Queues Introduction

A **message queue** is a kind of FIFO buffer that holds fixed-size data items. Also, the number of items a queue can hold is also fixed, after its initialization. Usually, **tasks write** data to the end of the buffer and read from the front end of the buffer. But it is also possible to write at the front end. Multiple writers and readers can write and read from the buffer.

But only one writer/reader can access buffer at a time and Other, tasks remain block. Therefore, blocking is possible both on reads and writes to buffer.

Blocking on Queue Reads

Blocking on reads possible in following scenarios:

1. If **multiple tasks** are ready to receive data from the message queue, then the highest priority task gets to read data first and lowest priority one read data at the end. Meanwhile, other tasks remain block. We can also specify the maximum blocking time of a task while sending a read request. But different tasks can also have different blocking time.
2. The other possible case is when a queue is empty. In such a case, all read requests go to a blocking state. Once data become available from the **message queue** (when another task places data into the ready queue), all readers moved to the ready state according to their priority.

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 in the queue should the queue already be full.

Queues can have multiple writers so it is possible that a full queue will 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 was waiting for space. If the blocked tasks have equal priority, then it will be the task that has been waiting for space the longest that is unblocked.

Use FreeRTOS Queue API

Till now we have covered the basics of Queues write and read process. Now let's begin with the main part of this tutorial. This section covers the following contents of **FreeRTOS queue API**:

- How to create queue
- How to write and read data/message

Creating Queue with FreeRTOS

xQueueCreate() API Function is use to create a queue with required number of buffer elements and size of each element.

```
QueueHandle_t xQueueCreate( UBaseType_t  Queue_Length, UBaseType_t Item_Size );
```

xQueueCreate() allocates memory according to queue length and data size. Assigned memory also holds the required data structure for a queue. Therefore, be careful while using it with Arduino Because Arduino comes with limited memory.

This function use reference by handles with a return type variable QueueHandle_t. This handler decides if enough memory is available on Arduino to create queue not.

Argument	Description
Return Value	Handler returns NULL value, if enough memory is not available on heap to create queue otherwise it returns Non-Null value
Queue_Length	The length of queue that is the total number of elements it can hold
Item_Size	Size of each item in bytes

For example, to **create a Queue**, create a variable of type xQueueHandle. This is used to store the handle to the queue return variable.

```
xQueueHandle  long_queue;
```

The queue is created to hold a maximum of 5 values, each of which is large enough to hold a variable of type long.

```
Long_queue = xQueueCreate( 5, sizeof( long ) );
```

Before creating tasks that read and write data to queue, you should check long_queue handler return value this:

```
if( long_queue != NULL ) { //create read and write message routines }
```

This is how we create queue with FreeRTOS in Arduino IDE. On top of that, xQueueReset() API function can also be used to reset it to its original state

FreeRTOS Sending data to Queue

FreeRTOS API provides these two functions to read and write message to/from queue.

1. xQueueSendToBack() or xQueueSendToFront() (both have same functionality)
2. xQueueSendToFront()

As their name suggests, these functions write data to the front or back end of a buffer. These function takes three arguments as an input and returns one value that is success or failure of message write.

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

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

Arguments	Details
xQueue	This provides the reference of the queue to which we want to write data. It is the same handle variable that we used while creating queue with xQueueCreate()
pvItemToQueue	A pointer variable to the message that we want to send to queue
TickType_t xTicksToWait	In case, if queue is full, it specifies the maximum blocking time of a task until space to become available. Passing value zero will force the task to return without writing, If queue is full. Similarly, passing portMAX_DELAY to function will cause the task to remain in blocking state indefinitely
Return value	It returns two values such as pdPASS (if data is successfully written) and errQUEUE_FUL(if data could not be written)

FreeRTOS Receiving data from Queue

Similar to write API functions, xQueueReceive() is used to read message/data from a queue.

Note: Once you read the data using xQueueReceive(), it will be removed from the queue.

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

All input arguments and return value of xQueueReceive() has same working as write API except second argument “const pvBuffer”. It is a pointer to a variable to which we want to store received data.

Arduino FreeRTOS queue Read/Write Data Example one

This Arduino example demonstrates a **queue being created**, data being sent to the queue from multiple tasks, and data being received from the buffer.

- The queue is created to hold data items of type long.
- The tasks that send data do not specify a block time, while the task that receives from the queue does.

