August 2010

Voice Controlled Release: Networking

Jonathan Kendall Low
Worcester Polytechnic Institute

VOICE CONTROLLED RELEASE: Networking

A Major Qualifying Project submitted to the Faculty of
Worcester Polytechnic Institute

In partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

Jonathan K. Low

12th August 2010

Advisor:
Professor William R. Michalson, Worcester Polytechnic Institute

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

ABSTRACT

This Major Qualifying Project is intended to solve electrical and mechanical issues with voice controlled release systems used in shooting sports. This report describes the networking portion of the system containing wireless communication, state machine logic, software controlled data flow and integrated power management to improve upon current voice controlled release systems based on user requirements.

INTRODUCTION

ABOUT THE PROJECT

Shooting sports, such as trap shooting and skeet shooting, date back to the mid 19th century and are popular in many countries. This project focuses on trap shooting and the equipment used by shooters and shooting clubs. In particular, this project seeks to improve the voice-controlled release systems that have become popular in different clubs around the United States, using the Canterbury system as a model this report seeks to improve. This report explores the networking aspect of an improved implementation, modeled using several microprocessor systems and wireless modules to simulate this improved product.

BACKGROUND

In order to understand the project it is necessary to understand the background of the sport, including origins, evolution, variety, necessary equipment, rules, and procedures for game play. Also included in this section are current mainstream and alternative implementations of voice controlled release systems for examination.

TRAP SHOOTING

The sport of trap has many variations of game play depending on region and available resources. The premise of each game is to break clay disks by firing at them using a shotgun. The goal of the game is to break as many targets as possible. In older variations the targets were live pigeons while the modern game is played with clay disks called by the same name (clay pigeons). This project will primarily deal with American trap as opposed to many of its other styles. American Trap is governed almost exclusively by the rules of the Amateur Trapshooting Association.[1]

To play the game in its modern form requires five firing rows, called lanes or runs, as well as a trap house as oriented in Figure 1.These rows are located at standardized distances from the trap house starting at 16 yards.

[1] The full Amateur Trapshooting Association rulebook is available on the online at the ATA website at http://www.shootata.com under About the ATA.

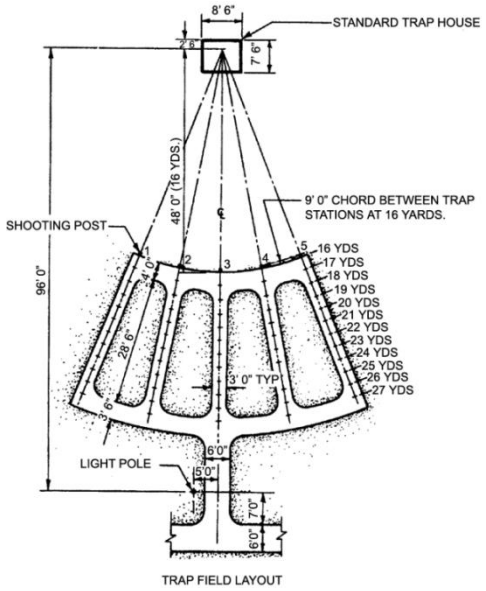

FIGURE 1: GENERIC TRAP FIELD SETUP

The three usual games within the sport of trap are Singles, Handicap, or Doubles. For singles, all shooters are located at the 16 yard distance. In their lane, each shooter is normally provided a total of 5 targets; however it is also possible to shoot 10 targets per post. Handicap is similar to singles; however the firing distance can range from 16 yards to 27 yards. In both singles and handicap the clay leaves the house in a random direction as defined by the ATA rulebook. In doubles, each shooter attempts to break 2 targets which leave the house simultaneously. In addition, for this game the position of the clays leaving the house is fixed and can be considered to follow the same flight path on every throw. In general, in a squad shooting handicap no shooter is ever more than 3 yards from another in order to minimize the muzzle shockwave. The typical number of targets thrown in a handicap or singles round is typically 25 per shooter (50 per round of doubles). Several rounds (called sub-events in the ATA rules) will make up an event which typically consists of 100 targets per shooter (however 50 and 200 target events are also common).

TYPICAL SHOOTING SEQUENCE

In order to play a game, a group of 1 to 5 shooters, called a squad, assume positions on the trap field with each shooter assigned to a particular shooting post, or station. Targets are shot in succession with each shooter firing a total of five shots from each station for a total of 25 shots per shooter, per round.[2] At the end of each subset of 5 shots, the shooters rotate shooting posts until each shooter has attempted to shoot 5 (or 10 in the case of doubles) targets from each station. This pattern of firing can be seen in Figure 2: Rotational Firing pattern. Each shooter

[2] 25 shots per shooter per round is typical, however there are common variations such as shooting 10 rounds per shooter per post, shooting 5 pairs of targets per post (10 shots per post taken two at a time when shooting doubles), and variations on doubles where shooters will shoot 2, 3, 2, 3, 2 pairs (24 targets) or 3,2,3,2,3 pairs (26 targets).

can be considered legal when that person can be expected to call for the next clay pigeon. This includes the next person on the line as well as the potential for the original shooter to call again due to one of the failure modes and sources.

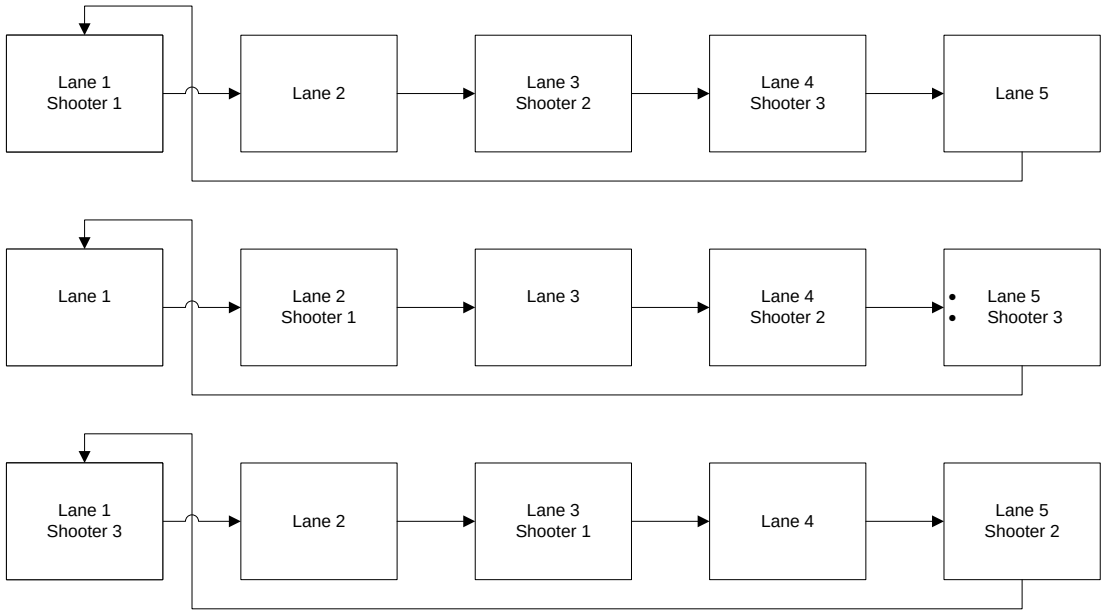

FIGURE 2: ROTATIONAL FIRING PATTERN

For a typical shooting sequence, one shooter follows the previous until the pattern would repeat itself, at which point the initial shooter starts the sequence again. Several conditions exist in which a shooter will call for multiple targets in a single pass; these are shown in failure modes and sources.

GAME HARDWARE

There is very little machinery required to play the game, however it is important to know what these pieces are. An essential piece of hardware is a trap machine, which comes in two varieties: automated and manual. A shotgun could be considered additional hardware however the system designed by this Major Qualifying Project is completely independent of this element in all forms.

MANUAL LAUNCHERS

Before automated systems, the trap house was activated manually via a lever and rope system. The switch from manual to automated systems involved a financial investment by trap shooting clubs. In smaller establishments the $3,000-$6,000 cost for an automatic trap machine is a substantial investment. However, in order to run a field with a manual launcher the presence of 2 volunteers is required. One member is used to load the clays and the other is used to

activate the machine typically with a pushbutton. These machines are normally spring driven and targets are loaded manually for each launch.

AUTOMATED LAUNCHERS

The trap house is a concrete structure that surrounds the trap machine shown in Figure 3, typically buried partially underground, launching targets using a hydraulic lever arm. The trap house itself is governed by its own physical dimension requirements as to meet the ATA rulebook. These dimensions are laid out in Section XIII, Article B of the ATA rulebook as quoted below:

> ***B. TRAPHOUSES***
> *Traphouses must adequately protect the trap loaders and shall not be higher than necessary for that purpose. It is recommended that traphouses constructed after September 1, 2003 shall conform to the following specifications:*
>
> *1. Length not less than 7 feet, 6 inches, nor more than 9 feet, 6inches.*
>
> *2. Width not less than 7 feet, 6 inches, nor more than 9 feet, 6 inches.*
>
> *3. Height not less than 2 feet, 2 inches, nor more than 3 feet, 0 inches, the height to be measured from the plane of the number 3 shooting position. It is recommended that the throwing surface (throwing arm or plate) of the trap machine be on the same level as that of Post 3 and the target height setting pad.*

FIGURE 3: AUTOMATED TRAP MACHINE

The trap machine is triggered by a relay connection through a twist pin within the trap house. That twist pin can be connected to a variety of sources; in some setups it is triggered by a manual push button, while in some systems it is triggered using a voice release system.

CURRENT VOICE RELEASE SYSTEMS

Though non-essential, it is common practice for fields and clubs to use voice release systems for convenience. This project is primarily based upon the Canterbury, the most popular voice release system with a few others taking distant second. The Canterbury is implemented in two variations: wired and wireless, both accomplishing the same goal only in slightly different manners. Another system that has been popular in the past is the Clay Mate, and there is an upcoming system called ERAD.

THE CANTERBURY

The construction of the Canterbury is composed of 3 physical parts in both the wired and wireless designs. In Figure 4 the functional block diagram of a wired Canterbury can be seen. In the wired Canterbury the system user end is a loudspeaker, which functions as semi-directional microphone to receive input from the shooter. Those signals are taken to the second stage of the wired Canterbury and processed to determine if a valid call has been received. The process by which the Canturbury determines the difference between noise and a call is unknown. Based solely on observation during operation, it can be suggested that the "brain box" performs a rough low pass filter to determine an acceptable firing condition.

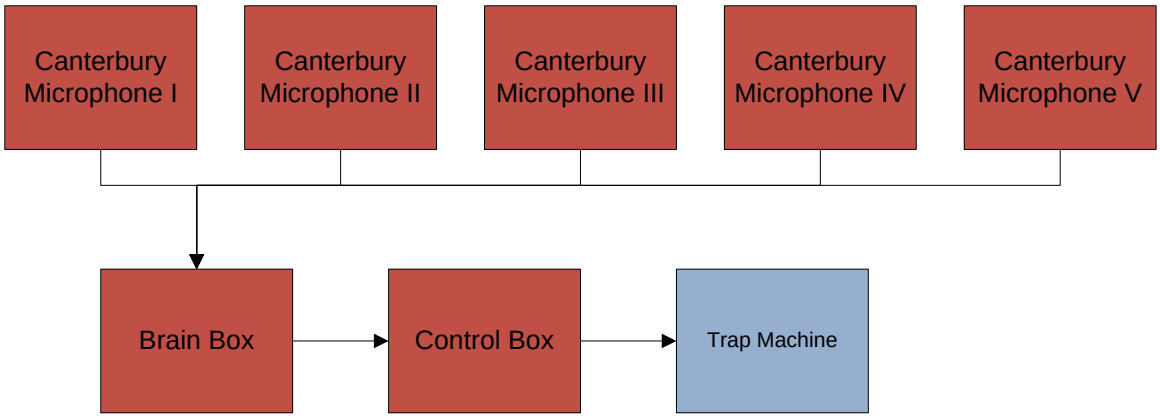

FIGURE 4: CANTERBURY SETUP

The last piece of the Canterbury wired system is the control box that sits within the trap house. It is connected to the brain box by a cable and its primary job is to take a fire signal from the brain box and tell the clay thrower to launch, with a secondary function to isolate the triggering mechanism from the rest of the system for a degree of electrical protection. This system is not complicated to install or maintain beyond ordering replacement parts.

FIGURE 5: WIRELESS CANTERBURY LANE ASSEMBLY

The wireless Canterbury system is similar to the wired system with only a few key differences. As opposed to the wired system where the microphone sends call signals directly to the "brain box", in the wireless system each speaker, referred to as a microphone transceiver by Canterbury shown in Figure 5, is fed into a plastic box attached to the stand with the microphone, which then relays the signal wirelessly to the trap transceiver.[3] It is unclear whether the decision engine is within the microphone stand or in the trap transceiver positioned within the trap house. Assuming it follows the same procedure and tests for threshold and the decision engine is within the first stage, the "brain" box portion of the wireless system is an enclosed transceiver positioned at the front of the lanes that relays that information to the trap house.

After the signal is transmitted, it is fed into another hop-along point at the trap house which receives the signal and transmits it to the control unit, which then activates the trap machine. Other than eliminating the wired portion of the system, the wireless provides a small amount of additional information via lights located on the transmit receiver boxes.

THE CLAY MATE

The Clay Mate is a lapel system similar in design to this project that is used in limited quantities, less commonly used than the Canterbury. It seemed to accomplish many of the goals of the Canterbury as well as providing a few extra features such as the ability to play "skeet", another style of sport shooting, and a counter for how many targets were launched. In addition the Clay Mate had an extra station per field in which you could manually control the launches as seen in Figure 6 as well as provide different firing styles easily.

[3] Specific terms used in this section are those used by the description of the system within the Canterbury manual available on the Canterbury

FIGURE 6: CLAY MATE CONTROL STATION

THE ERAD

The last system we will examine is the ERAD, a potential competitor and recent system with the same capabilities as this MQP's proposed one. Limited public information bounds what can be discussed but examining promotional materials ERAD looks like an equal product in everything but price. Instead of speakers, users use credit card sized modules, equipped with a 12 digit keypad and a liquid crystal display, carried on their person and transmitted to the house to set off targets.

PROJECT EXPLANATION AND GOALS

This section will provide the baseline for deriving the fundamental goals of this Major Qualifying Project. It includes the conversions of real problems into quantifiable solutions by means of the system architecture and implementation.

DATA GATHERING

In order to better define the problems to be solved by this current Major Qualifying Project, and to determine shooter expectations, a large amount of information was collected through various mediums. This information was gathered in three main ways: player input, online reference and personal observation. The majority of input gathered was via personal shooter experience therefore problems and suggestions are not independent of bias.

SHOOTER-SUBMITTED PROBLEMS

Several shooters as well as club operators have problems with the ability to diagnose a failure with the existing systems quickly to restore its operation. The primary reasons for system malfunction are hardware fatigue and failure due to use and environmental conditions. Particularly in the case of the wired Canterbury system there are several wires leading to and from various components as shown in Figure 7 Figure 7. In the wired Canterbury, if any particular connection is faulty or non operational it will lead to a stop in play, because each station is used even if there is only a single shooter. In the case of a wire failure, the unit in question is replaced with a different, functioning wire and the malfunctioning wire is repaired. Each wire in the Canterbury system is similar to that of a XLR or microphone cable with some weatherproofing to withstand some environmental conditions. In additional to the wire failure from stress on the cable, a problem noticed by shooters and operators are the connectors. Standard CBC microphone style connectors are subject to easy penetration by water. If a club has the need to replace a cable, in order for the system to be operational with 1 microphone transceiver per lane an overhead unit must be kept ready. Theoretically, if all 5 cables were to fail the club would need to stock 10 cables total for the system and 5 to replace when the originals failed. This is an additional overhead cost to the club applying the same concept for every element within the system.

