A Homemade Protocol Controlling Apple's iPhone for Audio Synthesis with 3rd Party Accessories

Sam Drazin
Music Engineering – University of Miami
MMI 593: DSP for Embedded Devices

***Abstract*—The primary goal of this project was to create and utilize a protocol which can transmit a serial data stream and send it to an iPhone through the headphone jack. Apart from sending stereo audio channels out of the device, the headphone jack features a line-in conductor which will act as our input to the iPhone's internal architecture. Methods for audio synthesis will then be explored using the received values sent from a micro-controller.**

***Index Terms*—iPhone, Arduino, Wii Nunchuck**

I. Introduction

ADVANCEMENT of mobile devices and the constant minification of personal computers enables a society endowed with an ever-increasing amount of computational power in the palm of one hand. This trend has been sculpted by electronics manufacturers and their push for computers to be made smaller and lighter, and for mobile phones and PDAs more robust and feature-rich. Convergence of these two market sectors culminated in June of 2007 with the release of Apple's iPhone: a multitouch smartphone with an open-development platform and user-interface experience that would revolutionize the world's view of what a mobile device could do.

The overnight success of the iPhone would give way to a saturated market of third-party products and applications. Today, the US *App Store* currently hosts over 225,000 applications, roughly 27% of which are available for free. Amongst the overwhelming number of applications available for download, a smaller subsection take advantage of the device's extensive hardware capabilities. The iPhone boasts an impressive laundry-list of hardware features, including (most notably) a multitouch capacitive touch-screen display, a three-axis accelerometer, a GPS tracking chip, and support for audio recording/playback/synthesis capabilities. The iPhone's extensive hardware components, as well as sleek form-factor, make it a penultimate display of embedded device design. For the same reason, the iPhone SDK is an attractive platform for development with much of the groundwork already completed and integrated into a polished system.

This report summarizes the efforts taken to create an application which harnesses the iPhone's hardware capabilities, as well as those of third-party devices collaboratively to provide the user an intelligible environment to synthesize audio comfortably and without inhibition.

II. Background

Where ever there is technology, there will be hackers. It took only eleven days after its release to jail-brake an iPhone; the extensions and modifications to a device as mature and capable as this or any other are vast and too broad to summarize. This was an equal difficulty when researching for project possibilities taking advantage of pre-assembled embedded devices doubling as development platforms. Nevertheless, several projects and articles were found that served as key influences towards choosing to pursue the aforementioned goals.

A. Related Work

Controlling an RC Car with an iPhone and Wii Components: Brothers Josef and Michal Prusa consider themselves simple DJs with technological urges, but the two obsess over the newest gadgets available and harnessing their inherent capabilities to do cool stuff. This project stuck out in particular, involving two retail embedded devices into a single objective. With the use of various Wii controllers, an Arduino micro-controller, an iPhone and a MacBookPro, the brothers Prusa have revamped the interface for controlling an RC car. The video posted on their project website highlights the full functionality of their work. Reading through the articles written about the modifications made in syncing the Wii controllers with the Arduino seemed relevant and manageable within the scope of a semester. Having recently explored the architecture of an Arduino for class assignments, its seemed like a stable interface for receiving data from a variety of sensors, and just as able of a hub for translating that information and forwarding it on through either digital or analog means.

III. Proposed System

The primary goal of this project is to utilize the iPhone's hardware and software development platform to synthesize audio with the use of a third-party accessory: the Wii Nunchuck. The iPhone SDK features several capable APIs providing methods for recording and processing an audio stream. Utilizing this input line into the devices architecture, a protocol will be designed to carry data sent from a micro-controller (reading data from the Nunchuck) to control parameters of an audio synthesis routine within a running application. The system permits flexibility in the choice of sensors/controllers feeding data to the micro-controller, and will ideally fuel an open-source repository of code segments that will provide guidance to those trying to accomplish similar goals.

