CC1350 TI-RTOS/RF LABS

Tasks to complete:
Complete Lab 1 of the CC1350 LAB document. For each task provide snapshot of the modifications in the cfg file XGCONF, Stack Usage for each case, log variables using ROV, execution graphs using RTOS Analyzer, and videos of the demo.

Follow the submission guideline to be awarded points for this Lab.
Submit the following for all Labs:
1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder CC1350-LABs, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.
3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (with all snapshots requested), b) source code file(s) with all include files, c) text file with youtube video links (see template).

TI-RTOS Basics

Introduction

The SimpleLink™ software development kits (SDKs) includes TI-RTOS support.

TI-RTOS accelerates development schedules by eliminating the need to create basic system software functions from scratch. TI-RTOS scales from a minimal footprint real-time multitasking kernel - TI-RTOS Kernel (formerly known as SYS/BIOS) - to a complete RTOS solution including protocol stacks, multi-core communications, device drivers and power management. By providing essential system software components pre-tested and pre-integrated, TI-RTOS enables developers to focus on differentiating their application.

Lab: Getting started

Software

- CCS 7.1 or higher (please use Desktop version, not CCS Cloud)
- Any SimpleLink SDK v1.40 or higher

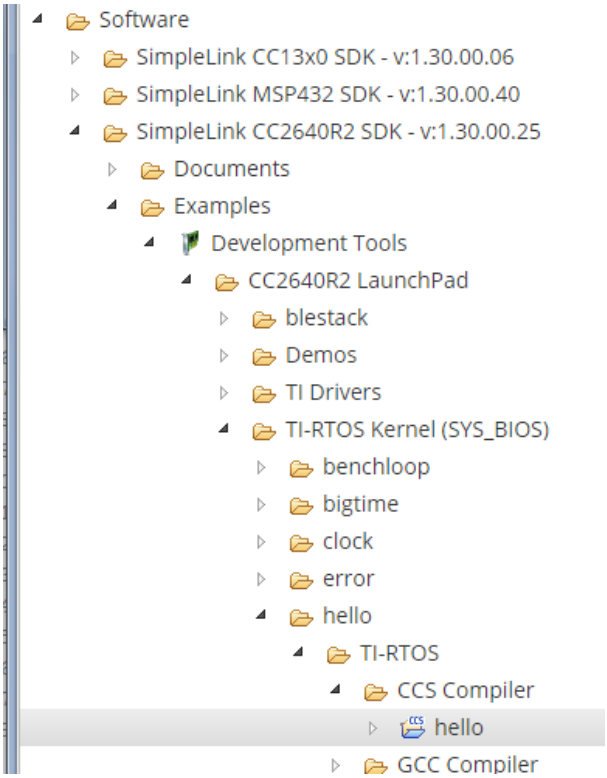
Hardware

- Any supported SimpleLink LaunchPad™ Development Kit

The below steps will use simplelink_cc2640r2_sdk_1_30_00_25 along with the CC2640R2-LAUNCHXL LaunchPad board. So some of the pictures/directory names/line numbers/sizes/etc. might be slightly different. Convert all reference or cc2640r2 to CC13X0

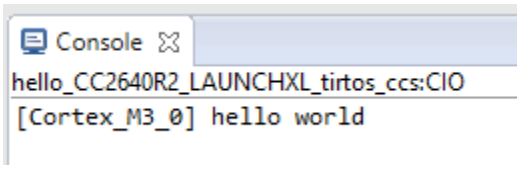
Making sure it works

Open your Desktop Code Composer Studio and import the TI-RTOS `hello` project from your SimpleLink SDK inside of Resource Explorer. We'll be changing the TI-RTOS kernel configuration file and this is not currently supported in CCS Cloud. Make sure to select the "CCS Compiler" version.



To test that the software and hardware pre-requisites are fulfilled we are going to build and run the project before going to the first task.

