Lab 4: Motion Controlled Audio Processing and FreeRTOS

Justin Schwiesow, 1864655 19-August-2022
Emmett Miller, 2076671

Assignment: ECE 474 Lab 4

Introduction:

Lab 4 comprised three sections, all of which served as an introduction to FreeRTOS. First, we demonstrated an example sketch using FreeRTOS to blink an external LED while also reading an analog value from a potentiometer. Building on this sketch in the next section, we implemented a series of increasingly complicated tasks. In addition to blinking an LED, our tasks played the theme from “Close Encounters of the Third Kind”, generated a random signal, and repeatedly took the FFT of this signal while measuring the task wall time. Carefully allocating stack space with FreeRTOS was the key to fixing runtime errors. Our final section was a self-guided project: to implement a system that allowed a user to control the playback and processing of a sound sample. Using a gyroscope/accelerometer module and an LCD, we made an interface for scrubbing through audio (similar to scratching a record) and manipulating reverb amount. Parameters for these processes were displayed on the LCD. All audio processing was implemented in SuperCollider, an integrated sound synthesis language and IDE.

Methods and Techniques:

Instruments:

- Arduino Mega
- LED and resistors
- Passive buzzer
- GY-521 accelerometer/gyroscope
- 16x2 LCD
- Auxiliary power supply
- FreeRTOS
- SuperCollider

Note: SuperCollider 3.12.2 is available here.

Lab section 4-1 required only to connect an LED and a potentiometer to the project board. All code for this assignment was provided. Uploading the sketch, we observed that turning the potentiometer resulted in values in the range 0-1023 being printed to the serial monitor.

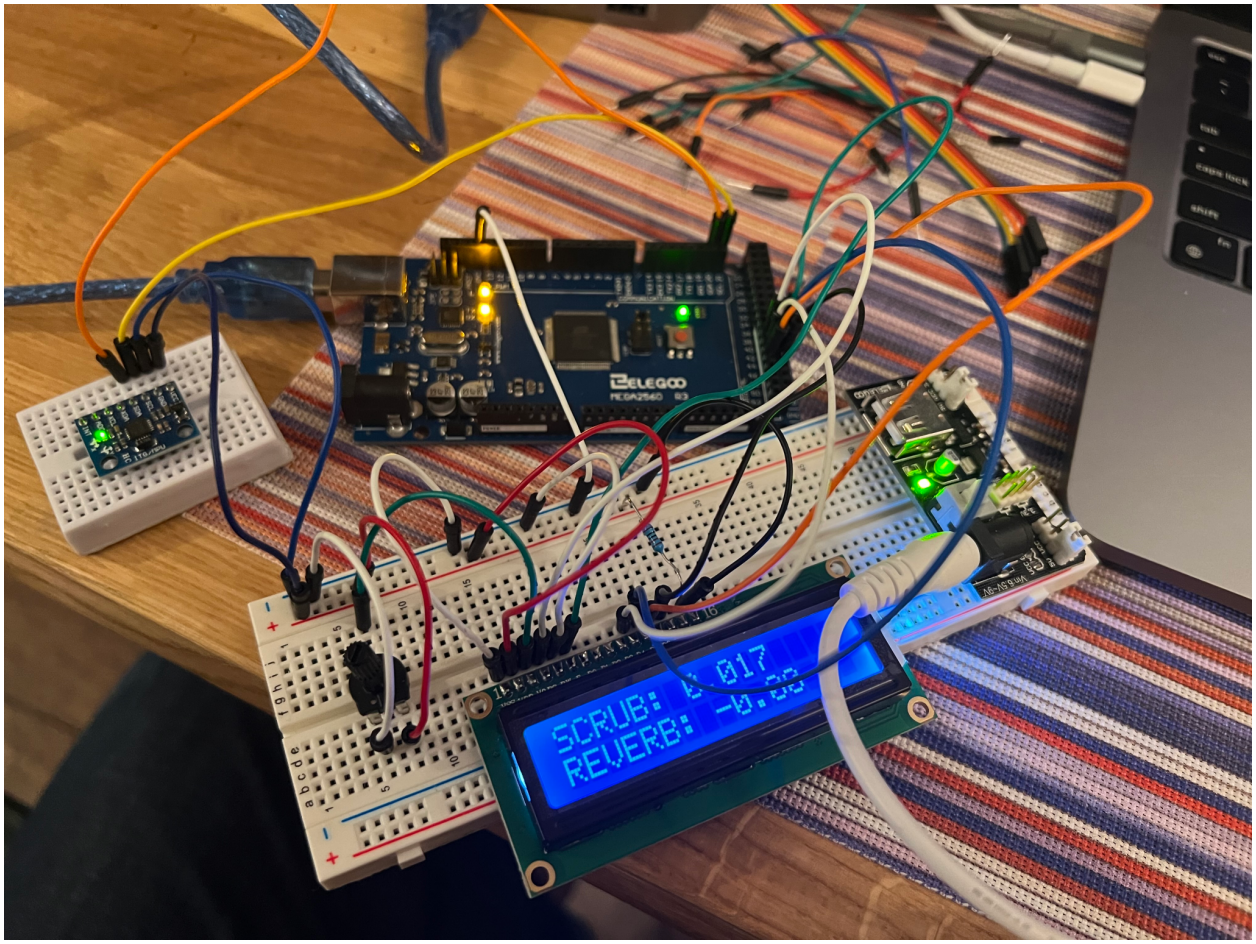
Setup for 4-2 was identical to that of the previous lab, using a buzzer driven by the PWM output of pin PH3. Though assembly of the hardware interface was simple, scheduling tasks in FreeRTOS was a delicate process. Tasks 1 and 2, blinking the LED and playing the “Close

