

FreeRTOS

A real time operating system for embedded systems

INTERRUPT MANAGEMENT

Interrupt Management

⌘ Topics covered:

- ❖ Which FreeRTOS API functions can be used from within an interrupt service routine (ISR).
- ❖ How a deferred interrupt scheme can be implemented.
- ❖ How to create and use binary semaphores and counting semaphores.
- ❖ The differences between binary and counting semaphores.
- ❖ How to use a queue to pass data into and out of an interrupt service routine.

The xHigherPriorityTaskWoken Parameter

- If a context switch is performed by an interrupt, then the task running when the interrupt exits might be different than the task that was running when the interrupt was entered
- Some FreeRTOS API functions can move a task from the Blocked state to the Ready state. This has already been seen with functions such as xQueueSendToBack()
- If the priority of a task that is unblocked by a FreeRTOS API function is higher than the priority of the task in the Running state, then, in accordance with the FreeRTOS scheduling policy, a switch to the higher priority task should occur.
- If the API function was called from a task:
If configUSE_PREEMPTION is set to 1 in FreeRTOSConfig.h then the switch to the higher priority task occurs automatically within the API function.

The xHigherPriorityTaskWoken Parameter

- If the API function was called from an interrupt: A switch to a higher priority task will not occur automatically inside an interrupt. Instead, a variable is set to inform the application writer that a context switch should be performed. Interrupt safe API functions (those that end in "FromISR") have a pointer parameter called `pxHigherPriorityTaskWoken` that is used for this purpose.
- If a context switch should be performed, then the interrupt safe API function will set `*pxHigherPriorityTaskWoken` to `pdTRUE`. To be able to detect this has happened, the variable pointed to by `pxHigherPriorityTaskWoken` must be initialized to `pdFALSE` before it is used for the first time.
- Use of the `pxHigherPriorityTaskWoken` parameter is optional. If it is not required, then set `pxHigherPriorityTaskWoken` to `NULL`.

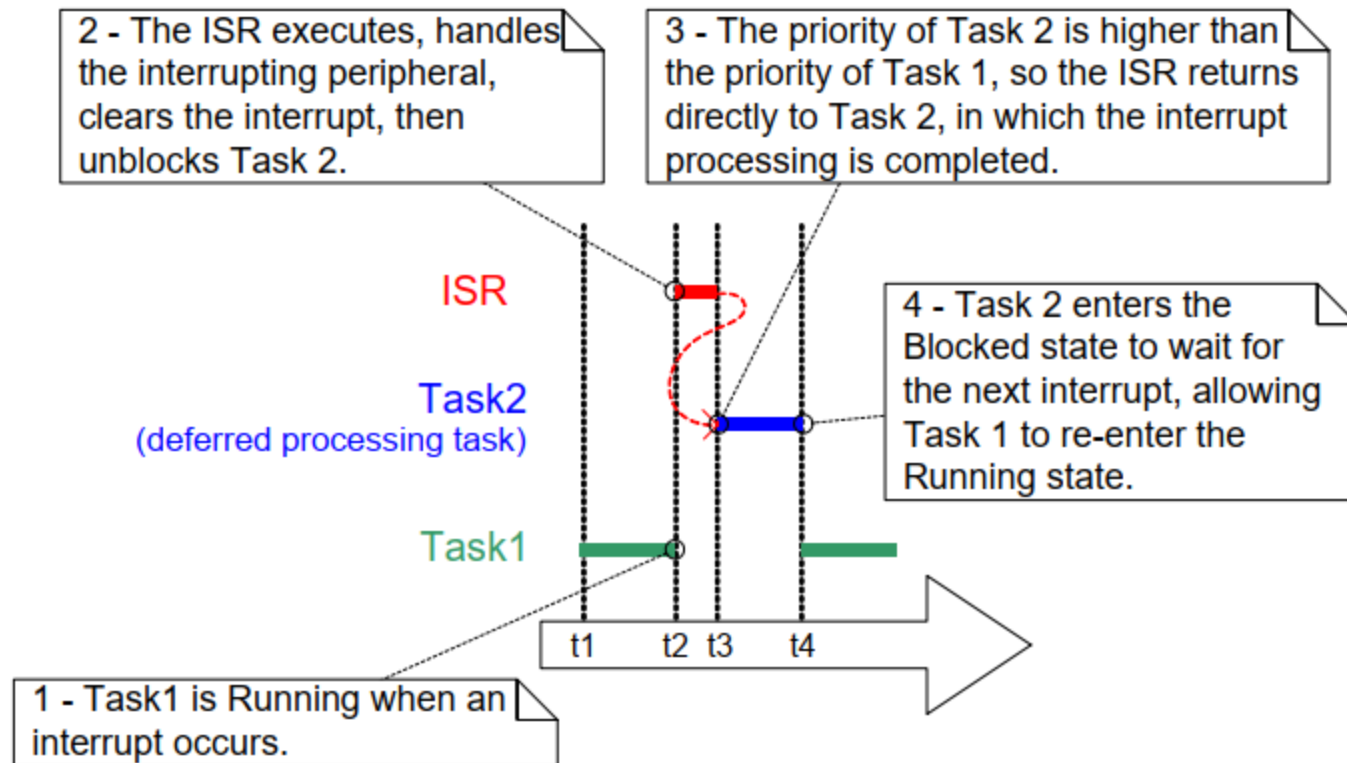
The portYIELD_FROM_ISR() and portEND_SWITCHING_ISR()

- taskYIELD() is a macro that can be called in a task to request a context switch.
- portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() are both interrupt safe of taskYIELD().
- They are both **used in the same way, and do the same thing.**
- Some FreeRTOS ports only provide one of the two macros. Newer FreeRTOS ports provide both macros.
- portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
- portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
- The xHigherPriorityTaskWoken parameter passed out of an interrupt safe API function can be used directly as the parameter in a call to portYIELD_FROM_ISR().
- If xHigherPriorityTaskWoken parameter is pdFALSE (zero), then a context switch is not requested, and the macro has no effect. If xHigherPriorityTaskWoken parameter is not pdFALSE, then a context switch is requested, and the task in the Running state might change

