Winter 2009

Wireless Personal Area Network-Based Assistance for the Visually Impaired

Kurt Matthew Peters
Old Dominion University

WIRELESS PERSONAL AREA NETWORK-BASED ASSISTANCE FOR THE VISUALLY IMPAIRED

by

Kurt Matthew Peters
M.S.E.E. December 1994, Air Force Institute of Technology
B.S.E.E. May 1990, United States Air Force Academy

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

ELECTRICAL AND COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
December 2009

Approved by:

_____
Sacharia Albin (Director)

_____
K. Vijayan Asari (Member)

_____
Mounir Laroussi (Member)

_____
John Cooper (Member)

ABSTRACT

WIRELESS PERSONAL AREA NETWORK-BASED ASSISTANCE FOR THE VISUALLY IMPAIRED

Kurt Matthew Peters
Old Dominion University, 2009
Director: Dr. Sacharia Albin

In this dissertation, a system allowing a visually impaired person to interact with his environment is developed using modern, low-power wireless communications techniques. With recent advances in wireless sensor networks, open-source operating systems, and embedded processing technology, low-cost devices have become practically feasible as a personal notification system for impaired people. Additionally, text-to-speech capabilities can now be employed without special application specific integrated circuits (ASICs), allowing low-cost, general-purpose processors to fill a niche that once required expensive semiconductors.

The system takes advantage of 802.15.4 and media access control (MAC) protocols offered by the open source operating system TinyOS. Important characteristics of these new standards that make them ideal for interface with humans are short range, low-power, and open-source software. To facilitate research and development in use and integration of such devices, we developed a hardware platform to allow exploration of possible future network architectures with multiple options for interfacing with the user. Our Visually Impaired Notification System (VINS) allows unprecedented awareness of the environment and has been simulated with multiple nodes using a modification of the TinyOS "Dissemination" protocol. This dissertation outlines the hardware platform, demonstration of a working prototype, and simulations of how the system would work in

its intended environment. We envision this system being used as a testbed allowing further research of other communications and message-delivery techniques. Additionally, the research has contributed directly to the TinyOS project and offered new insight into power management in embedded systems. Finally, through the research effort we were able to contribute to the open source movement and have produced software in four languages used in three countries with over 1500 downloads.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sacharia Albin, for his guidance and patience in helping me complete this dissertation. We have overcome many hurdles including some major changes in direction as the years passed.

I would like to acknowledge Dr. Mounir Laroussi, Dr. K. Vijayan Asari, and Dr. John Cooper for their insights as committee members and helpful discussions in achieving the goal of this research. I heartily thank my colleagues and my friends: Dr. Sachin Shetty and Dr. Min Song, who have contributed both in material and knowledge to this effort.

I dedicate my work to my loving family, Traci, Mayah, and now, Dean, the last two who were born while I was pursuing my PhD. Without their patience at the end, I would have never been able to complete the work.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

MOTIVATION AND SCOPE OF RESEARCH

Sight is, arguably, the most valuable of the five senses. Low vision affects about 14 million people and is among the ten most common causes of disability in the United States [1]. As life expectancy of the world's population increases, loss of vision will become an ever-increasing problem. Severe visual impairment can result from conditions such as age-related macular degeneration, cataracts, glaucoma, and diabetic retinopathy [2]. "Blindness and low vision affect approximately one in 28 Americans older than 40" [3]. As shown in Table 1, approximately a quarter of the population can expect some visual impairment once they are over the age of 80. This dissertation applies advances in the areas of haptics, wireless sensor networks, high-efficiency power supplies, and text-to-speech technology in an attempt to improve the quality of life of the vision impaired.

The device produced by this research applies the technology mentioned above in a miniature device, which can be worn by an individual to provide an interface to the outside world. One could eventually imagine such a device integrated into a walking stick or attached at the wrist. This technology could also apply, not only to someone suffering from a physical affliction, but to someone undergoing task saturation who needs added insight into his environment. For example, a soldier in an urban combat environment might use information from unattended ground sensors (UGS) to provide additional combat-related situational awareness (SA) such as detecting enemy proximity, automatic vehicle identification, or multi-hop, long-distance messaging. Figure 1

T. Park and M. Lee, "Power Saving Algorithms for Wireless Sensor Networks on IEEE 802.15.4," *IEEE Communications Magazine*, vol. 46, no. 6, June 2008, pp. 148-155.

Age	Blindness		Low Vision		All Vision Impaired	
(Years)	Persons	(%)	Persons	(%)	Persons	(%)
40-49	51,000	0.1%	80,000	0.2%	131,000	0.3%
50-59	45,000	0.1%	102,000	0.3%	147,000	0.4%
60-69	59,000	0.3%	176,000	0.9%	235,000	1.2%
70-79	134,000	0.8%	471,000	3.0%	605,000	3.8%
>80	648,000	7.0%	1,532,000	16.7%	2,180,000	23.7%
Total	937,000	0.8%	2,361,000	2.0%	3,298,000	2.7%

Table 1. Prevalence of blindness and low vision among adults 40 years and older in the United States [2].

shows how networked UGS could provide a combat soldier with unprecedented SA, possibly reducing numbers of deployed soldiers. The soldier in the scenario cares about his immediate surroundings, to ensure covert operations.

The goal of this research is to choose and implement an appropriate protocol and produce a device that can receive disseminated information from multiple base stations or peer devices. The received information provides added awareness of the environment that, otherwise, might go unnoticed. Specifically, we are examining how a small set of base stations, not limited in size, weight, or power, can transfer information periodically to a group of subscribers who stand to benefit from the information. The information, by its very nature, assumes close proximity to the base station for it to be of value. For example, a user near a bus stop needs to know that a bus is in-bound, its destination, and approximate arrival time. A patron to a vending machine may like to know relative location, whether it is within walking distance, and what goods are available. A blind

person approaching a construction site might like to know of impending obstacles or hazards and a recommended way to avoid them.

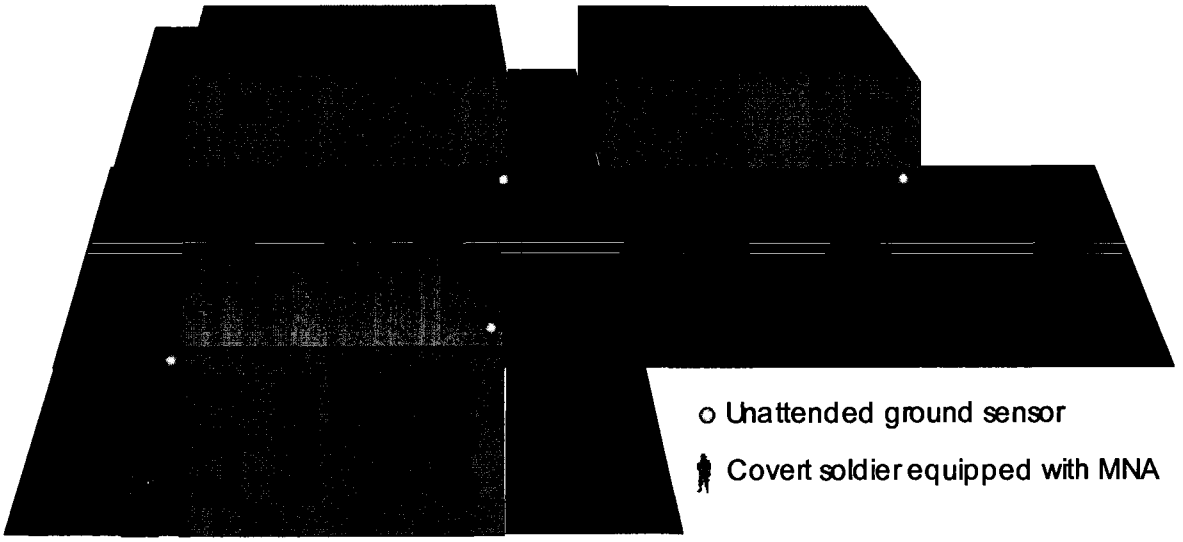


Figure 1. Example of how a soldier equipped with a miniature netted actuator (MNA) might benefit from this research.

Although the soldier scenario, above, seems similar, it differs in that any node can provide information to the soldier. The nodes sense information from the environment, and the soldier's receiver collects that information for presentation. In essence, it is the reverse of the "dissemination problem," in which a base station would like to disseminate information, for example a code update, to all the nodes in its network. These two problems, collection and dissemination, have received significant attention in the literature with a proposed set of solutions. We will concentrate on the dissemination problem in this dissertation.

WIRELESS PERSONAL AREA NETWORKS (WPAN)

Wireless personal area networks (WPAN) have achieved wide penetration in the marketplace, for example, most cell-phones and many automobiles are equipped with 802.15.1 devices, or the IEEE standard based on Bluetooth. A separate standard in the 802.15 family, but distinct from Bluetooth, has not received as much public acceptance yet, but is particularly designed for low-cost, low-bandwidth, low-power applications: 802.15.4. This technology is sometimes referred to as a low-rate, wireless personal area network (LR-WPAN), and only includes a description of the media access control (MAC) and physical (PHY) layers of the Open Systems Interconnection (OSI) stack model.

Unlike Bluetooth, 802.15.4 affords the user low latency and low energy consumption although sacrificing bandwidth. For instance, the node acquisition time for Bluetooth is approximately three seconds, while 802.15.4 is 30 ms. The wake-up time for Bluetooth is also three seconds, while it is 15 ms for 802.15.4. Bandwidth is significantly less than Bluetooth's 1-3 Mbps; the maximum bandwidth for 802.15.4 devices is 250 kbps in the 868/915 MHz and 2.4 GHz channels (Note: before release of the 2006 standard, the 868/915MHz channels of operation were limited to 20 or 40 kbps) [4]. Additionally, as shown in Figure 2, 802.15.4 has a flexible MAC layer allowing two topologies, Peer-to-Peer and Star. The Peer-to-Peer MAC allows Mesh networking technology to be employed where nodes forward messages to other nodes to reach a final destination. Additionally, Mesh networking allows for self-healing where the network can compensate for the loss of a node in real-time. Mesh networking compensates for the lack of range when using low-power radios. A drawback of using Mesh connectivity is

that the aggregate data rate significantly decreases as more nodes are added due to increased overhead costs of building routing tables and determining nearest neighbors. Gupta, et al. indicate that this degradation is proportional to $\frac{1}{n^{1.68}}$, where n is the number of nodes in a network [5,6].

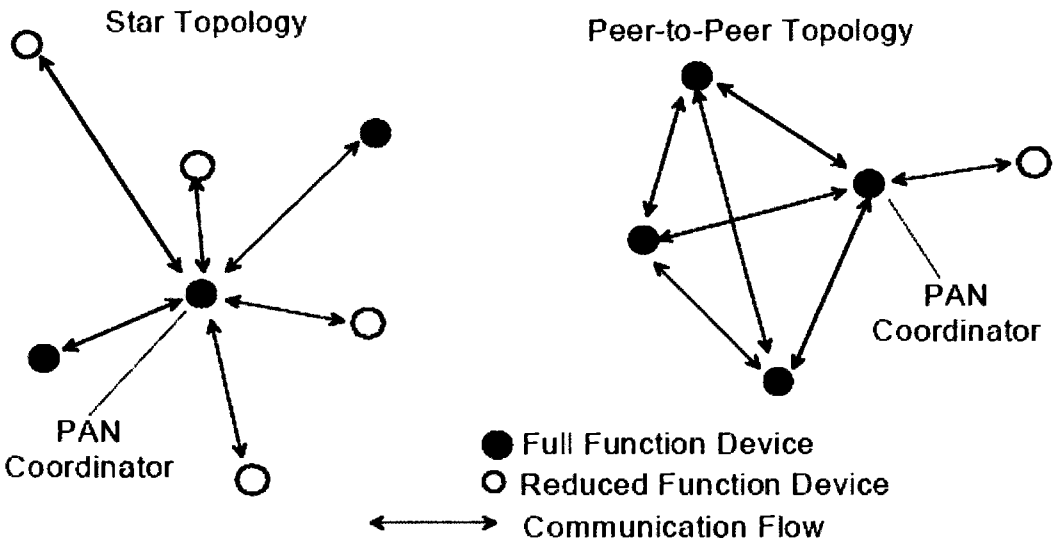


Figure 2. Star and peer-to-peer topology examples [4].

A sensor network protocol on top of 802.15.4 must support a variety of topologies, with random placement of nodes, with various densities of devices. A simple broadcast approach is: "Have every node in the network rebroadcast the message." This approach consumes too much bandwidth, causing a "broadcast storm problem" [7]. Based on the work of Ni, et al., [7] Levis, et al. [8] developed a protocol using "polite gossip" between nodes. This "Trickle protocol" has become a core networking primitive for practical dissemination of messages and code in a network [8]. We will examine the Trickle protocol in detail in the next chapter.

LOW-POWER REVOLUTION

Nothing short of a low-power revolution has taken place in the last decade. Besides 1) shrinking transistor sizes and development of "high-k" materials to achieve lower power consumption, 2) the advent and widespread use of buck-boost converters, 3) low-power or sleep modes, multiple operating clock speeds and voltages, and 4) new hardware capable of turning off portions of a chip not in use, techniques combining these technologies into a single system are coming to the forefront in engineering system design. Additionally, protocols synchronizing nodes to reduce listen time such as "low-power listen modes" are being developed. We will discuss the three prevalent low-power listening MACs in the next chapter. Our hardware platform takes advantage of each of these advances to supply a system that is arguably capable of lasting years on two AA batteries. Numerous commercial research platforms are available for both commercial and academic use, including Crossbow Motes (MICA, MicaZ, iMote). These boards take advantage of some of the hardware advances mentioned above, but require special attention on the part of the programmer to implement low-power processing aggressively. We combine the commercially available MicaZ mote with our board to create our communications platform.

BRAILLE, HAPTICS, AND TEXT-TO-SPEECH

Obviously, once the information is sent to the wearer's Mote, the next obstacle is how to deliver the message quickly, without the user having to divert their vision. In the case of a visually impaired person, we need to deliver the message to someone with limited ability to see. We review the prevailing literature examining proper placement, height, force, and duration of dots that would be used in a Braille device. We then review our

design of circuitry to drive a piezo-electric transducer to manipulate individual dots. Instead of Braille, a soldier in the scenario above, might, instead, receive a vibration on his shoulder alerting him of a preset condition. Additionally, we implement the Uncontracted (Grade 1) version of the English Braille American Addition developed under the Braille Authority of North America (BANA) [9].

Given the low bandwidth available with 802.15.4, we ascertained that transmitting voice would not be practical. Although, 802.15.4 specifies 250 kbps for node-to-node communications, actual aggregate throughput is more on the order of 10-20 kbps, especially when using a multi-hop network. Additionally, packets would have to be reordered once received, increasing storage requirements for nodes that are already memory starved. Instead, we examined availability of text-to-speech (TTS) technology. Our first board uses a Winbond TTS integrated circuit (IC) that is at the end-of-life. Digital signal processing has advanced to the degree that software-based TTS is now, not only practical, but makes these kinds of custom chips obsolete. We will review some of the available software-based TTS technologies in Chapter V.

EXPLOITING OPEN SOURCE

ELECTRONIC DESIGN AUTOMATION (EDA) SOFTWARE

All printed circuit boards were designed using open source software. Although numerous open-source packages are available, we chose gEDA to design our printed circuit boards [10]. gEDA provides all the tools necessary for analog and digital designs except a VHDL synthesis tool and simulator. Figure 3 shows a typical design flow using this EDA suite. We found the analog waveform viewer to be cumbersome and limited, so we wrote our own viewer, KJWaves. KJWaves is available on sourceforge.net and

has had over 1500 downloads and has received support to translate it into four different languages: English, Spanish, German, and Greek. A description of the design and techniques we used are described in Chapter III.

TINYOS

Developed by University of California at Berkley as standard programming interface for Motes, TinyOS has become accepted by the wireless sensor network community as a research standard for implementing 802.15.4 designs [11]. Although not a "true" operating system, TinyOS has an integrated scheduler and includes nesC application programming interface (API). nesC is a C-like language developed for TinyOS that allows the programmer to "wire" components using the nesC language constructs, simplifying designs. TinyOS is completely open source, and help and fixes are sometimes quickly available via the mail list. Additionally, tutorials and technical explanations are located on the world-wide web. The work from this dissertation has allowed contributions in source code for the I^2C interface as well as improving the online tutorials.

We use TinyOS version 2.x for this dissertation, which has implemented format changes in the implementation of Active Message, the standard message construct developed in the initial TinyOS 1.x. The new Active Message fixes flaws with TOS 1.x like dropped packets due to the version 1.x scheduler implementation and allows easier programming access to the MAC and Network layers.

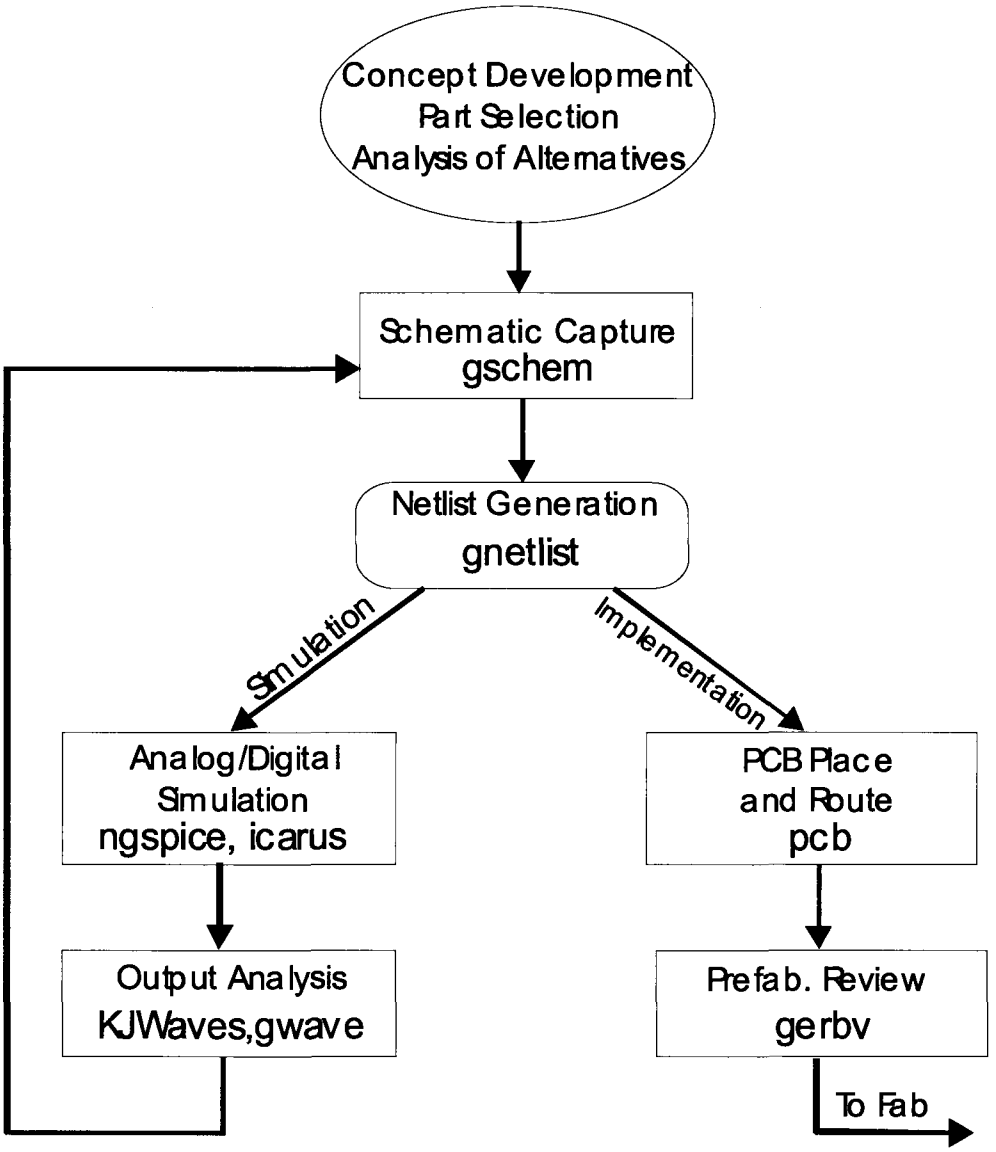


Figure 3. PCB design flow with gEDA. KJWaves, the analog SPICE waveform viewer, was created during this dissertation work and has over 1500 downloads.

TinyOS 2.x is used on a mote mated with our development board, which we call our daughter card. We wrote custom nesC code to handle message creation and to transfer the message via the I^2C interface to our board via the standard Crossbow Mote 51 pin connector. The nesC code integrates the dissemination algorithm called Trickle, which was discussed above and will be discussed more in chapter II [8,12].

RELATED WORK

Lo, et al. have published a paper describing how they developed body-sensor network hardware using mutli-sensor data fusion to monitor patients suffering from cardiovascular disease [13,14]. They developed hardware compatible with both the MicaZ and Telos motes devices based on the 802.15.4 standard. They found the standard provides both the bandwidth and latency required for appropriate medical responses. Responses are collected by a local PDA for information fusion. The PDA then transmits the data to a central server. Additionally, they used TinyOS as their operating system.

Wood, et al., examined the use of wireless devices for monitoring patients in assisted-living facilities [15]. Li, et al., have conducted significant research in the area of tactile feedback using voice coil motors [16]. Poupyrouv's *AmbientTouch* uses layers of piezoelectric to generate haptic feedback in PDAs [17]. Luk implemented an array of piezoelectric strips to stretch skin, allowing different sensations to be felt under the thumb [18]. Lee's *Haptic Pen* used a solenoid to simulate pressing down with a stylus [19]. Evreinova's dissertation provides an excellent review of visualization technology and the progression of improvements since the 1800's [20].

ORGANIZATION

This dissertation is divided into five chapters. We will review wireless sensor networks, the 802.15.4 standard, available operating systems, the broadcast-storm problem, and review of research related to solving it by implementing a dissemination protocol in Chapter II. In Chapter III, we will describe the hardware platform developed for this dissertation and interfaces to the MicaZ mote. In Chapter IV, we will

demonstrate results and experiments completed using the hardware. Finally in Chapter V, we will present conclusions and recommendations for future research.

CHAPTER II

THEORY: LOW-POWER WIRELESS COMMUNICATIONS

The Visually Impaired Notification System (VINS) was developed as a hardware platform to test dissemination of information to the visually impaired. In order to understand the system, we must review how low-power techniques are applied not only in hardware, but also in implementation of each layer in the OSI stack. As discussed in Chapter I, communications in the system rely upon the IEEE 802.15.4 standard. The operating system we used, TinyOS, allowed us to provide the application layer and adds additional features to the 802.15.4 MAC. This chapter reviews 1) the IEEE 802.15.4 standard, 2) the theory behind the "broadcast storm problem," 3) two dominant MAC protocols used in WPANs and the dissemination protocol called "Trickle" which attempts to solve "the broadcast storm problem", and 4) operating system alternatives available for WPANs.

OVERVIEW OF IEEE 802.15.4

IEEE 802.15.4 defines the MAC and PHY for low-rate WPANS. 802.15.4 was initially approved in 2003 and later was updated in 2006 for a number of reasons, including affording better spectrum usage, making clarifications, and simplifying implementation making the standard more applicable to the market. The key to this standard is low-bandwidth and low latency.

The PHY layer is discussed first below, since few modifications can be made to the physical layer. Following discussion of the PHY layer, an overview of the 802.15.4 MAC layer is given. The MAC layer is flexible enough to implement a number of

experimental standards: e.g., low-power listen, S-MAC, B-MAC, and X-MAC, which will then be discussed later in this chapter. The devices in this dissertation use the B-MAC, but it is important to understand alternative and competing MAC standards.

To support low-power consumption, the standard has a link quality indication (LQI) and the ability to perform energy detection (ED). Two different devices are specified: a full-function device (FFD) and a reduced-function device (RFD). As implied by the names, a FFD can perform any function in the network (PAN coordinator, coordinator, or node), while a RFD can only communicate with a FFD and act as a leaf or end node. As of the 2006 release of the standard, all communications take place in the industrial, scientific, and medical (ISM) bands 868 MHz, 915 MHz, and 2.4 GHz [4]. Sixteen channels are available in the 2.4 GHz band, 30 channels in the 915 MHz band, and 3 channels in the 868 MHz band. Data rates of 250, 100, 40, and 20 kbps are supported. Finally, two addressing modes: 16-bit short and 64-bit IEEE addressing are supported.

It should be noted that since the specification works in the ISM band, devices can receive interference from and give interference to other devices in the band. An example of this is shown in Figure 4, where the 2.4 GHz band 802.15.4 channels are shown as overlapping with channels specified in IEEE 802.11 (WiFi). For this reason, 802.15.4 includes an acknowledgement mechanism to ensure message transfers are reliable.

PHY LAYER OVERVIEW

The 802.15.4 PHY provides two services: the PHY data service and the PHY management service. The PHY data service supports the transmission and reception of PHY protocol data units (PPDUs) across the wireless channel. "The features of the PHY are activation and deactivation of the radio transceiver, ED, LQI, channel selection, clear

channel assessment (CCA), and transmitting as well as receiving packets across the physical medium" [4].

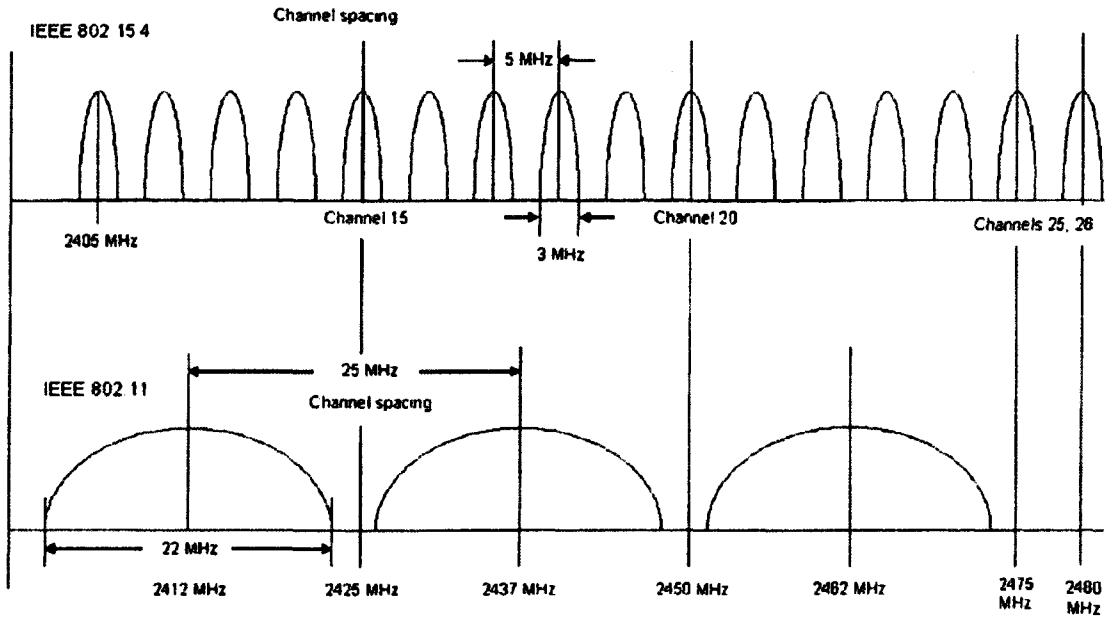


Figure 4. Overlapping spectrum usage in the 2.4 GHz band, showing 802.15.4 and 802.11 (WiFi) coexistance [21].

The radio specification supports the following waveforms: binary phase-shift keying (BPSK), amplitude shift keying (ASK), and offset quadrature phase-shift keying (O-QPSK). We will only discuss the PHY in the 2400-2483.5 MHz band since this is the band our device works in. This band solely uses O-QPSK. The O-QPSK waveform used in 802.15.4 in the 2.4 GHz band uses 4-bit symbols for a total of 16 symbols. These 16 symbols are represented by 32-chips as shown in Figure 5. These nearly-orthogonal chips are transferred at a rate of 2 Mchip/s and are broken into the in-phase and quadrature phases of the waveform, 16 chips into each. All 32 chips (4 bits) are transferred in 16 μs yielding a bit rate of 250 kbps.

Data symbol (decimal)	Data symbol (binary) (b_0 b_1 b_2 b_3)	Chip values (c_0 c_1 ... c_{30} c_{31})
0	0000	11011001110000110101001000101110
1	1000	11101101100111000011010100100010
2	0100	00101110110110011100001101010010
3	1100	00100001011101101100111000011010 1
4	0010	01010010000101110110110011100001 1
5	1010	00110101001000101110110110011100
6	0110	11000011010100100010111011011001
7	1110	10011100001101010010001011101101
8	0001	10001100100101100000011101111011
9	1001	10111000110010010110000001110111
10	0101	01111011100011001001011000000111
11	1101	01110111101110001100100101100000
12	0011	00000111011110111000110010010110
13	1011	01100000011101111011100011001001
14	0111	10010110000001110111101110001100
15	1111	11001001011000000111011110111000

Figure 5. Symbol to chip values from the 802.15.4 standard for O-QPSK at 2.5 GHz [4].

In order to understand the true aggregate data rate, we need to know how the data is packetized and framed, that is, how much of each packet is data versus overhead. We will first review the PHY packetization, and in the MAC section, how the actual data is framed. Each MAC frame is encapsulated in a PHY packet called the PHY protocol data unit (PPDU). The format of a PPDU is shown in Figure 6. The PPDU consists of a preamble of four bytes (or octets) to allow receivers to synchronize to the incoming message. The bits in the preamble field are binary zeros (the 32-chip sequence representing those zeros is actually a pattern of ones and zeros). The start-of-frame

delimiter (SFD) is hexadecimal A7. The frame length indicates the number of bytes in the PHY service data unit (PSDU) and is only seven bits long, indicating that the maximum MAC frame size can be 127 bytes. For an acknowledgement packet, the PSDU length will be five bytes, and for data and command packets, the PSDU length can be from nine to 127 bytes. The PSDU is where the MAC frame resides as shown in Figure 7.

Octets (bytes)				
4	1	1		VARIABLE
Preamble (128 μs)	Start-of-Frame Delimiter (SFD)	Frame Length (7 bits)	Reserved (1 bit)	PHY service data unit (PSDU)
Synchronization header (SHR)		PHY Header (PHR)		PHY Payload

Figure 6. PHY protocol data unit (PPDU) for the 2400 MHz band [4].

Transmitters must be capable of transmitting at least -3 dBm, or about 500 μW. Of course, to keep power consumption as low as possible, transmit power is adjustable. Additionally, receivers must be able to stand a signal as high as -20 dBm, or 10 μW. The ED measurement is a power average taken over eight symbol periods and is reported in a one-byte value. The LQI measurement is an estimate of the signal-to-noise ratio taken for each received packet. It is also a one-byte value, highest being the best quality, lowest is the worst quality. The radio reports this for each packet as received signal strength indication (RSSI) which is a term used often in the literature.

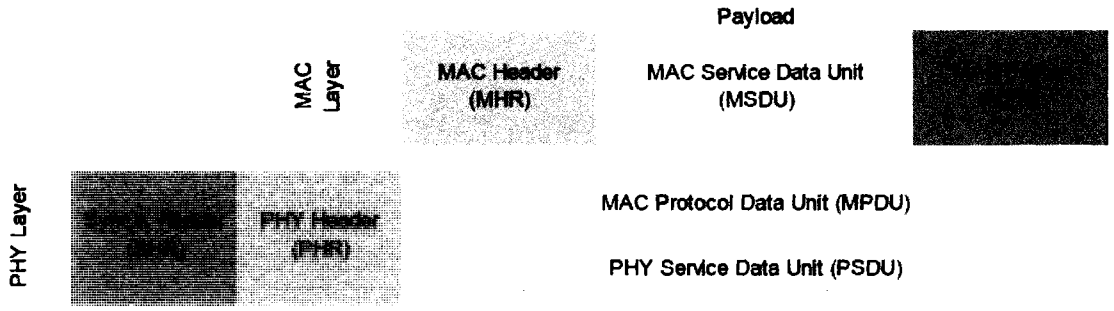


Figure 7. Diagram demonstrating the relationship of the MAC frame to the PHY packet [22].

Three methods are specified for performing a CCA:

1) CCA Mode 1: The CCA reports a busy medium if energy in the channel is above the ED threshold, which is, at most 10 dB above the receiver sensitivity (which, in turn, must be less than -85 dBm).

2) CCA Mode 2: The CCA reports a busy medium if a signal that has the same modulation and spreading characteristics is detected.

3) CCA Mode 3: Any combination of Mode 2 and/or Mode 1 above.

As with ED, CCA detection is over eight symbol periods.

MAC LAYER OVERVIEW

The MAC layer uses carrier sense multiple access with collision avoidance (CSMA-CA) in order to detect traffic on the radio channel and determine when it's appropriate to transmit. To accomplish this, the MAC layer uses either the ED or CCA of the PHY layer. Typically CCA is used to determine if a channel is busy and whether the device needs to wait a random amount of backoff units to attempt to transmit again. The basic unit for backing off is the unit *backoff period* and is 20 symbols (10 bytes). ED is

typically used by a PAN coordinator to choose a relatively clear channel to operate. Additionally, time slots can be allotted – called guaranteed time slots (GTSs) – to accommodate devices that the designer/developer does not wish to compete for bandwidth. The general MAC frame format is shown in Figure 8. The MAC footer (MFR) contains a frame check sequence (FCS), which is a 16 bit cyclical redundancy check (CRC).

