Real-time Embedded systems

Lab4 Queue Management

1. **Characteristics of a Queue in FreeRTOS**

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.

Writing data to a queue causes a byte-for-byte copy of the data to be stored in the queue itself. Reading data from a queue causes the copy of the data to be removed from the queue.

**Access by multiple tasks**

Queues are objects in their own right that are not owned by or assigned to any particular task. Any number of tasks can write to the same queue and any number of tasks can read from the same queue. A queue having multiple writers is very common, whereas a queue having multiple readers is quite rare.

**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 should 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 become available.

Queue 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, the task that has been waiting for data the longest will be unblocked.

**Blocking on Queue Writes**

As 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.

Queue 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 free 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, the task that has been waiting for space the longest will be unblocked.

1. **Using a Queue**

**xQueueCreate() API function**

A queue must be explicitly created before it can be used. xQueueCreate() is used to create a queue and returns an xQueueHandle to reference the queue it creates.

FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. xQueueCreate() will return NULL if there is insufficient heap RAM available for the queue to be created. The xQueueCreate(0 API function prototype is as the following.

xQueueHandle xQueueCreate(unsigned portBASE\_TYPE uxQueueLength,

unsigned portBASE\_TYPE uxItemSize);

/\* Return value, if NULL, the queue cannot be created due to insufficient RAM space; if non-NULL, the queue has been created successfully. \*/

**xQueueSendToBack () and xQueueSendToFront() API functions**

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. Their prototypes are as the following.

portBASE\_TYPE xQueueSendToFront( xQueueHandle xQueue,

const void \* pvItemToQueue, // a pointer to the data to be copied into the queue

portTickType xTicksToWait);

portBASE\_TYPE xQueueSendToBack( xQueueHandle xQueue,

const void \* pvItemToQueue, // a pointer to the data to be copied into the queue

portTickType xTicksToWait);

/\* xTicksToWait is the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

This block time is specified in tick seconds, so the absolute time it represents is dependent on the tick frequency. The constant portTICK\_RATE\_MS can be used to convert a time specified in milliseonds to a time specified in ticks. Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely, provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h. \*/

/\* Two possible return values: if data was successfully sent to the queue, pdPASS will be returned; if data could not be written to the queue as the queue was already full, errQUEUE\_FULL will be returned. \*/

**xQueueReceive () and xQueuePeek() API functions**

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

xQueuePeek() is used to receive an item from a queue without the item being removed from the queue. It receives the item from the head of the queue, without modifying the data that is stored in the queue, or the order in which data is stored in the queue.

portBASE\_TYPE xQueueReceive( xQueueHandle xQueue,

const void \* pvBuffer, // a pointer to the memory into which the received data to be copied

portTickType xTicksToWait);

portBASE\_TYPE xQueuePeek( xQueueHandle xQueue,

const void \* pvBuffer, // a pointer to the memory into which the received data to be copied

portTickType xTicksToWait);

/\* Two possible return values: if data was successfully read from the queue, pdPASS will be returned; if data could not be read from the queue as the queue was already empty, errQUEUE\_EMPTY will be returned. \*/

**uxQueueMessagesWaiting() API functions**

This is used to query the number of items that are currently in the queue. It prototype is as the following.

unsigned portBASE\_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue);

**/\*** Return value is the number of items that the queue being queried is currently holding. If zero is returned, the queue is empty.**\*/**

**Example 10 Blocking when receiving from a queue**

This part demonstrates a queue being created, data being sent to the queue from multiple tasks, and data being received from the queue. The queue is created to hold data items of type long. The tasks that send to the queue do not specify a block time, whereas 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 that the queue should never contain more than one item because, as soon as data is sent to the queue the receiving task will unlock, pre-empt the sending task, and remove the data- leaving the queue empty once again.

Please perform the following steps to edit the main.c file.

1. Include the *queue.h* header file to enable calling API functions related to Queue.

*#include “queue.h”*

1. Declare two functions prototypes *static void* vSenderTask(void \*pvParameters); and static void vReceiveTask(void \*pvParameters);*.*

*static void vSenderTask( void \*pvParameters );*

*static void vReceiverTask( void \*pvParameters );*

1. Declare a global variable *xQueue* with the type *xQueueHandle.*

*xQueueHandle xQueue;*

1. Declare two global constant integers data1 and data2 with the *long* type, and initialize them by 100 and 200 respectively. They are the data sent by two sender tasks to the queue.

