ECE 4760 Lab 1: Synthesizing and Synchronizing Snowy Tree Crickets

[redacted]

Introduction

The purpose of this lab was to synthesize two artificial Snowy Tree crickets on the Raspberry Pi Pico, a Software Development Kit (SDK) for the RP2040. In order to do this, two sine waves of a certain frequency were synthesized using direct digital synthesis. To separate threads, each sine wave was attributed to each of the two cores of the RP2040 in the Raspberry Pi Pico. From the Raspberry Pi Pico, the signal was transmitted to a DAC and audibly verified via a microphone. Then to replicate a cricket chirp, the amplitude of the waves were modulated through a core specific state machine to produce a series of syllables–a chirp–and pauses between syllables as well as between chirps. Crickets have the ability to synchronize their chirps with each other. In order to implement this capability, the microphone audio was sampled through an ADC and calculated through a Fast Fourier Transform (FFT). Utilizing this functionality as well as additional code, the RP2040 cores were able to detect each others' chirps as well as chirps that the cores did not produce. Finally, to synchronize chirps of cores or outside parties that were not in sync, a synchronization algorithm was also implemented within the code for calculating the FFT. Therefore, produced chirps other than its own could be recognized by the RP2040 core and would synchronize with each other over time.

Design and Testing Methods

Concept and Implementation Overview

With the ultimate goal of creating a cricket which can chirp and synchronize with other crickets just like an actual snowy tree cricket, it makes sense to first produce a realistic cricket chirp. The first half of this lab focused on producing a cricket chirp from each core on the Raspberry Pi Pico, simultaneously. When that was achieved, the lab moved its focus onto chirp *detection*, so that the cores could know when to synchronize to another cricket. When chirp detection was achieved, the synchronization algorithm was implemented, completing this lab. Below is a table summarizing the benchmarks achieved over time.

Week	Software Outcome	Hardware Outcome	Testing Method
1	Used Direct Digital Synthesis to output beeps on each of the two cores of the RP2040, simultaneously.	Connected GPIO pins on RP2040 to Digital to Analog Converter (DAC) and speakers.	Displayed the outputs from the DAC onto an oscilloscope to verify correct frequencies. Also connected and listened to outputs with speakers to ensure hardware connections.
2	Used Direct Digital Synthesis to generate cricket chirps from each core on the	Added buttons to pause each core's chirp.	Used an oscilloscope to verify chirp frequencies as well as syllable and pause

	RP2040. Created a state machine to correctly time the cricket chirps and pauses.		lengths. As for the hardware, manually tested it by pressing the button and noting pauses.
3	Added Fast Fourier Transform (FFT) algorithm to implement chirp detection on each core.	Used a VGA to display FFT spectra on a screen. Added a microphone so cores can hear other crickets. Connected SDK to a serial monitor to see when a chirp is detected.	Used a VGA display to test hardware connections and FFT code. Used a serial monitor to output both chirp detections and current states of cores during chirp detection.
4	Implemented the Mirollo and Strogatz synchronization algorithm.		Used an oscilloscope to watch chirps synchronize from a desynchronized state.

Fig. 1: Summary of software, hardware, and testing benchmarks achieved each week throughout the lab

Algorithms

Three main algorithms were used to support the final program: Direct Digital Synthesis (DDS), Fast Fourier Transform (FFT), and a synchronization algorithm by Mirollo and Strogatz (1990).

I. *Direct Digital Synthesis*

As aforementioned, DDS is used to generate amplitude modulated sine waves–the cricket sound. The cornerstone of DDS is that "a variable overflowing is isomorphic [the same or similar] to one rotation of a phasor."[1] In this lab, an accumulator, a 32 bit number, represents the angle of the phasor, where one rotation of the phasor is equivalent to an addition to the accumulator. To calculate the desired sine wave frequency, we use this methodology along with some dimensional analysis related to audio sampling.

$$F_{out} = \frac{1 \text{ overflow (i.e. sine period)}}{2^{32} \text{ accumulator units}} \cdot \frac{j \text{ accumulator units}}{1 \text{ audio sample}} \cdot \frac{F_s \text{ audio samples}}{1 \text{ sec}}$$
$$= \left(\frac{F_s}{2^{32}} \cdot N \right) Hz$$

Since F_s is known, the equation can be rearranged to find the only unknown left–N, which is the increment value. Note that F_s corresponds to the rate of generated audio samples that will be sent to the DAC. The final equation calculates N.

$$\boxed{N = \text{increment amount} = \frac{F_{out}}{F_s} \cdot 2^{32}}$$

For the sound to be generated correctly, there is a timer interrupt in the code that will fire and increment the accumulator. The increment amount calculation from above is used to check the

[1] Adams, H. (n.d.). Direct Digital Synthesis. ECE 4760 , Hunter Van Adams. Retrieved September 18, 2022, from https://vanhunteradams.com/DDS/DDS.html

value of a lookup table (the sine wave value at the specific phasor angle). Since the accumulator variable is 32 bits, it can have 2^{32} states. However, the power spectrum for 256 entries shows enough distance between the frequency of interest and the 1st error harmonic for a close approximation, even though there is still distortion from less entries. Therefore, only 8 bits are needed to index into the lookup table. The lookup value is stored and then sent to the DAC.

II. *Fast Fourier Transform*

The purpose of a fast fourier transform is to transfer a signal from one domain to some representation in the frequency domain. The FFT can also transfer the signal from the representation of the frequency domain back to the original domain. In this lab, the sine-cosine series is related to the Cooley-Tukey FFT. The sine-cosine series, f(t), is composed of a sum of sines and cosine, which can approximate any periodic function of period T_0.[2]

$$f(t) = \frac{a_0}{2} + \sum_{n=1,2,\ldots}^{\infty} a_n \cos\left(\frac{2\pi n}{T_0}t\right) + \sum_{n=1,2,\ldots}^{\infty} b_n \sin\left(\frac{2\pi n}{T_0}t\right)$$

Furthermore, a_m and a_n can be expressed as follows.

$$a_m = \frac{2}{T_0}\int_{t_0}^{t_0+T_0} \cos\left(\frac{2\pi m}{T_0}t\right) f(t)dt$$
$$b_m = \frac{2}{T_0}\int_{t_0}^{t_0+T_0} \sin\left(\frac{2\pi m}{T_0}t\right) f(t)dt$$

The sine-cosine series can be expressed exponentially as the following:

$$f(t) = \sum_{n=-\infty}^{\infty} \underline{c}_n e^{\frac{2\pi n i}{T_0}t}$$

$$c_n = \frac{1}{T_0}\int_{t_0}^{t_0+T_0} f(t)e^{-\frac{2\pi m i}{T_0}t}dt$$

In this lab, some continuous function is discretely sampled with an ADC. To computationally calculate the FFT, the infinite exponential sum was transformed into a "finite sum over sampled points."[3]

$$f_k = \sum_{n=0}^{2N-1} \underline{c}_n e^{\frac{2\pi n i}{T_0}t_k}$$

$$\underline{c}_m = \frac{1}{2N}\sum_{k=0}^{2N-1} e^{-\frac{2\pi m i}{T_0}t_k} f_k$$

Finally, the Cooley-Tukey method is used to further split sums into sums of sums.

---

[2] Adams, H. (n.d.). Understanding the Cooley-Tukey FFT. ECE 4760, van Hunter Adams. Retrieved September 18, 2022, from https://vanhunteradams.com/FFT/FFT.html

[3] Adams, H. (n.d.). Understanding the Cooley-Tukey FFT.

$$
\begin{aligned}
\underline{c}_m &= \frac{1}{2N}\left[\left[\left(\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k} + e^{-\frac{2\pi i}{N/2}m}\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+4}\right)\right.\right. \\
&\left.\left. + e^{-\frac{2\pi i}{N}m}\left(\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+2} + e^{-\frac{2\pi i}{N/2}m}\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+6}\right)\right]\right. \\
&\left. + e^{-\frac{2\pi i}{2N}m}\left[\left(\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+1} + e^{-\frac{2\pi i}{N/2}m}\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+5}\right)\right.\right. \\
&\left.\left. + e^{-\frac{2\pi i}{N}m}\left(\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+3} + e^{-\frac{2\pi i}{N/2}m}\sum_{k=0}^{\frac{N}{4}-1} e^{-\frac{2\pi ki}{N/2}m} f_{8k+7}\right)\right]\right] \\
&= \frac{1}{2N}\left[\left[\left(F^{eee} + e^{-\frac{2\pi i}{N/2}m}F^{eeo}\right) + e^{-\frac{2\pi i}{N}m}\left(F^{eoe} + e^{-\frac{2\pi i}{N/2}m}F^{eoo}\right)\right]\right. \\
&\left. + e^{-\frac{2\pi i}{2N}m}\left[\left(F^{oee} + e^{-\frac{2\pi i}{2N}m}F^{oeo}\right) + e^{-\frac{2\pi i}{N}m}\left(F^{ooe} + e^{-\frac{2\pi i}{N/2}m}F^{ooo}\right)\right]\right]
\end{aligned}
$$

Furthermore, this algorithm's main abstraction is that if the sample number is a power of two, the length of each of the resulting summations will be 1. Transformations of this length in the case of one audio sample results in an input replicated as the output. It turns out that the reordering of cases with more than one sample is synonymous to bit-reversal computations, meaning that the bits of a sample are mirrored such that 001 becomes 100, 111 becomes 111, and 110 becomes 011.

