HCMC University Of Technology, VNU-HCM
Faculty of Computer Science & Engineering

– Embedded System Lab 04 –

FreeRTOS Queue Management

Pham Hoang Anh & **Huynh Hoang Kha**

Goal In this lab, students are expected to understand and can use FreeRTOS queue for tasks intercommunication.

Content

- Characteristics of a Queue
- Using a queue
- Receiving data from multiple sources

Prerequisite Requirement

- Have basic knowledge of OS and micro-controller programming.
- Understand queue data structure.
- Finished previous lab.

Grading policy

- 40% in-class performance
- 60% report submission

1 Queue's Characteristics

Queues provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism. Only task-to-task communication is covered in this section. A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue. Figure 1 demonstrates data being written to and read from a queue that is being used as a FIFO. It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue.

There are two ways in which queue behavior could have been implemented:

1. Queuing by copy means the data sent to the queue is copied byte for byte into the queue.

2. Queuing by reference means the queue only holds pointers to the data sent to the queue, not the data itself.

FreeRTOS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queueing by reference because:

- Stack variable can be sent directly to a queue, even though the variable will not exist after the function in which it is declared has exited.

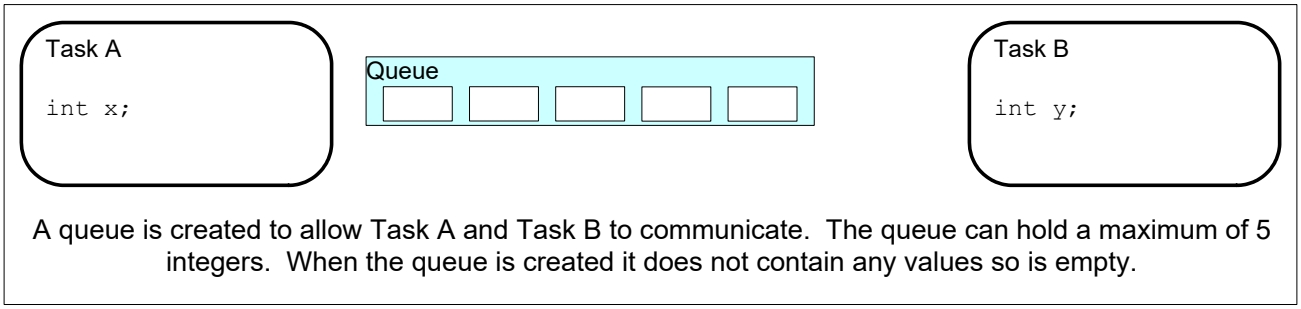
- Data can be sent to a queue without first allocating a buffer to hold the data, and then copying the data into the allocated buffer.

- The sending task can immediately re-use the variable or buffer that was sent to the queue.

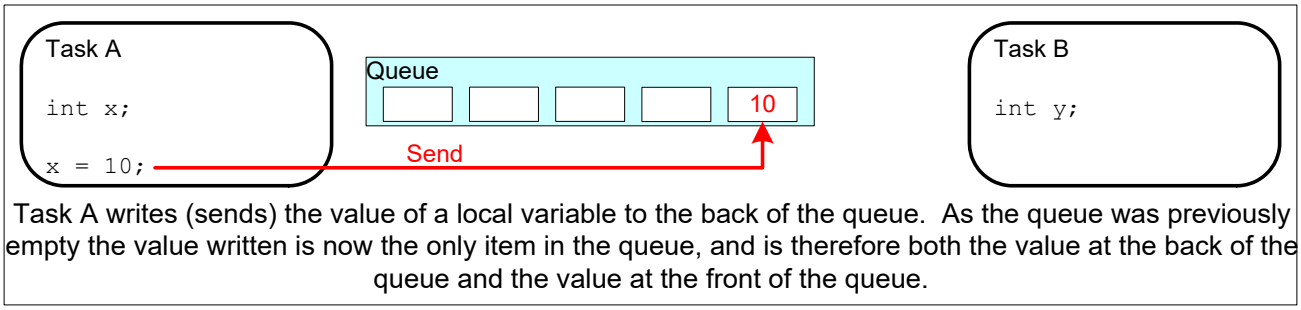
- The sending task and the receiving task are completely de-coupled—the application designer does not need to concern themselves with which task 'owns' the data, or which task is responsible for releasing the data.

- Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue instead.

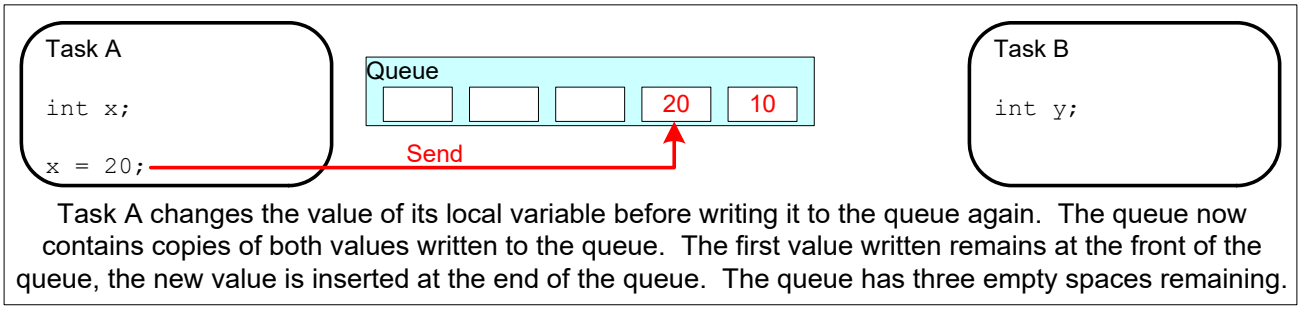
- The RTOS takes complete responsibility for allocating the memory used to store data.



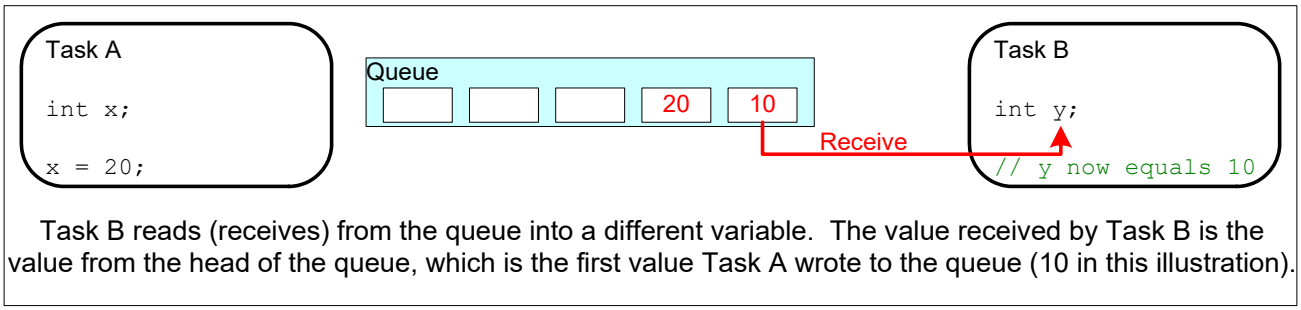
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



