Semaphore using FreeRTOS on LPC2148

e-Yantra Team

July 19, 2016

Contents

1 What is FreeRTOS

For statrters FreeRTOS is just a bunch of C files which enables us to implement RTOS in around 32 microcontrollers. FreeRTOS provides files which can be used in multiple microcontrollers with some microcontroller specific support files.

The main advantage of Implementing FreeRTOS in any microcontroller is the ability to multi-task.

MultiTasking enables the device to execute multiple "Tasks" at the same time. MultiTasking in single core systems is implemented by allocating each task a time slice of the processor, In this way Multiple Tasks can be executed at the "same time".

1.1 Advantage of FreeRTOS

1. Proper Utilization of Resources.
2. Low foot-print.
3. Priority based scheduling of Tasks.
4. API's for Semaphores.
5. API's for Making and managing queues.

1.2 Drawbacks/Disadvantages (write some fancy word)

1. Use of TaskNotification to implement MailBox.
2. Not readily Portable to all devices.
3. Limited source material.

2 Requirement

1. Knowledge of C++
2. FreeRTOS source files/API
3. Keil compiler
4. Flash magic
5. FireBird V (LPC2148)

3 Getting Started !

3.1 DataTypes

FreeRTOS defines counterparts of few basic data types

Data Type	**General Data Type**
portCHAR	char
portSHORT	short
portLONG	long
portTickType	This is used to store the tick count
portBASE_TYPE	Generally used for Bool type data,is 32 bit for 32 bit type uC

3.2 Variable Names

The Data type is prefixed to the name of a variable for e.g

In vTaskDelay "v" denotes the return type "void".

In xTaskCreate "x" denotes portBASE_TYPE.

3.3 Macros

Refer to page 168-169 of the RTOS document by Richard Barry.

3.4 Creating Tasks

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
const char * const pcName,
unsigned short usStackDepth,
void *pvParameters,
UBaseType_t uxPriority,
TaskHandle_t *pxCreatedTask
);
```

- BaseType_t :Can be used to check if the task has been created or not.If the returned value is pdTRUE the task has been created if pdFALSE is returned the task was not created.
- pvTaskCode:This parameter is a pointer to the task which has been created.
- pcName :Name given to a Task created so that user can easily identify a task,this parameter enables the programmer to easily identify a task.
- usStackDepth :The amount of memory/space which a given task is to be allocated is passed as a parameter through this value.
- uxPriority :Each task is assigned a priority on the basis of which it is allocated the processor time.priority assigned are natural numbers ,as the value of number increases priority increases.
- pxCreatedTask :Tasks are assigned handles using which they can be referred by other tasks.

Same Task can have multiple instances by varying the priority,parameters passed ,pcName.

3.5 Frequently used API's

- vTaskDelay :Takes the Clock Ticks as parameter and suspends the Task for those many cycles. e.g vTaskDelay(1000);
- vTaskSuspend :Takes Taskhandle as a parameter and Suspends the "passed" task indefinitely. e.g. vTaskSuspend(t1) : suspends task t1 vTaskSuspend(NULL) : suspends running task
- vTaskResume :Also Takes Taskhandle as a parameter resumes the Task from suspended state e.g. vTaskResumed(t1) : resumes task t1
- tskIDLE_PRIORITY: Priority of the idle task,used to fix priority of Tasks created.

4 Introduction to semaphore

There are a limited number of resources available to any system,Similarly any microcontroller has a limited number resources available.

As the complexity of the application Increases the number of Tasks running also Increases,more and more Tasks compete for the available Processor time or The I/O devices available.

To ensure equal availability of resources to all the Tasks Operating Systems provide a facilities through semaphores.
The Greek word sema means sign or signal, and -phore means carrier . So Semaphore = signalling.
Semaphores can be classified into

- Binary Semaphores
- Mutex
- Counting Semaphores

4.1 Binary Semaphores

Binary semaphores are used for Task synchronisation. If a process ocuppies a resource the value of Binary semaphore is 1 else 0 i.e it gives information only if the resource is available or not.

4.2 Mutex

Mutex stands for Mutual Exclusion.Any Task which requires a resource can "Block" the resource.when the Task uses the resource it can "Give" the resource.

4.3 Counting Semaphore

Counting semaphores are used to count resources and keep track of Multiple resources.

4.4 Mutex vs Binary Semaphore

- Mutexes are used for Resource Protection from other tasks//processes whereas Binary semaphores are used for task synchronistaion

- It is the responisibility of the occupying function to release the mutex,but a binary semaphore can be released even from ISR or any other functions.

- On the implementation level it is the Responibility of the Coder to ensure that the Mutex is only given by the task which takes it.

5 Binary Semaphore

5.1 Code :

