Wearable Multiple Sensor Acquisition Device

by

Christopher Lee Elledge

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by .....
Alex "Sandy" Pentland
Toshiba Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Wearable Multiple Sensor Acquisition Device

by

Christopher Lee Elledge

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The wearable multiple sensor aquisition device is targeted as the primary sensor hub for the next revision of the MIThril wearable computer architecture. The device provides 3-axis acceleration sensing, infrared transmission and reception of Tag IDs, and a 16kHz 8bit audio stream. To be usable for wearable applications, the WMSAD is contained in a 1.3"X1.35" package and operates at 35mA at 5V, and meets the requirements imposed by these applications. The WMSAD will operate directly with the SAK2 to provide sensor data collection and recording.

Thesis Supervisor: Alex "Sandy" Pentland
Title: Toshiba Professor of Media Arts and Sciences

Acknowledgments

Richard DeVaul provided previous work and guidance in wearable sensor design. Josh Weaver provided previous work in accelerometer and infrared sensor design. Ryan Aylward provided the microphone amplification circuit used in the audio capture subsystem. The hardware and software infrastructure at MIT Media Lab's BorgLab was used in the design, construction, and programming of the WMSAD.

Contents

Chapter 1

Introduction

Wearable Computing is a field of research that integrates the computing experience into a persons' everyday life. A wearable computer should be able to aid the user in their daily life by being able to interact with the user based on their context. In order for the computer to determine the user's context, a variety of sensors can gather information to be analyzed by the system.

The Wearable Multiple Sensor Acquisition Device (WMSAD) is designed to act as a compact sensor hub that bridges the gap between sensor output and the digital interfaces on the next generation MIThril wearable computing architecture. It centralizes several of the most commonly used sensors in wearable computing into a single device which can be placed in a convenient location on the body like a name tag on worn on the chest. These common sensors are accelerometers, infra-red transceivers, and microphones.

The remainder of this chapter discusses applications in which the data from the WMSAD can be utilized and how it relates to other work in the field.

The design requirements for the WMSAD are covered in Chapter 2. Chapter 3 states the final specifications for each of the WMSAD subsystems. Chapter 4 discusses the hardware design details and the engineering decisions and tradeoffs that were made. Chapter 5 covers the software design details. Chapter 6 details the testing procedures for the WMSAD. Finally, Chapter 7 discusses the contributions that this project makes for Wearable Computing.

1.1 Applications for Wearable Sensors

There are several applications that have used accelerometers, infrared transceivers, and microphones in wearable computing. The WMSAD will work with the future wearable systems to support continued research in this field.

Wearable systems can perform real-time activity classification using 3-axis accelerometer information. By obtaining the power spectrum of these signals and running it through a trained mixture model, the wearer's current activity can be determined from the trained states[3].

The acceleration data has also been used to in experiments with classifying the medication state of Parkinson patients. A patient's tremors can be detected and the times recorded so that the patient's doctor can have information useful for determining the patient's medication needs[6].

When groups of people wear sensing devices, the social network of the group can be analyzed. The flow of conversations can determine how people interact with one another in the group[2]. For these applications, audio and IR tags determine who is speaking with whom within the group. There has also been research into keyword recognition, so that the flow of information among the members can be analyzed[4].

1.2 Related Work

The Hoarder is a wearable device that collects data from on body sensors[5]. It can either record this data to an on-board compact flash memory or act as an interface between the worn sensors and a worn central computational resource such as a Zaurus PDA. A variety of sensors could be used with the Hoarder by complying with the MIThril architecture sensor design and the use of I2C as the communication protocol. The Hoarder is also commonly referred to as the SAK[1] Board.

The SAK2 is the second version of the Hoarder. It is currently in development and will be designed to work more closely with the Zaurus as a wearable platform and have a smaller package than the Hoarder. The primary communication protocol

[1]Swiss Army Knife

will still be I2C. The WMSAD combines the functionality of many of the sensors currently used with the Hoarder into a single device for use with the SAK2.

One of the primary sensors currently used is a 3-axis accelerometer[6]. This device uses the Analog Devices' ADXL202E to measure acceleration and a Microchip PIC micro-controller to digitize the values and make them available over I2C. The WMSAD uses the previous generation of 3-axis accelerometers as a reference platform for the acceleration sensing.

Another design currently in use is the infrared tag reader[1]. This sensor receives infrared broadcasts of IDs using the Sony-IR protocol. This reader is complemented by the Squirt[7] which is a small and low power board that broadcasts the IDs which the reader looks for. The IDs that these Squirts broadcast can represent locations inside a building or people who are wearing them as name tags. The WMSAD incorporates both of these designs to provide 2-way IR communication of IDs.

Chapter 2

Design Requirements

The Wearable Multiple Sensor Acquisition device has the following design requirements. It is required to provide three different types of information to the wearable computer regarding the user's current state: 3-axis acceleration, infrared tag identities, and a single 16kHz audio channel. The data that it outputs from its sensors is compatible with the previous generation of independent sensors. This minimizes the effort required in integrating the new hardware into our wearable computing systems.

Additionally the WMSAD is required to emit infrared tag identities for other tag reading devices to capture. This allows it to act as a name tag for other systems to identify the user and potentially allows for information transfers.

Further requirements are imposed to make the WMSAD a wearable device. First, it must be small and lightweight — less than 2 inches on a side and of minimal thickness. Additionally, to ensure low heat output and reduce battery consumption, the typical power use of the WMSAD is kept below 70mA.

To ensure operation with current systems and the future SAK2 data collection device, the WMSAD operates over the I2C protocol. The data from each sensor is made available at certain device IDs in accordance with the protocols used in the previous generation of sensing devices.

Chapter 3

Design Specification

The Wearable Multiple Sensor Acquisition Device is a small and low powered multi-sensor data source for the second revision of the Hoarder data collection device, the SAK2. This chapter details the specifications for the device that enable it to meet the requirements set out in Chapter 2.

The WMSAD must be capable of being worn on the body and handle passing requested sensor readings to the controlling device. The communications between this sensor and the controlling device is handled by an I2C two wire bus. It is assumed that a 5 Volt power rail and corresponding ground are provided. The power rail is also assumed to be capable of sourcing the required current to power the device, and the noise level of the power rail is assumed to be within $\pm 10\%$ tolerances.

The board layout should be between 1" and 2" square. In order to fit all the needed components into this area, a two board stack is used such that one board has a connector that allows it to be snapped into the other in a sandwich. The thickness of the WMSAD is defined by the infrared LEDs mounted to the bottom board, and the second board does not exceed this space.

3.1 Accelerometer

The three-axis acceleration data is captured using two separate two-axis accelerometers. These are the Analog Devices' ADXL202E accelerometers currently used in

various other wearable computing projects. They will accurately report accelerations of between ±2G on each axis. They can be read by timing the pulse width modulated output using a micro-controller's digital input pins, or by converting the analog output using an ADC equipped micro-controller. One of the accelerometers will be mounted on the main board; the other on a small companion board mounted perpendicular to the main board. This allows for the capture of all three axis of acceleration with one redundant axis.

3.2 IR Transceiver

The infrared transceiver is compatible with the Squirt's infrared tag protocol[7]. In order to accomplish this, the sensor uses several infrared LEDs as transmitters. Four LEDs, one in each corner of the board, provide a moderate spread of the transmission over an area. The tag supports broadcasting multiple one to four different IDs, and these IDs are programmable over I2C. These IDs are also retained when the device is power cycled. The frequency of ID broadcasts are also configurable over the I2C interface.

The transceiver also has support for one or two infrared receivers to capture transmissions from other WMSADs or Squirts. One of these receivers faces forward to catch tag information from devices worn on other people. This allows for the identification of people with whom the user may be conversing. A second receiver can be oriented facing upwards to allow the collection of tag IDs from devices on the ceiling. This configuration enables location tracking inside of instrumented areas. The most recently observed tag IDs can be obtained by querying the device over I2C.

3.3 Audio Capture

Audio is the highest bandwidth signal on the device. A single channel of audio sampled at 16kHz is required. The signal is quantized between 8 and 16 bits per sample. Most of the micro-controllers with embedded ADCs available for use in this

device have a 10 bit quantization limit for their analog inputs which is sufficient for this task. In order to support the necessary data rate for transmitting this audio, an I2C bus that runs faster than 160 kbps is used. Thus, it is necessary that both the micro-controllers on the WMSAD and on the SAK2 be capable of handling the faster I2C clock rates.

The audio input also requires a microphone on the board and that there be a clean power rail at the microphone's operating voltage. The audio signal from the microphone also needs to be amplified for the ADC. This is done with a set of opamps and a voltage reference to act as the zero crossing for the audio signal.

Chapter 4

Hardware Design

This chapter discusses the hardware design details for the WMSAD.

The final design for the device is based on three different boards. The main board is the largest at 2.3" by 2.35". It contains two micro-controllers and most of the components needed for the accelerometers and infrared subsystems. This board could function on its own as a two-axis accelerometer, and a infra-red transceiver with one upward facing receiver.

This board has five connection options. The primary connection is a four pad surface mount point with strain-relief mounting holes to allow the connection of a cable carrying power, ground, and I2C by direct soldering. While a locking connector could have been used to allow the arbitrary connection and disconnection of cabling, the available connectors for this are either large or are too flimsy for wearable applications. The second connection is similar to the first, but it is used to provide external infrared receiver options as discussed in Section 4.3.

The third connection is to the companion board that provides an additional infrared receiver and two-axis accelerometer. This connection is a .1" spaced 5 pin right angle connection to allow the companion board to be placed perpendicular to the main board.

The fourth and fifth connections are both Hirose 31 pin board to board connectors. The one on the top side of the board allows the connection of the audio daughter board (discussed in Section 4.4) to the main board. This connector is a standard

on several other devices used in the Wearable Computing Group, so each of the pins has a specific purpose. On this device, the connector has power, ground, and I2C connections. It also has carries a few digital lines and programming voltages to allow the micro-controllers to be programmed through the connector.

The Hirose connection on the back allows the addition of the future wireless daughter-board currently in development. That device will provide I2C protocol communication over a near field wireless interface to other components worn on the body. It will also have a voltage regulator and batteries, since wireless devices need their own power supplies, and will enable the WMSAD and the SAK2 to go wireless. This connector carries power, ground, and I2C.

4.1 Micro-controllers

The micro-controllers used in the WMSAD are from the Microchip PIC series. The PIC was chosen because there is support for it in the lab. Another option, the Atmel AVRs, would have required a new investment in both new equipment and support. Either chip set would have performed suitably in this task.

