UNIVERSITY OF LIMERICK
LIMERICK, IRELAND

Design of a Completely Wireless Security Camera System

A ***Major Qualifying Project*** Report

submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE
WORCESTER, MASSACHUSETTS

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

on the day of

Friday, October 12, 2007

by

Joseph A Bosman

Steven Olivieri

Ipek Ozil

Brandon C. Steacy

Advisor: Professor Donald R. Brown

Advisor: Professor Richard F. Vaz

Abstract

The Enterprise Research Center at the University of Limerick desired an energy-efficient and inexpensive way to monitor their car parks. During our stay in Ireland, we worked closely with the University and other WPI students to develop a ZigBee-enabled wireless security camera system to help fill this need. The main goal of our design was to develop a network that allowed for the transmitting and receiving of images from camera nodes to a base station.

Executive Summary

According to a recent national study, there were 12,600 vehicles that were stolen and an additional 37,900 thefts from motor vehicles were committed in Ireland in just one year [1]. Although these numbers alone are striking, they are not entirely representative of the situation. According to the same study only thirty-nine percent of thefts from motor vehicles were reported due to the victims' perception that local law enforcement could not, or would not, do anything related to the crime committed [1]. One step to reduce the frequency of automobile crime in Ireland is to begin installing security camera systems in car parks throughout the country. A security camera system in a car park would not only enhance the ability of local law-enforcement authorities to apprehend car thieves, but also deter the motivation of thieves to attempt the crime.

Facing similar automobile crimes in their car parks, the Enterprise Research Centre (ERC) at the University of Limerick (UL) desired a wireless security camera system. The system was designed to avoid the high installation cost of a wired security camera system while at the same time not inheriting the limitations of some wireless security camera systems on the market today. These limitations included insufficient battery life, surveillance area restrictions, and insufficient alerts.

In 2007, two student research teams from Worcester Polytechnic Institute (WPI) helped the ERC to design a proof-of-concept wireless security camera system. One team focused on the development of an energy efficient camera system that included a PIR sensor, camera, and memory. The other team focused on the design described in this document: a mesh network subsystem including TX/RX hardware to transmit images through the system to a PC.

At the core of our proof-of-concept design was the ZigBee wireless protocol, which is designed for use in low-cost, low power mesh networks. The EM260 Radio Control Module (RCM), designed by Ember Corporation, was selected to utilize the ZigBee protocol. The module offers a complete wireless solution for designing low power ZigBee applications. The EM260 RCM was combined with the MSP430F1611 microcontroller using the Serial Peripheral Interface (SPI) to create a fully-functional camera node and base station. Figure 1 illustrates the general system architecture of the camera node. The base station architecture is the same except it did not contain the camera, PIR sensor, or memory and was connected to a PC.

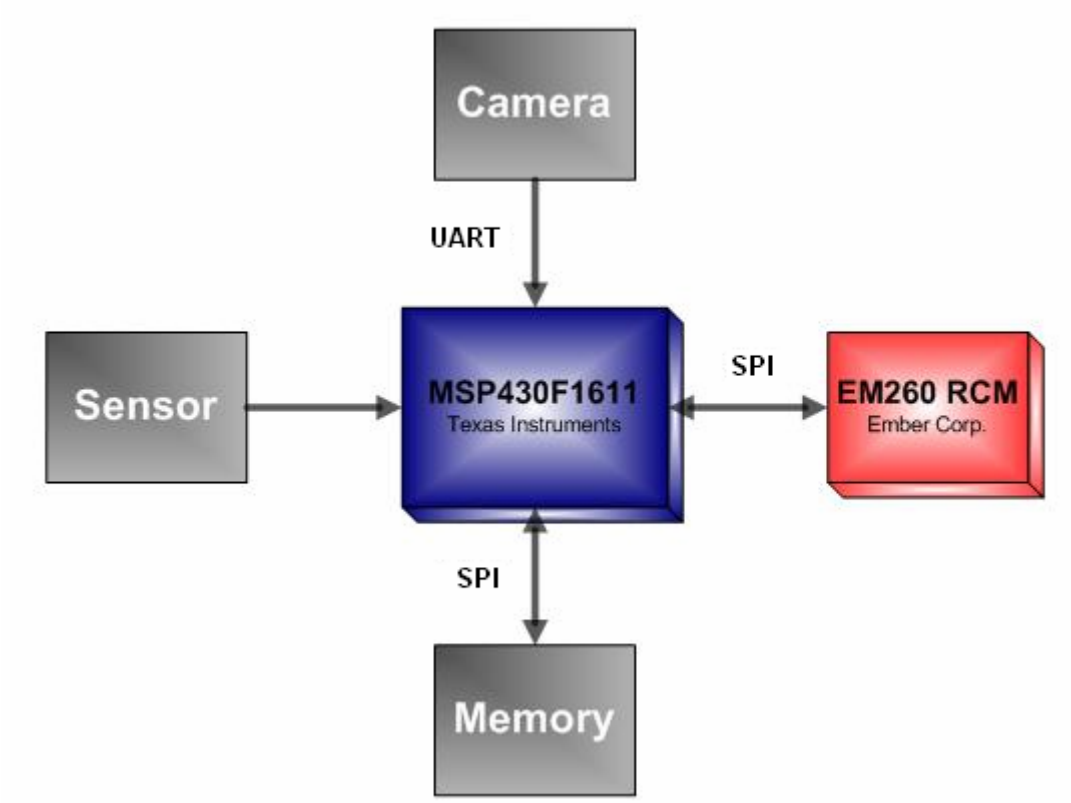

Figure 1: Camera node.

At the conclusion of the project, the teams were able to provide the Enterprise Research Centre with a proof-of-concept wireless security camera system. The final proof-of-concept system consisted of two camera nodes and a base station that together formed a functional wireless mesh network. Each node has a battery life of up to one year and has a transmission range of up to 90 meters. The system supports many nodes and has the ability to add or remove nodes from the network providing the system the capability of covering variable surveillance areas. All of the documentation on the wireless security camera system has been left with the ERC for future development. The teams are hopeful that research continues on this system and it is able to come to the market in the near future to aid in the reduction of car park crime not only in Ireland, but throughout the world.

Acknowledgements

We are grateful to be given the opportunity to partake in an incredible project that could not have been possible without the guidance of many different individuals and organizations. These individuals helped to keep us on track with the project as well as make us feel at home here in Limerick, Ireland. We extend our deepest thanks to all of the following participants:

The University of Limerick has provided a beautiful campus setting for us not only to complete our Major Qualifying Project, but also to enjoy a new culture and campus life. With much help and cooperation by many of the faculty and staff of the University we extend our sincerest thanks to all.

The Enterprise Research Centre located at the University of Limerick was where we worked for ten weeks on our project. Our thanks go out to three individuals that helped us to our goal of completing our Major Qualifying Project (MQP). Dr. Mark Southern, our chief sponsor, gave us the motivation and desire to complete this project. With all his enthusiasm about the project, it gave our team the drive that we needed to complete this very challenging task. We extend our thanks to Seamus Clifford who supplied us with endless ideas of how to enhance our project. He drove us to the limit with his knowledge and creativity. We also extend our appreciation to John Harris, the electronics expert, who gave us great insight about certain design concepts that aided in the decision making process for our project.

This one individual made this experience everything that we had hoped and more. Charlotte Tuohy spent months preparing for our arrival in Limerick and then spent ten weeks more helping us with anything and everything. From planning weekend trips to showing us where to buy the best meat, Charlotte was never more than a phone call away and always willing to help. Charlotte helped with an easy transition into the Irish culture which set a strong foundation for many memorable times here in Ireland. We thank you Charlotte for taking the time out of your life to make ours better.

Thanks to Worcester Polytechnic Institute (WPI) and the Interdisciplinary and Global Studies Division (IGSD) for giving us the opportunity to travel abroad to a beautiful country and complete our MQP. Without the assistance of Natalie Mello, Richard F. Vaz, and Donald R. Brown we would not have had this marvelous opportunity. We extend our thanks to Professor Richard F. Vaz and Professor Donald R. Brown for preparing us for this exciting adventure and providing us with helpful tips about the Irish ways. We would like to thank Professor Brown for

spending the first week with the group to help make the transition smooth. Also the dedication and support that both of the Professors showed throughout this challenging process were extremely helpful. We thank both of them for their great wisdom and knowledge that they shared in our weekly teleconferences. Thank you very much for your support throughout these ten marvelous weeks spent here in Ireland.

We would also like to thank Mr. David Egan of Ember Corporation, Mr. David Chalmers of Telegesis, and Mr. Andrew Pockson of Anglia Components for extending their knowledge and troubleshooting techniques. Simon Effler, a Ph.D. student at the University of Limerick, generously offered his time and knowledge day after day while we struggled to get our system working. Without Simon's unending help, witty humor, and excellent oscilloscope skills, our time in the lab would have been far less interesting. Vielen Dank, Simon.

Table of Contents

Abstract i
Executive Summary ii
Acknowledgements iv
1 Introduction 1
2 Prior Art 3
2.1 Big Bruin: 4 Channel Wireless Camera System 3
2.2 Wireless Digital Four 4
2.3 Observer IV 6
3 Requirements 8
3.1 Design Requirements 8
3.2 Project Proof-of-Concept 9
4 Design Considerations and Selection 10
4.1 Wireless Protocol 10
4.2 Integration with MSP430 12
4.3 Co-Processor Selection: EM260 14
4.4 Base Station Selection 15
4.4.1 Telegesis ZigBee ETRX2USB 16
4.4.2 MSP430 and EM260 RCM 16
5 ZigBee Wireless Protocol 18
5.1 ZigBee Description 18
5.2 ZigBee Applications 19
6 System Structure 23
6.1 Network-Level Design 24
6.2 Node-Level Design 25
6.3 Base Station Design 27
7 System Integration 28
7.1 Hardware Design 28
7.1.1 EM260 RCM 28
7.1.2 MSP430F1611 29
7.1.3 Pin Assignments 30
7.2 SPI with EM260 RCM 31
7.3 Base Station Programming 32
7.4 Camera Node Programming 35
8 Testing and Results 37
8.1 Basic Functionality 37
8.2 Range 37
8.3 Two Node Network 40
8.4 Multi-Hop 40
9 Final Proof-of-Concept 42
9.1 Cost Analysis 42
9.2 Final Product Description 43
9.3 System Limitations 44

10 Recommendations 46
10.1 PCB Design 46
10.2 Testing 46
10.3 Node Status Indicators 47
10.4 Wireless Signal Detector 47
10.5 Network User Interface Software 48
References 50
Appendix A: Mesh Networks 53
Appendix B: IEEE 802.15.4 57
Appendix C: Open Systems Interconnection (OSI) Model 58
Appendix D: Images of the Camera Node and the Base Station 60
Appendix E: Instructions for Security Camera System Code 61
Appendix F: MSP Base Station Code 64
Appendix G: PC Base Station Code 85
Appendix H: Camera Node Code 100
Appendix I: ETRX2USB Sender Code 130

Table of Figures

Figure 1: Camera node. iii
Figure 2: 4 Channel Wireless Camera System kit [6]. 4
Figure 3: Wireless Digital Four camera system [4]. 5
Figure 4: Observer IV [7]. 6
Figure 5: Single microcontroller configuration. 12
Figure 6: Dual microcontroller configuration. 13
Figure 7: Microcontroller with a co-processor configuration. 14
Figure 8: EM260 RCM. 15
Figure 9: Telegesis ETRX2USB. 16
Figure 10: MSP430 interfaces with the EM260 and the PC. 17
Figure 11: EmberZNet Model. 22
Figure 12: Node system structure. 23
Figure 13: Base station system structure. 24
Figure 14: Network level design. 25
Figure 15: Node level design. 26
Figure 16: Base station level design. 27
Figure 17: Two SPI 6-pin connectors on EM260 RCM. 28
Figure 18: Olimex MSP430F1611 Board. 30
Figure 19: SPI transaction for EM260 RCM [16]. 32
Figure 20: These images were taken from two distances, ten and 180 meters, respectively. There is no difference in quality. 38
Figure 21: These images were taken from two distances, ten and 90 meters, respectively. There is no difference in quality. 40
Figure 22: Multi-hop test. 41
Figure 23: Wireless Signal Detector. 48
Figure 24: Infrastructure architecture for WMN [23]. 53
Figure 25: IEEE 802.15.4 Standard model [26]. 57
Figure 26: OSI Model 58
Figure 27: Camera node with the enclosure. 60
Figure 28: Camera node inside the enclosure. 60
Figure 29: Base station 60

Table of Tables

Table 1: Comparison of the wireless protocols..... 11
Table 2: SPI pin descriptions for EM260 RCM. 29
Table 3: SPI pin descriptions for MSP430F1611. 30
Table 4: Pin connections between EM260 RCM and MSP430F1611..... 31
Table 5: Sending data in an open field..... 38
Table 6: Sending data in a car park..... 39
Table 7: Total cost per node for the given price breaks..... 42
Table 8: Total cost per base station for the given price breaks..... 43

1 Introduction

According to a recent national study, Irish Citizens reported 37,900 thefts from motor vehicles and an additional 12,600 vehicles stolen within one year [1]. Car owners run the risk of having their automobiles stolen or keyed, tires slashed, windows broken, antennas bent, and more. In many of these cases, the owner reports the vandalism knowing that there is little that the local authorities can do. In fact, according to the same survey, only thirty nine percent of the actual thefts from cars were reported due to victims' perception that local law-enforcement authorities could not, or would not, do anything in relation to the crime committed [1].

One step to reducing the frequency of automobile crime in Ireland is to begin installing security camera systems in car parks throughout the country. There have been many cases in which security cameras have aided in convicting criminals such as in the 1993 Bulger Case in which two eleven-year-old boys were caught on camera kidnapping a two-year-old [2]. A camera security system in a car park could not only heighten the ability of local law-enforcement authorities to apprehend car thieves, but also deter the motivation of thieves to attempt the crime. However, installing a wired camera system in a car park can be expensive. Not only does the camera system need to be purchased, but the car park must be dug up for laying wire and putting in foundations for the poles where the cameras are mounted. According to Surveillance Secure Inc., a full service provider for wired surveillance camera systems, installation of a basic wired security camera system, of approximately twenty cameras for a small car park, would cost up to €22,000 [3].

One alternative to installing a wired security camera system is to install a wireless security camera system. This would avoid the cost for construction with digging trenches for laying wire. Although there is the attraction of reduced cost, current wireless camera security camera systems still have numerous limitations. Self-installation systems, such as the Wireless Digital Four designed by 123 Security, are often composed of a fixed number of cameras which must all be located within a certain distance of a base receiving unit, which severely limits the range of the surveillance area [4]. Also, most of these systems require power cords to plug in each individual camera. This can lead to higher power costs and greatly restrict where cameras can be installed; this limitation greatly reduces the usefulness of a wireless system. In addition,

many wireless car park security camera systems, both professionally- and self-installed, on the market today send video that is captured to a central location, where it is viewed live or recorded. This requires the hiring of a security guard to monitor the live video or sift through recorded video in the event that a crime is reported in the car park, further increasing the cost of the overall system.

This project, sponsored by the University of Limerick in Ireland, was to design a proof-of-concept wireless security camera system to fill the void for an inexpensive, low power, and low maintenance car park security camera system solution. The design was focused on avoiding the high installation cost of a wired security camera systems while at the same time not inheriting the same limitations of some wireless security camera systems on the market today.

The project was divided between two teams: one team that designed the energy efficient camera system and a second team that designed a mesh network subsystem. This report includes the documentation for the mesh network design. The first team's report can be obtained by contacting the WPI Gordon Library at Worcester Polytechnic Institute in Worcester, Massachusetts [5]. The two teams worked together throughout the project in order to create one complete system that integrated both teams' designs. The camera team designed and implemented a fully functional camera node which captures images and saves them to a memory card. The mesh networking team designed a networking system that allowed these images to be sent to the base station. The complete proof-of-concept design is capable of having distances between nodes of 90 meters and costs approximately €2,185 for ten nodes and a base station. When compared to a wired system priced at €22,000 and systems with distance requirements such as the Wireless Digital Four, which can only transmit up to 180 meters from the base receiving unit, our system provides a cost effective, range efficient, wireless security system alternative. The completion of this proof-of-concept design lays the foundation for a low cost, low power security camera system which will aid in reducing car park crime not just in Ireland, but throughout the world.

2 Prior Art

A wireless security camera system is a desirable technology because of its low installation cost, when compared to a wired system. By becoming more familiar with the currently available wireless security camera systems, we were able to determine where we should make improvements. There are many products on the market today with features that help to enhance the performance of wireless security systems, but there are still numerous drawbacks to these systems. Each of the systems discussed in the following sections have their own strengths and weaknesses and by evaluating these strengths and weaknesses, we were better able to determine characteristics that would distinguish our product from the current products on the market.

2.1 Big Bruin: 4 Channel Wireless Camera System

The 4 Channel Wireless Camera System can be used in a variety of locations including homes, small offices, and even the outdoors. This system is available for €215.00 and includes the following components:

- One wireless receiver
- Four miniature cameras
- Four 9-Volt camera batteries
- Four antennas
- Five A/V cables
- Four camera AC adapter (120V AC, 60Hz; 8V DC, 200mA)
- One receiver AC adapter (100-240V, 50/60Hz, 0.25A)

Figure 2: 4 Channel Wireless Camera System kit [6].

The 4 Channel Wireless Camera System has four very small cameras that are easily mounted on shelves or placed on desks. The system uses either one monitor or four to view the video that is captured, though no monitors are included in the price of the system. In a single-monitor configuration the system scrolls through the four video feeds. Four monitors allow the user to view the footage captured by all four cameras at once. The system has a range up to 220 meters in line-of-sight conditions. When walls or other objects obstruct the line-of-sight between cameras, the system range is greatly decreased. The cameras are also mounted on brackets that allow the user to quickly change their vertical coverage area. Unfortunately, the mounting brackets do not allow the cameras to swivel horizontally, requiring that the user to remount the cameras if he or she desires to change their horizontal area of coverage. Each camera also equipped with an embedded microphone. Cameras can either be powered by nine-volt batteries, which allows them to operate for two hours, or by AC adapters, leaving them no longer wireless. Since the system can only last for two hours with the battery supply or must be plugged into an AC source, this system is not a viable solution for the University of Limerick. [6].

2.2 Wireless Digital Four

The next system that we analyzed as a possible solution was the Wireless Digital Four made by 123 Security Products. This product was intended for use by small businesses and is available for €1,450. This system is a general-purpose security system that can be used in a car park. The system includes the following:

- One 4 channel/160GB recorder
- Four BT176 wireless 2.4 GHz cameras
- Four 2.4 GHz transmitters/receivers
- One BW 12 security monitor

Figure 3: Wireless Digital Four camera system [4].

