WIRELESS PEOPLE COUNTING SENSOR NETWORK, PEOPLE ATTRACTOR, AND TEMPO INDICATOR

A Project for
Human Computer-Interaction Group

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirement for the Degree of
Master of Engineering

By
Arun Israel (Electrical and Computer Engineering)
and Eric Chieh-Yin Lee (Computer Science)
Project Advisor: Dr. Bruce R. Land
Degree Date: May 2005

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Wireless People Counting Sensor Network, People Attractor, and Tempo Indicator

Authors: Arun Israel and Eric Chieh-Yin Lee

Abstract: An interactive piece of art was developed to be deployed in a museum space. The system gathers data based on the number of people and their "presence" within the measured area and drives the generation and morphing of a visual piece of art using non-invasive technologies to maintain privacy and minimize annoyance. This system consists of 3 main components: a wireless people counter, a "birdie"- a device to attract people to interact with, and a tempo indicator to measure speed of movement. The Wireless People Counter was built using LEDs used as light sensors and RF links, the Attractor was built using a Pyroelectric Infrared (PIR) Sensor and Winbond Voice Coder chip, and Tempo Indicator was built using the PIR sensor. The design choices and implementation were motivated by keeping an eye towards minimal cost, maximum performance, and extensibility of system to incorporate future sensor technologies, and maintaining an easy implementation when more nodes need to be deployed in the system.

Report Approved by
Project Advisor: _____ Date: _____

Executive Summary

The idea was to have an interactive piece of art in a museum space. A PhD student at the Human Computer Interaction (HCI) Group of the Computing and Information Science Department wanted to design space for context aware computing that draws from advances in the development of ambient systems [5]. The general feature list was given by HCI. The idea is to have a piece of art that is generated and morphed based on the number of people and their "presence" in each section of the museum as well as the people flow trend in the Asian Gallery of Herbert Johnson Museum at Cornell University. This project was done under the direction of Kirsten Boehner (HCI) who detailed the feature requirements and budget constraints that influenced our design decisions.

Three different devices were built for this interactive museum experience. The first phase was to build an attractor, called "birdie", which contains a recorded bird sound which plays when no motion is detected in a two minute period. The idea is that the bird sound will attract people to that part of the museum where the birdie is deployed. The second device is needed to sense how fast people are moving in front of a piece of art. Finally, a people counting sensor network was built to track the amount of people present in each section of the museum. After much consideration of different technical design aspects, we were able to arrive at a design that would fulfill the feature requirements.

For the first phase, we had to choose appropriate technologies to meet the design criteria for the attractor and the tempo indicator. A motion detector is used in both devices, but for the second phase of the project, direction of movement is needed so an LED array used as light sensors was utilized. An intelligent sensor network was designed and built to count the number of people in each section of the gallery. The sensor network uses a basic transmitter and receiver laid out in a star topology – a primary and slave network. The base station is able to discover all the alive nodes and periodically poll the sensors on the network for data. The base station and the node package their communications based on our extensible RF protocol and the data is transmitted via a Manchester encoding scheme. The sensor information that the base station collects is fed through an RS-232 serial connection and can be displayed via HyperTerminal or another serial application.

All three devices use Atmel Mega32 microcontroller. The birdies are implemented using a Pyroelectric Infrared(PIR) Sensor, miniature speaker, and a Winbond Voice Coder. The speed detector is implemented using the PIR and the data is communicated through an RS-232 serial interface as well. The people counter is implemented using an Atmel Mega32 microcontroller, a Radiotronix 433MHz transmitter and receiver, and LEDs used as light sensors for detecting temporal lighting change.

Overall, this project was successful. In the prototype setup of the sensor network, the base station was able to communicate with two sensing nodes at two different locations that are 70 feet apart from the base station and it was able to get the precise people count. The birdies and speed detector have been deployed in the museum for collecting research data for the HCI group. We are very satisfied with the outcome of our design experience and also the interaction with our client to meet the project requirements.

Division of Labor

This Master of Engineering Design Project was done through the work of Arun Israel and Eric Lee. They both met with Professor Bruce Land to go over the system requirements, evaluation of various sensor technologies, and implementation strategies. There were several meetings with Professor Land to discuss the progress of all phases of the project over the course of the year.

In the fall semester, work began on developing the birdie and tempo indicator. Both members investigated various sensor technologies in terms of cost, performance, and ease of implementation to determine which solution would work best. The Pyroelectric Infrared(PIR) Sensor was chosen for both of these products. Arun worked on designing, testing, and implementing the hardware necessary for the PIR to function properly and interface correctly with the Atmel Mega32. Eric worked on developing the software interface to measure the voltage change of the PIR and he was responsible for designing, testing, and implementing the Winbond Voice Coder section, RS-232 serial interface, and overall system integration. Both members worked on debugging and testing the overall system operation and each member reviewed the other's work.

During the spring semester, work continued on developing a sensor technology to count the number of people in a given room and to transmit this information wirelessly to a central base station. Again, several sensor technologies and wireless communication standards were studied in terms of cost, performance, and ease of implementation to determine which would work best. LEDs used as light sensors were chosen to implement the people counter and RF links would transmit the information wirelessly.

Arun worked on the design, testing, and implementation of the LED from a hardware perspective, while Eric focused on developing the software and RS-232 interface to gather the data and interact with user commands. Both members reviewed the work of the other and helped each other with testing and building a breadboard and solder board version. Professor Land helped us resolve the technical problems we had with the LEDs and amplification circuitry. Concurrently, both members studied the Manchester encoded RF protocol developed by Dan Golden to determine modifications for use in our project. Eric began work on developing a breadboard based version of the base station and node to establish RF performance and testing of the communication

protocol while Arun finished work on hardware and software of the LED people counter. Arun then worked on modifying the software code at the base station and node to transmit repeatedly and rectify failures from lack of synchronization between transmitter and receiver. Eric and Arun worked on developing and testing the RF protocol to be used by both the base station and the node at a high level. Eric developed and implemented the software code and state machines for the base station, while Arun handled the same responsibilities on the node side. Both members worked on developing the ADC state machine for the LED circuitry to fit within the node's idle time and both members were responsible for testing and integrating the LED code within the node code.

The overall system testing was done by both members as part of a final validation. Both members contributed to this report, with each member writing the sections they were primarily responsible for. Numerous drafts of this report were made, with each member looking at the other's writing and making corrections as necessary. Each member contributed significantly to the overall success of this project.

Table of Contents

Introduction

New technology is often introduced in museum environments as a means to enhance existing practices and institutional roles [5]. For example, one of the applications in Museum of Science in Boston is a stair in which each step is a piano note, so one can create music by stepping on the steps going up or down. This installation successfully drew attention from children and adults coming down from upstairs. Some kids would run up and down on the steps to try to create different melodies. This is a clever way of utilizing simple technology but it stimulates creativity in children as well as adults [5].

Working with Human Computer Interaction Group in Cornell University, we have an idea of a virtual display that is generated from the number of people present and also the tempo in which people are moving in the display room. The virtual display could be composed of anything from very complex art to simple morphing of basic shapes.

This project has two phases. The first phase consists of building an attractor and a tempo indicator. The second phase of the project is to build a wireless people counting sensor network. All of these projects will be used on the fifth floor in the Herbert Johnson Museum at Cornell University.

Feature Requirements

The feature requirements were given by the Human Computer-Interaction group. The broad idea was first discussed and after understanding the context in which these devices will be used, we went to the drawing board to come up with the solution. After several meetings to discuss possible implementations, the following set of requirements were defined for each component of the system:

Attractor/Birdies

- Plays bird sound when no motion is detected for two minutes
- Stops singing when motion is detected
- Sleeps when there is no motion detected for 10 minutes, i.e, the bird has been singing for the past 10 minutes

- Collects data for 3 days of activity, keeping track of how many times the sensor is triggered as measured every half-minute of each day during the time period of 10am to 4pm

Tempo Indictor

- Get a sense of how fast people are moving in front of a particular piece of art
- The tempo information should be accessible via a serial interface

Wireless People Counter Sensor Network

- Tallies the number of people in every section of the museum
- Be able to deploy a low cost network with 15 nodes with information accessible via a serial link at a central location.

Background Information

There were a variety of technologies utilized for the completion of this project. Technical background information is provided on the main components such as the LEDs used as light sensors, RS-232 Serial Communication, RF communication, RF Receivers, RF Transmitters, Manchester Encoding, Clock Recovery, Synchronization, FCC Regulations, and the Pyroelectric Infrared Sensor with Fresnel Lens.

LEDs used as light sensors

A light-emitting diode (LED) is a semiconductor device that emits visible light when biased properly. This light can range in colors (wavelengths) from blue (~400nm) to red (~700nm). The LED itself consists of two regions, one P-type semiconductor and one N-type semiconductor. These regions are placed together as in a diode and packaged to allow visible light to pass through. However, this same structure can be used to sense light and produce a voltage corresponding to the brightness of the light. The LEDs are used because they are easy to control, they have a low power requirement, long life, and high efficiency.

The LED we have used in the people counting is the Agilent Red Water Clear 630nm 5mm LEDs with 23 degree of viewing angle. Black shrink tube covers the LED

cylindrical clear body is used to make sure the field of view of the LEDs is as narrow as possible. It also blocks out the ambient light that might shine in from the side of the LED, since we only want to detect the lighting condition change in the front part of the LED.

RS-232 Serial Communication

The AVR MCU has a hardware UART, which is used to talk to the PC ComPort. A simple terminal program, such as Hyperterminal, on the PC connected to COM1 can be used to talk to the MCU. The terminal program should be set to 9600 baud, no parity, one stop bit, and no flow control. The connection to the PC for MCU is on D0 and D1 pins. D0 is the RX and D1 is TX.

RF communication

Information is transmitted wirelessly at 433MHz. There are many important physical design considerations when building a wireless link such as path loss, multipath fading, interference, and antennas. Path loss is simply a loss of signal strength over distance where the signal level diminishes by a factor which is inversely proportional to distance. RF signals can be reflected, absorbed, or scattered by objects along the path between a receiver and a transmitter. This can lead to 1 signal taking multiple paths and arriving at different times to a receiver. This multipath fading can be destructive to a transmission if the signals along different paths have different phases. Strategies to deal with this are built into the receiver such as equalizers which account for variations in the transmission channel along a given path from transmitter to receiver.

Interference happens when multiple devices operate on the frequency band. In our case 433MHz is often used in RC cars, but the museum has no other sources of RF energy at this frequency and we can safely assume that we will not encounter interference from RC cars. Interference from out of band transmissions is possible, but filtering present on the antennas rejects those signals.

Antennas are crucial to high performance, wireless links. A simple ¼ wave wire antenna could be used, but the transmitter and receiver modules require a 50ohm load for maximum power transfer. We saw reliable communication of about 10-15 feet with the

¼ wave wire antennas. With the ¼ wave whip antennas purchased from Linx Technologies, we got an indoor range of about 75 feet in a very poor environment. Both of these types of antennas require a ground plane perpendicular to the antenna for reliable operation. The transmitter and receiver antennas need to spaced as far away from each other as possible and they require shielding from the internal circuitry.

RF Receiver

The Radiotronix receiver modules used make use of a super-regenerative AM detector which consists of a gain stage with positive feedback to ensure oscillation. This is used to demodulate the incoming AM carrier. Detection happens by measuring the emitter current of the gain stage and an RF signal at the frequency of the main oscillation will aid the main oscillation in restarting. For example, if the RF input amplitude increases, the main oscillation stays on longer resulting in more emitter current. So the original baseband signal that was upshifted to the RF baseband is recovered by low pass filtering the emitter current.

These receiver modules are constantly reading and have a noisy nominal voltage of around 2V when no transmission is detected. A digital data slicer converts the baseband signal for the MCU to read. To aid the receiver, the data is usually transmitted in non-return to zero (NRZ), pulse width modulation (PWM) or Manchester encoding.

RF Transmitter

The RF transmitters from Radiotraonix are small, cheap, and reliable. This module uses OOK modulation which is a binary form of amplitude modulation. A "0" is sent by the transmitter not sending anything (thereby suppressing the carrier) and when a "1" is sent, the carrier is on. The transmitter requires some time to start transmitting to account for the start up time of the oscillator which is a negative resistance LC oscillator whose center frequency is controlled by a surface acoustic wave resonator.

Manchester Encoding

This coding scheme is motivated by a desire to transmit the data and clock at the same time and to maintain a balance between received 1's and 0's at each node for the

receiver to operate properly. Traditional schemes like nonreturn to zero (NRZ) and pulse width modulation fail to accomplish this and in cases where we transmit all 1's or all 0's, the receiver could lose synchronization with the transmitter. Manchester encoding works around this by transitioning the logic level in the middle of every transmitted bit. For example, a "0" is sent by a falling edge or a transition from high to low and a "1" is sent by a rising edge or a transition from low to high.

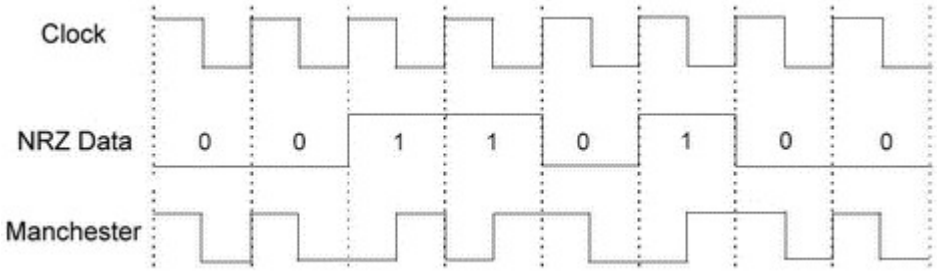


Figure 1: Various Encoding schemes[6]

Clock Recovery

The clock can be recovered by the receiver through hardware (PLL) or software by measuring the time between subsequent edges. Specifically, the external interrupt of the MCU is set to go off on any data edge received which then starts a timer which is stopped upon the reception of the next data edge. If the value held in the timer is ½ clock period, the receiver is unable to recover the clock at this point in time because this could happened between two different bits or in the middle of the transmission of 1 bit. (See Figure 2).

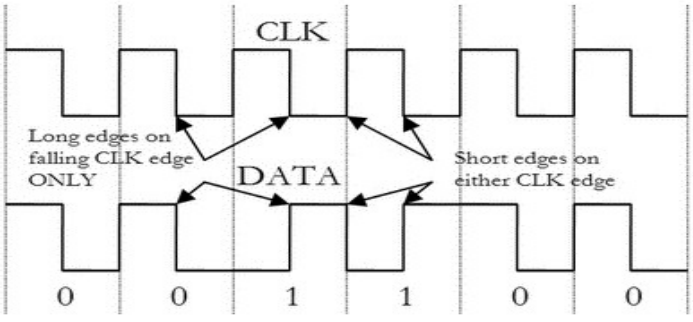


Figure 2: Diagram showing the same bit boundary edges (1-1 or 0-0) vs. different bit boundary edges (1-0 or 0-1). Notice the longer time between edges when the data changes from one level to the other[6]

If the timer value is one clock period or more (clock skew!), this means that it has counted the time between two bit centers with the current edge being the falling edge of the clock (Figure 2). So, the clock is fully recovered when the first transition between a 0 to 1 or a 1 to 0 is encountered and the bit boundaries can be determined to differentiate between edges between bits and edges in the middle of bits.

Synchronization

The receiver needs a way to break up the incoming data stream into bytes for processing. The boundaries between bytes can be recovered through the use of a synchronization stream. The transmitter sends 20 bytes of header (0xAA) for the AGC on the receiver to be set properly and to recover the clock, then it sends a five byte Manchester encoded sync stream of 0x00, 0xFF, 0xFF, 0xAA, 0xFF which we guarantee not to appear in the data at any point. So after receiving the sync stream, the receiver knows where the byte boundaries are located and splits the data stream accordingly. The received bits are shifted through a 5 byte register continuously and the receiver falls out of sync after each transmission to improve robustness. This helps because transmission delays or channel effects introduce clock skew into the edges which causes loss of sync anyway. The software attempts to handle this by labeling bytes that are too long or too short in width as junk bytes that are thrown away.

FCC Regulations

Our RF product needs to meet standards set by the Federal Communications Commission. Part 15 of the FCC guidelines requires that any device that radiates RF energy be tested for compliance [10]. The 433MHz band is one that is unlicensed, and since we use RadioTronix modules that have already been tested and approved by the FCC, our design satisfies this portion of the guidelines. The FCC guidelines detail fundamental power, harmonic levels, and allowed bandwidth for devices operating in our frequency band which are met by the RadioTronix modules [10]. However, for our implementation, we made sure not to transmit voice or video data and that all transmissions include ID codes (node #) to identify the appropriate recipients. In addition

to this, the FCC requires that transmissions cease within 5 seconds of activation which is done in our implementation [10].

Pyroelectric Infrared with Fresnel Lens

The pyroelectric sensor generates a surface electric charge on a crystalline material when exposed to infrared radiation. A FET device in the sensor detects the change in the amount of charge on the crystalline material. A filter window is added to the package to limit incoming radiation to that given off by humans. A typical circuit configuration is shown in Figure 3 where the amplifier is designed to reject high frequency noise and a MCU/Comparator is used to detect the positive and negative transitions of the output signal.

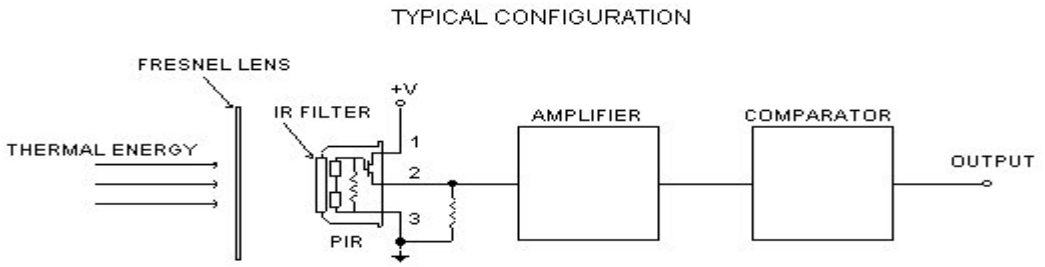


Figure 3: The typical setup for the Pyroelectric Infrared Sensor[1]

The figure below shows how the sensor uses two sensing elements to cancel signals caused by vibration, temperature changes and sunlight. Under typical operating conditions, a human passing in front of the sensor activates one element first, then the other. We use this same approach in the design of our people counter sensor using red LEDs. This is done to ensure that other IR sources are "cancelled" because they affect both elements at the same time. So a trigger condition can be set based upon a radiation source passing horizontally across the sensor array so that one element is affected by the IR source before the other. A Fresnel lens is used a focus in front of the sensor to decrease the effective field of view.

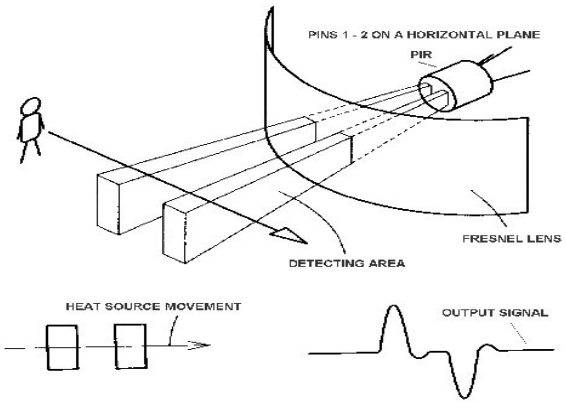


Figure 4: The output when a person walks across the detection field[1]

A Fresnel lens a convex lense collapsed on itself to form a flat lens that is not as thick, but still retains much of its original optical characteristics. The Glo-Lab Fresnel lens is made of an infrared transmitting material that is selective for sensitive to human body radiation. The lens has a focal length of 0.65 inches and a field of view of approximately 10 degrees.

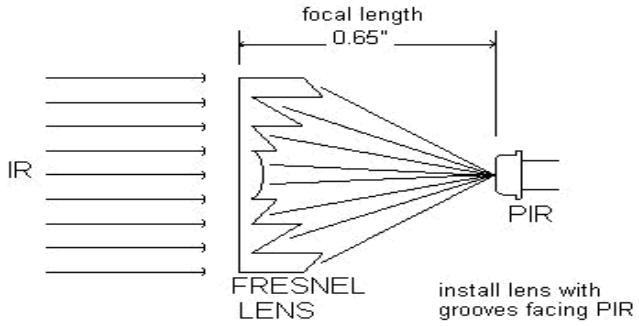


Figure 5: The Fresnel Lens has to be 0.65 inch from the sensor[1]

Evaluation of various sensor technologies

Sonar/Microphone

These types of sensors can be used to gauge the amount of sound activity in an area. However, this type of sensor could have a hard time differentiating from one loud person, and several people carrying on a conversation. Furthermore, in a museum environment, it is not likely that people will be talking very loudly or very much. This type of sensor technology would be hard to use in order to track the amount or "presence" of people in a given room since each person may behave differently acoustically.

Video

Cameras could be set up to capture video in each room and then software image processing could be done to count the number of people in the room. However, this solution is expensive and time intensive in terms of implementation.

Laser

This would require an emitter/detector pair across a hallway which would function by counting the number of times the beam is broken which each break in the beam counting as a person crossing through. This would require 2 beams at the entrance and exit of each hallway to sense the direction of movement (into or out of the room). If somebody blocks the detector pair on the opposite wall of the hallway, the count would not function properly.

