Inter Task Communication

Mojtaba Bagherzadeh, Adrien Lapointe

Royal Military College (RMC)

mojtaba@cs.queensu.ca,adrien.lapointe@rmc.ca

February 25, 2018

Overview

1. Communication Types

2. Task to Task Communication

3. Mailbox Creation

4. Q & A

Communication Type

- Task to Task

Communication Type

- Task to Task
- Task to Interrupt Service Routine (ISR)

Communication Type

- Task to Task
- Task to Interrupt Service Routine (ISR)
- ISR to Task

Task to Task Communication

Queue in FreeRTOS

- A queue is a space that can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its length.
- 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.

Task to Task Communication

Queue in FreeRTOS

- A queue is a space that can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its length.
- 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.

Queue Implementation

- **Queue by copy:** The data sent to the queue is copied byte for byte into the queue. ***FreeRTOS uses the queue by copy method.***
- **Queue by reference:** Queues only holds pointers to the data sent to the queue, not the data itself.

Queue by Copy

Pros

- Stack variables can be sent directly to a queue, even though the variables will not exist after the related function has exited.
- The sending task can immediately re-use the variable or buffer that was sent to the queue.
- Queuing by copy does not prevent the queue from also being used to queue by reference.
- The RTOS takes complete responsibility for allocating the memory used to store data.

Queue by Copy

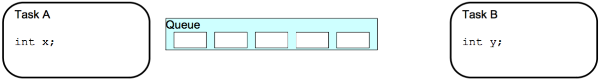
Pros

- Stack variables can be sent directly to a queue, even though the variables will not exist after the related function has exited.
- The sending task can immediately re-use the variable or buffer that was sent to the queue.
- Queuing by copy does not prevent the queue from also being used to queue by reference.
- The RTOS takes complete responsibility for allocating the memory used to store data.

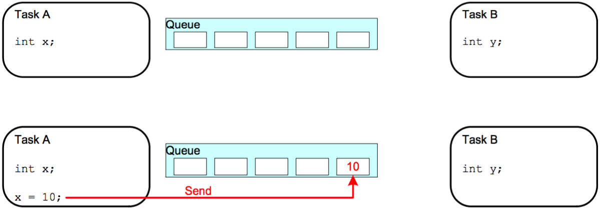
Cons

If the size of the data being stored in the queue is large, then it is preferable to use the queue to transfer pointers to the data, rather than copy the data itself into and out of the queue byte by byte.

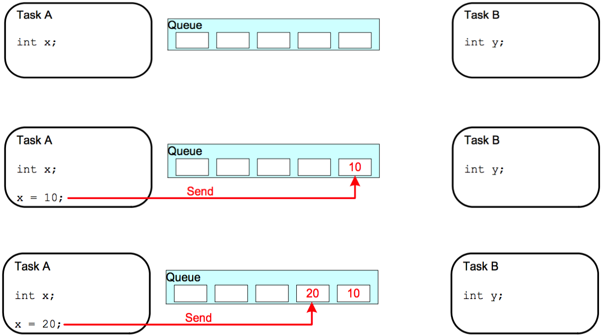
How Does Queue-based Communication work?



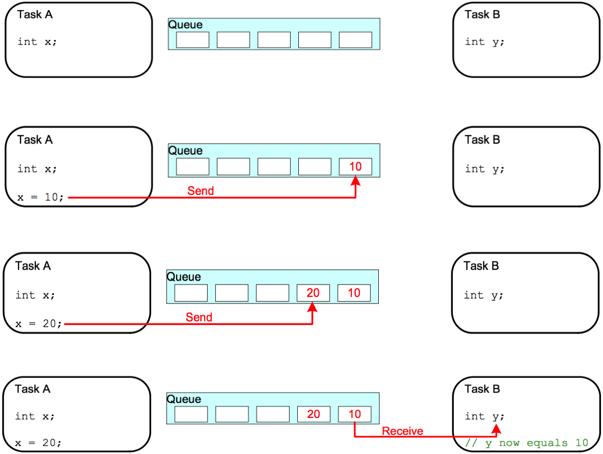
How Does Queue-based Communication work?



