Ultrasonic Speech Capture Board: Hardware Platform and Software Interface*

Carrick Detweiler and Iuliu Vasilescu

May 14, 2008

Abstract

This paper presents a hardware platform and software interface for ultrasonic aided speech recognition and speech activity detection. We call the device the Ultrasonic Speech Capture Board (USCB). The device plugs into a USB port and has an ultrasonic (40 kilohertz) transmitter and receiver as well as an audio microphone and an audio jack for an external microphone. The ultrasonic channel can be used in a pulsed mode to detect the presence and distance of the speaker. Additionally, in continuous mode, the Doppler shift in the ultrasonic channel can be used to detect speech and to aid in speech recognition.

1 Introduction

Speech detection and recognition is challenging in the most ideal environments. In noisy environments with lots of background noise or multiple, clearly audible, conversations it is even worse. Take, for instance, a kiosk in an atrium which provides directions for visitors. The kiosk must first detect that it is being spoken to and then must recognize the speech to provide the appropriate response.

A number of approaches have been taken to detect that the kiosk is being addressed. The first is to continually process received signals for valid phrases and respond when a valid phrase is detected. This is virtually impossible in a loud environment where multiple, unrelated conversations are ongoing. Related to this is the use of keywords to indicate to the system that it is being spoken to. Keying the system on particular words reduces the ease of use for first time users who may not know the keyword. Another method is to have the user press a button before speaking. All of these can be used for voice activity detection, however, they do not aid in speech recognition which is even more challenging in noisy environments.

Some have employed systems which can be used for both voice activity detection as well as speech recognition. Cameras can be used to detect the presence of a person and decide when they are speaking [3] and to aid in speech recognition (lipreading) [4, 7]. This can be costly computationally and raises privacy concerns.

Instead of using the aforementioned techniques we make use of an ultrasonic transmitter and receiver for voice activity detection [5, 8, 9] and speech recognition [6, 10]. This has the advantage that it is easy to process (single audio channel of limited frequency) and maintains privacy of the users.

In this paper we focus on a new hardware platform we developed for the capture of ultrasound data. We call the system the Ultrasonic Speech Capture Board (USCB). The USCB consists of an ultrasound emitter and receiver which operate at 40 kilohertz. It also has an audio microphone as well as a standard microphone jack for an external microphone. The system streams the data in real-time to a computer via a USB connection. In addition to the continuous ultrasound stream used in previous work, the USCB has a pulsed mode which can be used to obtain range measurements to objects in its field of view. This allows for a computationally simple method to determine if someone is in front of the kiosk.

The first part of this paper focuses on the hardware design of the USCB. We then present the Java and Matlab libraries which can be used to obtain real-time data from the USCB. A demo application which

*This paper is part of a final project for a class taught by Jim Glass at MIT

presents the real-time signal, Fourier transform, and spectrogram is also presented. Finally, we include detailed instructions for software and hardware installation.

2 Theory of Operation

2.1 Continuous Mode

In continuous mode the ultrasonic transmitter transmits continually at a fixed frequency of 40 kHz. If there is a fixed object in front of the transmitter then these will be reflected back and the receiver will receive a 40 kHz signal. If, instead, the object is moving towards the device then the signal will be Doppler shifted to a higher frequency. Similarly, an object moving away will cause a shift to a lower frequency.

When a speaker is in front of the device we see similar shifts due to the motion of the mouth. While the mouth is not moving towards or away from the USCB, the distances are changing fairly rapidly due to the mouth opening and closing, movement of the tongue, facial changes, etc. These changes in the spectrogram can then be used for voice activity detection [5, 8, 9] and speech recognition [10].

2.2 Pulsed Mode

In Pulsed mode the ultrasonic transmitter transmits a short pulsed 40 kHz signal. It then waits a period of time before emitting the next pulse. By measuring the time it takes for a reflection to return, the distance to the object can be measured. This is the method employed by ultrasonic range finders.

Pulsed mode has low computational overhead as using something as simple as a threshold on the received signal is sufficient to obtain the range. This means it can be easily used as a first pass filter to determine if a person is in front of the device. Additionally, we suspect that the obtained spectrogram from the continuous mode may vary based on the distance of the device to the speaker. Thus, this information can be used in conjunction with continuous mode to improve results or to provide user feedback such as "please move closer."

To the best of our knowledge having a pulsed mode to aid in speech detection and recognition is unique to our system.

3 Hardware

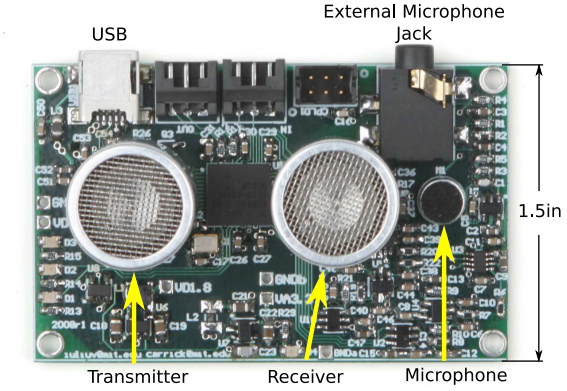


Figure 1: Picture of the USCB(life size).

Figure 1 shows an image of the USCB and Figure 2 shows a block diagram. At the heart of the device is a Xilinx XC2C256 CoolRunner II CPLD (complex programmable logic device). This generates a 40 kilohertz

square wave with variable duty cycle which is input into the ultrasonic emitter. By controlling the duty cycle the output power of the emitter can be controlled. The code for the CPLD can be found in appendix A.

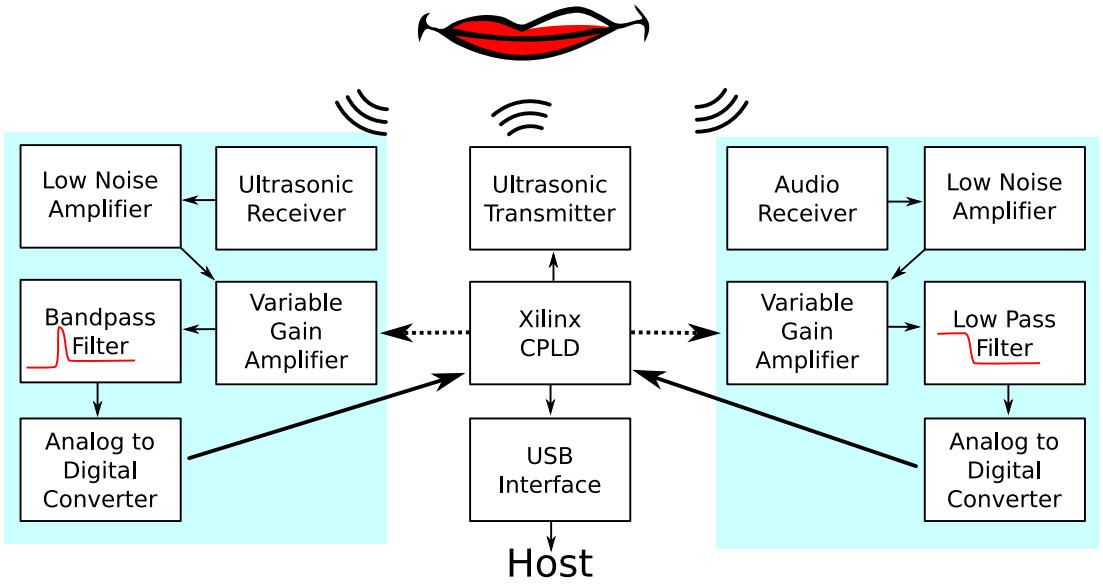


Figure 2: Block digram of the USCB.

The reflected signal is captured by the ultrasonic receiver. This signal passes through a low noise amplifier (LNA) followed by a variable gain amplifier (VGA) which allows us to control the sensitivity of the receiver (36dB range). It then passes through a 40 kHz bandpass filter. Finally, the filtered signal goes into a 16-bit analog to digital converter (Analog Devices AD7680). The CPLD reads the ADC at 24 kHz causing the 40kHz signal from the ultrasonic receiver to be aliased to 8kHz.

The audio is captured from an internal or external microphone and is processed similarly to the ultrasound channel. The main difference is that we use a low pass filter with a cutoff of around 8 kHz instead of the bandpass filter.

At this stage we have the digital representation of both the ultrasound and audio channels on the CPLD. The CPLD then formats the data for transfer over USB using an FTDI FT245 USB chip. The result is pure digital streams of both channels to the host computer.

Previous capture devices [5, 8, 9] mixed the aliased signals with the audio signal and transmitted this as an analog audio signal. This method has problems with noise introduced by long wires, differences in the quality of the captured data depending on the sound card of the computer, and the fact that an additional step to separate the ultrasonic data from the audio data must be performed.