- Our first mission is to build the imported project. Select the project in Project Explorer and choose `Project` → `Build Project` from the menu.
- When the project is built, we are going to make sure that the hardware and debugger work. To start debugging, press `Run` → `Debug`, or press F11.
- When the download is finished, press F8 or the green play button to run.
- You should see "hello world". The program terminates after printing the text. For example, this is what it should like like on a CC2640R2-LAUNCHXL LaunchPad.



On Building

Note that the first time you build the project the whole TI-RTOS kernel will also be built. This may take several minutes, but is only done the first time. Subsequent builds will re-use the compiled kernel unless a configuration change is done.

Task 1 - Replacing the contents of hello.c

Please select the below text and replace the entire contents of `hello.c` (you can leave the license banner if you want). Then rebuild/reload/run the project. You should get a flashing LED.

Select text

```
/* TI-RTOS Header files */
#include <xdc/std.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

#include <ti/drivers/GPIO.h>

/* Example/Board Header files */
#include "Board.h"

void myDelay(int count);

/* Could be anything, like computing primes */
#define FakeBlockingSlowWork()   myDelay(12000000)
#define FakeBlockingFastWork()   myDelay(2000000)

Task_Struct workTask;
/* Make sure we have nice 8-byte alignment on the stack to avoid wasting memory */
#pragma DATA_ALIGN(workTaskStack, 8)
#define STACKSIZE 1024
static uint8_t workTaskStack[STACKSIZE];

void doUrgentWork(void)
{
    GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_OFF);
    FakeBlockingFastWork(); /* Pretend to do something useful but time-consuming */
    GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_ON);
}

void doWork(void)
{
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
    FakeBlockingSlowWork(); /* Pretend to do something useful but time-consuming */
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
}

Void workTaskFunc(UArg arg0, UArg arg1)
{
    while (1) {

        /* Do work */
        doWork();

        /* Wait a while, because doWork should be a periodic thing, not continuous.*/
        myDelay(24000000);
    }
}

/*
```

```
 *  ======== main ========
 *
 */
int main(void)
{
    Board_initGeneral();
    GPIO_init();

    /* Set up the led task */
    Task_Params workTaskParams;
    Task_Params_init(&workTaskParams);
    workTaskParams.stackSize = STACKSIZE;
    workTaskParams.priority = 2;
    workTaskParams.stack = &workTaskStack;

    Task_construct(&workTask, workTaskFunc, &workTaskParams, NULL);

    /* Start kernel. */
    BIOS_start();

    return (0);
}

/*
 *  ======== myDelay ========
 *  Assembly function to delay. Decrements the count until it is zero
 *  The exact duration depends on the processor speed.
 */
__asm("    .sect \".text:myDelay\"\n"
      "    .clink\n"
      "    .thumbfunc myDelay\n"
      "    .thumb\n"
      "    .global myDelay\n"
      "myDelay:\n"
      "    subs r0, #1\n"
      "    bne.n myDelay\n"
      "    bx lr\n");
```

hello.c

Orienting ourselves in the code

The Lab 1 example comes preconfigured with one TI-RTOS `Task` already constructed in `main()`. This task is set up to use the `workTaskFunc` function as the task function, which in turn uses the GPIO Driver to toggle a LED.

The task is created using the `Task_construct` in the main function. The `main` function also initializes the hardware.

In the `main()` function, after `BIOS_start()` is called it never returns, but instead give control to the TI-RTOS scheduler which will call the Task functions of the tasks that are constructed (e.g. `workTaskFunc`). Normally, task functions will enter an infinite loop and never return, letting TI-RTOS switch to higher priority tasks or temporarily suspend the current task.

Task 2 - Debugging Tools

We are going to take a look at some of the built in features of CCS which can aid in the development of firmware running on TI-RTOS, and also give a better understanding of the multitasking.

