CPE403/ECG603/ECG710 – Advanced Embedded Systems/Real-Time Embedded Systems

Design Assignment 3

DO NOT REMOVE THIS PAGE DURING SUBMISSION:

Name: Joshua Martinez

Email: marti87@unlv.nevada.edu

Github Repository link (root): https://github.com/JoshuaMa2003/submission_da_0011

Youtube Playlist link (root):
https://www.youtube.com/playlist?list=PLZ09MA_uFt61Vw2JCAtw-_8vAdmoCKXpZ

Follow the submission guideline to be awarded points for this Assignment.

Submit the following for all Assignments:

1. In the document, for each task submit the modified or included code (from the base code) with highlights and justifications of the modifications. Also include comments. If no base code is provided, submit the base code for the first task only.
2. Create a private Github repository with a random name (no CPE403/603/710, Lastname, Firstname). Place all labs under the root folder MSP432E4/CC1352, sub-folder named Assignment1, with one document and one video link file for each lab, place modified c files named as asng_taskxx.c.
3. If multiple c files or other libraries are used, create a folder asng1_t01 and place these files inside the folder.
4. The folder should have a) Word document (see template), b) source code file(s) with startup_ccs.c and other include files, c) text file with youtube video links (see template).
5. Submit the doc file in canvas before the due date. The root folder of the github assignment directory should have the documentation and the text file with youtube video links.
6. Organize your videos as a playlist under the name “ADVEMBSYS”. The playlist should

have the video sequence arranged as submission or due dates.

7. Only submit the PDF. Upload this document in the github repository and in canvas.

1. **Code for tasks: For each task, submit the relevant modified/section or included code (from the base code) with highlights and justifications of the modifications. Also include the comments. If no base code is provided, submit the initialization and execution section of each task separately. Use a separate page for each task.**

Task 1 – ADC Task (5ms, ADC0 CH0 on joystick)

- **Original Code:**
  The TI-RTOS mutex_MSP_ESP432E401Y_tirtos_ccs example did not include any ADC usage. It only had two generic tasks (taskFxn0, taskFxn1) that printed messages and used a mutex.
- **Modified Code:**

```
/*
 *  ======== Configuration constants ========
 */

#define TASKSTACKSIZE           2048    // Stack size for each task
#define SYSTEM_TICK_MS          1       // BIOS Clock tick (1 ms)

#define ADC_PERIOD_MS           5       // ADC task release period
#define UART_PERIOD_MS          10      // UART task release period
#define PWM_PERIOD_MS           15      // PWM task release period
#define HEARTBEAT_PERIOD_MS     500     // Heartbeat blink period

#define ADC_MAX_VALUE           4095U   // 12-bit ADC full-scale value
```

```
/*
 *  ======== Globals: shared data and handles ========
 */

/* Most recent ADC sample (updated by ADC task, read by others) */
static volatile uint16_t gAdcValue = 0;

/* ADC value latched by the switch HWI for use by the PWM task */
static volatile uint16_t gLatchedAdcValue = 0;

/* Flag set in Hwi when a new PWM update is pending */
static volatile bool gPwmUpdatePending = false;

/* Driver handles (initialized in main) */
static UART_Handle uartHandle = NULL;
static PWM_Handle  pwmHandle  = NULL;

/* Semaphore handles (obtained from the constructed structs) */
static Semaphore_Handle adcSem       = NULL;
static Semaphore_Handle uartSem      = NULL;
static Semaphore_Handle pwmSem       = NULL;
static Semaphore_Handle adcDataMutex = NULL;
static Semaphore_Handle uartMutex    = NULL;
```

```c
/*
 *  ======== Clock callbacks (1 ms base tick → 5/10/15/500 ms periods) ========
 */

static void adcClockFxn(UArg arg0)
{
    (void)arg0;
    /* Trigger ADC Task at 5 ms instances of the 1 ms system tick */
    if (adcSem != NULL) {
        Semaphore_post(adcSem);
    }
}
```

```c
/*
 *  ======== ADC Task ========
 *
 *  - Waits on adcSem.
 *  - Every 5 ms (Clock driven), performs an ADC conversion on ADC0-CH0.
 *  - Stores the result in gAdcValue under protection of adcDataMutex.
 */

static void adcTaskFxn(UArg arg0, UArg arg1)
{
    (void)arg0;
    (void)arg1;

    ADC_Handle adcHandle;
    uint16_t sample;

    /* Open ADC driver instance configured as CONFIG_ADC_0 in SysConfig.
     * Make sure CONFIG_ADC_0 is mapped to PE3/AIN0 in your .syscfg file.
     */
    adcHandle = ADC_open(CONFIG_ADC_0, NULL);
    if (adcHandle == NULL) {
        /* If ADC fails to open, halt here */
        while (1) { }
    }

    for (;;) {
        /* Block until the 5 ms Clock posts the semaphore */
        Semaphore_pend(adcSem, BIOS_WAIT_FOREVER);

        /* Single ADC conversion */
        if (ADC_convert(adcHandle, &sample) == ADC_STATUS_SUCCESS) {

            /* Protect the shared variable while updating it */
            Semaphore_pend(adcDataMutex, BIOS_WAIT_FOREVER);
            gAdcValue = sample;
            Semaphore_post(adcDataMutex);
        }
    }
}
```

Description: The original mutex example had no ADC, so a new ADC task was created. A Clock object (*adcClockStruct*) with a 5 ms period posts a counting semaphore (*adcSem*) that wakes the ADC task at every 5 ms tick. Inside *adcTaskFxn* I open *CONFIG_ADC_0* (configured in SysConfig as ADC0 CH0 / joystick input), perform *ADC_convert(),* and write the result to the shared global *gAdcValue*. A binary semaphore (*adcDataMutex*) is used as a mutex so that other tasks (UART, PWM) can safely read this shared ADC value without race conditions.

Task 2 – UART Display Task (10 ms)

- **Original Code:**

  The mutex example only contained simple UART prints inside its demo tasks. These were not synchronized to a specific period and did not share ADC data.