Vibration/Pressure

These types of sensors could be placed on the ground and could be used to detect the presence of people in a given room and their movement within the room provided the sensors are spaced properly. However, it would be hard to wirelessly communicate this information to the base station since these types of sensors would be buried in the floor (which would also be expensive and time consuming to implement). Furthermore, these types of sensors would be hard to use to count the number of people entering or leaving a hallway if multiple people follow each other closely.

RFID

Radio frequency identification (RFID) is a way to remotely store and retrieve data using RFID transponders. These small RFID tags can be attached to a person and be either active or passive. Passive RFID lack their own power supply and instead use power generated by the incoming radio frequency signal and can get range up to 6m. Active RFID tags require their own power source and usually have longer ranges and are able to transmit/store more data than passive tags. Both devices are cheap and small and operate over a range of frequencies. The main problem with RFID is the lack of privacy

and the hassle of handing out and requiring the return of tagged badges. An RFID system was not pursued because of these concerns and limitations.

IR emitter/detector

The IR emitter is simply an LED which emits light with wavelength typically around 950nm. IR phototransistors act as variable current sources with the base voltage determined by the amount of light hitting the transistor and IR modules work such that their outputs are normally high and are set low when an IR signal at a specific frequency is detected. IR phototransistors are very sensitive to daylight and had a very small range of about 5cm, so both of these factors prevented its use in our project. The IR modules, while robust and capable of detecting signals farther than 3 m, require the transmitted signal to be frequency modulated. So to use this in terms of a people sensor would require us to set up a transmission link across a hallway which could be interrupted by a person walking through which we could detect. However, this is quite complex and unnecessary when simpler solutions are available. Another disadvantage is that the output of the modules is digital- either high or low output, so if we wanted to track velocity of movement through the field we would need an analog measure.

Overall Design

I. People Attractor

High Level Design

One of the ideas of getting the visitors more involved in a museum space is to attract people to a display/artifact that is less popular perhaps because of the location of the artifact. This is the motivation behind the birdie.

The idea of the original birdy is that a bird would sing when there is no human presence in front of an artifact for a period of time. The bird singing could get people's attention to seek the source of the bird sound, and thereby attract visitors to a particular piece of art. There should be no bird inside a museum environment, naturally visitors would be curious to find out where the bird sound is coming from. Once a visitor gets close to that artifact, motion is detected and the bird flies away (metaphorically).

The birdie is not only able to attract people with Thrushes or Orioles and others bird songs, but also records the rate at which people walk by the attractor. By using the data we can infer if people in the museum are being attracted by the birdie sound when it starts to sing.

The design was implemented with the Pyroelectric Infrared Sensor as the motion detection sensor. The signal from the sensor is sent to the MCU and the program decides which state to move to and when to playback the bird sound.

Hardware Design

We tried to make the Attractor as small as possible, so that it will fit in the recess area or a gap underneath the heater (see Appendix D.1) where it is not noticeable.

A dual element PIR sensor is used in this motion detection setup. The PIR's output signal is fed to Mega32's ADC input for analysis. If the voltage output from the PIR exceeds a threshold, which means motion has occurred, the 2 minute count down timer is reset. If there is no motion detected in 2 minutes, then the Mega32 chip triggers the voice chip to playback the pre-recorded bird sound.

The voice chip (Winbond ISD2560P) is set in loop mode, so the bird sound never runs out during the 10 minute period because it loops the 1 minute pre-recorded bird sound.

The user of the Attractor might want to test this device with different melodies. In accommodating that possibility, a recording button and a switch is incorporated into the design to allow the user to toggle between record or playback mode. The user can change the recording content by connecting a microphone using the 3.5mm standard microphone plug on the birdie. The user can easily get the output of a CD player to connect to the recording plug on the Attractor, and record the melody from the CD onto the voice chip. The Winbond Voice Coder, ISD2560P, is capable of recording and storing 1 minute of analog sound for playback.

Hardware Debugging

Much of the time was spent on getting the PIR to work and also building standalone birdies for real world usage. One problem we encountered was especially interesting. When the voice chip starts playing, it draws a substantial amount of power, thus the ADC would detect a voltage drop at the PIR reading, which gives a false alarm. The solution was to connect the whole device to its own power source with a 9V 500mA DC transformer with 5V voltage regulator.

Software Design

The Attractor will sing for 10 minutes in an attempt to draw attention from nearby visitors. However, if it fails to detect any motion in that 10 minute period, the birdie will go into sleep mode. The sleep mode is there to make sure the bird does not keep singing at night or when there are no visitors in the museum. The bird sound might be distracting to people who read books in the museum.

The program flow chart is shown below:

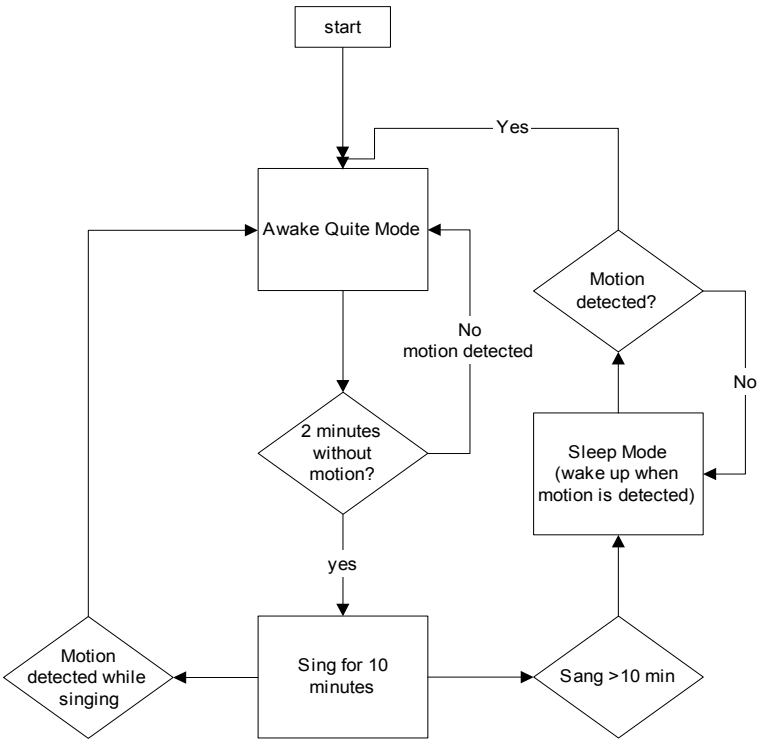


Figure 6: The program flow of the motion detector with built-in voice chip device or Birdie

The designed birdie sings when no motion is detected in a 2 minute period, then it starts singing for up to 10 minutes unless there is a motion detected (trigger event), which would stop the bird song.

The idea is that the user wants the capability to record the time that the Attractor is triggered, and from that data the user can infer when the Attractor is triggered and the actual time the bird is "singing." From this valuable data, we could get a feel for how the birdie is interacting with people.

Trigger Count Logging

We want to keep as much data as possible on the MCU. There are only 2K bytes of SRAM, which means the biggest unsigned char array we could have is about 2000 elements. However, because we have other variables declared as well as space requirements for program execution, we are effectively limited to an array of around 1500 bytes or 1500 elements of unsigned char elements. The number of trigger events is tallied every 30 seconds and is stored in this array.

The best way to save the most data is to compress 1 minute of data into 8 bytes, the first 4 bits is the first 30 seconds of a minute and the second 4 bits is the second 30 seconds of a minute. In other words, the number of trigger events that occur in the first 30 seconds of a given minute could be obtained via bit-masking the appropriate elements. For example, (element[i] & 0xF0) could be used to determine the data for the first 30 seconds while (element[i] & 0x0F) would reveal the data for the final 30 second time period. Using 4 bits to store the data, means we can only count up to 16 trigger events during each 30 second time period, however for our purposes this is more than enough and it is reasonable to assume that more than 16 triggers in 30 seconds is an unlikely event unless the museum space is extremely busy.

```
Starting time: 123 16:00
15 00 | 00 04 | 08 10 | 02 09 | 01 04 |
01 02 | 00 00 | 00 00 | 00 00 | 00 00 |
01 00 | 00 00 | 02 01 | 00 03 | 00 00 |
00 00 | 02 04 | 00 00 | 00 00 | 00 00 |
00 00 | 00 00 | 00 00 | 00 00 | 00 00 |
00 00 | 00 00 | 00 00 | 00 00 | 00 00 |
07 05 | 04 00 | 00 00 | 00 02 | 00 00 |
00 05 | 08 00 | 00 00 | 02 00 | 00 02 |
04 00 | 00 00 | 02 00 | 00 02 | 03 02 |
00 00 | 03 00 | 01 03 | 00 00 | 00 00 |
00 00 | 00 00 | 00 00 | 00 00 | 00 01 |
00 00 | 00 00 | 00 00 | 00 00 | 00 00 |
```

Figure 7: Triggering count data from birdie

Figure 7 shows data for one hour detailing the number of trigger events during each 30 second time period. The first row represents 5 minutes worth of triggering data (the 12 lines represent the data for 1 hour broken down into 5 minute sections), the first part of the data pair represents the first 30 seconds, and the second half is the number of trigger events during the last half of a given minute.

Serial Connection Menu

The user is able to obtain the number of trigger events, or trigger count, from the serial port via hyperterm. When the user presses enter they would see the following menu:

```
Birdy Manager Menu
--------------------------
Status: Idle
History Capacity: 3 Days (based on the starting and end time)
Options:
1. Dump all history since day: 123 hour: 10
2. Clear all the history since day:123 hour:10
3. Toggle the recording status
4. Set Start Recording Time - current start time = 10
5. Set End Recording Time - current end time = 17
6. Set time and day
```

Figure 8: Birdy Manager Menu. Option 6, the system time has to be set when the power is interrupted.

The options are self explanatory. The user must set the system time with option 6 when the power is interrupted. The start recording time (option 4) and end recording(option 5) time are stored in EEPROM because they are very unlikely to be changed on a daily basis.

Software Debugging

The software debugging is done by recording fake trigger counts and fast-forwarding the system time on the birdie by changing the system time on the birdies to 30 seconds jumps for every 1 second. So we could simulate three days worth of recording in less than 1 hour.

A functionality test was also carried out to make sure the birdies perform according to the specifications. We also let all 4 birdies run for three days in the HCI lab before it is deployed in the Johnson Museum.

II. Tempo Indicator

High Level Design

The virtual display shown in the appendix section requires a simple tempo indicator to change from rainy to sunny weather and the hue is adjusted based on the number of people in front of the scroll and how fast they are moving. The tempo detector reports how fast people are moving in front of the scroll and this data is used to control a cloudscape virtual display that is computer generated and projected onto the scroll. The idea is that people can move in front of the scroll and watch the display change based on fast they are moving across it.

The hue of the cloudscape would change from green to clear to red based on the fast tempo detected by the tempo detector. The fast tempo would also make the cloudscape change the weather from sunny to rainy. These effects are done by Eugene Medynskiy (HCI Group) in Macromedia Director with the tempo indicator connected to the serial port of the computer. There is a Java program (Appendix E) that reads the voltage level sent back from the indicator, and sends the appropriate command to the Macromedia Director software. The different weather cloudscapes are shown in Appendix C.

Hardware Design

By experimenting with the output voltage from the PIR, we found out that the PIR is less sensitive to fast motion but very sensitive to slow motion. A slow motion is equivalent to the walking speed in a museum environment; it's usually much slower than walking speed on a sidewalk because people are engaged in observing the art pieces. The output voltage swing is large when an infrared source (a hand, arm, or human walk across) is moving slowly in front of the PIR. By knowing this, we have set a threshold for each voltage that the ADC reports in a 10 second interval. In Figure 9, it shows two different kinds of motion – slow and fast. The slow motion would cross the slow threshold; therefore, the slow counter would increase. The software handles how each motion is distinguished if multiple thresholds are crossed.

Software Design

The tempo data is sent to a computer through RS-232 serial port. The data could be read via HyperTerm for testing purposes. The data reporting format is as follows:

SlowL x MidL x FastL x FastR x MidR x SlowR x.

Each "x" corresponds to the number of the ADC value that falls in each of the region indicated in Figure 9.

For example, a fast motion, e.g. some one running in the museum, in front of the PIR would be reported as: SlowL 0 MidL 0 FastL 3 FastR 4 MidR 1 SlowR 0. The voltage is sampled every 100ms, and the data is reported every 5 seconds. It might not give all non-zero count for mid and slow speed, it all depends on the speed of the person that is walking. When there is a person standing in front of the device, it would register a slow motion. When ever the slow counter is not 0, it is almost certain that there is someone standing in front or walking by slowly. Note that when the voltage falls below the fast threshold it is counted as no motion detected to avoid false alarms.

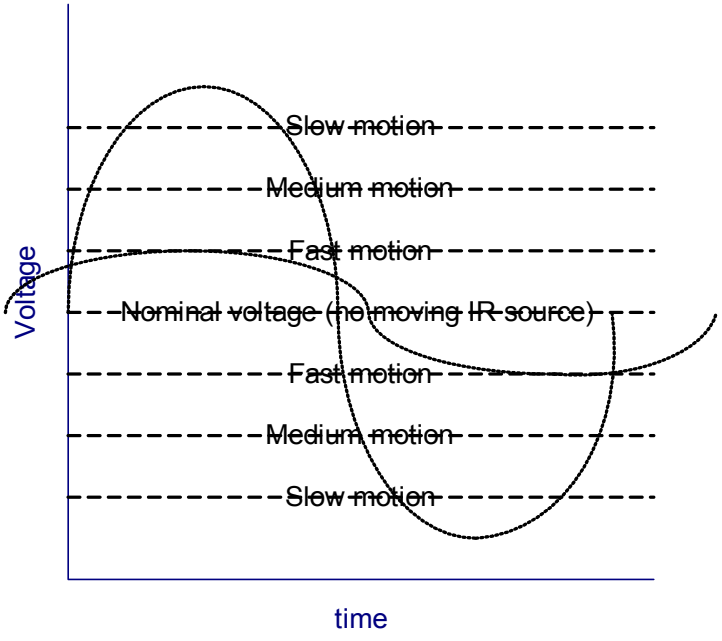


Figure 9: The tempo detector output region break down.

A better solution for future improvements is to check the slope of the voltage readings. If it was a positive slope, then we know the person has crossed the left element and we can also figure out the peak at which the voltage reaches, and use that as a measure for how fast the infrared source is moving.

A Java program was written to read input from the serial port. The data is parsed and processed to send keystrokes to the Macromedia Director program. The CloudControl program only reads in the fast movement frequency and converts that into keystrokes. The Java program uses the javax.comm communication API package released by Sun Microsystems. The java code is listed in Appendix E and the effects that Macromedia Director provides is in Appendix C . The cloudscape is done in Quicktime format and imported into Macromedia Director to have different effect, such as rain drops and changing hue, as well as making the transitions smooth from one cloudscape to another. The Director Projector in this prototype setup plays the cloudscape movie and also produces rainy and hue effects. The controls are A and D which alternate between rain and no rain, tapping Z repeatedly will change the hue from green to clear to red. If you stop hitting Z, it will slowly fade back to green.

Software Debugging

The software debugging was simple for this application, since there is no complex state machine. The most basic testing is carried out by walking or moving in front of the sensor and checking if the reading is as expected.

III. Wireless Networked People Counter

High Level Design

The museum wants a non-invasive sensor technology to measure and relay information regarding the number of people in a given space to a central location. The idea is to deploy these sensor nodes in all the entrances and exits of a given room. The sensor must be able to distinguish between people entering the room and people leaving the room. The networking component in this part of the project is needed to allow the two way communication for base station to discover and poll nodes for data. The design needs to be geared towards making it extensible if future nodes are necessary or if other types of sensor data which to be recorded and relayed. To this effect, a communication protocol will be needed to handle multiple transmissions which may cause interference among nodes since they share the same 433MHz frequency band. Furthermore, at the base station, the user should be given the option to discover, poll, and reset nodes in the network.

The sensor network is based on a star topology network. The base station is being the center commander who sends out commends, while the nodes only respond when the base station is requesting information from it. The figure below shows the sample network topology of our network.

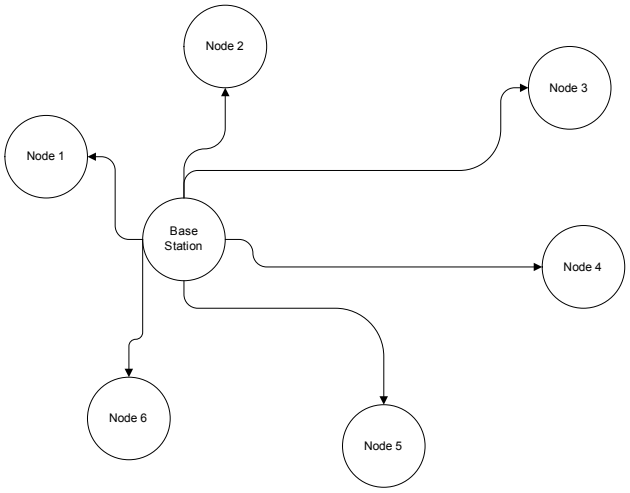


Figure 10: A start topology network example

Hardware Design

The hardware design can be broken down into two sections: the RF link consisting of the receiver and transmitter and the people counter circuit. The Radiotronix modules were chosen as our receiver and transmitter due to their low cost, reliable performance, and previous use in ECE 476. The LEDs were chosen as the sensing elements due to their range, low cost, and ease of implementation.

RF Transceiver Circuit

The Radiotronix receiver and transmitter are hooked up as specified in the data sheet (see schematics). Each node is a transceiver, operating with 1 receiver, and 1 transmitter, with each having its own antenna.

People Sensor Circuit

The schematic of the sensing elements is in Appendix B. This circuit consists of 2 clear red LEDs hooked up to two op-amps. The op-amps provide a variable gain of Rp/Rd through the potentiometer, Rp, which varies from 1Ω to 10kΩ and pull-down resistor, Rd = 47Ω. A variable gain is needed for our application due to the varying lighting conditions that are possible. For example, in the lab, the lighting is good, but the environment is darker than the museum or HCI lab where there is a strong presence of bright, ambient, white light. So in the lab environment, a higher gain is needed to amplify the signals since the light levels are so low which lead to smaller voltages across the LED. However, in the HCI lab or museum environment, a lower gain is needed since the strong presence of ambient light causes the output of the op-amp to rail very easily. Testing was done hooking up the scope and varying the potentiometer until a nominal voltage reading of about 2V to 2.5V was given at the op-amp output. This allows the op-amp output to swing to either rail (0V or 5V) depending on the person and the color of clothing they wear as they pass across the LED fields.

A 1uF capacitor is placed on the output node to the inverting input to shunt the high frequency signals to ground. This was needed because initially we were seeing a high frequency ringing on our output which caused invalid ADC readings. The circuit works by amplifying the small (1-2mV) change on the LED as someone walks by into a 0.5V-1V swing on the op-amp output. The output of the op-amp is fed to one of the

ADC inputs (PORTA.0 or A.1); The LEDs require a 10MΩ resistor in parallel in order to act like an ideal current source and prevent the output of the op-amp from railing under normal conditions. We had problems with the LM358 op-amp which had problems amplifying signal levels close to ground, but these problems were resolved by using the LMC7111, but this requires each LED to have its own op-amp since the 7111 has only one op-amp per DIP.

Overall, the circuit functions very well and shows a significant change in voltage as someone passes through either LED field from a distance of 10-15 feet away. However, the op-amp output of the circuit is susceptible to railing if the LED is pointed at direct or very bright light at which point, it still shows a drop in voltage when somebody walks across it, but the magnitude of the drop is not as large and is not enough to trigger a reading in our state machine.

ADC Sampling Frequency

We spaced our two LED sensors by approximately 1 inch. Given that this is to be deployed in a museum where people walk more slowly to observe the exhibits, if we take readings too often, the person might not have walked through both LED fields and the counter would miss this movement. Similarly, if we take only a few readings over a given time period, the person might have completely moved through both fields and again the counter would miss the movement. Our initial calculations showed that we would need to wait 0.0189 seconds between measurements (time = speed/distance) at our spacing of 1 inch and assuming people walk at 3 miles/hour. So our initial state machine took readings every 20ms, but we saw this was not fast enough and that the count was not accurate for fast movements. So we changed the state machine to take readings every 10ms, but this was also inaccurate because it failed to count slow movements through the field. Our robust solution was to sample every 10ms, and to modify our state machine to look for a period of 4 consecutive triggers (40ms) which accurately measures a person walking through both fields. This solution handled a wide range of walking speeds from very fast to very slow.

Hardware Debugging

For the breadboard and solderboard devices, check power and ground connections on each op-amp that the LED is connected to. Measure the voltage across the LED terminals to see it is changing as you vary the lighting conditions to see if the LED is fried or not. If the LED appears functional, but the op-amp output is railing at VDD, try covering the LED output completely to see if the output changes. If it does not, there might be a mistake in the wiring or soldering, so check the gain of the op-amp and feedback circuitry. If significant ringing appears in the op-amp output, increase the feedback capacitance or decrease the feedback resistance to reduce the overall gain. By connecting the PORTC pins on the STK-500 to the on-board LED port of the STK-500, this board can then be used to debug the overall people movement operation of the sensors as detailed in the attached code in node.c

Debugging the RF part requires attaching a scope to the analog output of the receiver on both the node and the base station. The pictures in the Appendix B show the waveforms you can expect to receive and how the signal strength varies. To establish a strong RF link, try for line of sight between the base station and node and try not to stand too close to either antenna as that "detunes" the antenna and corresponding received signal strength at each node.

