

# FreeRTOS

In this lab you will learn how to use a real time operating system on your AVR microprocessors. To implement this we will be using a free open source RTOS called FreeRTOS. FreeRTOS(<http://www.freertos.org/RTOS.html>) is an open-source RTOS library which can be easily ported onto the AVR microprocessors.

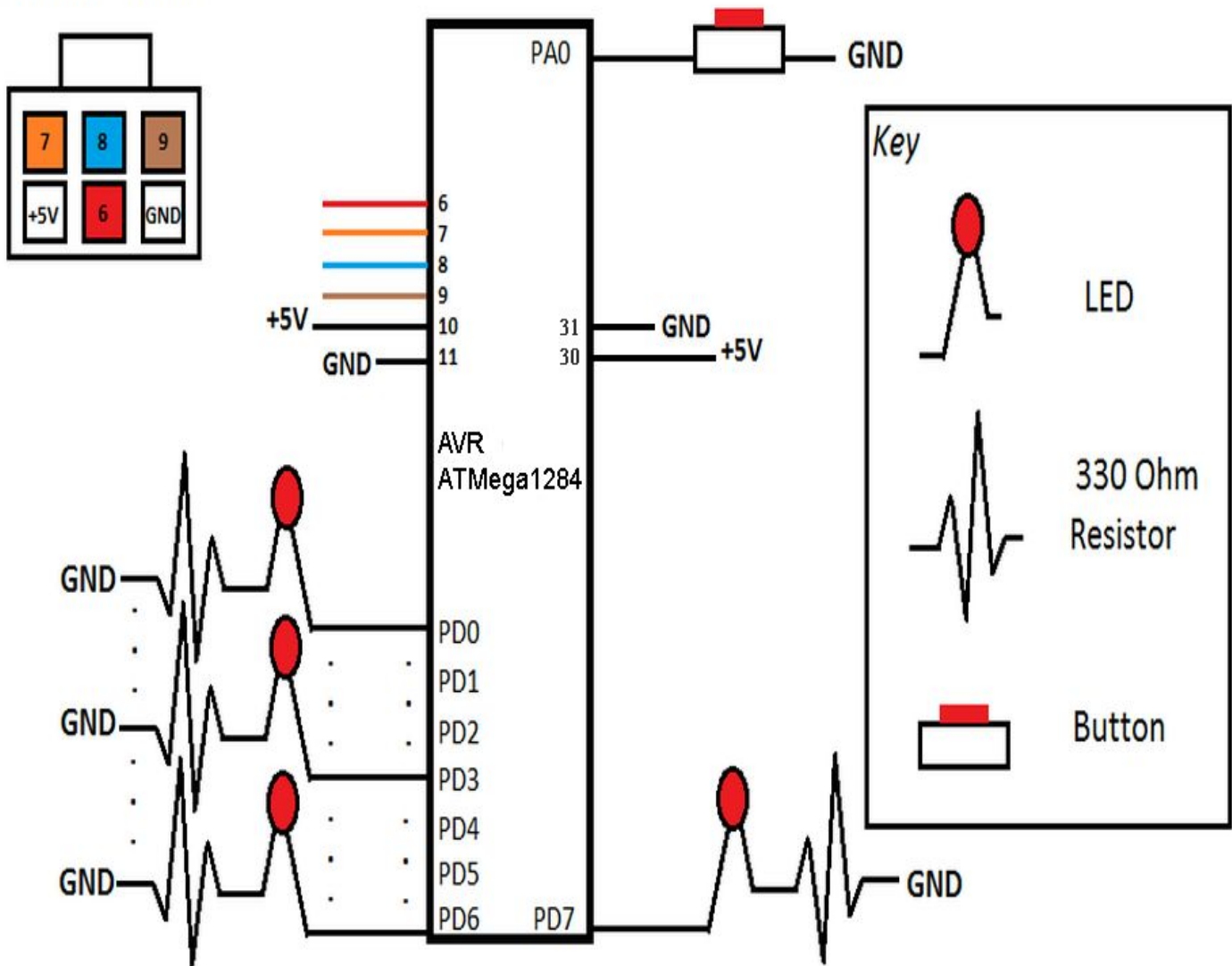
## What is a RTOS?