The three micro-controllers used have I2C slave support in hardware, and at least one of the micro-controllers used has an analog to digital converter. All the PICs taken into consideration with native I2C support also provide some number of analog pins muxed to an ADC, so the decision was simplified.

The number of micro-controllers to use on the device was also an important decision. The more devices that there are for individual tasks, the easier it is handle them concurrently. Since these sensors can be run independently, multiple micro-controllers can handle the work in parallel. This is also a simpler solution for programming since there is less competition for on chip resources.

The downside to choosing to use more micro-controllers is that more space n the physical layout is required, more power is consumed, and more noise is generated on the power rails and the signal lines. The final decision for this design was to have three micro-controllers, one for each individual subsystem discussed below.

The specific model of micro-controller used for the WMSAD is the PIC 16F876A. It has a three timers, a capture compare unit, hardware I2C support, a 10 bit ADC, and runs at 20MHz. This chip is available in a 28-pin TSSOP package which takes up very little layout space.

Each PIC also has its own oscillator to drive its clocks at 20MHz. The device chosen for this is the Panasonic EFJ-C2005E5B. It is a very small device which includes the required capacitors, so there are no additional external parts required to generate the clocks for the micro-controllers.

4.2 Accelerometer

The ADXL202E accelerometer reports the instantaneous acceleration using either a pulse width modulated signal or an analog output. Previous wearable accelerometer designs used the pulse width modulated outputs of this chip connected directly to digital input lines on the micro-controller. Since each ADXL202E has two-axis of information and we use two of them, we need to connect four digital inputs.

These past designs worked in most cases, but there were occasional glitches in their values. The PWM outputs are measured using delay loops that wait for each individual pin to change in a specific sequence. The problem occurs when that timing loop is interrupted by some outside event such as an I2C request for data. At higher sampling rates, this become even more of a problem.

One solution to this problem is to use the analog outputs of the ADXL202E connected to available analog inputs on the micro-controller. Unfortunately, after some design attempts, we discovered that the analog outputs are too high impedance to drive the ADC on the PICs with enough current to sample at a reasonable rate. The solution to this problem is analog buffering, but that would require the addition of a quad opamp package.

A second solution is to use the capture compare (CCP) units on the PIC to handle the timing instead of trying to do it in software. The CCPs capture timer values when edge transitions occur in a digital line. This does the same thing as the older designs,

but doesn't require tight timing loops that can be interrupted. Unfortunately the PIC has only two CCPs, and we need four to handle the four-axis of acceleration data. This was solved by using a digital MUX between the PWM outputs of the ADXL202E's and the CCP inputs.

For this device, I chose to use the MUX with the CCP units on the 16F876A. I chose this approach because the digital MUX requires fewer traces to be placed and much less power than the quad opamp would require. The PWM also has been shown to produce good results while the analog outputs still have yet to be tested in a design.

The PWM output of the ADXL202E's is connected to both the MUX and to four general purpose digit input lines on the PIC. These additional connections are to provide a backup solution in case the MUX or the CCP modules fail to operate properly. A Texas Instruments SN74CBT3257PWR was used as the MUX because it uses simple MOSFET switching for the logic. This makes it both small and power efficient. It can also be disabled when not in use for additional power savings. Only half of the MUX's switches are used, but smaller designs were not readily available.

Additional resistive and capacitive components were added to meet the reference specifications for the ADXL202E. Each of these devices has a filtered power rail from the primary 5 volt supply on the acquisition device.

4.3 IR Transceiver

The infrared transceiver is composed of three major components. The micro-controller controls most of the functionality of this subsystem. The infrared LEDs are used to broadcast the protocol's signal to other devices, and the receivers capture the broadcasts and relay the presence or lack of an infrared signal to the micro-controller.

The IR receivers used are the Sharp IS1U621. These receivers detect the presence of an infrared 38kHz square wave. Other devices considered were from Panasonic's PNA461*M line. While they come in many more frequency ranges and are pin compatible with the Sharp part, they all have worse footprints and large metal casings

which cannot be removed. This would be fine for most applications, but there is very little space to place the casings on the WMSAD. The infrared receivers also require a large amount of power rail filtering because they are noisy devices. Their outputs are connected to digital input pins on the PIC, which are also available to the capture compare units.

The IR LEDs used are the Light-On LTE-5208A. There are a variety of options that could work equally well. These were readily available, cheap, and in the optimal wavelength to work with the Sharp receiver. Their ideal brightness (without overloading) requires about 50mA of current each. Thus the digital output lines on the PIC cannot source or sink enough current to bring these LEDs to full brightness. Instead, the switching of these LEDs is provided by n-channel MOSFETS. The LED's anode are connected to the 5 volt supply through a current limiting resistor. The cathode is connected to the MOSFET which regulates the connection to ground. The MOSFETS used are the Zilog ZXMN10A07FTA. MOSFETS were used instead of BJTs because there was no need for additional current limiting resistors.

A surface mount connection is available to allow the IR receivers to be placed off of the board's surface. This allows the receivers to be placed in locations which will not be blinded by the output of the acquisition board's own IR LEDs when they are active. This connection is similar to the connection provided for the I2C and power connections, a strain-relieved cable mount to conserve space. The offboard receivers will need to have their own power filtering since the filters must be as close to the receivers as possible.

4.4 Audio Capture

The audio capture portion of the WMSAD is composed of a PIC 16F876A, a quad package opamp, and a cartridge microphone. The microphone goes through several amplification and filtering stages, and the buffered output is connected to the one of the ADC inputs on the micro-controller. These components, the related resistors and capacitors for the amplification stages, and power filtering components are all placed

on a separate board which clips into the Hirose 31 pin connector. This leaves about 4mm between the main board and the audio board. These boards were designed with the ground planes facing opposite directions so that any components on the inside that might accidently touch the other board would not cause a short.

The schematic used for the microphone amplifier and filter is a known working design by Ryan Aylward using the WM-52BM microphone and TLV2464IPW opamps. The first opamp stage buffers a 2.5 volt reference to provide power to the microphone and to act as a center crossing for amplification to the micro-controllers 5 volt scale.

The remaining opamps are all used to handle the microphone's output. Each stage is DC decoupled by a high-pass filter. The second opamp stage is a simple amplification stage for the signal. The third stage is a bandpass filter. This stage also has a potentiometer to allow some adjustment in the amplification. The final stage is a voltage follower to buffer the final DC decoupled output of the microphone circuit.

Chapter 5

Software Design

Even if the hardware all works, it still requires the micro-controller's to have software designed to accomplish the required tasks. Each subsystem has its own micro-controller, and each needs unique software to make their subsystems work.

All the software for the Microchip PIC micro-controllers in this multiple sensor acquisition device is written in Custom Compiler Service's PIC-C Compiler. The 16F876A micro-controllers are programmed with the compiled software using a PIC programmer through the Hirose 31 pin connector mentioned in Chapter 4.

5.1 Accelerometer

The software for the accelerometer sets up the acceleration subsystem and then runs in one of two modes. In the first mode, the micro-controller runs continuously and updates its values as frequently as possible. The second mode puts the micro-controller into low power mode until it receives a command to capture a new value. The advantages of the second mode are decreased power usage and the ability to latch the values of several accelerometers on the system at the same time. That allows accurate sampling of multiple accelerometers without sampling drift between components.

The I2C protocol works as specified for older accelerometer designs[6]. Command 2 resets the read value pointer to the first axis and command 8 causes the device to capture the acceleration values. Reading from the device will read the four axis of

acceleration data in order, and then wrap back around to the first.

In order to capture the acceleration data, the micro-controller enables the MUX and selects the first accelerometer's X and Y outputs. It sets up the capture compare unit (CCP) to capture the first rising edge for each line and resets the timer. An interrupt is generated when the PWM output of the accelerometer changes. The interrupt stores the value of the timer, and then sets the CCP to wait for the falling edge. The interrupt captures that value and then repeats to capture the following rising edge. This set of three data points for each axis provides the needed values for computing the duty cycles. These values are capture for both the X and Y axis at the same time.

Once these values have been captured, the micro-controller switches the MUX to select the second accelerometer's X and Y outputs and repeats the process above. Once all the captures are done, the MUX is disabled and the duty cycles are converted into signed 8bit integers to represent the acceleration value range of $\pm$2G.

5.2 IR Transceiver

The infrared subsystem requires that the micro-controller be able to handle two individual tasks at the same time. It controls the broadcasts of the infrared LEDs directly. It also reads the output of the two Sharp infrared receivers. This means that the WMSAD should handle timed events in a way that all these can occur with regular intervals and without affecting the results of the other tasks.

Unfortunately, during the time of the construction of this design and this paper, the Squirt infrared protocol was being redesigned. As such, I was unable to create a final version of the software for this device. Instead the software is designed to act as a framework for general infrared protocol support. In order to modify the software to support a different protocol, just two procedures need to be changed.

The first procedure is a method that converts a four byte ID into the internal representation for the infrared subsystem. This method is called whenever the device needs to change the ID that it is broadcasting. This method adds the required start

bits, error correction, etc. that the protocol requires to the internal representation.

The second procedure is a method that converts the internal representation to a four byte ID. This is called by the receiver code whenever an infrared pulse has ended. It should quickly determine if the internal representation does not have a complete protocol packet. If incomplete, it will be called again by the receiver when the next pulse ends. If complete, this method should parse the internal format and update the latest ID received variables. This method can also return an error to cause the receiver to flush the internal representation in case the captured data is bad or did not start at the beginning of a packet.

The internal representation is an odd length array that contains the number of event cycles that an infrared pulse should remain active, followed by the the number of event cycles that the system should wait before the next pulse. The representation should always start with a pulse and end with a pulse thus creating an odd length array.

The software design for this subsystem relies on a master event timer which determines the rate of polling for infrared reception and transmission events. Each timer event, the receivers are checked for changes in state and the transmitter's event counter is advanced.

When a change in the receiver's state occurs, it records the number of event cycles for that state change to occur into the internal representation for that receiver. Upon each end of an infrared pulse, the receiver calls the representation to ID method for the protocol to determine if the end has been reached, or if there has been an error in reception. On a completed receive or on an error, the state and internal representation for the receiver are cleared.

Whenever the transmitter's event counter reaches the value listed in the current entry in the internal representation it alternates the broadcast/silence state of the currently active infrared LEDs. Once the end is reached, the transmitter waits for a specified number of event cycles before beginning a new broadcast.