3.1 Modes and Configuration

The USCB has a number of configuration options which can be adjusted at runtime (see Section 3.3 the raw commands to set these parameters). The gain on both the audio and ultrasonic channel can be adjusted from 30dB to 66dB. Additionally, the ultrasonic transmitter input can be adjusted to vary between .2V and 10V peak to peak. Typically we use a gain on the audio channel of 50dB and a gain on the ultrasonic channel of 45dB. We use the ultrasonic transmitter at the minimum output level.

In addition to being able to adjust the power and gain we have two different modes. In the first mode we output the ultrasonic signal continuously. In the other mode we send out ultrasonic pulses. See Section 2 for more information. In addition to the power and gain settings, in pulsed mode we also control the number of periods in the pulse (even number between 2-512) and the delay between the pulses in periods (multiple

of eight between 8 and 2040). This gives us a pulse length between 1/20 and 12.8 milliseconds and a delay of .2 and 51 milliseconds. Using the maximum delay we can measure a distance of up to about 17 meters (assuming we can detect the reflected signal at that distance).

3.2 Raw Data

The USCB appears as a serial port on computers by using an FTDI Chip driver (see Section 6 for details on installation). By default no data is streamed over the serial port. This prevents operating systems from being confused by the data being sent (most notably, Windows sometimes decides that this is a serial port mouse, causing interesting mouse behavior). To enable the sending of data you must set the enable bit in the mode selection packet. See Section 3.3 for details on this.

The data is sent in five byte packets. The first byte is a status byte, the second byte is the most significant byte (MSB) of the audio channel, the third byte is the least significant byte (LSB) of the audio channel, the fourth is the MSB of the ultrasound channel, and finally the fifth byte is the LSB of the ultrasound channel. In summary the packet structure is:

```
STATUS, AUDIO MSB, AUDIO LSB, ULTRASOUND MSB, ULTRASOUND LSB
```

The status byte is 1 if the ultrasonic transmitter is currently transmitting and zero otherwise. This allows detection of when pulses are being sent when in pulsed mode. The audio and ultrasound data are the 14 most significant bits of the ADC readings. There is no start byte in this packet, but since we know the upper seven bits of the status byte and upper two bits of the MSBs are zero it is easy to synchronize on these bits.

3.3 Commands

Commands for setting parameters can be sent at any time. The USCB starts up with some default settings but it is recommended that you send the desired settings every time after power on. It will then remember the settings until it is unplugged from the USB port. The first two bits of a command indicate what the command is. Thus we have four commands: Mode Selection, Gain, Power, and Pulsed Mode Settings.

3.3.1 Mode Selection

The mode selection command controls whether the board is in pulsed mode or continuous mode. Additionally, it controls whether the board is streaming data over the serial port or not. This is a single byte command. Send `0x88` to enable and set to continuous mode. Send `0x98` to enable and set to pulsed mode. Send `0x80` to disable streaming of data over the serial port.

3.3.2 Gain

The gain on the ultrasound and audio receivers are set by sending one byte with the upper two bits set to zero, the next three bits represent the audio gain (0-7) and the lower 3 bits representing the ultrasound gain (0-7). For instance, sending `0x09` will set both gains to 1 and sending `0x14` will set the audio gain to 2 and the ultrasonic gain to 4.

3.3.3 Power

The power of the ultrasonic transmitter is set by sending one byte with the upper bit set to 0 and the next bit set to 1. The remaining 6 bits are used to set the power in a valid range between 0 and 50. Setting this to zero disables sending of ultrasonic pulses. For instance, sending the byte `0x48` will set the power to 8 and sending `0x54` will set the power to 20.

3.3.4 Pulsed Mode Settings

The pulsed mode settings command is a two byte command. This command is used to set the number of periods sent in pulsed mode and the length of the delay (see Section 3.1 for details). The first two bits of the first byte are 1. The next two bits are used to indicate setting the number oscillations (two bits zero) or the delay (two bits 01). The second byte is used to set the value of the parameter which has a valid range between 0 and 255. The number of periods sent is actually twice the parameter and the delay is eight times the parameter.

Sending `0xC005` will set the number of periods sent to $2 * 5$ and sending `0xC0BC` would set it to $2 * 188$. Sending `0xD008` would set the delay to $8 * 8$ periods and sending `0xD0C3` would set it to $8 * 195$.

4 Software

We have written a Java Library for easily obtaining the data from the USCB. In addition we have written Matlab wrappers for the Java code to allow for direct capture and processing of the data in Matlab. We have also implemented an example Java program which makes use of the library to display the signal, Fourier decomposition and spectrogram in real time. This example program also allows configuration of all of the parameters of the USCB and is useful to quickly see the effect of changing parameters or setups.

4.1 Java Library

In this section we summarize the most structure and important commands of the Java library for the USCB. The complete API reference can be found in the appendix section D and the code can be found in appendix section B. For information on installing the necessary software see Section 6.

Our Java library relies on the Java serial port library RXTX [2]. This library provides the necessary native drivers to allow cross-platform Java code to access serial ports. We use this library in our `SerialBasic` class which handles connecting to specified serial ports. The commands for controlling the USCB are contained in the `UltraSound` class. The constructor for this class takes as an argument the string representing the port name of the connected USCB. For example:

```
UltraSound us = new Ultrasound(‘‘/dev/ttyUSB0’’);
```

This creates the connection to the specified serial device. If there is a problem with the connection an error message will be printed on the console. It is also possible to check if it is properly connected by calling the function `isConnected()` which will return false if there is a problem with the serial connection.

The next step in using the library is to set the gain, power, mode and pulse parameters. For example:

```
int audioGain = 5;
int ultrasoundGain = 4;
int ultrasoundPower = 1;
int periodsOn = 40;
int periodsOff = 1600;

UltraSound us = new UltraSound(‘‘/dev/ttyUSB0’’);
if(!us.isConnected()){
  System.out.println(‘‘Error connected to device /dev/ttyUSB0’’);
  return;
}
us.setGain(audioGain,ultrasoundGain);
us.setUltraSoundPower(ultrasoundPower);
us.setPulseParams(periodsOn/2, periodsOff/8);
//Set the mode to continuous operation, this also enables data output
us.setContinuous(true);
```

The creation of the `UltraSound` objects creates a thread which continuously reads the data coming from the serial port. The data it reads is parsed and then cached. The size of the cache can be found in `UltraSound.CACHE_SIZE` and is 220000 which should be more than sufficient for most applications (this provides about 10 seconds of data caching). If the cache overflows an warning is printed to the console and the oldest data is discarded. The amount of data currently cached can be found by calling `us.available()`.

The number of overflows there have been can be determined by calling `us.getNumOverflows()`. The value returned should be zero or at least constant, otherwise data is not being read from the cache fast enough. Additionally the number of times the data stream has been resynchronized can be obtained by calling `us.getNumSyncs()`. This should be constant once the program starts. If this is changing (which would cause "synchronizing" to be printed to the console) then this indicates that data is not being read from the serial port fast enough. This is most likely caused by the reading thread being starved by some other process taking up too much time. By default warnings are printed to the console if there are overflows or resyncs. This can be disabled using the command `us.setWarnings(false)`.

Data can be obtained by subsequent calls to `us.read()`. This returns the oldest first `[audio,ultrasound,status]` values in an array of `int`s. This is a blocking call, which means if there is no data currently available it will not return until there is data. Caution should be exercised when calling this function if data streaming from the device has been disabled via `us.setEnabled(false)`. The safe way to read the data is as follows:

```
if(us.available() > 0){
  int data[];
  data = us.read();
  audioVal = data[0];
  ultrasoundVal = data[1];
  status = data[2];
}
```

The data read is the synchronized values, so the readings were taken at the same time. The audio and ultrasound values range from 0 to 16383 and the status is one if the ultrasonic transmitter is transmitting at this time and zero otherwise. The oldest value in the buffer is read first.

4.2 Matlab Code

We have also implemented a Matlab Library for capturing data. The code can be found in appendix C. This library calls the Java library by using Matlab's built in ability to use Java code. Unfortunately, because of the overhead involved in calling Java functions from Matlab, realtime capture of data in Matlab is not possible on the machines tested. Instead, we rely on the internal buffer Java maintains to allow capturing up to 10 seconds of data in the Java buffer and then transferring this into a Matlab array. While not ideal, this does allow quick analysis of data without having to load in data captured from another method such as our sample Java application described in Section 4.3.

The two functions for data acquisition are `ultrasoundInit` and `ultrasoundRead`. To initialize the capture run the command:

```
us = ultrasoundInit('/home/carrick/ultrasound/','/dev/ttyUSB0')
```

Where the first argument is the path to the Java code and the second is the path to the COM port of the USCB[1]. The result return is a pointer to the Java `UltraSound` object.