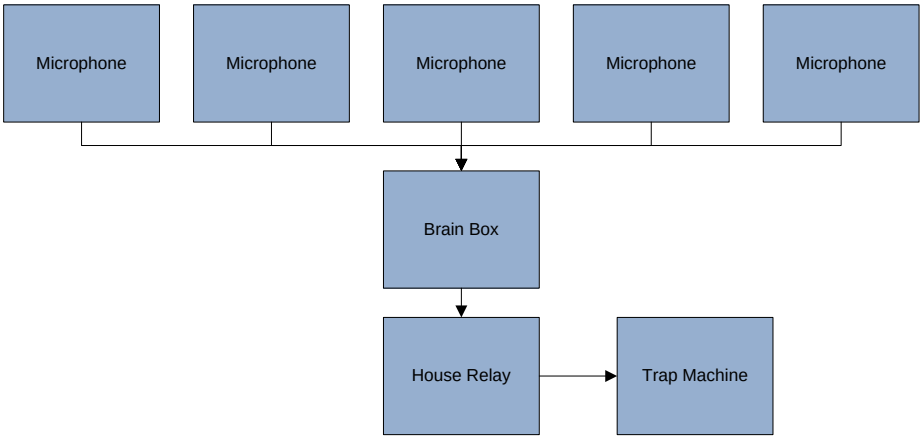

FIGURE 7: WIRE ORIENTATION

Dealing directly with the input portion of the Canterbury system, shooters reported a probability of the system being inadvertently triggered by banter on the line or other people on adjacent trap fields. In addition to the incorrect firing of a machine, this condition confuses shooters as to who is attempting to break that target. The cost, as mentioned earlier in this section, is deferred to the club as no shooters will be attempting to break this target or claim responsibility. The other complaint presented with the current system is the inability to play other variations of shotgun sports, however this project will not attempt to find a solution to those game play issues.

In order to gather information beyond the members of the Wallum Lake Rod and Gun Club, a popular website for trap shooters was used. Trapshooters.com is a forum style website dedicated to shotgun sports, culture, and member interests as well as serving as the second medium for data collection. By making several posts regarding people's opinions on the Canterbury and other voice controlled release systems, a lot of information was debated among shooters as well as operators. The first major conclusion of the online submissions was the opinion from many shooters that that the systems in question responded to a particular command or utterance. The most commonly used call is the word "pull," however via all 3 mediums several dozen possibilities were collected including: "bird," "call," "fire," "shoot," "go," and growls of many varieties.

The fact that many people believed the system responded to a particular word caused them to distrust the system. Also, in a related submission, people came to the conclusion that a shooter with a high or low pitched voice would be less likely to reliably trigger the machine. Most often this was suggested in the case of women and children, however from personal experience I cannot confirm this behavior. These shooters would often not trigger the machine on their first attempt according to many of the posters on trapshooters.com. Also online there was a variety of people discussing the topic outside topics related to this project including enthusiasts of other systems such as ERAD. Club operators had concern with the battery life of any wireless unit as they found money lost in batteries quickly adding up. This was due to a need to replace as well as the increased cost of replacement cells.

OBSERVATIONS

From personal observations there can be additional problems worth mentioning. The physical orientation of the Canterbury system can lead to 3 problems: tipping, tangling, and placement hazards. Each microphone station for shooter input is naturally top heavy as shown in Figure 8. The wired version (not pictured) is even more top heavy, with some tension being applied from the back of the microphone. Dependent on location, each stand can tip over due to wind and or accidental shooter contact, and this can lead to damage to different elements within the Canterbury in the form of broken casings, extra pull on wires as well as connection deformation. In addition, when a microphone tips over it causes a stop in play and potentially accidental firing of the trap machine. One solution to this taken by the Wallum Lake Rod and Gun Club is to attach 2 Lb lead weights to the base of the stands in order to prevent them tipping, however this solution was not thought of by the manufacturer.

FIGURE 8: CANTERBURY WIRELESS LANE STATION

In the case of the wired Canterbury there is a major concern with tangling after a day's worth of play. Due to the fact that the Canterbury's are stationary in relation to the firing distance of the shooter, they seem to be constantly moved within a single game of handicap or even singles depending on preferences each shooter has for microphone position. Each microphone stand, when moved, creates the possibility of tangled wires as well as creating pull on the connections. When the system is disassembled at the end of play, the tangled wires are often a hassle to the operator of the field. The last of the physical orientation problems was the placement of the system during inclement weather. In particular, snowy weather has pieces of the system placed where one must trudge the powder in order to set the system up. In the case of the wired system this would typically mean placing the brain box buried in the snow in front of the lanes with it possibly sinking into the weather testing its weatherproofing. For rain the speakers can often become waterlogged if their small drain hole is blocked or missing.

FAILURE MODES AND SOURCES

There are very few failures modes in the normal operation of the system. The failure to launch a legal target when one is expected or the launches of a target when none is expected are

the most obvious failure modes. The third failure mode is a stop in play when the system fails to respond to any input or behaves without pattern or logic. There are several sources for those failure modes in the Canterbury system.

In the case of a failure to launch, the flow of the game and concentration of the shooters are disturbed. For instance, a shooter playing a round of singles calls for a target and waits in firing position to see the clay pigeon leave the house. If the clay pigeon does not appear within a reasonable time the focus is broken and the shooter may reposition and call again. If you were to predict the location of the next shooter not knowing the machine failed to produce a legal target the prediction would be incorrect. In the case where a clay pigeon within the machine is broken before or while leaving the trap house, it is not a legal target therefore equivalent to no clay leaving the trap house. Figure 3 in *Automated Launchers* shows the rotating drum containing several hundred clays. If loaded unevenly, slots within the drum can run out sooner than others leading to empty slots. These empty slots, even if the voice controlled release and trap machine are working entirely as intended, would cause a failure to launch legal clay.

The second failure mode is launching a clay pigeon when no intentional triggering condition is present. This condition is larger in importance to the owners and operators of the club fields as it directly translates into profits lost and costs increased. In the Canterbury it appears any sufficiently loud source is a viable trigger. This is a major problem due to the fact noise sources are largely available on site. For example fields are often located adjacent to one another in various arrangements with no more than a few dozen yards between them shown in Figure 9. While these distances are not standard across all clubs the fact remains that the possibility of cross field noise is highly likely.

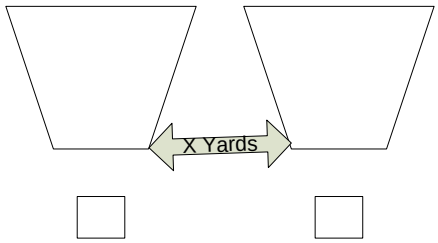

FIGURE 9: DISTANCE BETWEEN SHOOTERS

The third failure mode is the event of a system performing not to design. The system behaves radically regardless of input but also could mean a complete lack of response. This is a stop in play condition (SiP) where all actions are purely in effort to return the system to an operational status. This is caused by complete hardware failure or user incompetence.

IRREGULAR SHOOTING CONDITIONS

Several conditions exist where an uncommon event takes place. These events have the potential to change the flow of play without technically having a failure in the system anywhere. If a shooter enters the squad at some point after the round has started and each person in the initial squad has shot at least once; they would disturb the expected order as well as misrepresent the total number of shooters on the field. Another condition is a shooter leaving the round before

its completion. This can happen when the shooter has another urgent matter or possibly equipment failure. Another irregular shooting condition is the immediate shutdown of a field for weather or other reasons, while the system is not at fault it still may be in a condition to trigger the failure modes.

PROBLEM STATEMENT

The Canterbury, as with many other voice controlled release systems, fails to solve many of the problems associated with controlled flow game play as well as simply setup and maintenance. Networking solutions being developed in this project will be combined with a signals portion as to build a complete solution in another implementation. Below are the problems to be solved by the system:

- Reduce number of accidental fires due to noise and incorrect shooter calls
- Maintain accounting information for tracking
- Allow for remote control of a field by a master administration station
- Implement a System Design which requires less setup and maintenance
- Provide for an affordable end result based on progress in this project

DERIVED DESIGN REQUIREMENTS

This project aims to solve the networking problems with this system through a combined approach of wireless communications, state machine logic and administrative control. In order to achieve the solutions for the issues listed above in the problem statement a variety of system architecture requirements must be defined. The majority of the problems defined above can be solved by combing elements within a system design. The system will consist of a unit implementing wireless communications, allowing for the management of multiple fields, capable of handing any number of different actions in the game play, and establishing system architecture for simplicity and effectiveness.

PROPOSED SOLUTION

In order to satisfy the design requirements for this project the current, previous, and upcoming systems were analyzed. These systems each carried their own disadvantages and operation failures, and by combining the system's best attributes this project came to a feasible solution. This system can be separated into 2 distinct categories: Networking and Signal Analysis. This report will cover the networking responsibilities from the point past the recognition of a signal to the consequences of that data fully propagated through the system. The system will consist of 3 physical subsections, a lapel unit, a transceiver relay (located at the trap house) and an administration center as system architecture in order to accomplish its goals. The organization of these physical subsections in the proposed system is shown in Figure 10 as would be seen in a multiple field installation.

The first, the “Lapel Unit”, acts as a replacement for the microphone transceiver in the Canterbury system. As opposed to the fixed positions of Canterbury system, the Lapel Unit will be attached to each shooter. This unit will carry a wireless transceiver in order to eliminate the possibility of wire tangling and to reduce bulk and weight to a few ounces. This unit will be responsible for the signal determination, namely all signal processing and source identification. The wireless transceiver in this implementation is used as a transmitter, as receiving capabilities are unnecessary in this portion of the system.

The second unit within the system is the “Transceiver Relay” positioned at the trap house. This unit replaces the control box of the Canterbury and serves the same purpose with additional capabilities. This unit is responsible for activating the trap machine, serves as the main logic control of the system, and serves as a node for the accounting within the system. Due to its location it will need to be powered and protected to meet its surroundings. ATA rules state that a trap house must remained undamaged when shot at minimum range of 16 yards, the same applies for the Transceiver Relay module. In a later implementation of this project a protective covering can be designed as to protect the system at the house during use with live, not simulated, inputs. This design characteristic is one not solvable by electrical means and therefore should be considered non-material in relation to this project.

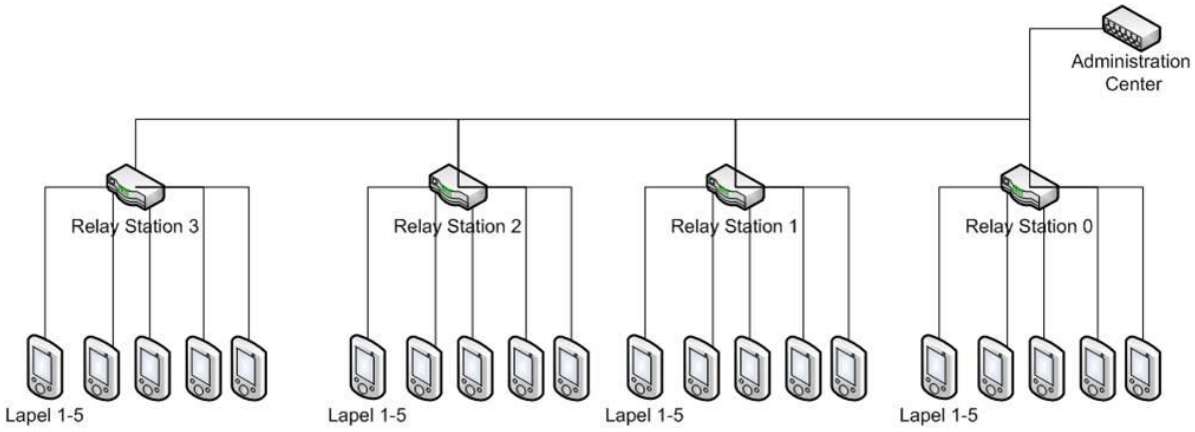

FIGURE 10: PHYSICAL SYSTEM ARCHITECHTURE

The third unit in the system is the administration control which adds a new element when compared to the Canterbury. This unit is a global replacement for the scorekeepers’ transceiver within the Canterbury architecture. As opposed to having a 1:1 ratio with the Canterbury Control Box, the Administration Control Unit has reign over all Receiver Relay units in that subset of fields. This unit will provide game manipulation actions as well as accounting features to better inform the users of their status. Discussed in the Future work section is the possibility of adding a hardware unit so that each field has a scorekeeper’s control. This hazard relates to the play of the game during competition but was not a concern when investigating the problems with the current voice controlled release systems.

This project is supported by very simple hardware architecture in which the system must be capable of several actions. The hardware architecture developed for this project is only meant to satisfy the networking constraints with signal processing being used as a placeholder in **Error! Reference source not found.** below. The system must be capable of external communication to a wireless module to be used as a node within the system as a whole. This communication is essential to implement the system as described in the proposed solution section. In addition to external communications, the receiver relay unit will need to have general purpose input/output pins available for the control of external devices including but not limited to the trap machine.

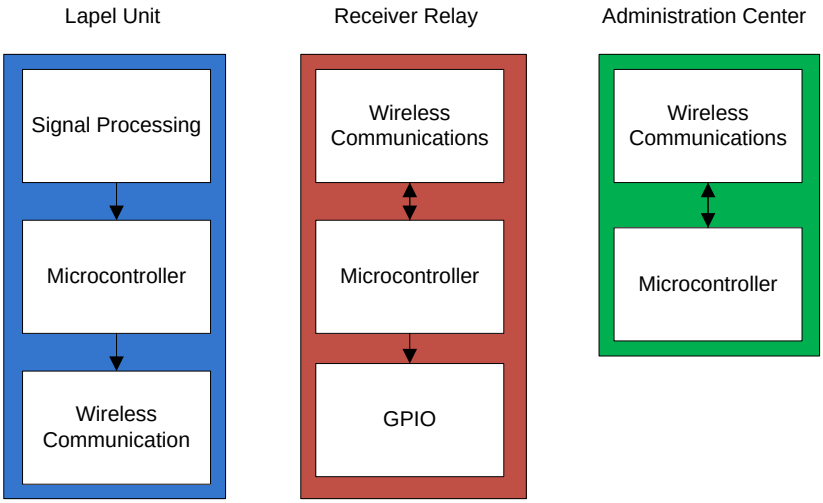

FIGURE 11: HARDWARE BLOCK DIAGRAM

In addition to external communication and control of peripherals, the system needs to receive input. This input is used to simulate the result of the signal portion of the system as well as provide controllable menus for the administration center. Two hardware requirements of the wireless module are the ability to transmit from the farthest lane position to the trap house where the receiver relay unit is located, and transmit data at a rate fast enough to reach the administrative center without losing information. In the lapel unit, where battery life is required to last multiple months on low duty cycle use on a daily basis, a low power solution has to be adopted.

Essentially, the hardware in each unit of the system is made of an operational microcontroller as well as a wireless module with some additional inputs on the lapel and Administration units of the system. Instead of developing a microcontroller platform on which to base our software elements a viable alternative was available. Some additional hardware circuits must be developed in a full scale production in order to perform voltage transformation and proper input configuration.

While only the administration center requires a display for the proof of concept of the system, an LCD is available on the chosen hardware on each unit to display various state changes and user firing information without digital I/O measurements.

Using the MSP430-449STK2 Development kits available in the electrical and computer engineering shops, we were provided all required attributes required by the hardware architecture as described above and pictured below in Figure 12. Retail wireless transmitters are very common in their construction and interfacing; most use serial data streams at fixed/standard baud rates. Several attributes are helpful in addition to being a functional wireless module which helps satisfy our system requirements. For instance an operating voltage which a transceiver runs should have an intersecting range with the operating voltage of the microcontroller. A major concern for the Lapel Unit of the system being developed is power, and therefore we should be able to reduce its draw on the power supply. This project incorporates the Xbee Xb-24 Transceiver from Digi in order perform wireless transmission of data.

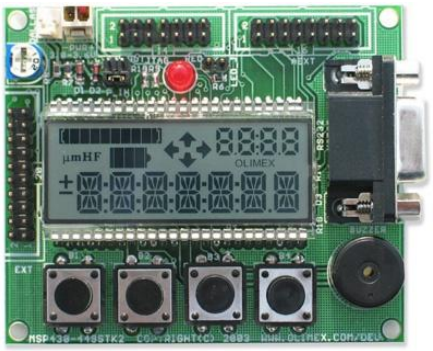

FIGURE 12: MSP430-449STK2 DEVLOPMENT KIT

MSP430 DEVELOPMENT BOARD

The MSP430 Development kit uses the MSP430F449 160 pin controller to operate its many peripherals. The peripherals include 4 momentary pushbuttons, rs-232 port, JTAG Debugging Interface, Buzzer, 7 character LCD display, as well as 2 extension headers including access to the UART0 Rx/Tx pins. This module uses one of several programming software packages including IAR Kick start used for this project. Due to these development kits being used in ECE2801: Embedded System Design as well as several authorities that are available on the use of these controllers, they were chosen for use in this project.

