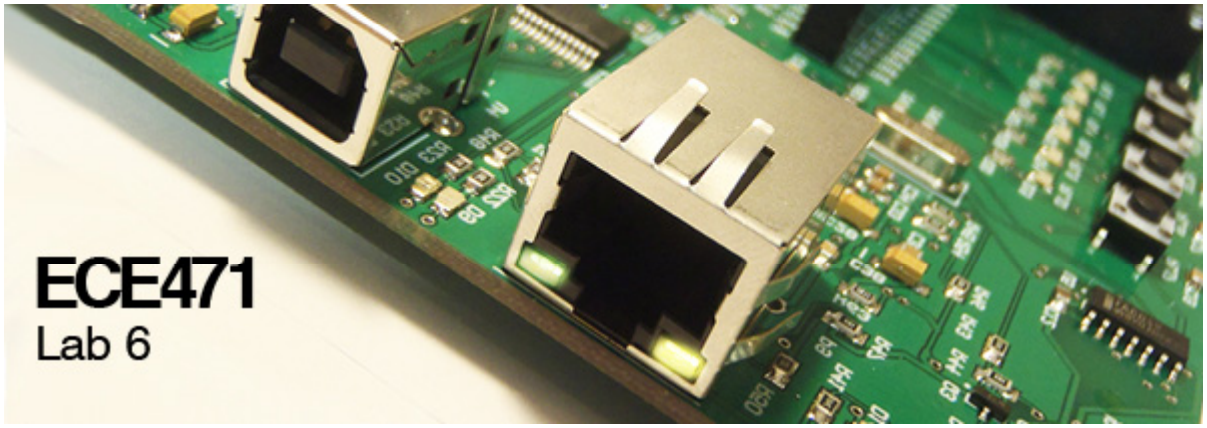


NDSU Embedded Systems II

Free RTOS

Multi-Tasking & Queue Usage

Objective: To obtain familiarity with the following:

- Free RTOS Tasks
- Free RTOS Queues

By
Nathan Zimmerman

Requirements for lab 6:

Introduction: Free RTOS is a popular industry grade real-time operating system designed for embedded level applications. Free RTOS has numerous features which are common to an RTOS such as tasks, adjustable priority levels, queues, semaphores. This RTOS also has preemptive and cooperative based elements which make it a simple and flexible OS. For starting out in RTOS, we will be focusing on task creation and queue usage.

Requirements for Lab 6:

Our goal is to replicate lab 3 (the clock/stopwatch lab) in FreeRTOS:

1. Create a Button task that will software poll a button GPIO every 10ms. Use **vTaskDelay** as opposed to forloop delay.
2. Create a Time Task that will keep track of 10ms, secs, mins, & hours. Use **vTaskDelayUntil.**
3. Create a LCD task to update the LCD display with time every 50ms. Use **vTaskDelay**
4. Use a Queue to send time information from the Time task to the LCD task. Use **xQueueSend**
5. Use another Queue to send information from the Buttons task to the LCD task. Use **xQueueSend**

Result should look similar to Lab6.axf .

Extra Credit:

1. Create & run an additional task which implements Lab 1 in a separate LED task.

Useful Resources:

Demo Project

http://interactive.freertos.org/attachments/token/uqn9wnetzurd6u4/?name=FreeRTOS-simple-demo-for-LPCXpresso-LPC1768.zip

Setup FreeRTOS application note for LPC1769:

http://interactive.freertos.org/attachments/token/rpjspilbw7qbvca/?name=FR-191-AN-RB-003i2-_A_simple_FreeRTOS_demo_for_the_LPCXpresso_.pdf

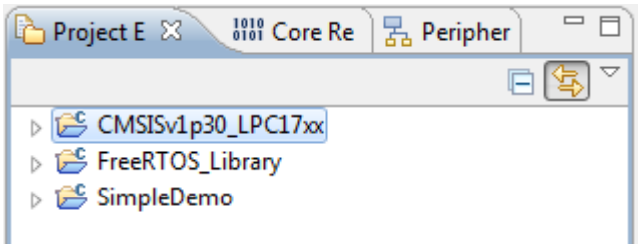
LPC1769 Free RTOS youtube video tutorial:

https://www.youtube.com/watch?v=WzKsMfn2sBE

Getting Started W/ Free RTOS

1. Download demo project
   http://interactive.freertos.org/attachments/token/uqn9wnetzurd6u4/?name=FreeRTOS-simple-demo-for-LPCXpresso-LPC1768.zip

2. Create a new work space for your RTOS projects. The reason for this is because we will be using a separate version of CMSIS (CMSISv1p30) which comes with the demo project for the LPCXpresso. Regrettably, it is rather painful to create and fully setup a FreeRTOS project in LPCXpresso. As such, we will simply rely on the demo program provided. Use the guide provided in the "Useful Resources" page for setup instructions regarding importing the demo project into your new work space. Your workspace should look like the following when you are done.