Encounters" melody, respectively, followed from our implementation in the previous lab. Setting up the tasks in FreeRTOS was a new challenge, and we relied on the OS's documentation throughout the lab. Next, we created a new task, rt3(), which in turn created another task (rt3p0()) and halted itself. rt3p0() creates the random signal to be processed by the FFT as well as the queue that the next task, rt3p1(), uses to send the signal to the FFT function. After creating tasks rt3p1() and rt4(), rt3p0() halts itself. rt3p1() creates the queue in which it receives wall times from the FFT and then runs through five iterations of sending the signal and receiving wall times. rt4() in turn receives the signal, computes the FFT, and transmits wall time back to rt3p1(). After each iteration, rt3p1() prints the wall time, finally halting itself and rt4().
We paid careful attention to task priority and stack allocation to ensure the program ran correctly. To simplify the debugging process, we introduced the tasks one at a time, noting the conditions that resulted in stack overflow or otherwise glitchy behavior.

For part 4-3, our self-guided project, we started simply, reading from a potentiometer and printing to the serial monitor as in 4-1. The next step was to determine how to send this data to SuperCollider on the PC. There are well-documented methods using *Open Sound Protocol* (OSC) to generate musical events in other applications from Arduino, but SuperCollider only has inbuilt classes for receiving OSC via UDP or TCP networking. Short of writing a SuperCollider class to receive OSC via the serial port, we would need to connect our project board to the PC via WiFi. Unfortunately, all efforts to get an ESP8266-based WiFi shield to work were unsuccessful. This may have been because the board needed a firmware update.

Our next strategy was to write the raw accelerometer data to the serial monitor with a series of character tags that we could use to parse the data (i.e. separate x, y, and z values) in SuperCollider manually. Doing so presented its own challenges, such as converting ascii values to characters to decimal integers, and determining when to apply negative signs. This video tutorial from Eli Fieldsteel was a great resource, illustrating a similar method. Once we had the accelerometer values in SuperCollider, we were able to generate OSC messages locally and then apply them to processing parameters on sample playback and reverb.

The final step was to display the parameter names and values on an LCD. Rather than try to message back to Arduino from SuperCollider (and further complicate the serial bus), we opted to display a normalized version of the raw accelerometer data, scaled to match the parameter values in SuperCollider. We connected the auxiliary power supply to drive the LCD. Using the LiquidCrystal library, writing to the LCD was straightforward.



