

Real Time Operating System

“FreeRTOS”

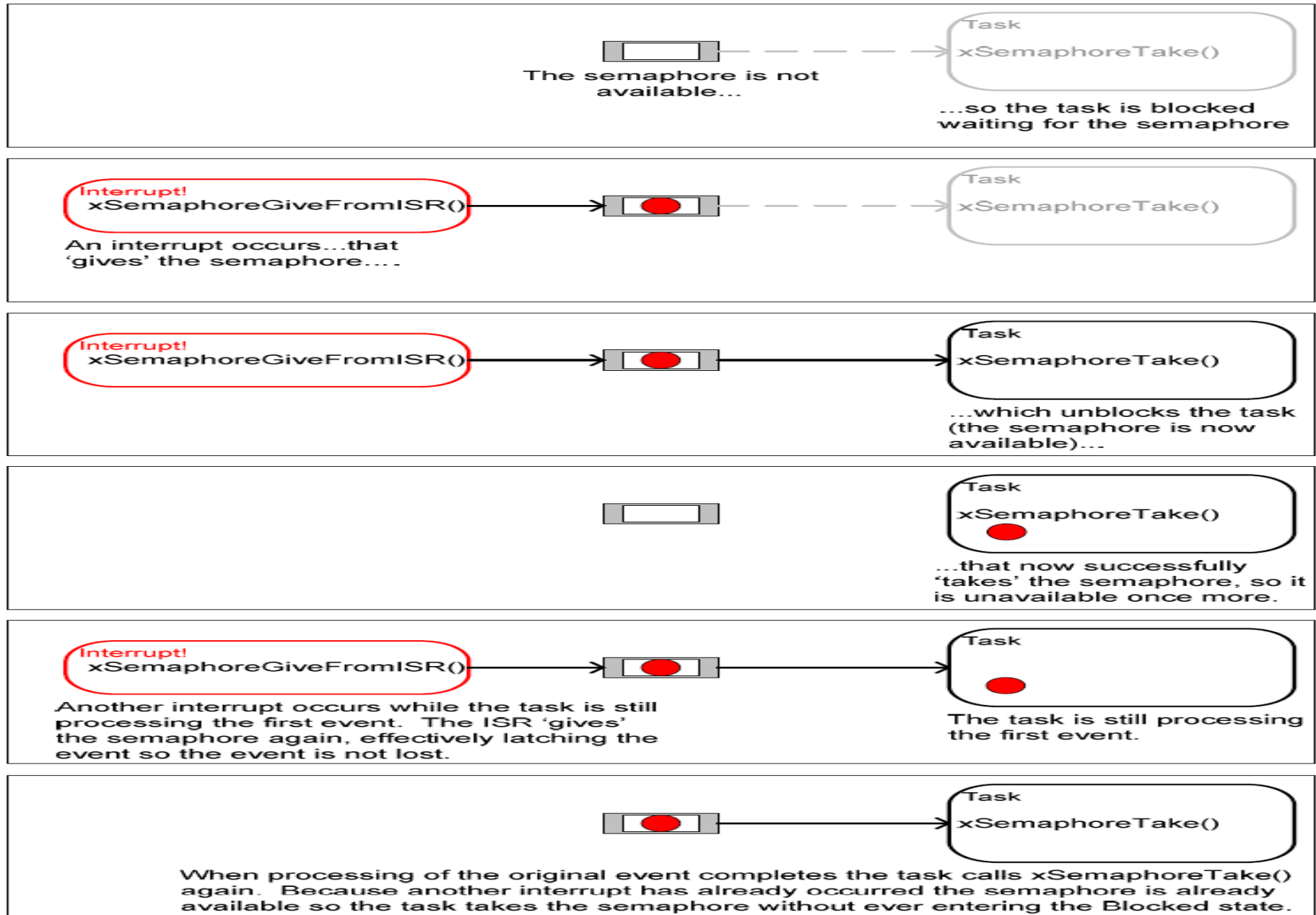
Interrupt Management

Counting Semaphores

# Agenda

- Binary semaphore pitfall
- Counting semaphore handling of fast interrupts
- Sending/Receiving to Queues from ISR

# Binary Semaphore Pitfall



**Figure 29. A binary semaphore can latch at most one event**

# Counting Semaphore

[The semaphore count is 0]



Task  
xSemaphoreTake()

The task is blocked waiting for a semaphore

Interrupt!  
xSemaphoreGiveFromISR()

[The semaphore count is 1]



Task  
xSemaphoreTake()

An interrupt occurs...that 'gives' the semaphore....

Interrupt!  
xSemaphoreGiveFromISR()

[The semaphore count is 1]



Task  
xSemaphoreTake()

...which unblocks the task (the semaphore is now available)...