3. Modify the "FreeRTOSconfig.h" and increase the heap size. To do this, rewrite the define heap size define to `#define configTOTAL_HEAP_SIZE( ( size_t ) ( 10 * 1024 ) )`. The heap size of the demo program is easy to overflow and can cause rather painful bugs sufficient heap size is not allocated.

4. Read up on the documentation provided in class and implement the lab requirements. Due to the massive amounts of documentation online already for FreeRTOS, this lab document will provide a very limited amount of information regarding FreeRTOS.

Free RTOS Tasking

Concept of a task & delay: Tasks in FreeRTOS are a critical fundamental block of the OS and are simply C functions that contain infinite loops. All repetitive requirements of an OS are generally separated and split up into tasks that continuously run. These tasks will continuously execute provided they are not in a blocked state. The concept of a blocked state is one element that can make OS-es very efficient. For example, say we wish to blink an LED every 60ms in the following sudo code.

```
Int main void
{
LPC_GPIO1 -> FIODIR |= LED0;  // Set LED as output
while(1) // Continuously loop
   {
        LPC_GPIO1 -> FIOPIN ^= LED0; //Toggle LED
        for (i=0;i<900000;i++) // delay 60ms.
   }
}
```

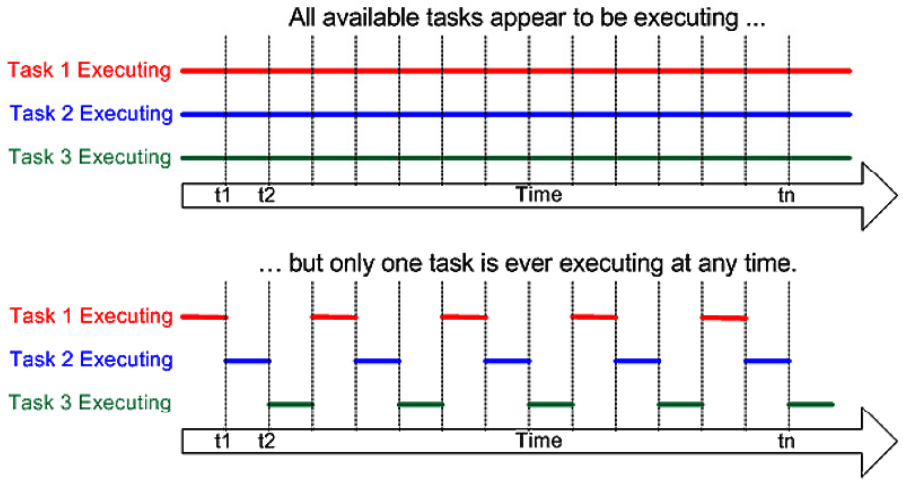
While the above code works, it is incredibly inefficient because the vast majority of our 120MHz CLK is spent doing nothing. Any type of delay simply expending clock cycles is incredibly wasteful and could be used to perform other operations. Hence the creation of multiple tasks & blocked states. The following is an example of how we would implement this same requirement in FreeRTOS with a task

```
static void LED_TASK( void *pvParameters )
{
      LPC_GPIO1 -> FIODIR |= LED0;  // Set LED as output
      while(1) // Continuously loop
      {
            LPC_GPIO1 -> FIOPIN ^= LED0; //Toggle LED
            vTaskDelay(60) ;
      } // TASK END
}
```

vTaskDelay(60) puts the task into a blocked state for 60 clock ticks. Provided the clock ticks occur every 1ms, this would be the equivalent to a 60 ms delay. However, as soon as this task is put in a blocked state, other tasks that are not currently blocked have the opportunity to run. This task could have been equally implemented with *vTaskDelayUntil* which allows for more precise timing than *vTaskDelay.* More information regarding implementing these functions should be present in the FreeRTOS manual.