```
#include<stdlib.h>
#include"FreeRTOS.h"
#include"task.h"
#include"LCD.h"
#include"semphr.h"

SemaphoreHandle_t xSemaphore;

//Look in the sample programs for Included functions variables etc

void forward(void *pvparam)
{        vTaskDelay(5); //Added so that Back Task can occupy the resource
        while(1)
        {
         if(xSemaphoreTake(xSemaphore,portMAX_DELAY)==pdTRUE)
                {        Stop();
                         Forward();
                   UART0_SendStr("Forward\n");
                vTaskDelay(5);//To avoid same Tasking Taking resources tu
                }


        }
}

void back(void *pvparam)
{
        while(1)
        {
         if(xSemaphoreTake(xSemaphore,portMAX_DELAY)==pdTRUE)
                {        Stop();
                         Back();
                  UART0_SendStr("Back\n");
                vTaskDelay(5);
                }

        }
}

void control_switcher(void *pvparam)
```

```
{
  while(1)
  {xSemaphoreGive(xSemaphore);
   UART0_SendStr("Semaphore given\n");
   vTaskDelay(1200);


   }
}


  int main()
{
 PINSEL0 = 0x00000000;           // Reset all pins as GPIO
 PINSEL1 = 0x00000000;
 PINSEL2 = 0x00000000;
 DelaymSec(40);
        Init_Peripherals();

        UART0_SendStr("\t\tBinary Semaphore\n");
    xSemaphore=xSemaphoreCreateBinary( );

        xTaskCreate(forward,"forward", 300 ,NULL, tskIDLE_PRIORITY + 1, N
        xTaskCreate(back,"back", 300 ,NULL, tskIDLE_PRIORITY + 1, NULL);
        xTaskCreate(control_switcher,"control_switcher", 300 ,NULL, tskIDL

        vTaskStartScheduler();  //Task Scheduling

        while(1);


}
```

5.2 Explanation

- **Variable declaration**

```
SemaphoreHandle_t  xSemaphore;
```

  This statement declares a variable of type "SemaphoreHandle_t"

- **Creation of the semaphore**

```
xSemaphore=xSemaphoreCreateBinary( );
```

- **Working of code**

  The forward function Waits for portMAX_DELAY i.e for maximum amount of time so that the control of Resources is available.

  Similarly the back function waits for maximum time to get access to the resources.

  As soon as execution of Tasks starts the resources are occupied by the back function(vTaskDelay restricts forward function),The control_switcher function is suspended for 1200 clock counts and Gives away the semaphore.

  As soon as the semaphore is released the forward function waiting for allocation of resources occupies them,the cycle continues with control_switcher releasing the semaphore.

- **Serial monitor Output**

```
                    Binary Semaphore
Semaphore given
Back
Semaphore given
Forward
Semaphore given
Back
Semaphore given
Forward
Semaphore given
Back
```

6 Mutex

6.1 Code :

```
#include<stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "LCD.h"
#include"semphr.h"