[The semaphore count is 0]



Task  
xSemaphoreTake()  


...that now successfully 'takes' the semaphore, so it is unavailable once more.

Interrupt!  
xSemaphoreGiveFromISR()

[The semaphore count is 2]

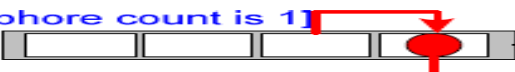


Another two interrupts occurs while the task is still processing the first event. The ISR 'gives' the semaphore each time, effectively latching both events so neither event is lost.

Task  


The task is still processing the first event.

[The semaphore count is 1]



Task  
xSemaphoreTake()  


When processing of the original event completes the task calls xSemaphoreTake() again. Another two semaphores are already 'available', one is taken without the task ever entering the Blocked state, leaving one more 'latched' semaphore available.

Figure 30. Using a counting semaphore to 'count' events

# Counting Semaphore

## ➤ Counting events

- An event handler will **'give'** a semaphore each time an event occurs—causing the **semaphore's count** value to be **incremented** on each 'give'.
- A handler task will **'take'** a semaphore each time it processes an event—causing the **semaphore's count** value to be **decremented** on each take.
- The **count value** is the **difference** between the number of events that have **occurred** and the number that have been **processed**.
- Counting semaphores that are used to count events are created with an **initial count value of zero**.

## ➤ Resource management.

- The count value indicates the **number of resources available**.
- To obtain **control of a resource** a task must first **obtain a semaphore—decrementing** the semaphore's count value.
- When the count value reaches **zero**, there are **no free resources**.
- When a **task finishes with the resource**, it **'gives' the semaphore back—incrementing** the semaphore's count value.
- Counting semaphores that are used to manage resources are created so that their **initial count value equals the number of resources that are available**.

# Counting Semaphore

---

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,  
                                           unsigned portBASE_TYPE uxInitialCount );
```

---

- uxMaxCount: The maximum value the semaphore will count to.
  - uxMaxCount value is effectively the **length of the “queue”**.
  - When the semaphore is to be used to count or **latch events**, uxMaxCount is the **maximum number of events that can be latched**.
  - When the semaphore is to be used to manage access to a collection of **resources**, uxMaxCount should be set to the **total number of resources that are available**.
  
- uxInitialCount: The initial count value of the semaphore after it has been created.
  - When the semaphore is to be used to count **or latch events**, uxInitialCount should be set **to zero—as**, presumably, when the semaphore is created, no events have yet occurred.
  - When the semaphore is to be used to manage access to a collection of **resources**, uxInitialCount should be set to equal **uxMaxCount—as**, presumably, when the semaphore is created, all **the resources are available**.

# Counting Semaphore; Example 13

```
/* Before a semaphore is used it must be explicitly created. In this example
a counting semaphore is created. The semaphore is created to have a maximum
count value of 10, and an initial count value of 0. */
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

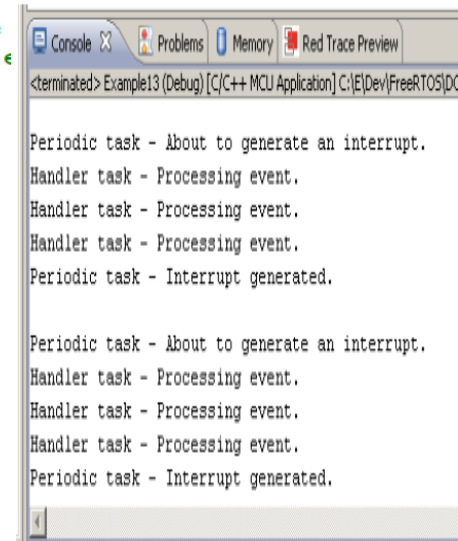
```
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

/* '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 );

/* Clear the software interrupt bit using the interrupt controllers Clear
Pending register. */
mainCLEAR_INTERRUPT();

/* Giving the semaphore may have unblocked a task - if it did and the
unblocked task has a priority equal to or above the currently executing
task then xHigherPriorityTaskWoken will have been set to pdTRUE and
portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
higher priority task.

NOTE: The syntax for forcing a context switch within an ISR varies between
FreeRTOS ports. The portEND_SWITCHING_ISR() macro is provided as part of
the Cortex-M3 port layer for this purpose. taskYIELD() must never be called
from an ISR! */
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```



Listing 51. The implementation of the interrupt service routine used by Example 13

# Using Queues within an Interrupt Service Routine

---

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,  
                                         void *pvItemToQueue  
                                         portBASE_TYPE *pxHigherPriorityTaskWoken  
                                         );
```

---

Listing 53. The xQueueSendToBackFromISR() API function prototype

- `pxHigherPriorityTaskWoken` It is possible that a single queue will have **one or more tasks blocked** on it waiting for data to become available.
- Calling `xQueueSendToFrontFromISR()` or `xQueueSendToBackFromISR()` can make data available, and so cause such a **task to leave the Blocked state**.
- If calling the API function causes a task to leave the Blocked state, and the **unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted)**, then, internally, the API function will set **`*pxHigherPriorityTaskWoken` to `pdTRUE`**.



