ECE 4DS4 – Embedded Systems

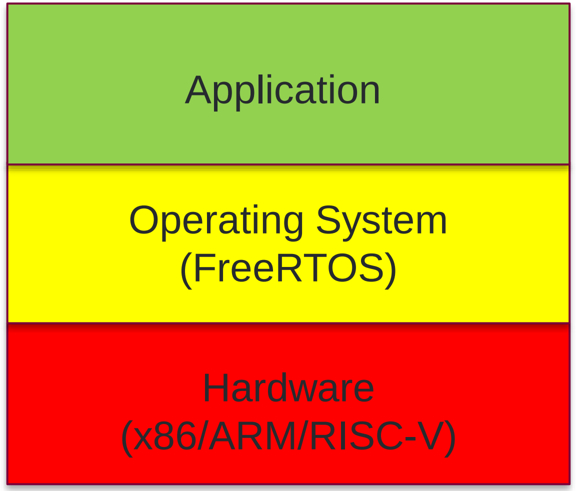
LAB 2 TUTORIAL FreeRTOS

Hazem Sharf

06-02-2024

Operating System (OS)

- Operating system is an intermediary layer between computer hardware and user application. And it offers the following functions:
  - managing memory
  - scheduling tasks
  - controlling input/output devices
  - providing a user interface
- The two Categories of Operating systems are:
  - General Purpose Operating System (GPOS)
  - Real-Time Operating System (RTOS)



RTOS vs GPOS

- Key differences between RTOS and GPOS are:
  - Determinism
    - RTOS is deterministic, ensuring tasks are executed within predictable timeframes
    - GPOS is non-deterministic focusing on high throughput of resource sharing
  - Priority based scheduling vs Dynamic scheduling
    - RTOS use priority based scheduling to meet real-time requirements, whereas GPOS use dynamic scheduling to optimize overall system performance and responsiveness
- Examples of GPOS
  - Windows
  - Linux
- Examples of RTOS
  - FreeRTOS
  - Zephyr
  - RIOT

RTOS

- A Real-Time Operating System (RTOS) is an operating system specifically designed to meet the requirements of real-time systems. RTOS must ensure
  - Correctness of system operation
  - Completing tasks within strict timing constraints
- Use cases include time-critical tasks, such as:
  - Embedded systems
  - Control systems
  - Aerospace applications

freeRTOS

RIOT

Zephyr®

Important Terminologies in RTOS

- Interrupt Service Routine
- Scheduling
- Context switching
- Dispatcher
- Task
- Idle Task
- Priority
- Preemption

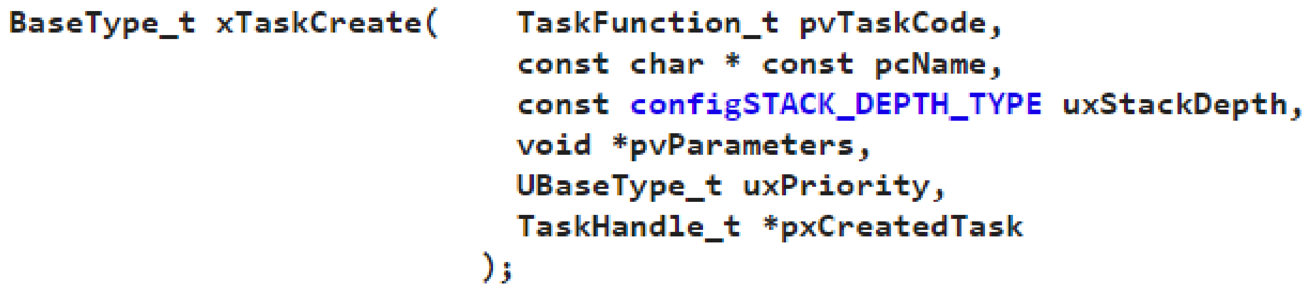
Scheduling

- Responsible for determining when to run which tasks.
- Allocating the available resources such as CPU time.
- Scheduler must meet timing constraints of each task.
- Scheduler uses a scheduling algorithm:
- Types of Schedulers include:
  - Static
    - Fixed-priority
    - Round-robin
  - Dynamic
    - Earliest deadline first (EDF)
    - Rate-monotonic: priorities based on periodicity

Hello World Task

```c
int main(void)
{
    BaseType_t status;

    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    status = xTaskCreate(hello_task, "Hello_task", 200, NULL, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    vTaskStartScheduler();
    while(1)
    {}
}
```

