



Micrium OS Kernel Labs

2018.04.16

Micrium OS is a flexible, highly configurable collection of software components that provides a powerful embedded software framework for developers to build their application on top of. There are few dependencies among the components, so developers are mostly free to select the software that makes sense for their projects. Within any particular component, there are numerous parameters that can be adjusted—at either compile time or run time in many cases—to ensure the most efficient use of resources.

Objectives

After completing this hands-on session, you will be able to:

- Build and flash your Giant Gecko Kit with Micrium OS Kernel projects
- Have a solid understanding of how to create Micrium OS Kernel tasks
- Write interrupt handlers using Micrium OS Kernel's ISR mechanism
- Properly use Semaphores and Event Flags in Tasks and ISRs
- Protect shared resources using Mutexes
- Pass data through Message Queues

Lab setup

The lab setup requires the following elements:

- A Giant Gecko GG11 Starter Kit (SLSTK3701A)

Software requirements

Some software tools are needed to complete the lab:

- Simplicity Studio Version 4 (with the most recent board updates)



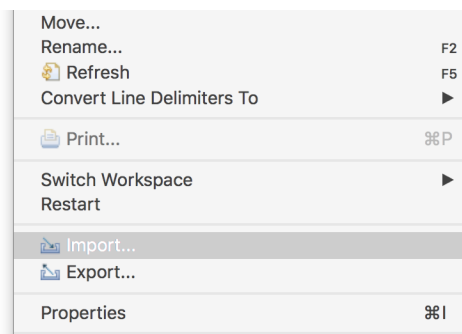
Micrium OS Kernel Lab 1

Overview

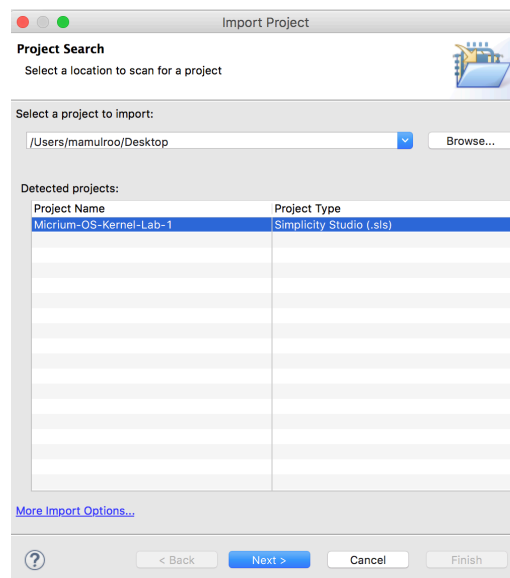
For this lab, you will just import an existing Micrium OS Kernel project and using that project flash your Giant Gecko Starter Kit. The goal of this lab is to ensure you are able to import a project, compile the project, flash your kit with the compiled project and then run the project via the debugger. After importing the project you will need to add some code provided in this lab manual to the Micrium OS Kernel Task to provide the heartbeat LEDs. Once the application is running on your Giant Gecko Starter Kit you will see a message displayed on the LCD and the LEDs on the board blinking.

Loading the Project into Simplicity Studio

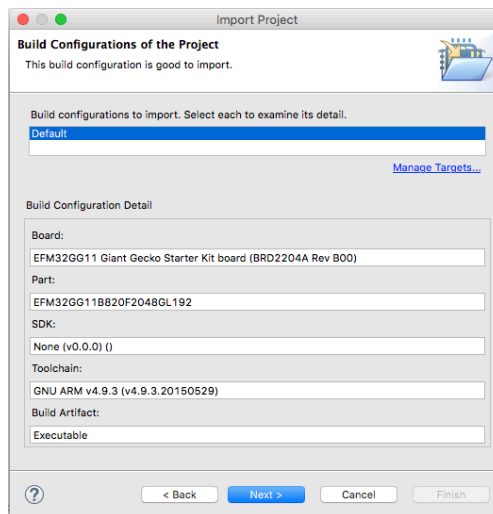
Start by unzipping the zip file that was provided by your instructor and then open up Simplicity Studio.



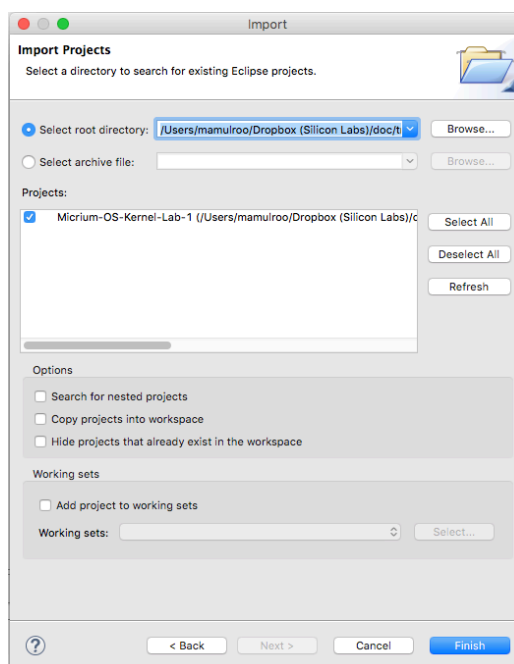
Under the file menu in Simplicity Studio, select **Import**.



