1-1-2023

Designing a Flexible Framework for Developing Acoustic Array Systems

Charles Fulton Gilliland

DESIGNING A FLEXIBLE FRAMEWORK FOR DEVELOPING ACOUSTIC ARRAY SYSTEMS

A Thesis Presented for the

Master of Science

Degree

The University of Mississippi

Charles Gilliland

May 2023

ABSTRACT

In recent years, research conducted by the Applied Acoustics group at the National Center for Physical Acoustics has involved the use of microphone arrays to study the propagation of sound through outdoor environments. In such research, there is need for data acquisition systems which can be reconfigured in both hardware and software. This work is an effort to develop a modular acoustic data acquisition framework which can be configured to accommodate a wide variety of acoustic array applications. In hardware, the framework provides modularity with a generic mainboard which uses a common interface to collect data from application-specific microphone boards. In software, a generic host-side USB driver allows various acoustic array systems to be integrated with signal processing algorithms through a simple API. Using this framework, a completely new array can be developed rapidly: the user simply designs the application-specific sensor boards and adjusts firmware parameters.

DEDICATION

This work is dedicated to my family, who have provided me with an excellent education, encouraged my interests, and always been there for me.

ACKNOWLEDGMENTS

Many thanks to Dr. Wayne Prather and Dr. Garth Frazier for their support and guidance throughout my time at the NCPA. Thanks to Dr. Kasem Khalil for advising me throughout the thesis process.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

In recent years, research conducted by the Applied Acoustics group at the National Center for Physical Acoustics has involved the use of microphone arrays to study the propagation of sound through outdoor environments. In such research, there is need for data acquisition systems which can be reconfigured in both hardware and software. This work is an effort to develop a modular acoustic data acquisition framework which can be configured to accommodate a wide variety of acoustic array applications.

A sensor array is a group of sensor elements, spaced in a known geometry, which collect signals. In the context of acoustics, the sensor elements are microphones, and the signals are sound waves. There are many applications of acoustic arrays, including beamforming, noise reduction, and source separation. Each of these applications leverage acoustic arrays by processing the data from the microphone elements and producing an output.

One prevalent example of an acoustic array is a home assistant device, such as Amazon Alexa, Google Assistant, and others. These devices use microphones to listen for a prompt from a user in the room. They can then use array signal processing techniques to amplify the audio signal in the direction of the user, effectively "pointing" to where the sound is coming from[1]. This is known as beamforming. In this case, beamforming effectively increases the signal to noise ratio in the recorded audio to improve the performance of voice recognition algorithms. This beamforming can even be directly observed in some devices that use a ring of LEDs to

[1] Amazon, Alexa Developer Documentation

indicate the direction of the speaker in relation to the device. A seemingly simple device like this demonstrates several common applications of acoustic arrays: sound source direction of arrival and source separation, beamforming or beamsteering, and noise reduction.

One might initially assume that developing an acoustic array would be simple: set up a few microphones in a known geometry, record some data, write a signal processing algorithm and it's done, right? With this assumption, it might seem that the most difficult task would be the signal processing, but when designing an acoustic array system, the properties of the array hardware are just as important as the signal processing algorithms used.

The most basic requirement of an acoustic array is that all channels are simultaneously sampled – that is, each data point from each microphone element is sampled at the exact same time. Many array processing algorithms operate under the assumption of simultaneous sampling, so this must be implemented by the hardware. Additionally, when deploying multiple arrays for the purpose of sound source localization, the arrays must be simultaneously sampled with relation to each other. This can be achieved by using a GPS module to provide a precise global time reference, without requiring that the arrays be physically connected.

Parameters such as sample rate and array geometry are dependent on the frequencies of interest in the application at hand. Following the Nyquist sampling theorem, microphones should be sampled at least twice the rate of the highest frequency of interest to avoid temporal aliasing. Similarly, the spacing between microphone elements should be less than one half wavelength of the highest frequency of interest to avoid spatial aliasing[2]. The wavelength depends on the speed of sound, which varies depending on the characteristics of the propagation medium – especially

[2] Consider a 10kHz signal. With C = 340m/s, its wavelength is 34mm. Thus, elements should be spaced less than 17mm apart to avoid spatial aliasing at 10kHz.

its temperature. In practice, a margin of error for the speed of sound should be considered when determining the element spacing. Naturally, elements should exist on all axes for which directions must be resolved – to give direction in three dimensions, the array must have elements placed in three dimensions.

One parameter on which the number of required elements is dependent is the number of sound sources that the array should be able to resolve. Generally, there should be at least one more element than the maximum number of sources, though it is beneficial to use more elements when practical.

With these hardware complexities in mind, it wouldn't be unreasonable to look for an off-the-shelf acoustic array solution instead of developing one from the ground up. In fact, some inspiration has been taken from the Zylia ZM-1 microphone array, which includes 19 microphone elements in a spherical arrangement[3]. This device is intended for use in a music recording environment, where several musicians can perform together at the same time and later, signal processing algorithms can be used on the recorded data to isolate each sound source into its own audio track. The ZM-1 array is simultaneously sampled with a sample rate of 48kHz, covering the audible range. From a music production perspective, the Zylia saves time and money since one does not have to set up, take down, and buy several microphones and an audio interface. Likewise, from a research perspective, one saves time and money by avoiding having to develop a system from scratch, with multiple custom hardware revisions and many hours of seemingly fruitless debugging. Using an off-the-shelf product such as the Zylia can be a good alternative to in-house development, as both the hardware and software are ready to use out of the box.

[3] Zylia, Technology White Paper [2]

Of course, the Zylia is not without its drawbacks. Firstly, its array geometry is fixed. Disassembly of the housing is possible, but the cabling built into the Zylia is quite restrictive, so any significant modification of the array geometry requires custom cables or adapter boards to be made. The user is also restricted to a maximum of 19 channels, and interfacing with sensors other than digital MEMS microphones is not possible. Additionally, the timing of the Zylia array is not referenced to a global GPS clock, thus multiple arrays could not perform precise localization of a sound source. Often, an acoustic array needs to be specifically tailored to each specific application, so hacking together off-the-shelf components is not always the most practical solution.