III. Synchronization

An integrate-and-fire oscillator can be defined as "a system that integrates a function until some threshold value is reached, at which point the system 'fires' and the integrator is reset to zero."[4] In this case, the function that the oscillators travel is $y = \sqrt{x}$, which is monotonic and concave. The other oscillators move up the curve by a certain number ε or fire once another oscillator fires. This process eventually results in synchronization. This is detailed further in the Mirollo and Strogatz 1990 paper.

In this lab, the calculation-on-event method is implemented. (1) First, a timer interrupt is set up. (2) Every time the timer interrupt fires, a counting variable increases. The firing threshold, a certain value of y, has a corresponding x-value. (3) Once the counting variable is greater than this x-value, the counting variable is set to zero and an oscillator fires. (4) Furthermore, when an oscillator fires, the square root of the counting variable is taken and ε is added to determine the new y-position. The new y-position is squared to find the new counting variable value. Step three occurs again depending on the counting variable.

Software Implementation

First focusing on using DDS to output a cricket chirp from each core, a state machine and a timer Interrupt Service Routine (ISR) were implemented on each core. Initial DDS code which outputted a beep

[4] Adams, H. (n.d.-b). Synchronization of integrate-and-fire oscillators. ECE 4760, Hunter Van Adams. Retrieved September 18, 2022, from https://vanhunteradams.com/Pico/Cricket/Synchronization.html

at a specific frequency in week one was provided from the lab handout[5]. In week two, the goal was to alter the parameters of the DDS program to output a chirping sound. Furthermore, the chirp required multiple points to be met: 8 syllables, 2300 Hz syllable frequency, 17 ms syllable length, 2 ms syllable repeat interval (pause between syllables), and 750 ms chirp repeat interval (pause between chirps). The pause between chirps was initially 260 ms, but it was much easier to do the final demo with a longer pause.

To output a constant chirp sound, the phase increment variable which was conveniently declared before any logic in the program needed to be altered to reflect a 2300 Hz frequency.

```
115   // the DDS units - core 1
116   // Phase accumulator and phase increment. Increment sets output frequency.
117   volatile unsigned int phase_accum_main_1;
118   volatile unsigned int phase_incr_main_1 = (2300.0*two32)/Fs ; // chirp frequency = 2300
119   // the DDS units - core 2
120   // Phase accumulator and phase increment. Increment sets output frequency.
121   volatile unsigned int phase_accum_main_0;
122   volatile unsigned int phase_incr_main_0 = (2300.0*two32)/Fs ; // chrip frequency = 2300
```

Fig. 2: Phase Increment variables were changed to output a 2300 Hz frequency

To follow the timing restrictions of these chirps, the timing parameters defined at the top of the program were changed, and a state machine to follow these timing parameters was implemented.

```
139   // Timing parameters for beeps (units of interrupts)
140   #define ATTACK_TIME             200
141   #define DECAY_TIME              200
142   #define SUSTAIN_TIME            10000
143   #define BEEP_DURATION           680
144   #define BEEP_REPEAT_INTERVAL    80
145   #define CHIRP_REPEAT_INTERVAL   30000
```

Fig. 3: Timing parameters for DDS were changed to output a cricket chirp. There are 40,000 interrupts in one second, so the 17 ms syllable length is reflected in BEEP_DURATION, the 2 ms syllable pause between syllables is reflected in BEEP_REPEAT_INTERVAL, and the 750 ms pause between chirps is reflected in CHIRP_REPEAT_INTERVAL.

With the timing parameters defined in units of ISR interrupts, the following state machine was designed.

[5] Adams, H. (n.d.-a). Synthesizing and Synchronizing Snowy Tree Crickets. ECE 4760, van Hunter Adams. Retrieved September 18, 2022, from https://vanhunteradams.com/Pico/Cricket/Crickets.html

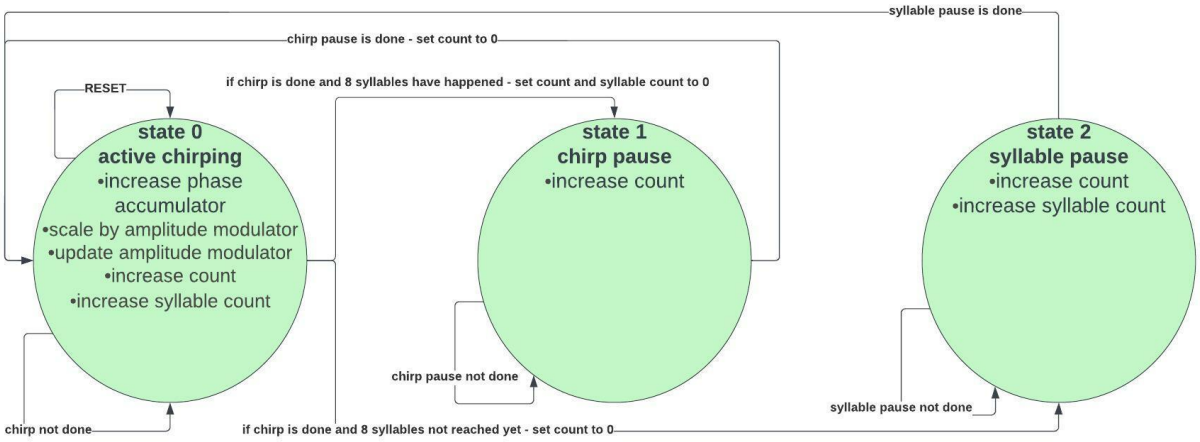

Fig. 4. State diagram for the timer ISR on each core

Timer ISRs were used to iterate through these state machines, incrementing a counter with every ISR iteration. In other words, each state machine was implemented inside an ISR. Depending on which state the core was currently in, when the counter reached the value of the durations defined in Fig. 4, the next state was entered. During the pause states, the DAC output amplitude would be set to zero so that a zero signal would be output on the speakers. Each core has its own ISR and outputs a chirp independently of the other chirp, but the ISRs of the cores are identical except that each core has its own set of variables denoted by a "_1" or "_0" suffix. See lines 188-348 in the program in Appendix.

To implement a manual pause feature with buttons, logic was added into the top of the ISRs such that when a button was pressed, the core would be stuck outside of the state machine and the counter would be set to zero. The core would restart at the beginning of the chirp state when the button was released.

Now focusing on chirp detection using FFT, once again an initial program with a working FFT algorithm was provided in the lab handout. The function FFTfix( ) performs the actual FFT algorithm on fixed point inputs. A proto-thread on core 0 called protothread_fft() then used the samples from the FFT algorithm function to locate the maximum frequency. One thread can be used for chirp detection on both cores since the chirp state machine states are global variables.

Detecting a chirp at frequency 2300 Hz means using an if statement when the maximum frequency is 2300 Hz. When this frequency is detected, each core accesses its ISR state machine state variable and determines whether or not it itself outputs that chirp. When the core is in a paused state or is being paused, then it "knows" that it did not output the chirp that was heard, and therefore an outside chirp was detected. Since using the Direct Memory Access (DMA) is time consuming and could outlast the end of a paused state or chirp state, the states of the cores are checked both before and after using the DMA.

```
525         //CHECK FOR CHIRP
526         if(max_freqency >= 2200 && max_freqency <= 2400) {
527             //does core 0 detect a chirp?
528             //LOGIC: if core is not in active chirping state, or if the core is paused and it hears a chirp,
529             //it means that it did not output that chirp
530             if(gpio_get(PAUSE_0) != 0 || before_0 != 0 || after_0 != 0) {
```

Fig. 5: Chirp detection login inside of FFT Thread. If the chirp frequency is heard, then the cores check whether they are paused and what their states were before and after using the DMA. For now imagine that line 531 outputs a "chirp detected on core 0" message to the serial monitor. There is also an identical if statement for core 1, meaning that chirps can be detected from both cores at the same time.