Octets (bytes)								
2	1	0/2	0/2/8	0/2	0/2/8	0-14	VARIABLE	2
Frame Control	Seq. Num.	Dest. PAN ID	Dest. Addr.	Src. PAN ID	Src. Addr.	Aux. Secu. HDR	Payload	FCS
MAC Header (MHR)							MAC Payload	MFR

Figure 8. MAC protocol frame [4].

It can be seen that the minimum number of octets added by the MAC to the PSDU is then nine bytes. This assumes the 16-bit source and destination addressing mode will be used with no PAN coordination. Assuming the minimum overhead for the MAC Frame, the PHY Packet overhead above, and maximum capacity for the PSDU (127) are used, for every 118 (127-9) bytes transferred, 15 bytes of overhead are sent. This makes the maximum transmission rate 221.8 kbps if the system could continually transmit packets with no dead time in between. Of course, the assumption of no dead time and continuous transmissions is not practical. Among other things, it does not account for acknowledgements (transmissions to confirm the packet was received), which are five bytes in length, plus the extra six bytes for the PHY packet. Additionally, the MAC

specifies the minimum interframe spacing (IFS) as 12 symbols and a minimum contention access period (CAP) length of 440 symbols (at four bits per symbol, this is 220 bytes) if a PAN coordinator with super frame is used. If we assume one packet is sent for each CAP, then the maximum aggregate data rate is more on the order of 134 kbps. The underlying point of this discussion is that if a user has a large amount of data to transfer (greater than 118 bytes), then the specified bit rate of 250 kbps is not valid.

As implied by the Auxiliary Security Header field in Figure 8, the MAC has the ability to encrypt transmissions using the Advanced Encryption Standard (AES) as its core cryptographic algorithm. The encryption protection specified can protect the confidentiality, integrity, and authenticity of MAC frames. The upper OSI layers set up keys to use and security level while the MAC layer does the security processing. The MAC layer will use the address of the message to retrieve the key associated with that address. It will then use the key to process the frame according to the security suite designated for the key being used. As can be seen, significant additional overhead is encountered with encryption enabled (up to 14 additional bytes). Further discussion of this is not in the scope of this research since we wish to disseminate messages that any user can read.

LOW-POWER MAC IMPLEMENTATIONS

The goal of this research is to transmit messages across the PAN indicating essential information about the environment to a visually impaired user. As mentioned in Chapter I, perhaps the simplest broadcast approach is to have each node rebroadcast the message. This approach consumes too much communication bandwidth, causing a broadcast storm problem [7]. In this section, we will discuss the broadcast storm problem and outline

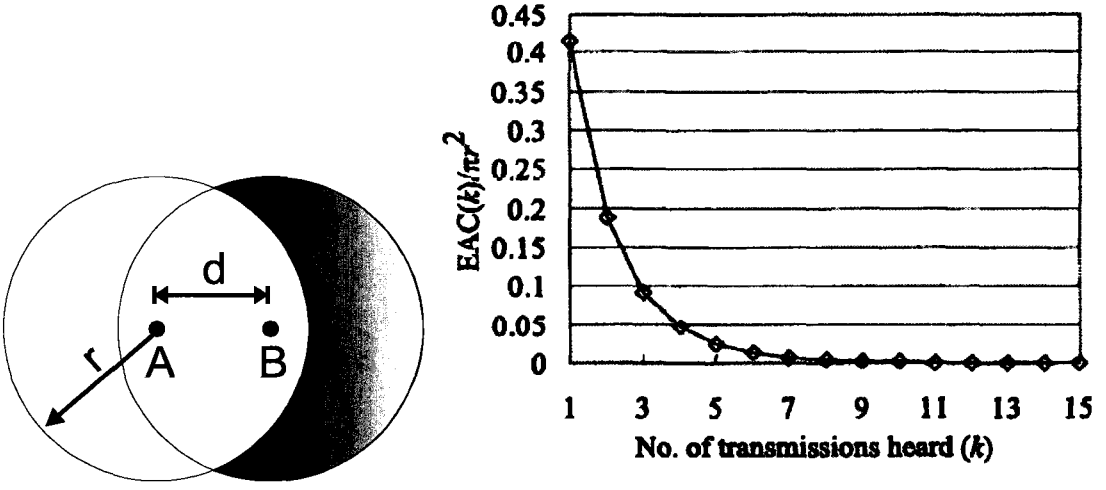
three MACs which have been implemented as an alternate to the 802.15.4 MAC to overcome the problem. Finally, we will examine the Trickle protocol which was available in TinyOS and implemented in this dissertation to transmit messages.

BROADCAST STORM PROBLEM

Ni, et al., [7] were one of the first to quantify the broadcast storm problem in depth. Clearly, in any wireless network, optimal placement of devices ensuring 100% delivery, but no coverage overlap, is extremely difficult. Additionally, some redundancy is desired in wireless sensor networks (WSNs) because, should a device stop functioning due to lock-up, battery being exhausted, or any other reason, the user would want the message to be rerouted through a working node. In our circumstance and for mobile ad hoc networks (MANETs), it's impossible to determine the layout a priori; therefore, the network, itself has to determine the most appropriate nodes to use to minimize retransmissions, but also do its "best effort" to ensure 100% of the working nodes have received the message. One of the key aspects of Ni's paper is that the broadcast is considered "unreliable", in that no acknowledgements are used. The network will make a best effort to ensure all nodes receive a message, but 100% reliability is unnecessary. This is the case for our system as well. Should a base station desire its message to be received by a greater number of recipients than "best effort", a priority field can be used in message delivery and robustness will occur at the expense of battery power.

Ni, et al. [7], start by examining the amount of redundancy that occurs by having all nodes in a defined area transmit. They model the transmit areas of each node as a circle with constant radius, r, as seen in Figure 9(a). As each new node is added, the additional area covered for the new node can be calculated. The average gain for the second

transmission is only about 41%. In a realistic scenario, a node might listen for how many times the same message was transmitted (the message would be tagged with a unique identification number) and that would determine if it should transmit. Ni, et al., simulated this scenario and determined that if a node heard the same message greater than or equal to four times, the "expected additional coverage" (EAC) for that node transmitting was less than 0.05%, as seen in Figure 9(b). This interesting outcome shows a practical limit on how a node should decide to retransmit or not.



(a) (b)

Figure 9. From Ni, et al. [7], (a) diagram showing the additional area gained by node B rebroadcasting a message from node A. (b) graph of the expected additional coverage after hearing *k* transmissions.

The second aspect of the broadcast storm problem examined in the paper is the contention problem. This occurs when multiple users try to use the same spectrum at the same time. Since the MAC performs a CCA before transmitting and then, if the channel is busy, does a random back-off, it can be some time before a message is delivered if

many nodes are trying to transmit the same message. Examining Figure 9(a) more closely; if another node, C, hears a message from A, and is going to contend with B, it must be in the intersecting area. From this, they computed the likelihood of contention for just two nodes as 59%. When the number of nodes becomes greater than five, the likelihood of contention becomes greater than 80%. It's interesting to note that they use only area to examine the likelihood of contention and not timing. For instance, if node C has already attempted to transmit another message and had to back-off, when it's necessary to transmit the repeated message, it might attempt it at a different time slot than B. This is a large oversight in their analysis and is perhaps why contention is not mitigated in their list of potential solutions.

They suggest four schemes for reducing redundancy, which simultaneously reduces contention: 1) a counter-based scheme, 2) a distance-based scheme, 3) a location-based scheme, and 4) a cluster-based scheme. Of these, the counter-based scheme is perhaps the most important because it is essentially implemented in the Trickle protocol as part of TinyOS implemented in this dissertation. Additionally, this scheme can be adjusted using a priority field (as mentioned above), which would increase the counter threshold, thus causing nodes to transmit more. (Note: this aspect of the counter-based scheme is not mentioned in their paper.) The location and cluster-based scheme require either external, power-hungry components or are too complicated for the computational power available in the low-power devices typically used in LR-WPANs. Additionally, the counter-based scheme performed almost as well as the location-based scheme and better than the cluster-based scheme in terms of reaching more nodes, especially in sparse clusters. Therefore, we will only review the counter and distance-based schemes.

The counter-based scheme performed better than the distance-based scheme in terms of saved rebroadcasts and lower latency [7]. To implement the counter-based scheme, a node would follow the following procedure.

S1: Message heard. Initialize counter. Set timer for random wait time. Go to S2.

S2: Wait; if time is up, go to S3. If message is heard again, increment counter. If counter is < C, go to S2, else go to S4.

S3: Transmit message.

S4: Do not retransmit message.

The distance-based scheme uses the distance between the receiver and any message re-transmitter (even the first host) as the metric to decide whether to transmit again. If the distance is small, the additional coverage gained by retransmission is small. The user chooses a minimum distance d_{min} to decide whether to retransmit. They assume signal strength is a reasonable indicator of distance, which is a valid assumption. To implement the distance-based scheme, a node would follow the following procedure.

S1: Message heard. Store distance measurement, d_{store}. If $d_{store} < d_{min}$, go to S4. Set timer for random wait time. Go to S2.

S2: Wait; if time is up, go to S3. If message is heard again, go to S1.

S3: Transmit message.

S4: Do not retransmit message.

They conclude the paper with some suggestions for the counter threshold, C, and minimum distance, d_{min}. For sparse networks, where the number of neighbors average about 2.4, all nodes that can be reached are never reached even for high counter thresholds i.e., six. For medium and high-density networks, reachability approaches

100% for C>3. The values for minimum distance were chosen to make a comparison between the two schemes by using their estimate for additional coverage for both methods. Even so, the counter-based scheme saves more in terms of rebroadcasts although the distance-based scheme did better in terms of reachability, especially in areas with sparse sensor populations.

LOW-POWER LISTEN

Hill and Culler [23] developed the concept of using a "low power listen" (LPL) mode to dramatically reduce power consumption in LR-WPANs. Their concept is simple but requires synchronizing nodes on the network. Using their time-synchronization method of tagging the sent data with a time stamp, and then comparing that to the time stamp at the receiver, they claim to achieve synchronization between two nodes of 2 μs. Initially, they suggest that a node would be scheduled to wake up at certain times, turn on its radio to listen, and listen for two packet lengths. This would allow the node to receive a packet scheduled for some time between the start and end of the two-packet lengths. This can be expensive in terms of power consumption with a duty cycle of 2.7%, especially if no message is present most of the time.

To alleviate this problem, they suggest that the transmitter may transmit its carrier (or preamble), only during a short time period if it has a message to send. It then can start the message after this short time. If the devices are synchronized to 2 μs, then the wake-up tone could only be 4 μs long. The receiver would use the ED feature to sample the channel for a much shorter period than above to determine whether the node should continue listening or "the wake up signal is present" [23]. If a four-second interval is used, they perform the ED in 50 μs yielding a very low duty cycle of 0.00125 [23]. This

method does not seem like it would be effective in a noisy environment but would be extremely effective in an outdoor environment with little electronic in-band noise. The energy-usage burden is transferred from the receiver to the sender which has to increase its preamble time to ensure the receiver wakes up in time to listen.

BERKELEY MAC (B-MAC)

Largely based on the work of the two papers above, Polastre, et al., developed the Berkeley or B-MAC [24]. Since idle listening consumes about the same power as transmitting, the amount of time nodes spend in listening mode can be costly in terms of network lifetime. Their MAC differs from other MACs in that, instead of supporting a general set of workloads and large network traffic, the B-MAC is designed to be effective specifically for small, low-duty cycle applications. They found that typically, wireless nodes have a duty cycle of about 1%. The B-MAC can be reconfigured by the upper-level application layer (OSI layer 7) to compensate for changing network conditions. This can be accomplished, real-time, to optimize for parameters like energy consumption to prolong network lifetime.

From reviewing previous work, Polastre claimed that time division multiple access (TDMA) and ALOHA (A University of Hawaii MAC protocol where subscribers transmit when they need to, and if the message does not make it through, they transmit later) could not scale to large networks. Since then, modifications to the ALOHA protocol include slotted ALOHA and ALOHA with preamble sampling. Polastre also included WiseMAC, which met all of B-MAC developer goals, except having the ability to be reconfigured based on changing demands of the network [24]. Polastre particularly compares B-MAC with S-MAC, which is a low-power request to send/clear to send

(RTS/CTS) protocol developed with 802.11 (WiFi) in mind. Each active period, when the radio wakes from the sleep mode to synchronize with neighbors and transmit messages, is 115 ms wide, while the sleep period is random. The protocol includes an adaptive sleep period where a neighbor "snoops" on other neighbor's packets and immediately transmits it own message(s) after their transmission is complete (using RTS/CTS). As the size of the network grows, the S-MAC must maintain an increasing number of each neighbor's schedule. This makes S-MAC less practical for large sensor networks considering the notorious lack of memory available in sensor devices. In S-MAC we can see though that listening to neighbor's communications (snooping/ eavesdropping/gossiping) can have benefits in a low-power network.

We can see from some of the above discussion, where B-MAC's key features of a. random delays (referred to as unit back-off from 802.15.4), b. forwarding delay (counter-based scheme from broadcast storm paper), c. link-quality estimation (RSSI from 802.15.4), and d. low-duty cycle through long preamble were developed. It should be noted that B-MAC avoids synchronization by lengthening the preamble and making the wake-up intervals (called "check time" by the authors) such that the radio always wakes up in time to hear a preamble, as shown in Figure 10. B-MAC also implements an improved CCA where it samples the channel at a time when it is assumed to be free, applying an exponentially-weighted moving average to samples. This allows them to create a better estimate of the noise floor and implement an outlier algorithm to more efficiently determine if a channel is free.

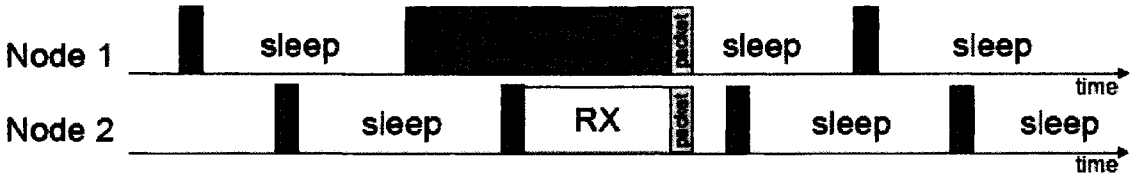


Figure 10. Illustration of B-MAC, showing key feature of lengthening the preamble to ensure the potential receive node wakes up in time to detect a pending message [24].

When initially attempting to send a message, if the channel is occupied, instead of using the 802.15.4 standard random back-off, B-MAC notifies the service (higher in the OSI stack) sending the package. If the notification is ignored, then the standard back-off is used. This gives the upper layer the ability to modify the standard back-off. After the initial back-off, the CCA outliers algorithm is used and a "congestion back-off" notification message is sent to the service. Again, the upper layer service can ignore the message or implement changes to adjust the fairness or available throughput of the network.

Using the LPL technique, B-MAC periodically samples the channel. If activity is detected, the radio stays on and receives the packet. After reception or a false positive, the node goes back to sleep. Clearly LPL requires synchronization and/or the ability to check the channel during the time of the PHY preamble. B-MAC chooses the latter, adjusting the preamble size and "check time" (time between sampling the channel for a message) based on density of the surrounding nodes. The length of the preamble and time between checks is adjustable but typically ranges between 20-100 ms (which is significantly longer than the 802.15.4 preamble specification in Figure 6), and the author provides an algorithm to compute the check time, and thus, the preamble length based on

usage cases. Nodes do snoop on other messages in an attempt to determine how best to synchronize with the parent node and determine network layout, but this is not formally included in the B-MAC. This can allow dynamic reduction in the preamble size during the lifetime of the network.

The authors provide two other insights and uses for the B-MAC: a. disabling the CCA to allow implementation of a scheduling protocol at a layer above B-MAC and b. cycling between long and short preambles in order to limit energy consumption. Polastre makes a very compelling case on the utility of B-MAC for sensor networks and clearly demonstrates the MAC's superiority over S-MAC and ALOHA. Although B-MAC is the default MAC used in TinyOS and used in this dissertation, continued development has occurred on low-power MACs. We describe one of the most promising below, X-MAC.

X-MAC

X-MAC was developed by the University of Colorado about two years after B-MAC in their technical report CU-CS-1008-06 [25]. The authors considered the B-MAC inferior because the long preamble "introduces excess latency at each hop, is suboptimal in terms of energy consumption, and suffers from excess energy consumption at non-target receivers" (an artifact from packet snooping) [25]. A large portion of energy wasted when using the B-MAC occurs because the receiver has to wait until the end of a long preamble to determine if the message is meant for it. If not snooping, this wastes significant amount of energy by those nodes that are not the intended recipients. For example, the worst case would occur if a receiver detects the preamble just as it is beginning. It must wait the full preamble time to eventually listen to the message. If it is

a multi-hop message, the per-hop latency is increased and bounded by the preamble length [25].

X-MAC goes back to a short preamble in an attempt to alleviate these problems. First, Buettner, et al., embed address information of the target into the preamble so that others that do not wish to snoop can go back to sleep (remember that the 802.15.4 preamble specification is for the preamble to consist symbols that are all 0's – repeating chips of 11011001110000110101001000101110). Second, they strobe the preamble, which allows the target recipient to notify the sender that it is awake (similar to a CTS signal). Third, similar to B-MAC, they suggest an automated algorithm for adapting the duty cycle of nodes to accommodate traffic density. It should be noted that the Trickle protocol, description to follow, significantly relies on listening to others' transmissions, making their first improvement of limited utility (since all nodes would want to snoop). One of the key aspects of the X-MAC is that it can be supported across a broad range of currently-available radios, without becoming non-compliant with the 802.15.4 PHY specification [25]. Although Buettner, et al., did not compare X-MAC directly with B-MAC, their method showed a significant improvement over the B-MAC implementation of LPL.

TRICKLE PROTOCOL

Now that we have reviewed some of the various MAC layer protocols that are used in LR-WPANs, we have to examine the means to achieve message dissemination most effectively. Levis, et al., [8,26] have developed the "Trickle" protocol based on the ideas in Ni's paper [7]. This scalable approach can consistently deliver messages or code

updates via a broadcast mechanism efficiently by using a "polite gossip" policy. Trickle can also be used for route maintenance and neighbor discovery.

The purpose of Trickle is to ensure all nodes have the same version of software or message (or "state") and resolve inconsistencies through local interactions with their neighbors. A key aspect of the protocol is that new nodes do not receive the entire history of updates, but, instead only the latest one. Similarly, if a node drops out for some reason, it will miss intermediate messages/updates, and will only receive the latest one. Key aspects to evaluate dissemination protocols are 1) how quickly they disseminate messages (and how long it is until the last node receives an update), 2) how reliably they are sent, 3) how much energy is consumed, and 4) how much bandwidth is consumed (reduce redundant transmissions and number of messages). Additionally, De Couto, et al., have shown that an effective metric to use to determine cost is "expected transmissions" or ETX [26], and, indeed, this is the metric Levis uses. The ETX of a link is the predicted number of transmissions required to successfully receive that packet at the destination. The ETX of a route is defined as the total of all ETXs for each link [27].

In order to determine if a node has the same state, a node listens to transmissions from its neighbors (mostly repeats of the broadcast message). If the "state" in the neighbor's message doesn't match the node's current state, the node will communicate to resolve the problem. If the states are the same between nodes, transmissions slow to a "trickle", sending only a few packets per hour. Since the protocol only relies on local broadcasts, instead of building large routing tables, it is robust enough to handle addition and loss of nodes. The general algorithm in an unsynchronized network for Trickle is below; notice the similarity to the counter-based algorithm from Ni, above.

S1: Initialize counter, c, to 0. Set timer for a random time in the range [$\tau/2$, τ]. Go to S2.

S2: Wait; if time is up, go to S3. If message is heard, either:

a. increment counter *if message is consistent with current state* and go to S2 or,

b. go to S4 *if message with a different state is heard.*

S3: If c < k, transmit message which contains metadata to detect inconsistent states. Double τ, but not greater than τ_h. Go to S1.

S4: Choose $\tau = \tau_l$. Note new state. Go to S1.

Note that k is the "redundancy constant" and was discussed in Ni's counter-based scheme. As can be seen, Trickle dynamically adjusts τ, the gossiping interval, based on network conditions. If a new message is heard, it sets τ to the lower limit τ_l, making a transmission, or gossip, more likely. If the gossiping interval is small, more communications have the potential to occur, and data propagates more quickly. Additionally, this means higher energy consumption in the network. Levis suggests both an upper and lower bound for τ, τ_h and τ_l, respectively. Notice that in S2b, the node will go to the lower bound if it hears a node with either an older or newer state. In the case of hearing an older state, the node will quickly respond with the new code. In the case of hearing a newer state, the node will ensure other nodes within its range also get the new message, having message transmission limited instead by the counter-based scheme.

We can see how this protocol is most compatible with the B-MAC described above due to the necessity of snooping on neighbor's messages. The X-MAC protocol could be modified to include the node's state (usually 4-11 bytes [8]) along with X-MAC's

broadcast address in each preamble packet. This should accomplish the same advantages of X-MAC over B-MAC with the corresponding energy savings.

Levis notes that a node observes $O\left(k \cdot \log_{1/PLR}(d)\right)$ transmissions over an interval for a synchronized network (where timers start at the same time), where d is the network density and PLR is the packet loss rate [8]. For an unsynchronized network, the number of transmissions scale as $2k \cdot O\left(\log_{1/PLR}(d)\right)$ [27]. This is because, for an unsynchronized network, a listening period of $\tau/2$ bounds the total transmissions in a lossless, single-hop network to $2k$ because it is assumed that a single transmission during a listen period prevents other nodes from transmitting in that time. Because of its utility and simplicity, Trickle has become a "network primitive" in LR-WPANs and is implemented in many projects [8].

TINYOS AND OTHER LR-WPAN OPERATING SYSTEMS

OVERVIEW OF TINYOS

TinyOS, developed by UC Berkely, has become the OS of choice for conducting research in the area of LR-WPANS and wireless sensor networks [11]. Despite that, numerous other operating systems and virtual machines have been developed. We will only briefly touch on some of these other OS's at the end of this section since TinyOS is used in this dissertation.

TinyOS is a scheduler for two predominant microcontrollers used in wireless sensor networks, the Atmel Atmega 128L and the Texas Instruments MSP430. A port of the code is in the works for the venerable 8051 microcontroller as well. It allows a programmer to manage concurrent data flows among hardware devices and provides

modularized software components which represent hardware components or messaging constructs. Using a simple run-to-completion, first-in-first-out concurrency model for time-consuming tasks, TinyOS capitalizes on the underlying capabilities of the microcontroller it resides on. Tasks are created which are associated with an event and placed in the scheduler. When the task is complete, the software-defined event is triggered by an interrupt from a timer, a peripheral, or an external device. This event-based model relies on the robust hardware interrupt structure that microcontrollers provide without blocking or polling while TinyOS ensures that the results of an interrupt are assigned to the correct event for response. It allows the CPU to dispatch jobs to peripherals, e.g., the direct memory access (DMA) controller or the inter-integrated circuit (I^2C) controller, and then go into a deep-sleep state to conserve energy. The CPU is then awoken by an interrupt when the peripheral is complete.

Additionally, the developers have provided a bit-level simulation for TinyOS called TinyOS Simulator (TOSSIM). Code created for TinyOS can be run in TOSSIM with minimal modifications, if code is sufficiently simple and does not include any non-supported external components. To run the code for this dissertation in TOSSIM, the I^2C components had to be disabled since those components were not supported in TOSSIM. I^2C module calls were replaced with debug statements to a file in the simulator. The University of California at Los Angeles has another compiler for the Mica mote called Avrora. It was not used or evaluated in this dissertation.

TinyOS is programmed using nesC (Networked Embedded System C) which allows connecting of built-in and user-defined "components". The user defines, in code, the interface between components and can instantiate components of their own. To increase

runtime efficiency, components are statically wired together to form the kernel based on the interfaces specified between components. Two types of components are used in nesC, and they may consist of interfaces, commands, and events:

- Modules which implement the application behavior (usually will contain the bulk of the user's code) and
- Configurations which wire components together through interfaces.

This methodology creates a hierarchy where commands flow downwards and events flow upwards. Since the OS is event-driven, tasks are created and "posted" to the scheduler with the keyword "post" which the scheduler executes and then waits for events. These events call commands but not vice-versa. Events, of course, can be ignored. For instance, when sending a message, upon message transfer to the transmitting radio, the interrupt response is a call to "*SendDone()*" which the user may ignore.

IMPLEMENTATION OF NOTIFICATION SYSTEM

We used TinyOS on a Crossbow MicaZ Mote as the radio component with a custom-designed daughter card for our notification system for the visually impaired. To further mitigate the broadcast storm problem and to localize messages, we used the Dissemination protocol included with TinyOS with a modification to limit the distance, by limiting the number of hops, a message is allowed to propagate. Messages are then forwarded to the daughter card for the human interface which is discussed in the next chapter.

A snippet of code from the nesC application created for this dissertation is shown below:

```
configuration EasyDisseminationAppC {}
implementation {
  components EasyDisseminationC as App;
  components DisseminationC;
  components MainC;
  App.Boot-> MainC.Boot;
  App.DisseminationControl -> DisseminationC;

  components ActiveMessageC;
  App.RadioControl -> ActiveMessageC.SplitControl;

  components new DisseminatorC(I2C_Dism_packet_t, 0x1234) as Diss16C;
  App.Value  -> Diss16C;
  App.Update -> Diss16C;

  components LedsC;
  App.Leds-> LedsC;

  components  new TimerMilliC() as Timer1;
  components  new TimerMilliC() as Timer2;
  App.Timer1-> Timer1;
  App.Timer2-> Timer2;

  components  new Atm128I2CMasterC() as I2CMaster ;
  App.I2CBasicAddr-> I2CMaster.I2CPacket;
  App.I2CResource -> I2CMaster.Resource;
...
```

The convention in nesC is to append a "C" after configuration files and an "M" after modules. The "*App*" in the file name indicates the highest level in the hierarchy of an application. It should, appropriately, also be a *configuration* wiring together underlying components. In the code snippet above, we can see a few things:

1) In the first set of component declarations, the programmer can choose to use the actual name of the component or redefine the name of the component for his particular application using the keyword "*as*" followed by the new name. So the application code we wrote to implement our messaging system is in the file *EasyDisseminationC*, but it is referred to as *App* in the nesC code due to the assignment in line 3. This is also useful if the programmer wishes to instantiate a number of different components (such as Timers).

2) The programmer can pass parameters, such as "*I2C_Dism_packet_t, 0x1234*" to the *DisseminatorC* component above which allows modifying a component before it is compiled. This allows components to be flexible, but one has to remember that more modifications cannot be made at runtime.

3) After the component declarations, the new name for the component is wired to application usage of the components using arrows: "->" or equals "=".

All applications have a "*MainC*" which handles initializing the application. In the above example, this component is "wired" to the application's "*boot*" component, meaning that "*MainC*" will be called when the microcontroller boots. In the application there will be an event "*void Boot.booted()*" which contains the actual C code that the application executes at boot time. The application's "*booted()*" section is below:

```
event void Boot.booted() {
    int ii;
    for (ii=0;ii<4;ii++) {
        I2CMsgIn[ii].nodeid  =  TOS_NODE_ID;
        I2CMsgIn[ii].counter =  0;
        I2CMsgIn[ii].priority =  1;
        I2CMsgIn[ii].numberhops    =  0;
    }
    I2CMsgOut.nodeid    =  TOS_NODE_ID;
    I2CMsgOut.counter   =  0;
    I2CMsgOut.priority  =  1;
    I2CMsgOut.numberhops =  0;
    atomic {
        busy = FALSE;
    }
    call Leds.set(LEDS_LED0 | LEDS_LED1 | LEDS_LED2);
    call I2CResource.request();
}
```

This is an opportunity to request resources and set up initial conditions. We use the I^2C port first so we send a request at the end of initialization for its use. When the microcontroller finishes setting up the port, it will call the event: *I2CResource.granted()*.

This example demonstrates how code flows in a TinyOS application by calling commands or posting tasks and waiting for responses through events, as described above.

Notice that three components: "*TimerMilliC, DisseminatorC, Atm128I2CMasterC*" are instantiated with the "*new*" command. "*New*" creates a realization of "virtualized abstraction" or "generic component" which is a set of code that is reused by creating a new, separate module for each instantiation. The programmer can send parameters to the virtual component creating a customized static component for their application. Using the keyword "*new*" will increase the size of the application because a separate component will be created each time it is used. This is actually used in the TinyOS library *Dissemination* to instantiate two receive-radio components, one for receiving data messages and one for receiving probe messages.

The main message construct of TinyOS is the "*Active Message*" which follows a TinyOS-defined C structure called *message_t*:

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Notice that the structure closely follows the MAC Frame in Figure 8. *ActiveMessage* uses the macro "define" *TOSH_DATA_LENGTH* to determine the MAC payload length which defaults to 28 bytes. The *ActiveMessage* construct handles all underlying communications allowing the user to call "*Send*" when a message needs to be sent and receive the event *sendDone()* when transferring the message to the radio is complete. A user with a custom application can modify the *message_t* structure (as is done in the Dissemination library), but most will choose what we have done which is have our

structure fit within the "*data*" field in *message_t*. We created our own structure to limit the distance packets are sent from the base station:

```
typedef nx_struct easydis_packet_t {
 nx_uint16_t nodeid;
 nx_uint8_t counter;
 nx_uint8_t priority;
 nx_uint8_t numberhops;
 nx_uint8_t data[TOSH_DATA_LENGTH-5];
} I2C_Dism_packet_t;
```

We have added a *priority* field, which would allow the dissemination protocol to increase the dispersion time for a broadcast packet (by immediately forcing selection of τ_l) and a *numberhops* field to limit the distance from the base station by counting hops. This structure is passed to *DisseminatorC*, which in-turn uses it as the data payload for *ActiveMessages*. For further information on *Dissemination* refer to the documentation at http://www.tinyos.net. Simulation and implementation results will be presented in Chapter IV.

OTHER OPERATING SYSTEMS

TinyOS has limitations that have been addressed by a number of different add-ons, communities, and universities. For instance, after linking, modifying the system is not possible, which caused Culler, et al., to create Maté, a virtual machine for TinyOS that allows new code segments to be created and downloaded to change the system during runtime. In contrast, Contiki, developed by the Swedish Institute of Computer Science, is a multi-tasking operating system that provides a dynamic structure which allows programs and drivers to be replaced after deployment without relinking [28]. Additionally, the University of Colorado developed MANTIS, a multithread embedded operating system which enables flexible and fast deployment of applications. With the

key goal of ease for programmers, MANTIS uses classical layered multithreaded structure and allows programmers to use C. Unlike TinyOS's FIFO structure, MANTIS has a prioritized scheduler allowing tasks to be arranged based on importance [29].

Many other operating systems and schedulers are available, such as MagnetOS which actually uses a Java virtual machine that includes static and dynamic components. The static components rewrite the application in byte-code level and add instructions to the original application. The dynamic components allow changes to be delivered as the system is running. This small sampling of operating systems hopefully demonstrates the growing research field of wireless netted sensors and WPANS. Small incremental improvements are continually occurring including the next release of TinyOS which will include safe memory execution and concurrent threads [30].

CHAPTER III

HARDWARE DESIGN

This chapter addresses the hardware designed for the Visually Impaired Notification System (VINS). The following features we envisioned for the system:

1) interface directly with commercially available Crossbow MicaZ Mote working in the 2.4 GHz band,
2) haptic feedback to the potential user in the form of 3x2 Braille,
3) visual feedback of Braille output,
4) text-to-speech (TTS) capability to allow visually impaired user to hear messages being sent, and
5) consume a minimal amount of additional power over the MicaZ.

We first review the Crossbow MicaZ Mote and hardware interface capabilities and then discuss our hardware implementation to implement the features above.

CROSSBOW MICAZ

Figure 11 shows the history of wireless sensor cards from DARPA's original program inception in 1997. The Crossbow MicaZ and the Telos motes are based on the Atmel 128L and Texas Instrument's MSP430 microcontrollers, respectively, and are arguably the most prolific devices used in research today. Numerous manufactures now make radios and integrated microcontrollers for the 802.15.4 specification, some implementing MAC protocols discussed in the previous chapter. We chose the MicaZ because of the wide variety of input/output (I/O) available through its 51-pin, on-board connector.