# Using Queues within an Interrupt Service Routine; Example 14

---

```
int main( void )
{
    /* Before a queue can be used it must first be created.  Create both queues
    used by this example.  One queue can hold variables of type unsigned long,
    the other queue can hold variables of type char*.  Both queues can hold a
    maximum of 10 items.  A real application should check the return values to
    ensure the queues have been successfully created. */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Enable the software interrupt and set its priority. */
    prvSetupSoftwareInterrupt();

    /* 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", 240, 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", 240, NULL, 2, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

# Using Queues within an Interrupt Service Routine; Example 14

---

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

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* This is a periodic task. Block until it is time to run again.
        The task will execute every 200ms. */
        vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );

        /* Send an incrementing number to the queue five times. The values will
        be read from the queue by the interrupt service routine. The interrupt
        service routine always empties the queue so this task is guaranteed to be
        able to write all five values, so a block time is not required. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }

        /* Force an interrupt so the interrupt service routine can read the
        values from the queue. */
        vPrintString( "Generator task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Generator task - Interrupt generated.\n\n" );
    }
}
```

# Using Queues within an Interrupt Service Routine; Example 14

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE 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\n",
    "String 1\n",
    "String 2\n",
    "String 3\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 );
}

/* Clear the software interrupt bit using the interrupt controllers Clear
Pending register. */
mainCLEAR_INTERRUPT();

/* xHigherPriorityTaskWoken was initialised to pdFALSE. It will have then
been set to pdTRUE only if reading from or writing to a queue caused a task
of equal or greater priority than the currently executing task to leave the
Blocked state. When this is the case a context switch should be performed.
In all other cases a context switch is not necessary.

NOTE: The syntax for forcing a context switch within an ISR varies between
FreeRTOS ports. The portEND_SWITCHING_ISR() macro is provided as part of
the Cortex-M3 port layer for this purpose. taskYIELD() must never be called
from an ISR! */
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

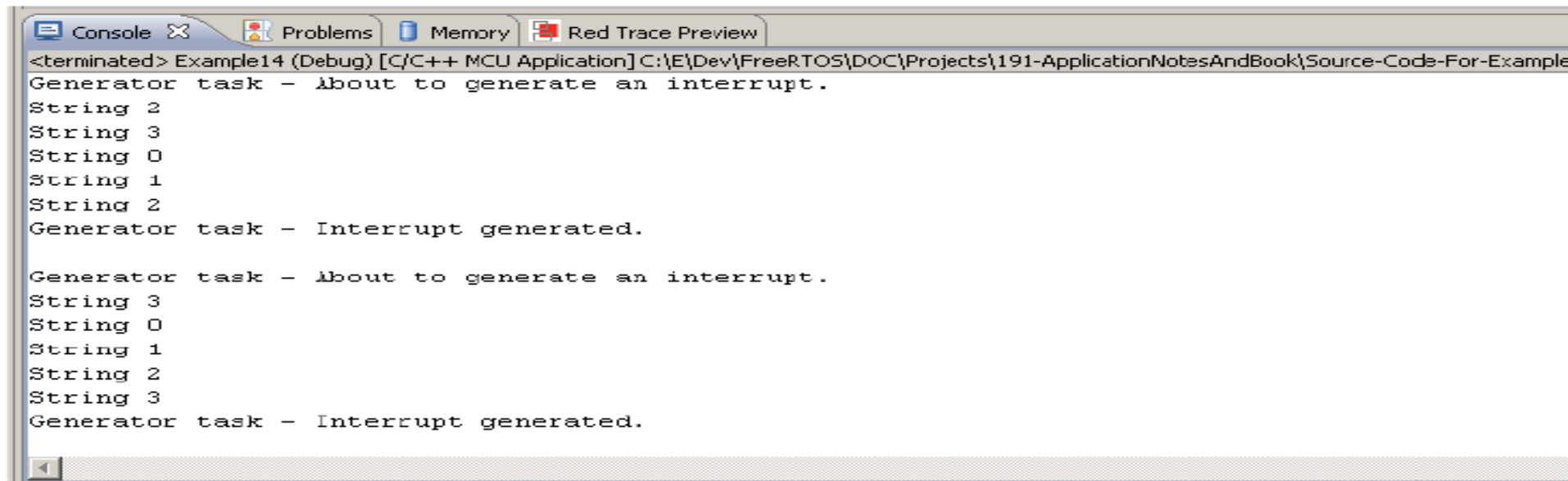
# Using Queues within an Interrupt Service Routine; Example 14

```
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. */
        vPrintString( pcString );
    }
}
```