To synchronize a core to a detected cricket chirp, the synchronization algorithm was implemented in the code:

```
530             if(gpio_get(PAUSE_0) != 0 || before_0 != 0 || after_0 != 0) {
531
532                 if(STATE_0 == 1) { //if in pause
533                     //disable isr
534                     spin0 = spin_lock_blocking(myspinlock);
535
536                     // change count for synch algo
537                     y_0 = (int)sqrt(count_0) + EPSILON;
538                     count_0 = y_0 * y_0;
539
540                     //enable isr
541                     spin_unlock(myspinlock, spin0);
542                 }
543             }
```

Fig. 6: Inside the chirp detection if statement, i.e. when a chirp is detected from a core, if the core is in between chirps, then it increases the counter incrementing in the ISR as described in the synchronization algorithm. After some testing, EPSILON was set to 20.

Notice the spin locking in Figure 6. Spin locking is used to disable the ISR before changing the counter. Without spin locking, the counter could be assigned a value at two different points in the program at the same time. This double assignment could result in egregious errors. The variables necessary for spin locking are assigned in the top portion of the code with the other global variable declarations.

Finally, everything is tied together in main( ) and core1_entry( ). See the code start at line 582 for this protocol.

Hardware Implementation

The final circuit includes multiple external components wired to the Raspberry Pi Pico. Below is a summary of each component as well as a full schematic.

Component	**Purpose**
Digital-to-Analog	Converts the digital chirp signals from the Raspberry Pi Pico to analog signals

Converter (MCP4802)	which can be played on speakers
Push Button (2)	One for each core; when pressed, the chirps pause and restart
Audio Socket	Connected signals from DAC to Audio Jack into which a speaker can plug
USB-to-UART (SJ1-355XNG)	Converts serial print data from Raspberry Pi Pico to a USB that can be connected to a Serial Monitor, for example the Arduino IDE Serial Monitor.
Microphone (MAX4466)	Detects outside chirps
VGA connector	Displays FFT spectra onto a screen

Fig. 7. Summary of components in final circuit excluding the Raspberry Pi Pico and resistors

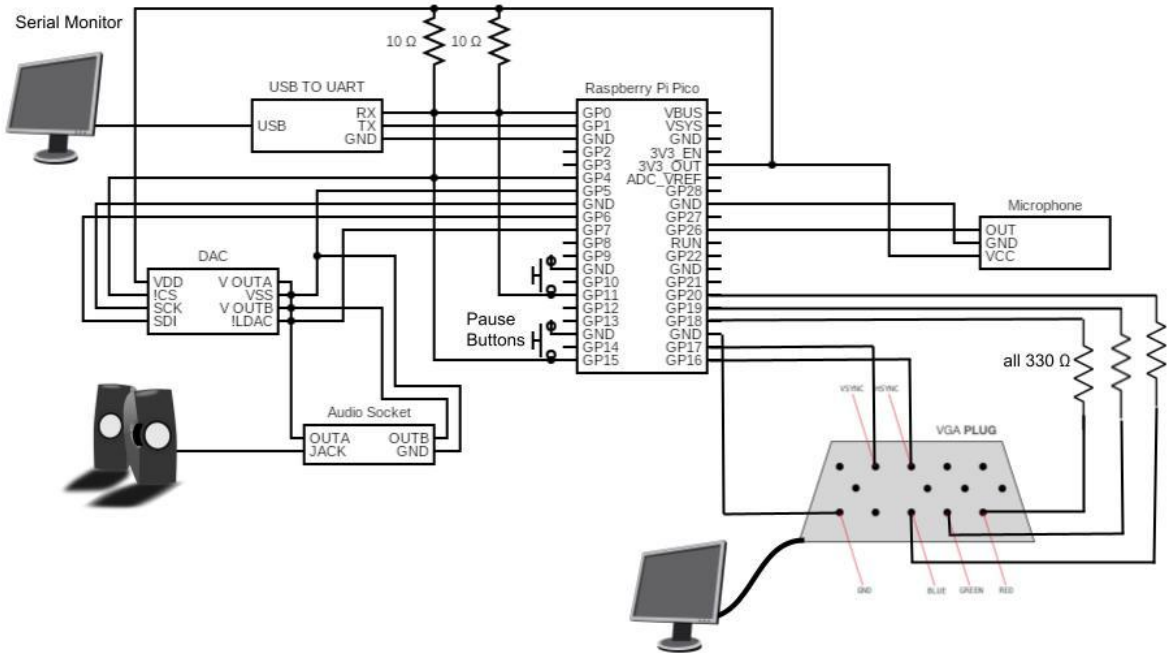

Fig. 8. Complete circuit diagram of the system

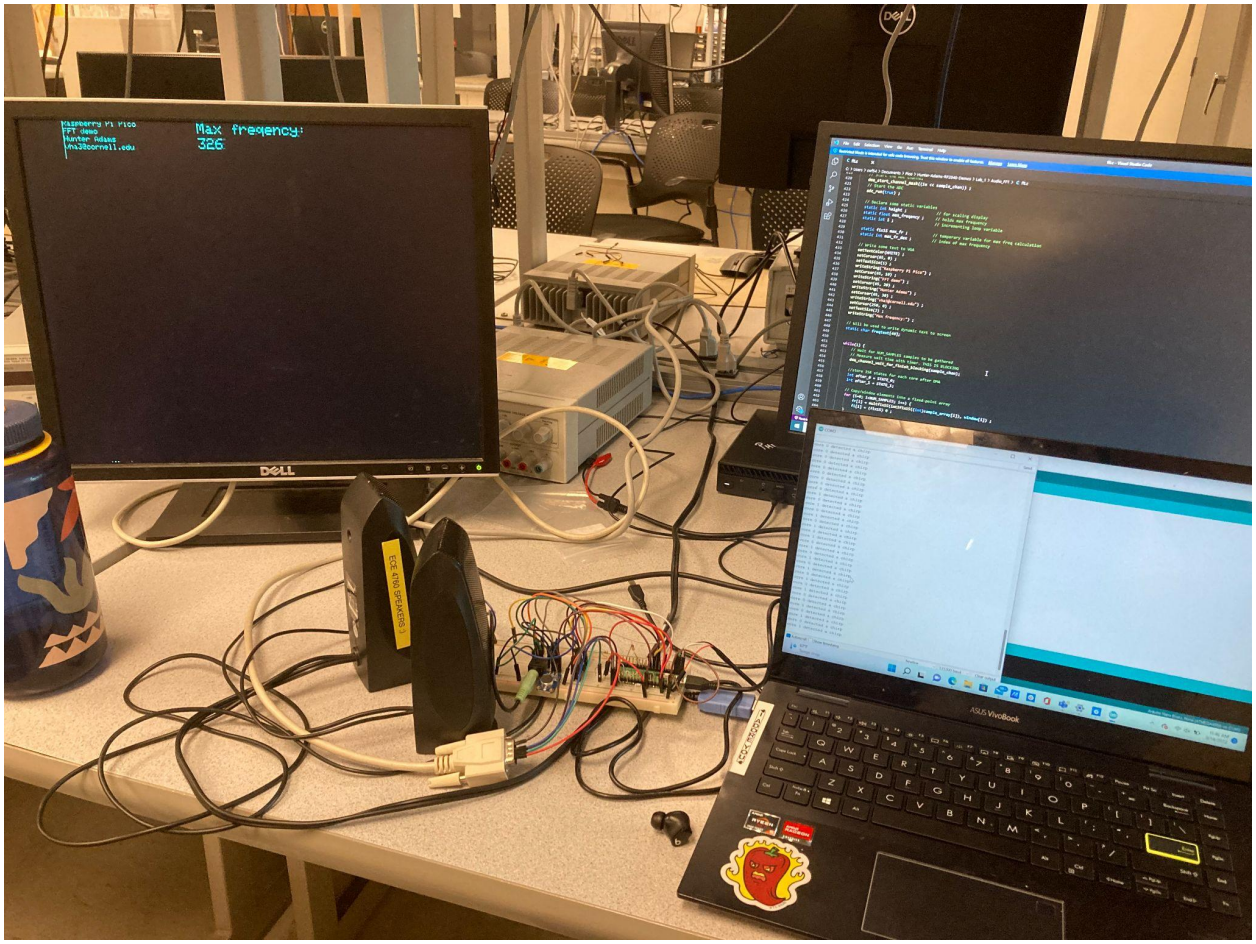

Fig. 9. Complete circuit and set up in the lab

Testing

A summary of the testing methods used is shown in Figure 1. The most important testing devices were: the oscilloscope, the serial monitor to view print statements, and the audio output from the speakers.