As previously mentioned, the proposed system features three main components suggested above: an iPhone, an Arduino micro-controller, and a Wii Nunchuck. While

running the application on the iPhone, the Nunchuck will connect to the micro-controller, which will gain access to the seven sensors the controller contains. These values are packaged into a formatted data stream and sent as a digital signal to the iPhone, which receives the signal through the line-in conductor from the headphone jack. Utilizing a constant buffering scheme to process the incoming audio, the signal is interpreted and sent to the corresponding controls within the audio processing environment to trigger and control stored audio samples and generate tones.

To use the complete system, a user will hold both the Wii Nunchuck in one hand and the iPhone in the other, run the Wii Nunchuck Application, and utilize the sensors from each of the hand-held devices to synthesize audio. Sounds ranging from DJ samples to sine waves can be generated by the application, and synthesis is controllable from acceleration data from both devices, as well as both button triggers and the joystick featured on the Wii Nunchuck. Figure 1 models the proposed system in use. The block-diagram in Figure 2 shows the an abstract flow of the signal throughout the system.

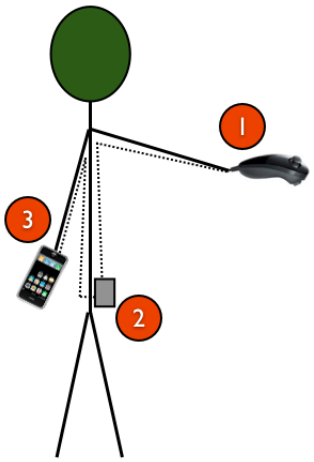

Fig. 1: A model of the system in use. The red markers indicate the following components: 1) the Wii Nunchuck controller, the primary source of sensory information sent to 2) the Arduino micro-controller, collects and packages the data received and transmits a digitally encoded signal to 3) the iPhone, device that receives, interprets, and utilizes sensory information to synthesize audio.

Although the possibilities are limitless for a device as established and supported as the iPhone, selecting it as a host device was not without costs. A principle function of the proposed system relies on the ability to transmit data to the iPhone from exterior devices. With ample processing power and software support/documentation for the iPhone SDK, it seemed best to take advantage of the assets that the mobile device had to offer. In considering possible methods of transferring data into the iPhone, careful consideration was taken before a final plan was chosen.

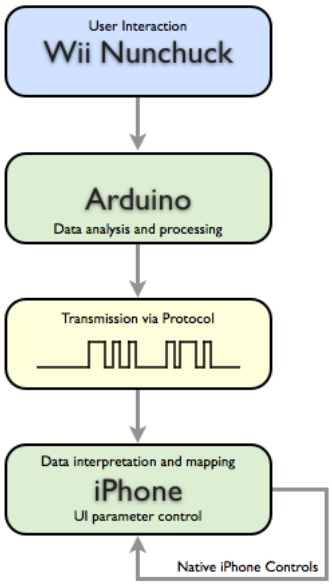

Fig. 2: Block diagram of the overall system.

IV. Execution and Evaluation

A. *iPhone*

A prominent reason in choosing the iPhone SDK as the platform for development was the amount of online documentation available, as well as the known capabilities for audio processing. Laying the framework for the finished application, an environment needed to be designed which could handle an input stream of data. Without experience in transmitting data into an iPhone, research was conducted in order to find a solution. After weighing out the options found through the documentation, three possible data entry-points were established: 1) the external microphone used for typical phone-communication), 2) the 30-pin dock connector, and 3) the headphone jack (featuring an extra conductor for line-in recording). For the sake of preserving the fidelity of data transmission, the external microphone was ruled out. Considering that any data transmitted would have to be amplified from the Arduino out into potentially free-space and then captured by a small electret microphone before getting into the device, the potential for interference was too great. The need to control values on the iPhone in (ideally) real time only added to the requirement of a clean data stream. For these reasons, the 30-pin dock connector was explored as a first possibility. Further research would reveal that this option would not be promising with the given timeline due to both software and hardware obstacles [1]. Apple provides neither sample code nor documentation for the utilization of the TX or RX pins (12 and 13, respectively) through the dock connector, a reality further emphasized by their labels as *iPod sending/receiving* lines on many of the dock connector pin-out diagrams available online. There were, however, several articles regarding certified Apple Accessory proto-

cols which, with proper conformity from the manufacturer, would allow third-party hardware to send and receive commands to running applications on the appropriate device [2]. The added complexity of making an application conform to the specified data protocols (in addition to the labor required to make an arbitrary third-party sensor play nicely with the iPhone) led to a prompt decision to explore the third possibility of the line-in headphone jack. Thankfully, an overwhelming amount of sample projects and documentation regarding audio recording and playback were available through a variety of sources. Before committing to a set of audio APIs supported by the iPhone SDK, the requirements of the proposed protocol were further explored.

