Lab 2: RTOS, Spectrum Analyzer

Yil Verdeja, ECE Box 349
April 16 2019

Table of Contents

List of Figures

1 Introduction

The objectives of this lab were to (1) port an existing real-time application to a real-time operating system (RTOS), (2) use fundamental RTOS objects such as Task, Hardware Interrupt (Hwi), Clock, Semaphore and Mailbox, (3) deal with shared data and other inter-task communication and (4) run a computationally intensive task without slowing down the user interface.

After porting the real-time application created in Lab 1 to RTOS, a fast fourier transform (FFT) mode was added that turned the system into a simple spectrum analyzer after the press of a button. This was completed by running all the code through TI-RTOS threads. The analog to digital converter (ADC) interrupt service routine (ISR) was converted to a Hwi, while the button handling and waveform generation were converted to tasks of different priorities.

2 Methodology, Results and Discussion

This section describes the implementation of each major step in the lab by going through them and describing how each was completed. The following image outlines the structure of the lab 1 digital oscilloscope ported to TI-RTOS where software threads and modules are represented by smooth shaded object. Important shared data and inter-task communication objects used textured shading and clear boxes indicate hardware.

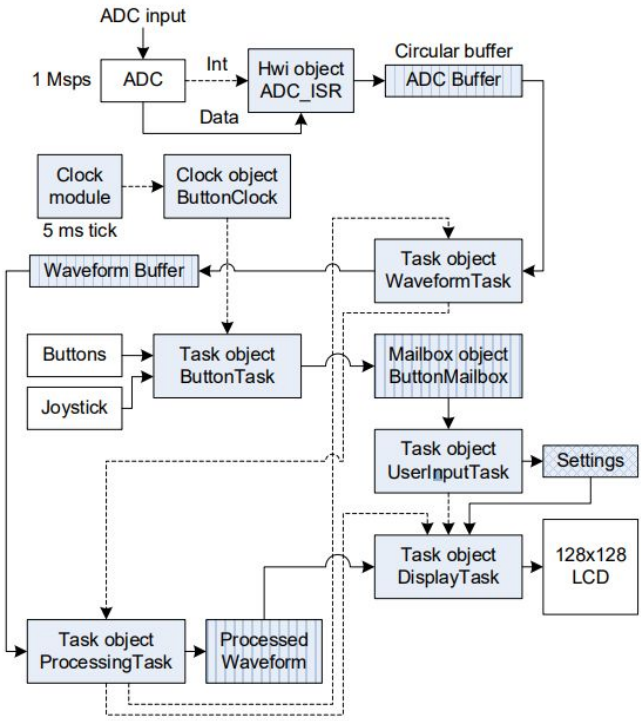

Figure 1. Top Level Design Implementation

As shown in the figure above, there are five different threads for each task: waveform, button, user input, display and processing. They are arranged vertically by priority. The top level design implementation in the figure was used to design the RTOS system.

To complete this project the ADC ISR was converted to a Hwi object, while the button, waveform generation, and display we're converted to tasks. Using user commands retrieved from a Mailbox queue, the oscilloscope settings can be altered and the waveform mode can be toggled between sinusoidal and spectrum.

2.1 ADC ISR to Hwi Object

The first task was to convert the ADC ISR to a Hwi object. Initially the ADC ISR ran periodically every 1 microsecond (µs) using a timer. By creating a Hwi object with highest priority attached to the ADC ISR function, it permits the ISR to execute every 1 µs without interference from critical sections in the operating system (OS). As found on Table 2-9 in the TM4C1294NCPTDT datasheet, the vector number of ADC1 has a value of 62. By placing this value inside the *interrupt number* setting, the Hwi interrupt was configured correctly for the ADC ISR. This step was complete after the Hwi object met all its deadlines. This was determined by looking at the *gADCError* variable in the ADC ISR which determines if deadlines are met. Having stayed at a value of zero proved that the Hwi met all its deadlines.