While debugging the chirp detection, in addition to printing to the serial monitor which core had detected a chirp, the state of the core during the detection was also outputted. There were errors with detecting a chirp at clearly wrong times. Namely, the biggest bug was the case where one core was paused with the pause button meaning that the paused core should be the only core detecting a chirp, but still both cores would detect chirps. There seemed to be issues with the time it took to change states and how long it took to use the DMA, so printing which state a core was in before and after the DMA was used was very helpful for this situation.

The speakers were helpful in situations in which something was very wrong. There were a few times when variables were wrongfully assigned, and the speakers would output obnoxious noise rather than anything close to a chirp. It was also good to be able to tell audibly whether two crickets were synchronizing.

That being said about the speakers, the oscilloscope was much more helpful in diagnosing errors and seeing that the crickets were indeed synchronizing. It will be discussed in the Results and Conclusion sections that there is some weird behavior with the final synchronizing. Without an oscilloscope to clearly display these waves, it would be very difficult to approach the issue. Furthermore, it is known exactly what a cricket chirp signal should look like in terms of the frequency, the syllable length, the pause length, etc. So when a signal does not look like that on a scope it is clear that something is wrong. On the other hand, when the signals do appear as expected, it is a reason to celebrate.

As an example, here are some snapshots of the oscilloscope displaying proper cricket chirps:

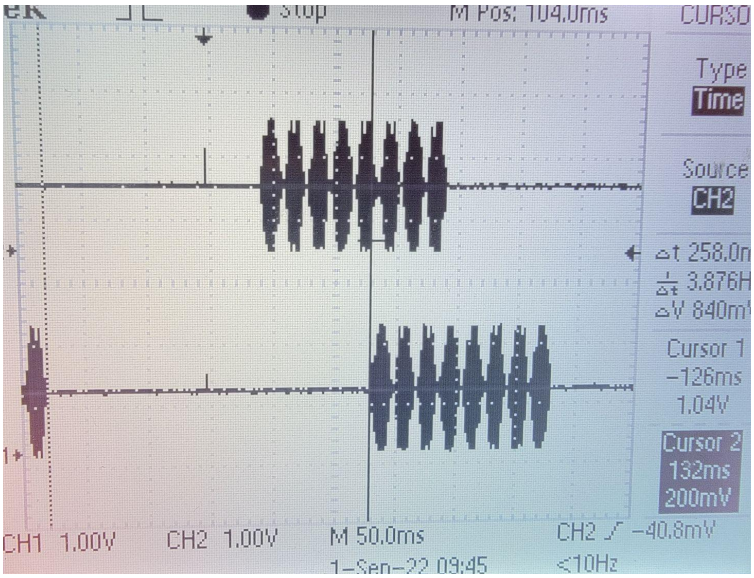

Fig. 10. Signal detected via the oscilloscope depicting 8 syllable chirp and 260 ms chirp pause time duration. This 260 ms pause is from before the pause duration was changed to 750 ms.

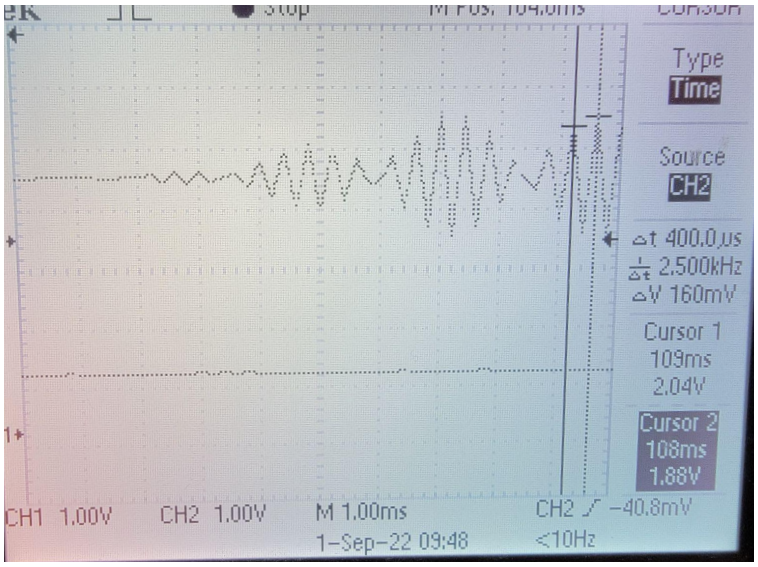

Fig. 11. Signal detected via oscilloscope depicting the 2300 Hz frequency. The system consistently outputs signals 200 Hz lower than expected, so 200 was added to the max frequency on line 524. This is why a frequency of 2500 Hz is seen here.

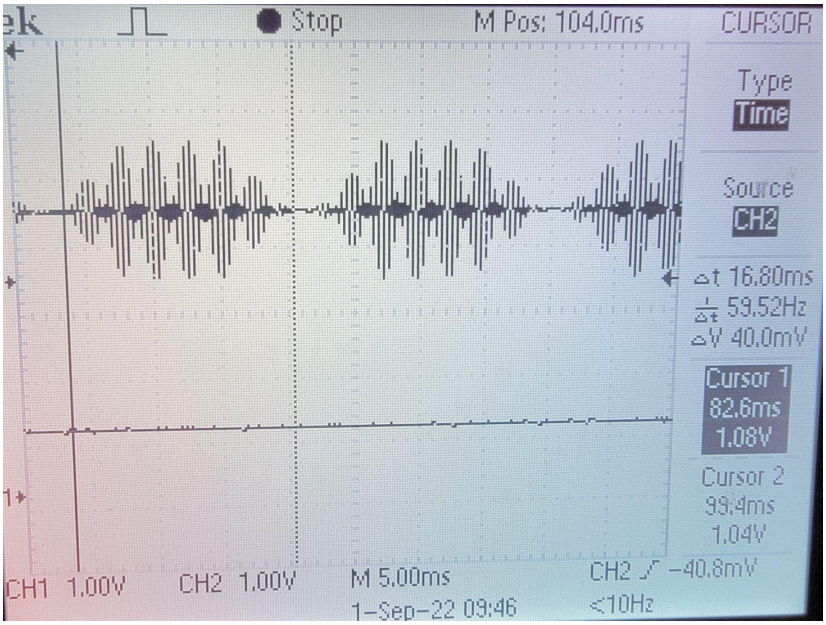

Fig. 12. Signal detected via oscilloscope depicting 17 ms syllable time duration

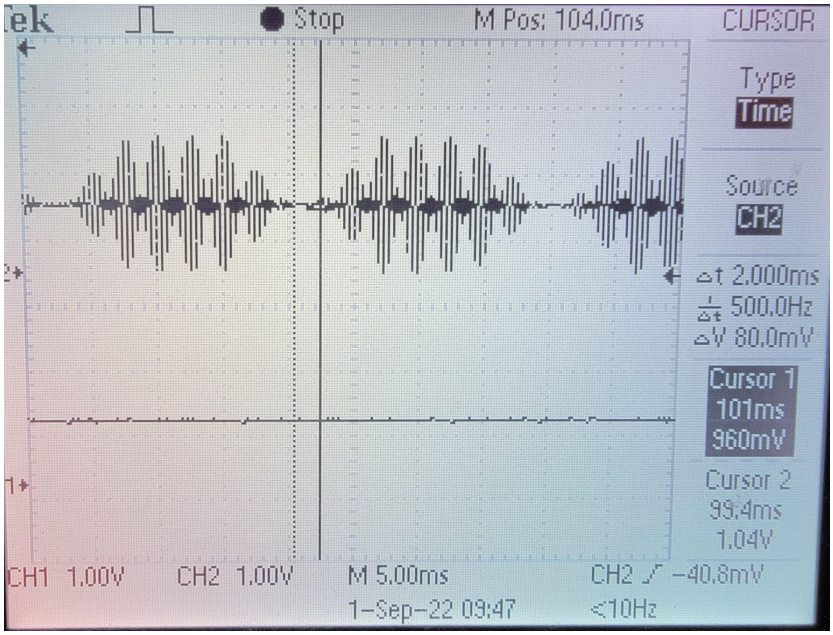

Fig. 13. Signal detected via oscilloscope depicting 2 ms syllable pause time duration

Lastly, since the lab room was often being used by multiple groups at a time, all outputting cricket chirps at 2300 Hz, it was helpful to test the system with different frequencies being output and detected by the cores. This method, suggested by a fellow student, ensured that the cricket chirps being detected or output were from only the cores on this system.

Results

The objective of this lab was to simulate snowy tree crickets that would generate chirps and synchronize to other chirps. Not only should the cores on the device synchronize with each other, but they should also synchronize with any outside crickets chirping in the vicinity of the microphone.