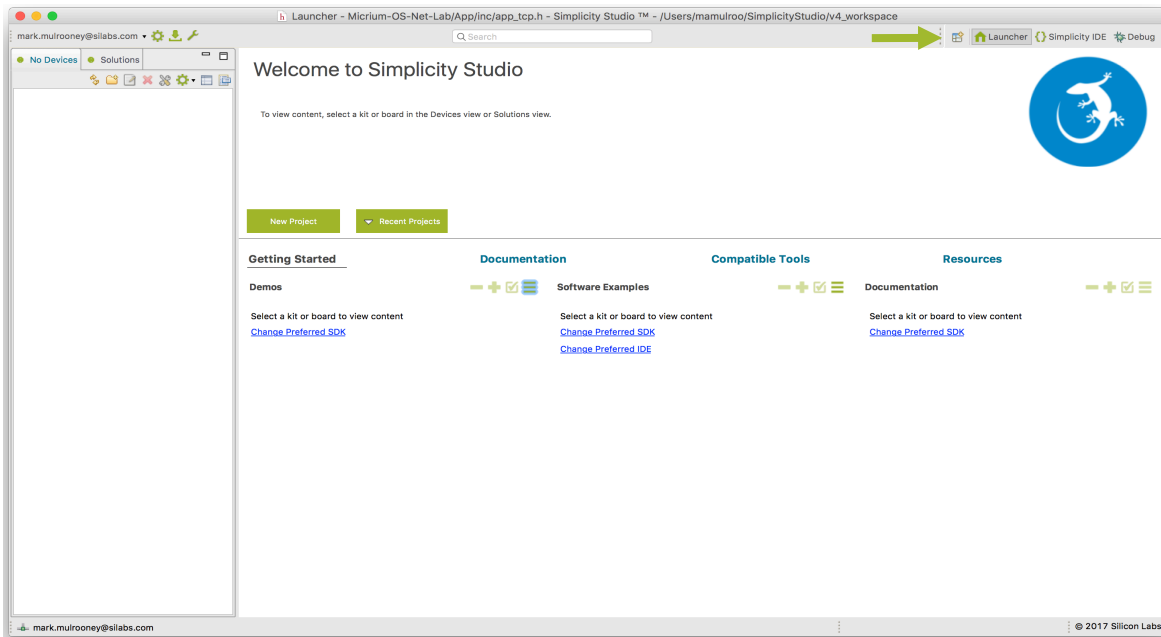
Browse to the location of the folder you unzipped. You should see all of the lab files under **Detected projects**. Select **Micrium-OS-Kernel-Lab-1** and then click **Next**.



Under the build configurations you should be able to leave everything at the default. If you do not have a specific version of a configuration item, attempt to change it to the version you have. If you are unsure or unable to get it to work ask your instructor for help. Once you've set the build configuration click **Next**.

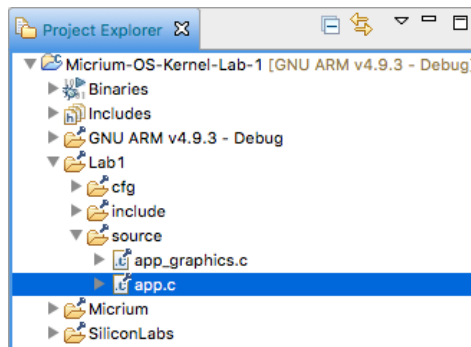


Click on the **Browse** button and select the directory where you previously unzipped the provided zip file. Once you load the directory you should see **Micrium-OS-Kernel-Lab-1** showing up under the **Projects** window. If you do not see any projects under the **Projects** window, verify that you have unzipped the provided zip file and are correctly selecting that directory. Once you've verified **Micrium-OS-Kernel-Lab-1** is displayed under **Projects** and the checkbox is checked, click the **Finish** button.



After clicking **Finish** you will be returned to this screen. Click the **Simplicity IDE** button in the top right corner. If the **Simplicity IDE** button is **NOT** there, click  next to **Launcher** to add **Simplicity IDE**.

Adding the Heartbeat Code



Open up the file **app.c**. This is where you will be adding code for the heartbeat LED. Locate the comment in the code for **Micrium OS Kernel Lab 1: Step 1**.