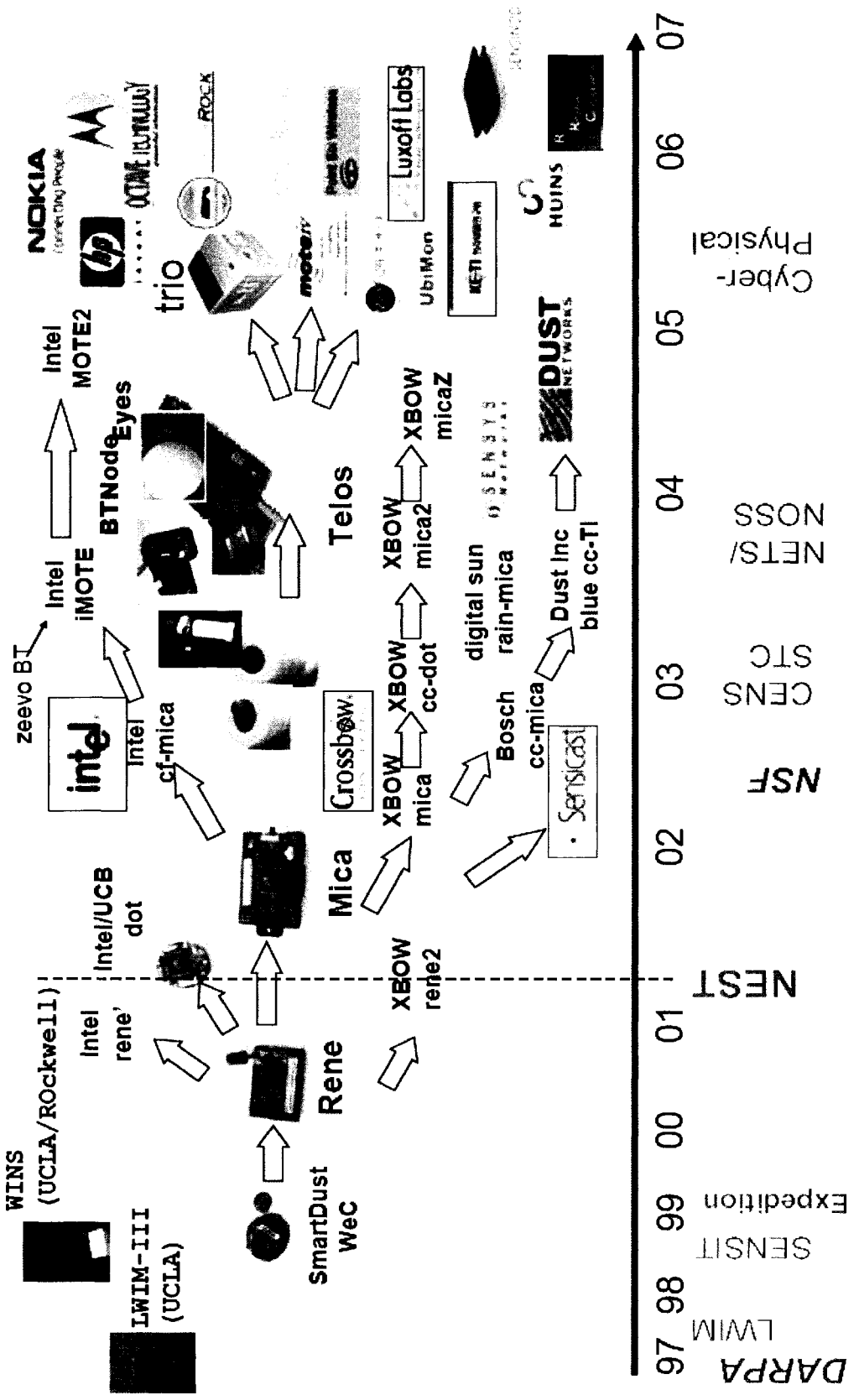


Figure 11. Short graphical history of wireless sensor networks showing Crossbow MicaZ development [31].

Once a concept is demonstrated, the entire validated design could eliminate unnecessary ports and connectors yielding a much more compact, power-saving design.

Figure 12 shows a picture of the MicaZ mote along with a cartoon showing the major components. The daughter card developed for this dissertation connects to the 51-pin expansion connector. This connector has inputs to the eight-channel, ten-bit, on-board analog-to-digital convertor (ADC) as well as 21 general purpose input/output (GPIO) pins. Additionally, the connector allows access to the microcontroller's I^2C and serial peripheral interface (SPI) ports. The SPI port is shared with the on-board radio making it necessary to ensure addressing commands are used to remove contention. Because of this, we decided to use the I^2C interface, also called "two-wire serial", to communicate with our daughter card. The I^2C interface allows up to 400kbps transfer speeds and can address up to 128 devices. Programming the I^2C bus for both the MicaZ and the daughter card will be discussed later.

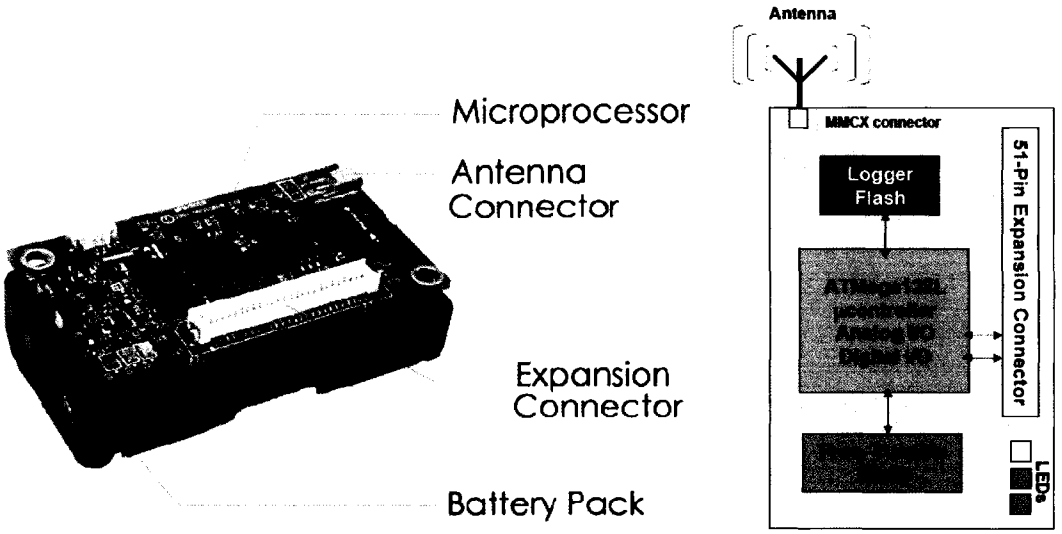


Figure 12. MicaZ description, from Crossbow documentation [32].

The provided schematic of the 51-pin connector is shown in Figure 13. Note that the figure includes both male and female connector pin-out descriptions, despite the fact that the MicaZ has no connector attached to J22. Instead, the battery holder is connected to that side of the board. The male connector is provided at the top (J21) of the MicaZ, while the user may remove the battery holder and solder their own connector on the pads provided on the side of the battery holder. We removed the battery holder in our design, and the daughter card sits where the battery holder would typically reside. Additionally, we soldered another DF9B-51P-1V, 51-pin connector to the bottom of the MicaZ. These modifications allow the daughter card to stay on the MicaZ during programming with the Stargate board described next. This allowed for easy viewing of the onboard LEDs on the Mote. Figure 14 shows the pin-out description of the DF9B-51S-1V, 51-pin connector used on our daughter card (for full schematics, see Appendix A). It should be noted that this is a mirror image of J21, and therefore, could not be connected to the MicaZ directly through connector J21. Figure 15 shows a diagram of the connection between the daughter card and MicaZ.

Additionally, we replaced the typical battery holder with a Digikey part number (P/N) SBH-321AS-ND and a connector to the 1.25 mm on-board power connector (J4), P/N W1720-ND. Due to clever power routing, the combination of daughter card and MicaZ can be powered in one of three ways:

1) From a direct-current (DC) plug connected to a wall outlet through the Stargate board,
2) From the battery pack connected to the MicaZ, or

3) From the same DC plug that powers the Stargate, connected to CONN202 on the daughter card.

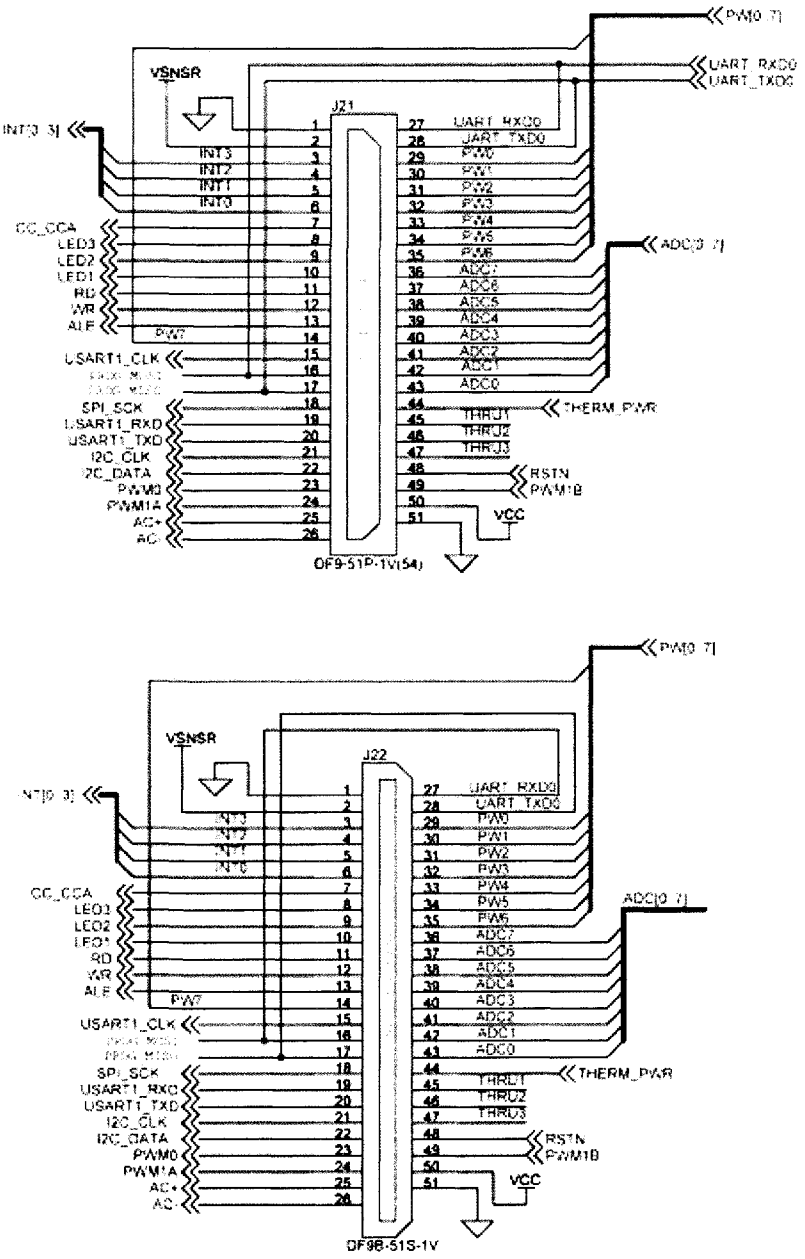


Figure 13. Schematic of the 51-pin connector taken from the MicaZ schematics provided by Crossbow [32].

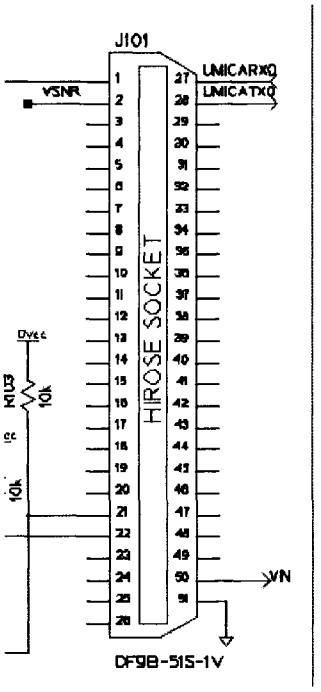


Figure 14. 51-pin connector used on daughter card.

The daughter card's power distribution and conditioning system will be discussed in more detail later.

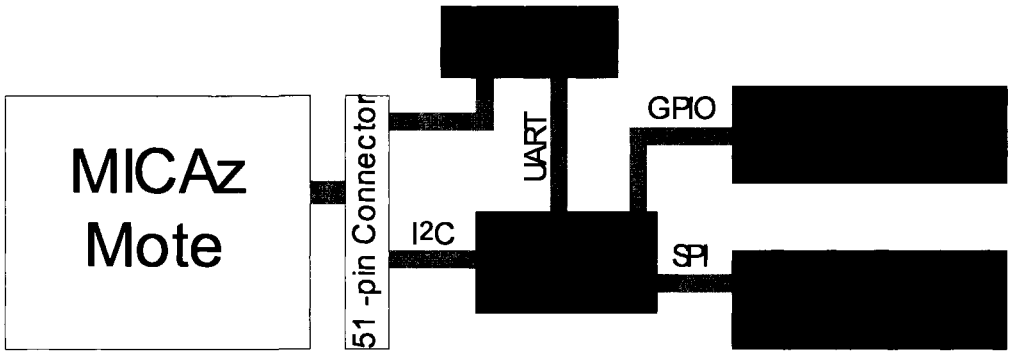


Figure 15. Diagram showing key components of the daughter card and connection to the MicaZ mote.

STARGATE

STARGATE DESCRIPTION

The Stargate board produced by Crossbow (P/N SPB400) is a Linux-based, single-board computer that has peripheral interfaces for a variety of functions. It contains a PCMCIA connector for a WiFi card and an Ethernet port, so it may connect a connected sensor network to an enterprise network without a PC or server. It has a 51-pin connector which allows it to use a Mica Mote to interface with the wireless sensor network. Additionally, it has a type II compact flash (CF) slot which enhances its memory capacity for network data storage and retrieval.

The Stargate has a 400 MHz , Intel PxA255 XScale processor based on ARM technology. In this project, the device is solely used to program the MicaZ mote through the UART port on the 51-pin connector. The procedure in the following section was used to compile the nesC code, assign the mote ID, and program the mote.

PROGRAMMING MOTES USING THE STARGATE

These instructions assume the user has already configured the Stargate according to the manual on the Stargate disk using *minicom* on your Linux host. That procedure initially sets up a static IP address and Apache (a web server). Additionally, we assume the user's nesC code has been compiled using the following command:

$ make micaz

To differentiate each Mote on the network, each should have a different identification number, the Mote ID. Therefore the user must set the Mote ID before downloading to the mote; this example sets it to one. For tinyOS 2.x we used *tos-set-symbols* instead of

set-mote-id; notice we appended the Mote ID to the end of the compiled file name for easier accounting later.

tinyOS 1.x: $ set-mote-id build/micaz/main.srec build/micaz/main.srec.out-1 1
tinyOS 2.x: $ tos-set-symbols build/micaz/main.srec build/micaz/main.srec.out-1 TOS_NODE_ID=1

The secure copy (SCP) protocol allows the user to copy the mote image (with the appropriate Mote ID applied) to the Stargate for programming the connected mote. The following line copies the compiled nesC code, *build/micaz/main.srec.out(-x)*, to the "/usr" directory on the Stargate, using the "root" user's account. The root-user's password is required to successfully connect.

$ scp build/micaz/main.srec.out root@<Stargate IP Address>:/usr

The response from the Stargate should look something like this:

user@host:/opt/tinyos-2.x/apps/BlinkToRadio/build/micaz
$ scp main.srec. root@192.168.1.91:/usr*
root@192.168.1.91's password:
main.srec.out-1 100% 26KB 26.4KB/s 00:00
main.srec.out-2 100% 26KB 26.4KB/s 00:00 ...

Once the images are copied to the Stargate, the user can use the secure shell (SSH) to connect protocol to the Stargate to write the image to the Mote. This line attempts to connect to the device using user, "root", at <IP.Address>. The root password should then be used to connect.

$ ssh -l root <IP.Address>

Stop xlisten-arm if running, since you need access to serial port for programming.

$ ps –A | grep xlisten-arm

$ kill <process id for xlisten-arm>

Change to the directory you stored the images.

$ cd /usr

Ensure the Mote is plugged into the Stargate 51-pin connector and execute:

$ uisp -dprog=sggpio -dpart=ATmega128 --wr_fuse_h=d9 --wr_fuse_e=ff –erase --upload if=main.srec.out[-x]

Once your motes are programmed, be sure to shutdown your Stargate before you turn it off by executing:

$ shutdown -h now

DAUGHTER-CARD, HARDWARE DESCRIPTION

MICROCONTROLLER

The microcontroller used on the daughter card is the Texas Instruments MSP430F1611. This 16-bit microcontroller was chosen for various low-power features such as four low-power modes (LPMs) and the ability to wake up from a low-power mode within 6 μs. The microcontroller includes I^2C, SPI, and universal synchronous/asynchronous receiver/transmitter (USART) ports and 48 pins of GPIO. Additionally, the processor can be run with supply voltages ranging from 1.8 to 3.6 Volts. In low-power mode 3 (LPM3), the processor can keep the external 32768 Hz crystal alive and receive interrupts at set timer intervals while consuming just 1.6 μA at 2.2 Volts (see Figure 16). In LPM4, the processor can receive external interrupts (such as from the MicaZ) and consume only 500 nA at 2.2 Volts.

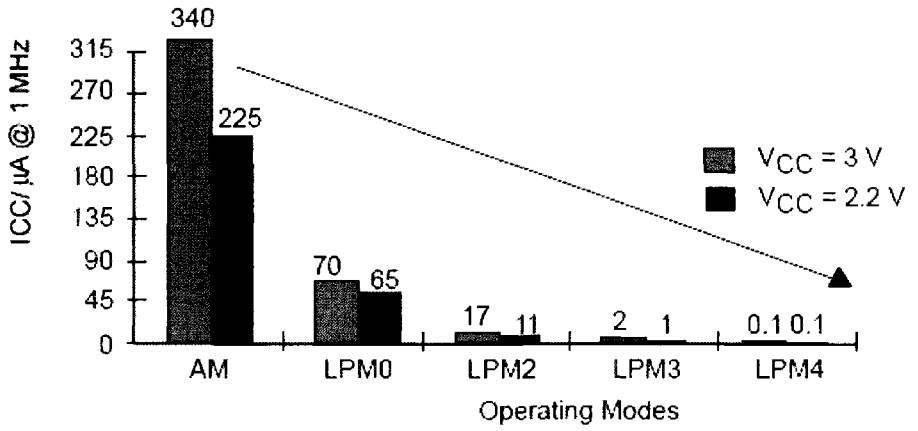


Figure 16. Example of current (and thus power) consumption of the MSP430 microcontroller in the four available low-power modes (compared to active mode (AM)) [33].

Care was taken to configure all unused pins properly to ensure lowest power consumption by following the MSP430 user's guide instructions, "Unused I/O pins should be configured as I/O function, output direction, and left unconnected on the PC board, to reduce power consumption" [33]. Additionally, other measures were taken to ensure low power consumption as described below.

MICROCONTROLLER POWER MANAGEMENT

In our design, we not only change the supply voltage based on the operation being performed, but also ensure operations were performed quickly, returning the microcontroller to the lowest power mode possible, in most cases LPM3. The processor only goes active or to a higher-power mode when awoken by an interrupt either from the I^2C port or USART. Figure 17 shows the state machine implemented in the microcontroller. Very few operations actually required the processor to enter active mode (AM) since the processor has a robust direct memory access (DMA) controller for

most peripherals. The processor only enters active mode to change states in the state machine. Therefore, the processor is left in LMP0 while the DMA transfers data to memory for use by the microcontroller. In the case of debugging, the memory used to store incoming messages from the I^2C interface can be the same memory used to write out to the UART, meaning that the processor is only awoken to transition between states, and the two respective DMAs can handle all data transfer.

Two scenarios require higher voltages: use of the chosen text-to-speech (TTS) chip and writing to onboard Flash non-volatile memory. Both of these require 2.7 V, and we run the system at 2.75 V under these circumstances. When not performing these functions, the chip runs at 2.25 V saving significant energy. Figure 18, taken from the MSP430 data sheet for the microcontroller used on the daughter card, shows the limitation in operating frequency based on applied voltage.

Power consumption scales linearly with frequency and with the square of voltage as shown in Equation 1:

$$P \propto C V_{DD}^2 f + V_{DD} I_{Leakage} \tag{1}$$

Where V_{DD} is the supply voltage,

f is the frequency of operation,

$I_{Leakage}$ is the static leakage current, and

C is the gate capacitance of the transistors (or load capacitance).

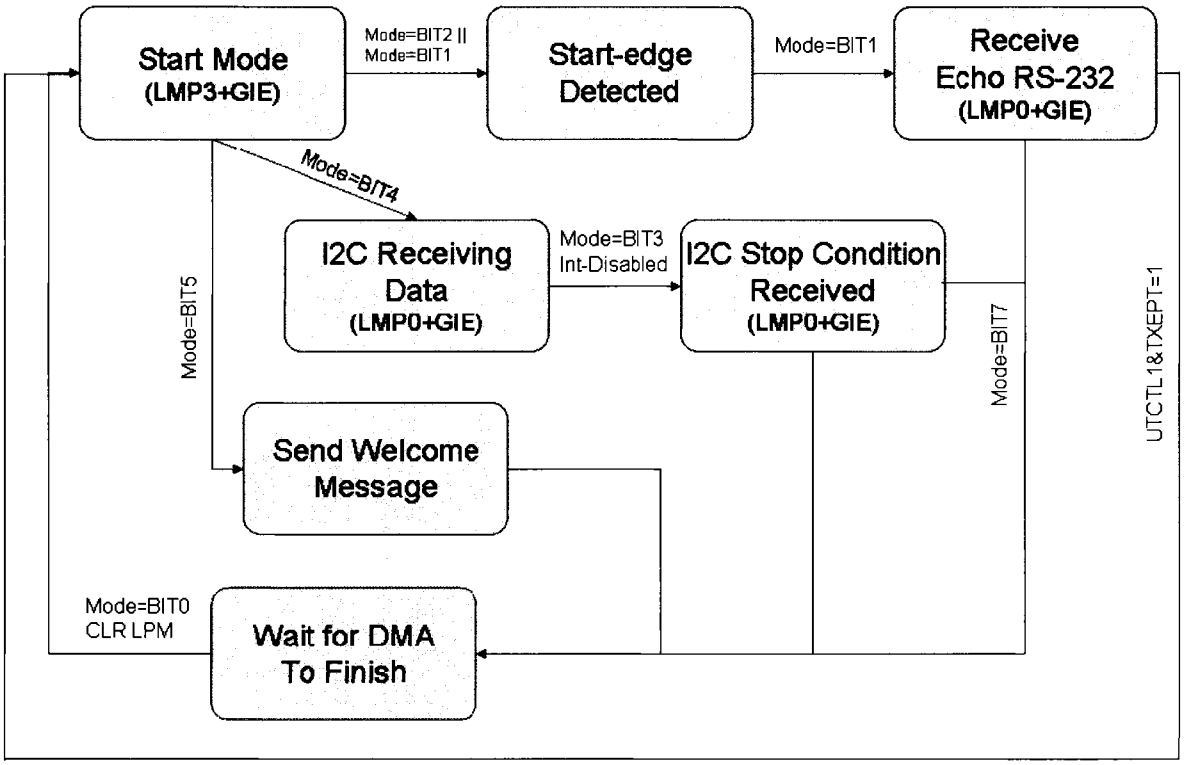


Figure 17. State machine running in daughter card microcontroller.

We run the chip at a frequency of 4.60 MHz which is 400 kHz below the maximum allowable frequency at 2.20 V of 5.01 MHz. This somewhat unusual clock speed was chosen to ensure the baud rate generator onboard the microprocessor could produce a consistent 38400 bit rate for the RS-232 debug port. Three registers control the baud rate of the UART: UBR0, UBR1, and UMCTL. UBR0 and UBR1 are the whole-number part of the fraction of system clock to baud rate for the on-board, baud rate calculator, while UMCTL is an additional modulation register that attempts to handle the fractional part of the above ratio. By choosing an even multiple of 38400, the master clock needs no additional modulation bits (UMCTL) since there is no fractional part of the clock rate to baud rate. Below is the code used to set the registers:

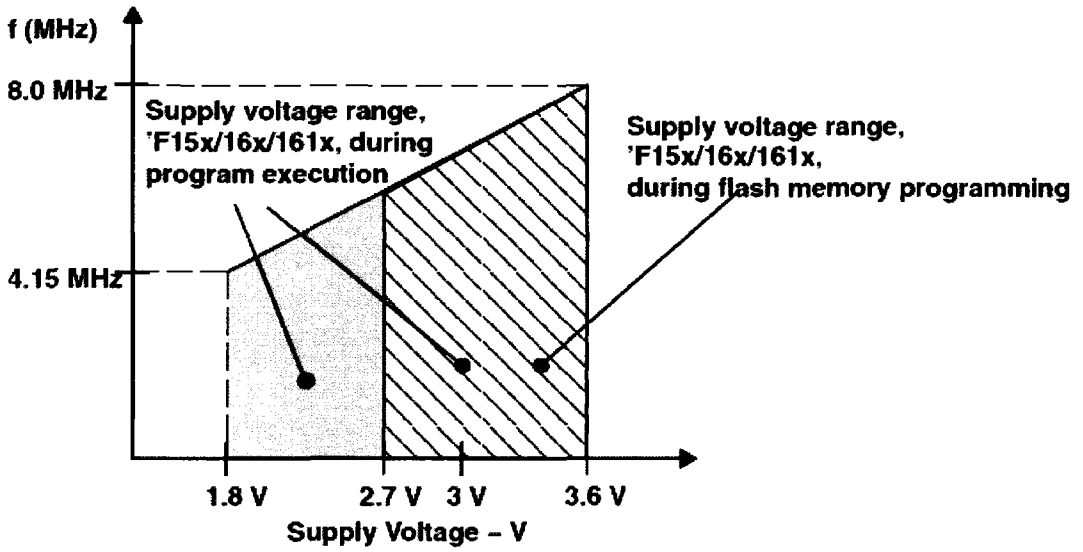


Figure 18. Limitations in operating frequency based on supply voltage [33].

```
UBR01 = 0x78;    // 4.6MHz/38400
UBR11 = 0x00;    //
UMCTL1 = 0x00;    // Modulation 4A
```

Since the microcontroller uses a RISC instruction set, keeping a constant clock frequency does little to increase power consumption versus using clock throttling. The amount of energy consumed by operating at the higher frequency is compensated by returning to sleep mode a proportional amount of time earlier. It should be noted the microcontroller uses a digitally controlled oscillator (DCO) with notoriously poor accuracy. Periodically, we synchronize this oscillator to the MicroCrystal MS1V-T1K 32.768 kHz crystal in active mode in the microcontroller. This crystal is rated at 10.00 *pF* with +20/-20 parts per million (ppm) frequency tolerance. The crystal has a temperature coefficient of: $(T-25C)^2 \cdot -0.025\frac{ppm}{^oC^2}$, where T is the operating temperature. Additionally, an external resistor was used to ensure added stability of the DCO, called ROSC.

In Equation 1, C and f are essentially constant. We can quantify the amount of power saved by changing voltage from 2.75 to 2.25 Volts (as opposed to running at 2.75 Volts continuously), by examining the predicted current consumption provided by the manufacturer, Texas Instruments:

$$I_{(AM)} = I_{(AM)[3V]} + K_{VS}\left(V_{CC} - 3V\right) \quad [33] \tag{2}$$

where *I(AM)[3V]* is nominally calculated at 2.3 mA for a frequency of 4.6 MHz and K_{VS} is the relationship of current consumption to supply voltage, supplied by the data sheet as $210\frac{\mu A}{V}$. Therefore, the active mode current for 2.25 V is 2.142 mA, while for 2.75 V is 2.248 mA. Additionally, we will assume the processor returns to LPM3 when all tasks are performed. The current consumption in LPM3 at 2.25 V is approximately 1.4 μA, while it's approximately 2.2 μA at 2.75 V from interpolating values from the data sheet [33]. Figure 19 shows the results of voltage scaling the microcontroller using the above scheme. As shown in chapter II, we can expect duty cycles on the order of <5% for a Mote using the MACs discussed. Since not all messages will necessarily wake the daughter card, we can expect an even lower duty cycle for it. From the graph, we can see the scheme performs best for lower duty cycles saving about 52% power at extremely low duty cycles.

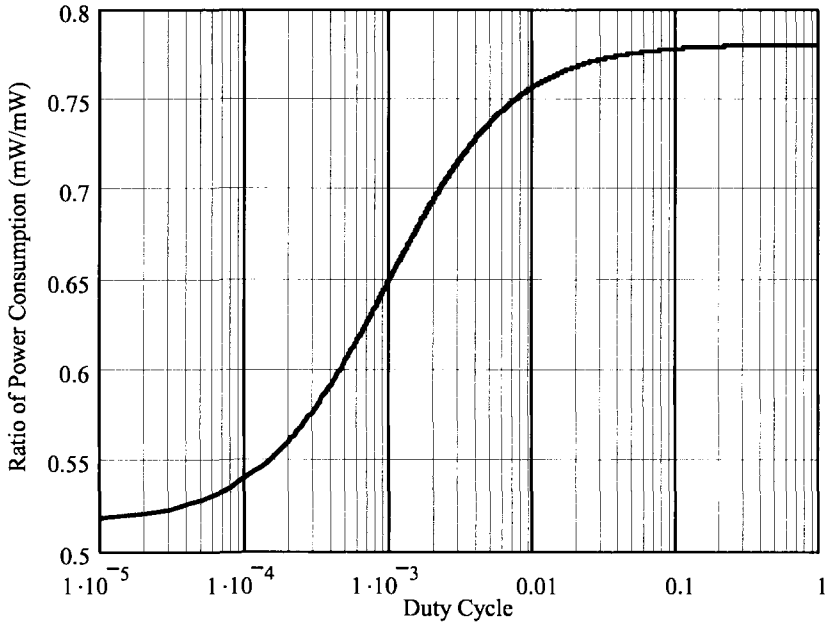


Figure 19. Ratio of power consumption at 2.25 V over 2.75 V of the MSP430 on the daughter card based on duty cycle.

Care was taken to ensure lowest power consumption on all pins by following the MSP430 user's guide instructions, "Unused I/O pins should be configured as I/O function, output direction, and left unconnected on the PC board, to reduce power consumption."

POWER MANAGEMENT OF OTHER COMPONENTS

The system would likely be powered by two AA batteries, in parallel, generating 3 V (1.5 V each). An Energizer ™ E91 Alkaline battery's capacity actually varies with load or discharge current, as shown in Figure 20, so keeping load current as small as possible is also important in increasing battery lifetime. In order to fully determine expected battery capacity, we should account for each component on the daughter card. The largest components to consume power are the light-emitting diodes, the voltage regulator,

the boost convertor to drive the piezo transducers, the RS-232 driver, and the Winbond WTS701 text-to-speech (TTS) integrated circuit (IC).

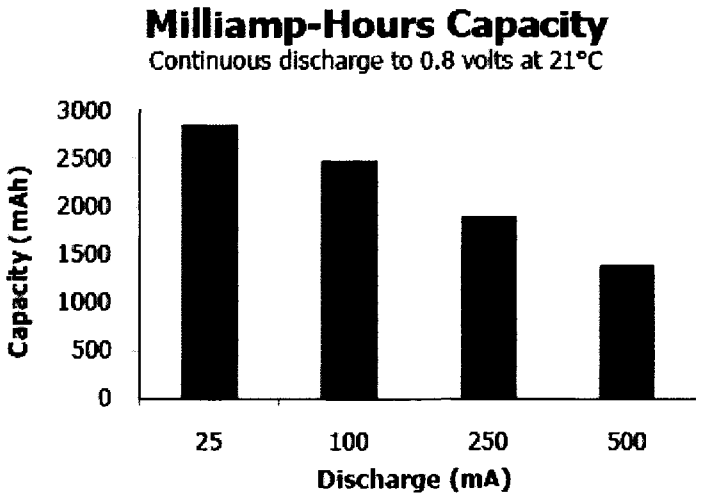


Figure 20. Alkaline battery capacity as a function of load current [34].

The Winbond TTS IC can be powered from 2.7 to 3.3 V, typically consuming 35 mA, and has the capability of being placed into sleep mode using less the 1 μA. The board has seven LEDs each biased with a 360 Ω resistor. These LEDs consume approximately 1.8 to 2.2 mA each of current depending on the supply voltage. The power consumed by the piezo-electric transducers is discussed later. Those transducers were not implemented in this effort but were modeled and components were placed on the daughter card to drive them. The choice of and design of the voltage regulator is in the next subsection.

We chose an RS-232 driver specifically designed for low-power operations, the Maxim 3318E. They may be powered with voltages ranging from 2.25 to 3.0 V and have an auto shutdown capability when no activity is detected for more than 30 seconds. The device uses only 1-10 μA when in shutdown mode and 300 to 1000 μA when in active

mode. To achieve this, the part does not meet EIA-232 requirements for transmitter voltage level which specifies ±5 V. The manufacturer states that the ±3.7-4.2 V output voltage that the part achieves should function properly with most modern transceivers, and, indeed, we noticed no problems with RS-232 functionality.

ACHIEVING VOLTAGE SWITCHING

As many hardware vendors profess, using a voltage regulator in a battery-powered application has many advantages that offset the additional component cost incurred. Those advantages include a more stable voltage level as battery voltage decreases, increased battery life, and the ability to dynamically change voltage, as described above, to conserve even more power. In order to efficiently lower the voltage, the designer generally has two choices: a switching regulator or a low-dropout (LDO) regulator.

The switching regulator uses field-effect transistors (FET) to synchronously switch on and off, charging an external inductor and capacitor to provide the appropriate voltage level to the load. These devices, although relatively expensive, are extremely efficient, usually achieving 85-95% efficiency when supplying a reasonable amount of current in most applications. The trade-off when using these devices is that the rapid switching can increase noise from the power supply causing the designer to use more robust decoupling capacitors and choose parts with a high power-supply rejection ratio. Another draw-back is their rapid drop in efficiency with low current loads.

LDO regulators are similar to traditional shunt regulators (similar to Zener diodes being placed in parallel with the load) using a FET in place of a diode. The efficiency of LDO regulators is proportional to the difference in voltage between the source and the load as shown in Equation 3. "Low dropout" means that the regulator is more efficient

than traditional shunt regulators because they are designed for a much smaller difference between source and load voltages. Of course, that means you must have an application where the source (batteries in our case) voltage is close to the desired voltage. The best efficiency that can be achieved, in our case of running at 2.25 V with two AA batteries, would be given by Equation 3.

$$\eta_{LDO} = \frac{I_o V_o}{(I_o + I_Q) V_I} \tag{3}$$

Where V_o and V_I are the output and input voltages respectively, I_o is the output current, and I_Q is the quiescent current. A LDO regulator considered for this project, the TPS715xx, has a quiescent current of 3.2 μA. Assuming active mode, with current draw of $I_o >> I_Q$, it would yield an efficiency of 75%. If we assumed that all LEDs are turned off and all components are in sleep or LMP3 for the microcontroller, the load current would be less than 50 μA. This yields an efficiency of about 70%. Figure 21 shows the drop-out voltage with respect to output current for this regulator. Assuming all LEDs are on, and the TTS IC is on, the output current would be close to 50 mA, yielding a dropout of 0.4 V at room temperature. Unfortunately, this means the drop-out required to run at 2.75 V was too high to consider this component.