A real-time operating system is an operating system that runs tasks in real-time, which allows programmers more control over process priorities.

Have the following circuit below wired up and ready to go. This should be the same hardware setup you had in the previous lab. Be aware that there should be LEDs wired to pins PD0-PD7.(8 LEDs in total)

The AVR ATmega1284 datasheet: <http://www.atmel.com/images/doc8059.pdf>

Quick Start Guide: <http://www.freertos.org/FreeRTOS-quick-start-guide.html>




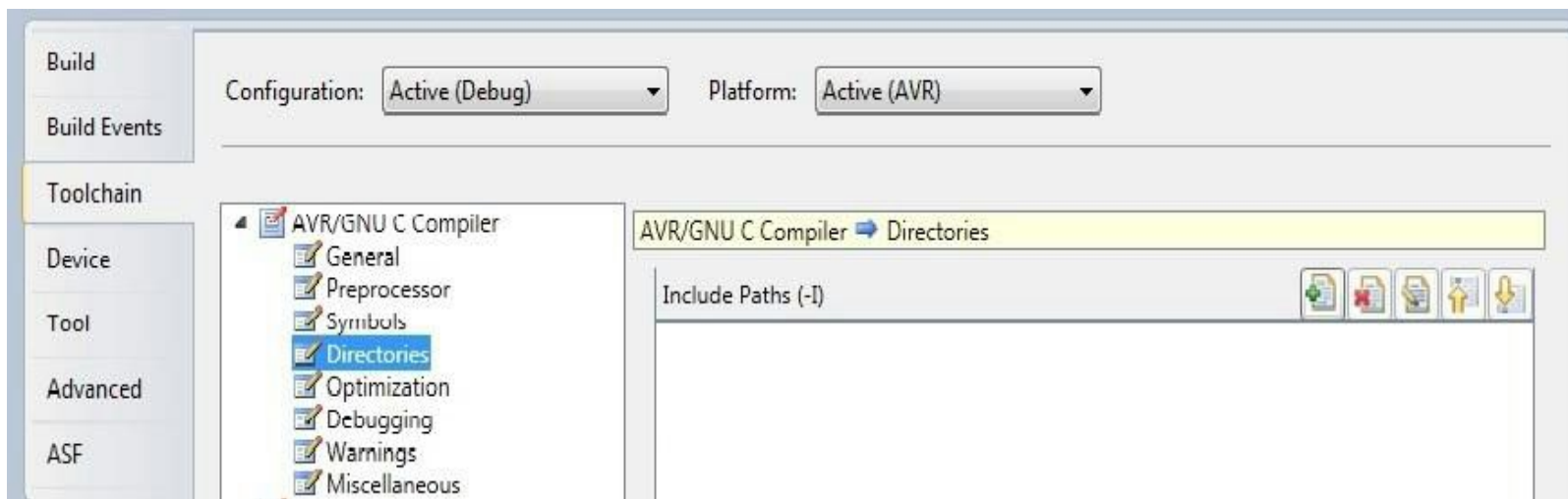
## Part I:


In this part, we will configure FreeRTOS to work with your AVR microcontroller. In order to do this you must download the [zip file](#) containing all the necessary source files. The provided zip file contains a modified subset of the files that can be downloaded from the FreeRTOS website.