B. Building an Audio Session

In order to modify parameters in real time controlling audio synthesis within the iPhone's architecture from an external controller, data needed to be transmitted and interpreted in an organized and consistent manner. The ability to access the input data stream from the line-in jack required recording capabilities of the phone, regardless of whether or not the data received was saved. Considering that the nature of the application's design is to output audio content, playback methods are needed as well. Requiring input and output lines of data from the iPhone, the search for an appropriate API was further reduced. The AudioToolbox and AudioUnit libraries were two frameworks that offered substantial control of an application's audio session instantiation within the iPhone SDK, and hence provided the necessary methods to construct a capable protocol.

Work from previous semesters at the University of Miami had paved the way in terms of establishing a tailored set of classes which initializes an audio session capable of recording, as well as creating an audio buffer structure with accessor methods [3]. Much thanks to Chris Santoro for his thorough work which enabled rapid prototyping with real-time audio pass-through on the iPhone. Once these files were integrated into the application, full access to the data-stream was made available. There was an initial concern that these methods would only work for acquiring audio data recorded in from the external microphone, but a few experiments eluded otherwise.

Every iPhone application requests an *AudioSession* object upon creation. When initialized, this AudioSession informs the device of how all audio configurations should be handled with regard to the host application (and any interfering system sounds) [4]. The AudioSession chosen for this application enabled both playback and recording capabilities, as well as set other audio-stream parameters such as buffer size and sample rate to match up with the data that would be sent to the device. The AudioSession also has methods declared for determining the currently active input source, or *route* for the device's IO, making audio input via the line-in jack as simple as plugging in a four-conductor 1/8" jack, as shown in Figure 3. When an 1/8" jack is connected into the headphone jack, the iPhone informs the AudioSession that a route change has occurred, and that data should now be expected from the line-in conductor rather than the external microphone [4]. A valid path for a data-stream into the iPhone led to the creation of a customized protocol, masquerading as audio content.

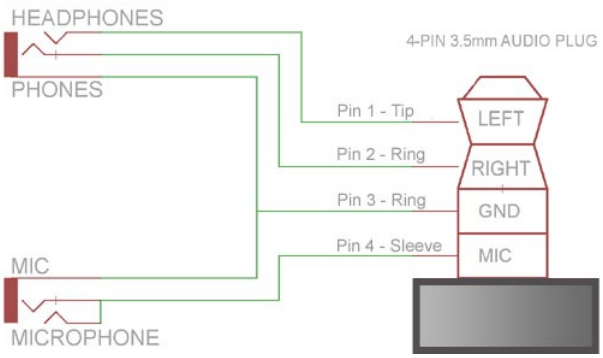

Fig. 3: The four conductors carry stereo channels out of the device, a mono channel in, and the last holds ground.

C. Protocol

Wikipedia describes a *protocol* as *"the set of standard rules for data representation, signaling, authentication and error detection required to send information over a communications channel."* The method used to transmit and interpret data within the iPhone aims to match this approach, attempting to not only read through incoming data, but to perform basic error-checking procedures and maintain a structured packet design to maximize accuracy and speed in signal interpretation. In developing the protocol, a test signal sent out of the Arduino was used which consisted of an encoded *binary* string of values with a simple pulse-train generator.

C.1 Data Transmission

The scope of this protocol would exist entirely between the Arduino micro-controller and the iPhone. The Arduino would read in the analog values from the Nunchuck's sensors and convert those values to binary strings. Those strings would then be used to generate the appropriate pulses to send out the calculated value. As the ranges of most of the Nunchuck's sensor were from 2^0 to 2^8, the transmitted values were kept as 8-bit values to avoid interpolating the data to a wider range than was naturally produced.