//Refer to actual code for necessary functions and codes

SemaphoreHandle_t xSemaphore=0;//Creation of Variable for semaphore

  void forward(void *pvparam)
{
        while(1)
        {
                if(xSemaphoreTake(xSemaphore,1000) == pdTRUE )
// if available then
                {   UART0_SendStr("Forward\n");
                        Forward();
                    vTaskDelay(1200);                   // task suspended
                        Stop();
                        xSemaphoreGive( xSemaphore );
// after resource task completed, return the semaphore
                }

                else
                {       UART0_SendStr("Forward function access denied\n")

                 vTaskDelay(200);
        }
}

void back(void *pvparam)
{          while(1)
        {
                        if(xSemaphoreTake(xSemaphore,1000) == pdTRUE )
                        {
                        UART0_SendStr("Back\n");
                        Back();
                        vTaskDelay(1200);                       // perform
data tasks();
```

```
                    Stop();
                    xSemaphoreGive( xSemaphore );
// after shared data task completed, return the semaphore
        }

        else        // if available then
                { UART0_SendStr("Back Function access denied\n");

        vTaskDelay(200);                                         /
}
}

  int main()
{
 PINSEL0 = 0x00000000;          // Reset all pins as GPIO
 PINSEL1 = 0x00000000;
 PINSEL2 = 0x00000000;
 DelaymSec(40);
        Init_Peripherals();

        UART0_SendStr("\t\tMutex\n");
        xSemaphore = xSemaphoreCreateMutex();    //Use the Handle as a MUT

        xTaskCreate(forward,"forward", 300 ,NULL, tskIDLE_PRIORITY + 1, N
        xTaskCreate(back,"back", 300 ,NULL, tskIDLE_PRIORITY + 1, NULL);

        vTaskStartScheduler();   //Task Scheduling

        while(1);
}
```

6.2 Explanation

- **Variable declaration**

  ```
  SemaphoreHandle_t xSemaphore;
  ```

  This statement declares a variable of type "SemaphoreHandle_t"

- **Creation of Mutex**

  ```
  xSemaphore = xSemaphoreCreateMutex();
  ```

- **Working of code**

  There are Two Tasks forward and back, when executed

  The forward function Waits for 1000 clock cycles for the resources,In case the resources are not available the Task sends a message about The lack of availability of resources. Similarly the back function waits for same amount of time for resources.

  As soon as execution of Tasks starts the resources are occupied by one of the the task and that task blocks the acess of those resources through a mutex.

  The task executes and when the execution is completed it "Gives" the Mutex and therefore the releases the resources,another waiting task then occupies those resources and blocks for a period of time it requires.

- **Serial monitor Output**

  ```
  Mutex
  Back
  Forward function access denied
  Forward
  Back Function access denied
  Back
  Forward function access denied
  Forward
  Back Function access denied
  Back
  ```

7 Counting Semaphore

:Implemented by dining Philosophers Problem

7.1 Code :

```
/*
Note: To use mutex semaphore you need to initialize  configUSE_MUTEXES to
*/

#include<stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "LCD.h"
#include"semphr.h"

SemaphoreHandle_t xSemaphore=0;//Creation of Variable for semaphore

int s=0;
int forks_avail[5]={0,0,0,0,0}; //The value of Variable is 0 if a fork is


void vfork( void * pvParameters )
{
        int i;
        const unsigned char* str;
        str = ( const unsigned char * ) pvParameters;


        //Assignment of forks available on the basis of name of Philosphe
        if(str[1]=='1')
        {i=0;}
        if(str[1]=='2')
        {i=1;}
        if(str[1]=='3')
        {i=2;}
        if(str[1]=='4')
        {i=3;}
        if(str[1]=='5')
        {i=4;}

        while(1)
        {
        //Waits for 1000 ticks for forks to be avaliable
```

```
        //If available checks if the fork is adjacent(Right) or not
        if(( xSemaphoreTake( xSemaphore, 1000 ) == pdTRUE )&&(forks_avail
{
                forks_avail[i]=1;