2.2 Task Object, Clock Object and Mailbox Object

In order to schedule periodic button scanning, a clock object instance was added. The clock module was set to a 5 millisecond (ms) clock tick period. Using the semaphore *semButtons*, the clock function signals the Button Task. Copying the functionality from the previous Button ISR function in lab 1, this task scan's the buttons and posts the button ID to a mailbox if a press is detected.

```
// signal button task periodically using a semaphore
void clock_func(UArg arg1){
    Semaphore_post(semButtons); // to buttons
}
```

Figure 2. Clock Function

```
// Button Task to handle button presses in Mailbox Queue
void buttonTask_func(UArg arg1, UArg arg2) { // pri 2
    while(true){
        Semaphore_pend(semButtons, BIOS_WAIT_FOREVER); // from clock
        // read hardware button state
        uint32_t gpio_buttons =
                ~GPIOPinRead(GPIO_PORTJ_BASE, 0xff) & (GPIO_PIN_1 | GPIO_PIN_0) | // EK-TM4C1294XL buttons in positions 0 and 1
                (~GPIOPinRead(GPIO_PORTH_BASE, 0xff) & (GPIO_PIN_1)) << 1 | // BoosterPack button 1
                (~GPIOPinRead(GPIO_PORTK_BASE, 0xff) & (GPIO_PIN_6)) >> 3 | // BoosterPack button 2
                ~GPIOPinRead(GPIO_PORTD_BASE, 0xff) & (GPIO_PIN_4); // BoosterPack buttons joystick select button

        uint32_t old_buttons = gButtons;    // save previous button state
        ButtonDebounce(gpio_buttons);       // Run the button debouncer. The result is in gButtons.
        ButtonReadJoystick();               // Convert joystick state to button presses. The result is in gButtons.
        uint32_t presses = ~old_buttons & gButtons;   // detect button presses (transitions from not pressed to pressed)
        presses |= ButtonAutoRepeat();      // autorepeat presses if a button is held long enough
        char button_char;

        if (presses & 2) {
            // trigger slope change
            button_char = 't';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }

        if (presses & 4) {
            // increment
            button_char = 'u';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }

        if (presses & 8) {
            // spectrum mode
            button_char = 's';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }
    }
}
```

Figure 3. Button Task

The button semaphore *semButtons* was set to binary mode with an initial count of 0. In this case, it would only be entered after the 5 ms interrupt from the clock. The debugger was used to verify the functionality of the mailbox posting button IDs. From the debugger, prior to any button presses, the number of free messages was at 10. After every button press, the number of free

messages would decrease by 1 as expected. Once there were no more free messages, the mailbox could no long post any more button presses.

2.3 Tasks

Looking at the top level design implementation in figure 1, the rest of the tasks were implemented. Prior to fully implementing all the tasks, the semaphore functionality was first set. The priorities from highest to lowest are:

1. Waveform Task
2. Button Task
3. User Input Task
4. Display Task
5. Processing Task

In addition to the the top level design implementation, the display task blocks again after drawing the waveform to the liquid crystal display (LCD). Due to the stack usage of the processing and display tasks, their stack sizes were changed to values of 2048 in order for proper function.

2.3.1 Critical Section Protection

To protect critical sections from shared data bug, a semaphore *sem_cs* was created to wrap around certain critical sections. It's a binary semaphore initially instantiated with a count of 1 so it can enter in the first run.

