|  |
| --- |
|  |
| **EE489 / EE589 Real-Time Embedded Systems Design** |
| Lab #4  *Dianzhi Yu* |
|  |
| *2/13/2018* |

|  |
| --- |
|  |

# Introduction

FreeRTOS applications are structured as a set of independent tasks and each task is effectively a mini program on its own right. Queue is the underlying primitive which can be used for communication and synchronization mechanisms in FreeRTOS. The first example in this lab shows a queue being created, data being sent to the queue from multiple tasks, and data being received from the queue. And the second example in this lab sets the receiving task lower priority than the sending tasks. And the queue is used to pass structures, rather than simple long integers, between the tasks.

1. **A list of all FreeRTOS API functions being used**

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

Create a queue which holds a maximum of 5 long type values and assign it to xQueue. The xQueue is declared as the type of xQueueHandle to make sure a value is returned, so a queue has been created successfully.

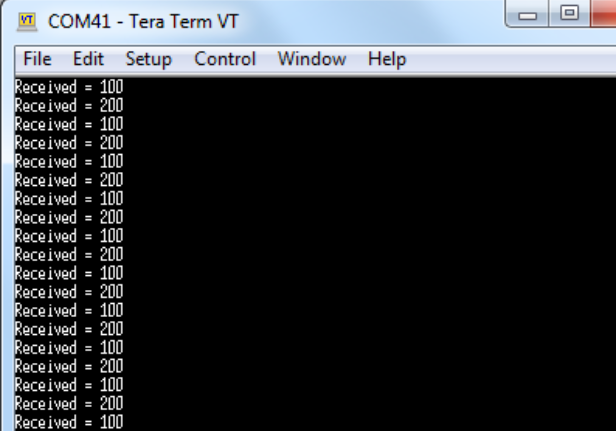
xStatus = xQueueSendToBack(xQueue, &1ValueToSend, 0);

The first parameter is the queue to which data is being sent. The queue "xQueue" was created before the scheduler was started. The second parameter is the address of the data to be sent which is equal to lValueToSend. The third parameter is the Block time, which is set to 0 means that there always is space in the queue. And the return value of this function is assigned to xStatus. If xStatus is not equal to pdPASS, the queue is empty so the data can not be read from the queue.

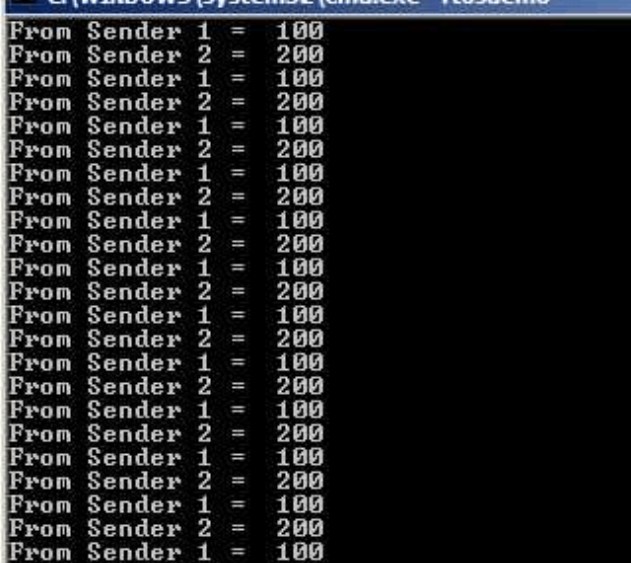
xStatus=xQueueReceive(xQueue,&lReceivedValue,xTicksToWait);

This function is used to receive (consume) an item from a queue. The first parameter is the queue from which data is being received. The second parameter is a pointer to the memory into which the received data will be copied. And it must be at least large enough to hold the data item held by the queue. The last parameter is the block time "xTicksToWait" – the maximum amount of time that the task should remain in the Blocked state to wait for data to be available should the queue already be empty.