The infrared pulse is actually a 38kHz square wave during the time of the active broadcast. In order to accomplish this a timer on the PIC generates an interrupt

at 76kHz. This timer alternates the state of the LEDs to create the square wave. Unfortunately this rate is quicker than the default interrupt handler that the CCS PIC-C compiler uses can run. In order to get around this limitation, a more specific hand coded assembly interrupt handler was written to handle the timer interrupt and the I2C interrupt.

5.3 Audio Capture

The audio capture software has two tasks to perform. The analog input from the microphone is sampled at a constant 16kHz and placed into a buffer, and the buffer is read over the I2C bus. In order to do this, one of the PIC's timers was setup to produce a 16kHz interrupt. That interrupt starts the analog-to-digital converter. Once the conversion is done, another interrupt is triggered to capture the value and store it into the buffer.

Communicating over I2C poses a problem with this setup. Normally the I2C communication for a slave device is handled as an interrupt. Unfortunately, the interrupt can't be interrupted by other interrupts. This means that when a device is reading buffer values from the audio capture software, the sampling rate is negatively affected.

In order to solve this, the I2C communication is moved into the main execution loop to allow the analog sampling to continue at the desired rate. The down side is that periodic failures in the I2C communication can arise. These errors appear as single sample errors and can be detected by the master device since the byte will not be acknowledged.

The software uses seven 40 byte buffers for the audio. These buffers must be segmented to fit into the micro-controller's RAM pages. The segmentation also allows for simplified book keeping. The requesting device can query how many buffers are ready and then read the corresponding number of bytes from the buffers. Overflow is also handled at a buffer level. Whenever all the buffers have become filled, the earliest buffer is purged and becomes the target buffer for new samples.

Chapter 6

Testing the WMSAD

The Wearable Multiple Sensor Acquisition Device underwent a series of tests to insure that the hardware was fully functional and the software performed according to specifications. These tests were intended to verify that the device will function as a wearable sensing device for the desired applications.

The BrightStar is a PowerPC based device for embedded design that has a Linux device for 100kHz I2C communication. It was used for testing the I2C functionality of the WMSAD and to analyze the values returned by the sensors.

6.1 Simple Hardware Tests

Each subsystem was assembled on separate boards to allow for individual testing and to aid in debugging. Then each subsystem was programmed with software that performed simple routines to insure that each component of the subsystem operated correctly.

The outputs of the accelerometers, infrared receivers, and microphone were verified to match specifications while operating in the appropriate subsystem. The infrared LEDs and their respective switching components were also tested to make sure that they could handle the necessary current and switching speeds for 38kHz broadcast. The I2C communication of each micro-controller was tested by connecting the device's I2C lines to a BrightStar and performing simple data exchanges.

6.2 Accelerometer Tests

To test the accelerometer, a board with a completed accelerometer subsystem was connected to a BrightStar, which was running software to capture data from previous accelerometer sensor board designs. WMSAD was then placed in various orientations to use gravity as a stable acceleration reference.

The outputs of the accelerometer were also viewed as a set of signals while being run through various motions. The results from the WMSAD were virtually identical to the results of the previous accelerometer designs.

6.3 IR Transceiver Tests

In order to test the infrared transceiver, two separate devices were used and placed facing each other several feet apart. Since the Squirt protocol was in a state of revision at the time, a variation of the old Squirt protocol was used. It did not perform error correction. The receiving device was connected to a BrightStar to read the received values over I2C.

The values received over I2C were compared to the original transmitted values from the transmitter. During the tests, these values matched consistently with errors resulting in dropped values instead of corrupted ones.

6.4 Audio Capture Tests

The audio capture board was connected to a BrightStar to capture the sampled audio data from the microphone. The BrightStar converted the I2C values read from the micro-controller's buffers into a signal which was then viewed using a real-time signal viewer.

The results of this test were that the audio capture device correctly captures audio from the microphone within the desired range. Unfortunately, the specified audio filters also amplify some undesired low frequency noise. This can be corrected

by replacing the passive components in the earlier stages of the amplifier to raise the cut-off frequency of the high-pass filters.

The device also occasionally fails to respond to an I2C request. This occurs approximately once every minute. The sample rate also skips a sample every few seconds. Both of these errors are caused by delays in the interrupt routines. This can be solved by moving to a specialized assembly interrupt dispatcher, as was used in Section 5.2 for the infrared software.

6.5 Total Package Performance

When all the subsystems are brought together onto the same device, the behavior in each subsystem remains within specifications. The only performance degradations are found in the infrared receivers and the primary power rail for the audio capture unit.

Whenever the infrared transmitters broadcast, they also broadcast straight into the infrared receivers on the same device. This means that one of the Tag IDs that these devices will see is always its own IDs. The device can be programmed to ignore its own IDs, or the user can mount the infrared receivers external to the device.

The power rail remains very stable except during the infrared 38kHz broadcasts. These broadcasts cause some noise at that frequency on the primary 5V power rail. This only poses a problem to the audio capture subsystem because it uses the rail for comparison in the analog to digital converter. On the audio capture board, the noise appears as a 100mV peak to peak 38kHz waveform. This does not significantly affect the results of the analog to digital conversion. Since the voltage driving the microphone is from a 2.5V reference, the microphone is also not noticeably affected by the power rail noise.

On average, each subsystem (excluding infrared broadcasts), consumes 10mA at 5V. Thus the typical continuous power consumption of the entire WMSAD is roughly 30 to 35mA. The infrared transmitters consume 50mA of current when they are active. Since they are pulse width modulated with a 50% duty cycle, their average

consumption is 30mA during transmission. With each transmitter making round-robin broadcasts, the peak power consumption peaks at 65mA.

Chapter 7

Contributions

The WMSAD is a device which consolidates the most commonly used sensors in Wearable Computing into a single package. These common sensor are the accelerometer, the infrared transceiver, and the audio capture device. This small package allows for convenient placement of these sensors on the body wherever they may be of use. It may also be placed on the body in a location independent of the primary computation and storage devices. It can be connected to these other devices by a very small four stranded cable or by wireless support to be implemented in the future wireless I2C daughter-board.

In my thesis, I contributed a completely functional hardware solution for a Wearable Multiple Sensor Acquisition Device to the research field of Wearable Computing. I also provided a software framework for the various sensor subsystems on the device to provide the necessary sensing and communication functionality of each subsystem. These frameworks can be built upon to create more application specific functionality.

The WMSAD has already been used in an additional sensing role: to capture a person's heart rate using a polar heart monitor. The infrared receiver subsystem was modified to read the data pulse of the polar heart monitor's output. This timing was captured and made available for acquisition to a Hoarder through I2C. The entire device was then waterproofed and used in a hypothermia study.

Appendix A

Schematics

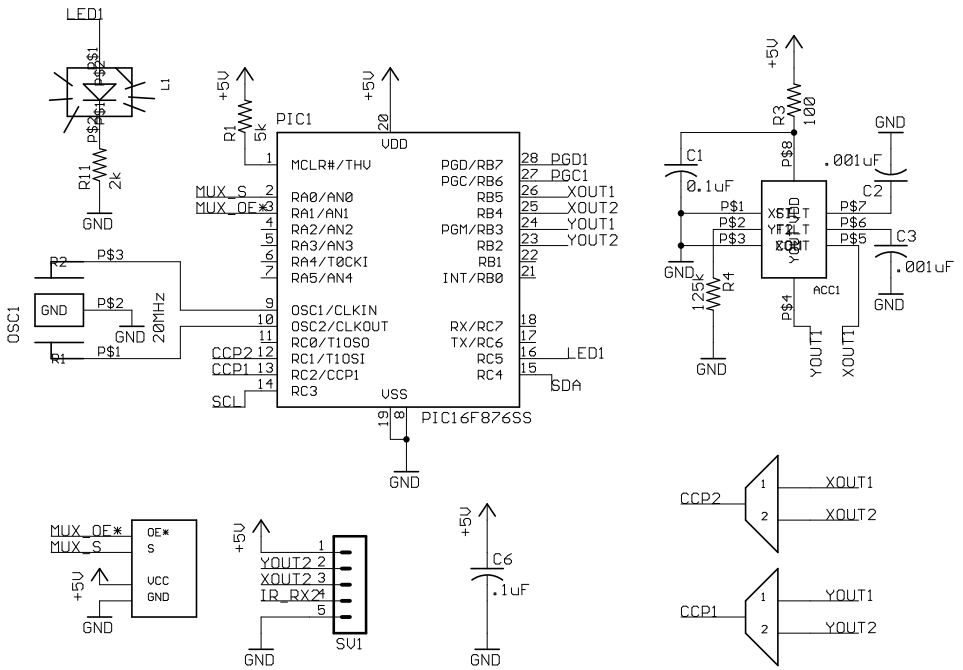

Figure A-1: Schematic for Main Board — Sheet 1.

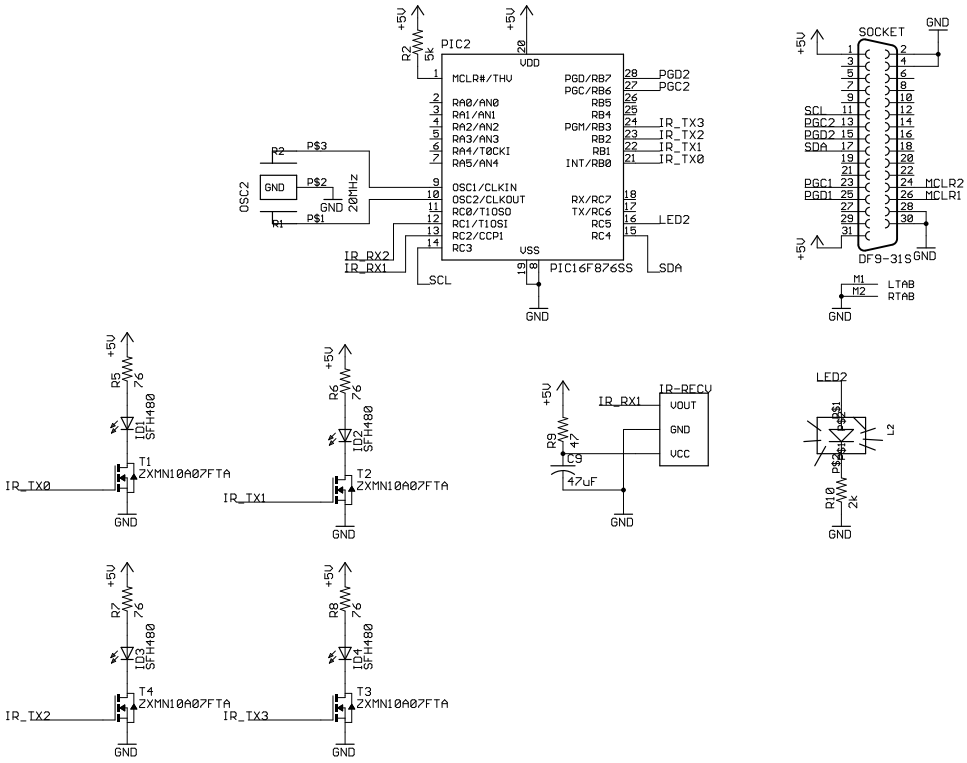

Figure A-2: Schematic for Main Board — Sheet 2.