2.3.2 User Input Task

As shown below, the user input task function checks whether there are any presses in the mailbox by pending it. If any button presses exist in the queue, the corresponding action is taken. It checks both the button press in the mailbox and the current global button press to verify what is being pressed. In figure 4, the actions that can be taken are changing the voltage per division scale, changing the slope, or changing the mode of the waveform.

```
// User input task to read button presses and take actions accordingly
void userInputTask_func(UArg arg1, UArg arg2) { // pri 3
    char bpresses[10]; // holds fifo button presses
    while(true){
        if (Mailbox_pend(mailbox0, &bpresses, TIMEOUT)) {
            // read bpresses and change state based on bpresses buttons and whether it is being pressed currently
            int i;
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < 10; i++){
                if (bpresses[i]==('u') && gButtons == 4){ // increment state
                    stateVperDiv = (++stateVperDiv) & 3;
                } else if (bpresses[i]==('t') && gButtons == 2){ // trigger
                    risingSlope = !risingSlope;
                } else if (bpresses[i]==('s') && gButtons == 8){ // spectrum mode
                    spectrumMode = !spectrumMode;
                }

            }
            Semaphore_post(sem_cs);
        }

        Semaphore_post(semDisplay); // to display

    }
}
```

Figure 4. User Input Task

Since *gButtons, stateVperDiv, and risingSlope* are global shared data, the critical section of the *for* loop is being protected by the *sem_cs* semaphore. After reading from the mailbox, the user input task signals to the display. To verify the functionality, the debugger was used to check the values of the settings after each button press. Since this task pends on the mailbox often, it was expected to always have the number of free messages stay at a value of 10.

2.3.3 Waveform Task

The waveform task is the highest priority because it has to run as often as the ADC ISR function. It makes sure the trigger search completes before the ADC overwrites the buffer. Using trigger search, it searches for the trigger in the ADC Buffer and copies the triggered waveform into a waveform buffer.

```
// Waveform Task to handle waveform mode and creating waveform to print
void waveformTask_func(UArg arg1, UArg arg2) { // pri 1
    IntMasterEnable();
    while(true){
        Semaphore_pend(semWaveform, BIOS_WAIT_FOREVER); // from processing
        trigger_value = zero_crossing_point(); // Dynamically finds the ADC_OFFSET
        if (spectrumMode){
            int i;
            int buffer_ind = gADCBufferIndex;

            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < NFFT; i++){
                fft_samples[i] = gADCBuffer[ADC_BUFFER_WRAP(buffer_ind - NFFT + i)];
            }
            Semaphore_post(sem_cs);

        } else {
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            triggerSearch(); // searches for trigger
            Semaphore_post(sem_cs);
        }
        Semaphore_post(semProcessing); // to processing

    }
}
```

Figure 5. Waveform Task

As shown in figure 6, the *triggerSearch* function was rewritten which fixed any of the glitches from Lab 1. Since it utilizes the *gADCBuffer* global shared data, the *sem_cs* semaphore wraps around it to protect that critical section.

```
// Searches the trigger
void triggerSearch(void){
    int32_t trigger_index;
    int32_t i;

    // Goes backwards through the whole gADCBuffer array, and finds the zero-crossing point index and shifts it
    trigger_index = gADCBufferIndex - LCD_HORIZONTAL_MAX/2;
    if (risingSlope) { // Rising slope trigger search
        for (i = 0; i < ADC_BUFFER_SIZE/2; i++, trigger_index--) {
            if (gADCBuffer[ADC_BUFFER_WRAP(trigger_index)] <= trigger_value &&
                    gADCBuffer[ADC_BUFFER_WRAP(trigger_index + 1)] > trigger_value) {
                break; // if found, stop looking
            }
        }
    }
    else { // Falling slope trigger search
        for (i = 0; i < ADC_BUFFER_SIZE/2; i++, trigger_index--) {
            if (gADCBuffer[ADC_BUFFER_WRAP(trigger_index)] >= trigger_value &&
                    gADCBuffer[ADC_BUFFER_WRAP(trigger_index + 1)] < trigger_value) {
                break; // if found, stop looking
            }
        }
    }

    if (i == ADC_BUFFER_SIZE/2) { // If trigger not found, set to previous value
        trigger_index = gADCBufferIndex - LCD_HORIZONTAL_MAX/2;
    }

    // A local buffer retrieves 128 samples of the gADCBuffer from the trigger_index previously found
    for (i = 0; i < ADC_TRIGGER_SIZE; i++){
        trigger_samples[i] = gADCBuffer[ADC_BUFFER_WRAP(trigger_index - (ADC_TRIGGER_SIZE - 1) + i)];
    }

}
```

Figure 6. Trigger Search Rewritten

The functionality of the waveform task was verified by checking that the *trigger_samples* array gets frequently updated as much as the gADCBuffer array updates.