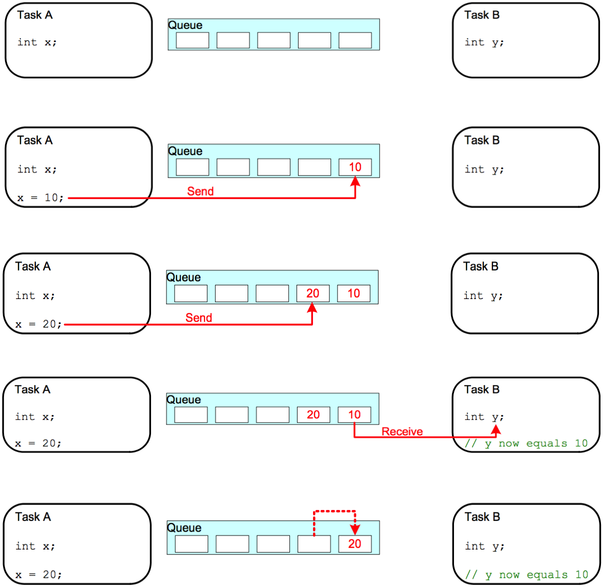
How Does Queue-based Communication work?



How Does Queue-based Communication work?



How Does Queue-based Communication work?



Create a Queue

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

Create a Queue

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

Argument	**Description**
uxQueueLength	the maximum number of items that the queue can hold at any time.
uxItemSize	the size in bytes of each data item that can be stored in the queue.

Create a Queue

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

Argument	**Description**
uxQueueLength	the maximum number of items that the queue can hold at any time.
uxItemSize	the size in bytes of each data item that can be stored in the queue.

Return values of the function:

- `NULL`: the queue has not been created successfully because of insufficient heap memory.
- `non-NULL`: the queue has been created and the queue handle is returned.

Blocking Read and Write

Blocking on Queue Reads

- A task can specify a 'block' time when reading from a queue. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue.
- The task be moved to the Ready state when data is written into the queue. It also is moved from the Blocked state to the Ready state if the specified block time expires before data becomes available.
- If a Queue has multiple readers, only one task (highest priority task) will be unblocked when data becomes available. If blocked tasks have equal priority, then the task that with longest waiting time will be unblocked.

Blocking on Queue Reads

Similarly, tasks can specify blocking time for writing in a queue.

Send a Message

```
xQueueSend*( QueueHandle_t xQueue, const void *
    pvItemToQueue, TickType_t xTicksToWait )
```

Description

`xQueueSendToBack()` sends data to the back (tail) of a queue.

`xQueueSendToFront()` sends data to the front (head) of a queue.

`xQueueSend()` is equivalent to `xQueueSendToBack()`.

Send a Message

```
xQueueSend*( QueueHandle_t xQueue, const void *
    pvItemToQueue, TickType_t xTicksToWait )
```

Description

`xQueueSendToBack()` sends data to the back (tail) of a queue.

`xQueueSendToFront()` sends data to the front (head) of a queue.

`xQueueSend()` is equivalent to `xQueueSendToBack()`.

Argument	**Description**
xQueue	the handle of the queue.
pvItemToQueue	a pointer to the data to be copied into the queue.
xTicksToWait	the maximum amount of time the task should remain in the Blocked state.

Send a Message

```
xQueueSend*( QueueHandle_t xQueue, const void *
    pvItemToQueue, TickType_t xTicksToWait )
```

Description

`xQueueSendToBack()` sends data to the back (tail) of a queue.

`xQueueSendToFront()` sends data to the front (head) of a queue.

`xQueueSend()` is equivalent to `xQueueSendToBack()`.

Argument	**Description**
xQueue	the handle of the queue.
pvItemToQueue	a pointer to the data to be copied into the queue.
xTicksToWait	the maximum amount of time the task should remain in the Blocked state.

Return values of the function:

- `pdPASS`: data was successfully sent to the queue.
- `errQUEUE_FULL`: data could not be written to the queue because the queue was already full.

Receive a Message

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

Description

`xQueueReceive()` receives (read) an item from a queue. ***The received item is removed from the queue.***

Receive a Message

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

Description

`xQueueReceive()` receives (read) an item from a queue. ***The received item is removed from the queue.***

Argument	**Description**
xQueue	the handle of the queue.
pvBuffer	a pointer to the memory into which the received data will be copied.
xTicksToWait	the maximum amount of time the task should remain in the Blocked state.