To capture 5 seconds of data run the command:

```
data = ultrasoundRead(us,5)
```

[1]Note that the code path must already have the compiled class files `SerialBasic.class` and `UltraSound.class` as well as the `RXTXcomm.jar` jar file. Depending on the version of the Java compiler `javac` and the version of Matlab the Java files may need to be compiled with the flag `-target 1.5` to specify that they need to be compatible with the 1.5 version of the JVM that Matlab uses. This is already done properly in the Java example application Makefile described in Section 4.3, there is no need to recompile these files if the sample application runs.

The data returned is the samples where `data(:,1)` are the audio readings, `data(:,2)` are the ultrasound readings and `data(:,3)` are the status bytes. This command uses the default values of the audio gain (5), ultrasonic gain (4) and ultrasonic power (1). However, it is possible adjust these when calling `ultrasoundRead`, the full format of the command is:

```
data = ultrasoundRead(us, seconds, audiogain, ultrasonicgain, ultrasonicpower)
```

When running this command a message "Recording data started" will be printed indicating that the USCB has started recording data (the lights on the USCB will also turn on). After the specified number of seconds has elapsed the message "Recording data stopped" will be printed and the lights will turn off. The data will then be downloaded from the cache of Java ultrasound object. This typically takes about twice as long as the capture time (depending on the speed of the machine used). Note that there is a maximum of about 10 seconds of recording time.

4.3 Example Java Application

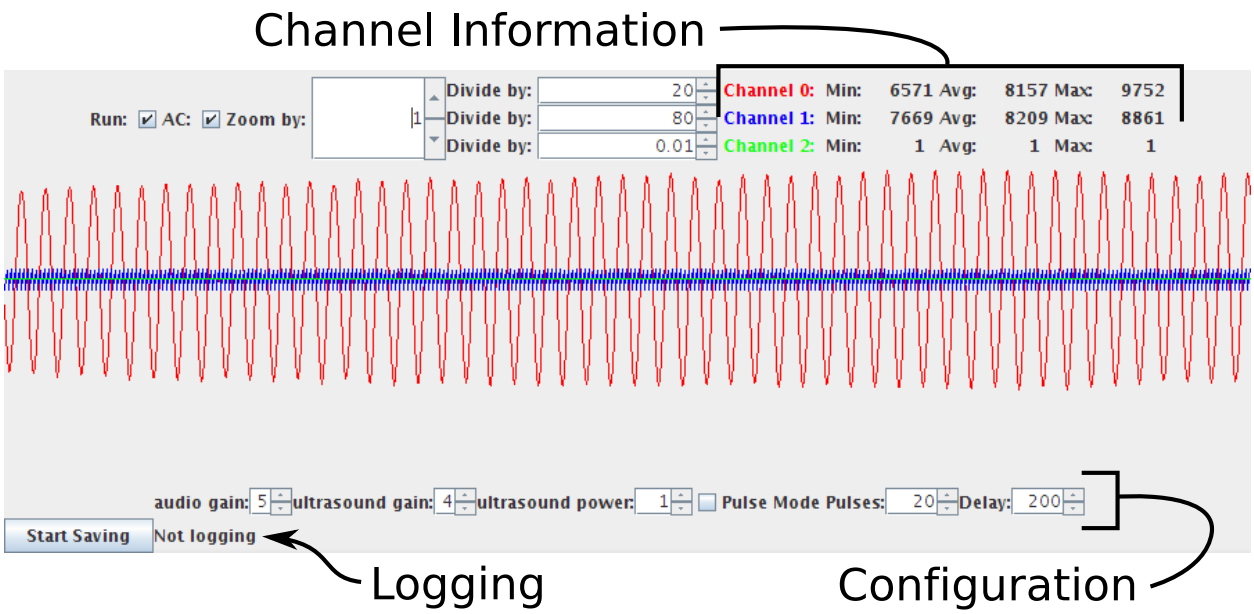


Figure 3: The main window of the Java Application. This shows the received signals and allows configuration of the parameters. The red signal is the audio and the blue is the ultrasonic signal.

We used our Java library to implement an example application which displays the data from the USCB in real time and allows configuration of all of the parameters. The application has two windows. The top window, shown in Figure 3, displays a real time view of the signals received from the audio channel (red), the ultrasound channel (blue) and the status byte (green). The figure shows the signal when whistling. Additionally, the lower portion of the window allows configuration of the gains, power, and pulse mode parameters. There is also a button to start saving data[2].

This window also displays raw channel information such as the minimum, average and maximum over the displayed window. To the left of this are boxes which allow the raw values to be divided by a value, this controls the height of the displayed signals. Further to the left is a box to select the zoom (from .01 to 10) which allows increasing or decreasing the amount of time over which the signal is displayed.

[2]Note when saving data if packets are dropped or lost (indicated by error messages on the console) it may be advisable to uncheck the run box on the signal and FFT displays.

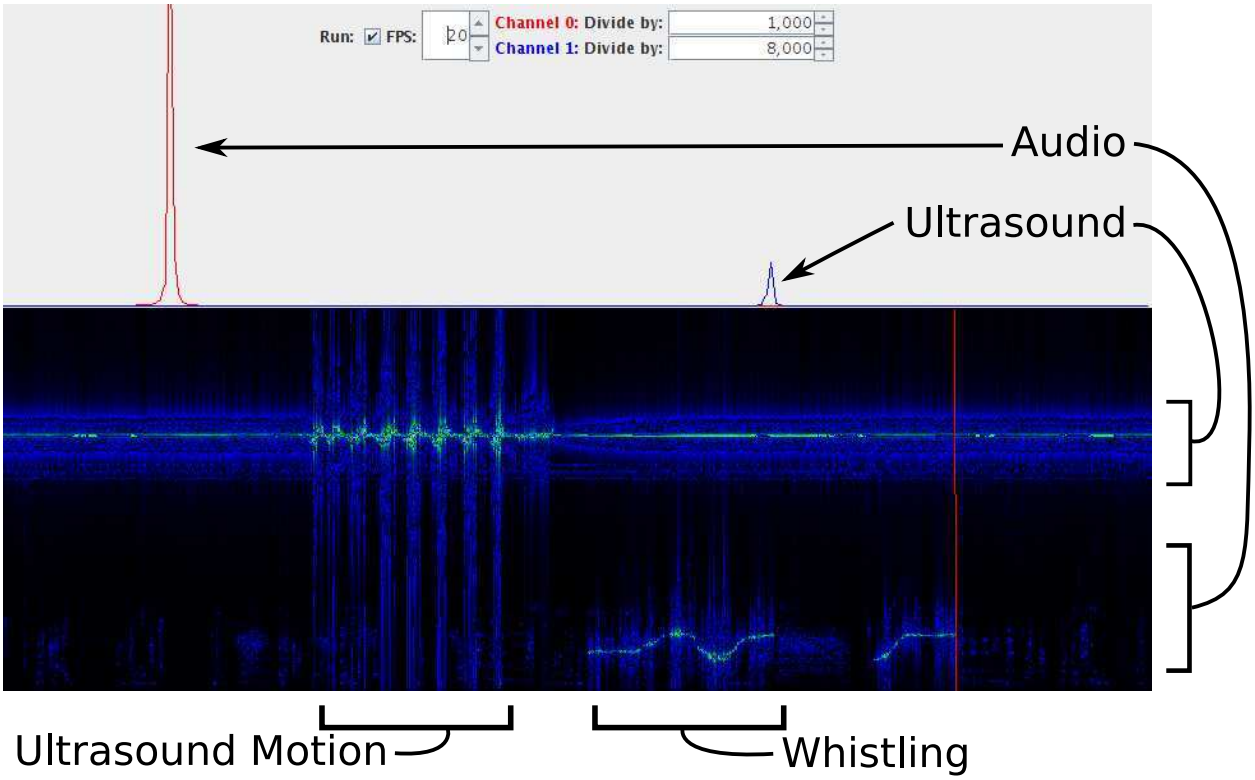


Figure 4: The spectrogram window of the Java Application. This shows the received signals Fourier transform (top) as well as the spectrogram (bottom). At the top the red signal is the audio and the blue is the ultrasonic signal. Below the spectrogram shows the result on the ultrasound of a hand moving back and forth and whistling on the audio channel. The red line on the spectrogram shows the current time.

The other window, shown in Figure 4, displays the realtime Fourier transform of the signals and the resultant spectrogram. These make use of Michael Flanagan Fourier transform library [1]. We plot the power spectrum produced on a sliding square window of 512 samples.

The sample application requires the drivers and programs described in Section 6 as well as GNU Make utilities. To run it use the command:

```
make run PORT=/dev/ttyUSB0
```

where `/dev/ttyUSB0` should be replaced by the port where the USCB is connected. Rebuilding the Java programs should not be required, however, if changes are made, running the command `make` will rebuild the program.