This system allows for each of the four cameras to be placed up to 180 meters from the receiver, which would be sufficient for one of the smaller University car parks. The user then has the option to view each signal on the monitor individually, or all at once, and the option to view signals in real-time or in a time lapse mode. The four-channel receiver with a 160 GB recorder also offers features such as two weeks of data storage and motion record. The motion record feature eliminates the need for a motion detector on the camera. Each camera is also equipped with night vision of up to six meters. Unfortunately, the cameras require AC power adapters which restrict the flexibility of the system and may require that cable be laid. In addition, this system can only support four cameras, which means that adding cameras to the network requires purchasing another full system, which would be very costly when trying to survey a large car park. Therefore, the Wireless Digital Four would not be a good solution for the University of Limerick because the system cannot operate without wires and cameras cannot be easily added to the system [4].

2.3 Observer IV

The last and final option that we considered was the Observer IV, which is a wireless camera that is capable of taking pictures and making them available on the web via a GSM cellular telephone network. This system only contains one camera. The camera is battery operated and is enclosed in a weatherproof case. The Observer IV, available for €740, has been used by wildlife and game management for monitoring animal numbers and movement, illegal trespassing, current weather conditions, and to take pictures for guide books. The camera can be set to take pictures at set time intervals, or when detecting heat or motion. The Observer IV can send images through e-mail attachments, through text messages via SMS, or upload them to a server using File Transfer Protocol (FTP). The Observer IV can take between 350-650 images before needing a new battery supply. The unit requires five AA batteries or also offers the option of an AC adapter plug in [7].

Figure 4: Observer IV [7].

The Observer IV seems to be a viable option for the University of Limerick except for the cost that would be associated with purchasing many of these cameras to survey a car park. In addition, the cameras battery supply is only capable of capturing 350-650 pictures, which means the system will not have the battery life that the university desires. For these reasons, this system was not an adequate solution for the university.

After considering the options that the University of Limerick had, we determined that there was a need for a system that was completely wireless, featured a battery life of at least one year, and the ability to support multiple nodes. The camera systems discussed in this section did not meet the needs of the university and therefore, were not adequate solutions. The following

chapter discusses the requirements set forth by the University of Limerick needed to create a product that would be highly desired by customers.

3 Requirements

While designing our proof-of-concept system, we needed to consider a number of specifications and requirements. We considered requirements set by the University of Limerick as well as requirements determined by our prior art research to produce a product that would meet the needs of the University as well as customers in need of a wireless security camera system.

3.1 Design Requirements

In order to design a product that would meet not only the needs of our sponsor, but of a consumer, we first had to determine what those needs were [8]. Through discussion with our sponsor and research, we determined that the system needed to meet the following requirements:

- Wireless operation in the 2.4 GHz unlicensed ISM frequency band
- A transmission range of at least 20 meters
- A low cost of no more than €250 per node
- A battery life of at least one year
- A data rate sufficient for transmitting images

In addition, the system must possess the following features:

- The ability to support at least as many nodes as an average-sized car park would require for full coverage
- The ability to automatically recognize new nodes added to the system as well as the removal of nodes from the system
- The ability to interface with a standard PC running Microsoft's Windows operating system
- The ability to alert the system's administrator of new pictures via SMS or e-mail

The University desired a wireless system to avoid the high costs for installation that are inherent in wired security camera systems. In addition they desired a system that was low-maintenance, leading to the requirement of a battery-operated system with camera modules that

could last for approximately one year without the need to change the batteries. Because the size and shape of the car parks at the University vary greatly, our sponsors desired a system that is easy to install and is adaptable to any area in need of surveillance. Finally, the University sponsors wanted to be able to view the images on a PC and have an alert sent to a mobile phone.

The motivation for this system came from the amount of crime on campus, especially car thefts and break-ins. However we did not want to limit the design any more than necessary. Therefore, we designed the wireless security camera system to be function in any generic car park.

3.2 Project Proof-of-Concept

The goal of this project was to design a proof-of-concept wireless security camera system. In order to demonstrate the full functionality of the system we determined that it needed to contain a base station and at least two camera nodes. These nodes needed to be capable of forming a mesh network, which allows them to send images to the base station. In addition, the network needed to have the abilities to automatically recognize new nodes and update routing tables in the event that a node is taken offline. The proof-of-concept needed to encompass all the previously mentioned system requirements. That is, the cameras in our security system needed to be able to operate for extended periods of time on a portable power supply.

4 Design Considerations and Selection

In order to design the best possible system, we needed to consider a number of design options. This chapter discusses the options that were available to use, which ones we selected, and why we selected them.

4.1 Wireless Protocol

There are a number of wireless protocols that can be used to transmit data from one camera to another. The most popular are WiFi, Bluetooth, and ZigBee. The 802.11 working groups of the Institute of Electrical and Electronics Engineers (IEEE) developed these three protocols. While all three of them make use of standard radio frequency (RF) technology, they differ in nearly every other way.

WiFi (802.11g) is easily the most common of the three. Developed for use in wireless local area networks (WLANs), WiFi allows users to connect to the Internet without cables. In addition, WiFi can be used to create an ad-hoc mesh network amongst a number of devices. Typical WiFi networks have a range of approximately 300 meters in an outdoor setting [9]. In the European Union, it is illegal to transmit signals in the 2.4GHz band with strength greater than 20 dBm, so a range of 100 meters is more realistic [10]. The WiFi protocol sacrifices power efficiency for speed and is capable of data rates as high as 54 Mbit/s [9]. WiFi signals can be protected with WPA2 encryption, to which there is no known weakness. However, the transmit power of a WiFi device is 20 dBm (100 mW) and as a result WiFi transceivers require a lot of power. Since the transceivers must always be on to listen for incoming data, using WiFi transceivers in our camera nodes would drain the batteries too quickly, making WiFi an impractical solution [11].

Bluetooth (802.15.1) was designed to replace the cables connecting portable devices such as cell phones, keyboards, and MP3 players [12]. Three classes of Bluetooth devices exist, each differing in its maximum range and transmission power. Most devices are Class 2, which uses approximately 2.5mW of power and transmits at distances of up to 10 meters [12]. In order to keep power requirements low, Bluetooth sacrifices speed: it can only transmit at speeds up to 2.1Mbit/s, far slower than the 54Mbit/s offered by WiFi. Bluetooth devices form small, ad-hoc networks called piconets. A typical piconet can only contain seven Bluetooth devices, although

each device may be a member of more than one piconet. As such, larger networks are possible. However, the Bluetooth protocol was not designed for mesh networking; the devices in each piconet must be configured in a star formation. Since each piconet can only support seven devices and our system must support far more, multiple piconets would be required. The complexity of transmitting data through multiple piconets, in addition to the short transmission range of each device, makes Bluetooth an inadequate solution.

In December 2004, IEEE ratified the 802.15.4 protocol. This protocol describes the small, low-power radios that the ZigBee specification is based on. ZigBee was designed for use in low-cost, low power mesh networks that do not require fast data transmission rates. As such, ZigBee devices are only able to transmit data at speeds of up to 250 Kbit/s. However, most devices are capable of transmitting as far as 100 meters with 0 dBm (1mW) of power. In addition, ZigBee allows devices to sleep once they have associated with a network. As a result, many ZigBee devices have a battery life of months or even years [13]. ZigBee messages are protected with multiple levels of security to which there is no known weakness, making it a viable solution for industrial and security applications [14]. Like Bluetooth, the 802.15.4 and ZigBee protocols specify a very low transmission power: 0 dBm (1 mW). In addition, ZigBee was designed with mesh networks in mind. As a result, ZigBee networks are capable of supporting up to 64,000 devices in a single mesh network configuration. While the transmission rate of a ZigBee network is only 250 Kbit/s, we estimate that the size of each JPEG image produced by our cameras will be within 20-25 KB and at the very most 100 KB. Therefore, it would take each node eight tenths of a second to transmit a 25 KB image and a maximum of three point two seconds to transmit a 100 KB image to the next node. Since our data is not very time sensitive, this data rate should be sufficient. For these reasons, ZigBee is the best fit for our camera system. Table 1 shows a comparison of these three protocols.

Table 1: Comparison of the wireless protocols.

Protocols	Range	Data Rate	Power Consumption	Transmit Frequency	Channels
WiFi	90 m	54 Mbit/s	100 mW	2.4 GHz ISM	4
Bluetooth	10 m	2.4 Mbit/s	2.5 mW	2.4 GHz ISM	79
ZigBee	75 m	250 Kbit/s	1 mW	2.4 GHz ISM	16

4.2 Integration with MSP430

With the ZigBee protocol chosen, other considerations also needed to be taken into account before choosing specific components for our design. These considerations included discussing interface options with the camera team and choosing how the overall system would be configured. The camera team had chosen their microcontroller, the MSP430F1611 from Texas Instruments [15]. With this in mind, we determined three possible configurations for each node.

The first configuration, depicted in Figure 5, involves a single microcontroller that is responsible for the operation of everything in the system. This configuration requires a microcontroller that has enough resources to access and control a ZigBee transceiver, a camera, input sensors, and the system's memory. While some MSP microcontrollers have enough peripheral ports to accommodate these devices, the MSP430F1611 does not. Additional peripheral ports would need to be created in software. As a result, this configuration requires a lot more programming than the other two.

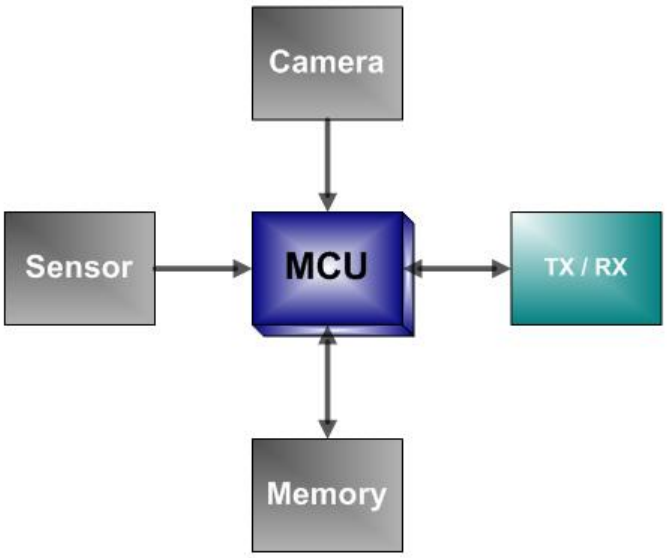

Figure 5: Single microcontroller configuration.

The second possible configuration, Figure 6, involves two microprocessors. The first processor is the MSP430, which is responsible for the camera, input sensors, and system memory; the other processor is responsible only for networking routines. Splitting these tasks allows the system to process image data while simultaneously transmitting data, something that

is not possible with the single-processor configuration. Unfortunately, interfacing two microcontrollers can be very difficult. In addition, one of the two processors would not have access to the system memory. This limitation would almost certainly result in much more overhead than necessary.

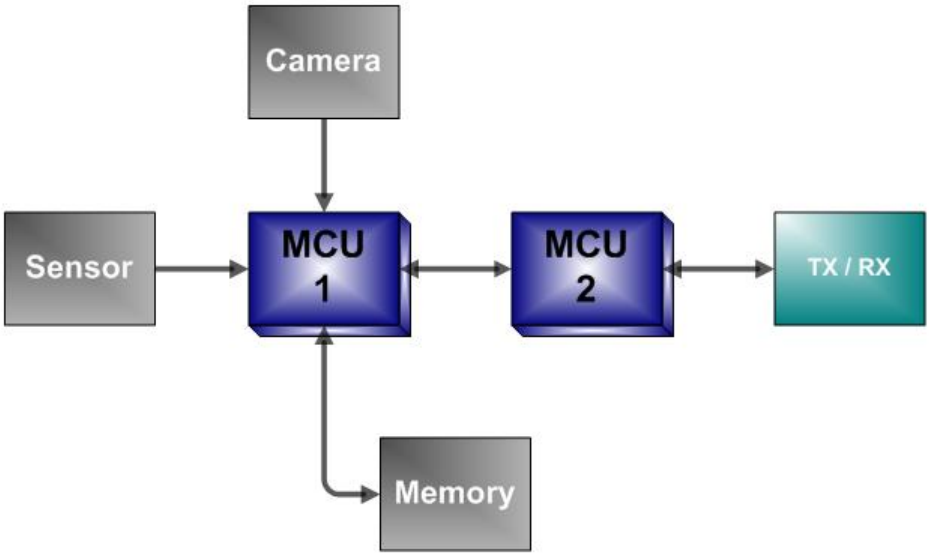

Figure 6: Dual microcontroller configuration.

A final configuration, Figure 7, utilizes a microcontroller and a networking co-processor. In this configuration, the MSP430 acts as a central processing unit and the co-processor as a peripheral device. The MSP430 is responsible for handling camera, sensor, and memory routines while the co-processor provides the system with networking capabilities. This configuration allows the system to process image data while simultaneously transmitting data, as in the two-processor system above, and does not have the system memory access limitation. The networking co-processor has no need to access the system memory because the central processor controls all of its functions. We decided to use this configuration method of using the MSP430 with a co-processor.

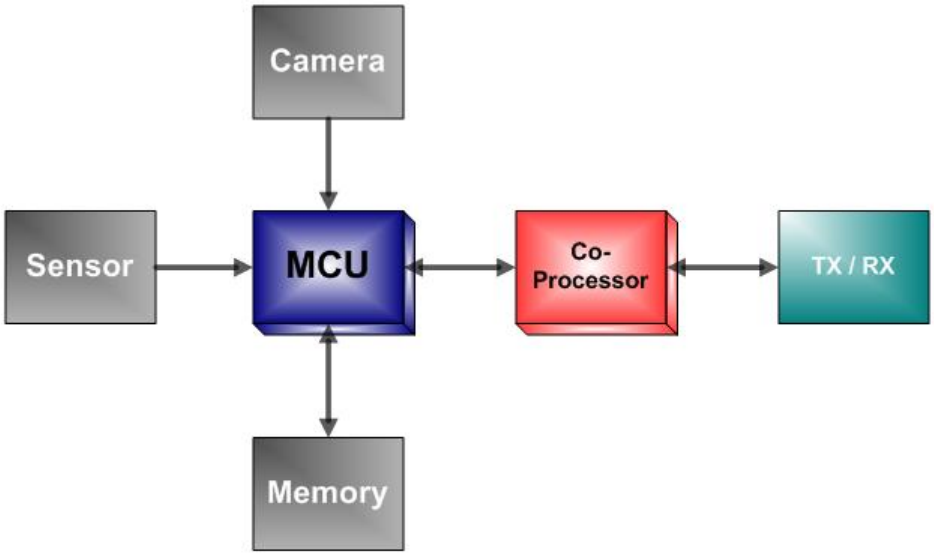

Figure 7: Microcontroller with a co-processor configuration.

In order to interface our co-processor with the MSP430, we needed to determine whether we would use the universal asynchronous receiver/transmitter bus (UART) or the serial peripheral interface bus (SPI). The MSP430 supports both busses and either would be suitable for our needs. The particular processor chosen, the MSP430F1611, has two peripheral ports. Each of these ports has enough pins to connect devices with UART and SPI simultaneously, though only one can be enabled at any given time. Our system has three peripherals (the camera, the system memory, and the co-processor), so it was necessary that two of them share a port. Both the camera and the co-processor need to read and write system memory (through the MSP430), so the memory needed its own port. The camera is only capable of using the UART interface, forcing us to use SPI for the co-processor.

With the system architecture decided, we next needed to choose our components. The most important piece of the system is the networking co-processor.

4.3 Co-Processor Selection: EM260

Based on our protocol selection and the configuration option chosen above, we needed to choose a ZigBee-enabled networking co-processor. This co-processor needed to have SPI support and ideally a low price and low power consumption. We chose the EM260 ZigBee co-

processor from Ember Corporation because it is the only ZigBee-enabled co-processor currently available [16].

The EM260 Radio Control Module (RCM), shown in Figure 8, is a full-featured networking node with an EM260 co-processor, an antenna, an activity LED, and all of the necessary external components. This module is capable of connecting to other microcontrollers via SPI, as required, and does not contain any unnecessary features. As a result, its cost and power consumption are both low. The EM260 RCM was priced at €88.00. A development kit containing ten RCMs, a development board, software, and support was also available from Ember Corporation for approximately €1800.00, which was beyond the budget of this project. We chose to purchase three EM260 RCMs for our proof-of-concept.

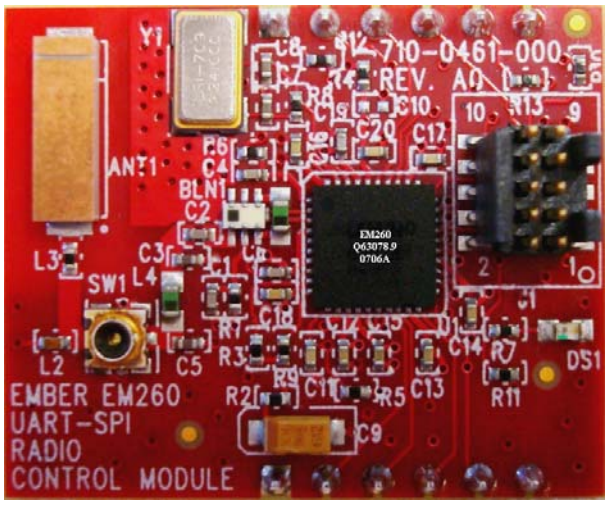

Figure 8: EM260 RCM.

4.4 Base Station Selection

The base station is the final destination for each image that is sent through the network. Because each of the nodes in the system utilizes ZigBee technology, the base station too must be ZigBee-enabled in order for it to communicate with the nodes. Since the nodes contain EM260 chips from Ember Corporation, using an EM260 (or similar chip) in the base station would greatly reduce compatibility issues. Knowing this, we initially designed the base station to use an EM250 ZigBee processor that would communicate with the PC over a USB interface. However, due to compatibility issues with different versions of the ZigBee stack software, we needed to modify this design. The final base station contains an MSP430 and an EM260 and

connects to the PC with a serial cable. The following sections describe both the initial and final designs and the reasons for each of the design decisions.

4.4.1 Telegesis ZigBee ETRX2USB

The first design for the base station included the Telegesis ETRX2USB stick shown in Figure 9 [17]. The ETRX2USB interfaces with a standard PC over USB and allows it to communicate with other ZigBee devices. The hardware for the ETRX2USB includes the EM250, which is very similar to the EM260 except that it is a full system-on-chip solution instead of a co-processor. This provides a way to interface the network to a personal computer using Ember technology, which is the reason that we considered it as a base station solution. Unfortunately, the ETRX2USB contains version 2.5 of Ember's ZNet ZigBee stack and the RCMs contain version 3.0, which is not backwards compatible. As a result, we were unable to use the ETRX2USB in our proof-of-concept. However, Telegesis is currently in the process of upgrading to version 3.0, which will be fully compatible with our camera nodes. Refer to Appendix I for the base station code.