Ideally, an acoustic array framework would be able to overcome the limitations of the Zylia. It would be able to accurately timestamp samples to a GPS clock reference, interface with a large number of microphone elements, and be easily extensible to other sensor types if desired. This work proposes and demonstrates an acoustic array framework that not only meets the minimum requirements of an acoustic array, but also adds functionality not available with commercial devices. It is also highly reconfigurable, reusable, and can be easily augmented to interface with sensors other than digital MEMS microphones. At the center of the framework is a data acquisition board containing the essential components for an array system and connectors for interfacing with microphone boards of varying shapes and sizes. A host computer powers the board over USB and receives data from the array over a custom USB endpoint.

This framework is referred to as “xdaq” – where “x” refers to the XMOS microcontroller used, and “daq” meaning data acquisition.

II. HARDWARE DEVELOPMENT

1. Modularity in Hardware

In order to provide a standardized hardware platform, and to reduce the amount of effort required to design and deploy a new array, it is necessary to make a distinction between the hardware elements that are common between different arrays, and those elements that are unique to each array. Designing and laying out a PCB which includes a microcontroller and a GPS module is not a trivial task: specifically, one must consider sensitive traces which require controlled impedance and length matching, such as USB differential pairs and antenna traces. These sensitive traces require careful consideration of the circuit board layer stack-up, which can vary depending on the PCB manufacturer, since the impedance of a trace is primarily affected by the trace width, its height above a reference plane, and the dielectric properties of the material between copper layers[4]. On the other hand, a PCB with only digital MEMS microphones and an aggregator IC is much more tolerant to variations in layer stack-up, as it does not require controlled-impedance traces[5].

With these considerations in mind, the xdaq framework defines two types of boards:

i. The Mainboard – which includes a microcontroller, GPS module, power supply, and other supporting circuitry.

[4] Texas Instruments, High Speed Layout Guidelines. [3]
[5] Signal bandwidth of single-ended PDM and TDM signals is much lower than that of USB differential pairs.

ii. The Microphone Board – which includes digital MEMS microphones and a codec conversion IC.

This distinction of boards allows the same mainboard to be used with many different microphone boards, so that a new array requires only microphone boards to be designed. Another advantage of this distinction is that the mainboard could interface with sensor boards other than digital MEMS, provided the appropriate firmware support. The firmware framework which complements this hardware modularity is described in Chapter III, Section 1.

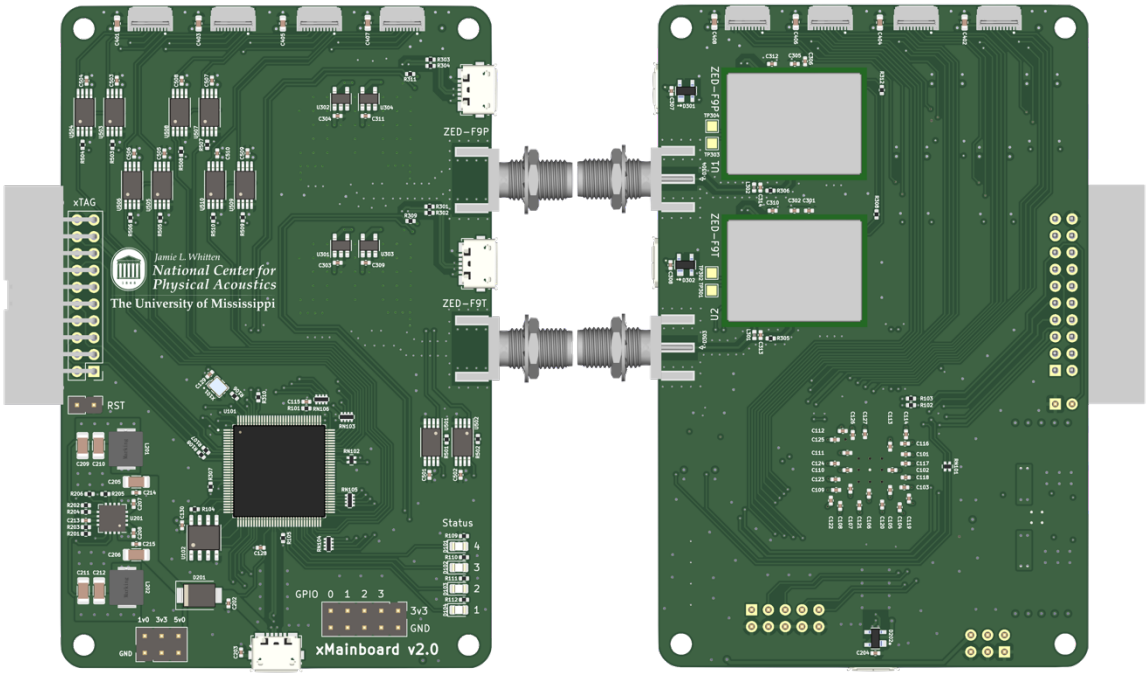

Figure 1: Renders of a Mainboard.

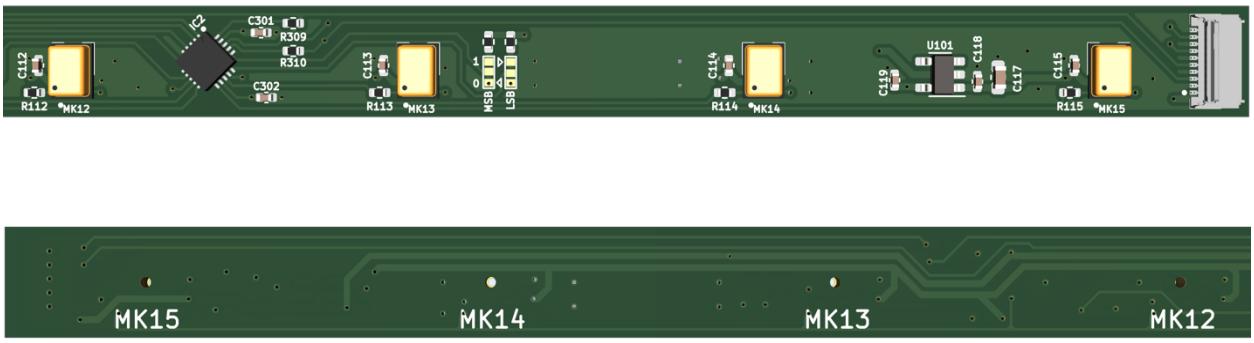

Figure 2: Close-up renders of a 15-channel linear microphone board.