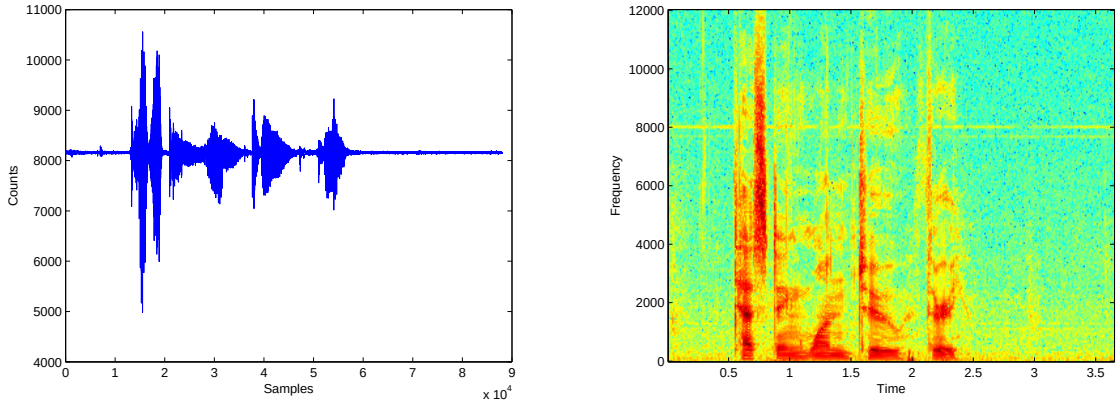
5 Sample Data

In this section we present some samples of collected data both in continuous mode with speech and in pulsed mode for range finding.

5.1 Continuous Mode

Figure 5 shows the time and frequency domain plots for the audio channel when speaking the phrase "Testing one two three." This was a recording over 4 seconds with an audio and ultrasound gain of 4 and an ultrasonic

power level of 1. In the spectrogram there is a slight line at 8kHz which is due to the microphone picking up some of the signal from the ultrasound. This can be minimized further by decreasing the gain on the audio channel or by using an external microphone.

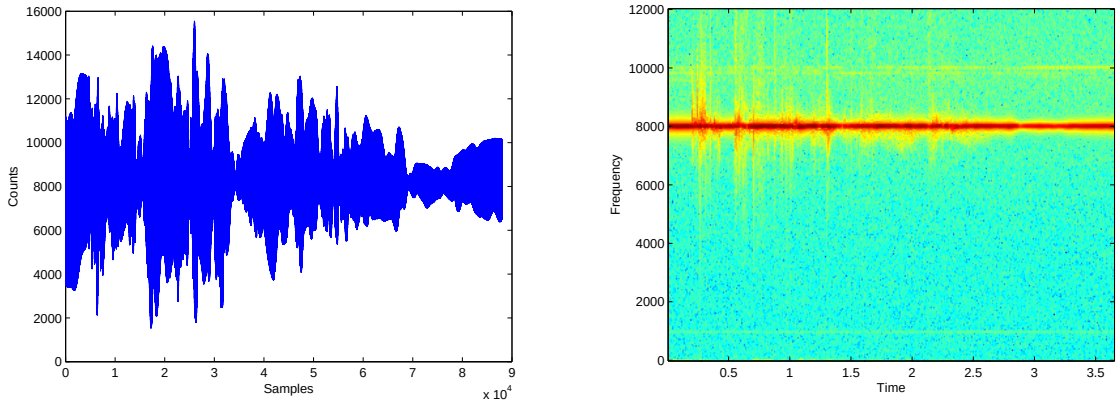


(a) Plot of the captured speech data.

(b) Spectrogram of captured speech data.

Figure 5: Speech data from the phrase "Testing one two three" captured using the Matlab interface.

Figure 6 shows the time and frequency domain plots for the ultrasonic channel for the same experiment. There is clear variation in the time and frequency domain plots.



(a) Plot of the captured ultrasound data.

(b) Spectrogram of captured ultrasound data

Figure 6: Ultrasound data from the phrase "Testing one two three" captured using the Matlab interface.

5.2 Pulsed Mode

Figure 7 shows the output of our Java user interface when in ranging mode. The top part of the figure is ranging to an object at a half meter and the lower part is ranging to an object about a meter away. The green line shows when the ultrasonic transmitter is transmitting (when it is low). The blue is the received signal.

Shortly after transmitting starts there is some signal received. This is the direct coupling between the transmitter and the receiver. Shortly thereafter is a larger pulse which is the reflected signal. By measuring

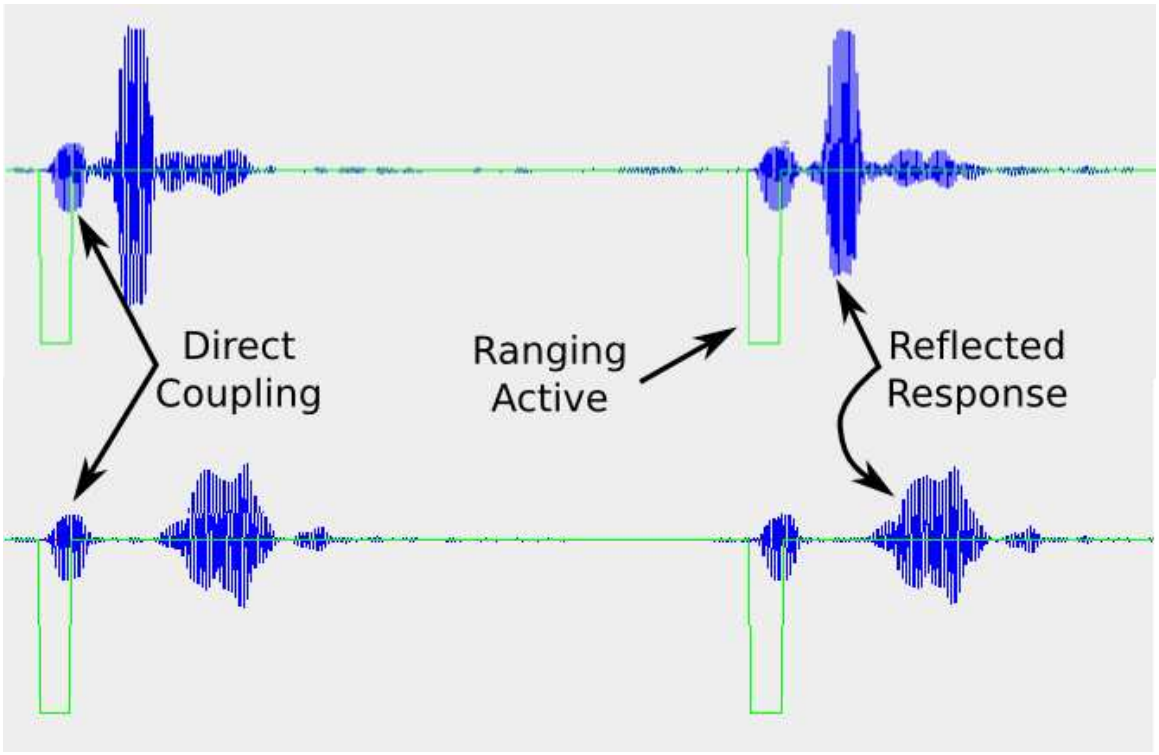


Figure 7: On top the signal from ranging to an object at approximately half meter, below ranging to an object a meter away. The green line indicates pulses being sent when low, the blue is the received ultrasound pulses.

the time between the start of sending the pulse and the start of the response the range to the object can be computed.

In the upper part of the figure the response has a stronger intensity and occurs more quickly after pulse was sent than in the lower part of the figure where it ranging to an object further away. There are also some signals received after the main initial response. These are caused by secondary reflections off of objects further away or from multi-path (bouncing off of multiple objects before being received).

6 Software Installation

Software installation is fairly straight forward. The main requirements are the FTDI VCP serial port drivers (not needed for linux), Java 1.5 JDK or greater (not needed for OS X), and libRXTX. These can be found at the following websites:

```
http://www.ftdichip.com/Drivers/VCP.htm
http://java.sun.com/javase/downloads/index.jsp
http://users.frii.com/jarvi/rxtx/
```

Download the latest version of each of these. When building and running java programs with the library make sure to include RXTXcomm.jar in the class path. On a Debian GNU/Linux machine you can install libRXTX by doing:

```
sudo apt-get install librxtx-java
```

On an Ubuntu GNU/Linux machine you can install everything needed by doing:

```
sudo apt-get install librxtx-java sun-java6-jdk
```

The only caveat on Ubuntu is you may have to change the default version of Java which is used by using the command:

```
sudo update-alternatives --config java
```

and setting the version of Java used to the Sun jvm.

References

[1] Michael thomas flanagan's java library: Fourier transforms. website. `http://www.ee.ucl.ac.uk/~mflanaga/java/FourierTransform.html`.

[2] RXTX: The prescription for transmission. website. `http://users.frii.com/jarvi/rxtx/`.

