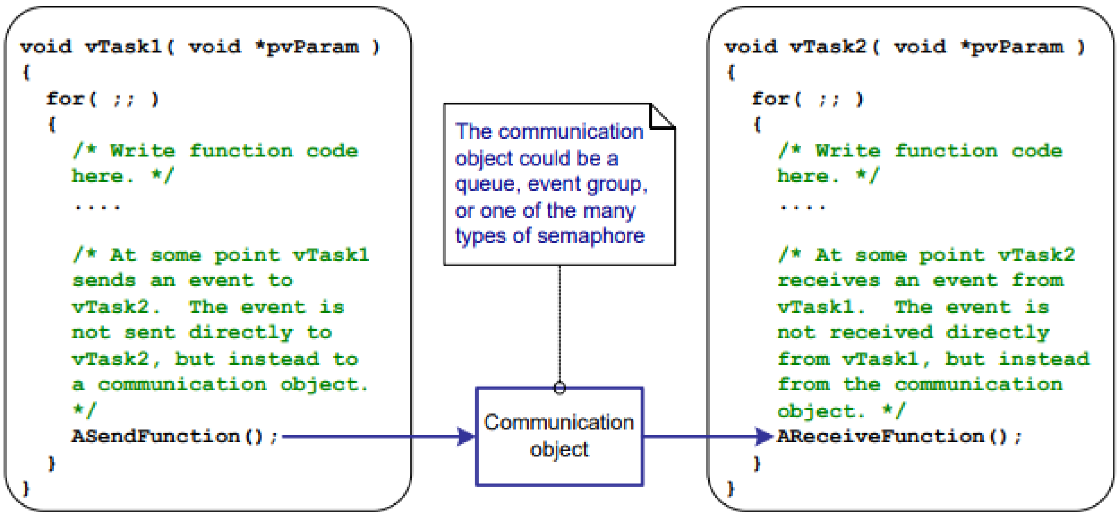
TASK NOTIFICATIONS

INDEX:

1. Introduction

1.1 Communicating Through Intermediary Objects

- There are various ways in which tasks can communicate with each other. The methods described so far have required the creation of a communication object.
- Examples of communication objects include queues, event groups, and various different types of semaphore.
- When a communication object is used, events and data are not sent directly to a receiving task, or a receiving ISR, but are instead sent to the communication object. Likewise, tasks and ISRs receive events and data from the communication object, rather than directly from the task or ISR that sent the event or data



```
void vTask1( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask1
    sends an event to
    vTask2.  The event is
    not sent directly to
    vTask2, but instead to
    a communication object.
    */
    ASendFunction();
  }
}
```

```
void vTask2( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask2
    receives an event from
    vTask1.  The event is
    not received directly
    from vTask1, but instead
    from the communication
    object. */
    AReceiveFunction();
  }
}
```

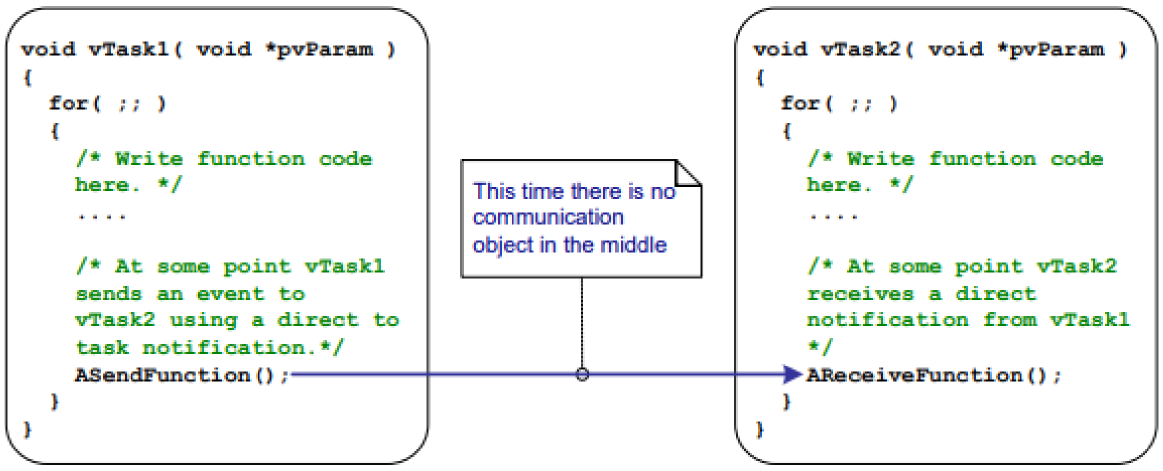
The communication object could be a queue, event group, or one of the many types of semaphore
Communication object