The switching regulator used in the daughter card design is the TI TPS62200. The efficiency of this switching regulator with respect to load current is shown in Figure 22. As can be seen by the figure, when the daughter card is in sleep mode, consuming less than 100 μA, this regulator can be less efficient than the LDO regulator. This is largely due to the higher quiescent current. The no-load quiescent current is approximately 16 μA. Assuming a single LED is on with a 50% duty cycle when the card is in sleep

mode would correspondingly increase the load current and allow the regulator to boost efficiency up to the 90% region. Additionally, if the mote board consumes more than a few tens of micro Amps, the switching regulator would operate more efficiently.

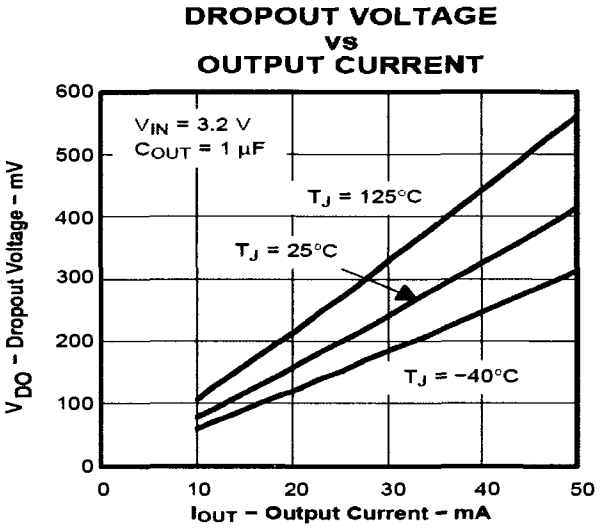


Figure 21. Dropout voltage vs. load current for LDO regulator considered for this project [35].

Performing a duty cycle vs. power consumption calculation will allow us to determine approximately when selecting one type of regulator may be better than the other. Figure 23 shows that the duty cycle would have to be greater than approximately 2% for the switching regulator to have a longer battery life than the LDO. This assumes a minimum of sleep current drain (no LEDs on and low power consumption by the Mote). Additionally, a battery capacity of 2700 mA-hr was used, referencing Figure 20. In low duty-cycle applications, it becomes clear that the LDO regulator is more efficient than using a switching regulator. We chose the switching regulator because the duty cycle and load currents were not initially known, and the LDO regulator could not

support operations at the higher voltage required (2.75 V). Additionally, the switching regulator has significantly less drop out, allowing operation when the batteries have lost significant amounts of charge. Both regulators allow dynamically adjusting voltage based on the mode of operation, but the LDO regulator would generate more heat in active mode.

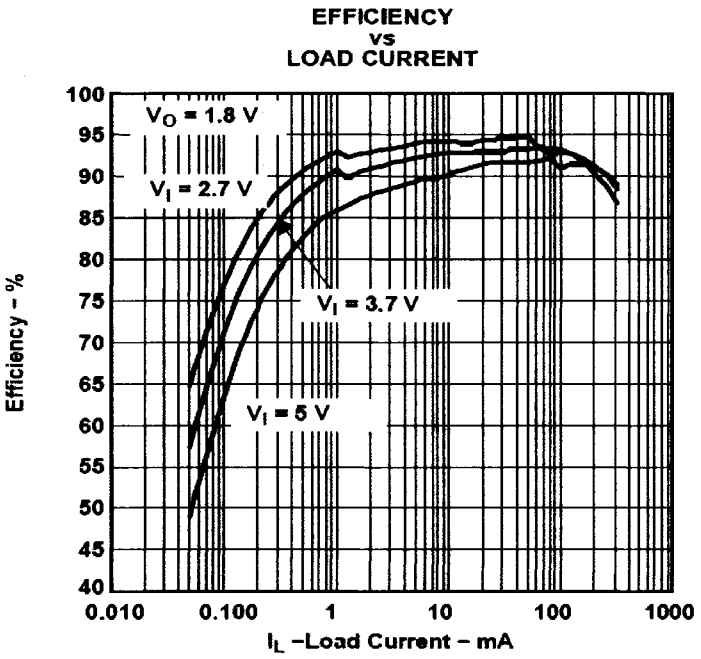


Figure 22. Efficiency of the switching regulator with respect to load current [36].

The switching regulator uses a voltage-divider circuit to determine the output voltage as shown in Figure 24. The internal circuitry will automatically attempt to adjust the output voltage to make the voltage at the feedback pin (FB) 0.5 V. In order to dynamically change the output voltage, a resistor is added to the voltage divider and connected to a GPIO pin on the microcontroller. This resistor is labeled $R_{Control}$ in - Figure 24. R_1 and R_2 should be kept as large as possible to reduce quiescent current, but

less than 1 MΩ to ensure stability. The output capacitors must be chosen based on the voltage-divider resistors to ensure stable operation. R_1, R_2, and $R_{Control}$ were also chosen from standard resistor values in the 1% tolerance range.

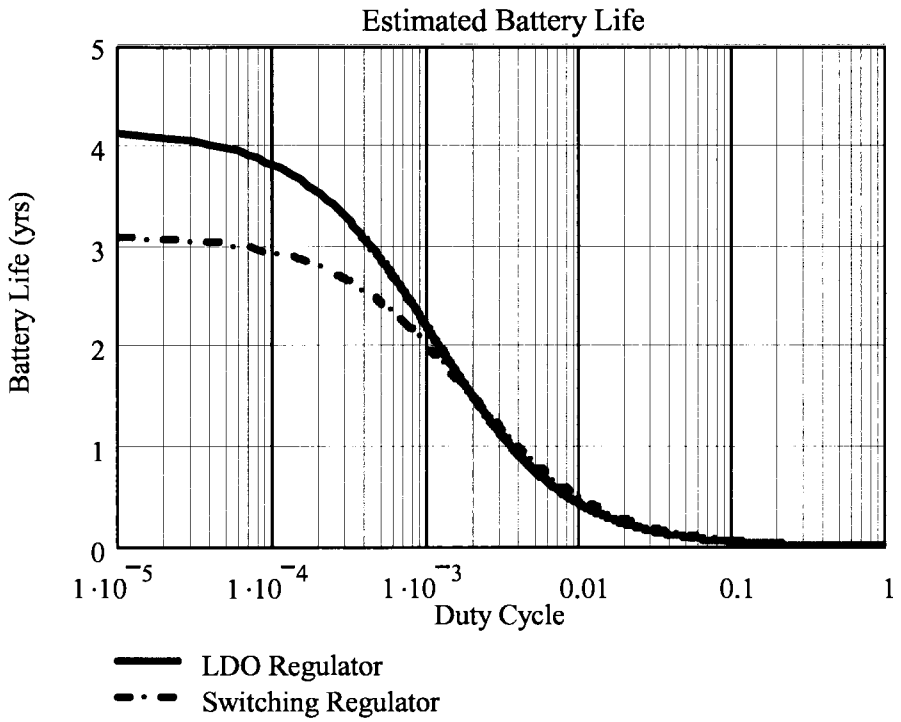


Figure 23. Estimated battery life of the daughter card, alone, assuming a choice of either the LDO or the switching regulator chosen. This assumes that all LEDs are off during sleep mode and the device consumes 50 mA during active mode.

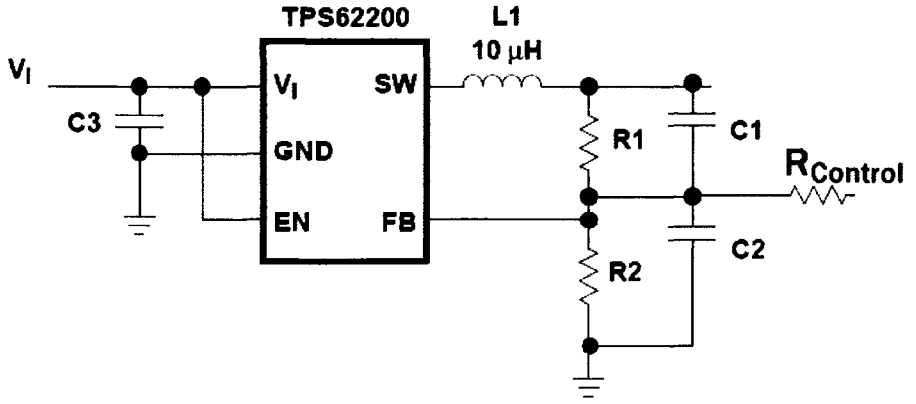


Figure 24. Configuration of switching regulator to allow for dynamic voltage adjustment [modified from 36].

To accomplish this, we have two equations with three unknowns:

$$V_{2.75} = \left(\frac{R_1}{R_p\left(R_2, R_{Control}\right)} + 1 \right) \cdot V_{FB} \tag{4}$$

$$V_{2.25} = \left(\frac{R_p\left(R_1, R_{Control}\right)}{R_2} + 1 \right) \cdot V_{FB} \tag{5}$$

where $R_p\left(R_a, R_b\right) = \frac{R_a R_b}{R_a + R_b}$, is simply the equivalent resistance of two parallel resistors, and V_{FB} is the voltage at the input of the onboard comparator to be compared with the input of the **FB** pin, nominally 0.5 *V*. Figure 25 shows these equations graphed for R_1 = 43 kΩ and R_2 = 10 kΩ. A control resistor, $R_{Control}$, value of 200 kΩ will yield the values of 2.75 V and 2.25 V desired. This creates a quiescent current of at least 52 µA. Figure 26 shows oscilloscope output of the voltage supply when commanded by the microcontroller. Page two of the schematics in the Appendix shows the final switching regulator design, where *R208*, *R209*, and *R210* are R_1, R_2, and $R_{Control}$ respectively.

MOTE-TO-DAUGHTER-CARD INTERFACE

As shown in Figure 15, communications between the Mote and the Daughter-card take place using the onboard I^2C bus. This relatively low-speed bus provides its own clock in a master-slave configuration and has two addressing modes allowing either seven or ten-bit addresses. We only use the seven-bit addressing mode. Data on the I^2C-bus can be transferred at rates of up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, or up to 3.4 Mbit/s in the High-speed mode [37]. The Mote is always the Master on the bus and determines the transfer clock speed. We found that there were two conflicting header files in the TinyOS 2.x release, named "Atm128I2C.h," causing a compiler error. To get the compiler to work, we renamed the one in the *{$TOSROOT}/tos/chips/atm128* directory so the compiler would not recognize it.

Due to a flaw and non-standard address forming in implementing the I^2C bus in TinyOS, significant debugging was required to get it working. Since the address is only seven bits, standard practice for address forming in software uses the lower seven bits of the addressing byte. TinyOS was using the upper seven bits. So an address that should be hexadecimal 0x48 when programmed on our daughter card microcontroller, and according to standard I^2C addressing, was actually interpreted as address 0x24 in TinyOS. After placing the I^2C bus on an oscilloscope and solving the address-resolution problem, we reviewed the source code for I^2C implementation to ensure it additionally conformed to accepted norms. From this review, we found another flaw in the I^2C implementation. In Atm128I2CMasterPacketP.nc [on line 333, shown below], there was an error with setting the microcontroller for read/writing:

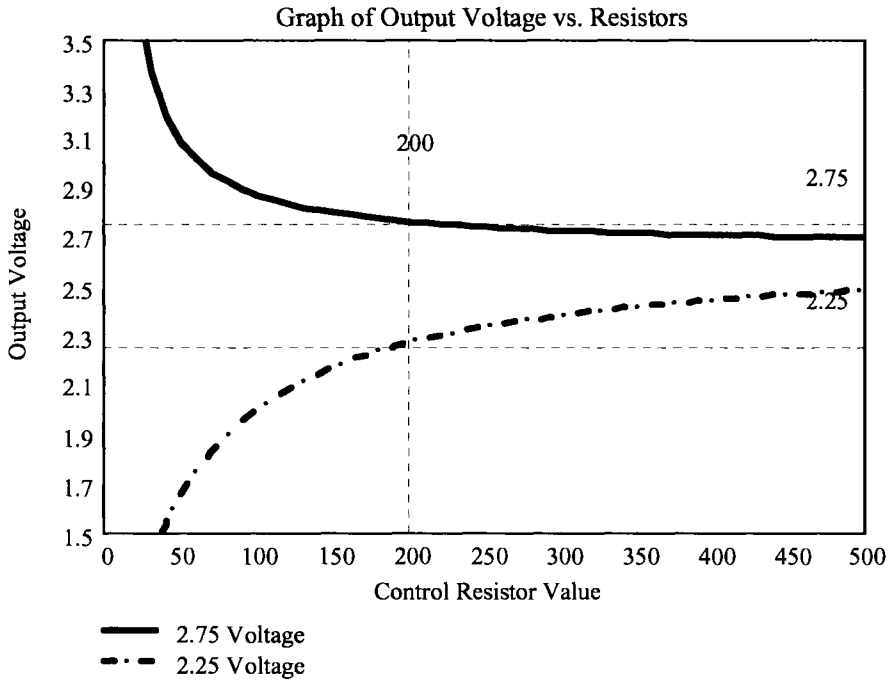


Figure 25. Graph of output voltage vs. control resistor for R_1 = 43 kΩ and R_2 = 10 kΩ.

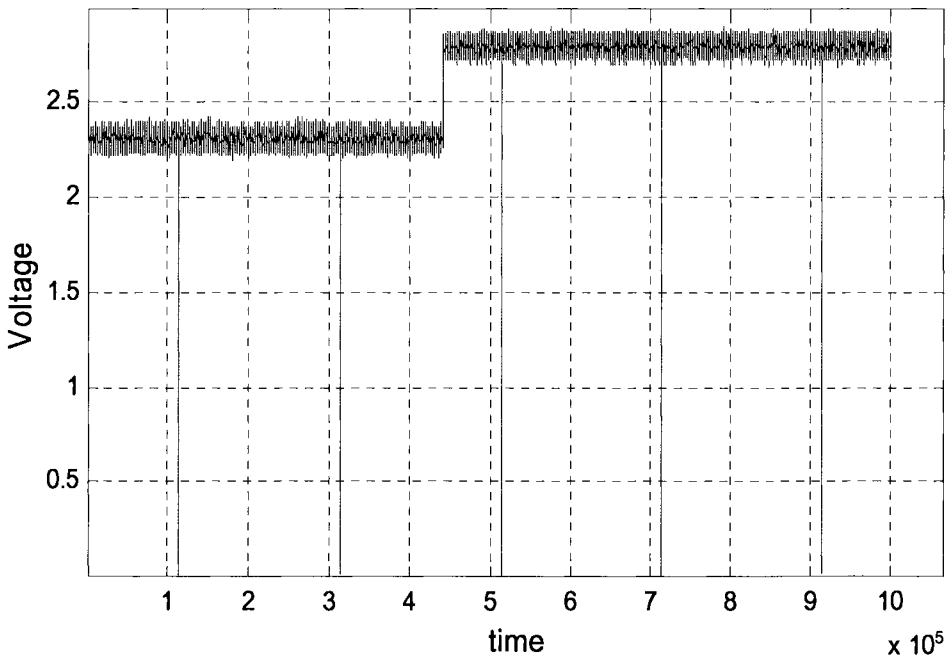


Figure 26. Actual performance of voltage switching on the daughter card.

```
if (reading == TRUE) {
    call I2C.write((packetAddr & 0xff) | ATM128_I2C_SLA_READ);
  }
  else
    call I2C.write((packetAddr & 0xff) | ATM128_I2C_SLA_WRITE);
```

In this code segment, *ATM128_I2C_SLA_READ(0x00)* is supposed to *call I2C.write* with a "0" as the least significant bit (LSB) to enable read mode. Should *packetAddr* contain an odd address (ending in '1'), then *call I2C.write* would always be in write mode. This bug was fixed in the latest release of TinyOS due to our input to the mail-list. Additionally, TinyOS was modified to standard addressing in later versions.

TEXT-TO-BRAILLE TRANSLATION

Since we envisioned the device to receive text messages from nearby sensors equipped with IEEE 802.15.4 transceivers, the daughter card must be able to translate received text to Braille. At this time, we consider the text to be encoded as standard ASCII. We address a single 8-bit port from the microcontroller to drive a 3 by 2 set of pins or vibrational devices to emulate Braille. These same pins drive six LEDs onboard the daughter card to provide additional feedback to the user (should they maintain some ability to see) and for debugging purposes. Additionally, the timing of addressing the actuators allows for variation so, for instance, the Braille letter may be "scrolled in" from the left, right, etc. to give the reader the illusion of running their hand over the text in a certain direction. Finally, the level of actuation could be made variable allowing not only for different pressures but also different heights. It should be noted that should the sensor need to tell the transmitter it has received the message, acknowledgments can be turned on in the 802.15.4 protocol.

According to the Braille Authority of North America (BANA), two grades of Braille are typically used where Grade 2 allows the use of contractions to simplify word representation [9]. Additionally, Braille ASCII and other variations on Braille exist, such as increasing the standard cell from three rows by two columns to four rows by two columns. We concentrated only on Grade 1 Braille using the standard 3x2 cell. According to Gardner, Braille cell dimensions vary from country to country [38]. After review of the literature available, we determined the dimensions in Figure 27 would provide a user with a typical Braille experience [39]. Additionally, we determined that presenting characters for about a quarter of a second per character to the reader is enough time to identify the character. In fact, Foulke found that typically a single character could be identified within 30-40 msec [40] while Flanigan quotes the optimal presentation time to be 600 msec [41]. Flanigan used 300 msec as his "control" presentation time in his experiments, so 250 msec in our system seemed like a reasonable trade between memory buffer requirements and intelligibleness of the output.

ASCII TO BRAILLE

We chose Port four on the MSP430 to drive the Braille output and LEDs. Keeping all I/O lines together in a contiguous port allows for a single microcontroller write for each manipulation of the Braille cell. In order to determine the correct Braille cell to display for each character, we assume two bytes are needed for each character and store 94, two-byte, values to translate from ASCII to Port 4 on the microcontroller. The programmer can easily modify readout speed through the variable:

#define BRAILLE_WRITE_TIME 8192

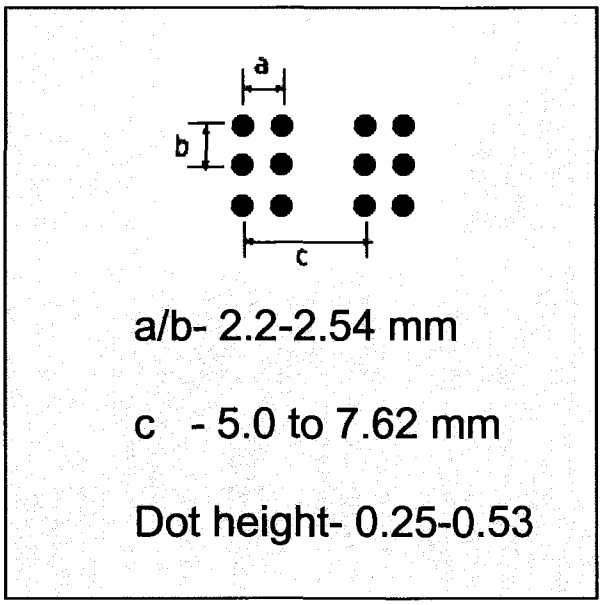


Figure 27. Standard or normal Braille cell dimensions for English Braille [38,39].

As indicated, the current readout speed is 8192/32768 or 250 msec. We wanted to ensure we chose a value for read-out speed commensurate with established research in the field as described above. Two bytes were chosen since many characters require two Braille cells for correct display; the most obvious being numerical characters. Figure 28 shows the standard Braille English alphabet. Note that letters "a" through "j" may also represent the numbers "1" through "0". The way the two uses are distinguished is through the use of a prefix cell. A number sign, shown at the bottom of the figure, precedes the corresponding letter cell to indicate a number. This means it may take 500 msec to display a single number. Similarly, the same is true for capital letters. It should be noted that the code is written such that if only one cell is necessary to represent a character, then only the second byte will be displayed, and the character will only be displayed for 250 msec.

We chose to represent 94 total characters and variants based on the BANA standard Braille codes [42]. Since memory is limited in the microcontroller, we limited the characters stored to those in standard ASCII. Based on the layout of the LEDs, the function *convertASCIIToHW()* converts the incoming I^2C data to the hardware mapping of the I/O lines in Port four to the Braille cell.

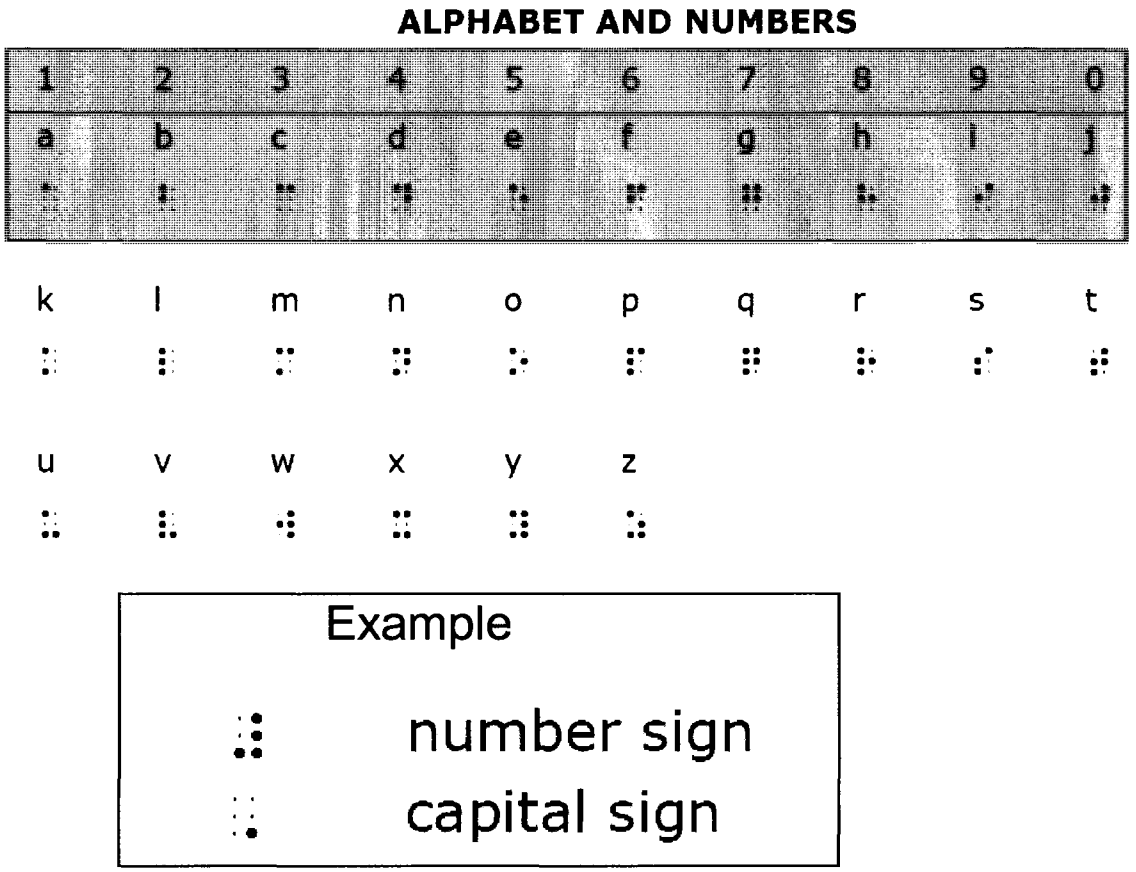


Figure 28. Standard Braille characters with modifier prefixes [9].

PIEZOELECTRIC BRAILLE DRIVER DESIGN

In order to provide enough force to stimulate the user's fingers, we decided to investigate the use of piezoelectric transducers. The piezoelectric effect was first

discovered by Jacques and Pierre Curie in 1880. The effect occurs in asymmetric crystals where compression along the "hemihedral axes produces electric polarization" [43]. Additionally, when the crystal is stretched in the same direction, the opposite electric effect occurs. It was later discovered that a voltage applied across a piezoelectric crystal produces either compression or expansion [44]. The crystal essentially is polarized with positive and negative sides by applying a polarizing electric field while the crystal is heated past the Curie temperature. The stretching effect can be increased by stacking crystals either in series or parallel. A standard bending lever is shown in Figure 29. This figure shows two elements in series. The crystals can be thought of as capacitors placed together in series and electrically appear that way when integrated into electronic circuitry.

Using equations 6 and 7 for parallel and series deformation, we calculated the amount of lever action that could be achieved for two different piezoelectric materials designated *PSI-5A4E PIEZOCERAMIC,* "A" (Navy Type II, Lead-Zirconate Titanate), and *PSI-5H4E PIEZOCERAMIC* ,"H" (Navy Type VI, Lead-Zirconate Titanate) [45]. Note that the Navy designations are defined in DOD-STD-1376A(SH) Ceramic Types I-VI. The results in Figure 30 clearly show a parallel combination of material "H", using the thinnest available material, yields the most deflection.

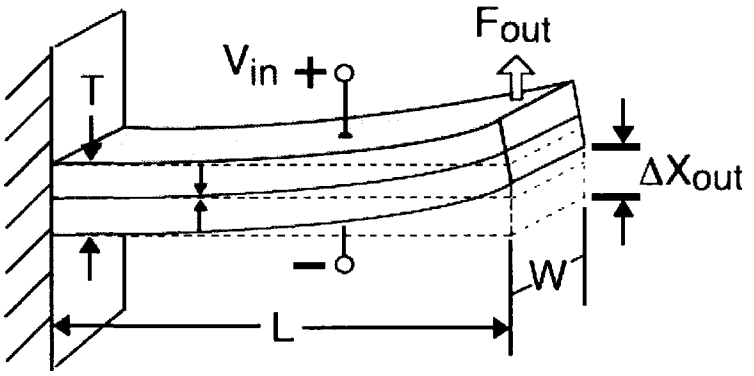


Figure 29. Bending element consisting of two piezoelectric crystals in series configuration. The arrows in the crystals indicated the direction of polarization [45].

$$\Delta X_{outPar} = \frac{3L^2 V d_{31}}{T^2} \tag{6}$$

$$\Delta X_{outSerial} = \frac{1}{2}\Delta X_{outPar} \tag{7}$$

Where T is the thickness of the piezoelectric stack,

d31 is the piezo "d" coefficient or Strain Produced / Electric Field Applied,

and the other variables are described in Figure 29.

We can see from the figure that it would be necessary to attach a mechanical lever to achieve the 0.25 mm deflection necessary to simulate a Braille dot, making piezoelectric motors difficult to implement as a Haptic device. It may be possible, though, for the user to detect a vibrating pin instead of a stationary pin, so the electronic drive circuit created allows for applying the drive voltage rapidly to provide a vibrational tactile response to the user for experimentation.

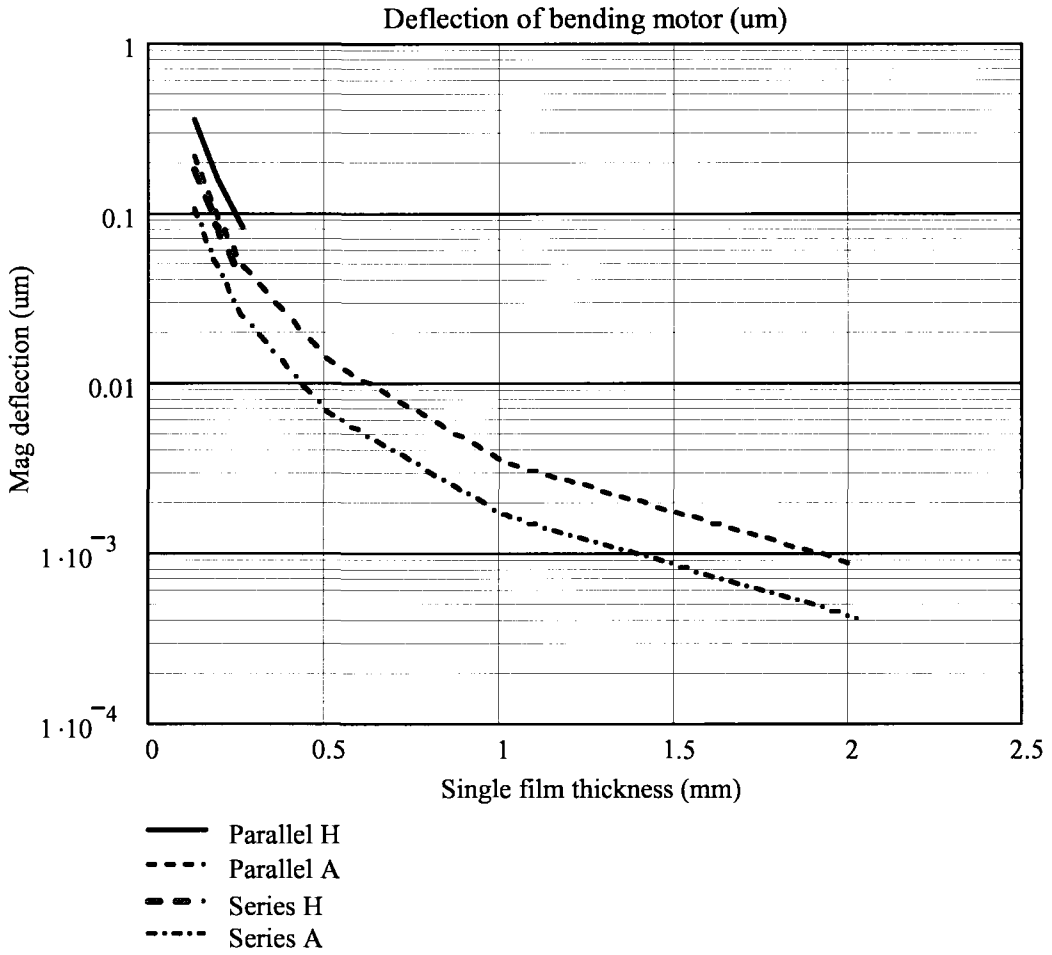


Figure 30. Calculated amount of deflection for an applied voltage of 18 V. Note that the "H" material was limited in available thicknesses by the manufacturer.

The circuit designed for applying the signal to the highly-capacitive piezo material is shown in Figure 31. V_{pI} or V_{in} is a voltage from an IO pin of the MSP430 microcontroller. This pin can be driven from an on-board pulse-width modulator (PWM) peripheral, allowing variable output power to drive the capacitor. AV_{cc} is 18 V provided by a boost-convertor on the daughter card. When the V_{in} is low, **X2** and **X3** turn off, causing a AV_{cc} to be applied to the gate of **X1**. This turns **X1** on, supplying AV_{cc} to the piezo capacitor. When V_{in} is high, **X2** and **X3** turn on, driving both the capacitor voltage

and the voltage at the gate of **X1** low. This circuit was modeled with ngspice, a free and open-source version of Berkeley's Spice 3f5 [46]. Transistor models for ngpice were downloaded from vender web sites and virtual ammeters (using voltage sources with 0 V) were placed in the circuit to measure transient currents. The only difference between the model and the daughter card design was changing AV_{cc} from 20 V in the simulator to 18 V on the daughter card.

The interface provided for the free and open-source ngspice package is totally text-based and the recommended graphical interface was difficult to compile and build to meet all of its Linux dependencies. As part of this project, we wrote a Java-based interface to ngspice, called KJWaves, which does the following:

1) imports netlists generated from the schematic capture tool,

2) allows the user to add various additional analysis and mathematical functions, and

3) reads the resulting file and provides graphical display.

KJWaves has received support from the open-source community and now is available in English, German, Spanish, and Greek. It has had over 1500 downloads from the repository on sourceforge.net [47].

Figure 32 shows the output of the circuit for an input signal of:

VP1 Vin 0 PWL 0 2.75 24u 2.75 25u 0 49u 0 50u 2.75 74u 2.75 75u 0 99us 0 100us 2.75 124u 2.75 125u 0 149u 0 150u 2.75 174u 2.75 175u 0 199u 0 200u 2.75 249u 2.75 250u 0 299u 0 300u 2.75 349u 2.75 350u 0 399 0 400u 2.75

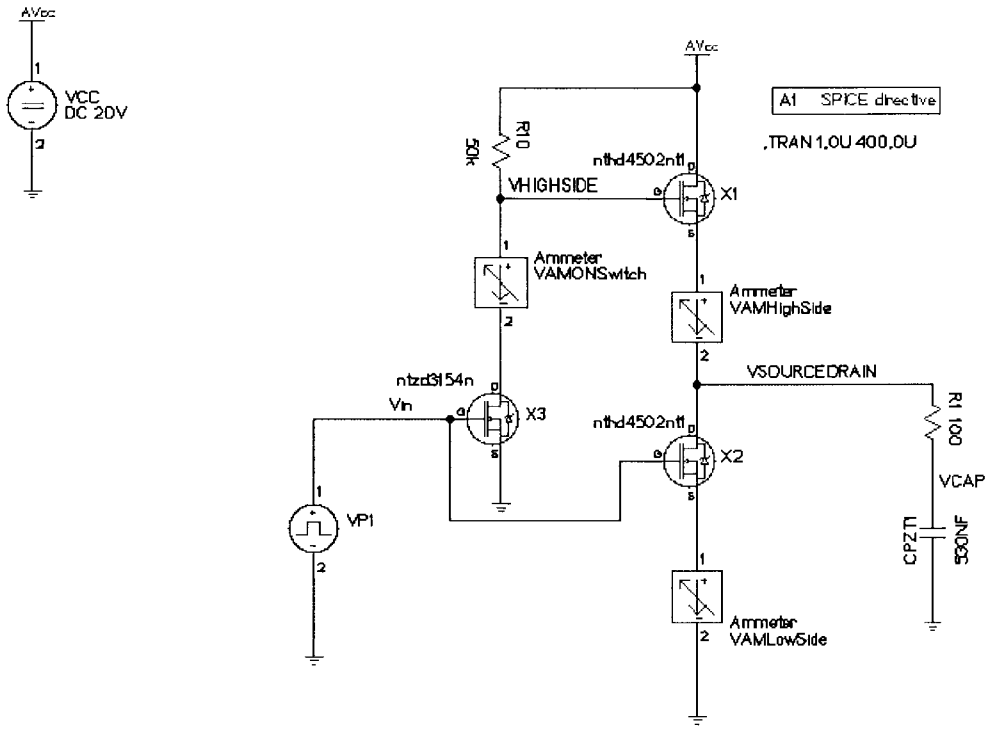


Figure 31. Piezoelectric driver circuit.

