ECE 3849 D2018
Real-Time Embedded Systems
Lab 2: RTOS, Spectrum Analyzer

In this lab you will port the Lab 1 application, 1 Msps digital oscilloscope, to an RTOS (TI-RTOS). You will also add an FFT mode that turns your system into a simple spectrum analyzer. Have the course staff fill in your scores in the following signoff sheet before the signoff deadline. This lab requires a report. See Canvas for due dates. When submitting the source code, first "Clean" the project, and remove all files from the Debug subfolder to reduce the .zip file size.

Step	Max pts	Score
ADC ISR converted into an Hwi object, meets all deadlines.	10	
Button ISR converted into a **Task**, signaled by a Clock object. Button IDs posted to a Mailbox object.	10	
Waveform and Display Tasks implemented. Oscilloscope is drawing waveforms.	12	
User commands from the button Mailbox are processed and control the oscilloscope settings.	8	
Spectrum mode (FFT) implemented. Grid and scale text should be appropriate to the FFT mode. See assignment.	20	
Button Task timing measurements (extra credit; see assignment)	3	
Trigger search timing measurements (extra credit; see assignment)	3	
Question about this lab answered.	5	Student1: Student2:
Report and source code (separate due date)	35	
Total points	100 + 6	Student1: Student2:

Student 1: ______________________________ Student 2: ______________________________

Grader: ________________________________ Date: _______________ Mailbox: ___________

Learning objectives

- Port an existing real-time application to an RTOS
- Use fundamental RTOS objects: Task, Hwi, Clock, Semaphore, Mailbox
- Deal with shared data and other inter-task communication
- Run a computationally intensive task without slowing down the user interface

Assignment

Port all the Lab 1 oscilloscope functionality to TI-RTOS, breaking the project up into multiple Task, Hwi and Clock threads. You do not need to port the Lab 1 CPU load or extra credit code. Required differences in implementation from Lab 1:

- All code must run in TI-RTOS threads: Task, Swi or Hwi.
- Convert the ADC ISR into an Hwi, but preserve its low latency by configuring it as a “zero-latency interrupt”
- Button handling
    - Schedule the periodic button scanning using the Clock module. The Clock function should signal the Button Task using a Semaphore.
    - Create the Button Task (highest-priority) that scans the buttons and places button IDs into a Mailbox object.
    - Create the User Input Task (mid-priority) that processes the user input that it receives through the Button Mailbox. If no buttons are being pressed, this Task remains blocked waiting on the Mailbox. When the user makes changes to the settings, this task signals the Display Task.
- Create the Waveform Task (high-priority) that searches for the trigger and copies the triggered waveform into a waveform buffer (a shared global array). It then signals the Processing Task.
- Create the Processing Task (lowest-priority) that scales the captured waveform in oscilloscope mode, or applies FFT to the waveform in spectrum mode. (Use a separate buffer for the processed waveform.) It then signals the Display Task and the Waveform Task.
- Create the Display Task (low-priority) that draws a complete frame (grid, settings and waveform) to the LCD screen.

An additional feature compared to Lab 1 is **spectrum (FFT) mode**. Switch between the FFT and oscilloscope modes using one of the buttons. In FFT mode, some of the existing Task functionality should change:

- The Waveform Task copies the 1024 newest ADC samples without searching for trigger.
- The Processing Task performs a 1024-point FFT on the captured samples and converts the lowest 128 frequency bins to an integer 1 dB/pixel scale for display. A floating point FFT library is provided to you together with the assignment. This Task is meant to take significant time to execute to test the prioritization and preemption of RTOS Tasks.

- The Display Task should show the frequency and dB scales in FFT mode (instead of time and voltage). These scales do not need to be adjustable. The frequency grid should start at x = 0 (zero frequency).