As seen in the above Figure 12 the MSP430 development kit contains pushbutton inputs that can be used in order to simulate the result of the signals portion of the system attached to pins P3.4 -P3.7. In addition these buttons will serve as the source of menu selection when using the boards as an administration unit within the system. On each extension header of the MSP development kit there is a pair of UART pins capable of streamlined serial transmission. This capability will serve as the external communication to the XB-24 wireless modules by Digi discussed in the XB-24 Wireless Module section of this report. The alternative to which is to have the system implement a software UART substitute with timers and pin interrupts.

In terms of GPIO pins available on the MSP Development board there are 17 including the UART pins this system will use to interface with its wireless module. This number is low compared to the maximum possible 48 GPIO pins. This system will only make use of 2 GPIO pins: one for the sleep mode of the wireless module and one for the output to the trap machine to control the release of clay pigeons.

XB-24 WIRELESS MODULE

The choice to use Xbee wireless transceivers for serial transmission was derived from requirements shown in section *Prototype Hardware Architecture*. The transmitters in the lapel units must operate in the range of 2.1-3.6 Volts, and must have a low power consumption (sleep) mode. In addition to satisfying the operating voltage and transmission rate there are several advantages to the using the Xbee modules that satisfy some of the software requirements listed in the Software System Architecture section. In addition to the standard features of any wireless transmitter the Xbee provides two beneficial features: randomly seeded retransmit times and an internal addressing network that will work in conjunction with a software addressing mode.

FIGURE 13: STANDARD XBEE MODULE

Each Receiver Relay as well as the administration station will use both the transmit and receive functions of their wireless serial links. In order to configure the Xbee modules, Xbee manufacturer Digi provided software that allows for direct access to all registers on the wireless module which control its behavior. This software provides a high level of configurability for later updates to this project however in the current implementation we will be using only a few registers shown in **Error! Reference source not found.**.

TABLE 1: XBEE REGISTER CONFIGURATION

Register	Label	Purpose
Retries	RR	Set number of retries the modem will execute in addition to the 3 retries provided by the 802.15.4 MAC. For each XBee retry, the 802.15.4 MAC can execute up to 3 retries.
Source Address	MY	Set the 16-bit source address for the modem. Set MY = 0xFFFF to disable reception of packets with 16-bit addresses. 64-bit source address is the serial number and is always enabled.
Destination Address (Low)	DL	Set/read the lower 32 bits of the 64 bit destination address. Set the DH register to zero and DL less than 0xFFFF to transmit using a 16 bit address. 0x000000000000FFFF is the broadcast address for the PAN.
Sleep Mode	SM	Set/read sleep mode: Pin Hibernate is lowest power, Pin Doze provides the fastest wake up, Cyclic Sleep Remote with or without pin wake up. Sleep Coordinator setting is for SM parameter compatibility with version 106; ATCE should be used going forward.

Xbee XB-24 units in their simplest implementation are just a wireless replacement for an RS-232 serial link. However, the XB-24 has many layers of complexity as well as capabilities similar to that of a microcontroller. The most important registers in dealing with the goals of this project are the Source/ Destination Registers as well as Xbee Re-tries and Sleep Cycle attributes.

COMBINED XB-24/MSP430 ATTRIBUTTES

Using the MSP430 Microcontroller and the XB-24 wireless modules the overall system can be run using a supply voltage no lower than 2.1V, the lower limit as recorded for reliable operation of the XB-24 datasheet. In order to supply the necessary battery life as defined by the user requirements as well as convenience for cell availability and cost, the system will operate at 3V using two AA or AAA sized battery cells.

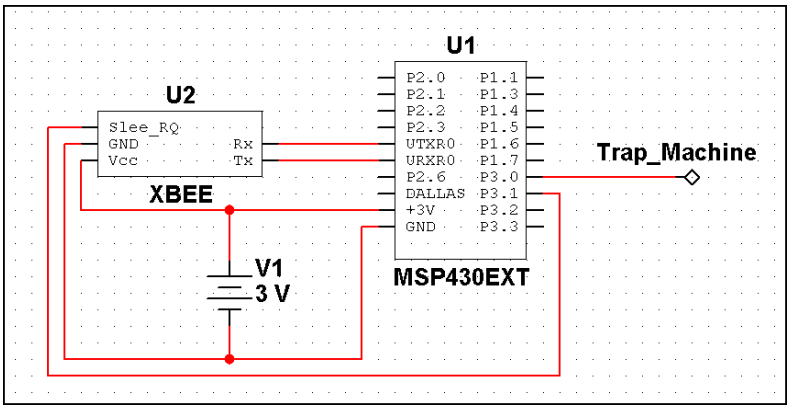

FIGURE 14: MSP EXT/XBEE CONNECTIONS SOFTWARE

In figure 14 we can see a schematic between the MSP extension header and the relevant Xbee Pins. This unit represents all 3 subsections within the system as the Administration and Transceiver units will not require pin to drive their XBEE's wireless power mode as they will be in a constant on state. In addition to that P3.0 in the lapel and administration units will be disconnected as they have no function which would change the level on that pin.

SOFTWARE SYSTEM ARCHITECTURE

The following sections detail the software architectures used in the development of an improved voice control release. These base concepts will allow for a software implementation to solve the majority of the problems with the current voice controlled release products on the market. Similar to the prototype hardware architecture, the software architecture diagram can be found below in Figure.

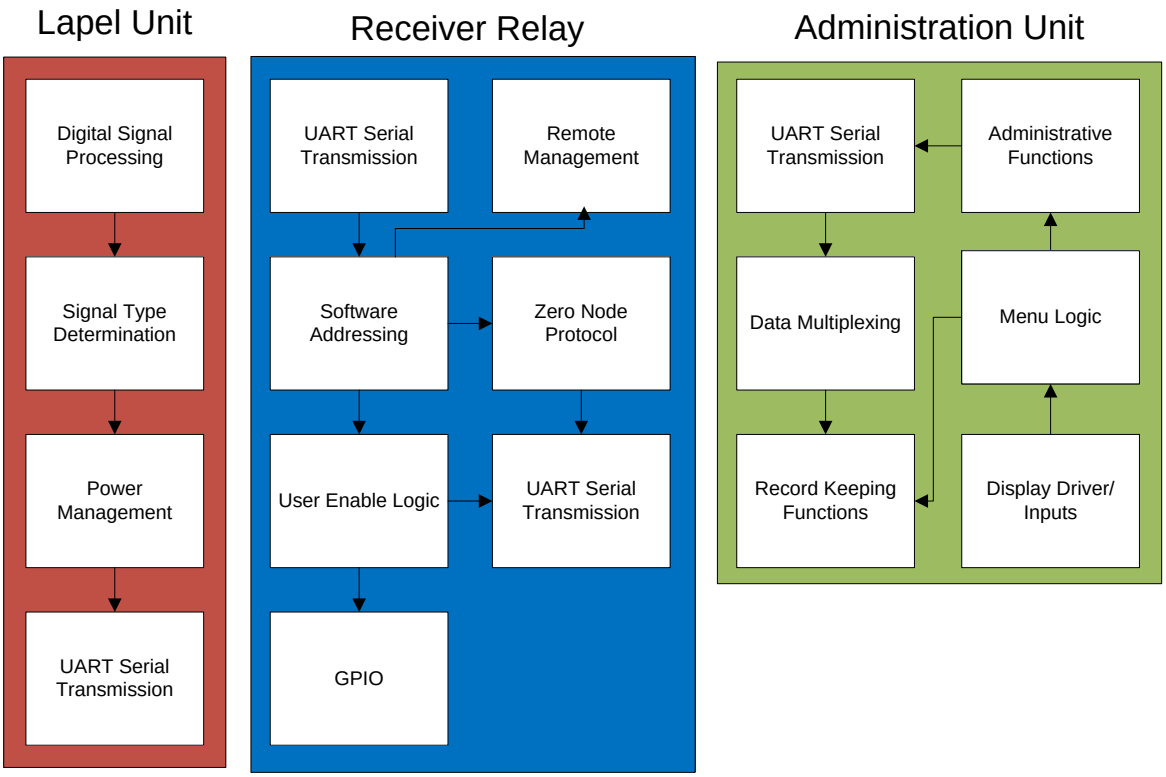

FIGURE 15: SOFTWARE ARCHITECTURE FLOWCHART

The software architecture from user input to the final propagation of data is essential for the system in order to accomplish its goals over the long term.

POWER MANAGEMENT

Starting with the power management software architecture, the main way the system can conserve energy is by entering low power modes for peripherals. In software this means enabling and disabling power consuming units with externally controlled sleep modes. This would be ideal for the wireless module as it consumes roughly 2 orders of magnitude more current, shown in the software critical issues section.

The first critical register is the retries (RR) register which controls the number of retries a module attempts to transmit the applicable data and receive a confirmation of receive signal. In addition to the 3 attempts an Xbee module automatically will try, the Xbee will perform additional attempts to successfully transmit data based on the value of the RR register as seen in Equation 1 below.

EQUATION 1

$$\#\ of\ attempts = 1\ Original + 3\ Automatic\ Retries + Value\ of\ RR$$

This register is set to its maximum value in order to decrease the chance for a corruption of the data. If two shooters trigger the system at the exact same time, the retransmit time for each of their data is reseeded randomly therefore making conflicting data signals highly unlikely for the additional repetitions.

The source and destination registers define the personal identifications and expected end unit for any particular Xbee unit. The source register identifies the source of a transmission and should be paired with the destination register on the receiving unit for correct operation shown in Figure.

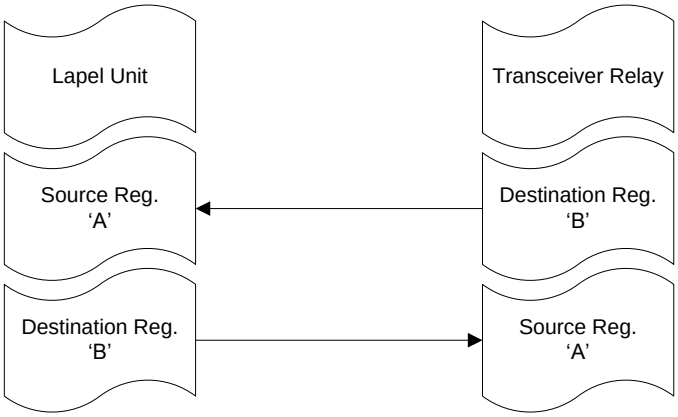

FIGURE 16: SOURCE-DESTINATION REGISTER RELATIONSHIP

This is important because only transmissions from addresses listed in a chips destination register will receive the end packets out of their D_{out}. This allows for the limiting of information per field without additional cross field data contamination safeguards. This aspect of the Xbee allows for the use of its internal addressing in combination with software addressing into a 2 stage addressing system.

The last important register is the sleep cycle register allowing for the system to enter a low current sleep mode for power efficiency. This register can be configured in both a pin activated sleep mode as well as a cycle style where it wakes on a periodic schedule. This register is configured for pin sleep in its lowest current drawing form.

USER-ENABLE LOGIC

The User Enable Logic is the core decision-making software once a trigger condition has been received. This software resides within the "Transceiver Relay" portion of the system at the trap house. The User Enable Logic's primary job is to take a user's attempt to activate and determine whether it is a legal shooter, then, if legal, trigger the trap machine. The User Enable Logic is based upon the expectation of the users to fire in a sequential order. In order for the logic to keep an account of the legal shooters at any moment it must maintain the order of shooters as well as any changes to that order and stalls in the pipeline of expected events. This system uses a state machine as the architecture for which to base this User Enable Logic. It is intended to determine all possible outcomes of the next shooter and has a pre-determined list of actions to compensate for any user whom triggers the trap machine.

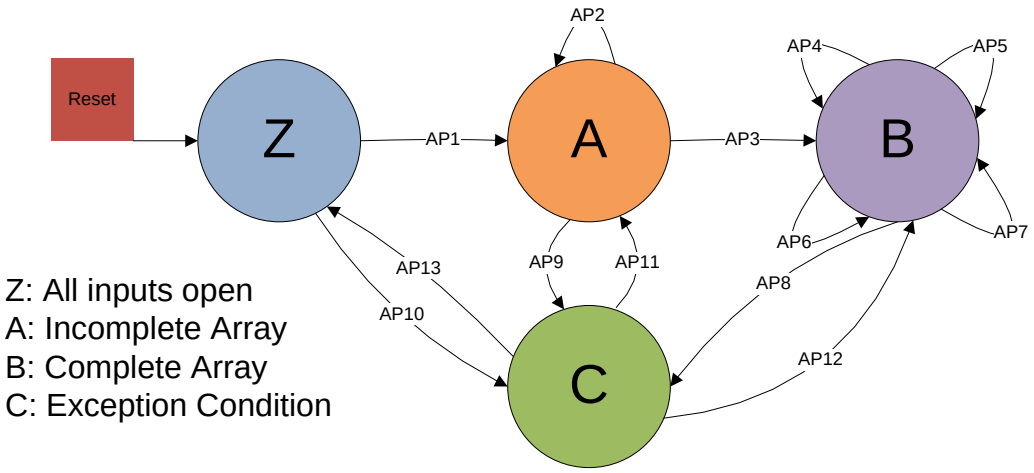

FIGURE 17: USER ENABLE LOGIC STATE MACHINE

STATE Z

Once the state machine has been initialized to a reset condition it is known as Z or Zero State, a state which only serves to be a base point for the rest of the state machine. State Z exists where all inputs are open; the system allows any shooter assigned to that field to successfully trigger a target. No particular user has priority when it comes to the order of play. Because of this any shooter whom calls in state Z will cause a clay pigeon to be thrown. The state diagram of all conditions State Z where it would be changed to another state is seen below in

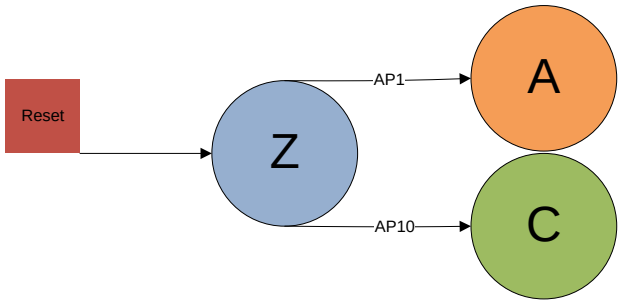

Figure.

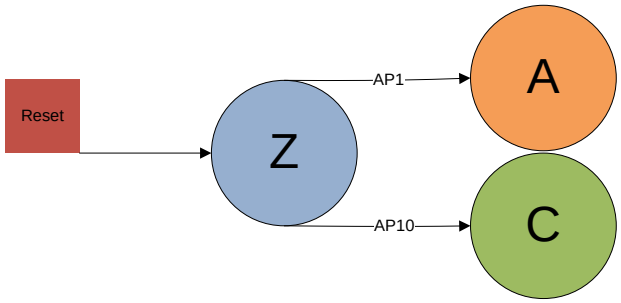

FIGURE 18: STATE Z DIAGRAM

When a user's 'call' is received by the state machine, it then uses AP1 which corresponds directly with the first shooter within a squad. AP1 is the only user driven action path which can be executed from state Z. Since there are no inclusive action paths that are available from state Z, the state machine will terminate in a different state then in started after a reset condition. Action Path 10 will be discussed later in this chapter due to its relationship to state C.

STATE A

State A is used under the incomplete array category; it is used when there are indeterminate amounts of shooters yet to fire their first shot. Under this condition the system can only consider the total amount of shooters in a squad final in the case of a repetition in the initial shooter which originally caused the change from state Z or a full 5 person squad.