The setup for 4-3

Experimental Results:

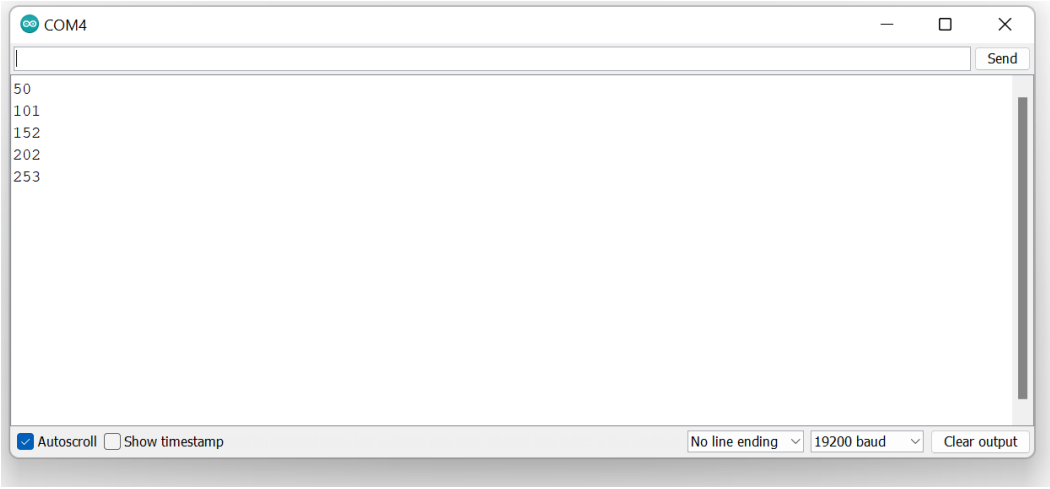
Section 4-1 behaved as expected, but getting 4-2 to run correctly took a few days of troubleshooting. One breakthrough was using a volatile int for wall time. Using LEDs to indicate when certain tasks were starting was an effective diagnosis tool: some bugs were a result of forgetting to suspend tasks. Careful attention to task priority and stack allocation were the key to correct behavior of 4-2. During the in-lab demonstration, we were able to print the wall time of the FFT on a 128-sample buffer. The melody playback and LED were glitch free.

```
285         q2 = xQueueCreate(LOOP_TIMES, sizeof(unsigned int));
286
287         //loop 5 times:
288         for (int i = 0; i < LOOP_TIMES; i++) {
289
290             // Send a pointer to the data to task rt4() (below) using a
291             // FreeRTOS queue function.
292             // NOTE: task will be blocked if queue is full
293             xQueueSend(q1, randoms, WAIT_TICKS);
294
295             // receive current wall time from rt4()
296             // NOTE: task will be blocked if queue is empty
297             xQueueReceive(q2, thisFFTtime, WAIT_TICKS);
298
299             // Report back to laptop/PC, via the Serial link, how much "wall
300             // clock time" elapsed for the 5 FFTs.
301             Serial.println(totalTime);
```