This input signal was chosen to examine the frequency response of the circuit as well as to determine if any voltage decay occurred in the holding voltage during operation. It's interesting to see that the output voltage, in red, is delayed slightly as the voltage at the gate of **X1** builds. Resistor **R1** is in place to limit current draw by the piezo capacitor. This resistor was chosen as a 1 W resistor instead of the standard $1/8^{th}$ W due to higher current and voltage loads.

To achieve the voltage required for the deflection, an Analog Devices ADP1611 DC-to-DC switching converter was chosen. This device is 90% efficient with the ability to convert an input voltage of 2.5 V to 20 V with a switching frequency of 1.2 MHz. A 22 nF soft-start capacitor was chosen which makes the startup response time of the device approximately 1.2 ms. The device is turned on an off through the use of the shutdown

pin connected to the microcontroller (to ensure the board is operating above 2.5 V before turning on) and will draw about 10 nA in shutdown mode.

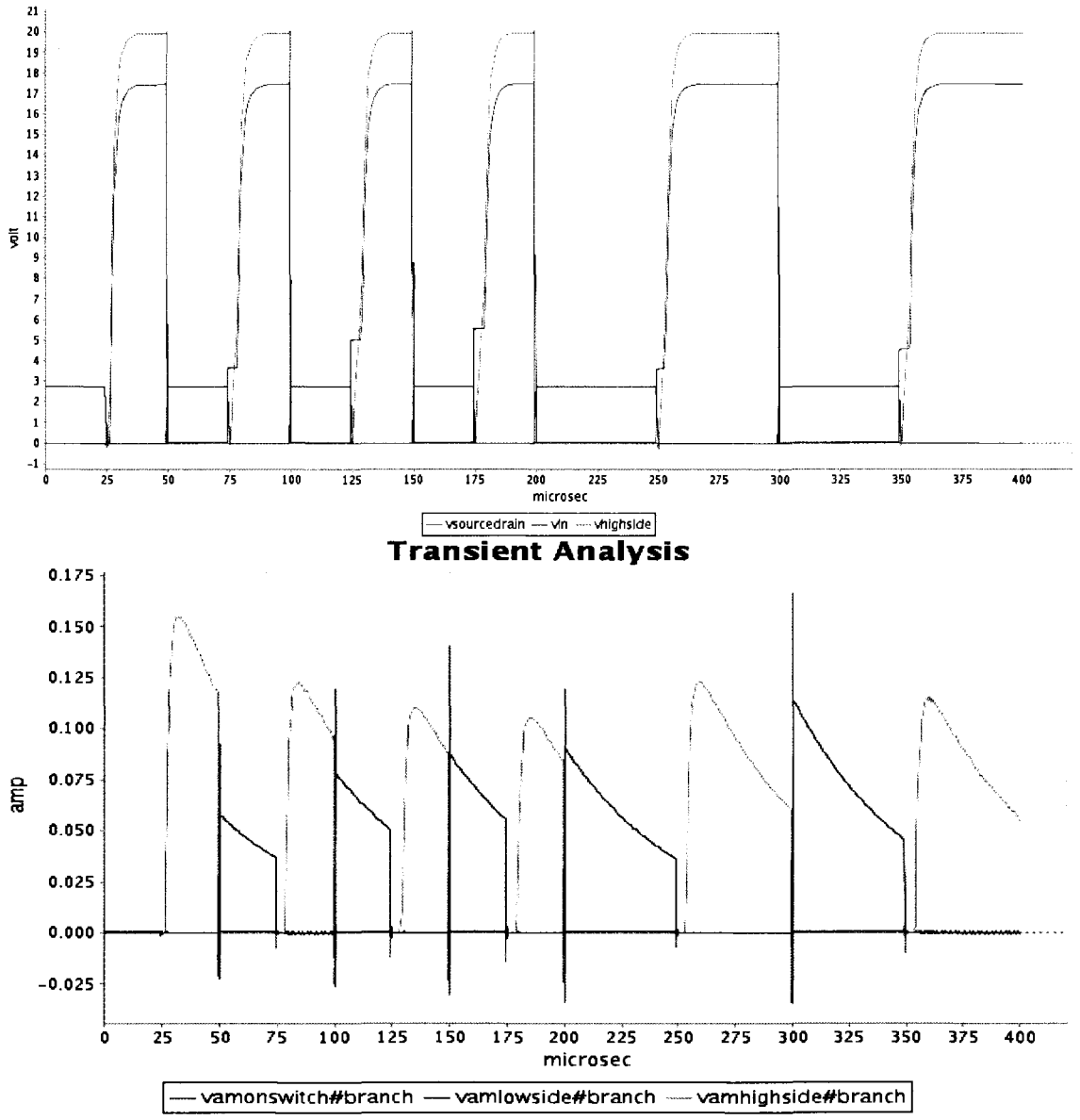


Figure 32. Transient analysis of piezo driver circuit showing annotated voltages (upper) and currents (lower).

TEXT-TO-SPEECH TRANSLATION

Finally, the daughter card adds a Winbond WTS701 text-to-speech (TTS) chip connected to the microcontroller's SPI port. Although the MicaZ board has an SPI port as well, it is used for communications with the radio, making contention on the port an issue. The WTS701 comes in two varieties for male and female voices and will directly drive external speakers. It does require an external 24.576 MHz crystal oscillator as well as some external capacitors to function though. In standby, it consumes less than 1 μA, and can be operated with voltages as low as 2.7 V.

This chip has been discontinued largely for of two reasons: 1) its high cost (it cost over $35 US) and 2) many, if not all, of its functions can now be performed as efficiently with a low-cost digital signal processor (DSP) coupled with a low-cost audio amplifier/DAC. Therefore, we suggest, in a future design to replace much of the functionality in both the TTS processor and the microcontroller to be performed in an on-board DSP such as the Analog Devices Blackfin.

CHAPTER IV

SIMULATION AND HARDWARE RESULTS

This chapter describes results received with the actual hardware produced as well as simulations demonstrating behavior with multiple nodes and a single base station. The first section describes the hardware and results achieved by coupling the daughter card with a mote. Both the simulation and hardware demonstration use the same code as much as practical. The one exception is that the I^2C port was simulated with debug statements in the simulation since TOSSIM I^2C supporting does not exist.

Packets were periodically sent from the base station at a rate long enough for the furthest node available to receive data through multiple hops. In the simulation, number of hops and message receive times were recorded to determine how effectively we limited range and congestion by limiting the number of hops. In all, the hardware worked as planned, but the simulation code required many runs to ensure the noise model reflected a realistic environment. Simulation verified that the distance a message was transmitted could be limited by limiting the number of hops. To demonstrate the ability to disseminate base station messages while limiting the distance, we used TOSSIM with a 64 element grid with a spacing of four meters between nodes. To aid in debugging message transmission, a linear "chain" was used to ensure maximum number of hops.

HARDWARE RESULTS

The Gerber files for the daughter card (see Appendix for the stacked Gerber image) were sent to a printed circuit board (PCB) manufacturing house for fabrication. Once received, all parts, such as the microcontroller and RS-232 transceiver, were populated by

hand and tested as they were placed. All parts used, except connectors, were surface-mount. As mentioned in the previous chapter, in order to couple the daughter card with the MicaZ mote, the battery holder on the mote had to be removed, and a 51-pin connector was added to the back side. Figure 33 shows the daughter card mated with the Mote micaZ.

Hardware demonstrations showed reception of the transmitted base-station message through both the debug serial (RS-232) port to a PC (either running Window $XP_{[TM]}$ and HyperTerminal or running Linux and minicom) and through the 3x2 Braille LEDs.

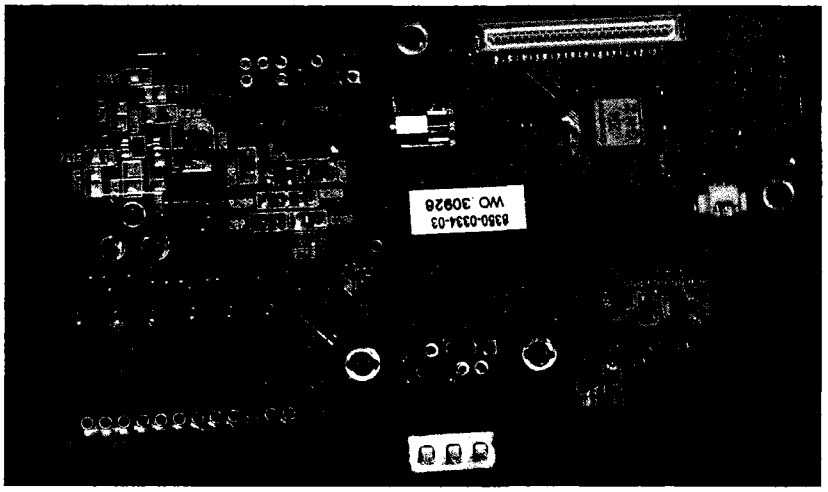


Figure 33. Daughter card coupled with a mote transceiver.

Three MicaZ motes were programmed with the Braille and daughter card interface code. Node zero in the code was hard-coded as the base station and transmits a message every *I2C_SEND_PERIOD_MILLI* milliseconds, which is set to 4096. This means that the base station transmits a "new" message every four seconds. During that time, receivers can receive a message and forward it on to their neighbor based on Trickle

(described in chapter II). When a node receives a new message, it will buffer it for transmission to the Braille LEDs and output the buffered text through the serial port. The standard Active Message packet was augmented with a payload of:

```
typedef nx_struct easydis_packet_t {
  nx_uint16_t nodeid;
  nx_uint8_t counter;
  nx_uint8_t priority;
  nx_uint8_t numberhops;
  nx_uint8_t data[TOSH_DATA_LENGTH-5];
} I2C_Dism_packet_t
```

where number of hops a packet takes and priority of the packet can be tracked as the packet transgresses the network. It should be noted that *TOSH_DATA_LENGTH* is modified to be 100 bytes by specifying *CFLAGS += -DTOSH_DATA_LENGTH=100* in the Makefile. Typical data size in a TinyOS packet is 28 bytes. Key aspects of the hardware test were that:

1) only new messages were presented to the user (despite the fact that the node could receive the same message multiple times from its neighbors),

2) Braille output faithfully represented the text transmitted, and

3) serial port output reflected the same values sent over the base station.

Figure 34 shows the LEDS displaying text sent by the base station. Characters were faithfully represented where each number was preceded by the appropriate number prefix. The delay between Braille characters was approximately 250 ms.

Listing One shows the serial port output from the mote, through the I^2C bus, to daughter card, and, finally, out the RS-232 port. The counter is incremented for each new message sent from the base station. Two buffers are used in the software: 1) on board the mote receiving and storing up to four messages and 2) onboard the daughter

card. Numerous, long, and/or quickly-changing messages can take too long to transcribe to the Braille LEDs, and thus cause an overflow. This overflow mechanism was also tested during hardware testing. Listing Two shows the output of the daughter card through the RS-232 port when an overflow is received. The software recovers gracefully once the buffer is emptied. A circular buffer is used so heap memory will not be corrupted. On the other hand, the user will likely receive poorly transcribed information.

Figure 34. LEDs on daughter card displaying the "numeric prefix" in Braille. Two Braille characters are displayed for numbers as described in the previous chapter.

Listing 1

MNS Braille Receiver
ver 0.1
<I2C Message counter: 1<I2C Message counter: 2
Local still alive.
<I2C Message counter: 3<I2C Message counter: 4
Local still alive.
<I2C Message counter: 5<I2C Message counter: 6
Local still alive.

<I2C Message counter: 7<I2C Message counter: 8
Local still alive.
<I2C Message counter: 9<I2C Message counter: 10
Local still alive.
<I2C Message counter: 11<I2C Message counter: 12

Listing 2

Local still alive.
<I2C Message counter: 30<I2C Message counter: 31
Local still alive.
<I2C Message counter: 32
Error: Buffer Overflow!
<I2C Message counter: 33
Error: Buffer Overflow!

SIMULATION RESULTS

SIMULATION DESCRIPTION

In the simulation, we assumed 64 nodes available to receive the message from the base station which was hard-coded as node "0". Each node is running the TinyOS 2.1 Drip library which is the same as the Dissemination library used TinyOS 2.0.1. Version 2.1 includes two additional dissemination libraries using the Trickle protocol described in chapter II, Dip and DHV. Nodes were separated by four meters, and the environment was assumed a noisy office environment, using the "Meyer Heavy" noise file provided with TinyOS, which sampled the spectrum at the Stanford Meyer library. This noise file has a noise floor of approximately -98 dBm [48].

Figure 35 shows a diagram of the motes used in the simulation, including the order and numbering of nodes distributed in simulation. Note that node zero is the base station, and reception gain to it was arbitrarily assigned for proper scaling of the legend. Assuming a noise floor of -98 dBm, we can see that a transmission from node 0 might at best reach nodes 24, 17, 10, and 3. Nodes further out would have to receive the message

via an adjacent node hop. The simulation was first run with no modifications to the Drip library to determine how long it takes to transfer a message to node 63. This allowed us to determine how much simulation time is required to realistically determine if distance can be limited by limiting hops and how to modify the library to limit the number of hops.

In order to create the grid in Figure 35, the TinyOS program "LinkLayerModel.java" was used. This propagation model creates a two "topology" files. One file lists the physical layout of the motes while the other contains "gains" between each node, giving an "RF layout." This file, which has the default name of *linkgain.out*, is imported into each node in TOSSIM for a realistic layout. Listing Three shows the parameters file used in *LinkLayerModel* that was used to create the topology used in the simulation. The link between nodes is considered symmetric, in that transmitting to a node has the same gain as receiving from that node.

The noise model applied to each mote is based on closest pattern matching (CPM). CPM uses actual measured noise traces and creates a statistical model from it [49]. Therefore, every simulation will be different because the simulation works with a probabilistic model of reception. TOSSIM develops conditional packet delivery functions (CPDFs) which is the probability that a packet will be delivered successfully given a certain number of successes or failures [49]. CPM, coupled with the propagation model, produces a very realistic simulation environment for sensor networks.

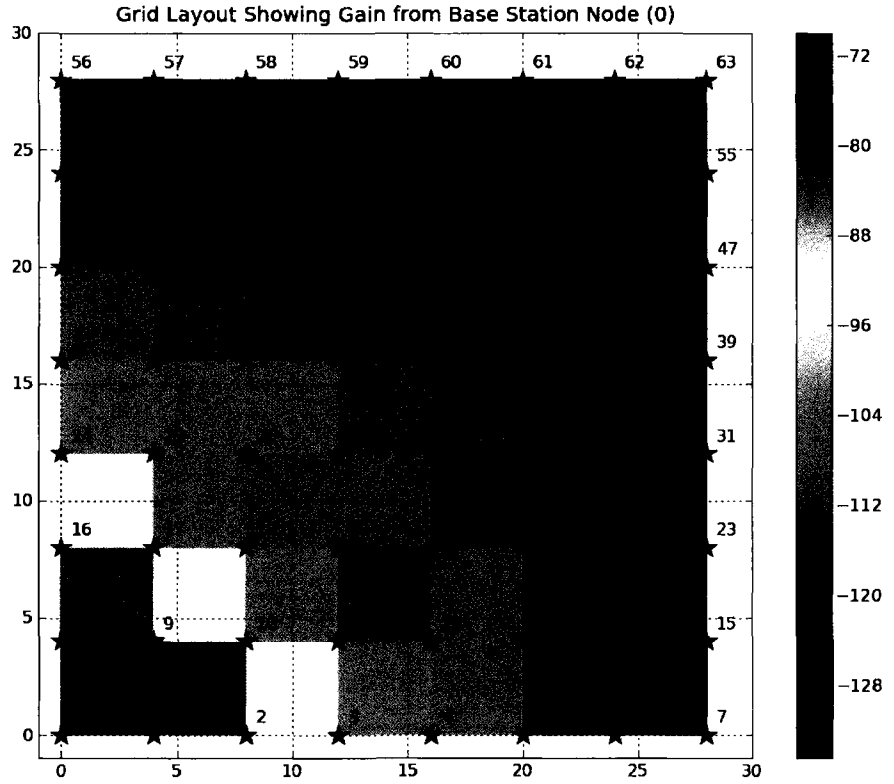


Figure 35. Diagram of mote layout used in the simulation. Colors indicate reception gain of each node from the base station. Node 0 is considered the base station and the reception gain was set arbitrarily for the base station to aid in graph auto-scaling.

Listing 3

% 64 node grid topology is deployed with an internode distance of 4m and symmetric links are assumed. The resulting configuration file is:
%%%%%%%%%%%%%%%%%%%%%%%%
% Channel Parameters
%%%%%%%%%%%%%%%%%%%%%%%%
PATH_LOSS_EXPONENT = 4.7;
SHADOWING_STANDARD_DEVIATION = 3.2;
D0 = 1.0;
PL_D0 = 55.4;

%%%%%%%%%%%%%%%%%%%%%%%%
% Radio Parameters
%%%%%%%%%%%%%%%%%%%%%%%%

```
NOISE_FLOOR = -105.0;
S11 = 0;
S22 = 0;
WHITE_GAUSSIAN_NOISE = 4;
%%%%%%%%%%%%%%%%%%%%%%%%
% Topology Parameters
%%%%%%%%%%%%%%%%%%%%%%%%
% available topologies :
% - GRID (1)
% - UNIFORM (2)
% - RANDOM (3)
% - FILE (4)
TOPOLOGY = 1;
GRID_UNIT = 4.0;
NUMBER_OF_NODES = 64;
```

A Python script (*testscript.py*, see Appendix) was created to allow the user to run TOSSIM specifying different topology files and run times. This script applies the *meyerheavy.txt* noise file to each node and then runs TOSSIM writing debugging messages to a file (which can be the screen when specified as std.out). An example invocation of this script in Linux is:

nice –n 15 ./testscript.py –o output.txt –m 4 –s 5 –t linkgain.out

This invocation uses "nice" to ensure the user can run other programs as the simulation runs (our simulations took about 3.5 hours on a 2.54 GHz Pentium 4). The switches "-m" and "-s" is the time for the simulation to run in minutes and seconds, respectively. The switch "-t" specifies the topology file created by *LinkLayerModel.*

Applying these models to our grid, the average message delivery time to the furthest node was 23.47 seconds with a standard deviation of 9.32 seconds in ten simulation trials. The typical number of hops to reach node 63 is about 9-10. A hop is defined as each time a node receives a message. If a message was received, it is assumed it was received from a hop, and, thus, the hop counter is incremented.

LIMITING HOPS

We then had to determine the best way to limit message transmissions once a message had achieved a certain number of hops (and, thus, achieved a certain distance from the base station). To accomplish testing of different hop-limitation modifications to the Drip library, a linear chain of nodes was used to reduce simulation time. A number of different library-modification scenarios were tried to limit radio transmissions, but since the Trickle algorithm is designed to not only send probe messages, but also to send full messages when the Trickle timer ended, limiting transmissions proved problematic.

Listing Four shows the parameters file used in *LinkLayerModel* that was used to create the topology used representing the chain.

Listing 4

```
% 64 node chain topology is deployed with an internode distance of 4m and symmetric
links are assumed. The resulting configuration file is:
%%%%%%%%%%%%%%%%%%%%%%%%
% Channel Parameters
%%%%%%%%%%%%%%%%%%%%%%%%
PATH_LOSS_EXPONENT = 4;
SHADOWING_STANDARD_DEVIATION = 3.2;
D0 = 2.0;
PL_D0 = 55.4;

%%%%%%%%%%%%%%%%%%%%%%%%
% Radio Parameters
%%%%%%%%%%%%%%%%%%%%%%%%
NOISE_FLOOR = -88.0;
S11 = 0;
S22 = 0;
WHITE_GAUSSIAN_NOISE = 4;
%%%%%%%%%%%%%%%%%%%%%%%%
% Topology Parameters
%%%%%%%%%%%%%%%%%%%%%%%%
% available topologies :
% - GRID (1)
% - UNIFORM (2)
% - RANDOM (3)
```

```
% - FILE (4)
TOPOLOGY = 4;
TOPOLOGY_FILE = topochain.out
```

The two key implementation files for the Drip library are *DisseminationEngine-Imp.nc* and *DisseminatorP.nc*. In Drip, two different messages are sent out, as described in chapter II, a probe message and the actual data message. The data types for these are defined as *dissemination_probe_message_t* and *dissemination_message_t*, respectively. The actual data message allows us to place our own structure in the data field of *dissemination_message_t*. The same function, *sendObject(uint16_t key),* is used to send out both probe and data messages, but since the radio receiver is instantiated as two separate components, probe messages and data messages are received in two separate events in TinyOS. The event for data messages is *Receive.receive(args)* while the event for probe messages is *ProbeReceive.receive(args)*. We wanted to ensure that nodes far from a base station would not receive messages that underwent too many hops, but as the node moved closer to a base station, it would start receiving messages with lower hop values.

To implement distance or hop-limiting, we first attempted to stop all transmissions over a certain number of hops in *DisseminationEngineImp.nc* in the send-function, written in C, *sendObject(uint16_t key)*. This proved problematic in that probe messages used the same function without the payload overhead. We would have to add code to determine the type of transmission to avoid segmentation faults since we cast the "arbitrary payload" of the received message to our data structure (*I2C_Dism_packet_t*). Indeed, the code worked until probe messages were sent, causing a segmentation fault.

Next we attempted to modify the reception code in the event *Receive.receive(args)* (where we increment the number of hops upon message reception) to not forward on messages to the disseminator cache function if they were over a certain hop limit. This did little to limit traffic since the node's Trickle timer would fire because it would never receive a "normalizing" message (that is, it would never know the nodes around it have the same message, and after a while, would transmit the data message).

After the above unsuccessful attempts, we realized that radio message reception occurred independently of whether the Disseminator component was running or not. In other words, the radio will continue receiving messages with the Trickle timer disabled and the Dissemination cache system off. We were able to limit the number of hops a message took by disabling the Dissemination engine when the node receives messages over a certain number of hops. Listing Five shows the modifications made to the event *Receive.receive(args)*. This code calls *StdControl Start()* and *Stop()* which controls turning the Dissemination engine and Trickle timers on and off, but leaves the radios receive function normally (using low-power listen). The variable *m_running* is set if the Dissemination engine is running. Using a macro of LIMIT_HOPS allows the code to be turned on and off by adding the macro in any header file.

Listing 5

```
event message_t* Receive.receive( message_t* msg,
                                  void* payload,
                                  uint8_t len ) {
<snip>
#ifdef LIMIT_HOPS
  error_t result;
#endif
<snip>
#ifdef LIMIT_HOPS
  if (((I2C_Dism_packet_t*) dMsg->data)->numberhops >= HOPS_TO_ALLOW) {
```

```
    result = call StdControl.stop();
  } else {
    if (!m_running) {
      result = call StdControl.start();
    }
  }
#endif
```

Using this code and propagation conditions, with a limit of three hops (*HOPS_TO_ ALLOW =4*), we would expect messages to go about as far as the ridge defined by nodes: 40, 35, 36, 28, 20, 13, and 5, refer to Figure 35. The processed results of 10 message transmissions from the base station is shown in Figure 36. This demonstrated that limiting hops can limit the nodes receiving messages, and thus, the distance messages propagate. The figure shows that nodes 20, 27, and 34 were the furthest nodes reached.

Sample text output (Listing Six) of a particularly long-running simulation (five minutes and 30 seconds in simulation time) is shown below. In this case, three different messages were transmitted by the base station; the counter value indicates the message version from the base station. The value in parenthesis after the word "DEBUG" is the node number of the mote processing the received value.

Listing 6

```
DEBUG (8): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (1): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (16): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (9): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (2): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (10): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (25): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (8): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (17): Value Changed: I2C Message: I2C Message counter: 1
DEBUG (1): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (2): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (9): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (16): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (11): Value Changed: I2C Message: I2C Message counter: 2
```

DEBUG (10): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (24): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (17): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (3): Value Changed: I2C Message: I2C Message counter: 2
DEBUG (9): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (2): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (17): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (3): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (1): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (8): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (10): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (25): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (11): Value Changed: I2C Message: I2C Message counter: 3
DEBUG (16): Value Changed: I2C Message: I2C Message counter: 3

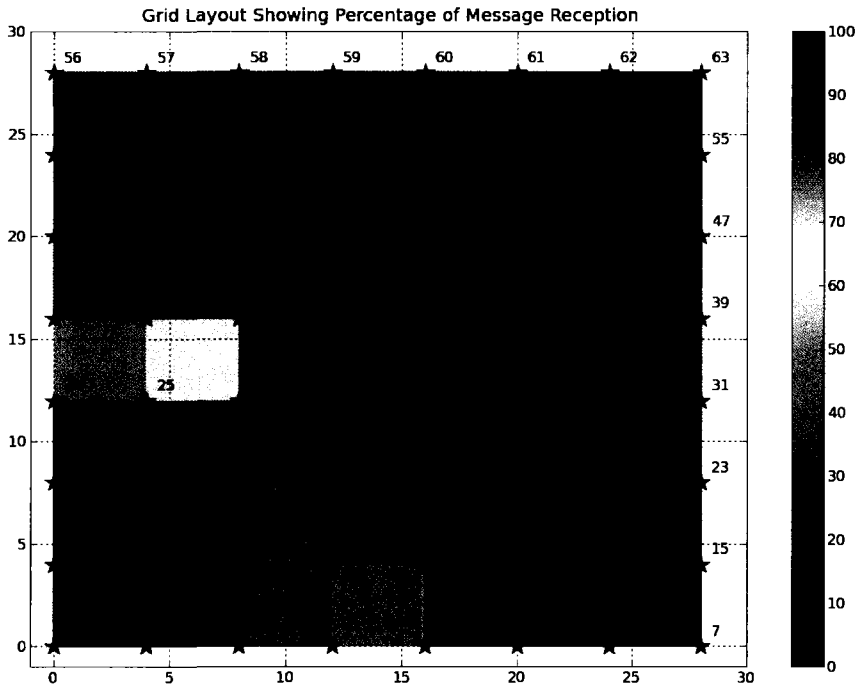


Figure 36. Percentage of messages received in ten transmissions from the base station. This is with hop-limiting enabled and set to three hops.

RESULTS AND DISCUSSION

We were successful in exploiting the capabilities of TinyOS to create hardware which can be useful to someone suffering from visual impairment. The hardware provides the capability to receive messages from a wireless mote and transcribe to a number of different formats, including: LEDs, RS-232 port, SPI port to a text-to-speech IC, and general-purpose IO driving piezo-electric or other transducers. The software in the daughter card provides the ability to operate in dense environments with minimal power consumption. The software will gracefully degrade when faced with buffer overflows, ensuring continued functionality. Additionally, a priority function is available which allows for priority delivery of messages should the need arise.

To relieve congestion and prevent messages from being received too far away from a relevant site, we were able to demonstrate the ability to limit reception distance by limiting the number of hops a message takes in the network. Our simulation showed that we can effectively limit messages received by the user based on number of hops a message undergoes while preserving the power-saving features built into the Trickle algorithm and low-power listen.

CHAPTER V

CONCLUSION AND FUTURE WORK

CONCLUSION

As the world population changes, more resources will be spent to ensure quality of life for the elderly and those suffering from physical disabilities. This dissertation provides the foundation for future work in using low-power, low-data rate wireless protocols to provide new capabilities for people with disabilities to interact with their surroundings. All designs were accomplished using open-source software to produce what is essentially open-source hardware. Those who want to capitalize on this work can load the design, make changes, and create new Gerber files to produce their own printed circuit board (PCB) to meet their design needs and expand capabilities. This chapter contains additional ideas for research as well as ways to apply current technology to provide novel capabilities to the impaired.

FUTURE WORK

Geolocation capability is proving to be a ubiquitous capability that is establishing itself as indispensible in personal devices. Significant research already is available to determine the location of sensor nodes. Knowing precise sensor locations is particularly useful when placing unattended ground sensors. For one of our purposes of developing the VINS, providing critical information to the visually impaired, it is essential that the user has relative position information with regards to obstacles and hazards. For instance, since these wireless radios are so inexpensive, we could envision base stations being placed on street signs. A blind person receiving information from a street sign base

station could receive, via Braille, distance and relative bearing (see Figure 37) as well as the street name. Different proposed geolocation schemes can be investigated for the most applicable to mobile nodes. Individual inertial navigation units such as accelerometers and micro-electrical mechanical (MEMS)-base gyros could also be examined to complement GPS. Signals from the sensor network would provide compensation for gyro drift on such a device. This rudimentary inertial navigation system (INS) would be especially useful indoors where GPS is not available.

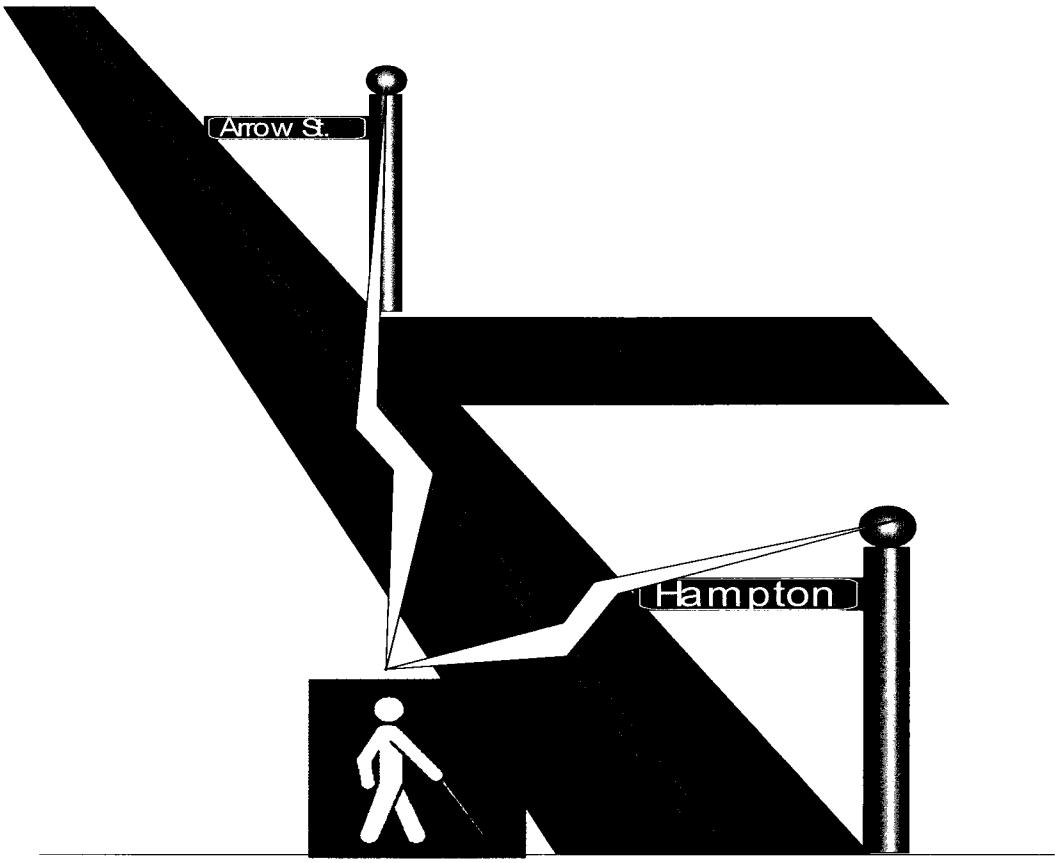


Figure 37. Notional concept of application of geolocation capability.

Current code allows a certain set of mote addresses to be reserved for base stations. Future work could include details about how multiple base stations affect receiver performance. We can determine how the node would respond in a highly saturated environment. Additionally, the base station code can be changed to detect new nodes that enter its area and throttle its message transmission rate based on new-node traffic.

Since we're mainly working with dissemination, nodes carrying the same address are likely not an issue. Nevertheless, having multiple nodes with the same identification number may be a problem with the TinyOS Drip library because the sequence number (to determine if a node has the same value) is based on node address. This issue should be investigated in more detail.

We recommend investigating different uses of piezo transducers to provide haptic feedback to a user. Future research should investigate the force required by a pin in a Braille cell to determine if the piezo driver designed can provide, not only the deflection required, but also the force. Additionally, since the transducers are piezo-electric, their repetition frequency can be varied. It's possible that a vibrating pin may have a similar effect as a stationary one. An experiment can be designed to determine if a piezo driver running close to resonance might be able to provide sufficient feedback to a user while minimizing energy required for large deflection.