[3] P. D. Cuetos, C. Neti, and A. Senior. Audio-visual intent-to-speak detection for human-computer interaction. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, volume 6, pages 2373–2376 vol.4, 2000.

[4] T. Hazen, K. Saenko, C. La, and J. Glass. A segment-based audio-visual speech recognizer: Data collection, development and initial experiments. In *Proceedings of ICMI*, State College, PA, Oct. 2004.

[5] R. Hu and B. Raj. A robust voice activity detector using an acoustic doppler radar. In *Automatic Speech Recognition and Understanding, 2005 IEEE Workshop on*, pages 319–324, 2005.

[6] D. Jennings and D. Ruck. Enhancing automatic speech recognition with an ultrasonic lip motion detector. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 868–871 vol.1, 1995.

[7] S. jong Lee, J. Park, and E. kyeu Kim. Speech activity detection with lip movement image signals. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 403–406, 2007.

[8] K. Kalgaonkar, R. Hu, and B. Raj. Ultrasonic doppler sensor for voice activity detection. *Signal Processing Letters, IEEE*, 14:754–757, 2007.

[9] K. Kalgaonkar and B. Raj. An acoustic doppler-based front end for hands free spoken user interfaces. In *Spoken Language Technology Workshop, 2006. IEEE*, pages 158–161, 2006.

[10] B. Zhu, T. Hazen, and J. Glass. Multimodal speech recognition with ultrasonic sensors. Antwerp, Belgium, Aug. 2007.

A Verilog Code

The code for the Xilinx CPLD is written in Verilog and follows:

```
////file main.v////

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:44:37 04/22/2008
// Design Name:
// Module Name:    main
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module main(CLK, LED, NTXE, NRXF, WR, NRD, USBDATA,
            AU_GAIN, AU_N_CS, AU_DATA, AU_SCK,
US_GAIN, US_N_CS, US_DATA, US_SCK,
TRANSMIT);
   `include "../lib/utils.v"
   parameter             LED_COUNT       = 3;
parameter             USB_SIZE        = 8;
parameter             GAIN_SIZE       = 3;
parameter             TRANSMIT_SIZE   = 2;
parameter             STATE_SIZE      = 5;
parameter             DATA_SIZE       = 16;
parameter             TR_TICKS        = 50;
localparam            TR_DIV_SIZE     = clog2(TR_TICKS);
parameter             CMD_SIZE        = 2;
parameter             CMD_GAIN        = 2'b00;
parameter             CMD_DUTY        = 2'b01;
parameter             CMD_MODE        = 2'b10;
parameter             CMD_SET         = 2'b11;
parameter             CMD_1_SIZE      = 2;
parameter             CMD_1_TICKS     = 2'b00;
parameter             CMD_1_PAUSE     = 2'b01;
parameter             MODE_SIZE       = 2;
parameter             MODE_CONTINUOUS = 2'b00;
parameter             MODE_PULSED     = 2'b01;
parameter             PULSE_SIZE      = 11;
parameter             STREAM_POS      = USB_SIZE-CMD_SIZE-MODE_SIZE-1;
parameter             STATUS_US_RUN   = 0;


input                       CLK;
output [LED_COUNT-1:0]      LED;
input                       NTXE, NRXF;
output                      WR, NRD;
inout  [USB_SIZE-1:0]       USBDATA;
output [GAIN_SIZE-1:0]      AU_GAIN;
output [GAIN_SIZE-1:0]      US_GAIN;
output                      AU_N_CS, AU_SCK, US_N_CS,  US_SCK;
input                       US_DATA, AU_DATA;
output [TRANSMIT_SIZE-1:0] TRANSMIT;


reg    [STATE_SIZE-1:0]     state      = 24;
   reg                         ncs        = 1;
reg    [DATA_SIZE-1:0]      au_buf;
reg    [DATA_SIZE-1:0]      us_buf;
reg    [USB_SIZE-1:0]       out_buf;
reg                         wr         = 1;
reg                         NRD        = 1;
reg                         tr_side    = 0;
reg    [TR_DIV_SIZE-1:0]    tr_counter = 0;
reg    [TRANSMIT_SIZE-1:0] TRANSMIT   = 0;
reg                         out_en     = 1;
reg    [GAIN_SIZE-1:0]      AU_GAIN    = 7;
reg    [GAIN_SIZE-1:0]      US_GAIN    = 4;
reg    [TR_DIV_SIZE-1:0]    tr_duty    = 0;
reg    [USB_SIZE-1:0]       last_usb   = 0;
reg    [MODE_SIZE-1:0]      mode       = MODE_CONTINUOUS;
reg                         streaming  = 0; // by default no data is streamed on the USB
reg    [USB_SIZE-1:0]       pulsed_ticks  = 10;
reg    [USB_SIZE-1:0]       pause_ticks   = 10;
reg    [USB_SIZE-1:0]       status        = 0;
reg    [PULSE_SIZE-1:0]     pulse_counter = 0;

wire                        clk2400k, clk600k, clk4000k;
   wire   [USB_SIZE-1:0]       usb_in     = USBDATA;
wire   [1:0]                usb_cmd    = usb_in[USB_SIZE-1:USB_SIZE-2];

assign  AU_N_CS          = ncs;
assign  US_N_CS          = ncs;
assign  AU_SCK           = ncs | clk600k;
assign  US_SCK           = ncs | clk600k;
assign  USBDATA          = (out_en && NRD)?out_buf:8'bz;
assign  LED[0]           = (~NTXE) & NRXF;
assign  LED[1]           = ~streaming;
assign  LED[2]           = tr_duty==0;
assign  WR               = wr | ~streaming;

divider       #(.COUNTER_SIZE(3), .DIVIDER(5))
div1(.CLK(CLK), .OUT(clk2400k));

power2divider #(.COUNTER_SIZE(2))
div2(.CLK(clk2400k), .OUT(clk600k));

divider       #(.COUNTER_SIZE(2), .DIVIDER(3))
div3(.CLK(CLK), .OUT(clk4000k));


always @(posedge clk600k)
if(state == 24) state <= 0;
else            state <= state+1;

always @(posedge clk600k)
case(state)
0:  begin
ncs <= 0;
if(!NRXF) out_en <= 0;
      NRD <= 1; // for safety
    end
1:  if(!out_en) NRD    <= 0;
2:  if(!NRD) begin // reading input data
if(last_usb[USB_SIZE-1:USB_SIZE-CMD_SIZE] == CMD_SET) begin
case(last_usb[USB_SIZE-CMD_SIZE-1:USB_SIZE-CMD_SIZE-CMD_1_SIZE])
CMD_1_TICKS: pulsed_ticks <= usb_in;
CMD_1_PAUSE: pause_ticks  <= usb_in;
endcase
last_usb <= 0;
end else begin
case(usb_cmd)
CMD_GAIN: {AU_GAIN, US_GAIN} <= usb_in[GAIN_SIZE*2-1:0];
CMD_DUTY: tr_duty            <= usb_in[TR_DIV_SIZE-1:0];
CMD_MODE: begin
          mode               <= usb_in[USB_SIZE-CMD_SIZE-1:USB_SIZE-CMD_SIZE-MODE_SIZE];
 streaming          <= usb_in[STREAM_POS];
 end
CMD_SET : last_usb           <= usb_in;
endcase
end
    end
3:  NRD    <= 1;
4:  begin out_en <= 1; out_buf <= status; end
5:  wr     <= 0;
6:  wr     <= 1;
8:  out_buf <= au_buf[7:0];
9:  wr <= 0;
10: begin wr <= 1;  out_buf <= us_buf[10:2]; end
11: wr <= 0;
12: wr <= 1;
16: begin ncs <= 1; out_buf <= au_buf[7:0]; end
17: wr <= 0;
18: begin wr <= 1;  out_buf <= us_buf[7:0]; end
19: wr <= 0;
20: wr <= 1;
endcase

always @(negedge clk600k) if(~ncs) begin
au_buf <= {au_buf[DATA_SIZE-2:0], AU_DATA};
us_buf <= {us_buf[DATA_SIZE-2:0], US_DATA};
end

always @(posedge clk4000k)
if(tr_counter == TR_TICKS-1) tr_counter <= 0;
else                         tr_counter <= tr_counter + 1;

always @(posedge clk4000k) if(tr_counter == TR_TICKS-1 && tr_side) begin // each period
case(mode)
MODE_CONTINUOUS: status[STATUS_US_RUN] <= 1;
MODE_PULSED: begin
if(pulse_counter==0) begin
if(status[STATUS_US_RUN]) begin
pulse_counter <= pause_ticks<<3;
end else begin
pulse_counter <= pulsed_ticks<<1;
end
status[STATUS_US_RUN] <= ~status[STATUS_US_RUN];
end else begin
pulse_counter <= pulse_counter-1;
end
end
endcase
end

always @(posedge clk4000k) begin
if(tr_counter == 0 && tr_duty != 0 && status[STATUS_US_RUN])
TRANSMIT[~tr_side] <= 1;
if(tr_counter == tr_duty || tr_counter == 0)
TRANSMIT[tr_side]  <= 0;
if(tr_counter == 0)
tr_side            <= ~tr_side;
end

endmodule


////file divider.v////
module divider(CLK, OUT);
  parameter DIVIDER = 434;
  parameter COUNTER_SIZE = 16;
```

```
  input CLK;
  output OUT;

  reg [COUNTER_SIZE-1:0] counter = DIVIDER-1;
  reg                    OUT     = 0;

  always @(posedge CLK) begin
   if(counter == DIVIDER-1) begin
     counter <= 0;
  OUT     <= 1;
   end else  begin
  counter <= counter+1;
  if(counter == DIVIDER/2-1)
    OUT<=0;
end
  end

endmodule


////file power2divider.v////
`timescale 1ns / 1ps

module power2divider(CLK, OUT);
  parameter COUNTER_SIZE = 4;

  input  CLK;
  output OUT;

  reg [COUNTER_SIZE-1:0] counter = 0;

  assign OUT = counter[COUNTER_SIZE-1];

  always @(posedge CLK)
    counter <= counter + 1;

endmodule


////file utils.v////
  function integer clog2;
    input [31:0] value;
    for(clog2=0; value>0; clog2=clog2+1)
      value = value >> 1;
  endfunction
```