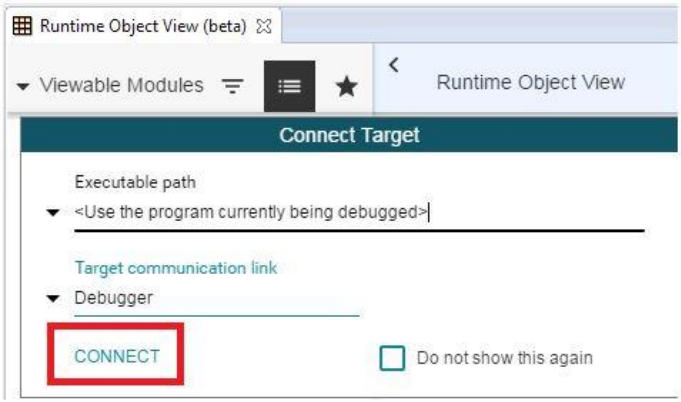
Runtime Object View

As discussed above, Runtime Object Viewer (we'll call it ROV2 here for short) can be used to get a snapshot of the whole RTOS. By default the information is only updated via JTAG when the target is halted. First we are going to halt the code as we are toggling the led.

- Put a breakpoint in the `workTaskFunc` on the `doWork` line. Do this by double clicking on the area on the left of the line number.

```
30 void doWork(void)
31 {
32     GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
33     FakeBlockingSlowWork(); /* Pretend to do something useful but time-consuming */
34     GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
35 }
```

- Run so you hit that breakpoint. Next open the ROV2 by going to `Tools` → `Runtime Object View (beta)`. You'll be prompted to connect.

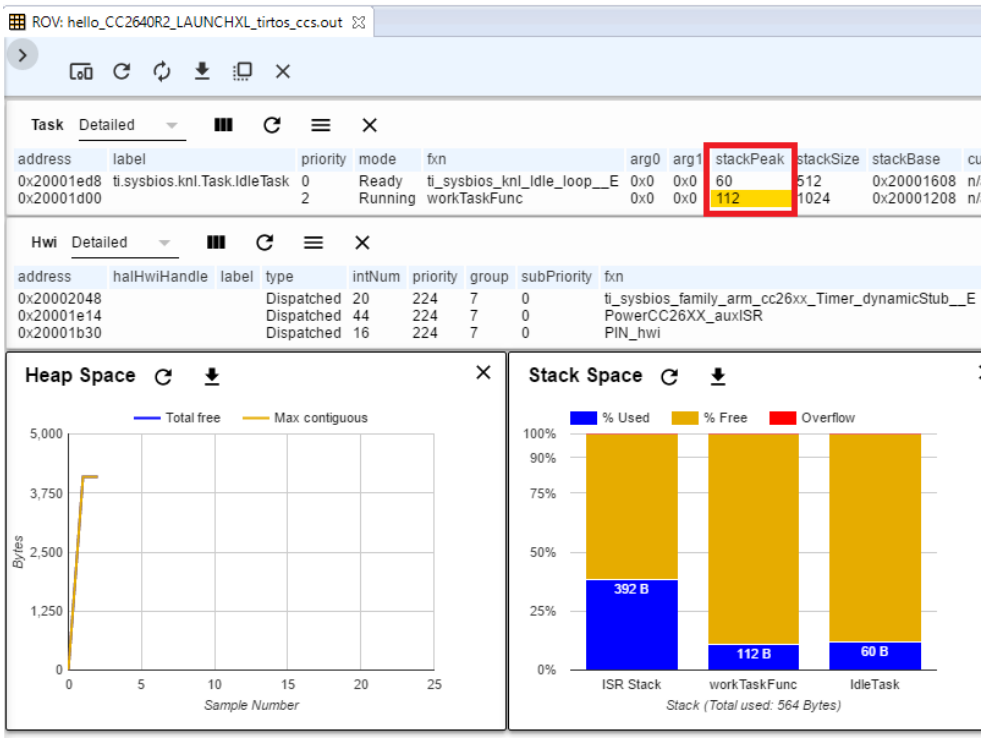


- In ROV2, select the "Import a dashboard" icon and select the `overview.rov.json` file that is in the project.

