EMBEDDED SYSTEMS LABORATORY EXPERIMENTS

|  |  |
| --- | --- |
| **Ex No:8.a** | **RTOS PROGRAMMING ENVIRONMENT** |
| **Date:** |

# Aim:

To Study about the task life cycle, creation of tasks and deletion of tasks, Queues and Resource

Management in FREE RTOS.

# Introduction:

FreeRTOS is an free and opensource RealTime Operating system developed by Real Time Engineers Ltd. Its design has been developed to fit on very small embedded systems and implements only a very minimalist set of functions: very basic handle of tasks and memory management, just sufficient API concerning synchronization, and absolutely nothing is provided for network communication, drivers for external hardware, or access to a filesystem.

However, among its features are the following characteristics: preemptive tasks, a support for 23 microcontroller architectures1 by its developers, a small footprint2 (4.3Kbytes on an ARM7 after compilation3), written in C and compiled with various C compiler (some ports are compiled with gcc, others with openwatcom or borland c++). It also allows an unlimited number of tasks to run at the same time and no limitation about their priorities as long as used hardware can afford it. Finally, it implements queues, binary and counting semaphores and mutexes.

## A task in FreeRTOS

FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks. In RTOS, a task is defined by a simple C function, taking a void\* parameter and returning nothing (void).

Several functions are available to manage tasks: task creation (vTaskCreate()), destruction (vTaskDelete()), priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()). More options are available to user, for instance to create a critical sequence or monitor the task for debugging purpose.

# Life cycle of a task

This section will describe more precisely how can a task evolve from the moment it is created to when it is destroyed. In this context, we will consider to be available only one microcontroller core, which means only one calculation, or only one task, can be run at a given time. Any given task can be in one of two simple states : “running” or “not running”. As we suppose there is only one core, only one task can be running at a given time; all other tasks are in the “not running task. Figure 1 gives a simplified representation of this life cycle. When a task changes its state from “Not running” to running, it is said “swapped in” or “switched in” whereas it is called “swapped out” or “switched out” when changing to “Not running” state. As there are several reasons for a task not to be running, the “Not running” state can be expanded as shows Figure 2. A task can be preempted because of a more priority task (scheduling is described in section 2), because it has been delayed or because it waits for a event. When a task can runs but is waiting for the processor to be available, its state is said “Ready”. This can happen when a task has it needs everything to run but there is a more priority task running at this time. When a task is delayed or is waiting for another task (synchronisation through semaphores or mutextes) a task is said to be “Blocked”. Finally, a call to vTaskSuspend() and vTaskResume() or xTaskResumeFromISR() makes the task going in and out the state “Suspend”.

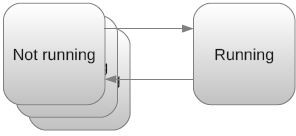


Fig 1 Simplified life cycle of a task : Only one task can be "running" at a given time, whereas the “not running state can be expanded“.

It is important to underline that a if a task can leave by itself the “Running” state (delay, suspend or wait for an event), only the scheduler can “switch in” again this task. When a task wants to run again, its state turns to “Ready” an only the scheduler can choose which “Ready” task is run at a given time.

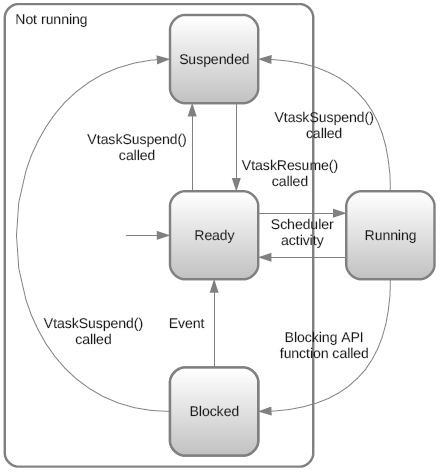


Fig 2 Life Cycle of a Task

## Creating and deleting a task

A task defined by a simple C function, taking one void\* argument and returning nothing. Any created task should never end before it is destroyed. It is common for task's code to be wrapped in an infinite loop, or to invoke vTaskDestroy(NULL) before it reaches its final brace. As any code in infinite loop can fail and exit this loop, it is safer even for a repetitive task, to invoke vTaskDelete() before its final brace.