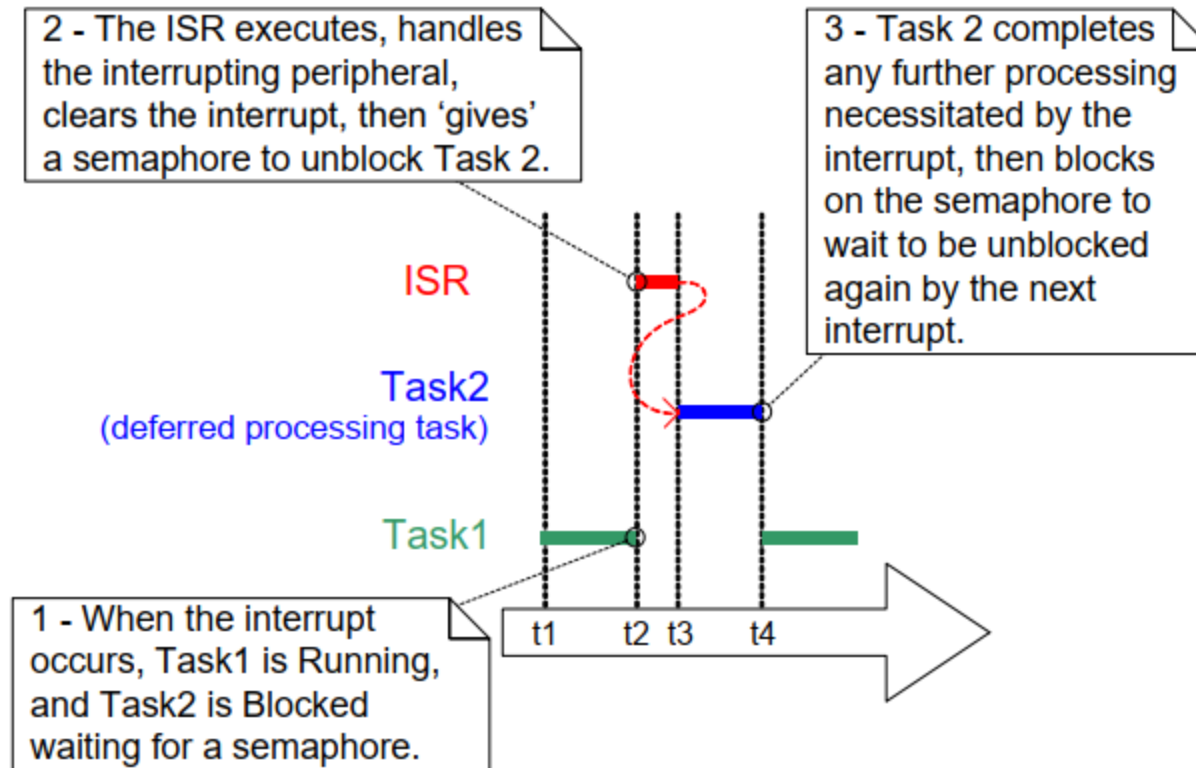
Deferred Interrupt Processing

- An interrupt service routine must record the cause of the interrupt, and clear the interrupt. Any other processing necessitated by the interrupt can often be performed in a task, allowing the interrupt service routine to exit as quickly as is practical. This is called 'deferred interrupt processing', because the processing necessitated by the interrupt is 'deferred' from the ISR to a task.
- Deferring interrupt processing to a task also allows the application writer to prioritize the processing relative to other tasks in the application, and use all the FreeRTOS API functions.
- If the priority of the task to which interrupt processing is deferred is above the priority of any other task, then the processing will be performed immediately, just as if the processing had been performed in the ISR itself.

Deferred Interrupt Processing



Interrupt and Handler Task



Deferred Interrupt Processing

There is no absolute rule as to when it is best to perform all processing necessitated by an interrupt in the ISR, and when it is best to defer part of the processing to a task. Deferring processing to a task is most useful when:

- The processing necessitated by the interrupt is not trivial. For example, if the interrupt is just storing the result of an analog to digital conversion, then it is almost certain this is best performed inside the ISR, but if result of the conversion must also be passed through a software filter, then it may be best to execute the filter in a task.
- It is convenient for the interrupt processing to perform an action that cannot be performed inside an ISR, such as write to a console, or allocate memory.
- The interrupt processing is not deterministic—meaning it is not known in advance how long the processing will take.

Binary Semaphores Used for Synchronization

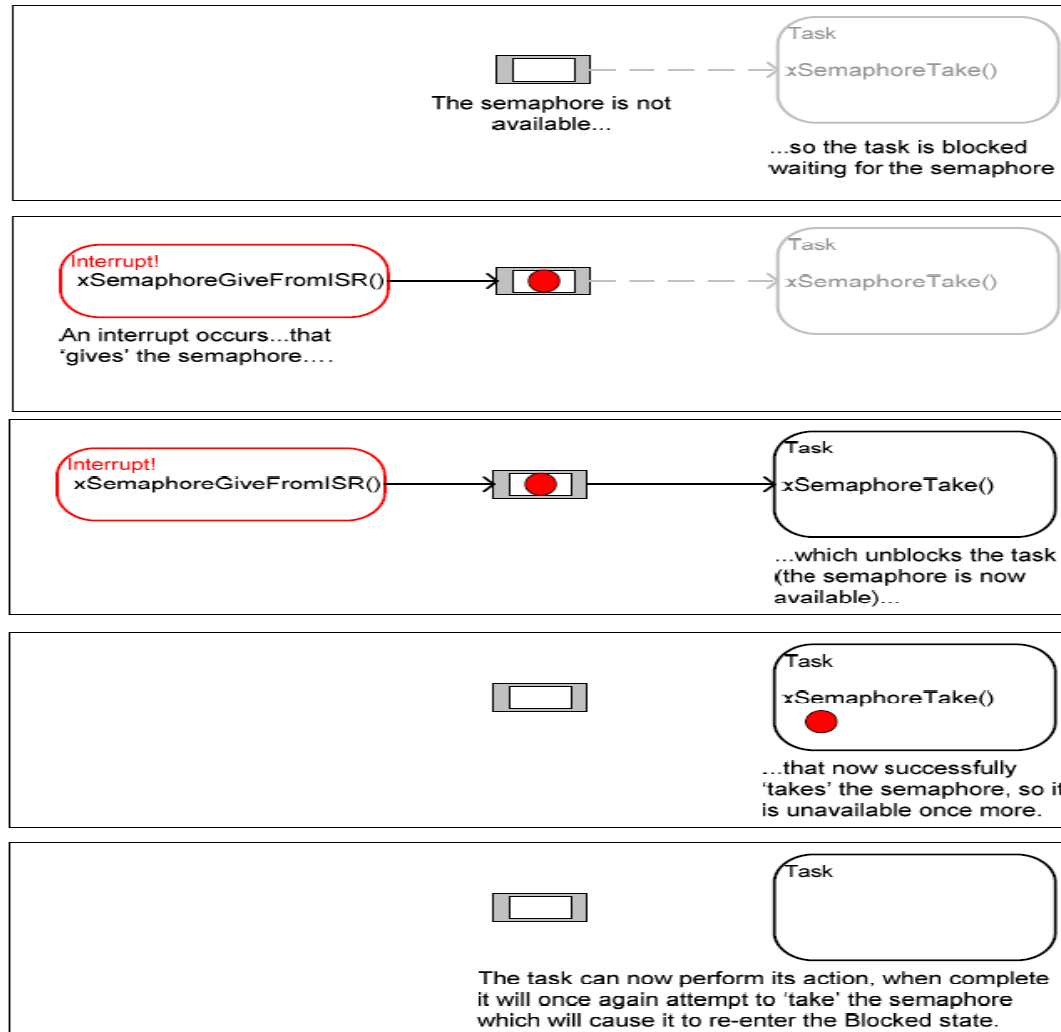
- ⌘ A Binary Semaphore can be used to unblock a task each time a particular interrupt occurs
- ⌘ The majority of the interrupt event processing can be implemented within the synchronized task.
- ⌘ Only a very fast and short portion remaining directly in the ISR.
- ⌘ The interrupt processing is said to have been 'deferred' to a 'handler' task
- ⌘ The handler task uses a blocking 'take' call to a semaphore and enters the Blocked state to wait for the event to occur
- ⌘ When the event occurs, the ISR uses a 'give' operation on the same semaphore to unblock the task

Deferred Interrupt Processing

'Taking a semaphore' and 'giving a semaphore' are concepts that have different meanings depending on their usage scenario. In this interrupt Synchronization scenario:

- ⌘ The Binary Semaphore can be considered conceptually as a queue with a length of one.
- ⌘ By calling `xSemaphoreTake()`, the handler task effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty.
- ⌘ When the event occurs, the ISR uses the `xSemaphoreGiveFromISR()` to place a semaphore into the queue, making the queue is full.
- ⌘ This causes the handler task to exit the Blocked state and remove the semaphore, leaving the queue empty once more.
- ⌘ When the handler task has completed its processing, it once more attempts to read from the queue, and, finding the queue empty, re-enters the Blocked state to wait for the next event.

Interrupt and Handler Task



Create a Binary Semaphore

- Before a semaphore can be used, it must be created. Use the **xSemaphoreCreateBinary()** API function to create a binary semaphore

SemaphoreHandle_t xSemaphoreCreateBinary(void);

- Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.
- If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
- If a non-NULL value is returned, it indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

'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

pdPASS is returned only if the call to xSemaphoreTake() 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(  
    SemaphoreHandle_t 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`

If `xSemaphoreGiveFromISR()` sets this value to `pdTRUE`, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