Figure 9: Telegesis ETRX2USB.

4.4.2 MSP430 and EM260 RCM

Unable to use the ETRX2USB, we needed to find a new solution for our base station. The development boards that the MSP430 processors came with had a serial port that we could utilize to connect a PC via UART. Since we knew that the EM260 and MSP430 were compatible, we determined that a solution similar to that used for the camera nodes would work. This new base station configuration, shown in Figure 10, contains the same MSP430F1611 and EM260 RCM as the camera nodes do, but lacks the additional peripherals (camera, system memory, sensors).

This solution is not quite as simple and elegant as our initial design. Modern PCs typically have an abundance of available USB ports, but lack the older serial ports that this solution requires. In addition, the MSP430 board is rather large, unlike the ETRX2USB. However, this solution is the best alternative until Telegesis finishes updating its software to version 3.0 of the EZNet ZigBee stack.

Figure 10: MSP430 interfaces with the EM260 and the PC.

5 ZigBee Wireless Protocol

This chapter will provide an overall background for the ZigBee protocol and how it is used in the project. Much of the routing, as will be seen, is handled by ZigBee, which allowed the project to utilize mesh networking.

5.1 ZigBee Description

Most wireless communication protocols until recently have focused on creating high speed and long range devices. However, the ZigBee Alliance has been taking a different approach. The ZigBee Alliance is a group of ninety companies which has been developing a wireless network standard for residential and commercial control [13].

The best way to describe ZigBee technology is using the seven layer OSI model which is shown in Appendix C. The bottom three layers: Physical, Data Link, and Network are produced by the ZigBee Alliance, along with the part of the application layer, which allows customers to develop their own custom applications using the capabilities of the bottom three layers. The ZigBee Alliance chose to use IEEE 802.15.4 standards or low-rate PANs. The IEEE 802.15.4 uses the Industrial, Scientific, and Medical (ISM) radio bands: 868 MHz, 916 MHz, and 2.4 GHz. The frequency band used in this application was the 2.4 GHz which offers a data rate of 250 Kbps. Applications using ZigBee typically have an indoor, non line-of-site, range of thirty meters and an outdoor line-of-site range of eighty meters [18].

The data link layer is the layer which specifies how ZigBee communicates between nodes. The IEEE 802.15.4 protocol, which is based on Carrier Sense Multiple Access (CSMA), is used on the data link layer. CSMA is a protocol in which nodes communicate with the next node to verify its existence before sending data. This means that when packets are being sent, nodes first check to see if there is a node that will receive the data. When data is sent there is also an acknowledgement returned to the sender to indicate a successful transmission of the data.

Much of the new development within the ZigBee Alliance has been within the network layer. This is the layer which allows ZigBee to have its mesh networking capabilities. The ZigBee network layer is responsible for the protocol for when nodes leave and enter a network and also the routing protocol for packets making their way to its final destination.

However, there are certain limitations to the Zigbee protocol. The physical layer only supports transmission of packets limited to 127 bytes. Also with the overhead, MAC and physical layers, each packet can only contain up to 89 bytes of application data. This means that information being sent in the application must be broken up into packets and reassembled which is not done in the ZigBee protocol [19]. For more information on the IEEE 802.15.4 standard can be found in Appendix B.

5.2 ZigBee Applications

Within a ZigBee network nodes have three different responsibilities: coordinators, routers, and end devices. Each ZigBee network has exactly one coordinator, which is responsible for establishing and maintaining the network. The EmberZNet libraries allow any ZigBee device to act as coordinator. The coordinator is selected through application code routines which allow the device to assume the roll of a coordinator. In our network, the base station acts as the network coordinator. Routers are fully functional ZigBee devices that can talk to any other ZigBee device within its range. Each of the camera nodes in our network acts as a ZigBee router, enabling them to communicate as a true mesh network. ZigBee end devices have limited functionality and are not capable of routing. As a result, end devices will not be a part of our network. Each of the nodes on our network (the coordinator and all of the routers) are fully functional ZigBee devices, running complete ZigBee software stacks.

To begin a ZigBee network a device first must be appointed a coordinator. If there is no coordinator in operation there can not be a network. Once a network is started then other ZigBee devices can join this network. Upon being powered, the ZigBee node will actively begin searching for existing ZigBee networks by sending out broadcasts. When the ZigBee device detects a network, it has to verify that the network detected is the correct network [20]. To do this, the node will attempt to associate with the closest router or the coordinator in that network. If successful, two devices will begin the authentication procedure. Authentication procedures can be done in different ways by the user and they are defined in the ZigBee specification. For our devices, this authentication of a network will be determined by an encryption key. Should this process succeed, the node will become a member of the network, in other words it will bind to the network. Binding always will happen after a communication link is established between

the nodes already in the network and the additional node [20]. Once binding is completed, routing tables will then be generated which is provided in the three bottom layers of the OSI model as discussed before. Routing tables are formed when a node sends out a routing request. In our case the routing request would be sent through the network to the base station when a node first powers up and also before attempting to send data through the network. These routing tables are what allow packets to be sent back to the coordinator through a routing path. To remove a node from the network, a node is simply turned off. Neighboring nodes will then recognize the dead link and generate new routing tables for the network.

There are several routing methods that can be found in the ZigBee specification. The user can choose which one to use and this selection is made as a part of the network design. EmberZNet supports three types of routing procedures: mesh, multicast, and broadcast. In our project, the routing procedure used was mesh. In the mesh procedure, routes are formed when one node sends out a request to find a path to another node, which is done until the route taken reaches the coordinator. When this path is formed, the initiating node sends the data to the first node in the path. The intermediate nodes between the source and the destination use their own routing tables to relay the data. However, if a route fails, an error message is sent back to the source of the data and a new request for a path to the destination is created. The other two routing procedures involve sending the data to many nodes at the same time, which makes the system less efficient and increases its power consumption [20]. For more information on mesh networks refer to Appendix A.

Once a device is associated with a network, it can begin transmitting and receiving messages. There are four ways to transfer data in a ZigBee network: direct addressing, indirect addressing, broadcast addressing, and group addressing. In direct addressing, the transmitting node specifies the address of the node that it wants to transfer data to and the source. On the other hand, in indirect addressing the node does not know the exact address of the destination node and has to make use of the routing tables. Broadcast addressing and group addressing both send messages to multiple recipients [21]. We used the direct addressing because the address of each node is known. Since our application involves sending data that is greater than 127 bits, we needed to divide it up into smaller packages. To not to lose data, we needed to ensure that these packets arrive at the destination in a timely and an orderly manner. The ZigBee stack does not ensure an in-order message delivery or filtering of duplicate packages. These need to be taken

care of by the user. That is why the message also included a sequence number. This sequence number is initialized to a random value and incremented with each message transmission. These three values allow the ZigBee stack to determine where to send messages and in what order any receiving devices should consider received transmissions. If an application needs to send a message, it passes a request to the Network Layer (as well as the payload of the transmitted message). Once the message has been constructed in the network layer, it passes through the security sub-layer and finally to the MAC layer for transmission.

Upon receiving a message, a ZigBee router (or coordinator) evaluates the parameters described above to determine whether to retransmit the message. If the message was not intended for that specific router or coordinator then the message must be retransmitted, but if it is then the message will continue to the end unit. In the latter case, the ZigBee router will pass the received message through its security sub-layer and eventually to its network layer [21]. Here, the message may be used by an application. In our network, all messages are sent to the network coordinator (the base station). As a result, the software for evaluating the payload of each message need not be implemented on the other devices. The procedures for transmission, security, reception, and routing are all defined in the ZigBee specification.

In order to ensure proper delivery ZigBee employs a number of technologies to ensure that messages are delivered. More detail is found in the following chapter when describing our process of ensuring message delivery. In addition to the addressing and sequence numbers described above, ZigBee devices send acknowledgment packets to the sender of any messages that they receive. Before transmitting a message, the user needs to set the acknowledgement flag to '1' if he desires an acknowledgement packet. If this flag is set to '1', the receiver sends out a packet back to its source address upon reception of a packet [21]. If a transmitting device does not receive such a packet, it considers the transmission a failure and attempts it again for a previously determined amount of time. If the transmitter does not receive any acknowledgement during this time, it considers transmission a failure and stops transmission [21]. For our application, this time is set to 200ms, which gives the nodes ample time to send everything they have. The more detailed procedures for generating, sending, and reading these acknowledgment packets are all defined in the ZigBee specification. Also ZigBee provides a128-AES encryption. For our devices, the encryption keys are generated on a PC and then programmed into each

ZigBee node. As a result, any one of our devices can communicate with any other device on the network, but outside devices will not be able to participate in the network.

Most of the network processes are accomplished using the EmberZNet ZigBee software. However, there are still some applications that need to be done by the user:

- Defining callback functions: Callback functions are called once a certain process is completed like reception or transmission of a message.
- Setting up a main program loop: SPI is initialized in the main program loop and all the necessary functions are kept there.
- Managing network associations: Even though ZigBee software handles detection of networks, the type of routing must be specified by the user. After a network scan, user has to call the functions that would allow the node to become a part of the network.
- Message protocol: The user defines how the message is to be transmitted in the network and how to process it once it is received.
- Housekeeping tasks: The user keeps track of the processes in the co-processor as well as the host device managing the input/outputs and the clock. [20]

Figure 11 summarizes which processes are handled by the application layer, the EmberZNet stack, and the 802.15.4 protocol. As can be seen, EmberZNet ZigBee software handles most of the network functionalities and the user has to add in some application functions to have a functional ZigBee network.

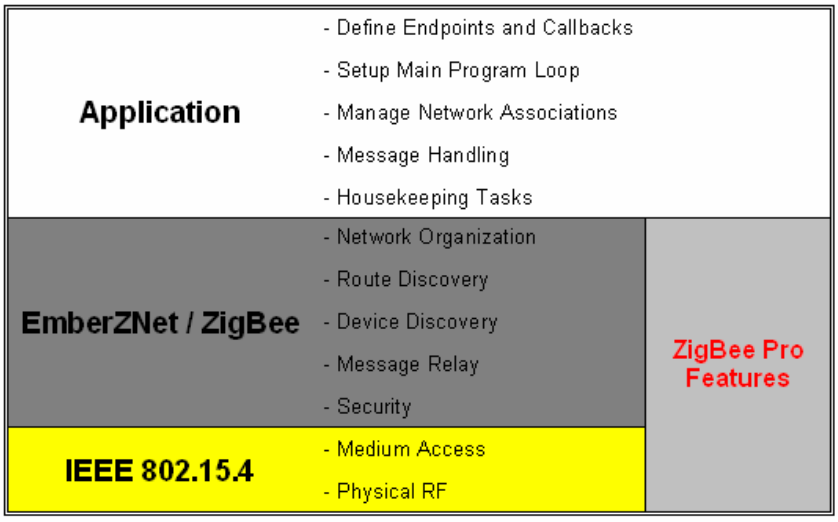

Figure 11: EmberZNet Model.

6 System Structure

Figure 12 shows the overall system layout for the wireless security camera system. The diagram displays each of the components used in the design of a node with all of the power lines and the signal lines that connect them. The camera team was responsible for the solar panel, charge controller, battery, MSP430, SD memory card, PIR sensor, and the camera module. Our team was responsible for setting up the network and the transceiver module.

When the PIR sensor detects motion, it triggers the camera to take pictures. These pictures are then transmitted to the base station via the EM260, where they can be viewed at a later time.

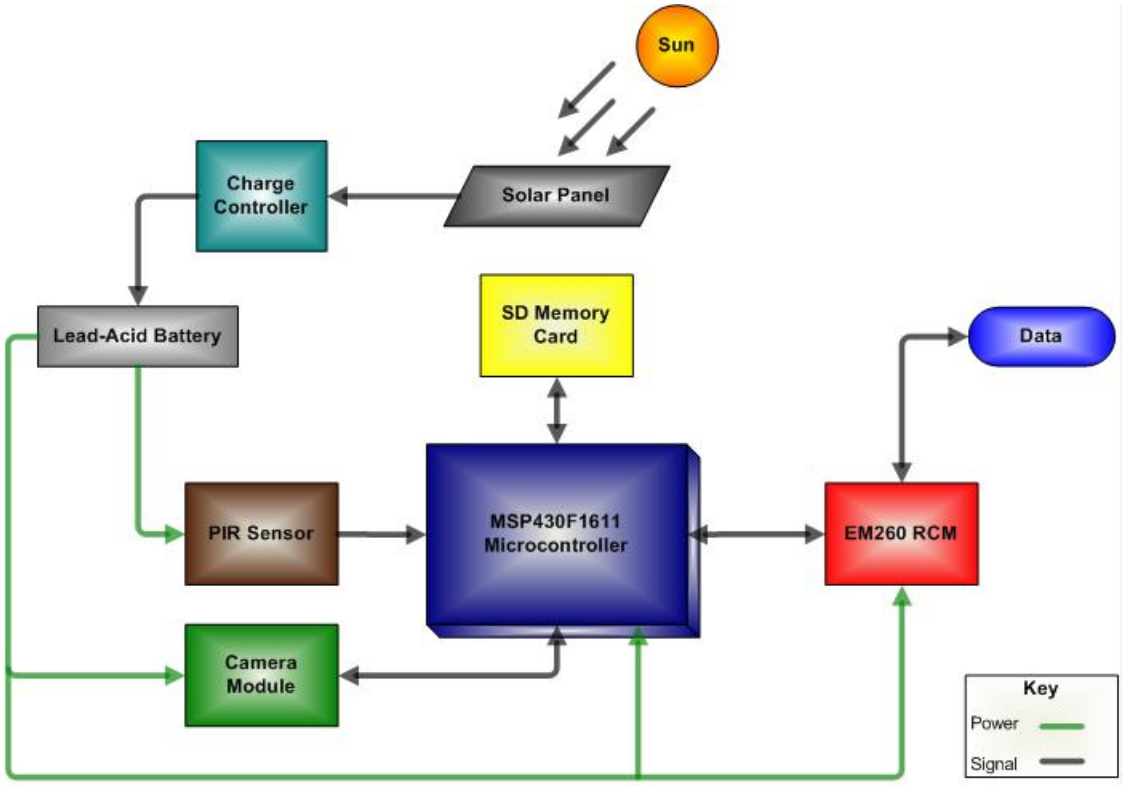

Figure 12: Node system structure.

Figure 13 represents the base station. This consists of the MSP430, the EM260 RCM, and a standard PC that runs windows. A six volt adaptor is used to power the base station. The images that are transmitted over the network end up at the base station. Here the images are stored in a folder on the computer. From here, images and log files can be uploaded to the Internet for remote viewing or simply opened on the PC for immediate access.

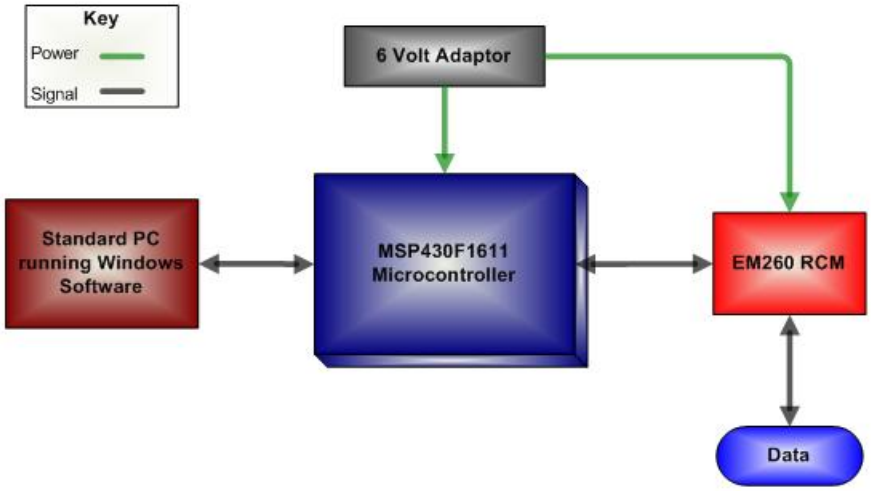

Figure 13: Base station system structure.

6.1 Network-Level Design

The network only contains one input, which is motion. Any node in the network can detect motion. The output of the network includes JPEG images. Figure 14 shows the inputs and outputs, as well as the overall flow of the system.

The first event that begins all processes in the system is the detection of motion. When a node detects motion, it takes a series of pictures and stores them in memory. The node then transmits these images as a series of packets, relying on the ZigBee protocol to help them reach the base station. Essentially, each packet is forwarded node by node until they reach the base station. When it arrives, the base station generates and sends an acknowledgment packet to inform the node that a packet was received. This process is repeated until all of the packets of an image have been sent. While this system does introduce some added overhead to the network protocol, it ensures that all packets arrive in order regardless of the route they take.

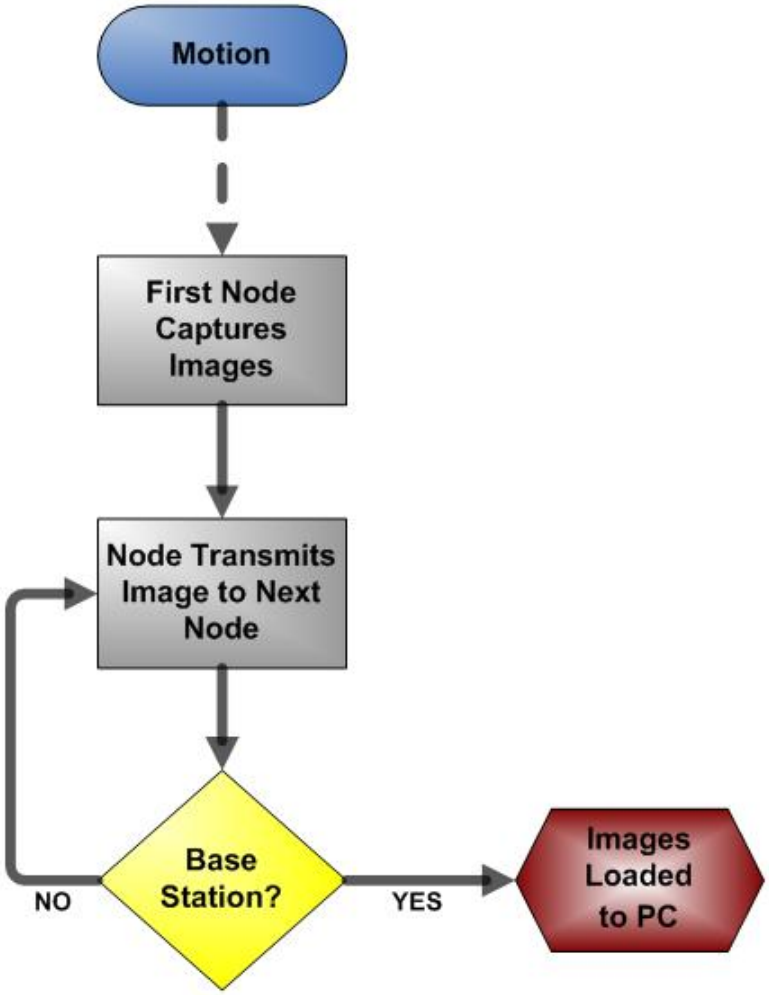

Figure 14: Network level design.