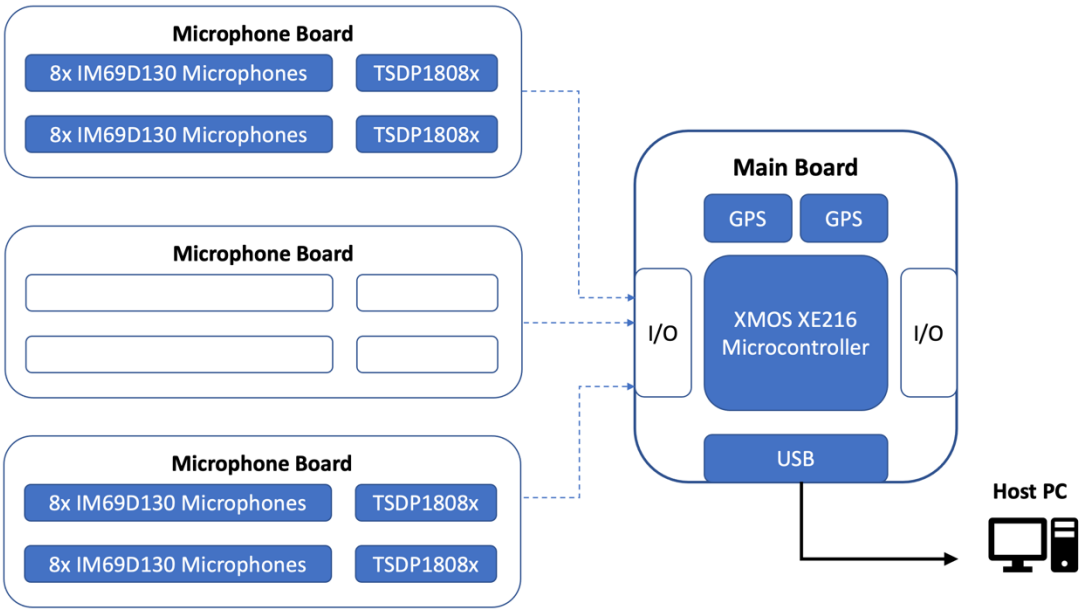

Figure 3: System Level Block Diagram.

2. The Microphone Board

It is necessary to first understand the function of the microphone board before seeing how it interfaces with the mainboard. The digital MEMS microphone used in this design is the Infineon IM69D130. These microphones have a small footprint, low power consumption, and have a PDM output clocked between 1.536 and 3.072 MHz, depending on the desired audio rate. Additionally, two microphones can be configured to share a single data line, where one microphone outputs its data on the rising edge of the clock, and the other on the falling edge. Sharing a data line simplifies board layout. According to the datasheet, "the flat frequency response (28Hz low-frequency roll-off) and tight manufacturing tolerance result in close phase matching of the microphones, which is important for multi-microphone (array) applications." [6] Notably, the analog to digital conversion (ADC) is done on-chip – a distinct advantage to laying out ADC circuitry manually, which can be prone to error, increases power requirements, and increases board size and complexity.

The pulse-density-modulation, or PDM serial interface is relatively simple as it only requires two connections: clock and data. However, interpreting PDM data is not trivial. With a traditional audio format such as PCM, or pulse code modulation, a word of data is represented by a sequence of bits. For example, 24-bit PCM would have a word length of 24 bits, where the first bit is the most significant, and the last is the least significant. With PCM, a sample represents the sound pressure level directly as a word. In contrast, with PDM each individual bit is a word – that is, the word length is 1 bit. With PDM from a MEMS microphone, "positive pressure increases density of 1's, [and] negative pressure decreases density of 1's in data output." [7] Naturally, PDM has a very low bit depth (only 1 bit per sample), whereas PCM can have a much

[6] Infineon, IM69D130 Datasheet [4]
[7] Infineon, IM69D130 Datasheet [4]

higher bit depth (24 bits per sample in the previous example). On the other hand, PDM data from the MEMS microphone comes at a much higher sample rate of 3.072MHz as compared to 48kHz with PCM. The challenge is then to translate the high sample rate, low bit depth PDM format into low sample rate, high bit depth PCM format. This is done by applying a low pass filter and decimation, also known as down-sampling. While this process is generally well-understood, it can become computationally expensive in applications with many channels.

In order to reduce the number of data lines needed, and to reduce computational load on the microcontroller, the PDM outputs from the digital MEMS microphones are interfaced with a PDM to TDM (time division multiplexing) converter IC. The converter used here is the Tempo Semiconductor TSDP1808x. This IC is specifically designed for the task of aggregating PDM data from up to 8 microphones, converting to PCM format, and sending the converted samples over a TDM serial interface. Based on the clock frequencies given on the TDM bus, the TSDP1808 determines the PDM clock frequency and decimation factor. For the desired sample rate of 48kHz, the PDM clock will be 3.072MHz.

The TDM serial interface is composed of at least 3 signals: bit clock (BCLK aka SCLK), frame-sync (FSYNC aka LRCLK), and data (SDOUT). The rising edge of the FSYNC signal indicates the start of a frame of data, where a frame is one sample from each channel. In this application, the sample rate is 48kHz, so FSYNC pulses at the same rate. Each sample in a frame is aligned on a 32-bit word with respect to BCLK.[8] The IC is configured for 8-channel output, so there are 8 words in a frame. To determine the necessary BCLK frequency, the sample rate, bits-per-word, and words-per-frame are multiplied, giving 12.288MHz for this example. Figure 4 shows the signal timing of the TDM bus.

[8] It is worth noting that the effective bit depth is only 24 bits, so the least-significant 8 bits are discarded. This is done for simplicity in firmware.