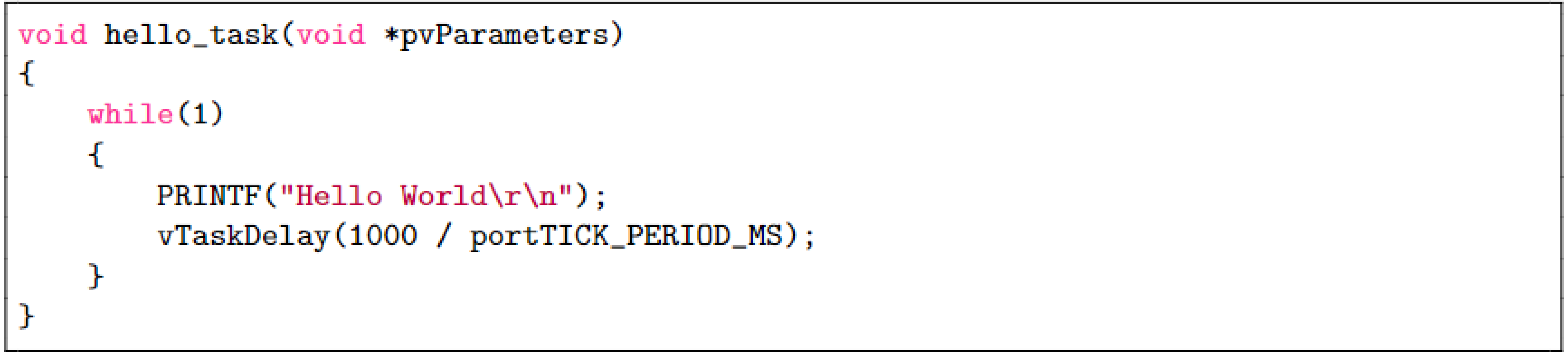
Hello World Task



```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           const configSTACK_DEPTH_TYPE uxStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                         );
```

- pvTaskCode: Pointer to the task entry function
- pcName: descriptive name
- uxStackDepth: The number of words
- pvParameters: A value that is passed as the paramater to the created task
- uxPriority: priority of task
- *pxCreatedTask:* handle to the created task