6.2 Node-Level Design

Nodes accept two forms of input: motion, and data packets. There is one type of output that nodes can produce: data packets. Figure 15 outlines the overall flow of a node.

The events that trigger the activation of a node are the detection of motion by the PIR sensor and the reception of a packet from another node. In the event that a packet is received from another node, the system receives the packet and forwards it to the transmitter to be transmitted to the next node in the system. The packet continues to be transmitted through the network until it reaches the base station. The EZNet ZigBee stack that's programmed into the EM260 co-processors handles this routing process.

In the event that the PIR sensor detects motion, the camera captures an image and stores it in memory. The node then sends an "Image Ready" signal through the network to the base station and waits for a "Send Image" signal from the base station. Upon receiving the "Send Image" signal form the base station the images are then taken from storage and divided up into data packets. The first packet is sent from the node through the network and the node waits for acknowledge from the base station that it has received the packet. When the acknowledgement is received from the base station of a successful transmission, the next packet is sent and the node waits again for conformation from the base station. The node follows this same process for each packet until it receives a confirmation for the last packet of the image. Once this image has been sent to the base station the node returns to an idle state and waits for the next input.

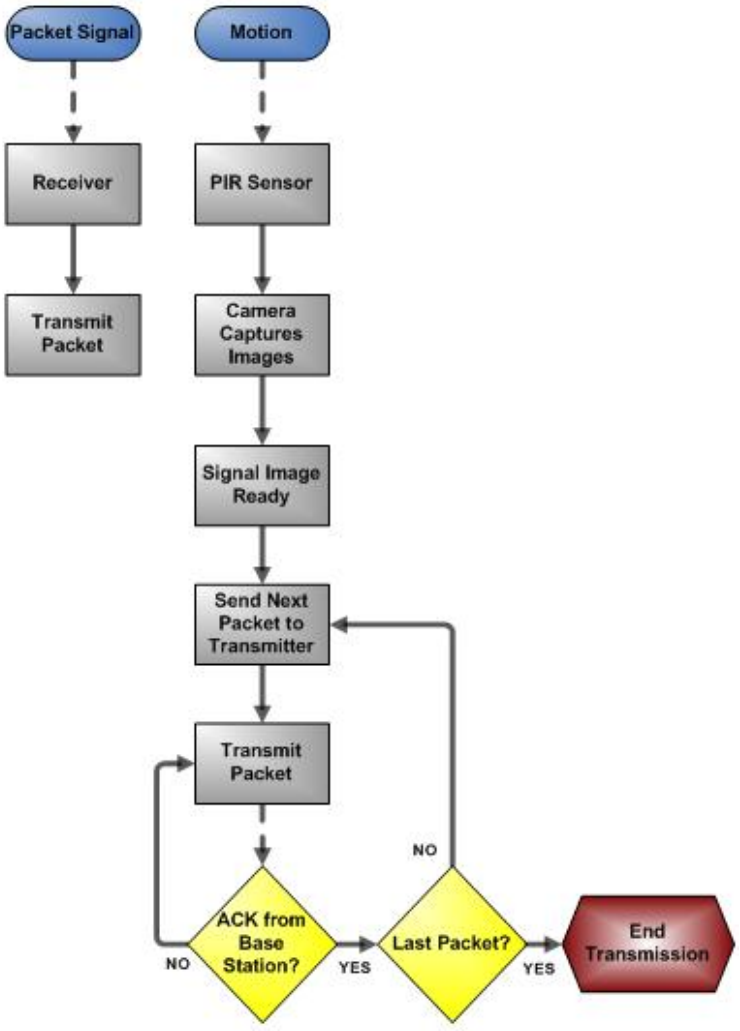

Figure 15: Node level design.

6.3 Base Station Design

The base station is the end node in the system. There is only one input to the base station: data packets. Likewise, the base station can produce only one output: JPEG files. Figure 16 shows the overall flow of the base station system.

The base station remains idle until it receives a data packet from another node in the network. If this data packet is not the "Image Ready" signal, the base station simply ignores it. However, if it is the "Image Ready" signal, the base station begins the process of creating a new JPEG file. Once the new file has been created, the base station transmits a "Send Image" signal back to the node to let it know that it can begin sending the image data. As each packet is received, the base station writes its contents to the open image file and sends an acknowledgment signal back to the sender. Upon receiving the last packet, the base station closes the file and releases it for viewing.

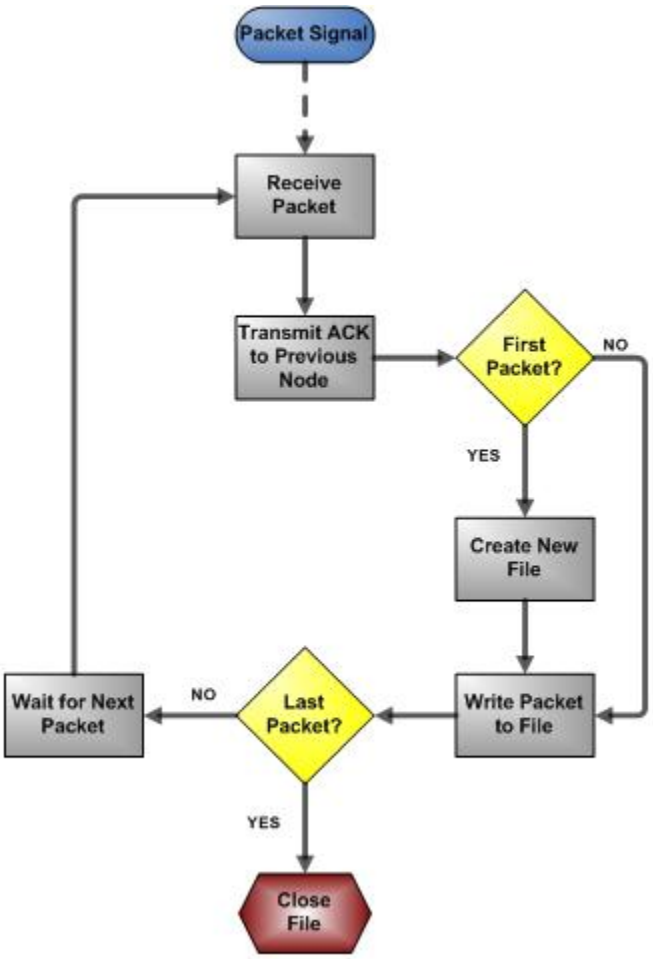

Figure 16: Base station level design.

7 System Integration

This chapter discusses the components selected for both the camera nodes and the base station, and the steps taken to construct them. The first section discusses how the components were physically assembled in both the nodes and the base station. The second section discusses the programming aspect for each of the components.

7.1 Hardware Design

The following section describes both the EM260 RCM and the MSP430F1611 and the physical connections made between them. The physical connections made between the EM260 and the MSP430F1611 were similar for both the base station and the camera node in the system because they both were connecting to the MSP through the Serial Peripheral Interface (SPI). However, the hardware on the node differs slightly because it includes an SD memory card, a camera, and a PIR sensor while the base station has none of these components.

7.1.1 EM260 RCM

The EM260 RCM contains two 6-pin connectors for access to the EmberZNet Serial Protocol (EZSP), shown in Figure 17, which allow for connection to a host microcontroller by UART or SPI. Table 2 gives the general description for each pin when using SPI, which is the connection made to the MSP430F1611 in this project.

Figure 17: Two SPI 6-pin connectors on EM260 RCM.

Table 2: SPI pin descriptions for EM260 RCM.

Pin	Signal Name	General Description
1	VBRD	2.1 to 3.6V DC power supply for RCM
2	MOSI	SPI data, master out/slave in (from host to EM260)
3	MISO	SPI data, master in/slave out (from EM260 to Host)
4	SCLK	SPI clock (Host to EM260)
5	nSSEL_INT	SPI mode: Active low SPI slave select (Host to EM260)
6	GND	Ground connection
1	TXD	Not Connected (UART only)
2	RXD	SPI mode: Host interrupt (from EM260 to Host)
3	nWAKE	Wake interrupt (from Host to EM260)
4	nRESET	Active low chip reset (internal pull-up)
5	nRTS	Not Connected (UART only)
6	GND	Ground Connection

7.1.2 MSP430F1611

The MSP430F1611 contains two peripheral ports that can be used to connect other devices with UART or SPI. The MSP430 that was purchased by the camera team came on a preassembled breakout board which can be seen in Figure 18. The SPI pins which were used to connect to the EM260 RCM to the camera node are shown in Table 3 with their general description; the pins used to connect the base station through SPI have the same general description except they are on port five instead of port three. These pins are accessed using the row of pins on the center of the Olimex board.

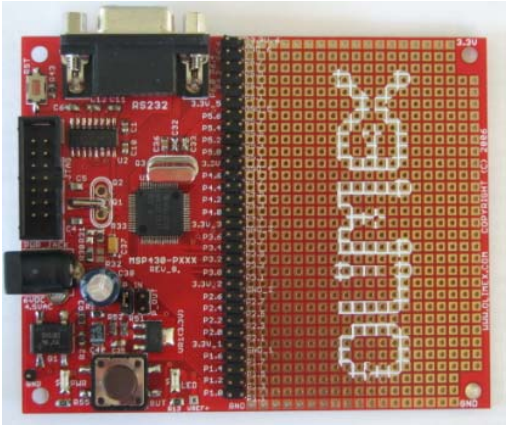

Figure 18: Olimex MSP430F1611 Board.

Table 3: SPI pin descriptions for MSP430F1611.

Pin	Signal Name	General Description
3.3v_2	Power	3.3V
P3.1	SIMO	General-purpose digital I/O pin/slave in/master out
P3.2	SOMI	General-purpose digital I/O pin/slave out/master
P3.3	SCL	General-purpose digital I/O pin/external clock input
P3.0	STE1	General-purpose digital I/O pin/slave transmit enable
gnd_2	Ground	Ground Connection
P1.3	TA2	General-purpose digital I/O pin/Timer_A
P4.0	TBCLK	General-purpose digital I/O pin/Timer_B
P4.1	TB6	General-purpose digital I/O pin/Timer_B
gnd_3	Ground	Ground Connection

7.1.3 Pin Assignments

In order to physically connect the two chips the correct pins needed to be matched. Table 4 displays which pins on the EM260 RCM were connected to which pins on the MSP430F1611. However, the base station and the camera node are connected to the EM260 RCM through different ports due to the fact that when the MSP430F1611 is connected serially to a PC it gains access through port three. As Table 4 shows, the base station utilizes pins in port five to

function, while the camera nodes utilize the pins in port three. Refer to Appendix D for the hardware design of the camera node and the base station.

Table 4: Pin connections between EM260 RCM and MSP430F1611.

EM260 RCM		MSP430F1611			
		Camera Node		Base Station	
Pin	Signal Name	Pin	Signal Name	Pin	Signal Name
1	VBRD	**3.3v_2**	Power	**3.3v_4**	power
2	MOSI	**P3.1**	SIMO	**P5.1**	SIMO
3	MISO	**P3.2**	SOMI	**P5.2**	SOMI
4	SCLK	**P3.3**	SCL	**P5.3**	SCL
5	nSSEL_INT	**P3.0**	STE0	**P5.0**	STE0
6	GND	**gnd_2**	Ground	**gnd_2**	ground
1	TXD	**N.C.**	not connected	**N.C.**	not connected
2	RXD	**P1.3**	TA2	**P1.3**	TA2
3	nWAKE	**P4.1**	TB1	**P4.7**	TBCLK
4	nRESET	**P4.0**	TB0	**P4.6**	TB6
5	nRTS	**N.C.**	not connected	**N.C.**	not connected
6	GND	**gnd_3**	ground	**gnd_3**	ground

7.2 SPI with EM260 RCM

Before programming the chips we first needed to understand how to communicate with the EM260 RCM. The SPI transaction for the EM260 is half-duplex meaning either the master is sending data or the slave is sending data. Both can never happen at the same time. The basic transaction is composed of three different sections which are called command, wait, and response. The command section begins with the host, the MSP430, signaling the slave select pin (nSSEL_INT) and then sending a command to the EM260 which can be any length between 2 and 136 bytes, but cannot begin with 0xFF. During the command section the EM260 will be responding with 0xFF on the MISO pin. Once the host has completed its message the wait section begins. The wait section, which can last no longer than 200 milliseconds, is the period

when the EM260 is processing the command from the host. During the wait section the EM260 continues to respond with 0xFF. When the EM260 is ready to respond it will respond to the host with something other than 0xFF, which indicates an official transition from the wait section to the respond section. The data format for the response section is the same as that of the command section. Once the response is given, the host must then de-assert the chip select pin. Figure 19 displays a typical SPI transaction.

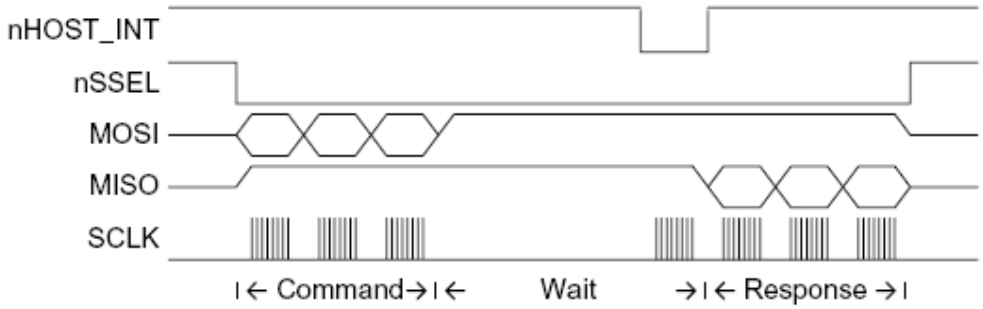

Figure 19: SPI transaction for EM260 RCM [16].

7.3 Base Station Programming

The base station's main functionality consists of forming networks, receiving images from the camera nodes, and storing those images on the PC. To accomplish these tasks, the MSP430 needed to be able to communicate both with the PC and the EM260. The pseudo-code below shows the general flow of the program.

```
main()
{
  initMSP();           // initializes MSP430
  initUART0();         // initializes UART
  initSPI1();          // initializes SPI
  initEM260();         // initializes EM260

  networkInit();       // initializes network

  /* If in a network already, leave it.  It might
   * not be the right network.
   */
  if(inNetwork)
  {
    leaveNetwork();
  }

  /* Stay in this loop until powered off, so that
```

```
     * the MSP430 never resets.
     */
    while(1)
    {
      /* If we have a message from the PC, figure out
       * which message it is and then handle it.  We
       * only accept three commands; others are ignored.
       */
      if(messageFromPC)
      {
        if(message = "Establish Network")
          establishNetwork();
        else if(message = "Leave Network")
          leaveNetwork();
        else if(message = "Send Unicast")
          sendUnicast();
        else
          continue;
      }

      /* The only messages that we get from the EM260
       * are messages that we should pass to the PC.
       * So, we format them and pass them along.
       */
      if(messageFromEM260)
      {
        formatMessage();
        sendMessageToPC();
      }

      continue;
    }
  }
```

The first two functions in the program initialize the MSP430 and enable UART so that it can communicate with the PC. The next two functions initialize SPI and finally the EM260. Once these procedures were completed, we could begin sending and receiving messages between the PC and the EM260, using the MSP430 as an intermediate. In this way, the PC can control the formation and destruction of networks and the EM260 can deliver image data to the PC.

At this point, the EM260 is not part of a network. In order to rectify this problem, we call a special function named networkInit(). This function rejoins the EM260 to the network that it was a member of when last powered off, or informs us that the EM260 was not part of a network previously. In either case, we do not wish to remain on an old network so we immediately call a leaveNetwork() function. We remain disconnected from all networks until instructed by the PC to form a new PAN.

When the MSP430 receives the signal to form a new network, it passes the request to the EM260, which immediately begins to scan various channels to determine which has the least interference. Once an appropriate channel is found, the EM260 establishes a new network and

informs the PC of its success. Should this call fail, the PC is informed so that it may again send the request to from a network.

Once the systems are running and the network is formed, the MSP430 becomes nothing more than a message relay device. The PC can signal it to leave its current network, form a new network, or send a message to one of the camera nodes. The EM260 can deliver any message to the PC. The pseudo-code below displays the general flow of the program written for the PC.

```
main()
{
  initCOM();   // opens the serial port
  createLog(); // creats a log file

  /* We instruct the base station to leave its network
   * and form a new one.
   */
  leaveNetwork();
  establishNetwork();

  /* Receive messages from the EM260.  If we get an
   * image ready, we inform the EM260 that we're ready
   * to accept image data.  We then accept the image
   * data, close the file, and return to waiting.  If
   * the EM260 does not relay any data for 10 seconds,
   * we time out and return to the top of the loop.
   */
  while(1)
  {
    if(messageFromEM260)
    {
      writeMessageToLog();
      if(message = "Image Ready")
      {
        createImageFile();
        sendMessage("Send Image");
        while(notFinished && notTimeOut)
        {
          receiveNextDataPacket();
          writeDataToFile();
        }

        closeImageFile();
      }
    }
  }
}
```

Once the MSP430 signals that it has initialized its systems and that it is not on a network, the administrator may start the PC software. This program begins by signaling the EM260 to leave its network, if it is on one. By doing so, we ensure that the EM260 is not on any networks regardless of when it was last rebooted. Next, the PC instructs the EM260 to form a new PAN. Once successful, the PC enters its main loop. Here, it accepts messages from the EM260. Should it receive an "Image Ready" signal, it enters a sub-routine where it accepts image data from the EM260. The PC remains in this sub-routine until the entire image has been received or

the connection to the node sending the image is lost, after which it returns to waiting for messages from the EM260.

7.4 Camera Node Programming

The camera node contains an SD card and a camera, and must send images through the network. The main section of code written for the camera node is shown below. The bolded and red text is used to indicate the functions that the mesh team wrote and the text in blue indicates the code that was written by the camera team.

```
main()
{
  initMSP();           // initializes MSP430
  enableSPI0();        // enables SPI0
  initEM260();         // initializes EM260
  disableSPI0();       // disables SPI0
  enableUART0();       // enables UART0
  initSystems();       // initializes camera, SD
  disableUART0();      // disables UART0

  /* We enable SPI0 and tell the EM260 to join
   * a new network.
   */
  enableSPI0();
  joinNetwork();
  disableSPI0();

  /* From now on, we stay in this main loop.  The
   * MSP enters a sleep state until it is woken by
   * the PIR sensor.  Then, the camera takes pictures
   * and the EM260 transmits them to the BS.  Finally,
   * the MSP goes back to sleep.
   */
  while(1)
  {
    sleep();
    takePictures();
    for(each picture)
    {
      sendPicture();
    }
  }
}
```

Like the base station, the camera node begins by initializing its systems. Unlike the base station, we cannot simply initialize UART or SPI. Instead, UART and SPI we must enable and disable them as necessary. This method allows the camera and EM260 to share a single peripheral port on the MSP430.