We suggest that a change to the MAC protocol used can be investigated to determine its impact on message delivery and energy consumption. As discussed in the third section of Chapter II, the X-MAC actually is a more efficient MAC than the B-MAC used because it puts information such as the destination address in the preamble. The Drip Dissemination protocol sends probe messages with a short "sequence number"

representing the current state of data on the particular mote. A version of the X-MAC could be investigated where the preamble contains the sequence number instead of destination address. This could dramatically improve our dissemination capability. For future work, we could implement a modified X-MAC to include the node's data-state in the preamble where the probing nodes would examine the preamble, not just as a CCA, but also to change the state of its trickle timer. This procedure would be used instead of "snooping" and could significantly reduce energy consumption since shorter preambles would be used than with the B-MAC.

Additionally, we did not use the "priority" field although we discussed how it could aid in delivery of hazard warnings or important messages. Additional work could implement a priority-based scheme where the Trickle protocol is modified to ensure more robust message delivery. When a high-priority message is in the queue, the τ_h value can be reduced to increase the number of transmissions. We can investigate how priority changes delivery time by varying trickle values in real time.

The current design actually uses two microcontrollers: the Atmel processor on the mote and the MSP430 on the daughter card. If the TI CC2530 radio is used, three microcontrollers are used since it contains an 8051 core. Although these processors are designed for low-power applications, we believe their functionality can be consolidated into a single, more powerful processor. Additionally, the text-to-speech chip could easily be replaced with a processor running Linux and resident text-to-speech software. For instance, the Audio Desktop Reference Implementation and Networking Environment (ADRIANE) is now being implemented as an accessibility add-on [50].

A single Analog Devices Blackfin DSP with and necessary CODEC and audio amplifier would cost about a quarter of the cost of the Winbond WTS701. This DSP has more than enough processing power to perform 100% of all functions of the current system while running uCLinux [51]. It also has a sleep mode allowing the processor to sleep with a mere 0.8 V supply. If the X-MAC discussed above was implemented in this embedded Linux solution significant cost savings could be achieved. Additionally, the necessary drivers would need to be created to interface with standard 802.15.4 radios available.

REFERENCES

[1] Research to Prevent Blindness, http://www.rpbusa.org/rpb/eye_info/low_vision, last visited 30 Oct 09.

[2] M.M. Whiteside, M.I. Walhagen, and E. Pettengill, "Sensory Impairment in Older Adults," *American Journal of Nursing*, vol.106, no. 11, Nov. 2006, pp. 52-61.

[3] F.L. Ferris and J. M. Tielsch, "Blindness and Visual Impairment: A Public Health Issue for the Future as Well as Today," *Arch Ophthalmol.*Vol. 122, Apr. 2004, pp. 451-452.

[4] IEEE Standard 802.15.4-2006, "Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)," 2006.

[5] P. Gupta, R. Gray, and P. Kumar, "An Experimental Scaling Law for Ad Hoc Networks," *Modeling and design of wireless networks.* Conference, Denver CO, *ETATS-UNIS2001*, Vol. 4531, Aug 2001, pp. 14-21.

[6] http://www.bbwexchange.com/meshnetworks/meshnetwork_performance_analysis.asp, last visited Jan 2009.

[7] S. Ni, Y. Tsing, Y. Chen, and J. Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," *Proc. Eighth ACM Int'l Conf. Mobile Computing and Networking (MobiCom)*, 1999.

[8] P. Levis, E. Brewer, D. Culler, D. Gay, S. Madden, N. Patel, J. Polastre, S. Shenker, R. Szewczyk, and A. Woo, "The Emergence of a Networking Primitive in Wireless Sensor Networks," *Communications ACM*, vol. 51, no. 7, July 2008, pp. 99-106.

[9] English Braille American Addition, developed under the Braille Authority of North America, (BANA), American Printing House for the Blind, Louisville, KY, 1994 (Rev. 2007).

[10] http://www.gpleda.org/ , last visited Jan 2009.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System architecture directions for networked sensors", *Architectural Support for Programming Languages and Operating Systems, ASPLOSIX*, vol. 11, 2000, pp. 93-104.

[12] P. Levis, Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," *First*

Symposium on Networked Systems Design and Implementation (NSDI '04), 2004, pp. 15-28.

[13] B. Lo, S. Thiemjarus, R. King, and G. Yang, "Body Sensor Network – A Wireless Sensor Platform for Pervasive Healthcare Monitoring," *3rd International Conference on Pervasive Computing*, May 2005.

[14] B. Lo and G. Yang, "Architecture for Body Sensor Networks," *Perspective in Pervasive Computing*, Oct 2005, pp. 23-28.

[15] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic, "ALARM-NET: Wireless Sensor Networks for Assisted-Living and Residential Monitoring," *Technical Report CS-2006-13*, Department of Computer Science, University of Virginia (no date provided).

[16] K. Li, P. Baudisch, W. Griswold, and J. Hollan, "Tapping and Rubbing: Exploring New Dimensions of Tactile Feedback with Voice Coil Motors," *UIST* Oct. 2008.

[17] I. Poupyrev, S. Maruyama, and J. Rekimoto, "Ambient touch: designing tactile interfaces for handheld devices," *Proc. UIST '02*, 2002, pp. 51–60.

[18] J. Luk, J. Pasquero, S. Little, K. MacLean, V. Levesque, and Hayward, V. "A role for haptics in mobile interaction: initial design using a handheld tactile display prototype," *Proc. CHI '06*, 2006, pp. 171–180.

[19] J. C.Lee, P. H. Dietz, D. Leigh, W. S. Yerazunis, and S. E. Hudson, "Haptic pen: a tactile feedback stylus for touch screens," *Proc UIST '04*, 2004, pp. 291–294.

[20] T. Evreinova, "Alterative Visualization of Textual Information for People with Sensory Impairment," Dissertation, Department of Computer Sciences, University of Tempere, 2005.

[21] Crossbow MICAz-Based ZigBee and WiFi Coexistance, Crossbow. http://www.xbow.com, last visited Nov 2009.

[22] Zigbee Alliance, Inc., "Status of 802.15.4 and Zigbee," May 2004.

[23] J. Hill and D. Culler, "MICA: A Wireless Platform for Deeply Embedded Networks," *IEEE Micro*, vol. 22, no. 6, Nov./Dec. 2002, pp. 12-24.

[24] J. Polastre, J. Hill, and D. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks," *Sensys '04*, (Nov 2004).

[25] M. Buettnew, G. Yee, E. Anderson, and R. Han, "X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks," University of Colorado at Boulder, *Technical Report CU-CS-1008-06*, 2006.

[26] P. Levis, N. Patel, S. Shenker, D. Culler, "Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," UC Berkeley, *Tech Report Number CSD-03-1290*, no date provided.

[27] D. De Couto, D. Aguayo, J. Bicket, R. Morris, "A High-throughput Path Metric for Multi-hop Wireless Routing," *MobiCom '03*, Sep. 2003.

[28] A. Dunkels, B. Gronvall, T. Voigt, "Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors," Swedish Institute of Computer Science, 2004.

[29] Sohraby, Kazem, Daniel Minoli, and Taieb Znati. "Medium Access Control Protocols for Wireless Sensor Networks," *Wireless Sensor Networks: Technology, Protocols, and Applications*. John Wiley & Sons. 2007.

[30] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient Memory Safety for TinyOS," *Proceedings of the 5th international conference on Embedded networked sensor systems,* Sydney, Australia, 2007, pp. 205-218.

[31] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, "The Mote Revolution: Low Power Wireless Sensor Network Devices," *Hot Chips 2004*, Aug. 22-24, 2004.

[32] Crossbow MPR-MIB Users Manual, Revision B, PN: 7430-0021-07, June 2006.

[33] Texas Instruments, "MSP430x161x Datasheet," SLAS368E, rev. Aug 2006.

[34] Energizer, "Product Datasheet for E91," http://data.energizer.com.

[35] Texas Instruments, "TPS715xx Datasheet, SLVS338P," rev. Nov. 2008.

[36] Texas Instruments, "TPS62200 Datasheet, SLVS417E," rev. May 2006.

[37] Philips Semiconductor, "The I^2C Bus Specification," document order number 9398 393 40011, Version 2.1, January 2000.

[38] J. Gardner, "Braille, Innovations, and Over-specified Standards," Department of Physics, Oregon State University, 2005.

[39] J.C. Stevens, E. Foulke, and M. Patterson, "Tactile Acuity, Aging, and Braille Reading," *Journal of Experimental Psycology: Applied*, vol. 2, no. 2, 1996, pp. 91-106. From site, http://www.braille.org/papers/.

[40] E. Foulke, "Investigative Approaches to the Study of Braille Reading," *Journal of Visual Impairment and Blindness, American Foundation for the Blind*, vol.73, no. 8, Oct.1979, pp. 298-308.

[41] P. Flanigan and E. Joslin, "Patterns of Response in the Perception of Braille Configurations," *Outlook*, Oct. 1969.

[42] BANA Braille Codes Update, Developed under the sponsorship of the Braille Authority of North America, Effective January 1, 2008, www.brailleauthority.org, 2007.

[43] S. Katzir, *The Beginnings of Piezoelectricity: A Study in Mundane Physics*, Springer, 1st Ed. 2006.

[44] W. Cady, *Piezoelectricity*; *An Introduction to the Theory and Applications of Electromechanical Phenomena in Crystals,* New York, Dover Publications 1964.

[45] Piezo Systems, Inc, http://www.piezo.com , last visited Jul. 2009.

[46] http://ngspice.sourceforge.net/index.html - last visited Aug. 2009.

[47] http:// sourceforge.net/projects/ kjwaves - last visited Aug. 2009.

[48] "TOSSIM Tutorial," http://docs.tinyos.net/index.php/TOSSIM, last visited Oct. 2009.

[49] Hyung Lee, A. Cerpa, and P. Levis, "Improving Wireless Simulation through Noise Modeling," *IPSN*, Apr. 2007, pp. 21-30.

[50] K. Knopper, "The Adriane Desktop for the Sight Impaired," Linux*Pro Magazine*, vol. 92, July 2008, pp. 59-63.

[51] http://www.uclinux.org/, last visited Oct. 2009.

APPENDICES

DAUGHTER CARD SCHEMATICS

NOTES:

Px.0 to Px.7 – Switch to port function, output direction for low power

Jumper next to JTAG connector:

If no external power = connect to pin 2–3 – which connects JTAG pwr to board

If external power is available connect pin 1–2 – which connects pin 4 to ext pwr

ADP1611 RT set to 700kHz to achieve highest duty cycle to reach high voltage

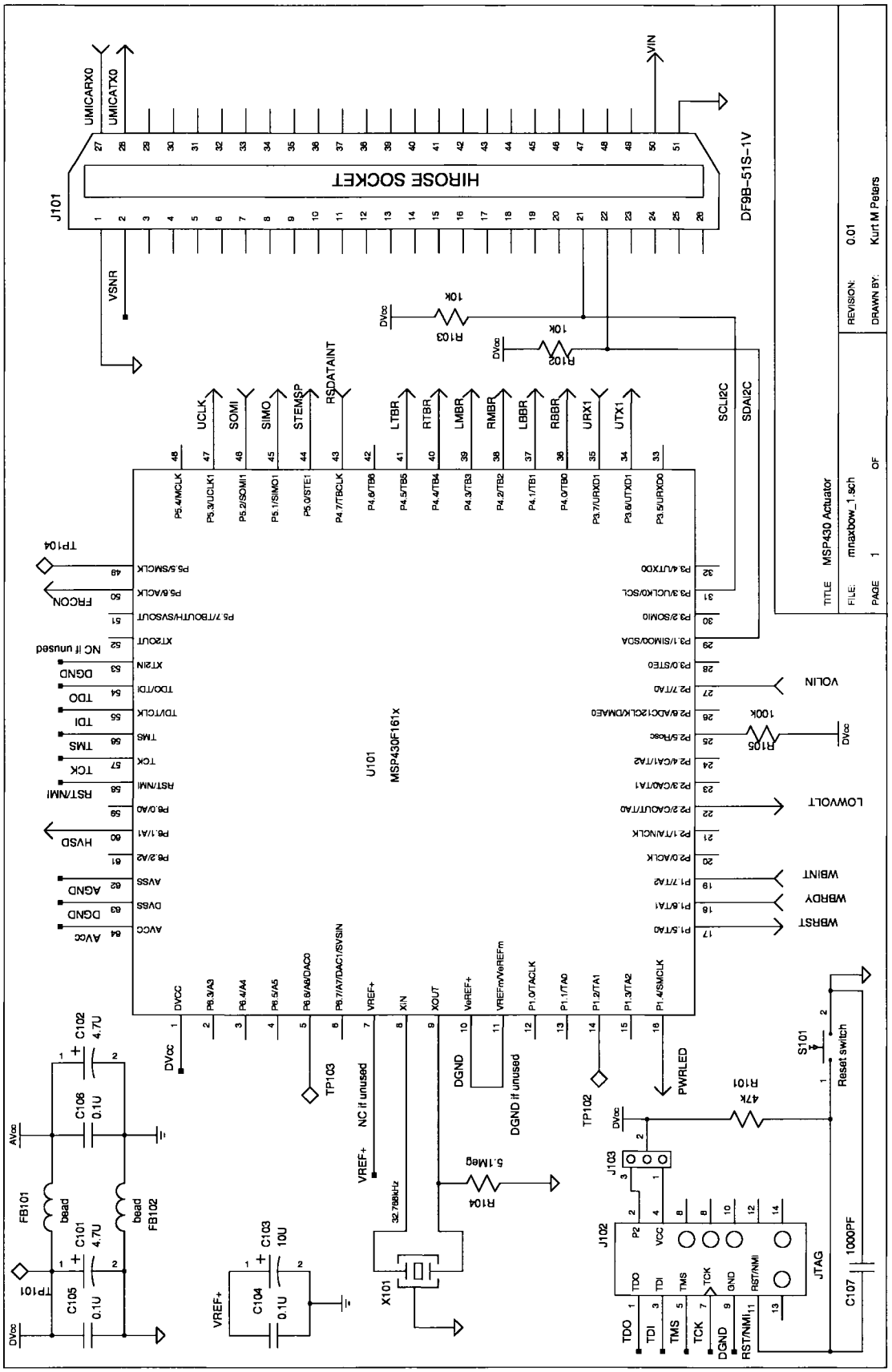
-- datasheet page 10.

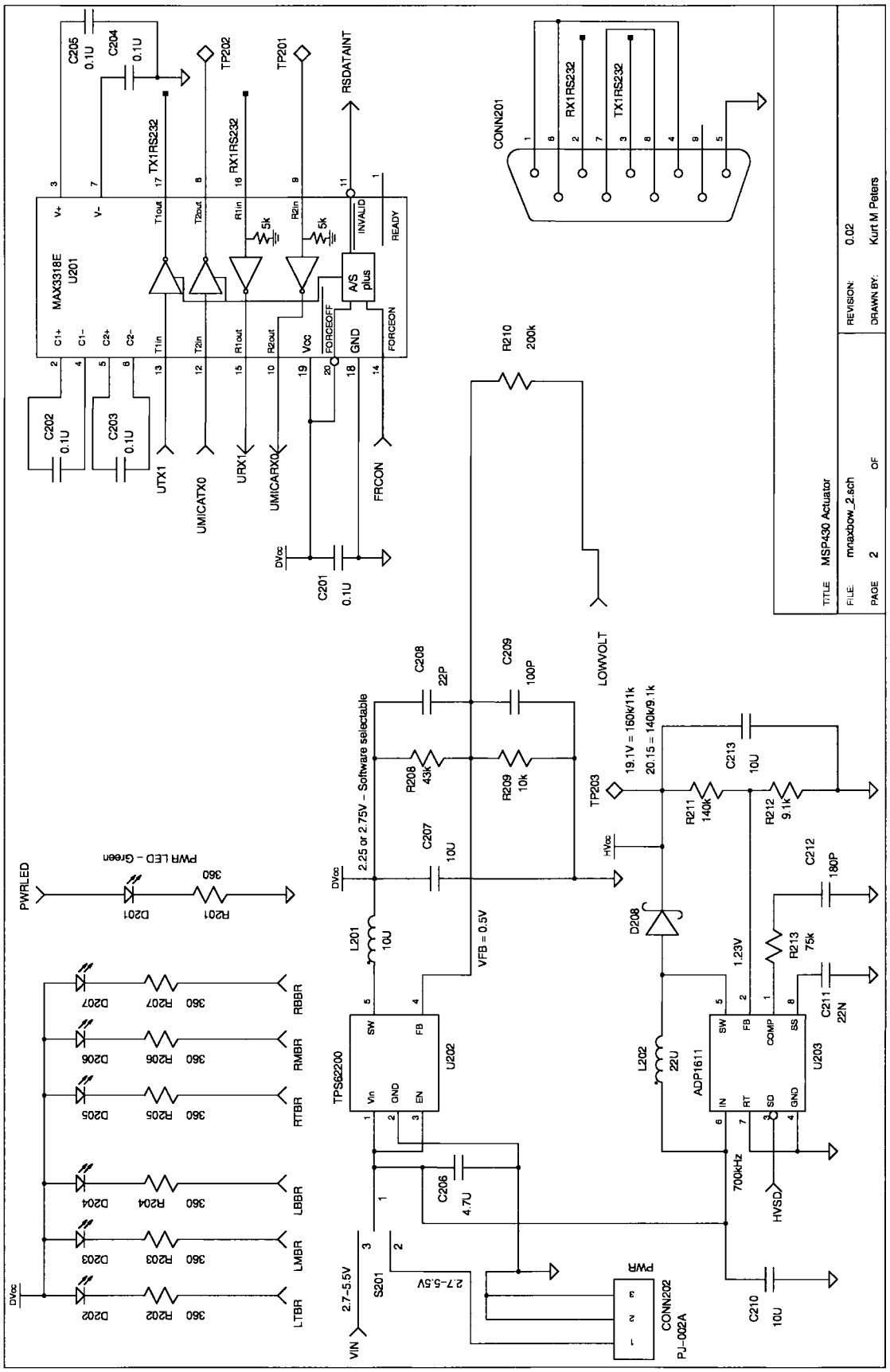
On WINBOND chip pins 11 and 12 must be independently bypassed and not connected together.

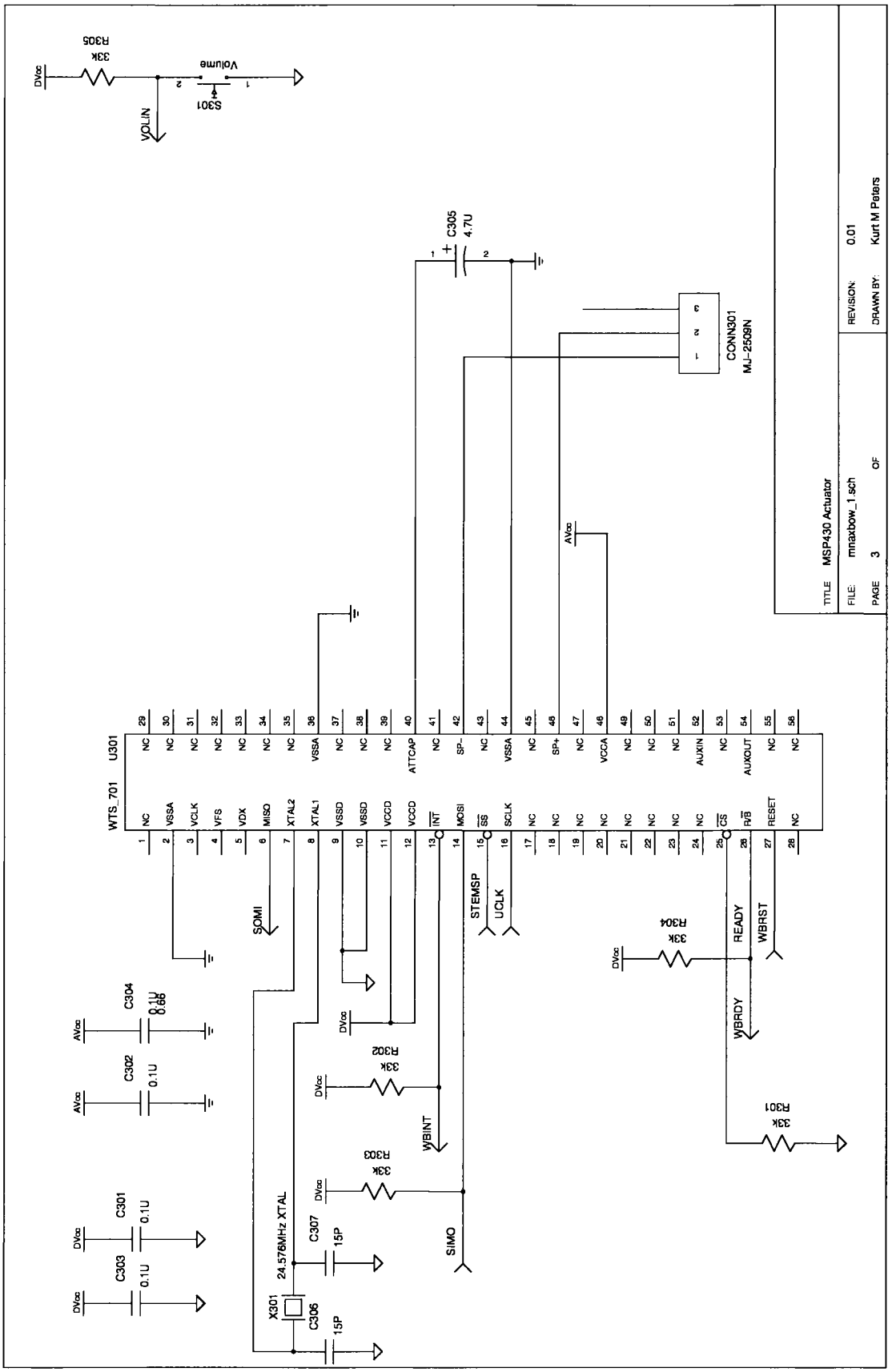
5.1MOhm Res added to XOUT of MSP430 based on Users guide

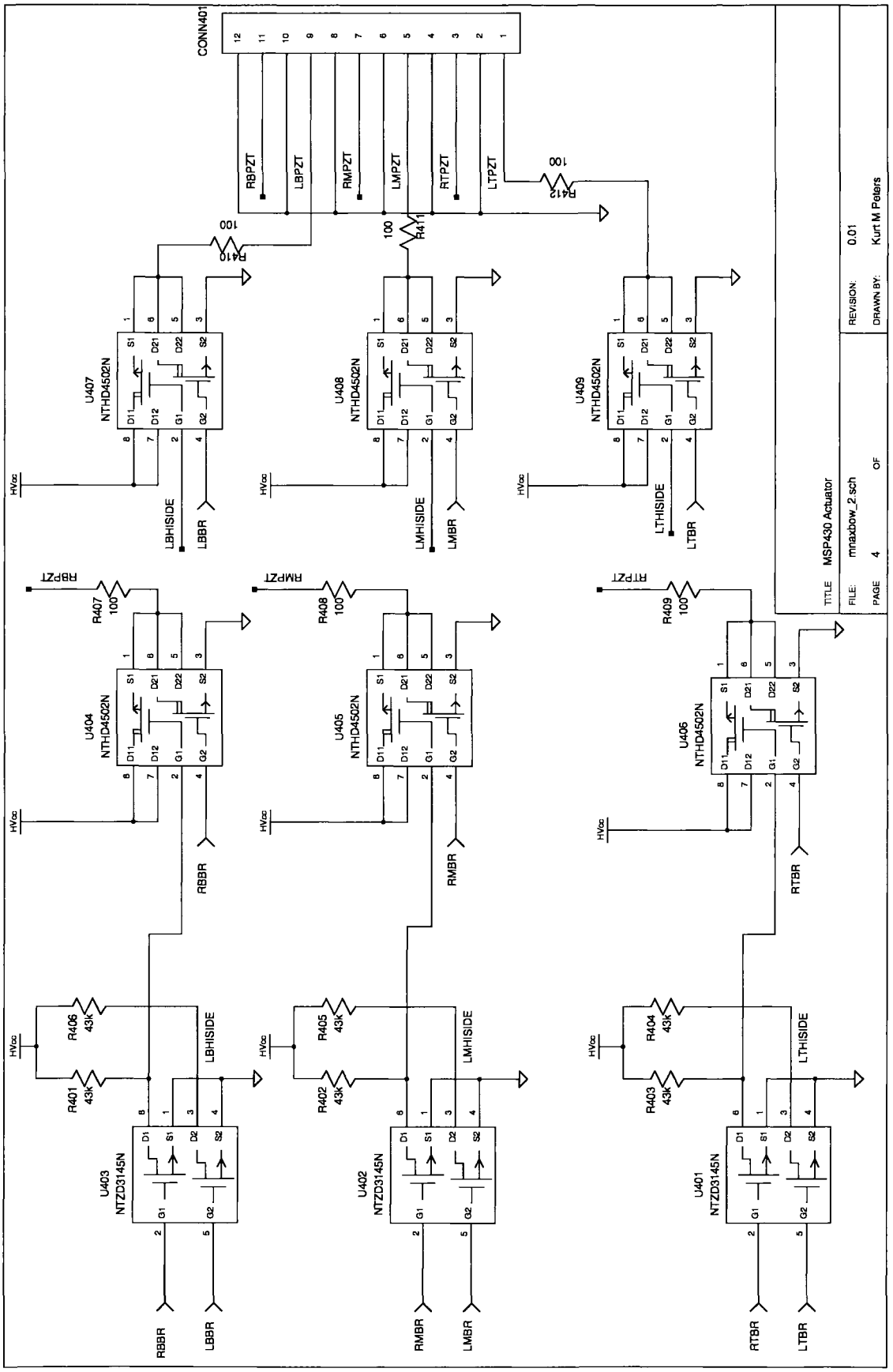
and expected VCC < 2.5 V.

TITLE	MSP430 Actuator		
FILE:	mnaxbow_0.sch	REVISION:	0.01
PAGE	0 OF	DRAWN BY:	Kurt M Peters


J101
HIROSE SOCKET
DF9B-51S-1V
UMICARX0
UMICATX0
VIN
VSNR
R103
10k
R102
10k
RSDATAINT
UCLK
SOMI
SIMO
STEMSP
LTBR
RTBR
LMBR
RMBR
LBBR
RBBR
URX1
UTX1
SCLI2C
SDAI2C
U101
MSP430F161x
TP104
FRCON
NC if unused
DGND
TDO
TDI
TMS
TCK
RST/NMI
HVSD
AGND
DGND
AVcc
VOLIN
R105
100k
LOWVOLT
WBINT
WBRDY
WBRST
PWRLED
S101
Reset switch
R101
47k
TP102
J103
J102
JTAG
TDO
TDI
TMS
TCK
DGND
RST/NMI
C107
1000PF
TP103
NC if unused
VREF+
X101
32.768kHz
R104
5.1Meg
DGND
DGND if unused
C102
4.7U
C106
0.1U
FB101
bead
FB102
bead
C101
4.7U
C105
0.1U
TP101
C103
10U
C104
0.1U
TITLE
MSP430 Actuator
FILE:
mnaxbow_1.sch
PAGE
1
OF
REVISION:
0.01
DRAWN BY:
Kurt M Peters


MAX3318E
U201
C205 0.1U
C204 0.1U
C202 0.1U
C203 0.1U
C201 0.1U
TP202
TP201
RSDATAINT
TX1RS232
RX1RS232
5k
INVALID
READY
A/S plus
FORCEOFF
FORCEON
Vcc
GND
DVcc
UTX1
URX1
UMICATX0
UMICARX0
FRCON
CONN201
R210 200k
LOWVOLT
C208 22P
C209 100P
R208 43k
R209 10k
2.25 or 2.75V - Software selectable
C207 10U
TP203
19.1V = 160k/11k
20.15 = 140k/9.1k
R211 140k
R212 9.1k
C213 10U
HVcc
C212 180P
D208
L201 10U
VFB = 0.5V
TPS62200
U202
Vin
GND
EN
SW
FB
L202 22U
ADP1611
U203
IN
RT
SD
GND
COMP
SS
1.23V
R213 75k
C211 22N
700kHz
HVSD
C206 4.7U
C210 10U
S201
2.7-5.5V
VIN
PWR
CONN202
PJ-002A
PWRLED
PWR LED - Green
D201
R201 360
LTBR
LMBR
LBBR
RTBR
RMBR
RBBR
D202
D203
D204
D205
D206
D207
R202 360
R203 360
R204 360
R205 360
R206 360
R207 360
TITLE MSP430 Actuator
FILE: mnaxbow_2.sch
PAGE 2 OF
REVISION: 0.02
DRAWN BY: Kurt M Peters


R305
33k
DVcc
Volume
S301
VOLIN
C305
4.7U
CONN301
MJ-2509N
U301
WTS_701
AVcc
DVcc
SOMI
STEMSP
UCLK
READY
WBRST
WBRDY
WBINT
SIMO
R304
R302
R303
R301
C304
C302
C301
C303
C307
C306
X301
24.576MHz XTAL
15P
0.1U
TITLE MSP430 Actuator
FILE mnaxbow_1.sch
PAGE 3 OF
REVISION: 0.01
DRAWN BY: Kurt M Peters

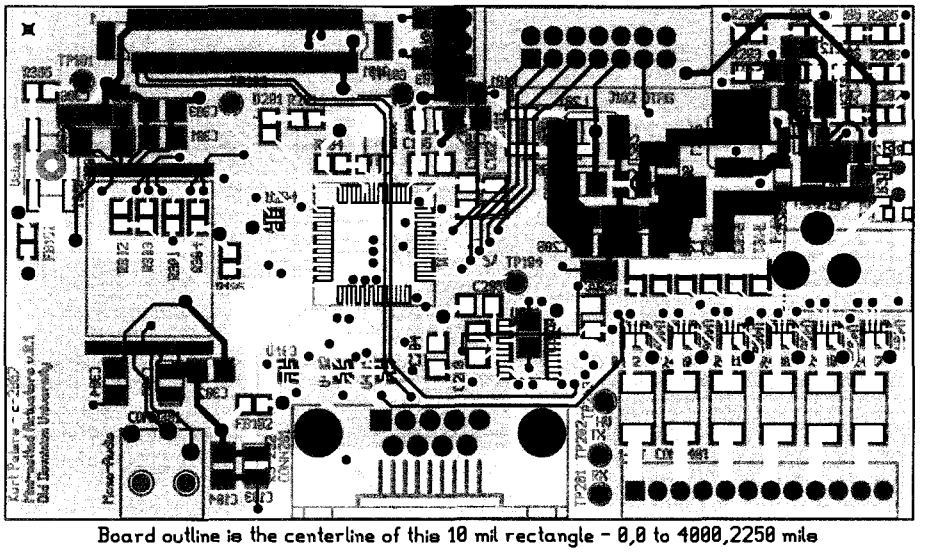

CONN401
RBPZT
LBPZT
RMPZT
LMPZT
RTPZT
LTPZT
R410
R411
R412
100
U407
NTHD4502N
U408
NTHD4502N
U409
NTHD4502N
HVcc
LBHISIDE
LMHISIDE
LTHISIDE
LBBR
LMBR
LTBR
R407
R408
R409
U404
NTHD4502N
U405
NTHD4502N
U406
NTHD4502N
RBBR
RMBR
RTBR
R406
43k
R405
43k
R404
43k
R401
43k
R402
43k
R403
43k
U403
NTZD3145N
U402
NTZD3145N
U401
NTZD3145N
TITLE
MSP430 Actuator
FILE:
mnaxbow_2.sch
PAGE
4
OF
REVISION:
0.01
DRAWN BY:
Kurt M Peters