1.2 Task Notifications—Direct to Task Communication

- ‘Task Notifications’ allow tasks to interact with other tasks, and to synchronize with ISRs, without the need for a separate communication object.
- By using a task notification, a task or ISR can send an event directly to the receiving task
- Task notification functionality is optional. To include task notification functionality set configUSE_TASK_NOTIFICATIONS to 1 in FreeRTOSConfig.h.
- When configUSE_TASK_NOTIFICATIONS is set to 1, each task has a ‘Notification State’, which can be either ‘Pending’ or ‘Not-Pending’, and a ‘Notification Value’, which is a 32-bit

unsigned integer. When a task receives a notification, its notification state is set to pending. When a task reads its notification value, its notification state is set to not-pending.

- A task can wait in the Blocked state, with an optional time out, for its notification state to become pending



```
void vTask1( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask1
    sends an event to
    vTask2 using a direct to
    task notification.*/
    ASendFunction();
  }
}
```

```
void vTask2( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask2
    receives a direct
    notification from vTask1
    */
    AReceiveFunction();
  }
}
```

2. Task Notifications - Benefits and Limitations

2.1 Performance Benefits of Task Notifications

Using a task notification to send an event or data to a task is significantly faster than using a queue, semaphore or event group to perform an equivalent operation.

2.2 RAM Footprint Benefits of Task Notifications

Likewise, using a task notification to send an event or data to a task requires significantly less RAM than using a queue, semaphore or event group to perform an equivalent operation. This is because each communication object (queue, semaphore or event group) must be created before it can be used, whereas enabling task notification functionality has a fixed overhead of just eight bytes of RAM per task.

2.3 Limitations of Task Notifications

Task notifications are faster and use less RAM than communication objects, but task notifications cannot be used in all scenarios. This section documents the scenarios in which a task notification cannot be used:

2.3.1 Sending an event or data to an ISR

- Communication objects can be used to send events and data from an ISR to a task, and from a task to an ISR.
- Task notifications can be used to send events and data from an ISR to a task, but they cannot be used to send events or data from a task to an ISR.
- By limiting communication to one-way from ISR to task, real-time and embedded systems can achieve efficient, deterministic, and reliable operation while minimizing the potential for synchronization, priority inversion, responsiveness, resource management, and complexity issues

2.3.2 Enabling more than one receiving task

- A communication object can be accessed by any task or ISR that knows its handle (which might be a queue handle, semaphore handle, or event group handle). Any number of tasks and ISRs can process events or data sent to any given communication object.
- Task notifications are sent directly to the receiving task, so can only be processed by the task to which the notification is sent. However, this is rarely a limitation in practical cases because, while it is common to have multiple tasks and ISRs sending to the same communication object, it is rare to have multiple tasks and ISRs receiving from the same communication object.

2.3.3 Buffering multiple data items

- A queue is a communication object that can hold more than one data item at a time. Data that has been sent to the queue, but not yet received from the queue, is buffered inside the queue object. Task notifications send data to a task by updating the receiving task's notification value. A task's notification value can only hold one value at a time.

2.3.4 Broadcasting to more than one task

- An event group is a communication object that can be used to send an event to more than one task at a time.

Task notifications are sent directly to the receiving task, so can only be processed by the receiving task.

2.3.5 Waiting in the blocked state for a send to complete

- If a communication object is temporarily in a state that means no more data or events can be written to it (for example, when a queue is full no more data can be sent to the queue), then tasks attempting to write to the object can optionally enter the Blocked state to wait for their write operation to complete.
- If a task attempts to send a task notification to a task that already has a notification pending, then it is not possible for the sending task to wait in the Blocked state for the receiving task to reset its notification state. As will be seen, this is rarely a limitation in practical cases in which a task notification is used