Once everything has been initialized, the MSP and camera enter a low-power sleep state to conserve battery life. The EM260 remains active, as it must be able to relay messages from other nodes. The PIR sensor also remains active and, upon sensing motion, wakes the rest of the system.

When the system is awake, the camera takes a number of pictures and saves them to the SD memory card. The size and starting location of each image is stored for later access. Once the camera is finished, the EM260 prepares to send the new images. If it is not already on a network, or if the base station is not responding, it will attempt to join a new network. Once successful, the EM260 sends an "Image Ready" signal to the base station, indicating that it has data to send. Once the base station replies with a "Send Image" signal, the EM260 sends the image data in 64-byte packets until the entire image has been transferred. This process is repeated for each image stored on the SD card. When there are no more images to send, the MSP returns to its low-power sleep state and the image pointer is set back to zero, effectively deleting any data stored on the memory card.

The code presented in the section is a very basic overview of the more important code written to create a functioning project. The actual code written for this project is found in Appendices E, F, G, and H.

8 Testing and Results

This section describes the tests that we performed to test the functionality of the proof-of-concept. These procedures test the basic functionality of the system as well as its performance.

8.1 Basic Functionality

The first test that we performed demonstrates the following:

- The base station is able to form a ZigBee network
- The camera node is able to join the base station's network
- An image can be transmitted over the network

Before performing this test, we loaded an image onto the SD memory card and recorded its size. We then placed the base station on one end of a table and a camera node on the other end of the table. We powered on the base station and instructed it to form a new ZigBee network. Then, we turned on the camera node and waited for it to join the base station's network. A few moments later, we received a notice that the node was ready to transmit an image to the base station. The base station responded with a "Send Image" signal and the node began sending data. A few seconds later, the base station notified us that the transfer was complete. We compared the size of the received image with the size of the original, and they matched. The image that we received looked exactly like the image that we loaded onto the SD card, indicating that our test was a success.

8.2 Range

With the basic functionality of the proof-of-concept established, we needed to test the system's performance. The first performance test involved both range and speed. We set up a camera node and the base station following the same procedure used in the initial test, with one exception: we performed this test in an open field, rather than on a table. This environment allowed us to transmit data at greater ranges than before. We performed tests at distances ranging from 10 to 180 meters. These tests allowed us to measure how long it took to transfer an

image at various distances and to determine whether any data loss would occur because of the long distance separating the node and the base station. Table 5 summarizes the results of these tests.

Table 5: Sending data in an open field.

Data sent in open field	
Distance (meters)	**Time (seconds)**
100	15.282
130	19.625
150	18.906
180	15.516

The open field tests showed that the maximum distance between two nodes in the best-case conditions was one hundred and eighty meters. Since the detection range of the PIR sensors on the camera nodes limits the distance between them to approximately 20 meters, the range of our transmitters is more than sufficient. Figure 20, below, displays the images that were transmitted at distances of ten meters and one hundred and eighty meters. Clearly, there is no quality loss resulting from of the longer distance. The time taken to transmit images at different ranges was nearly constant. This result is likely a side effect of the delays programmed into the MSP430 to prevent errors in the SPI exchanges. The time to transmit an image without these delays would likely be much less and would vary more with distance.

Figure 20: These images were taken from two distances, ten and 180 meters, respectively. There is no difference in quality.

In order to obtain a more accurate test of real-world performance, we repeated the previous tests in a car park at the University. Signal quality was not as strong in the car park as it was in the field because of the vehicles, pedestrians, and other obstructions present in the area. Table 6 summarizes the results of these tests.

Table 6: Sending data in a car park.

Data sent in car park	
Distance (meters)	**Time (seconds)**
10	19.313
20	19.313
30	18.484
40	19.328
50	16.265
60	15.266
70	15.344
80	20.406
90	15.36
100	unsuccessful

The car park test showed that the maximum distance between nodes in the realistic conditions of a car park was ninety meters. This distance still more than fulfills the twenty meters required for the actual security camera system. As before, we noticed that the time taken to transmit the images in regards to the distance varied little. Again, this is likely a result of the delays intentionally coded into the MSP430 to prevent SPI transaction errors. In an ideal setting, transmission time should vary directly as a result of the distance between two nodes. Figure 21, below, displays the images that were transmitted at distances of ten meters and ninety meters. Again, there is no discernable difference between the two images.

Figure 21: These images were taken from two distances, ten and 90 meters, respectively. There is no difference in quality.

8.3 Two Node Network

Since networks typically contain more than one node and a base station, we needed to test our system with multiple nodes. To do so, we loaded pre-loaded two camera nodes with images and set up them in the same way that we did in our initial tests. Once both nodes were powered on and joined to the base station's network, we pressed a button on one node that instructed it to send its image to the base station. Once it finished, we instructed the second node to do the same. Both images were received by the base station, indicating that the two nodes did not interfere with one another.

8.4 Multi-Hop

In order to test the routing and node recognition features of the system, we needed to form a network that contained a camera node that was not in range of the base station. To do so, we placed one node within range of the base station and a second node further out, as shown in Figure 22. We established a network on the base station and had the first node join. Then, we attempted to join the network with the second node. This was successful and demonstrated that the network could automatically recognize new nodes. Next, we attempted to send preloaded data to the base station. The data arrived as expected, demonstrating that routing functions as desired.

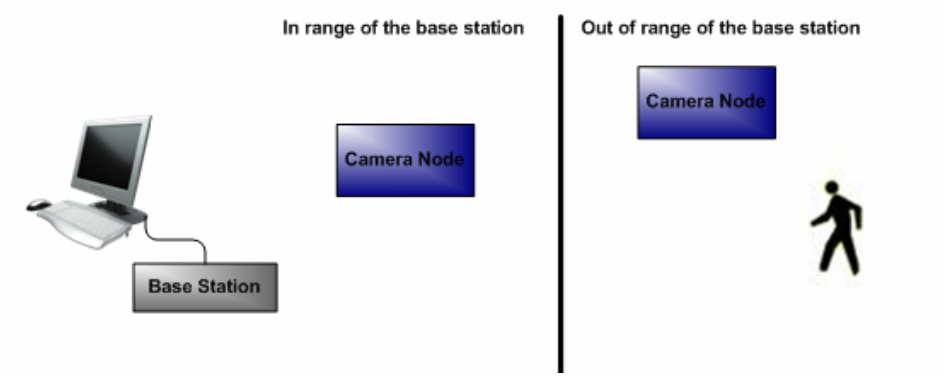

Figure 22: Multi-hop test.

9 Final Proof-of-Concept

In order to determine how well our product could compete on the market, we provide a cost analysis that describes the components necessary to reconstruct our system as well as the cost of constructing a larger system. This analysis shows that our solution is more cost effective than others on the market and that its costs will be further reduced in a full-scale production environment.

To demonstrate the full functionality of our system, we constructed two camera nodes and a base station. These units allowed us to perform the tests described in Chapter 8 and to determine the limitations of the system. Tests that are more rigorous can be completed with the addition of more camera nodes, including tests to determine the reliability of the network and the interference caused by having more than a handful of nodes transmitting in close range.

9.1 Cost Analysis

The final production cost of the security camera system can be broken down into the major components purchased by both the camera and mesh networking teams. Each of these components is listed in Table 7, which shows the price breaks for the quantities one, ten, one hundred, and one thousand. Table 7 also shows the total cost of building a camera node when the parts are purchased in different quantities.

Table 7: Total cost per node for the given price breaks.

		Price Break Quantities			
		1	10	100	1000
Components for each node	Camera	€ 33.89	€ 30.68	€ 28.54	€ 24.96
	PIR Sensor	€ 19.96	€ 17.41	€ 15.75	€ 14.26
	MSP430	€ 32.82	€ 29.71	€ 26.89	€ 24.33
	SD Card	€ 7.95	€ 7.95	€ 7.95	€ 7.95
	Battery	€ 6.18	€ 6.18	€ 6.18	€ 6.18
	Solar Panel	€ 28.54	€ 28.54	€ 25.83	€ 23.38
	Charge Controller	€ 7.11	€ 7.11	€ 6.43	€ 5.82
	EM260 RCM	€ 87.20	€ 78.92	€ 71.43	€ 64.65
Total Node Cost for each Price Break Quantity		€ 223.65	€ 206.49	€ 188.99	€ 171.53

Table 8 displays the cost of purchasing the components for the base station. Table 8 is broken down similarly to Table 7, but only shows the two components that make up the base station: the MSP430 and the EM260 RCM. The price breaks show that if we were to buy 1,000 base station units, the cost for each base station would be €88.98.

Table 8: Total cost per base station for the given price breaks.

		Quantity Price Breaks			
		1	10	100	1000
Components for the Base Station	MSP430	€ 32.82	€ 29.71	€ 26.89	€ 24.33
	EM260 RCM	€ 87.20	€ 78.92	€ 71.43	€ 64.65
Total Node Cost for each Price Break		€ 120.02	€ 108.63	€ 98.32	€ 88.98

* This table does not take into account the price for a PC. Assumption is that the user has a computer.
* Software for the computer is also not figured into the price.

The overall price that it would cost for our team to purchase and build the system that contains ten nodes and one base station would be about €2,185. Although the system can support more nodes, we had to set a standard number of nodes for the system. If additional nodes need to be purchased they would cost approximately €220. These are not the prices that our product would be sold at if it were on the market today, but rather the prices that it would cost to purchase the components and assemble the system. These prices are based on buying a thousand units of each component. Our product is well inside the price range of the products that are on the market today and supply the user with more hardware and equipment.

9.2 Final Product Description

A fully operational system contains a network of nodes, ranging from one node to thousands of nodes, and a single base station. Each node contains a camera, sensor ZigBee transceiver, battery, solar panel, and memory. The base station contains a ZigBee transmitter and can be interfaced with a standard PC. The interface allows users to store captured images in a folder of their choice and view them at their leisure.

Our system is extremely power efficient and boasts an estimated battery life of approximately one year. The entire system is enclosed in a box, to prevent weather damage and vandalism. There are various ways to attach the system to poles, walls, and many other structures, which include permanent installation as well as temporary. For instance, users may attach the camera nodes to poles using a set of brackets that can be bolted into the system enclosure.

Our system is very adaptable and allows users to add enough nodes to survey nearly any car park. With automatic detection of nodes, automatic wake up and sleep modes, and re-routing in the case of a node failure, the system is completely self-sustaining. The setup for each of the nodes is automatic and involves only setting up only the base station and allowing the system to detect each node as it is powered on. The system configures itself with the necessary routing tables and communication lines to get the images to the base station correctly.

9.3 System Limitations

Unfortunately, a perfect environment almost never exists. Strange weather, large objects, radio interference, and other factors may disrupt the operation of the mesh network. Knowing this, we have made every attempt to ensure that the network is able to operate in most conditions. Specifically, we have addressed lost packets, unexpected power loss, and simultaneous image capture.

In the event that a packet is dropped in transmission, the node sending the packet will attempt to do so again. This process repeats ten times, after which the node attempts to rejoin the network. If successful, the transmission is attempted once more. If not, the node aborts the image transfer and any unsent data is lost. Packets sent from the base station are attempted ten times before the process is aborted. Fortunately, no data loss results in an abortion on the base station.

Should a node lose its connection to the base station, either because of an unexpected power failure or for another reason, the base station will wait ten seconds before considering the image transfer a failure. At this time, the data received thus far is saved into a partial image file and the base station returns to waiting for "Image Ready" signals from other nodes in the network. Due to the nature of JPEG images, these partial files are typically viewable as partial

images. The only way to access the unsent data is to manually remove the SD memory card from the camera node before it takes any new pictures.

In the event that multiple nodes capture an image simultaneously, the node which is able to transmit an "Image Ready" signal first will be allowed to transfer its image first. All other "Image Ready" signals will be ignored by the base station and written to the log file. Once the base station is finished receiving an image, other nodes may request an image transmission by sending their "Image Ready" signals. If a node does not receive the "Send Image" signal from the base station, it will wait for ten seconds and then attempt to send it again. This process repeats ten times before the transfer is aborted and the data lost.

10 Recommendations

This chapter describes recommendations for future improvements to the proof-of-concept device. It is our hope that these recommendation will help to both move the proof-of-concept design to a marketable base line product and inspire future upgrades to enhance the system.

10.1 PCB Design

To move the proof-of-concept design to a marketable product, we recommend that custom PCBs be designed for both the camera node and the base station. The proof-of-concept devices use pre-assembled modules. These modules provide the necessary functionality, but they are large and rather difficult to connect properly. A custom PCB design would allow for a decrease in the size of the camera nodes and base station and an increase in the reliability of both units.

10.2 Testing

Because there were only ten weeks allocated for the completion of this project more extensive testing needs to be completed on the proof-of-concept design before becoming a marketable product. The tests should include testing a system which consists of multiple camera nodes, testing in live car park conditions, extensive image transmission tests, and node failure tests. Each of these tests should be done to ensure that the system will function in any situation.

To perform the car park test the system should be set up in peak hours of the day when there is the most activity as well as during the night when there is little to no activity. The idea here is to get a sense of how many pictures the cameras will be taking during these hours and how many of those images will actually reach the base station. This test should be done with four camera nodes and a base station. Having four nodes would allow for a fairly large area to be covered while also showing that when there are multiple nodes in the system the images captured will all eventually reach the base station.

Node failure should be studied in more detail. This should be tested by having a node send an image but then fail to send the all of the packets of the image. The base station should

acknowledge that it did not receive all the packets and send a message back to the failed node. This node will then try to send the image once again. This test would prove that if a node fails then the node will attempt to send the image again until it is successful. By further examining these tests it will help to enhance the overall system.

10.3 Node Status Indicators

We also recommend connecting more status indicators to each node. These indicators can help determine possible errors in the network and notify users if a node needs maintenance. For example, one might attach sending and receiving indicators to each node. These new indicators would benefit the user when testing the system for possible data transmission errors. Other possible uses for additional indicators include battery status and connection status. Simple LEDs make fine indicators and are rather inexpensive.

Alternatively, a small LCD screen could be added as an upgrade in following versions of the system. This LCD screen could be used to display messages about the status of the node, in regards to sending images, receiving images, battery life, and system failures. Each of these situations would have readouts on the module to aid the user in obtaining information about the node.

10.4 Wireless Signal Detector

A wireless signal strength indicator would be a fine accessory to the system because it would make installing new camera nodes significantly easier. This device would be held in hand and indicate the strength of the wireless signal from the closest node. This would remove a lot of the guesswork involved in designing a new wireless network and ensure the user that each of the camera nodes would be able to transmit data to the base station.

Figure 23 displays an artist's rendition of the product. The three LEDs on the top of the device would indicate when the user had an excellent signal, a weak signal, and had no signal at all. A screen would display data such as how far from the previous node the user was and the precise strength of the wireless signal

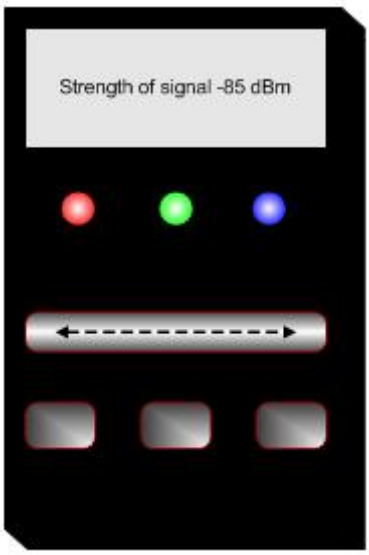

Figure 23: Wireless Signal Detector.

10.5 Network User Interface Software

Stemming from previous sending and receiving indictors and also low battery indicators, a network user interface software needs to be implemented as a future upgrade in the system. This would be very useful in maintaining the systems function from the convenience of a personal computer. The software should be loaded onto the base station computer to interact with the nodes in the network. This software should be used to denote errors that occurred in the network such as when a node has left the network and a possible indication for why this has happened. It could also be used to indicate that the nodes battery is low and should be changed. This software could also be used to suggest optimal network configurations for the users lay out in the car park. A software interface is a necessity to make system maintenance as simple as possible and provide an extra edge in wireless security camera systems.

We recommend that a more comprehensive software package be developed for the base station. This software would allow users to interact with the network from a PC and allow them to see the whether a node's battery needs replacement, if a node is offline, or if an error has occurred somewhere in the network. Ideally, this software should be able to make recommendations to help the user improve his or her network.

These recommendations provide ideas for future development and expansion on the wireless security system that we developed during out ten-week stay in Ireland. It is our hope

that with continued development, the system will become a marketable product that will help the University of Limerick and other organizations deter crime.

References

[1] "Theft of motor vehicles classified by regional authority". Quarterly national household survey, 2004. Central Statistics Office Ireland. 4a 12-15.

[2] Marcus Baram. **"**Eye on the City: Do Cameras Reduce Crime?". ABC News. 9 July 2007. Viewed 4 September 2007. <http://www.abcnews.go.com/US/Story?id=3360287&page=1>.

[3] Gazzola, Robert. Surveillance Secure Inc. Personal contact: e-mail. 12 September 2007.

[4] 123 Security Products. Wireless digital four, August 2007. <http://www.bizrate.com/homesecurity/oid14605114.html>.

[5] Mathew Conway, Phong Damn, Janelle Tavares. Worcester Polytechnic Institute. "Efficient Surveillance System" <http://www.wpi.edu/Academics/Projects/projects.html>.

[6] Big Bruin. 4 Channel Wireless Camera System. 16 September 2007. <http://www.bigbruin.com/reviews05/wirelesscams/index.php?file=1>.

[7] Gprs / gsm remote mobile wireless internet snapshot camera - observer iv, August 2007. <www.colorado-video.com/observerapps.html>.

[8] Clifford, Seamus. University of Limerick. Personal contact: interview. 8 August 2007.

[9] WiFi Alliance. "What is the range of a wireless network". 16 August 2007. <http://www.wi-fi.org/knowledge_center_overview.php?docid=3278>.

[10] European Commission. Directive 1999/5/ec. 16 August 2007. <http://ec.europa.eu/enterprise/rtte/dir99-5.htm>.

[11] Microsoft TechNet. "Wifi protected access 2 (wpa2) overview". 16 August 2007. <http://www.microsoft.com/technet/community/columns/cableguy/cg0505.mspx>

[12] Bluetooth SIG. "Bluetooth basics". 16 August 2007. <http://www.bluetooth.com/Bluetooth/Learn>.