1. **Screen shots of the program execution results or debug windows of Keil µVision**



The receiver task has higher priority so it runs first. It read data from the queue until the queue is empty. And then the receiver enters the blocked state to wait. Send tasks will write the data to the queue.



The example 11 demonstrates the sequence of execution that results from having the priority of the sending tasks above that of the receiving task.

1. **Conclusion**

A queue can hold a finite number of fixed size data items and can be read and written by functions. And according to the different priorities, the tasks can run to receive data from the queue or send the data to the queue.

1. **Appendix: The source code (main.c) with sufficient comments.**

Ex 10

/\*\*

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "queue.h"

void vApplicationIdleHook()

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

/\* Two instances are created from the sender task in main(). \*/

/\* To-do:

Declare the prototype of the task function "vSenderTask".\*/

static void vSenderTask( void \*pvParameters );

/\* A single instance is created from the receiver task in main(). \*/

/\* To-do:

Declare the prototype of the task function "vReceiverTask".\*/

static void vReceiverTask( void \*pvParameters );

/\*-----------------------------------------------------------\*/

/\* This variable xQueue is used to store the queue data structure that is

accessed by all three tasks. \*/

/\* To-do:

Declare a Queue named as "xQueue" with the type of xQueueHandle. \*/

xQueueHandle xQueue;

/\* To-do:

Declare two global constant variables data1 and data2 with long type;

and initialize data1 with the constant 100, and data2 with the constant 200 \*/

const long data1=100;

const long data2=200;

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 10 starts\r\n");

/\* To-do:

Call the xQueueCreate() API to create a queue that holds a maximum of 5 long

type values. And assign the created queue to to xQueue. \*/

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

if( xQueue != NULL )

{

/\* To-do:

Create one task instance of vSenderTask at priority 1. The 4th parameter is used

to pass the address of variable "data1". This task should continuously write

this variable to the queue. \*/

xTaskCreate(vSenderTask,"Sender1", 200, (void\*)data1,1,NULL );

/\* To-do:

Create another task instance of vSenderTask at priority 1. The 4th parameter is used

to pass the address of variable "data2". This task should continuously write this

variable to the queue. \*/

xTaskCreate(vSenderTask,"Sender2", 200, (void\*)data2,1,NULL );

/\* To-do:

Create one task instance of vReceiverTask at priority 2. The 4th parameter is NULL.

This task will read data from the queue. \*/

xTaskCreate( vReceiverTask, "Receiver",200,NULL,2,NULL);

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

else

{

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

}

for( ;; );

}

/\*-----------------------------------------------------------\*/

static void vSenderTask( void \*pvParameters )

{

/\*To-do:

Declare a long type pointer variable named as lValueToSend. \*/

long lValueToSend;

/\*To-do:

Declare a variable named as xStatus with the type of portBASE\_TYPE. It holds

the return value of the function xQueueSendToBack() \*/

portBASE\_TYPE xStatus;

/\* Two instances are created of this task so the value that is sent to the

queue is passed in via the task parameter rather than be hard coded. This way

each instance can use a different value. Cast the parameter to the required

type. \*/

/\*To-do:

Assign the value of input parameter "pvParamters" to the local variable

"lValueToSend", and cast the parameter to the (long \*) type.\*/

lValueToSend=(long \*) pvParameters;

/\* As per most tasks, this task is implemented within an infinite loop. \*/

for( ;; )

{

/\*To-do:

Call the xQueueSendToBack() API function to send data to the queue by passing

three parameters as described above; and assign the returned value to xStatus.

The first parameter is the queue to which data is being sent. The

queue "xQueue" was created before the scheduler was started.

The second parameter is the address of the data to be sent which is equal

to lValueToSend.