Hello World Task

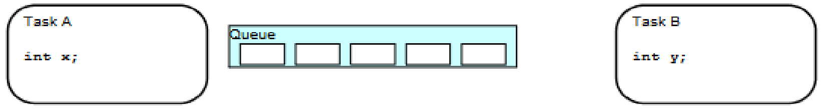


```
void hello_task(void *pvParameters)
{
    while(1)
    {
        PRINTF("Hello World\r\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

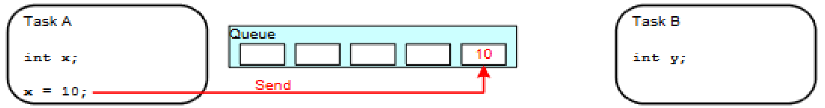
Inter-Task Communication and Synchronization

- Synchronized access to shared resources is necessary to ensure that the resources are accessed in a consistent and predictable manner.
- Tasks may execute concurrently and access the same resources.
- If two or more tasks try to access the same resource simultaneously, the result can be data corruption, unpredictable behavior, or other errors.
- Methods for synchronizing tasks and inter-Task communication:
  - Queues
  - Semaphores
  - Event Groups

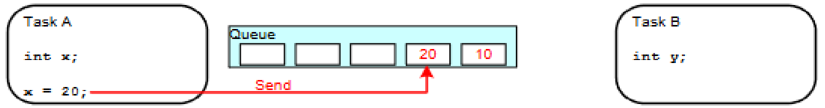
Queues



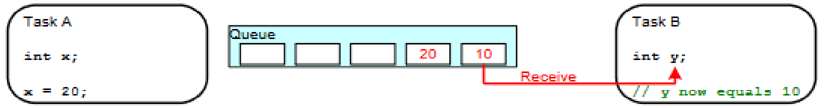
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



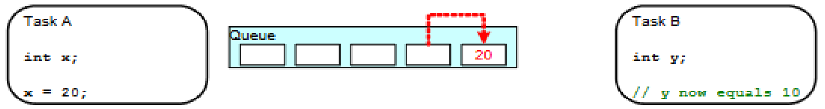
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



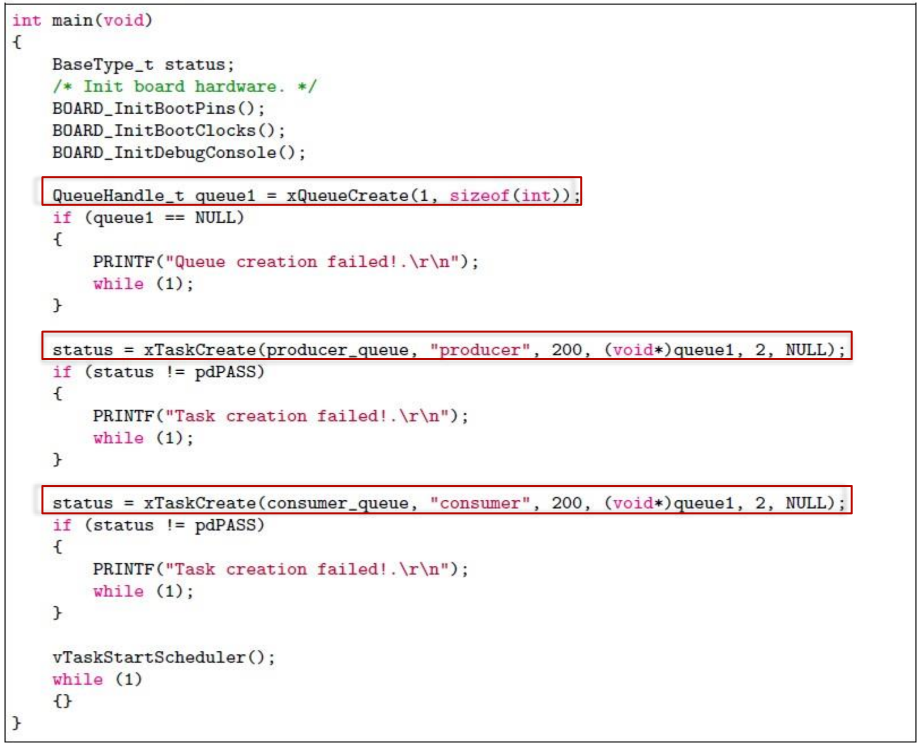
Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

McMaster University

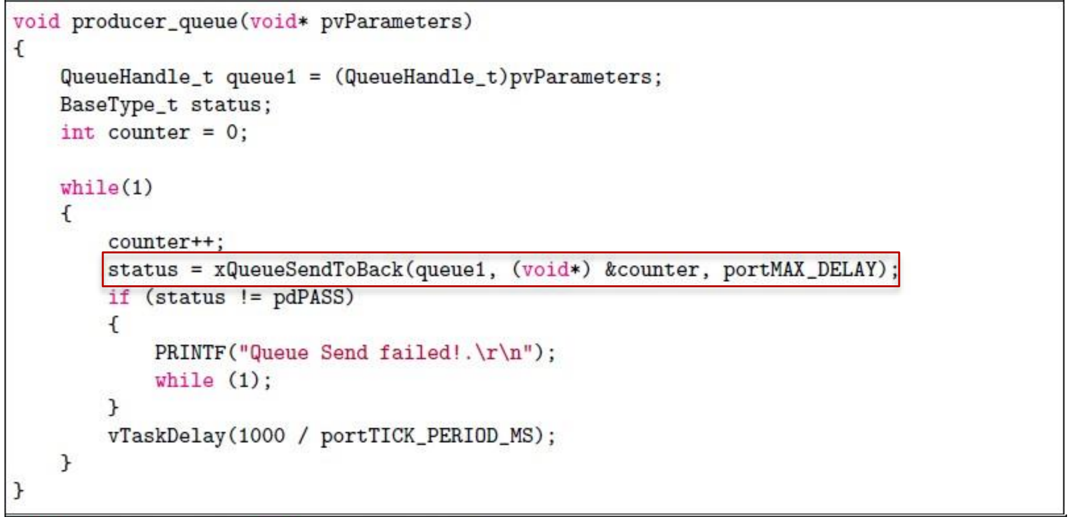
Queues



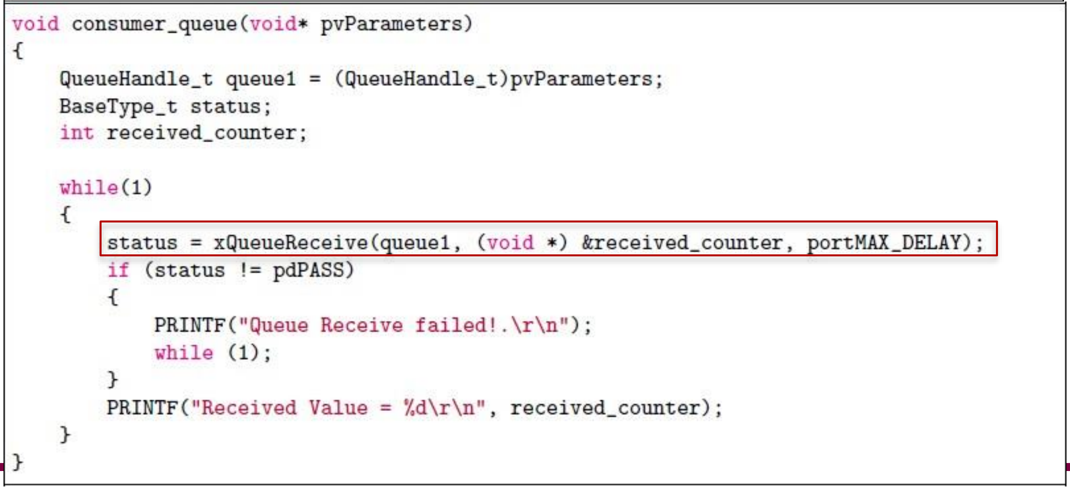
```
int main(void)
{
    BaseType_t status;
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    QueueHandle_t queue1 = xQueueCreate(1, sizeof(int));
    if (queue1 == NULL)
    {
        PRINTF("Queue creation failed!.\r\n");
        while (1);
    }