**portBASE\_TYPE xTaskCreate( pdTASK\_CODE pvTaskCode, const signed portCHAR \* const pcName, unsigned portSHORT usStackDepth, void \*pvParameters, unsigned portBASE\_TYPE uxPriority, xTaskHandle \*pxCreatedTask );**

A task can be created using vTaskCreate(). This function takes as argument the following list:

* pvTaskCode: a pointer to the function where the task is implemented.
* pcName: given name to the task. This is useless to FreeRTOS but is intented to debugging purpose only.
* usStackDepth: length of the stack for this task in words. The actual size of the stack depends on the
* micro controller. If stack with is 32 bits (4 bytes) and usStackDepth is 100, then 400 bytes (4 times 100) will be allocated for the task.
* pvParameters: a pointer to arguments given to the task. A good practice consists in creating a dedicated
* uxPriority: priority given to the task, a number between 0 and MAX\_PRIORITIES – 1. pxCreatedTask: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.

A task is destroyed using xTaskDestroy() routine. It takes as argument pxCreatedTask which is given when the task was created. When a task is deleted, it is responsibility of idle task to free all allocated memory to this task by kernel.

# Queue Management

**void vTaskDelete( xTaskHandle pxTask );**

Queues are an underlying mechanism beyond all tasks communication or synchronization in a FreeRTOS environment. They are an important subject to understand as it is unavoidable to be able to build a complex application with tasks cooperating with each other. They are a mean to store a and finite number (named “length”) of fixedsize *run after each other in turn.* They are able to be read and written by several different tasks, and don't belong to any task in particular. A queue is normally a FIFO which means elements are read in the order they have been written. This behavior depends on the writing method: two writing functions can be used to write either at the beginning or at the end of this queue.

# Reading in a queue

**portBASE\_TYPE xQueueReceive( xQueueHandle xQueue, const void \* pvBuffer, portTickType xTicksToWait );**

When a single task reads in a queue, it is moved to “Blocked” state and moved back to “Ready” as soon as data has been written in the queue by another task or an interrupt. If several tasks are trying to read a queue, the highest priority task reads it first. Finally, if several tasks with the same priority are trying to read, the first task who asked for a read operation is chosen. A task can also specify a maximum waiting time for the queue to allow it to be read. After this time, the task switches back automatically to “Ready” state. xqueue is the identifier of the queue to be read pvBuffer is a pointer to the buffer where the read value will be copied to. This memory must be allocated and must be large enough to handle the element read from the queue. xTicksToWait defines the maximum time to wait. 0 prevents the task from waiting even if a value is not available, whereas if INCLUDE\_vTaskSuspend is set and xTicksToWait equals MAX\_DELAY, the task waits indefinitely. pdPASS is returned if a value was sucessfully read before xTicksToWait is reached. If not, errQUEUE\_EMPTY is returned from xQueueReceive(). After reading an element in a queue, this element is normally removed from it; however, an other read function given in allows to read an element without having it to be deleted from the queue.

# Writing to a queue

**portBASE\_TYPE xQueueSend( xQueueHandle xQueue, const void \* pvItemToQueue, portTickType xTicksToWait);**

Writing on a queue obeys to the same rules as reading it. When a task tries to write on a queue, it has to wait for it to have some free space: the task is blocked until another task reads the queue and free some space. If several tasks attempt to write on the same queue, the higher priority task is chosen first. If several tasks with the same priority are trying to write on a queue, then the first one to wait is chosen. It describes the normal method to write on a queue. xQueue is the queue to write on. This value is returned by the queue creation method. pvItemToQueue is a pointer to an element which is wanted to be copied (by value) to the queue. xticksToWait is the number of ticks to wait before the task gives up to write on this queue. If xTicksToWait is 0, the task won't wait at all if the queue is full. XqueueSend returns

pdPASS if the element was successfully written to the queue before the maximum waiting time was reached, or errQUEUE\_FULL if the maximum time was elapsed before the task could write on the queue.