Within the Arduino environment, several methods enabled simple pulse generation which could be sent directly to an analog output on the board. In particular, the *analogWrite()* method was used exclusively to generate the binary values via a calculated pulse train. Transmitting a *1* required passing the *analogWrite()* function a value of 1, or *HIGH*, while a *0* was sent with a value of 0 or *LOW*. Making the assumption that these values would be represented somewhat similarly in the iPhone, differentiation

between pulses would be made based on a certain threshold as to whether a given pulse was to represent a *1* or a *0*.

C.2 Pulse Conditioning

Sending the desired data-stream using this command once for each pulse yielded disappointing results; no intelligible data was interpreted. To further inspect what data was actually being received on the iPhone versus what was being sent out of the Arduino, the signal was measured directly out of the micro-controller and from within Xcode, and both were plotted in MATLAB. The differential between sent and received data-streams was substantial. The data seen at the iPhone's line-in conductor consisted of a weak signal of staggering pulses. A bit of experimentation led to a more successful sustained transmission of high pulses to strengthen the signal sent from the micro-controller. With lengthened pulses being transmitted, however, the data still lacked clarity and definition. The primary issue lied in the inconsistency of the output pin of the Arduino. Sending any pulse for an given number of cycles would usually transmit the corresponding pulse to the iPhone successfully. If the duration of the consistent pulse-train was longer than a few hundred cycles, though, the pulse's amplitude would begin to decay, and a rebounding of the signal would also occur once the pulse was no longer sent, producing skewed and inaccurate values. The inconsistencies of alternating pulses from the Arduino led to an alteration that, although negatively impacted packet duration, highly increased a message's accuracy.

Rather than send high and low pulses to transmit a binary ones and zeros respectively, a high pulse would be sent to signify either. Distinguishing between the two would be a factor of the pulse's duration. Given a certain threshold, any pulse seen longer than that threshold was considered a *1*, and any pulse shorter (but above a minimum-pulse threshold) was considered a *0*. After either pulse was sent, a low pulse was written for a fixed duration, which served to separate the pulses and distinguish them further from any idly decaying signal from a previous high pulse. This convention drastically improved accuracy of the packets sent, and performed at a speed that was not hindering. With the ability to send intelligible bits of information serially, a structure was needed to wrap a collection of bits together to form words and packets. Let the following terms be defined for the context of this passage:

- Word: a string of binary pulses representing a singular value
- Packet: a grouping of N words which are transmitted together all at once (where $N \geq 1$)

Using high pulses to transfer each bit, a length of low pulses was chosen to separate each data packet. A new packet would be recognized by the iPhone when a length of low pulses with sufficient duration was encountered. With the stated conventions in place, the Arduino's output became intelligible from within the audio-process function of the iPhone. A sample pulse is displayed in Figure 4, shown in Audacity.

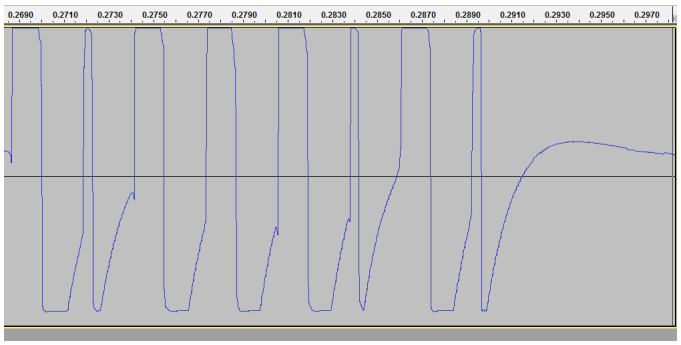

Fig. 4: A single packet transmitted from the Arduino. Wider pulses signify 1 while narrow pulses signify 0. The span of low-level values at the end of pulses represents the initializing train of 0s that separate each packet. The image portrays the following message 10111010_2, or 186_{10}.