- **Modified Code:**

```
/*
 *  ======== Configuration constants ========
 */

#define TASKSTACKSIZE           2048    // Stack size for each task
#define SYSTEM_TICK_MS          1       // BIOS Clock tick (1 ms)

#define ADC_PERIOD_MS           5       // ADC task release period
#define UART_PERIOD_MS          10      // UART task release period
#define PWM_PERIOD_MS           15      // PWM task release period
#define HEARTBEAT_PERIOD_MS     500     // Heartbeat blink period

#define ADC_MAX_VALUE           4095U   // 12-bit ADC full-scale value
```

```
/*
 *  ======== Globals: shared data and handles ========
 */

/* Most recent ADC sample (updated by ADC task, read by others) */
static volatile uint16_t gAdcValue = 0;

/* ADC value latched by the switch HWI for use by the PWM task */
static volatile uint16_t gLatchedAdcValue = 0;

/* Flag set in Hwi when a new PWM update is pending */
static volatile bool gPwmUpdatePending = false;

/* Driver handles (initialized in main) */
static UART_Handle uartHandle = NULL;
static PWM_Handle  pwmHandle  = NULL;

/* Semaphore handles (obtained from the constructed structs) */
static Semaphore_Handle adcSem       = NULL;
static Semaphore_Handle uartSem      = NULL;
static Semaphore_Handle pwmSem       = NULL;
static Semaphore_Handle adcDataMutex = NULL;
static Semaphore_Handle uartMutex    = NULL;
```

```
static void uartClockFxn(UArg arg0)
{
    (void)arg0;
    /* Trigger UART Task at 10 ms instances */
    if (uartSem != NULL) {
        Semaphore_post(uartSem);
    }
}
```

```c
/*
 *  ======== UART Task ========
 *
 *  - Waits on uartSem.
 *  - Every 10 ms, reads current ADC value from gAdcValue
 *    (guarded by adcDataMutex).
 *  - Prints the value and percentage over UART.
 *  - Uses uartMutex so PWM task and UART task do not interleave text.
 *
 */

static void uartTaskFxn(UArg arg0, UArg arg1)
{
    (void)arg0;
    (void)arg1;

    uint16_t adcCopy;
    char msg[64];

    for (;;) {
        /* Wait for 10 ms period tick */
        Semaphore_pend(uartSem, BIOS_WAIT_FOREVER);

        /* Take a safe snapshot of the ADC value */
        Semaphore_pend(adcDataMutex, BIOS_WAIT_FOREVER);
        adcCopy = gAdcValue;
        Semaphore_post(adcDataMutex);

        /* Convert reading to % of full-scale for easier interpretation */
        uint32_t adcPercent =
            ((uint32_t)adcCopy * 100U) / ADC_MAX_VALUE;

        int len = snprintf(msg, sizeof(msg),
                           "[UART] ADC = %4u (%2lu%%)\r\n",
                           adcCopy,
                           (unsigned long)adcPercent);

        if (len > 0) {
            /* UART is a shared resource - guard with mutex semaphore */
            Semaphore_pend(uartMutex, BIOS_WAIT_FOREVER);
            UART_write(uartHandle, msg, (size_t)len);
            Semaphore_post(uartMutex);
        }
    }
}
```

Description: Instead of simple prints from the original demo tasks, I created a dedicated UART task that is triggered every 10 ms via a Clock + semaphore. The task acquires *adcDataMutex*, reads a snapshot of *gAdcValue*, converts it into a percentage of ADC full-scale, and formats a status line. All UART writes are serialized through *uartMutex*, so this task and the PWM task can both print without interleaving characters.

Task 3 – PWM Task (PF0, 15 ms)

- **Original Code:**

  The mutex template did not generate PWM; PF0 was unused. So PWM setup and PF0 behavior were added entirely.

- **Modified Code:**

```
/*
 *  ======== Configuration constants ========
 */

#define TASKSTACKSIZE           2048    // Stack size for each task
#define SYSTEM_TICK_MS          1       // BIOS Clock tick (1 ms)

#define ADC_PERIOD_MS           5       // ADC task release period
#define UART_PERIOD_MS          10      // UART task release period
#define PWM_PERIOD_MS           15      // PWM task release period
#define HEARTBEAT_PERIOD_MS     500     // Heartbeat blink period

#define ADC_MAX_VALUE           4095U   // 12-bit ADC full-scale value
```

```
/*
 *  ======== Globals: shared data and handles ========
 */

/* Most recent ADC sample (updated by ADC task, read by others) */
static volatile uint16_t gAdcValue = 0;

/* ADC value latched by the switch HWI for use by the PWM task */
static volatile uint16_t gLatchedAdcValue = 0;

/* Flag set in Hwi when a new PWM update is pending */
static volatile bool gPwmUpdatePending = false;

/* Driver handles (initialized in main) */
static UART_Handle uartHandle = NULL;
static PWM_Handle  pwmHandle  = NULL;

/* Semaphore handles (obtained from the constructed structs) */
static Semaphore_Handle adcSem       = NULL;
static Semaphore_Handle uartSem      = NULL;
static Semaphore_Handle pwmSem       = NULL;
static Semaphore_Handle adcDataMutex = NULL;
static Semaphore_Handle uartMutex    = NULL;
```

```
static void pwmClockFxn(UArg arg0)
{
    (void)arg0;
    /* Trigger PWM Task at 15 ms instances */
    if (pwmSem != NULL) {
        Semaphore_post(pwmSem);
    }
}
```

```c
/*
 *  ======== PWM Task ========
 *
 *  - Waits on pwmSem (15 ms period).
 *  - Checks if the switch Hwi has latched a new ADC value.
 *  - If so, converts that latched ADC value to a PWM duty fraction
 *    and updates PWM on PF0 (CONFIG_PWM_0).
 *  - Also prints info to UART about the new duty cycle.
 *
 *  interaction: PWM duty only changes when the switch has triggered
 *  and latched a value, and stays fixed otherwise.
 */

static void pwmTaskFxn(UArg arg0, UArg arg1)
{
    (void)arg0;
    (void)arg1;

    uint32_t currentDuty = 0;

    /* Start with duty = 0% (LED off), per assignment */
    PWM_setDuty(pwmHandle, currentDuty);

    for (;;) {
        /* Release every 15 ms */
        Semaphore_pend(pwmSem, BIOS_WAIT_FOREVER);

        /*
         * Safely snapshot the "update pending" flag and the latched ADC
         * value. We briefly disable interrupts to avoid racing the Hwi.
         */
        UInt key        = Hwi_disable();
        bool pending    = gPwmUpdatePending;
        uint16_t latchedVal = gLatchedAdcValue;
        if (pending) {
            gPwmUpdatePending = false;   // consume the event
        }
        Hwi_restore(key);

        if (pending) {
            /* Convert 12-bit ADC value into PWM fraction.
             * PWM_DUTY_FRACTION_MAX is defined by the TI PWM driver when
             * dutyUnits == PWM_DUTY_FRACTION.
             */
            uint64_t scaled =
                ((uint64_t)latchedVal * (uint64_t)PWM_DUTY_FRACTION_MAX) /
                (uint64_t)ADC_MAX_VALUE;

            currentDuty = (uint32_t)scaled;
            PWM_setDuty(pwmHandle, currentDuty);