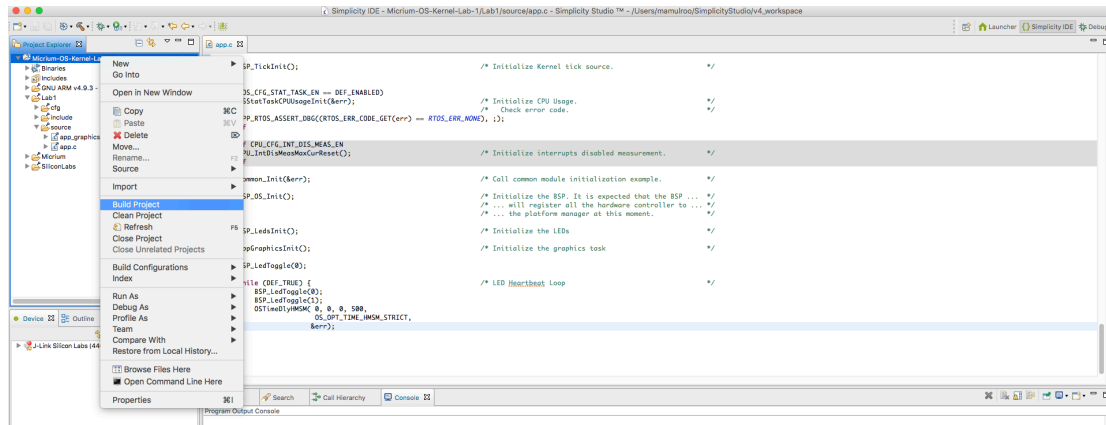
Step 1:

Add the following code inside the while loop. This will toggle the LEDs at an interval of 500ms. Feel free to change the delay to any value you wish. Be sure to check out the documentation for `OSTimeDlyHMSM()` to see what the valid parameters are.

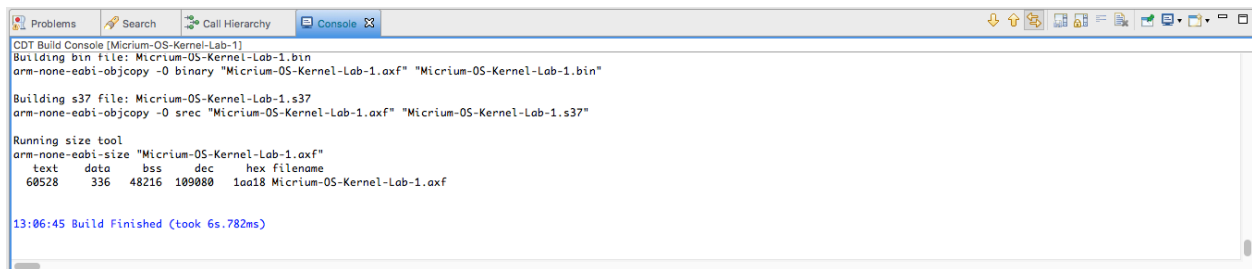
```
BSP_LedToggle(0);
BSP_LedToggle(1);
OSTimeDlyHMSM(0, 0, 0, 500,
               OS_OPT_TIME_HMSM_STRICT,
               &err);
```

Building the Application Code

Once you have added the necessary code to **app.c**, you can build your project.



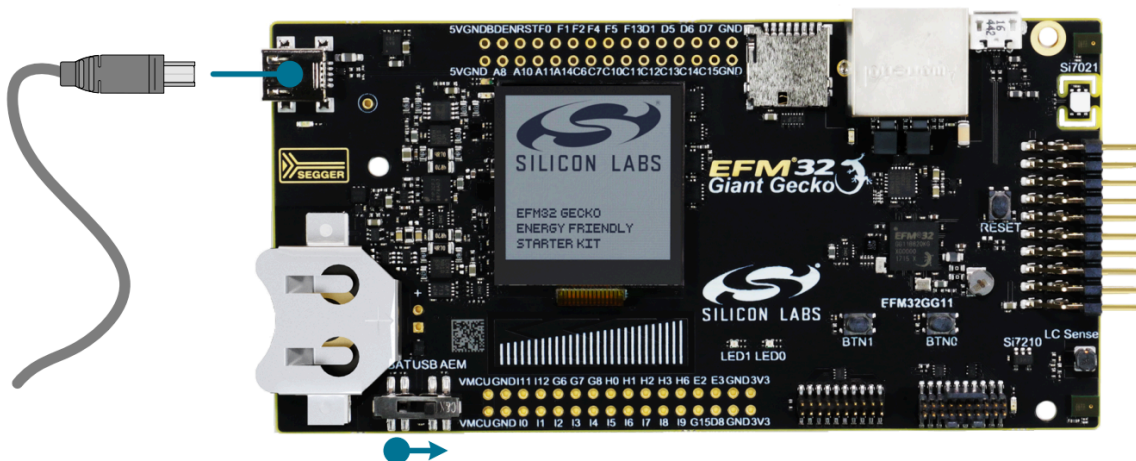
Right click on **Micrium-OS-Kernel-Lab-1** under the project explorer and click the **Build Project** option. You can also access **Build Project** under the Project menu.



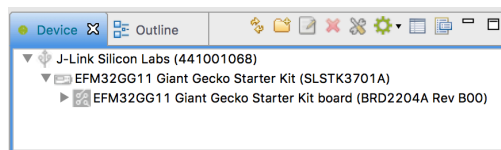
After building the project, you should see a similar output on the Console window. If it reports errors during the build, check the modifications you've made in the code to ensure there are no errors. If you're still unable to get it to build, ask your instructor for help.