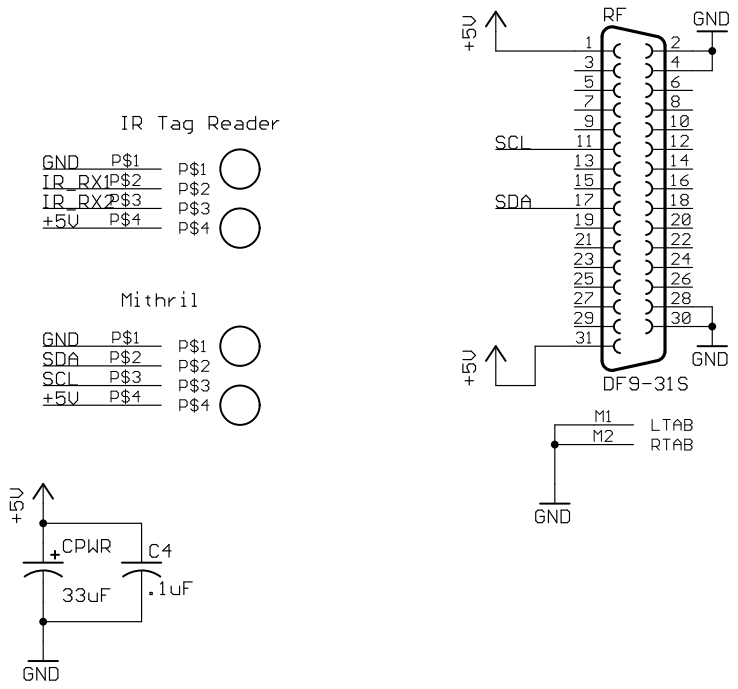

Figure A-3: Schematic for Main Board — Sheet 3.

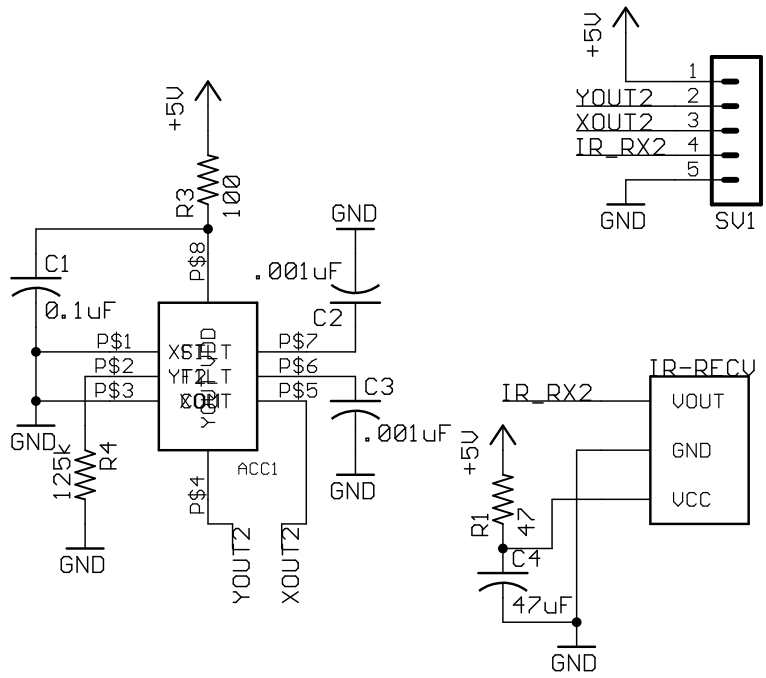

Figure A-4: Schematic for Companion Board.

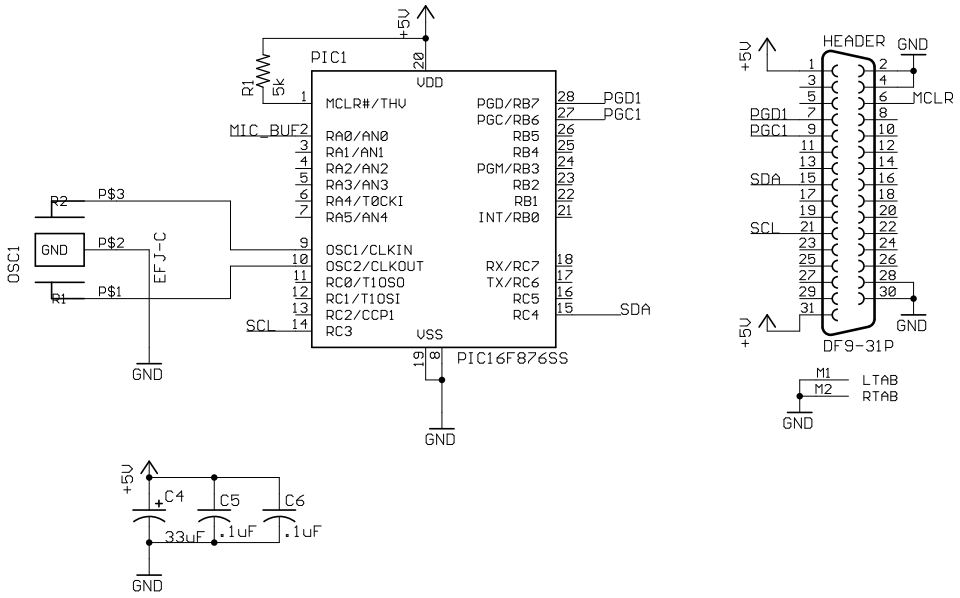

Figure A-5: Schematic for Audio Board — Sheet 1.

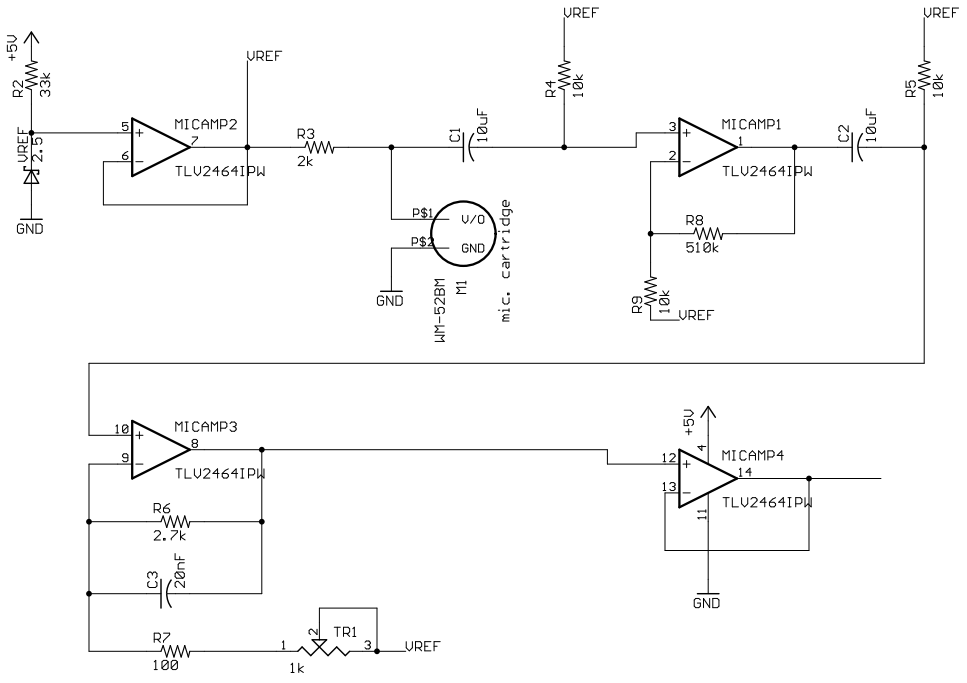

Figure A-6: Schematic for Audio Board — Sheet 2.

Appendix B

Layouts

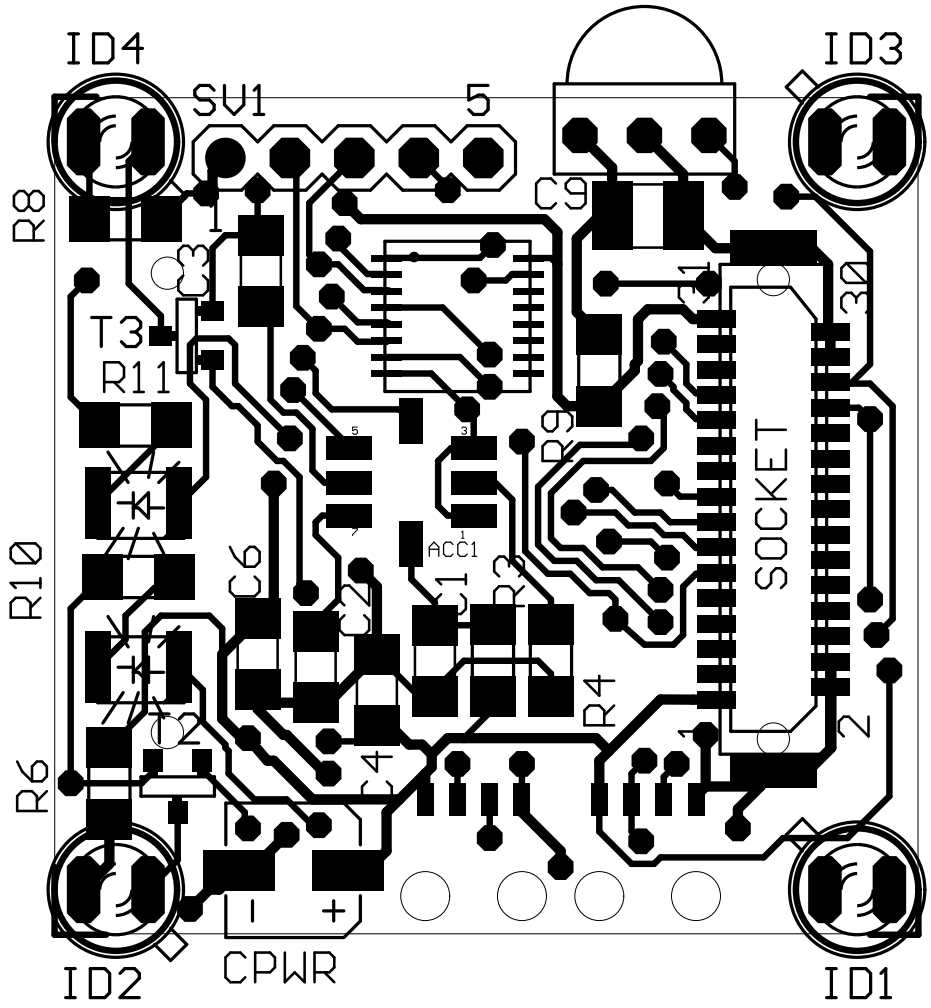

Figure B-1: Layout for Main Board — Component Side.