3. Task Notification API Options

- Task notifications are a very powerful feature that can often be used in place of a binary semaphore, a counting semaphore, an event group, and sometimes even a queue. This wide range of usage scenarios can be achieved by using the xTaskNotify() API function to send a task notification, and the xTaskNotifyWait() API function to receive a task notification.
- However, in the majority of cases, the full flexibility provided by the xTaskNotify() and xTaskNotifyWait() API functions is not required, and simpler functions would suffice. Therefore, the **xTaskNotifyGive()** API function is provided as a simpler but less flexible alternative to **xTaskNotify()**, and the **ulTaskNotifyTake()** API function is provided as a simpler but less flexible alternative to **xTaskNotifyWait().**

3.1 xTaskNotifyGive()

- xTaskNotifyGive() sends a notification directly to a task, and increments (adds one to) the receiving task’s notification value.
- Calling xTaskNotifyGive() will set the receiving task’s notification state to pending, if it was not already pending.
- The xTaskNotifyGive() API function is provided to allow a task notification to be used as a lighter weight and faster alternative to a binary or counting semaphore

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

Parameter Name/ Returned Value	Description
xTaskToNotify	● The handle of the task to which the notification is being sent—see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.
Returned value	● xTaskNotifyGive() is a macro that calls xTaskNotify(). The parameters passed into xTaskNotify() by the macro are set such that pdPASS is the only possible return value. xTaskNotify() is described later in this book.

3.2 vTaskNotifyGiveFromISR()

- vTaskNotifyGiveFromISR() is a version of xTaskNotifyGive() that can be used in an interrupt service routine.

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,
BaseType_t *pxHigherPriorityTaskWoken );
```

Parameter Name/ Returned Value	Description
xTaskToNotify	● The handle of the task to which the notification is being sent—see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.
pxHigherPriorityTaskWoken	● If the task to which the notification is being sent is waiting in the Blocked state to receive a

	notification, then sending the notification will cause the task to leave the Blocked state. ● If calling vTaskNotifyGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the priority of the currently executing task (the task that was interrupted), then, internally, vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. ● If vTaskNotifyGiveFromISR() sets this value to pdTRUE, then 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. ● As with all interrupt safe API functions, the pxHigherPriorityTaskWoken parameter must be set to pdFALSE before it is used

3.3 ulTaskNotifyTake()

- ulTaskNotifyTake() allows a task to wait in the Blocked state for its notification value to be greater than zero, and either decrements (subtracts one from) or clears the task's notification value before it returns.
- The ulTaskNotifyTake() API function is provided to allow a task notification to be used as a lighter weight and faster alternative to a binary or counting semaphore.

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t
xTicksToWait );
```

Parameter Name/ Returned Value	Description
xClearCountOnExit	● If xClearCountOnExit is set to pdTRUE, then the calling task's notification value will be cleared to zero before the call to ulTaskNotifyTake() returns. ● After calling ulTaskNotifyTake() with

	xClearCountOnExit set to pdTRUE, regardless of the number of pending notifications, the task's notification count will be cleared to zero. ● If xClearCountOnExit is set to pdFALSE, and the calling task’s notification value is greater than zero, then the calling task’s notification value will be decremented before the call to ulTaskNotifyTake() returns. ● If the task's notification count is greater than zero when ulTaskNotifyTake() is called with xClearCountOnExit set to pdFALSE, one notification will be consumed, and the count will be decremented by one. ● Subsequent calls to ulTaskNotifyTake() will continue to consume notifications one by one until the count reaches zero, at which point the task will block until new notifications arrive.
xTicksToWait	● The maximum amount of time the calling task should remain in the Blocked state to wait for its notification value to be greater than zero. ● The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. ● Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	● The returned value is the calling task’s notification value before it was either cleared to zero or decremented, as specified by the value of the xClearCountOnExit parameter. ● If a block time was specified (xTicksToWait was

	not zero), and the return value is not zero, then it is possible the calling task was placed into the Blocked state, to wait for its notification value to be greater than zero, and its notification value was updated before the block time expired. ● If a block time was specified (xTicksToWait was not zero), and the return value is zero, then the calling task was placed into the Blocked state, to wait for its notification value to be greater than zero, but the specified block time expired before that happened

3.4 xTaskNotify() and xTaskNotifyFromISR()

- xTaskNotify() is more flexible and powerful than xTaskNotifyGive(), and because of that extra flexibility and power, it is also a little more complex to use.
- xTaskNotifyFromISR() is a version of xTaskNotify() that can be used in an interrupt service routine, and therefore has an additional pxHigherPriorityTaskWoken parameter.
- Calling xTaskNotify() will always set the receiving task's notification state to pending, if it was not already pending.