# Creating a queue

Length of a queue and its width (the size of its elements) are given when the queue is created. uxQueueLenght gives the number of elements this queue will be able to handle at any given time. uxItemSize is the size in byte of any element stored in the queue. xQueueCreate returns NULL if the queue was not created due to lack

# xQueueHandle xQueueCreate( unsigned portBASE\_TYPE uxQueueLength, unsigned portBASE\_TYPE uxItemSize);

**Resource Management**

# Semaphore

Binary semaphores are the simplest effective way to synchronize tasks, an other even more simple, but not as effective, consists in polling an input or a resource. A binary semaphore can be seen as a queue which contains only one element.

# Mutex

Mutexes are designed to prevent mutual exclusion or deadlocking. A mutex is used similarly to a binary semaphore, except the task which take the semaphore must give it back. This can be though with a token associated with the resource to access to. A task holds the token, works with the resource then gives back the token; in the meanwhile, no other token can be given to the mutex.

# Prelab Questions

1. What is RTOS?
2. List the unique features of FREE RTOS.
3. How will you add the FREE RTOS library in Arduino?
4. What is a task in FREE RTOS?
5. Describe the life cycle of a task.

# Postlab Questions

1. How is scheduling done in FREE RTOS?
2. How can data be written and read from a queue?
3. How are resources managed in FREE RTOS?
4. What is a binary semaphore? How is it handled in FREE RTOS?
5. What is a mutex? How is it handled in FREE RTOS?

|  |  |
| --- | --- |
| **Ex No:8.b** | **RTOS – CREATING MULTIPLE TASKS** |
| **Date:** |

# Aim:

To create multiple tasks using FREE RTOS.

# Components Required:

**Hardware:** ARDUINO UNO, LED, RGB Led, Photo Resistor, Bread board and Jumper cables.

**Software:** Arduino IDE.

# A task in FreeRTOS

FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks. In RTOS, a task is defined by a simple C function, taking a void\* parameter and returning nothing (void).

Several functions are available to manage tasks: task creation (vTaskCreate()), destruction (vTaskDelete()), priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()). More options are available to user, for instance to create a critical sequence or monitor the task for debugging purpose.

# Creating and deleting a task

A task defined by a simple C function, taking one void\* argument and returning nothing. Any created task should never end before it is destroyed. It is common for task's code to be wrapped in an infinite loop, or to invoke vTaskDestroy(NULL) before it reaches its final brace. As any code in infinite loop can fail and exit this loop, it is safer even for a repetitive task, to invoke vTaskDelete() before its final brace.

**portBASE\_TYPE xTaskCreate( pdTASK\_CODE pvTaskCode, const signed portCHAR \* const pcName, unsigned portSHORT usStackDepth, void \*pvParameters, unsigned portBASE\_TYPE uxPriority, xTaskHandle \*pxCreatedTask );**

A task can be created using vTaskCreate(). This function takes as argument the following list:

* pvTaskCode: a pointer to the function where the task is implemented.
* pcName: given name to the task. This is useless to FreeRTOS but is intented to debugging purpose only.
* usStackDepth: length of the stack for this task in words. The actual size of the stack depends on the
* micro controller. If stack with is 32 bits (4 bytes) and usStackDepth is 100, then 400 bytes (4 times 100) will be allocated for the task.
* pvParameters: a pointer to arguments given to the task. A good practice consists in creating a dedicated
* uxPriority: priority given to the task, a number between 0 and MAX\_PRIORITIES – 1. pxCreatedTask: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.

A task is destroyed using xTaskDestroy() routine. It takes as argument pxCreatedTask which is given when the task was created. When a task is deleted, it is responsibility of idle task to free all allocated memory to this task by kernel.