                UART0_SendStr(&str[0]);
                UART0_SendStr(":Right fork obtained\n");

        if(( xSemaphoreTake( xSemaphore, 2000 ) == pdTRUE )&&(forks_avail
        { //Waits for 2000 ticcks for Left fork to be available

                        forks_avail[(i+1)%5]=1;

                        UART0_SendStr(&str[0]);
                        UART0_SendStr(":Left fork obtained Eating :)\n");

                        vTaskDelay(2000);
                        UART0_SendStr(&str[0]);
                        UART0_SendStr(":Ate \n");

                        xSemaphoreGive(xSemaphore);
                        xSemaphoreGive(xSemaphore);
                        forks_avail[i]=0;
                        forks_avail[(i+1)%5]=0;
                    UART0_SendStr(&str[0]);
                        UART0_SendStr(":Thinking \n");
                        vTaskDelay(3000);
}

                else
                {
            UART0_SendStr(&str[0]);
                UART0_SendStr(":Returned Right fork:( \n");
                xSemaphoreGive(xSemaphore);
                forks_avail[i]=0;
                }

        }

    else
            {
            UART0_SendStr(&str[0]);
            UART0_SendStr(":Hungry\n");
```

```
            vTaskDelay(3000);
                  }
                  }
 }



  int main()
{
 PINSEL0 = 0x00000000;              // Reset all pins as GPIO
 PINSEL1 = 0x00000000;
 PINSEL2 = 0x00000000;
 Init_Peripherals();


        UART0_SendStr("\t\tCounting Semaphore\n");

        xSemaphore = xSemaphoreCreateCounting( 5, 5 );

         if( xSemaphore != NULL )
    {   UART0_SendStr("\tSemaphore Created\n");


xTaskCreate(vfork,"Philospher 1", 300 ,"P1", tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(vfork,"Philospher 2", 300 ,"P2", tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(vfork,"Philospher 3", 300 ,"P3", tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(vfork,"Philospher 4", 300 ,"P4", tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(vfork,"Philospher 5", 300 ,"P5", tskIDLE_PRIORITY + 1, NULL);

                vTaskStartScheduler();   //Task Scheduling
     }

        while(1)//Never reaches this Part of the main
        {UART0_SendStr("\t\tSemaphore not Created\n");    }

}
```

7.2 Explanation

- **Variable declaration**

  ```
  SemaphoreHandle_t xSemaphore;
  ```

  This statement declares a variable of type "SemaphoreHandle_t"

- **Creation of Counting semaphore**

  ```
  xSemaphore = xSemaphoreCreateCounting( 5, 5 );
  ```

  Here 1st parameter gives the maximum count and 2nd parameter is the initial count. If the semaphore is used for counting events 2nd parameter would be 0 and if used for resources management it would be equal to maximum or initial count.

- **Task Creation**

  ```
  xTaskCreate(vfork,"Philospher 1", 300 ,"P1",
   tskIDLE_PRIORITY + 1, NULL);
  .
  .
  ```

  Here vfork is a single Task which on variation of Parameter P1,P2...etc behaves as a different task,ecah task has its own stack and act as if they are independent.All the tasks have same priority and get equal time at the processor.

- **Working of code**

  The Tasks created are by changing the parameters of a single task.

  When each time a "Philosopher" is allocated the processor time it checks for the number of available "Forks".If the forks are available and then check for the Right fork and the philosopher "picks up the left fork" then when the "Philosopher" again gains the processor time it waits for Left fork to be available and proceeds to eat.

  when 5 "Philosophers" are allocated simulatenously the semaphore keeps track of the available forks .

- **Serial monitor output**