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify, uint32_t
ulValue, eNotifyAction eAction ); BaseType_t

xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue,
eNotifyAction eAction, BaseType_t *pxHigherPriorityTaskWoken );
```

- The task notification value is a 32-bit unsigned integer (uint32_t). This provides a range of 0 to 4,294,967,295, which allows for a wide variety of uses such as counting events, setting bit flags, or storing other types of data.

Parameter Name/ Returned Value	Description
xTaskToNotify	● The handle of the task to which the notification is being sent—see the pxCreatedTask parameter of

	the xTaskCreate() API function for information on obtaining handles to tasks.
ulValue	● ulValue is used is dependent on the eNotifyAction value.
eNotifyAction	● An enumerated type that specifies how to update the receiving task’s notification value
Returned value	● xTaskNotify() will return pdPASS except in the one case

Table : xTaskNotify() eNotifyAction Parameter Values, and Their Resultant Effect on the Receiving Task’s Notification Value

eNotifyAction Value	Resultant Effect on Receiving Task
eNoAction	● The receiving task’s notification state is set to pending without it’s notification value being updated. The xTaskNotify() ulValue parameter is not used. ● The eNoAction action allows a task notification to be used as a faster and lighter weight alternative to a binary semaphore.obtaining handles to tasks.
eSetBits	● The receiving task’s notification value is bitwise OR’ed with the value passed in the xTaskNotify() ulValue parameter. For example, if ulValue is set to 0x01, then bit 0 will be set in the receiving task's notification value. As another example, if ulValue is 0x06 (binary 0110) then bit 1 and bit 2 will be set in the receiving task's notification value. ● The eSetBits action allows a task notification to be used as a faster and lighter weight alternative to an event group.
eIncrement	● The receiving task’s notification value is

	incremented. The xTaskNotify() ulValue parameter is not used. ● The eIncrement action allows a task notification to be used as a faster and lighter weight alternative to a binary or counting semaphore, and is equivalent to the simpler xTaskNotifyGive() API function.
eSetValueWithoutOverwrite	● If the receiving task had a notification pending before xTaskNotify() was called, then no action is taken and xTaskNotify() will return pdFAIL. ● If the receiving task did not have a notification pending before xTaskNotify() was called, then the receiving task's notification value is set to the value passed in the xTaskNotify() ulValue parameter.
eSetValueWithOverwrite	● The receiving task's notification value is set to the value passed in the xTaskNotify() ulValue parameter, regardless of whether the receiving task had a notification pending before xTaskNotify() was called or not.

3.4.1 xTaskNotify Functionalities

1. **Increment Notification Value:**
    - This functionality increments (adds one to) the receiving task's notification value. It is equivalent to xTaskNotifyGive().
    - Use Case: This is useful when you want to count events or signals. For example, if multiple events need to be processed by a task, each event can be notified by incrementing the notification value.

```
// Increment the notification value of xHandlerTask

xTaskNotify(xHandlerTask, 0, eIncrement);
```

2. **Set Bits in Notification Value:**
    - This sets one or more bits in the receiving task's notification value. It allows using the notification value as a lightweight and faster alternative to an event group.

- Use Case: This can be used to signal specific conditions or flags. Each bit in the notification value can represent a different event or condition that the task needs to handle.

  ```
  #define BIT_0 (1 << 0)

  #define BIT_1 (1 << 1)
  ```

```
// Set BIT_0 and BIT_1 in the notification value of
xHandlerTask
xTaskNotify(xHandlerTask, BIT_0 | BIT_1, eSetBits);
```

3. **Write a New Value if the Value Has Been Read:**
   - This writes a completely new number into the receiving task's notification value but only if the receiving task has read its notification value since it was last updated. This is similar to having a queue with a length of one.
   - Use Case: This ensures that the receiving task processes the current notification value before being updated with a new one, useful for state information or commands that must not be overwritten.