- The priority of the tasks that send to the queue is lower than the priority of the task that receives from the queue.
- This means the queue should never contain more than one item because as soon as data is sent to the queue the receiving task will unblock, pre-empt the sending task, and remove the data leaving the queue empty once again.

Writing to Queue Task

This code shows the implementation of the task that writes to the queue. Two instances of this task are created, one that continuously writes the value 100 to the queue, and another that continuously writes the value 200 to the same queue. The task parameter is used to pass these values into each task instance.

```
void vSenderTask( void *pvParameters ){

long lValueToSend;



lValueToSend = ( long ) pvParameters;

for( ;; )

{

        xQueueSend( xQueue, &lValueToSend, portMAX_DELAY );

Serial.print( "Sent = ");

Serial.println(lValueToSend);

taskYIELD();}}
```

Reading from Queue Task

This Arduino code shows the implementation of the task that receives data from the queue. The receiving task specifies a block time of 100 milliseconds so will enter the Blocked state to wait for data to become available. It will leave the Blocked state when either data is available on the queue, or 100 milliseconds passes without data becoming available. In this example, the 100 milliseconds timeout should never expire as there are two tasks that are continuously writing to the queue.

```
void vReceiverTask( void *pvParameters )

{

long lReceivedValue;
```

```
for( ;; ){

if (xQueueReceive( xQueue, &lReceivedValue, portMAX_DELAY ) == pdPASS){

Serial.print( "Received = ");

Serial.println(lReceivedValue);

}taskYIELD();

}}
```

Creating Queue List

This code contains the definition of the Arduino setup() function. This simply creates the queue and the three tasks before starting the scheduler. The queue is created to hold a maximum of 5 long values even though the priorities of the tasks are set so the queue will never actually contain more than one item at a time.

```
#include <Arduino_FreeRTOS.h>

#include <queue.h>

QueueHandle_t xQueue;void setup() {

 Serial.begin(9600);

 xQueue = xQueueCreate( 5, sizeof( long ) );if( xQueue != NULL ){xTaskCreate( vSenderTask,
"Sender1", 240, ( void * ) 100, 1, NULL );xTaskCreate( vSenderTask, "Sender2", 240, ( void * )
200, 1, NULL );xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 2,
NULL );vTaskStartScheduler();}else{/* The queue could not be created. */}

}

void loop(){

 // put your main code here, to run repeatedly:

}
```

Arduino complete Sketch

This is complete Arduino sketch to read/write message to/from queue.

```
#include <Arduino_FreeRTOS.h>

#include <queue.h>
```

```
QueueHandle_t xQueue;

void setup() {

 Serial.begin(9600);

 xQueue = xQueueCreate( 5, sizeof( long ) );

if( xQueue != NULL )

{

        xTaskCreate( vSenderTask, "Sender1", 240, ( void * ) 100, 1, NULL );

xTaskCreate( vSenderTask, "Sender2", 240, ( void * ) 200, 1, NULL );

xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 2, NULL );

vTaskStartScheduler();

}

else

{

/* The queue could not be created. */}

}

void loop()

{

 // put your main code here, to run repeatedly:

}

void vSenderTask( void *pvParameters )

{

long lValueToSend;
```

```
lValueToSend = ( long ) pvParameters;

for( ;; )

{

xQueueSend( xQueue, &lValueToSend, portMAX_DELAY );

Serial.print( "Sent = ");

Serial.println(lValueToSend);

taskYIELD();

}

}




void vReceiverTask( void *pvParameters )

{

long lReceivedValue;

for( ;; )

{

if (xQueueReceive( xQueue, &lReceivedValue, portMAX_DELAY ) == pdPASS)

{

Serial.print( "Received = ");

Serial.println(lReceivedValue);}taskYIELD();

}}
```

The tasks that send to the queue call taskYIELD() on each iteration of their infinite loop. taskYIELD() informs the scheduler that a switch to another task should occur now rather than keeping the executing task in the Running state until the end of the current time slice. A task that calls taskYIELD() is in effect volunteering to be removed from the Running state.