Packaging

Our current implementation of the base station and nodes are on solder boards. Each node contains 1 MCU, 1 Transmitter, and 1 Receiver while the base station has the same components as well as an RS-232 connector. We place this inside a small box and we poked holes for each antenna while attaching a sheet of aluminum foil inside the box to act as a ground plane. Similarly, there are holes through which the AC adaptor provides power to the module. In the front of the box, two holes are cut for placement of the LED sensors.

Software Design

There are two parts to the software design for the wireless people counter – the base station code and the node code. The base station sends discover, poll and reset

packets to each of the node to gather data. Each node responds to the base station packet by sending the corresponding respond. Software debugging can be done by utilizing the USART interrupt and the appropriate printf commands in the C files which are commented out for clarity.

Base Station

The base station is hard-coded with the possible maximum number of nodes in the network. The base station polls each node on this list, waits for a response, times out and tries again 3 times. After that, the node is declared dead and the base station moves on to the next node. After this process of discovering the alive nodes in the network, the base station is ready to poll each alive node for people counting information. The third process of the base station involves resetting the count of each node (at the end of the day, for example).

Node

Each node is hard-coded with a specific number in software. This is done to prevent interference among the nodes when the base station requests a count. For example, when the base station sends a request, it is received and decoded by all nodes, but only the node it is intended for will send a response ensuring that there is no interference among nodes while transmitting. This establishes a 1 to 1 link between the base station and a given node at any point in time.

Transmission Protocol

The protocol we designed is extensible for this small scale network. Our design was motivated by minimized the amount of work the node should do since it takes and ADC reading every 10ms to record the number of people that pass its field of view.

Each packet consists of 4 bytes as shown in the figure below. In each byte of the packet, the very first bit is always the parity bit. Odd parity is used – If the byte has an even numbers of 1, the parity bit will be 1, otherwise, it will be a 0. Because of the parity bit, the actual data that each byte can contain is just 7 bits.

Packet Type	Node Number	Data 1	Data 2

A list of possible Packet Types

1. DSC_PKT: Discovery Packet
2. CNT_PKT: Counting Packet
3. RST_PKT: Reset Packet
4. ACK_DSC_PKT: Discovery Packet Acknowledgment
5. ACK_CNT_PKT: Counting Packet Acknowledgment
6. ACK_RST_PKT: Reset Packet Acknowledgment

The first three types are used when the base station communicates with the nodes. The nodes would reply an acknowledgement packet based on the type of packet it gets, which is the last three types of packets. In other words, the base station can never send an ACK_DSK_PKT nor can the node ever send a DSC_PKT.

The Node Number field would be the node number that the packet is destined for when the base station is sending to the node. When node is sending to the base station, it indicates which node the packet is coming from. This implies that the only communication link is between the base station and a specific node; the nodes can not communicate with each other in version. However, this could easily be added into the protocol by separating the node number field into source and destination fields.

Data 1 and Data 2 are 7 bits each, since the first bit in each byte is the parity bit. The two data bytes are combined in software which means we can have 14 bits worth of data which is enough for our purpose. The most important piece of data that we transfer is the counting packet which contains the number of people that have crossed the node's field of view. 14 bits gives a data range of has a range of [-8191, 8192]. Negative numbers are used to signify movement out of a given room, while a positive count is used to signify movement into a room. Each transmission consists of sending the given packet 4 times so that a best match scheme could be implemented if necessary to handle corruptions in the data.

Serial Connection Menu

The user is able to execute the three most basic functions that base station provides, namely, discover nodes, get current people count statistics, and reset all the ndoes, from the serial port via hyperterm. When the user presses enter in the hyperterm, the following menu is shown:

```
People Counter Basestation
Number of Alive Nodes: 3
Options:
1. Discover Nodes
2. Get Current People Count from All Nodes
3. Reset All Nodes' Count to Zero
4. Copyright Information
```

Figure 11: The People Counter Basestation Manager

Common Functions in both Base station and Node

Receiver Interrupt

Any edge (rising or falling) triggers the external interrupt and runs the receiver interrupt code. Timer Counter 1 is used to measure the time between successive edges. This timer is used to aid in the synchronization and recovery of the data stream. It runs at the same clock rate as Timer Counter 0 which means that edges should appear at 256us for same bit boundaries and 512us for opposing bit boundaries. We factored in clock skew and channel losses so TCNT1 looks for opposing bit boundaries less than 680us and same bit boundaries below 360us, but above 40us. Data is sampled on the rising clock edge which could correspond to a logic low or logic high level at the receiver input (depending on whether we are sending a 0 or a 1). The data is shifted into a receiving stream to be stored/decoded later and the bit number is changed accordingly. This interrupt scans the receiving stream for the synchronization sequence at which point it allows the receiving stream to be stored into the final received data array.

Transmitter Interrupt

The Timer0 Compare Interrupt is used to transmit information by toggling the PORTB.0 pin which is connected to the Radiotronix transmitter. The output compare register, OCR0, is set to 64; Timer0 is set to CTC mode (clear timer on compare match) via the TCCR0 register by setting the WGM01 bit to 1. The clock divisor factor is set to

64, meaning the counter tied to timer0 is incremented every 1/(CLK/64) seconds = 4us. So the counter counts to 64 (which takes 4us *64 = 256us), then the Timer0 Compare Interrupt is triggered where the transmitting code is located. So this interrupt is taken every 256us. However, the transmitter output is toggled only if the sendPacket flag is set. We use a variable called halfcycle to determine when the transmitter should move on to the next byte. Since we are using Manchester encoding, 1 bit is sent by the transition from either high to low or low to high which requires the Timer0 interrupt to be taken twice (once to set the initial logic level, and the second time to change the logic level). So when halfcycle reaches 16, that means we have transmitted 8 bits since halfcycle is incremented with each transmitted edge. The actual toggling of PORTB.0 is determined by shifting the byte to get the appropriate bit and using masking and inverting the bit if necessary to get the right logic levels. For example, on the falling edge of the clock, the transmitter output is driven directly by the bit to be transmitted. However, on the rising edge of the clock, the transmitter output is driven by the inverse of the bit to be transmitted.

Parity Generator/Checker

As we mentioned in the transport protocol section of the report, we have a parity bit for every byte of data we send. The most significant bit is the parity bit. When the packet is form, we add a parity bit to every byte before the data is transmitted. The parity bit is checked against Parity (0x7F & DataByte) at the receiver end. Odd parity is used to generate the parity bit. If (0x7F & DataByte) has even numbers of 1, then the party bit is 1, otherwise it is 0.

Packet Creation

We transmit a 4 byte packet from node to base or base to node. The format of the packet is given earlier. The base station has functions create_dsc_packet, create_cnt_packet, and create_rst_packet which simply set the appropriate bytes in the sending array which the transmitter interrupt traverses when a packet is to be sent. Similarly, on the node there are functions create_dsc_ack_packet, create_cnt_ack_packet,

and create_rst_ack_packet which set the appropriate bytes (according to our protocol and definitions) in the sending array, send_packet[].

Node Specific Functions

ADCReading: This function sets the appropriate bits in the ADMUX and ADSCRA registers to request and record ADC readings from channel 1 and then channel 2 where channel 1 and channel 2 are connected to LED1 and LED2 through ports A.0 and A.1

ADCStateMachine: See the next section state machine description and Figure 14.

ADCCompare: This function compares the current reading (in LED_ADC[x]) to the values in LED_BASE_ADC[x] to see if a deviation in either direction of LED_DELTA_ADC[x] has occurred indicating motion at which point, this function sets the appropriate bits in LED_TRIG[x] (where x is 0 or 1 depending on LED1 or LED2). In debugging mode, this function toggles the STK-500 LEDs via PORTC.0 and C.1

ADCInitAvg: This function gets the initial reading off both LEDs. It requests ADC_AVERAGES readings delayed by 2ms each and averages the results into a base led reading for both leds from 0 (darkest) to 255 (brightest). It also sets the sensitivity level of each LED. For example, if a base reading is 120 and sensitivity is set to 10 percent, then the LED_DELTA value is 120* 0.1 = 12, so if later on, the reading from the LEDs deviate by 12 in either direction, that sets a trigger (LED_TRIG) which the state machine will use to determine if a movement has taken place.

Program Flow Charts and ADC State Machine

The base station flow chart is shown below in Figure 12. The base station is powered on and enters the discovery state. Initially, the base station polls each node in the system to determine which ones are “alive” and which ones are “dead”. The total number of nodes in the system is hard coded in software and can easily be changed. “Alive” nodes are counted as nodes that receive the base station’s request and send a reply that is received and successfully decoded by the base station. For example, the base

station sends a discovery packet requesting an ACK from node 1 in the system. All nodes receive this transmission and decode it, but only node 1 sends a reply because it sees that the node number of the packet matches its own hard coded node number. The base station receives and decodes the DSC_ACK_PKT from node 1 and sets its internal status array to indicate that node 1 is alive and should be polled for count info later on. The base station polls each node 3 times and waits for a response before declaring the node dead. The base station then moves to an idle state where it waits for either user intervention or a pre-determined time interval to elapse before either polling each node for data or resetting the count at each node. These time intervals can be modified in the software and are in place to avoid constant polling to meet FCC regulations for unlicensed wireless transmission [10].

The base station polls each node that is alive for data by sending out a CNT_PKT and storing the data sent back by each node via the CNT_ACK_PKT locally in its own array. After a set time interval, the base station sends out a RST_PKT to each node to set its internal count to 0 (at the end of the day for example) and awaits the node's reply of ACK_RST_PKT to acknowledge that this has happened

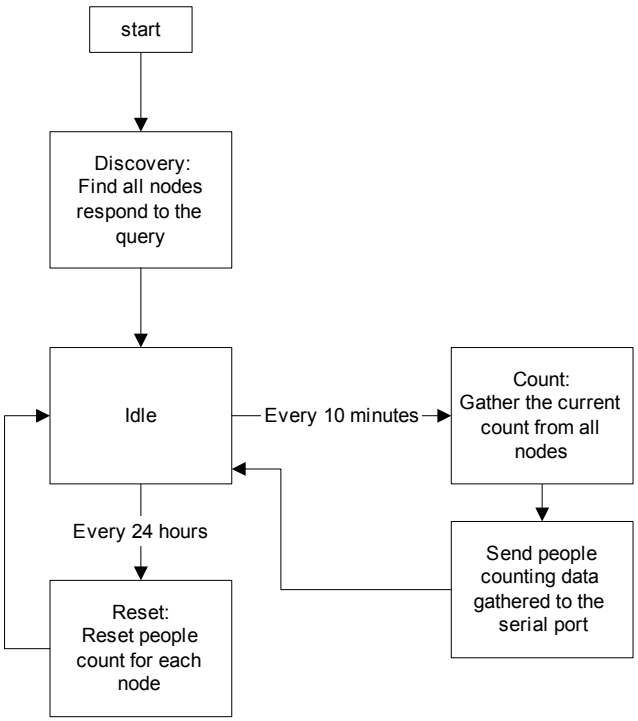


Figure 12: Base Station Program Flow Chart

The program flow chart for the node is shown in Figure 13 below. The node is designed as a passive component that waits for the base station to initiate contact before sending out a reply. The node is always in a default waiting state where it is gathering data from the LED sensors regarding the number of people that have crossed its field of view. When a packet is received and decoded, the node checks to see if it is the intended recipient of the packet and it checks the parity bits in each byte (packet integrity check) to see if the data was corrupted or not. If the packet is not corrupted and the node is the intended recipient, it sends out an ACK packet depending on the packet that was received. For example, if a DSC_PKT is received, the node sends a DSC_ACK_PKT back to the base station. The node does not send multiple packets to the base station, if the node's reply is corrupted, it simply waits for the base station to request the data again and then the node sends out a response. This greatly simplifies the code and protocol and helps establish reliable communication links between the base station and each node. After the node sends its reply, it then goes back to waiting and continues taking readings off the ADC to count the people that cross its field of view.

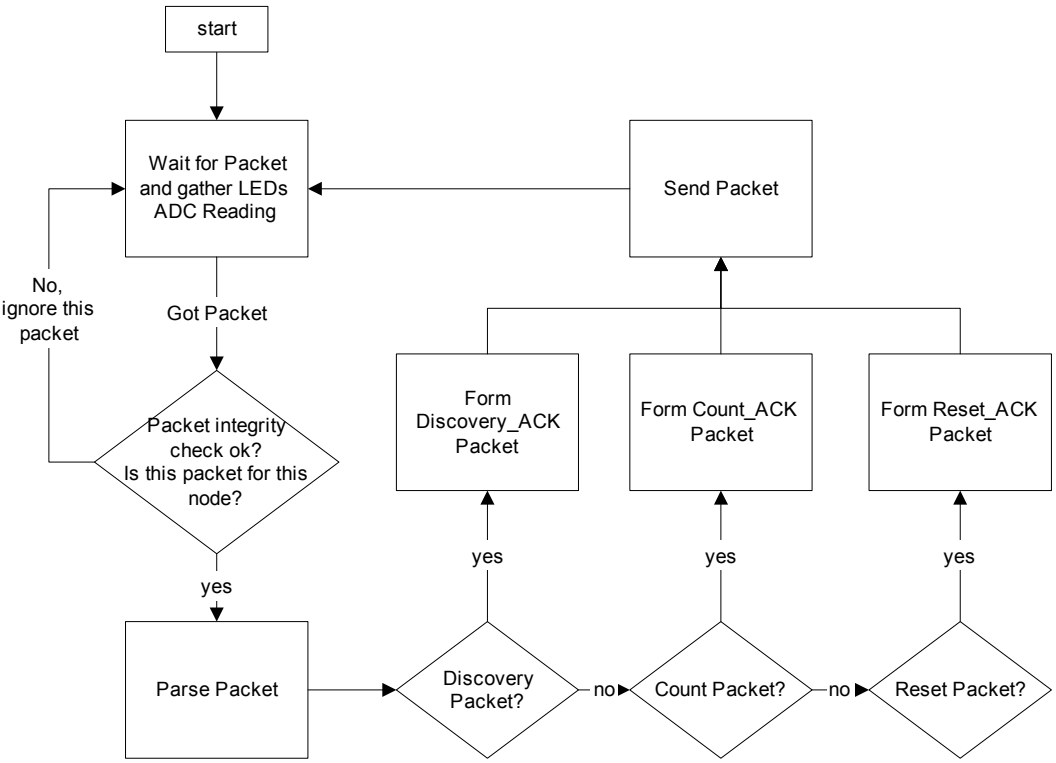


Figure 13: Node Program Flow Chart

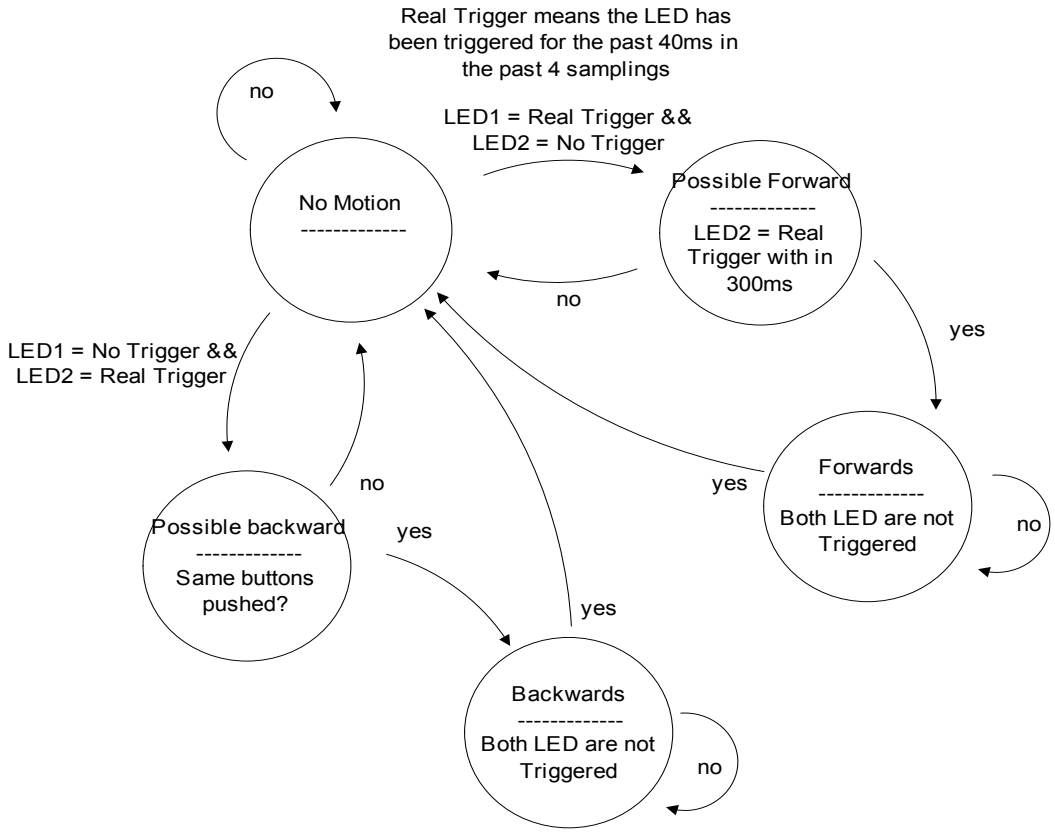


Figure 14: LED State Machine

The LED State Machine Diagram above is implemented on each node. The no motion state is the default state. The 2 LEDs on each node are used to establish forward and backwards motion depending on which LED is triggered first as a person passes through both fields of view. ADC readings tied to each LED are taken every 10ms. A trigger is taken to be when the LED reading deviates from the base LED reading by a certain percentage set in software via the sensitivity multiplier. For example, if the base reading is 150 ADC units and the sensitivity multiplier is 0.1 or 10 %, then this LED is considered triggered if the reading falls below 140 ADC units or rises above 160 ADC units indicating movement across this LED's field of view. The Real Trigger condition is for each LED and is meant to track movement over the entire field of view which is why it is done by looking at the past 4 readings (in essence making sure the LED is triggered for 40 ms).

Movement into the Possible Forward or Possible Backward state happens when one LED is real triggered while the other is not. In these intermediate states, the other LED has to be real triggered within 300ms. This is done in order to handle the case where somebody walks into the field of one LED, and then backs out without crossing the field of the other LED. In other words, this handles the case where somebody thinks about entering the room, but backs out instead. So a person is effectively counted when one LED is real triggered before the other within a time span of 300ms. This is done to account for even the slowest of museum visitors. To prevent one person from triggering each LED multiple times (i.e. counting one person more than once as they walk across both fields of view) the state machine is kept in the Forwards or Backwards state until the person completely leaves each LED's field of view.

Future Improvements

There are many improvements can be made to this people counting sensor network. Due to time and budget constraint, some of the tasks were unable to complete. Here is a list of possible improvements and the importance of each:

PCB

By putting all the elements on to a single PCB boards would make the whole system, especially the wireless transmitters and receivers, more reliable. However, the cost of using SMD components is a lot higher than using prototype boards. We think this is the single most important prerequisite for doing wireless RF communication. PCB would provide a stable and controlled environment for the transmitter and receiver. Every wire we put on the protoboard will absorb the transmission energy and changes the RF environment. PCB would also shrink the size of the overall design which is important for our application.

RF Switch and Single Antenna

When two antennas are present in a system, they interfere with each other. Performance can be improved if a single antenna is used for both the transmitter and receiver by putting a RF switch into the design.

Loop Antennas

Loop Antennas have better resistance to human or proximity interference than the 1/4 wave whip antenna currently deployed. However, they have a shorter range than the whip antenna, but loop antennas would make the people counter smaller by eliminating the large, visible, ¼ whip antennas. The antenna we suggest is the Linx Technologies Splatch surface mount antenna[11].

Single Dual OpAmp

Single Dual OpAmp should be used instead of two single OpAmps that we are currently using for the sensor element. A better OpAmp with substantially faster slew rate should also be used for better performance.

T-pad on the Transmitter antenna path

From talking to a senior member in the Radiotronix Forum, he highly recommends using a T-Pad for the antenna circuit. The T-pad helps to isolate the impedance of the transmitter's output from the antenna [3]. This will also attenuate the signal by 3db. The AS modules are very sensitive to antenna matching. If the VSWR is > 2, the module may not oscillate properly, giving the appearance of a dead module. You might not need it if your antenna is a good one. It is also a good idea to AC couple the antenna [3]. Furthermore, these cheap modules tie the antenna directly to the oscillator whereas more expensive modules isolate these elements for better performance.

Battery Powered Nodes

It is important to make the node is as small as possible as it will be deployed in a museum setting. When a battery is used, a better designed voltage regulator should also be used. The current design is desirable for short term prototype development, but it is not suitable for battery powered devices, as the voltage regulator itself draws too much power.

Sound Level Sensor and other technologies

A microphone could be attached to every people counting node to obtain a sense of sonar activity (footsteps, conservations, etc) in addition to our current measurement of people count.

Narrow Beam LEDs

Currently, the LEDs used in our setup have a 23 degrees viewing angle. A narrower viewing angle LED would work better, since the field of view of the two LEDs won't cross as much when they are placed 1 inch apart.

Node Number Indicating Switch

A 6 bit DIP switch should be added to hardwire each node's number rather than our current implementation which does it in software. A DIP switch should be connected

to one of the pulled-up port pins on Mega32 and another end to GND, so the node ID can be changed anytime.