```

```
            /* Compute some human-readable info for UART */
            char msg[96];
            uint32_t dutyPercent =
                (uint32_t)(((uint64_t)currentDuty * 100U) /
                           (uint64_t)PWM_DUTY_FRACTION_MAX);

            Semaphore_pend(uartMutex, BIOS_WAIT_FOREVER);

            int len = snprintf(msg, sizeof(msg),
                               "[SW1] Latched ADC = %4u (%2lu%%)\r\n",
                               latchedVal,
                               (unsigned long)((uint32_t)latchedVal *
                                               100U / ADC_MAX_VALUE));
            if (len > 0) {
                UART_write(uartHandle, msg, (size_t)len);
            }

            len = snprintf(msg, sizeof(msg),
                           "[PWM] Duty = %4lu (%2lu%%)\r\n",
                           (unsigned long)currentDuty,
                           (unsigned long)dutyPercent);
            if (len > 0) {
                UART_write(uartHandle, msg, (size_t)len);
            }

            Semaphore_post(uartMutex);
        }
    }
}
```

Description: The PWM task is woken every 15ms by a Clock + semaphore, but it only changes PF0's duty cycle when the switch HWI has latched a new ADC value. The ISR stores the joystick reading into *gLatchedAdcValue* and sets *gPwmUpdatePending*. The PWM task enters a small critical section using *Hwi_disable() / Hwi_restore()* to automatically copy and clear the flag, then maps the 12-bit ADC reading to *PWM_DUTY_FRACTION_MAX* and calls *PWM_setDuty()*. This guarantees that the duty cycle changes only on switch events, and stays fixed between presses.

Task 4 – Switch Read Task (HWI on SW1)

- **Original Code (Template):**
  The mutex template already had a GPIO button callback, but it only toggled an LED and posted a semaphore. For this, I changed the handler so that it captures the current ADC value for the PWM task.
- **Modified Code:**

```
180 void SwitchHwiFxn(uint_least8_t index)
181 {
182     (void)index;
183
184     /* Latch the current ADC reading for the PWM task */
185     gLatchedAdcValue   = gAdcValue;
186     gPwmUpdatePending  = true;     // mark that a new duty update is pending
187 }
```

```
98     /* ----- Hook SW1 to our Hwi callback and enable its interrupt ----- */
99     GPIO_setCallback(CONFIG_GPIO_SW1, SwitchHwiFxn);
00     GPIO_enableInt(CONFIG_GPIO_SW1);
01
```

Description: Instead of toggling LEDs, the SW1 interrupt now serves as a Switch Read Task in hardware-interrupt form. When the user presses the switch, *SwitchHwiFxn* runs immediately, grabs the latest ADC sample (*gAdcValue*), and stores it in *gLatchedAdcValue* while also setting *gPwmUpdatePending = true.* The PWM task checks this flag on its 15 ms wakeup and uses that latched value to compare a new duty cycle. Because the ADC task runs periodically and the HWI only latches on button events, the PWM output remains fixed until the next switch press even though the ADC continues sampling.

Task 5 – Heartbeat Function (PF4 LED)

- **Original Code (Template):**
  The mutex template used the board LED as a simple "alive" indicator in the tasks. For this, I repealed that with a separate Clock-driven heartbeat on PF4 so the LED is independent of the other tasks's timing.
- **Modified Code:**

```
/*
 *  ======== Configuration constants ========
 */

#define TASKSTACKSIZE           2048    // Stack size for each task
#define SYSTEM_TICK_MS          1       // BIOS Clock tick (1 ms)

#define ADC_PERIOD_MS           5       // ADC task release period
#define UART_PERIOD_MS          10      // UART task release period
#define PWM_PERIOD_MS           15      // PWM task release period
#define HEARTBEAT_PERIOD_MS     500     // Heartbeat blink period

#define ADC_MAX_VALUE           4095U   // 12-bit ADC full-scale value
```