```
P3:Hungry
P5:Ate
P5:Thinking
P4:Left fork obtained Eating :)
P2:Right fork obtained
P4:Ate
P4:Thinking
P2:Left fork obtained Eating :)
P1:Right fork obtained
P3:Hungry
P5:Hungry
P2:Ate
P2:Thinking
P1:Left fork obtained Eating :)
P4:Right fork obtained
P1:Ate
P1:Thinking
P4:Left fork obtained Eating :)
P3:Right fork obtained
P5:Hungry
P2:Hungry
P4:Ate
P4:Thinking
P3:Left fork obtained Eating :)
P1:Right fork obtained
P3:Ate
P3:Thinking
P1:Left fork obtained Eating :)
P5:Right fork obtained
P2:Hungry
P4:Hungry
P1:Ate
P1:Thinking
```

8 Task Notification

There occurs instances when tasks needs to communicate with each other.Semaphores are one of the methods by which tasks communicate with each other.Two other methods by which tasks communicate with each other are

1. MailBox
2. Queues

Tasks in mailbox communicate by sending "Mails" to each other.In FreeRTOS mailbox is implemented by Task Notification.

Each Task has an associated notification value using which they can be "notified".When a task is notified, Task notifications can update the receiving task's notification value in the following ways:

- Set the receiving task's notification value without overwriting a previous value
- Overwrite the receiving task's notification value
- Set one or more bits in the receiving task's notification value
- Increment the receiving task's notification value

8.1 Code:

```
#include<stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include"semphr.h"

TaskHandle_t xHandle = NULL;


void vnoticer( void * pvParameters )
 {
  uint32_t ulNotifiedValue=0x01;

    while(1)
    {

       if( xTaskNotifyWait( 0x00,0xffff,&ulNotifiedValue,1000 )==pdTRUE)
 {


                 if( ( ulNotifiedValue | 0x01 ) == 0x01 )
//checking if the received message is same as the sent
  {  UART0_SendStr("Received   MSG from N1 \n");            }

        else if( ( ulNotifiedValue | 0x02 ) == 0x02 )
  {  UART0_SendStr("Received   MSG from N2\n");             }

                 else if( ( ulNotifiedValue | 0x03 ) == 0x03 )
  {  UART0_SendStr("Received   MSG from N3\n");             }

        else if( ( ulNotifiedValue | 0x04 ) == 0x04 )
  {   UART0_SendStr("Received   MSG from N4\n");   }

           else
  {   UART0_SendStr("Learn Programming !\n");}

}

else
{ UART0_SendStr("No Notice\n"); }

 }
}
```

```
void vn1( void * pvParameters )
{

   xHandle = xTaskGetHandle( "Noticer" );

   while(1)
   {
         vTaskDelay(4000);
         UART0_SendStr("N1 sent a Message\n");
         xTaskNotify(xHandle, 0x01, eSetBits);


         }
}
void vn2( void * pvParameters )
{

   xHandle = xTaskGetHandle( "Noticer" );

   while(1)
   {
         vTaskDelay(5000);
         UART0_SendStr("N2 sent a Message\n");
         xTaskNotify(xHandle, 0x02, eSetBits);


         }
}
void vn3( void * pvParameters )
{

   xHandle = xTaskGetHandle( "Noticer" );

   while(1)
   {
         vTaskDelay(6000);
         UART0_SendStr("N3 sent a MSG\n");
         xTaskNotify(xHandle, 0x03, eSetBits);


         }
}
void vn4( void * pvParameters )
```

```
{

   xHandle = xTaskGetHandle( "Noticer" );

   while(1)
   {
         vTaskDelay(7000);
         UART0_SendStr("N4 sent a MSG\n");
         xTaskNotify(xHandle, 0x04, eSetBits);


         }
}


  int main()
{
 PINSEL0 = 0x00000000;              // Reset all pins as GPIO
 PINSEL1 = 0x00000000;
 PINSEL2 = 0x00000000;
 Init_Peripherals();