**void vTaskDelete( xTaskHandle pxTask );**

# Including the Library:

* Free RTOS Zip file
* Extract it and paste it into the libraries folder of arduino.
* Example programs are available in examples->FreeRTOS

# Common functions Used:

**Header File: Arduino\_FreeRTOS.h**

* XTaskCreate()
  + This function is used to create multiple tasks with priority. The various arguments taken by this function is explained in the above section.
* VTaskDelay()
  + This function is used to set delay for the multiple tasks created.
  + Eg: VtaskDelay(1000/PortTICK\_PERIOD\_MS);

# Program 1: Demonstration of Creating 2 Tasks

#include <Arduino\_FreeRTOS.h> void TaskBlink();

void TaskAnalogRead(); void setup()

{

xTaskCreate( TaskBlink

, (const portCHAR \*)"Blink"

, 128 // Stack size

, NULL

, 2 // priority

, NULL );

xTaskCreate( TaskAnalogRead

, (const portCHAR \*) "AnalogRead"

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

, NULL

, 1 // priority

, NULL );

}

void loop()

{

// Empty. Things are done in Tasks.

}

void TaskBlink() // This is a task.

{

pinMode(3, OUTPUT); for (;;)

{

digitalWrite(3, HIGH);

vTaskDelay( 1000 / portTICK\_PERIOD\_MS ); digitalWrite(3, LOW);

vTaskDelay( 1000 / portTICK\_PERIOD\_MS );

}

}

void TaskAnalogRead() // This is a task.

{

pinMode(A0,INPUT); Serial.begin(9600); for (;;)

{

int sensorValue = analogRead(A0); Serial.println(sensorValue); vTaskDelay(1);

}

}

**Circuit Connection**

|  |  |  |
| --- | --- | --- |
| **Arduino** | **LED** | **LDR** |
| Vcc |  | 1st LEG |
| Gnd | LED –ve Leg | 2nd Leg + Resitor |
| A0 |  | 2nd Leg |
| 3 | LED +ve +  Resistor |  |

# Schematic

**Prelab Questions**

1. How will you create multiple tasks using FREE RTOS?
2. Explain the various parameters passed in XTaskCreate().
3. How can be provide delay between multiple tasks created?

# Exercise

* + Create 2 task – Blink of LED, Photo resistor
  + Create 2 task – RGB Blink , Fade LED
  + Create 3 task – Simulate Traffic light (sync all the 3 LED glow)

# Postlab Questions

1. Write an arduino sketch to perform the following,
   1. Create 2 independent tasks.
   2. Task 1 – implement 2 bit counter. (Use 2 LED to display the result)
   3. Task 2 – implement 3 bit counter. Display the result on the serial monitor.
   4. Draw the appropriate Circuit diagram.
2. Write an arduino sketch to perform the following,
   1. Create 2 independent tasks.
   2. Task 1 – Read the value from the keypad and display it on the serial monitor.
   3. Task 2 – Read the value from the photo resistor. If the value is above 100 – Green LED must glow. If the value is less than 100 – Red LED must glow.
   4. Draw the appropriate Circuit diagram.

|  |  |
| --- | --- |
| **Ex No:9** | **RTOS – INTER TASK COMMUNICATION** |
| **Date:** |

# Aim:

To perform inter task communication using FREE RTOS.

# Components Required:

**Hardware:** ARDUINO UNO, LED, Touch Sensor, Bread board and Jumper cables.

**Software:** Arduino IDE.

# Inter task Communication:

Different tasks in an embedded system typically must share the same hardware and software resources or may rely on each other in order to function correctly. For these reasons, embedded OSs provide different mechanisms that allow for tasks in a multitasking system to intercommunicate and synchronize their behavior so as to coordinate their functions, avoid problems, and allow tasks to run simultaneously in harmony.

Embedded OSs with multiple intercommunicating processes commonly implement inter task communication (ITC) and synchronization algorithms based upon one or some combination of memory sharing, message passing, and signaling mechanisms. The most commonly used mechanism among these is by using message queues.

# Queue Management

Queues are an underlying mechanism beyond all tasks communication or synchronization in a FreeRTOS environment. They are an important subject to understand as it is unavoidable to be able to build a complex application with tasks cooperating with each other. They are a mean to store a and finite number (named “length”) of fixedsize *run after each other in turn.* They are able to be read and written by several different tasks, and don't belong to any task in particular. A queue is normally a FIFO which means elements are read in the order they have been written. This behavior depends on the writing method: two writing functions can be used to write either at the beginning or at the end of this queue.

