

A03: Applying the μ C/OS-II RTOS

Lab Guide

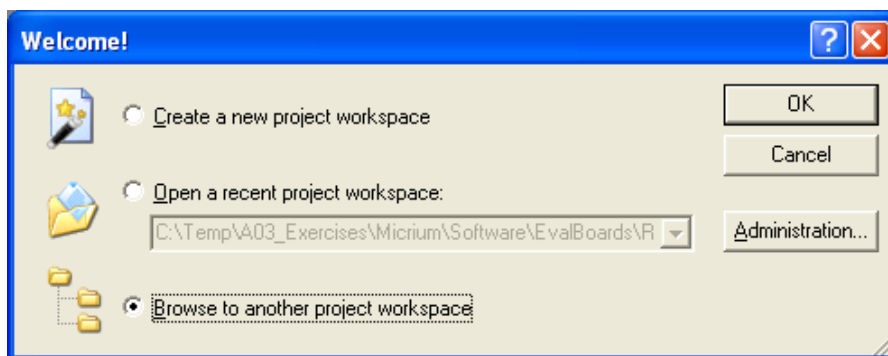
This document describes the three exercises that constitute the lab portion of “Applying the μ C/OS-II RTOS.” These brief but engaging projects, the objectives of which are listed below, are intended to complement this class’s lecture material. Accordingly, you should be able to complete the exercises simply by applying the concepts that the lectures introduce. In case you need to review any of these concepts, you can consult the class’s PowerPoint slides. For additional μ C/OS-II-related information, beyond what the slides contain, you can peruse the thorough documentation that accompanies the operating system, or even consult the source code itself.

Lab 1: Becoming Familiar with μ C/OS-II

Objectives: This exercise, which involves building and running a μ C/OS-II-based application, will introduce you to the tools and source code upon which all of this class’s coding assignments are based.

Procedure:

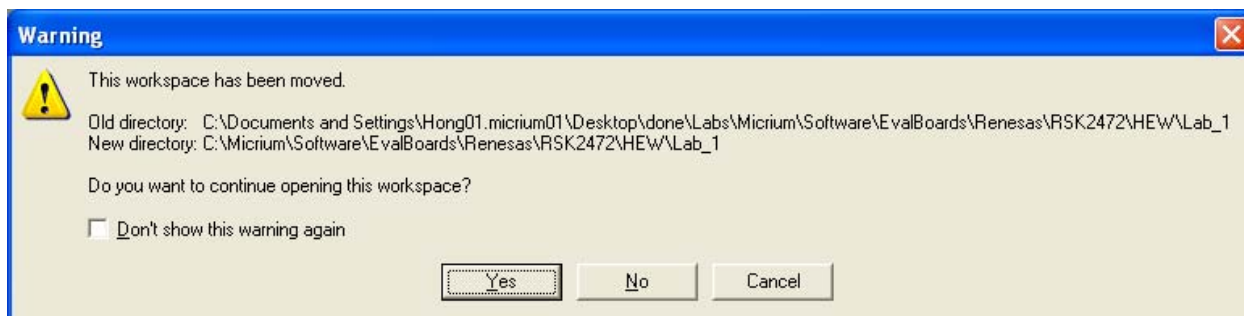
1. Throughout this class, you will be using an H8S/2472-based Renesas Starter Kit (RSK) board and an E10A debugger. If you have not already connected your E10A to both your RSK board and your PC, you should do so now. You should also make sure that power is being supplied to your board.
2. Once you have set up your hardware, you should start High-performance Embedded Workshop (HEW) by double-clicking the appropriate desktop icon. Alternatively, you could start HEW by selecting **All Programs > Renesas > High-performance Embedded Workshop > High-performance Embedded Workshop** from your PC’s Start menu.
3. The below dialog box will appear after you have started HEW.