Intelligent Adaptive base reading average adjustment

Each LED takes a base ADC reading every morning at 10am when the museum opens. The base reading is then used for comparison when there is any lighting change. When there is a drastic ADC reading change, it must mean that a person is walking past the people counter. However, the lighting condition changes throughout the day, so we have to adjust the base reading periodically.

We need to update the base reading every 10 minutes for accurate comparison. We can not just blindly take 20 ADC readings and average them to use as the base reading, since there might be a person walking by the people counter in 5 of those ADC readings. The proposed way of doing this is to have a threshold from the previous base reading. If the current ADC base reading is less than 2% (should be fine tuned to look for the optimal percentage) different from the previous base reading then we update the previous base reading to the current base reading. There is another caveat, we should only update if both LEDs current ADC readings are within this threshold. We have wrote a method in the node.c called ADCAverageUpdate(). It's the general algorithm and implementation of the algorithm described above, however, it is not currently being used by the nodes.

Results

Four Birdies have been installed in Hubert Johnson Museum at Cornell University [5]. They are installed in the recesses as shown in Figure d1 in Appendix D. The visitors and security guards at the museum found them to be very interesting. The data has been collected from these birdies, but has not been analyzed by HCI. Kirsten Boehner (HCI Group) created questionnaires to understand the impact of these birdies on the movement of visitors in the museum.

One of the tempo detectors was used in Herbert Johnson Museum as well. The device was connected to a laptop to control the Macromedia Director program used to make transition between different effects on the cloudscape. The whole setup was shown to Kirsten's communication class at the museum. Feedback from the students about the virtual art was gathered.

The wireless networked people counter was built on a small scale (base station + 2 nodes) and tested under this implementation and worked fine. Each node keeps an accurate count of people movement across its field of view and relays the information wirelessly to the base station where the user can interact with the system. The user is able to discover nodes, poll nodes, and reset nodes from the base station.

The LED sensor was tested thoroughly at a varying range of walking speeds from very slow (less than 1 mile per hour) to very fast (about 7 miles per hour) and it maintained an accurate count of people movements. However, under the current setup, the LED sensor and corresponding state machine are unable to distinguish two simultaneous movements in opposite directions across the sensors. For example, if two people are crossing the fields of view, with one person entering the given space, while the other is exiting results in varying data where neither is counted, or one or both are counted in the same direction. This can be remedied by mounting the sensors overhead the given hallway or entrance so that two different streams of movement can be distinguished which allows for the two people entering and exiting at the same time to be counted accurately.

Also, our LED sensors work by detecting the amount of light that is reflected off a person as they walk through each sensor's field of view. If the sensors are mounted horizontally, there are some considerations to take into account. First is the height of the

mounting. If the sensors are mounted at foot level, each person would be counted twice as they moved through the sensor since each leg would essentially count as a different person. If the sensors are mounted too high, no movement would be registered as the person walks through. Given that a museum audience consists of a varying range of people with different heights, we suggest a mounting distance of approximately 3 feet which would be triggered as toddlers walk through and this height would also pick up the upper torso or waist of taller individuals.

Our testing was done by mounting the sensors horizontally to the movement direction at a height of approximately 3-4 feet above the ground. Another implementation would be to do as recommended above and mount the sensors on the ceiling overhead so that they point downward and would pick up lighting conditions reflected off the person's head or hair. Since the LED is picking up changes in lighting conditions, a problem might occur when a person is wearing a shirt that matches the background closely. However, we did not see this in our testing where we used a bright white background and walked across the sensors field of view wearing white shirts.

On the wireless side, we had some problems establishing a clean, long range RF link. At very close distances (less than 10 feet), there were no signal integrity issues in the transmitted and received signal, and the network is able to function properly. As we increased the distances, we have to deal with more channel effects (multipath, interference, etc) which degrade the quality of the signal. To deal with this, we used ground planes on the receiver and transmitter antennas and tried to keep the wire lengths short on the solder board to minimize RF emissions. In addition, we tried to shield the circuitry from the antennas and we used decoupling capacitors on the power supply. Despite our best efforts, the range is highly variable where at some points we could get reliable communication up to 70 feet away, while at other times we could only get reliable links at distances of 20-30 feet. We feel these signal integrity issues can be resolved by moving the design completely to PCB through the use of an RF switch and Linx Technologies Splatch antenna which would also significantly reduce the size of the design to allow it to be able to be used in the museum environment [11].

Conclusion

The Birdie and the Tempo Indicator met the design requirements. They were successfully deployed in the Herbert Johnson Museum. Further research will be done by interviewing individuals who notice the bird sounds in the museum.

The wireless people counter met the design requirements given to us, but the size needs to be significantly reduced before it can be used in the museum environment. We have established a proof of concept for this through our design and implementation of a small scale network. The size issues can be resolved by migrating the design to PCB which would compact the design and then allow it be housed in a device only moderately bigger than a cell phone, rather than its current packaging of a small box.

The LED sensors work very well and they achieved our goal of great performance at low cost. They provide a very accurate count of people movement and their range is far greater than necessary for our application. The software protocol we developed to handle the wireless communication is extremely robust and extensible. New nodes and functionality can be added easily and the polling/reset schemes can be modified by the user if they wish. The transmission protocol we developed allows us to send and receive signals reliably. Furthermore, we have built a great deal of automation into both the base station and node so that the end user can simply grab the results with little interaction. The wireless RF link between the base station and nodes suffers from a lack of reliability at long ranges, but functions very well within a medium range of operation. If we had more time, we would move the design to PCB to resolve the range and size issues in our implementation while improving the level of performance in the RF link. We could not initially have started with a PCB design due to budget constraints so we showed proof of concept by building a small scale network consisting of a base station and two nodes on solderboards.

Acknowledgements

We would like to thank Professor Bruce Land, for his guidance throughout this entire project. Additional thanks goes out to Kirsten Boehner and Eugene Medynskiy for providing us with specifications for the birdies and tempo indicator and creative inspiration.

References

1. Pyroelectic Infrared Sensor datasheet <http://www.glolab.com/pirparts/infrared.html>

2. Winbond Voice Playback Chip Datasheet ISD2560P <http://www.winbond.com/e-winbondhtm/partner/PDFresult.asp?Pname=919>

3. Radiotronix 433Mhz Receiver/Transmitter Datasheets <http://www.radiotronix.com/support/click_download.asp?contentid=146>

4. Radiotronix 433Mhz Receiver/Transmitter Reference Circuit <http://www.radiotronix.com/support/click_download.asp?ContentId=168>

5. Boehner, Kirsten. *Bridging Art and Tool: A Case Study of Reflective Design in an Art Museum*, (2005).

6. Dan Golden Code/Website <http://instruct1.cit.cornell.edu//courses/eceprojectsland/STUDENTPROJ/2004to2005/dig4/dig4rfid.htm>

7. Bird sounds <http://www.math.sunysb.edu/~tony/birds/>

8. Radiotronix Forum <http://www.radiotronix.com/support/forums.asp?ForumId=7&TopicId=218>

9. LED Digikey SKU: 516-1369-ND <http://dkc3.digikey.com/PDF/T051/1522.pdf>

10. "FCC Regulations, Part 15" <http://www.access.gpo.gov/nara/cfr/waisidx_04/47cfr15_04.html>

11. Linx Technologies Splatch Antenna <http://www.linxtechnologies.com/index.php?section=products&category=antennas&subcategory=embeddable&series=sp_series>

12. Various RF Design Guide and Application Notes http://www.linxtechnologies.com/index.php?section=support

Appendix A – Cost:

Attractor or Birdies

Part Name	Manufacturer	Cost	Description
Mega32	Atmel	$5.20	Microcontroller
PIR325	Glolab	$4.00	Pyroelectric Infrared Sensor
FL65	Glolab	$4.00	Infrared Fresnel Lens
ISD2560	Winbond	$14.85	Voice Coder – Voice Playback Chip
MAX233CPP	Maxim	$6.19	RS232 Communication Chip
EAS-4D05A0	Panasonic	$7.98	Speaker 1W 16 OHM 40x30mm
Transformer	Jameco	$4.50	110AC to 9V 500mA
Misc		$2.00	Audio jacks, crystals, resister, capacitor
	TOTAL	$48.72	

Tempo indicator

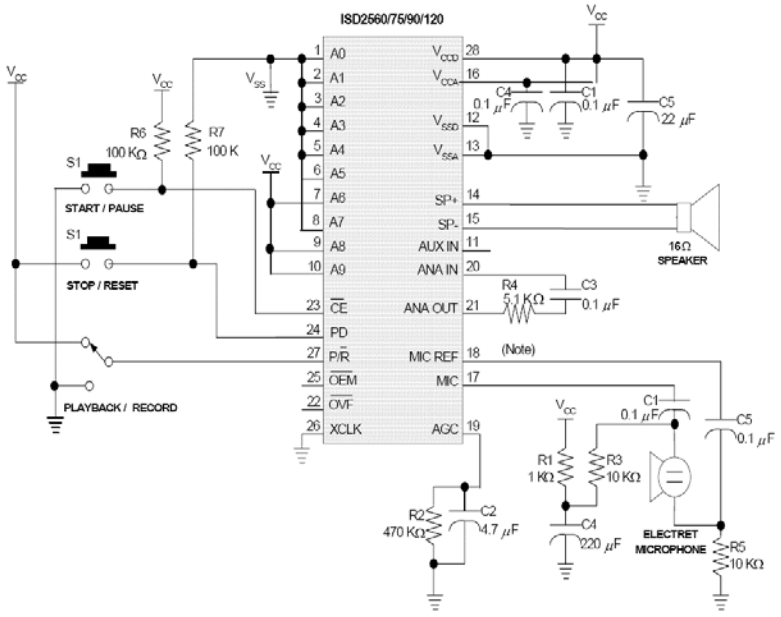
Part Name	Manufacturer	Cost	Description
Mega32	Atmel	$5.20	Microcontroller
PIR325	Glolab	$4.00	Pyroelectric Infrared Sensor
FL65	Glolab	$4.00	Infrared Fresnel Lens
MAX233CPP	Maxim	$6.19	RS232 Communication Chip
Transformer	Jameco	$4.50	110AC to 9V 500mA
Misc		$1.50	Crystals, resister, capacitor, etc
	TOTAL	$25.39	

Wireless People Counter Sensor Network (cost per node or base station)

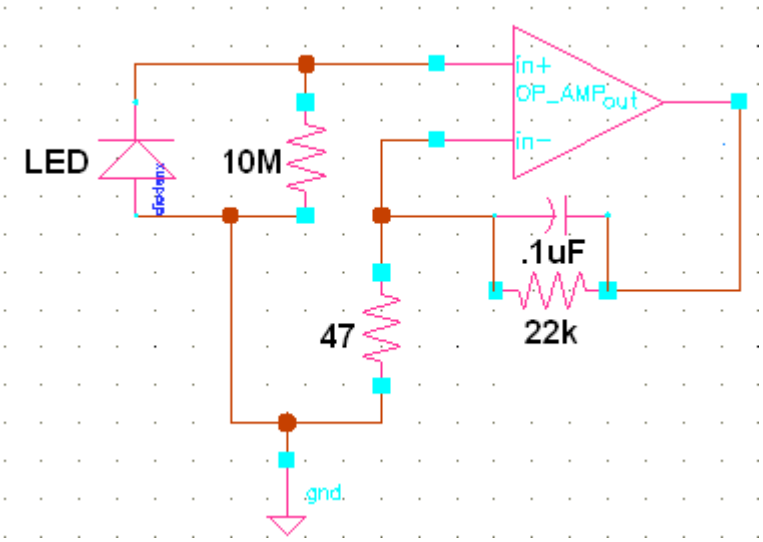
Part Name	Manufacturer	Cost	Description
Mega32	Atmel	$5.20	Microcontroller
MAX233CPP	Maxim	$6.19	RS232 Communication Chip
RCR-433-RP	Radiotronix	$4.99	433MHz Receiver
RCT-433-AS	Radiotronix	$3.99	433MHz Transmitter
ANT-433-PW-QW	Linx Technologies	$6.89 x2	433MHz Permanent Mount 1/4 Wavelength Antenna
HLMT-ED23-TW000	Aglient Technologies	$0.49 x2	Ultra-bright 4850mcd Red Water Clear LED 630nm
LMC7111BIN	National Semi	$1.31x2	OpAmp
Transformer	Jameco	$4.50	110AC to 9V 500mA
Misc		$5.50	Crystals, resister, capacitor, trimpot, etc
	TOTAL	$47.75	

Appendix B – Schematics and Waveform:

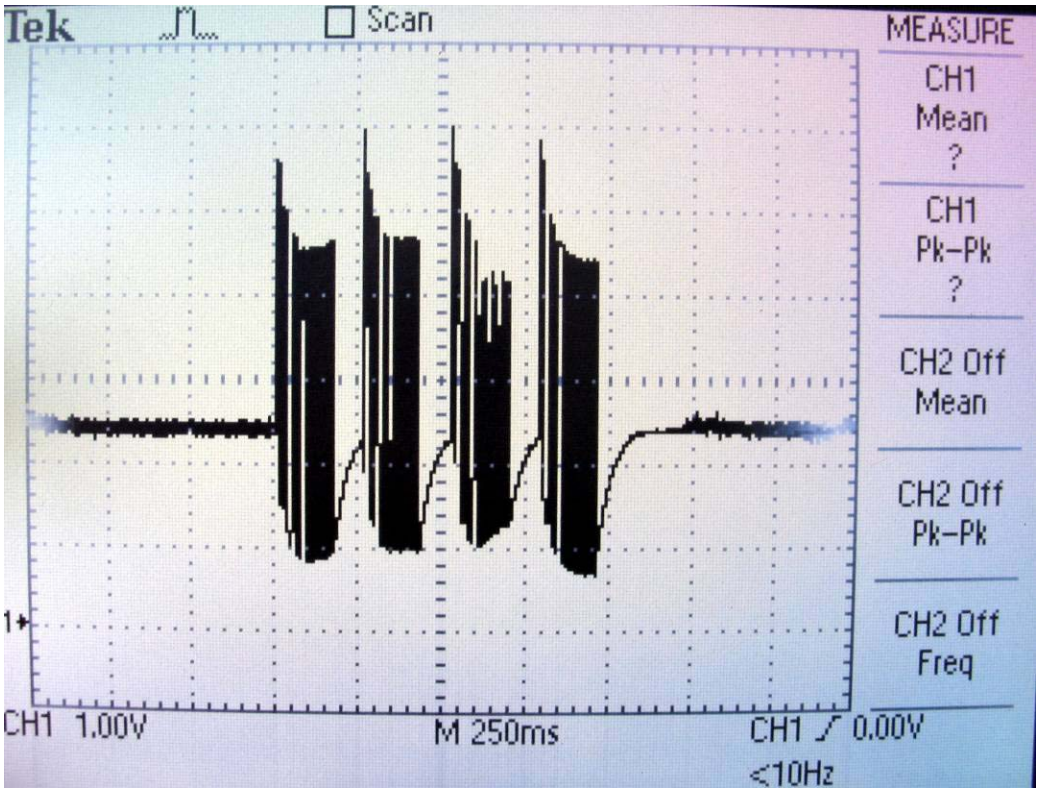
Voice Coder Schematic:



LED People Counting Sensor Schematic:



Receiver Waveform:



This wave is captured via a receiver's analog pin while the base station and the nodes are communicating. There are 4 packets shown in this graph. The first packet is the base station transmitting count packet to node 1, the second packet is the count ack packet from node 1, the third packet is the base station transmitting to node 2, and the fourth packet is the count ack packet from node 2. We can see the quality of the wave varies from packet to packet. The worst packet out of these four is the third packet.

Appendix C – Cloudscape pictures:

Figure C.1: what the cloudscape looks like from a projector

Figure C.2: screen capture of the cloudscape movie - a little red with rain.

Figure C.3: screen capture of the cloudscape movie - rainy and red hue overlay.

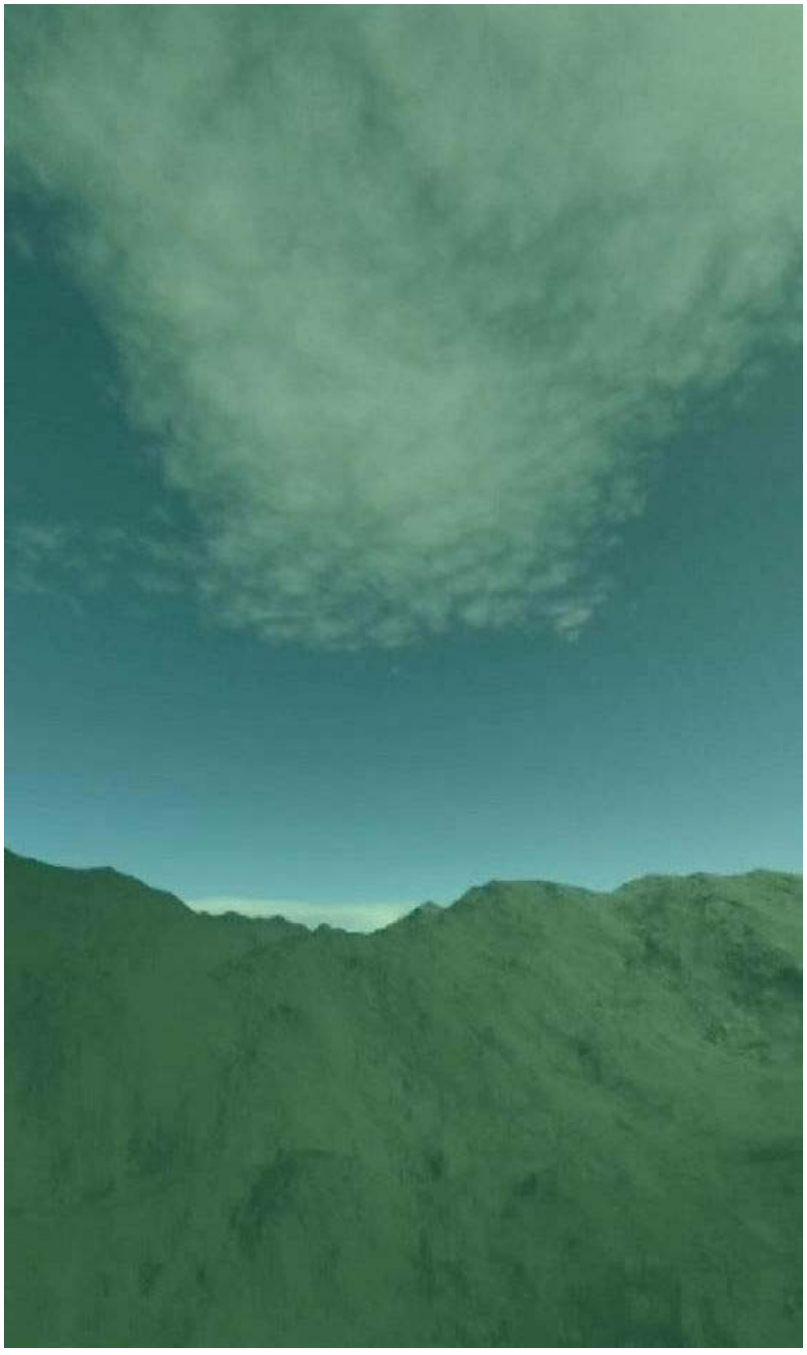


Figure C.4: screen capture of the cloudscape movie - Green hue overlay, no rain.

Appendix D – Device Pictures:

Attractor:

Figure D.1: One of the birdies that has been installed in Herbert Johnson Museum

Tempo Indicator:

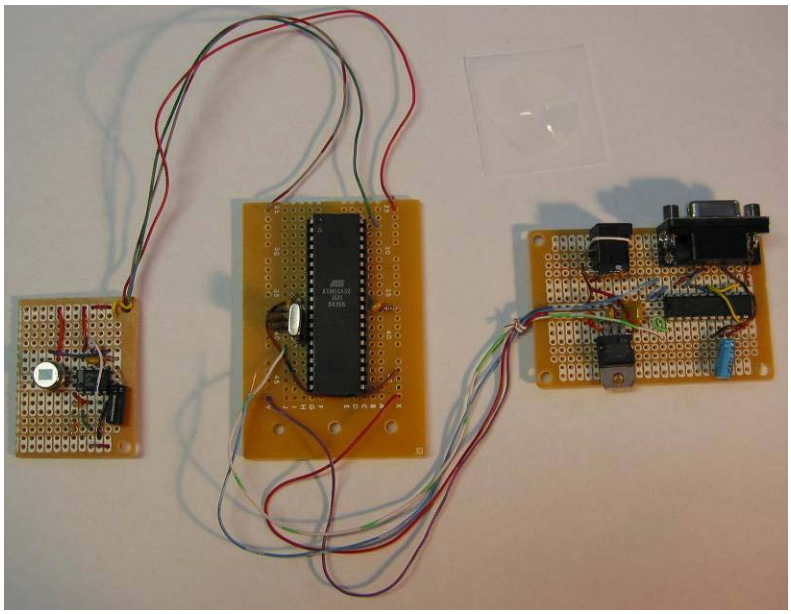


Figure D.2: Bare Tempo Indicator. PIR, Mega32, Power and Serial are the items on the boards left to right.

Figure D.3: Boxed Tempo Indicator with Transformer and RS232 cable for transmission real time data.