    status = xTaskCreate(producer_queue, "producer", 200, (void*)queue1, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    status = xTaskCreate(consumer_queue, "consumer", 200, (void*)queue1, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    vTaskStartScheduler();
    while (1)
    {}
}
```

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

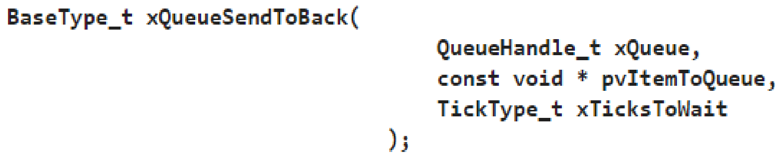
- uxQueueLength: number of items in queue
- uxItemSize: Item size



```c
void producer_queue(void* pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    int counter = 0;

    while(1)
    {
        counter++;
        status = xQueueSendToBack(queue1, (void*) &counter, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Send failed!.\r\n");
            while (1);
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```



```c
void consumer_queue(void* pvParameters)
{
    QueueHandle_t queue1 = (QueueHandle_t)pvParameters;
    BaseType_t status;
    int received_counter;

    while(1)
    {
        status = xQueueReceive(queue1, (void *) &received_counter, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Receive failed!.\r\n");
            while (1);
        }
        PRINTF("Received Value = %d\r\n", received_counter);
    }
}
```



```c
BaseType_t xQueueSendToBack(
                                    QueueHandle_t xQueue,
                                    const void * pvItemToQueue,
                                    TickType_t xTicksToWait
                                );
```

- xQueue: The handle to the queue
- pvItemToQueue: Pointer to the item in queue
- xTicksToWait: amount of time task is blocked until a place is available in the queue

Semaphore

- Semaphore is a counter.
- Counter is incremented when the resource is available and decremented when the resource is acquired by a task.

Two main operations:

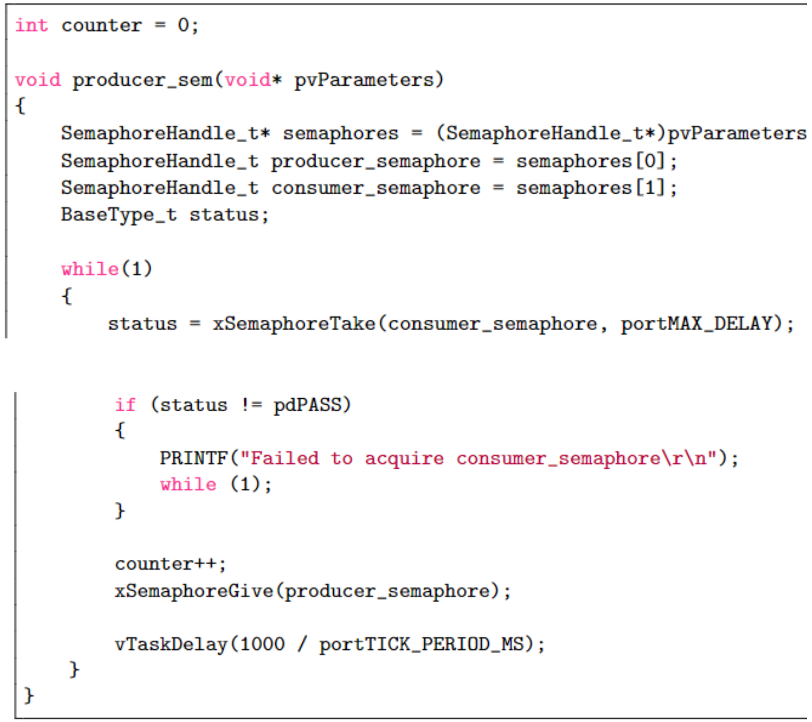
- Wait (or "acquire"): When a task wants to access a shared resource, it must first wait on the semaphore. If the semaphore's counter is greater than zero, the task decrements the counter and acquires the resource. If the counter is zero, the task is blocked until the resource becomes available.
- Signal (or "release"): When a task has finished using a shared resource, it signals the semaphore by incrementing the counter. This makes the resource available to other tasks that may be waiting for it.