C.3 Data Reception

Much time was spent correlating the behaviors of the signal received by the iPhone versus that originally transmitted by the Arduino. The algorithm used to interpret an incoming signal had to function within the flow of a function that would process incoming data on a buffer-by-buffer basis, and the data propagated through several checks to sift through potential noise and variance before considering any values as valid. Settings were chosen to minimize latency in order to maintain a *real-time* sense of control of the device. This requirement led to a buffer-size of 512 bytes. Putting this into perspective, sending two 8-bit words would take the space of nearly four buffers to transmit the entire packet. To accommodate this issue, the algorithm maintained statistical information about the most recent pulse events witnessed at the input terminal. With the sense of a *rolling* scope of the input, full packets were able to be sent and interpreted in an efficient manner.

As mentioned earlier, there existed a notable margin of difference between the properties of the signal sent out from the Arduino, and the signal read into the iPhone. This variance, although consistent, required in-depth testing and experimentation to develop a thorough understanding of the relationship connecting the behaviors of a stimulus and its corresponding response read through a transmission line. For example, a long pulse measured at the output terminal of the Arduino plotted in Audacity showed a span around 80 samples above an amplitude threshold of 0.5. The same signal measured from reading the raw output of the iPhone's processing function ranged anywhere from 35 to 60 samples in width. This reduction of signal length is likely due to the iPhone's hardware available for processing audio from the line-in jack, as well as any faults in the transmission line used to carry the data. It is also worth noting that a small biasing circuit was used to condition the signal out of the Arduino to the iPhone [5], shown in Figure 5.

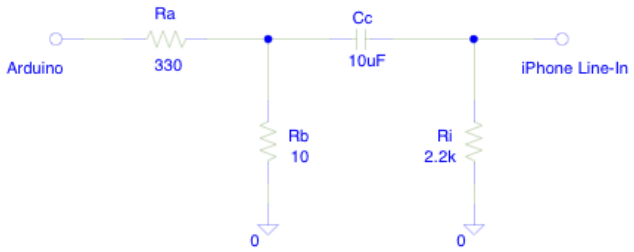

Fig. 5: This circuit scales the output voltage from the micro-controller, removes the DC offset from the device, and matches the input impedance to the device's line-in conductor.

D. Controller

With a working protocol in place, an effective communication line was paved from the Arduino to the iPhone's architecture. In order to complete the implementation of controlling iPhone synthesis parameters with an intuitive controller, the Wii Nunchuck was introduced to the system. The Wii Nunchuck is a diverse and compact controller with a multitude of functionality. The device contains a three-axis accelerometer, as well as a two-axis joystick and two push buttons. Integrating the Nunchuck to communicate with the Arduino was a painless addition thanks to existing code online which handled synchronizing the Nunchuck's output and forwards that information to the analog pins of the micro-controller [6]. Utilizing a manufactured break-out board that inserts into the Nunchuck's proprietary connector, a secure connection was made between the Nunchuck's pin-outs and the Arduino. Thanks to the open-source header file with methods that process and return the values of each Nunchuck sensor, polling the values of the controller was tied to the binary conversion functionality of the existing protocol methods for the protocol. At last, a Wii Nunchuck successfully transmitted data to the iPhone.

E. Interface

The application created to receive the data stream sent from the Arduino featured several tabs, each of which displays a page of differing functions. Aside from *Welcome* and *About* screen display basic information about the application itself, and the *Synthesize* and *Settings* pages are designed to allow the user to interact with the application's synthesis behaviors. The *Synthesize* view displays a button allowing the user to initiate a connection between the iPhone and an external data source. Once pressed, the device will wait for an incoming stream of data received through the line-in conductor on the headphone jack. Receiving a constant data stream from its line-in jack, the iPhone application's interface needed to adequately reflect it's implicit functionality. Primarily for debugging purposes, the interpreted values received within the iPhone were printed to the *Synthesize* view of the application (shown in Figure 6), allowing a user to confirm a valid data stream. Seeing the data constantly update on screen added a layer of security, and was reassuring of the protocols effectiveness; the value meter was left on the main screen of the interface to convey a similar sense of security to the end user.