During the final lab checkout, the system was tested for reliability. When the system is first turned on, both chirps from cores zero and one will be synchronized both audibly and on the oscilloscope. Then, either core zero or one would be paused, thus desynchronizing the chirps. After the button is released, the two chirps will begin to move closer in sync with one another and therefore become synchronized again.

To test the accuracy of the system, these signals were viewed on an oscilloscope. As described in the Testing section, the oscilloscope displayed each produced chirp as well as how well the chirps were synchronizing.

Through this, it was noticed that the system had some variations in the time the signals took to synchronize again. In some cases it would take less than 10 seconds for the two signals to match. The expected synching behavior and resulting sync can be seen in Fig. 14. In other cases, it would be nearly a minute long – which is much slower than hoped. In this case, the one signal trace lagged behind and did not look like expected synchronization behavior. This can be seen in Fig. 15.

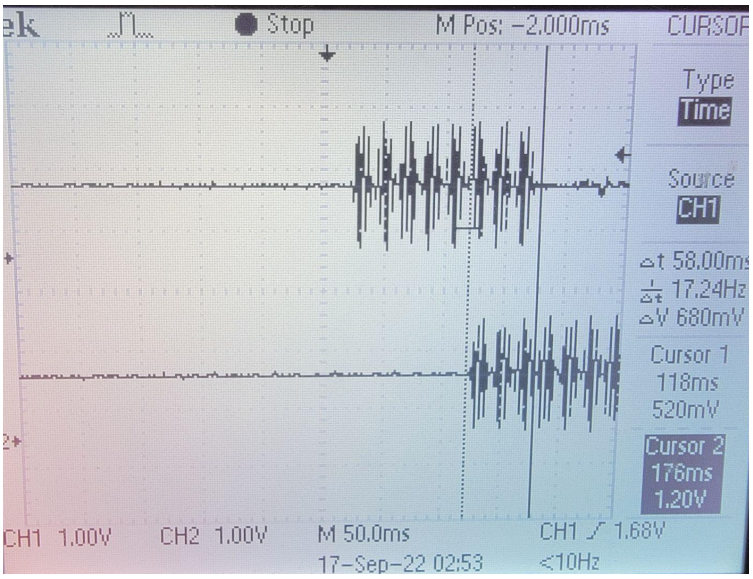

Fig. 14. Signals from both RP2040 cores depicting synchronization when Core 0 leads

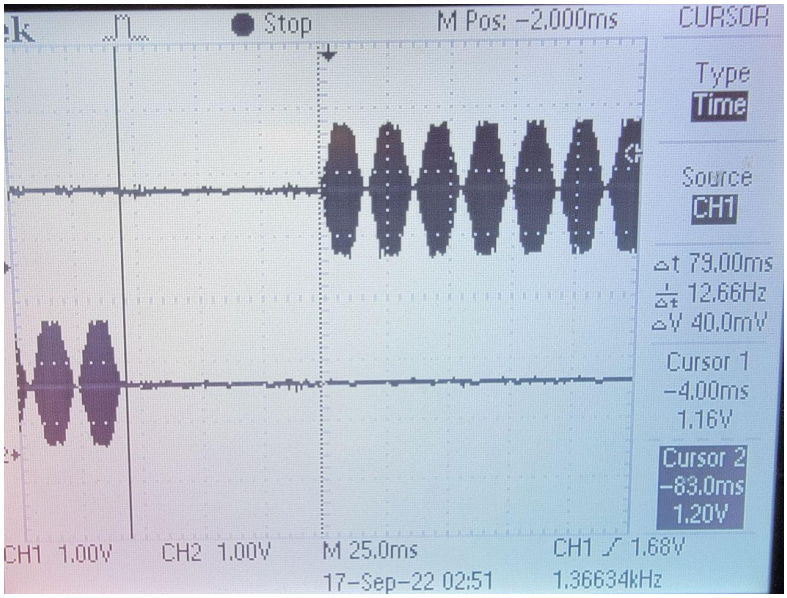

Fig. 15. Signals from both RP2040 cores depicting synchronization when Core 1 leads

An area of improvement for this lab would be centered around timing. As stated before, there were some disparities when it came to the timing of synchronization. In the process of desynchronization, when a button was pressed to pause core one, the time it took to resynchronize with core zero's chirp was short, often around 10 seconds. However, when core was paused via button, the time it took for core one's chirp to match core zero's chirp was about a minute. Though eventually both instances eventually led to synchronization, it is an issue that could be explored further. Although this behavior is without explanation for now, the crickets do indeed synchronize with both each other and outside crickets.

Conclusion

By the end of this lab, the system could successfully detect and synchronize to chirps from anywhere, including chirps outside of the system. The system, which consisted of just a few components, provided insights not only about direct digital synthesis, but allowed exploration of the RP2040's multicore system, which will come in handy for future projects.

Through this lab, a large breadth of topics from elements within the RP2040 to mathematical algorithms that generate realistic, digital sounds were covered. Learning about concepts such as Fast Fourier Transforms and synchronization algorithms required use of mathematical background. Furthermore, this background was also used in order to understand how the microcontroller can detect a call to one (or both) of its cores using interrupts and how that detection can be reflected by multiple systems around the lab. Learning about Direct Digital Synthesis only emphasizes how digital sounds can be produced to sound realistic. Even in coding the microcontroller, it was learned the hard way why syntax is important. Many times the system didn't produce a working output because there was one equal sign instead of two in an if statement condition.

Some of these many issues that were produced as a result of a missing equal sign included the output signal showing up as noise once a chirp was detected, having the chirp pauses changing amplitudes, and getting no output from the speakers. All of these were debugged by going through the code and seeing if the logic made sense, and if it did, then checking the syntax. The largest issue, which was described in the Results section, was a difference in timing when a specific core was paused. A way this could have been mitigated was by implementing a timer that would track where in the chirp the unpaused core, right before the paused core, is turned back on and that timer would basically act as a shift. But right now, it is unknown how that idea would work.

Further improvements that could have been made to the system would be taking into consideration optimization. For this system, producing a functional working system was prioritized over optimizing and figuring out the complexities of the code. Though not taken into consideration this time, for future labs, taking into account how certain implementations, like potentially adding a new timer, is something to look into.

A final thought and potential improvement for this lab is to allow the students to have less of the sample code. While there were cases where the algorithms were solved by the students, a lot of it felt as though it was already figured out in the sample code. To further deepen an understanding of the math and technology, having less on the sample code would've immersed students more into design thinking. Additionally, another interesting idea that could be added to the lab would be having crickets, whose chirps are harmonic, all synchronize with each other.