Types of Semaphores

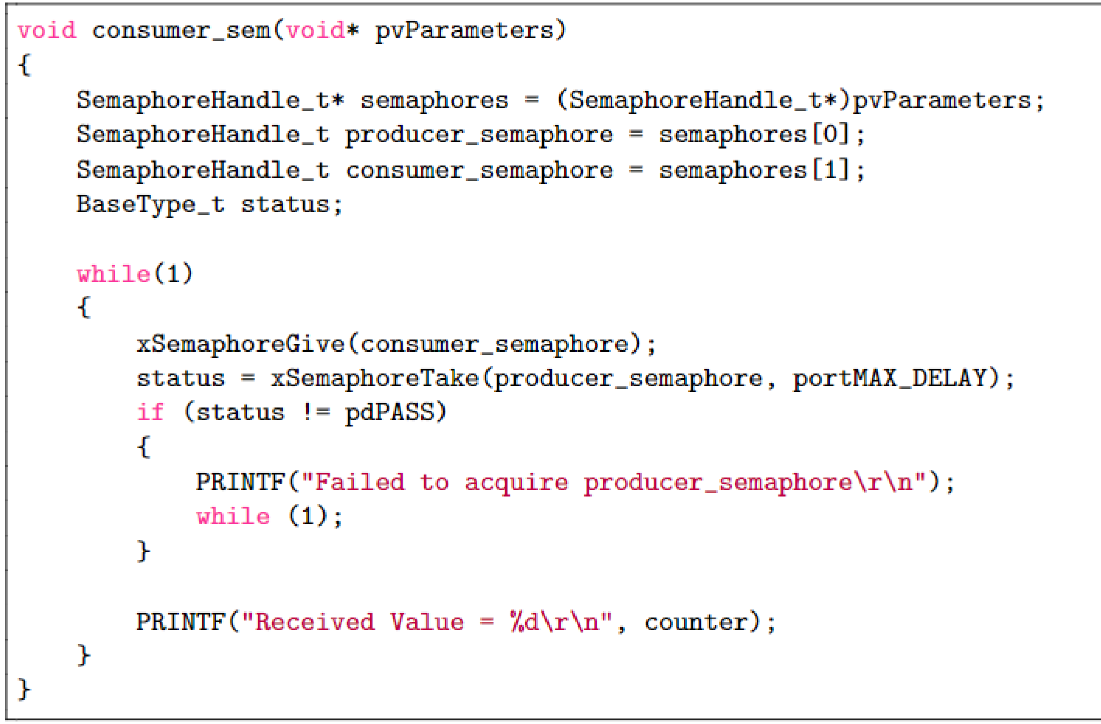
- Binary semaphores
- Counting semaphores