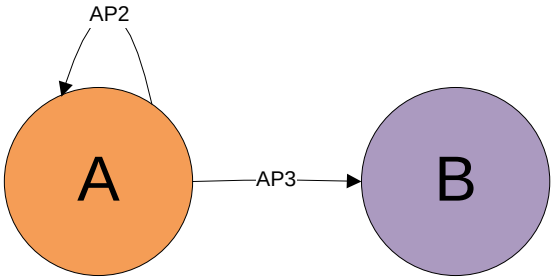

FIGURE 19: STATE A DIAGRAM

Figure above is a diagram of all conditions and action paths involving State A excluding those involving state C. The main function of State A is to acquire all users and establish a shooting order before moving into State B. Action Path 2 functions to fill the array with all shooters on the line, for that reason state A has access to action paths which enter state B when the array is complete. Action Path 3 detects either a full squad or the repetition the initial shooter therefore signifying the completion of the

squad. AP3 allows for the expected user in the typical shooting sequence because of its transition to state B.

STATE B

State B is the most major of the states within the machine. It handles the most events as well as is responsible for manipulation of the expected next shooter in the case of a failure mode or irregular game play. This state is not escapable without SiP conditions and therefore is the terminal destination for each round.

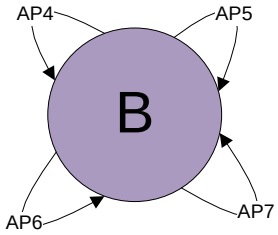

FIGURE 20: STATE B DIAGRAM

Figure above details any action paths that originate from State B, and how it is affected by those actions. AP4 is considered standard play where either the next expected user calls or the most recent user calls again in certain failure mode conditions. In this action path you can expect that if a shooter called and received a broken target the other shooters would recognize the fact that shooter needs to attempt again without penalty. In action path 5 if a new shooter were to enter the line once the order has been established, everyone past the point of entry will need to wait one more shot until they are legal shooters. In the case of AP6, if a shooter leaves the line for whatever reason it seems inappropriate to wait each time that shooter's position comes up. For that reason shooters are effectively removed from the game when missing their second shot in a row. In AP7 there may be a temporary break in the typical cycle for a shooter. This shooter may participate out of turn, in that case a shooter will call for a target, not receive one and be added to an exception listing where should they call again they shall receive a target.

STATE C

As you can see from Figure state C affects every single other state though a direct link. However state C is not a game play mechanism, but rather a SiP condition where a field enters a state unable to activate the trap house. This SiP is intentional given such causes as emergencies and operator's preference. When a field enters state C all conditions relating to play are simply held in limbo.

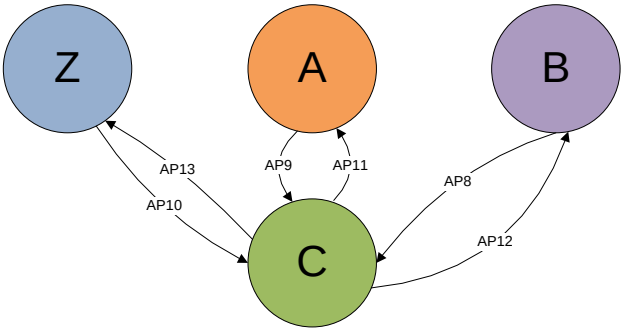

FIGURE 21: STATE C DIAGRAM

When the users in a particular field have left the system inactive for a period, the administration resets a field, or the maximum number of triggers has elapsed, the system will automatically revert to the Z state. However in the case that a field should be shut down temporarily, the administration alone can force a field to go into State C, which serves no purpose but to disable triggering of the house by any source. Above in Figure all action paths involving State C are shown.

ZERO-NODE PROTOCOL

This system makes use of a Zero Node Protocol to redirect the flow of information as described in the system requirements as a sub requirement of the administration control. This control flow operates on the premise that certain nodes have positional values which are used to determine if the information is propagating toward the Zero node. The assumption is that within the system the "zero" node will always be in range of the administration center, from this node to be transmitted to the administration station and vice-versa when applying to outgoing commands. When using zero node protocol the system should never generate multiple copies of the same information because of non-specified destinations. These non specified destinations will cause a recursive effect where nodes generate additional packets elsewhere in the system.

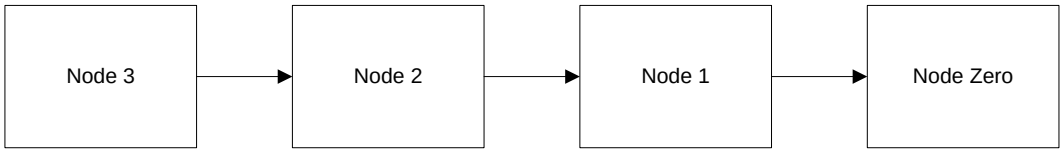

FIGURE 14: ZERO NODE PROTOCOL

A potential problem in the addressed in the future work section of this report is the capability of a node to reach 2 other nodes each qualifying to retransmit the package when using the zero node protocol. This causes the same recursive data as the non specified destination model where a single instance of data may multiply into many copies.

TWO STAGE ADDRESSING

The system takes advantage of two addressing systems when using the Zero Node Protocol. Using addressing within the Xbee XB-24 Units as well as the data addressing defined within the code the system effectively separates the system into a 2 stage process regardless of the operation. The purpose of the 2 stage addressing is to allow for the system to operate with the same set of identifying tags on every instance of a field. Otherwise each set of lapels would need to be explicitly paired with its transceiver with a restructuring of the administration in order to assign targets to each user based on a permanent ID.

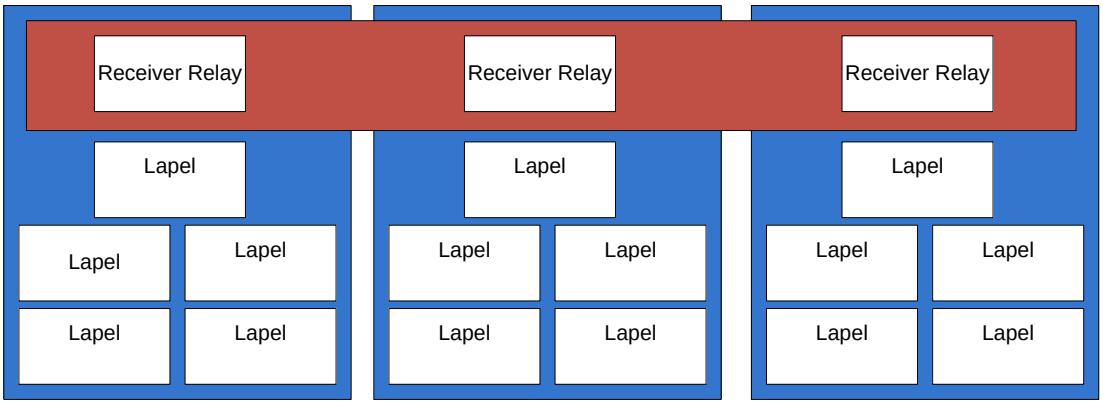

FIGURE 22: 2 STAGE ADDRESSING CONNECTIVITY

In the above Figure 22 the connectivity of each field is displayed. All elements on the same color pads are linked together. Once the information is transferred by the Xbee XB-24 Internal Addressing from the lapels, the receiver relay nodes linked by the same addressing use the software addressing to propagate the data in the correct directions.

ADMINISTRATION CONTROL

One of the major improvements to the current system is the increase in accountability. The administration control provides an insight into the operation of the game as well as the ability to perform routine game operations remotely. The administration control is a key for solving a few of the design requirements as listed in the software architecture section. The administration control's responsibility is divided into actions: Accounting and Game Manipulation. The increase in accountability is most directly a result of the increase of available information by the administration center. In the administration center it will serve 3 purposes: the displaying of scores by field and shooter, placing remote fields into standby or state C, and removing users from fields where they would have left voluntarily.

ACCOUNTING

The accounting modes of the administration control deal primarily with the shot totals of each user on a field. They are intended to be a monitoring tool for system operators in order to press accountability on users whom abuse the lack of observation at a field using automated

equipment. It is the position of this project based on the requirements derived from user accounts, that the ability to account for losses is the highest priority of correctable solution in regards to the operation of a trap club.

In the administration control unit each 2 byte package contains the originating field as well as the user who fired. The data is stored in a 5 by (X field) integer array which is updated as the system is constantly running. The navigation of capabilities is run though a simple rotating software driver activated by buttons. This software driver is similar to a cryptex where each option corresponds to a drum; each drum represents a type (function, field, user) which holds a value, and the combination of those values give us the function of the machine.

$$Function\ Select + Field\ Select + User\ Select = administration\ function\ selection$$

$$Call\ Clay + Field\ 2 + User\ 4 = The\ number\ of\ targets\ user\ 4\ on\ field\ 2\ has\ called\ for.$$

This code can be found in the appendix. This type of menu system can generate every combination for the administration center in this implementation. One note to mention is that when the administration center attempts to standby a field it is of no consequence which value the user drum is holding as it will standby all users on that field simultaneously.

GAME MANIPULATION

The other function of the administration center is to act upon games without being a direct shooter on the field. Using the same menu system as described above the administration center can send out character based hex values tied to field ID's that will be interpreted by the destination to accomplish certain tasks. Each of these messages is sent out with an address therefore only the transceiver relay with the correct address will be affected by the game manipulation software.

The following sections will document the technologies required in the development of our improved voice controlled release system. These technologies satisfy the specifications for power consumption, reduction in connections, correcting of failure in triggering, increasing user accountability and making the system implementation practical for users and operators.

SOFTWARE ADDRESSING & REDIRECTION

Originally the information to leave the lapel and to be received by the transceiver relay would be an indication of field and user who activated the trap machine. In addition to that the system was originally intended to indicate the probability for signal to be either a call or shot. This would add an additional element of information, the purpose of the aforementioned probabilities to be turned into the base for an active filter. In order to create an exclusive band of values which can be interpreted as identified values for field ID's a portion of the standard ACII table was reserved by the system. 0x61 through 0x7A translate into 'a' to 'z' where now the value of the zero node is actually 'a' as an ID representing the lowest point within a system.

In practice, the system checks for the presence of an incoming character whose value is within the identification range. If so, the expectation of the next character received on the serial line is the shooter associated with that call. When those are paired together the result is the basis for a field user data string. Shown below in Figure 23 is the code associated with determining if an incoming packet warrants being retransmitted.

```
#pragma vector=UART0RX_VECTOR
__interrupt void usart0_rx (void)
{
  temp=U0RXBUF;
  if(temp>0x3A)
  {QVB=1; IDhold=temp;}//potential conflict in timing discuss meeting 8-6-10
  else if(QVB==1)
  {Queuel=U0RXBUF; QVB=0;
        if(IDhold>ID){
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = IDhold;
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = Queuel;}
  }
  else
  {active=1; }
  if(temp==0x10  && state!='C'){
    LastState=state; state='C'; active=0;}
  else if(temp==0x10 && state==C)
  {state=LastState; active=0;}

}
```

FIGURE 23: DATA ADDRESSING CODE

Each node is hardcoded with an identifier as to allow for the comparison from the incoming packet. In the above code example this node is 'a' therefore will take any inputs from 'b' to 'z'. In the *Future Work* section of this report there is the detailing of a potential hazard as a result of Zero Node Protocols.

IMPLEMENTING THE STATE MACHINE

This section details the software implementation which using data structures created and manipulated in C on the MSP430F449 Development Board.

```
case 'Z':

    ExpectedUserArray[0]= internal; Apointer=1; state ='A'; ArrayLimit=1; FirePulse();

    if (OVERRIDE ==1) { state='C';}
```

FIGURE 24: Z STATE CODE INITALIZATION

Starting in state Z shown in Figure 24, the system for implementing a state machine is fairly simple. Whenever a particular user calls, they transmit their integer based unique identifier of their unit to the trap house. Since each unit has a different identifier, it is possible just to build the array based on the received one at a time, building the array which governs the action paths taken in states A and B. Each Action Path carries with it a set number of actions to manipulate variables which control the logical decisions of the User Enable Logic. In AP1 the system uses the user information received in order to assign that user to the first firing position within the *ExpectedUserArray* as shown below in **Error! Reference source not found.**. Without it implementing a state machine architecture would be impossible for this project.

TABLE 2: ACTION PATH 1- EXPECTED USER ARRAY

	ExpectedUserArray (Before AP1)		ExpectedUserArray (After AP1)
[0]	0x00	[0]	0x02 “Example ID”
[1]	0x00	[1]	0x00
[2]	0x00	[2]	0x00
[3]	0x00	[3]	0x00
[4]	0x00	[4]	0x00

The *ExpectedUserArray* is an integer array data structure in C which corresponds to the sequential firing rotation on a trap field given a standard game play as described in typical shooting sequence. Each user is given its own integer, assigned as a permanent identifier: in the **Error! Reference source not found.** example “User 2” is the first shooter. The identification integer associated with each user is completely independent of their firing position. For example an array of 5 shooters could be 2,1,4,5,3, which behaves exactly the same as an array of 1,2,3,4,5. In addition to the assignment of the initial user to the first position within the array, the system adjusts the array pointer (Apointer) which determines the next location to be assigned while in State A. The Array Pointer, with variable Apointer, points to the array location and

therefore can only range from 0-4; in addition the Apointer will serve as the point of reference for the expected shooter in State B, as well; in AP1 the Array Limit is increased from zero to one allowing for a looping mechanism to take place at the end of each rotation. AP1 will only take place once when the initial shooter's trigger signal has been received leaving the system in state A after its completion.

In state A, AP2 also detects for 2 fault conditions: non-initial user repetition and exceeding array limits. If a user not assigned to *ExpectedUserArray[0]* is detected twice before an initial user repetition, the system will assume a fault in game play and reset the system to Z state. This will allow for the firing order to be preserved without interrupting game play. In the case where the system detects that the array location [4] within the *ExpectedUserArray* has been filled it will change to State B bypassing AP3. This bypassing of AP3 is only possible when a full Squad of 5 shooters is on the line or hardware failure. In the case of a squad of less than 5 users, AP3 detects the repetition of the user in initial position and changes states, as well as modifying the Apointer to reflect *ExpectedUserArray*[1] to be the next user. This Action Path 3 leaves the system in State B, the primary firing state.

TABLE 3: EXAMPLE FIRING SEQUENCE

Call Total	State	User	Action Path	ExpectedUserArray
1	Z	1	1	{1,0,0,0,0}
2	A	2	2	{1,2,0,0,0}
3	A	3	2	{1,2,3,0,0}
4	A	1	3	{1,2,3,0,0}
5	B	2	4	{1,2,3,0,0}
6	B	3	4	{1,2,3,0,0}

The system will spend the majority of its time per round in State B. This state signifies the completed array and therefore allows the firing pattern to be maintained throughout this stage. There are a total of 5 AP's possible from State B. These correspond to 5 functions that are necessary in order for the game to play out as expected. Action Path 4 represents standard play; it increments the Apointer allowing for the previous shooter to be eligible in case of a broken bird. In order to be considered a valid user, the ID associated with *ExpectedUserArray [Apointer]* must match either directly Apointer or Apointer-1; see **Error! Reference source not found.** below.**Error! Reference source not found.**

TABLE 4: ENABLED USER EXAMPLE

	ExpectedUserArray Ex.1		*ExpectedUserArray*
[0]	0x02	[0]	0x02 <<Apointer>>
[1]	0x01	[1]	0x01
[2]	0x04<<Apointer>>	[2]	0x04
[3]	0x05	[3]	0x05
[4]	0x00	[4]	0x03

Whether the Apointer increments or resets back to the start of the array is dependent on the position the array. If the user firing, was at the end of the array and the state machine tried to increment the Apointer the array would end up floating in user-less array locations. These user-less array positions have the potential to stall the system due to the inability to reset the Apointer. In order to reset the position of Apointer it is compared to the ArrayLimit which holds the number of shooters in a squad. This action path also resolves the condition where for any shooter broken clay would force them to call for another target. Since the user did not have a legal opportunity to attempt the first clay and the system incremented the Apointer regardless, enabling the -1 location in the array solves that problem. If the Apointer-1 user is detected a unique section of AP4 will not increment the Apointer due to the fact it would already be in the correct position from the broken target triggering. AP4 will be the most used Action Path in the system; since the number of broken targets per round is typically within 5 for a full squad, other AP conditions will be rare occasions by comparison. Within the designated requirements for the system is the ability for the system to seamlessly add an additional user to a squad of 4 or less. Action Path 5 takes a user not previously in the array and slides the array forward using a positional sliding counter as shown below in Table 5.**Error! Reference source not found. Error! Reference source not found.**

TABLE 5: ACTION PATH 5 RESULTS

	Expected User Array		Expected User Array
[0]	0x02	[0]	0x02
[1]	0x03 << Apointer>>	[1]	0x05
[2]	0x01	[2]	0x03<<Apointer>>

[3]	0x04	[3]	0x01
[4]	0x00	[4]	0x04

Another inclusive state B function is the remove user function, or Action Path 6. It takes 2 cycles of users in order to trigger this AP because of the special case exception as allowed by AP8. The system recognizes after 2 missed appearances that the player has been removed from the game and therefore should not be afforded any more enable locations in the array. Should that user return before the completion of that round, the system treats them as if they were a new user.