- ❖ Return value

`pdPASS` will only be returned if successful

`pdFAIL` if a semaphore is already available and cannot be given

Example 12. Using a Binary Semaphore to Synchronize a Task with an Interrupt

```
static void vPeriodicTask(void *pvParameters) {  
    //const TickType_t xDelay = 500 / portTICK_PERIOD_MS;  
    /* As per most tasks, this task is implemented within an infinite loop. */  
    for (;;) {  
        /* This task is just used to 'simulate' an interrupt. This is done by  
        periodically generating a software interrupt. */  
        vTaskDelay(500 / portTICK_PERIOD_MS);  
        /* Generate the interrupt, printing a message both before hand and  
        afterwards so the sequence of execution is evident from the output. */  
        Serial.print("Periodic task - About to generate an interrupt.\r\n");  
        digitalWrite(interruptPin, LOW);  
        digitalWrite(interruptPin, HIGH);  
        delay(20);  
        Serial.print("Periodic task - Interrupt generated.\r\n\r\n\r\n\r\n");  
  
    }  
}
```

xSemaphoreTake()

```
static void vHandlerTask(void *pvParameters) {  
    /* Note that when you create a binary semaphore in FreeRTOS, it is ready  
    to be taken, so you may want to take the semaphore after you create it  
    so that the task waiting on this semaphore will block until given by  
    another task. */  
    xSemaphoreTake(xBinarySemaphore, 0);  
  
    /* As per most tasks, this task is implemented within an infinite loop. */  
    for (;;) {  
        /* Use the semaphore to wait for the event. The semaphore was created  
        before the scheduler was started so before this task ran for the first  
        time. The task blocks indefinitely meaning this function call will only  
        return once the semaphore has been successfully obtained - so there is no  
        need to check the returned value. */  
        xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);  
  
        /* To get here the event must have occurred. Process the event (in this  
        case we just print out a message). */  
        Serial.print( "Handler task - Processing event.\r\n" );  
    }  
}
```

xSemaphoreGiveFromISR()

```
static void IRAM_ATTR vExampleInterruptHandler(void) {  
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;  
    xSemaphoreGiveFromISR(xBinarySemaphore, (BaseType_t  
*)&xHigherPriorityTaskWoken);  
    if (xHigherPriorityTaskWoken == pdTRUE) {  
        /* Giving the semaphore unblocked a task, and the priority of the  
        unblocked task is higher than the currently running task - force  
        a context switch to ensure that the interrupt returns directly to  
        the unblocked (higher priority) task.
```

NOTE: The syntax for forcing a context switch is different depending on the port being used. Refer to the examples for the port you are using for the correct method to use! */

```
    portYIELD_FROM_ISR();  
}  
}
```

Example 12. Using a Binary Semaphore to Synchronize a Task with an Interrupt

```
static void vPeriodicTask(void *pvParameters);
static void vHandlerTask(void *pvParameters);
static void IRAM_ATTR vExampleInterruptHandler(void);
SemaphoreHandle_t xBinarySemaphore;
const uint8_t interruptPin = 2;
void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600);
    pinMode(interruptPin, OUTPUT);
    digitalWrite(interruptPin, LOW);
    attachInterrupt(interruptPin, vExampleInterruptHandler, RISING);
}
```

Example 12. Using a Binary Semaphore to Synchronize a Task with an Interrupt

```
xBinarySemaphore = xSemaphoreCreateBinary();
if( xBinarySemaphore != NULL )
{
    /* Create the 'handler' task. This is the task that will be synchronized
    with the interrupt. The handler task is created with a high priority to
    ensure it runs immediately after the interrupt exits. In this case a
    priority of 3 is chosen. */
    xTaskCreate( vHandlerTask, "Handler", 1024, NULL, 3, NULL );

    /* Create the task that will periodically generate a software interrupt.
    This is created with a priority below the handler task to ensure it will
    get preempted each time the handler task exist the Blocked state. */
    xTaskCreate( vPeriodicTask, "Periodic", 1024, NULL, 1, NULL );

}
for (;;)
}
```

Example12 – Serial monitor

example012_modified_esp32 | Arduino IDE 2.3.2

File Edit Sketch Tools Help

ESP32 Dev Module

example012_modified_esp32.ino

```
63 portYIELD_FROM_ISR();
```

Output Serial Monitor x

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6') New Line 9600 baud

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.