Sensor Network:

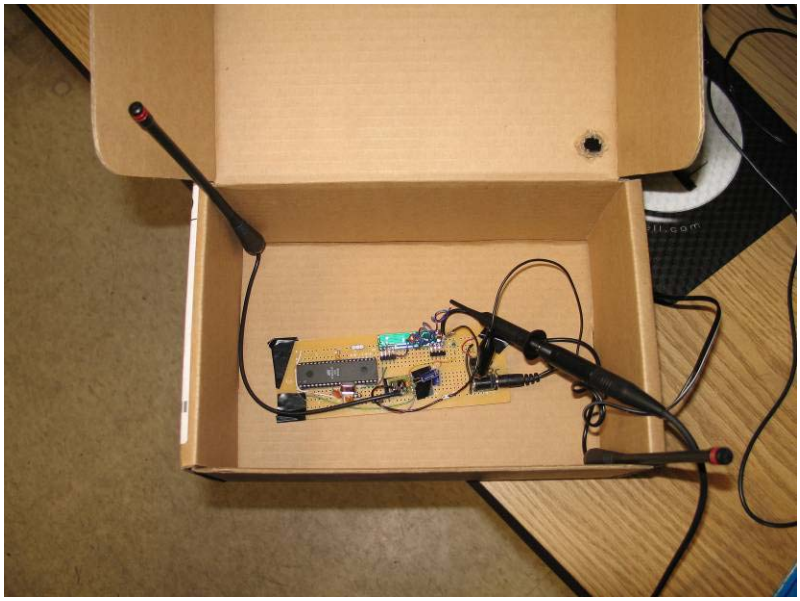


Figure D.4: One of the people counting nodes

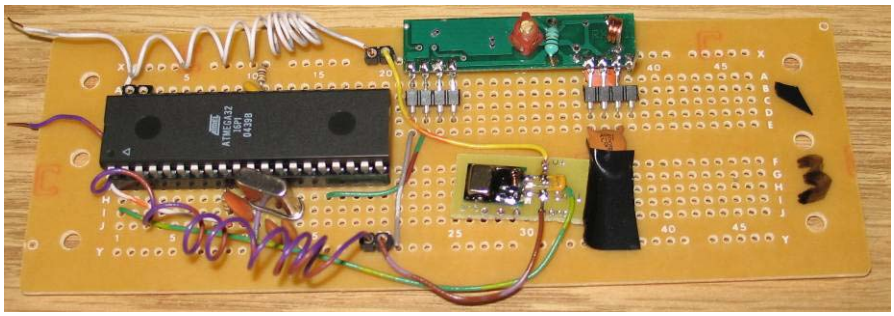


Figure D.5: One of the people counting nodes

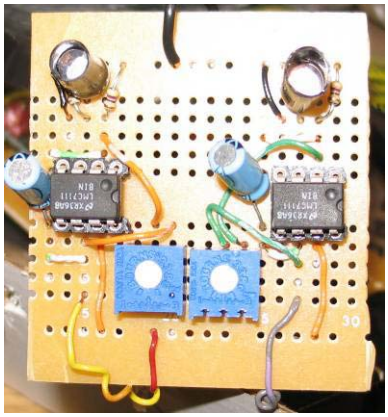


Figure D.6: Shrink-tube wrapped LED Sensors. The two trim pots are there for manual gain adjustments.

Appendix E – Source Code:

Attractor Source Code: birdy.c

```
// Birdies
// Author: Eric Lee CEL27@cornell.edu
// An HCI Group Project


#include <Mega32.h>
#include <stdio.h> // sprintf
#include <stdlib.h>
#include <delay.h>
//#asm
//    .equ __lcd_port=0x15
//#endasm
//#include <lcd.h> // LCD driver routines
#define SYSCLOCK 16000000

#define TIMEARRAY_SIZE 1400

#define LCDwidth 20

#define DEFAULT_TIME_BEFORE_TRIGGER 120  // wait 2 minutes before starting to play bird
sound
#define TIME_TO_SLEEP 720  //Time before it goes into sleep mode. 10min + 2 minutes

#define TRUE 1
#define FALSE 0

#define PIR_NOMINAL 52  // nominal voltage of the PIR
#define PIR_SWING 7           // swing + and -
#define PIR_LOW_BOUND (PIR_NOMINAL - PIR_SWING)
#define PIR_HIGH_BOUND (PIR_NOMINAL + PIR_SWING)

#define FREQ_PER_MIN 2
#define CNT_PER_ELEM 2


enum {PLAY_H, PLAY_L, STOP_H, STOP_L, DONE};
enum {DUMP=1, CLEAR, TOGGLE_REC, SET_START_TIME, SET_END_TIME, SETTIME, MENU, NORMAL,
WAIT, READY};

unsigned char led;     //light states
unsigned char playDelay;  //timing variable for lcd

unsigned int oneSecond;
unsigned char birdySingTime;
unsigned char adcDelay;



unsigned int noMotionDetectedTime;

unsigned short birdyIsOn;
unsigned short prevBirdyIsOn;
unsigned short birdyInSleep;

int voltPIR;

unsigned int recStartingDay;
unsigned int day;
// eeprom variable to keep start recording time and stop recording time.
eeprom unsigned char recStartingHour=10, recEndingHour=17;
unsigned char dayRecorded;
unsigned char hour, minute, sec;

unsigned char playState;
```

```
//SERIAL COMM DECL
unsigned char inputChar;
unsigned char cmdString[20];

unsigned char char_count, cmd_ready;
unsigned char stateTask;

unsigned char commandNum;

unsigned char triggerTimeArray[TIMEARRAY_SIZE];
unsigned char triggerCount, location;
unsigned int index;
unsigned char prevTrigger, currTrigger;
unsigned char triggerCountDelay;

unsigned char recording;

unsigned char manualStartDay, manualStartHour, manualStartMin;

unsigned char manualStart, manualStop;
unsigned char toggle;

int Ain;
int clock;
int ticked = 0;
//unsigned char ledbuf[21];


//Function prototype

void initialize(void); //all the usual mcu stuff
void menu(void);
void get_cmd_init(void);

unsigned int calcArrayUsage();
void startPlayState(void);
void processTasks(void);
int ctoi(char x);
void printTime(void);
void initTriggerTimeArray();

void clearHistory();

int ctoi(char x)
{
        //printf("%d\r\n", (int)(x - '0'));
        return (int)(x - '0');
}

// put number of PIR triggers into array. Every char is divided into two parts, which
stores two triggers
// or 1 minutes worth of trigger.
// count = trigger count
// index = the array index where to put the data,
// location = in that index, is it at the first or second half. the range of location
should be [0,1]
void triggerCountToArray(unsigned char count, unsigned int arrayIndex, unsigned char
location)
{
        if(count > 0x0F)
                count = 0x0F;
        if(location >= 2)
            putsf("location out of bound! in triggerCountToArray");
        if(arrayIndex < TIMEARRAY_SIZE)
        {
                if(location == 0)
                {
                        triggerTimeArray[arrayIndex] |= (count & 0x0F);
                }
```

```
                else if(location == 1)
                {
                        triggerTimeArray[arrayIndex] |= (count & 0x0F) << 4;
                }
        }

}


// dump all the triggering data to serial
void dumpCountToSerial()
{
        int i=0;

        if(index == 0)
        {
                putsf("There is no History");
                return;
        }

        if(manualStart)
        {
                printf("Manual Starting time: %02d %02d:%02d\r\n", manualStartDay,
manualStartHour, manualStartMin);
        }
        else
        {
                printf("Starting time: %02d %02d:%02d\r\n", recStartingDay,
recStartingHour, 0);
        }

        for(i=0; i<index; i++)
        {
                printf("%02d %02d | ", triggerTimeArray[i] & 0x0F, (triggerTimeArray[i] &
0xF0) >> 4);

                // five data on a line
                if((i+1)%5 == 0)
                {
                        putsf("\r");
                }
        }
}



interrupt [TIM1_COMPA] void cmpA_overflow(void){
        //blink();

        ++sec;
        //sec+=30; // FOR TESTING ONLY
        //when we are actually recording
        if(recording)
        {
                // dump the data into the array every 30 seconds
                if(sec == 30 || sec==60)
                {
                        /// triggerCount = minute%15;//+++   FOR TESTING ONLY
                        triggerCountToArray(triggerCount, index, location);
                        location++;
                        // +++ should reset trigger count here
                    triggerCount = 0;
                }
        }


        if(sec==60)
        {
                // reset the location
                if(recording)
                {
```

```
                        location=0;
                        index++;
                }

                ++minute;
                sec=0;

                // start the manual recording
                if(!recording && manualStart)
                {
                        manualStartDay = day;
                        manualStartHour = hour;
                        manualStartMin = minute;
                        recording = 1;
                }

        }


        if(minute==60)
        {
                ++hour;
                minute=0;

                // start and end recording
                if(hour == recStartingHour)
                {
                        // only trun on recording if the remaining space is enough for a
full day of data.
                        if(calcArrayUsage() < (TIMEARRAY_SIZE-index-1))
                        {
                                recording = 1;

                                if(dayRecorded ==0)
                                        recStartingDay = day;

                                dayRecorded++;

                        }
                }

                if(hour == recEndingHour)
                {
                        recording = 0;
                }

        }

        if(hour==24)
        {
                ++day;
                hour=0;
        }


}


// Calculate array usage.  The amount of time remaining will be shown.
unsigned int calcArrayUsage()
{
        int hourPerDay;
        hourPerDay = recEndingHour - recStartingHour;
        return (unsigned int)(hourPerDay * 60 * FREQ_PER_MIN / 2);

}
```

```
//*********************************************************
//timer 0 overflow ISR
interrupt [TIM0_COMP] void timer0_compare(void)
{
        ticked = 1;
        playDelay++;
        //btndelay++;
        adcDelay++;
        oneSecond++;
        triggerCountDelay ++;



        // milisecond time increment, get 1 second from 1000 mili
        if(oneSecond >=1000)
        {
                oneSecond = 0;
                noMotionDetectedTime++;
        }

        //When motion is detected reset the noMotionDetectedTime.
        if(voltPIR < PIR_LOW_BOUND || voltPIR > PIR_HIGH_BOUND )
        {
                // +++ increament the trigger count HERE!
                currTrigger = 1;


                //PORTB.7 = 0;
                noMotionDetectedTime = 0;

                //once the motion is detected again, stop the birdy
                //PORTB.0 = 1;
                birdyIsOn = FALSE;
                playState=STOP_H;
                //if the birdy is in sleep mode, wake it up
                birdyInSleep = FALSE;
        }
        else
        {
                //PORTB.7 = 1;
                if(triggerCountDelay > 10)
                {
                        currTrigger = 0;
                        triggerCountDelay =0;
                }
        }

        if(currTrigger && !prevTrigger)
                triggerCount++;

        prevTrigger = currTrigger;

        //Ain=ADCH;
        //ADCSRA = ADCSRA | 0x40;
}

//*********************************************************
//ADC interrupt
interrupt [ADC_INT] void adc_done(void)
{
   Ain = ADCH;
   ADCSRA.6 = 1;
}

//*********************************************************
//Entry point and task scheduler loop
void main(void)
{

  initialize();
  menu();
```

```
  //main task scheduler loop
  while(1)
  {

        if(prevBirdyIsOn == FALSE && birdyIsOn == TRUE)
        {
                playState=PLAY_H;
                //startPlay();
        }
        else if(prevBirdyIsOn == TRUE && birdyIsOn == FALSE)
        {
                playState=STOP_H;
        //      stopPlay();
        }
        prevBirdyIsOn = birdyIsOn;


        if(playDelay >=10)
        {
                startPlayState();
                //stopPlayState();
                playDelay =0;
                //displayBirdySettings(voltPIR, noMotionDetectedTime, timeBeforeTrigger);
                //blink();
        }


        if(adcDelay >= 100)
        {
                adcDelay = 0;
                #asm
                        sleep
                #endasm

                //get the voltage from the ADC
                voltPIR = Ain;

                //displayBirdySettings(voltPIR, noMotionDetectedTime, timeBeforeTrigger);

            // 50/256 * 4.9 = 1 volt

         }

        if(noMotionDetectedTime >= DEFAULT_TIME_BEFORE_TRIGGER && birdyInSleep == FALSE)
                birdyIsOn = TRUE;
        else
                birdyIsOn = FALSE;


        if(noMotionDetectedTime > TIME_TO_SLEEP)
        {
                birdyIsOn = FALSE;
                birdyInSleep = TRUE;
        }


        //wake up the birdy when motion detected.
        if(noMotionDetectedTime == 0 && birdyInSleep==TRUE)
                birdyInSleep = FALSE;



        //When the user have input a string at the birdy console
        if (cmd_ready)
        {
                processTasks();
        }

        }
}
```

```
void blink()
{
        if(toggle == 0)
        {
                PORTB.5=1;
                toggle =1;

        }
        else
        {
                PORTB.5=0;
                toggle=0;
        }
}

/*
// PORTB.1 = play_low
void startPlay(void)
{
        // assert low then high
        PORTB.1=0;
        delay_us(1);
        PORTB.1=1;

}

void stopPlay(void)
{
        //assert reset high
        PORTB.2 = 1;
        delay_us(1);
        PORTB.2 = 0;
}

*/

void startPlayState(void)
{
 //time_start=t3;     //reset the task timer


  switch (playState)
  {
    case PLAY_H:
         PORTB.1 = 0;
         playState = PLAY_L;
         break;
        case PLAY_L:
                PORTB.1 = 1;
                playState = DONE;
         break;

    case STOP_H:
         PORTB.2 = 1;
         playState = STOP_L;
         break;
    case STOP_L:
         PORTB.2 = 0;
         playState = DONE;
         break;
    case DONE:
        playState = DONE;
        break;
  }
}

  /*
void displayBirdySettings(int voltPIR, int noMotionDetectedTime, int timeBeforeTrigger)
{
```

```
        //print the current room tempearture
        lcd_clear();
        lcd_gotoxy(0,0);
        lcd_putsf("PIR adc=");

        lcd_gotoxy(15,0);
        sprintf(ledbuf, "%d", voltPIR);
        lcd_puts(ledbuf);

        //print the setpoint temperature
        lcd_gotoxy(0,1);
        lcd_putsf("noMoDetTime=");

        lcd_gotoxy(15,1);
        sprintf(ledbuf, "%d", noMotionDetectedTime);
        lcd_puts(ledbuf);


        lcd_gotoxy(0,2);
        lcd_putsf("timeBeforeTrig=");

        lcd_gotoxy(15,2);
        sprintf(ledbuf, "%d", timeBeforeTrigger);
        lcd_puts(ledbuf);



        lcd_gotoxy(0,3        );
        if(birdyInSleep == TRUE)
                lcd_putsf("SLEEEEP MODE");
        else
                lcd_putsf("AWAKE MODE");

        lcd_gotoxy(15, 3);
        if(birdyIsOn == TRUE)
                lcd_putsf("SING");
        else
                lcd_putsf("QUITE");

}


 */

 //*********************************************************
//interrupt for UART RXC
interrupt [USART_RXC] void serialInterrupt(void)
{
        inputChar = UDR;
        UDR = inputChar;

        if(inputChar != '\r') //when did not encounter a carrage return
                cmdString[char_count++] = inputChar;
        else
        {
                putchar('\n');
                //terminate the string
                cmdString[char_count] = 0;
                cmd_ready = 1;
                UCSRB.7 = 0;
        }

}


//process security manager's task
void processTasks(void)
{
//      unsigned int tempCode = 0;
        int i;
```

```
        i=0;

        switch (stateTask)
        {
                //at the menu state
                case MENU:
                        commandNum = ctoi(cmdString[0]);
                        if (cmdString[0] == 0 || cmdString == 0)
                        {
                                menu();

                                break;
                        }

                        switch(commandNum)
                        {
                                case DUMP:
                                        putsf("=================\r\n");
                                        //get_cmd_init();
                                        dumpCountToSerial();
                                case stateTask = MENU;
                                menu();

                                        //stateTask = DUMP;
                                        break;
                                case CLEAR:
                                        if(index != 0)
                                        {
                                                putsf("ARE YOU SURE? [y/n]\r\n");
                                                get_cmd_init();
                                                stateTask = CLEAR;
                                        }
                                        else
                                        {
                                                putsf("There is no history to clear");
                                        }

                                        break;
                                case TOGGLE_REC:
                                        putsf("ARE YOU SURE, you want to start or stop
recording? [y/n]\r\n");
                                        get_cmd_init();
                                    stateTask = TOGGLE_REC;
                                        break;
                                case SET_START_TIME:
                                        if(!recording)
                                        {
                                                putsf("Please enter START TIME in hh format.
Example 07 = 7am : \r\n");
                                                get_cmd_init();
                                                stateTask = SET_START_TIME;
                                        }
                                        else
                                        {
                                                putsf("You have to stop recording before you
can change the start time.");
                                                stateTask = MENU;
                                        }
                                        break;

                                case SET_END_TIME:
                                        if(!recording)
                                        {
                                                putsf("Please enter END TIME in hh format.
Example 17 = 5pm : \r\n");
                                                get_cmd_init();
                                                stateTask = SET_END_TIME;
                                        }
                                        else
                                        {
```

```
                                        putsf("You have to stop recording before you
can change the ending time.");
                                        stateTask = MENU;
                                }
                                break;

                        case SETTIME:
                                putsf("Input time in hhmmddd format where hh=hour,
mm=minutes, ddd=day of the year.\r\nEample: 0908032 = 9:08am, Feb 1st\r\n");
                                stateTask = SETTIME;
                            break;
                        default:
                                printf("\n\r**Invalid Input** %d %d\n\r",
commandNum, SETTIME);
                                menu();
                                break;
                }


                break;

        case DUMP:

                stateTask = MENU;
                break;

        case CLEAR:

                if(cmdString[0] == 'y')
                {
                        clearHistory();
                        putsf("All The History Has Been Cleared\r\n");
                        stateTask = MENU;
    }
    else
    {
                        stateTask = MENU;
                menu();
                        break;

                }
                break;


        case TOGGLE_REC:

                if(cmdString[0] == 'y')
                {
                        if(recording && !manualStop)
                        {
                                recording = 0;
                                manualStop = 1;
                                putsf("STOP Recording, The history must be cleared
before you can start recording again\r\n");
                        }
                        else
                        {
                                //recording is set to 1 in the interrupt
                                manualStart = 1;
                                putsf("START Recording.\r\n");
                        }
                }
                stateTask = MENU;
                menu();

                break;

        //set the system current time hhmmddd
        case SET_START_TIME:
                if(cmdString[2] != 0)
                {
```

```
                                putsf("Invalid format - please input in hh format, try
again.\r\n");
                                stateTask = SET_START_TIME;
                                break;
                        }
                        recStartingHour = ctoi(cmdString[0]) *10 +  ctoi(cmdString[1]);

                        printf("Recording Start Time Has Been Changed to %02d\r\n",
recStartingHour);
                        stateTask = MENU;
                        menu();

                        break;

                //set the system current time hhmmddd
                case SET_END_TIME:
                        if(cmdString[2] != 0)
                        {
                                putsf("Invalid format - please input in hh format, try
again.\r\n");
                                stateTask = SET_END_TIME;
                                break;
                        }
                        recEndingHour = ctoi(cmdString[0]) *10 +  ctoi(cmdString[1]);

                        printf("Recording End Time Has Been Changed to %02d\r\n",
recEndingHour);
                        //printf("Based on the starting and ending time you set,\r\nthe
birdie can keep %d days of data \r\n", TIMEARRAY_SIZE/calcArrayUsage());
                        stateTask = MENU;
                        menu();

                        break;


                //set the system current time hhmmddd
                case SETTIME:
                        if(cmdString[7] != 0)
                        {
                                putsf("Invalid format - please input in hhmmddd format, try
again.\r\n");
                                stateTask = SETTIME;
                                break;
                        }
                        hour = ctoi(cmdString[0]) *10 +  ctoi(cmdString[1]);
                        minute = ctoi(cmdString[2]) *10 +  ctoi(cmdString[3]);


                        day = ctoi(cmdString[4]) *100 +  ctoi(cmdString[5])*10 +
ctoi(cmdString[6]);


                        putsf("System Time Has Been Changed\r\n");
                        stateTask = MENU;
                        menu();

                        break;

                default:
                        //wrong command, return some sensable message to SM.
                        putsf("Invalid command");

                        break;
        }

        get_cmd_init();
}


//*********************************************************
//print out the current system time
```

```
void printTime(void)
{
	printf("Current System Time: Day %03d, %02d:%02d %02d\r\n", day, hour, minute,
sec);
}



//*********************************************************
//  -- non-blocking keyboard check initializes ISR-driven
// receive. This routine merely sets up the ISR, which then
//does all the work of getting a command.
//*********************************************************
void get_cmd_init(void)
{
   char_count=0;
   cmd_ready=0;
   UCSRB.7=1;
}

void initTriggerTimeArray()
{
	int i=0;
	for(i = 0; i< TIMEARRAY_SIZE; i++)
		triggerTimeArray[i] = 0;
}


void clearHistory()
{
	initTriggerTimeArray();
	recStartingDay = 0;
	index = 0;
	location = 0;
	manualStop = 0;
	manualStart = 0;
	recStartingDay = 999;
	dayRecorded=0;
	recording = 0;

}


//*********************************************************
// birdie Manager menu
void menu(void)
{
	putsf("\n\n\rBirdy Manager Menu\r\n");


	printTime();
	if(recording)
		printf("Status: Recording %dst day\r\n", dayRecorded);
	else
		putsf("Status: Idle\r");

	if(recEndingHour != 0 && recStartingHour != 0)
		printf("History Capacity: %d Days (based on the starting and end
time)\r\n", TIMEARRAY_SIZE/calcArrayUsage());
	else
		putsf("**Please set the start and end recording time.\r\n");

	putsf("Options:\r");
	printf("1. Dump all history since day:%03d hour:%02d \r\n", recStartingDay,
recStartingHour);
	printf("2. Clear all the history since day:%03d hour:%02d \r\n", recStartingDay,
recStartingHour);
	putsf("3. Toggle the recording status\r");
```

```
        printf("4. Set Start Recording Time - current start time = %d\r\n",
recStartingHour);
        printf("5. Set End Recording Time - current end time = %d\r\n", recEndingHour);
        putsf("6. Set time and day\r");
}



//*********************************************************
//Set it all up
void initialize(void)
{
//set up the ports
  DDRD=0x00;   // PORT D is an input
  DDRB=0xff;    // PORT B is an ouput

  //init the LED status (all off)
  led=0xFF;
  PORTB = led;
  PORTB.1=1;
  PORTB.2=0;

  //timer 1 setup
  TIMSK = 0b00010000;
  //turn on output compare A match
  TCCR1A = 0;
  TCCR1B = 8+5;
  //CTC mode, prescalar = 1024
  OCR1A = SYSCLOCK/1024;


  //set up timer 0
  TIMSK = TIMSK | 2;          //turn on timer 0 cmp match ISR
  OCR0 = 250;          //set the compare re to 250 time ticks
  TCCR0 = 0b00001011;  //prescalar to 64 and turn on clear-on-match



  playState = DONE;

  toggle=0;
  //init variables
  clock = 0;

  noMotionDetectedTime=0;
  adcDelay =0;
  oneSecond = 0;
  birdyIsOn = FALSE;
  prevBirdyIsOn = FALSE;
  birdyInSleep = FALSE;
  birdySingTime = 0;

  triggerCountDelay = 0;
  voltPIR = 0;

  stateTask = MENU;


  //init the LCD
  //lcd_init(LCDwidth);       //initialize the display
  //lcd_clear();                      //clear the display

  //setup RS232
  UCSRB = 0x10 + 0x08 ;
  UBRRL = 103;                //9600 baud rate


  //init time
  sec=0;
  minute=0;
```

```
  hour=0;
  day=0;
  prevTrigger=0;
  currTrigger=0;
  triggerCount = index = location = 0;
  // init toggle array stuff.
  clearHistory();
  recording = 0;


  ADMUX =  0b01100000; //read voltage on A.0
  ADCSRA = 0b11001111; //ADC on,
  //enable sleep mode
  MCUCR = 0b10000000;

  get_cmd_init();
  //crank up the ISRs
  #asm
        sei
  #endasm
}
```