**Listing 56.** The task that prints out the strings received from the interrupt service routine in Example 14



```
<terminated> Example14 (Debug) [C/C++ MCU Application] C:\E\Dev\FreeRTOS\DOC\Projects\191-ApplicationNotesAndBook\Source-Code-For-Example
Generator task - About to generate an interrupt.
String 2
String 3
String 0
String 1
String 2
Generator task - Interrupt generated.

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

**Figure 32.** The output produced when Example 14 is executed

# Using Queues within an Interrupt Service Routine; Example 14

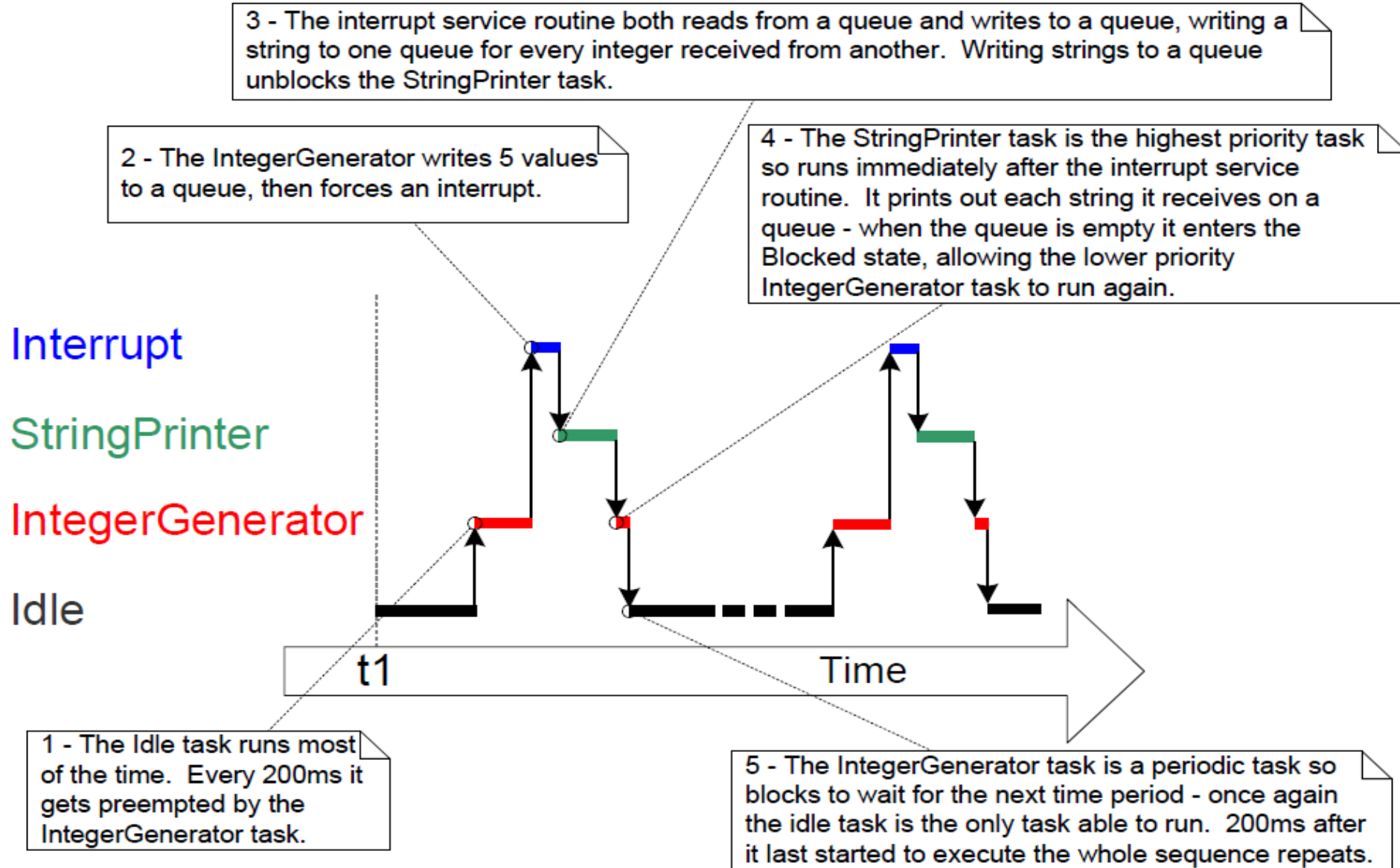


Figure 33. The sequence of execution produced by Example 14