Semaphore

```c
int main(void)
{
    BaseType_t status;
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*) malloc(2 * sizeof(
    SemaphoreHandle_t));
    semaphores[0] = xSemaphoreCreateBinary(); //Producer semaphore
    semaphores[1] = xSemaphoreCreateBinary(); //Consumer semaphore

    status = xTaskCreate(producer_sem, "producer", 200, (void*)semaphores, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    status = xTaskCreate(consumer_sem, "consumer", 200, (void*)semaphores, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1);
    }

    vTaskStartScheduler();
    while (1)
    {}
}
```



```c
int counter = 0;

void producer_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters
    SemaphoreHandle_t producer_semaphore = semaphores[0];
    SemaphoreHandle_t consumer_semaphore = semaphores[1];
    BaseType_t status;

    while(1)
    {
        status = xSemaphoreTake(consumer_semaphore, portMAX_DELAY);


        if (status != pdPASS)
        {
            PRINTF("Failed to acquire consumer_semaphore\r\n");
            while (1);
        }

        counter++;
        xSemaphoreGive(producer_semaphore);

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

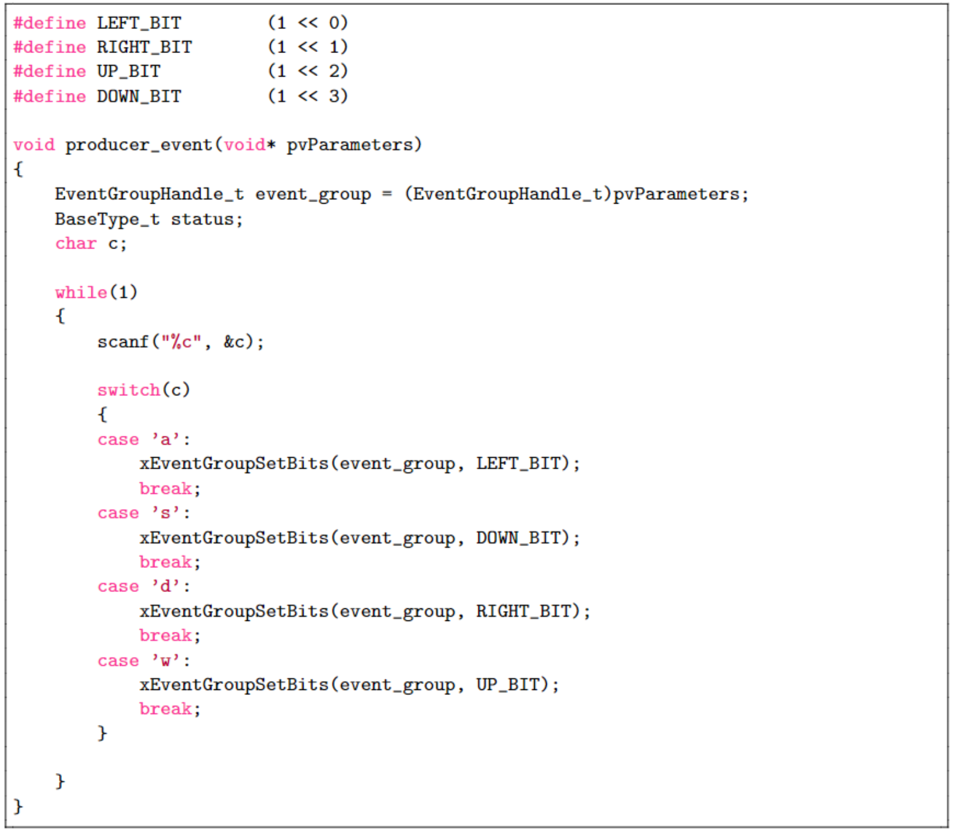


```c
void consumer_sem(void* pvParameters)
{
    SemaphoreHandle_t* semaphores = (SemaphoreHandle_t*)pvParameters;
    SemaphoreHandle_t producer_semaphore = semaphores[0];
    SemaphoreHandle_t consumer_semaphore = semaphores[1];
    BaseType_t status;

    while(1)
    {
        xSemaphoreGive(consumer_semaphore);
        status = xSemaphoreTake(producer_semaphore, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Failed to acquire producer_semaphore\r\n");
            while (1);
        }