Ln 63, Col 26 ESP32 Dev Module on COM6 2

Windows taskbar: 7:56 AM 5/6/2024

Sequence of execution

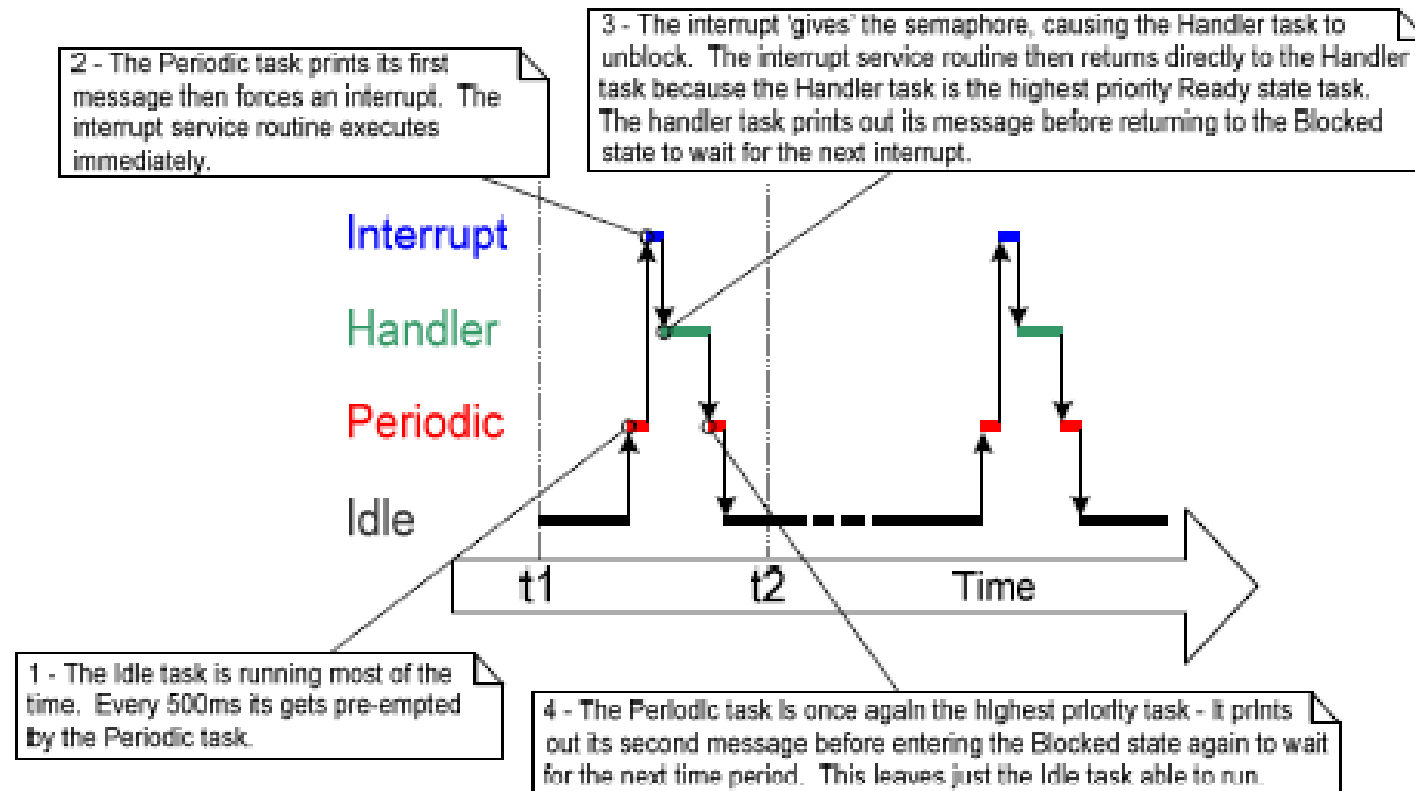


Figure 29 The sequence of execution when Example 12 is executed

Improving the Implementation of the Task Used in Example 12

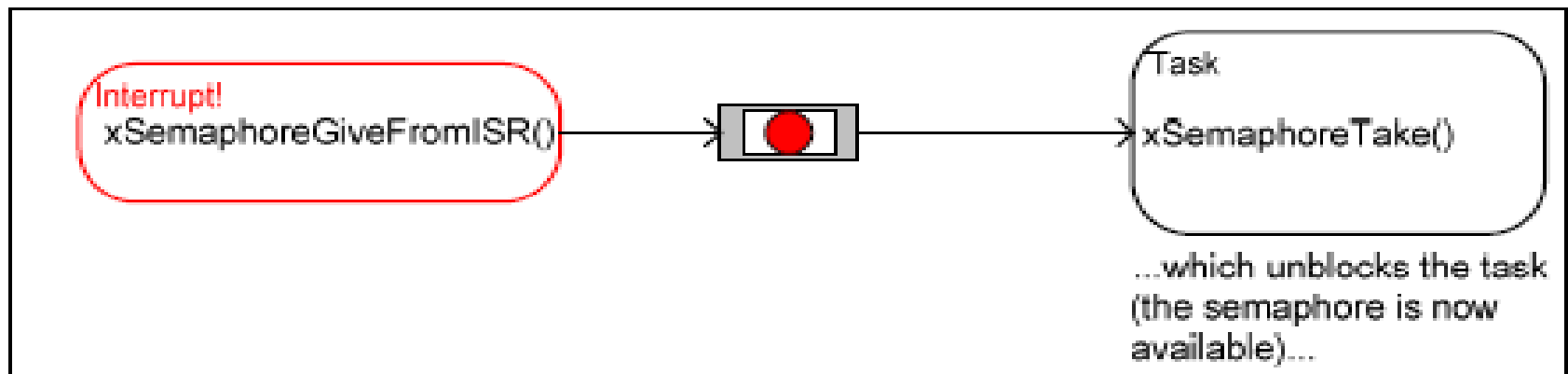
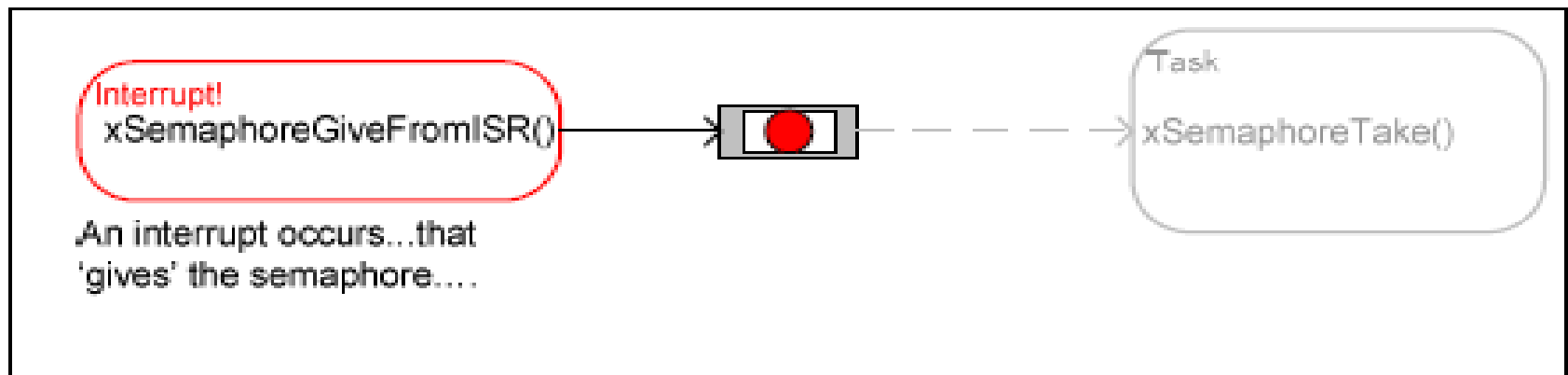
Example 12 used a binary semaphore to synchronize a task with an interrupt. The execution sequence was as follows:

1. The interrupt occurred.
2. The ISR executed and 'gave' the semaphore to unblock the task.
3. The task executed immediately after the ISR, and 'took' the semaphore.
4. The task processed the event, then attempted to 'take' the semaphore again—entering the Blocked state because the semaphore was not yet available (another interrupt had not yet occurred).

Improving the Implementation of the Task Used in Example 12

The structure of the task used in Example 12 is adequate only if interrupts occur at a relatively low frequency. To understand why, consider what would happen if a second, and then a third, interrupt had occurred before the task had completed its processing of the first interrupt:

1. When the second ISR executed, the semaphore would be empty, so the ISR would give the semaphore, and the task would process the second event immediately after it had completed processing the first event. That scenario is shown in Figure below.
2. When the third ISR executed, the semaphore would already be available, preventing the ISR giving the semaphore again, so the task would not know the third event had occurred. That scenario is shown in Figure below.