Fig. 6: "Synthesize View" of the Wii Nunchuck iPhone application.

The *Settings* view of the application gives the user access to several control parameters of the application's synthesis routines. The primary mode of the application can be changed, altering the app between three possible modes: *DJ* mode, *Sine* mode, and *Drum* mode. DJ mode hooks DJ samples into the application's triggering methods, while Drum mode connects drum samples instead. Sine mode, instead of playing back pre-recorded samples, was designed to allow the user to freely produce pitches in the form of sine waves. Once sounded, these pitches could be altered with intelligible motions of either the iPhone or Wii Nunchuck. Aside from the mode-selector, the ability to utilize the iPhone's built-in accelerometer can be turned on or off. When selected to be on, the user is given a choice of reading acceleration values from either the X, Y, or Z axis of the iPhone. With these configurable settings, the user has a significant amount of control over the tones that the application will produce, as well as the behaviors that it will respond to.

V. Discussion

Though the finished product exhibits functions distinctly different than that of my original plans, I am pleased with the outcome. Utilizing third-party controllers to control an iPhone represents the backlash against the boundaries imposed by Apple and other phone manufacturers, illustrating that there is often a solution to seemingly impossible problems. Although the marketability of an Arduino rig with a Wii Nunchuck to control one application on the iPhone may not be vast, the protocol built to facilitate communication between the devices will hopefully bridge a gap, and open avenues for others to develop and experiment. The most rewarding outcome from this project would be to serve as a useful resource to another research project attempting similar feats. For those interested in browsing through the documentation for the protocol and iPhone application, consult the online repository.

A. Future Work

There are many undeveloped aspects of this project which, with time, will hopefully flourish. One of the applications biggest weaknesses is the limited capability to synthesize audio. Currently, the application features sample playback methods, as well as a sine generator. Although variation in the samples used can be modified programmatically, the end-user has no control over which samples are available, and the timbre of the tone generator. Future releases of the app will hopefully incorporate the ability to record samples from the iPhone itself, or upload them from the iTunes library. As well, a multi-waveform tone generator would broaden the scope for tonal production of the application, varying the user's aural experience.

Alongside improving the iPhone application's ability to play new and different sounds, enhancing the control of sensor data to synthesis parameters is another improvement to be made. Currently, the user is restricted to reading in a maximum of five values; three of which can be driven by the Wii Nunchuck's axial sensors (accelerometer in three axes, and joystick in two), and the two additional push buttons. Should the user want to incorporate the iPhone's onboard accelerometer data, that value replaces one of the three complex values received, and only one value from the onboard accelerometer can be read at once. Developing an interface that would provide an intelligible mapping scheme from sensor to parameter within the application is another future goal. This implementation would loosen the confines of the current application, allowing for complete user-customization in the mapping of sensor values to control audio synthesis.

These two improvements, amongst several others, would greatly enhance the functionality and usability of the application. Nonetheless, the current product is a proud example of overcoming obstacles in the hardware and software domains.

References

[1] H. Gilke, "iphone serial communication," 2 2010. [Online]. Available: http : / / hcgilje . wordpress . com / 2010 / 02 / 15 / iphone-serial-communication/

[2] A. Inc. (2010, 5) iphone programming guide. Online Documentation. [Online]. Available: http : / / developer . apple . com / iphone

[3] C. Santoro, "Mueaudioio class," June 2008.

[4] A. Inc. (2010, 5) Audio units programming guide. Online Documentation. [Online]. Available: http : / / developer . apple . com / mac / library / documentation / MusicAudio / Conceptual / AudioUnitProgrammingGuide / Introduction / Introduction . html

[5] D. A. Mann, *Audio Application Design for the iPhone Platform*, Music Engineering, University of Miami, 2009.

[6] T. E. Kurt, ""wiichuck" wii nunchuck adapter," 2 2008. [Online]. Available: http : / / todbot . com / blog / 2008 / 02 / 18 / wiichuck-wii-nunchuck-adapter-available/