        PRINTF("Received Value = %d\r\n", counter);
    }
}
```

Event Groups

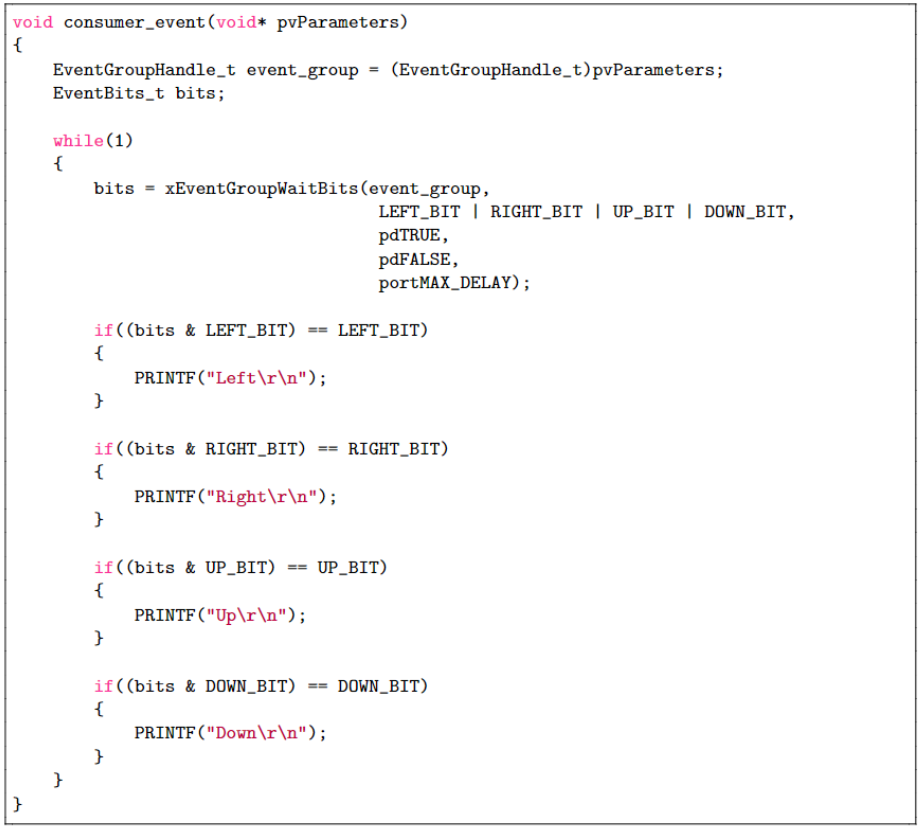
- Provide a way for tasks to wait for one or more specific events to occur.
- Each event is represented by a bit in the event group.
- When a task waits for an event, it is blocked until the event occurs.
- When the event occurs, the RTOS sets the corresponding bit in the event group and unblocks the task.

Event Groups



```
#define LEFT_BIT        (1 << 0)
#define RIGHT_BIT       (1 << 1)
#define UP_BIT          (1 << 2)
#define DOWN_BIT        (1 << 3)

void producer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    BaseType_t status;
    char c;

    while(1)
    {
        scanf("%c", &c);

        switch(c)
        {
        case ’a’:
            xEventGroupSetBits(event_group, LEFT_BIT);
            break;
        case ’s’:
            xEventGroupSetBits(event_group, DOWN_BIT);
            break;
        case ’d’:
            xEventGroupSetBits(event_group, RIGHT_BIT);
            break;
        case ’w’:
            xEventGroupSetBits(event_group, UP_BIT);
            break;
        }

    }
}
```

Event Groups



```
void consumer_event(void* pvParameters)
{
    EventGroupHandle_t event_group = (EventGroupHandle_t)pvParameters;
    EventBits_t bits;

    while(1)
    {
        bits = xEventGroupWaitBits(event_group,
                                    LEFT_BIT | RIGHT_BIT | UP_BIT | DOWN_BIT,
                                    pdTRUE,
                                    pdFALSE,
                                    portMAX_DELAY);

        if((bits & LEFT_BIT) == LEFT_BIT)
        {
            PRINTF("Left\r\n");
        }

        if((bits & RIGHT_BIT) == RIGHT_BIT)
        {
            PRINTF("Right\r\n");
        }

        if((bits & UP_BIT) == UP_BIT)
        {
            PRINTF("Up\r\n");
        }

        if((bits & DOWN_BIT) == DOWN_BIT)
        {
            PRINTF("Down\r\n");
        }
    }
}
```

```
EventBits_t xEventGroupWaitBits(
                        const EventGroupHandle_t xEventGroup,
                        const EventBits_t uxBitsToWaitFor,
                        const BaseType_t xClearOnExit,
                        const BaseType_t xWaitForAllBits,
                        TickType_t xTicksToWait );
```

- xEventGroup: The event group
- uxBitsToWaitFor: bits to be tested
- *xClearOnExit*: clearing the bits in the event group
- *xWaitForAllBits: wait for all bits to be set*
- *xTicksToWait:* amount of time task is blocked until a place is available in the queue

Timers

- Timers are used to track the passage of time.
- Timers can be used to schedule tasks to run at specific times, or to trigger an action after a certain time period has elapsed.

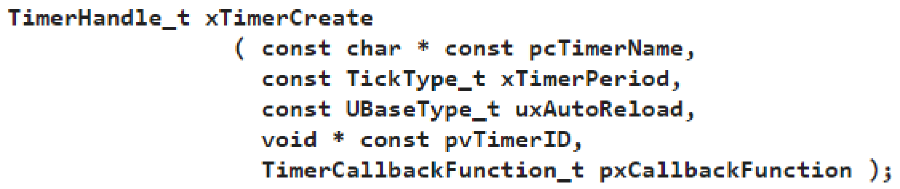
Callback Functions

- It is called when a specific event occurs.
- For example: When a timer expires.
- The callback function can then perform the necessary actions to run the task.

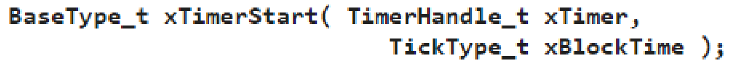
FreeRTOS Timers:

- One shot timer
- Periodic Timer

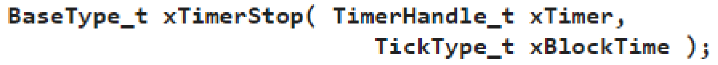
Timers