As both tasks that send to the queue have an identical priority each time one calls taskYIELD() the other starts executing – the task that calls taskYIELD() is moved to the Ready state as the other sending task is moved to the Running state. This causes the two sending tasks to send data to the queue in turn. The output produced by Example 10 is shown in Figure below:

```
Received = 100
Received = 200
Sent = 100
Sent = 200
Received = 100
Received = 200
Sent =S100
Received = 200
Received = 100
Sent = 2S0
Received = 100
Received = 200
Sent = 100S
Received = 200
Received = 100
Sent = 200
Rent = 100
```

How to use FreeRTOS structure Queue to Receive Data from Multiple Tasks

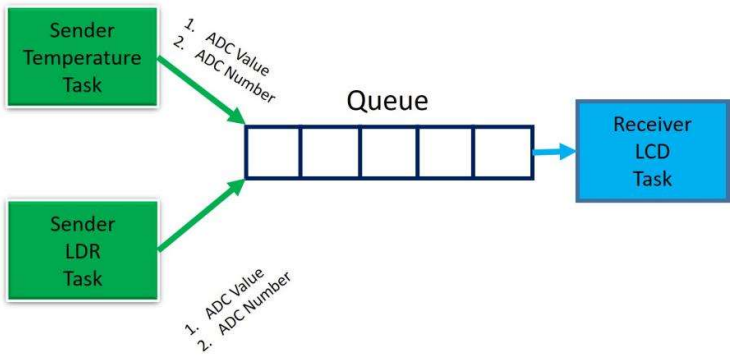
Why do we used queues Structure?

Why is it necessary for the receiver task to the know the source of data?

Because otherwise there is no other way, the receiver can differentiate data resources and can perform a meaningful operation on that data. The simplest solution is to use structures data type to write and read data from a single queue. The structure should contain a source of data and the actual value that we want to communicate with the receiver task.

Structure Queue Example to Receive Data from Multiple Resources

This block diagram demonstrates an example to transfer structure to a single queue.



We create an example with Arduino that measure temperature and light sensor value from A0 and A1 channels of Arduino, respectively. After that, prints sensors value on 16×2 LCD.

- First, we create a queue that can hold the structure. This structure elements consist of data value and tag of resource (such as analog channel number)
- In this example, we create two sender tasks such as Temperature and LDR.
- The temperature sender reads the LM35 sensor value from analog channel A0 of Arduino IDE. After that, it writes structure to the queue that contains ADC value and channel number (A0).
- Similarly, a light-dependent resistor task measures light intensity with the ADC channel (A1) and writes data in a structured format.
- LCD task (Receiver) also reads this structure data and processes data to display sensor value after identifying the source.

Implementation of FreeRTOS structure Queue in Arduino

FreeRTOS Structure Queue Code

First include FreeRTOS, Queue and LCD header files.

```
// Include Arduino FreeRTOS library
#include <Arduino_FreeRTOS.h>
// include LCD library function
#include<LiquidCrystal.h>
#include <queue.h>
```

After that define the pin name that connect with LCD.

```
//Define LCD pins

LiquidCrystal lcd (7,6,5,4,3,2); //RS, E, D4, D5, D6, D7
```

We define a structure that consists of two elements such as sensor value and pin number. This structure willbe passed to the queue to write and read sensor output values.

```
// Define a struct

struct pinRead

{

   int pin;  // analog channel indentifier

 int value; // sensor output value

};
```

This lines defines the hanlder for structQueue to access it for tasks reference.

```
QueueHandle_t structQueue;
```

These two functions initialize LCD and Uart communication library.

```
Serial.begin(9600); // Enable serial communication library.

lcd.begin(16, 2);  // Enable LCD library
```

Create a structure queue with xQueueCreate() API and pass above defined structure in sizeof() argument.

```
 // create Structure Queue

structQueue = xQueueCreate(10, // Queue length

          sizeof(struct pinRead) // Queue item size

           );
```

In the setup function, we create three tasks that use structure_queue to write and read data. Firstly, “TaskLCD” is created that reads structure elements from the queue and it has the highest priority. Conversely, TaskTempReadPin0() and TaskLightReadPin0() pins are created with equal priority and lower than the receiver task. I will explain the reason later on.

```
if (structQueue != NULL) {

// Create tasks that consume the queue if it was created.

xTaskCreate(TaskLcd, // Task function

          "Displaydata", // A name just for humans
```

```
        128,  // This stack size can be checked & adjusted by reading the Stack Highwater

        NULL,

        2, // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the
lowest.

        NULL);



// Create task that publish data in the queue if it was
created.xTaskCreate(TaskTempReadPin0, // Task function

        "AnalogReadPin0", // Task name

        128,  // Stack size

        NULL,

        1, // Priority

        NULL);

// Create other task that publish data in the queue if it was
created.xTaskCreate(TaskLightReadPin1, // Task function

        "AnalogReadPin1", // Task name

        128,  // Stack size

        NULL,

        1, // Priority

        NULL);}
```

