ECE 3849 B2023
Real-Time Embedded Systems
Lab 4: Spectrum Analyzer and DMA

In this lab, a spectrum analyzer mode will be added to the oscilloscope, and ADC I/O will be optimized with the use of DMA.

To complete all labs in this course, you will need to submit your source for that lab, and demonstrate the lab in operation to one of the course staff (a TA or tutor). You will be graded on the completion of the steps in this lab document. After you have received a signoff, follow the instructions at the end of the document to submit your source code.

This lab is due on **Wednesday, December 6**. Late assignments will receive a 20% penalty.

Step	**Max Points**	**Score**
Spectrum (FFT) mode implemented	20	
Grid and scale text are appropriate to the FFT mode (see assignment).	5	
CPU Load and Relative Deadline measurements for all sampling modes.	15	
Use DMA to transfer ADC1 samples directly to memory.	20	
Source Code & Report	25	
No shared data bugs handling options data for waveform data	15	
Total	100	

Objectives

- Run a computationally intensive task without slowing down the user interface
- Use DMA to optimize ADC I/O

Assignment

In this lab, you will exercise your real-time system by adding **spectrum (FFT) mode**. You will 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 the 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 shows 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 = 0 (zero frequency).

Once you have completed this, you will record measurements of the CPU load of the system in the following table:

ADC Configuration	**Sampling Rate**	**CPU Load**	**ISR Relative Deadline**
Single-sample ISR	1 Msps		
DMA	1 Msps		
DMA	2 Msps		

Then, you will optimize the ADC sampling using DMA, using the DMA controller to replace your ADC ISR. Measure the CPU load in this configuration. Then, increase the sampling rate to 2 Msps, and repeat the measurement. Note that the DMA also requires an ISR, though different from the previous ISR.

Part 0: Copy CCS Project

Copy your Lab 3 CCS project by selecting it in Project Explorer and pressing Ctrl+c, then Ctrl+v. CCS should prompt you to rename the new project: use the convention for online code submission “ece3849_lab3_username1_username2,” substituting your username. When the time comes to submit your source code, follow the instructions at the end of the Lab.

Part 1: Spectrum (FFT) Mode

Here, you will add a spectrum mode that takes significant time to execute to test the prioritization and preemption of tasks in your system.

The Kiss FFT package has been provided to compute the FFT for spectrum mode. It 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 and computes one FFT of a preprogrammed 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))
```

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

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

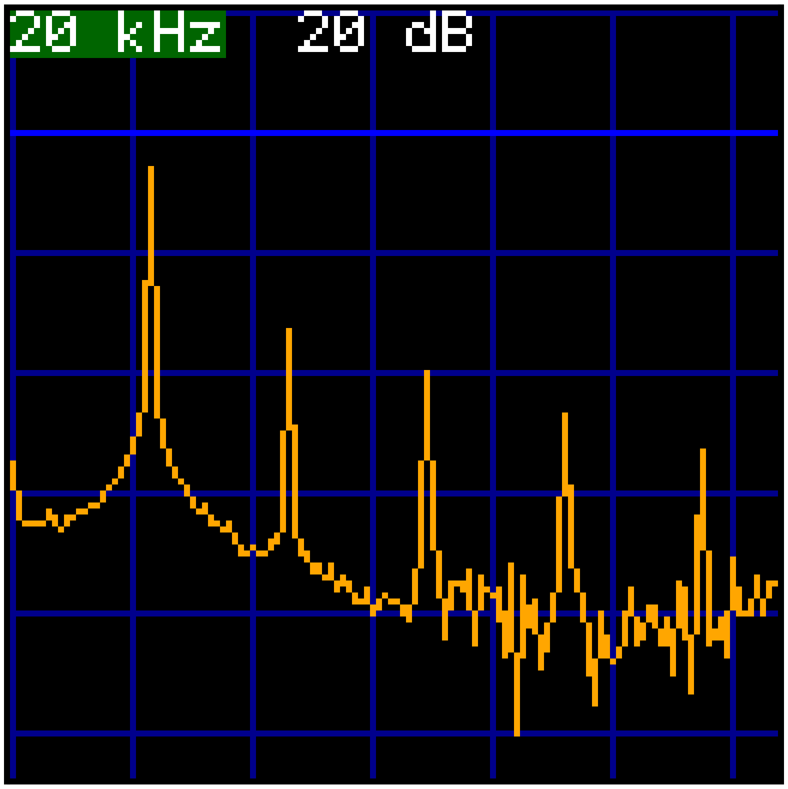
```
for (i = 0; i < NFFT; i++) { // generate an input waveform
    in[i].r = sinf(20*PI*i/NFFT); // real part of waveform
    in[i].i = 0; // imaginary part of waveform
}

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 test input sinusoid has a period of 102.4 samples (1/10 of the total number of samples). The output should be 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 one point at a time as follows:

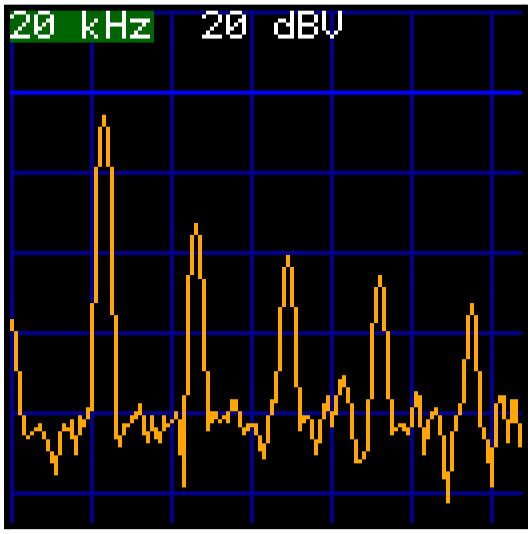
```
out_db[i] = 10 * log10f(out[i].r * out[i].r + out[i].i * out[i].i);
```

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. 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, use the "f" suffix on numbers and math functions.)



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, then 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));
}
```



Part 2: Performance Measurement

With the CPU-intensive spectrum mode implemented, you will now measure the performance of the system.

Take the measurements necessary to fill out the following table. Recall that the relative deadline for your ADC ISR is how much time can pass after the hardware has triggered the ISR before data will be lost.

ADC Configuration	**Sampling Rate**	**CPU Load**	**ISR Relative Deadline**
Single-sample ISR	1 Msps		
DMA	1 Msps		
DMA	2 Msps		

Part 3: ADC Optimization

Now you will use DMA to optimize your ADC sampling, transferring from the ADC to memory with almost no CPU usage.

The Direct Memory Access (DMA) controller can perform most of the duties of the Lab 2 ADC ISR without using any CPU time. While an ISR is still required, its relative deadline is nowhere near as short. A DMA interrupt occurs only when a long DMA transfer completes, not every ADC sample. The DMA setup code is given to you in the following code block.

```
#include "driverlib/udma.h"
```

```
#pragma DATA_ALIGN(gDMAControlTable, 1024) // address alignment required
tDMAControlTable gDMAControlTable[64]; // uDMA control table (global)
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
uDMAEnable();
uDMAControlBaseSet(gDMAControlTable);
// assign DMA channel 24 to ADC1 sequence 0
uDMAChannelAssign(UDMA_CH24_ADC1_0);
uDMAChannelAttributeDisable(UDMA_SEC_CHANNEL_ADC10, UDMA_ATTR_ALL);
// primary DMA channel = first half of the ADC buffer
uDMAChannelControlSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT,
                      UDMA_SIZE_16 | UDMA_SRC_INC_NONE | UDMA_DST_INC_16 |
                      UDMA_ARB_4);
uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT,
                       UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                       (void*)&gADCBuffer[0], ADC_BUFFER_SIZE/2);
// alternate DMA channel = second half of the ADC buffer
uDMAChannelControlSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT,
                      UDMA_SIZE_16 | UDMA_SRC_INC_NONE | UDMA_DST_INC_16 |
                      UDMA_ARB_4);
uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT,
                       UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                       (void*)&gADCBuffer[ADC_BUFFER_SIZE/2],
                       ADC_BUFFER_SIZE/2);
uDMAChannelEnable(UDMA_SEC_CHANNEL_ADC10);
```

The DMA controller configuration is as follows:

- Allocate the DMA control table in RAM (#pragma specifies alignment)
- Assign a supported DMA channel to ADC1 sequence 0
- Configure the primary and alternate DMA control blocks for this channel:
    - Source = ADC1 Sample Sequence Result FIFO 0 register (direct access)
    - Primary channel destination = first half of the ADC buffer
    - Alternate channel destination = second half of the ADC buffer
    - Data size = 16 bits (one ADC sample)
    - Source address increment = 0 (always read the same register)

- Destination address increment = 16 bits (same as data size)
- Arbitration size = 4 transfers
- DMA mode = ping-pong

This configuration is fairly straightforward. In ping-pong mode, DMA operates on one of the two buffers, while the CPU is supposed to access the other. Then they swap, achieving a continuous transfer. Note that DMA transfer size is limited to **1024 items**. An ADC buffer size of 2048 samples is the maximum possible for this simple static setup, where each DMA channel is assigned to a fixed buffer.

The only mysterious parameter is "arbitration size." The ADC sequence 0 can be configured as an 8-sample FIFO. Normally, DMA will transfer from the ADC FIFO to RAM even if only one sample is available. If this FIFO becomes half-full (4 samples), this indicates that the DMA subsystem is very busy. At this point, the ADC DMA is given higher priority so it could go ahead despite other DMA traffic competing for bandwidth. For the ADC, the arbitration size defines this priority increase threshold, as well as the longest DMA burst length. Longer bursts are more bandwidth efficient but introduce longer latency to other DMA transfers. With only the ADC using DMA, this setting has no real effect.

In ping-pong mode, we are configuring two DMA channels that are assumed linked (complementary). When a ping-pong DMA channel reaches the end of transfer, it starts its complementary channel, triggers an interrupt, then stops and waits for further instructions. It does not automatically reset itself for another transfer. If left alone, the DMA transfer will halt when the alternate channel finishes its transfer. Resetting the DMA channels, such that continuous DMA is maintained, is the job of the new ADC ISR.

DMA interrupts are passed on to the peripheral where the DMA transfer is occurring. **Instead of the normal ADC1 sequence 0 interrupt**, enable the ADC1 sequence 0 **DMA interrupt**. The DMA feature for ADC1 sequence 0 must also be enabled in the ADC peripheral. The additional ADC setup code is partially given in the following code block.