The Action Path 6 function is a problem in that if you allow an Apointer+1 category for enabling the user logic, you increase the percentage of active IDs greater to that of inactive IDs in cases of a full squad. This is true of all Array Limits as shown in Table 6. Even with a full squad the chance to deter an incorrect call is limited to 40% with 3 of 5 enabled users at given moment.

TABLE 6: LIMITING STATION BREAKDOWN

Number of Users	% Active with 0/-1 Enable	% Active with 0/-1/ +1 Enable
1	100%	100%
2	100%	100%
3	66%	100%
4	50%	75%
5	40%	60%

Action Path 7 performs a simple exception condition in which a non-enabled user triggers the User Enable Logic. On the first call the system will disallow their attempt but insert a special condition in which if they were to call again would fire. This is to allow for skipping of a user for a brief period but then to resume normal play, for instance if a player had to momentarily remove his or herself from the game. Similar to AP1, AP2 uses the *Apointer* to assign the next user ID into the next available slot in *ExpectedUserArray*, however it will not cause a state change until the user ID assigned to *ExpectedUserArray[0]* is detected, signifying a full rotation of the line.

In order to communicate with the X-Bees, the system takes advantage of the internal UART (Universal Asynchronous Receiver Transmitter) configuration from the MSP430 microcontroller. Once the XB-24 Units are configured there are several steps on the way to having an operational UART port. As listed in the MSP430 Family User guide there are several initialization steps required for the setup of one of its 2 UART Pairs. Only 1 UART pair is available externally per board however this satisfies the requirements as each lapel only requires 1 pair of serial connections.

In our system we set pins P2.4 and P2.5 to their alternate functionality which is a UART Tx and Rx Pins by writing to P2SEL. Below in Figure 25 the initialization of the UART pins can be seen along with the along with comments as to each operations function.

```
66 void init_sys(void)
67 {
68   FLL_CTL0 |= XCAP18PF;                    // Configure load caps
69   P2SEL |= 0x30;                           // P2.4,5 = USART0 TXD/RXD
70   ME1 |= UTXE0 + URXE0;                    // Enable USART0 TXD/RXD
71   UCTL0 |= CHAR;                           // 8-bit character
72   UTCTL0 |= SSEL1;                         // UCLK = SMCLK
73   UBR00 = 0x6d;                            // 1MHz 9600
74   UBR10 = 0x00;                            // 1MHz 9600
75   UMCTL0 = 0x03;                           // modulation
76   UCTL0 &= ~SWRST;                         // Initialize USART state machine
77   IE1 |= URXIE0;                           // Enable USART0 RX interrupt
78   P2DIR |= 0x10;                           // P2.4 output direction
79   _BIS_SR(GIE);
80 }
```

FIGURE 25: SYSTEM INITALIZATION

Also available to the system designer in the MSP430 user guide was the register configuration for commonly known data rates given a typical clock speed and what is used in our system. Because we are using the 1.048MHz ACLK shown the default settings when using a 32 KHz input crystal we can safely assign the registers as follows shown on page 514 of the user guide.

	Divide by		A: BRCLK = 32,768 Hz						B: BRCLK = 1,048,576 Hz				
Baud Rate	A:	B:	UxBR1	UxBR0	UxMCTL	Max. TX Error %	Max. RX Error %	Synchr. RX Error %	UxBR1	UxBR0	UxMCTL	Max. TX Error %	Max. RX Error %
1200	27.31	873.81	0	1B	03	−4/3	−4/3	±2	03	69	FF	0/0.3	±2
2400	13.65	436.91	0	0D	6B	−6/3	−6/3	±4	01	B4	FF	0/0.3	±2
4800	6.83	218.45	0	06	6F	−9/11	−9/11	±7	0	DA	55	0/0.4	±2
9600	3.41	109.23	0	03	4A	−21/12	−21/12	±15	0	6D	03	−0.4/1	±2
19,200		54.61							0	36	6B	−0.2/2	±2
38,400		27.31							0	1B	03	−4/3	±2
76,800		13.65							0	0D	6B	−6/3	±4
115,200		9.1							0	09	08	−5/7	±7

FIGURE 26: UART REGISTER SPREADSHEET

The configuration for the UART ports is straightforward and allows for the communication of all the modules within a field through the use of the XB-24 modules. In order to transmit once the system has been initialized the program will need to write to the UART TX 0 buffer which will send the information via Xbee shown in Figure 27.

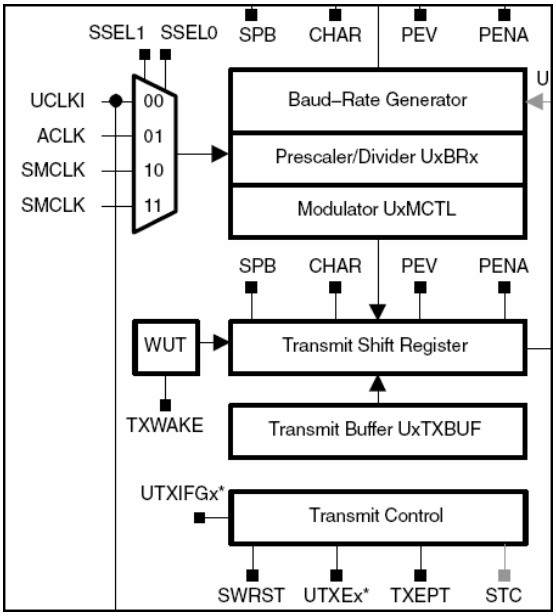

FIGURE 27: UART TX MULTIPLEXED SYSTEM

Receiving information comes from an interrupt vector where the contents of the RX buffer are transferred to a variable of your choosing in a 2-3 operation ISR as shown below in Figure 28. Once the interrupt is received it enables the system while is stuck in a while(1) to perform a single iteration of the various functions associated with array management.

```
191
192 #pragma vector=UART0RX_VECTOR
193 __interrupt void usart0_rx (void)
194 {
195 temp = U0RXBUF;
196 active=1;
197 }
198
```

FIGURE 28: RX INTERRUPT SERVICE ROUTINE

TIMER SETUP

The use of timers is a required element within the system. This project makes use of timers for two purposes in the development of a Voice Controlled Release. The first is to delay the system in order fully allow the XB-24 to initialize once woken from sleep mode. The second purpose of timers are disabling of the input to the system for a given period after a successful fire. This timer is in order to ignore the expected gunshot approximately two seconds after a call.

To implement the timers associated with the lockout and power conservation modules the included "Timer B" was used. Timer B was the only available choice as the LCD uses the basic timer and the buzzer uses the Timer A, in case the system was to use that as a demonstration tool. In order to set the parameters of Timer B, three particular registers are relevant: TBCTL, TBCCR0, and TBCCTL0. TBCTL controls the majority of the attributes of the timer including clock source selection, number of bits, type of counting, and enabling its operations. In order to set TBCTL to the parameters desired, predefined labels provided in the MSP430 header file were used, as seen in **Error! Reference source not found.**, as well as the equation found in Equation 2.

EQUATION 2: TIMER B CONTROL REGISTER

$$TBCTL = TBSSEL\_1 + CNTL\_0 + ID\_0 + MC\_1$$

TBSSEL_1 is the source selection header for the timer choosing ACLK running at 32,768 Hz. Other choices included TBCLK and SMCLK; however SMCLK is configured at 1.0485 MHz, too high a speed for our timing needs to be satisfied without adding functions. CNTL_0 controls counter length, and this label specifies 16 bits, which is necessary due to the fact we need all 65,536 places in order to generate the 2 second lockout timer, shown in Table 7.

Bits	Locations	Maximum Value	t_{max} 32.768KHz in s	t_{max} 1.0485MHz in s
16	2^{16}	65,536	2	0.062500536
12	2^{12}	16,384	0.5	0.015625134
10	2^{10}	1,024	0.03125	0.000976571
8	2^{8}	256	0.0078125	0.000244143

TABLE 7: MAXIMUM TIMER INTERRUPT VALUES

ID_0 sets the internal divisor, up to a maximum of 8, in order to modify the length of the timer interrupt. The system uses a divisor of 1; even with an interrupt 8 times slower the largest counter length could not satisfy the 2 second timer required for our purpose. Finally, MC_1 sets the timer to operate in "up mode" counting to the specified maximum in TBCCR0 as opposed to defaulting to maximum counter value or up-down maximum value. Since Timer B will be operating in up mode, the system needs to specify a maximum value in TBCCR0. The value of TBCCR0 depends on the time between interrupts; this system implements 2 distinct timers: 15 milliseconds and 2 seconds. The first, 15 milliseconds, is the listed wake up time within the Digi XBee module user's guide of 13.2ms from the lowest power mode, as seen in Figure 29.

Pin Hibernate (SM = 1)

- Pin/Host-controlled
- Typical power-down current: < 10 µA (@3.0 VCC)
- Wake-up time: 13.2 msec

FIGURE 29: XBEE WAKE UP TIME AND SLEEP CURRENT

In order to implement the 15ms timer, TBCCR0 is set to 0x01EC Hexadecimal or 492 decimal. Below in Equation 3 **Error! Reference source not found.** is the calculation for the value in TBCCR0. The resolution on the 32.768KHz clock does not give us an exact representation of 15 ms however, since each use of this timer is not cumulative, the error is never past 1 tick, and the approximation of 492 ticks as opposed to 491.52 gives a larger margin of error for those past the 13.2 second listed wake up time.

EQUATION 3: 15 MILLISECOND TBCCR0 CONFIGURATION

$$\frac{1}{32{,}768} * 492 = 15.014648ms$$

$$(15 * 10^{-3}) * 32{,}768 = 491.52 \text{ ticks}$$

To configure the 2 second stall timer the system uses after a trigger to lockout accidental firing by a gunshot, the same calculations are performed, show in Equation 4. The register is set to the maximum value of the up counter, 0xFFFF or 65,536. In this case the TBCCR0 register could have not been configured and the timer could be run in continuous mode, because in this case they have the same behavior.

EQUATION 4: 2 SECOND TBCCR0 CONFIGURATION

$$\frac{1}{32768} * 65536 = 2s$$

$$(2) * 32768 = 65536 \text{ ticks}$$

POWER CONSUMPTION

In this section we will describe the power consumption attributes of the modules as well as the choices run operate the modules in according to a specific constraint.

Figure 2–8. Typical Current Consumption of 41x Devices vs Operating Modes

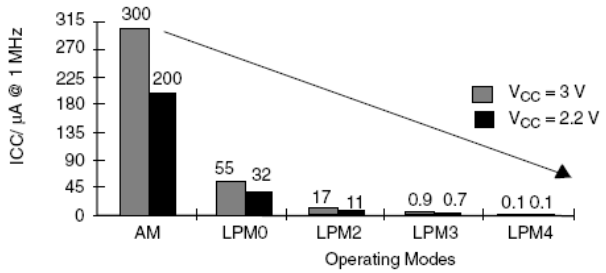

FIGURE30: TYPICAL CURRENT CONSUMPTION OF MSP430 DEVICES

LPM0 – LPM4 are MSP defined low power modes within the hardware architecture however since this project will not implement them, their specific attributes are irrelevant. The current draw of the XB-24 Module as compared to the MSP in active mode is at least 2 orders of magnitude larger as shown in Figure. However the operational time of the XB is shorter than that of the MSP considering the time between shooters in relation to the transmit time as shown in **Error! Reference source not found.**. The operational time of the XB-24 units including sleep time is approximately 20ms, therefore having a potential maximum pull of 2.7mJ, also seen in Figure. Considering the difference in current draw as well as the active periods of each module, the decision was made to run the MSP430 in an active state as long as the system has power.

EQUATION 5: XB-24/MSP430 CURRENT DRAWS

$\frac{45mA}{300uA} = 150x\ Transmitt\ current\ consumption$
$\frac{40mA}{300uA} = 133.3x\ Recieve\ current\ consumption$

EQUATION 6: XB-24 POWER DRAW

$\left(15mS + \left(10\ bits * \left(\frac{1}{9800}\right) baud * 4\ characters\right) + Power\ Down\ time\right) * 35mA =$
$(15mS + (4.0816mS) + 1mS) * 35mA =$ ~20mS*35mA= 2.7mJ

In the software, the system asserts the GPIO P3.0 sleep pin, and then initializes the timer B indefinitely stalling the system. In order to escape the while loop the system is hung on, timer B must trigger an interrupt signifying 15 milliseconds has passed. At which point the system transmits the package of data as shown in Figure 31. This way the system will never prematurely attempt to send UART data when the wireless chip is not ready, also allowing for minimal power consumption when the system is not transmitting anything.

```
/*****************************WirelessEnable()*****************************/
void WirelessEnable(void)
{
  P3OUT=0x01;
  TimerB15ms();
  while(wait==1);
  Tx(btn+0x30);
}
```

FIGURE 31: WIRELESS WAKEUP FUNCTION

In the case of each lapel unit the system will run in a while(1) state while the power is on, only waking the Xbee units when the system is attempting to send information. Even though each XB-24 Unit can act as a Transceiver on the lapel units, there will be no expectation of receiving or acting as an ad-hoc node. Each administration center as well as receiver relay station will be continuously powered including constant running of the wireless modules.

CONFIGURING THE GPIO PINS

In order to use the available pins on the MSP430 Development Board a degree of configuration must be executed. These pins will control two external sources: the trap machine and the sleep pin in on the Xbee wireless module. For convenience the two pins selected for GPIO configuration are P3.0 and P3.1 because of their location on the same port as well as physical location adjacent to each other on the extension header as seen in Figure 32.

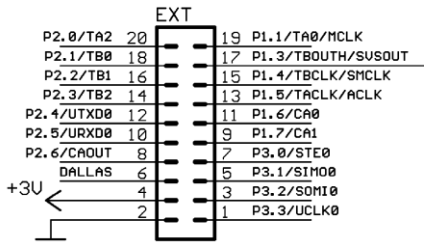

FIGURE 32: MSP430 DEVELOPMENT BOARD EXTENSION HEADER

The first step in setting the GPIO pins as configured is to set their function, as most pins on the MSP430F449 are able to multitask depending on the application. The PxSEL register is responsible for function selection; setting the bits low corresponds to I/O functionality. This can be seen in Figure 33 where the lower 4 bits of port3 are set to I/O functionality. Once the function has been chosen each I/O pin is set to be an output so that if can function as a control signal for another device. Now that the P3.0 and P3.1 are configured to be outputs in order to change the value on them write to the P3OUT register.

```
7
8   P3SEL &= ~(BIT0|BIT1|BIT2|BIT3);          // P3.0-3 I/O Function
9   P3DIR = 0x03;                             // P3.0-1 Output Direction
0 }
```

FIGURE 33: PORT 3 GPIO CONFIGURATION CODE

TABLE 8: GPIO SUMMARY TABLE

P3.0	Output	Lapel
P3.1	Output	Trap Machine
P3.2	Output	All units
P3.3	Output	All units

CONCLUSION AND FUTURE WORK

RELIABILITY

This system cannot be considered reliable for several reasons. One, the hardware in which it is being developed is insufficient to be put into a full scale production. Two, the development of the technologies used in this project are based solely on observation and therefore have not been tested using live data to revise the software if required.

COST EFFECTIVENESS

Obviously the intention when building a system is to be cost effective. However due to the nature of the hardware involved in the development of this project, being cost effective was not a viable option. Given the current retail pricing for the components in the system, each lapel unit would cost approximately $100 before markup. For a full scale production in which the microprocessor foundation was custom built the whole product would satisfy the original intention to make this an option for even the smallest of clubs.