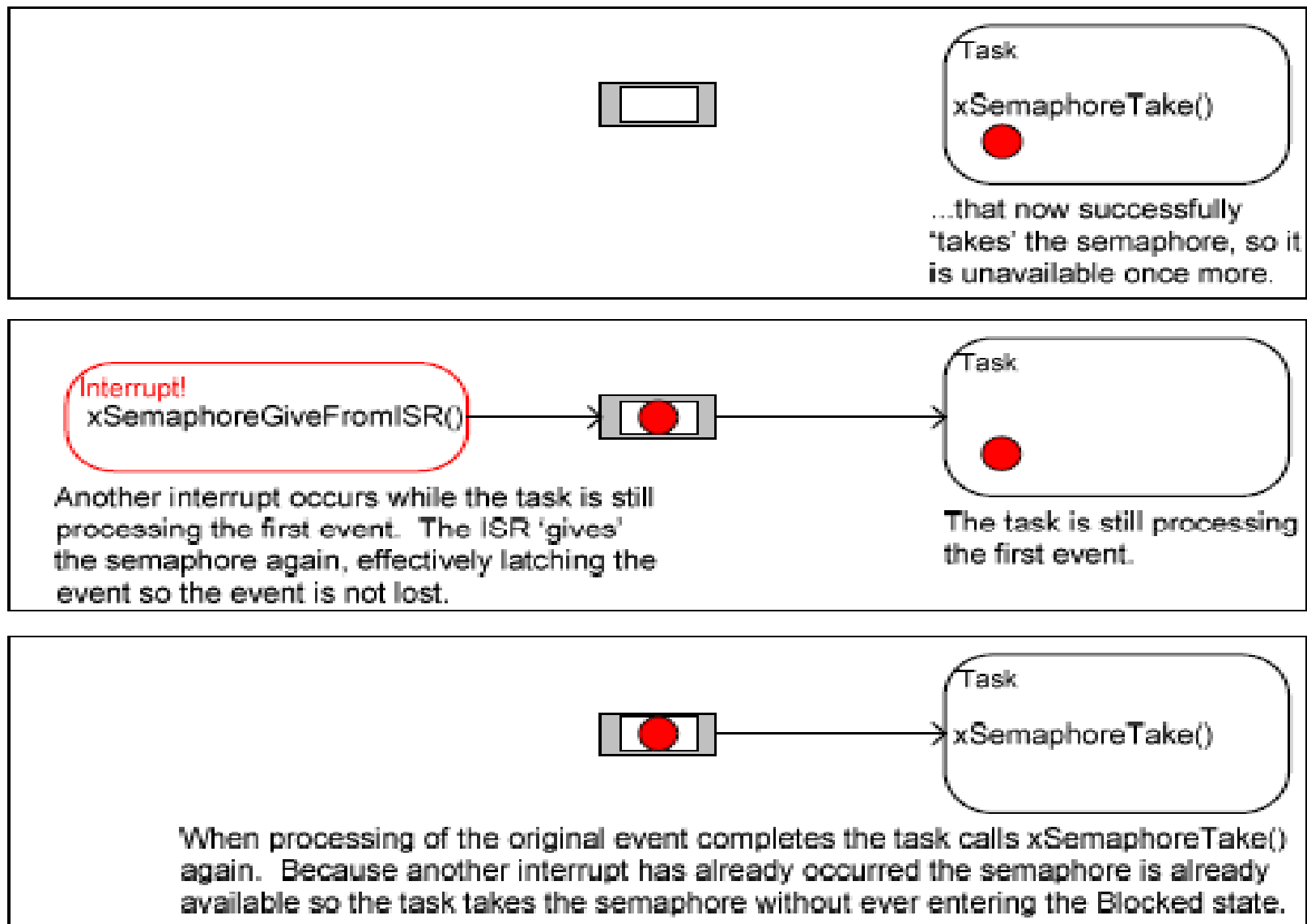
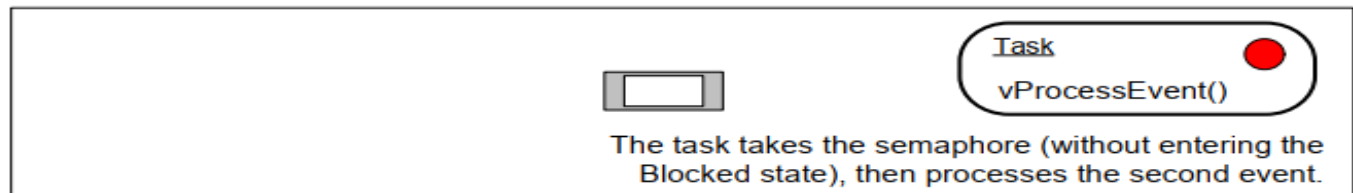
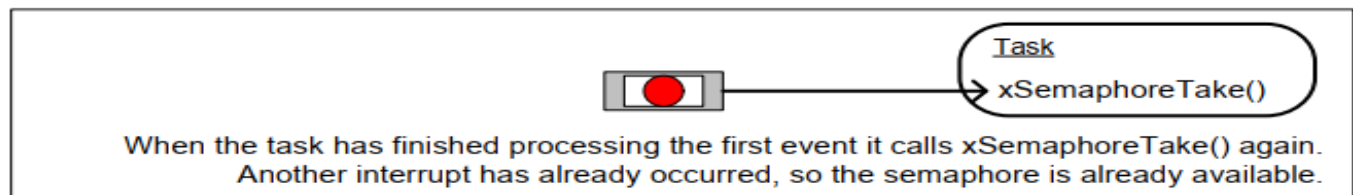
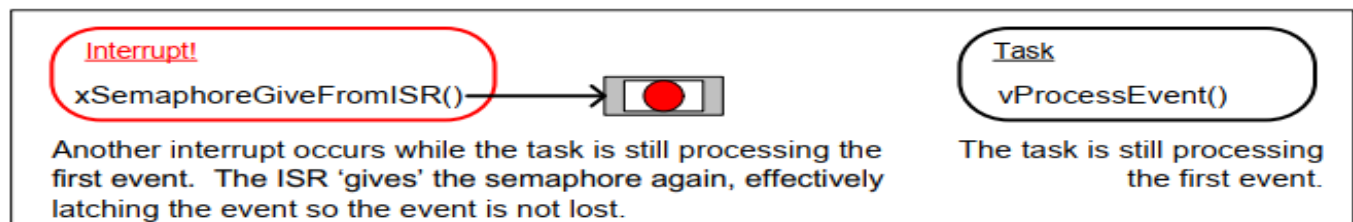
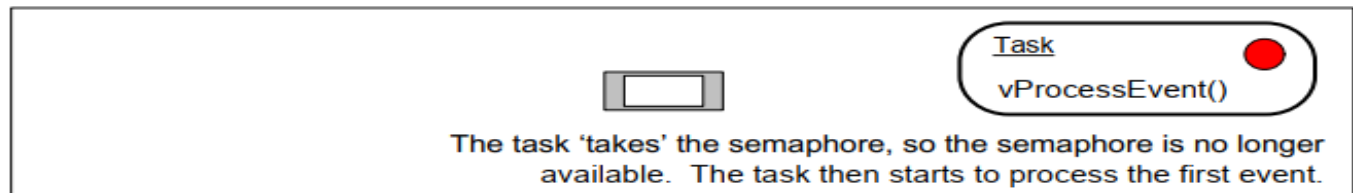
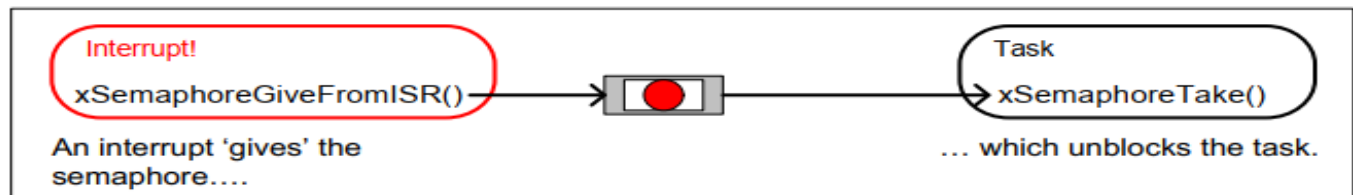
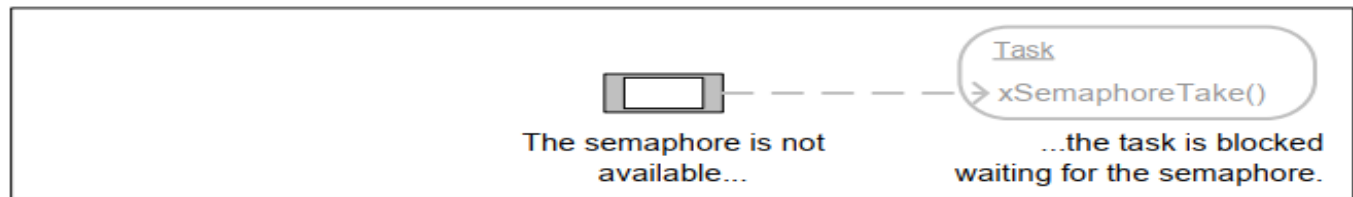
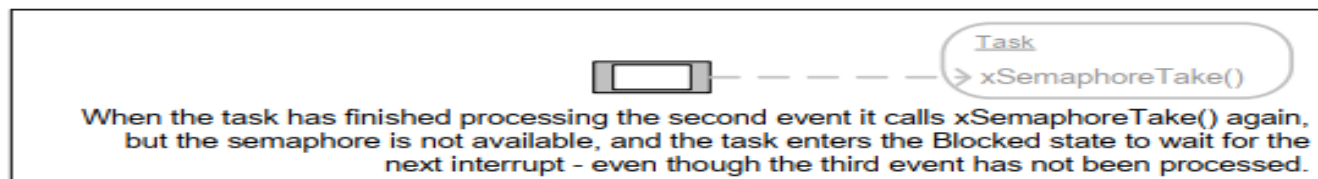
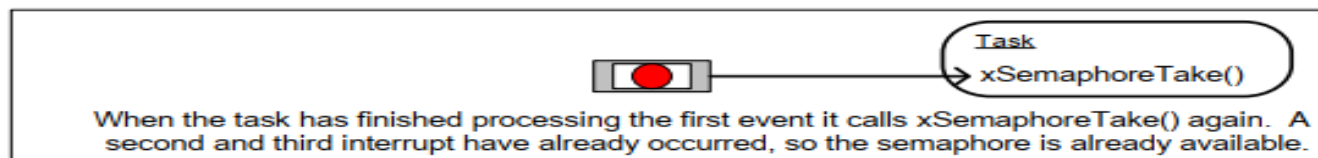
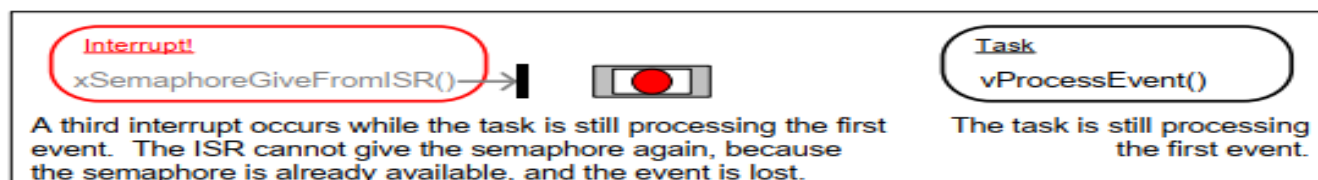
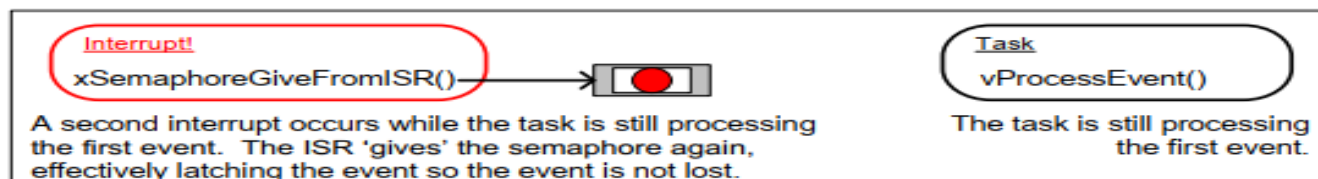
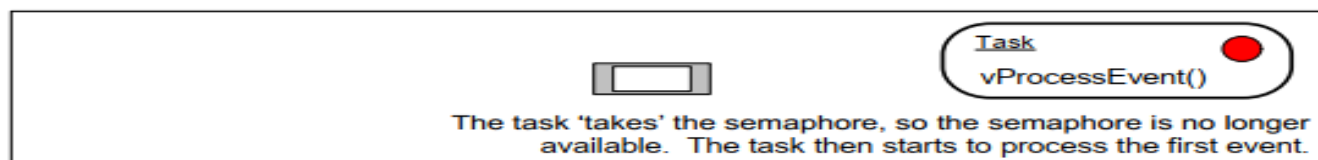
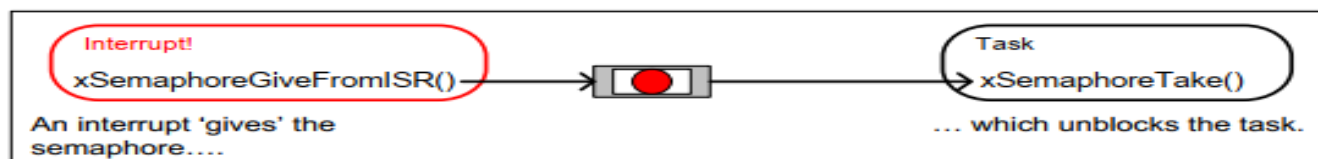