*const long data1 = 100;*

*const long data2 = 200;*

1. Define the function *static void vSenderTask(void \*pvParameters).*

- First declare two variables: 1) *lValueToSend* with *long* type pointer to hold the address of the variable to be written to the queue; 2) *xStatus* with *portBASE\_TYPE* type to hold the return value of *xQueueSendToBack()* function;

- Then cast input parameter *pvParameters* from (*void \**) to (*long* \*) type and assign it to *lValueToSend*;

- Inside the infinite *for(;;)* loop, Firstly, call *xQueueSendToBack*() function to write the data *lValueToSend* to the *xQueue* with *xTicksToWait* set to 0, and assign the return value to *xStatus*. Then, check if the write operation is successful, if not, print a string “Could not send to the queue.\n”. The last operation is to call *taskYIELD*() to allow the other sender task to execute.

1. Define another function *static void vReceiveTask(void \*pvParameters);.*

- First declare three variables *lReceivedValue* with *long* type to hold the value to be received from the queue, *xStatus* with *portBASE\_TYPE* type to hold the return value of *xQueueReceive()* function, and a constant *xTicksToWait* of the type *portTickType* and initialized by *100/portTICK\_RATE\_MS*.

*long lReceivedValue;*

*portBASE\_TYPE xStatus;*

*const portTickType xTicksToWait = 100 / portTICK\_RATE\_MS;*

- Inside the *for(;;){}* loop body, firstly, check whether the queue is empty by calling *uxQueueMessagesWaiting*() function, if not empty (i.e. 0), print a string “Queue should have been empty!”.

*if( uxQueueMessagesWaiting( xQueue ) != 0 ) {*

*puts ( "Queue should have been empty!\r\n" );*

*}*

- Then, call *xQueueReceive*() function to receive *lValueReceive* from the *xQueue* with *xTicksToWait* waiting time, and assign the return value to *xStatus*;

*xStatus = xQueueReceive ( xQueue, &lReceivedValue, xTicksToWait );*

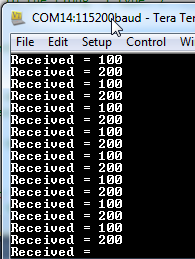
* Then, check if the receive operation is successful, if yes, print a string and a number with *“Received = ”* and *lReceivedValue* respectively; if not, print a string “Could not receive from the queue.”.

1. Define the *main()* function.

- First, create a queue by calling *xQueueCreate*() with length 5 and itemSize as long type and assign the function return to the variable *xQueue*.

- Then, check if xQueue is empty, if not, create two vSenderTask() tasks named “Sender1” and “Sender2” whose priorities are set to 1, and one vReceiveTask() task named as “Receiver” before starting *vTaskStartScheduler()* whose priority is set to 2. The fourth argument of “Sender 1” task and “Sender 2” is data1 and data2 declared in step 5, respectively.

Expected outputs are shown here.



Using Queues to transfer compound types

It is common for a task to receive data from multiple sources on a single queue. Often, the receiver of the data needs to know where the data came from, to allow it to determine how the data should be processed. A simple way to achieve this is to use the queue to transfer structures where both the value of the data and the sources of the data are contained in the structure fields.

**Example 11 Blocking when sending to a queue or sending structures on a queue**

This part is similar to Example 10, but the task priorities are reversed so the receiving task has a lower priority than the sending tasks. Also the queue is used to pass structures, rather than simple long integers, between the tasks.

The structure used in this part is as the following.

*typedef struct // define the structure type that will be passed on the queue*

*{*

*unsigned char ucValue;*

*unsigned char ucSource;*

*}xData;*

*const xData xStructsToSend[2] =*

*{*

*{100, mainSENDER\_1},*

*{200, mainSENDER\_2}*

*};*

Please note: The sending task specifies a block time of 100 milliseconds instead of 0 in part1. It will only enter the Blocked state when the queue is full so this task always expects the number of items in the queue to be equal to the queue length = 3, so the receiving task can start to execute.