2.3.4 Display Task

Set as the second lowest priority, the display task basically draws the waveform on the LCD after settings have been changed and/or waveform processing is complete. Whilst all the other tasks are implemented in Buttons.c, the display task is implemented in main.c.

```
void displayTask_func(UArg arg1, UArg arg2) { // prd 4
    char tscale_str[50];   // time string buffer for time scale
    char vscale_str[50];   // time string buffer for voltage scale
    char tslope_str[50];   // time string buffer for trigger edge
    while(true){
        Semaphore_pend(semDisplay, BIOS_WAIT_FOREVER); // from user input

        tRectangle rectFullScreen = {0, 0, GrContextDpyWidthGet(&sContext)-1,
                                     GrContextDpyHeightGet(&sContext)-1}; // full-screen rectangle

        GrContextForegroundSet(&sContext, ClrBlack);
        GrRectFill(&sContext, &rectFullScreen); // fill screen with black
        GrContextForegroundSet(&sContext, ClrWhite); // yellow text

        // draw grid in blue
        GrContextForegroundSet(&sContext, ClrBlue); // yellow text
        int i;
        for (i = 1; i < 128; i+=21){
            GrLineDraw(&sContext, i, 0, i, 128);
            GrLineDraw(&sContext, 0, i, 128, i);
        }

        // draw center grid lines in dark blue
        GrContextForegroundSet(&sContext, ClrDarkBlue); // blue
        if (spectrumMode){
            GrLineDraw(&sContext, 0, 22, 128, 22);
        } else {
            GrLineDraw(&sContext, 64, 0, 64, 128);
            GrLineDraw(&sContext, 0, 64, 128, 64);
        }

        // draw waveform
        GrContextForegroundSet(&sContext, ClrYellow); // yellow text
        int x;
        int y_old;
        Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
        for (x = 0; x < LCD_HORIZONTAL_MAX - 1; x++) {
            if (x!=0)
                GrLineDraw(&sContext, x-1, y_old, x, processedWaveform[x]);
            y_old = processedWaveform[x];
        }
        Semaphore_post(sem_cs);

        // Time Scale, Voltage Scale, Trigger Slope and CPU Load
        GrContextForegroundSet(&sContext, ClrWhite); // yellow text
        if (spectrumMode){
            snprintf(tscale_str, sizeof(tscale_str), "20kHz"); // convert time scale to string
            snprintf(vscale_str, sizeof(vscale_str), "20dB"); // convert vscale to string
        } else {
            snprintf(tscale_str, sizeof(tscale_str), "20us"); // convert time scale to string
            snprintf(vscale_str, sizeof(vscale_str), gVoltageScaleStr[stateVperDiv]); // convert vscale to string
            snprintf(tslope_str, sizeof(tslope_str), gTriggerSlopeStr[risingSlope]); // convert slope to string
            GrStringDraw(&sContext, tslope_str, /*length*/ -1, /*x*/ LCD_HORIZONTAL_MAX/2 + 20, /*y*/ 5, /*opaque*/ false);

        }

        GrStringDraw(&sContext, tscale_str, /*length*/ -1, /*x*/ 7, /*y*/ 5, /*opaque*/ false);
        GrStringDraw(&sContext, vscale_str, /*length*/ -1, /*x*/ LCD_HORIZONTAL_MAX/2 - 20, /*y*/ 5, /*opaque*/ false);

        GrFlush(&sContext); // flush the frame buffer to the LCD
        Semaphore_pend(semDisplay, BIOS_WAIT_FOREVER); // block again
    }
}
```