Receiver Task

"TaskLcd" is a receiver task that reads structure data from struct_queue. The receiving task has the highest priority. Therefore, at the start of program execution, it will run first, but it will enter the blocking state. Because the queue will be empty. Hence, it will execute as soon as when one of the senders will write data to the queue. This task always expects to receive data, therefore, we specify blocking time to the maximum value that is "portMAX_DELAY".

```
void TaskLcd(void * pvParameters){

 (void) pvParameters;



 for (;;)

 {

  int TEMP=0; // temporary variable to hold Temperature adc0 value

  int LDR=0; // temporary variable to hold LDR adc1 value

   struct pinRead currentPinRead; // structure to hold receiv data

   // Read structure elements from queue and check if data received successfully

   if (xQueueReceive(structQueue, &currentPinRead, portMAX_DELAY) == pdPASS)

   {

     // print received data elements on serial montor

     Serial.print("Pin: ");

     Serial.print(currentPinRead.pin);

     Serial.print(" Value: ");

     Serial.println(currentPinRead.value);



     // If condition checks,  if data receive from channel zero

     // If yes, store sensor value member of structure in temporary temp variable

     if(currentPinRead.pin==0)

     {

       TEMP = (currentPinRead.value * 500)/1024; // convert adc value into temperature
```

```
// dispay temperature sensor output in first line of 16x2 LCD
lcd.setCursor(0, 0);
lcd.print("Temp = ");
lcd.setCursor(7, 0);
lcd.print(TEMP);
lcd.print("'C");
}

// If condition checks,  if data receive from channel one (A1)
// If yes, store sensor value member of structure in temporary LDR variable
if(currentPinRead.pin==1)
{

LDR = map(currentPinRead.value, 0, 1023, 0, 255); //map ADC1 value to light value
// dispay light dependent resistor output in first line of 16x2 LCD
lcd.setCursor(0, 1);
lcd.print("LIGHT = ");
lcd.setCursor(7, 1);
lcd.print(LDR);
lcd.print("LUX");

}
}
```

```
    taskYIELD(); // terminate the task and inform schulder about it

  }

 }
```

Sender Tasks

These are the definitions of two sender tasks. As we mentioned earlier, writer tasks have equal priority but lower than receiver tasks. Therefore, when a high priority task will be in the blocking state, then one of the low priority tasks will execute by following time sharing or time-slicing scheduling algorithm. Because FreeRTOS is based on Prioritized Pre-emptive Scheduling with Time Slicing.

```
// Temperature value and adc number writing

void TaskTempReadPin0(void *pvParameters){

 (void) pvParameters;



 for (;;)

 {



   struct pinRead currentPinRead;  // define a structure of type pinRead

   currentPinRead.pin = 0; // assign value '0' to pin element of struct

   currentPinRead.value = analogRead(A0); // Read adc value from A0 channel and store it in value element of struct

   xQueueSend(structQueue, &currentPinRead, portMAX_DELAY); //write struct message to queue

   Serial.println("Channel_0"); //print Channel_0 on serial monitor

   taskYIELD(); //terminate the task and inform schulder about it

 }}

// Light Sensor value and adc number writing
```

```
void TaskLightReadPin1(void *pvParameters){

 (void) pvParameters;



 for (;;)

 {



   struct pinRead currentPinRead; // define a structure of type pinRead

   currentPinRead.pin = 1; // assign value '1' to pin element of struct

   currentPinRead.value = analogRead(A1);// Read adc value from A1 channel and store it in value element of struct

   xQueueSend(structQueue, &currentPinRead, portMAX_DELAY); //write struct message to queue

   Serial.println("Channel_1");//print Channel_1 on serial monitor

   taskYIELD(); //terminate the task and inform schulder about it

 }}
```