Appendix

Code:

```
/*
    final code for cricket chirp synchronization
    Kaitlyn Beiler (keb282), Wanda Field (cwf54), Nia Reid Vicars (nr346)
    ECE 4760 9/21/2022


* HARDWARE CONNECTIONS
DAC CONNECTIONS:
 *  - GPIO 4  ---> DAC MISO
 *  - GPIO 5  ---> DAC CS
 *  - GPIO 6  ---> DAC SCK
 *  - GPIO 7  ---> DAC MOSI
 *  - GPIO 8  ---> DAC LDAC
PAUSE BUTTONS:
 *  - GPIO 15 ---> CORE 1 PAUSE (left bottom button)
 *  - GPIO 11 ---> CORE 0 PAUSE (left middle button)
VGA CONNECTIONS:
 *  - GPIO 16 ---> VGA Hsync
 *  - GPIO 17 ---> VGA Vsync
 *  - GPIO 18 ---> 330 ohm resistor ---> VGA Red
 *  - GPIO 19 ---> 330 ohm resistor ---> VGA Green
 *  - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
 *  - RP2040 GND ---> VGA GND
 *  - GPIO 26 ---> Audio input [0-3.3V]
 *
```

```
 * RESOURCES USED
 *  - PIO state machines 0, 1, and 2 on PIO instance 0
 *  - DMA channels 0, 1, 2, and 3
 *  - ADC channel 0
 *  - 153.6 kBytes of RAM (for pixel color data)
 */

// Include VGA graphics library
#include "vga_graphics.h"
// Include standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
// Include Pico libraries
#include "pico/stdlib.h"
#include "pico/multicore.h"
// Include hardware libraries
#include "hardware/pio.h"
#include "hardware/dma.h"
#include "hardware/adc.h"
#include "hardware/irq.h"
#include "hardware/sync.h"
#include "hardware/spi.h"
// Include protothreads
#include "pt_cornell_rp2040_v1.h"

// === the fixed point macros (16.15) ======================================
typedef signed int fix15 ;
#define multfix15(a,b) ((fix15)((((signed long long)(a))*((signed long long)(b)))>>15))
#define float2fix15(a) ((fix15)((a)*32768.0)) // 2^15
#define fix2float15(a) ((float)(a)/32768.0)
#define absfix15(a) abs(a)
#define int2fix15(a) ((fix15)(a << 15))
#define fix2int15(a) ((int)(a >> 15))
#define char2fix15(a) (fix15)(((fix15)(a)) << 15)
#define divfix(a,b) (fix15)( (((signed long long)(a)) << 15) / (b))

/////////////////////////// ADC configuration ////////////////////////////////
// ADC Channel and pin
#define ADC_CHAN 0
#define ADC_PIN 26
// Number of samples per FFT
#define NUM_SAMPLES 1024
// Number of samples per FFT, minus 1
#define NUM_SAMPLES_M_1 1023
// Length of short (16 bits) minus log2 number of samples (10)
#define SHIFT_AMOUNT 6
// Log2 number of samples
#define LOG2_NUM_SAMPLES 10
// Sample rate (Hz)
#define FFT_Fs 10000  // fix - original 10000
// ADC clock rate (unmutable!)
#define ADCCLK 48000000.0

// DMA channels for sampling ADC (VGA driver uses 0 and 1)
int sample_chan = 2 ;
int control_chan = 3 ;

// Max and min macros
#define max(a,b) ((a>b)?a:b)
#define min(a,b) ((a<b)?a:b)

//spin lock for changing counter variable when chirp detected
spin_lock_t* myspinlock;
uint32_t spin0;
uint32_t spin1;

// 0.4 in fixed point (used for alpha max plus beta min)
fix15 zero_point_4 = float2fix15(0.4) ;
```

```
96
97  // Here's where we'll have the DMA channel put ADC samples
98  uint8_t sample_array[NUM_SAMPLES] ;
99  // And here's where we'll copy those samples for FFT calculation
100 fix15 fr[NUM_SAMPLES] ;
101 fix15 fi[NUM_SAMPLES] ;
102
103 // Sine table for the FFT calculation
104 fix15 Sinewave[NUM_SAMPLES];
105 // Hann window table for FFT calculation
106 fix15 window[NUM_SAMPLES];
107
108 // Pointer to address of start of sample buffer
109 uint8_t * sample_address_pointer = &sample_array[0] ;
110
111 //Direct Digital Synthesis (DDS) parameters
112 #define two32 4294967296.0  // 2^32 (a constant)
113 #define Fs 40000            // sample rate
114
115 // the DDS units - core 1
116 // Phase accumulator and phase increment. Increment sets output frequency.
117 volatile unsigned int phase_accum_main_1;
118 volatile unsigned int phase_incr_main_1 = (2300.0*two32)/Fs ; // chirp frequency = 2300
119 // the DDS units - core 2
120 // Phase accumulator and phase increment. Increment sets output frequency.
121 volatile unsigned int phase_accum_main_0;
122 volatile unsigned int phase_incr_main_0 = (2300.0*two32)/Fs ; // chrip frequency = 2300
123
124 // DDS sine table (populated in main())
125 #define sine_table_size 256
126 fix15 sin_table[sine_table_size] ;
127
128 // Values output to DAC
129 int DAC_output_0 ;
130 int DAC_output_1 ;
131
132 // Amplitude modulation parameters and variables
133 fix15 max_amplitude = int2fix15(1) ;    // maximum amplitude
134 fix15 attack_inc ;                      // rate at which sound ramps up
135 fix15 decay_inc ;                       // rate at which sound ramps down
136 fix15 current_amplitude_0 = 0 ;         // current amplitude (modified in ISR)
137 fix15 current_amplitude_1 = 0 ;         // current amplitude (modified in ISR)
138
139 // Timing parameters for beeps (units of interrupts)
140 #define ATTACK_TIME             200
141 #define DECAY_TIME              200
142 #define SUSTAIN_TIME            10000
143 #define BEEP_DURATION           680
144 #define BEEP_REPEAT_INTERVAL    80
145 #define CHIRP_REPEAT_INTERVAL   30000
146 #define EPSILON                 20 //in synthesis formula
147
148 // core 0 state machine and synthesis variables
149 volatile unsigned int STATE_0   = 0 ;
150 volatile unsigned int count_0   = 0 ;
151 volatile unsigned int syl_0     = 0 ;  //syllable counter
152 volatile unsigned int y_0       = 0 ;  //synthesis alg
153
154 // core 1 state machine and synthesis variables
155 volatile unsigned int STATE_1   = 0 ;
156 volatile unsigned int count_1   = 0 ;
157 volatile unsigned int syl_1     = 0 ;  //syllable counter
158 volatile unsigned int y_1       = 0 ;  //synthesis alg
159
160 // SPI data
161 uint16_t DAC_data_1 ; // output value
162 uint16_t DAC_data_0 ; // output value
163
164 // DAC parameters (see the DAC datasheet)
165 // A-channel, 1x, active
```

```
#define DAC_config_chan_A 0b0011000000000000
// B-channel, 1x, active
#define DAC_config_chan_B 0b1011000000000000

//SPI configurations (note these represent GPIO number, NOT pin number)
#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7
#define LDAC     8
#define LED      25
#define SPI_PORT spi0
#define PAUSE_1  15//LEFT BOTTOM BUTTON
#define PAUSE_0  11//LEFT MIDDLE BUTTON

// Two variables to store core number
volatile int corenum_0  ;
volatile int corenum_1  ;

// Global counter for spinlock experimenting
volatile int global_counter = 0 ;

// This timer ISR is called on core 1
// contains state machine, ISR is used to ensure states are times properly
bool repeating_timer_callback_core_1(struct repeating_timer *t) {
    // when pause button is pressed, stop outputting a chirp and start state machine over
    if(gpio_get(PAUSE_1) == 0) {
        STATE_1 = 0;
        count_1 = 0;
    }

    // in S = 0, cricket is actively chirping
    else if (STATE_1 == 0) {

        // DDS phase and sine table lookup
        phase_accum_main_1 += phase_incr_main_1  ;
        DAC_output_1 = fix2int15(multfix15(current_amplitude_1,
            sin_table[phase_accum_main_1>>24])) + 2048 ;

        // Ramp up amplitude
        if (count_1 < ATTACK_TIME) {
            current_amplitude_1 = (current_amplitude_1 + attack_inc) ;
        }
        // Ramp down amplitude
        else if (count_1 > BEEP_DURATION - DECAY_TIME) {
            current_amplitude_1 = (current_amplitude_1 - decay_inc) ;
        }

        // Mask with DAC control bits
        DAC_data_1 = (DAC_config_chan_A | (DAC_output_1 & 0xffff))  ;

        // SPI write (no spinlock b/c of SPI buffer)
        spi_write16_blocking(SPI_PORT, &DAC_data_1, 1) ;

        // Increment the counter
        count_1 += 1 ;

        // State transition: when chirp is over, check syllables to know which pause to go to
        if (count_1 == BEEP_DURATION) {
            // if full 8 chirps are done, go to long pause
            if(syl_1 == 7) {
                STATE_1 = 1 ;
                count_1 = 0 ;
                syl_1   = 0;
            }

            // otherwise go to short pause
            else {
                STATE_1 = 2 ;
                count_1 = 0 ;
```

```
                syl_1 += 1;
            }
        }
    }

    // S = 1 means long pause
    else if(STATE_1 == 1){
        count_1 += 1;
        // pause chirp
        current_amplitude_1 = 0;
        if (count_1 >= CHIRP_REPEAT_INTERVAL) {
            current_amplitude_1 = 0 ;
            STATE_1 = 0 ; // go back to chirp
            count_1 = 0 ;
            syl_1   = 0 ;
        }
    }
    // S = 2 means short pause between syllables
    else {
        count_1 += 1;
        // pause SYLLABLE
        current_amplitude_1 = 0;
        if (count_1 == BEEP_REPEAT_INTERVAL) {
            current_amplitude_1 = 0 ;
            STATE_1 = 0 ; // go back to chirp
            count_1 = 0 ;
        }
    }
    // retrieve core number of execution
    corenum_1 = get_core_num() ;
    return true;
}

// This timer ISR is called on core 0