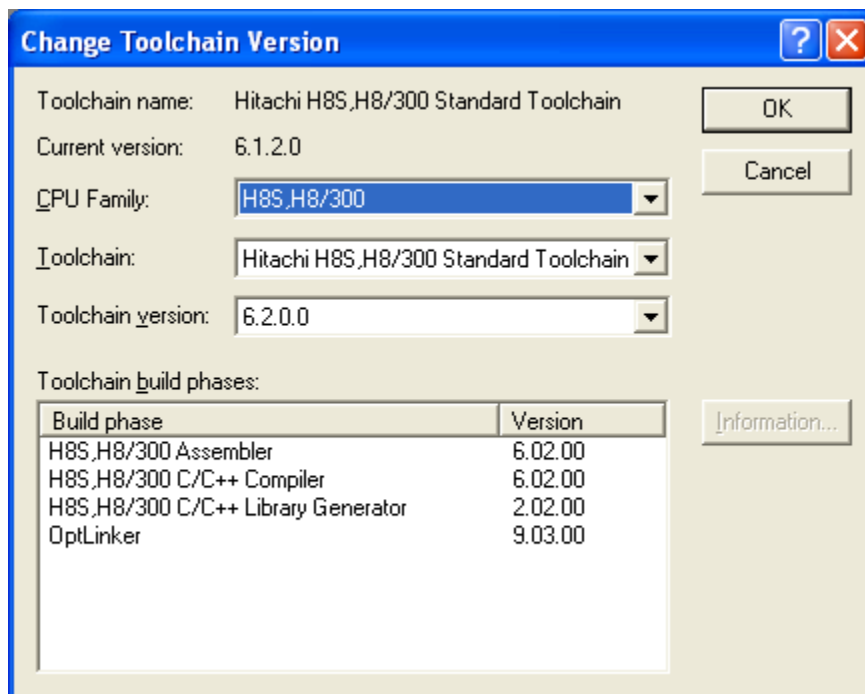
Within this box, you should select **Browse to another project workspace** and then click **OK**. Next, you should browse to the following folder, where you can find Lab 1's workspace:

Micrium\Software\EvalBoards\Renesas\RSK2472\HEW\Lab_1

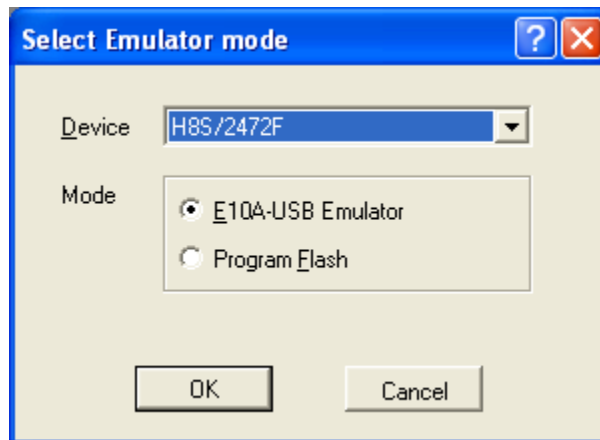
4. A warning message, indicating that **This workspace has been moved**, might be displayed after you attempt to open the above workspace. You should simply click the **Yes** button that is located at the bottom of this message.



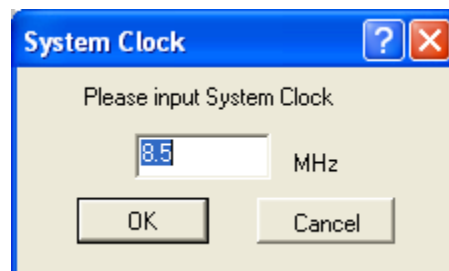
5. If you are presented with a dialog box entitled **Toolchain missing**, you should click **OK**. You should then use the below settings to change toolchain versions.



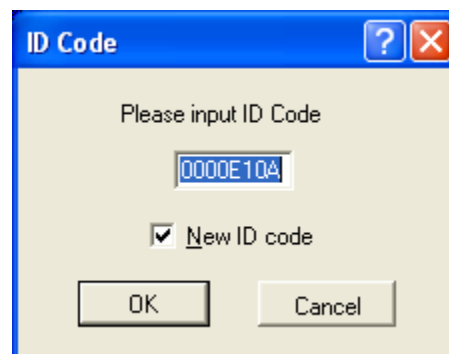
- When you are prompted to **Select Emulator mode**, you should choose the **Device** and **Mode** shown below.



- After the above selections have been made, a dialog box, indicating that HEW is **Connecting**, will be shown. You will then be asked to specify a clock frequency. You should enter a frequency of 8.5 MHz.

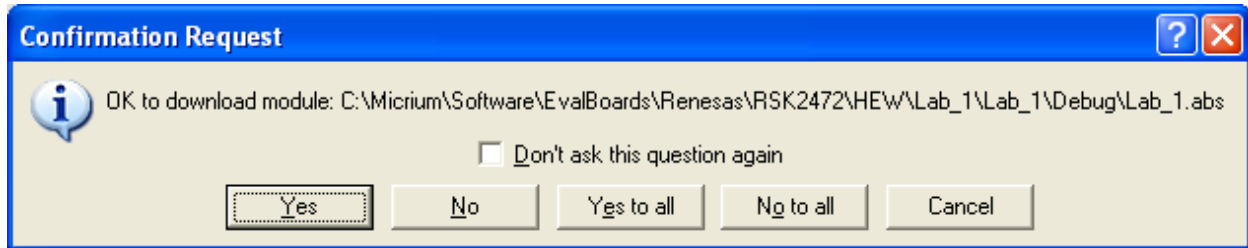


- While HEW is **Connecting** you will also be asked for an **ID Code**. The appropriate response for this request is shown below.