Flashing your Giant Gecko GG11 (SLSTK3701A) from Simplicity Studio

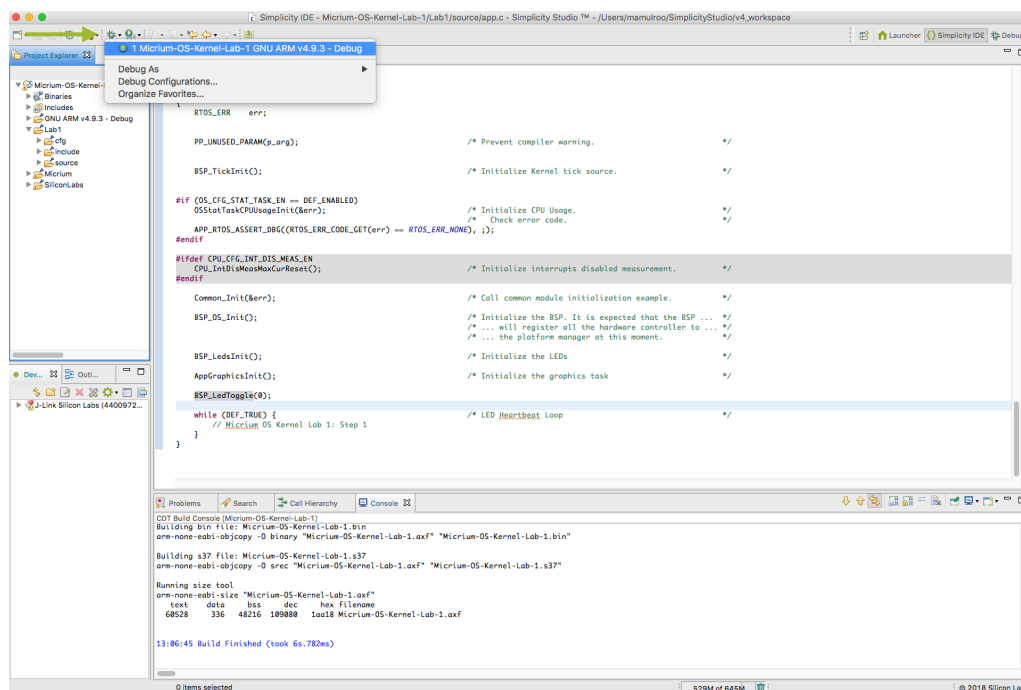
If you have not already, take your Giant Gecko board out of the kit.



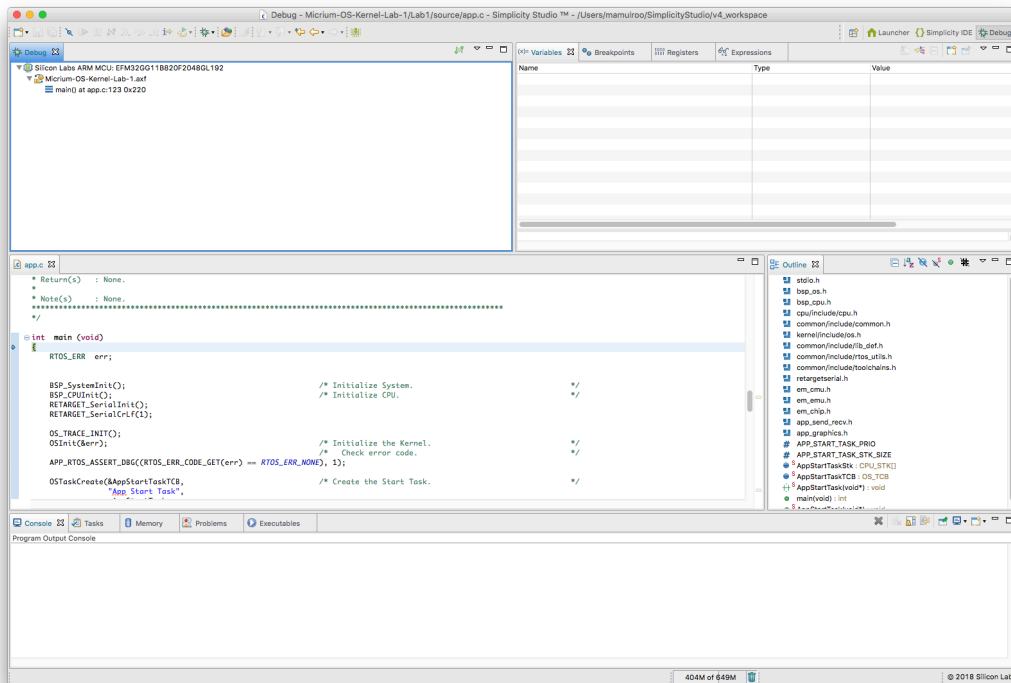
Connect the USB mini cable as show above. Also ensure that the switch highlighted above is set to **AEM**. Once you've connected the USB mini cable and set the switch to **AEM**, plug the USB cable into your computer.