Figure 7. Display Task Implementation

Using the semaphore *semDisplay* at an initial count of 0, the display task draws the grid, and the corresponding waveform: spectrum or sinusoidal. If the sinusoidal mode is chosen, the time scale, voltage scale and slope are shown. If the spectrum mode is chosen, the frequency and decibels is printed on the screen. Since the *for* loop in drawing the waveform uses the *processedWaveform* global shared data, it is protected with the *sem_cs* semaphore. Ideally, a GateMutexPri would have been used as it is preferred over longer critical sections such as this one.

2.3.5 Processing Task

Finally the processing task was implemented. At the lowest priority, the processing task blocks on the *semProcessing* semaphore initialized at 0. As shown in figure 1, processing occurs after the waveform is retrieved from the ADC Buffer. It simply scales the captured waveform for display and stores the processed waveform in another global buffer named *processingWaveform*. It signals the display task to draw the waveform, and then signals the waveform task to capture another waveform.

```
// Processing Task to correct spectrum wave and sine wave to LCD
void processingTask_func(UArg arg1, UArg arg2) { // pri 5
    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

    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)));
    }

    while(true){
        Semaphore_pend(semProcessing, BIOS_WAIT_FOREVER); // from waveform
        if (spectrumMode){

            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < NFFT; i++) { // generate an input waveform
             in[i].r = ((float)fft_samples[i] - trigger_value) * w[i]; // real part of waveform
             in[i].i = 0; // imaginary part of waveform
            }
            Semaphore_post(sem_cs);

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

            // convert first 128 bins of out[] to dB for display
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < ADC_TRIGGER_SIZE - 1; i++) {
                processedWaveform[i] = (int)roundf(128 - 10*log10f(out[i].r*out[i].r +out[i].i*out[i].i));
            }
            Semaphore_post(sem_cs);

        } else {
            fScale = (VIN_RANGE/(1 << ADC_BITS))*(PIXELS_PER_DIV/fVoltsPerDiv[stateVperDiv]); // determines fScale
            int i;

            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < ADC_TRIGGER_SIZE - 1; i++) {
                processedWaveform[i] = ((int)(ADC_TRIGGER_SIZE/2)
                        - (int)roundf(fScale*(int)(trigger_samples[i] - trigger_value)));
            }
            Semaphore_post(sem_cs);
        }
        Semaphore_post(semWaveform); // to waveform
        Semaphore_post(semDisplay); // to display
    }
}
```

Figure 8. Processing Task

The else statement in figure 8 shows how the the sinusoidal waveform was processed to fit into the LCD screen. Since *trigger_samples* is accessed, the *for* loop is wrapped around the *sem_cs* semaphore to protect the critical section from shared data bugs.

2.3.6 Overall Functionality

To verify it's overall functionality, the program was uploaded on the board. The following images show that the porting the real-time program to RTOS proved successful.