DAUGHTER CARD GERBER

There are 10 different drill sizes used in this layout, 188 holes total

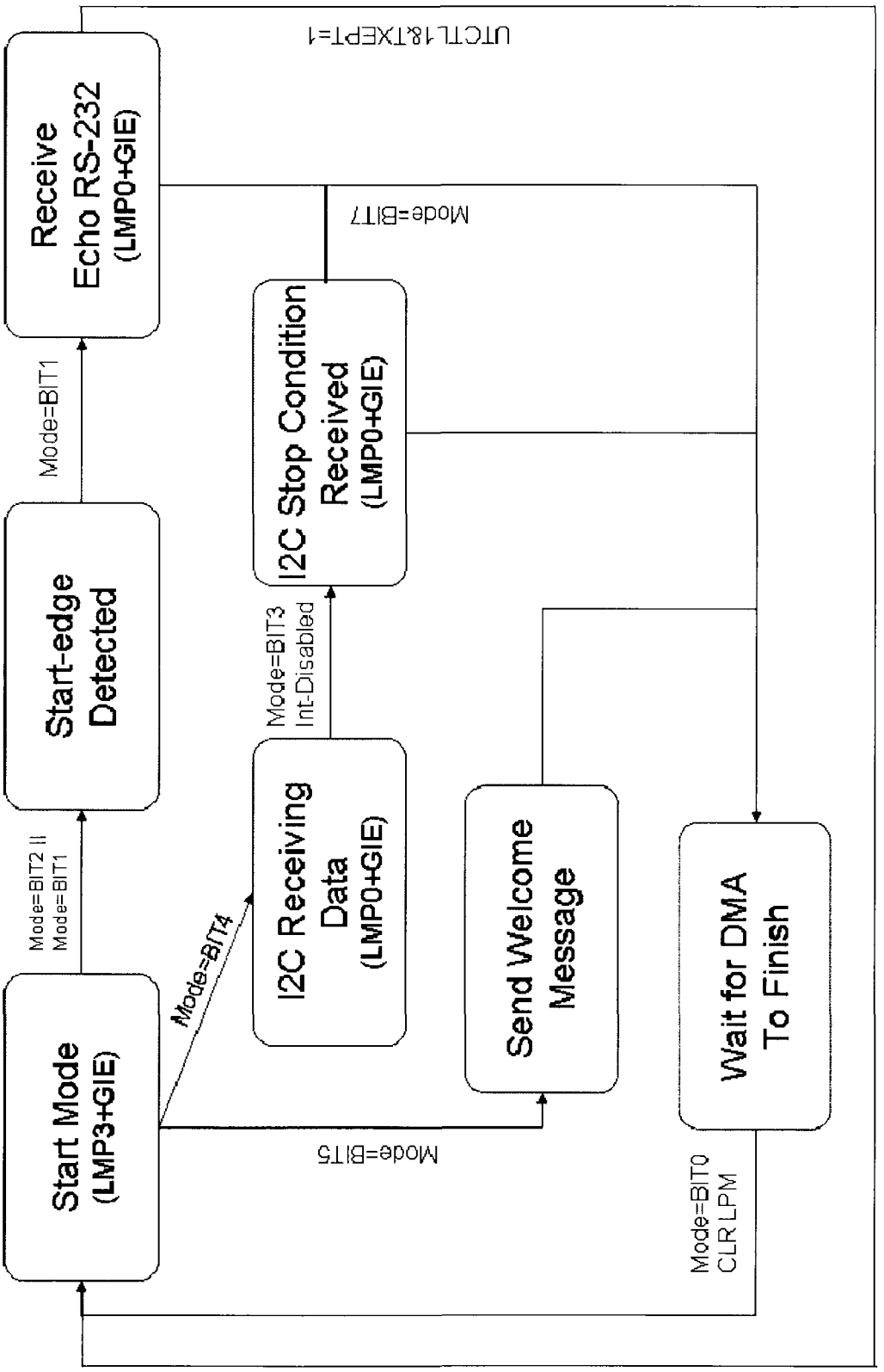
Symbol	Diam. (Inch)	Count	Plated?
▾	0.020	100	YES
+	0.035	2	NO
×	0.035	30	YES
•	0.043	3	YES
□	0.046	36	YES
▼	0.052	9	YES
✚	0.063	2	NO
✖	0.110	1	NO
◎	0.120	2	YES
⧈	0.125	2	YES
⛉	0.140	1	YES

Title: (unknown) - Fabrication Drawing
Author: Kurt Peters
Date: Tue Feb 13 15:13:44 2007 UTC
Maximum Dimensions: 4000 mils wide, 2250 mils high



Board outline is the centerline of this 10 mil rectangle - 0,0 to 4000,2250 mils

MSP430 SOURCE CODE



```
/*****************************************************************
*  MSP430F1611 - Braille translator - interface to MicaZ
*
*  Description; Toggles LED at 0.25 sec interval
*  DCO frequency used for SMCLK and set to 4 MHz by TI routine.
*  I2C is used to communicate with Mote in Slave mode only
*  RS-232 is used to echo input and outputs everything to Braille
*  SPI is used to communicate with TTS chip
*  ACLK = 32768 = TACLK, MCLK = SMCLK  = DCO~ 4000k
*
*
*           MSP430F169/11
*         ---------------
*     /|\|              XIN|-
*      | |                 |
*      --|RST          XOUT|-
*        |                 |
*        |             P1.4|-->LED
*        |             P2.2|-->LOWVOLT
*
*  Kurt Peters
*  Old Dominion University
*  February 2007
*  Built with CCE for MSP430 Version: 2.0.0.21
*****************************************************************/

#include "msp430x16x.h"
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include "DCO_Library.h"

#define LowByte(i)    ( (uint8_t) i )
#define HiByte(i)    ( (uint8_t) ( ((uint16_t) i) >> 8) )
/*
 * This is the default payload size of a TinyOS 2.x packet
 */
#define TOSH_DATA_LENGTH 28
#define BRAILLE_BUFFER_LENGTH 0x80*4
#define BRAILLE_BUFFER_MASK 0x7F
/* This is the rate the Braille will be written */
#define BRAILLE_WRITE_TIME 8192
#define SPACE_SYMBOL 0x3F
#define SLAVE_I2C_ADDRESS 0x0048          // change to 0x24 to account
for a bug in TinyOS 2.0.2 for address 0x48
#define DELTA 1125                                    // 4608000 Hz

//TI_DCO_4MHZ               // target DCO = 256*(4096) = 1048576 |
2097152
                                        // the 4.8MHz allows 400kHz
I2C Master (if required)
typedef struct
{
  uint8_t first;
  uint8_t second;
} brailleStruct;
```

```
/*
 *  Prototypes for int services
 *  Timer A ISR
 */
__interrupt void Timer_A(void);

// UART1 RX/TX ISR
__interrupt void usart1_rx(void);

// I2C Module ISR
__interrupt void I2C_ISR(void);

//DMA Interrupt
__interrupt void DAC_DMA_ISR(void);

// watchdog Interrupt
__interrupt void watchdog_timer(void);

inline void configUART1forRS232(void);
inline void configDMAToSendUART1(void);
inline void configUART0forI2C(void);

inline void forceRS232On(void);
inline void forceOnRS232Off(void);
inline void setDVoltage(uint8_t DCOCTLVal, uint8_t DCOBCVal, uint8_t
P2Val);
//inline void setHighVoltage(void);
//inline void setLowVoltage(void);
brailleStruct convertASCIIToHW(uint8_t ASCIIin);
inline void sendUART1Msg(const char *anyMSG, size_t length);

uint16_t numberofseconds = 0;
volatile uint8_t mode = 0;
uint8_t braillehalf = 0;
/*
 * Definition of modes
 * bit      - definition
 * 0  - DMA Finished
 * 1  - RS-232 Receive in progress
 * 2  - RS-232 Start-edge Detect
 * 3  - I2C Receive finished and stop condition received
 * 4  - I2C Receiving Character
 * 5  - Send Welcome Message
 * 6  - NA
 * 7  - overflow error
 **/

//brailleStruct
static char longBrailleBuffer[BRAILLE_BUFFER_LENGTH];
static char I2CBuffer[BRAILLE_BUFFER_LENGTH];
/*
 * index is the current position to write
 * out the Braille code
 * size is the number of bytes left
 *
 * size can be less than index when the
 * buffer becomes circular
```

```
 */
uint16_t indexBBWrite=0;
uint16_t indexBBRead=0;
uint16_t sizeBB=0;
uint16_t sizeI2CB=0;

/*
 *  These constants are used to convert from the
 * standard Braille cell (1-6) to a HW version.
 * In our case, we read from rt to left starting from
 * the bottom -- because our Port 4 is wired that
 * way to the LEDs
 */
static const uint8_t andMatrix[] = {0x02,0x10,0x04,0x20,0x08,0x40};
static const int8_t shiftAmount[] = {-4,0,-1,3,2,6};
//static const char errorMSGDCO[] = {"Error with DCO\r\n"};
static const char repeatMSG[] = {"\r\nLocal still alive.\r\n"};
static const char errorMSGBB[] = {"\r\nError: Buffer Overflow!\r\n"};
static const char welcomemessage[] = {"\r\nMNS Braille Receiver\r\nver
0.1\r\n"};



/*
 *  BRAILLE Translation
 *  Goal is to just add '32' to the ASCII value
 *  Note: The function convertBToHW() is used to
 *        convert the cell to a HW representation
 *        From BANA - Braille Authority of North America
 *        English Braille American Edition 1994
 *        Revised 2002
 */
static const uint16_t displayDigits [] = {
                      0x0000,   // space
                              0x002C,       // exclamation
                              0x004C,       // " opening quote 3F34
                              0x1078,   // # pound or "number follows"
2F28
                              0x0064,   // $ dollar 3F32
                              0x0052,   // % percent - note two-
characters needed 3305
                              0x005E,   // & uses an and instead of
amperstamp 04
                              0x0008,   // ' apostrophe 3F3D
                              0x006C,   // ( 3F30
                              0x006C,   // )
                              0x2828,   // * - for math this could be
2F28 3939
                              0x1058,   // + dot 4 and 3-4-6 2F2C
                              0x0004,   // , 3F37
                              0x0048,   // - 3F3C
                              0x0064,   // . 3F32
                              0x0018,   // / 3F2D
                              0x7834,   // 0 2823
                              0x7802,   // 1 281F
                              0x7806,   // 2 2817
                              0x7812,   // 3 280F
```

```
0x7832,   // 4 280B
0x7822,   // 5 281B
0x7816,   // 6 2807
0x7836,   // 7 2803
0x7826,   // 8 2813
0x7814,   // 9 2827
0x0024,   // : 3F33
0x000C,   // ; 3F35
0x0046,   // < 3F16
0x007E,   // =
0x0038,   // > 3F29
0x004C,   // ? 3F34
0x0010,   // @ 1F21
0x4002,   // A 3E1F
0x4006,   // B 3E17
0x4012,   // C 3E0F
0x4032,   // D 3E0B
0x4022,   // E 3E1B
0x4016,   // F 3E07
0x4036,   // G 3E03
0x4026,   // H 3E13
0x4014,   // I 3E27
0x4034,   // J 3E23
0x400A,   // K 3E1D
0x400E,   // L 3E15
0x401A,   // M 3E0D
0x403A,   // N 3E09
0x402A,   // O 3E19
0x401E,   // P 3E05
0x403E,   // Q 3E01
0x402E,   // R 3E11
0x401C,   // S 3E25
0x403C,   // T
0x404A,   // U
0x404E,   // V
0x4074,   // W
0x405A,   // X
0x407A,   // Y
0x406A,   // Z
0x406C,   // [ or 2-4-6
0x0066,   // backslash
0x6C08,   // ] or 1-2-4-5-6
0x0030,   // ^
0x7070,   // _
0x7010,   // `
0x0002,   // a
0x0006,   // b
0x0012,   // c
0x0032,   // d
0x0022,   // e
0x0016,   // f
0x0036,   // g
0x0026,   // h
0x0014,   // i
0x0034,   // j
0x000A,   // k
0x000E,   // l
```

```
                                    0x001A,    // m
                                    0x003A,    // n
                                    0x002A,    // o
                                    0x001E,    // p
                                    0x003E,    // q
                                    0x002E,    // r
                                    0x001C,    // s
                                    0x003C,    // t
                                    0x004A,    // u
                                    0x004E,    // v
                                    0x0074,    // w
                                    0x005A,    // x
                                    0x007A,    // y
                                    0x006A,    // z
                                    0x7054,    // {
                                    0x7066,    // |
                                    0x7076,    // {
                                    0x7030     // ~
                                    };

//static char outMessage[] = { "Hello World\r\n" };
//char outMessage[] = "test RS-232";

void main(void) {

      enum States {
            Start_Mode,
            I2C_Receiving_Data,
            I2C_Stop_Received,
            Wait_For_DMA_Finish,
            Start_Edge_Received,
            Receive_Echo_RS232,
            Send_Welcome_Message};
      uint16_t result;
      uint16_t DCOerrors;
      uint8_t DCOCTLLowVolt, DCOCTLHighVolt;
      uint8_t DCOBC1LowVolt, DCOBC1HighVolt;

      enum  States current_state;
//    enum  States last_state = Start_Mode;
   WDTCTL = WDT_ARST_1000 + WDTHOLD;
  //WDTCTL = WDTPW + WDTHOLD;              // Stop WDT
  _disable_interrupts();
  P1DIR = BIT0+BIT1+BIT2+BIT3+BIT4+BIT5;     // P1.0-5 output
  P2DIR = BIT0+BIT1+BIT2+BIT3+BIT4+BIT6;    // P2.0-4,6 output
  P3DIR = BIT0+BIT2+BIT4+BIT6;  // P3.0,2,4-6 outputs, 1 and 3 and 5
are inputs for I2C
  P4DIR = BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6;    // P4.0-6 output
  P5DIR = BIT0+BIT1+BIT3+BIT4+BIT5+BIT6+BIT7;     // P5.0,1,3-7 output
  P6DIR = 0xFF;           // P6.0-7 output
      forceOnRS232Off();
      /* Initialize the settings for functions */
      DCOCTLLowVolt=0xDA;
      DCOBC1LowVolt=0x86;
      DCOCTLHighVolt=0xBB;
      DCOBC1HighVolt=0x86;
```

```
   /* Configure DCO at both low and high voltage modes
    * and record the values
    * Set the SMCLK to P5.5 to see clock on O-scope
    * Should be TP104 on board
    */
      P5SEL |= BIT5;
      /* This sets the DCO to use the external ROSC
       * Resistor = 140k nom
       */
      P2SEL |= BIT5;
      BCSCTL2 |= DCOR;    // Rosc

      /* Do high voltage first */
    //setHighVoltage();
      setDVoltage(DCOCTLHighVolt, DCOBC1HighVolt, (P2OUT & (~BIT2)));

      DCOerrors=0;
   // Set HW Clock to right Freq using TI asm routine
      result = TI_SetDCO(DELTA);
      if( result != TI_DCO_NO_ERROR ) {    // returned result was in
error
          DCOerrors++;
      }
      DCOCTLHighVolt=DCOCTL;
      DCOBC1HighVolt=BCSCTL1;

      /* Now do low voltage */
      //setLowVoltage();
      setDVoltage(DCOCTLLowVolt, DCOBC1LowVolt, (P2OUT | BIT2));

      DCOerrors=0;
   // Set HW Clock to right Freq using TI asm routine
      result = TI_SetDCO(DELTA);
      if( result != TI_DCO_NO_ERROR ) {    // returned result was in
error
          DCOerrors++;
      }
      DCOCTLLowVolt=DCOCTL;
      DCOBC1LowVolt=BCSCTL1;


      /* Initialize all ports used */
   configUART1forRS232();
   configDMAToSendUART1();
   configUART0forI2C();


    /* Turn off Braille outputs */
    P4OUT = 0xFF;
    CCTL0 = CCIE;             // CCR0 interrupt enabled
    CCR0 = 32768>>1;       // after 1/2 second first CCR0 will occur
    TACTL = TASSEL_1 + MC_2;              // ACLK, contmode <- ACLK is
32768 crystal
 //setLowVoltage();
      mode =0;
      sizeI2CB=1;
      I2CBuffer[0] = 60;
```

```
      current_state = Start_Mode;
      forceRS232On();                                  // Force the RS-
232 Max3318E on
   /* Set HW Clock to right Freq using TI asm routine
   if (DCOerrors>0){
            result = TI_SetDCO(DELTA);
            if( result != TI_DCO_NO_ERROR ) {     // returned result was
in error
            sendUART1Msg(errorMSGDCO, sizeof errorMSGDCO-1);
//          DCOerrors++;
            }

                                 if( result == TI_DCO_SET_TO_SLOWEST ) {
// returned result if DCO registers hit min
                                   sendErrorMessage();
// trap the CPU if hit
                                 }
                                 else if( result ==
TI_DCO_SET_TO_FASTEST ) { // returned result if DCO registers hit max
                                   sendErrorMessage();
// trap the CPU if hit
                                 }
                                 else if( result == TI_DCO_TIMEOUT_ERROR
) { // returned result if DCO takes >10000 loops
                                   sendErrorMessage();
// trap the CPU if hit
                                 }

   }*/
    sendUART1Msg(welcomemessage, sizeof welcomemessage-1);
      /* Play the welcome message */

      while (1)  {
      switch( current_state ) {
            case  Start_Mode:
      // Starting up or wait for something to happen
                  if (mode&BIT4) {        // Give I2C higher priority
                        current_state=I2C_Receiving_Data;

                  }
                  else if (mode&BIT2) {
      // Start edge detected -- needed since we're using SMA clock
                        mode &= ~BIT2;
            // Clear BIT2 Saying we're in right state
                        current_state=Start_Edge_Received;
                  }
                  else if (mode&BIT5) {
                        P1OUT ^= BIT4;
      // Toggle P1.4
                        current_state=Send_Welcome_Message;
                  } else {
                        forceOnRS232Off();
                        _bis_SR_register(LPM3_bits + GIE);
                  }
            break;
            /* Transmitter Empty From here down */
```

```
            case  I2C_Receiving_Data:
      // Receive of I2C in progress
                  /*
                   *  Display the text received on I2C
                   */
                  forceRS232On();
      // Force the RS-232 Max3318E on
                  while (I2CDCTL&I2CBUSY);
      // Wait for I2C to become "idle"
                  mode &= ~BIT4;
            // Clear BIT4 Saying we're in right state
                  result = sizeI2CB;
                  sizeI2CB = 1;
            // Set to 0 in case wake up is something other than DMA
finished
                  sendUART1Msg(I2CBuffer, (size_t) result);       //
send received data out
//                _enable_interrupts();
                  P1OUT ^= BIT4;                // Toggle P1.4
                  current_state=Wait_For_DMA_Finish;
            break;
            case Wait_For_DMA_Finish:
      // Assuming DMA is transmitting
                  if (mode&BIT0) {
                        mode &= ~BIT0;
                        if (mode & BIT7) {// Check overflow
                              mode &= ~BIT7;
                              sendUART1Msg(errorMSGBB, sizeof
errorMSGBB-1);
                        }
                        else {
                        current_state=Start_Mode;
                              while (!(UTCTL1 & TXEPT));
      // Confirm no TXing before --> LPM3
                        }
                  } else {
                        _bis_SR_register(LPM0_bits +GIE);
      // DMA wasn't finished, we were interrupted by something else -
it will have to wait
                  }

                  //    _bic_SR_register_on_exit(LPM4_bits);
      // start DCO, CPU, and everything on exit
            break;
            case Start_Edge_Received:
                  if (mode&BIT1) {
      // Check to see if we're already echoed char
                        current_state=Receive_Echo_RS232;
                  } else {
                        _bis_SR_register(LPM0_bits + GIE);
                  }
            break;
            case Receive_Echo_RS232:
                  if (UTCTL1 & TXEPT) {         // TX Empty
                        mode &= ~BIT1;
                        if (mode & BIT7) {// Check overflow
                              mode &= ~BIT7;
```

```
                              sendUART1Msg(errorMSGBB, sizeof
errorMSGBB-1);
                              current_state=Wait_For_DMA_Finish;
                        }
                        else {
                              current_state=Start_Mode;
                        }
                  }
            break;
            case Send_Welcome_Message:
                  /*
                   *  Display the text received on I2C
                   */
                  mode &= ~BIT5;    // Turn off Send Welcome Msg
                  forceRS232On();
                  // Force the RS-232 Max3318E on
                  sendUART1Msg(repeatMSG, sizeof repeatMSG-1);    //
Send DMA test message through UART1
                  current_state=Wait_For_DMA_Finish;
            break;
            default: _never_executed();
      }
      }
}

/*
 * Timer A0 interrupt service routine
 */
TIMERA0_ISR(Timer_A)
__interrupt void Timer_A (void) {
        brailleStruct displayBraille;
        displayBraille.first=0;
        displayBraille.second=0;
        numberofseconds++;
        TACTL = TASSEL_1 + MC_2;              // ACLK, contmode
        /*
         * Now every blink will occur at
         * BRAILLE_WRITE_TIME/32768 or
         *  0.25 of a second for 8192=BRAILLE_WRITE_TIME
         */
        CCR0 += BRAILLE_WRITE_TIME;

        /* Toggles Braille lights
         * P4OUT = numberofseconds; // Change the braille lights to
count the number of seconds
         */
          /*
             * Really need to display both characters if the
             * first isn't a "space"
             * So first check if there's data in the buffer (sizeBB>0)
             * If so, check whether mode.4 = 0 and not a space
             *  and write the first Braille Char
             * If not, write the second Braille char and increment
             *
             */
            if (sizeBB>0) {
```

```
                displayBraille =
convertASCIIToHW(longBrailleBuffer[indexBBRead]);
                if (
(!(braillehalf&BIT4))&&(displayBraille.first!=SPACE_SYMBOL) ){
                      P4OUT = displayBraille.first;
                      braillehalf |= BIT4;
                }
                else {
                      P4OUT = displayBraille.second;
                      sizeBB--;
                      indexBBRead++;
                      braillehalf &= ~BIT4;    // Clear bit 4 so we
start with the first one next time
                }
                      /* Do a roll-over if we reached the end of the
buffer */
                if (indexBBRead>BRAILLE_BUFFER_LENGTH-1) {
                      indexBBRead = 0;
                }
            /*
                if (sizeBB<0) {
                       error - wrote more data than we should have
                       *  to the Braille pad

                       sendUART1Msg(errorMSGOF, sizeof errorMSGOF-1);
                }
             */
            }

        /*
         * Place code to test here
         *
         */
      if (numberofseconds>15) {
            numberofseconds=0;
           //P6OUT ^= BIT1;         // Toggle a test point
           //P2OUT ^= BIT2;         // Determine if voltage change is
working
             //P1OUT ^= BIT4;     // Toggle P1.4 - main pwer LED

                mode |= BIT5;           // set RS-232 need to check
xmit full
      }
       //  WDTCTL = WDTPW+WDTCNTCL;
      _bic_SR_register_on_exit(LPM4_bits + GIE);      // start DCO,
CPU, and everything on exit
}

/*
 * Force the RS-232 Chip on
 * Assumed to be a Max3318E
 */
inline void forceRS232On(void) {
      P5OUT |= BIT6;
}

/*
```

```
 * Turn the Force on bit off
 * Assumed to be a Max3318E
 */
inline void forceOnRS232Off(void) {
            P5OUT &= ~BIT6;
}

/*
 *  For a high voltage, we need to set the
 *  bit low, causing the resistor to be
 *  in parallel with the lower resistor
 */


inline void setDVoltage(uint8_t DCOCTLVal, uint8_t DCOBCVal, uint8_t
P2Val) {
      DCOCTL= DCOCTLVal;
      BCSCTL1= DCOBCVal;
    P2OUT = P2Val;
}

/*
 *  For a high voltage, we need to set the
 *  bit low, causing the resistor to be
 *  in parallel with the lower resistor
 *
inline void setHighVoltage(uint8_t DCOCTLH, uint8_t DCOBCH) {
      DCOCTL= DCOCTLH;
      BCSCTL1= DCOBCH;
    P2OUT &= ~BIT2;
}
 */

/*
 * For a high voltage, we need to set the
 * bit high, causing the top resistor
 * to become "smaller".

inline void setLowVoltage(uint8_t DCOCTLH, uint8_t DCOBCH) {
      DCOCTL = DCOCTLH;
      BCSCTL1= DCOBCH;
      P2OUT |= BIT2;
}
*/

inline void configDMAToSendUART1(void) {
        DMACTL0 = DMA0TSEL_10; // + DMA1TSEL_3;       // UTXIFG1
        DMA0SA = (unsigned int)welcomemessage;  // Source block address
        DMA0SZ = sizeof welcomemessage-1;       // Block size
        DMA0DA = (uint16_t) &TXBUF1;        // Dest single address
        DMA0CTL = DMASRCINCR_3 + DMASBDB + DMALEVEL + DMAIE;  // sngl,
inc src
        //set up two DMA channels

        /*DMA1SA = (unsigned int)&I2CDRB;    // Source block address
        DMA1SZ = 1;     // Block size
```

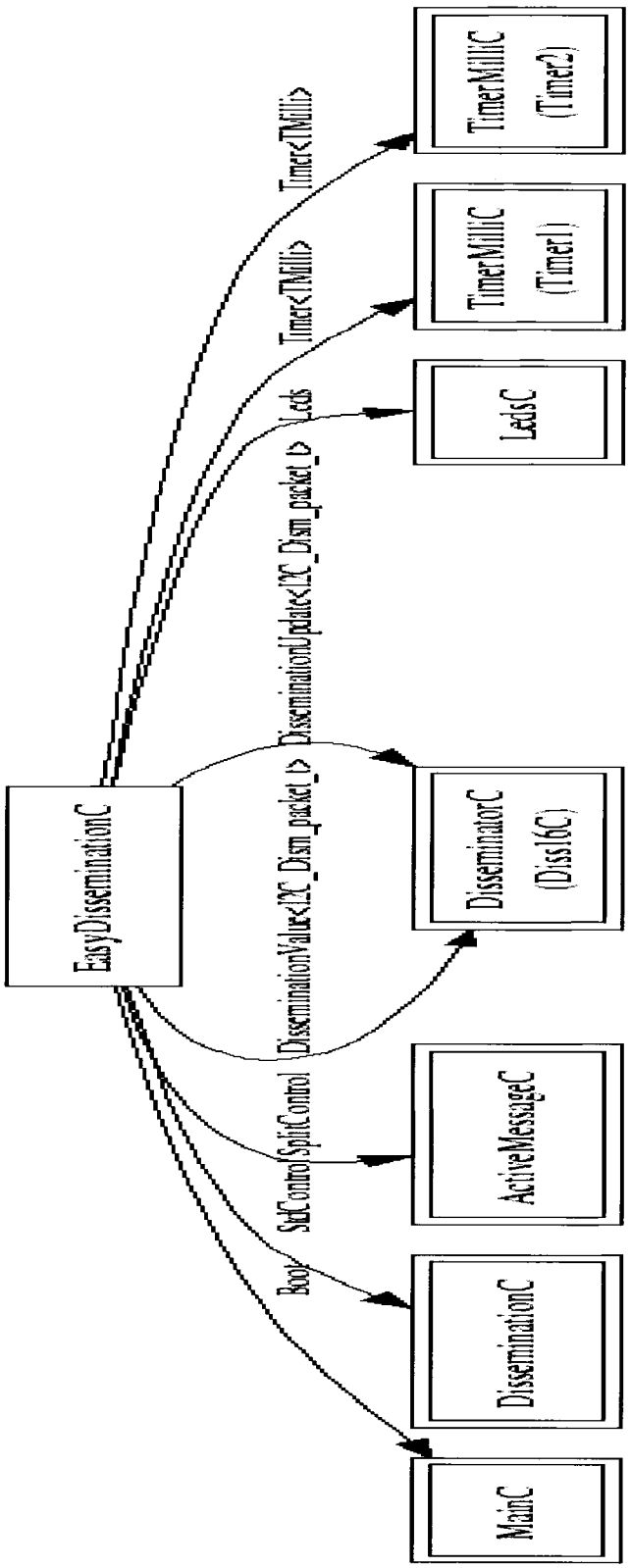
```
        DMA1DA = (uint16_t) RS232Buffer;                    // Dest
single address
        DMA1CTL = DMADT_4 + DMADSTINCR_3 + DMASBDB + DMAIE + DMAEN;  //
sngl, inc src
        */
        return;
}

// DMA interrupt service routine

DACDMA_ISR(DAC_DMA_ISR)
__interrupt void DAC_DMA_ISR(void) {
      if (DMA0CTL & DMAIFG) {
            DMA0CTL &= ~DMAIFG;// Clear DMA0 interrupt flag
            mode |= BIT0;     // set RS-232 need to check xmit full
            _bic_SR_register_on_exit(LPM4_bits+GIE);        // start
DCO and everything
      }

      /*if (DMA1CTL & DMAIFG)
              DMA1CTL &= ~DMAIFG;     // Clear DMA0 interrupt flag
   */

}

inline void configUART1forRS232(void) {
            U1CTL |= SWRST;           // Initialize USART state machine
            P3SEL |= 0xC0;            // P3.6,7 = USART1 TXD/RXD
            ME2 |= UTXE1 + URXE1;          // Enable USART1 TXD/RXD
            U1CTL |= CHAR;       // 8-bit character (8N1 UART)
            UTCTL1 |= SSEL1 + SSEL0 +  URXSE;           // UCLK =
SMCLK, start edge detect
            //URCTL1 = 0;                    // Receive normal
            UBR01  = 0x78;          // 4.6MHz/38400
            UBR11  = 0x00;          //
            UMCTL1 = 0x00;          // Modulation  4A
            U1CTL &= ~SWRST;        // Initialize USART state machine
            IE2 |= URXIE1;          // Enable USART1 RX interrupt
            return;
  }

USART1RX_ISR(usart1_rx)
__interrupt void usart1_rx (void) {
            uint8_t conASCIIToBraille;

            if(URXIFG1 & IFG2)       {// 0 Start-edge detected
                                      // 1 char received
                                      // RXBUF1 to TXBUF1
            conASCIIToBraille = RXBUF1;
            while (!(IFG2 & UTXIFG1));     // USART1 TX buffer ready?
            TXBUF1 = conASCIIToBraille;// Echo the character back out

                  longBrailleBuffer[indexBBWrite++]=conASCIIToBraille;
                  /* Do a roll-over if we reached the end of the buffer
*/
                  if (indexBBWrite>BRAILLE_BUFFER_LENGTH-1) {
                        indexBBWrite = 0;
```

```
                }
                sizeBB++;
                if (sizeBB>BRAILLE_BUFFER_LENGTH-1) {
                      /* error - received more than two packets worth
of data
                       * before writing anyout to the Braille pad
                       */
                      //sizeBB = BRAILLE_BUFFER_LENGTH-1;       //
since we have an overflow - use max size of buffer
                      mode |= BIT7;     // set overflow flag
                }
          mode |= BIT1;           // set RS-232 receive mode
                _bic_SR_register_on_exit(LPM4_bits+GIE);  // start
DCO
          } else {                // Start-edge detected
                UTCTL1 &=   ~URXSE;     // clear URXS
                UTCTL1 |=   URXSE;      // enable SE again URXS
                mode |= BIT2;           // SSEL0 = 0, RX activity
                _bic_SR_register_on_exit(LPM4_bits);      // start
DCO - interupts enabled to echo asap
          }

}

brailleStruct convertASCIIToHW(uint8_t ASCIIin) {
      int ii=0;
      uint8_t lowByte, highByte, tempByte;
      brailleStruct twoToReturn;
      twoToReturn.first=0;
      twoToReturn.second=0;
      /*
       * Make sure the received ASCII is a character
       * we're prepared for
       */
      if (( (uint8_t) ASCIIin) <32) {
            tempByte=60;                              // < sign
      }
      else if  (( (uint8_t) ASCIIin) >126) {
            tempByte=62;                              // > sign
      }
      else {
            tempByte = ASCIIin-32;
      }
      lowByte = LowByte(displayDigits[tempByte]);
      highByte = HiByte(displayDigits[tempByte]);
      for(ii=0; ii<(sizeof andMatrix); ii++) {
            if (shiftAmount[ii]<0) {
                  twoToReturn.first+=((highByte&andMatrix[ii])<<(-
1*shiftAmount[ii]));
                  twoToReturn.second+=((lowByte&andMatrix[ii])<<(-
1*shiftAmount[ii]));
            }
            else {

      twoToReturn.first+=((highByte&andMatrix[ii])>>shiftAmount[ii]);

      twoToReturn.second+=((lowByte&andMatrix[ii])>>shiftAmount[ii]);
```

```
            }

      }
      twoToReturn.first|=0xC0;
      twoToReturn.second|=0xC0;
      twoToReturn.first =~twoToReturn.first;
      twoToReturn.second=~twoToReturn.second;
      return twoToReturn;
}

inline void sendUART1Msg(const char *anyMSG, size_t length) {
            DMA0SA = (unsigned int)anyMSG;      // Source block address
            DMA0SZ = length;                    // Block size
            DMA0CTL |= DMAEN;             // Send DMA test message
through UART1
            _enable_interrupts();
}


inline void configUART0forI2C(void) {
      P3SEL |= 0x0A;                          // Select I2C pins
      //U0CTL |= SWRST;                       // Reset USART state
machine I2CTRX Bit must be cleared
    //U0CTL &= ~CHAR;                                 // 7 bit
addressing & 7 bit chars
      U0CTL |= I2C + SYNC;// + SWRST;   // Recommended init procedure
      U0CTL &= ~I2CEN;                  // Recommended init procedure
      I2CTCTL |= I2CSSEL1+I2CSSEL0;             // SMCLK
      //I2CDR Register : I2CWORD and I2CTRX =0
      I2COA = SLAVE_I2C_ADDRESS;                // Own Address is 048h
      I2CIE = RXRDYIE;// + ARDYIE;              // Enable  RXRDYIFG and
Register Access Ready interrupt (stop condition)
      U0CTL |= I2CEN;                           // Enable I2C
      return;
}

USART0TX_ISR(I2C_ISR)
__interrupt void I2C_ISR(void) {
      uint8_t tempRCByte;
      switch( I2CIV ) {
            case  I2CIV_AL: break;              // Arbitration lost
            case  I2CIV_NACK: break;            // No Acknowledge
            case  I2CIV_OA: break;              // Own Address
            case  I2CIV_ARDY:
      // Register Access Ready - Receive stop condition
                  P1OUT ^= BIT4;
      // Toggle P1.4
//                mode |= BIT3;
      // Set BIT3 Saying we're done receiving I2C
            break;
            case I2CIV_RXRDY:                   // Receive Ready
                  tempRCByte = I2CDRB;          // add data to buffer
                  //while (!(IFG2 & UTXIFG1));                    //
USART1 TX buffer ready?
                //TXBUF1 = tempRCByte;     // RXBUF1 to TXBUF1
                  I2CBuffer[sizeI2CB++]               =     tempRCByte;
                  longBrailleBuffer[indexBBWrite++]=  tempRCByte;
```

```
                  if (sizeI2CB > (BRAILLE_BUFFER_LENGTH-1)) {
                        mode |= BIT7;
      // set overflow flag
                        sizeI2CB = BRAILLE_BUFFER_LENGTH-1;       //
Buffer is linear
                  }
                  /* Do a roll-over if we reached the end of the buffer
*/
                  if (indexBBWrite> (BRAILLE_BUFFER_LENGTH-1)) {
                        indexBBWrite = 0;
      // Buffer is circular
                  }
                  sizeBB++;
                  if (sizeBB>BRAILLE_BUFFER_LENGTH-1) {
                        /* error - received more than two packets worth
of data
                         * before writing anyout to the Braille pad
                         */
                        mode |= BIT7;     // set overflow flag
                  }
                  mode |= BIT4;
      // Set BIT4 Saying we're receiving I2C

                  //while (!(UTCTL1 & TXEPT));// Confirm no TXing
before exit
            break;
            case I2CIV_TXRDY:  break;            // Transmit Ready
            case I2CIV_GC: break;                      // General Call
            case I2CIV_STT: break;             // Start Condition
            default: break; //_never_executed();
      }
      _bic_SR_register_on_exit(LPM4_bits);                  // start
DCO, CPU, and everything on exit
      _bis_SR_register_on_exit(GIE);                  // start DCO,
CPU, and everything on exit
}

WDT_ISR(watchdog_timer)
__interrupt void watchdog_timer(void) {
            P1OUT ^= BIT4;                            // Toggle P1.4
}
```