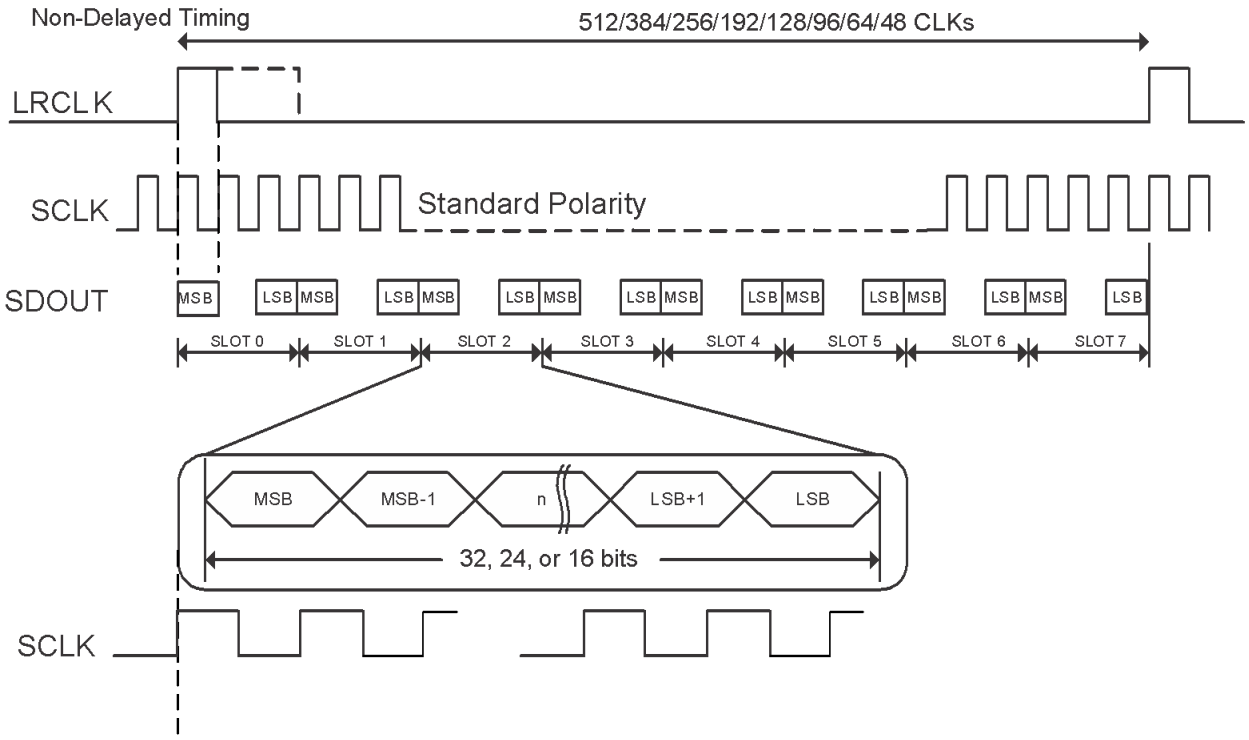

Figure 4: TDM Audio Output of TSDP1808x.[9]

Using the TSDP1808, data from 8 microphones can be carried by 3 conductors (BCLK, FSYNC, DATA), whereas 5 would be required with PDM (CLK, DATA1-2, DATA3-4, DATA5-6, DATA7-8). Another 8 microphones can be added to the same TDM bus with the addition of a second DATA signal. For improved signal integrity in the flat flex cables connecting the microphone board to the mainboard, there is an additional ground conductor adjacent to each signal conductor, giving the smallest possible loop area for the signal and its return path. The final 10-pin cable pinout is shown in figure 5, where DIN_F* represents DATA signals.

[9] Tempo Semiconductor, TSDP1808x Datasheet [5]

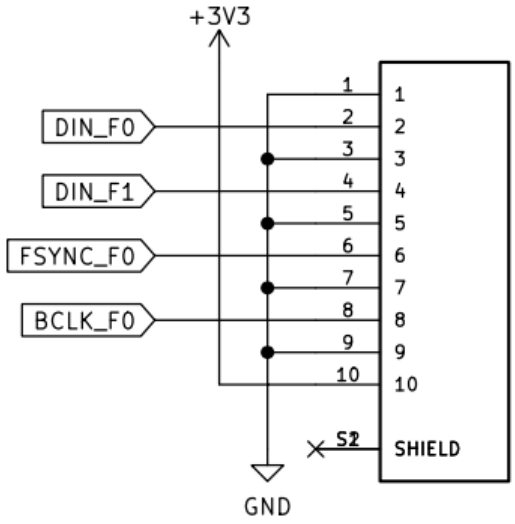

Figure 5: Microphone board connector pin description.

3. The Mainboard

The mainboard has three major responsibilities: powering and interfacing with the microphone boards, synchronizing data acquisition with the GPS clock, and sending data from the microphones and GPS to a host computer over USB. The board is powered over USB, and 3v3 and 1v0 supplies are generated by a dual buck converter.[10]

At the center of the mainboard is the XMOS XE216, which is a 32-bit multi-core microcontroller capable of precisely timed and deterministic IO. In addition, it contains a built-in USB2.0 High-Speed physical layer. The architecture of this microcontroller is unique in several ways. To begin with, it is composed of two xCORE two tiles, each of which has between 5 to 8 xCOREs. Each xCORE is capable of executing real-time computational and IO tasks[11]. The number of xCOREs on a tile can change based on the computational requirements of the tasks being run. Figure 6 shows the block diagram of the XMOS XE216 microcontroller. The left-hand and right-hand sides of the figure show the two tiles, each with 8 xCOREs and input/output ports. The two tiles are connected via the xCONNECT switch, which facilitates data transfer between xCOREs on different tiles.

[10] Admittedly, the buck converter used is capable of providing much more current than is used by the mainboard itself, but the extra power budget could be used to supply other circuits.
[11] XMOS, XE216 Datasheet [6]

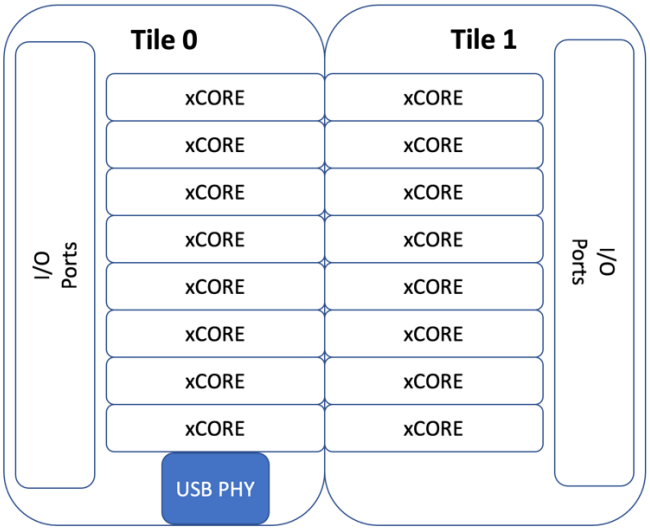

Figure 6: XE216 Block Diagram.[12]

For synchronizing data acquisition to a global clock, a GPS module is used. The u-blox ZED-F9T was chosen for this task, as it has two phase-locked time pulse outputs, one of which is programmed as a pulse-per-second (PPS) and the other as the audio bit clock (12.288MHz). The rising edge of the PPS indicates the top of a second, accurate to 5ns[13]. These clock signals are buffered and connected to IO ports on both tiles of the microcontroller, which can then clock the TDM interfaces with reference to the precise PPS signal. The GPS also has a UART interface, which sends messages including timestamp and position to the microcontroller. The microcontroller is then able to accurately timestamp the samples received from the microphone boards. This GPS is capable of RTK (real-time kinematic) positioning which, when paired with a second on-board GPS module, allows orientation of the array to be calculated.