Under the **Device** tab you should see **J-Link Silicon Labs** and underneath that, the **EFM32GG11** part listed. If you do not see your board show up, verify your USB connection and that your computer can find the USB device. If it is reported as an **Unknown Device**, ensure that you have the most recent updates for Simplicity Studio installed.



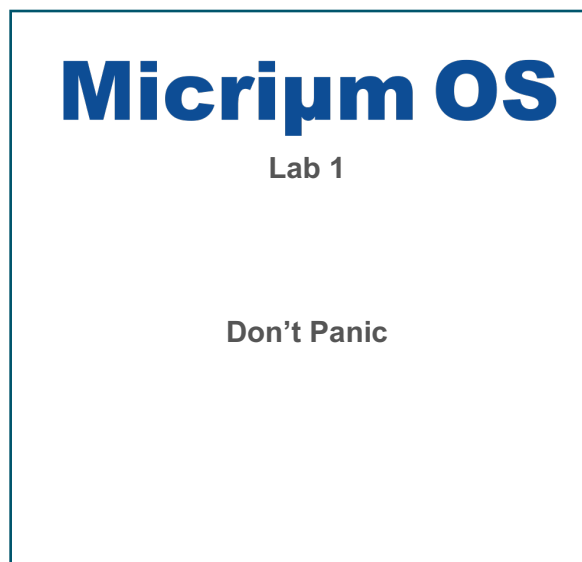
Once the board is powered up, click the debug button in Simplicity Studio and select the **Micrium-OS-Kernel-Lab-1 GNU ARM** debug target.



After clicking the Debug button and selecting the target, Simplicity Studio will switch you to the Debug window perspective. After the board is flashed, the application will halt at main as shown above.

Application Status

Once your Giant Gecko kit is flashed with the Micrium OS application, it will run the application when you click **Resume** while debugging or it will start automatically when not in debug mode. Upon boot-up, Micrium OS goes through initializations to start up the kernel. Once it is successfully initialized, the output on the LCD screen should be the same as below. Also, both LEDs should be toggling at the rate specified in the call to OSTimeDlyHMSM().



Micrium OS Kernel Lab 2

Overview

Micrium OS Kernel applications are made up of tasks, each of which is simply a C function. Application developers use Micrium OS Kernel's task creation routine, **OSTaskCreate()**, to indicate that a particular function should be treated as a task. In this lab, you'll work with **OSTaskCreate()** as you add a new task to an existing application.

The goal of this lab is to create a task to control a traffic light. The traffic light will operate on a simple time schedule: 10 seconds red, 10 seconds green, 1.5 seconds yellow.

The code for the Traffic Light Task has already been written, your job is to create the task so uC/OS-III schedules the task to run. You will need to declare a stack, priority and TCB for the task and then create the task.

Loading your project into Simplicity Studio

Load the Lab 2 project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Writing the Task Code

From the project explorer open up **app.c**. Locate the comments for the lab and follow the steps below.

Step 1:

Add the following defines to the **Step 1** comment. This will specify the priority and task size for the Traffic Light task.

```
#define APP_CFG_TRAFFIC_LIGHT_TASK_PRI0    2u
#define APP_CFG_TRAFFIC_LIGHT_STK_SIZE    512u
```

Step 2:

Add the following global variables to the **Step 2** comment. These variables will be used for the LED Task's stack and Task Control Block (TCB)

```
static CPU_STK  TrafficLightTaskStk[APP_CFG_TRAFFIC_LIGHT_STK_SIZE];

static OS_TCB   TrafficLightTaskTCB;
```

Step 3:

Add the following function declaration to the **Step 3** comment. This is the function that will be called by Micrium OS Kernel when multitasking starts. Once multitasking has started the kernel will begin scheduling tasks to run, including the newly added LED Task.

```
static void AppLEDTask (void *p_arg);
```

Step 4:

In this step you will need to add the **OSTaskCreate()** call at the **Step 4** comment. You will notice that we are creating the LED Task from the Start Task. It is common for most Micrium OS projects to create one task, start the OS and then create the other application tasks after the kernel has started.

The code for **OSTaskCreate()** is not being provided like the previous steps. Instead, you can use **AppStartTask's** **OSTaskCreate()** call as a reference (be sure to change all of the variables from start to LED!) or you can use <https://doc.micrium.com> to look up all of the parameters for **OSTaskCreate()**.

Step 5:

Now that we've created the LED Task, we need to add code to the task's loop to toggle the LED. Copy the following code into **Step 5**.

```
BSP_LedToggle(1);  
OSTimeDlyHMSM(0, 0, 0, 500,  
              OS_OPT_TIME_HMSM_STRICT,  
              &err);
```

Running the Application

After adding code following the steps above, compile and flash the Giant Gecko Starter Kit similar to what was done in Lab 1.

Once your Giant Gecko kit is flashed with the Micrium OS application, it will run the application when you click **Resume** while debugging or it will start automatically when not in debug mode. Upon boot-up, Micrium OS goes through initializations to start up the kernel. Once it is successfully initialized, you will see an output on the LCD screen similar to the one below. Also, both LEDs should be toggling at the rate specified in the call to **OSTimeDlyHMSM()** from the **AppStartTask** and **AppLEDTask**.