[13] ZigBee Alliance. "Zigbee FAQ".17 August 2007. <http://www.zigbee.org/en/about/faq.asp>.

[14] ZigBee Alliance. "Security Layer Technical Overview, 2005". <http://www.zigbee.org/en/events/documents/Dec2005_Open_House_Presentations/ZigBee_Security_Layer_Technical_Overview.pdf>.

[15] Texas Instruments. "Mixed Signal Microcontroller." 5 September 2007. <http://www.ti.com/>.

[16] Ember Corporation. "EM260 Radio Communication Module Technical Specification: SPI/UART." < http://ember.com/>.

[17] Telegesis. "ETRX2USB Wireless Mesh Networking USB Stick/BaseStation." < http://www.telegesis.com/>.

[18] Embedded Development Community. "Home Networking With ZigBee." September 2007. <http://www.embedded.com/columns/technicalinsights/18902431?_requestid=44555>.

[19] Sahinoglu, Zafer. Mitsubishi Electric Research Laboratories. "Image Transmission over IEEE 802.15.4 and ZigBee Networks", May 2005.

[20] Ember Corp. "Developing Ember Applications." Building Application pg 7. < http://ember.com/>.

[21] ZigBee Alliance. "ZigBee Specification." Version r13.

[22] Search Networking. "Data link layer." 16 August 2007. <http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211899,00.html>.

[23] Linktionary. "Network layer protocols." 16 August 2007. <http://www.linktionary.com/n/network_layer.html>.

[24] "The presentation layer." 17 August 2007. <http://homepages.uel.ac.uk/u0324781>.

[25] Rad Data Communications. "The transport layer." 16 August 2007. <http://www.raduniversity.com/networks/1994/osi/transp.htm>.

[26] Tritech. "ZigBee Alliance Platform Layers." 29 August 2007. < http://www.tritech.se/>.

[27] How Stuff Works. "How OSI Works: The layers." 15 August 2007. <http://computer.howstuffworks.com/osi1.htm>.

Appendix A: Mesh Networks

Mesh Networks are one of the many ways to transfer data from one point to another. They are composed of nodes that are wirelessly connected to each other and have the capabilities to receive and transfer data. The definition of a wireless mesh network is that of a fully wireless network that employs multi-hop communications to forward traffic en route to and from wired nodes [22].

Wireless mesh networks (WMN) can have two types of nodes; mesh routers and mesh clients. Mesh routers contain gateway/bridge functions which allow them to connect to other equipment such as laptops and other types of networks. However, a mesh router is different from a conventional wireless router because it contains additional routing functions to support mesh networking. It also has the ability to cover a wider range of transmission using less power. Mesh routers are easy to use because they can be built on same platform as a regular wireless router. Mesh clients have almost all the same functions as a mesh router since they need to support mesh networking, except they do not have gateway/bridge functions [23]. Mesh clients cannot be used to connect to other platforms; they just provide a means of transferring data.

Based on the connections between the routers and the clients, WMNs have three types of architectures. Figure 24 shows how the infrastructure/backbone architecture of WMN may look.

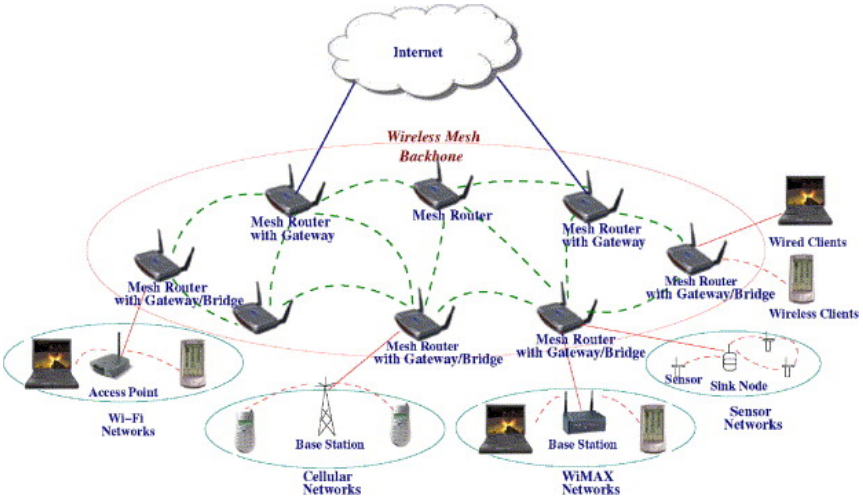

Figure 24: Infrastructure architecture for WMN [23].

This type of architecture is the most common. Mesh routers with gateway functions are connected to the Internet and they relay information by using other mesh routers, they then transfer this information to another platform by using mesh routers with gateway/bridge functions. The mesh routers form the backbone of this network and it is possible to transfer information to any other wireless system. The other two types of WMNs are client and hybrid. Client networks occur between mesh clients and provide peer-to-peer networking. This sort of network does not require a mesh router. Hybrid networks combine client infrastructure networks [23].

Regardless of the architecture employed all WMNs contain the following characteristics:

- Multi-hop routing: The main objective in developing wireless mesh networks is to extend the coverage area without losing channel capacity and to provide connectivity between users who are not in each others' line-of-sight.
- Capability of self-organizing: It is possible to add new nodes to the mesh network and other nodes can automatically recognize the new node. This provides flexibility in the network.
- Different types of network access: With mesh routers that have gateway/bridge functions, it is possible to connect to the Internet, WiFi networks, cellular networks etc.

Wireless mesh networks are closely related to ad hoc networks and are sometimes called multi-hop ad hoc networks. Ad hoc refers to the ability of nodes to both transfer and receive data and not be designated a specific task; multi-hop refers to the ability of the nodes to relay information from one node to another to reach the desired destination. However, mesh networks differ from ad hoc networks by the powerful capabilities of their routers. In an ad-hoc network, the end nodes have to perform configuration for the other nodes whereas a mesh router handles the entire configuration on its own. It is also possible to separate the network traffic in mesh networks by having routers with two radios. One could be used for transmission and the other could be used for users to connect to the network, which would improve the capacity of the network [23].

Applications

Mesh networks can provide solutions that other types of networks cannot. An example is a home network that contains zones that do not have coverage. These dead zones can be easily equipped with mesh routers, which make the connections possible. This idea can be extended to community/neighborhood networks [23]

Even though mesh networking is a fairly new concept, it has already been used in important projects. Massachusetts Institute of Technology (MIT) has developed a project called Roofnet. In this project students have implemented nodes over various buildings and created an "experimental multi-hop IEEE 802.11b mesh network" [24]. This project uses all of the primary features of a mesh network. A user can turn on a node and it will become a part of the network. Thus far, the network has been working quite well, with a good throughput of hundreds of kilobytes per second. The initial aim of Roofnet was to cover 1.25 square miles, near the MIT campus. Each node that a user was given contained an antenna and an Ethernet port [24].

Another recent application of a wireless mesh network has been created by Microsoft Research Center. Similar to MIT, Microsoft is trying to install a neighborhood network by utilizing mesh networking technologies. The advantage to this is that users will not each have to get an Internet connection. Instead they can share one Internet connection, which would prove to be cheaper and easier for the user. In this type of a network, everybody in the community contributes to the network and co-operates [4]. A similar project is being conducted by the Champaign-Urbana Community Wireless Network (CUWiN) to bring Internet to people in Ghana. Currently, ten nodes have been implemented over a range of 20 km, providing many schools, houses and businesses with the Internet [25].

Portsmouth Real-Time Travel Information System (PORTAL) is a system that aims to provide information on bus schedules accurately to the passengers. To realize this system, about 300 buses were equipped with mesh network devices that were provided by MeshNetworks Inc. Users can display real-time information on bus services in forty locations throughout the city. This solution can be extended to monitor traffic and give detailed information.
These projects are still being worked on and there is much improvement still to be completed. However, they do show that mesh networks are becoming important in research areas and they will be used more frequently as the technology improves over time.

Disadvantages to Mesh Networks

Despite all the advantages that WMNs provide, they also have a few disadvantages that have not yet been solved. One of the main disadvantages is security. WMNs need extra security due to the vulnerability of nodes in a wireless medium [23]. Attackers can change the routing table, forward a packet to a destination that it was not meant to go, or become a part of the network to disturb the way it works. Even a cryptographic network protocol may not be enough to stop an attacker. In a cryptographic protocol, there is a constant exchange of information with a trustworthy source [23]. However, in any wireless network there is always a danger of attackers intercepting information passed between the trustworthy source and users.

Another problem that occurs is the network capacity. Kleinrock and Silvester, from the department of Computer Science at UCLA, have shown that if all the nodes are stationary in a network, then the optimal number of nodes in a network is six and if there are fewer nodes, then the capacity in the network decreases. The capacity of a network can be increased by insuring that the routing protocol provides the fastest path to its final destination. This would mean that additional nodes that have been added to the network would increase the capacity; however, it would also increase the cost of the system [23].

Appendix B: IEEE 802.15.4

Institute of Electrical and Electronics Engineers (IEEE) is a non-profit organization that has more than 370,000 members. This organization was formed in 1963 and is the world's largest association for advancement the technology. The IEEE 802.15.4 standard is a low cost, low power wireless communication protocol. The Media Access Control Layer (MAC) and the Physical Layer are specifically defined in the 802.15.4 protocol. Figure 25 shows these layers as well as the ZigBee Alliance Platform layers.

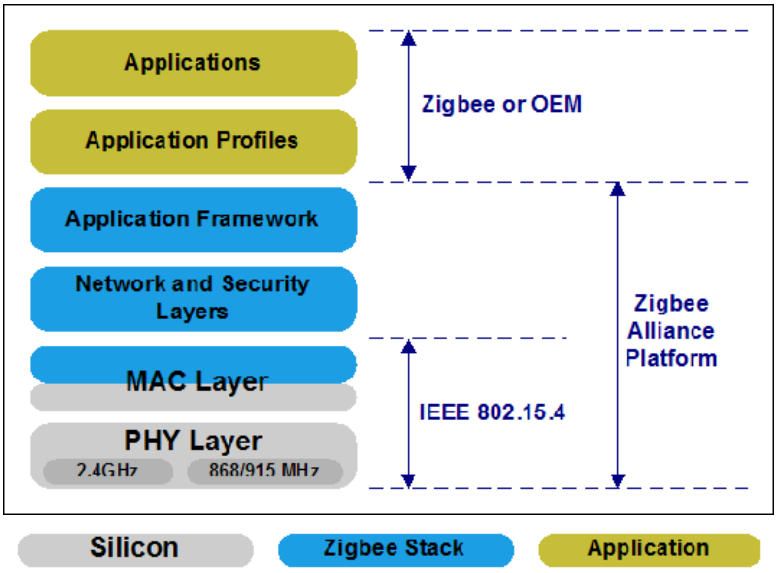

Figure 25: IEEE 802.15.4 Standard model [26].

The data rate for devices that are compliant with the standard can range from 20 Kbit/s to 250 Kbit/s. The frequency bands that the standard uses are 868 MHz, 915 MHz, and 2.4 GHz. The data rates that relate to these frequencies are 20 Kbit/s, 40 Kbit/s, and 250 Kbit/s, respectively [23]. The devices used for this wireless security system are all 802.15.4 IEEE compliant.

Appendix C: Open Systems Interconnection (OSI) Model

The Open Systems Interconnection (OSI) Model was designed to layout the description for digital communications and network protocol design. This protocol design consists of seven layers: physical, data link, network, transport, session, presentation, and application. Figure 26 below shows these layers in the appropriate order.

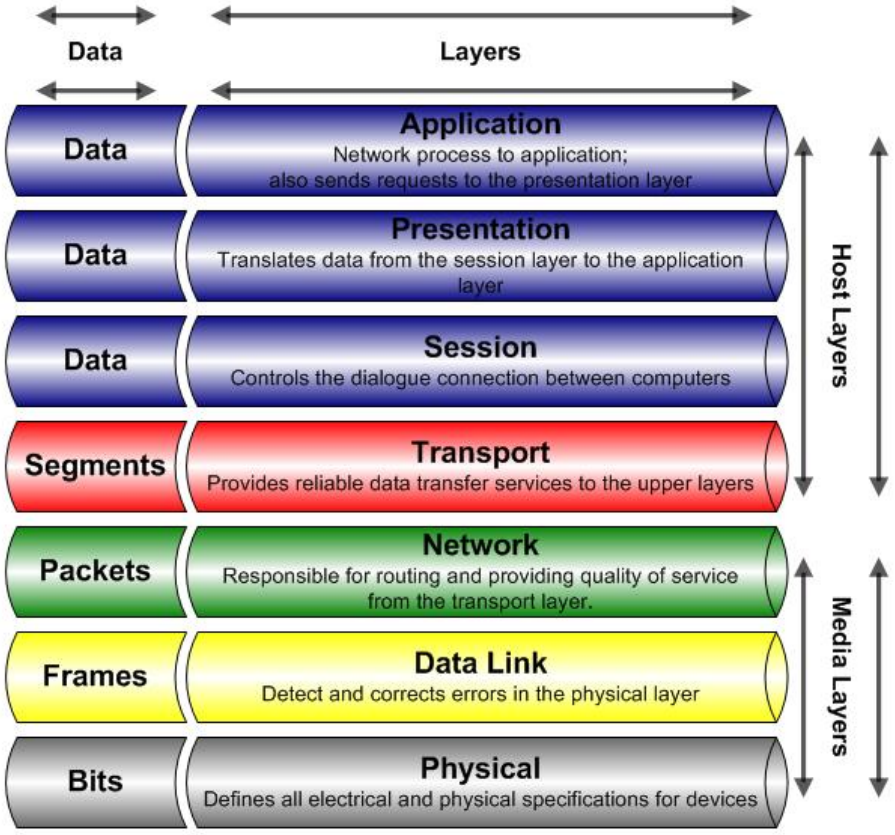

Figure 26: OSI Model

The Physical Layer is the first layer in the OSI model. This layer is the most basic layer of the model. The Physical Layer is the actual hardware, such as a transceiver, for transmitting raw data in bit form. This layer is responsible for carrying out the requests from the Data Link Layer, which is the next layer on the OSI model. The Data Link Layer contains two subparts: the media access control and the logical link control. This layer is in charge of moving data in and out of the physical link in the network. When the data is sent out this layer will get an acknowledgment that the data has in fact arrived at its destination. The next layer of the OSI

model is the Network Layer. This layer determines how the data will be sent to its intended receiver and is responsible for routing and network addressing. The fourth layer, also known as the Transport Layer, maintains the flow of data while checking for any errors to ensure all pieces of the data arrive properly. The Session Layer manages and ends any connections or sessions that are in progress. This means that when data is flowing between two software applications processes, the Session Layer will manage this connection by either allowing communication or ending the session. The sixth layer in the OSI model is the Presentation Layer. This layer is in charge of preparing incoming data for the Application Layer. The Presentation Layer is accountable for data translation and defining data formatting. The last and final layer of the OSI Model is the Application Layer. This layer forms the interface between user applications and the operating system's network stacks [26].

Appendix D: Images of the Camera Node and the Base Station

Figure 27: Camera node with the enclosure.

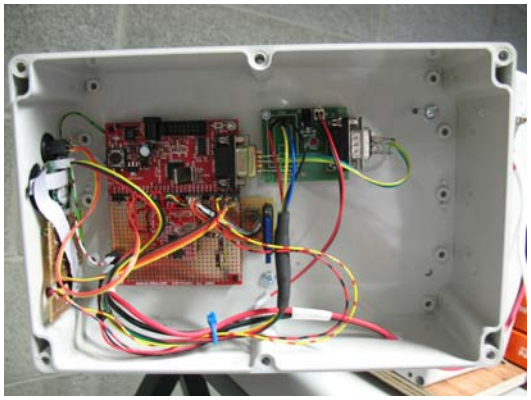

Figure 28: Camera node inside the enclosure.

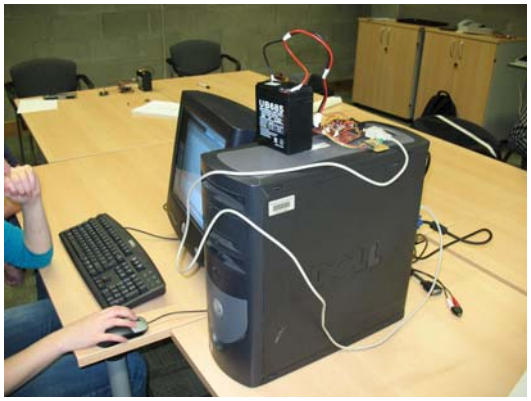

Figure 29: Base station

Appendix E: Instructions for Security Camera System Code

PC Software:

The base station software may only be run on a PC with Microsoft Windows installed. Because this software makes use of the Windows API, it will not run in MacOS, Linux, or any other operating system.

Compiling:

To compile the source, you must first install a compiler. Three popular compilers for Windows are MinGW (a port of the GCC), Microsoft Visual Studio, and Intel's ICC. The Makefile included with this distribution is suited for use in MinGW, though it can easily be modified to run with any other compiler.

Running:

Before running the em250.exe program, the base station must be connected to a COM port. When the large LED is lit, the base station is ready.

To run this software, open a Command Prompt. In Windows XP, this can be accomplished by clicking on Start -> All Programs -> Accessories -> Command Prompt. There, run the following command:

em250.exe -c (NUM) -d (DIR)

where NUM is the COM port number that the base station is connected to and DIR is the directory to save images and log files to.

At any time, you may press ctrl+c to terminate the program. Upon restarting, the program will instruct the base station to form a new network.

Files:

The em250.exe program creates a log file each time it is started. These files have names of the form LOG_DDMMYYYY_HHMMSS_NN.txt and are located in the directory specified at program start. These files will be useful for debugging.

Image files are located in the same directory and have names of the form DDMMYYYY_HHMMSS_NN.jpg.

MSP Base Station Software:

This software allows the PC to interact with the ZigBee network. It is intended for use on an MSP430 connected to an EM260 co-processor.

Compiling:

To compile this software, you will need either IAR or MSPGCC. Note that the trial (Kickstart) version of IAR will not be sufficient as the binary is more than 4KB. The Makefile provided with this distribution is suitable for MSPGCC.

Running:

To run this software, simply flash it onto an MSP430 via a JTAG cable. The power provided by the JTAG cable is enough to flash the chip. An alternate power source will be required to successfully run the software.

MSP Node Software:

This software runs on an MSP430 microcontroller with an attached EM260 co-processor.

Compiling:

To compile this software, you will need either IAR or MSPGCC. Note that the trial (Kickstart) version of IAR will not be sufficient as the binary is more than 4KB. The Makefile provided with this distribution is suitable for MSPGCC.

Running:

To run this software, simply flash it onto an MSP430 via a JTAG cable. The power provided by the JTAG cable is enough to flash the chip. An alternate power source will be required to successfully run the software.