Figure 30 A binary semaphore can latch at most one event





- The deferred interrupt handling task used in Example 12 is structured so that it only processes one event between each call to `xSemaphoreTake()`. That was adequate for Example 12, because the interrupts that generated the events were triggered by software, and occurred at a predictable time.
- In real applications, interrupts are generated by hardware, and occur at unpredictable times. Therefore, to minimize the chance of an interrupt being missed, the deferred interrupt handling task must be structured so that it processes all the events that are already available between each call to `xSemaphoreTake()`⁶.
- This is demonstrated by Listing below, which shows how a deferred interrupt handler for a UART could be structured. In Listing below, it is assumed the UART generates a receive interrupt each time a character is received, and that the UART places received characters into a hardware FIFO (a hardware buffer).

```
static void vUARTReceiveHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime holds the maximum time expected between two
       interrupts. */
    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* The semaphore is 'given' by the UART's receive (Rx) interrupt.
           Wait a maximum of xMaxExpectedBlockTime ticks for the next
           interrupt. */
        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS )
        {
            /* The semaphore was obtained. Process ALL pending Rx events before
               calling xSemaphoreTake() again. Each Rx event will have placed a
               character in the UART's receive FIFO, and UART_RxCount() is
               assumed to return the number of characters in the FIFO. */
```

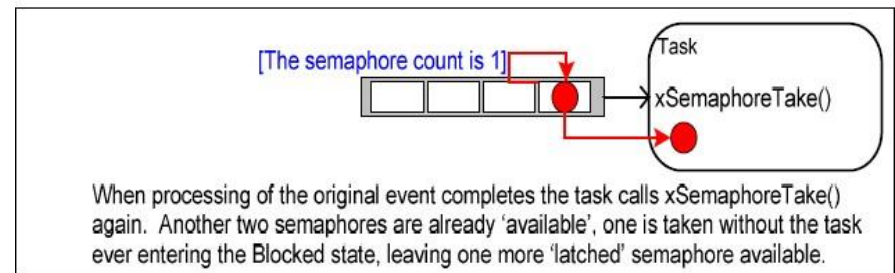
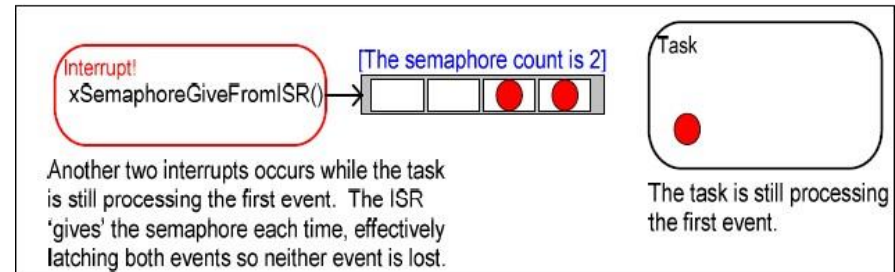
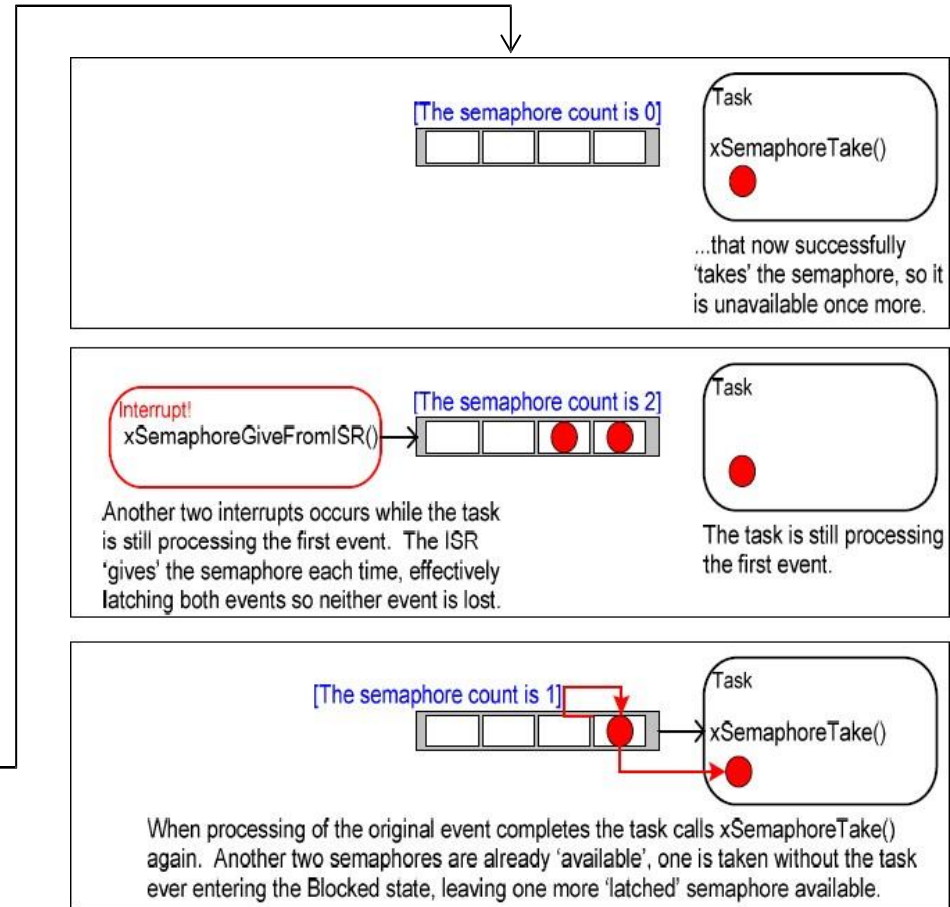
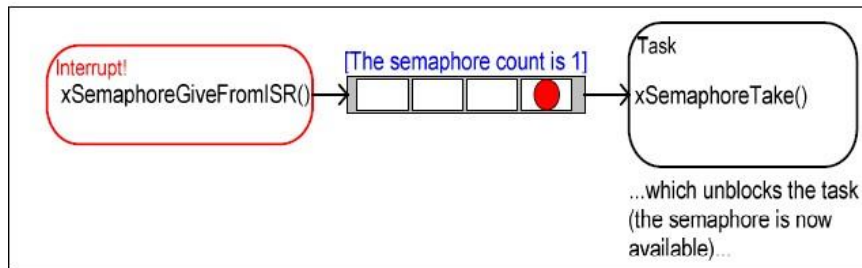
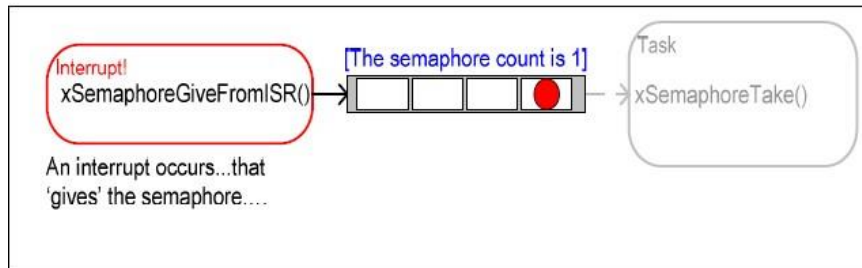
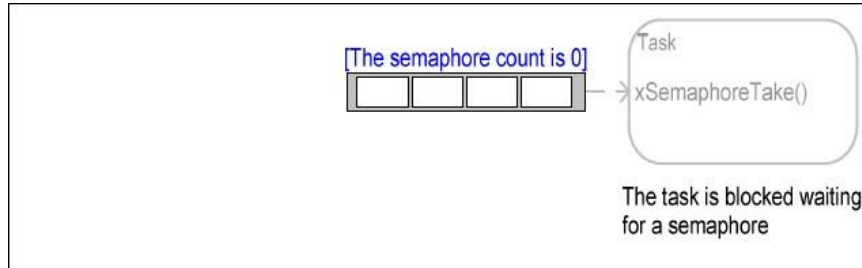
```
while( UART_RxCount() > 0 )
{
    /* UART_ProcessNextRxEvent() is assumed to process one Rx
       character, reducing the number of characters in the FIFO
       by 1. */
    UART_ProcessNextRxEvent();
}

/* No more Rx events are pending (there are no more characters in
   the FIFO), so loop back and call xSemaphoreTake() to wait for
   the next interrupt. Any interrupts occurring between this point
   in the code and the call to xSemaphoreTake() will be latched in
   the semaphore, so will not be lost. */
}
else
{
    /* An event was not received within the expected time. Check for,
       and if necessary clear, any error conditions in the UART that
       might be preventing the UART from generating any more
       interrupts. */
    UART_ClearErrors();
}
}
}
```