# Reading in a queue

**portBASE\_TYPE xQueueReceive( xQueueHandle xQueue, const void \* pvBuffer, portTickType xTicksToWait );**

When a single task reads in a queue, it is moved to “Blocked” state and moved back to “Ready” as soon as data has been written in the queue by another task or an interrupt. If several tasks are trying to read a queue, the highest priority task reads it first. Finally, if several tasks with the same priority are trying to read, the first task who asked for a read operation is chosen. A task can also specify a maximum waiting time for the queue to allow it to be read. After this time, the task switches back automatically to “Ready” state. xqueue is the identifier of the queue to be read pvBuffer is a pointer to the buffer where the read value will be copied to. This memory must be allocated and must be large enough to handle the element read from the queue. xTicksToWait defines the maximum time to wait. 0 prevents the task from waiting even if a value is not available, whereas if INCLUDE\_vTaskSuspend is set and xTicksToWait equals MAX\_DELAY, the task waits indefinitely. pdPASS is returned if a value was sucessfully read before xTicksToWait is reached. If not, errQUEUE\_EMPTY is returned from xQueueReceive(). After reading an element in a queue, this element is normally removed from it; however, an other read function given in allows to read an element without having it to be deleted from the queue.

# Writing to a queue

**portBASE\_TYPE xQueueSend( xQueueHandle xQueue, const void \* pvItemToQueue, portTickType xTicksToWait);**

Writing on a queue obeys to the same rules as reading it. When a task tries to write on a queue, it has to wait for it to have some free space: the task is blocked until another task reads the queue and free some space. If several tasks attempt to write on the same queue, the higher priority task is chosen first. If several tasks with the same priority are trying to write on a queue, then the first one to wait is chosen. It describes the normal method to write on a queue. xQueue is the queue to write on. This value is returned by the queue creation method. pvItemToQueue is a pointer to an element which is wanted to be copied (by value) to the queue. xticksToWait is the number of ticks to wait before the task gives up to write on this queue. If xTicksToWait is 0, the task won't wait at all if the queue is full. XqueueSend returns pdPASS if the element was successfully written to the queue before the maximum waiting time was reached, or errQUEUE\_FULL if the maximum time was elapsed before the task could write on the queue.

# Creating a queue

Length of a queue and its width (the size of its elements) are given when the queue is created. uxQueueLenght gives the number of elements this queue will be able to handle at any given time. uxItemSize is the size in byte of any element stored in the queue. xQueueCreate returns NULL if the queue was not created due to lack

# xQueueHandle xQueueCreate( unsigned portBASE\_TYPE uxQueueLength, unsigned portBASE\_TYPE uxItemSize);

**Common functions Used:**

# Header File: Arduino\_FreeRTOS.h

* XTaskCreate()
  + This function is used to create multiple tasks with priority. The various arguments taken by this function is explained in the above section.
* VTaskDelay()
  + This function is used to set delay for the multiple tasks created.
  + Eg: VtaskDelay(1000/PortTICK\_PERIOD\_MS);
* vTaskDelete( NULL )
  + This function is used to delete the task that is created.
* xQueueCreate( queueSize, size of element)
  + It creates a queue of the specified size and type.
* xQueueSend(queue, &i, portMAX\_DELAY)
  + This functions inserts the given value into the queue that is created.
* xQueueReceive(queue, &element, portMAX\_DELAY)
  + This functions removes the given value into the queue that is created.

# Program 1: Demonstration of ITC using Global Variables

#include <Arduino\_FreeRTOS.h> void Task1();

void Task2(); int g=0;

void setup()

{

// Now set up two tasks to run independently.

xTaskCreate( Task1, (const portCHAR \*)"task 1" , 128, NULL , 1 , NULL); xTaskCreate( Task2, (const portCHAR \*) "task 2", , NULL , 2 , NULL);

}

void loop()

{

// Empty. Things are done in Tasks.

}

void Task1()

{

pinMode(3,INPUT); for (;;)

{

int value=digitalRead(3); if(value==HIGH)

{

g=1;

vTaskDelay( 2000 / portTICK\_PERIOD\_MS );

}

else

{

g=0;

}

vTaskDelay(1);

}

}

void Task2() // This is a task.