         UART0_SendStr("\t\tMailBox using Task Notification\n");

                  xTaskCreate(vn1,"Notifier", 300 ,NULL, tskIDLE_PRIORITY +
                  xTaskCreate(vn2,"Notifier", 300 ,NULL, tskIDLE_PRIORITY +
                  xTaskCreate(vn3,"Notifier", 300 ,NULL, tskIDLE_PRIORITY +
                  xTaskCreate(vn4,"Notifier", 300 ,NULL, tskIDLE_PRIORITY +
         xTaskCreate(vnoticer,"Noticer", 300 ,NULL, tskIDLE_PRIORITY + 1,




                  vTaskStartScheduler();   //Task Scheduling

         while(1)//Never reaches this Part of the main
         {
         UART0_SendStr("\t\tMailBox Bypassed\n");
         }

}
```

8.2 Explanation

Above is a simple code which has four tasks(vn1,vn2,vn3,vn4) which notify a 5th task 'noticier',The 5th task prints which task notified it.

- **xTaskNotify()** This function is used to notify other tasks general format is as specified below

  ```
  xTaskNotify(xHandle, 0x03, eSetBits);
  \\(Task Handle,Notification value,eAction)
  ```

  Parameters

  - **Task handle**:The handle of the task which needs to be notified.
  - **Notification value**:Value used for notification
  - **eAction**:The type of action which is to be carried out upon the specified task.The types are as specified below:
    - eNoAction :The Task receives the value but no action takes place,can used to Resume a suspened task.
    - eSetBits :The existing Notification value will be Bitwise OR-ed with the Notified value to obtain a new value.
    - eIncrement :Increments the existing value.
    - eSetValueWithOverwrite :Overwrites the existing Notification value.

  Return value : Returns pdTRUE if Task has been Notified else pdFALSE.

- **xTaskNotifyWait()** The Function waits to receive a Notification and has parameters which govern the actions upon the received data.

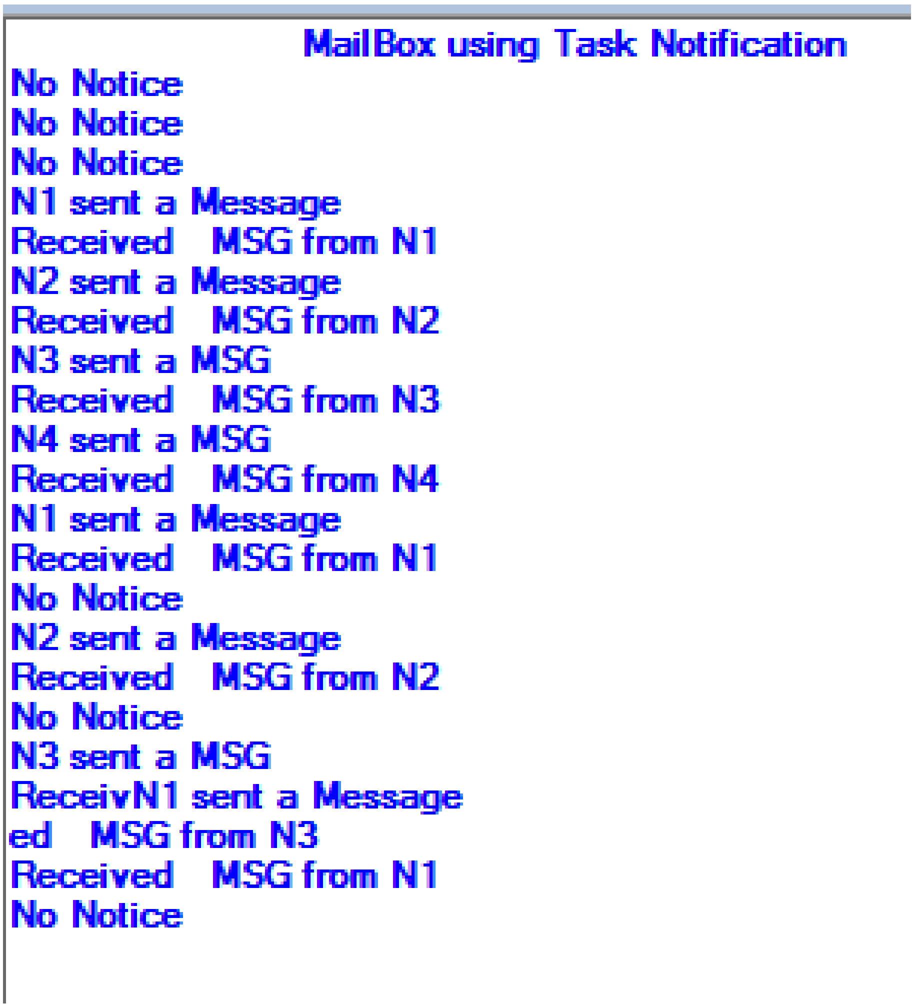
  ```
  xTaskNotifyWait( 0x00,0xffff,&ulNotifiedValue,1000 )
  \\(clear bits on entry,clear bits on exit,notified value,time
  ```

  Parameters

  - **ulBitsToClearOnEntry**:Specifies the Bit position which needs to be cleared as soon as the Notification is received.
  - **ulBitsToClearOnExit** :Specifies the Bit position which needs to be cleared before xTaskNotifyWait() function exits if a notification was received
  - **pulNotificationValue**:The Notification value before exit is taken and stored in this.

– **xTicksToWait** :This specifies the timeout period for which the function call waits for a notification.

- **Working**
  Tasks vn1,vn2,vn3,vn4 With different frequencies send a ”message” to a noticer task through a hex value,these hex values are compared to find out which task sent the message.
  For the first few ticks no task is sending a notification so the noticier prints a ”No Notice” message,as It starts receiving messsages it starts acknowledging the received messages.



9 Queue

9.1 Intro

In RTOS, inter-process communication is possible. It means that you can communicate between two task and control them on the basis of this communication.
But first we need to understand **what is queue?**
Consider a dynamic buffer. Dynamic in the sense of memory allocation. We can allocate the size of the buffer as per our requirements. There are two things that we can change, one is the size of each data the buffer can carry and other is the number of data the buffer can send with each data of the size defined by us. this buffer is known as queue.

9.2 Code