FUTURE WORK

In the interest of continuance of this project the availability of upgrades should be considered. For one, the ability to turn on not only the system but also the trap machine remotely would prove hugely beneficial for clubs and ranges. In addition, with more time the system could be redesigned with the required inputs and less for the rapid development of the system. In a future adaptation of this project the system should address the data addressing hazard by implementing either a mandatory node the administrative control unit or assigning each transmission an authenticator as to compare transmissions at the final node with each other for repetition. A simple addition to the hardware and software architecture is the need to a scorekeepers unit capable of triggering the house without attributing the call to any particular user incorrectly. Using any information available, going forward the system should reflect the best attributes of any system that would compete with it. This means the system goals should be ever changing to reach its full potential.

CONCLUSION

In conclusion the project addresses several issues plaguing the current voice controlled release systems available today. However there are several constraints introduced in this project that were unable to be addressed due to lack of time. The fact this system was a prototype caused the introduction of several sources of failure when compared to our original guidelines. This report only aims to cover half the problem posed by Voice controlled release systems and should be taken as a stepping stone for future works.

BIBLIOGRAPHY

Amateur Trapshooting Association. (2009). *ATA Rules, Bylaws, Policies and other shooter information.* Vandalia, OH: Amateur Trapshooting Association.

Digi International. (2010). *Xbee/Xbee-PRO ZB RF Modules.* Minnetonka: Digi International.

Instruction Manual for the Canterbury wireless voice release for SKEET. Christchurch, New Zealand: Canterbury Voice Release International Ltd.

Texas Insruments. (2010). *MSP430x4xx Family User Guide.* Texas Instruments.

APPENDIX

The following pages contain the code, relevant datasheets, promotional material and other external materials essential for the completion of this project. It is limited by space and importance and therefore some materials shall be listed by their publication instead in the bibliography section.

LAPEL.C

The following code is the final version of the Lapel of the system used for MSP implementation for the demonstration:

```
#include "msp430x44x.h"      // Definitions, constants, etc for msp430F449
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <in430.h>

// ************************************* FUNCTION DECLARATIONS
**********************************************
void init_sys(void);      // MSP430 Initialization routine

void clearLCD(void);         // Clears LCD memory segments so that LCD is blank
void initLCD(void);          // Setup code to interface LCD with MSP430F449
void writeLetter(int position,char letter);  // display single character on LCD
void writeWord(const char *word);    // displays words upto 7 chars on LCD. Can
int  getBTN(void);                   // also display numbers passed as text
void buzzerOn(void);    // turns buzzer on
void buzzerOff(void);   // turns buzzer off
void swDelay(unsigned int max_cnt);   // simple SW delay loop

void UserDetection(void); //Detects user presses simulation for post signal identification
void ArrayManagement(void); // Checks array state changes
void SpecialArray(void); // Checks B state Array Exception Conditions
void FirePulse(void); //The Firing trigger mechanism for the trap machine
void NewUser(void); //Checks if the user is not in the array and ajusts firing order
void RemoveUser(void); //Removes a user if they miss thier spot more then twice
void ErrorDec(void); // Detects if an illegal repetition has occured in array formation
void reset(void); // software reset without power loss or interupt vector
void Tx1(char TxTemp); // Generic UART Tx Replacement with Lockout and Wireless Enable
void Tx2(char TxTemp); // Superficial UART Tx Replacement with Wireless Enable
void mode(int MdTemp, int mode); // Transmitts correct information based on mode
void TimerB15ms(void); //
void TimerB2s(void);
void TimerBStop(void);
void WirelessEnable(void);


/*-------------------------------------------------------------------------*/
/*                                                                         */
/* Global variable declarations                                            */
/*                                                                         */
/*-------------------------------------------------------------------------*/

char *LCD = LCDMEM;       // pointer to LCD Memory Segments
int btn=0, lastbtn=0;
int modesel=0; int wait=0;
/*-------------------------------------------------------------------------*/
/*                                                                         */
/* main() variable declarations                                            */
/*                                                                         */
/*-------------------------------------------------------------------------*/
```

```
//****************************************** LCD CONSTANTS
****************************************************
// From Muneem Shahriar's LCD driver code from Olimex.com
#define     a       (0x80)     // definitions for LCD seegments on the Olimex LCD. 4-Mux operation
is assumed
#define     b       (0x40)     // For more details on 4-Mux operation, gather your LCD datasheet,
#define     c       (0x20)     // TI's MSP430F449 User Guide (look for LCD Controller, then 4-Mux),
#define     d       (0x01)     // and MSP-449STK-2 schematic. You will need ALL these 3 when
defining
#define     e       (0x02)     // each number or character. Remember, the Olimex LCD doesn't use a
LCD driver!
#define     f       (0x08)     // You tell the LCD what characters to display. It's very time
consuming!!
#define     g       (0x04)
#define     h       (0x10)

/******************** MAIN FUNCTION *********************/
void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
  init_sys();                  // Initialize the MSP430

/* Loop forever waiting for input (where would you go if you exited?) */
  while(1)
  {
    getBTN();
    if(btn!=lastbtn)
    {
    switch(btn)
    {
      case 1:
        writeWord("USER 1");
        mode(btn, modesel);
        swDelay(2);
        clearLCD();
        break;
      case 2:
        writeWord("USER 2");
        mode(btn, modesel);
        swDelay(2);
        clearLCD();
        break;
      case 3:
        writeWord("USER 3");
        mode(btn, modesel);
        swDelay(2);
        clearLCD();
        break;
      case 4:
        clearLCD();
        writeWord("SWITCH");
        if(modesel==1)
          {modesel=0;}
        else
          {modesel=1;}
        swDelay(2);
        clearLCD();
        break;
      default: writeWord("MODE   "); writeLetter(2, modesel+0x30);
    }
    }
  }
}

/******************** initSys() ****************************/
void init_sys(void)
{
  initLCD();                   // Setup LCD for work
  clearLCD();                  // Clear LCD display
  FLL_CTL0 |= XCAP18PF;                     // Configure load caps
```

```
  P2SEL |= 0x30;                            // P2.4,5 = USART0 TXD/RXD
  ME1 |= UTXE0 + URXE0;                     // Enable USART0 TXD/RXD
  UCTL0 |= CHAR;                            // 8-bit character
  UTCTL0 |= SSEL1;                          // UCLK = SMCLK
  UBR00 = 0x6d;                             // 1MHz 9600
  UBR10 = 0x00;                             // 1MHz 9600
  UMCTL0 = 0x03;                            // modulation
  UCTL0 &= ~SWRST;                          // Initialize USART state machine
  IE1 |= URXIE0;                            // Enable USART0 RX interrupt
  P2DIR |= 0x10;                            // P2.4 output direction
  P3SEL &= ~(BIT0|BIT1|BIT2|BIT3);          // P3.0-3 I/O Function
  P3DIR = 0x0F;                             // P3.0-3 Output Direction
  P3OUT=0x01;
  _BIS_SR(GIE);                             //General Interupt Enable
}


/* The functions below are from MSL90: MSP430F449 LCD Driver Code for
   MSP430F449STK-2 Starter Kit from Olimex.com
   ------------------------------------------------------------------------
*  Author Details   :  Muneem Shahriar
                       Electrical Engineering & Mathematics (Senior)
                       Texas Tech University, Lubbock, TX, USA
                       Email: muneem.shahriar@ttu.edu
* ------------------------------------------------------------------------ */

// *************************** initLCD ************************************
void initLCD(void)   // initialize the various registers for LCD to work
{                    // (code obtained from sample demos of MSP430F449)
    FLL_CTL0 = XCAP10PF;          //set load capacitance for 32k xtal
    // Initialize LCD driver (4Mux mode)
    LCDCTL = LCDSG0_7 + LCD4MUX + LCDON; // 4mux LCD, segs16-23 = outputs
    BTCTL  = BT_fLCD_DIV128;             // set LCD frame freq = ACLK
    P5SEL  = 0xFC;                       // set Rxx and COM pins for LCD
}


// *************************** clearLCD ***********************************
void clearLCD(void) // makes the LCD blank
{                   // clear LCD memory to clear display
    unsigned int iLCD;
    for (iLCD =0; iLCD<20; iLCD++)  // clears all 20 LCD memory segments
    {
      LCD[iLCD] = 0;
    }
}


// ************************  writeLetter **********************************
void writeLetter(int position,char letter) // writes single character on the LCD.
{                                          // User can specify position as well
    // DO NOT PLAY WITH THE CODE BELOW ----------------------------------------
    if (position == 1)  // this is position adjustment for compatibility
      position = position + 6;
    else
      if ( (position > 1) &  (position < 8) )
        position = ((position * 2) - 1) + 6;  // adjust position
    // ------------------------------------------------------------------------

    switch(letter)
    {
       // letter  // LCDM7                          // LCDM8                          // End
       case 'A':  LCD[position-1] = a + b + c + e;    LCD[position] = b + c + g;        break;
       case 'B':  LCD[position-1] = c + h + e;        LCD[position] = b + c + g;        break;
       case 'C':  LCD[position-1] = a + h;            LCD[position] = b + c;            break;
       case 'D':  LCD[position-1] = b + c + h + e;    LCD[position] = c + g;            break;
       case 'E':  LCD[position-1] = a + h + e;        LCD[position] = b + c + g;        break;
       case 'F':  LCD[position-1] = a;                LCD[position] = b + c + g;        break;
       case 'G':  LCD[position-1] = a + c + h + e;    LCD[position] = b + c;            break;
       case 'H':  LCD[position-1] = b + c + e;        LCD[position] = b + c + g;        break;
```

```
        case 'I':  LCD[position-1] = a + h + f;          LCD[position] = d;              break;
        case 'J':  LCD[position-1] = b + h + c;          LCD[position] = c;              break;
        case 'K':  LCD[position-1] = d + g;              LCD[position] = b + c + g;      break;
        case 'L':  LCD[position-1] = h;                  LCD[position] = b + c ;         break;
        case 'M':  LCD[position-1] = b + c + g;          LCD[position] = b + c + f;      break;
        case 'N':  LCD[position-1] = b + c + d;          LCD[position] = b + c + f;      break;
        case 'O':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;          break;
        case 'P':  LCD[position-1] = a + b + e;          LCD[position] = b + c + g;      break;
        case 'Q':  LCD[position-1] = a + b + c + h + d;  LCD[position] = b + c;          break;
        case 'R':  LCD[position-1] = a + b + d + e;      LCD[position] = b + c + g;      break;
        case 'S':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;          break;
        case 'T':  LCD[position-1] = a + f + b;          LCD[position] = d + b;          break;
        case 'U':  LCD[position-1] = b + c + h;          LCD[position] = b + c;          break;
        case 'V':  LCD[position-1] = g;                  LCD[position] = b + c + e;      break;
        case 'W':  LCD[position-1] = b + c + d;          LCD[position] = b + c + e;      break;
        case 'X':  LCD[position-1] = d + g;              LCD[position] = e + f;          break;
        case 'Y':  LCD[position-1] = b + c + h + e;      LCD[position] = f;              break;
        case 'Z':  LCD[position-1] = a + h + g;          LCD[position] = e;              break;

        // number  // LCDM7                             // LCDM8                        // END
        case '0':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;          break;
        case '1':  LCD[position-1] = b + c;              LCD[position] = d & a;          break;
        case '2':  LCD[position-1] = a + b + e + h;      LCD[position] = c + g;          break;
        case '3':  LCD[position-1] = a + b + c + e + h;  LCD[position] = g;              break;
        case '4':  LCD[position-1] = b + c + e;          LCD[position] = b + g;          break;
        case '5':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;          break;
        case '6':  LCD[position-1] = a + c + h + e;      LCD[position] = b + c + g;      break;
        case '7':  LCD[position-1] = a + b + c;          LCD[position] = d & a;          break;
        case '8':  LCD[position-1] = a + b + c + e + h;  LCD[position] = b + c + g;      break;
        case '9':  LCD[position-1] = a + b + c + e ;     LCD[position] = b + g;          break;

        // others
        case '.':                                        LCD[position] = h;              break;
// decimal point
        case '^':                                        LCDM2 = c;                      break;
// top arrow
        case '!':                                        LCDM2 = a;                      break;
// bottom arrow
        case '>':                                        LCDM2 = b;                      break;
// right arrow
        case '<':                                        LCDM2 = h;                      break;
// left arrow
        case '+':                                        LCDM20= a;                      break;
// plus sign
        case '-':                                        LCDM20= h;                      break;
// minus sign
        case '&':                                        LCDM2 = d;                      break;
// zero battery
        case '*':                                        LCDM2 = d + f;                  break;
// low battery
        case '(':                                        LCDM2 = d + f + g;              break;
// medium battery
        case ')':                                        LCDM2 = d + e + f + g;          break;
// full battery */
     }
}


// ****************************  writeWord ***********************************
void writeWord(const char *word)  // displays a word upto 7 characters -- why 7?
                                  // words must be in upper case (why?)
{
   unsigned int strLength = 0;   // variable to store length of word
   unsigned int i;               // dummy variable

   strLength = strlen(word);     // get the length of word now
   for (i = 1; i <= strLength; i++) // display word
   {
     writeLetter(strLength - i + 1,word[i-1]);  // displays each letter in the word
   }
```

```
}
int  getBTN()
{
   lastbtn=btn;
   char   inReg;

   /* Remember: Buttons are implemented "active low". That is, the input pin
      will equal logic 0 when the button is pressed and 1 otherwise*/

   inReg = (P3IN >> 4) & 0x0F;  /* Read the input register and shift bits 7-4 to be bits 3-0 then
mask out
      just the low nibble */
   if (inReg == 0x0E)
     btn = 1;
   else if (inReg == 0x0D)
     btn = 2;
   else if (inReg == 0x0B)
     btn = 3;
   else if (inReg == 0x07)
     btn = 4;
   else
     btn = 0;
   return(btn);
}

/****************** swDelay() ************************/
void swDelay(unsigned int max_cnt)
{
   unsigned int  cnt1=0, cnt2;

   while (cnt1 < max_cnt)
   {
     cnt2 = 0;
     while (cnt2 < 65535)
       cnt2++;
     cnt1++;
   }
}

/******************** Tx1()*************************/
void Tx1(char TxTemp)
{
  WirelessEnable();
  while ((IFG1 & UTXIFG0) == 0); {U0TXBUF = TxTemp;}
  P3OUT=0x01;
  writeWord("LOCKOUT");
  TimerB2s();
  while(wait==1);
  clearLCD();

}
/******************** Tx2()*************************/
void Tx2(char TxTemp)
{
  WirelessEnable();
  while ((IFG1 & UTXIFG0) == 0); {U0TXBUF = TxTemp;}
  P3OUT=0x00;
}


/*********************Mode************************/
void mode(int MdTemp, int mode)
{
  if(mode==0)
  {
      switch(MdTemp){
        case 1: Tx1('1'); break;
        case 2: Tx1('2'); break;
        case 3: Tx1('3'); break;}
  }
  else if(mode==1){
```

```
      switch(MdTemp){
        case 1: Tx2('a'); Tx2('2');  break;
        case 2: Tx2('c'); Tx2('1');  break;
        case 3: Tx2('c'); Tx2('2');  break;}
  }
}


/*****************************TimerB15ms()*****************************/
void TimerB15ms(void)
{
  TBCTL= TBSSEL_1 + CNTL_0 + ID_0 + MC_1;
  TBCCR0=0x1EC;
  TBCCTL0=CCIE;
  wait=1;
}

/*****************************TimerB2s()*****************************/
void TimerB2s(void)
{
  TBCTL= TBSSEL_1 + CNTL_0 + ID_0 + MC_1;
  TBCCR0=0xFFFF;
  TBCCTL0=CCIE;
  wait=1;
}
//*************************** Timer B interupt*****************
#pragma vector=TIMERB0_VECTOR
__interrupt void timerB0_ISR(void)
{
wait=0;
TimerBStop();
}
/*****************************TimerBStop()*************************/
void TimerBStop(void)
{
  TBCTL= MC_0;
}

/*************************WirelessEnable()*************************/
void WirelessEnable(void)
{
  P3OUT=0x00;
  TimerB15ms();
  while(wait==1);
}
```