The third parameter is the Block time – the time the task should be kept

in the Blocked state to wait for space to become available on the queue

should the queue already be full. In this case we specify a block

time as 0 because there should always be space in the queue. \*/

xStatus = xQueueSendToBack(xQueue,&lValueToSend,0);

if( uxQueueMessagesWaiting( xQueue ) != 0 ) {

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

}

/\* To-do:

Call the API function taskYIELD() to allow the other sender task to execute. \*/

taskYIELD();

}

}

/\*-----------------------------------------------------------\*/

static void vReceiverTask( void \*pvParameters )

{

/\* Declare the variable with the long type that will hold the values received

from the queue. \*/

long lReceivedValue;

/\* Declare a variable named as xStatus with the type of portBASE\_TYPE. It holds

the return value of the function xQueueReceive() \*/

portBASE\_TYPE xStatus;

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

/\* This task is also defined within an infinite loop. \*/

for( ;; )

{

/\* As this task unblocks immediately that data is written to the queue this

call should always find the queue empty. \*/

if( uxQueueMessagesWaiting( xQueue ) != 0 )

{

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

}

/\*To-do:

Call the xQueueReceive() API function to send data to the queue by passing

three parameters as described above; and assign the returned value to xStatus.

The first parameter is the queue from which data is to be received. The

queue "xQueue" is created before the scheduler is started.

The second parameter is the buffer into which the received data will be

placed. In this case the buffer is simply the address of the variable

"&lReceivedValue" that has the required size to hold the received data.

the last parameter is the block time "xTicksToWait" – the maximum amount of time that the

task should remain in the Blocked state to wait for data to be available should

the queue already be empty. \*/

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

/\*To-do: check whether the value of xStatus is equal to pdPASS. If yes, print out the following string. \*/

if(xStatus == pdPASS)

{

/\* Data was successfully received from the queue, print out the current value of lReceivedValue. \*/

printf( "Received = %ld\r\n", lReceivedValue );

}

else

{

/\* We did not receive anything from the queue even after waiting for 100ms.

This must be an error as the sending tasks are free running and will be

continuously writing to the queue. \*/

printf( "Could not receive from the queue.\r\n" );

}

}

}

EX11

/\*\*

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

void vApplicationIdleHook()

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

/\*To-do:

Define a macro named as "mainSENDER\_1" with value 1

It is used as a unique ID of the sender task 1. \*/

//unsigned char mainSENDER\_1=1;

#define mainSENDER\_1 1

#define mainSENDER\_2 2

/\*To-do:

Define another macro named as "mainSENDER\_2" with value 2

It is used as a unique ID of the sender task 2. \*/

//unsigned char mainSENDER\_2=2;

/\* The tasks to be created. Two instances are created of the sender task while

only a single instance is created of the receiver task. \*/

static void vSenderTask( void \*pvParameters );

static void vReceiverTask( void \*pvParameters );

/\*-----------------------------------------------------------\*/

/\* Declare a variable of type xQueueHandle. This is used to store the queue

that is accessed by all three tasks. \*/

xQueueHandle xQueue;

/\* To-do:

Declare a struct type "xData" with two elements that will be passed on the queue.

The first element "ucValue" is of unsigned char type that holds the data sent to the queue.

The second element "ucSource" is of unsigned char type that holds the ID of the sender.

Refer to the lab sheet. \*/

typedef struct{

unsigned char ucValue;

unsigned char ucSource;

}xData;

/\* To-do:

Define a constant array named as xStructsToSend of the type "xData" with two elements that will be passed on the queue.

Refer to the lab sheet. \*/

static const xData xStructsToSend[2]=

{

{100, mainSENDER\_1},

{200, mainSENDER\_2}

};

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

puts("Experiment 11 starts\r\n");

/\* The queue is created to hold a maximum of 3 structs of type xData. \*/

/\* To-do:

Call the xQueueCreate() API to create a queue that holds a maximum of 3 xData

type data. And assign the created queue to to xQueue. \*/

xQueue= xQueueCreate(3,sizeof(xData));

if( xQueue != NULL )

{

/\* Create two instances of the task that will write to the queue. The

parameter is used to pass the struct that the task should write to the

queue, so one task will continuously send xStructsToSend[ 0 ] to the queue

while the other task will continuously send xStructsToSend[ 1 ].