Figure 7 shows the resources required for a 32-channel TDM receiver task. A single TDMrx task runs on one xCORE. Each of these tasks use one 4-bit port for the data inputs, two 1-bit ports for BCLK and FSYNC generation, and another 1-bit port which can be shared

[12] XMOS, XE216 Datasheet [6]
[13] U-blox, ZED-F9T Datasheet [7]

between multiple tasks for PPS synchronization. Generally in the xCORE architecture, ports cannot be shared by different tasks, but with careful locking this can be achieved. FSYNC is generated according to the common master clock signal provided by the GPS. BCLK and FSYNC are then buffered so that all microphone boards share the same clock signals.

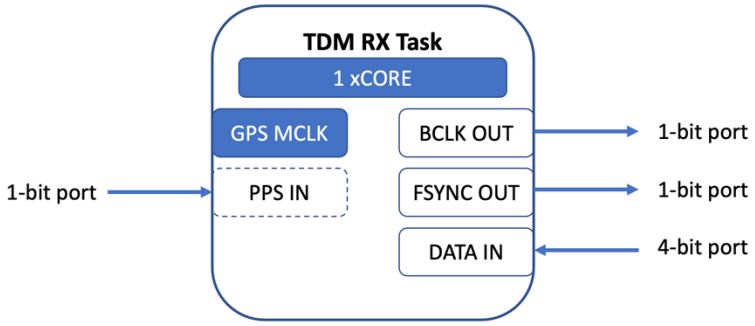

Figure 7: TDM RX task resources.

It is worth mentioning the parallel 4-bit ports used for TDM aggregation. Four data input signals are connected to this port. As shown in figure 8, a read operation returns a 32-bit word, with the bits interleaved. Since it is a 4-bit port, a 32-bit read operation only reads 8-bits with respect to the bit clock. Thus, it takes 4 read operations to read a 32-bit word from each data input pin. Once 32 bytes have been read, or 4 bytes for each pin, they must be unzipped to form

8, 32-bit words. Figure 9 gives a simplified example where a 4-byte zipped word is unzipped into 4, 1-byte words. This operation is repeated to accommodate 8 channels with 4-byte words.

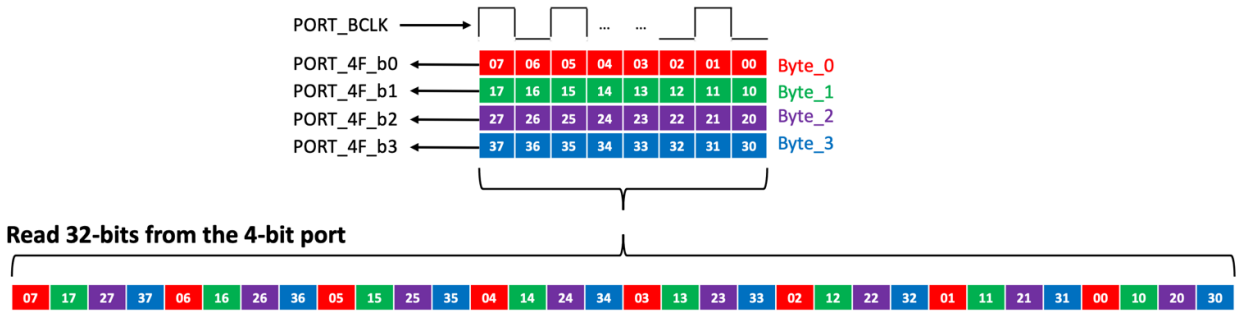

Figure 8: A 4-bit read operation. Colors represent data pins. First digit indicates channel index, second digit indicates bit index.

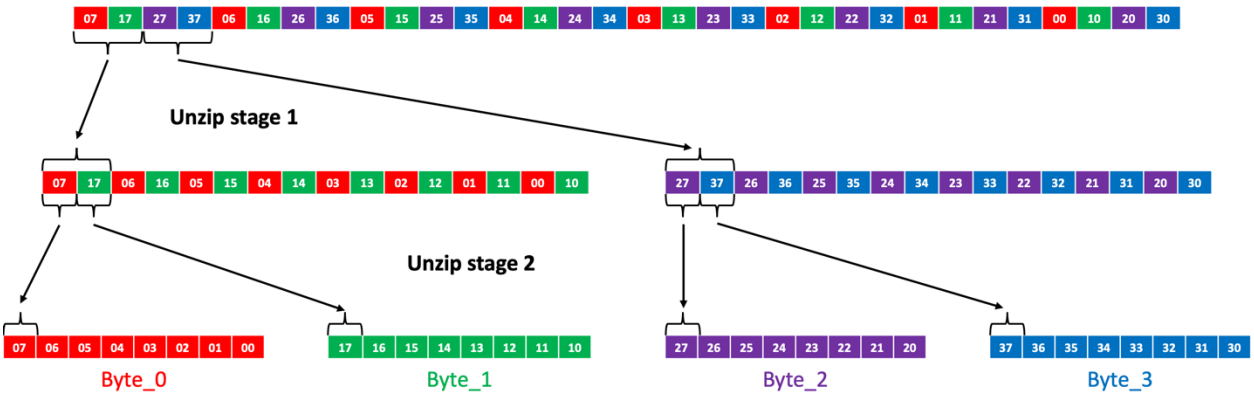

Figure 9: Unzipping an interleaved word into four bytes.

III. SOFTWARE DEVELOPMENT

1. Mainboard Firmware

The firmware that runs on the mainboard is split into two major parts. Code that is common to all array implementations is in the ‘module_xdaq’ directory. Application-specific code, such as data acquisition drivers, are in the application directory. Application directories are prefixed with ‘app_.’ The xdaq module contains the code which is responsible for decoupling the audio data rate from the USB data rate, embedding GPS metadata into USB packets, and queuing USB transfers to the host computer. In order to interface with the xdaq module, the application code must set the appropriate #defines, and send its audio data using the standard interface provided by module_xdaq. Code for acquiring the audio data is left up to the user to implement, but TDM firmware for digital MEMS microphones is implemented here. The basic structure of a TDM based application is shown in figure 10. Similar to the hardware, this structure allows for minimal effort required when writing firmware for a new array. Figures 12 and 13 describe the organization of tasks and data flow in an xdaq application.

