 ----------------------------------------------------\r\n",
  "Message printed from the tick hook interrupt ##############\r\n"
};
QueueHandle_t xPrintQueue;
```

```
void setup( void )
{
  Serial.begin(9600);
   xPrintQueue = xQueueCreate( 5, sizeof( char * ) );
  srand( 567 );

  if( xPrintQueue != NULL )
  {
    xTaskCreate( prvPrintTask, "Print1", 200, ( void * ) 0, 1, NULL );
    xTaskCreate( prvPrintTask, "Print2", 200, ( void * ) 1, 2, NULL );

    xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 200, NULL, 0, NULL );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
  }

    for( ;; );
}
```

```
static void prvPrintTask( void *pvParameters )
{
 int iIndexToString;

 iIndexToString = ( int ) pvParameters;

  for( ;; )
  {
    xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

    vTaskDelay( ( rand() & 0x1FF ) );
  }
}
```

```
void vApplicationTickHook( void )
 {
 static int iCount = 0;
 portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

 iCount++;
   if( iCount >= 200 )
   {
     /* In this case the last parameter (xHigherPriorityTaskWoken) is not
     actually used but must still be supplied. */
     xQueueSendToFrontFromISR( xPrintQueue, &( pcStringsToPrint[ 2 ] ),
(BaseType_t*)&xHigherPriorityTaskWoken );


     iCount = 0;
   }
 }
```


example016_Arduino_Mega2560 | Arduino IDE 2.3.2
File Edit Sketch Tools Help
Arduino Mega or Meg...
example016_Arduino_Mega2560.ino
148 void vApplicationTickHook(void)
Output Serial Monitor
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM3')
10:49:14.119 -> Task 1 **********
10:49:15.368 -> Message printed from the tick hook interrupt ##########
10:49:15.787 -> Task 2 ----------
10:49:18.133 -> Task 2 ----------
10:49:18.937 -> Message printed from the tick hook interrupt ##########
10:49:20.771 -> Task 1 **********
10:49:22.525 -> Message printed from the tick hook interrupt ##########
10:49:23.039 -> Task 2 ----------
10:49:24.883 -> Task 1 **********
10:49:25.820 -> Task 2 ----------
10:49:26.079 -> Message printed from the tick hook interrupt ##########
10:49:29.658 -> Message printed from the tick hook interrupt ##########
10:49:31.751 -> Task 1 **********
10:49:32.272 -> Task 2 ----------
10:49:33.243 -> Message printed from the tick hook interrupt ##########
10:49:36.821 -> Message printed from the tick hook interrupt ##########
10:49:39.724 -> Task 2 ----------

References (example001-016)

https://github.com/greiman/FreeRTOS-Arduino/tree/master/libraries/FreeRTOS_AVR/examples/FreeRTOSBook