This dashboard shows which task is currently running on the system, what tasks are blocked as well as what tasks are ready to run.

We can see that the workTaskFunc is currently running and we can also see that the stack usage for the task has peaked at 112 of (1024) STACKSIZE bytes so far, so no

risk of stack overflow Note: we generally recommend you start with a larger stack size and then trim it down once everything is working properly.



The 112 has a yellow background which means it has changed since the last time it was read.

The memory and stack usage graphs can be very useful also. If you run again to the breakpoint, the values will be updated.

Execution graph

While the ROV is handy to get a snapshot view over the current state of the system it is not as easy to get any information about the state of the system over time. For this we use the Execution graph. For this, we'll need to edit the .cfg file.

Edit the .cfg as text or graphically

You can right-click the .cfg file and edit it as a text file (recommended for this lab) or graphically.

- First we need to enable the logging by changing the comments. Open the `hello.cfg` and enable the kernel's logging feature. Note: this is turned on in the debug TI-RTOS kernel configuration project used by the driver examples.

```
/*
 * Enable logs in the BIOS library.
 *
 * Pick one:
 *  - true (default)
 *      Enables logs for debugging purposes.
 *  - false
 *      Disables logging for reduced code footprint and improved runtime
 *      performance.
 */
BIOS.logsEnabled = true;
//BIOS.logsEnabled = false;
```

For CC13xx/CC26xx devices, the kernel in the ROM has logs disabled.

To enable kernel to log events, you cannot use the kernel in the ROM. By removing (or commenting out) the following lines in the .cfg file, the kernel will be placed in the CC26xx's flash instead (and the kernel can log events).

Select text

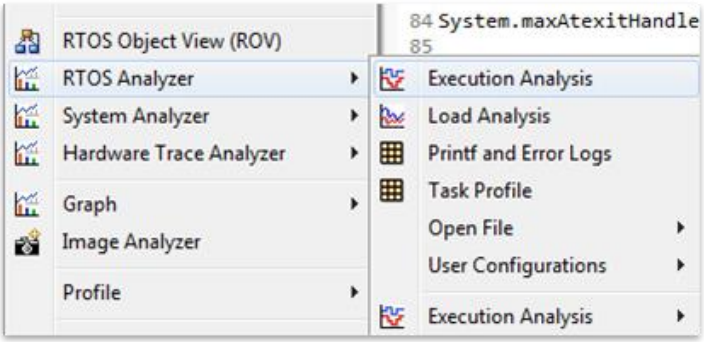
```
var ROM = xdc.useModule('ti.sysbios.rom.ROM');
if (Program.cpu.deviceName.match(/CC2640R2F/)) {
    ROM.romName = ROM.CC2640R2F;
}
else if (Program.cpu.deviceName.match(/CC26/)) {
    ROM.romName = ROM.CC2650;
}
else if (Program.cpu.deviceName.match(/CC13/)) {
    ROM.romName = ROM.CC1350;
}
```

- Add the following lines to the bottom of the `hello.cfg` file. This will configure the kernel to maintain buffers on the target where the log records will reside. Note: this is also in the debug TI-RTOS kernel configuration project used by the driver examples. We're only interested in the kernel's logging so we'll disable the CPU Load logging.