// contains state machine, ISR is used to ensure states are times properly
bool repeating_timer_callback_core_0(struct repeating_timer *t) {
    // when pause button is pressed, stop outputting a chirp and start state machine over
    if(gpio_get(PAUSE_0) == 0) {
        STATE_0 = 0;
        count_0 = 0;
    }

    // in S = 0, cricket is actively chirping
    else if (STATE_0 == 0) {

        // DDS phase and sine table lookup
        phase_accum_main_0 += phase_incr_main_0  ;
        DAC_output_0 = fix2int15(multfix15(current_amplitude_0,
            sin_table[phase_accum_main_0>>24])) + 2048 ;

        // Ramp up amplitude
        if (count_0 < ATTACK_TIME) {
            current_amplitude_0 = (current_amplitude_0 + attack_inc) ;
        }
        // Ramp down amplitude
        else if (count_0 > BEEP_DURATION - DECAY_TIME) {
            current_amplitude_0 = (current_amplitude_0 - decay_inc) ;
        }

        // Mask with DAC control bits
        DAC_data_0 = (DAC_config_chan_B | (DAC_output_0 & 0xffff))  ;

        // SPI write (no spinlock b/c of SPI buffer)
        spi_write16_blocking(SPI_PORT, &DAC_data_0, 1) ;

        // Increment the counter
        count_0 += 1 ;

        // State transition: when chirp is over, check syllables to know which pause to go to
        if (count_0 == BEEP_DURATION) {
```

```
            // if full 8 chirps are done, go to long pause
            if(syl_0 == 7) {
                STATE_0 = 1 ;
                count_0 = 0 ;
                syl_0   = 0;
            }

            // otherwise go to short pause
            else {
                STATE_0 = 2 ;
                count_0 = 0 ;
                syl_0 += 1;
            }
        }
    }

    // S = 1 means long pause
    else if(STATE_0 == 1){
        count_0 += 1;
        // pause chirp
        current_amplitude_0 = 0;
        if (count_0 >= CHIRP_REPEAT_INTERVAL) {
            current_amplitude_0 = 0 ;
            STATE_0 = 0 ; // go back to chirp
            count_0 = 0 ;
            syl_0   = 0 ;
        }
    }
    // S = 2 means short pause between syllables
    else {
        count_0 += 1 ;
        // pause SYLLABLE
        current_amplitude_0 = 0;
        if (count_0 == BEEP_REPEAT_INTERVAL) {
            current_amplitude_0 = 0 ;
            STATE_0 = 0 ; // go back to chirp
            count_0 = 0 ;
        }
    }
    // retrieve core number of execution
    corenum_0 = get_core_num() ;
    return true;
}


// Peforms an in-place FFT. For more information about how this
// algorithm works, please see https://vanhunteradams.com/FFT/FFT.html
void FFTfix(fix15 fr[], fix15 fi[]) {

    unsigned short m;   // one of the indices being swapped
    unsigned short mr ; // the other index being swapped (r for reversed)
    fix15 tr, ti ; // for temporary storage while swapping, and during iteration

    int i, j ; // indices being combined in Danielson-Lanczos part of the algorithm
    int L ;    // length of the FFT's being combined
    int k ;    // used for looking up trig values from sine table

    int istep ; // length of the FFT which results from combining two FFT's

    fix15 wr, wi ; // trigonometric values from lookup table
    fix15 qr, qi ; // temporary variables used during DL part of the algorithm

    ////////////////////////////////////////////////////////////////////////
    ////////////////////////// BIT REVERSAL ////////////////////////////////
    ////////////////////////////////////////////////////////////////////////
    // Bit reversal code below based on that found here:
    // https://graphics.stanford.edu/~seander/bithacks.html#BitReverseObvious
    for (m=1; m<NUM_SAMPLES_M_1; m++) {
        // swap odd and even bits
        mr = ((m >> 1) & 0x5555) | ((m & 0x5555) << 1);
```

```
        // swap consecutive pairs
        mr = ((mr >> 2) & 0x3333) | ((mr & 0x3333) << 2);
        // swap nibbles ...
        mr = ((mr >> 4) & 0x0F0F) | ((mr & 0x0F0F) << 4);
        // swap bytes
        mr = ((mr >> 8) & 0x00FF) | ((mr & 0x00FF) << 8);
        // shift down mr
        mr >>= SHIFT_AMOUNT ;
        // don't swap that which has already been swapped
        if (mr<=m) continue ;
        // swap the bit-reveresed indices
        tr = fr[m] ;
        fr[m] = fr[mr] ;
        fr[mr] = tr ;
        ti = fi[m] ;
        fi[m] = fi[mr] ;
        fi[mr] = ti ;
    }
    //////////////////////////////////////////////////////////////////////////
    ////////////////////////// Danielson-Lanczos //////////////////////////////
    //////////////////////////////////////////////////////////////////////////
    // Adapted from code by:
    // Tom Roberts 11/8/89 and Malcolm Slaney 12/15/94 malcolm@interval.com
    // Length of the FFT's being combined (starts at 1)
    L = 1 ;
    // Log2 of number of samples, minus 1
    k = LOG2_NUM_SAMPLES - 1 ;
    // While the length of the FFT's being combined is less than the number
    // of gathered samples . . .
    while (L < NUM_SAMPLES) {
        // Determine the length of the FFT which will result from combining two FFT's
        istep = L<<1 ;
        // For each element in the FFT's that are being combined . . .
        for (m=0; m<L; ++m) {
            // Lookup the trig values for that element
            j = m << k ;                          // index of the sine table
            wr =  Sinewave[j + NUM_SAMPLES/4] ; // cos(2pi m/N)
            wi = -Sinewave[j] ;                 // sin(2pi m/N)
            wr >>= 1 ;                          // divide by two
            wi >>= 1 ;                          // divide by two
            // i gets the index of one of the FFT elements being combined
            for (i=m; i<NUM_SAMPLES; i+=istep) {
                // j gets the index of the FFT element being combined with i
                j = i + L ;
                // compute the trig terms (bottom half of the above matrix)
                tr = multfix15(wr, fr[j]) - multfix15(wi, fi[j]) ;
                ti = multfix15(wr, fi[j]) + multfix15(wi, fr[j]) ;
                // divide ith index elements by two (top half of above matrix)
                qr = fr[i]>>1 ;
                qi = fi[i]>>1 ;
                // compute the new values at each index
                fr[j] = qr - tr ;
                fi[j] = qi - ti ;
                fr[i] = qr + tr ;
                fi[i] = qi + ti ;
            }
        }
        --k ;
        L = istep ;
    }
}

// Runs on core 0
static PT_THREAD (protothread_fft(struct pt *pt))
{
    // Indicate beginning of thread
    PT_BEGIN(pt) ;
    printf("Starting capture\n") ;

    //store ISR states for each core before DMA
```

```
    unsigned int before_0 = STATE_0;
    unsigned int before_1 = STATE_1;
    // Start the ADC channel
    dma_start_channel_mask((1u << sample_chan)) ;
    // Start the ADC
    adc_run(true) ;

    // Declare some static variables
    static int height ;             // for scaling display
    static float max_freqency ;     // holds max frequency
    static int i ;                  // incrementing loop variable

    static fix15 max_fr ;           // temporary variable for max freq calculation
    static int max_fr_dex ;         // index of max frequency

    // Write some text to VGA
    setTextColor(WHITE) ;
    setCursor(65, 0) ;
    setTextSize(1) ;
    writeString("Raspberry Pi Pico") ;
    setCursor(65, 10) ;
    writeString("FFT demo") ;
    setCursor(65, 20) ;
    writeString("Hunter Adams") ;
    setCursor(65, 30) ;
    writeString("vha3@cornell.edu") ;
    setCursor(250, 0) ;
    setTextSize(2) ;
    writeString("Max freqency:") ;

    // Will be used to write dynamic text to screen
    static char freqtext[40];

    while(1) {
        // Wait for NUM_SAMPLES samples to be gathered
        // Measure wait time with timer. THIS IS BLOCKING
        dma_channel_wait_for_finish_blocking(sample_chan);

        //store ISR states for each core after DMA
        unsigned int after_0 = STATE_0;
        unsigned int after_1 = STATE_1;

        // Copy/window elements into a fixed-point array
        for (i=0; i<NUM_SAMPLES; i++) {
            fr[i] = multfix15(int2fix15((int)sample_array[i]), window[i]) ;
            fi[i] = (fix15) 0 ;
        }

        // Zero max frequency and max frequency index
        max_fr = 0 ;
        max_fr_dex = 0 ;

        //store ISR states for each core before DMA for next cycle
        unsigned int new_before_0  = STATE_0;
        unsigned int new_before_1  = STATE_1;

        // Restart the sample channel, now that we have our copy of the samples
        dma_channel_start(control_chan) ;

        // Compute the FFT
        FFTfix(fr, fi) ;

        // Find the magnitudes (alpha max plus beta min)
        for (int i = 0; i < (NUM_SAMPLES>>1); i++) {
            // get the approx magnitude
            fr[i] = abs(fr[i]);