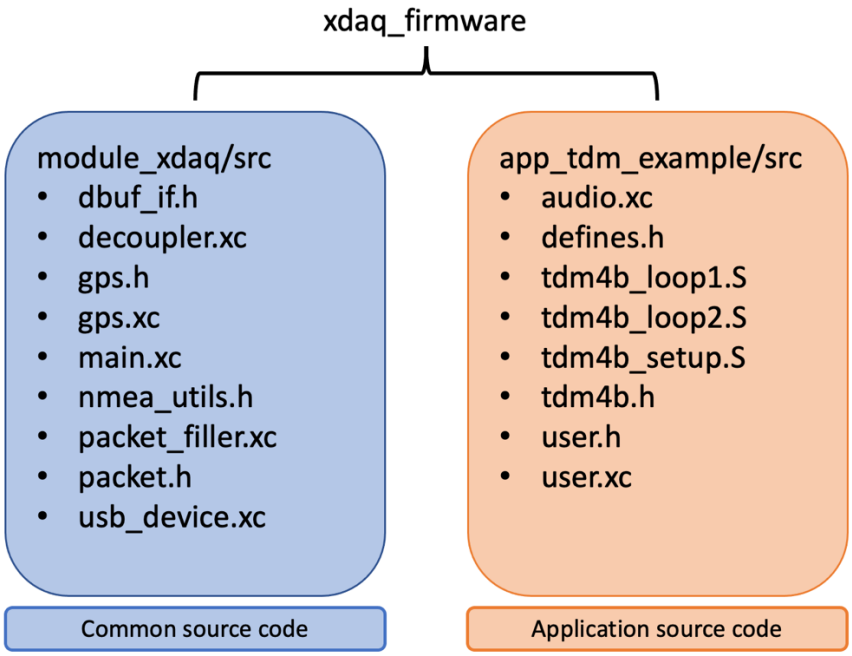

Figure 10: File structure of an xdaq application.

The most time-sensitive section of code is the audio ingestion task, implemented in audio.xc. This task must run in real-time with respect to the audio rate – that is, it must acquire, unzip, and send an audio frame to the decoupling task before another frame of data is available on the serial audio interface. For this reason, the acquisition and unzipping function is written in assembly to take full advantage of the dual-issue mode of the xCORE processors. See figure 11 for an example of acquire and unzip in dual-issue mode.

```
// tdm4b_loop1.S

// Input two samples, retrieve two samples stored previously
{ ldw    f, inp_array[1] ; in     d, res[inp_port] }
{ ldw    e, inp_array[3] ; in     c, res[inp_port] }

// Unzip the samples
// aeim bfjn cgko dhlp -> (abcd) (efgh) (ijkl) (mnop)
  unzip  e, f, 0
  unzip  c, d, 0
  unzip  d, f, 0
  unzip  c, e, 0

// Bit-reverse and store the recieved samples
{                        ; bitrev f, f            }
{ stw    f, inp_array[7] ; bitrev e, e            }
{ stw    e, inp_array[5] ; bitrev d, d            }
{ stw    d, inp_array[3] ; bitrev c, c            }
{ stw    c, inp_array[1] ;                        }
```

Figure 11: Assembly code using dual issue mode.

The audio task is also responsible for synchronizing the serial audio interface with the PPS signal from the GPS. This is done by taking a timestamp, with respect to BCLK, at the rising edge of the PPS signal. With this timestamp, the audio task is able to schedule future transfers with reference to the PPS. Then, each frame received is marked with a frame index variable, which ranges from 0 to SAMPLE_RATE-1. Here, a frame index of 0 represents the very first frame in a second, and that index is incremented for each successive frame until SAMPLE_RATE, when it is reset to 0. With this method, the samples are timestamped at the earliest possible stage in the data acquisition.

After data has been acquired, it is sent to the decoupler task, implemented in decouple.xc, via the fswap interface, which behaves as a simple double buffer. In the decoupler task, several audio frames are again buffered into a ring buffer, with size defined by DEC_MULT. As its name suggests, this task decouples the audio ingestion rate from the rest of the tasks involved in data transfer. Once again, the frame indices are preserved through this task. This is an important

step in the data transfer process, as timing requirements for both the audio task and the USB task must be met.

The packet filler task, implemented in packet_filler.xc, is responsible for packaging the data into structured packets to be sent over USB. Additionally, this is the step in which GPS metadata are combined with the audio data. Since the GPS timestamp message arrives on the UART interface before the PPS rising edge[14], this data has already been received and parsed by the GPS task. In the packet filler, the GPS data is injected into the packet which contains the frame indexed 0. The packets will later be processed by the host USB driver, described in the next section. Once again, several packets are buffered before being sent to the USB task. This is done to take full advantage of the bandwidth available over USB 2.0 High-Speed.

As previously mentioned, samples received from the audio task are aligned to 32-bit words; however, the least significant 8-bits are simply padding – the true bit-depth of a sample is 24-bits, or 3 bytes. To pack as many frames as possible into a packet, these samples can be rearranged such that the padding is removed, and words are aligned to 24-bits. Of course, this requires the host driver to unpack the samples before they can be used.

Finally, the USB device task, implemented in usb_device.xc, receives several packets from packet filler. This task has the sole responsibility of sending packets over the USB interface. The USB endpoint is implemented as a Bulk IN endpoint with a packet size of 512 bytes. The first 128 bytes are reserved for metadata such as the number of channels, number of frames in a packet, sample rate, frame index, etc. The remaining 384 bytes are reserved for audio frames. Figure 15 gives the definitions of the data_packet and data_header structures.