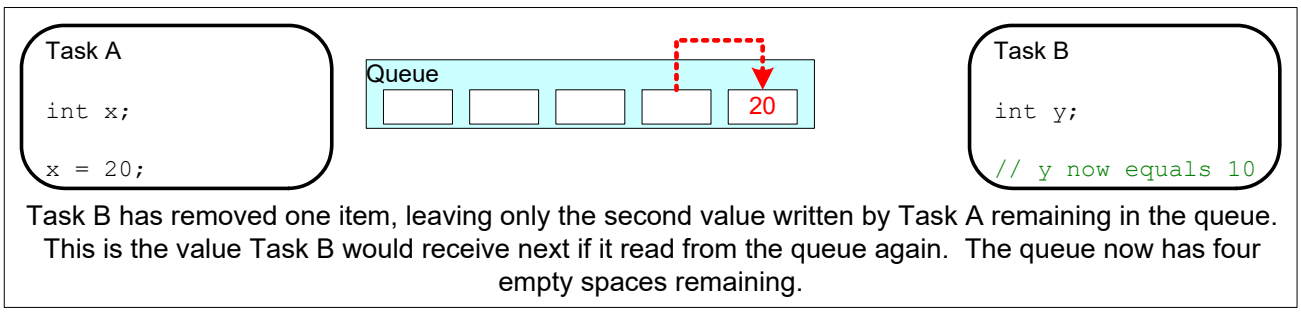
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



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.

Figure 1: An example sequence of writes to, and reads from a queue

- In a memory protected system, the RAM that a task can access will be restricted. In that case queueing by reference could only be used if the sending and receiving task could both access the RAM in which the data was stored. Queuing by copy does not impose that restriction; the kernel always runs with full privileges, allowing a queue to be used to pass data across memory protection boundaries.

Access by Multiple Tasks

Queues are objects in their own right that can be accessed by any task or ISR that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

Blocking on Queue Reads

When a task attempts to read from a queue, it can optionally specify a 'block' time. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty. A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue. The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data. If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked.

Blocking on Queue Writes

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task

that is waiting for space. If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked,

Blocking on Multiple Queues

Queues can be grouped into sets, allowing a task to enter the Blocked state to wait for data to become available on any of the queues in the set.

2 Using a queue

The `xQueueCreate()` API Function

Queues must be explicitly created before it can be used. Queues are referenced by handlers which are variables of type `QueueHandle_t`. The `xQueueCreate()` API function creates a queue and returns a `QueueHandle_t` that references the queue it created.

FreeRTOS V9.0.0 also includes the `xQueueCreateStatic()` function, which allocates the memory required to create a queue statically at compile time: FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. The RAM is used to hold both the queue data structures and the items that are contained in the queue. `xQueueCreate()` will return `NULL` if there is insufficient heap RAM available for the queue to be created.

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

Code 1: The `xQueueCreate()` API function prototype

- `uxQueueLength`: The maximum number of items that the queue being created can hold at any one time.
- `uxItemSize`: The size in bytes of each data item that can be stored in the queue.
- Return Value: If `NULL` is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.
  A non-`NULL` value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

After a queue has been created the `xQueueReset()` API function can be used to return the queue to its original empty state.

The `xQueueSendToBack()` and `xQueueSendToFront()` API Functions

As might be expected, `xQueueSendToBack()` is used to send data to the back (tail) of a queue, and `xQueueSendToFront()` is used to send data to the front (head) of a queue.

`xQueueSend()` is equivalent to, and exactly the same as, `xQueueSendToBack()`.

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

Code 2: The `xQueueSendToFront()` API function prototype

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

Code 3: The `xQueueSendToBack()` API function prototype

The xQueueReceive() API Function

`xQueueReceive()` is used to receive (read) an item from a queue. The item that is received is removed from the queue.

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait );
```

Code 4: The `xQueueReceive()` API function prototype

Parameters and return value in Code 2, Code 3, and Code 4

- `xQueue`: The handle of the queue to which the data is being sent or read. The queue handle will have been returned from the call to `xQueueCreate()` used to create the queue.

- `pvItemToQueue`: A pointer to the data to be copied into the queue. The size of each item that the queue can hold is set when the queue is created.

- `xTicksToWait`: The maximum amount of time the task should remain in the Blocked state to wait.

  If `xTicksToWait` is zero and there is operation failure, those function will return immediately.

  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 into 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: `pdPASS` would be returned if operation is successfully. Else, `errQUEUE_FULL` or `errQUEUE_EMPTY` would be returned according to the kind of error.