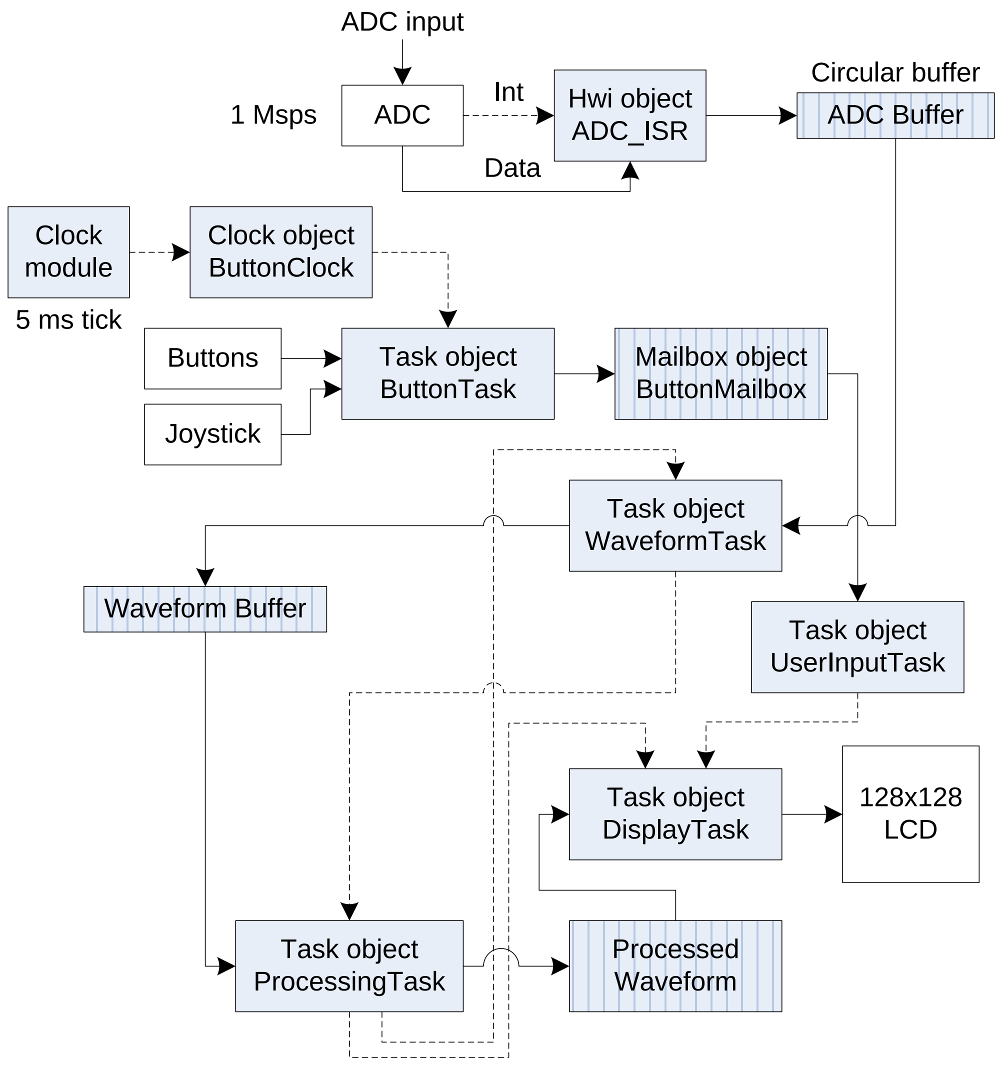
For **extra credit**, measure the real-time parameters of the Button Task and the trigger search code in Waveform Task, and fill the table below. For trigger search, measure the max time your Waveform Task takes to search for the trigger and copy the waveform from the ADC Buffer. Determine the relative deadline for each of these tasks.

Task	Latency	Response Time	Relative Deadline
Button Task			
Trigger search and waveform copy in Waveform Task	N/A		

Write a detailed lab report documenting your lab implementation. A lab report guide is available in the Assignment/Labs section of the course website.

Recommended Implementation

The following figure outlines the recommended structure of the Lab 1 digital oscilloscope ported to TI-RTOS.



Software threads and modules are represented by smooth shaded boxes. If an OS object is to be used, it is indicated in the box. Important shared data and inter-task communication objects use textured shading. Clear boxes indicate hardware. Two types of connections are distinguished: solid lines = data flow, dotted lines = signaling/synchronization. Threads are arranged vertically by priority.