- Once all of your emulator settings have been established, you should attempt to build the sole project that Lab 1's workspace contains. To this end, you should either select **Build > Build All** from the menu bar or right-click the project's name (**Lab_1**) and then choose **Build > Build All**.

10. Assuming that the build process completed without any major problems, a dialog box entitled **Confirmation Request** will appear. You should simply click this dialog box's **Yes** button, thereby instructing HEW to download executable code to your board.



11. After the aforementioned downloading has taken place, you should click HEW's **Reset CPU** button, and then the **Go** button. At this point, one of your board's LEDs (**LED0**) should begin blinking.

Lab 2: Creating Tasks

Objectives: After completing this exercise, you will be capable of using $\mu\text{C}/\text{OS-II}$'s task-management routines, especially those that facilitate task creation. This exercise will also introduce you to `OSTimeDly()` and `OSTimeDlyHMSM()`, two of $\mu\text{C}/\text{OS-II}$'s time-management routines.

Procedure:

1. For this exercise, you will be adding a new task to an existing $\mu\text{C}/\text{OS-II}$ -based application. Although the application code that will serve as the foundation for this addition is essentially identical to that encompassed by Lab 1's workspace, there is actually a separate workspace for this exercise. This new workspace can be found in the following location:

Micrium\Software\EvalBoards\Renesas\RSK2472\HEW\Lab_2

2. Once you have opened Lab 2's workspace, you should add two new `#define` constants to *app_cfg.h*. One of these constants will indicate your new task's priority, while the other will represent the size of this task's stack. From the perspective of $\mu\text{C}/\text{OS-II}$, the name chosen for each of these constants is not overly important. The values that you assign to the constants, however, will help to determine whether or not your code ultimately functions properly. Initially, then, you should rely on the following values, each of which Micrium has tested for this exercise: a priority of 6 and a stack size of 384. After you have completed the exercise, you can experiment with other values.

3. Although the size of your stack should be defined in *app_cfg.h*, you will need to declare the stack itself in *app.c*. Other stacks, such as `AppTaskStartStk[]`, are already declared in this file, so you will have plenty of relevant example code at your disposal. As you add your declaration to *app.c*, you should keep in mind that, like both of the above `#define` constants, your stack can be named somewhat arbitrarily. Micrium, however, would recommend that you follow a well-defined convention when naming any such variables or constants.
4. In $\mu\text{C}/\text{OS-II}$, tasks are simply C functions. Thus, in order to implement your task, you will need to add a new function to *app.c*. This function, which can be given nearly any name of your choosing (although, again, the use of naming conventions is recommended), will be similar to `AppTaskStart()`, the first task that the code contained in *app.c* creates. However, your new task should be devoid of the initialization code that `AppTaskStart()` incorporates. In other words, your code will simply blink one of the RSK board's LEDs. Since `AppTaskStart()` already toggles **LED0**, you should focus on **LED1**. Additionally, whereas `AppTaskStart()` toggles its LED every 100 ms, your task should employ an interval of 50 ms. Before attempting to verify that your code correctly implements this functionality, you should proceed to the next step of this exercise.
5. For every task that your application incorporates, there must be a corresponding call to either `OSTaskCreate()` or `OSTaskCreateExt()`. Thus, on behalf of your newly written task, a call to one of these functions must be added to *app.c*. This new function call, which should invoke `OSTaskCreateExt()`, rather than the older `OSTaskCreate()`, should be placed in `AppTaskCreate()`, a routine that `AppTaskStart()` uses solely to create additional tasks. If you would like to consult an example of such a function call before writing your code, you should simply review the existing contents of `AppTaskCreate()`. Once your new call has been added to this function, you should build and run your application and confirm that your code properly blinks **LED1**.

Lab 3: Interrupt Handlers

Objective: This exercise will help you to understand how $\mu\text{C}/\text{OS-II}$ -based applications typically handle interrupts. Since peripheral devices figure heavily into interrupt handling, this exercise will also prepare you to write application code that, in addition to incorporating the operating system's API functions, directly invokes hardware-specific routines.