```
1 UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);
```

Code 5: The `uxQueueMessagesWaiting()` API function prototype

In Code 5 above:

- `xQueue` The handle of the queue being queried. The queue handle will have been returned from the call to `xQueueCreate()` used to create the queue.
- Returned value: The number of items that the queue being queried is currently holding. If zero is returned, then the queue is empty.

3 Receiving data from multiple sources

It is common in FreeRTOS designs for a task to receive data from more than one source. The receiving task needs to know where the data came from to determine how the data should be processed. An easy design solution is to use a single queue to transfer structures with both the value of the data and the source of the data contained in the structure's fields. This scheme is demonstrated in Figure 2.

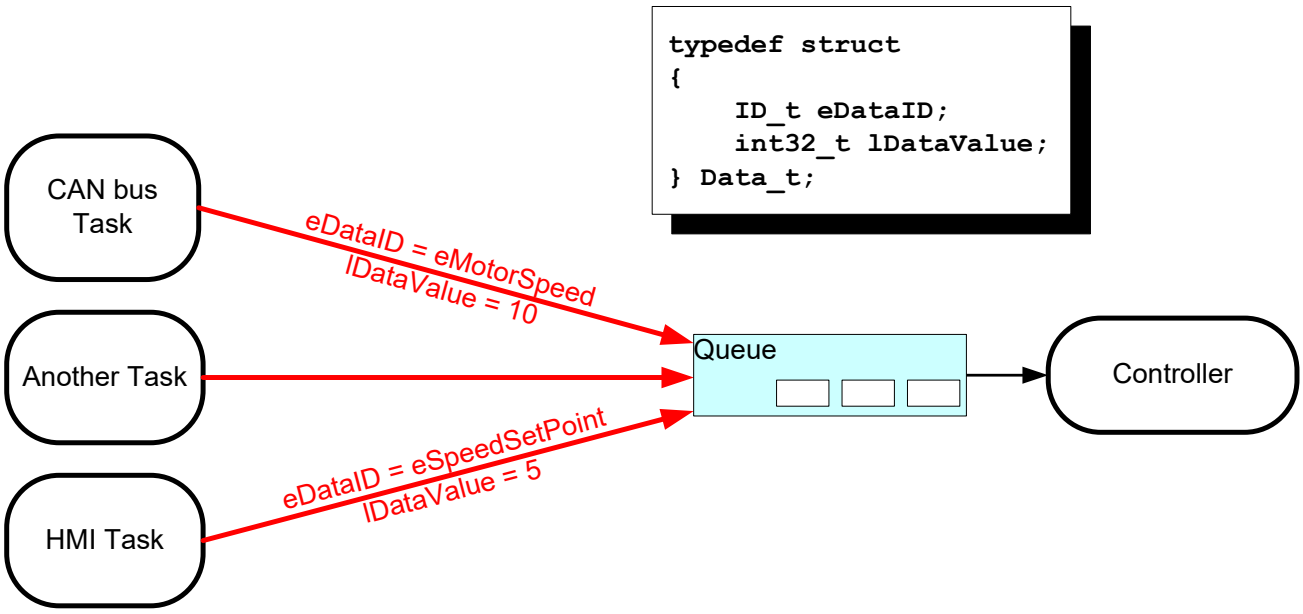


Figure 2: An example scenario where structures are sent on a queue

4 Exercises

Suppose that we are designing an application on the ESP32 development board in which there are one reception task and some functional tasks. The reception task is designed to receive requests from somewhere and classify these tasks and send them to a queue. Each functional task would check if the next request is for it or not. If yes, it will receive the request and handle it. Vice versa, it does nothing. If no functional task receives the request, raise an error and simply ignore that request. Students are asked to implement an example for the application model mentioned above.