Step 1: TI-RTOS Project

While it is possible to start with one of the pre-configured TI-RTOS example projects, you would have to remove a lot of features we will not use. The main feature we will ignore is the set of device drivers that comes with TI-RTOS. For our purposes, these device drivers are just a different interface to TivaWare, and are also incomplete. Instead, we will simply use TivaWare.

Rather than starting from one of the TI-RTOS examples, download the **ece3849_lab2_starter** project from the Canvas Lab 2 area. Rename your project to "ece3849d18_lab2_username1_username2," substituting your usernames.

You now have an almost empty project linked with TI-RTOS. The primary files you will modify are main.c and **rtos.cfg**. Opening rtos.cfg should bring up the TI-RTOS configuration GUI. If rtos.cfg shows up as a script instead, close it, then right-click it and select Open With → XGCONF. The basic features of rtos.cfg are explained as you navigate the different objects in it. You can click the ⓘ icon while viewing any part of rtos.cfg GUI to bring up detailed documentation on every object you encounter. All the objects controlled by the rtos.cfg GUI are listed in the Outline window. You can add more TI-RTOS modules to your system through the Available Products window (if not visible, open the menu View → Other... and type in "Available Products"). The starter project features only a single Task task0 to demonstrate its configuration in rtos.cfg and its code in main.c.

Perform hardware initialization with **interrupts** globally **disabled** in main() before starting the OS. It is recommended to globally **enable interrupts** in one of the **Tasks**, to prevent ISRs from running before the OS is initialized. The OS does not globally enable interrupts during initialization (despite what the comments seem to say). Note that TI-RTOS configures the **FPU**, **clock generator**, **interrupt controller** and (periodic/one-shot) **timers** on its own. You should **not** initialize these peripherals manually.

Copy your Lab 1 source (.c and .h) files into your new Lab 2 project. **Do not copy** tm4c1294ncpdt_startup_ccs.c or main.c. Copy the text of your Lab 1 main.c into the Lab 2 main.c, and comment out that old code. You will gradually move pieces of that code into their final locations and then uncomment them.

Step 2: ADC Hwi

The ADC ISR and setup code should remain nearly unchanged from Lab 1. The only difference is that the ADC ISR must be configured as a TI-RTOS Hwi object. Use the **M3 specific Hwi module**. The Hwi module sets up the interrupt vector table and priorities, so **none of the interrupt controller setup code should remain** from Lab 1 (no driver functions starting with **Int**, except for IntMasterDisable() and IntMasterEnable()).

In the Hwi Module settings, specify "Priority threshold for Hwi_disable()" = 32. This leaves all higher priority ISRs enabled even while TI-RTOS runs the scheduler. Make the ADC Hwi priority 0 (highest priority on the interrupt controller). This should permit this ISR to execute every 1 μs without interference from the critical sections in the OS, at the expense of forbidding any OS calls from the ISR.

When configuring the ADC Hwi object (Instance), you will need to specify the interrupt number. Look it up in the TM4C1294NCPDT datasheet in Table 2-9 under **Vector Number** (not Interrupt Number – yes, confusing).

Step 3: Button Scanning

You will configure a **Clock object** (Instance) to schedule periodic button scanning. First, initialize the Clock Module to a 5 ms clock tick period and "Internally configure a Timer to periodically call Clock_tick()." Specify a different "Timer Id" if you would like to use Timer0 for something else. Note that you should remove your Lab 0/1 button timer initialization code. The RTOS handles this now.

Add a Clock Instance. The Clock function should signal the Button Task using a Semaphore. This Task should scan the buttons, and if a press is detected, posts the button ID to a Mailbox for further processing by a lower-priority Task. Make sure the Clock function and the Button Task no longer access any timer hardware (this is now the RTOS's responsibility). The Clock Instance requires an initial timeout of at least 1 tick. Check the box to start the Clock Instance at boot time.

Add the **Button Task** (highest-priority) under Task → Instance (or the right-click context menu for Task in the Outline window). Add a **Semaphore** for the button Clock function to signal the Button Task. Similarly, add a **Mailbox** where the Button Task posts button IDs.

Step 4: Breaking the Lab 1 main() into Tasks

The **User Input Task** (medium priority) should pend on the button Mailbox (blocking call). When it receives a button ID, it should modify the oscilloscope settings and signal the Display Task (using a Semaphore) to update the LCD screen. Be careful here as the oscilloscope settings are shared data. Make sure no serious shared data bugs arise.