B Java Code

The following is the code found in UltraSound.java and SerialBasic.java.

```
/**
 *
 * <p>Copyright (C) 2007 Carrick Detweiler and
 *                       Massachusetts Institute of Technology</p>
 *
 * <p>This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.</p>
 *
 * <p>This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.</p>
 *
 * <p>You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc., 59
 * Temple Place - Suite 330, Boston, MA 02111-1307, USA.</p>
 *
 * $Id: ultrasonicspeech.tex 1160 2008-05-14 19:41:28Z carrick $
 **/

import java.util.*;

public class UltraSound{
  /**
   * <p>SVN version info.</p>
   **/
  private static final String svnid =
    "$Id: ultrasonicspeech.tex 1160 2008-05-14 19:41:28Z carrick $";

  /**
   * Our serial port
   **/
  SerialBasic serial;

  /**
   * <p>Number of samples to cache internally.  If we overflow our cache
   * we just discard the oldest data.</p>
   **/
  public static final int CACHE_SIZE = 220000;

  /**
   * <p>Cache for the audio channel</p>
   **/
  private int audio[] = new int[CACHE_SIZE];

  /**
   * <p>Cache for the status byte.</p>
   **/
  private int status[] = new int[CACHE_SIZE];

  /**
   * <p>Cache for the ultrasound channel.</p>
   **/
  private int ultrasound[] = new int[CACHE_SIZE];

  /**
   * <p>Number of elements we currently have cached</p>
   **/
  private volatile int numCached = 0;

  /**
   * <p>Index into both audio and ultrasound which indicates the next
   * place we will write data.</p>
   **/
  private volatile int writeIndex = 0;

  /**
   * <p>True to enable warning outputs on buffer overflows, etc.</p>
   **/
  private boolean shouldWarn = true;

  /**
   * <p>The number of times our buffer has overflowed.</p>
   **/
  private long overflowCount = 0;

  /**
   * <p>The number of times we have resynchronized.</p>
   **/
  private long syncCount = 0;

  /**
   * <p>Blocks waiting for the next pair of readings from the serial port
   * and then adds them to our local cache.</p>
   **/
  private void getNextReadings(){
    int b0, b1, b2;

    b0 = serial.readByteBlock();
    b1 = serial.readByteBlock();
    b2 = serial.readByteBlock();
    //Synchronize on the known zero bits in our data
    while(((b0 & 0xFE) != 0) ||((b1 & 0xC0) != 0) ||((b2 & 0xC0) != 0)){
      b0 = b1;
      b1 = b2;
      b2 = serial.readByteBlock();
      syncCount++;
      if(shouldWarn){ System.out.println("resynchronizing"); }
    }

    //Synchronize this section so we don't have conflicts with other
    //threads reading our data
    synchronized(this){
      //Perform the update
      status[writeIndex] =  b0;
      audio[writeIndex] =  (b1<<8) + serial.readByteBlock();
      ultrasound[writeIndex] =  (b2<<8) + serial.readByteBlock();
      writeIndex++;
      //Wrap
      if(writeIndex >= CACHE_SIZE){
        writeIndex = 0;
        //System.out.println("over");
      }

      //Handle overflows appropriately, just dump oldest data
      if(numCached < CACHE_SIZE){
        numCached++;
      }else{
        overflowCount++;
        if(shouldWarn){ System.out.println("UltraSound cache overflow"); }
      }
    }
  }

  /**
   * <p>Collect data thread.  This continuously reads the serial port
   * looking for data.  On startup this sleeps for a couple of seconds
   * before reading data to let everything else startup so we don't
   * overflow our internal buffer.</p>
   **/
  Thread collectThread = new Thread(){
      public void run(){
        //Sleep at first to let other things start up
        try{
          sleep(2000);
        }catch(Exception e){
          e.printStackTrace();
          return;
        }
        if(serial.readByteBlock() == -1) return;
        while(true){
          getNextReadings();
        }
      }
    };

  /**
   * <p>Returns the number of samples available.</p>
```

```
 *
 * @return the number of samples in our cache
 **/
public int available(){
  return numCached;
}

/**
 * <p>Blocking read of the values of the audio and ultrasound.  The
 * audio and ultrasound readings are 14 bit values.  The status byte
 * is also returned which will be 1 if it is pulsing and 0 if it is
 * not.  The status bit allow synchronization with the sending of
 * the pulse and the receipt of a pulse in pulsed mode.</p>
 *
 * <p>The other version of read ({@link #read(int[] vals)}) is
 * slightly more efficent as new arrays don't have to be allocated.</p>
 *
 * @return array containing the [audio,ultrasound,status] reading
 **/
public int[] read(){
  int vals[] = new int[3];
  read(vals);
  return vals;
}

/**
 * <p>Blocking read of the values of the audio and ultrasound.  The
 * audio and ultrasound readings are 14 bit values.  The status byte
 * is also returned which will be 1 if it is pulsing and 0 if it is
 * not.  The status bit allow synchronization with the sending of
 * the pulse and the receipt of a pulse in pulsed mode.</p>
 *
 * <p>The array passed as an argument is used to store the results,
 * and is returned.</p>
 *
 * @param vals the array to put the data into, must be of at least size 3
 * @return array containing the [audio,ultrasound,status] reading
 **/
public int[] read(int[] vals){
  while(true){
    synchronized(this){
      if(numCached > 0){
        vals[0] = audio[(writeIndex-numCached+CACHE_SIZE)%CACHE_SIZE];
        vals[1] = ultrasound[(writeIndex-numCached+CACHE_SIZE)%CACHE_SIZE];
        vals[2] = status[(writeIndex-numCached+CACHE_SIZE)%CACHE_SIZE];
        numCached--;
        return vals;
      }
    }

    try{
      Thread.sleep(10);
    }catch (Exception e){
      e.printStackTrace();
    }
  }
}

/**
 * <p>Set the gain of the audio and ultrasound.  These must be in the
 * range 0-7, if they are not they are clipped.</p>
 **/
public void setGain(int audio, int ultrasound){
  if(audio < 0) audio = 0;
  if(audio > 7) audio = 7;
  if(ultrasound < 0) ultrasound = 0;
  if(ultrasound > 7) ultrasound = 7;
  //Command bits 8 and 7 zeros, 3 bits audio, 3 bits ultrasound
  serial.writeByte((audio<<3)+ultrasound);
}

/**
 * <p>Sets the ultrasound output power level.  Must be between 0-50,
 * otherwise clipped.</p>
 **/
public void setUltraSoundPower(int level){
  if(level < 0) level = 0;
  if(level > 50) level = 50;
  //Comand bits 8 and 7 are 01, 6 value bits
  serial.writeByte(0x40+level);

}

/**
 * <p>Set it to continuous mode if true, else to pulsed mode.  Pulsed
 * mode is setup via {@link #setPulseParams}.  This also enables
 * streaming of data over the serial port.  Note, this just sends
 * the bytes to set this mode, it does not verify that we are
 * actually in this mode.</p>
 *
 * <p>In continuous mode the ultrasound is going all the time.  In
 * pulsed mode a number of pulses are sent, then there is a delay,
 * and then this repeats (see {@link #setPulseParams}).</p>
 *
 * @param c true to set to continuous mode, false to set to pulsed
 * mode.
 **/
public void setContinuous(boolean c){
  if(c){
    serial.writeByte(0x88);
  }else{
    serial.writeByte(0x98);
  }
}

/**
 * <p>Enables/disables the output of the data from the ultrasound
 * board.  When enabling this sets to continuous mode.  To enable
 * you can also just use {@link #setContinuous}.</p>
 *
 * <p>Note, this just sends the bytes to set this mode, it does not
 * verify that we are actually in this mode.</p>
 *
 * @param e true to enable output of data in continuous mode, false
 * to disable
 **/
public void setEnabled(boolean e){
  if(e){
    serial.writeByte(0x88);
  }else{
    serial.writeByte(0x80);
  }
}

/**
 * <p>Set the pulsed mode parameters.  There will be osc*2 pulses and
 * then a pause of pause*8 periods.  You must also do {@link
 * #setContinuous} to false to set to pulsed mode.</p>
 *
 * <p>Pulsed mode is useful for getting range information in
 * conjunction with the status byte.</p>
 *