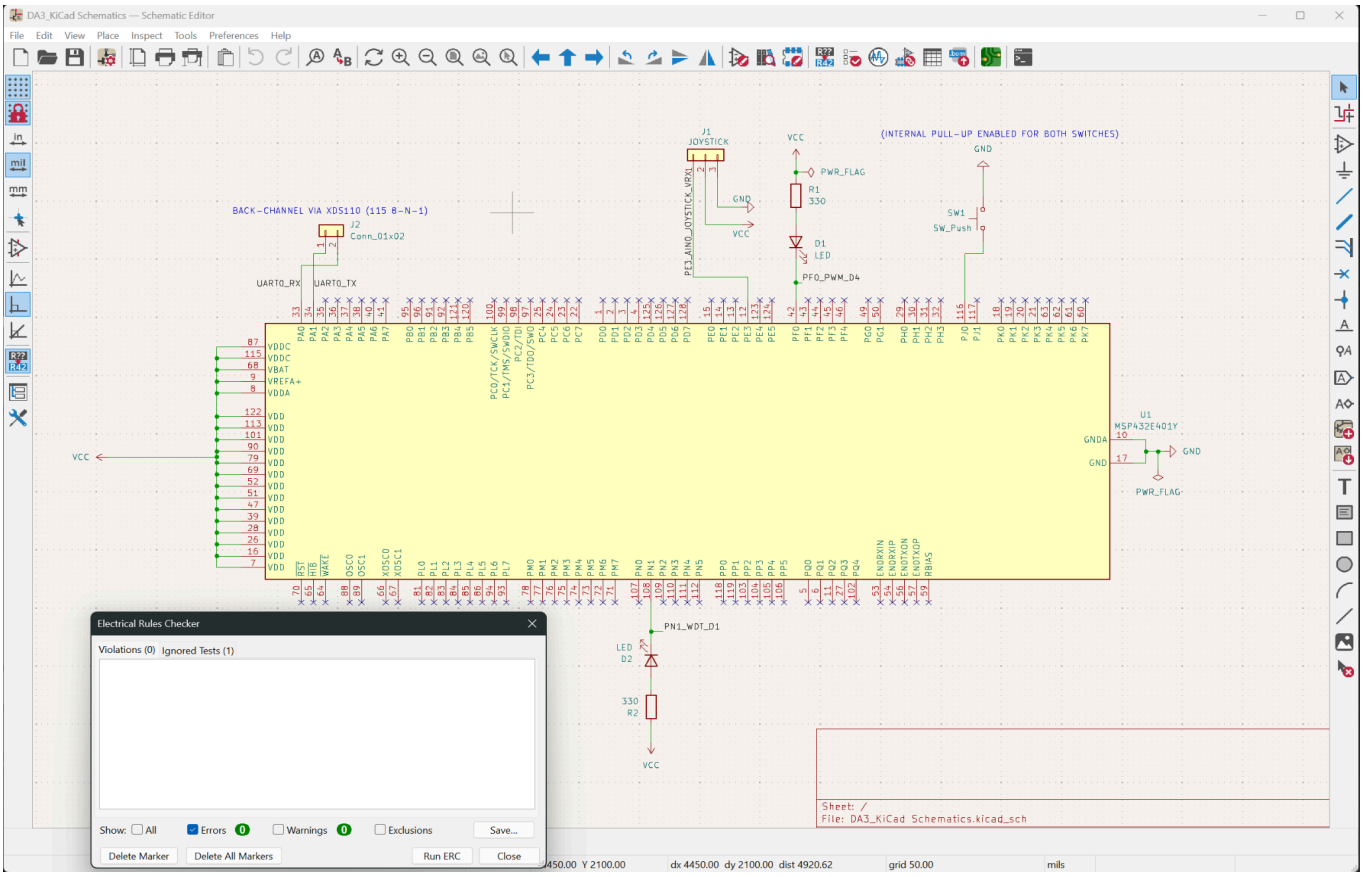
```
static void heartbeatClockFxn(UArg arg0)
{
    (void)arg0;
    /* Heartbeat: blink PF4 LED independently of other tasks */
    GPIO_toggle(CONFIG_GPIO_LED_0);
}
```

```
    /* Heartbeat Clock: 500 ms period */
    Clock_Params hbClkParams;
    Clock_Params_init(&hbClkParams);
    hbClkParams.period    = HEARTBEAT_PERIOD_MS / SYSTEM_TICK_MS;
    hbClkParams.startFlag = TRUE;
    Clock_construct(&heartbeatClockStruct,
                    heartbeatClockFxn,
                    HEARTBEAT_PERIOD_MS / SYSTEM_TICK_MS,
                    &hbClkParams);
```

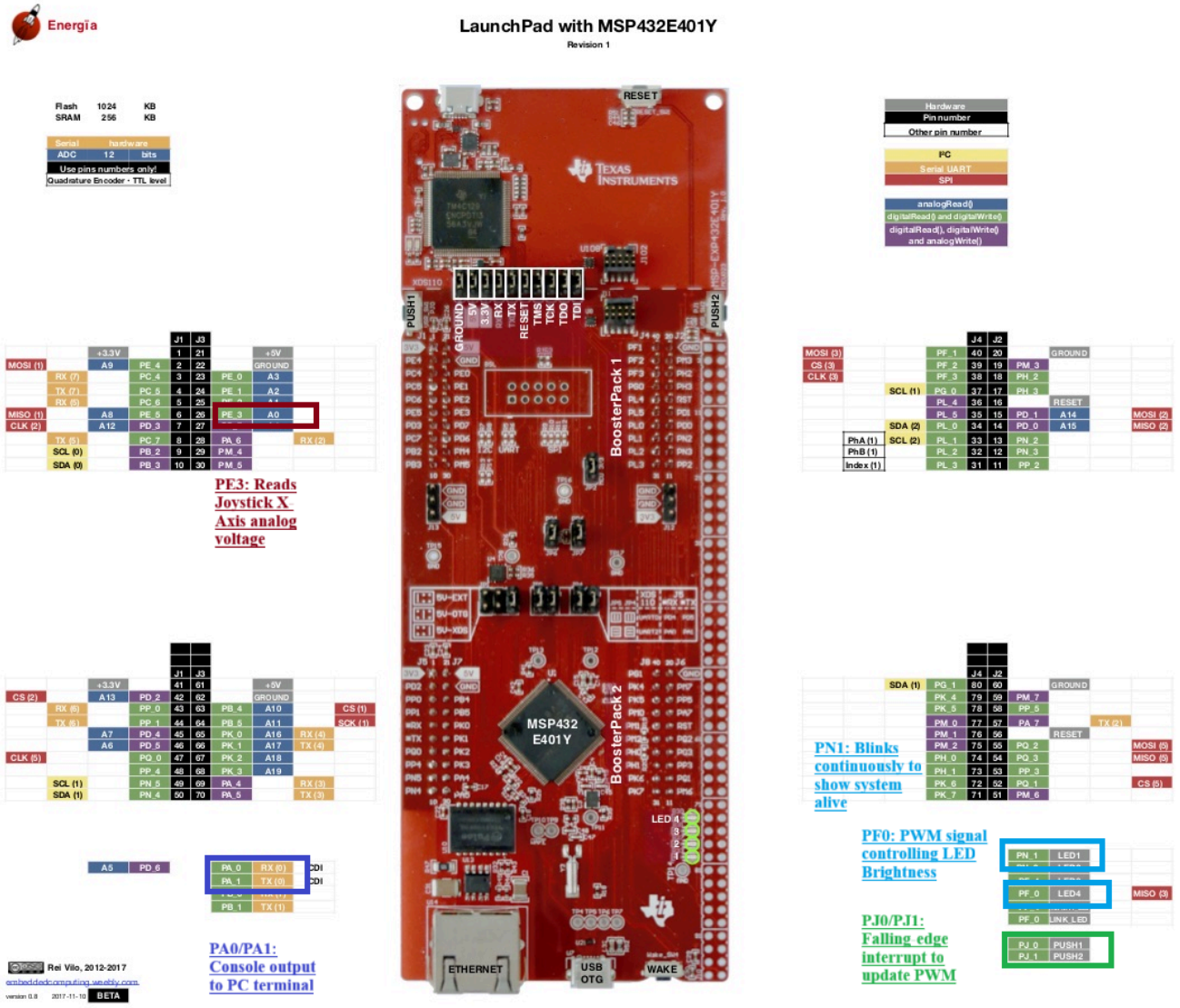
Description: In order to meet the "heartbeat" requirement, I created a dedicated Clock object that fires every 500 ms, independent of the ADC/UART/PWM scheduling. Its callback *heartbeatClockFxn* simply toggles *CONFIG_GPIO_LED_0* (PF4), producing a steady blink that shows the RTOS is running even when other tasks are blocked or busy. Because it is Clock-driven, this heartbeat does not interfere with the timing of ADC acquisitions, UART prints, or PWM updates.