Receive a Message

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

Description

xQueueReceive() receives (read) an item from a queue. ***The received item is removed from the queue.***

Argument	**Description**
xQueue	the handle of the queue.
pvBuffer	a pointer to the memory into which the received data will be copied.
xTicksToWait	the maximum amount of time the task should remain in the Blocked state.

Return values of the function:

- pdPASS: data was successfully read from the queue.
- errQUEUE_EMPTY: data cannot be read from the queue because the queue is already empty.

Check Queues Status

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

Description

`uxQueueMessagesWaiting` queries the number of items that are currently in a queue.

Check Queues Status

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

Description

`uxQueueMessagesWaiting` queries the number of items that are currently in a queue.

Argument	Description
xQueue	the handle of the queue.

Example: Create Queues

```c
QueueHandle_t xQueue;
int main( void ){
 /* The queue is created to hold a maximum of 5 values, each
      of which is large enough to hold a variable of type
     int32_t. */
 xQueue = xQueueCreate( 5, sizeof( int32_t ) );
 if( xQueue != NULL ){
        // rest of the code
 }
 else{
   //The queue could not be created.
 }
 for( ;; );
}
```

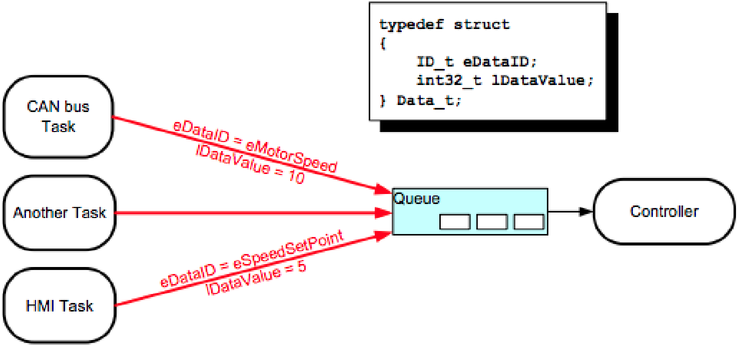
Example: Write to a Queues

```
 void vSenderTask( void *pvParameters ){
 int32_t lValueToSend;
 BaseType_t xStatus;
 lValueToSend = ( int32_t ) pvParameters;
 for( ;; ){
  xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
  if( xStatus != pdPASS ){
   vPrintString( "Could not send to the queue.\r\n" );
  }
 }
}
```

Example: Read from a Queues

```
static void vReceiverTask( void *pvParameters ){
 int32_t lReceivedValue;
 BaseType_t xStatus;
 const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );
 for( ;; )
 {
  xStatus = xQueueReceive( xQueue, &lReceivedValue,
      xTicksToWait );
  if( xStatus == pdPASS ){
   vPrintStringAndNumber( "Received = ", lReceivedValue );
  }
  else{
  vPrintString( "Could not receive from the queue.\r\n" );
  }
 }
}
```

Receiving Data From Multiple Sources



```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;
```

Receiving Data From Multiple Sources

```
/* Define an enumerated type used to identify the source of
    the data. */
typedef enum{
 eSender1,
 eSender2
} ID_t;

//Define the structure type that will be passed on the queue
    .
typedef struct{
 uint8_t edataValue;
 ID_t eDataSource;
} Data_t;

// create a queue
xQueue = xQueueCreate( 3, sizeof( Data_t ) );
```

Receiving Data From Multiple Sources

```c
xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
if( xStatus == pdPASS ){
 if( xReceivedStructure.eDataSource == eSender1 {
   vPrintStringAndNumber( "From Sender 1 = ",
       xReceivedStructure.edataValue );
 }
 else{
   vPrintStringAndNumber( "From Sender 2 = ",
       xReceivedStructure.edataValue );
 }
}
```

Create a Mailbox

Mailbox

A mailbox is used to hold data that can be read by any task, or any interrupt service routine. The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten.

Create a Mailbox

Mailbox

A mailbox is used to hold data that can be read by any task, or any interrupt service routine. The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten.

Overwrite Data in queue

`xQueueOverwrite()` API can be used to overwrite data in a queue.

```
xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue )
```

Read Data without Remove

`xQueuePeek()` API can be used to read data in a queue without removing it.

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

Question?