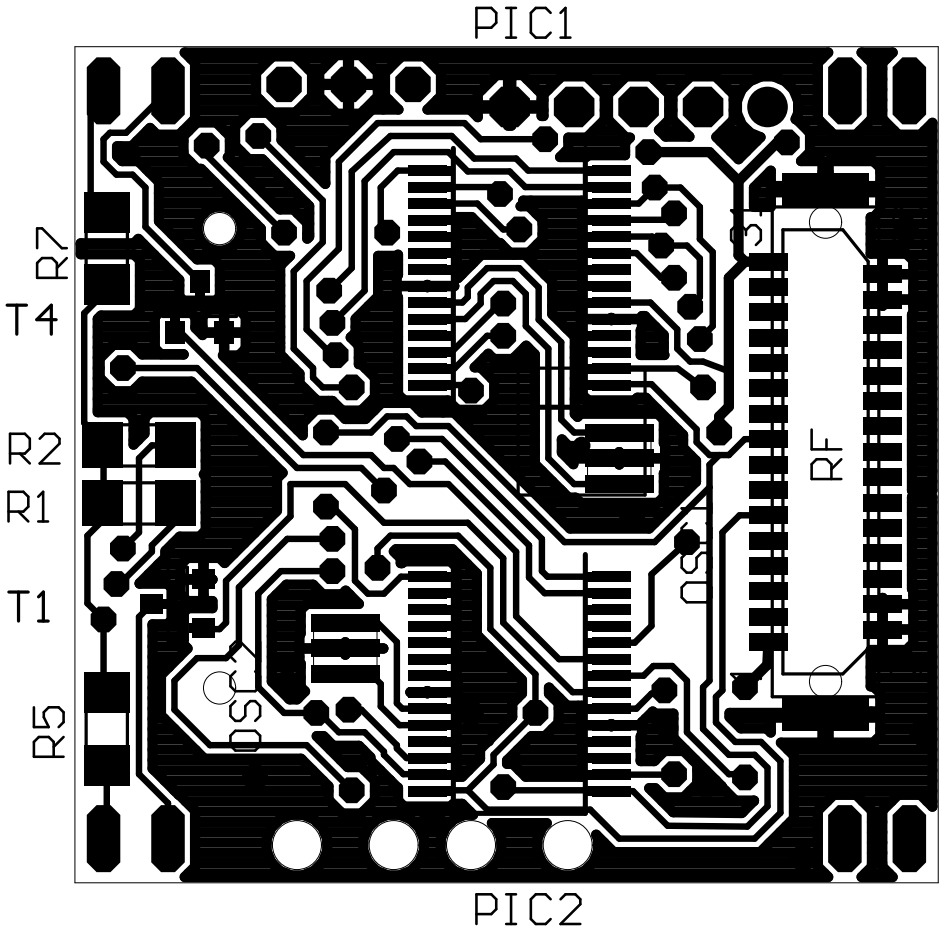

Figure B-2: Layout for Main Board — Solder Side.

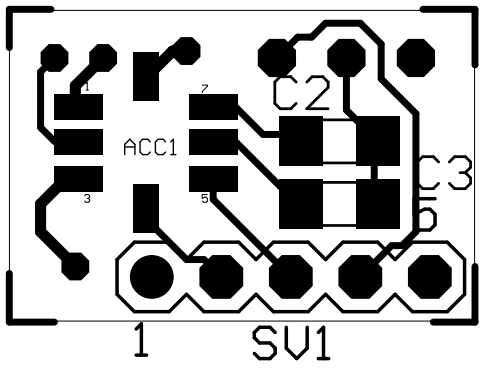

Figure B-3: Layout for Companion Board — Component Side.

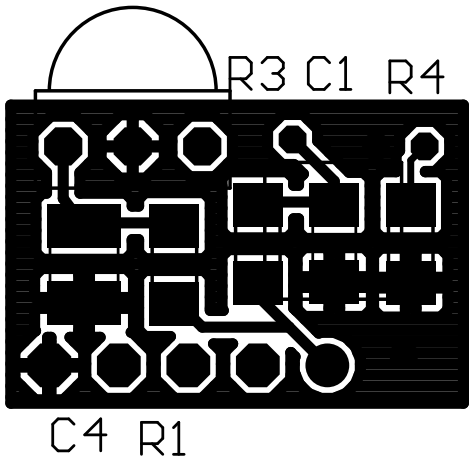

Figure B-4: Layout for Companion Board — Solder Side.

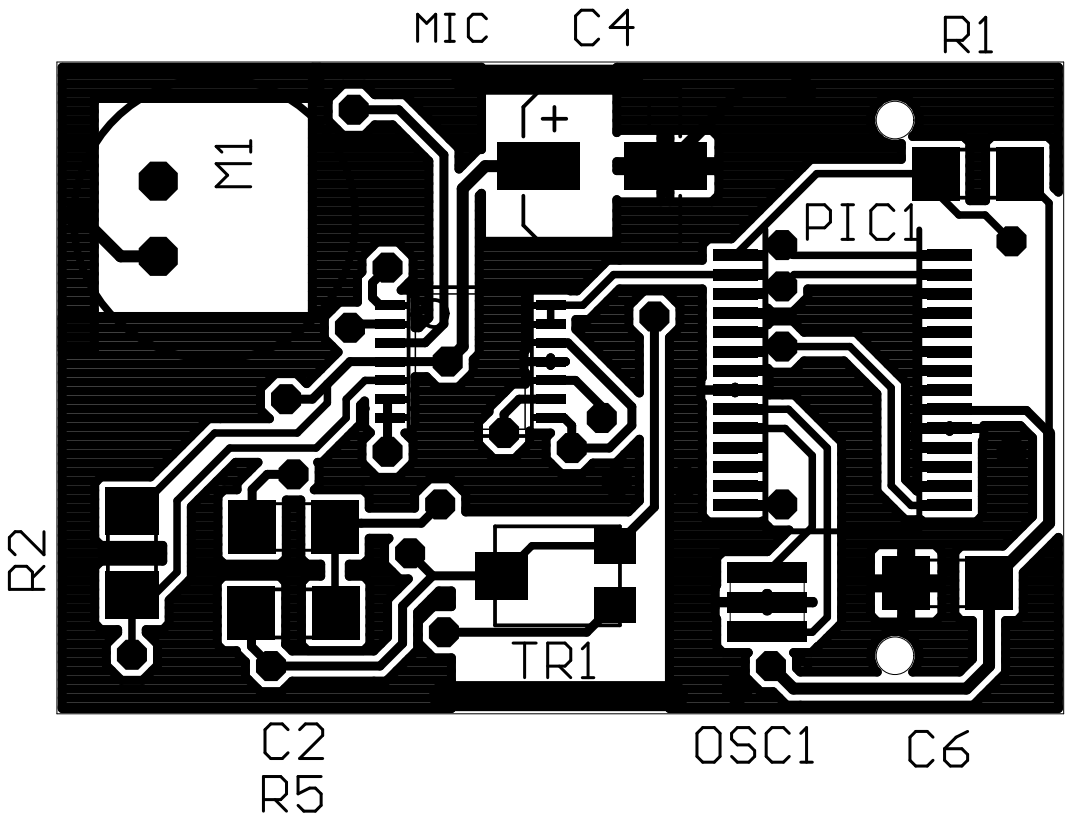

Figure B-5: Layout for Audio Board — Component Side.

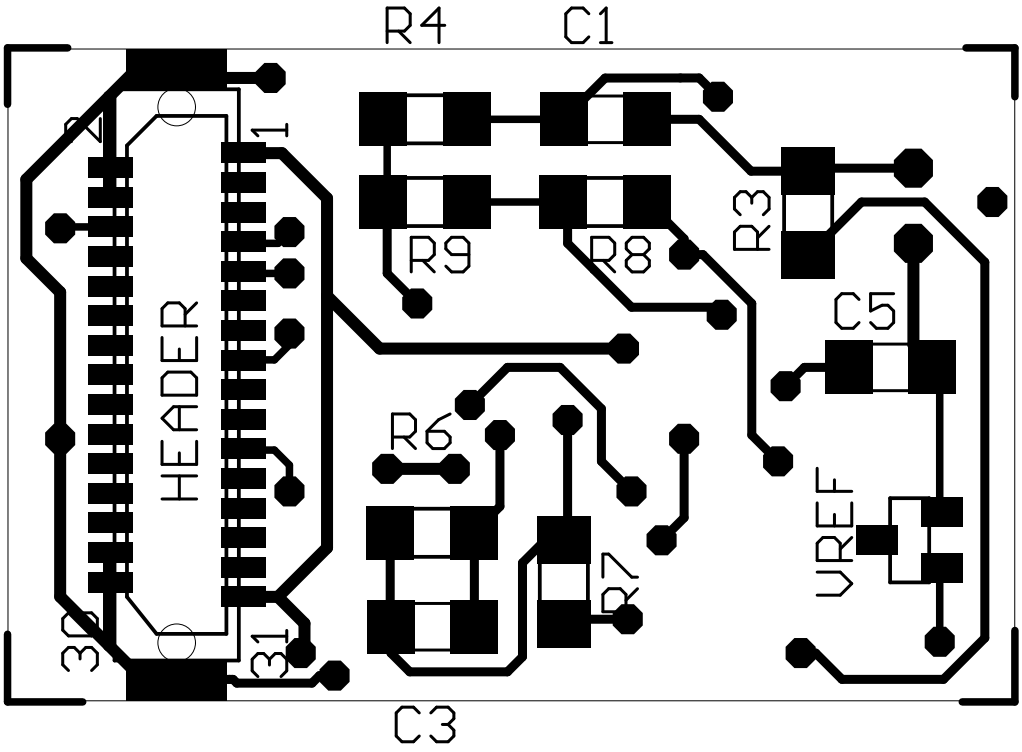

Figure B-6: Layout for Audio Board — Solder Side.