```
uint32_t newValue = 0xABCD1234;
// Write newValue to xHandlerTask's notification value only
if it has been read
if (xTaskNotify(xHandlerTask, newValue,
eSetValueWithoutOverwrite) != pdPASS) {
    // Handle the case where the previous value has not been
read
}
```

4. **Overwrite with a New Value:**
   - This writes a completely new number into the receiving task's notification value, even if the receiving task has not read its notification value since it was last updated. It provides functionality similar to xQueueOverwrite().
   - Use Case: This is useful for scenarios where the latest information is always the most relevant, such as updating sensor readings or state variables.

```
uint32_t newValue = 0xABCD1234;
// Overwrite xHandlerTask's notification value with newValue
```

```
    xTaskNotify(xHandlerTask, newValue, eSetValueWithOverwrite);
```

3.5 xTaskNotifyWait()

xTaskNotifyWait() is a more capable version of ulTaskNotifyTake(). It allows a task to wait, with an optional timeout, for the calling task's notification state to become pending, should it not already be pending. xTaskNotifyWait() provides options for bits to be cleared in the calling task's notification value both on entry to the function, and on exit from the function.

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                           uint32_t ulBitsToClearOnExit,
                           uint32_t *pulNotificationValue,
                           TickType_t xTicksToWait );
```

Parameter Name/ Returned Value	Description
ulBitsToClearOnEntry	● If the calling task did not have a notification pending before it called xTaskNotifyWait(), then any bits set in ulBitsToClearOnEntry will be cleared in the task's notification value on entry to the function. ● For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared. As another example, setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0
ulBitsToClearOnExit	● If the calling task exits xTaskNotifyWait() because it received a notification, or because it already had a notification pending when xTaskNotifyWait() was called, then any bits set in ulBitsToClearOnExit will be cleared in the task's notification value before the task exits the xTaskNotifyWait() function. ● The bits are cleared after the task's notification

	value has been saved in *pulNotificationValue (see the description of pulNotificationValue below). ● For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits. ● Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0
pulNotificationValue	● Used to pass out the task's notification value. The value copied to *pulNotificationValue is the task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting. ● pulNotificationValue is an optional parameter and can be set to NULL if it is not required.
xTicksToWait	● The maximum amount of time the calling task should remain in the Blocked state to wait for its notification state to become pending. ● The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. ● Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h
Returned value	There are two possible return values: **1. pdTRUE** ● This indicates xTaskNotifyWait() returned because a notification was received, or because the calling task already had a notification pending when xTaskNotifyWait() was called.

	● If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for its notification state to become pending, but its notification state was set to pending before the block time expired. **2. pdFALSE** ● This indicates that xTaskNotifyWait() returned without the calling task receiving a task notification. ● If xTicksToWait was not zero then the calling task will have been held in the Blocked state to wait for its notification state to become pending, but the specified block time expired before that happened.

4. Example codes

4.1 use of task notification

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h>

/* Function prototypes */
static void vPeriodicTask(void *pvParameters);
static void vHandlerTask(void *pvParameters);
static uint32_t ulExampleInterruptHandler(void);
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

/* Define the handler task handle */
TaskHandle_t xHandlerTask = NULL;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
```

```
}

void vApplicationIdleHook(void) { }

/* Main function */
int main(void) {
    /* Create the 'handler' task, which is the task to which
interrupt processing is deferred */
    xTaskCreate(vHandlerTask, "Handler", configMINIMAL_STACK_SIZE,
NULL, 3, &xHandlerTask);

    /* Create the task that will periodically generate a software
interrupt */
    xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);

    /* Start the scheduler */
    vTaskStartScheduler();

    /* As normal, the following line should never be reached */
    for (;;) {
    }
}

/* Periodic task function */
static void vPeriodicTask(void *pvParameters) {
    const TickType_t xDelay250ms = pdMS_TO_TICKS(500UL); // Shorter
delay

    /* Infinite loop */
    for (;;) {
        /* Print a message before generating the "interrupt" */
        vPrintString("Periodic task - About to generate an
interrupt.\r\n");

        /* Block until it is time to generate the "interrupt" again
*/
        vTaskDelay(xDelay250ms); // Shorter delay

        /* Generate the "interrupt" */
        ulExampleInterruptHandler();
        /* Print a message after generating the "interrupt" */
        vPrintString("Periodic task - Interrupt
generated.\r\n\r\n\r\n");
    }
}

/* Handler task function */
static void vHandlerTask(void *pvParameters) {
    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency +
pdMS_TO_TICKS( 100 );
```