```
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "queue.h"
//assuming necessary functions and header files have been included
unsigned char Temp=0;
int count=0;
QueueHandle_t xQueue= 0;

void Txtask(char *);
void Rxtask(void *);


char *tx1={"Task 1 "};
char *tx2={"Task 2"};
char *tx3={"Task 3"};
char *tx4={"Task 4"};



void Txtask(char *p)                    // task which writes data on to t
{
        while(1)
        {
```

```
                  if(xQueueSend(xQueue,p,1000) == pdTRUE)
// wait for 1000ms to tx queue message
                  {         UART0_SendStr("\nData sent to Queue : \t");
                            UART0_SendStr(p);
                  vTaskResume("RxTask");
                  //Data added to Q
                                  }
                  else
                  {
                  }
         //       vTaskDelay(2000);
         }

}

void Rxtask(void *p)                         // task which reads data from the

{

unsigned char rx_success_count[11]={0};
unsigned char *rxptr;
rxptr = rx_success_count;

         while(1)
         {
                  if(xQueueReceive(xQueue,rxptr,1000) == pdTRUE)           /
                  {         UART0_SendStr("\n");
                            UART0_SendStr("Data read from Queue : \t");
                            UART0_SendStr(rxptr);
                            vTaskDelay(40);
// if RX success then display  rx_success_count
                  }
                  else