            fi[i] = abs(fi[i]);
            // reuse fr to hold magnitude
            fr[i] = max(fr[i], fi[i]) +
                    multfix15(min(fr[i], fi[i]), zero_point_4);
```

```
            // Keep track of maximum
            if (fr[i] > max_fr && i>4) {
                max_fr = fr[i] ;
                max_fr_dex = i ;
            }
        }
        // Compute max frequency in Hz
        max_freqency = max_fr_dex * (FFT_Fs/NUM_SAMPLES) + 200 ; // our FFT is always 200 Hz low, consistently
        //CHECK FOR CHIRP
        if(max_freqency >= 2200 && max_freqency <= 2400) {
            //does core 0 detect a chirp?
            //LOGIC: if core is not in active chirping state, or if the core is paused and it hears a chirp,
            //it means that it did not output that chirp
            if(gpio_get(PAUSE_0) != 0 || before_0 != 0 || after_0 != 0) {

                if(STATE_0 == 1) { //if in pause
                    //disable isr
                    spin0 = spin_lock_blocking(myspinlock);

                    // change count for synch algo
                    y_0 = (int)sqrt(count_0) + EPSILON;
                    count_0 = y_0 * y_0;

                    //enable isr
                    spin_unlock(myspinlock, spin0);
                }
            }
            //does core 1 detect a chirp?
            if(gpio_get(PAUSE_1) != 0 || before_1 != 0 || after_1 != 0) {

                if(STATE_1 == 1) {
                    //disable isr
                    spin1 = spin_lock_blocking(myspinlock);

                    // change count for synch algo
                    y_1 = (int)sqrt(count_1) + EPSILON;
                    count_1 = y_1 * y_1;

                    //enable isr
                    spin_unlock(myspinlock, spin1);
                }
            }
        }
        // Display on VGA
        fillRect(250, 20, 176, 30, BLACK); // red box
        sprintf(freqtext, "%d", (int)max_freqency) ;
        setCursor(250, 20) ;
        setTextSize(2) ;
        writeString(freqtext) ;

        // Update the FFT display
        for (int i=5; i<(NUM_SAMPLES>>1); i++) {
            drawVLine(59+i, 50, 429, BLACK);
            height = fix2int15(multfix15(fr[i], int2fix15(36))) ;
            drawVLine(59+i, 479-height, height, WHITE);
        }

        //set vars for next cycle
        before_0 = new_before_0;
        before_1 = new_before_1;

    }
    PT_END(pt) ;
}

// This is the core 1 entry point. Essentially main() for core 1
void core1_entry() {

    // create an alarm pool on core 1
```

```
    alarm_pool_t *core1pool ;
    core1pool = alarm_pool_create(2, 16) ;

    // Create a repeating timer that calls repeating_timer_callback.
    struct repeating_timer timer_core_1;

    // Negative delay so means we will call repeating_timer_callback, and call it
    // again 25us (40kHz) later regardless of how long the callback took to execute
    alarm_pool_add_repeating_timer_us(core1pool, -25,
        repeating_timer_callback_core_1, NULL, &timer_core_1);

    // Start scheduler on core 1
    pt_schedule_start ;
}

// Core 0 entry point. don't need one for Core 1 because there is no entry point
int main() {
    // Initialize stdio
    stdio_init_all();

    // Initialize the VGA screen
    initVGA() ;

    myspinlock = spin_lock_init(spin_lock_claim_unused(true));

    // Initialize SPI channel (channel, baud rate set to 20MHz)
    spi_init(SPI_PORT, 20000000) ;
    // Format (channel, data bits per transfer, polarity, phase, order)
    spi_set_format(SPI_PORT, 16, 0, 0, 0);

    // Map SPI signals to GPIO ports
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
    gpio_set_function(PIN_CS, GPIO_FUNC_SPI) ;

    // Map LDAC pin to GPIO port, hold it low (could alternatively tie to GND)
    gpio_init(LDAC) ;
    gpio_set_dir(LDAC, GPIO_OUT) ;
    gpio_put(LDAC, 0) ;

    gpio_init(PAUSE_1) ;
    gpio_set_dir(PAUSE_1, GPIO_IN) ;
    gpio_init(PAUSE_0) ;
    gpio_set_dir(PAUSE_0, GPIO_IN) ;

    /////////////////////////////////////////////////////////////////////////////
    // ============================== ADC CONFIGURATION ============================
    /////////////////////////////////////////////////////////////////////////////
    // Init GPIO for analogue use: hi-Z, no pulls, disable digital input buffer.
    adc_gpio_init(ADC_PIN);

    // Initialize the ADC harware
    // (resets it, enables the clock, spins until the hardware is ready)
    adc_init() ;

    // Select analog mux input (0...3 are GPIO 26, 27, 28, 29; 4 is temp sensor)
    adc_select_input(ADC_CHAN) ;

    // Setup the FIFO
    adc_fifo_setup(
        true,    // Write each completed conversion to the sample FIFO
        true,    // Enable DMA data request (DREQ)
        1,       // DREQ (and IRQ) asserted when at least 1 sample present
        false,   // We won't see the ERR bit because of 8 bit reads; disable.
        true     // Shift each sample to 8 bits when pushing to FIFO
    );

    // Divisor of 0 -> full speed. Free-running capture with the divider is
    // equivalent to pressing the ADC_CS_START_ONCE button once per `div + 1`
```

```
    // cycles (div not necessarily an integer). Each conversion takes 96
    // cycles, so in general you want a divider of 0 (hold down the button
    // continuously) or > 95 (take samples less frequently than 96 cycle
    // intervals). This is all timed by the 48 MHz ADC clock. This is setup
    // to grab a sample at 10kHz (48Mhz/10kHz - 1)
    adc_set_clkdiv(ADCCLK/FFT_Fs);

    // set up increments for calculating bow envelope
    attack_inc = divfix(max_amplitude, int2fix15(ATTACK_TIME)) ;
    decay_inc =  divfix(max_amplitude, int2fix15(DECAY_TIME)) ;

    // Build the sine lookup table
    // scaled to produce values between 0 and 4096 (for 12-bit DAC)
    int ii;
    for (ii = 0; ii < sine_table_size; ii++){
         sin_table[ii] = float2fix15(2047*sin((float)ii*6.283/(float)sine_table_size));
    }

    // Populate the sine table and Hann window table
    int ii_fft;
    for (ii_fft = 0; ii_fft < NUM_SAMPLES; ii_fft++) {
        Sinewave[ii_fft] = float2fix15(sin(6.283 * ((float) ii_fft) / (float)NUM_SAMPLES));
        window[ii_fft] = float2fix15(0.5 * (1.0 - cos(6.283 * ((float) ii_fft) / ((float)NUM_SAMPLES))));
    }

    ////////////////////////////////////////////////////////////////////////////////
    // ================================ ADC DMA CONFIGURATION ========================
    ////////////////////////////////////////////////////////////////////////////////

    // Channel configurations
    dma_channel_config c2 = dma_channel_get_default_config(sample_chan);
    dma_channel_config c3 = dma_channel_get_default_config(control_chan);


    // ADC SAMPLE CHANNEL
    // Reading from constant address, writing to incrementing byte addresses
    channel_config_set_transfer_data_size(&c2, DMA_SIZE_8);
    channel_config_set_read_increment(&c2, false);
    channel_config_set_write_increment(&c2, true);
    // Pace transfers based on availability of ADC samples
    channel_config_set_dreq(&c2, DREQ_ADC);
    // Configure the channel
    dma_channel_configure(sample_chan,
        &c2,            // channel config
        sample_array,   // dst
        &adc_hw->fifo,  // src
        NUM_SAMPLES,    // transfer count
        false            // don't start immediately
    );

    // CONTROL CHANNEL
    channel_config_set_transfer_data_size(&c3, DMA_SIZE_32);      // 32-bit txfers
    channel_config_set_read_increment(&c3, false);                // no read incrementing
    channel_config_set_write_increment(&c3, false);               // no write incrementing
    channel_config_set_chain_to(&c3, sample_chan);                // chain to sample chan

    dma_channel_configure(
        control_chan,                         // Channel to be configured
        &c3,                                // The configuration we just created
        &dma_hw->ch[sample_chan].write_addr,  // Write address (channel 0 read address)
        &sample_address_pointer,                    // Read address (POINTER TO AN ADDRESS)
        1,                                  // Number of transfers, in this case each is 4 byte
        false                               // Don't start immediately.
    );

    // Launch core 1
    multicore_launch_core1(core1_entry);

    // Create a repeating timer that calls
    // repeating_timer_callback (defaults core 0)
```

```
    struct repeating_timer timer_core_0;

    // Negative delay so means we will call repeating_timer_callback, and call it
    // again 25us (40kHz) later regardless of how long the callback took to execute
    add_repeating_timer_us(-25,
        repeating_timer_callback_core_0, NULL, &timer_core_0);

    // Add and schedule core 0 threads
    pt_add_thread(protothread_fft) ;
    pt_schedule_start ;

}
```