```
 // Longer delay
    uint32_t ulEventsToProcess;

    /* Infinite loop */
    for (;;) {
        /* Wait to receive a notification sent directly to this task
from the interrupt service routine */
        ulEventsToProcess = ulTaskNotifyTake(pdTRUE,
xMaxExpectedBlockTime);

        /* Print a message before processing events */
        vPrintString("Handler task - About to process events.\r\n");

        /* Check if events were received */
        if (ulEventsToProcess != 0) {
            /* Process each event */
            while (ulEventsToProcess > 0) {
                vPrintString("Handler task - Processing
event.\r\n");
                ulEventsToProcess--;
            }
        } else {
            /* Handle error recovery operations if an interrupt did
not arrive within the expected time */
        }

        /* Print a message after processing events */
        vPrintString("Handler task - Events processed.\r\n");

    }
}


/* Interrupt handler function */
static uint32_t ulExampleInterruptHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Give the task notification to unblock the handler task */
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    /* Request a context switch if the handler task has higher
priority */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

    return 0;
}
```

Output:

```
Periodic task - About to generate an interrupt.

Handler task - About to process events.
Handler task - Processing event.
Handler task - Events processed.
Periodic task - Interrupt generated.


Periodic task - About to generate an interrupt.
Handler task - About to process events.
Handler task - Processing event.
Handler task - Events processed.
Periodic task - Interrupt generated.


Periodic task - About to generate an interrupt.
Handler task - About to process events.
Handler task - Processing event.
Handler task - Events processed.
Periodic task - Interrupt generated.
```

4.2 Multiple task notifications

```c
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h>

/* Function prototypes */
static void vPeriodicTask(void *pvParameters);
static void vHandlerTask(void *pvParameters);
static uint32_t ulExampleInterruptHandler(void);
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

/* Define the handler task handle */
TaskHandle_t xHandlerTask = NULL;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void) { }

/* Main function */
int main(void) {
    /* Create the 'handler' task, which is the task to which
interrupt processing is deferred */
    xTaskCreate(vHandlerTask, "Handler", configMINIMAL_STACK_SIZE,
NULL, 3, &xHandlerTask);

    /* Create the task that will periodically generate a software
interrupt */
    xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);

    /* Start the scheduler */
    vTaskStartScheduler();

    /* As normal, the following line should never be reached */
    for (;;) {
    }
}

/* Periodic task function */
static void vPeriodicTask(void *pvParameters) {
    const TickType_t xDelay250ms = pdMS_TO_TICKS(1000UL); // Shorter
delay

    /* Infinite loop */
    for (;;) {
        /* Print a message before generating the "interrupt" */
        vPrintString("Periodic task - About to generate an
interrupt.\r\n");

        /* Block until it is time to generate the "interrupt" again
*/
        vTaskDelay(xDelay250ms); // Shorter delay

        /* Generate the "interrupt" */
        ulExampleInterruptHandler();
        /* Print a message after generating the "interrupt" */
        vPrintString("Periodic task - Interrupt
generated.\r\n\r\n\r\n");
    }
}

/* Handler task function */
static void vHandlerTask(void *pvParameters) {
    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency +
pdMS_TO_TICKS( 100 ); // Longer delay
    uint32_t ulEventsToProcess;

    /* Infinite loop */
    for (;;) {
        /* Wait to receive a notification sent directly to this task
from the interrupt service routine */
        ulEventsToProcess = ulTaskNotifyTake(pdFALSE,
xMaxExpectedBlockTime);



        /* Check if events were received */
        if (ulEventsToProcess != 0) {
            /* Process each event */
                vPrintString("Handler task - Processing
event.\r\n");

        } else {
            /* Handle error recovery operations if an interrupt did
not arrive within the expected time */
        }

    }
}


/* Interrupt handler function */
static uint32_t ulExampleInterruptHandler(void) {
     BaseType_t xHigherPriorityTaskWoken;
     xHigherPriorityTaskWoken = pdFALSE;
     /* Send a notification to the handler task multiple times. The
first 'give' will
     unblock the task, the following 'gives' are to demonstrate that
the receiving
     task's notification value is being used to count (latch) events
- allowing the
     task to process each event in turn. */
     vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken
);
     vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken
);
     vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken
);
     portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    return 0;
}
```

Output:

```
Periodic task - About to generate an interrupt.