Appendix C

Micro-controller Code

```
/* accel.c
 *
 * Source code for the accel portion of the multisensor for
 * the SAK2. 16F876 reads two ADXL202JE accelerometers through
 * the Capture Unit. The two acceleromters are muxed to the
 * CCP inputs. Alternatively the original accel code could be
 * modified to just use the lines on port B.
 *
 * Chris Elledge <celledgemit.edu>
 * 7/30/2003
 *
*/

#include <16F876.h>

//20MHz clock
#use DELAY(clock=20000000)

#fuses HS, NOWDT, NOPROTECT, PUT, NOLVP

// PIN configuration
#DEFINE LED    PIN_C5
#DEFINE MUX_S  PIN_A0
#DEFINE MUX_OE PIN_A1

//Global Values
int acquire;
int timer_overflow;

//temporary ccp values
int ccp1_state;
long ccp1_start, ccp1_pulse, ccp1_end;
int ccp2_state;
long ccp2_start, ccp2_pulse, ccp2_end;

// end values
int data_out[4];
int readNext;

//errors
typedef enum {NO_ERROR, TIMEOUT_ACCEL1, TIMEOUT_ACCEL2, TIMEOUT_BOTH} ERROR_CODES;
ERROR_CODES errorno;

// I2C configuration
#define SLAVEID 0xB0
#define I2C_CLK PIN_C3
#define I2C_DTA PIN_C4

#use I2C(slave,address=SLAVEID,SCL=I2C_CLK,SDA=I2C_DTA,FORCE_HW)
typedef enum {NOTHING, ADDRESSED} I2C_STATE;
I2C_STATE fState;

// I2C interrupt handler
#INT_SSP
void ssp_interrupt() {
  byte incoming;

  if (i2c_poll() == FALSE) {
    // Reading values
    if (readNext == 80) {
```

```
      i2c_write(errorno);
    } else {
      i2c_write(data_out[readNext]);
      readNext=(readNext+1)%4;
    }
    fState=NOTHING;
  } else {
    // Writing values
    incoming = i2c_read();
    switch(fState) {
    case NOTHING:
      //expecting this address
      //check in case of broken state
      if (incoming == SLAVEID)
        fState = ADDRESSED;
      break;
    case ADDRESSED:
      //expecting a command
      switch(incoming) {
      case 0x00: //LED OFF
        break;
      case 0x01: //LED ON
        break;
      case 0x02: //Read X next
        readNext=0;
        break;
      case 0x03: //Read Y next
        readNext=1;
        break;
      case 0x04: //Read Z next
        readNext=2;
        break;
      case 0x05: //Read K next
        readNext=3;
        break;
      case 0x08: //Latch
        acquire=1;
        break;
      case 0x80: //Errorno
        readNext=80;
        break;
      case 0x81: //Clear Errorno
        errorno=NO_ERROR;
        output_low(LED);
        break;
      }
      fState=NOTHING;
      break;
    }
  }
}

#INT_CCP1
void ccp1_interrupt() {
  switch(ccp1_state) {
  case 0:
    ccp1_state = 1;
    ccp1_start = CCP_1;
    setup_ccp1(ccp_capture_fe);
    break;
  case 1:
    ccp1_state = 2;
    ccp1_pulse = CCP_1;
    setup_ccp1(ccp_capture_re);
    break;
  case 2:
    ccp1_state = 3;
    ccp1_end = CCP_1;
    break;
  }
}

#INT_CCP2
void ccp2_interrupt() {
  switch(ccp2_state) {
  case 0:
    ccp2_state = 1;
    ccp2_start = CCP_2;
    setup_ccp2(ccp_capture_fe);
    break;
  case 1:
    ccp2_state = 2;
    ccp2_pulse = CCP_2;
    setup_ccp2(ccp_capture_re);
    break;
  case 2:
    ccp2_state = 3;
    ccp2_end = CCP_2;
    break;
  }
```

```
}

#INT_TIMER1
void overflow() {
  // in case T1 overflows raise a flag
  timer_overflow = 1;
}

void capture() {

  //initialize states
  ccp1_state=0;
  ccp2_state=0;
  setup_ccp1(CCP_CAPTURE_RE);
  setup_ccp2(CCP_CAPTURE_RE);

  //clear timer1
  set_timer1(0);
  timer_overflow = 0;
  enable_interrupts(INT_TIMER1);

  // start capture
  enable_interrupts(INT_CCP1);
  enable_interrupts(INT_CCP2);

  // wait for capture to finish
  // or timer to overflow
  while(((ccp1_state+ccp2_state)<6) && (timer_overflow==0)) {
    delay_cycles(10);
  }

  disable_interrupts(INT_TIMER1);
  disable_interrupts(INT_CCP1);
  disable_interrupts(INT_CCP2);

  if (timer_overflow == 1) {
    output_high(LED);
  }

  // values are ready.
  return;
}

//Sets the MUX channel
void setup_mux(int accel) {
  output_low(MUX_OE);
  if (accel==0) {
    output_low(MUX_S);
  } else {
    output_high(MUX_S);
  }
}

//Captures the accelerometer values
void capture_all() {
  //temporary values
  long period1, period2, period3, period4;
  long pulse1, pulse2, pulse3, pulse4;
  signed long temp;
  signed int xo, yo, zo, ko;

  // capture first accelerometer
  // setup MUX
  setup_mux(0);
  capture();

  //overflow check
  if ((timer_overflow==1) && (errorno==NO_ERROR))
    errorno = TIMEOUT_ACCEL1;
  if ((timer_overflow==1) && (errorno==TIMEOUT_ACCEL2))
    errorno = TIMEOUT_BOTH;

  // compute values;
  period1=ccp1_end-ccp1_start;
  pulse1=ccp1_pulse-ccp1_start;
  period2=ccp2_end-ccp2_start;
  pulse2=ccp2_pulse-ccp2_start;

  // capture second accelerometer
  // setup MUX
  setup_mux(1);
  capture();

  //overflow check
  if ((timer_overflow==1) && (errorno==NO_ERROR))
    errorno = TIMEOUT_ACCEL2;
  if ((timer_overflow==1) && (errorno==TIMEOUT_ACCEL1))
    errorno = TIMEOUT_BOTH;

  // Turn off MUX
```

```
  output_high(MUX_OE);
  // compute values
  period3=ccp1_end-ccp1_start;
  pulse3=ccp1_pulse-ccp1_start;
  period4=ccp2_end-ccp2_start;
  pulse4=ccp2_pulse-ccp2_start;

  //xvalue
  temp = (signed long)( (( (float)pulse1/(float)period1)-0.5)*512.0);
  if (temp > 127)
    xo = 127;
  else if (temp < -127)
    xo = -127;
  else xo = (signed int) temp;

  //yvalue
  temp = (signed long)( (( (float)pulse2/(float)period2)-0.5)*512.0);
  if (temp > 127)
    yo = 127;
  else if (temp < -127)
    yo = -127;
  else yo = (signed int) temp;

  //zvalue
  temp = (signed long)( (( (float)pulse3/(float)period3)-0.5)*512.0);
  if (temp > 127)
    zo = 127;
  else if (temp < -127)
    zo = -127;
  else zo = (signed int) temp;

  //kvalue
  temp = (signed long)( (( (float)pulse4/(float)period4)-0.5)*512.0);
  if (temp > 127)
    ko = 127;
  else if (temp < -127)
    ko = -127;
  else ko = (signed int) temp;

  data_out[0]=xo;
  data_out[1]=yo;
  data_out[2]=zo;
  data_out[3]=ko;
}

main() {
  acquire = 0;
  ccp1_state = 0;
  ccp2_state = 0;
  fState = NOTHING;

  //startup test
  output_high(LED);
  delay_ms(500);
  output_low(LED);
  delay_ms(500);
  output_high(LED);
  delay_ms(200);
  output_low(LED);

  setup_timer_1(T1_INTERNAL);
  enable_interrupts(INT_SSP);
  enable_interrupts(GLOBAL);

  while(true) {
    delay_cycles(10);
    acquire=1;      //Free Run Mode
    if (acquire == 1) {
      capture_all();
      acquire = 0;
    }
    //sleep();
    delay_cycles(2);
  }
}
```

```
/* ir.c
 *
 * Source code for the IR portion of the multisensor for
 * the SAK2. 16F876 reads two IR receiver modules. It
 * expects the protocol to be the MIThril derivative of
 * the Sony IR protocol.
 *
 * Chris Elledge <celledgemit.edu>
 * 8/4/2003
 *
*/

#include <16F876.h>

//20MHz clock
#use DELAY(clock=20000000)

#fuses HS, NOWDT, NOPROTECT, PUT, NOLVP

//IR variables
int IR_LEDS=0b1111;
short IR_STATE=0;

//State I2C reading
short reader;
int readoffset;
int A[4]={1,2,3,4};
int B[4]={5,6,7,8};

//State variables for IR Receiver A
short IN_READ_A=0;
int A_array[20];
int A_counter=0;
short A_direction=0;
int A_element=0;

//State variables for IR Receiver B
short IN_READ_B=0;
int B_array[20];
int B_counter=0;
short B_direction=0;
int B_element=0;

//State variables for IR Transmitter
short IN_WRITE=0;
int TX_array[20];
int TX_counter=0;
short TX_direction=0;
int TX_element=0;
int TX_length=10;

// PIN configuration
#DEFINE LED PIN_C5
#DEFINE IR_TX0 PIN_B0
#DEFINE IR_TX1 PIN_B1
#DEFINE IR_TX2 PIN_B2
#DEFINE IR_TX3 PIN_B3
#DEFINE IR_RX1 PIN_C2
#DEFINE IR_RX2 PIN_C1
#BYTE PORT_B = 0x06

#use fast_io(B)
#define IR_B_TRIS 0b11110000
#define IR_C_TRIS 0b11011111


// I2C configuration
#define SLAVEID 0xA0
#define I2C_CLK PIN_C3
#define I2C_DTA PIN_C4

#use I2C(slave,address=SLAVEID,SCL=I2C_CLK,SDA=I2C_DTA,FAST,FORCE_HW)
typedef enum {NOTHING, ADDRESSED,WRITING} I2C_STATE;
I2C_STATE fState;

//definitions
void i2c_int_handler();

#INT_GLOBAL
void dispatch() {

 #ASM
  //Save W
  movwf 0x7F
  //Save Status
  movf 3,0
  bcf 3,5
  bcf 3,6
  movwf 0x21
  //Save high PC
```

```
  movf 0x0A,0
  movwf 0x20
  clrf 0x0A

  //Test TIMER2IF to determine interrupt
  btfss 0x0c,1 //Branch if not set
  goto testi2c

  //TIMER2 IR TOGGLE
  btfss IR_STATE //Branch if not set                                    100
  goto ir_is_off
 ir_is_on: //is set
  //bcf 7,5 //clear gLED
  movlw 0 // clear LEDs states
  movwf 0x06 //turn off IR
  bcf IR_STATE
  goto ir_end
 ir_is_off:
  //bsf 7,5 //set gLED
  movf IR_LEDS,0 //set LEDs states per flags                            110
  movwf 0x06 //turn on IR
  bsf IR_STATE
 ir_end:
  bcf 0x0C,1 //clear timer2 IF

 testi2c: //Test SSPIF to determine interrupt
  btfss 0x0c,3 //Branch if not set
  goto cleanup

  //Save Pointer                                                        120
  movf 0x04,0
  movwf 0x22
#ENDASM

i2c_int_handler();

#ASM
  bcf 3,5
  bcf 3,6
  bcf 0x0C,3 //clear i2c IF                                            130
  //Restore Pointer
  movf 0x22,0
  movwf 0x04
 cleanup:
  //Restore Pointer
  movf 0x20,0
  movwf 0x0A
  //Restore Status