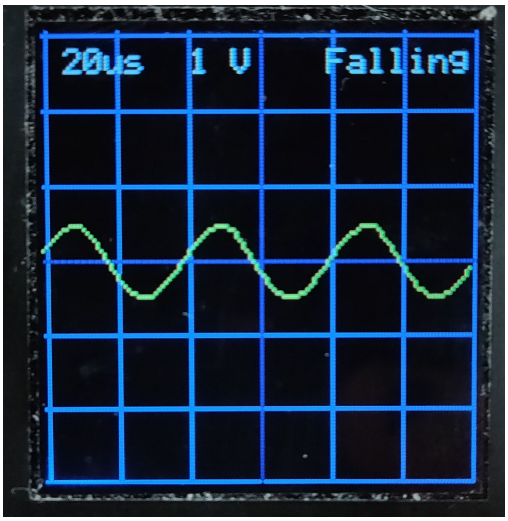

Figure 9. 1 V Sine Waveform Falling Trigger

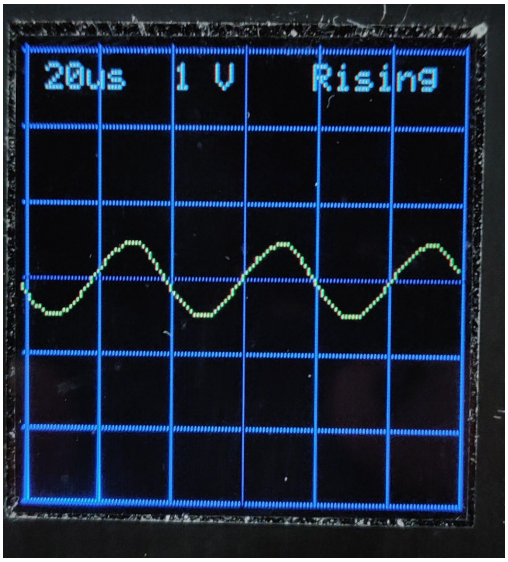

Figure 10. 1 V Sine Waveform Rising Trigger

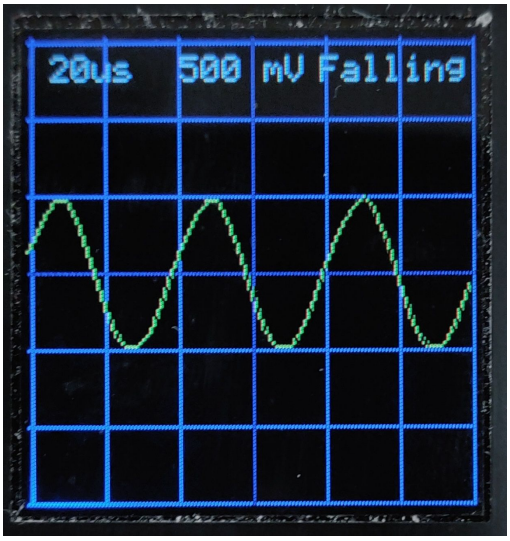

Figure 11. 500 mV Sine Waveform Falling Trigger

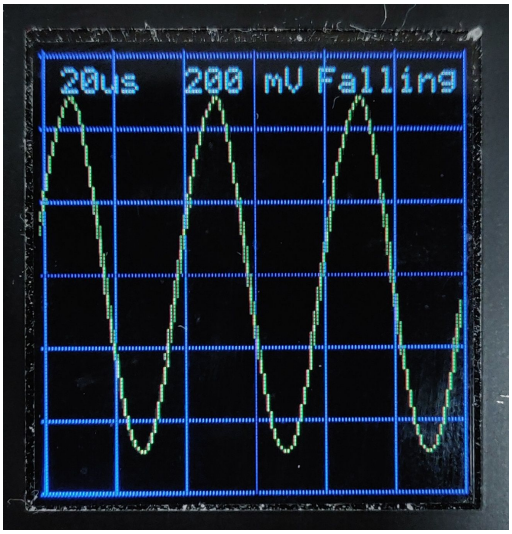

Figure 12. 200 mV Sine Waveform Falling Trigger

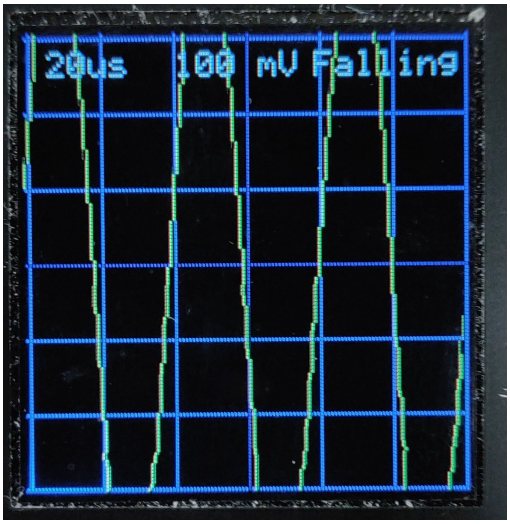

Figure 13. 100 mV Sine Waveform Falling Trigger

2.4 Spectrum (FFT) Mode

An additional button was assigned to switch between oscilloscope mode and **spectrum mode**. In spectrum mode, the functionality was changed. As shown in figure 5, the waveform task captures the newest 1024 samples without searching for the trigger. These samples are placed in a global array *fft_samples* and the critical section is thus protected using the *sem_cs* semaphore.