Tempo Indicator Source Code: tempo.c

```
#include <Mega32.h>
#include <stdio.h> // sprintf
#include <string.h>
#include <delay.h>

#define SYSCLOCK 16000000

#define maxkeys 16
#define t 1
#define t1 30

//the terminal
#define TERMINATOR
#define NO_BTN 88

//set time for each task
#define KEYPAD_DECODE 30

//commend decoding
#define MAX_NUM_CMD 6

//MAX number of pass code
#define MAX_NUM_CODE 8
#define MAX_NUM_KEYS 16

#define TRUE 1
#define FALSE 0



#define PIR_NOMINAL_VOLT 44
#define PIR_DIVIDER 4
#define PIR_ADJUST 3


//Function prototype
void initialize(void); //all the usual mcu stuff

//LED
unsigned char led;

//KEYPAD VARIABLE DECL
unsigned char keypadTask;
unsigned char keypadState, butnum, key_count, keypad0000;
unsigned char keyString[6];

//SERIAL COMM DECL
unsigned char inputChar, keypadReadState;
unsigned char cmdString[80];

unsigned char char_count, cmd_ready,accessGranted;
unsigned char stateTask;
unsigned char SMLockState;
unsigned char commandNum;
//Time related variables
int time, time1, day, hour, minute, sec, attempts, num_code, lockOneMin;



int Ain, oneSecond, voltPIR;
unsigned int speed70, speed60, speed50, speed40, speed30, speed20;

unsigned char adcDelay;
unsigned char reportDelay;

int clock;
int ticked = 0;
unsigned char ledbuf[17];
```

```
//*********************************************************
//ADC interrupt
interrupt [ADC_INT] void adc_done(void)
{
    Ain = ADCH;
    ADCSRA.6 = 1;
}


//*********************************************************
//timer 0 compare overflow
interrupt [TIM0_COMP] void cmpA_overflow(void)
{

    adcDelay++;
    oneSecond++;
    reportDelay++;

    if(oneSecond >=1000){
        oneSecond=0;
        sec++;
    }

}


//*********************************************************
//interrupt for UART RXC
interrupt [USART_RXC] void serialInterrupt(void)
{
    inputChar = UDR;
    UDR = inputChar;

    if(inputChar != '\r') //when did not encounter a carrage return
        cmdString[char_count++] = inputChar;
    else
    {
        putchar('\n');
        //terminate the string
        cmdString[char_count] = 0;
        cmd_ready = 1;
        UCSRB.7 = 0;
    }

}




//*********************************************************
//Entry point and task scheduler loop
void main(void)
{

    initialize();

    while(1)
    {

        if(adcDelay >= 100)
        {

            adcDelay = 0;
#asm
            sleep
#endasm
```

```
                //get the voltage from the ADC
                voltPIR = Ain;


            reportDelay=0;
            //printf("current volt %d\r\n", voltPIR);
            // DIFFERENT THRESHOLDS
            if(voltPIR > PIR_NOMINAL_VOLT+PIR_DIVIDER*2+PIR_ADJUST)
            {
                speed70++;
            }
            else if(voltPIR > PIR_NOMINAL_VOLT+PIR_DIVIDER+PIR_ADJUST)
            {
                speed60++;
            }
            else if(voltPIR > PIR_NOMINAL_VOLT+PIR_ADJUST)
            {
                speed50++;
            }
            else if (voltPIR < PIR_NOMINAL_VOLT-PIR_ADJUST && voltPIR > PIR_NOMINAL_VOLT-
PIR_DIVIDER-PIR_ADJUST)
            {
                speed40++;
            }
            else if (voltPIR < PIR_NOMINAL_VOLT-PIR_ADJUST && voltPIR > PIR_NOMINAL_VOLT-
PIR_DIVIDER*2-PIR_ADJUST)
            {
                speed30++;
            }
            else if(voltPIR < PIR_NOMINAL_VOLT-PIR_ADJUST)
            {
                speed20++;
            }


        }

        //RESETS THE THRESHOLD
        if(sec >=5)
        {
            sec =0;
            printf("slowH %d midH %d fastH %d fastL %d midL %d slowL %d\r\n", speed70,
speed60, speed50, speed40, speed30, speed20);

            speed70=0;
            speed60=0;
            speed50=0;
            speed40=0;
            speed30=0;
            speed20=0;


        }


    }
}



//*********************************************************
//Set it all up
void initialize(void)
{
    //Init port B to show keyboard result
    DDRB = 0xff;
    PORTB = 0xff;


    //set up timer 0
```

```
    TIMSK=2;           //turn on timer 0 cmp match ISR
    OCR0 = 250;               //set the compare re to 250 time ticks
    TCCR0 = 0b00001011;       //prescalar to 64 and turn on clear-on-match



    UCSRB = 0x10 + 0x08 ;
    UBRRL = 103;                //9600 baud rate

    stateTask = MENU;
    keypadState = DONE;
    butnum = NO_BTN;
    SMLockState = NORMAL;

    key_count = 0;
    keypadReadState = WAIT;

    //init task timers
    time=t;
    time1=t1;

    //init time
    sec=0;
    minute=0;
    hour=0;
    day=0;
    attempts = 0;

    oneSecond=0;
    adcDelay=0;


    speed70=0;
    speed60=0;
    speed50=0;
    speed40=0;
    speed30=0;
    speed20=0;




    ADMUX =  0b01100000; //read voltage on A.0
    ADCSRA = 0b11001111; //ADC on,
    //enable sleep mode
    MCUCR = 0b10000000;


    //crank up the ISRs

#asm
    sei
#endasm
}
```

Tempo Indicator to Macromedia Director Source Code: CloudControl.java

```
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import javax.comm.*;

public class CloudControl extends JFrame {

       public class SerialComm implements SerialPortEventListener,
CommPortOwnershipListener {
              private CloudControl ctrl;
              private InputStream is;

              private int numSinceChange; // number of times serial event occured since
we last changed
                                                                  // to or from rain

           private CommPortIdentifier portId;
           private SerialPort sPort;

              public SerialComm(CloudControl control) throws SerialConnectionException {
                     if(control == null) throw new IllegalArgumentException("Need valid
CloudControl");

                     ctrl = control;

                     numSinceChange = 0;

                     ////// Set up and open serial port //////

                     // Obtain a CommPortIdentifier object for the port we want to open.
                     try {
                            portId = CommPortIdentifier.getPortIdentifier("COM1");
                     } catch (NoSuchPortException e) {
                            throw new SerialConnectionException(e.getMessage());
                     }

                     // Open serial port
                     try {
                         sPort = (SerialPort)portId.open("CloudControl", 30000);
                     } catch (PortInUseException e) {
                         throw new SerialConnectionException(e.getMessage());
                     }

                     // Set parameters of serial port
                     try {
                         sPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
SerialPort.STOPBITS_1,
                                                   SerialPort.PARITY_NONE);
                     } catch (UnsupportedCommOperationException e) {
                            sPort.close();
                         throw new SerialConnectionException("Unsupported parameter");
                     }

                     // Set flow control.
                     try {
                         sPort.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);
                     } catch (UnsupportedCommOperationException e) {
                            sPort.close();
                         throw new SerialConnectionException("Unsupported flow
control");
                     }

                     // Open the input stream for the connection
                     try {
                         is = sPort.getInputStream();
                     } catch (IOException e) {
```

```
                        sPort.close();
                        throw new SerialConnectionException("Error opening i/o
streams");
                    }

                    // Add this object as an event listener for the serial port.
                    try {
                        sPort.addEventListener(this);
                    } catch (TooManyListenersException e) {
                        sPort.close();
                        throw new SerialConnectionException("too many listeners
added");
                    }

                    // Set notifyOnDataAvailable to true to allow event driven input.
                    sPort.notifyOnDataAvailable(true);

                    // Set receive timeout to allow breaking out of polling loop during
input handling.
                    try {
                        sPort.enableReceiveTimeout(30);
                    } catch (UnsupportedCommOperationException e) { }

                    // Add ownership listener to allow ownership event handling.
                    portId.addPortOwnershipListener(this);
            }

            public void serialEvent(SerialPortEvent e) {
                    numSinceChange++;

                    // Create a StringBuffer and int to receive input data.
                    StringBuffer inputBuffer = new StringBuffer();
                    int newData = 0;

                    // Determine type of event.
                    if(e.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
                            while (newData != -1) {
                                    try {
                                        newData = is.read();
                                        if (newData == -1) {
                                            break;
                                        }
                                        if ('\r' == (char)newData) {
                                            inputBuffer.append('\n');
                                        } else {
                                            inputBuffer.append((char)newData);
                                        }
                                    } catch (IOException ex) {
                                        System.err.println(ex);
                                        return;
                                    }
                            }
                            // Deal with input
                            String data = new String(inputBuffer);

                            int fastHEnd = data.indexOf("fastH ") + 6;
                            int fastLBegin = data.indexOf(" fastL");
                            int fastLEnd = data.indexOf("fastL ") + 6;
                            int midLBegin = data.indexOf(" midL");

                            int amtActivity = -1;
                            try {
                                    int amtH = Integer.parseInt(data.substring(fastHEnd,
fastLBegin));
                                    int amtL = Integer.parseInt(data.substring(fastLEnd,
midLBegin));

                                    amtActivity = amtH + amtL;
                            } catch (NumberFormatException ife) {
                                    System.out.println("Poorly formatted input");
                            }
```

```
                        // Tell director ctrl what buttons to press
                        if(amtActivity > 0) ctrl.Activity(amtActivity);

                        if(numSinceChange == 3) {
                                if(ctrl.isRaining()) {
                                        ctrl.StopRain();
                                } else {
                                        ctrl.MakeRain();
                                }
                                numSinceChange = 0;
                        }
                }
        }

        public void ownershipChange(int arg0) { /* We keep ownership... */ }
}

public class CloudControl extends JFrame {
        private Robot directorCtrl;
        private boolean isRaining = false;

        public CloudControl() throws AWTException, SerialConnectionException {
                super("CloudControl");
                JButton quit = new JButton("EXIT");
                quit.setActionCommand("quit");

                getContentPane().add(quit);

                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                quit.addActionListener(new ActionListener() {

                        public void actionPerformed(ActionEvent e) {
                                if("quit".equals(e.getActionCommand())) System.exit(0);
                        }
                });
                pack();
                setVisible(true);

                directorCtrl = new Robot();

                SerialComm driver = new SerialComm(this);
        }

        public boolean isRaining() {
                return isRaining;
        }

        public void MakeRain() {
                directorCtrl.keyPress(KeyEvent.VK_D);

                isRaining = true;
        }

        public void StopRain() {
                directorCtrl.keyPress(KeyEvent.VK_A);

                isRaining = false;
        }

        public void Activity(int ticks) {
                for(int i = 0; i < ticks; i++) {
                        directorCtrl.keyPress(KeyEvent.VK_Z);
                }
        }

        public static void main(String[] args) throws AWTException,
SerialConnectionException, IOException {
                CloudControl me = new CloudControl();
                Runtime.getRuntime().exec("director.exe");
        }
}
```

SerialConnectionException.java

```
public class SerialConnectionException extends Exception {

    /**
     * Constructs a <code>SerialConnectionException</code>
     * with the specified detail message.
     *
     * @param   s   the detail message.
     */
    public SerialConnectionException(String str) {
        super(str);
    }

    /**
     * Constructs a <code>SerialConnectionException</code>
     * with no detail message.
     */
    public SerialConnectionException() {
        super();
    }
}
```