```
ADCSequenceDMAEnable(ADC1_BASE, 0); // enable DMA for ADC1 sequence 0
ADCIntEnableEx(...); // enable ADC1 sequence 0 DMA interrupt
```

The new ADC ISR is partially implemented in the following code block. Complete the missing functionality.

```
// is DMA occurring in the primary channel?
volatile bool gDMAPrimary = true;
void ADC_ISR(void) // DMA
{
    ADCIntClearEx(...); // clear the ADC1 sequence 0 DMA interrupt flag
    // Check the primary DMA channel for end of transfer, and
    // restart if needed.
    if (uDMAChannelModeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT) ==
        UDMA_MODE_STOP) {
        // restart the primary channel (same as setup)
        uDMAChannelTransferSet(...);
        // DMA is currently occurring in the alternate buffer
        gDMAPrimary = false;
    }
    // Check the alternate DMA channel for end of transfer, and
    // restart if needed.
    // Also set the gDMAPrimary global.
    <...>

    // The DMA channel may be disabled if the CPU is paused by the debugger
    if (!uDMAChannelIsEnabled(UDMA_SEC_CHANNEL_ADC10)) {
        // re-enable the DMA channel
        uDMAChannelEnable(UDMA_SEC_CHANNEL_ADC10);
    }
}
```

Once you complete the DMA controller setup, ADC1 sequence 0 DMA interrupt setup and the ISR, run your code. You should see a waveform being continuously sampled into the ADC buffer. Unfortunately, gADCBufferIndex is no longer updated by the DMA mechanism. So, your trigger search does not know where to find the newest sample in the ADC buffer.

Finding where DMA is currently writing samples into the ADC buffer is not too difficult. We can query the number of items **remaining** to transfer in a DMA channel using the `uDMAChannelSizeGet()` function. The only problem is that there are two DMA channels in ping- pong mode. We need to know which channel is currently sampling, and which is waiting for its turn. The ADC ISR supplies this information through the `gDMAPrimary` global. If we check which channel is currently sampling and then read the remaining transfer size for that channel, we run into the **non-atomic read** issue. By the time we get to read the remaining transfer size, the channel may finish and be restarted by the ISR. The following function returns the location of the newest DMA sample in the ADC buffer but suffers from the non-atomic read issue. **Fix it using the appropriate TI-RTOS Gate object.** Change the priority of the ADC Hwi to make is **no longer a "zero-latency Hwi."** The worst thing that can happen then is that the DMA channel we detected as currently active finishes, and its remaining transfer size reads 0. Since the other channel has just started, our estimate of the newest sample location is then only slightly off.

```
int32_t getADCBufferIndex(void)
{
    int32_t index;
    if (gDMAPrimary) { // DMA is currently in the primary channel
        index = ADC_BUFFER_SIZE/2 - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 |
                                   UDMA_PRI_SELECT);
    } else { // DMA is currently in the alternate channel
        index = ADC_BUFFER_SIZE - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 |
                UDMA_ALT_SELECT);
    }
    return index;
}
```

Use the above function instead of reading the `gADCBufferIndex` variable in the Waveform Task. You should comment out the definition of `gADCBufferIndex` to make sure you are not using it anywhere anymore.
Measure the CPU load with this new DMA implementation of ADC sampling. Now increase the sampling rate to 2 Msps by adjusting the ADC clock. Measure the CPU load again and fill in the table given above. You should now have plenty of CPU available for other demanding real-time tasks.

Part 4: Submitting Source Code

Make sure your Lab 4 project is named "**ece3849_lab4_username**" with your username substituted. **Clean** your project (remove compiled code). Right click on your project and select "Export..." Select General → "Archive File." Click Next. Click Browse. Find a location for you archive and specify the exact same file name as your project name. Click Finish. Submit the resulting .zip file to Canvas. This concludes Lab 4.