[14] U-blox, ZED-F9T Integration Manual. [8]

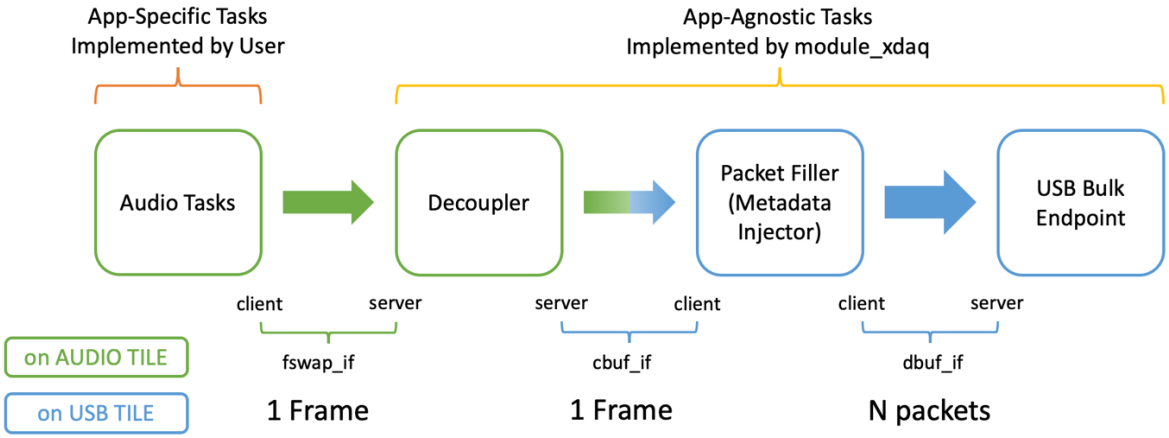

Figure 12: Organization of tasks, data flow and interfaces provided by module_xdaq.

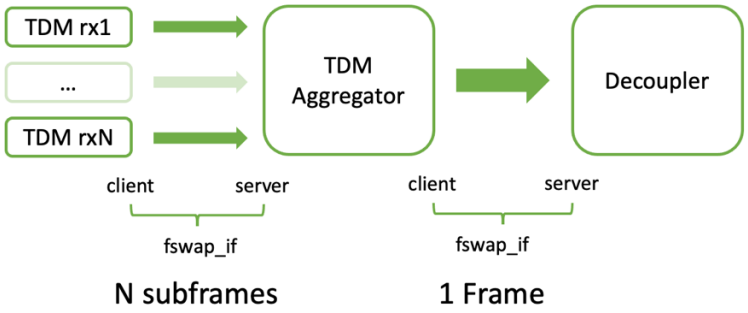

Figure 13: Application with multiple TDM receiver tasks and an aggregator task.

```
/*
Interface used for double buffer WITHOUT frame index
(packet_filler -> usb_device)
*/
interface dbuf {
    void swap(char * movable &x);
};

/*
Interface used for double buffer WITH frame index
(audio -> decoupler)
*/
interface fswap {
    void swap(uint32_t f_idx, int32_t * movable &x);
};

/*
Interface used for ring buffer WITH frame index
(decoupler -> packet_filler)
*/
interface cbuf {
    [[clears_notification]]
    uint32_t get_buffer(int32_t buf[]); // returns frame index

    [[notification]] slave void data_ready ( void );
};
```

Figure 14: Interfaces defined by module_xdaq used for transferring data between tasks.

```
struct data_header {
    uint8_t n_chan;                // number of channels
    uint8_t n_frame;               // number of frames in this packet
    uint8_t prod_id;               // product ID (unique to array type)
    uint32_t frm_idx;              // index of the first frame in this packet
    uint8_t extra_len;             // length of used extra_data in bytes
    unsigned char extra_data[119]; // extra space for other metadata (GPS)
};

struct data_packet {
    struct data_header header;
    unsigned char samples[384];
};
```

Figure 15: Packet and header structures.

2. Host-side USB Driver

The host-side USB driver is responsible for receiving USB packets from the array hardware, buffering, and sending the data to a user application for further processing. This framework provides a driver, libxdaq, which has been implemented using the libusb library. Libusb is available for Linux, Windows, MacOS, and other operating systems. Libusb, and libxdaq by extension, runs in user-space, so it does not require root privileges to run, provided that the user has permission to the USB device itself. The current implementation of the host driver has been tested on MacOS (x86) and Linux (x86, ARM) operating systems. This driver is able to handle various types of arrays without requiring modifications or recompilation of the driver code. Thus, new arrays can be developed without having to rewrite the driver. Additionally, it is capable of running at a high speed to keep up with the data rate of the array, otherwise packets would be dropped.

Libxdaq is intended to allow a user to easily write programs that use the xdaq framework, without requiring extensive knowledge of the firmware itself. As such, the user code must only call a few functions from libxdaq to initialize, start, and read data from the device. Included with libxdaq is a program called xrecord, which serves as both a utility program for recording audio to the hard disk, and an example for users to follow when integrating libxdaq into a signal processing algorithm. The application first attempts to connect to a specific xdaq device using the unique product ID. Using different product IDs for different array types allows a host computer to receive data from multiple arrays simultaneously. Once libxdaq has successfully connected to a device, the user can begin data acquisition by calling the xdaq_start function. Internally, libxdaq will create one thread for handling libusb events, and another thread for decoupling libxdaq from the user program. When a packet arrives, libusb calls a callback

function which pushes the received packet to the tail of a linked list. The linked list is implemented as a simple FIFO, with push and pop operations taking O(1) time[15] – allowing the callback function to return as soon as possible. The decoupling thread performs pop operations on the linked list, filling up a buffer of packets. Once the buffer is full, it is written to a Unix pipe, which is configured in non-blocking mode. Finally, the user application can call the xdaq_read function, which blocks until a buffer of packets is read from the pipe. The xrecord application simply writes the unprocessed data packets to a file to be used later. Other user applications could parse the data packets for use in a signal processing algorithm. Utility functions are available for the user to easily parse the packets into usable audio samples and metadata. The block diagram of libxdaq is shown in figure 16.