Procedure:

1. This exercise, much like its predecessor, involves augmenting an existing μ C/OS-II-based application. However, contrary to the last project, in which you wrote a new task, this exercise will require you to prepare an interrupt handler. The workspace for this exercise encompasses the code to which you will be adding your interrupt handler. This workspace is located in the following folder:

Micrium\Software\EvalBoards\Renesas\RSK2472\HEW\Lab_3

2. Responding to interrupts typically involves saving and loading CPU registers, so your interrupt handler, which will be responsible for retrieving data from the H8S/2472's A/D converter, will need to be written in assembly language. Since the only assembly language file incorporated by Lab 3's workspace is *os_cpu_a.src* (which, as part of the μ C/OS-II port for the H8S family of MCUs, should not, under most circumstances, be modified), you will actually need to add a new file to this workspace. You can give this file almost any name that you choose, as long as you use the appropriate *.src* extension.
3. Before you actually write your interrupt handler, you will need to add a handful of assembler directives to your new file. Additionally, you should update your file with the definitions of two macro functions, `PUSHALL` and `POPALL`. These functions, which are used by most of the interrupt handlers that Micrium's engineers have prepared for the H8S/2472, will afford you with a convenient means of pushing and popping CPU registers. The definitions of `PUSHALL` and `POPALL` are shown on the next page of this document, as are the aforementioned assembler directives. All of this text should simply be added to the top of your new assembly language file.

```

;*****
;
;                               Cross references
;*****
;
    .EXPORT    _ADCISR

    .IMPORT    _OSIntExit
    .IMPORT    _OSTCBCur
    .IMPORT    _OSIntNesting
    .IMPORT    _ADCClr

;*****
;                               MACROS
;
; Note(s): 1) Save 32-bit registers in case they were used in the application
; code.
;*****
;

    .MACRO     PUSHALL
    PUSH.L    ER0
    PUSH.L    ER1
    PUSH.L    ER2
    PUSH.L    ER3
    PUSH.L    ER4
    PUSH.L    ER5
    PUSH.L    ER6
    .ENDM

    .MACRO     POPALL
    POP.L     ER6
    POP.L     ER5
    POP.L     ER4
    POP.L     ER3
    POP.L     ER2
    POP.L     ER1
    POP.L     ER0
    .ENDM

```

4. In accordance with the assembler directives that you added to your file, your new interrupt handler must be named `ADCISR` (to which an underscore will need to be added in the label preceding your declaration). You should model your code after $\mu\text{C}/\text{OS-II}$'s tick interrupt handler, `OSTickISR()`. As one of multiple assembly language routines that the $\mu\text{C}/\text{OS-II}$ H8S port incorporates, `OSTickISR()` resides in `os_cpu_a.src`, the file mentioned in step 2 of this exercise. For the most part, your interrupt handler should be identical to this function. Unlike `OSTickISR()`, though, your code will process A/D converter interrupts, rather than timer interrupts. Thus, instead of invoking `TickClr()` and `OSTimeTick()`, which are the two C functions that `OSTickISR()` uses to clear tick interrupts, your routine will invoke `ADCClr()`, a routine that Micrium's engineers have already written for you. If you would like to review this routine, you can find it in `bsp.c`.

5. If your new function, `ADCISR()`, is to effectively service the interrupts that the A/D converter produces, you will need to associate this routine with the proper interrupt vector, number 28. Fortunately, in HEW, you can easily associate any interrupt handler with the appropriate vector via `#pragma` directives. Thus, in order for `ADCISR()` to be established as the interrupt handler for the A/D converter, you will not need to write any additional assembly language routines. Instead, you will only need to make a few modifications to a C file, `bsp_isr.c`. The modicum of code that should be added to this file is shown below.

```

/*
*****
*
*                               ADC ISR
*
* Description      : This function is part of the ISR for the ADC.
* Argument        : None
* Returns         : None
*****
*/

#pragma noregsave INT_ADI
#pragma interrupt (INT_ADI(vect=28))

void INT_ADI (void)
{
    ADCISR();
}

```

6. Once you have finished updating `bsp_isr.c`, you should turn your attention to `app.c`. Within this file, you will be writing a new task that will have a now familiar responsibility: blinking an LED. However, unlike the last task that you wrote, which toggled **LED1** at a predetermined rate, your new task will blink its LED (**LED2**) at a rate determined by the A/D converter. In order to obtain the latest reading of the A/D converter, the new task will need to rely on `AdcDly`, a global variable that `ADCCLR()` (the routine invoked by your interrupt handler) regularly updates. As indicated below, your task should not pass `AdcDly` directly to `OSTimeDlyHMSM()`; this variable's value must be scaled first.

```

Delay (seconds)          = AdcDly / 1000
Delay (milliseconds)    = AdcDly % 1000

```

7. In order to determine whether both your new task and your interrupt handler are functioning correctly, you will need to build your code and then download it to your RSK board. Once the code is running on your board, you should make sure that **LED0**, **LED1**, and **LED2** are continuously blinking. Furthermore, you should confirm that, by tweaking the board's potentiometer, you are able to effect variations in **LED2**'s frequency.