Handler task - Processing event.

Handler task - Processing event.

Handler task - Processing event.

Periodic task - Interrupt generated.
```

```
Periodic task - About to generate an interrupt.

Handler task - Processing event.

Handler task - Processing event.

Handler task - Processing event.

Periodic task - Interrupt generated.
```

4.3 TaskNotify() - eAction parameters

```c
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h>

TaskHandle_t xHandlerTask = NULL;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void)
{
}


static void vPeriodicTask(void *pvParameters) {
    const TickType_t xDelay = pdMS_TO_TICKS(1000UL);

    for (;;) {
        vPrintString("Periodic task - About to notify the handler
task.\r\n");
```

```
        // Increment the notification value
        xTaskNotify(xHandlerTask, 0, eIncrement);
        vTaskDelay(xDelay);

        // Set specific bits (bit 0 and bit 1)
        xTaskNotify(xHandlerTask, 1 | 2, eSetBits);
        vTaskDelay(xDelay);

        // Set the notification value to a new value (10)
        xTaskNotify(xHandlerTask, 10, eSetValueWithOverwrite);
        vTaskDelay(xDelay);

        // Set the notification value to a new value only if not
updated (5)
        xTaskNotify(xHandlerTask, 5, eSetValueWithoutOverwrite);
        vTaskDelay(xDelay);

        vPrintString("Periodic task - Notifications sent.\r\n");
    }
}


static void vHandlerTask(void *pvParameters) {
    uint32_t ulNotificationValue;
    BaseType_t xResult;
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS(2000);

    for (;;) {
        // Wait for a notification value to be received
        xResult = xTaskNotifyWait(0x00, 0xFFFFFFFF,
&ulNotificationValue, xMaxBlockTime);

        if (xResult == pdPASS) {
            vPrintString("Handler task - Notification received with
value: %u\r\n", ulNotificationValue);

            // Check if the notification was an increment
            if (ulNotificationValue == 1) {
                vPrintString("Handler task - Increment notification
received.\r\n");
            }

            // Check specific bits
            if (ulNotificationValue == 3)
            {
                vPrintString("Handler task - Bit 0 is set.\r\n");
                vPrintString("Handler task - Bit 1 is set.\r\n");
            }

            // Check if notification value was set with overwrite
            if (ulNotificationValue == 10) {
                vPrintString("Handler task - Notification value set
with overwrite: %u\r\n", ulNotificationValue);
            }

            // Check if notification value was set without overwrite
            if (ulNotificationValue == 5) {
                vPrintString("Handler task - Notification value set
without overwrite: %u\r\n", ulNotificationValue);
            }

            // Process the notification value
            vPrintString("Handler task - Processing notification
value: %u\r\n", ulNotificationValue);
        } else {
            vPrintString("Handler task - No notification received
within the block time.\r\n");
        }
    }
}


int main(void) {
    xTaskCreate(vHandlerTask, "Handler", configMINIMAL_STACK_SIZE,
NULL, 3, &xHandlerTask);
    xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);
    vTaskStartScheduler();

    for (;;) {
    }
}
```

Output:

```
Periodic task - About to notify the handler task.

Handler task - Notification received with value: 1

Handler task - Increment notification received.

Handler task - Processing notification value: 1

Handler task - Notification received with value: 3

Handler task - Bit 0 is set.

Handler task - Bit 1 is set.

Handler task - Processing notification value: 3

Handler task - Notification received with value: 10

Handler task - Notification value set with overwrite: 10
Handler task - Processing notification value: 10
Handler task - Notification received with value: 5
Handler task - Notification value set without overwrite: 5
Handler task - Processing notification value: 5
Periodic task - Notifications sent.
Periodic task - About to notify the handler task.
Handler task - Notification received with value: 1
Handler task - Increment notification received.
Handler task - Processing notification value: 1
Handler task - Notification received with value: 3
Handler task - Bit 0 is set.
Handler task - Bit 1 is set.
Handler task - Processing notification value: 3
Handler task - Notification received with value: 10
Handler task - Notification value set with overwrite: 10
Handler task - Processing notification value: 10
Handler task - Notification received with value: 5
Handler task - Notification value set without overwrite: 5
Handler task - Processing notification value: 5
```