Micrium OS

Lab 2

**Great
Scott!**

Micrium OS Kernel Lab 3

Overview

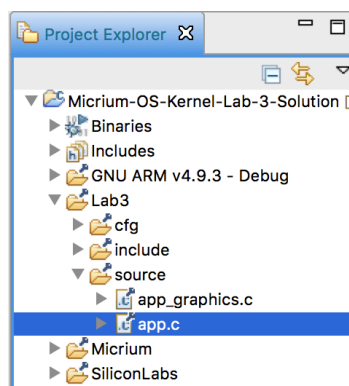
The interrupt handlers in a Micrium OS Kernel application somewhat resemble those found in other applications. However, in general, there are a few differences between interrupt handling in single-threaded applications and kernel-based code. In this exercise, you'll get to experience some of these differences as you add an interrupt handler to an existing Micrium OS Kernel application.

For this lab, the goal is to configure GPIO interrupts to display text to the LCD screen when a button is pressed. You will take advantage of the Silicon Labs GPIO interrupt driver to handle part of the interrupt, but then you will use your own separate handler to perform actions during the button press interrupt.

Loading your project into Simplicity Studio

Load the Lab 3 project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Writing Kernel-based Interrupts



From the project explorer open up **app.c**. Locate the comments for the lab and follow the steps below.

Step 1:

You'll notice code has been provided to configure the GPIOs for PB0 and PB1. This code is using the Silicon Labs emdrv library for GPIO interrupts. The provided code initializes the necessary clocks, registers a callback for the GPIO pins associated with the push buttons, and it configures the interrupts in the GPIO module.

Your task is to enable the GPIO_EVEN_IRQHandler and GPIO_ODD_IRQHandler using the Micrium OS CPU module. The GPIO_EVEN_IRQHandler and GPIO_ODD_IRQHandler are the two interrupt handlers inside the GPIO interrupt module that then dispatch to the function specified in the GPIOINT_CallbackRegister() function.

You should first set the interrupt handlers using CPU_IntSrcHandlerSetKA(). For this function, when specifying the interrupt ID you must add 16 to account for the 16 default ARM interrupts (ex. CPU_IntSrcHandlerSetKA(GPIO_EVEN_IRQn + 16)). Make calls for both the even and odd interrupt handlers.

After enabling the even and odd interrupt handlers, use the **CPU_IntSrcEn()** call to enable the even and odd interrupt handlers. In this call you do not need to add 16 to the interrupt id.

Step 2:

Inside of our interrupt handler for the button press we need to tell the kernel that we've entered an interrupt. Add the following code to **Step 2**.

```
CPU_SR_ALLOC();  
CPU_CRITICAL_ENTER();  
OSIntEnter();  
CPU_CRITICAL_EXIT();
```

Step 3:

After completing everything we need to do inside of our interrupt handler we need to tell the kernel that we're now exiting the interrupt. If we're not in a nested interrupt, this will trigger the kernel to run the scheduler to see if anything has changed inside the system. Add the following code to **Step 3** to signal to the kernel we're exiting the interrupt.

```
OSIntExit();
```

Running the Application

After adding code following the steps above, compile and flash the Giant Gecko Starter Kit similar to what was done in the previous labs.

Once your Giant Gecko kit is flashed with the Micrium OS application, use **BTN0** to display text on the LCD screen. After three button presses the screen will not add anymore text. Use **BTN1** to clear the LCD screen.



Micrium OS

Lab 3

Alright,
alright,
alright.

Micrium OS Kernel Lab 4

Overview

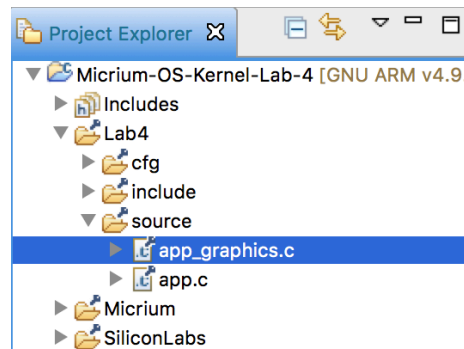
Application developers working with a real-time kernel often find it helpful to synchronize the actions of two tasks, or of a task and an ISR. The semaphore is a kernel primitive through which such synchronization is often accomplished. In this lab, you'll use a semaphore to synchronize a task with an ISR.

The goal of this lab is to create an application that has a similar behavior to Lab 3, but instead of using two of the push buttons only one will be used. Every time a button is pressed, a signal will be sent via a semaphore from the ISR to the LCD task.

Loading your project into Simplicity Studio

Load the Lab 4 project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Adding Semaphore code to the application



From the project explorer open up **app_graphics.c**. Locate the comments for the lab and follow the steps below.

Step 1:

Declare a Semaphore object similar to how **OS_TCB** is declared. The object type you should declare is **OS_SEM**.

Step 2:

Create the Semaphore using a kernel API call. Remember to check your error codes!