The **Display Task** (low priority) should block on a Semaphore waiting for other tasks to signal a screen update. When signaled, it should draw one frame including the settings and the waveform (output of the Processing Task) to the LCD, then block again. Be careful accessing the settings and the processed waveform buffer, as they are shared data.

The **Waveform Task** (high priority) should also block on a Semaphore. When signaled, it should search for trigger in the ADC buffer, copy the triggered waveform into the waveform buffer, signal the Processing Task, and block again. This task is high priority to make sure the trigger search completes before the ADC overwrites the buffer. Protect access to the shared waveform buffer, if necessary.

The **Processing Task** (lowest priority) should block on a Semaphore. When signaled, it should scale the captured waveform for display, but not draw it. Instead, store the processed waveform in another global buffer. Then, signal the Display Task to actually draw this waveform. Finally, signal the Waveform Task to capture another waveform. The Processing Task may seem redundant, but it does a lot more processing in spectrum mode. This time-consuming processing should be done at a lower priority than the user interface (User Input Task & Display Task).

You may need to increase the **stack** size for the Processing and Display Tasks. You can monitor the Task max stack usage in the Runtime Object View (see the end of this document).

Step 5: Spectrum (FFT) Mode

Pressing one of the user buttons should switch your oscilloscope into **spectrum mode**. Pressing the same button again should switch back oscilloscope mode. In spectrum mode, some of the task functionality should change:

- The Waveform Task should capture the newest 1024 samples without searching for trigger.
- The Processing task should use the Kiss FFT package to compute the 1024 point FFT of the captured waveform in single precision floating point. It should then convert the lowest 128 frequency bins of the complex spectrum to magnitude in dB and save them into the processed waveform buffer, scaled at 1 dB/pixel for display.
- The Display Task should print the frequency and dB scales rather than the time and voltage scales. It should also modify the grid, such that the first vertical grid line is at x = 0, corresponding to zero frequency. Keep the grid spacing 20 pixels in both directions. The waveform drawing code should remain unchanged (rely on the Processing Task to prepare the waveform for display), aside from the color.

The Kiss FFT package is relatively easy to run. You only need to copy into your project the .c and .h files from the **root kiss_fft130 folder**, not subfolders. Read the README and the comments in kiss_fft.h. The only annoying issue is how to allocate the Kiss FFT cfg/state buffer. The following example initializes the Kiss FFT library for 1024 samples (adjustable) and computes one FFT of a pre-programmed sine wave.

```
#include <math.h>
#include "kiss_fft.h"
#include "_kiss_fft_guts.h"

#define PI 3.14159265358979f
#define NFFT 1024         // FFT length
#define KISS_FFT_CFG_SIZE (sizeof(struct kiss_fft_state)+sizeof(kiss_fft_cpx)*(NFFT-1))
```

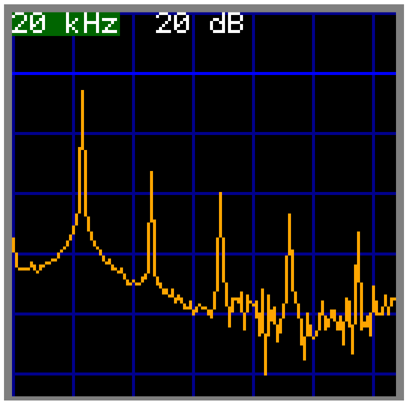
```
static char kiss_fft_cfg_buffer[KISS_FFT_CFG_SIZE]; // Kiss FFT config memory
size_t buffer_size = KISS_FFT_CFG_SIZE;
kiss_fft_cfg cfg;                          // Kiss FFT config
static kiss_fft_cpx in[NFFT], out[NFFT]; // complex waveform and spectrum buffers
int i;

cfg = kiss_fft_alloc(NFFT, 0, kiss_fft_cfg_buffer, &buffer_size); // init Kiss FFT
```

```
for (i = 0; i < NFFT; i++) {    // generate an input waveform
  in[i].r = sinf(20*PI*i/NFFT); // real part of waveform
  in[i].i = 0;                  // imaginary part of waveform
}

kiss_fft(cfg, in, out);         // compute FFT

// convert first 128 bins of out[] to dB for display
```

The three sections are includes/defines, initialization (place this code before the infinite loop of the Processing Task), and computation (in the Processing Task's infinite loop).