{

pinMode(8, OUTPUT); for (;;)

{

if(g==1)

{

digitalWrite(8,HIGH);

}

else

{

digitalWrite(8,LOW);

}

vTaskDelay(1);

}

}

**Circuit Connection**

|  |  |  |
| --- | --- | --- |
| **Touch Sensor** | **Arduino** | **LED** |
| + | 5 v |  |
| G | Gnd | -ve leg |
| D0 | 3 |  |
|  | 8 | +ve Leg + resistor |

# Program 2: Demonstration of ITC using Message Queues

#include <Arduino\_FreeRTOS.h> QueueHandle\_t queue;

int queueSize = 10;

void setup() { Serial.begin(112500);

queue = xQueueCreate( queueSize, sizeof( int ) ); if(queue == NULL){

Serial.println("Error creating the queue");

}

xTaskCreate(producerTask,(const portCHAR \*)"producer",128, NULL, 1, NULL); xTaskCreate(consumerTask,(const portCHAR \*)"consumer",128, NULL, 1, NULL);

void loop() { delay(1000);

}

void producerTask( void \* parameter )

{

for( int i = 0;i<queueSize;i++ )

{

Serial.print("Element into Queue:"); Serial.println(i);

xQueueSend(queue, &i, portMAX\_DELAY);

}

vTaskDelete( NULL );

}

void consumerTask( void \* parameter)

{

int element;

for( int i = 0;i<queueSize;i++ )

{

xQueueReceive(queue, &element, portMAX\_DELAY); Serial.print("From Queue:"); Serial.println(element);

}

vTaskDelete( NULL );

}

**Exercise:**

* Demonstrate the Inter task communication using the concept of global variable.
  + Use touch sensor.
  + Task 1: Read the value of **Touch sensor**.
  + Task 2: Get the value of the Touch sensor. If it is High **LED** must glow.
* Demonstrate the Inter Task Communication between producer and a consumer via message queues.

# Task 1: Producer

* + **Task 2: Consumer**

|  |  |
| --- | --- |
| **Ex No:10** | **RTOS – TASK SYNCHRONIZATION** |
| **Date:** |

**Aim:**

To perform task synchronization using FREE RTOS.

**Components Required:**

**Hardware:** ARDUINO UNO, Push Button, Photo Resistor, Bread board and Jumper cables.

**Software:** Arduino IDE.

**Binary semaphores:**

Binary semaphores are the simplest effective way to synchronize tasks, an other even more simple, but not as effective, consists in polling an input or a resource. A binary semaphore can be seen as a queue which contains only one element.

**Creation of a semaphore:**

**void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );**

xSemaphore: semaphore to be created.

**Taking a semaphore:**

This operation is equivalent to a P() operation, or if compared to queues, to a Receive() operation. A task taking the semaphore must wait it to be available and is blocked until it is or until a delay is elapsed (if applicable).

**portBASE\_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xTicksToWait**

**);**

xSsemaphore is the semaphore to take. xTicksToWait is the time, in clock ticks, for the task to wait before it gives up with taking the semaphore. If xTicksToWait equals MAX\_DELAY and INCLUDE\_vTaskSuspend is 1, then the task won't stop waiting. If the take operation succeed in time, the function returns pdPASS. If not, pdFALSE is returned.

**Giving a semaphore:**

Giving a semaphore can be compared to a V() operation or to writing on a queue.

**portBASE\_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );**

xSemaphore is the semaphore to be given. The function returns pdPASS if the give operation was successful, or pdFAIL if the semaphore was already available, or if the task did not hold it.

**Mutexes:**

Mutexes are designed to prevent mutual exclusion or deadlocking. A mutex is used similarly to a binary semaphore, except the task which take the semaphore must give it back. This can be though with a token associated with the resource to access to. A task holds the token, works with the resource then gives back the token; in the meanwhile, no other token can be given to the mutex.