Both tasks are created at priority 2 which is above the priority of the receiver. \*/

xTaskCreate( vSenderTask, "Sender1", 240, ( void \* ) &( xStructsToSend[ 0 ] ), 2, NULL );

xTaskCreate( vSenderTask, "Sender2", 240, ( void \* ) &( xStructsToSend[ 1 ] ), 2, NULL );

/\* Create the task that will read from the queue. The task is created with

priority 1, so below the priority of the sender tasks. \*/

xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 1, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

else

{

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

}

for( ;; );

}

/\*-----------------------------------------------------------\*/

static void vSenderTask( void \*pvParameters )

{

portBASE\_TYPE xStatus;

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

/\* As per most tasks, this task is implemented within an infinite loop. \*/

for( ;; )

{

/\*To-do:

Call the xQueueSendToBack() API function to send data to the queue by passing

three parameters as described below; and assign the returned value to xStatus.

The 1st parameter is the queue to which data is being sent. The

queue xQueue was created before the scheduler was started, so before this task

started to execute.

The 2nd parameter is the address of the struct being sent. The

address is passed in as the task parameter "pvParameters".

The 3rd parameter is the Block time xTicksToWait" that the task should be kept

in the Blocked state to wait for space to become available on the queue

should the queue already be full. A block time is specified as the queue

will become full. Items will only be removed from the queue when both

sending tasks are in the Blocked state.. \*/

xStatus=xQueueSendToBack(xQueue, pvParameters,xTicksToWait);

if( xStatus != pdPASS )

{

/\* We could not write to the queue because it was full - this must

be an error as the receiving task should make space in the queue

as soon as both sending tasks are in the Blocked state. \*/

puts( "Could not send to the queue.\r\n" );

}

/\* Allow the other sender task to execute. \*/

taskYIELD();

}

}

/\*-----------------------------------------------------------\*/

static void vReceiverTask( void \*pvParameters )

{

/\*To-do:

Declare a variable "xReceivedStruct" with the type of "xData" struct.

It will hold the values received from the queue. \*/

xData xReceivedStruct;

portBASE\_TYPE xStatus;

/\* This task is also defined within an infinite loop. \*/

for( ;; )

{

/\* As this task only runs when the sending tasks are in the Blocked state,

and the sending tasks only block when the queue is full, this task should

always find the queue to be full. 3 is the queue length. \*/

/\* To-do:

Check whether xQueue is full (equal to 3) by calling the uxQueueMessagesWaiting() with the input

parameter xQueue.

If the return value of this function is not equal to 3, print a message "Queue should have been full!\r\n". \*/

if (uxQueueMessagesWaiting(xQueue)!=3 )

{

printf("Queue should have been full!\r\n");

}

/\*To-do:

Call the xQueueReceive() API function to send data to the queue by passing

three parameters as described below; and assign the returned value to xStatus.

The first parameter is the queue from which data is to be received. The

queue is created before the scheduler is started, and therefore before this

task runs for the first time.

The second parameter is the buffer into which the received data will be

placed. In this case the buffer is simply the address of a variable

-- &xReceivedStruct. This variable has the required size to hold the received struct.

The last parameter is the block time - the maximum amount of time that the

task should remain in the Blocked state to wait for data to be available

should the queue already be empty. A block time is set to 0 as this

task will only run when the queue is full so data will always be available. \*/

xStatus= xQueueReceive(xQueue, &xReceivedStruct,0);

if(xStatus == pdPASS)

{

/\* Data was successfully received from the queue, print out the received

value and the source of the value. \*/

if( xReceivedStruct.ucSource == mainSENDER\_1 )

{

printf( "From Sender 1 = %d \r\n", xReceivedStruct.ucValue );

}

else

{

printf( "From Sender 2 = %d \r\n", xReceivedStruct.ucValue );

}

}

else

{

/\* We did not receive anything from the queue. This must be an error

as this task should only run when the queue is full. \*/

puts( "Could not receive from the queue.\r\n" );

}

}

}