         {        vTaskSuspend(NULL);
         }
         }
}



int main()
{
 Init_UART0();
```

```
                                  /*create queue of length=3 and of size i
    xQueue = xQueueCreate(7,40);
    UART0_SendStr("Queue\n");
                                  /* creating the 2 task with the same prio
    xTaskCreate(Txtask,"TxTask_1", configMINIMAL_STACK_SIZE,tx1, tskID
    xTaskCreate(Txtask,"TxTask_2", configMINIMAL_STACK_SIZE,tx2, tskID
    xTaskCreate(Txtask,"TxTask_3", configMINIMAL_STACK_SIZE,tx3, tskID
    xTaskCreate(Txtask,"TxTask_4", configMINIMAL_STACK_SIZE,tx4, tskID
    xTaskCreate(Rxtask,"RxTask", configMINIMAL_STACK_SIZE,NULL, tskIDL

  vTaskStartScheduler();       /* Start the scheduler so the tasks star
*/

  while(1)
    {
      ; /* Should never get here!
    }
}
```

9.3 Explanation

- **QueueHandle_t** :A predefined data type used to reference a queue.

  ```
  QueueHandle_t xQueue= 0;
  ```

- **xQueueCreate(a,b)**:Creates a queue of 'a' continuous memory locations,where each memory location is of 'b' bytes each.It returns a 'pointer' to the queue which is stored in the queuehandle.

  ```
  xQueue = xQueueCreate(7,40);
  ```

- **xQueueSend(QueueHandle,data,timeout period)**:This function is used to send data to queue,It has three parameters

  1. **QueueHandle**:It gives the address of the queue in which data has to be stored.
  2. **Data**:The data which needs to be stored in the queue.
  3. **Timeout period**:This specifies the amount of time for which the function waits if the queue is unavailable(i.e data is being sent to queue or queue is full)

  Return value : Returns pdTRUE if Data has been sent to Queue else pdFALSE e.g..

  ```
  if(xQueueSend(xQueue,p,1000) == pdTRUE)
  ...
  ```

- **xQueueReceive(QueueHandle,data,timeout period):**This function is used to receive data from a queue,It has three parameters

    1. **QueueHandle**:It gives the address of the queue from which data has to be obtained.
    2. **Data**:A variable in which the poped data has to be stored.
    3. **Timeout period**:This specifies the maximum amount of time for which the function waits if the queue is unavailable(i.e data is being received by another task or queue is empty)

    Return value : Returns pdTRUE if Data has been Received from the Queue, pdFALSE id queue is empty e.g..

    ```
    if(xQueueReceive(xQueue,rx,1000) == pdTRUE)
    ...
    ```

- **Working of code:** Initially a queue is created which can accomodate 7 elements in which each of them can occupy 40 Bytes.

    There are 4 tasks which send data to the queue and one task which receives data from the queue.

    As the tasks are created data is pushed into the queue and the receiving task is resumed which inturn pops the data and prints them through the serial comm port.

    From the Output screenshots it can be observed how data pushed 1st is popped out 1st (Queue mechanism).

Queue

Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data sent to Queue : Task 1
Data read from Queue : Task 1
Data sent to Queue : Task 1
Data read from Queue : Task 1
Data sent to Queue : Task 2
Data read from Queue : Task 1
Data sent to Queue : Task 3
Data read from Queue : Task 1
Data sent to Queue : Task 4
Data read from Queue : Task 1
Data sent to Queue : Task 1
Data read from Queue : Task 1
Data sent to Queue : Task 2
Data read from Queue : Task 1
Data sent to Queue : Task 3
Data read from Queue : Task 1
Data sent to Queue : Task 4
Data read from Queue : Task 2
Data sent to Queue : Task 1
Data read from Queue : Task 3
Data sent to Queue : Task 2
Data read from Queue : Task 4
Data sent to Queue : Task 3
Data read from Queue : Task 1
Data sent to Queue : Task 4

10 References

1. http://www.rtos.be/2013/05/mutexes-and-semaphores-two-concepts-for-two-different-use-cases/

2. http://www.ocfreaks.com/cat/embedded/lpc2148-tutorials/

3. http://www.freertos.org/Inter-Task-Communication.html

4. http://tinymicros.com/

5. http://www.profdong.com/elc4438_spring2016/USINGTHEFREERTOSREALTIMEKERNEL.pdf