```
TimerHandle_t xTimerCreate
                ( const char * const pcTimerName,
                  const TickType_t xTimerPeriod,
                  const UBaseType_t uxAutoReload,
                  void * const pvTimerID,
                  TimerCallbackFunction_t pxCallbackFunction );
```



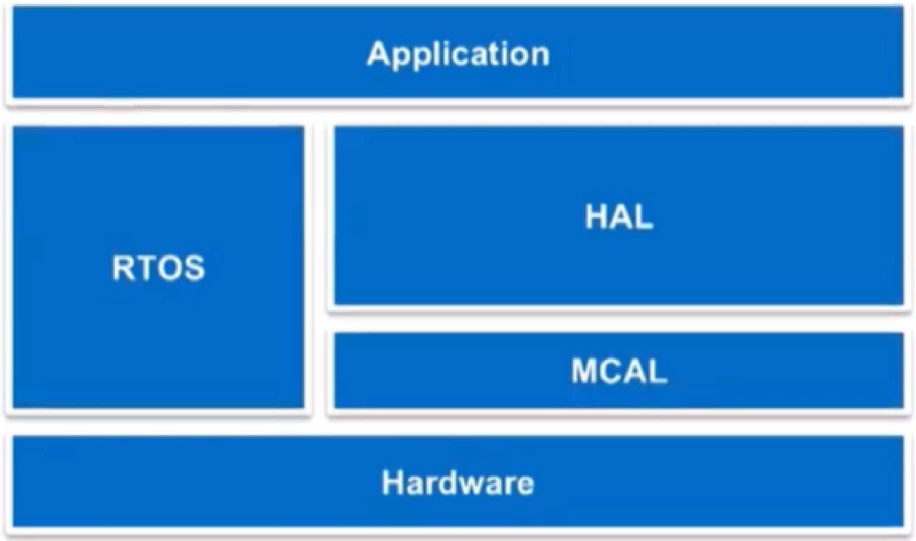
```
BaseType_t xTimerStart( TimerHandle_t xTimer,
                         TickType_t xBlockTime );
```



```
BaseType_t xTimerStop( TimerHandle_t xTimer,
                        TickType_t xBlockTime );
```

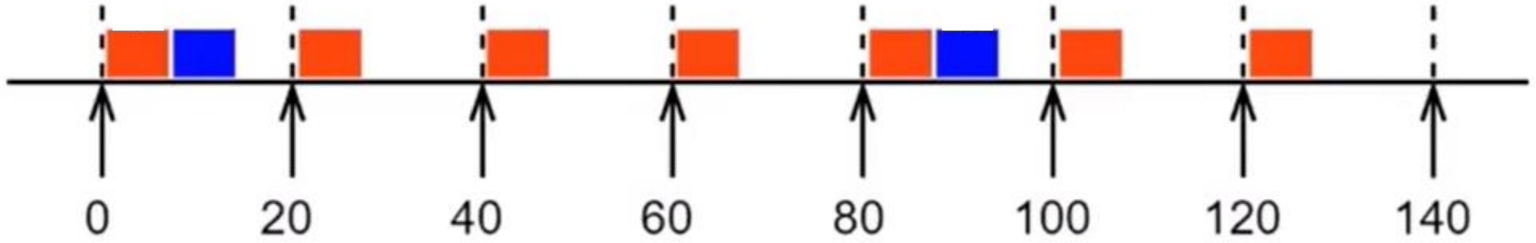
Thank You!

Real-Time operating systems (RTOS)



Scheduling example

- T1{P:20, E:5, D:20}, T2{P:80, E:5, D=80}


0
20
40
60
80
100
120
140