FreeRTOS Structure Queue Arduino Code

This is a complete Code. You can also download complete code from a download link give below.

```
// Include Arduino FreeRTOS library

#include <Arduino_FreeRTOS.h>

// include LCD library function

#include<LiquidCrystal.h>

//Define LCD pins

LiquidCrystal lcd (7,6,5,4,3,2); //RS, E, D4, D5, D6, D7

// Include queue support

#include <queue.h>
```

```
// Define a struct
struct pinRead
{
 int pin;  // analog channel indentifier
 int value; // sensor output value
};

QueueHandle_t structQueue;

void setup()
{

 Serial.begin(9600); // Enable serial communication library.
 lcd.begin(16, 2);  // Enable LCD library
 // create Structure Queue
 structQueue = xQueueCreate(10, // Queue length
                    sizeof(struct pinRead) // Queue item size
                    );

 if (structQueue != NULL) {

  // Create task that consumes the queue if it was created.
  xTaskCreate(TaskLcd, // Task function
          "Displaydata", // A name just for humans
          128,  // This stack size can be checked & adjusted by reading the Stack Highwater
          NULL,
          2, // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the
lowest.
          NULL);
```

```
  // Create task that publish data in the queue if it was created.
  xTaskCreate(TaskTempReadPin0, // Task function
          "AnalogReadPin0", // Task name
          128,  // Stack size
          NULL, yvv
          1, // Priority
          NULL);

  // Create other task that publish data in the queue if it was created.
  xTaskCreate(TaskLightReadPin1, // Task function
          "AnalogReadPin1", // Task name
          128,  // Stack size
          NULL,
          1, // Priority
          NULL);

 }

 vTaskStartScheduler();

}

void loop()
{

}

// Temperature value and adc number writing
void TaskTempReadPin0(void *pvParameters)
{
```

```
 (void) pvParameters;

 for (;;)
 {

  struct pinRead currentPinRead;  // define a structure of type pinRead
  currentPinRead.pin = 0; // assign value '0' to pin element of struct
  currentPinRead.value = analogRead(A0); // Read adc value from A0 channel and store it in value element of struct
  xQueueSend(structQueue, &currentPinRead, portMAX_DELAY); //write struct message to queue
  Serial.println("Channel_0"); //print Channel_0 on serial monitor
  taskYIELD(); //terminate the task and inform schulder about it
 }
}

// Light Sensor value and adc number writing
void TaskLightReadPin1(void *pvParameters)
{
 (void) pvParameters;

 for (;;)
 {

  struct pinRead currentPinRead; // define a structure of type pinRead
  currentPinRead.pin = 1; // assign value '1' to pin element of struct
  currentPinRead.value = analogRead(A1);// Read adc value from A1 channel and store it in value element of struct
  xQueueSend(structQueue, &currentPinRead, portMAX_DELAY); //write struct message to queue
  Serial.println("Channel_1");//print Channel_1 on serial monitor
  taskYIELD(); //terminate the task and inform schulder about it
```

```
  }
}

void TaskLcd(void * pvParameters)
{
  (void) pvParameters;

  for (;;)
  {
   int TEMP=0; // temporary variable to hold Temperature adc0 value
   int LDR=0; // temporary variable to hold LDR adc1 value
    struct pinRead currentPinRead; // structure to hold receiv data
    // Read structure elements from queue and check if data received successfully
    if (xQueueReceive(structQueue, &currentPinRead, portMAX_DELAY) == pdPASS)
    {
      // print received data elements on serial montor
      Serial.print("Pin: ");
      Serial.print(currentPinRead.pin);
      Serial.print(" Value: ");
      Serial.println(currentPinRead.value);

      // If condition checks,  if data receive from channel zero
      // If yes, store sensor value member of structure in temporary temp variable
      if(currentPinRead.pin==0)
      {
        TEMP = (currentPinRead.value * 500)/1024; // convert adc value into temperature

      // dispay temperature sensor output in first line of 16x2 LCD
      lcd.setCursor(0, 0);
      lcd.print("Temp = ");
```

```
    lcd.setCursor(7, 0);
    lcd.print(TEMP);
    lcd.print("'C");
    }

    // If condition checks,  if data receive from channel one (A1)
    // If yes, store sensor value member of structure in temporary LDR variable
    if(currentPinRead.pin==1)
    {

    LDR = map(currentPinRead.value, 0, 1023, 0, 255); //map ADC1 value to light value
    // dispay light dependent resistor output in first line of 16x2 LCD
    lcd.setCursor(0, 1);
    lcd.print("LIGHT = ");
    lcd.setCursor(7, 1);
    lcd.print(LDR);
    lcd.print("LUX");

    }
    }
    taskYIELD(); // terminate the task and inform schulder about it
  }
}
```