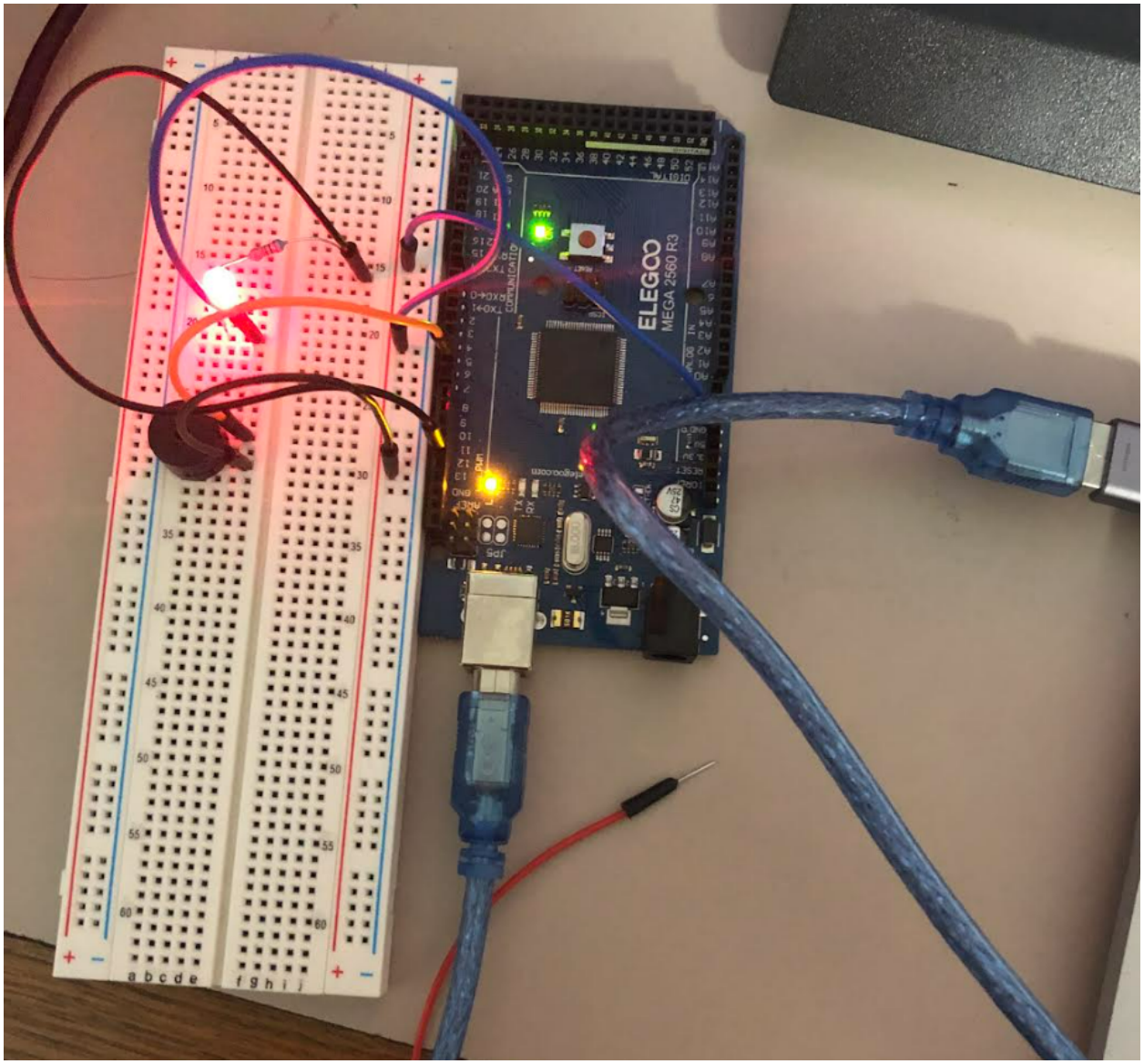
We used Queues to communicate between the tasks for the FFT.

```
// FFT.
time1 = t;

wallTime = time1 - time0;

// Send a message back to RT-3 using the second queue with the
// wall-clock time time for the completed FFT.
totalTime += wallTime;
```

A global variable was used to calculate the total time



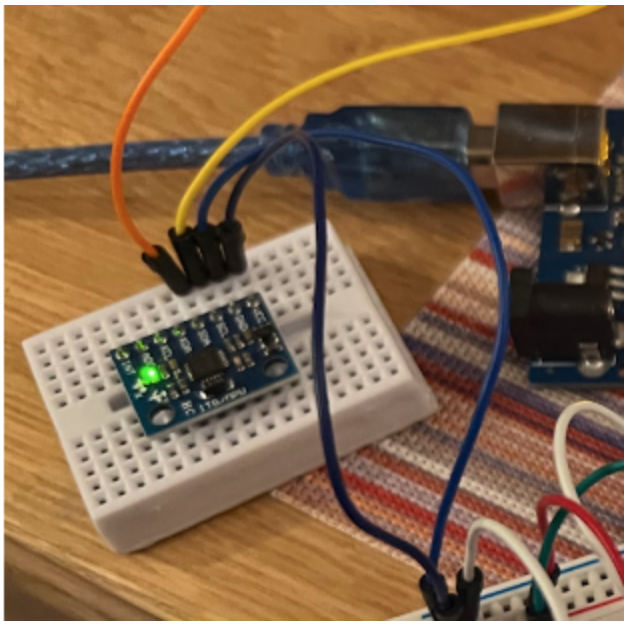
We printed the total elapsed time each loop for the Five FFT loops while running the LED and playing the music. The last value was the total time it took to run all five FFTs.