  Select text

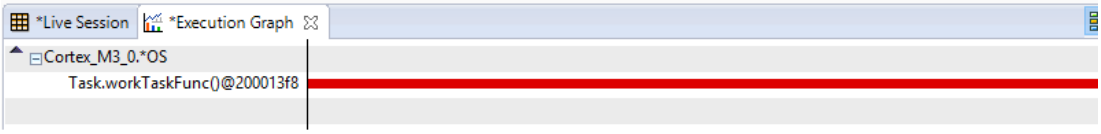
  ```
  var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');
  LoggingSetup.sysbiosLoggerSize = 1024;
  LoggingSetup.loadLogging = false;
  ```

- Rebuild, reload, and run the example for 10-15 seconds (to get a good amount of log data). Remember to remove any breakpoints. Press Halt/Suspend/Pause button to pause the execution of the program.
- Next, open the Execution Analysis menu by going to `Tools` → `RTOS Analyzer` → `Execution Analysis`. Note: you be asked to do a one-time setup.



Select only the `Execution Graph` in the next window, leave everything else as it was and press **Start**.

In the new Execution Graph tab, expand `Cortex_M3_0.*OS` to see that `Task.workTaskFunc` has been executing! In fact, it's the only task executing. The `Idle` task has not gotten any time at all during the execution of the program.



If there is nothing in the execution graph...

We have a small race condition in the tool. If there are no records, please select the Live Session tab and stop/start the collection.

No records

There is only one task running the entire time. It's hogging all the processor.

How does the logging work?

The TI-RTOS module LoggingSetup, which is part of the Universal Instrumentation Architecture (UIA), sets the UIA module LoggerStopMode up as an interface for the XDC Runtime Log module, which has hooks into the Task, Hwi and Swi modules.

The TI-RTOS configuration script parsing acts as an extra precompile step which can add, remove and configure RTSC (Real-Time Software Components) modules by outputting .c and .h files used for later compilation and linking.

Task 3 - Sleeping well

After looking at the Execution Graph, we can see that we have a problem with one of our tasks hogging all CPU resources. Let's take a look at our `workTaskFunc`.

Select text

```
void doWork(void)
{
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
    FakeBlockingWork(); /* Pretend to do something useful but time-consuming */
```

```
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
}

Void workTaskFunc(UArg arg0, UArg arg1)
{
    while (1) {

        /* Do work */
        doWork();

        /* Sleep */
        myDelay(24000000);
    }
}
```

Work, "sleep", work.

The only thing the task does is execute the `doWork` function and then goes back to "sleep", except it never does go to sleep. The `myDelay` function is simply a function which burns CPU cycles in a loop. This is not the correct way to pass time in the RTOS.

One of the easiest ways to pass time in a task is to call `Task_sleep(numTicks)`. `Task_sleep` will simply make the current task sleep for as many system ticks as is specified in the argument. The current tick rate of the system is needed in order to know how long you will sleep. This is a constant value available via the `Clock_tickPeriod` variable. The value is the amount of microseconds per clock tick.

Clock_tickPeriod

To use `Clock_tickPeriod`, remember to include the kernel `Clock module` header: `#include <ti/sysbios/knl/Clock.h>`

The value of this variable [µs/tick] is determined when the TI-RTOS .cfg file is parsed. If `Clock.tickPeriod = nn;` is not present in the .cfg file, the default value is used. Since the tick period can vary between projects, it's useful to include the variable `Clock_tickPeriod` in calculations that depend on system clock ticks.

Task 3.1

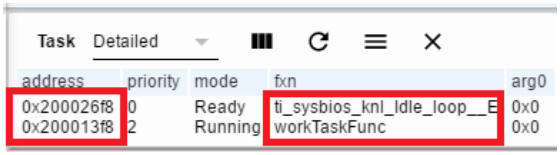
- Replace the use of `myDelay` to sleep with `Task_sleep` and use it to sleep for 500ms.
    - How do you convert an argument in microseconds to an argument in system ticks?

      Answer if you're stumped

- Rebuild/reload/run. Let the code run for a while and have another look at the Execution Graph, does it look any different? Note: you may have to zoom out to see the transitions.

Now we see that idle was given a chance to run. For low power devices, going into idle will allow the device to transition to lower power modes.