  movf 0x21,0
  movwf 3                                                              140
  //Restore W
  movf 0x7F,W
 #ENDASM
}

void i2c_int_handler() {
  int incoming;
  if (!i2c_poll()) {
    // Reading values
    if(!reader) {                                                      150
      //Read ID A
      i2c_write(A[readoffset]);
    } else {
      //Read ID B
      i2c_write(B[readoffset]);
    }
    readoffset=(readoffset+1)%4;
    fState=NOTHING;
  } else {
    //Writing values                                                   160
    incoming = i2c_read();
    switch(fState) {
    case NOTHING:
      //expecting this address
      //check in case of broken state
      if (incoming == SLAVEID)
        fState = ADDRESSED;
      break;
    case ADDRESSED:
      switch(incoming) {                                               170
      case 0x0:
        output_low(LED);
        fState=NOTHING;
        break;
      case 0x1:
        output_high(LED);
        fState=NOTHING;
        break;
      case 0x2:
        //setup to read A                                              180
```

```
        reader=0;
        fState=NOTHING;
        readoffset=0;
        break;
      case 0x3:
        //setup to read B
        reader=1;
        fState=NOTHING;
        readoffset=0;
        break;
      case 0x4:
        //setup to write #IDs
        fState=WRITING;
        break;
      }
      break;
    case WRITING:
      // Get some number of bytes
      break;
    }
  }
}

int LED_STATE;
//LED toggler
toggle_LED() {
  if(LED_STATE==1) {
    LED_STATE=0;
    output_low(LED);
  } else {
    LED_STATE=1;
    output_high(LED);
  }
}



/**
 * Definition area for API functions
 **/
#define API_delay_limit 40
#define start 12
#define t_hold 4
#define d_zero 4
#define d_one 12
#define byte_delay 12

int ID1[4];
int IDA[4];
int IDB[4];

int tx_curr_byte;
int A_curr_byte;
int B_curr_byte;
int A_temp_byte;
int B_temp_byte;

/* **API**
 * This should initialize the state for the API
 * specific variables.
 */
void setup_API() {
  tx_curr_byte=0;
  A_curr_byte=0;
  B_curr_byte=0;
  ID1[0]=18;
  ID1[1]=85;
  ID1[2]=44;
  ID1[3]=3;
}

/* **API** ID to internal representation function
 * Change this function to match the needed protocol
 * The internal representation is an array of ints
 * that specifies how long the LED should be on and
 * off in order.
 *
 * {T1h, T1l, T2h, T2l, T3h ...}
 *
 * This will be called by the transmission code
 * should return 0 if the transmission is over
 * & 1 if it has set the tx_array to transmit.
 */
int byte_to_rep(short restart) {
  int temp = 0;
  int i;
  int a;

  if (restart)
    tx_curr_byte=0;
```

```
  if (tx_curr_byte > 3) {
    tx_curr_byte=0;
    return 0;
  }
  a = ID1[tx_curr_byte];

  tx_array[temp]=start;
  temp++;

  for(i=0; i<8; i++) {
    if(bit_test(a,i)) {
      tx_array[temp]=d_one;
    } else {
      tx_array[temp]=d_zero;
    }
    temp++;
    tx_array[temp]=t_hold;
    temp++;
  }
  tx_array[temp]=byte_delay;
  tx_length=18;

  tx_curr_byte++;
  return 1;
}

/* **API** internal rep to ID function
 * Change this function to match the needed protocol
 *
 * This will be called every tick to determine if a byte
 * has been read by the receiver. array_elements is the number
 * of elements that have been filled in the array. reader is
 * the IR reader buffer to look at.
 *
 * return 0 when a full byte in,
 * return 1 when waiting for more,
 * return 2 on error to bail on transmission early
 *  (this is useful if it doesn't start with a start bit)
 */
int rep_to_byte(short reader, int array_elements) {
  int temp,i,element;

  if (array_elements<18)
    return 1;

  if (reader) {
    //read A buffer
    temp = A_array[0];

    if ((temp<10)||(temp>14)) {
      A_curr_byte=0;
      return 2; //first bit is not a start bit
    }

    //get the bits
    element = 1;
    for (i=0; i<8; i++) {
      temp = A_array[element];
      if(temp<8) {
        //it is a zero
        bit_clear(A_temp_byte,i);
      } else if(temp<18) {
        //it is a one
        bit_set(A_temp_byte,i);
      } else {
        A_curr_byte=0;
        return 2; //bad length
      }
      element++;
      temp = A_array[element];
      if ((temp<2)||(temp>8)) {
        A_curr_byte=0;
        return 2;
      }
      element++;
    }
    //It seems good, so use it.
    IDA[A_curr_byte]=A_temp_byte;
    A_curr_byte++;
    if (A_curr_byte>3) {
      //finished ID
      IN_READ_A=0;
      A[0]=IDA[0];
      A[1]=IDA[1];
      A[2]=IDA[2];
      A[3]=IDA[3];
    }
    toggle_LED();
    return 0;
```

```
  } else {
    //read B buffer
    temp = B_array[0];

    if ((temp<10)||(temp>14)) {
      B_curr_byte=0;
      return 2; //first bit is not a start bit
    }

    //get the bits
    element = 1;
    for (i=0; i<8; i++) {
      temp = B_array[element];
      if(temp<8) {
        //it is a zero
        bit_clear(B_temp_byte,i);
      } else if(temp<18) {
        //it is a one
        bit_set(B_temp_byte,i);
      } else {
        B_curr_byte=0;
        return 2; //bad length
      }
      element++;
      temp = B_array[element];
      if ((temp<2)||(temp>8)) {
        B_curr_byte=0;
        return 2;
      }
      element++;
    }
    //It seems good, so use it.
    IDB[B_curr_byte]=B_temp_byte;
    B_curr_byte++;
    if (B_curr_byte>3) {
      //finished ID
      IN_READ_B=0;
      B[0]=IDB[0];
      B[1]=IDB[1];
      B[2]=IDB[2];
      B[3]=IDB[3];
    }
    toggle_LED();
    return 0;
  }
}


// Reading State Machine
void readA() {
  int temp;
  //Are we already reading something?
  if(!IN_READ_A) {
    //if not, then we are waiting for a transmission
    if (!input(IR_RX1)) {
      //transmission detected
      IN_READ_A=1;
      A_element=0;
      A_counter=0;
      A_direction=0;
    }
    return;
  }

  // We are currently reading a transmission
  A_counter++; //every clock tick increment
  if (!A_direction) {
    // we are waiting for signal to pause

    if (input(IR_RX1)) {
      //signal paused
      A_array[A_element]=A_counter;
      A_counter=0;
      A_direction=1;
      A_element++;
    } else {
      //nothing interesting has happened
      return;
    }

  } else {
    // we are waiting for signal to resume

    if (!input(IR_RX1)) {
      //signal resumed
      A_array[A_element]=A_counter;
      A_counter=0;
      A_direction=0;
      A_element++;
    }
```

```
  //check if we have gone over the API's
  // delay time limit
  else if (A_counter>API_delay_limit) {
    //we should try to end the byte.
    A_array[A_element]=A_counter;
    A_counter=0;
    A_element++;
    IN_READ_A=0;
  } else {
    //nothing interesting
    return;
  }
 }

 // if the signal was there and just ended
 // then we may be at the end
 // (case A_direction==0 && IR_RX1==1)

 temp = rep_to_byte(1,A_element);

 if(temp==0) {
   // We got a full Byte in
   A_element=0;
 }
 if(temp==2) {
   // We were told to bail
   IN_READ_A=0;
 }
}

void readB() {
 //Are we already reading something?
 if(!IN_READ_B) {
   //if not, then we are waiting for a transmission
   if (!input(IR_RX2)) {
     //transmission detected
     IN_READ_B=1;
     B_element=0;
     B_counter=0;
     B_direction=0;
   }
   return;
 }

 // We are currently reading a transmission
 B_counter++; //every clock tick increment
 if (!B_direction) {
   // we are waiting for signal to pause

   if (input(IR_RX2)) {
     //signal paused
     B_array[B_element]=B_counter;
     B_counter=0;
     B_direction=1;
     B_element++;
   } else {
     //nothing interesting has happened
     return;
   }

 } else {
   // we are waiting for signal to resume

   if (!input(IR_RX2)) {
     //signal resumed
     B_array[B_element]=B_counter;
     B_counter=0;
     B_direction=0;
     B_element++;
   }
   //check if we have gone over the API's
   // delay time limit
   else if (B_counter>API_delay_limit) {
     //we should try to end the byte.
     B_array[B_element]=B_counter;
     B_counter=0;
     B_element++;
     IN_READ_B=0;
   } else {
     //nothing interesting
     return;
   }
 }

 // if the signal was there and just ended
 // then we may be at the end
 // (case B_direction==0 && IR_RX2==1)

 temp = rep_to_byte(0,B_element);
```

```
  if(temp==0) {
    // We got a full Byte in
    B_element=0;
  }
  if(temp==2) {
    // We were told to bail
    IN_READ_B=0;
  }
}

// Transmitter State Machine
write() {

  //do we have something to write?
  if (!IN_WRITE)
    return;

  TX_counter++; //increment every cycle

  //check the counter
  if (TX_counter >= TX_array[TX_element]) {

    //It is time to change the output
    if (!TX_direction) {

      toggle_LED();
      // resume signal
      IR_LEDs = 0xff;
      TX_direction=1;
      TX_counter=0;
      TX_element++;

    } else {

      toggle_LED();
      //pause signal
      IR_LEDs = 0x00;
      TX_direction=0;
      TX_counter=0;
      TX_element++;


    }
    //check if we are done transmitting