Counting Semaphores

- ⌘ For Binary Semaphore, when interrupts come at a speed faster than the handler task can process, events will be lost.
- ⌘ Counting semaphore can be thought of as queue that have a length of more than one. Tasks are not interested in the data that is stored in the queue—just the number of items in the queue.
- ⌘ Each time a counting semaphore is 'given', another space in its queue is used, **count value** is the number of items in the queue.
- ⌘ Counting semaphores are typically used for two things:
 - ⌘ **Counting events:** the count value is the difference between the number of events that have occurred and the number that have been processed
 - ⌘ **Resource management:** the count value indicates the number of resources available; a task must first obtains a semaphore before obtains the control of a resource; a task returns a semaphore when finishing with the resource

Using a counting semaphore to 'count' events



Create a Counting Semaphore

- ✂ Use `xSemaphoreCreateCounting()` to create a counting semaphore

```
SemaphoreHandle_t    xSemaphoreCreateCounting(  
                        UBaseType_t uxMaxCount,  
                        UBaseType_t uxInitialCount);
```

- ❖ `uxMaxCount`

The maximum value the semaphore will count to

- ❖ `uxInitialCount`

The initial count value of the semaphore after it has been created

- ❖ Return value

If NULL is returned then the semaphore could not be created because there was insufficient heap memory available

A non-NULL value being returned indicates that the semaphore was created successfully. The returned value should be stored as the handle to the created semaphore.

Example 13. Using a Counting Semaphore to Synchronize a Task with an Interrupt

```
static void IRAM_ATTR vExampleInterruptHandler( void )
{
    static BaseType_t xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore multiple times. The first will unblock the handler
    task, the following 'gives' are to demonstrate that the semaphore latches
    the events to allow the handler task to process them in turn without any
    events getting lost. This simulates multiple interrupts being taken by the
    processor, even though in this case the events are simulated within a single
    interrupt occurrence.*/
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Example13 – Serial monitor

example013_esp32 | Arduino IDE 2.3.2

File Edit Sketch Tools Help

ESP32 Dev Module

example013_esp32.ino

84

85

Output Serial Monitor X

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6') New Line 9600 baud

erodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.

Ln 90, Col 1 ESP32 Dev Module on COM6 2

Deferring Work to the RTOS Daemon Task

- The deferred interrupt handling examples presented so far have required the application writer to create a task for each interrupt that uses the deferred processing technique.
- It is also possible to use the `xTimerPendFunctionCallFromISR()`⁸ API function to defer interrupt processing to the RTOS daemon task, which removes the need to create a separate task for each interrupt. Deferring interrupt processing to the daemon task is called 'centralized deferred interrupt processing'.

USING QUEUES WITHIN AN INTERRUPT SERVICE ROUTINE

- Semaphores are used to communicate events.
- Queues are used to both communicate events and **transfer data**.
- **xQueueSendToFrontFromISR()**, **xQueueSendToBackFromISR()** and **xQueueReceiveFromISR()** are versions of **xQueueSendToFront()**, **xQueueSendToBack()** and **xQueueReceive()** respectively that **are safe to use within an interrupt service routine**.

Example 14. Sending and Receiving on a Queue from Within an Interrupt