Free RTOS Tasking

Muli-Tasking: When using FreeRTOS, source code simply becomes a compilation of multiple tasks. It is important to note though that multi-tasking is different than multiple threads since we only have a single core. Consequently, all tasks must execute in a sequential order that is dependent on block states and priority levels.



Priority level dictates which task will operate next when a system tick occurs or when a task that is currently running enters a blocked state. Many RTOS API can cause a task to enter a blocked state such as vTaskDelay, and vTaskSuspend(). Other API can optionally block a task depending on user settings. For example, if it is desired for a program to be blocked until data is received from a queue, one can use the following: xQueueReceive(xQueue, &ulReceivedValue, portMAX_DELAY); versus reading a queue without blocking: xQueueReceive(xQueue, &ulReceivedValue, 0); . Reading up on the RTOS manual should help significantly behind the RTOS API and how to implement them. On the following page is a simple FreeRTOS demo program I wrote to demonstrate some functionality of this API.

Sample FreeRTOS Program: Blinks LED, SW0 is enables/disables sequence.

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

static void LED_Task( void *pvParameters ); // prototype for led task
static void BUTTON_Task( void *pvParameters ); // prototype for button task

static xQueueHandle xQueue = NULL; // prototype for queue

#define LED_TASK_PRIORITY  ( tskIDLE_PRIORITY + 1 ) // assign priority to led task
#define BUTTON_TASK_PRIORITY  ( tskIDLE_PRIORITY + 2 ) // assign priority to button task
#define mainQUEUE_LENGTH 1

int main(void)
{
	xQueue  = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) ); // create queue
	xTaskCreate( LED_Task, ( signed char * ) "LED", configMINIMAL_STACK_SIZE, NULL, LED_TASK_PRIORITY, NULL );
	xTaskCreate( BUTTON_Task, ( signed char * ) "BUTTON", configMINIMAL_STACK_SIZE, NULL, LED_TASK_PRIORITY, NULL );
	vTaskStartScheduler(); // Starts task system
	while(1); // Should never reach this point
}

static void BUTTON_Task( void *pvParameters )
{
	uint32_t data =5; // random data
	portTickType xNextWakeTime; //necessary for vtaskdelayuntil
	xNextWakeTime = xTaskGetTickCount(); //necessary for vtaskdelayuntil
	while(1){
		if(!(((LPC_GPIO0->FIOPIN)&(1<<2))>>2)) // IF button 2 pressed
		{
			xQueueSend( xQueue, &data, 0 ); // Send Data to LED task
			vTaskDelay(250); // Debounce delay ~250ms
		}
	vTaskDelayUntil( &xNextWakeTime,10); //Delay for 10ms
	}
}

static void LED_Task( void *pvParameters )
{
	uint32_t data =0;
	uint8_t LED_Sequence_ON = 1;
	LPC_GPIO1 ->FIODIR |= (1<<19);
	while(1){
			if(xQueueReceive( xQueue, &data, 0)) // If new data in queue, do something
				LED_Sequence_ON=!LED_Sequence_ON; // Toggle Sequence on/off

			if(LED_Sequence_ON)
			{
				LPC_GPIO1->FIOPIN ^= ( 1 << 19);
			}
			else
				LPC_GPIO1->FIOSET = (1 << 19); // LED off;
			vTaskDelay(250); // Delay for ~50ms
	}
}
```

Lab Report Guidelines:

1. Upload your code files to your assigned GIT repo. You must first create an account on github and email me your account name in order to gain access to your repo.

2. If you are having trouble getting GIT to work initially, email me your code at nathan.zimmerman@my.ndsu.edu . The code should be attached in the form of .c and .h files or in a complete zip archive . Your email title should be : Lab_Number : Myname. For example: Lab_1 : Nathan Zimmerman. This method of turning in assignments is cumbersome and will not be available long term so attempt to get GIT working ASAP.

   You are allowed to work in groups but you are all expected to turn in your own individual code. You should write the code yourself and it should not be a duplication of someone elses code.

Grading:

100% **: Code runs and by majority meets requirements**

Xx% **: An attempt was made to implement as many of the requirements as possible for partial credit.**

0% **: Code was not turned in or significantly falls short of meeting requirements.**