2. Block diagram and/or Schematics showing the components, pins used, and interface. You can use KiCAD/Eagle/Altium to get the schematics. KiCAD Symbol libraries for TI uCs are @ https://kicad.github.io/symbols/MCU_Texas.html and https://www.snapeda.com/parts/CC1352P1F3RGZT/Texas%20Instruments/view-part/

KiCad Design Schematics (ERC Compliant):

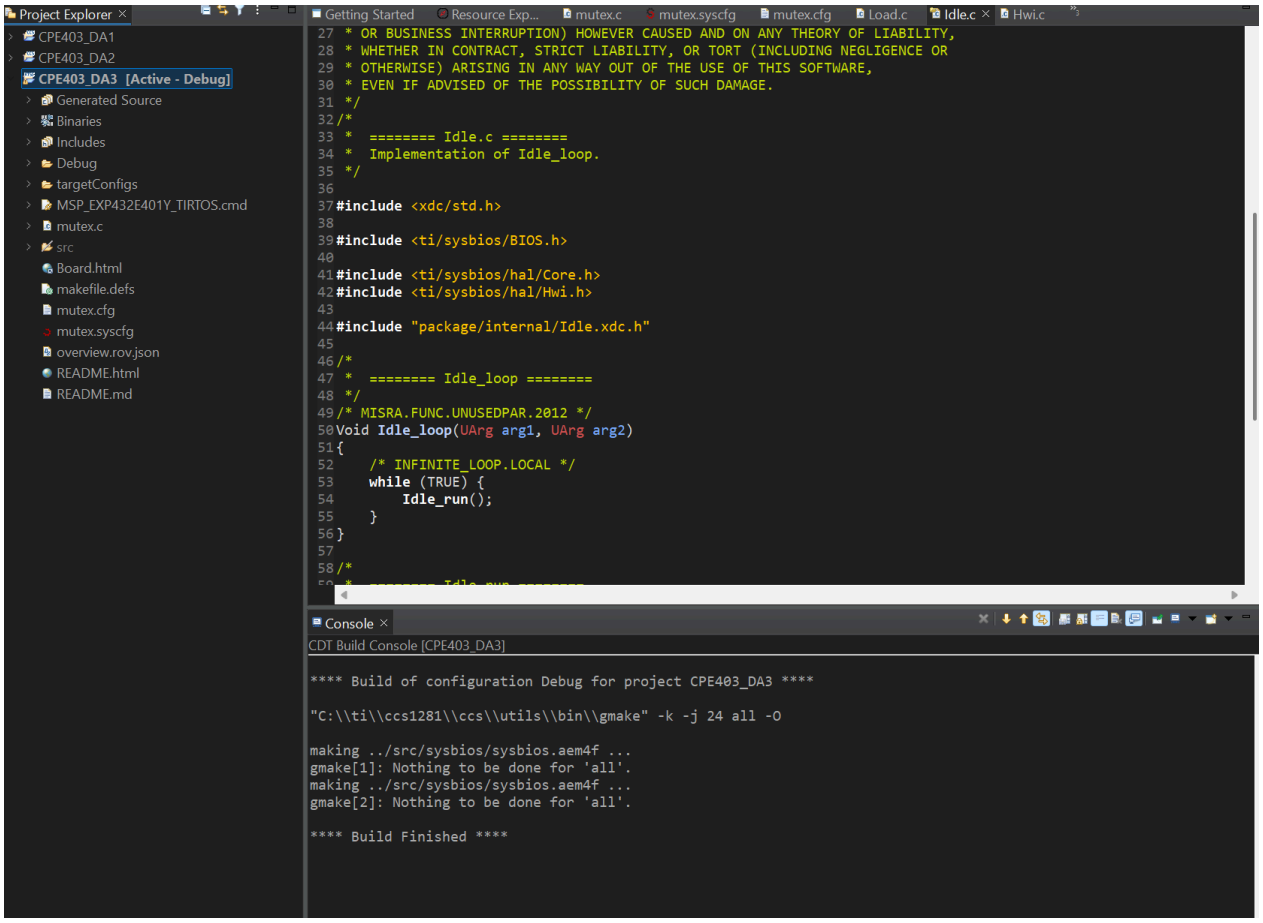


Block Diagram:

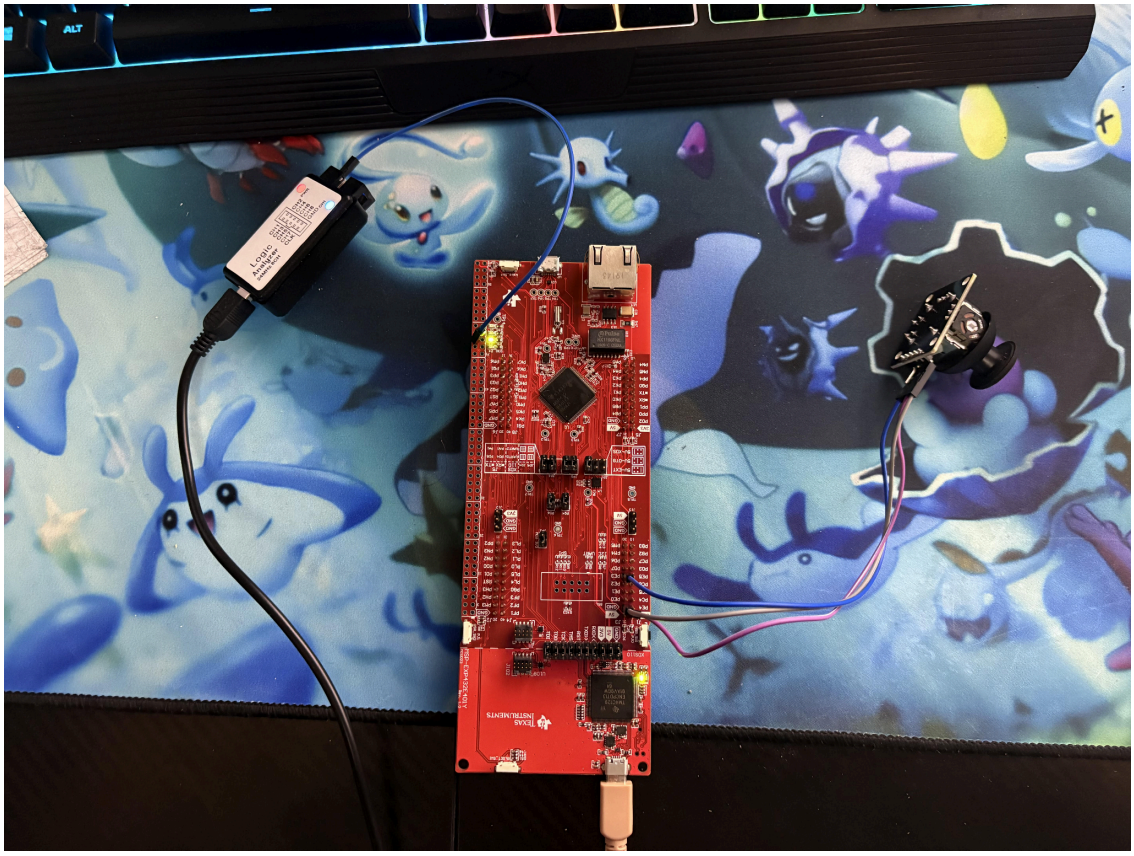


3. Screenshots of the IDE, physical setup, debugging process – Provide screenshot of successful compilation, screenshots of registers, variables, graphs, etc.

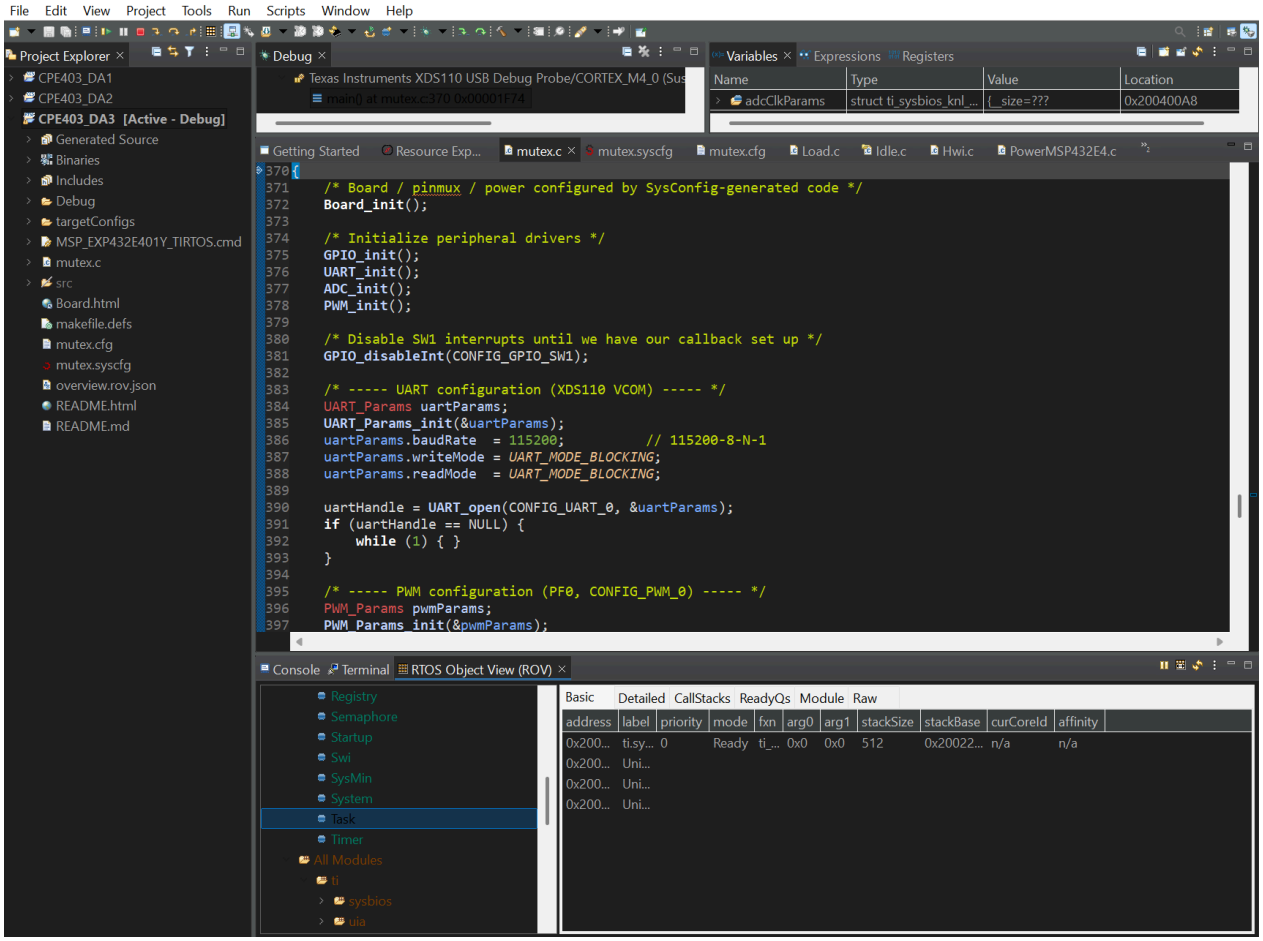
IDE Workspace:



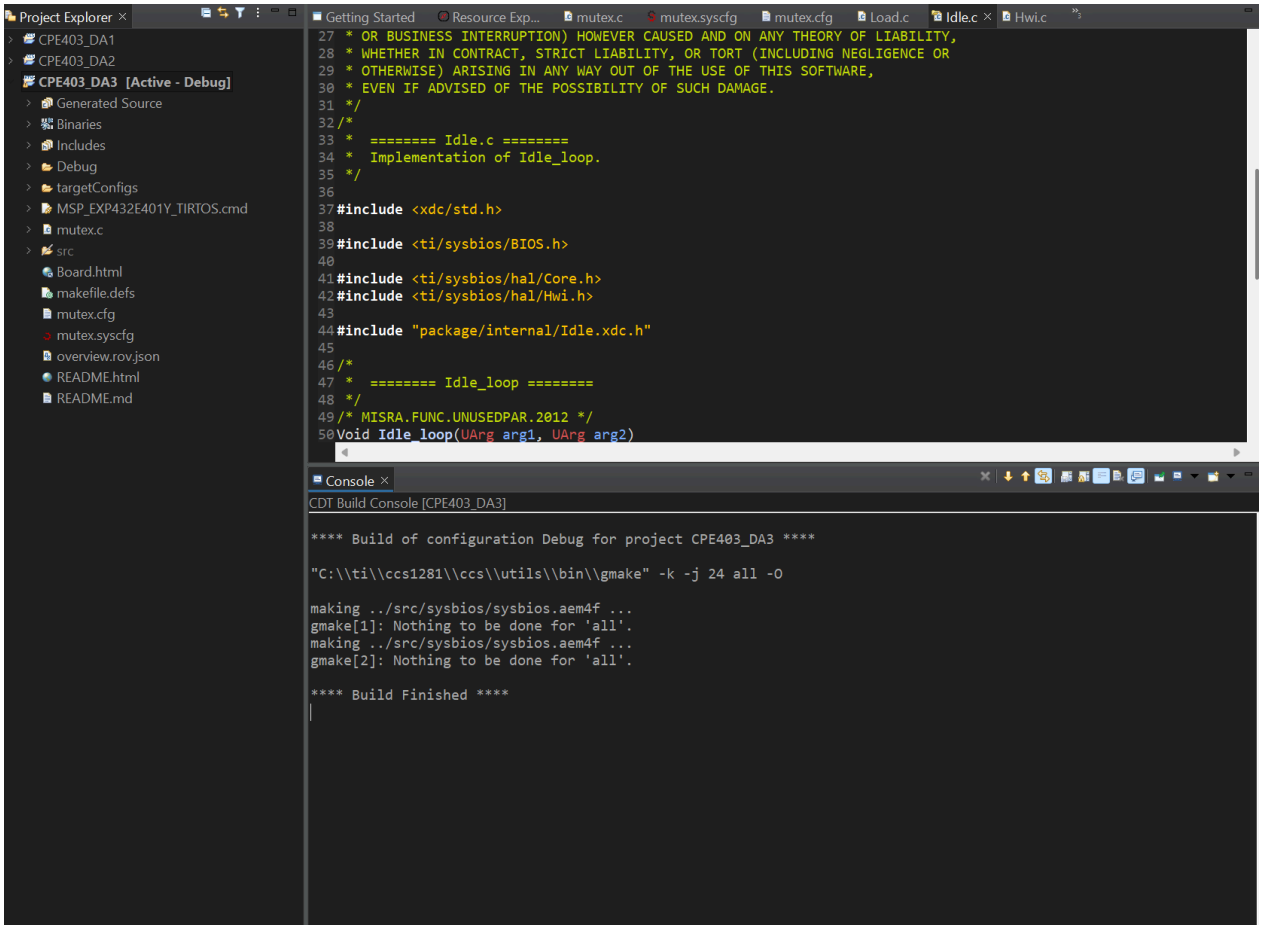
Physical Setup:



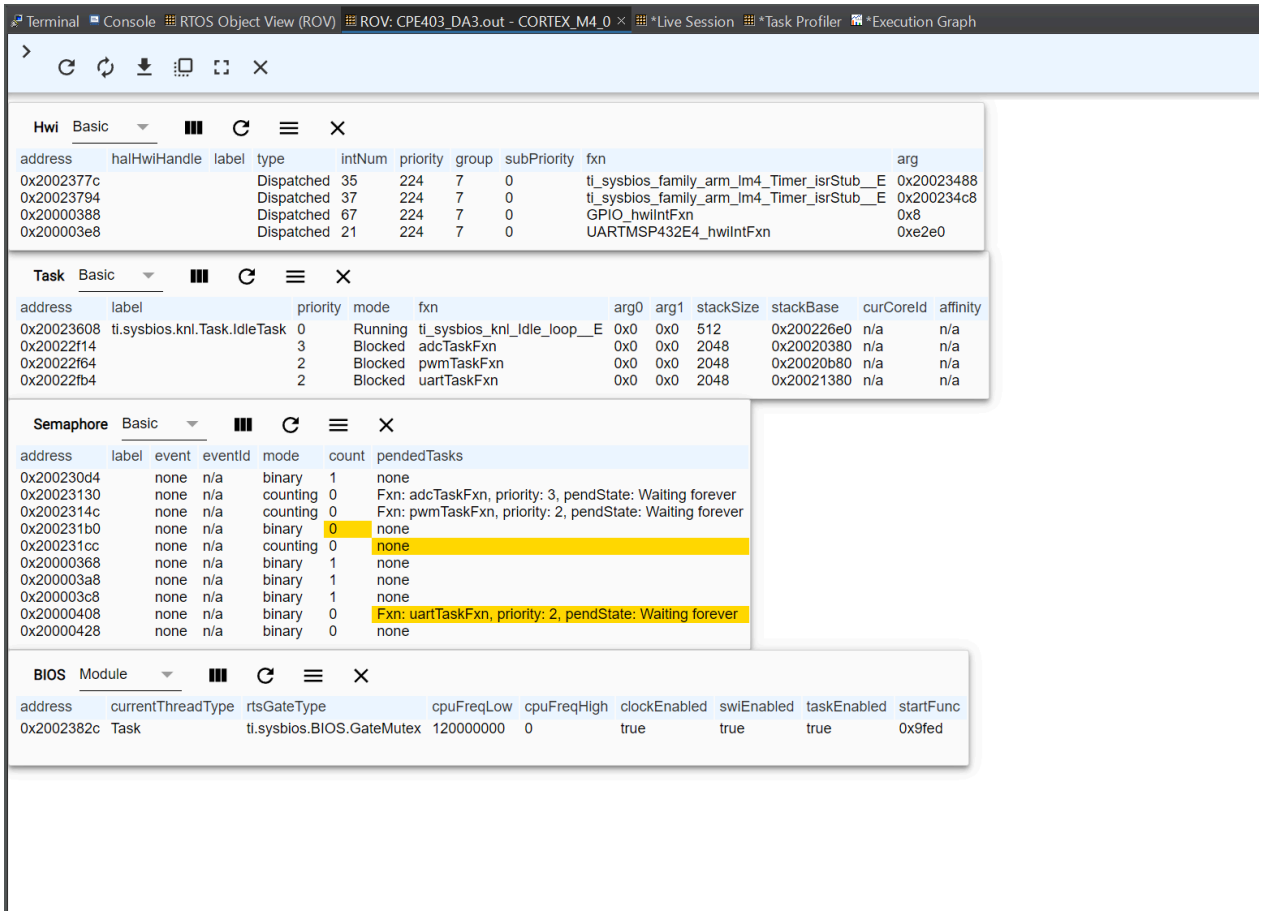
Debugging Process:



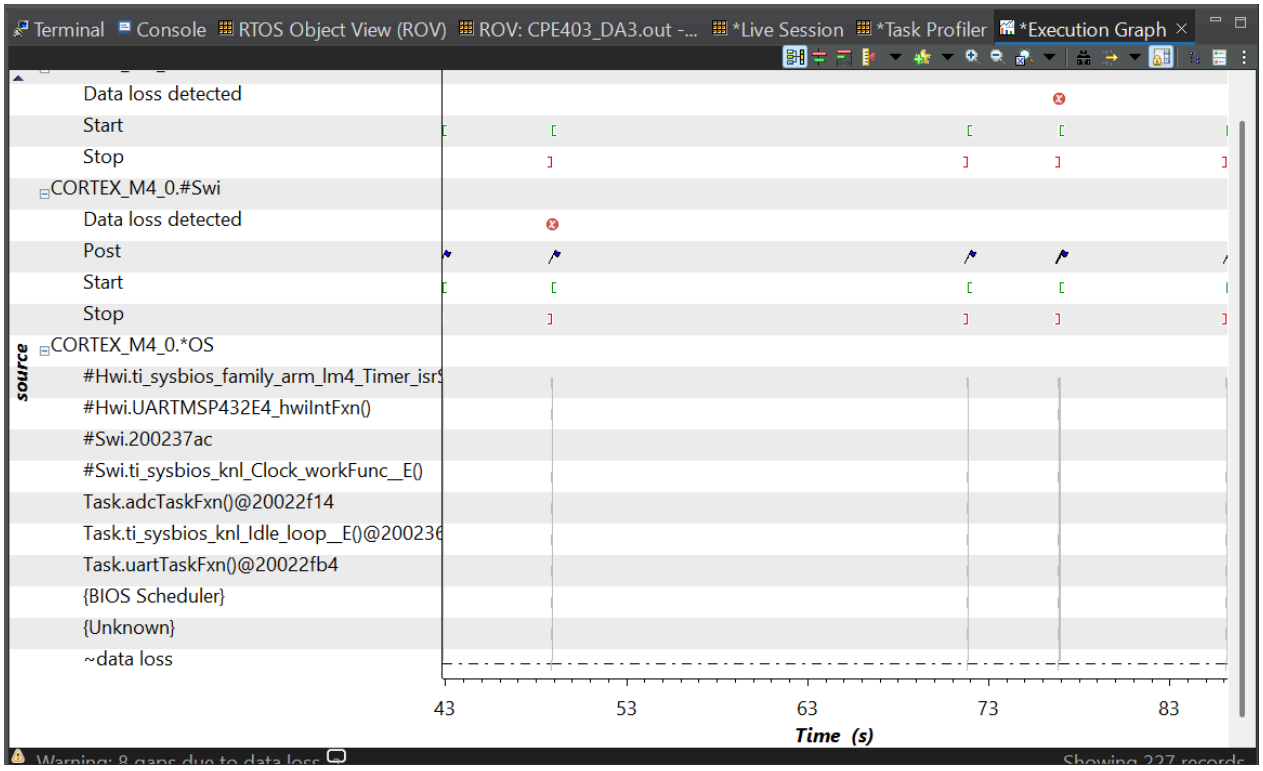
Successful Compilation:



RTOS Object Viewer:



Execution Graph:



4. Declaration

I understand the Student Academic Misconduct Policy - http://studentconduct.unlv.edu/misconduct/policy.html

"This assignment submission is my own, original work".
Joshua Martinez