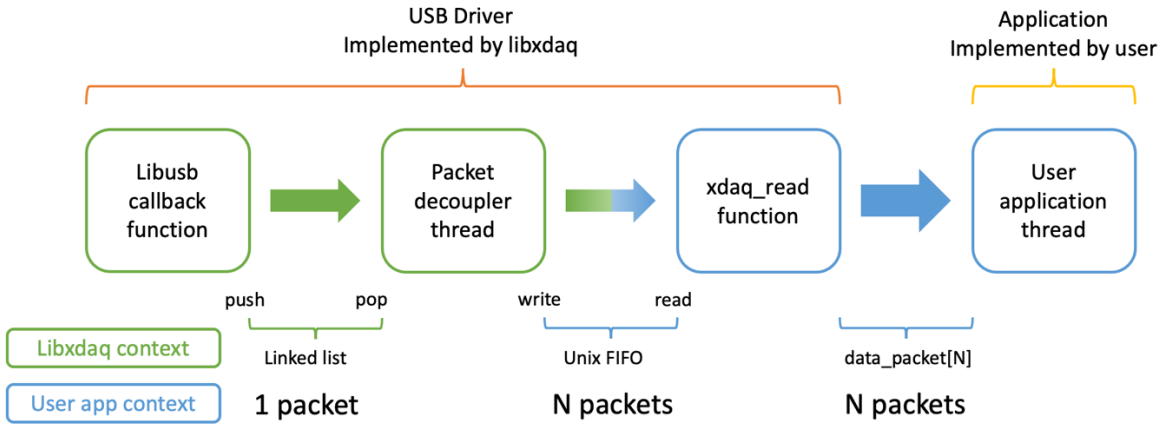

Figure 16: libxdaq host-side USB driver block diagram.

Due to the structure of the data packets, there is extra space available in which metadata can be stored. The current implementation of GPS metadata does not use the entire extra space, so it is possible for the user to customize the firmware and user application to give additional

[15] Since the linked list is used by multiple threads, it uses a mutex lock to ensure thread safety. The decouple thread is busy waiting while attempting to pop a packet off the list. Therefore, opportunities for performance improvements do exist, such as back-off or lock-free techniques, but a functional implementation is prioritized first.

functionality. Even in this case, the underlying libxdaq driver need not be changed or recompiled.

IV. FUTURE WORK & CONCLUSIONS

There do exist opportunities for improvement of this work. As previously mentioned, the USB implementation currently uses a bulk endpoint. A bulk endpoint was chosen for its simplicity, guaranteed delivery, and error checking of data packets; however bulk endpoints do not guarantee bandwidth. On the other hand, isochronous endpoints guarantee bandwidth, but delivery of packets is not guaranteed nor is error checking done[16]. Using a networking analogy, bulk is to isochronous as TCP is to UDP. It is possible that using isochronous transfers could yield better performance on a congested USB bus, but this was not explored due to the simplicity of bulk transfers.

There is also room for improvement within the host driver. Currently, the linked list implementation within libxdaq relies on mutex locks for thread safety. While this is functional, it is likely that a lock-free implementation would have much better performance. Additionally, there are some sections of code that use busy waiting until new data is available. Again, while functional, there are probably better ways to implement this.

Due to the many-core nature of the XMOS microcontroller, there is often unused processing power in arrays with less than 64 channels. In these cases, it is entirely possible to run some signal processing steps on the mainboard itself, such as decimation, filtering, and even FFT. One can imagine the potential benefits of running preprocessing steps on the XMOS before sending data to the host computer. It might even be possible to run a beamforming application

[16] Jan Axelson, USB Complete Developer's Guide [9]

solely on the XMOS. These possibilities do not fall within the scope of this work, but there is clearly potential for improving the feature set of this framework.

While not discussed here, a separate data acquisition system using differential analog to digital converters instead of digital MEMS microphones was developed using the xdaq framework. In only a few days, new driver code interfacing with the ADCs over SPI was developed and integrated with the existing firmware module, proving the flexibility of the framework. Clearly, use of this framework saves a significant amount time and effort, making reconfigurable acoustic arrays easily accessible even to users with a beginner level of hardware and software experience.

The xdaq framework sets out to provide users with a firm foundation on which to develop highly customizable acoustic array systems. Using this framework, a completely new array can be developed rapidly: the user simply designs the application-specific microphone boards and adjusts firmware parameters. Integrating an array signal processing algorithm with the array hardware is made simple by using the generic host-side driver library, and existing signal processing algorithms can be easily ported to a different array by changing only a few parameters.

LIST OF REFERENCES

LIST OF REFERENCES

[1] *Alexa Developer Documentation*, Audio Hardware Configurations, Amazon.com Inc., 2021. [Online]. Available: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/audio-hardware-configurations.html.

[2] *Zylia Portable Recording Studio - Technology White Paper*, Zylia Sp, z o.o, 2017. [Online]. Available: https://www.zylia.co/white-paper.html.

[3] *High-Speed Layout Guidelines*, Rev. A, Texas Instruments Inc., 2017. [Online]. Available: https://www.ti.com/lit/ml/scaa082a/scaa082a.pdf.

[4] *High performance digital XENSIV™ MEMS microphone*, IM69D130, Infineon Technologies AG, 2017. [Online]. Available: https://www.infineon.com/dgdl/Infineon-IM69D130-DataSheet-v01_00-EN.pdf?fileId=5546d462602a9dc801607a0e46511a2e.

[5] *Octal PDM to 24-bit TDM converter*, TSDP18xx, Rev. 0.97, Tempo Semiconductor Inc., 2019. [Online]. Available: https://temposemi.com/wp-content/uploads/2019/09/TSDP18xx_DS.pdf.

[6] *XE216-512-TQ128 Datasheet*, Rev. 1.16, XMOS Ltd., 2018. [Online]. Available: https://www.xmos.ai/download/XE216-512-TQ128-Datasheet(1.16).pdf.

[7] *ZED-F9T Datasheet*, u-blox F9 high accuracy timing module, Rev. R06, u-blox AG, 2022. [Online]. Available: https://content.u-blox.com/sites/default/files/ZED-F9T-10B_DataSheet_UBX-20033635.pdf.

[8] *ZED-F9T Integration Manual*, u-blox F9 high accuracy timing module, Rev. R01, u-blox AG, 2022. [Online]. Available: https://content.u-blox.com/sites/default/files/ZED-F9T_IntegrationManual_UBX-21040375.pdf.

[9] Jan Axelson, *USB Complete The Developer's Guide*, Fourth Edition, Madison, WI: Lakeview Research LLC, 2009, pp. 61-88.

VITA

Charles Gilliland grew up in Memphis, TN, where he spent much of his free time tinkering with electronics and playing guitar. His interest in programming began in high school with an Arduino microcontroller and a Raspberry Pi. He graduated from the University of Mississippi in 2021 with a bachelor's degree in computer science and decided to continue his studies with a master's degree specializing in computer engineering. Charles gained invaluable experience during his time working at the National Center for Physical Acoustics as a research assistant, where he learned circuit board design, digital signal processing, and embedded systems design. Charles is looking forward to a career as an electronics engineer, where he hopes to continue working with both hardware and software.