

# LAB: Periodic Scheduler

The objective of this assignment is:

- Set up your development environment
- Learn how to run unit-tests
- Trial how to input your code to the Periodic Scheduler

For CmpE243, we will not be focusing on typical RTOS tasks like CmpE244. The reason is that we wish to use an approach that is typically seen in the Automotive industry, which is to design the logic of your autonomous RC car based on software instructions that occur periodically and consistently.

---

## Part 0: Build Environment

Set up your development environment for this portion of the lab. Follow through and read all of the README files carefully that are [linked here](#). Make sure you are able to run the unit tests, and also compile a hex file that you can load onto your board.

You can watch the following video to get started:

- [Youtube: Compile project](#)
- [Youtube: Scons build system](#)

---

## Part 1: Blink LEDs

For this portion, edit the code such that it will start to blink four LEDs driven by the periodic scheduler. In particular, read the documentation of the `main.c` file, and enable the code for the periodic scheduler.

Study the overall structure of `main.c`, and then switch a `#if (1)` to `#if (0)` such that it will disable two blinky tasks, and instead run the periodic scheduler. The name "periodic scheduler" may sound fancier than what it actually is, but this is just a trivial piece of code that invokes function at `periodic_callbacks.c` file.

```
// main.c
static void create_blinky_tasks(void) {
    /**
```

```

    * Use '#if (1)' if you wish to observe how two tasks can blink LEDs
    * Use '#if (0)' if you wish to use the 'periodic_scheduler.h' that will spawn 4 periodic tasks, one
    */
#if (0)
    // ...
#else
    periodic_scheduler__initialize();
    UNUSED(blink_task);
#endif
}

// periodic_scheduler.c
void periodic_scheduler__initialize(void) {
    /**
     * ...
     */
    static StackType_t hz1_stack[4096 / sizeof(StackType_t)];
    static StackType_t hz10_stack[4096 / sizeof(StackType_t)];
    static StackType_t hz100_stack[4096 / sizeof(StackType_t)];
    /**
     * ...
     */
}

```

There are a few things to note for future reference:

- The stack size is chosen with a same value, and depending on the complexity of the functions you invoke at the `periodic_callbacks.c` file, you may have to increase this memory size. Also note that there are five tasks total that run the periodic callbacks, so if you input 2K, then you will end up using 10K for the memory footprint. Recommended size is 2-4K.
- The logic at `periodic_callbacks.c` the file should be function calls into your other code modules. This way, unit tests of this file will remain simple. You do not want to input branch statements here because this would make your code less modular, and difficult to unit-test.

## Part 2: Switch and LED code module

Insert additional code to one of the periodic callbacks, and then observe its operation. In the example below, we are going to demonstrate the right way to build a module that reads a switch and lights up an LED.

DO NOT do the following because what you have done is that cluttered all the things that need to occur periodically. If we go down this path, you will end up creating a giant `periodic_callbacks.c` file that will be difficult to test, and your code will not be modular or broken down into these pieces. Unit-testing code will also be difficult because now you have to not only test the switch and LED logic but also test more unrelated

subsequent code.

```
// periodic_callbacks.c -- BAD example
static gpio_s my_led;
static gpio_s my_switch;
void periodic_callbacks__initialize(void) {
    my_led = gpio__construct_as_output(GPIO__PORT_2, 0);
    my_switch = gpio__construct_as_input(GPIO__PORT_2, 1);
}
void periodic_callbacks__1Hz(uint32_t callback_count) {
    gpio__toggle(board_io__get_led0());
    if (gpio__get(my_switch)) {
        gpio__set(my_led);
    } else {
        gpio__reset(my_led);
    }
}
```

Instead, follow good code design, and create "modules" for your code. Using this approach, you have refactored your switch and LED logic to a new code module: `switch_led_logic.h`. You can test this code module separately and then testing the `periodic_callbacks.c` a code module is also straightforward since you only have to set up a couple of "expect" function calls.

```
// periodic_callbacks.c -- Good example
#include "switch_led_logic.h"
void periodic_callbacks__initialize(void) {
    switch_led_logic__initialize();
}
void periodic_callbacks__1Hz(uint32_t callback_count) {
    gpio__toggle(board_io__get_led0());
    switch_led_logic__run_once();
}
```

Of course, you are not done yet, and you also have to modify `test_periodic_callbacks.c`

```
#include "Mockboard_io.h"
#include "Mockgpio.h"

// Add mock of your new code module
```

```

#include "Mockswitch_led_logic.h"
#include "periodic_callbacks.h"
// Add expect during the periodic_callbacks__initialize() function
void test__periodic_callbacks__initialize(void) {
    switch_led_logic__initialize_Expect();
    periodic_callbacks__initialize();
}
void test__periodic_callbacks__1Hz(void) {
    gpio_s gpio = {};
    board_io__get_led0_ExpectAndReturn(gpio);
    gpio__toggle_Expect(gpio);
    switch_led_logic__run_once_Expect();
    periodic_callbacks__1Hz(0);}

```

---

## Part 3: Experiment with Task Overrun

Deliberately overrun one of the periodic tasks and observe that your board will reboot. Since this will be sort of a "throw-away" code, you can opt to skip the unit-tests. Here is a sample code that will deliberately reboot the processor because of the missed deadline of the 1Hz function.

```

// periodic_callbacks.c
// Include these files for RTOS task delay function
#include "FreeRTOS.h"
#include "task.h"
void periodic_callbacks__1Hz(uint32_t callback_count) {
    gpio__toggle(board_io__get_led0());
    // On the fifth function call to this function, sleep for 1000ms
    if (callback_count >= 5) {
        vTaskDelay(1000);
    }
}

```

It is **strongly advised** NOT to skip the unit-tests in general. But if you are purely doing a code prototype to try things out, then use the `scons --no-unit-test` command.

---

## What did you learn?

- Work with the periodic callbacks to add your code
- Design small code modules, and set up their expectation in unit-test code
- The first-hand account of what happens when you miss the deadline of a periodic callback

---

Revision #13

Created 4 years ago by [Preet Kang](#)

Updated 10 months ago by [isa\\_team](#)