```
static void vIntegerGenerator( void *pvParameters )
{
    TickType_t xLastExecutionTime;
    unsigned long ulValueToSend = 0;
    int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();
    for( ;; )
    {
        vTaskDelayUntil( &xLastExecutionTime, 200/portTICK_PERIOD_MS );
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }
        Serial.print( "Generator task - About to generate an interrupt.\r\n" );
        digitalWrite(outputPin, LOW);
        digitalWrite(outputPin, HIGH);
        Serial.print( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```



```
static void vExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    static unsigned long ulReceivedNumber;

    /* The strings are declared static const to ensure they are not allocated to the
    interrupt service routine stack, and exist even when the interrupt service routine
    is not executing. */
    static const char *pcStrings[] =
    {
        "String 0\r\n",
        "String 1\r\n",
        "String 2\r\n",
        "String 3\r\n"
    };
};
```

```
/* Loop until the queue is empty. */
while ( xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber,
&xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
{
    /* Truncate the received value to the last two bits (values 0 to 3 inc.), then
    send the string that corresponds to the truncated value to the other
    queue. */
    ulReceivedNumber &= 0x03;
    xQueueSendToBackFromISR( xStringQueue, &pcStrings[ ulReceivedNumber ],
&xHigherPriorityTaskWoken );
}
vPortYield();
}
```

```
static void vStringPrinter( void *pvParameters )
{
char *pcString;

for( ;; )
{
    /* Block on the queue to wait for data to arrive. */
    xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

    /* Print out the string received. */
    Serial.print( pcString );
}
}
/*-----*/
```

```
void setup( void )
{
    Serial.begin(9600);

    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Create the task that uses a queue to pass integers to the interrupt service
    routine. The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 200, NULL, 1, NULL );

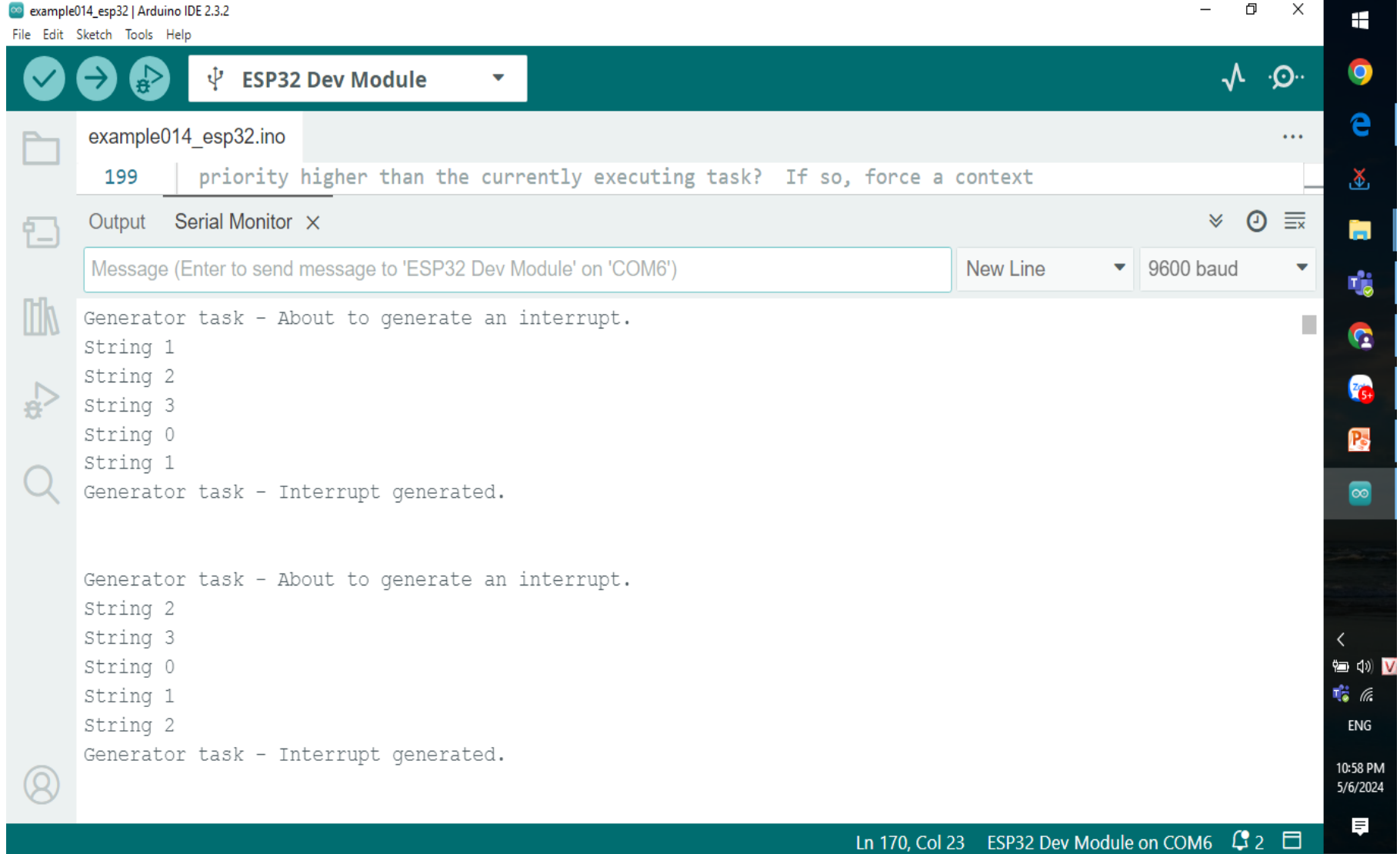
    /* Create the task that prints out the strings sent to it from the interrupt
    service routine. This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 200, NULL, 2, NULL );
```

```
/* Install the interrupt handler. */
pinMode(inputPin, INPUT);
pinMode(outputPin, OUTPUT);
digitalWrite(outputPin, HIGH);
bool tmp = digitalRead(inputPin);
digitalWrite(outputPin, LOW);
if (digitalRead(inputPin) || !tmp) {
    Serial.println("pin 2 must be connected to pin 3");
    while(1);
}
attachInterrupt(digitalPinToInterrupt(inputPin), vExampleInterruptHandler, RISING);

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

for( ;; );
}
```

Example 14. If there is a connection between pin 2 and pin 4



The screenshot shows the Arduino IDE 2.3.2 interface. The top bar indicates the current sketch is `example014_esp32.ino` and the target board is `ESP32 Dev Module`. The Serial Monitor is open, showing the output of the sketch. The output consists of two identical sequences of messages, each preceded by a line number (199).

```
199 priority higher than the currently executing task? If so, force a context
```

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6') New Line 9600 baud

Generator task - About to generate an interrupt.
String 1
String 2
String 3
String 0
String 1
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 2
String 3
String 0
String 1
String 2
Generator task - Interrupt generated.

Ln 170, Col 23 ESP32 Dev Module on COM6 2

Example 14. If there is no connection between pin 2 and pin 4

example014_esp32 | Arduino IDE 2.3.2

File Edit Sketch Tools Help

ESP32 Dev Module

example014_esp32.ino

199 | priority higher than the currently executing task? If so, force a context

Output Serial Monitor X

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6') New Line 9600 baud

Generator task - About to generate an interrupt.
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
Generator task - Interrupt generated.

Ln 170, Col 23 ESP32 Dev Module on COM6 2

Sequence of Execution

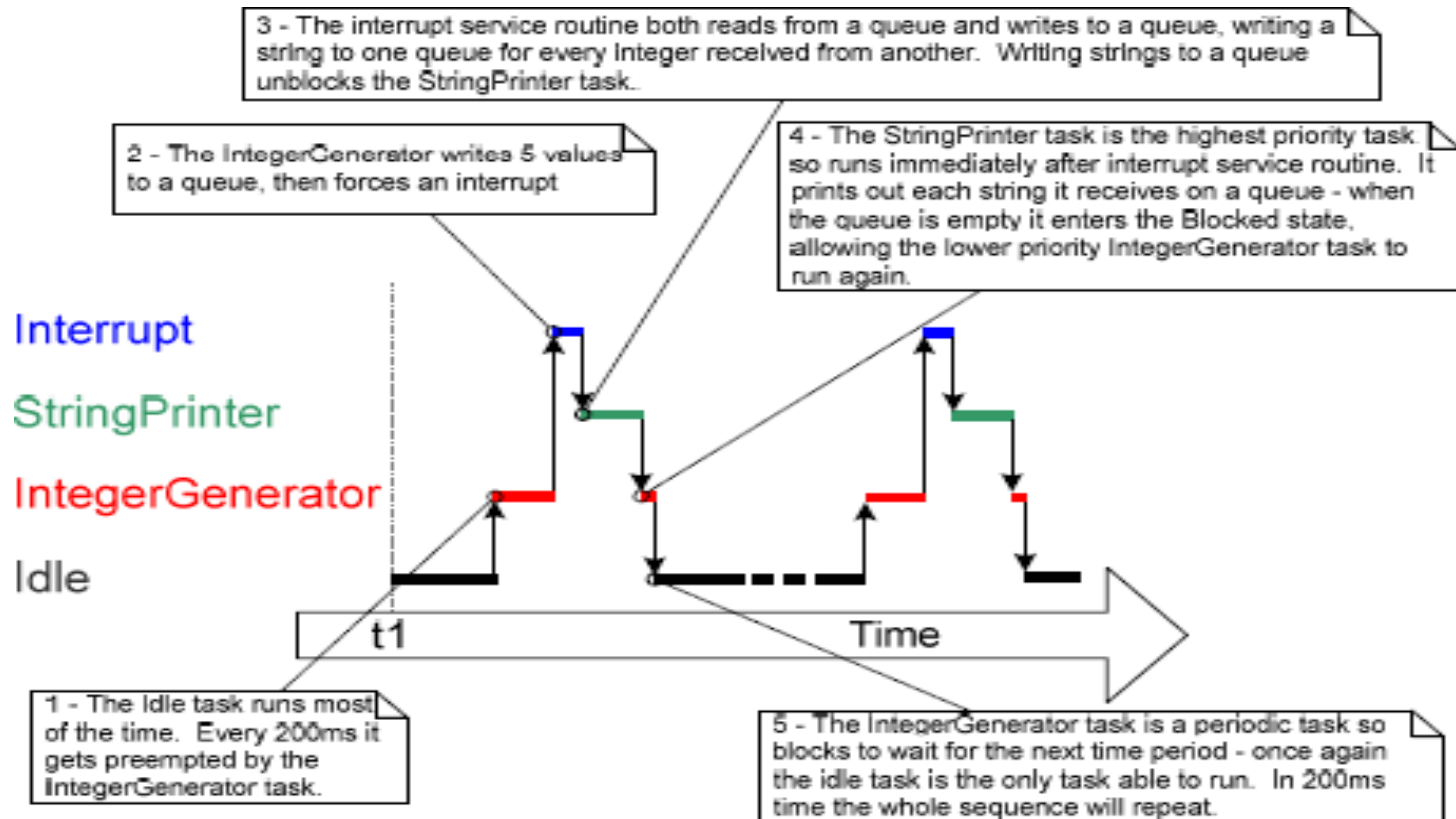


Figure 34 The sequence of execution produced by Example 14