In the processing task, the Kiss FFT package is utilized to compute the 1024 point FFT of the captured waveform in single precision floating point. Figure 8 shows the initialization of the kiss FFT config memory, and the complex waveform and spectrum buffers. Prior to the tasks while loop, a window function is created to reduce spectral leakage. This function is multiplied by the 1024 time domain samples retrieved later. The processing task converts the lowest 128 frequency bins of the complex spectrum to magnitude in dB, and saves them into the processed waveform buffer scaled at 1 dB/pixel for display. The processing to the processed waveform is protected with the *sem_cs* semaphore as it accesses a shared global data. The spectrum is converted to dB scale using the log10f() function. The processed waveform is then shifted to fit within the LCD screen by subtracting from the height of the screen.

Finally, as shown in figure 7, the display task prints the frequency and dB scales of the spectrum mode. Unlike the oscilloscope mode, the dark blue grid is modified so that the first vertical line is at x = 0, corresponding to zero frequency.

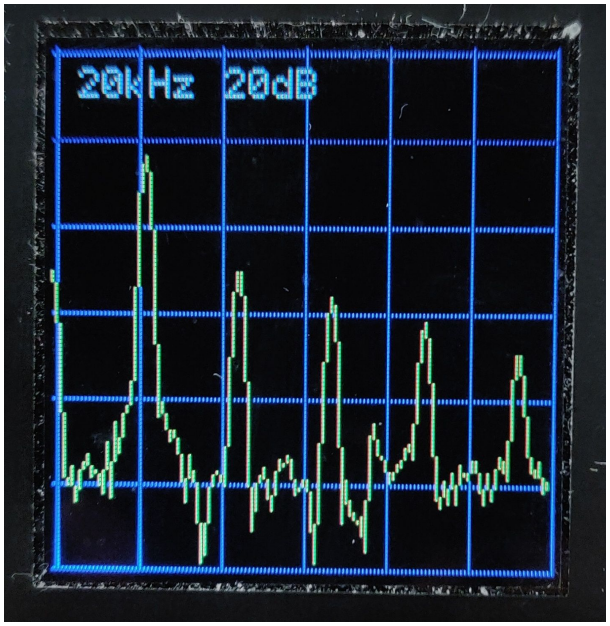

Figure 14. Spectrum (FFT) Mode

3 Conclusion

The most important outcome of this project was porting a real-time application to RTOS by using fundamental RTOS objects such as Tasks, Hwi, Clocks, Semaphores and Mailboxes. Debugging in real-time proved to be a great tool because it allowed to better understand what was happening in the code, but it also helped in fixing errors when they occured.

A major problem encountered was either not taking a semaphore or taking the incorrect semaphore. Even though the tasks had the right functionality, since the semaphores were placed in the wrong order, they program wasn't doing what it was supposed to do. This proved to be very hard to debug since there was too much going on in the code, and errors like these can run smoothly without notice. To fix this issue, each task was re-written to only blocking and signaling of semaphores. After running through the program and verifying that each task was being called appropriately, the main functionality of each task was re-implemented.

Another outcome of this lab was identifying critical sections with shared data bugs and protecting them with the *sem_cs* semaphore. For the purposes of this lab, this simple implementation was fine. However to protect against priority inversion, a Gate Mutex Pri or Gate Task object should have been utilized to protect certain critical sections.