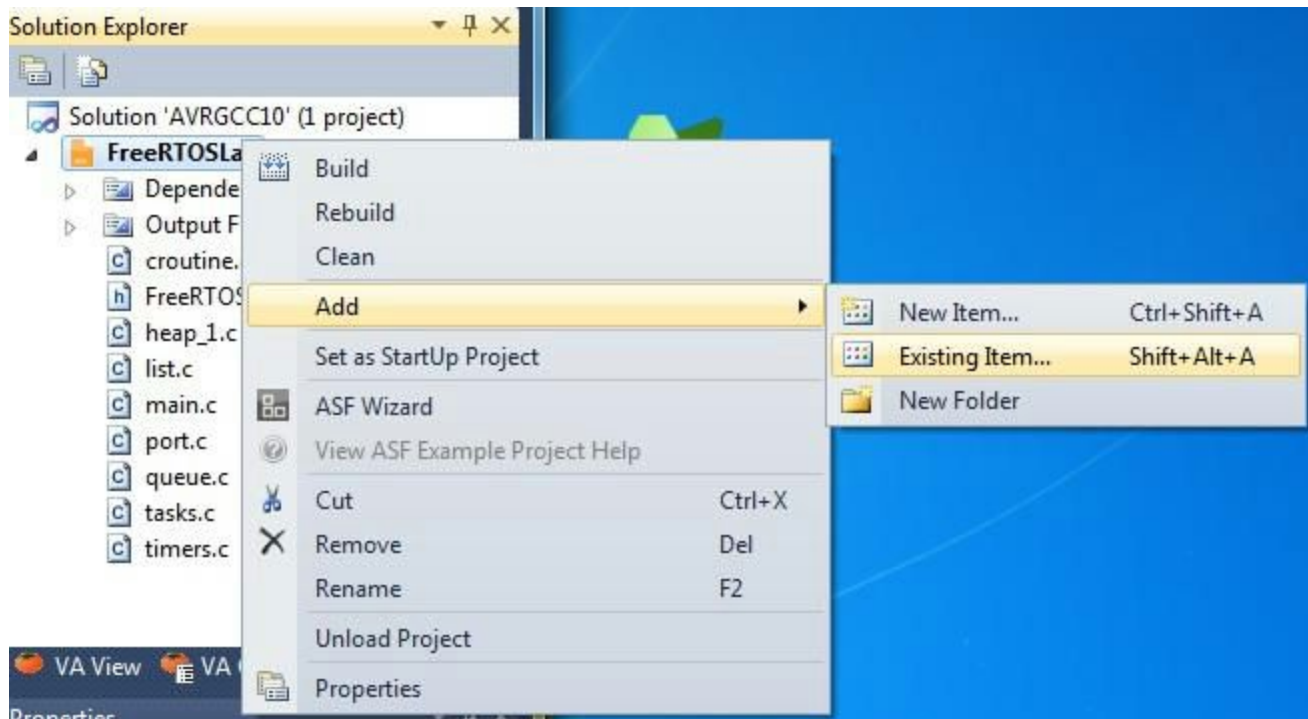
### Lab Procedures:

1. Create a new folder named FreeRTOS on your desktop. Extract the given FreeRTOS\_Lab.zip into your FreeRTOS folder.
2. Build the above circuit design on your breadboard.

3. Double check your circuit design with your partner to assure it has been wired correctly.
4. Open AVR Studio and create a new C Project. If you do not remember how to do this please refer to the Lab Reference Guide.
5. Go to Project->Properties and select Toolchains->Directories. Press the Add Item button  and include the following directories using absolute paths by unchecking "relative paths".
  - FreeRTOS/Source
  - FreeRTOS/Source/include
  - FreeRTOS/Source/portable/GCC/ATMega323



6. On your Solution Explorer window, Right-Click your main Project File .
  - Select Add > Existing Item and select the following files in your FreeRTOS directory.Note that you can add multiple items at the same time by selecting them all.
  - croutine.c
  - heap\_1.c
  - list.c
  - port.c
  - queue.c
  - tasks.c
  - timers.c
  - main.c



7. Examine main.c. main.c is a file we created that contains a state machine that lights LEDs in sequence. The other files come with FreeRTOS.