Note: for the CC26xx, there are setting to minimize flash usage (e.g. `Text.isLoaded`). We can tell which task is which by looking in ROV2's Task Details



Task 4 - Executing urgent work

Next we are going to expand on the original code by adding a `doUrgentWork` function and task. In our system, this will represent the most important work processing the system needs to do. This is more important than the work done by the `workTask` and should execute as quickly as possible.

Setting up the new task

- First copy, paste and rename the `workTaskFunc` function to create a new task function called `urgentWorkTaskFunc`.
- Let `urgentWorkTaskFunc` call `doUrgentWork`.
- Copy, paste and rename the `Task_Struct` and the task stack storage as well for `urgentTask`.
- Construct the new task (copy and rename the parameters and the construction) and set the priority of the new task to 1. We'll play with this later... **Note:** Higher priority number means higher priority.
- Reduce the `Task_sleep()` time to 50ms in `urgentWorkTaskFunc`.

CC3220 LaunchPad LEDs

There is a pin conflict with the LEDs and I2C/PWM. To get LED1 to work, please read the comments in the *Board* files and adjust accordingly. For example, with the CC3220SF-LAUNCHXL, change the following lines in the following files

- `Board.h`: Change LED1's define `#define Board_GPIO_LED1 CC3220SF_LAUNCHXL_GPIO_LED_D6`
- `CC3220SF_LAUNCHXL.h`: Uncomment `CC3220SF_LAUNCHXL_GPIO_LED_D6,`

- `CC3220SF_LAUNCHXL.c`: Uncomment `GPIOCC32XX_GPIO_10 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_HIGH | GPIO_CFG_OUT_LOW,`

Tasks

A `Task` has some information associated with it. This is stored in the `Task_Struct`, which holds the variables the TI-RTOS kernel needs to act on the Task, for example to make it pend on a Semaphore, place it in a Ready queue, or just check the current priority.

A `Task` also needs a `Stack` to place function-local variables. The stack is just an array of bytes that we tell TI-RTOS to use. When a specific Task is running, the CPU's stack pointer will point into the memory area of this array. This is a part of how multi-threading is accomplished, because each Task thinks on a low level that it is operating independently. For example `workTaskFunc` uses `workTaskStack` for its local variables and function calls.

For more information on tasks, refer to the TI-RTOS kernel documentation in the *SimpleLink_SDK_Install_dir*/docs/documentation_overview.html file.
Rebuild/load/run!

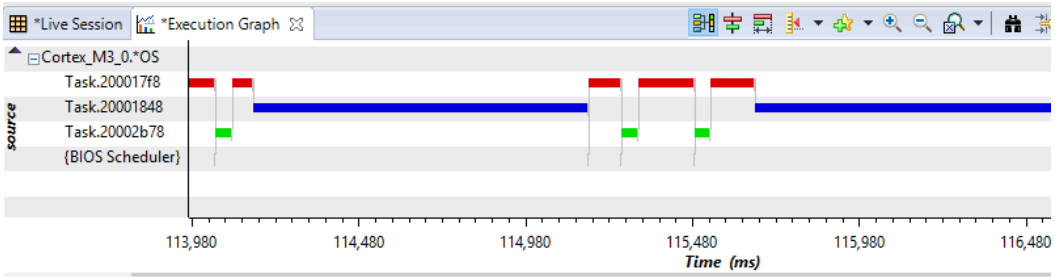
Which LED is flashing at the desired rate?

LED0LED1

Why is LED1 not running at the desired rate?

The urgentWorkTaskFunc task has a lower priorityI messed up

Let's look at the execution graph again.



Solution if your not sure

Changing priority

Let's just change the `workTaskParams.priority` from 1 to 3 and rebuild/reload/run again.

```
    workTaskParams.priority = 3;
    workTaskParams.stack = &urgentWorkTaskStack;

    Task_construct(&urgentWorkTask, urgentWorkTaskFunc, &workTaskParams, NULL);
```

The "urgent" LED1 is now flashing at the desired rate (because it's an higher priority task now).