Step 3:

Add a Semaphore Pend. This will cause the LCD to draw the initial screen and then only update when a button is pressed.

Step 4:

Post a Semaphore from the ISR to signal the button press.

Running the Application

After adding code following the steps above, compile and flash the Giant Gecko Starter Kit similar to what was done in the previous labs.

Once your Giant Gecko kit is flashed with the Micrium OS application, use BTN0 to display text on the LCD screen. After the third press the screen should clear itself and subsequent button presses should then begin to add more text again.

Micrium OS

Lab 4

Alright,
alright,
alright.

Micrium OS Kernel Lab 5

Overview

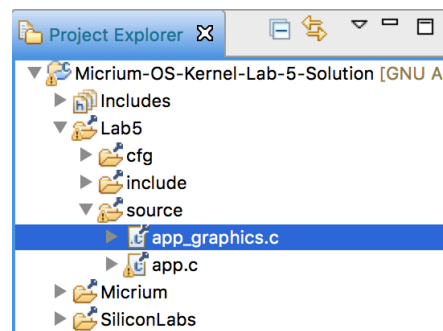
If two tasks in a Micrium OS Kernel application both use a particular resource (such as a global variable or a peripheral device), the developers of the application must make sure that each of the tasks has exclusive access to that resource. Oftentimes, developers in this situation turn to a mutual exclusion semaphore, or mutex. This exercise will afford you the opportunity to work with Micrium OS Kernel's mutex functions.

The goal of this lab is protect the variable **AppCount** that is shared between **AppCountTask()** and **AppGraphicsTask()**. The **AppCount** variable is incremented by a **BTN0** press and the value of **AppCount** is printed to the LCD screen from **AppGraphicsTask()**.

Loading your Project into Simplicity Studio

Load the Lab 5 project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Using a Mutex to Protect a Count Variable



Open up **app_graphics.c** from the project explorer. All of the code you'll be writing for this lab will be in this file.

You will find a variable called **AppCount** has already declared. This variable is used in the **AppGraphics** task and will be needed by a task you create. The code you will need to write for this lab will include:

- Declare a Mutex object and create the Mutex using the kernel API.
- In the **AppCount Task**, before incrementing the **AppCount** variable, be sure to use a **Mutex** to protect the variable.
- In the **AppGraphics Task**, protect the copy of **AppCount** to **count_local** using a **Mutex**

Running the Application

After adding code following the steps above, compile and flash the Giant Gecko Starter Kit similar to what was done in the previous labs.

Once your Giant Gecko kit is flashed with the Micrium OS application, use BTN0 to increment the count variable.

Micrium OS

Lab 5

Count: 0

Micrium OS Kernel Lab 6

Overview

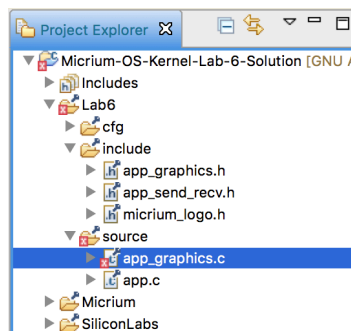
Most of the services offered by Micrium OS Kernel fall into one of three primary categories: synchronization (which includes the semaphore services used in Lab 4), mutual exclusion (the category encompassing the mutex services of Lab 5) and inter-task communication. In this exercise, you'll be working with the primitives through which the kernel facilitates inter-task communication: message queues. Via message queues, tasks can easily share practically any type of data.

In this lab you will use a message queue to send a single 32-bit value from an interrupt to the graphics task. When **BTN0** is pressed, the message queue should signal to the graphics task to increment the count by 1. When **BTN1** is pressed, the message queue should signal to the graphics task to increment the count by 2.

Loading your project into Simplicity Studio

Load the Lab 6 project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Using a Message Queue to Send Button Press Data



Open up **app_graphics.c** from the project explorer. All of the code you'll be writing for this lab will be in this file.

You will find that code to handle the GPIO interrupt has already been declared in **AppGPIOCallback()** but currently no action is being taken when the button press occurs. You will also find that in **AppGraphics_Task()**, the task currently does not have a delay in it and there is a variable called count local that is passed into the update screen function. Using a message queue, write code that increments the count variable by one when **BTN0** is pressed and by two when **BTN1** is pressed.

Running the Application

After adding code following the steps above, compile and flash the Giant Gecko Starter Kit similar to what was done in the previous labs.

Once your Giant Gecko kit is flashed with the Micrium OS application, use **BTN0** to increment the count variable by one and **BTN1** to increment the count variable by two.

Micrium OS

Lab 6

Count: 0

Micrium OS Kernel Lab Final Project

Overview

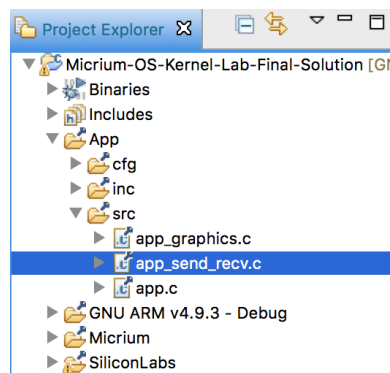
Now that we've covered all of the main aspects of Micrium OS Kernel, it is time to put them all together in a more complicated project.