MICAZ NESC GRAPHVIS CODE DIAGRAM



MICAZ NESC SIMULATION CODE

Header File: EasyDissemination.h

```
#ifndef __EASY_DIS__
#define __EASY_DIS__
// Uncomment out below to limit the hops in a simulation
#define LIMIT_HOPS
enum {
      I2C_SEND_PERIOD_MILLI = 61440, // 1 minute * 2=122880
      I2C_ADDRESS_TO_SEND = 0x0048,
      BUFFER_LENGTH = 4,            // Note: this needs to be radix-2
      COMMAND_NODE_MASK = 1,
      HOPS_TO_ALLOW = 4
};

typedef nx_struct easydis_packet_t {
  nx_uint16_t nodeid;
  nx_uint8_t counter;
  nx_uint8_t priority;
  nx_uint8_t numberhops;
  nx_uint8_t data[TOSH_DATA_LENGTH-5];
} I2C_Dism_packet_t;

#endif
```

nesC Application File: EasyDisseminationApp.nc

```
configuration EasyDisseminationAppC {}
implementation {
      components EasyDisseminationC as App;
      components DisseminationC;
      components MainC;
      App.Boot-> MainC.Boot;
      App.DisseminationControl -> DisseminationC;

      components ActiveMessageC;
      App.RadioControl -> ActiveMessageC.SplitControl;

      components new DisseminatorC(I2C_Dism_packet_t, 0x1234) as
Diss16C;
      App.Value  -> Diss16C;
      App.Update -> Diss16C;

      components LedsC;
      App.Leds-> LedsC;

      components  new TimerMilliC() as Timer1;
      components  new TimerMilliC() as Timer2;
      App.Timer1-> Timer1;
      App.Timer2-> Timer2;
}
```

nesC Implementation File: EasyDisseminationC.nc

```
#include "EasyDissemination.h"
#include <Timer.h>
#include <stdlib.h>
#include <string.h>
//#include <I2C.h>
//#include <Atm128I2C.h>


module EasyDisseminationC {
      uses interface Boot;
      uses interface DisseminationValue<I2C_Dism_packet_t> as Value;
      uses interface DisseminationUpdate<I2C_Dism_packet_t> as Update;
      uses interface Leds;
      uses interface Timer<TMilli> as Timer1;
      uses interface Timer<TMilli> as Timer2;
//    uses interface Timer<TMilli> as Timer3;
      //uses interface I2CPacket<TI2CBasicAddr> as I2CBasicAddr;
      //uses interface Resource as I2CResource;
      uses interface StdControl as DisseminationControl;
      uses interface SplitControl as RadioControl;
}

implementation {
      uint16_t addrI2C  = I2C_ADDRESS_TO_SEND;
      uint8_t counter         = 32;
      bool busy         = FALSE;
      uint8_t errcnt          = 0;
      uint8_t bufferIndex     = 0;
      uint8_t flag            = 1;
      I2C_Dism_packet_t I2CMsgIn[BUFFER_LENGTH];
      I2C_Dism_packet_t I2CMsgOut;
      uint8_t buffer[2];
      /* reverse:  reverse string s in place
      * found from internet
      * compensate for fact of TOSSIM not using libc
      */
      void reverse(char s[]) {
            int c, i, j;
            for (i = 0, j = strlen(s)-1; i<j; i++, j--) {
                  c = s[i];
                  s[i] = s[j];
                  s[j] = c;
            }
      }
      /* itoa:  convert n to characters in s */
      void itoa(int n, char s[], int k) {
            int i, sign;
            if ((sign = n) < 0)  /* record sign */
                  n = -n;          /* make n positive */
            i = 0;
            do {       /* generate digits in reverse order */
                  s[i++] = n % 10 + '0';   /* get next digit */
            } while ((n /= 10) > 0);     /* delete it */
```

```
        if (sign < 0)
            s[i++] = '-';
        s[i] = '\0';
        reverse(s);
    }

    /**
     * Requests I2C resource as soon as booted
     */
    event void Boot.booted() {
        int ii;
        for (ii=0;ii<BUFFER_LENGTH;ii++) {
            I2CMsgIn[ii].nodeid     =     TOS_NODE_ID;
            I2CMsgIn[ii].counter    =     0;
            I2CMsgIn[ii].priority   =     1;
            I2CMsgIn[ii].numberhops =     0;
        }
        I2CMsgOut.nodeid  =     TOS_NODE_ID;
        I2CMsgOut.counter =     0;
        I2CMsgOut.priority      =     1;
        I2CMsgOut.numberhops    =     0;
        atomic {
            busy = FALSE;
        }
        dbg("Boot,ShowCounter,WritePayload", "Application booted.
ID = %d at time: %s\n",TOS_NODE_ID,sim_time_string());
        call Leds.set(LEDS_LED0 | LEDS_LED1 | LEDS_LED2);

        call RadioControl.start();

    }

    event void RadioControl.startDone(error_t err) {
        if (err != SUCCESS) {
            call RadioControl.start();
        } else {
            call Timer1.startOneShot(2*I2C_SEND_PERIOD_MILLI);
            call Leds.set(LEDS_LED1);
        }
    }


    event void Timer1.fired() {
        call DisseminationControl.start();
        if ( TOS_NODE_ID  < COMMAND_NODE_MASK ) {
            call Timer2.startPeriodic(I2C_SEND_PERIOD_MILLI);
            call Leds.set(LEDS_LED2);
        }
        else {
            call Leds.set(LEDS_LED0);
        }
    }

    task void ShowCounter() {
        I2C_Dism_packet_t* I2CMsg;
        if ( TOS_NODE_ID  < COMMAND_NODE_MASK ) {
            I2CMsg = &I2CMsgOut;
```

```
                dbg("ShowCounter", "Server node received message.
Sending back out.\n");
            } else {
                I2CMsg = &I2CMsgIn[bufferIndex];
                dbg("ShowCounter", "Regular node received message
buffering: %d  at time: %s\n",bufferIndex,sim_time_string());
                dbg("ShowCounter", "Priority: %d  - Hops: %d  -
Counter %d .\n", I2CMsg->priority,I2CMsg->numberhops,I2CMsg->counter);
            }

            if (I2CMsg->counter & 0x1)
                call Leds.led0On();
            else
                call Leds.led0Off();
            if (I2CMsg->counter & 0x2)
                call Leds.led1On();
            else
                call Leds.led1Off();
            if (I2CMsg->counter & 0x4)
                call Leds.led2On();
            else
                call Leds.led2Off();
      }


      task void ShowCounterServer() {
            I2C_Dism_packet_t* I2CMsg;
            I2CMsg = &I2CMsgOut;
            dbg("ShowCounter", "Server node sending message.  Counter:
%d\n",I2CMsg->counter);
      }

      /**
      * This fires the timer used to broadcast
      * Characters across 802.15.4
      */
      event void Timer2.fired() {
            char buf[5];
            char sentence[] = "I2C Message counter: ";
            I2CMsgOut.data[0] =     '\0'; // end with Null so we can do
a string concatenate
            I2CMsgOut.counter=I2CMsgOut.counter+1;
            // convert 123 to string [buf]
            itoa(I2CMsgOut.counter, buf, 10);
            strncat((char*) (I2CMsgOut.data), sentence,
TOSH_DATA_LENGTH-7);
            //http://www.delorie.com/djgpp/doc/libc/libc_881.html#fn_S
            strncat((char*) (I2CMsgOut.data), buf, TOSH_DATA_LENGTH-
strlen(sentence)-7);
            dbg("WritePayload", "I2C Message sent by timer:
\"%s\"\n",(char *) (I2CMsgOut.data));
            // show counter in leds
            post ShowCounterServer();
            // disseminate counter value
            call Update.change(&I2CMsgOut);

      }
```

```
	/**
	 * Checks if address is the same sent and makes the I2C not busy
	 * and turns off LED0 to signal message sending is complete
	 * keeps a count of the errors for each write event and turns all
LEDs
	 * on if greater than 4
	 */
	task void I2CBasicAddrwriteDone() {
		if (1) {
			//if (addr == addrI2C) {
				atomic {
					busy = FALSE;
				}
				errcnt=0;
		}
		else {
			errcnt++;
			/* if there are more than for tries with failure
light the lights */
			if (errcnt>4) {
				call Leds.set(LEDS_LED0 | LEDS_LED1 |
LEDS_LED2);
			}
			else {
				call Leds.set(errcnt);
			}
		}
		return;
	}

	/**
	* This sends via I2C the payload message with a start and stop
	* and lights LED0 to signal sending has started
	* lights LED1 if there's a problem sending
	*/

	task void WritePayload() {
		// This was created to allow for atomic reading of busy

		I2C_Dism_packet_t* I2CMsg;
		bool oldBusy;
		atomic {
			oldBusy = busy;
			I2CMsg = &I2CMsgIn[bufferIndex];
		}

		if (!oldBusy) {
			//buffer[0]=counter;
		// i2c_flags_t flags, uint16_t addr, uint8_t len, uint8_t*
data  - Atm128I2CMasterPacketP.nc
			dbg("WritePayload", "I2C Message out I2C port:
%s\n",(char *) (I2CMsg->data));
			//dbg("WritePayload",(char *) (I2CMsg->data));
			if (1) {
				atomic {
					busy = TRUE;
```

```
                        }
                        post I2CBasicAddrwriteDone();
                        atomic {
                              bufferIndex++;
                              bufferIndex &= BUFFER_LENGTH-1;
//BUFFER_LENGTH
                        }

                        //call Leds.led0Toggle();   // set  red toggle
= green and yellow off
                  }
                  else {
                        call Leds.set(LEDS_LED1);  // set  green on =
red and yellow off
                  }
            }
            else {  //  BUSY
                  call Leds.set(LEDS_LED1|LEDS_LED0);  // set  green
and red on = yellow off
                  // repost because we're busy with old post
                  post WritePayload();
            }


      }

      event void Value.changed() {
            const I2C_Dism_packet_t* newVal = call Value.get();
            //The above is a nonconstant pointer to data that cannot be
changed
            I2CMsgIn[bufferIndex] = *newVal;
            // show new counter in leds and write out I2C
            if ( TOS_NODE_ID  >= COMMAND_NODE_MASK ) {

                  dbg("WritePayload", "Value Changed: I2C Message:
%s\n",(char *) (newVal->data));
                  post ShowCounter();
                  post WritePayload();
            }
      }

      event void RadioControl.stopDone(error_t err) {
      }

}
```

MICAZ NESC LIVE CODE

nesC Application File: EasyDisseminationApp.nc

```
configuration EasyDisseminationAppC {}
implementation {
      components EasyDisseminationC as App;
      components DisseminationC;
      components MainC;
      App.Boot-> MainC.Boot;
      App.DisseminationControl -> DisseminationC;

      components ActiveMessageC;
      App.RadioControl -> ActiveMessageC.SplitControl;

      components new DisseminatorC(I2C_Dism_packet_t, 0x1234) as
Diss16C;
      App.Value  -> Diss16C;
      App.Update -> Diss16C;

      components LedsC;
      App.Leds-> LedsC;


      components  new TimerMilliC() as Timer1;
      components  new TimerMilliC() as Timer2;
//    components  new TimerMilliC() as Timer3;
      App.Timer1-> Timer1;
      App.Timer2-> Timer2;
//    App.Timer3-> Timer3;

      components  new Atm128I2CMasterC() as I2CMaster ;
      App.I2CBasicAddr-> I2CMaster.I2CPacket;
      App.I2CResource -> I2CMaster.Resource;
}
```

nesC Implementation File: EasyDisseminationC.nc

```
#include "EasyDissemination.h"
#include <Timer.h>
#include <I2C.h>
#include <Atm128I2C.h>


module EasyDisseminationC {
      uses interface Boot;
      uses interface DisseminationValue<I2C_Dism_packet_t> as Value;
      uses interface DisseminationUpdate<I2C_Dism_packet_t> as Update;
      uses interface Leds;
      uses interface Timer<TMilli> as Timer1;
      uses interface Timer<TMilli> as Timer2;
//    uses interface Timer<TMilli> as Timer3;
      uses interface I2CPacket<TI2CBasicAddr> as I2CBasicAddr;
      uses interface Resource as I2CResource;
```

```
	uses interface StdControl as DisseminationControl;
	uses interface SplitControl as RadioControl;
}

implementation {
	uint16_t addrI2C		= I2C_ADDRESS_TO_SEND;
	uint8_t counter			= 32;
	bool busy			= FALSE;
	uint8_t errcnt			= 0;
	uint8_t bufferIndex		= 0;
	uint8_t flag				= 1;
	I2C_Dism_packet_t I2CMsgIn[4];
	I2C_Dism_packet_t I2CMsgOut;
	uint8_t buffer[2];

	/**
	 * Requests I2C resource as soon as booted
	 */
	event void Boot.booted() {
		int ii;
		for (ii=0;ii<4;ii++) {
			I2CMsgIn[ii].nodeid	=	TOS_NODE_ID;
			I2CMsgIn[ii].counter	=	0;
			I2CMsgIn[ii].priority	=	1;
			I2CMsgIn[ii].numberhops	=	0;
		}
		I2CMsgOut.nodeid	=	TOS_NODE_ID;
		I2CMsgOut.counter	=	0;
		I2CMsgOut.priority	=	1;
		I2CMsgOut.numberhops	=	0;
		atomic {
			busy = FALSE;
		}
		call Leds.set(LEDS_LED0 | LEDS_LED1 | LEDS_LED2);
		call I2CResource.request();
	}

	event void RadioControl.startDone(error_t err) {
		if (err != SUCCESS) {
			call RadioControl.start();
		} else {
			call Timer1.startOneShot(2*I2C_SEND_PERIOD_MILLI);
			call Leds.set(LEDS_LED1);
		}
	}

	/**
	 * Once access to I2C is granted, turns
	 * yellow LED off and creates a 4 second timer
	 * to allow wait for MSP430 to set itself up for receiving
	 */
	event void I2CResource.granted(){
		call RadioControl.start();
		call Leds.set(LEDS_LED0);
		return;
	}
```

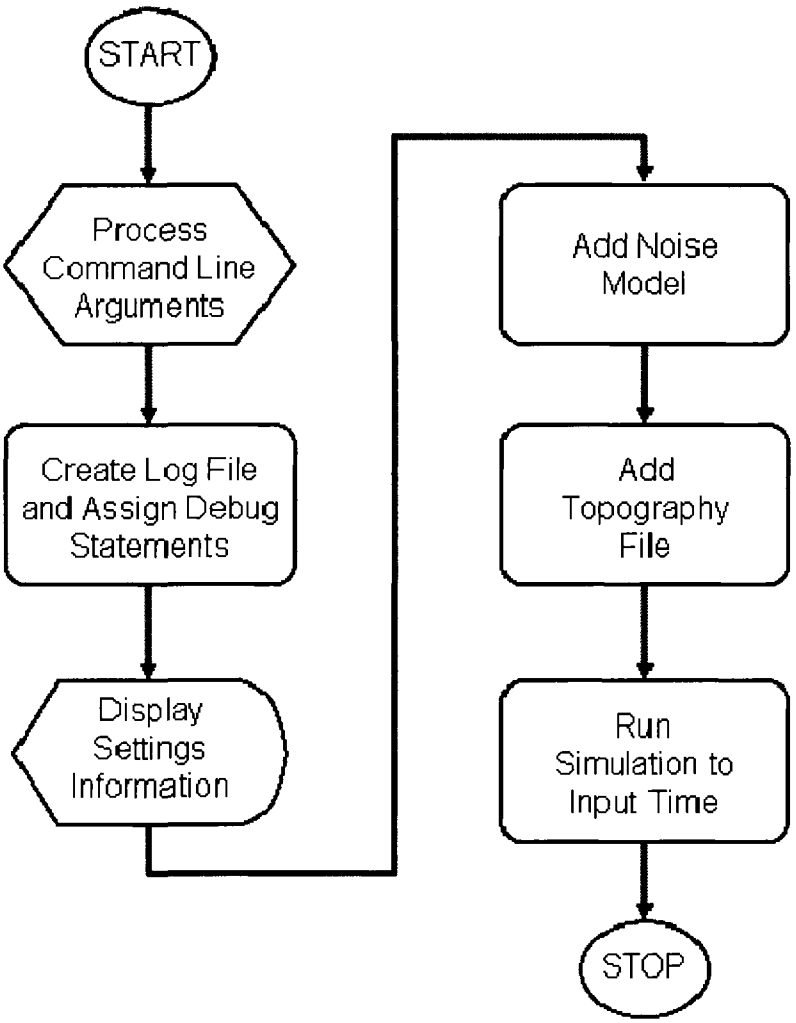
```
	event void Timer1.fired() {
		call DisseminationControl.start();
		if ( TOS_NODE_ID  == 1 ) {
			call Timer2.startPeriodic(I2C_SEND_PERIOD_MILLI);
			call Leds.set(LEDS_LED2);
		}
		else {
			call Leds.set(LEDS_LED0);
		}
		//call Timer3.startPeriodic(I2C_SEND_PERIOD_MILLI>>1);
	}

	task void ShowCounter() {
		I2C_Dism_packet_t* I2CMsg;
		if ( TOS_NODE_ID  == 1 ) {
			I2CMsg = &I2CMsgOut;
		} else {
			I2CMsg = &I2CMsgIn[bufferIndex];
		}


		if (I2CMsg->counter & 0x1)
			call Leds.led0On();
		else
			call Leds.led0Off();
		if (I2CMsg->counter & 0x2)
			call Leds.led1On();
		else
			call Leds.led1Off();
		if (I2CMsg->counter & 0x4)
			call Leds.led2On();
		else
			call Leds.led2Off();
	}

	/**
	* This fires the timer used to broadcast
	* Characters across 802.15.4
	*/
	event void Timer2.fired() {
		char buf[5];
		I2CMsgOut.data[0] =     '\0'; // end with Null so we can do
a string concatenate
		I2CMsgOut.counter=I2CMsgOut.counter+1;
		// convert 123 to string [buf]
		//itoa(int value, char *string, int radix)
		itoa(I2CMsgOut.counter, buf, 10);
		strncat((char*) (I2CMsgOut.data), "I2C Message counter: ",
TOSH_DATA_LENGTH-7);
		//http://www.delorie.com/djgpp/doc/libc/libc_881.html#fn_S
		strncat((char*) (I2CMsgOut.data), buf, TOSH_DATA_LENGTH-
27);
		// show counter in leds
		post ShowCounter();
		// disseminate counter value
		call Update.change(&I2CMsgOut);
```

```
	}

	/**
	 * This fires off a single character with a start and stop
	 * and lights LED0 to signal sending has started
	 * lights LED1 if there's a problem sending

	event void Timer3.fired() {
		// This was created to allow for atomic reading of busy
		bool oldBusy;
		atomic {
			oldBusy = busy;
		}

		if (!oldBusy) {
			if (counter++ >=127)  {		// set back to
32
				counter= 32;
			}

			buffer[0]=counter;
			if (call I2CBasicAddr.write(I2C_START | I2C_STOP,
addrI2C, 1,buffer ) == SUCCESS) {
				atomic {
					busy = TRUE;
				}
				call Leds.led0Toggle();	// set  red toggle =
green and yellow off
			}
			else {
				call Leds.set(LEDS_LED1);  // set  green on =
red and yellow off
			}
		}
		else {  //  BUSY
			call Leds.set(LEDS_LED1|LEDS_LED0);  // set  green
and red on = yellow off
		}


	}

	 */


	/**
	* This sends via I2C the payload message with a start and stop
	* and lights LED0 to signal sending has started
	* lights LED1 if there's a problem sending
	*/

	task void WritePayload() {
		// This was created to allow for atomic reading of busy

		I2C_Dism_packet_t* I2CMsg;
		bool oldBusy;
```

```
            atomic {
                  oldBusy = busy;
                  I2CMsg = &I2CMsgIn[bufferIndex];
            }

            if (!oldBusy) {
                  //buffer[0]=counter;
            // i2c_flags_t flags, uint16_t addr, uint8_t len, uint8_t*
data  - Atm128I2CMasterPacketP.nc
                  if (call I2CBasicAddr.write(I2C_START | I2C_STOP,
addrI2C, (uint8_t) strlen((char *) (I2CMsg->data)),(uint8_t *) &I2CMsg-
>data[0] ) == SUCCESS) {
                        atomic {
                              busy = TRUE;
                        }
                        //call Leds.led0Toggle();   // set  red toggle
= green and yellow off
                  }
                  else {
                        call Leds.set(LEDS_LED1);  // set  green on =
red and yellow off
                  }
            }
            else {  //  BUSY
                  call Leds.set(LEDS_LED1|LEDS_LED0);  // set  green
and red on = yellow off
                  // repost because we're busy with old post
                  post WritePayload();
            }


      }

      event void Value.changed() {
            const I2C_Dism_packet_t* newVal = call Value.get();

            I2CMsgIn[bufferIndex] = *newVal;
            // show new counter in leds and write out I2C
            if ( TOS_NODE_ID  != 1 ) {
                  post ShowCounter();
                  if (post WritePayload()==SUCCESS) {
                        atomic {
                              bufferIndex++;
                              bufferIndex &= 0x03;
                        }
                  }
            }
      }

      /**
       * Checks if address is the same sent and makes the I2C not busy
       * and turns off LED0 to signal message sending is complete
       * keeps a count of the errors for each write event and turns all
LEDs
       * on if greater than 4
       */
```

```
      async event void I2CBasicAddr.writeDone(error_t error, uint16_t
addr, uint8_t length, uint8_t* data) {
            if (error == SUCCESS) {
                  //if (addr == addrI2C) {
                        atomic {
                              busy = FALSE;
                        }
                        errcnt=0;
                        //call  Leds.led2Toggle();  // set  yellow
toggle = red and green off
            }
            else {
                  errcnt++;
                  /* if there are more than for tries with failure
light the lights */
                  if (errcnt>4) {
                        call Leds.set(LEDS_LED0 | LEDS_LED1 |
LEDS_LED2);
                  }
                  else {
                        call Leds.set(errcnt);
                  }
            }
            return;
      }

      async event void I2CBasicAddr.readDone(error_t error, uint16_t
addr,uint8_t length, uint8_t* data) {
            return;
      }

      event void RadioControl.stopDone(error_t err) {
      }
}
```

FLOWCHART FOR PYTHON SIMULATION SCRIPT



PYTHON SCRIPTS

Simulation Script: testscript.py

```
#!//usr/bin/env python
from __future__ import with_statement
import sys
import getopt
from TOSSIM import *
import time


def main(argv):
    try:
        opts, args  = getopt.getopt(argv,
"ho:m:s:t:",["help","output=","minutes=","seconds=","topo="])
    except getopt.GetoptError, err:
        print(str(err))
        usage()
        sys.exit(2)

    #Set default options should they not be specified
    topoFileToUse="linkgain.out"
#r"/home/kurt/workspace/tossimwork/src/blinktoradio/linkgain.out"
    outputFileName = "output_ns.txt"
    minutesToStop = 3
    secondsToStop = 0

    for opt,arg in opts:
        if (opt in ("-h", "--help")):
            usage()
            sys.exit()
        elif (opt in ("-o","--output")):
            outputFileName = arg
        elif (opt in ("-m","--minutes")):
            minutesToStop = arg
      elif (opt in ("-s","--seconds")):
            secondsToStop = arg
        elif (opt in ("-t","--topo")):
            topoFileToUse = arg


    #timeToStop = ".".join((minutesToStop,secondsToStop))
    timeToStop = float(minutesToStop)
    tenthsOfMinutesToStop = float(secondsToStop)/60.0
    timeToStop = timeToStop + tenthsOfMinutesToStop
    print("Output file name is: %s"%outputFileName)
    print("Simulation stop time is: %d minutes and %d
seconds."%(int(minutesToStop),int(secondsToStop)))
    print("Topofile is %s"%topoFileToUse)


    t = Tossim([])
    # Set up channels to display results
    debugFile=open(outputFileName,"w")
```

```
    #t.addChannel("Boot", sys.stdout)
    t.addChannel("Boot", debugFile)
    t.addChannel("RadioCountToLedsC", debugFile)
    t.addChannel("Dissemination", debugFile)
    t.addChannel("ShowCounter", debugFile)
    t.addChannel("WritePayload", debugFile)
    t.timeStr()
    #Print start time of sim to file
    s=" ".join((time.strftime('%X %x %Z'),'\n'))
    debugFile.write(s)
    #get the radio and create a noise model
    radiolist=t.radio()
    try:
        with open(r"/opt/tinyos-2.x/tos/lib/tossim/noise/meyer-
heavy.txt","r") as noise:
            lines=noise.readlines()
    except:
        print("Major problem reading file for noise.")
        sys.stderr.write("Topography file error.")
        sys.exit()
    print("Adding noise to nodes... please wait.")
    for line in lines:
        str=line.strip()
        if (str!=""):
            val=int(str)
            for ii in range(0,64):
                t.getNode(ii).addNoiseTraceReading(val)

    #print("line %s is %s"%(it,line))
    #if (it>10):
    # break
    print("Creating noise models... please wait.")
    for ii in range(0,64):
        t.getNode(ii).createNoiseModel()

    # supposedly, by using with, this is not necessary:noise.close()
    try:
        with open(topoFileToUse,"r") as topo:
            lines=topo.readlines()
    except:
        print("Problem reading in topography file.")
        sys.stderr.write("Topography file error.")
        sys.exit()
    print("Adding topography... please wait.\n Skipping printing
gains...")
    for line in lines:
        topovals=line.strip().split()
        if (len(topovals)>0):
            if (topovals[0]=="gain"):
            #print("node> %s : %s
%s"%(topovals[1],topovals[2],topovals[3]))

radiolist.add(int(topovals[1]),int(topovals[2]),float(topovals[3]))
    print("Booting all nodes... please wait. Starting simulation.")
    for ii in range(0,64):
        m  = t.getNode(ii)
        numberOfSeconds=float(ii)/float(32)
```

```
        m.bootAtTime(round(numberOfSeconds*t.ticksPerSecond())+4)
    #time = numberofSeconds*t.tickPerSecond()
    print("Running...")
    #Run for only 3 minutes or less
    hms=t.timeStr().split(':')
    ss=hms[2].split('.')  #this just gives seconds part and not past
decimal in ss[0]
    tenthsOfMinutesToStop = float(ss[0])/60.0
    #Actual time in minutes and tenths of minutes
    resTime=float(hms[1])+tenthsOfMinutesToStop  #Combine to give a
decimal number
    print resTime
    oldNumSeconds=int(hms[1])
    while (resTime<timeToStop):
    #for ii in range(0,200000):
        t.runNextEvent()
        hms=t.timeStr().split(':')
        ss=hms[2].split('.')  #this just gives seconds part and not
past decimal in ss[0]
        tenthsOfMinutesToStop = float(ss[0])/60.0
        resTime=float(hms[1])+tenthsOfMinutesToStop  #Combine to give a
decimal number
        #print hms
        #print("hms[1] is : %s"%hms[1])
        if (int(hms[1])!=oldNumSeconds):
            oldNumSeconds=int(hms[1])
            print("Processing in simulation minute: %d."%oldNumSeconds)
    #while (m.isOn()==0):
    #    t.runNextEvent()
    strvalue=('Simulation Ended at simulation time: ',t.timeStr(),'\n')
    s=" ".join(strvalue)
    debugFile.write(s)
    s=" ".join((time.strftime('%X %x %Z'),'\n'))
    debugFile.write(s)
    print("Simulation Ended at simulation time: %s \n"%t.timeStr())
    print("Local time is: %s \n"%time.strftime('%X %x %Z'))
    debugFile.flush()
    debugFile.close()

def usage():
    print("Usage: ", sys.argv[0])
    print("-h --help: This message.")
    print("-o --output: the file for ouput messages. Can be std.out.
[output_ns.txt]")
    print("-m --minutes: in minutes to stop: an integer for run time.
[default = 3]")
    print("-s --seconds: in seconds to stop: an integer for run time.
[default = 0]")
    print("-t --topo: topography file to use.  This is the file with
\"gain\" as first")
    print("           line items.    [default= linkgain.out]")

if __name__ == "__main__":
    main(sys.argv[1:])
```

Grid Plotting Script: plotnodelocations.py

```
#!//usr/bin/env python
from __future__ import with_statement
#from __future__ import print function
import sys
from matplotlib import *
from pylab import *
#ion()
#turns on interactive mode
# supposedly, by using with, this is not necessary:noise.close()


try:
    with
open(r"/home/kurt/workspace/tossimwork/src/blinktoradio/topology.out","
r") as topo:
        topolines=topo.readlines()
except:
    print("Problem reading in topography file.")
    sys.stderr.write("Topography file error.")
    sys.exit()
try:
    with
open(r"/home/kurt/workspace/tossimwork/src/blinktoradio/linkgain.out","
r") as topo:
        gainlines=topo.readlines()
except:
    print("Problem reading in linkgains file.")
    sys.stderr.write("Topography file error.")
    sys.exit()
print("Adding topography... please wait.\n Skipping printing gains...")
maxnode=0
gainsforZero=zeros(64,'f')
gainsforZero[0]=-70
for line in gainlines:
    topovals=line.strip().split()
    if (len(topovals)>0):
        if (topovals[0]=="gain"):
            if (int(topovals[1])>maxnode):
                maxnode = int(topovals[1])
          if (int(topovals[1])==0):
                print("topovals[0]: %s topovals[2]:%s and topovals[3]
is %s"%(topovals[0],topovals[2],topovals[3]))
                gainsforZero[int(topovals[2])]=float(topovals[3])
for ii in range(64):
    print("%d : %s"%(ii,gainsforZero[ii]))
print("Max node value is: %s"%maxnode);
x=zeros(64,'f')
y=zeros(64,'f')
z={}
for line in topolines:
    topovals=line.strip().split()
    if (len(topovals)>0):
      x[int(topovals[0])]=(int(round(float(topovals[1]))))
      y[int(topovals[0])]=(int(round(float(topovals[2]))))
```

```
        print("node> %s :x= %d y=
%d"%(topovals[0],x[int(topovals[0])],y[int(topovals[0])]))
      #z[[x[int(topovals[0])]],[y[int(topovals[0])]]]=
gainsforZero[int(topovals[0])]
        #(int(topovals[1]),int(topovals[2]),float(topovals[3]))
print(len(y),"  ",len(x),"   ",len(gainsforZero))
x1=arange(0,32,4)
y1=arange(0,32,4)
X,Y=meshgrid(x1,y1)
print("printing x")
print(X)
#for jj in X:
#    for ii in Y:
#        print(jj)
#        print(ii)
#print("done with IJ")
#Z=func3(z,X,Y)
pcolor(X,Y,gainsforZero.reshape(8,8))
colorbar()
hold(True)
plot(x,y,ls='None',marker='*',ms=15,markerfacecolor='red')
axis([-1, 30, -1, 30])
grid(True)
for ii in range(64):
    text(x[ii]+0.5,y[ii]+0.5,ii,fontsize=12)
title('Grid Layout Showing Gain from Base Station Node (0)')
show()
```

VITA

Kurt Matthew Peters
ECE department, KH 231
Old Dominion University
Norfolk, VA 23529

EDUCATION

B.S. Electrical Engineering, May 1990
United States Air Force Academy, Colorado Springs, Colorado

M.S. Electrical Engineering, December 1994
Air Force Institute of Technology, Wright-Patterson AFB, Ohio

Ph.D. Electrical Engineering, December 2009
Old Dominion University, Norfolk, Virginia

SELECTED PUBLICATIONS AND CONFERENCE PRESENTATIONS

1. Makhin Thitsa, Haider Ali, Feng Wu, Kurt Peters and Sacharia Albin, "Modeling the Effect of Oxidation and Etching of Silicon Photonic Crystals", in *Multiscale Modeling of Materials,* edited by R. Devanathan, M. J. Caturla, A. Kubota, A. Chartier, S. Phillpot, *Mater. Res. Soc. Symp. Proc.* **978E**, Warrendale, PA, 2007.
2. Kurt Peters and Sacharia Albin, 802.15.4 Based Visually Impaired Notification System," Paper presented at *ICST*, New Zealand, November 2007.
3. K. Peters, M. Creighton, "Monitoring lithography product data for real-time focus control", *Microlithography World*, Nov. 2004.
4. K. Peters, M. Creighton, "Practical Tips for Evaluation of Lithography Product Data for Real Time Dose and Focus Control and Estimation of Optical Aberrations," *Interface 2003*, Paper 22, Sept. 2003.
5. K Peters, "Modeling and Simulation of NVS Lithography APC System with real-world Parameters", *2003 Intel APC Summit,* Best Paper Award, 2003.

PATENTS

1. Corning Inc. for Apparatus for Estimating Bit Error Rate by Sampling in WDM Communication System - US Pub. Num. 6,295,614.
2. Orasee Inc. for Method for Scaling and Interlacing Multidimensional and Motion Images – Intl. Pub. Num. WO 02/063560.