For 4_2 and 4_1 this was our Hardware Setup

Once our motion-sound interface was working, the system took some tweaking to produce musical results. We experimented with accelerometer layout and wiring to keep the leads from unplugging during rapid movement. We also experimented using processing other than reverb. Stereo panning was much harder to perceive, especially over laptop speakers in a noisy lab. We found that human speech was an effective processing source: the results were both recognizable and musically interesting.

This was our LCD screen displaying the amount of processing(Reverb) and playback speed/Direction(Scrub)



This is the GyroScope that we used I2C to communicate with.

Code Documentation: SuperCollider

See included HTML file for 4-2 and Arduino portion of 4-3 documentation.

Just as we select a serial port when using the Arduino IDE, we must tell SuperCollider to listen to a port for incoming messages. We also specify the local IP address for reading OSC messages. After booting the server for real-time synthesis and processing, we initialize a three-channel control bus that will store the raw x, y, and z accelerometer values. We also

initialize a function to respond to OSC messages, updating the values in the control bus on each message reception.

```
SerialPort.listDevices


// SETUP
(
// set up Net Address
~local = NetAddr.localAddr;

// initialize serial port
// NOTE: usbmodem number can change! Check with SerialPort.listDevices above
// NOTE: match baud rate to arduino sketch
~port = SerialPort.new("/dev/tty.usbmodem1101", 115200);
)



(
// boot server if not running
// control bus and output proxy
s.waitForBoot({
    ~controlBus = Bus.control(s, 3);
    ~proxy = ProxySpace(s);
});

 // Receive OSC: read values into control bus
~receiver = OSCdef(
    key: \gyrUpdate,
    func: {|msg| ~controlBus.setnSynchronous(msg[1..]);},
    path: '\readGyr',
    srcID: ~local
);
)

```

Server and control bus setup

A task loop allows us to read from the serial bus continually, one byte at a time. The data in the serial bus are strings: sequences of decimal digit characters followed by the axis to which those digits belong (i.e. x, y, or z). A string 1, 4, 3, 9, ‘y’, for example, would correspond to a y-value of 1,439. To parse this data, we start with an empty character array and append any decimal digit characters. The negative sign character ‘-’ at the start of a digit sequence indicates that whatever final integer value that results should be multiplied by -1. Once an x, y, or z is seen, the character array is converted to an integer, normalized to the parameter space, and written to the corresponding channel on the control bus.

```
(
~charArray = [ ];
~getValue = Routine.new({
    var ascii;
    var neg;
    var valueScaler = 2048.0;
    var startOfVal = true;
    var max = 0;
    var accMax = 15.75;

    {
        ascii = ~port.read.asAscii;

        if(startOfVal, {
            if(ascii == $-, {neg = -1; startOfVal = false;}, {neg = 1; startOfVal = false;});
        });

        if(ascii.isDecDigit, {~charArray = ~charArray.add(ascii);});

        if(ascii == $x, {
            ~valx = ~charArray.collect(_.digit).convertDigits;
            ~valx = ~valx * neg;
            ~valx = ~valx / valueScaler;
            ~valx = ~valx / accMax;

            ~charArray = [ ];

            startOfVal = true;
        });
        if(ascii == $y, {
            ~valy = ~charArray.collect(_.digit).convertDigits;
            ~valy - ~valy * neg;
            ~valy = ~valy / valueScaler;
            ~valy = ~valy / accMax;

            ~charArray = [ ];

            startOfVal = true;
        });
        if(ascii == $z, {
            ~valz = ~charArray.collect(_.digit).convertDigits;
            ~valz - ~valz * neg;
            ~valz = ~valz / valueScaler;
            ~valz = ~valz / accMax;

            ~charArray = [ ];

            startOfVal = true;
        });

        ~local.sendMsg('\readGyr', ~valx, ~valy, ~valz);
    }.loop;

}).play;
)
```