    if (TX_element >= TX_length) {
      if(byte_to_rep(0)==0) {
        IN_WRITE=0;
        disable_interrupts(INT_TIMER2);
        PORT_B=0;
        toggle_LED();
      }
      TX_element=0;
    }
  }
}

main() {
  long cycles=0;

  set_tris_b(IR_B_TRIS);
  fState = NOTHING;

  output_low(IR_TX0);
  output_low(IR_TX1);
  output_low(IR_TX2);
  output_low(IR_TX3);

  IN_WRITE=0;
  TX_array[6] = 150;
  TX_counter=0;
  TX_direction=0;
  TX_element=0;
  TX_length=7;
  IR_LEDs=0x00;

  setup_API();

  //bootup test
  output_high(LED);
  delay_ms(500);
  output_low(LED);
  delay_ms(500);
  output_high(LED);
  delay_ms(200);
  output_low(LED);

  //setup timer two for 38kHz period (76kHz update rate)
  //200ns increment. need 66 iterations in a period
  setup_timer_2(T2_DIV_BY_1, 65, 1);
```

```
  enable_interrupts(INT_TIMER2);
  enable_interrupts(INT_SSP);
  enable_interrupts(GLOBAL);

  //Stup Timer 0 to determine when to start transmissions
  //setup_timer_0(T0_DIV_BY_8);
  setup_counters(RTCC_INTERNAL,RTCC_DIV_4);

                                                                                    640
  while(true) {
    //Wait for timer0 pulse as our heartbeat
    while(get_timer0()<125) {
      delay_cycles(1);
    }
    set_timer0(0);

    //Call the IR writing function
    write();
                                                                                    650
    //Call the IR reading function
    readA();
    readB();

    cycles++;
    if (cycles > 65000) {
      //start writing the ID
      cycles=0;
      IN_WRITE=1;
      TX_counter=0;                                                                 660
      TX_element=0;
      TX_direction=1;
      IR_LEDs = 0xff;
      byte_to_rep(1);
      enable_interrupts(INT_TIMER2);
      toggle_LED();
    }
  }
}
```

```
/* audio.c
 *
 * Source code for the audio portion of the multisensor
 * for the SAK2. The 16F876 reads one analog channel
 * at a specific frequency. There should be a reasonable
 * amount of buffering provided by this device.
 *
 * The amount required depends upon the sampling rate of
 * the audio and on the frequency that the device is
 * queried at. 16kHz with 10ms of buffering, we need
 * 160bytes of buffering. At an I2C rate of 400kHz, this
 * would take a little less than 4ms to transfer. If this
 * buffer is broken into fourths, then it should only
 * take 1ms to transfer at the faster 3ms query rate.
 *
 * This code supports 4 buffers of 50 bytes each to
 * handle the audio stream.
 *
 * Chris Elledge <celledgemit.edu>
 * 8/16/2003
 *
 */

#include <16F876.h>
#device PIC16F876 *=16

//20MHz clock
#use DELAY(clock=20000000)

#fuses HS, NOWDT, NOPROTECT, PUT, NOLVP

//I2C flag
short doI2C;

//audio buffers
int buf0[40];
int buf1[40];
int buf2[40];
int buf3[40];
int buf4[40];
int buf5[40];
int buf6[40];
#DEFINE BUF_SIZE 40
#DEFINE BUF_NUM 7

//current buffer
int current_buf;
//buffer with earliest data
int first_buf;

//write offset for current buffer
int writer;

//read offset for reading buffer
int reader;

//overflow flag
int overflow;

//analog stuff
#bit ADC_GO = 0x001F.2
#bit ADFM = 0x009F.7
#byte ADRESH = 0x001E

//TRIS
#define AUDIO_TRIS_B 0b11111110
//#use fast_io(B)

//I2C configuration
#bit I2C_DA = 0x0094.5
#bit I2C_OV = 0x0014.6
#define SLAVEID 0x70
#define I2C_CLK PIN_C3
#define I2C_DTA PIN_C4

#use I2C(slave,address=SLAVEID,SCL=I2C_CLK,SDA=I2C_DTA,FORCE_HW)
typedef enum {NOTHING, ADDRESSED} I2C_STATE;
I2C_STATE fState;
int readNext;

#INT_TIMER2
void sample_signal() {
  //set the go bit in the ADC register
  ADC_GO=1;
  output_high(PIN_B0);
}

#INT_AD
void sample_done() {
  //the sample is done
```

```
  //read the sample
  //write it to the buffer
  switch(current_buf) {
  case 0:
    buf0[writer]=ADRESH;
    break;
  case 1:
    buf1[writer]=ADRESH;
    break;
  case 2:
    buf2[writer]=ADRESH;
    break;
  case 3:
    buf3[writer]=ADRESH;
    break;
  case 4:
    buf4[writer]=ADRESH;
    break;
  case 5:
    buf5[writer]=ADRESH;
    break;
  case 6:
    buf6[writer]=ADRESH;
    break;
  }
  //increment writer with buffer advance
  writer++;
  if (writer>=BUF_SIZE) {
    //advance buffer
    writer=0;
    current_buf=(current_buf+1)%BUF_NUM;
    //test for overflow
    if (current_buf==first_buf) {
      overflow=1;
      first_buf=(first_buf+1)%BUF_NUM;
    }
  }
  output_low(PIN_B0);
}

#INT_SSP
void i2c_interrupt() {
   doI2C=1;
}

void i2c_handler() {
  byte incoming;
  signed int temp;
  doI2C=0;
  I2C_OV=0;

  if (i2c_poll() == FALSE) {
    //Reading Values
    switch(readNext) {
    case 2:
      //# of ready buffers
      temp = current_buf - first_buf;
      if (temp < 0)
        temp+=BUF_NUM;
      i2c_write(temp);
      break;
    case 3:
      i2c_write(BUF_SIZE);
      break;
    case 4:
      //write buffers

      //test if there is a reading buffer
      if (first_buf == current_buf) {
        i2c_write(127);
        break;
      }
      //pick the current buffer
      switch(first_buf) {
      case 0:
        i2c_write(buf0[reader]);
        break;
      case 1:
        i2c_write(buf1[reader]);
        break;
      case 2:
        i2c_write(buf2[reader]);
        break;
      case 3:
        i2c_write(buf3[reader]);
        break;
      case 4:
        i2c_write(buf4[reader]);
        break;
```

```
      case 5:
        i2c_write(buf5[reader]);
        break;
      case 6:
        i2c_write(buf6[reader]);
        break;
      default:
        i2c_write(127);
      }

      // increment reader with buffer advance
      reader++;
      if (reader>=BUF_SIZE) {
        //advance buffer
        reader=0;
        first_buf=(first_buf+1)%BUF_NUM;
      }
      break;
    case 5:
      i2c_write(overflow);
      break;
    case 7:
      i2c_write(current_buf);
      break;
    case 8:
      i2c_write(first_buf);
      break;
    }
    fState=NOTHING;
  } else {
    //Writing Values
    incoming = i2c_read();

    if (!I2C_DA) //This is an address
      fState=NOTHING;

    switch(fState) {
    case NOTHING:
      //expecting this address
      //check in case of broken state
      if (incoming == SLAVEID)
        fState = ADDRESSED;
      break;
    case ADDRESSED:
      //expecting a command
      switch(incoming) {
      case 0x0:
        //Yeah, I forgot an LED on this PIC
        break;
      case 0x1:
        break; //ditto
      case 0x2:
        //setup to read number of ready buffers
        readNext=2;
        break;
      case 0x3:
        //setup to read length of buffers
        readNext=3;
        break;
      case 0x4:
        //setup to read buffers
        readNext=4;
        reader=0;
        break;
      case 0x5:
        //setup to read overflow
        readNext=5;
        break;
      case 0x6:
        //reset overflow
        overflow=0;
        break;
      case 0x7:
        readNext=7;
        break;
      case 0x8:
        readNext=8;
        break;
      }
      break;
    }
  }
}

main() {
  //variable initialization
  //set_tris_B(AUDIO_TRIS_B);
  fState=NOTHING;
  current_buf=0;
  first_buf=0;
```

```
  writer=0;
  reader=0;
  doI2C=0;

  //setup analog (PORTS & channel)
  setup_adc_ports(A_ANALOG);
  setup_adc(ADC_CLOCK_DIV_32);
  ADFM=0; //left justify output for 8-bit read
  set_adc_channel(0);
                                                                                                    280
  delay_ms(5);

  //setup timer2 for 16kHz interrupt
  // 1/(200ns*2*156)=16025.64kHz
  setup_timer_2(t2_div_by_1, 155, 2);
  // 1/(200ns*4*156)=8012.82kHz
  //setup_timer_2(t2_div_by_1, 155, 4);

  //setup interrupts
  enable_interrupts(INT_SSP);                                                                       290
  enable_interrupts(INT_AD);
  enable_interrupts(INT_TIMER2);

  //Start
  set_timer2(0);
  enable_interrupts(GLOBAL);

  while(true) {
    if (doI2C)
      i2c_handler();                                                                                300
  }
}
```

Bibliography

[1] Ir-suck: The squirt 2 tag reader. http://www.media.mit.edu/wearables/mithril/hardware/.

[2] Tanzeem Choudhury and Alex Pentland. The sociometer: A wearable device for understanding human networks. Technical report, MIT Media Lab. http://web.media.mit.edu/ tanzeem/TR-554.pdf.

[3] Richard W. DeVaul and Steve Dunn. Realtime motion classification for wearable computing. http://www.media.mit.edu/wearables/mithril/realtime.pdf, 2001.

[4] Nathan Eagle and Alex Pentland. Social network computing. In *Fifth International Conference on Ubiquitous Computing (UbiComp)*, Seattle, WA., 2003.

[5] Vadim Gerasimov. Hoarder data acquisition system. http://vadim.www.media.mit.edu/Hoarder/Hoarder.htm.

[6] Josh Weaver. A wearable health monitor to aid parkinson disease treatment. M.s. thesis, Massachussetts Institure of Technology, June 2003.

[7] Josh Weaver and Ari Benbasat. Squirt 2. http://www.media.mit.edu/wearables/mithril/hardware/, June 2002.