 * @link param osc oscillation pulses(*2) valid range 1-255
 * @link param osc pause periods(*8) valid range 1-255
 **/
public void setPulseParams(int osc, int pause){
  if(osc < 1) osc = 1;
  if(osc > 255) osc = 255;
  if(pause < 1) pause = 1;
  if(pause > 255) pause = 255;
  //Write the osc
  serial.writeByte(0xC0);
  serial.writeByte(osc&0xff);
  //Write the pause
  serial.writeByte(0xD0);
  serial.writeByte(pause&0xff);
}

/**
 * <p>Checks to see if the serial port is connected and properly
 * initialized.</p>
 *
 * @return true if connected and initilized
 **/
public boolean isConnected(){
  return serial.isConnected();
}

/**
 * <p>Enables/disables printing of warnings to the console if there
 * is a cache overflow, etc.  By default warnings are enabled.</p>
 *
 * @param w true to enable warnings, false to disable.
 **/
public synchronized void setWarnings(boolean w){
  shouldWarn = w;
}

/**
 * <p>Gets the number of times the data stream has had to be
 * resynchronized.  If this is continually increasing then there is
 * a problem with the serial stream dropping bytes.</p>
 *
 * @return the number of syncs
 **/
public synchronized long getNumSyncs(){
  return syncCount;
}

/**
 * <p>Gets the number of overflows of our internal buffer.  If this
 * continually increases then the data is not being read quickly
 * enough or the interal buffer size is too small.</p>
 *
 * @return the number of overflows
 **/
public synchronized long getNumOverflows(){
  return overflowCount;
}

/**
 * <p>Clears our buffer of all data</p>
 **/
public synchronized void clear(){
  numCached = 0;
}

/**
 * <p>Constructor to create a connection to an ultrasound device on
 * the specified serial port</p>
 *
 * @param port to connect to
 **/
public UltraSound(String port){
```

```
    serial = new SerialBasic(port);
    collectThread.setPriority(Thread.MAX_PRIORITY);
    collectThread.start();
  }

}

/**
 *
 * <p>Copyright (C) 2007 Carrick Detweiler and
 *                       Massachusetts Institute of Technology</p>
 *
 * <p>This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.</p>
 *
 * <p>This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.</p>
 *
 * <p>You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc., 59
 * Temple Place - Suite 330, Boston, MA 02111-1307, USA.</p>
 *
 * <p>Some snips taken from the rxtx example on the rxtx wiki
 *
 * $Id: ultrasonicspeech.tex 1160 2008-05-14 19:41:28Z carrick $
 **/

import java.util.*;

import gnu.io.CommPort;
import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;

import java.io.FileDescriptor;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;

public class SerialBasic{

  /**
   * <p>SVN version info.</p>
   **/
  private static final String svnid =
    "$Id: ultrasonicspeech.tex 1160 2008-05-14 19:41:28Z carrick $";

  /**
   * <p>The default serial port.</p>
   **/
  public final static String DEFAULT_PORT = "/dev/tty.usbserial-FTC9MFOI";
  //public final static String DEFAULT_PORT = "/dev/tty.usbserial-FTCDC5GC";
  //public final static String DEFAULT_PORT = "/dev/tty.BlueRadios-COM0-4";

  /**
   * <p>The default baud rate.</p>
   **/
  public final static int DEFAULT_BAUD = 3000000;
  //public final static int DEFAULT_BAUD = 230400*4;

  /**
   * <p>Input.</p>
   **/
  private BufferedInputStream input;

  /**
   * <p>Output.</p>
   **/
  private BufferedOutputStream output;

  /**
   * <p>True if the serial port has been inited.</p>
   **/
  private boolean serialInitilized = false;

  /**
   * <p>Connects to the specified port.  On OS X (and others?) must have
   * rw access to /var/lock for port lock checking.</p>
   *
   * @param portName the port to connect to
   * @param baud to connect at
   **/
  void connect(String portName, int baud) throws Exception{
    System.out.println("Trying to connect to port " + portName + " with baud " + baud);
    CommPortIdentifier portIdentifier = CommPortIdentifier.getPortIdentifier(portName);

    if ( portIdentifier.isCurrentlyOwned() ){
      System.out.println("Error: Port is currently in use");
    }else{
      CommPort commPort = portIdentifier.open(this.getClass().getName(),2000);

      if ( commPort instanceof SerialPort ){
        SerialPort serialPort = (SerialPort) commPort;
        serialPort.setSerialPortParams(baud,SerialPort.DATABITS_8,SerialPort.STOPBITS_1,SerialPort.PARITY_NONE);

        input = new BufferedInputStream(serialPort.getInputStream());
        output = new BufferedOutputStream (serialPort.getOutputStream());

      }else{
        System.out.println("Error: Not a serial port");
      }
    }
  }

  /**
   * <p>Blocking read of the next byte.</p>
   *
   * @return -1 on error, else unsigned byte value read
   **/
  public int readByteBlock(){
    if(!serialInitilized){
      System.err.println("ERROR: Serial not initialized");
      return -1;
    }
    try{
      int val = input.read();
      //System.out.print("0x" + Integer.toHexString(0xff& val) + " ");
      return val;
    }catch(Exception e){
      e.printStackTrace();
      return -1;
    }
  }

  /**
   * <p>Reads the next available byte.  Returns -1 if there are none
   * available otherwise the value read between 0-255.</p>
   *
   * @return -1 if no byte available, else unsigned byte value read
   **/
  private int readByte(){
    if(!serialInitilized){
      System.err.println("ERROR: Serial not initialized");
      return -1;
    }
    try{
      if(input.available() <= 0){
        return -1;
      }
      int val = input.read();
      //System.out.print("0x" + Integer.toHexString(0xff& val)
      //+ "," + (char)val + " ");
      //System.out.print((char)val);
      return val;
    }catch(Exception e){
      e.printStackTrace();
      return -1;
    }
  }

  /**
   * <p>Gets the number of bytes which are available to read.
   * Negative on serial error.</p>
   **/
  public int available(){
    if(!serialInitilized) return 0;
    try{
      return input.available();
    }catch(Exception e){
      e.printStackTrace();
      return -1;
    }
  }

  /**
   * <p>Writes byte b.</p>
   *
   * @param b byte to write (value of int b & 0xff)
   **/
  public void writeByte(int b){
    if(!serialInitilized){
      System.err.println("ERROR: Serial not initialized");
      return;
    }
    try{
      output.write(b);
      output.flush();
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  /**
   * <p>Writes bytes b of length b.length.</p>
   *
   * @param b bytes to write (value of int b & 0xff)
   **/
  public void writeBytes(byte[] b){
    if(!serialInitilized){
      System.err.println("ERROR: Serial not initialized");
      return;
    }
    try{
      output.write(b);
      output.flush();
      /*
      System.out.println("Sending packet: ");
      for(int i=0;i<b.length;i++){
```

```
      System.out.print(Integer.toHexString(0xff& b[i]) + " ");
    }
    System.out.println("");
    */

  }catch(Exception e){
    e.printStackTrace();
  }
}

/**
 * Gets any bytes that we have and makes a string of it
 **/
public String toString(){
  String s = "";
  int val;
  while((val = readByte()) >= 0){
    byte b[] = new byte[1];
    b[0] = (byte)val;
    s += new String(b);
  }
  return s;
}

/**
 * <p>Checks to see if the serial port is connected and properly
 * initialized.</p>
 *
 * @return true if connected and initilized
 **/
public boolean isConnected(){
  return serialInitilized;
}

/**
 * <p>Default constructor of the serial port object.  Exits on fail</p>
 **/
public SerialBasic(){
  this(DEFAULT_PORT);
}

/**
 * <p>Constructor with the port set</p>
 **/
public SerialBasic(String port){
  try{
    connect(port,DEFAULT_BAUD);
    serialInitilized = true;
  }catch (Exception e){
    serialInitilized = false;
    e.printStackTrace();
    System.out.println("Error initializing serial port, port disabled");
  }

}

}
```