For this lab, you will use a number of different kernel services to allocate data when a button press is received, store the current tick value, send the data to a receive task and free it when the other button is pressed. The recommended kernel services to use will be: Event Flags, Counting Semaphore, Mutex, Memory Allocation and Message Queue.

Loading your project into Simplicity Studio

Load the Final Project from your lab zip file into Simplicity Studio. If you need step-by-step instructions refer to Lab 1 Section 1-2.

Lab Requirements



Open up the file **app_send_recv.c**. This is where you will be writing all of the code for this lab.

You will find that there are two tasks created for you, **AppSendTask** and **AppRecvTask**. The objective of the lab is to be able to send data from the send task to the receive task. The LCD task has already been created for you, you just need to ensure you update the **AppSendRecv_Stats** structure so data is appropriately represented on the screen.

AppSendRecvInit:

- Initialize all of the kernel objects necessary for the lab **before** the **OSTaskCreate()** calls

AppSendTask:

- Wait for a **BTN0** press.
- Allocate an **AppSendRecv_Data_t** structure from the Dynamic Memory Pool **or** wait until a block is free. Do not queue up the number of button presses while waiting for a memory block, only **one** button press will be stored while waiting.
- Update the **AppSendRecv_Stats** structure fields: **newest_tick**, **free**, **in_use** and **total_allocated**. Ensure the structure is not read by the graphics task while being updated.
- Send the data structure to the receive task.

AppRecvTask:

- Wait for a **BTN1** press
- Check if there is any data to read/free. If no data, wait for the next button press. Otherwise, process the data.
- Update the **AppSendRecv_Stats** structure fields: `freed_tick`, `free`, `in_use` and `total_freed`. Ensure the structure is not read by the graphics task while being updated.
- Return the data to the Dynamic Memory Pool

AppSendRecvGetStats:

- Copy the contents of **AppSendRecv_Stats** to `p_data`. Ensure the data **does not** change while copying.

AppSendRecv_GPIOInterrupt:

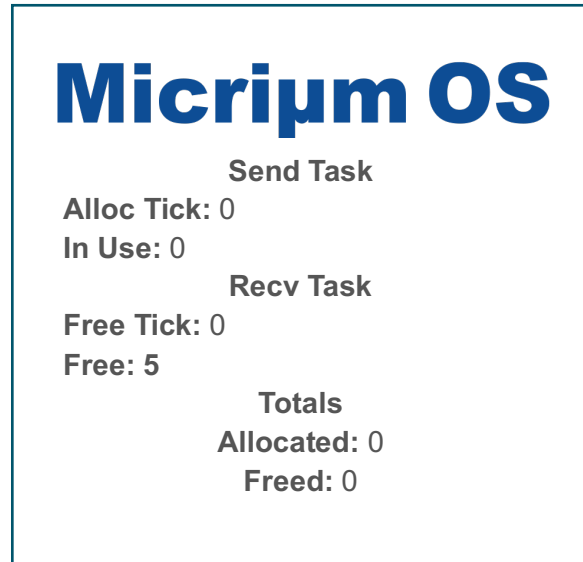
- On **BTN0** press, send a signal to `SendTask`.
- On **BTN1** press, send a signal to `RecvTask`.

Hints:

- All of the kernel objects used in the instructor's solution have been declared under Local Global Variables
- All kernel objects should be created in **AppSendRecvInit()** and not in the `Send` or `Recv` Task
- Using **Mem_Copy()** in **AppSendRecvGetStats()** is a good way to copy to `p_data`. Be sure to protect the copy! If you're unsure what this means, ask your instructor.
- A counting semaphore and dynamic memory pool work very nicely together.

Application Status

Once your Giant Gecko kit is flashed with the Micrium OS application, it will run the application when you click **Resume** while debugging or it will start automatically when not in debug mode. Upon boot-up, Micrium OS goes through initializations to start up the kernel. Once it is successfully initialized, a status page similar to the one below will be displayed on the LCD screen.



Send Task:

Pressing BTN0 will attempt to allocate a memory struct to store the current tick in. If it's able to grab one it updates the **In Use/Free** values and the **Alloc Tick** value. If one is not available, the press is stored until memory becomes available. It **does not** keep track of how many button presses occur while waiting.

Recv Task:

Pressing BTN1 will attempt to free any messages waiting. If none are available, nothing happens. If there is a message waiting, the **Free Tick** is set to the received tick value and **In Use/Free** are updated.

Totals:

Keeps track of the total number of times memory has been allocated/freed.

Conclusion

We hope you enjoyed the Micrium OS Kernel Labs.

For more information on Micrium OS please refer to the following link:

<https://www.silabs.com/products/development-tools/software/micrium-os>

For all of the online documentation for Micrium OS:

<https://doc.micrium.com/>

For any other questions you may have, feel free to reach out to me:

Mark Mulrooney

E-Mail: mark.mulrooney@silabs.com

Thank you!