Wireless People Counting Network – Base Station Source Code:

```
/*************************************************************
*****       HCI Wireless People Counting Project       *******
*****               Base Station Code                  *******
*****                                                  *******
*****          By Eric Lee and Arun Israel             *******
*****                  05/06/2005                      *******
*************************************************************/

/***********************************
*****                  Includes      *******
***********************************/
#include <Mega32.h>
#include <stdio.h>
#include <Math.h>
#include <stdlib.h>
#include <delay.h>

/***********************************
*****        Definitions     *******
***********************************/

#define dataInput PIND.2             //  Receiver Input to edge triggered interrupt 0
#define LEDs PORTC                          //  Used for debugging

// discovery, counter, reset, ack for each type of packets
#define DSC_PKT                             1
#define CNT_PKT                             2
#define RST_PKT             3
#define ACK_DSC_PKT                 4
#define ACK_CNT_PKT                 5
#define ACK_RST_PKT                 6

// (lengths are in bits)
#define PKT_TYPE_LENGTH                     3
#define NODE_NUM_LENGTH             8
#define DATA_LENGTH                 15
#define PARITY_LENGTH               1
#define BYTE_LENGTH                                 8

#define PKT_BYTE_LEN                        4
#define TOTAL_NODES                                 5
#define DATA_BITS                                   0x7F
#define NUM_DISC_RETRY                      3

/***********************************
*****     Global Variables   *******
***********************************/

unsigned char count;
        // General indexing variable

unsigned char bitStream[5];
        // Used to parse input data
unsigned char bitNumber, byteNumber;                                        // pointer to
current bit/byte
unsigned char lastEdge, thisEdge, CLKedge, getData;

unsigned char data;
        // data received at base from node
unsigned char dataReady;
        // Set when the data is ready
unsigned char junkByte;
        // Set if the current byte is invalid
unsigned char junkCounter;
        // # of successive junk bytes
char sync;
        // Sync level where 0 = not syncing (receiving data)
unsigned int pktcount;
        // -1: not synced (waiting for sync pulse)
```

```
unsigned char recCount;
        // number of received bytes

enum pkt_format {PKT_TYPE=0, NODE_NUM, DATA1, DATA2};
enum base_states {INIT=0, DISCOVERY, COUNTING, RESET, DIE, IDLE};
enum error_states {WAITING=0, DSC_ACK_RECVD, DSC_ACK_ERROR, CNT_ACK_RECVD, RST_ACK_RECVD,
CNT_ACK_ERROR, RST_ACK_ERROR,RCV_ERROR, PKT_RECVD};
enum serialOptions{ OPT_DISC_NODES = 1, OPT_NODES_COUNT, OPT_RESET_COUNTS, OPT_COPYRIGHT,
MENU};

enum waitPktState {WAIT = 0, READY, DONE, FAIL};

unsigned char recv_packet[16], send_packet[4];                          // holds
packet type, node #, data, parity, and padding
unsigned char final_recv_packet[4];                                             // the
final received packet that is parsed from a given node
unsigned char CURRENT_STATE;                                                    // hold
current state of base station state machine

unsigned char curr_byte_num;                                                    //
current byte being stored from data stream
bit pktReady;
        // set when the given packet is ready for parsing

unsigned char node_status[TOTAL_NODES];                                         //
holds status (alive or dead) for each node in network
unsigned char second, minute, hour, nextpkt;                    // timer variables
used in state machine
unsigned int day, one_sec, timeout_counter;

unsigned char do_count, do_reset;                                               //
boolean variables set when counting or reset pkts are to be sent

unsigned char keyIn_timer;
        // keyboard polling for user input
unsigned char pktRecvd;
        // set when packet is received
unsigned char WAIT_PKT_STATE;

int p;
                // indexing variable

unsigned char retries;

int countPeople;
        // holds current count for a given node
unsigned short people_count[TOTAL_NODES];                               // holds count
for each node in network

unsigned char sendPacket, send_packet_count;                    // flags that are set
to send packet and # of times to be sent
unsigned char index, byte, halfcycle;                                   // used by
transmitter code

unsigned char aliveNodesCount;                          // # of alive nodes in
network


// Serial interface variables.
unsigned char inputChar;
unsigned char cmdString[20];

unsigned char char_count, cmd_ready;
unsigned char stateTask, commandNum;

/***********************************
*****   Function Prototypes  *******
***********************************/

void initialize(void);
                // sets up all the variables/functions
```

```
unsigned char parity(unsigned char x);
        // returns parity bit for given 7 bit input
unsigned char addParityBit(unsigned char x);                                    // adds
parity bit to given 7 bit input
unsigned char checkParity(unsigned char x, unsigned char byteNum);  // checks parity of
received packet

unsigned char checkPacketIntegrity();
        // determines if pkt is corrupted or not
void initRecvPacket();
                // clears variables for next incoming packet

short display_people_count();
        // sends count to serial connection

/************************************
*****       Interrupts       *******
************************************/

/* This interrupt is used to find the clock from the sync stream */
interrupt [EXT_INT0] void data_edge(void)
{
        //      Sample data after triggering
        thisEdge = dataInput;

        // If the detected period is too low (T < 40 us), then label
        // this byte as junk.
        if(TCNT1 < 10)
                junkByte = 1;           // Label the byte as junk.

        // 40 us < T < 360 us. This period occurs after a clock edge or a same-bit
        // boundary (e.g., 1-1 or 0-0).
        else if (TCNT1 < 90)
        {
                if (CLKedge)
                {
                        CLKedge = 0;
                        getData = 1;
                }
                else
                        CLKedge = 1;
        }
        // 360 us < T < 680 us.
        // This period occurs after an opposing-bit boundary ONLY (rising clock edge)
        else if (TCNT1 < 170)
        {
                CLKedge = 0;
                getData = 1;
        }
        // If the pulse period is very different, label the byte is junk.
        else
        {
                junkByte = 1;
                //      printf("\r\nj:%d %d",dataInput,TCNT1);
        }

        // Reset timer1
        TCNT1 = 0;

        // Rising clock edge means data is to be sampled
        if (getData)
        {
                getData = 0;

                // Shift bitstream down
                bitStream[0] >>= 1;
                bitStream[0] |= (bitStream[1] << 7);

                bitStream[1] >>= 1;
                bitStream[1] |= (bitStream[2] << 7);
```

```
                bitStream[2] >>= 1;
                bitStream[2] |= (bitStream[3] << 7);

                bitStream[3] >>= 1;
                bitStream[3] |= (bitStream[4] << 7);

                bitStream[4] >>= 1;
                bitStream[4] |= (dataInput << 7);

                // Insert the most recent bit into bitStream
                bitNumber++;
        }

        // If we have sync, and we've just updated bitStream and bitStream has
        // all 8 bits of data, then read the data. If the data has been
        // marked as junk for any reason, then set it to 0xff.
        if ((~CLKedge) && (sync != -1) && (bitNumber == 8))
        {
                // If the data is junk, make it 0xff.
                if (junkByte)
                {
                        data = 0xff;
                        //printf("%d ",bitStream[4]);
                        junkCounter++;
                }
                else
                {
                        data = bitStream[4];
                        //LEDs = ~data;
                        if(pktReady && curr_byte_num <16 && !pktRecvd && !sendPacket)
                        {
                                //printf("\r\ndata%d = %d", curr_byte_num, data);
                                recv_packet[curr_byte_num++] = data;
                                //LEDs = ~data;
                        }
                        junkCounter = 0;
                }

                byteNumber++;
                dataReady = 1;
                bitNumber = 0;
                junkByte = 0;

                // End of sync byte stream
                if (curr_byte_num == 7)
                {
                        sync = -1;      // look for sync again
                        //printf("\r\nO:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                        pktReady = 0;               // wait for after next sync stream
                        pktRecvd = 1;                           // current pkt is ready to be
decoded/parsed
                        curr_byte_num = 0;
                        for (p=0; p<4; p++)
                        {
                                final_recv_packet[p] = recv_packet[p+3];     // copy over
data to final recv packet
                        }
                }
        }
        // If we're searching for the sync stream...
        else if ((~CLKedge) && (sync == -1) && !pktRecvd && !sendPacket)
        {
                // If we've found the sync, stop searching and get ready for data.
                if ( (bitStream[0] == 0x00) && (bitStream[1] == 0xff) && (bitStream[2] ==
0xff) && (bitStream[3] == 0xaa) && (bitStream[4] == 0xff))
                {
                        sync = 0;
                        //TIMSK &= 0b11111101; // Stop sending 0xff's
                        bitNumber = 0;
```

```
                        byteNumber = 0;
                        pktReady =1;
                        //putsf("\r\nSYNCED");
                }
        }
        //LEDs = ~0xaa;
}

/* This function parses and prints the appropriate data for a given packet */
void parsePacket()
{
        switch(final_recv_packet[PKT_TYPE])
        {
        case ACK_DSC_PKT:
                printf("DSC node %d: LED1=%d, LED2=%d\n", final_recv_packet[NODE_NUM],
(((char)final_recv_packet[DATA1])<<1) >>1, (((char)final_recv_packet[DATA2])<<1) >>1 );
                break;
        case ACK_CNT_PKT:
                printf("CNT from %d:\t%d\n", final_recv_packet[NODE_NUM],
display_people_count());
                break;
        case ACK_RST_PKT:
                printf("RST from %d:\t%d\n", final_recv_packet[NODE_NUM],
final_recv_packet[DATA2]);
                break;
        default:
                break;
        }
}

//*********************************************************
//interrupt for UART RXC
interrupt [USART_RXC] void serialInterrupt(void)
{
        inputChar = UDR;
        UDR = inputChar;

        if(inputChar != '\r') //when did not encounter a carrage return
                cmdString[char_count++] = inputChar;
        else
        {
                putchar('\n');
                //terminate the string
                cmdString[char_count] = 0;
                cmd_ready = 1;
                UCSRB.7 = 0;
        }

}

/************************************
*****         Main           *******
************************************/

// create discovery packet for node i
void create_dsc_packet(unsigned char i)
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(DSC_PKT);
        send_packet[1] = addParityBit(i);
        send_packet[2] = addParityBit(7);
        send_packet[3] = addParityBit(9);
}

// create count packet for node i
void create_cnt_packet(unsigned char i)
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(CNT_PKT);
        send_packet[1] = addParityBit(i);
        send_packet[2] = addParityBit(0);
```

```
        send_packet[3] = addParityBit(0);
}

// create reset packet for node i
void create_rst_packet(unsigned char i)
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(RST_PKT);
        send_packet[1] = addParityBit(i);
        send_packet[2] = addParityBit(0x0f);
        send_packet[3] = addParityBit(0x0f);
}

// wait for packet from node i,parse the data, and check parity
void waitforPacket()
{
        switch(WAIT_PKT_STATE)
        {
        case WAIT:
                WAIT_PKT_STATE = WAIT;
                if(timeout_counter >= 2)
                {
                        WAIT_PKT_STATE = FAIL;
                        putsf("\n\rfail");
                        initRecvPacket();
                }

                if(pktReady)
                {
                        WAIT_PKT_STATE = READY;

                }

                break;

        case READY:
                WAIT_PKT_STATE = READY;
                if (pktRecvd)  // pkt received, ready to be decoded
                {
                        if(checkPacketIntegrity())
                        {
                                // parse the packet
                                parsePacket();
                                WAIT_PKT_STATE = DONE;
                        }
                        else
                        {
                                putsf("FAIL");
                                WAIT_PKT_STATE = FAIL;
                        }
                        printf("\r\nR:%d %d %d
%d",(char)final_recv_packet[0],(char)final_recv_packet[1],(char)final_recv_packet[2],(cha
r)final_recv_packet[3]);
                        initRecvPacket();

                        pktRecvd = 0;


                }
                break;

        case DONE:
                break;

        case FAIL:
                break;

        default:
                putsf("\r\ndefault");
                break;
        }
```

```
}

//*********************************************************
// This interrupt is used to transmit data and set the appropriate clocks
interrupt [TIM0_COMP] void timer0_com(void)
{
        // Three sync bytes (00-ff-ff), then 100 data bytes.
        // output 2kHz clock
        PORTB.1 = ~PORTB.1;
        if (sendPacket)
        {
                // Byte boundary.
                if (halfcycle == 16)
                {
                        halfcycle = 0;
                        // AGC
                        if (index < 20)               byte = 0xaa;
                        // SYNC
                        else if (index == 20)  byte = 0x00;
                        else if (index < 23 || index == 24) byte = 0xff;
                        else if (index == 23) byte = 0xaa;
                        // DATA send 16 bytes,
                        else if (index < 41) byte = send_packet[send_packet_count++];
                        // should never get into this else case
                        else byte = 0x00;

                        // send the packet 4 times.
                        if (send_packet_count == 4)
                                send_packet_count = 0;

                        index++;

                        if (index >= 41)
                        {
                                index = 0;
                                sendPacket = 0;
                                send_packet_count = 0;
                                // turns off the transmitter
                                PORTB.0 = 0;
                        }
                }
                /* MANCHESTER ENCODING */
                // On the falling edge of the clock, the output matches the data.
                if(sendPacket)
                {
                        if (halfcycle & 1)
                                PORTB.0 = (byte >> (halfcycle >> 1)) & 0x01;
                        // On the rising edge of the clock, the output is the data
inverted.
                        else
                                PORTB.0 = (~(byte >> (halfcycle >> 1))) & 0x01;
                        halfcycle++;
                }
        }

        /* Timer and Keyboard counters */
        one_sec++;
        keyIn_timer++;

        if(keyIn_timer >=195)
        {
                keyIn_timer = 0;
                //keyboardInput();
        }

        if(one_sec >=3906)
        {
                one_sec = 0;
                second++;

                timeout_counter++;
```

```
                // count flag
                if(second%2 == 0)
                        do_count = 1;

        }
        if(second >= 60)
        {
                second = 0;
                minute++;
        }
        if(minute >=60)
        {
                hour++;
                minute=0;
                //do_reset = 1;
        }
}

/* This function initializes the appropriate variables for the next transmission */
void initRecv()
{
        sync = -1;      // look for sync again
        //printf("\r\nO:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
        pktReady = 0;                // wait for after next sync stream
        pktRecvd = 0;                           // current pkt is ready to be decoded/parsed
        curr_byte_num = 0;
}

/* Turns off receiver interrupt, Sends packet, waits for response, turns on receiver
interrupt */
void send_and_wait_packet()
{
        GICR = 0b00000000;
        sendPacket = 1;
        while (sendPacket){}
        initRecv();
        GICR = 0b01000000;
        WAIT_PKT_STATE = WAIT;
        timeout_counter = 0;
        while(WAIT_PKT_STATE != DONE && WAIT_PKT_STATE != FAIL )
        {
                waitforPacket();
        }

}

/* Creates discovery packet for each node, transmits it, waits for response, retries if
necessary */
void discover_nodes()
{
        int i;

        aliveNodesCount = 0;

        for(i = 0; i<TOTAL_NODES; i++ )
                node_status[i]=0;

        for (i=0; i<TOTAL_NODES; i++)
                        {
                                retries = 0;
                                /**++++ retry if fails **/
                                while(!node_status[i] && retries < NUM_DISC_RETRY)
                                {
                                send_packet_count=0;
                                // create discovery packet for each node
                                create_dsc_packet(i);
                                // broadcast discovery packet
                                printf("\r\nS:%d %d %d
%d",send_packet[0],send_packet[1],send_packet[2],send_packet[3]);
```

```
                                // SEND A PACKET
                                send_and_wait_packet();

                                // set the node status to be alive == 1, dead == 0
                                if(WAIT_PKT_STATE == DONE)
                                {


                                        node_status[i] = 1;
                                        aliveNodesCount++;
                                }
                                else
                                        node_status[i] = 0;

                                retries++;
                                }
                                delay_ms(100);
                        }
}

/* Creates reset packet, sends it to each node, waits for response */
void reset_nodes()
{
        int i;
                for (i=0; i<TOTAL_NODES; i++)
                        {
                                if(node_status[i])
                                {

                                // create reset packet for each node
                                create_rst_packet(i);
                                // broadcast reset packet
                                send_and_wait_packet();
                                }
                                delay_ms(100);


                        }
}

/* creates count packet for each node, sends it and waits */
void count_nodes()
{
        int i;
                        for (i=0; i<TOTAL_NODES; i++)
                        {
                                if(node_status[i])
                                {
                                        // create count packet for each node
                                        create_cnt_packet(i);
                                        // broadcast count packet
                                        // SEND THE PACKET
                                        send_and_wait_packet();
                                }
                                delay_ms(100);
                        }
}

/* Gets keyboard command from user */
void get_cmd_init(void)
{
   char_count=0;
   cmd_ready=0;
   UCSRB.7=1;
}

//*********************************************************
/* Prints User Interface menu */
void menu(void)
{
        putsf("\n\n\rPeople Counter Basestation \r\n");
```

```
        printf("Number of Alive Nodes: %d\r\n", aliveNodesCount);
        putsf("Options:\r");
        putsf("1. Discover Nodes\r");
        putsf("2. Get Current People Count from All Nodes\r");
        putsf("3. Reset All Nodes' Count to Zero\r");
        putsf("4. Copyright Information\r");
}

/* Prints copyright info */
void copyright(void)
{
        putsf("People Counter basestation and node \n\rBy Arun Israel and Eric Lee\n\rat
Cornell University Spring 2005\n\r");
        putsf("Code Available upon Request\r\n");
}

//process security manager's task
void processTasks(void)
{

        switch (stateTask)
        {
                //at the menu state
                case MENU:
                        commandNum = cmdString[0] - '0';
                        if (cmdString[0] == 0 || cmdString == 0)
                        {
                                menu();
                                break;
                        }

                        switch(commandNum)
                        {
                                case OPT_DISC_NODES:
                                        putsf("=================\r\n");
                                        discover_nodes();
                                stateTask = MENU;
                                menu();

                                        break;
                                case OPT_NODES_COUNT:
                                        putsf("=================\r\n");
                                        count_nodes();
                                stateTask = MENU;
                                menu();
                                        break;
                                case OPT_RESET_COUNTS:
                                        putsf("=================\r\n");
                                        reset_nodes();
                                stateTask = MENU;
                                menu();
                                        break;
                                case OPT_COPYRIGHT:
                                        putsf("=================\r\n");
                                        copyright();
                                stateTask = MENU;
                                menu();
                                        break;
                        }
                        break;
                default:
                        //wrong command, return some sensable message to SM.
                        putsf("Invalid command");
                        break;
        }

        get_cmd_init();
}


//*******************************************************
```

```
/************************************
*****  Function Definitions  *******
************************************/

void main(void)
{
        initialize();
        while(1)
        {
                if (cmd_ready)
                {
                        processTasks();
                }

                switch(CURRENT_STATE)
                {
                // base station discovers each node
                case DISCOVERY:
                        // send out discovery packet to each node

                        discover_nodes();
                        CURRENT_STATE = IDLE;
                        break;

                // base station requests count from each node
                case COUNTING:
                        do_count = 0;
                        // send out count request packets to each node
            count_nodes();
                        CURRENT_STATE = IDLE;
                        break;

                // base station resets each node
                case RESET:
                        do_reset = 0;
                        // send out reset packet to each node
                        reset_nodes();
                        CURRENT_STATE = IDLE;
                        break;
                case IDLE:
                        if(do_count) CURRENT_STATE = COUNTING;
                        else if (do_reset) CURRENT_STATE = RESET;
                        else CURRENT_STATE = IDLE;
                        break;
                default:
                        break;

                }

        }
}
/************************************
*****     Initialization     *******
************************************/

void initialize(void)
{
        /**** Port initialization ****/
        DDRD = 0x00;   // D.2 is data input from receiver
        DDRC = 0xff;   // Port C is LED output for debugging

        //set up the ports
        DDRB.0=1;       // PORTB.0 and PORTB.1 are outputs for data to transmitter and
clock
        DDRB.1=1;
        PORTB.0=0;
        PORTB.1=0;

        LEDs = 0xff;   // All off initially

        /**** Interrupt initialization ****/
```

```
        MCUCR = 0b00000001;    // Any edge on INT0 triggers interrupt
        GICR = 0b01000000;     // Enable INT0

        /* Timer 1 is used for synchronizing with the transponder's data clock,
        which should run at around (125e3/64) = 1.95 KHz, which means that
        edges appear at 3.91 KHz (no opposing bit boundary) or 1.95KHz
        (at opposing bit boundaries). */
        TCCR1B = 0b00000011;   // CLK/64 (250 KHz)

        /* Timer 0 is used to output 0xff data points at the same frequency as
        the usual data rate when the base is searching for the sync stream. */

        //set up timer 0
        OCR0 = 64;                    //f = 240 Hz (almost 238.5 Hz, the data rate)
        TCCR0=0b00001011;             //CTC mode
        TIMSK=0b00000010;             //enable compare match

        /**** USART initialization ****/
        UCSRB = 0b00011000 ; // TXC, RXC enabled
        UBRRL = 103 ; // 57.6 kbaud @ 16MHz

        /***** Packet init *****/
        curr_byte_num = 0;
        pktReady =0;
        countPeople =0;
        CURRENT_STATE = DISCOVERY;
        stateTask = MENU;

        send_packet_count = 0;
        //PacketSent = 0;
        sendPacket = 0;
        /**** Variable initialization ****/
        nextpkt = 0;
        lastEdge = 0;
        CLKedge = 0;
        bitNumber = 0;
        pktcount = 0;
        byteNumber = 0;
        junkByte = 0;
        do_count = do_reset = 0;
        for (count = 0; count < 3; count++)
                bitStream[count] = 0;

        //clock init
        one_sec = second = minute = hour = day = 0;
        timeout_counter =0;

        WAIT_PKT_STATE = WAIT;
        pktRecvd = 0;
        data = 0;
        dataReady = 0;
        sync = -1;             // Initially, we don't have sync
        recCount = 0;
        get_cmd_init();
        #asm("sei");
}

/* Sends people count info to serial connection */
short display_people_count()
{
        short people_count = 0;

        short mask14th_bit = 0x2000;

        people_count = ((unsigned short)final_recv_packet[2]) << 7 | final_recv_packet[3];
        //when people count is a negative number
        if(people_count & mask14th_bit)
                people_count |= 0xC000;

        return people_count;
}
```

```
//**************************************************************************
//Compute the odd parity of a character.  If the character has even numbers
//of 1, the parity bit will be 1, otherwise it will be 0.
//**************************************************************************

unsigned char parity(unsigned char x)//calculate the parity of a character
{
        unsigned char temp, i;
        temp=1;
        for(i=0;i<8;i++)
        {
                temp=temp^(x&1);
                x>>=1;
        }
        return temp;
}


// ADD parity bit to the packet
unsigned char addParityBit(unsigned char x)
{
        return (x & 0x7f) | (parity(x) << 7);
}

// checks parity of given byte in packet
unsigned char checkParity(unsigned char x, unsigned char byteNum)
{
        unsigned char data, parityBit;

        data = x & DATA_BITS;
        parityBit = (x >> 7) & 1;
        // remove parity bit from the byte.
        final_recv_packet[byteNum] = data;
        return parityBit == parity(data);
}

// checks integrity of received packet
unsigned char checkPacketIntegrity()
{
        int i = 0;
        for(i=0;i<PKT_BYTE_LEN;i++)
        {
                if(!checkParity(final_recv_packet[i], i))
                        return 0;
        }
        return 1;
}

// clears variables for next packet
void initRecvPacket()
{
        int i = 0;
        for(i=0;i<PKT_BYTE_LEN;i++)
        {
                final_recv_packet[i] = 0;

        }
        for(i=0;i<16;i++)
        {
                recv_packet[i] = 255;

        }
}
```