ADMINISTRATION.C

```
#include "msp430x44x.h"      // Definitions, constants, etc for msp430F449
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <in430.h>

void init_sys(void);
void clearLCD(void);         // Clears LCD memory segments so that LCD is blank
void initLCD(void);          // Setup code to interface LCD with MSP430F449
void writeLetter(int position,char letter);  // display single character on LCD
void writeWord(const char *word);    // displays words upto 7 chars on LCD. Can
int  getBTN(void);                   // also display numbers passed as text
void buzzerOn(void);    // turns buzzer on
void buzzerOff(void);   // turns buzzer off
void swDelay(unsigned int max_cnt);   // simple SW delay loop
void display(void);  //screen refresh multipler
void Excecute(void); // Preforms the action based on the Multi(tier) Variables
void Tx(char TxTemp); //Cosmetic Tx Operation
```

```
void calc(int CalcTemp);

//****************************************** LCD CONSTANTS
******************************************************
// From Muneem Shahriar's LCD driver code from Olimex.com
#define     a       (0x80)    // definitions for LCD seegments on the Olimex LCD. 4-Mux operation
is assumed
#define     b       (0x40)    // For more details on 4-Mux operation, gather your LCD datasheet,
#define     c       (0x20)    // TI's MSP430F449 User Guide (look for LCD Controller, then 4-Mux),
#define     d       (0x01)    // and MSP-449STK-2 schematic. You will need ALL these 3 when
defining
#define     e       (0x02)    // each number or character. Remember, the Olimex LCD doesn't use a
LCD driver!
#define     f       (0x08)    // You tell the LCD what characters to display. It's very time
consuming!!
#define     g       (0x04)
#define     h       (0x10)

char *LCD = LCDMEM;       // pointer to LCD Memory Segments
int    lastbtn=0;
int  Scores[3][5]= {0,0,0,0,0};
char ID='0', User='0';
int stage=0, field; int btn=0;
int multi1=0, multi2=0, multi3=0;
int Tier=3;
char fieldt='0';
int ten, hun, one;

int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  init_sys();
  clearLCD();
  display();
  while(1)
  {
  getBTN();
  ////////////////////////////////////////////////////////////
  if(ID!='0' && User!='0')                  //////////////////
  {                                         //////////////////
      switch(ID)                            //////////////////
      {                                     //////////////////
         case 'a': field=0; break;          //////////////////
         case 'b': field=1; break;          //////////////////
         case 'c': field=2; break;          //////////////////
         default: break;                    //////////////////
      }                                     //////////////////
                                            //////////////////
      Scores[field][User-0x30-1]+=1;          //////////////////
      ID='0';                               //////////////////
      User='0';                             //////////////////
  }                                         //////////////////
  ////////////////////////////////////////////////////////////
  if(btn!=lastbtn)
  {
    switch(btn)
    {
    case 1: Excecute(); break;
    case 2:
      if(Tier==3){
        if(multi3==2)
        {multi3=0;}
        else{multi3+=1;}}
      else if(Tier==2){
        if(multi2==2)
        {multi2=0;}
        else{multi2+=1;}}
      else if(Tier==1){
        if(multi1==4)
        {multi1=0;}
```

```
        else{multi1+=1;}}
      display();
       break;

    case 3: if(Tier<3){Tier+=1;  display();} break;
    case 4: if(Tier>1){Tier-=1;  display();} break;
    }

  }
}
}



void init_sys(void)
{
  initLCD();
  FLL_CTL0 |= XCAP18PF;                     // Configure load caps
  P2SEL |= 0x30;                            // P2.4,5 = USART0 TXD/RXD
  ME1 |= UTXE0 + URXE0;                     // Enable USART0 TXD/RXD
  UCTL0 |= CHAR;                            // 8-bit character
  UTCTL0 |= SSEL1;                          // UCLK = SMCLK
  UBR00 = 0x6d;                             // 1MHz 9600
  UBR10 = 0x00;                             // 1MHz 9600
  UMCTL0 = 0x03;                            // modulation
  UCTL0 &= ~SWRST;                          // Initialize USART state machine
  IE1 |= URXIE0;                            // Enable USART0 RX interrupt
  P2DIR |= 0x10;                            // P2.4 output direction
  _BIS_SR(GIE);
}

#pragma vector=UART0RX_VECTOR
__interrupt void usart0_rx (void)
{
  if(stage==0)
  {ID = U0RXBUF; stage=1;}
  else
  {stage=0; User=U0RXBUF;}
}


/* The functions below are from MSL90: MSP430F449 LCD Driver Code for
   MSP430F449STK-2 Starter Kit from Olimex.com
   ------------------------------------------------------------------------------
*  Author Details   :  Muneem Shahriar
                       Electrical Engineering & Mathematics (Senior)
                       Texas Tech University, Lubbock, TX, USA
                       Email: muneem.shahriar@ttu.edu
* ------------------------------------------------------------------------------ */

// *************************** initLCD ************************************
void initLCD(void)   // initialize the various registers for LCD to work
{                    // (code obtained from sample demos of MSP430F449)
    FLL_CTL0 = XCAP10PF;          //set load capacitance for 32k xtal
    // Initialize LCD driver (4Mux mode)
    LCDCTL = LCDSG0_7 + LCD4MUX + LCDON; // 4mux LCD, segs16-23 = outputs
    BTCTL  = BT_fLCD_DIV128;             // set LCD frame freq = ACLK
    P5SEL  = 0xFC;                       // set Rxx and COM pins for LCD
}


// *************************** clearLCD ***********************************
void clearLCD(void) // makes the LCD blank
{                   // clear LCD memory to clear display
    unsigned int iLCD;
    for (iLCD =0; iLCD<20; iLCD++)  // clears all 20 LCD memory segments
    {
      LCD[iLCD] = 0;
    }
}
```

```
// *************************  writeLetter ***********************************
void writeLetter(int position,char letter) // writes single character on the LCD.
{                                          // User can specify position as well
    // DO NOT PLAY WITH THE CODE BELOW ----------------------------------------
    if (position == 1)  // this is position adjustment for compatibility
      position = position + 6;
    else
      if ( (position > 1) &  (position < 8) )
        position = ((position * 2) - 1) + 6;  // adjust position
    // ------------------------------------------------------------------------

    switch(letter)
    {
       // letter  // LCDM7                              // LCDM8                          // End
       case 'A':  LCD[position-1] = a + b + c + e;      LCD[position] = b + c + g;        break;
       case 'B':  LCD[position-1] = c + h + e;          LCD[position] = b + c + g;        break;
       case 'C':  LCD[position-1] = a + h;              LCD[position] = b + c;            break;
       case 'D':  LCD[position-1] = b + c + h + e;      LCD[position] = c + g;            break;
       case 'E':  LCD[position-1] = a + h + e;          LCD[position] = b + c + g;        break;
       case 'F':  LCD[position-1] = a;                  LCD[position] = b + c + g;        break;
       case 'G':  LCD[position-1] = a + c + h + e;      LCD[position] = b + c;            break;
       case 'H':  LCD[position-1] = b + c + e;          LCD[position] = b + c + g;        break;
       case 'I':  LCD[position-1] = a + h + f;          LCD[position] = d;                break;
       case 'J':  LCD[position-1] = b + h + c;          LCD[position] = c;                break;
       case 'K':  LCD[position-1] = d + g;              LCD[position] = b + c + g;        break;
       case 'L':  LCD[position-1] = h;                  LCD[position] = b + c ;           break;
       case 'M':  LCD[position-1] = b + c + g;          LCD[position] = b + c + f;        break;
       case 'N':  LCD[position-1] = b + c + d;          LCD[position] = b + c + f;        break;
       case 'O':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;            break;
       case 'P':  LCD[position-1] = a + b + e;          LCD[position] = b + c + g;        break;
       case 'Q':  LCD[position-1] = a + b + c + h + d;  LCD[position] = b + c;            break;
       case 'R':  LCD[position-1] = a + b + d + e;      LCD[position] = b + c + g;        break;
       case 'S':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;            break;
       case 'T':  LCD[position-1] = a + f + b;          LCD[position] = d + b;            break;
       case 'U':  LCD[position-1] = b + c + h;          LCD[position] = b + c;            break;
       case 'V':  LCD[position-1] = g;                  LCD[position] = b + c + e;        break;
       case 'W':  LCD[position-1] = b + c + d;          LCD[position] = b + c + e;        break;
       case 'X':  LCD[position-1] = d + g;              LCD[position] = e + f;            break;
       case 'Y':  LCD[position-1] = b + c + h + e;      LCD[position] = f;                break;
       case 'Z':  LCD[position-1] = a + h + g;          LCD[position] = e;                break;

       // number  // LCDM7                             // LCDM8                          // END
       case '0':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;            break;
       case '1':  LCD[position-1] = b + c;              LCD[position] = d & a;            break;
       case '2':  LCD[position-1] = a + b + e + h;      LCD[position] = c + g;            break;
       case '3':  LCD[position-1] = a + b + c + e + h;  LCD[position] = g;                break;
       case '4':  LCD[position-1] = b + c + e;          LCD[position] = b + g;            break;
       case '5':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;            break;
       case '6':  LCD[position-1] = a + c + h + e;      LCD[position] = b + c + g;        break;
       case '7':  LCD[position-1] = a + b + c;          LCD[position] = d & a;            break;
       case '8':  LCD[position-1] = a + b + c + e + h;  LCD[position] = b + c + g;        break;
       case '9':  LCD[position-1] = a + b + c + e ;     LCD[position] = b + g;            break;

       // others
       case '.':                                        LCD[position] = h;                break;
// decimal point
       case '^':                                        LCDM2 = c;                        break;
// top arrow
       case '!':                                        LCDM2 = a;                        break;
// bottom arrow
       case '>':                                        LCDM2 = b;                        break;
// right arrow
       case '<':                                        LCDM2 = h;                        break;
// left arrow
       case '+':                                        LCDM20= a;                        break;
// plus sign
       case '-':                                        LCDM20= h;                        break;
// minus sign
       case '&':                                        LCDM2 = d;                        break;
// zero battery
```

```
      case '*':                                          LCDM2 = d + f;                  break;
// low battery
      case '(':                                          LCDM2 = d + f + g;              break;
// medium battery
      case ')':                                          LCDM2 = d + e + f + g;          break;
// full battery */
    }
}


// ****************************  writeWord *********************************
void writeWord(const char *word)  // displays a word upto 7 characters -- why 7?
                                  // words must be in upper case (why?)
{
   unsigned int strLength = 0;   // variable to store length of word
   unsigned int i;               // dummy variable

   strLength = strlen(word);     // get the length of word now
   for (i = 1; i <= strLength; i++) // display word
   {
     writeLetter(strLength - i + 1,word[i-1]);  // displays each letter in the word
   }

}
int  getBTN()
{
   lastbtn=btn;
   char   inReg;

   /* Remember: Buttons are implemented "active low". That is, the input pin
      will equal logic 0 when the button is pressed and 1 otherwise*/

   inReg = (P3IN >> 4) & 0x0F;  /* Read the input register and shift bits 7-4 to be bits 3-0 then
mask out
      just the low nibble */
   if (inReg == 0x0E)
     btn = 1;
   else if (inReg == 0x0D)
     btn = 2;
   else if (inReg == 0x0B)
     btn = 3;
   else if (inReg == 0x07)
     btn = 4;
   else
     btn = 0;
   return(btn);
}

/****************** swDelay() ***********************/
void swDelay(unsigned int max_cnt)
{
   unsigned int  cnt1=0, cnt2;

   while (cnt1 < max_cnt)
   {
     cnt2 = 0;
     while (cnt2 < 65535)
       cnt2++;
     cnt1++;
   }
}

/******************Display()*****************************/
void display(void)
{
  clearLCD();
  if(Tier==3){
    if     (multi3==0){writeWord("CALLCLY");}
    else if(multi3==1){writeWord("STANDBY");}
    else if(multi3==2){writeWord("RM USER");}
  }
```

```
  if(Tier==2){
    if      (multi2==0){writeWord("FIELD A"); fieldt='a';}
    else if(multi2==1){writeWord("FIELD B"); fieldt='b';}
    else if(multi2==2){writeWord("FIELD C"); fieldt='c';}
  }
  if(Tier==1){
    if      (multi1==0){writeWord("USER 1");}
    else if(multi1==1){writeWord("USER 2");}
    else if(multi1==2){writeWord("USER 3");}
    else if(multi1==3){writeWord("USER 4");}
    else if(multi1==4){writeWord("USER 5");}
  }
}
/************************Excecute()*********************/
void Excecute(void)
{
  if(multi3==0)
  {
    calc(Scores[multi2][multi1]);
    clearLCD(); writeWord("FIELD  "); writeLetter(1,0x30+multi2+1); swDelay(3);
    clearLCD(); writeWord("USER   "); writeLetter(1,0x30+multi1+1); swDelay(3); clearLCD();
    writeLetter(2,ten+0x30);
    writeLetter(3,hun+0x30);
    writeLetter(1,one+0x30);
    swDelay(3); clearLCD();
  }
  if(multi3==1)
  {Tx(field); Tx(0x2E); }
  if(multi3==2)
  {Tx(field); Tx(0x2F); Tx(multi1+0x30);}
  display();
}



/************************Tx()*********************/
void Tx(char TxTemp)
{
  while ((IFG1 & UTXIFG0) == 0); {U0TXBUF = TxTemp;}
}

/********************calc()***************/
void calc(int CalcTemp)
{
  hun=0; ten=0; one=0;
  while(CalcTemp>=100)
  {CalcTemp-=100; hun+=1;}
  while(CalcTemp>=10)
  {CalcTemp-=10; ten+=1;}
  while(CalcTemp>=1)
  {CalcTemp-=1; one+=1;}
}
```