8. Now build the project and upload it to your AVR. If all these steps are done correctly you should see your LEDs cycling through. (You can delete the generated .c file from creating a new AVR project.)

**Note:** If you are getting the compiler error "multiple definitions of main()" You may need to remove the original .c file created when you generated your project.

**Note:** The registers to set up the timer differ between the ATmega1284 and ATmega32. The code has been designed to handle this, but if you are having issues you may need to change settings in the port.c file. Specifically the settings for the TIMSK(1) and ucLowByte.

## Part II:

Now that Part I has been completed let us break down how FreeRTOS works. There are three main functions that you will be using to implement this part. Read through the FreeRTOS Quick-Start if more information is needed.

```
// This function creates a task that will be added to the list
// of tasks that are ready to run.
xTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters,
            uxPriority, pxCreatedTask )
```

- **pvTaskCode**: Pointer to the task entry Function. In main.c of Part 1, this parameter was set to **LedSecTask**.
- **pcName**: A label or name used for the task.(Used for debugging purposes.)
- **usStackDepth**: The size of the task stack.(Use the same parameter used in main.c from Part I.)
- **pvParameters**: Pointer that will be used as the parameter for the task being created.(Set this to NULL.)
- **uxPriority**: The priority at which the task should run. (Higher the number, higher priority.)
- **pxCreatedTask**: Used to pass back a handle by which the created task can be referenced.(Set this to NULL.)

```
// This function delays a task until a specified time has passed
vTaskDelay( portTickType xTicksToDelay );
```

- **portTickType xTicksToDelay**: The amount of time, in tick periods, that the calling task should block.(This takes in time as milliseconds. Recall 1000ms = 1s)

```
// This function starts the real time kernel tick processing.
// This function is the equivalent of TimerOn() in our task
// scheduler.
void vTaskStartScheduler( void );
```

### Lab Procedures:

1. You will be using the same circuit design and template as in main.c of Part I.
2. Draw out the finite state machines for the given exercise problems below. Both you and your partner should draw out the finite state machines and compare results to assure correctness.
3. Implement the state machines as separate tasks. Be sure to always have an infinite loop for each one of your tasks (while(1) or for(;;) will work). Refer to the following functions in main.c from Part 1 to see how a complete task is implemented.
  - **void LEDS\_init()**
  - **void LEDS\_Tick()**
  - **void LedSecTask()**
  - **void StartSecPulse(unsigned portBASE\_TYPE Priority)**
4. Have your TA check you off once you have finished each exercise. Please write your name on the board and specify which exercise you want checked off. Also, be prepared to show your source code.

### Exercises:

1. Blink three LEDs connected to PD0,PD2, and PD4 at a rate of 1000ms. Use only one task to complete this functionality.

**Video Demonstration:** <http://youtu.be/2g9epHEkY3A>

2. From the previous lab we had you implement three LEDs that blinked at different rates. PD0 at a rate of 500ms, PD2 at a rate of 1000ms and PD4 at a rate of 2500ms. Implement the same functionality using the FreeRTOS library. Use multiple tasks to complete this functionality and make them visible in your code.

**Video Demonstration:** <http://youtu.be/296uqczZjCw>

3. Now we want to implement a state machine design where the LEDs will cycle through each LED one after another. Once it reaches the last LED it will bounce and go in the opposite direction. (Try simplifying your designs by using the shift operator.)

**Video Demonstration:** <http://youtu.be/uveVMEDDoQE>

4. (Challenge) Expand upon exercise 3 by adding a button which will reverse the direction of the LED cycle whenever it is pressed. (Hint: Review priorities to implement this.)

**Video Demonstration:** <http://youtu.be/tYKldfoPBXs>

5. (For fun - No credit) Implement two tasks that use the same LED and assign different priorities to them. Then play around with some implementations where you can see how priority matters. (You don't have to demo this.)

**Each person must submit their .c source files according to instructions in the Lab submission guidelines.**