Wireless People Counting Network – Node Source Code:

```
/*************************************************************
*****       HCI Wireless People Counting Project       *******
*****               Node Code                          *******
*****                                                  *******
*****           By Arun Israel and Eric Lee            *******
*****                  05/06/05                        *******
*************************************************************/

/* This code is for the Atmel Mega32 Microcontroller */
/************************************
*****                  Includes      *******
************************************/
#include <Mega32.h>
#include <stdio.h>
#include <delay.h>

/************************************
*****        Definitions     *******
************************************/

#define dataInput PIND.2
#define debugOut PORTA.0
#define LEDs PORTC

// discovery, counter, reset, ack for each type of packets
#define DSC_PKT                              1
#define CNT_PKT                              2
#define RST_PKT             3
#define ACK_DSC_PKT                  4
#define ACK_CNT_PKT                  5
#define ACK_RST_PKT                  6

// (lengths are in bits)
#define PKT_TYPE_LENGTH                      3
#define NODE_NUM_LENGTH              8
#define DATA_LENGTH                  15
#define PARITY_LENGTH                1
#define BYTE_LENGTH                                  8

#define PKT_BYTE_NUM                         4

// some constants
#define DATA_BITS                    0x7F

// ADC Constants
#define SENSTIVITY 0.05
#define ADC_UPD_BOUND 0.01

#define ADC_AVERAGES 10
#define ADC_TIMEOUT 30
#define ADC_DELAY 38
#define ADC_RESET_COUNT 6
#define REAL_TRIGGER 4
#define THIS_NODE_NUM 3                      // CHANGE THIS VARIABLE FOR EACH NODE
PROGRAMMED SO TWO NODES DON'T SHARE THE SAME NUMBER

#define NUM_LEDS 2


/************************************
*****     Global Variables   *******
************************************/

unsigned char count;                                 // General indexing variable

unsigned char bitStream[5];                          // Used to parse input data
unsigned char bitNumber, byteNumber;  // pointer to current bit/byte num
unsigned char lastEdge, thisEdge, CLKedge, getData;
```

```
unsigned char data;
       // data byte received at the Base Station from the

unsigned char dataReady;              // True when the data is ready
unsigned char junkByte;               // True if the current byte is invalid
unsigned char junkCounter;    // # of successive junk bytes since last sync
char sync;                            // Sync level
// 0: not syncing (receiving data)
unsigned int pktcount;                                                  // -1: not
synced (waiting for sync pulse)
unsigned char recCount;

enum pkt_format {PKT_TYPE=0, NODE_NUM, DATA1, DATA2};
enum base_states {INIT=0, WAIT_FOR_PACKET,DISCOVERY_ACK, COUNTING_ACK, RESET_ACK,
DIE_ACK};
enum error_states {WAITING=0, DSC_ACK_RECVD, DSC_ACK_ERROR, CNT_ACK_RECVD, RST_ACK_RECVD,
CNT_ACK_ERROR, RST_ACK_ERROR,RCV_ERROR, PKT_RECVD};

enum waitPktState {WAIT = 0, READY, DONE, FAIL};

unsigned char recv_packet[16], send_packet[PKT_BYTE_NUM];        // holds packet type,
node #, data, parity, and padding- checksum?
unsigned char final_recv_packet[PKT_BYTE_NUM];
unsigned char CURRENT_STATE;        // hold state of node i

unsigned char curr_byte_num;
bit pktReady;

unsigned char second, minute, hour, nextpkt;
unsigned int day, one_sec, timeout_counter;

unsigned char pktRecvd;
unsigned char WAIT_PKT_STATE;


int p;
int countPeople;

unsigned char sendPacket, send_packet_count;
unsigned char index, byte, halfcycle;

/* ADC VARIABLES */
unsigned char LED_upd_flag;
int  LED_ADC[NUM_LEDS], LED_TRIG[NUM_LEDS];
float LED_BASE_ADC[NUM_LEDS], LED_BASE_UPD_ADC[NUM_LEDS], LED_DELTA_ADC[NUM_LEDS],
LED_DELTA_UPD_ADC[NUM_LEDS];

enum ADCStates{NO_MOTION,POSSIBLE_FORWARD,POSSIBLE_BACKWARD,FORWARDS,BACKWARDS};
short people_count;
unsigned char ADCdelay, ADCState, ADCwaitCounter;


/***********************************
*****   Function Prototypes  *******
***********************************/

void initialize(void);


// PROTOCOL PROTOTYPES
unsigned char parity(unsigned char x);
unsigned char addParityBit(unsigned char x);
unsigned char checkParity(unsigned char x, unsigned char byteNum);

unsigned char checkPacketIntegrity();


/* ADC PROTOTYPES */
void ADCReading();
void ADCStateMachine();
void ADCCompare();
```

```
void ADCInitAvg();

/***********************************
*****       Interrupts       *******
***********************************/

// Finds clock from sync stream
interrupt [EXT_INT0] void data_edge(void)
{
        thisEdge = dataInput;

        // If the detected period is too low (T < 200 us), then label
        // this byte as junk.
        if(TCNT1 < 10)
                junkByte = 1;          // Label the byte as junk.

        // 200 us < T < 384 us. This period occurs after a clock edge or a same-bit
        // boundary (e.g., 1-1 or 0-0).
        else if (TCNT1 < 90)
        {
                // If we're at a clock edge...
                if (CLKedge)
                {
                        CLKedge = 0;
                        getData = 1;
                }
                else
                        CLKedge = 1;
        }
        // 384 us < T < 600 us.
        // This period occurs after an opposing-bit boundary ONLY.
        else if (TCNT1 < 170)
        {
                CLKedge = 0;
                getData = 1;
        }
        // If the pulse period is too different from any legal values, the byte is junk.
        else
        {
                junkByte = 1;
        }

        // Reset timer1
        TCNT1 = 0;

        // If we're on a rising clock edge, it's time to get the data
        if (getData)
        {
                getData = 0;

                // Shift bitstream down
                bitStream[0] >>= 1;
                bitStream[0] |= (bitStream[1] << 7);

                bitStream[1] >>= 1;
                bitStream[1] |= (bitStream[2] << 7);

                bitStream[2] >>= 1;
                bitStream[2] |= (bitStream[3] << 7);

                bitStream[3] >>= 1;
                bitStream[3] |= (bitStream[4] << 7);

                bitStream[4] >>= 1;
                bitStream[4] |= ((dataInput) << 7);

                // Insert the most recent bit into bitStream
                bitNumber++;
        }

        // If we have sync, and we've just updated bitStream and bitStream has
```

```
        // all 8 bits of data, then read the data. If the data has been
        // marked as junk for any reason, then set it to 0xff.
        if ((~CLKedge) && (sync != -1) && (bitNumber == 8))
        {
                // If the data is junk, make it 0xff.
                if (junkByte)
                {
                        data = 0xff;
                        junkCounter++;
                }
                else
                {
                        data = bitStream[4];
                        //LEDs = ~data;
                        if(pktReady && curr_byte_num<16 && !pktRecvd && !sendPacket)
                        {
                                //printf("\r\ndata%d = %d", curr_byte_num, data);
                                recv_packet[curr_byte_num++] = data;
                                //LEDs = ~data;
                        }
                        junkCounter = 0;
                }

                byteNumber++;
                dataReady = 1;
                bitNumber = 0;
                junkByte = 0;

                if (curr_byte_num == 7)
                {
                        sync = -1;      // look for sync again
                        //      printf("\r\nO:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                        pktReady = 0;   // wait for after next sync stream
                        pktRecvd = 1;  // current pkt is ready to be decoded/parsed
                        curr_byte_num = 0;
                        for (p=0; p<4; p++)
                        {
                                final_recv_packet[p] = recv_packet[p+3];      // copy over
data to final recv packet
                        }
                }
        }
        // If we're searching for the sync stream...
        else if ((~CLKedge) && (sync == -1)&& !pktRecvd && !sendPacket)
        {

                // If we've found the sync, stop searching and get ready for data.
                if ( (bitStream[0] == 0x00) && (bitStream[1] == 0xff) && (bitStream[2] ==
0xff) && (bitStream[3] == 0xaa) && (bitStream[4] == 0xff))
                {
                        sync = 0;
                        //TIMSK &= 0b11111101; // Stop sending 0xff's
                        bitNumber = 0;
                        byteNumber = 0;
                        pktReady =1;
                        //putsf("\r\nSYNCED");
                }
        }
}

/***********************************
*****         Main           *******
***********************************/

// create discovery packet for this node
void create_dsc_ack_packet()
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(ACK_DSC_PKT);
        send_packet[1] = addParityBit(THIS_NODE_NUM);
```

```
        send_packet[2] = addParityBit(((int)LED_BASE_ADC[0]-255/2) & DATA_BITS);
        send_packet[3] = addParityBit(((int)LED_BASE_ADC[1]-255/2) & DATA_BITS);
}

// create count packet for node i
void create_cnt_ack_packet()
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(ACK_CNT_PKT);
        send_packet[1] = addParityBit(THIS_NODE_NUM);
        send_packet[2] = addParityBit((unsigned char)((people_count & 0x3F80)>>7));
        send_packet[3] = addParityBit((unsigned char)(people_count & DATA_BITS));
}

// create reset packet for node i
void create_rst_ack_packet()
{
        // create bytes for each piece of packet
        send_packet[0] = addParityBit(ACK_RST_PKT);
        send_packet[1] = addParityBit(THIS_NODE_NUM);
        // send basestation the current count, basestation will check if it's indeed 0
        send_packet[2] = addParityBit((unsigned char)((people_count & 0x3F80)>>7));
        send_packet[3] = addParityBit((unsigned char)(people_count & DATA_BITS));
}

// wait for packet from node i,parse the data, and check parity
void waitforPacket()
{
        //timeout_counter = 0;
        switch(WAIT_PKT_STATE)
        {
        case WAIT:
                WAIT_PKT_STATE = WAIT;
                if(ADCdelay >= ADC_DELAY)
                {
                        ADCdelay = 0;
                        ADCReading();
                        ADCCompare();
                        ADCStateMachine();

                }
                //putsf("wailt");
                if(pktReady)
                {

                        WAIT_PKT_STATE = READY;
                        //putsf("ready!\r\n");
                }
                break;

        case READY:
                WAIT_PKT_STATE = READY;
                if (pktRecvd)  // pkt received, ready to be decoded
                {       /*
                                printf("\r\nR1:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                                printf("\r\nR2:%d %d %d
%d",recv_packet[4],recv_packet[5],recv_packet[6],recv_packet[7]);
                                printf("\r\nR3:%d %d %d
%d",recv_packet[8],recv_packet[9],recv_packet[10],recv_packet[11]);
                                printf("\r\nR4:%d %d %d
%d",recv_packet[12],recv_packet[13],recv_packet[14],recv_packet[15]);
                                */

                        if(checkPacketIntegrity()) // check integrity
                        {
                                //LEDs = 0xaa;

                                //printf("\r\nG:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                                WAIT_PKT_STATE = DONE;
```

```
                        }
                        else
                        {
                                LEDs = 0x77;
                                //printf("\r\nB:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                                WAIT_PKT_STATE = FAIL;

                                //return RCV_ERROR; // ERROR;
                        }
                        printf("\r\nR:%d %d %d
%d",final_recv_packet[0],final_recv_packet[1],final_recv_packet[2],final_recv_packet[3]);

                        pktRecvd = 0;


                }
                break;

        case DONE:
                //LEDs = ~recv_packet[2];

                //      printf("\r\nDONE:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                //      pktRecvd = 0;
                break;

        case FAIL:
                //printf("\r\nFAIL:%d %d %d
%d",recv_packet[0],recv_packet[1],recv_packet[2],recv_packet[3]);
                //      pktRecvd = 0;
                break;

        default:
                putsf("\r\ndefault");
                break;
        }
}

// Take readings off each channel for both LEDs
void ADCReading()
{
        ////////////////////// LED1 ///////////////////////
        ADMUX =  0b01100000; //read voltage on A.0 for LED1
        ADCSRA.6 = 1;  // start conversion

        //ADCSRA.6 bit will be clear after the conversion is done.
        while(ADCSRA.6 == 1){}
        LED_ADC[0] = ADCH;     //record the ADC value

        ////////////////////// LED2 ///////////////////////
        ADMUX =  0b01100001; //read voltage on A.1 for LED2
        ADCSRA.6 = 1;
        while(ADCSRA.6 == 1){}
        LED_ADC[1] = ADCH;     //record the ADC value
}

// Compares current readings to base readings to see if trigger condition is met
void ADCCompare()
{
        int i;
        // compare current reading to base
        // if current reading > +/- 10 delta units from base, set led_triggered
        for(i=0;i<NUM_LEDS;i++)
        {
                if ((LED_ADC[i] > LED_BASE_ADC[i]+LED_DELTA_ADC[i]) || (LED_ADC[i] <
LED_BASE_ADC[i]-LED_DELTA_ADC[i]))
                {
                        // set led triggered state, reset base reading to current reading,
modify delta?
                        LED_TRIG[i]++;
```

```
                        if(LED_TRIG[0] >= REAL_TRIGGER) PORTC.0 = 0;
                        if (LED_TRIG[1] >= REAL_TRIGGER) PORTC.1 = 0;
                }
                else
                {
                        LED_TRIG[i] = 0;
                        if(i == 0) PORTC.0 = 1;
                        if (i == 1) PORTC.1 = 1;
                }
        }

}

// state machine to detect people movement across both LED fields of view
void ADCStateMachine()
{
        ADCwaitCounter++;
        switch(ADCState)
        {
        case NO_MOTION: // neither trigger condition is met
                PORTC.7 =1;
                PORTC.6 = 1;
                if (LED_TRIG[0] >= REAL_TRIGGER && LED_TRIG[1] == 0)
                {
                        ADCState = POSSIBLE_FORWARD;
                        ADCwaitCounter = 0;
                }
                else if (LED_TRIG[1] >= REAL_TRIGGER && LED_TRIG[0] == 0)
                {
                        ADCState = POSSIBLE_BACKWARD;
                        ADCwaitCounter = 0;
                }
                else
                        ADCState = NO_MOTION;
                break;

        case POSSIBLE_FORWARD: // first LED has been triggered
                if (ADCwaitCounter >= ADC_TIMEOUT)    // wait for movement through second
field, if no movement then no motion
                {
                        ADCState = NO_MOTION;
                }
                else if (LED_TRIG[1] >= REAL_TRIGGER) // second movement occurred goto
forwards state
                {
                        ADCState = FORWARDS;
                        ADCwaitCounter = 0;
                }
                else
                        ADCState = POSSIBLE_FORWARD;
                break;

        case POSSIBLE_BACKWARD:   // second led has been triggered
                if (ADCwaitCounter >= ADC_TIMEOUT)    // wait for movement into first
field, if no movement, then no motion
                {
                        ADCState = NO_MOTION;
                }
                else if (LED_TRIG[0] >= REAL_TRIGGER) // second movement occurred goto
backwards state
                {
                        ADCState = BACKWARDS;
                        ADCwaitCounter = 0;
                }
                else
                        ADCState = POSSIBLE_BACKWARD;
                break;

        case FORWARDS:
                PORTC.7 = 0;
                //if (ADCwaitCounter >= ADC_RESET_COUNT)
```

```
                if (LED_TRIG[0] == 0 && LED_TRIG[1] == 0)     // leave this state when
person leaves both fields
                {
                        people_count++;
                        ADCState = NO_MOTION;
                }
                else
                {
                        ADCState = FORWARDS;
                }
                break;


        case BACKWARDS:
                PORTC.6 = 0;
                //if (ADCwaitCounter >= ADC_RESET_COUNT)
                if (LED_TRIG[0] == 0 && LED_TRIG[1] == 0)     // leave this state when
person leaves both fields
                {
                        people_count--;
                        ADCState = NO_MOTION;
                }
                else
                {
                        ADCState = BACKWARDS;
                }

                break;

        default:
                // should never be here!

                break;
        }
}

// Update average to accomodate for lighting changes
void ADCAverageUpdate()
{
        int i = 0;
        int j = 0;
        for (i = 0; i<NUM_LEDS;i++)
        {
                LED_BASE_UPD_ADC[i] = 0;
        }

        for(i=0;i<ADC_AVERAGES;i++)
        {
                ADCReading();
                delay_ms(2);
                for(j=0;j<NUM_LEDS;j++)
                {
                        LED_BASE_UPD_ADC[j] += LED_ADC[j];
                }
        }

        for(i=0;i<NUM_LEDS;i++)
        {
                LED_BASE_UPD_ADC[i] /= (float)ADC_AVERAGES;
                //LED_DELTA_UPD_ADC[i] = SENSTIVITY*LED_BASE_UPD_ADC[i];
        }

        LED_upd_flag = 1;
        // the update logic here
        for(i=0;i<NUM_LEDS;i++)
        {
                // if the new adc reading is with in 1% of the previous reading.
                if ((LED_BASE_UPD_ADC[i] < LED_BASE_ADC[i]+LED_BASE_ADC[i]*ADC_UPD_BOUND)
&& (LED_BASE_UPD_ADC[i] > LED_BASE_ADC[i]-LED_BASE_ADC[i]*ADC_UPD_BOUND))
                        ;
                else
```

```
                        LED_upd_flag = 0;
        }

        // update the average if both LEDs agrees to update.
        if(LED_upd_flag)
        {
                for(i=0;i<NUM_LEDS;i++)
                {
                        LED_BASE_ADC[i] = LED_BASE_UPD_ADC[i];
                        LED_DELTA_ADC[i] = SENSTIVITY*LED_BASE_ADC[i];
                }
        }

}


// get the initial base reading of the adc
void ADCInitAvg()
{
        int i = 0;
        int j = 0;
        for (i = 0; i<NUM_LEDS;i++)
        {
                LED_BASE_ADC[i] = 0;
        }

        for(i=0;i<ADC_AVERAGES;i++)
        {
                ADCReading();
                delay_ms(2);
                for(j=0;j<NUM_LEDS;j++)
                {
                        LED_BASE_ADC[j] += LED_ADC[j];
                }

        }

        for(i=0;i<NUM_LEDS;i++)
        {
                LED_BASE_ADC[i] /= (float)ADC_AVERAGES;
                LED_DELTA_ADC[i] = SENSTIVITY*LED_BASE_ADC[i];
                LED_TRIG[i] = 0;
        }
}

//*********************************************************
//timer 0 compare ISR
interrupt [TIM0_COMP] void timer0_com(void)
{
        // Three sync bytes (00-ff-ff), then 100 data bytes.
        // output 2kHz clock
        PORTB.1 = ~PORTB.1;

        // Send packet
        if (sendPacket)
        {
                if (halfcycle == 16)
                {
                        halfcycle = 0;
                        // AGC
                        if (index < 20)               byte = 0xaa;
                        // SYNC
                        else if (index == 20)  byte = 0x00;
                        else if (index < 23 || index == 24) byte = 0xff;
                        else if (index == 23) byte = 0xaa;
                        // DATA send 16 bytes,
                        else if (index < 41) byte = send_packet[send_packet_count++];
                        // should never get into this else case
                        else byte = 0x00;

                        // resend the packet 4 times.
```

```
                        if (send_packet_count == 4)
                                send_packet_count = 0;

                        index++;

                        if (index >= 41)
                        {
                                index = 0;
                                sendPacket = 0;
                                send_packet_count = 0;
                                // transmitter sends low/0
                                PORTB.0 = 0;
                        }
                }


                if(sendPacket)
                {
                        // On the falling edge of the clock, the output matches the data.
                        if (halfcycle & 1)
                                PORTB.0 = (byte >> (halfcycle >> 1)) & 0x01;
                        // On the rising edge of the clock, the output is the data
inverted.
                        else
                                PORTB.0 = (~(byte >> (halfcycle >> 1))) & 0x01;
                        halfcycle++;
                }
        }

        ADCdelay++;

        timeout_counter++;
        one_sec++;

        // one_sec is based on the 2KHz clock for this interrupt
        if(one_sec >=3906)
        {
                one_sec = 0;
                second++;
        }
        if(second >= 60)
        {
                second = 0;
                minute++;
        }
        if(minute >=60)
        {
                hour++;
                minute=0;
        }


}
/* Reset the receive variables.*/
void initRecv()
{
        sync = -1;      // look for sync again
        pktReady = 0;                // wait for after next sync stream
        pktRecvd = 0;                        // current pkt is ready to be decoded/parsed
        curr_byte_num = 0;
}

/* Send packet in send_packet array to the basestation*/
void send_ack_packet()
{
        // disable the receive interrupt
        GICR = 0b00000000;
        delay_ms(100);
        // semaphore
        sendPacket = 1;
        while (sendPacket){}
```

```
        initRecv();
        // enable the receive interrupt
        GICR = 0b01000000;

}
//************************************************************
/************************************
*****  Function Definitions  *******
************************************/

void main(void)
{
        initialize();
        //      ADCInitAvg();
        while(1)
        {

                switch(CURRENT_STATE)
                {
                        // base station discovers each node
                case WAIT_FOR_PACKET:
                        initRecv();
                        // listen for packet from base station

                        WAIT_PKT_STATE = WAIT;
                        while(WAIT_PKT_STATE != DONE && WAIT_PKT_STATE != FAIL )
                        {
                                waitforPacket();
                        }
                        // if pkt destined for this node, parse it and send response
                        if(final_recv_packet[NODE_NUM] == THIS_NODE_NUM && WAIT_PKT_STATE
== DONE)
                        {
                                switch (final_recv_packet[PKT_TYPE])
                                {
                                case DSC_PKT:
                                        CURRENT_STATE = DISCOVERY_ACK;
                                        break;
                                case CNT_PKT:
                                        CURRENT_STATE = COUNTING_ACK;
                                        break;
                                case RST_PKT:
                                        CURRENT_STATE = RESET_ACK;
                                        break;
                                default:
                                        CURRENT_STATE = WAIT_FOR_PACKET;
                                        break;
                                }
                        }
                        break;
                case DISCOVERY_ACK:
                        // create and send out discovery ack packet to base station
                        ADCInitAvg();
                        create_dsc_ack_packet();
                        send_ack_packet();
                        CURRENT_STATE = WAIT_FOR_PACKET;
                        break;

                case COUNTING_ACK:
                        // create count ack packet to send to base station
                        create_cnt_ack_packet();
                        send_ack_packet();
                        CURRENT_STATE = WAIT_FOR_PACKET;
                        break;

                        // create reset ack pkt to send to base station
                case RESET_ACK:
                        people_count = 0;
                        create_rst_ack_packet();
                        send_ack_packet();
                        CURRENT_STATE = WAIT_FOR_PACKET;
```

```
                        break;

                default:
                        break;
                }
        }
}
/***********************************
*****     Initialization     *******
***********************************/

/* Initialize ADC State Machine */
void ADCinit()
{
        ADCState = NO_MOTION;
        ADCdelay = 0;
        ADCwaitCounter = 0;
}

void initialize(void)
{
        /**** Port initialization ****/
        DDRD = 0x00;   // D.2 is data input
        DDRC = 0xff;   // Port C is LED output

        //set up the ports
        DDRB.0=1;       // PORT.0 and PORT.1 are outputs
        DDRB.1=1;
        //DDRB.2=0; //PORTB.2 is an input
        PORTB.0=0;
        PORTB.1=0;
        //PORTB.2=0; //no pull-up

        LEDs = 0xff;   // All off initially

        /**** Interrupt initialization ****/
        MCUCR = 0b00000001;    // Any edge on INT0 triggers interrupt
        GICR = 0b01000000;     // Enable INT0

        /* Timer 1 is used for synchronizing with the transponder's data clock,
        which should run at around (125e3/64) = 1.95 KHz, which means that
        edges appear at 3.91 KHz (no opposing bit boundary) or 1.95KHz
        (at opposing bit boundaries). */
        TCCR1B = 0b00000011;   // CLK/64 (250 KHz)

        /* Timer 0 is used to output 0xff data points at the same frequency as
        the usual data rate when the base is searching for the sync stream. */
        OCR0 = 64;     //f = 240 Hz (almost 238.5 Hz, the data rate)
        TCCR0=0b00001011;             //CTC mode
        TIMSK=0b00000010;             //enable compare match

        /**** USART initialization ****/
        UCSRB = 0b00011000 ; // TXC, RXC enabled
        UBRRL = 103 ; // 57.6 kbaud @ 16MHz

        /***** Packet init *****/

        curr_byte_num = 0;
        pktReady =0;
        countPeople =0;
        CURRENT_STATE = WAIT_FOR_PACKET;
        send_packet_count = 0;
        sendPacket = 0;
        /**** Variable initialization ****/
        nextpkt = 0;
        lastEdge = 0;
        CLKedge = 0;
        bitNumber = 0;
        pktcount = 0;
        byteNumber = 0;
        junkByte = 0;
```

```
        for (count = 0; count < 5; count++)
                bitStream[count] = 0;

        //clock init
        one_sec = second = minute = hour = day = 0;
        timeout_counter =0;

        WAIT_PKT_STATE = WAIT;
        pktRecvd = 0;
        data = 0;
        dataReady = 0;
        sync = -1;              // Initially, we don't have sync
        recCount = 0;
        ADCinit();
        //read voltage on A.0
        ADMUX =  0b01100000;
        //ADC on
        ADCSRA = 0b11000111;
#asm("sei");
}

//*********************************************************************
//Compute the odd parity of a character.  If the character has even numbers
//of 1, the parity bit will be 1, otherwise it will be 0.
//*********************************************************************
unsigned char parity(unsigned char x)//calculate the parity of a character
{
        unsigned char temp, i;
        temp=1;
        for(i=0;i<8;i++)
        {
                temp=temp^(x&1);
                x>>=1;
        }
        return temp & 1;
}


// ADD parity bit to the packet
unsigned char addParityBit(unsigned char x)
{
        return (x & 0x7f) | (parity(x) << 7);
}

// check parity bit for each byte in packet
unsigned char checkParity(unsigned char x, unsigned char byteNum)
{
        unsigned char data, parityBit;

        data = x & DATA_BITS;
        parityBit = (x >> 7) & 1;
        // remove parity bit from the byte.
        final_recv_packet[byteNum] = data;
        return parityBit == parity(data);
}

// check parity bits for a given packet
unsigned char checkPacketIntegrity()
{
        int i = 0;
        for(i=0;i<PKT_BYTE_NUM;i++)
        {
                if(!checkParity(final_recv_packet[i], i))
                        return 0;
        }
        return 1;
}
```