RRELAY.C

```
//Inclusive Logic Demonstration
#include "msp430x44x.h"      // Definitions, constants, etc for msp430F449
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <in430.h>

// ************************************* FUNCTION DECLARATIONS
**********************************************
void init_sys(void);      // MSP430 Initialization routine

void clearLCD(void);         // Clears LCD memory segments so that LCD is blank
void initLCD(void);          // Setup code to interface LCD with MSP430F449
void writeLetter(int position,char letter);  // display single character on LCD
void writeWord(const char *word);    // displays words upto 7 chars on LCD. Can
int  getBTN(void);                   // also display numbers passed as text
void buzzerOn(void);    // turns buzzer on
void buzzerOff(void);   // turns buzzer off
void swDelay(unsigned int max_cnt);   // simple SW delay loop

void UserDetection(void); //Detects user presses simulation for post signal identification
void ArrayManagement(void); // Checks array state changes
void SpecialArray(void); // Checks B state Array Exception Conditions
void FirePulse(void); //The Firing trigger mechanism for the trap machine
void NewUser(void); //Checks if the user is not in the array and ajusts firing order
void RemoveUserDetection(void); //Removes a user if they miss thier spot more then twice
void ErrorDec(void); // Detects if an illegal repetition has occured in array formation
void reset(void); // software reset without power loss or interupt vector
void RMT(int RT);
void Tx(char TxTemp);


/*************************************************************************/
/*                                                                       */
/* Global variable declarations                                          */
/*                                                                       */
/*************************************************************************/

char *LCD = LCDMEM;       // pointer to LCD Memory Segments
int    lastbtn=0, btn=0, ACTIVE;
int ExpectedUserArray[5] = {0,0,0,0,0};
int RemoveVector[5]={0,0,0,0,0};
int internal=0, Apointer=0, ArrayLimit, OVERRIDE, TempEx, internal;
int triggercount=0, active=0, QVB='0';  int RMinc=0;
char state = 'Z', temp='0';
const char ID='b';
char Queue1='0', IDhold; char LastState='Z';
/*************************************************************************/
/*                                                                       */
/* main() variable declarations                                          */
/*                                                                       */
/*************************************************************************/

//****************************************** LCD CONSTANTS
*****************************************************
// From Muneem Shahriar's LCD driver code from Olimex.com
#define     a      (0x80)    // definitions for LCD seegments on the Olimex LCD. 4-Mux operation
is assumed
#define     b      (0x40)    // For more details on 4-Mux operation, gather your LCD datasheet,
#define     c      (0x20)    // TI's MSP430F449 User Guide (look for LCD Controller, then 4-Mux),
#define     d      (0x01)    // and MSP-449STK-2 schematic. You will need ALL these 3 when
defining
#define     e      (0x02)    // each number or character. Remember, the Olimex LCD doesn't use a
LCD driver!
#define     f      (0x08)    // You tell the LCD what characters to display. It's very time
consuming!!
#define     g      (0x04)
#define     h      (0x10)

/******************** MAIN FUNCTION *********************/
void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
  init_sys();                  // Initialize the MSP430
/* Loop forever waiting for input (where would you go if you exited?) */
```

```
  while(1)
  {
    if(active==1){

      switch(temp)
      {
      case '1': writeWord("TRIG 1"); swDelay(2); clearLCD(); break;
      case '2': writeWord("TRIG 2"); swDelay(2); clearLCD(); break;
      case '3': writeWord("TRIG 3"); swDelay(2); clearLCD(); break;
      case '4': writeWord("TRIG 4"); swDelay(2); clearLCD(); break;
      case '5': writeWord("TRIG 5"); swDelay(2); clearLCD(); break;
      default: clearLCD();
      }
      internal=temp-0x30;
      ErrorDec();
      if(active==1){
      ArrayManagement();
      SpecialArray();
      Tx(ID); Tx(internal+0x30);
      }
      internal=0; temp='0'; active=0;
    }
  }
}

#pragma vector=UART0RX_VECTOR
__interrupt void usart0_rx (void)
{
  temp=U0RXBUF;
  if(temp>0x60 && temp<0x7B)
  {QVB=1; IDhold=temp;}
  else if(QVB==1  && temp>0x2D && temp<0x36)
  {
    Queue1=U0RXBUF; QVB=0;
    if(IDhold>ID)
    {
      clearLCD(); writeWord("ZNP    "); writeLetter(4,ID-0x20); writeLetter(2,IDhold-0x20);
writeLetter(1,Queue1);
      while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = IDhold;
      while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = Queue1;
      swDelay(5);
      clearLCD();
    }
    else if(IDhold==ID)
    {
      if(temp==0x2E  && state!='C'){
        LastState=state; state='C'; active=0; clearLCD(); writeWord("STANDBY");}
      else if(temp==0x2E && state=='C')
      {state=LastState; active=0; clearLCD(); writeWord("WAKE UP");}
      else if(temp==0x2F){RMinc=1;}
    }
  }
  else if(RMinc==1)
  {RMT(temp-0x30); RMinc=0;}
  else if(temp>0x30 && temp<0x36)
  {active=1;}
}

/******************ArrayManagement() ***********************/
void ArrayManagement(void)
{
  if(internal !='0'){
  switch (state){
  case 'Z':
      ExpectedUserArray[0]= internal; Apointer=1; state ='A'; ArrayLimit=1; FirePulse();
      if (OVERRIDE ==1) { state='C';}
      break;
  case 'A':
    if (OVERRIDE ==1) { state='C';}
    if(ExpectedUserArray[0]==internal || Apointer==5) // Doesnt check if non first shooter
repeats
```

```
      { state='B'; ArrayLimit=Apointer; Apointer=1; FirePulse(); }
    else
      {ExpectedUserArray[Apointer]= internal; Apointer+=1; ArrayLimit=Apointer; FirePulse();}
      break;
  case 'B':
    if (OVERRIDE ==1) { state='C';}
    else if ( ExpectedUserArray[Apointer]==internal)
      {
      FirePulse();
      Apointer+=1;}
    else if (ExpectedUserArray[Apointer-1]==internal)
       {
         FirePulse();
       }
    else if (Apointer==0)
    {
      if (ExpectedUserArray[ArrayLimit-1]==internal)
      {
        FirePulse();
      }
    }
    else if (internal==TempEx)
    {FirePulse();
    Apointer+=2;
     TempEx=0;}
    else{TempEx=internal;}
    if(Apointer==ArrayLimit){Apointer=0;}
    if(Apointer==ArrayLimit+1){Apointer=1;}
    break;
  case 'C': clearLCD(); writeWord("STANDBY");
  default: break;}
  }
}

/******************UserDetection()*************************/
void UserDetection(void)
{
  if(btn!=lastbtn){
    clearLCD();
        if      (btn==1) {
          clearLCD(); writeWord("USER 1 ");
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = btn+0x30;}
        else if (btn==2) {
          clearLCD(); writeWord("USER 2 ");
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = btn+0x30;}
        else if (btn==3) {
          clearLCD(); writeWord("USER 3 ");
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = btn+0x30;}
        else if (btn==4) {
          clearLCD(); writeWord("USER 4 ");
          while ((IFG1 & UTXIFG0) == 0);  U0TXBUF = btn+0x30;}
        else             { clearLCD(); writeWord("       ");}
  }
}
/******************* initSys() ****************************/
void init_sys(void)
{
  initLCD();                    // Setup LCD for work
  clearLCD();                   // Clear LCD display
  FLL_CTL0 |= XCAP18PF;                       // Configure load caps
  P2SEL |= 0x30;                              // P2.4,5 = USART0 TXD/RXD
  ME1 |= UTXE0 + URXE0;                       // Enable USART0 TXD/RXD
  UCTL0 |= CHAR;                              // 8-bit character
  UTCTL0 |= SSEL1;                            // UCLK = SMCLK
  UBR00 = 0x6d;                               // 1MHz 9600
  UBR10 = 0x00;                               // 1MHz 9600
  UMCTL0 = 0x03;                              // modulation
  UCTL0 &= ~SWRST;                            // Initialize USART state machine
  IE1 |= URXIE0;                              // Enable USART0 RX interrupt
  P2DIR |= 0x10;                              // P2.4 output direction
  _BIS_SR(GIE);
```

```
}

/* The functions below are from MSL90: MSP430F449 LCD Driver Code for
   MSP430F449STK-2 Starter Kit from Olimex.com
   ------------------------------------------------------------------------
*  Author Details   :  Muneem Shahriar
                       Electrical Engineering & Mathematics (Senior)
                       Texas Tech University, Lubbock, TX, USA
                       Email: muneem.shahriar@ttu.edu
* ------------------------------------------------------------------------ */

// *************************** initLCD ************************************
void initLCD(void)   // initialize the various registers for LCD to work
{                    // (code obtained from sample demos of MSP430F449)
    FLL_CTL0 = XCAP10PF;          //set load capacitance for 32k xtal
    // Initialize LCD driver (4Mux mode)
    LCDCTL = LCDSG0_7 + LCD4MUX + LCDON; // 4mux LCD, segs16-23 = outputs
    BTCTL  = BT_fLCD_DIV128;             // set LCD frame freq = ACLK
    P5SEL  = 0xFC;                       // set Rxx and COM pins for LCD
}


// *************************** clearLCD ***********************************
void clearLCD(void) // makes the LCD blank
{                   // clear LCD memory to clear display
    unsigned int iLCD;
    for (iLCD =0; iLCD<20; iLCD++)  // clears all 20 LCD memory segments
    {
      LCD[iLCD] = 0;
    }
}


// ***************************  writeLetter ********************************
void writeLetter(int position,char letter) // writes single character on the LCD.
{                                          // User can specify position as well
    // DO NOT PLAY WITH THE CODE BELOW ----------------------------------------
    if (position == 1)  // this is position adjustment for compatibility
      position = position + 6;
    else
      if ( (position > 1) &  (position < 8) )
        position = ((position * 2) - 1) + 6;  // adjust position
    // ------------------------------------------------------------------------

    switch(letter)
    {
       // letter  // LCDM7                          // LCDM8                          // End
       case 'A':  LCD[position-1] = a + b + c + e;      LCD[position] = b + c + g;        break;
       case 'B':  LCD[position-1] = c + h + e;          LCD[position] = b + c + g;        break;
       case 'C':  LCD[position-1] = a + h;              LCD[position] = b + c;            break;
       case 'D':  LCD[position-1] = b + c + h + e;      LCD[position] = c + g;            break;
       case 'E':  LCD[position-1] = a + h + e;          LCD[position] = b + c + g;        break;
       case 'F':  LCD[position-1] = a;                  LCD[position] = b + c + g;        break;
       case 'G':  LCD[position-1] = a + c + h + e;      LCD[position] = b + c;            break;
       case 'H':  LCD[position-1] = b + c + e;          LCD[position] = b + c + g;        break;
       case 'I':  LCD[position-1] = a + h + f;          LCD[position] = d;                break;
       case 'J':  LCD[position-1] = b + h + c;          LCD[position] = c;                break;
       case 'K':  LCD[position-1] = d + g;              LCD[position] = b + c + g;        break;
       case 'L':  LCD[position-1] = h;                  LCD[position] = b + c ;           break;
       case 'M':  LCD[position-1] = b + c + g;          LCD[position] = b + c + f;        break;
       case 'N':  LCD[position-1] = b + c + d;          LCD[position] = b + c + f;        break;
       case 'O':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;            break;
       case 'P':  LCD[position-1] = a + b + e;          LCD[position] = b + c + g;        break;
       case 'Q':  LCD[position-1] = a + b + c + h + d;  LCD[position] = b + c;            break;
       case 'R':  LCD[position-1] = a + b + d + e;      LCD[position] = b + c + g;        break;
       case 'S':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;            break;
       case 'T':  LCD[position-1] = a + f + b;          LCD[position] = d + b;            break;
       case 'U':  LCD[position-1] = b + c + h;          LCD[position] = b + c;            break;
       case 'V':  LCD[position-1] = g;                  LCD[position] = b + c + e;        break;
       case 'W':  LCD[position-1] = b + c + d;          LCD[position] = b + c + e;        break;
       case 'X':  LCD[position-1] = d + g;              LCD[position] = e + f;            break;
```

```
        case 'Y':  LCD[position-1] = b + c + h + e;      LCD[position] = f;                break;
        case 'Z':  LCD[position-1] = a + h + g;          LCD[position] = e;                break;

        // number  // LCDM7                              // LCDM8                          // END
        case '0':  LCD[position-1] = a + b + c + h;      LCD[position] = b + c;            break;
        case '1':  LCD[position-1] = b + c;              LCD[position] = d & a;            break;
        case '2':  LCD[position-1] = a + b + e + h;      LCD[position] = c + g;            break;
        case '3':  LCD[position-1] = a + b + c + e + h;  LCD[position] = g;                break;
        case '4':  LCD[position-1] = b + c + e;          LCD[position] = b + g;            break;
        case '5':  LCD[position-1] = a + c + h + e;      LCD[position] = b + g;            break;
        case '6':  LCD[position-1] = a + c + h + e;      LCD[position] = b + c + g;        break;
        case '7':  LCD[position-1] = a + b + c;          LCD[position] = d & a;            break;
        case '8':  LCD[position-1] = a + b + c + e + h;  LCD[position] = b + c + g;        break;
        case '9':  LCD[position-1] = a + b + c + e ;     LCD[position] = b + g;            break;

        // others
        case '.':                                        LCD[position] = h;                break;
// decimal point
        case '^':                                        LCDM2 = c;                        break;
// top arrow
        case '!':                                        LCDM2 = a;                        break;
// bottom arrow
        case '>':                                        LCDM2 = b;                        break;
// right arrow
        case '<':                                        LCDM2 = h;                        break;
// left arrow
        case '+':                                        LCDM20= a;                        break;
// plus sign
        case '-':                                        LCDM20= h;                        break;
// minus sign
        case '&':                                        LCDM2 = d;                        break;
// zero battery
        case '*':                                        LCDM2 = d + f;                    break;
// low battery
        case '(':                                        LCDM2 = d + f + g;                break;
// medium battery
        case ')':                                        LCDM2 = d + e + f + g;            break;
// full battery */
    }
}


// ******************************  writeWord ***********************************
void writeWord(const char *word)  // displays a word upto 7 characters -- why 7?
                                  // words must be in upper case (why?)
{
   unsigned int strLength = 0;   // variable to store length of word
   unsigned int i;               // dummy variable

   strLength = strlen(word);     // get the length of word now
   for (i = 1; i <= strLength; i++) // display word
   {
     writeLetter(strLength - i + 1,word[i-1]);  // displays each letter in the word
   }

}
int  getBTN()
{
   lastbtn=btn;
   char   inReg;

   /* Remember: Buttons are implemented "active low". That is, the input pin
      will equal logic 0 when the button is pressed and 1 otherwise*/

   inReg = (P3IN >> 4) & 0x0F;  /* Read the input register and shift bits 7-4 to be bits 3-0 then
mask out
      just the low nibble */
   if (inReg == 0x0E){
     btn = 1; active=1;}
   else if (inReg == 0x0D){
     btn = 2; active=1;}
```

```
   else if (inReg == 0x0B){
     btn = 3; active=1;}
   else if (inReg == 0x07){
     btn = 4; active=1;}
   else
     btn = 0;
   return(btn);
}

/****************** swDelay() ************************/
void swDelay(unsigned int max_cnt)
{
   unsigned int  cnt1=0, cnt2;

   while (cnt1 < max_cnt)
   {
     cnt2 = 0;
     while (cnt2 < 65535)
       cnt2++;
     cnt1++;
   }
}

/******************FirePulse()**************************/
void FirePulse(void)
{
 clearLCD();
 writeWord("TRIGGER");
 triggercount+=1;
 swDelay(2);
 clearLCD();
}
/******************SpecialArray()*******************/
void SpecialArray(void)
{
  NewUser();
  RemoveUserDetection();
}

/******************NewUser()************************/
void NewUser(void)
{
  int count=0;
  if(state=='B' && internal!=0)
  {
    for(int i=0; i<=ArrayLimit-1; i++)
    {
      if(internal!=ExpectedUserArray[i])
      {count+=1;}
    }
    if(count==ArrayLimit)
    {
      for(int i=ArrayLimit; i>Apointer; i--)
      {ExpectedUserArray[i]=ExpectedUserArray[i-1];}
      ExpectedUserArray[Apointer]=internal;
      Apointer+=1; ArrayLimit+=1;
    }
  }
}
/*******************************RemoveUser()**********************/
void RemoveUserDetection(void)
{
    if      (internal==ExpectedUserArray[Apointer+1] && state=='B')
    {RemoveVector[Apointer]+=1;}
    if      (internal==ExpectedUserArray[0]&& state=='B' && Apointer==ArrayLimit-1)
    {RemoveVector[Apointer]+=1;}
  for(int x=0; x<5; x++)
  {
    if(RemoveVector[x]>1)
    {
      int RemoveTarget=0;
```

```
      RemoveTarget=ExpectedUserArray[x];
      RMT(RemoveTarget);
    }
  }
}

/***************************RemoveTarget*****************************/
void RMT(int RT)
{
  for(int i=0; i<5; i++)
  {
    if(RT==ExpectedUserArray[i])
    {
      ExpectedUserArray[i]=ExpectedUserArray[i+1];
      ExpectedUserArray[i+1]=ExpectedUserArray[i+2];
      ExpectedUserArray[i+2]=ExpectedUserArray[i+3];
      ExpectedUserArray[i+3]=ExpectedUserArray[i+4];
      Apointer+=1; ArrayLimit-=1; RemoveVector[i]=0;
    }
    if(Apointer==ArrayLimit){Apointer=0;}
    if(Apointer==ArrayLimit+1){Apointer=1;}
  }
}
/*******************************ErrorDec()***************************/
void ErrorDec(void)
{
  int Ecnt=0;
  if(ExpectedUserArray[1]==internal)
  {Ecnt+=1;}
  if(ExpectedUserArray[2]==internal)
  {Ecnt+=1;}
  if(ExpectedUserArray[3]==internal)
  {Ecnt+=1;}
  if(ExpectedUserArray[4]==internal)
  {Ecnt+=1;}
  if(Ecnt>0 && state=='A')
  {
     clearLCD();
     writeWord("ERROR  ");
     writeLetter(1,internal+0x30);
     swDelay(2);
     reset();
     active=0;}
}

/***************************Reset()********************************/
void reset(void)
{
  ExpectedUserArray[0]=0;
  ExpectedUserArray[1]=0;
  ExpectedUserArray[2]=0;
  ExpectedUserArray[3]=0;

  ExpectedUserArray[4]=0;
  Apointer=0; ArrayLimit=0;
  lastbtn=0; temp='0'; internal=0; TempEx=0; state='Z'; btn=0; clearLCD();
}
/***********************Tx()*********************************/

void Tx(char TxTemp)
{
  while ((IFG1 & UTXIFG0) == 0); {U0TXBUF = TxTemp;}
}
```