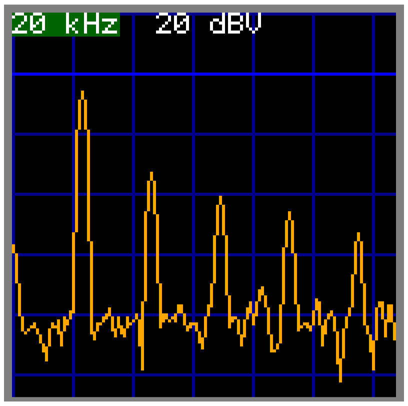
The input sinusoid has a period of 102.4 samples (1/10 of the total number of samples). The output (you can verify in MATLAB) is mostly zeros, except `out[10].i = -512` and `out[1014].i = 512`. When done testing, convert the waveform generator code to copying from the Waveform Buffer.

Convert the spectrum to dB using the `log10f()` function. You may need to add an offset to move the spectrum into the y-coordinate range of the display. The dB scale does not need to be calibrated to any specific voltage level, but the spectrum peaks and the noise floor should be visible on the screen. See the screen capture below for an example of what the spectrum analyzer output should approximately look like. (Performance note: The Cortex-M4F supports only single-precision floating point natively. Floating point calculations in C default to double precision. To enforce single precision (`float`), use the "f" suffix on numbers and math functions. You may also want to change "Floating point mode" to "relaxed" under Project Properties → Build → ARM Compiler → Optimization to convert the type `double` to mean `float`.)



Optionally, you may window your time-domain waveform before the FFT to reduce spectral leakage (the skirt-like signal drop-off around spectral peaks). Multiply your 1024 time-domain samples by the corresponding window function samples given to you below. FFT the resulting waveform. The spectrum should look like that in the following screen capture.

```
static float w[NFFT]; // window function
for (i = 0; i < NFFT; i++) {
    // Blackman window
    w[i] = 0.42f
           - 0.5f * cosf(2*PI*i/(NFFT-1))
           + 0.08f * cosf(4*PI*i/(NFFT-1));
}
```



Extra Credit: Real-time Measurements

We have been carefully monitoring ADC1 for missed deadlines. Now do the same for the remaining two real-time deadlines in this system: button scanning and trigger search.

For button scanning, measure the max latency and response time of the Button Task. You may use the **ece3849_rtos_latency** CCS project on the course website as a reference. The RTOS Clock module uses a hardware timer to schedule system tick interrupts. Read the timer count to determine the time since the last interrupt. Determine if a deadline was missed by checking the count of the Semaphore used to signal the Button Task. Exercise your code by pressing buttons.

For trigger search, measure the max time your Waveform Task takes to search for the trigger and copy the waveform from the ADC Buffer in Oscilloscope mode. You may use the TI-RTOS Timestamp module for this measurement (need to add it in rtos.cfg). Review the Lab 1 assignment for an explanation of the real-time requirements for trigger search. Note that the worst case trigger search time is when the trigger is not found. To force this condition, disconnect your AIN3 input.

Fill in the following table to show to the grader. Use actual time units, not CPU clock cycles. Also, be ready to explain your code for performing these measurements. Experiment with how the compiler optimization settings affect these measurements.

Task	Latency	Response Time	Relative Deadline
Button Task			
Trigger search and waveform copy in Waveform Task	N/A		

Debugging Features of the RTOS

When debugging your software running on an RTOS, it helps to be able to examine the state of all RTOS objects: which Tasks are blocked, Semaphore counts and waiting lists, etc. TI-RTOS provides a specialized watch window for RTOS objects in CCS. Navigate the menu to Tools → Runtime Object View, then click Connect. To find out what is blocking your Tasks, you will need to examine the Semaphores and Mailboxes, not just the Tasks. Under Detailed information for a Task you see what it is blocked on, but the blocking object is identified only by its address, not name.