**Common functions Used:**

**Header File: Arduino\_FreeRTOS.h, semphr.h**

* XTaskCreate()
  + This function is used to create multiple tasks with priority. The various arguments taken by this function is explained in the above section.
* VTaskDelay()
  + This function is used to set delay for the multiple tasks created.
  + Eg: VtaskDelay(1000/PortTICK\_PERIOD\_MS);
* vTaskDelete( NULL )
  + This function is used to delete the task that is created.
* xSemaphoreGive( Semaphore\_variable )
  + This function is used to release the acquired semaphore.
* xSemaphoreTake(Semaphore\_variable, ( TickType\_t ) 5 )
  + This function is used to acquire the semaphore.
* xSemaphoreCreateMutex()
  + This function is used to create the semaphore.

**Program 2: Demonstration of ITC using Message Queues**

#include <Arduino\_FreeRTOS.h> #include <semphr.h>

SemaphoreHandle\_t xSerialSemaphore;

void TaskDigitalRead( void \*pvParameters ); void TaskAnalogRead( void \*pvParameters );

void setup()

{

Serial.begin(9600);

if ( xSerialSemaphore == NULL ) .

{

xSerialSemaphore = xSemaphoreCreateMutex(); if ( ( xSerialSemaphore ) != NULL )

xSemaphoreGive( ( xSerialSemaphore ) );

}

xTaskCreate(TaskDigitalRead,(const portCHAR \*)"DigitalRead",128,NULL,2,NULL ); xTaskCreate(TaskAnalogRead,(const portCHAR \*)"AnalogRead",128,NULL,1,NULL );

}

void loop()

{

// Empty. Things are done in Tasks.

}

void TaskDigitalRead( void \*pvParameters attribute ((unused)) )

{

uint8\_t pushButton = 2; pinMode(pushButton, INPUT);

for (;;)

{

int buttonState = digitalRead(pushButton);

if ( xSemaphoreTake( xSerialSemaphore, ( TickType\_t ) 5 ) == pdTRUE )

{

Serial.println(buttonState); xSemaphoreGive( xSerialSemaphore );

}

vTaskDelay(1);

}

}

void TaskAnalogRead( void \*pvParameters attribute ((unused)) ) // This is a Task.

{

for (;;)

{

int sensorValue = analogRead(A0);

if ( xSemaphoreTake( xSerialSemaphore, ( TickType\_t ) 5 ) == pdTRUE )

{

Serial.println(sensorValue); xSemaphoreGive( xSerialSemaphore );

}

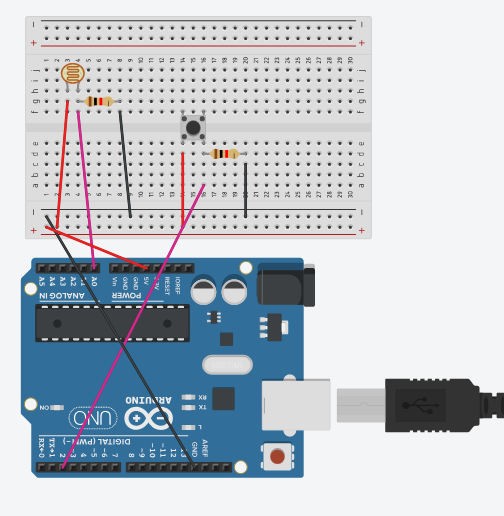
**chematic:**

**S**

vTaskDelay(1);

}

}



**Circuit Connection:**

|  |  |  |
| --- | --- | --- |
| **Push Button** | **Arduino Pin** | **Photo Resistor** |
| (Leg 1 + Resistor) 2nd Leg | Gnd | 1st Leg + Resistor (2nd End) |
| (Leg 1 + Resistor) 1st Leg | 2 |  |
| Leg 2 | 5 V | 2nd Leg |
|  | A0 | 1st Leg + Resistor (1st End) |

**Exercise:**

Write an arduino sketch to perform the following:

1. Connect a push button and a photo resistor.
2. Create 2 independent tasks such that,
   1. Task 1 – Gets the data from the push button and displays it on the serial monitor.
   2. Task 2 – Gets the data from the photo resistor and displays it on the serial monitor.
3. As, both the tasks use the same resource, perform task synchronization between the two tasks using semaphore.