Appendix F: MSP Base Station Code

basestation.h:

```
MSP430 Base Station
Copyright (C) 2007 Joseph Bosman, Steve Olivieri, Ipek Ozil, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include <msp430x16x.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef BASE_STATION_H
#define BASE_STATION_H

/* nHOST_INT interrupt boolean */
volatile unsigned char nHOST_INT;

/* Tracks potential callbacks for the EM260 */
volatile unsigned char potentialCallback;

/* Interrupt boolean for the PC */
volatile unsigned char pcINT;

/* Buffer for storing commands from the PC */
unsigned char pcBuff[150];

/* Number of characters currently in pcBuff */
volatile unsigned char pcI;

/* Node ID of the last IMAGE_READY signal */
unsigned char irID[2];

/* IMAGE_READY boolean */
unsigned char imageReady;

/* Interrupt routines. */
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector=USART0RX_VECTOR
    __interrupt void usart0_rx(void);
  #pragma vector=PORT1_VECTOR
    __interrupt void em260(void);
#else
  interrupt (USART0RX_VECTOR) usart0_rx(void);
  interrupt (PORT1_VECTOR) em260(void);
#endif

/* Wait for a while.  Specifically, perform two nop() opertions for each
 * unit of 'time.'  How long is a unit?  About 6 operations in GCC without
 * optimizations.  This is a rather poor way of waiting, but it'll do for
```

```
 * now.
 *      @param time: the number of units of time to wait
 */
void delay(unsigned int time);

/* Initializes UART0 on Port 3 of the MSP430. */
void initUART0(void);

/* Initializes SPI1 on Port 5 of the MSP430. */
void initSPI1(void);

/* Initializes the EM260 by performing the following tasks:
 *      - resets the EM260
 *      - performs a reset transaction over SPI
 *      - performs a "verion" transaction over SPI
 *      - performs a "SPI status" transaction over SPI
 */
unsigned int initEM260(void);

/* Cleans up the MSP430 after a reboot by performing the following tasks:
 *      - sets all I/O pins to I/O mode
 *      - sets all I/O pins to output
 *      - sets all I/O pins to low
 *      - turns off all interrupts
 *      - sets interrupt flags to low
 *      - sets interrupts modes to low-to-high
 */
void initValues(void);

/* Initializes the MSP430 by performing the following tasks:
 *      - sets the MSP to use the 8MHz crystal at 4MHz
 *      - turns off the watchdog timer
 *      - globally enables interrupts
 *      - clears registers (see initValues())
 *      - sets up pins for EM260 operation
 *      - clears nHOST_INT
 */
void initMSP(void);

/* Sends a string to the EM260 over SPI1.
 *      @param buff: a pointer to the string to send
 *      @param len: the length of the string
 */
void toEM260(unsigned char* buff, unsigned int len);

/* Receives data from the EM260 over SPI1.
 *      @param buff: a pointer to the buffer to store the response in
 *      @return: the number of bytes received (and stored in buff)
 */
unsigned char fromEM260(unsigned char* buff);

/* Sends a string to the PC over UART0.
 *      @param buff: a pointer to the string to send
 *      @param len: the length of the string
 */
void toPC(unsigned char* buff, unsigned int len);

/* Establishes a ZigBee PAN with this EM260 as the coordinator.
 * The extended PAN ID is NNNNNNNNNNNNNNNN and the PAN ID is NNNN.
 * The network exists on Channel 13.  This function uses EZSP to
 * call the emberFormNetwork() command.
 *      @param sq: the sequence number of the EZSP frame
 *      @return: the status of the network
 */
unsigned char EM260_establishNetwork(unsigned char sq);

/* Sets a configuration value on the EM260.  The params buffer
 * must contain the following: EZSP sequence number (1 byte),
 * the ConfigId (1 byte), and the new value for that ConfigId
 * (2 bytes).  ConfigIds are specified in the EM260 data sheet.
```

```
 * This function returns the status value from the EZSP response.
 *      @param params: the arguments for setConfigurationValue()
 *      @return: the status of the command
 */
unsigned char EM260_setConfigurationValue(unsigned char* params);

/* Calls the EZSP networkInit() function.  The result of calling
 * this function depends on whether or not the EM260 was joined
 * to a network on last power off.  If so, a callbac must follow
 * this command.  In addition, the EM260 will once again be on
 * that network.  If not, no callback should be issued and the
 * EM260 will not be joined to any networks.
 *      @param sq: the sequence number of the EZSP frame
 *      @return: the network status
 */
unsigned char EM260_networkInit(unsigned char sq);

/* Calls the EZSP leaveNetwork() function.  The result of this
 * call is the return value for EM260_leaveNetwork().  If this
 * call is successful, the EM260 will leave any network that it
 * is joined to.  Else, the state of the network is unknown.
 *      @param sq: the sequence number of the EZSP frame
 *      @return: the status of the command
 */
unsigned char EM260_leaveNetwork(unsigned char sq);

/* Forms a network via the EZSP scanAndFormNetwork() command.
 * The extended PAN ID of the network is F1332548CA02121F,
 * the short PAN ID is chosen randomly, and the Channel is
 * automatically selected after scanning all available channels
 * and determining which one has the least amount of interference.
 * After issuing this command, a callback must be made.
 *      @param sq: the sequence number of the EZSP frame
 */
void EM260_formNetwork(unsigned char sq);

/* Permits nodes to join the network (as routers or end-devices).
 * This function must be called after EM260_formNetwork() or else
 * the network will not function properly.
 *      @param sq: the sequence number for the EZSP frame
 *      @return: the status of the command
 */
unsigned char EM260_permitJoining(unsigned char sq);

/* Gets the parameters of the network that the EM260 is joined to.
 * Since the EM260 should always be acting as a coordinator, it
 * is not necessary to obtain the NodeId (which is always 0000).
 * The PAN ID is stored in bytes 15-16 (little endian), the channel
 * is stored in byte 18 of buff.  The extended PAN ID is stored
 * in bytes 7-14 of buff.
 *      @param buff: pointer to the buffer to store the parameters in
 *      @param sq: the sequence number for the EZSP frame
 */
void EM260_getNetworkParameters(unsigned char* buff, unsigned char sq);

/* Handles callbacks from the EM260.  Currently, it ignores all
 * callbacks except for the incomingMessageHandler callback as
 * that's all we really care about.
 *      @param buff: the callback to handle (result of EM260_callback())
 *      @param sq: the sequence number of any EZSP frames that we send
 */
void handleCallback(unsigned char* buff, unsigned char sq);

/* Issues a callback() command to the EM260 and receives a response.
 * Typically, callbacks are made after calling other commands.  In the
 * case that the EM260 must send us a message asynchronously, the
 * nHOST_INT interrupt will be triggered and a callback must be made.
 *      @param buff: pointer to the buffer to store the respnse in
 *      @param sq: the sequence number for the EZSP frame
 */
```

```
unsigned char EM260_callback(unsigned char* buff, unsigned char sq);

/* Sends a unicast message to another node on the network via the
 * EZSP sendUnicast() function.  If this function is successful,
 * a callback must follow.  This function always uses ProfileId 0,
 * ClusterId 0, and EndPoint 0x01 (source and destination).  Messages
 * are always tagged with 0x64, route discovery is enabled, retries
 * are enabled, and the source EUI64 is always sent before the message.
 *      @param addr: the two-byte NodeID to send the message to
 *      @param message: the message to send
 *      @param len: the length of message
 *      @param sq: the sequence number for the EZSP frame
 *      @param sq2: the sequence number for the unicast
 */
unsigned char EM260_sendUnicast(unsigned char* addr, unsigned char* message,
        unsigned char len, unsigned char sq, unsigned char sq2);

#endif /* BASE_STATION_H */
```