C Matlab Library

```
function us = ultrasoundInit(codepath, port)
% us = ultrasoundInit(codepath,port).  Codepath is the location of the java class files
% and port is the serial port to connect to. Returns the ultrasound java
% object.

javaaddpath(codepath)
javaaddpath(strcat(codepath,'/RXTXcomm.jar'))

us = UltraSound(port);
us.setWarnings(false);
us.setContinuous(true);
us.setGain(4,4);
us.setUltraSoundPower(1);

us.setEnabled(false);

function data = ultrasoundRead(us, seconds, audiogain, ultrasonicgain, ultrasonicpower)
% data = ultrasoundRead(us,seconds)
% data = ultrasoundRead(us, seconds, audiogain, ultrasonicgain, ultrasonicpower)
%
% Read from the ultrasound object us for
% the specified number of seconds.  Result is returned in data with the
% format [audio,ultrasound,status].  The audiogain (default 5),
% ultrasonic gain (default 4) and power (default 1)
% can also optionally be set.

if nargin < 5, ultrasonicpower = 1; end
if nargin < 4, ultrasonicgain = 4; end
if nargin < 3, audiogain = 5; end

%samples per second
sps = 22000;

numsamples = sps*seconds;

'Allocating array'

data = zeros(numsamples,3);


us.setEnabled(false);
us.clear();
us.setGain(audiogain,ultrasonicgain);
us.setUltraSoundPower(ultrasonicpower);


%enable warnings
us.setWarnings(true);

'Recording data started'
tic

starttime = cputime;
us.setEnabled(true);
while (cputime - starttime) < seconds
end
us.setEnabled(false);

'Recording data stopped'
toc

'Downloading data'
tic

%read the rest
for i = 1:numsamples
    if(us.available() > 0)
        data(i,:) = us.read();
    end
end


'Done downloading data'
toc

%disable warnings
us.setWarnings(false);
```

D Java API Docs

Class UltraSound

java.lang.Object
  └ **UltraSound**

public class **UltraSound**
extends java.lang.Object

Field Summary

private int[]	**audio** Cache for the audio channel
static int	**CACHE_SIZE** Number of samples to cache internally.
(package private) java.lang.Thread	**collectThread** Collect data thread.
private int	**numCached** Number of elements we currently have cached
private long	**overflowCount** The number of times our buffer has overflowed.
(package private) SerialBasic	**serial** Our serial port
private boolean	**shouldWarn** True to enable warning outputs on buffer overflows, etc.
private int[]	**status** Cache for the status byte.
private static java.lang.String	**svnid** SVN version info.
private long	**syncCount** The number of times we have resynchronized.
private int[]	**ultrasound** Cache for the ultrasound channel.

private int	**writeIndex** Index into both audio and ultrasound which indicates the next place we will write data.

Constructor Summary

UltraSound(java.lang.String port) Constructor to create a connection to an ultrasound device on the specified serial port

Method Summary

int	**available**() Returns the number of samples available.
void	**clear**() Clears our buffer of all data
private void	**getNextReadings**() Blocks waiting for the next pair of readings from the serial port and then adds them to our local cache.
long	**getNumOverflows**() Gets the number of overflows of our internal buffer.
long	**getNumSyncs**() Gets the number of times the data stream has had to be resynchronized.
boolean	**isConnected**() Checks to see if the serial port is connected and properly initialized.
int[]	**read**() Blocking read of the values of the audio and ultrasound.
int[]	**read**(int[] vals) Blocking read of the values of the audio and ultrasound.
void	**setContinuous**(boolean c) Set it to continuous mode if true, else to pulsed mode.
void	**setEnabled**(boolean e) Enables/disables the output of the data from the ultrasound board.
void	**setGain**(int audio, int ultrasound) Set the gain of the audio and ultrasound.
void	**setPulseParams**(int osc, int pause) Set the pulsed mode parameters.

void	**setUltraSoundPower**(int level) Sets the ultrasound output power level.
void	**setWarnings**(boolean w) Enables/disables printing of warnings to the console if there is a cache overflow, etc.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

svnid

```
private static final java.lang.String svnid
```

SVN version info.

See Also:
Constant Field Values

---

serial

```
SerialBasic serial
```

Our serial port

CACHE_SIZE

```
public static final int CACHE_SIZE
```

Number of samples to cache internally. If we overflow our cache we just discard the oldest data.

See Also:
Constant Field Values

---

audio

```
private int[] audio
```

Cache for the audio channel

---

status

`private int[] status`

Cache for the status byte.

---

ultrasound

`private int[] ultrasound`

Cache for the ultrasound channel.

---

numCached

`private volatile int numCached`

Number of elements we currently have cached

---

writeIndex

`private volatile int writeIndex`

Index into both audio and ultrasound which indicates the next place we will write data.

---

shouldWarn

`private boolean shouldWarn`

True to enable warning outputs on buffer overflows, etc.

---

overflowCount

`private long overflowCount`

The number of times our buffer has overflowed.

---

syncCount

```
private long syncCount
```

The number of times we have resynchronized.

---

collectThread

```
java.lang.Thread collectThread
```

Collect data thread. This continuously reads the serial port looking for data. On startup this sleeps for a couple of seconds before reading data to let everything else startup so we don't overflow our internal buffer.

Constructor Detail

UltraSound

```
public UltraSound(java.lang.String port)
```

Constructor to create a connection to an ultrasound device on the specified serial port

Parameters:

`port` - to connect to

Method Detail

getNextReadings

```
private void getNextReadings()
```

Blocks waiting for the next pair of readings from the serial port and then adds them to our local cache.

---

available

```
public int available()
```

Returns the number of samples available.

Returns:

the number of samples in our cache

---

read

```
public int[] read()
```

Blocking read of the values of the audio and ultrasound. The audio and ultrasound readings are 14 bit values. The status byte is also returned which will be 1 if it is pulsing and 0 if it is not. The status bit allow synchronization with the sending of the pulse and the receipt of a pulse in pulsed mode.

The other version of read (`read(int[] vals)`) is slightly more efficent as new arrays don't have to be allocated.

Returns:

array containing the [audio,ultrasound,status] reading

---

read

```
public int[] read(int[] vals)
```

Blocking read of the values of the audio and ultrasound. The audio and ultrasound readings are 14 bit values. The status byte is also returned which will be 1 if it is pulsing and 0 if it is not. The status bit allow synchronization with the sending of the pulse and the receipt of a pulse in pulsed mode.

The array passed as an argument is used to store the results, and is returned.

Parameters:

`vals` - the array to put the data into, must be of at least size 3

Returns:

array containing the [audio,ultrasound,status] reading

---

setGain

```
public void setGain(int audio,
                    int ultrasound)
```

Set the gain of the audio and ultrasound. These must be in the range 0-7, if they are not they are clipped.

---

setUltraSoundPower

```
public void setUltraSoundPower(int level)
```

Sets the ultrasound output power level. Must be between 0-50, otherwise clipped.

---

setContinuous

```
public void setContinuous(boolean c)
```

Set it to continuous mode if true, else to pulsed mode. Pulsed mode is setup via `setPulseParams(int, int)`. This also enables streaming of data over the serial port. Note, this just sends the bytes to set this mode, it does not verify that we are actually in this mode.

In continuous mode the ultrasound is going all the time. In pulsed mode a number of pulses are sent, then there is a delay, and then this repeats (see `setPulseParams(int, int)`).

Parameters:

`c` - true to set to continuous mode, false to set to pulsed mode.

---

setEnabled

```
public void setEnabled(boolean e)
```

Enables/disables the output of the data from the ultrasound board. When enabling this sets to continuous mode. To enable you can also just use `setContinuous(boolean)`.

Note, this just sends the bytes to set this mode, it does not verify that we are actually in this mode.

Parameters:

`e` - true to enable output of data in continuous mode, false to disable

---

setPulseParams

```
public void setPulseParams(int osc,
                           int pause)
```

Set the pulsed mode parameters. There will be osc*2 pulses and then a pause of pause*8 periods. You must also do setContinuous(boolean) to false to set to pulsed mode.

Pulsed mode is useful for getting range information in conjunction with the status byte.

---

isConnected

```
public boolean isConnected()
```

Checks to see if the serial port is connected and properly initialized.

Returns:
true if connected and initilized

---

setWarnings

```
public void setWarnings(boolean w)
```

Enables/disables printing of warnings to the console if there is a cache overflow, etc. By default warnings are enabled.

Parameters:
`w` - true to enable warnings, false to disable.

---

getNumSyncs

```
public long getNumSyncs()
```

Gets the number of times the data stream has had to be resynchronized. If this is continually increasing then there is a problem with the serial stream dropping bytes.

Returns:
the number of syncs

---

getNumOverflows

```
public long getNumOverflows()
```

Gets the number of overflows of our internal buffer. If this continually

increases then the data is not being read quickly enough or the interal buffer size is too small.

Returns:

the number of overflows

---

clear

```
public void clear()
```

Clears our buffer of all data