Reading from the serial bus

After reading an audio buffer onto the server, a function scans through the buffer at a given sample, modulated by the accelerometer roll value. This allows the user to scrub through the sound forwards or backwards by turning their wrist. The pitch angle controls the amount of reverb on the sound before output to the audio bus.

```
~path = "/Users/stlp/Documents/UW/DXARTS 499/Field Recordings/FOA/SPTL023.WAV"; // personal field recording

// read buffer for sample playback
~buffer = Buffer.readChannel(s, ~path, channels: 0);

~buffer.plot; // verify that buffer contains signal

(
// playhead is a pointer to a sample in the buffer
~proxy.ar(\playheadPosition, 1); // define audio-rate proxy
~proxy[\playheadPosition] = 0; // start at beginning of sample

// reverb mod
~proxy.ar(\rotation, 1); // audio-rate proxy for rotation parameter
~proxy[\rotation] = 0; // start rotation at zero
)

// define output
(
~proxy[\out] = {
    var snd, snd2; // L and R channels
    var out;       // output
    var buf = ~buffer; // sound to process

    // read the contents of the input buffer at a given sample
    snd = BufRd.ar(1, buf, ~proxy[\playheadPosition] * BufFrames.kr(buf));
    snd2 = BufRd.ar(1, buf, ~proxy[\playheadPosition] * BufFrames.kr(buf) * LFNoise2.kr(0.1).range(0.99, 1.01));

    // apply reverb to output signal and control
    // wet/dry mix with parameter proxy
    FreeVerb2.ar(snd, snd2, mix: ~proxy[\rotation], room: 0.4);
}
)

~proxy[\out].free; // stop playback

// start sample playback
~proxy[\out].play;

~proxy[\out].scope; // view output

// map x-acc to playhead position
~proxy[\playheadPosition] = {Slew.ar(~proxy[\gyr].ar(1, 0, 0/*X axis*/).lag3(0.3).linlin(-1, 1, 0, 1), 1, 1)};

// map y-acc to rotation angle
~proxy[\rotation] = {Slew.ar(~proxy[\gyr].ar(1, 1, 0/*Y axis*/).lag3(0.3).linlin(-1, 1, 0, 1, clip: \min))};
```

Sample processing: scrub and reverb

Overall Performance Summary:

Our in-lab demonstration went very well. We had worried about setting up demonstrations for parts 4-2 and 4-3 on a single board, but we were able to set up the hardware properly and switch quickly between section demos. Our 4-2 tasks all behaved correctly, and we were able to calculate the FFT of a random signal of 128 samples. Our final project went smoothly, without any wire disconnections or synth server crashes that would have required restarting the software. We are pleased with the results. Not only did we learn a great deal from this lab, but we were able to build something interesting and fun to use!

Teamwork Breakdown:

This lab presented a unique challenge: Justin attended a robotics competition in Alberta, Canada during the last week of class. Despite this, we were able to coordinate work remotely, with Justin attending lab demonstrations via Zoom. We worked on 4-2 in parallel, noting any major changes we made to the code. In hindsight, some form of version control would have been useful. When progress stalled on 4-2, we switched focus to our final project. Justin researched I2C devices and relevant libraries, while Emmett read up on OSC protocol and serial communication. Once we had settled on our hardware setup, Justin switched to debugging 4-2, while Emmett worked in SuperCollider.

Discussion and Conclusions:

Our initial goal for the project was to use a Nintendo WiiMote rather than the gy-521 module for the user interface. Even though we weren't able to accomplish this, we still feel that our project was a success, and we learned a great deal in the process. The WiFi difficulty set our progress back a few days, but we are proud of the solution we came up with as a result. Writing our own process to generate OSC messages locally in SuperCollider was more rewarding than using existing libraries. We feel especially proud of our recovery after days of being stuck. This lab has made us stronger troubleshooters. In addition to what we learned about using FreeRTOS, the process of brainstorming hardware and library alternatives when we hit a dead end was a great experience.