basestation.c:

```
MSP430 Base Station
Copyright (C) 2007 Joseph Bosman, Steve Olivieri, Ipek Ozil, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include "basestation.h"

#ifdef __IAR_SYSTEMS_ICC__
 #pragma vector=PORT1_VECTOR
 __interrupt void em260 (void)
#else
 interrupt (PORT1_VECTOR) em260(void)
#endif
{
        nHOST_INT = 1;
        //if(!potentialCallback)
                potentialCallback++;
        P1IFG &= ~0x08;
}

#ifdef __IAR_SYSTEMS_ICC__
 #pragma vector=USART0RX_VECTOR
 __interrupt void usart0_rx (void)
#else
 interrupt (USART0RX_VECTOR) usart0_rx(void)
#endif
{
        pcBuff[pcI] = U0RXBUF;
        if(pcBuff[pcI] == 0x0D)
                pcINT = 1;
        ++pcI;
}

void delay(unsigned int time)
{
        for( ; time > 0; time--)
        {
                nop();
                nop();
        }
}

unsigned char charToHex(unsigned char* buff)
{
        return ((unsigned char)(((int)(buff[0] - (buff[0] > '9' ? 55 : 48)) * 16)
                + (int)(buff[1] - (buff[1] > '9' ? 55 : 48))));
}

void initUART0(void)
{
        /* Use P3.4 and P3.5 for UART0 */
        P3SEL |= 0x30;

        /* P3.4 must be OUT (TXD), and P3.5 (RXD) must be IN */
```

```
        P3DIR |= 0x10;
        P3DIR &= ~0x20;

        /* Put UART0 in a reset state. */
        U0CTL |= SWRST;

        /* Enable USART0 TXD/RXD. */
        ME1 |= UTXE0 + URXE0;

        /* Set the UART parameters to:
         *      8-bit character
         *      1 stop bit
         *      no parity
         */
        U0CTL |= CHAR;

        /* UCLK = SMCLK */
        U0TCTL |= SSEL1;

        /* Set the baud rate for UART0 to ~38400 bps. */
        UBR00 = 0x68;
        UBR10 = 0x00;

        /* Modulation. */
        U0MCTL = 0x4;

        /* Initialize UART0 state machine. */
        U0CTL &= ~SWRST;

        /* Wait for the Tx buffer to be ready. */
        while(!(IFG1 & UTXIFG0));

        /* Enable the UART0 RX interrupt. */
        IE1 |= URXIE0;
}

void initSPI1(void)
{
        /* Use pins 5.1-5.3 for USART1. */
        P5SEL |= 0x0E;

        /* P5.1 (MOSI) and P5.3 (SCLK) must be OUT, P5.2 (MISO) must be IN */
        P5DIR |= 0x0A;
        P5DIR &= ~0x04;

        /* Put USART1 into a reset state. */
        U1CTL |= SWRST;

        /* Enable SPI. */
        ME2 |= USPIE1;

        /* Set USART parameters to:
         *      8-bit character
         *      1 stop bit
         *      no parity
         *      SPI mode
         *      Master mode
         */
        U1CTL |= CHAR + SYNC + MM;

        /* BRCLK = SMCLK, 3-pin mode, phase shifting */
        U1TCTL |= SSEL1 + STC + CKPH;

        /* Set the baud rate to 500 Kbps. */
        UBR01 = 0x08;
        UBR11 = 0x00;

        /* Turn off modulation. */
        U1MCTL = 0x00;
```

```
	/* Set P5.0 to be OUT (nSSEL). */
	P5DIR |= 0x01;
	P5OUT |= 0x01;

	/* Initialize USART1 state machine. */
	U1CTL &= ~SWRST;

	/* Wait until the buffer is clear. */
	while(!(IFG2 & UTXIFG1));

	/* Enable the SPI1 RX interrupt. */
	//IE2 |= URXIE1;
}

unsigned int initEM260(void)
{
	/* Buffer for storing commands and responses */
	unsigned char buff[10];

	/* How many bytes we received */
	unsigned char i;
	i = 0;

	/* Reset the EM260 (P4.6 = 0). */
	P4OUT &= ~0x40;

	/* Wait for a while. */
	delay(0x0014);

	/* Release the reset pin. */
	P4OUT |= 0x40;

	/* Wait for EM260 to assert nHOST_INT */
	while(!nHOST_INT);
	nHOST_INT = 0;

	/* Now, we must transmit a specific series of commands to finish
	 * initializing the EM260.  The first command is the "version"
	 * command.  The first step is to load the "version" command (0x0A)
	 * and the frame terminator (0xA7) into the buffer.
	 */
	buff[0] = 0x0A;
	buff[1] = 0xA7;

	/* Assert nSSEL. */
	P5OUT &= ~0x01;

	/* Send the command to the EM260. */
	toEM260(buff, 2);
	//toPC(buff, 2);

	/* Clear enough buffer space to receive the response. */
	memset(buff, 0, 3);

	/* Now, receive the response. */
	i = fromEM260(buff);
	//toPC(buff, i);

	/* Deassert the nSSEL signal. */
	nHOST_INT = 0;
	P5OUT |= 0x01;

	/* If the first byte is not 0x00 and the second byte is not 0x02,
	 * there's a problem.
	 */
	if(buff[0] != 0x00 || buff[1] != 0x02)
		return 1;

	/* We must wait at least 1ms between commands. */
	delay(0x00FF);
```

```
	/* We must again transmit the "version" command. */
	buff[0] = 0x0A;
	buff[1] = 0xA7;
	P5OUT &= ~0x01;
	toEM260(buff, 2);
	//toPC(buff, 2);
	memset(buff, 0, 3);
	i = fromEM260(buff);
	//toPC(buff, i);
	nHOST_INT = 0;
	P5OUT |= 0x01;

	/* If the first byte of our response is not 0x82, there is
	 * a problem.
	 */
	if(buff[0] != 0x82)
		return 2;

	/* Again, we must delay for at least 1ms between commands. */
	delay(0x00FF);

	/* Now, transmit the "SPI status" command. */
	buff[0] = 0x0B;
	buff[1] = 0xA7;
	P5OUT &= ~0x01;
	toEM260(buff, 2);
	//toPC(buff, 2);
	memset(buff, 0, 3);
	i = fromEM260(buff);
	//toPC(buff, i);
	nHOST_INT = 0;
	P5OUT |= 0x01;

	/* If the first byte of the response is not 0xC1, there is
	 * a problem.
	 */
	if(buff[0] != 0xC1)
		return 3;

	/* Wait for at least 1ms. */
	delay(0x00FF);

	/* Finally, we must send the EZSP "version" command. */
	buff[0] = 0xFE;
	buff[1] = 0x04;
	buff[2] = 0x00;
	buff[3] = 0x00;
	buff[4] = 0x00;
	buff[5] = 0x02;
	buff[6] = 0xA7;
	P5OUT &= ~0x01;
	toEM260(buff, 7);
	//toPC(buff, 7);
	memset(buff, 0, 10);
	i = fromEM260(buff);
	//toPC(buff, i);
	nHOST_INT = 0;
	potentialCallback = 0;
	P5OUT |= 0x01;

	/* If bytes 5 and 6 are not each 0x02, something is wrong. */
	if(buff[5] != 0x02 || buff[6] != 0x02)
		return 4;

	/* Return successfully. */
	return 0;
}

/* Initializes the MSP430's I/O registers. */
```

```
void initValues(void)
{
	/* I/O pins aren't controlled by modules */
	P1SEL = 0x00;
	P2SEL = 0x00;
	P3SEL = 0x00;
	P4SEL = 0x00;
	P5SEL = 0x00;
	P6SEL = 0x00;

	/* All I/O pins are outputs */
	P1DIR = 0xFF;
	P2DIR = 0xFF;
	P3DIR = 0xFF;
	P4DIR = 0xFF;
	P5DIR = 0xFF;
	P6DIR = 0xFF;

	/* All output values are zero */
	P1OUT = 0x00;
	P2OUT = 0x00;
	P3OUT = 0x00;
	P4OUT = 0x00;
	P5OUT = 0x00;
	P6OUT = 0x00;

	/* Turn off P1 and P2 interrupts */
	P1IE = 0x00;
	P2IE = 0x00;

	/* All interrupts flags are reset */
	P1IFG = 0x00;
	P2IFG = 0x00;

	/* Interrupt modes are all low-to-high */
	P1IES = 0x00;
	P2IES = 0x00;
}

/* Initialize the MSP430. */
void initMSP(void)
{
	/* Use the 8MHz crystal at 4MHz. */
	BCSCTL1 &= ~XT2OFF;
	BCSCTL2 |= SELM_2 + SELS + DIVS_1;

	/* Turn the watchdog timer off. */
	WDTCTL = WDTPW | WDTHOLD;

	/* Enable interrupts. */
	_EINT();

	/* Quickly clear all I/O and interrupt registers. */
	initValues();

	/* P6.0 (LED) is OUT. */
	P6DIR |= 0x01;

	/* Init the LED to off. */
	P6OUT |= 0x01;

	/* For debugging, we also enable P2.4-2.7 as LEDs. */
	/* TODO: DEBUG */
	P2DIR |= 0xF0;
	P2OUT |= 0xF0;

	/* P4.6 (nRESET) and P4.7 (nWAKE) are OUT. */
	P4DIR |= 0xC0;
	P4OUT |= 0xC0;
```

```
        /* P1.3 (nHOST_INT) is a H->L interrupt and IN. */
        P1DIR &= ~0x08;
        P1SEL &= ~0x08;
        P1IES |= 0x08;
        P1IE |= 0x08;

        /* Initialize nHOST_INT */
        nHOST_INT = 0x00;
}

void toEM260(unsigned char* buff, unsigned int len)
{
        /* Temporary receive buffer. */
        unsigned char tmp;

        /* Loop variable */
        unsigned int i;

        for(i = 0 ; i < len; i++)
        {
                /* Wait until the Tx buffer is empty. */
                while(!(IFG2 & UTXIFG1));

                /* Transmit the next byte in the buffer. */
                U1TXBUF = *(buff+i);

                /* Receive the idle response */
                while(!(IFG2 & URXIFG1));
                tmp = U1RXBUF;

                /* Transmit to the PC. */
                //while(!(IFG1 & UTXIFG0));
                //U0TXBUF = *(buff+i);
        }
}

unsigned char fromEM260(unsigned char* buff)
{
        /* EZSP frame boolean */
        unsigned char ezsp;

        /* Length of the response */
        unsigned char len;

        /* Loop variable */
        unsigned char i;

        /* Initialize i to 2 to account for the first two bytes
         * that we read.
         */
        i = 0x02;
        ezsp = 0;
        len = 0;

        /* Transmit 0xFF and receive 0xFF to clock the SPI.  Do so until
         * we receive a byte that is not 0xFF.  This byte is the first
         * byte of the response.
         */
        *buff = 0xFF;
        while(*buff == 0xFF)
        {
                while(!(IFG2 & UTXIFG1));
                U1TXBUF = 0xFF;

                while(!(IFG2 & URXIFG1));
                *buff = U1RXBUF;
        }

        /* Determine whether or not we're reading an EZSP frame. */
        if(*buff == 0xFE)
```

```
                ezsp = 1;

        /* Send another 0xFF to get the length of the message. */
        while(!(IFG2 & UTXIFG1));
        U1TXBUF = 0xFF;
        while(!(IFG2 & URXIFG1));
        *(buff+1) = U1RXBUF;

        /* If our response is a non-EZSP command, it may only
         * be two bytes in length.  So, we'll just return here.
         */
        if(*(buff+1) == 0xA7)
                return i;

        /* Else, determine the length of the message.  It it's an EZSP
         * message, the length is the value of buff[1].  Else, the
         * length of the message must be 0.  We add three to this
         * value to account for the inital byte, the length byte,
         * and the frame terminator.
         */
        len = ezsp ? *(buff+1) : 0;
        len += 3;

        /* Now, transmit 0xFF to clock the SPI and receive the next
         * byte of the response.  Do so until we have the entire
         * response.
         */
        do
        {
                /* Transmit 0xFF to the EM260. */
                while(!(IFG2 & UTXIFG1));
                U1TXBUF = 0xFF;

                /* Receive a response. */
                while(!(IFG2 & URXIFG1));
                *(buff+i) = U1RXBUF;

                /* Increment the number of bytes we've read. */
                ++i;
        } while(i < len);

        /* Return the number of bytes that we received. */
        return (i);
}

void toPC(unsigned char* buff, unsigned int len)
{
        /* Loop variable */
        unsigned int i;

        for(i = 0 ; i < len; i++)
        {
                /* Wait until the Tx buffer is empty. */
                while(!(IFG1 & UTXIFG0));

                /* Transmit the next byte in the buffer. */
                U0TXBUF = *(buff+i);
        }
}

unsigned char EM260_setConfigurationValue(unsigned char* params)
{
        /* Command buffer */
        unsigned char buff[9];

        /* Set up the command buffer. */
        buff[0] = 0xFE;       /* EZSP frame */
        buff[1] = 0x06;               /* Length of the EZSP frame */
        buff[2] = params[0];  /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
```

```
        buff[4] = 0x53;              /* EZSP: setConfigurationValue() command */
        buff[5] = params[1];   /* EZSP: Param: EZSP ConfigId */
        buff[6] = params[2];   /* EZSP: Param: new value (LSB) */
        buff[7] = params[3];   /* EZSP: Param: new value (MSB) */
        buff[8] = 0xA7;              /* Frame terminator */

        /* Send the command and receive the response. */
        P5OUT &= ~0x01;
        toEM260(buff, 9);
        //toPC(buff, 9);
        memset(buff, 0, 7);
        fromEM260(buff);
        //toPC(buff, 7);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

unsigned char EM260_networkInit(unsigned char sq)
{
        /* Command buffer */
        unsigned char buff[7];

        /* How many bytes we read from the 260 */
        unsigned char i;
        i = 0x00;

        /* Set up the command buffer. */
        buff[0] = 0xFE;              /* EZSP frame */
        buff[1] = 0x03;              /* Length of the EZSP frame */
        buff[2] = sq;         /* EZSP: sequence number */
        buff[3] = 0x00;              /* EZSP: command with no sleeping */
        buff[4] = 0x17;              /* EZSP: networkInit() command */
        buff[5] = 0xA7;              /* Frame terminator */

        /* Send the command and receive a response. */
        P5OUT &= ~0x01;
        toEM260(buff, 6);
        //toPC(buff, 6);
        memset(buff, 0, 6);
        i = fromEM260(buff);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;
        //toPC(buff, i);

        /* Return the result of the command. */
        return buff[5];
}

unsigned char EM260_leaveNetwork(unsigned char sq)
{
        /* Command buffer */
        unsigned char buff[7];

        /* Set up the command buffer. */
        buff[0] = 0xFE;              /* EZSP frame */
        buff[1] = 0x03;              /* Length of the EZSP frame */
        buff[2] = sq;         /* EZSP: sequence number */
        buff[3] = 0x00;              /* EZSP: command with no sleeping */
        buff[4] = 0x20;              /* EZSP: leaveNetwork() command */
        buff[5] = 0xA7;              /* Frame terminator */

        /* Send the command and receive a response. */
        P5OUT &= ~0x01;
        toEM260(buff, 6);
        //toPC(buff, 6);
```

```
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;
        //toPC(buff, 7);

        /* Return the result of the command. */
        return buff[5];
}

void EM260_formNetwork(unsigned char sq)
{
        /* Command buffer */
        unsigned char buff[19];

        /* Set up the command buffer. */
        buff[0] = 0xFE;        /* EZSP frame */
        buff[1] = 0x10;                /* Length of the EZSP frame */
        buff[2] = sq;          /* EZSP: sequence number */
        buff[3] = 0x00;                /* EZSP: command with no sleeping */
        buff[4] = 0x4F;                /* EZSP: scanAndFormNetwork() command */
        buff[5] = 0x00;                /* EZSP: Param: Channel Mask (LSB) */
        buff[6] = 0xF8;                /* EZSP: Param: Channel Mask */
        buff[7] = 0xFF;                /* EZSP: Param: Channel Mask */
        buff[8] = 0x07;                /* EZSP: Param: Channel Mask (MSB) */
        buff[9] = 0x03;                /* EZSP: Param: Radio Tx Power (dBm) */
        buff[10] = 0x1F;       /* EZSP: Param: Extended PAN ID (LSB) */
        buff[11] = 0x12;       /* EZSP: Param: Extended PAN ID */
        buff[12] = 0x02;       /* EZSP: Param: Extended PAN ID */
        buff[13] = 0xCA;       /* EZSP: Param: Extended PAN ID */
        buff[14] = 0x48;       /* EZSP: Param: Extended PAN ID */
        buff[15] = 0x25;       /* EZSP: Param: Extended PAN ID */
        buff[16] = 0x33;       /* EZSP: Param: Extended PAN ID */
        buff[17] = 0xF1;       /* EZSP: Param: Extended PAN ID (MSB) */
        buff[18] = 0xA7;       /* Frame terminator */

        /* Send the command. */
        P5OUT &= ~0x01;
        toEM260(buff, 19);
        nHOST_INT = 0;
        potentialCallback--;
        delay(0x0FFF);
        //toPC(buff, 19);
        P5OUT |= 0x01;
}

unsigned char EM260_permitJoining(unsigned char sq)
{
        /* Command buffer */
        unsigned char buff[7];

        /* Set up the command buffer. */
        buff[0] = 0xFE;        /* EZSP frame */
        buff[1] = 0x04;                /* Length of the EZSP frame */
        buff[2] = sq;          /* EZSP: sequence number */
        buff[3] = 0x00;                /* EZSP: command with no sleeping */
        buff[4] = 0x22;                /* EZSP: permitJoining() command */
        buff[5] = 0xFF;                /* EZSP: Param: duration (infinite) */
        buff[6] = 0xA7;                /* Frame terminator */

        /* Send the command and receive the response. */
        P5OUT &= ~0x01;
        toEM260(buff, 7);
        //toPC(buff, 7);
        memset(buff, 0, 7);
        fromEM260(buff);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;
```

```
	//toPC(buff, 7);

	/* Return the result of the command. */
	return buff[5];
}

void EM260_getNetworkParameters(unsigned char* buff, unsigned char sq)
{
	unsigned char i;

	/* Set up the command buffer. */
	buff[0] = 0xFE;		/* EZSP frame */
	buff[1] = 0x03;		/* Length of the EZSP frame */
	buff[2] = sq;		/* EZSP: sequence number */
	buff[3] = 0x00;		/* EZSP: command with no sleeping */
	buff[4] = 0x28;		/* EZSP: getNetworkParameters() command */
	buff[5] = 0xA7;		/* Frame terminator */

	/* Send the command and receive a response. */
	P5OUT &= ~0x01;
	toEM260(buff, 6);
	//toPC(buff, 6);
	memset(buff, 0 , 6);
	i = fromEM260(buff);
	nHOST_INT = 0;
	potentialCallback--;
	P5OUT |= 0x01;
	//toPC(buff, i);
}

void EM260_lookupNodeID(unsigned char* EUI, unsigned char* nodeID, unsigned char sq)
{
	/* Command bufer */
	unsigned char command[14];

	/* Set up the command buffer. */
	command[0] = 0xFE;	/* EZSP frame */
	command[1] = 0x0B;	/* Length of the EZSP frame */
	command[2] = sq;	/* EZSP: sequence number */
	command[3] = 0x00;	/* EZSP: command with no sleeping */
	command[4] = 0x60;	/* EZSP: lookupNodeIdByEui64() command */
	command[5] = EUI[7];	/* EZSP: Param: EUI64 (LSB) */
	command[6] = EUI[6];	/* EZSP: Param: EUI64 */
	command[7] = EUI[5];	/* EZSP: Param: EUI64 */
	command[8] = EUI[4];	/* EZSP: Param: EUI64 */
	command[9] = EUI[3];	/* EZSP: Param: EUI64 */
	command[10] = EUI[2];	/* EZSP: Param: EUI64 */
	command[11] = EUI[1];	/* EZSP: Param: EUI64 */
	command[12] = EUI[0];	/* EZSP: Param: EUI64 (MSB) */
	command[13] = 0xA7;	/* Frame terminator */

	/* Send the command and receive a response. */
	P5OUT &= ~0x01;
	toEM260(command, 14);
	//toPC(command, 14);
	memset(command, 0, 14);
	fromEM260(command);
	nHOST_INT = 0;
	potentialCallback--;
	P5OUT |= 0x01;
	//toPC(command, 8);

	/* Get the NodeID and put it in the return buffer. */
	memcpy(nodeID, command+5, 2);
}

void handleCallback(unsigned char* buff, unsigned char sq)
{
	/* The EUI64 of an incoming message's sender */
	unsigned char EUI[8];
```

```
/* The NodeID of an incoming message's sender */
unsigned char nodeID[2];

/* Buffer for outgoing messages */
unsigned char buff2[136];

/* Loop variable and counter */
unsigned char i;

/* Determine which callback we got. */
if(buff[4] == 0x00 || buff[4] == 0x07)
{
	/* There were no callbacks, or we have an error.
	 * Either way, we can ignore it.
	 */
	return;
}
else if(buff[4] == 0x62)
{
	/* We have an incoming EUI64, which will be
	 * followed by an incoming unicast.  We need
	 * to store this address, format the message,
	 * and send everything to the PC.
	 *
	 * We read the EUI64 in backwards because it
	 * was sent to us in a little endian format
	 * and the PC program wants it in big endian
	 * format.
	 */
	for(i = 0; i < 8; i++)
	{
		EUI[i] = buff[13-i];
	}

	/* Now we clear the buffer, wait at least 1ms,
	 * and get the actual message from the EM260.
	 */
	memset(buff, 0, 136);
	while(buff[4] != 0x45)
	{
		memset(buff, 0, 136);
		delay(0x00FF);
		i = EM260_callback(buff, sq);
	}

	/* The 24th byte is the length of the message. */
	i = *(buff+23);

	/* Get the sender's NodeID so we can reply to them. */
	memcpy(nodeID, buff+24, 2);

	/* Now, we must format our string to look like
	 * the one the PC is expecting.
	 */
	snprintf(buff2, 26, "\r\nUCAST:%.2x%.2x%.2x%.2x%.2x%.2x%.2x%.2x=",
		(int)EUI[0], (int)EUI[1], (int)EUI[2], (int)EUI[3],
		(int)EUI[4], (int)EUI[5], (int)EUI[6], (int)EUI[7]);

	memcpy(buff2+25, buff+26, i-4);
	memcpy(buff2+21+i, "\r\n", 2);

	/* Now we send the message to the PC. */
	toPC(buff2, 23+i);
	potentialCallback--;

	/* If the message was an IMAGE_READY signal, we need to store
	 * the node's ID and set imageReady so that we can reply
	 * later.
	 */
```

```
                if(strncmp(buff2+25, "IMAGE_READY", 11) == 0)
                {
                        if(!imageReady)
                        {
                                imageReady = 1;
                                memcpy(irID, nodeID, 2);
                        }
                }

                /* Finally, we sent a message back to the node to let it know
                 * that it's okay to send another message.
                 */
                memset(buff, 0, 136);
                strncpy(buff, "OK", 3);
                EM260_sendUnicast(nodeID, buff, 2, sq++, 0);
        }
        else
        {
                /* We can ignore all other callbacks. */
                return;
        }
}

unsigned char EM260_callback(unsigned char* buff, unsigned char sq)
{
        /* How many bytes the response is */
        unsigned char i;

        /* Set up the command buffer. */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x03;               /* Length of the EZSP frame */
        buff[2] = sq;          /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x06;               /* EZSP: callback() command */
        buff[5] = 0xA7;               /* Frame terminator */

        /* Send the command and receive a response. */
        P5OUT &= ~0x01;
        toEM260(buff, 6);
        //toPC(buff, 6);
        memset(buff, 0, 6);
        //delay(0x0FFF);
        i = fromEM260(buff);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;
        //toPC(buff, i);

        /* Return how many bytes the callback response is. */
        return i;
}

unsigned char EM260_sendUnicast(unsigned char* addr, unsigned char* message, unsigned char len,
unsigned char sq, unsigned char sq2)
{
        /* Command buffer */
        unsigned char command[22+len];

        /* Loop variable and counter */
        unsigned int i;

        /* Set up the command buffer. */
        command[0] = 0xFE;     /* EZSP frame */
        command[1] = 19 + len; /* Length of the EZSP frame */
        command[2] = sq;       /* EZSP: sequence number */
        command[3] = 0x00;     /* EZSP: command with no sleeping */
        command[4] = 0x34;     /* EZSP: sendUnicast() command */
        command[5] = 0x00;     /* EZSP: Param: message type (direct) */
        command[6] = addr[0];  /* EZSP: Param: nodeID (LSB) */
        command[7] = addr[1];  /* EZSP: Param: nodeID (MSB) */
```

```
        command[8] = 0x00;     /* EZSP: Param: profileID (LSB) */
        command[9] = 0x00;     /* EZSP: Param: profileID (MSB) */
        command[10] = 0x00;    /* EZSP: Param: clusterID (LSB) */
        command[11] = 0x00;    /* EZSP: Param: clusterID (MSB) */
        command[12] = 0x01;    /* EZSP: Param: source endPoint */
        command[13] = 0x01;    /* EZSP: Param: destination endPoint */
        command[14] = 0x40;    /* EZSP: Param: APS options (LSB) */
        command[15] = 0x05;    /* EZSP: Param: APS options (MSB) */
        command[16] = 0x00;    /* EZSP: Param: groupID (LSB) */
        command[17] = 0x00;    /* EZSP: Param: groupID (MSB) */
        command[18] = sq2;     /* EZSP: Param: sequence number */
        command[19] = 0x64;    /* EZSP: Param: message tag */
        command[20] = len;     /* EZSP: Param: message length */
        for(i = 0; i < len; i++)
        {
                command[i+21] = message[i];   /* EZSP: Param: message */
        }
        command[21+len] = 0xA7;       /* Frame terminator */

        /* Send the command and receive a response. */
        P5OUT &= ~0x01;
        toEM260(command, 22+len);
        //toPC(command, 22+len);
        memset(command, 0, 22+len);
        i = fromEM260(command);
        //toPC(command, i);
        nHOST_INT = 0;
        potentialCallback--;
        P5OUT |= 0x01;

        /* Return the result of the command. */
        return command[5];
}

void main(void)
{
        /* Message buffers */
        unsigned char buff1[136];
        unsigned char buff2[136];

        /* Our network information */
        unsigned char channel;
        unsigned char panID[2];

        /* Sequence number for SPI transactions */
        unsigned char sq;

        /* Sequence number for unicast messages. */
        unsigned char sq2;

        /* Length of unicast messages, or number of bytes read with
         * fromEM260() or fromPC()
         */
        unsigned char len;

        /* The status of the network (0 = DOWN, 1 = UP) */
        unsigned char netStat;

        /* Status returned from EM260 command */
        unsigned char status;

        status = 0x01;
        sq2 = 0x00;
        imageReady = 0;

        /* Initialize hardware. */
        initMSP();
        initUART0();
        initSPI1();
        while(status != 0x00)
```

```
        status = initEM260();

/* Initialize the sequence number to 1 since we used sq = 0 in
 * the initEM260() call.
 */
sq = 0x01;

/* Configure the EM260 to function properly.  The fist configuration
 * call turns on boost mode (EZSP_CONFIG_TX_POWER_MODE while the
 * second enables meissage relaying (EZSP_CONFIG_DISABLE_RELAY).
 * Finally, we turn off security.
 */
delay(0xFFFF);
buff1[0] = sq++;
buff1[1] = 0x17;
buff1[2] = 0x01;
buff1[3] = 0x00;
status = 0x01;
while(status != 0x00)
        status = EM260_setConfigurationValue(buff1);
memset(buff1, 0, 4);

delay(0xFFFF);
buff1[0] = sq++;
buff1[1] = 0x18;
buff1[2] = 0x00;
buff1[3] = 0x00;
status = 0x01;
while(status != 0x00)
        status = EM260_setConfigurationValue(buff1);
memset(buff1, 0, 4);

delay(0xFFFF);
buff1[0] = sq++;
buff1[1] = 0x0D;
buff1[2] = 0x00;
buff1[3] = 0x00;
status = 0x01;
while(status != 0x00)
        status = EM260_setConfigurationValue(buff1);
memset(buff1, 0, 4);

/* Now, we make a call to networkInit().  This should be done every
 * time we reboot, regardless of whether or not we used to be on
 * a network.  If we are on a network after networkInit(), we
 * must leave that network.
 */
delay(0xFFFF);
status = EM260_networkInit(sq++);
if(status != 0x93)
{
        P2OUT &= ~0x10; /* TODO: DEBUG */
        delay(0xFFFF);
        while(!nHOST_INT);
        len = EM260_callback(buff1, sq);
        memset(buff1, 0, len);

        status = 0x01;
        while(status != 0x00)
        {
                delay(0xFFFF);
                status = EM260_leaveNetwork(sq++);
        }

        delay(0xFFFF);
        while(!nHOST_INT);
        len = EM260_callback(buff1, sq);
        memset(buff1, 0, len);
}
```

```
/* Now, we're certainly not on a network. */
netStat = 0;
channel = 0x00;
panID[0] = 0x00;
panID[1] = 0x00;

P2OUT |= 0x10; /* TODO: DEBUG */

/* Turn the LED on to let the adminstrator know that we're ready. */
P6OUT &= ~0x01;
potentialCallback = 0;

/* TODO: DEBUG */
P2OUT &= ~0x20;

/* Now, we simply process input from the PC and from the EM260. */
while(1)
{
	/* Determine whether or not we have a command from the PC. */
	if(pcINT)
	{
		/* If so, we'll deal with it. */
		pcINT = 0;
		len = pcI;
		pcI = 0;

		/* Read the command from the PC. */
		if(strncmp(pcBuff, "AT+UCAST", 8) == 0)
		{
			/* We attempt to send the message to the
			 * NodeID that we stored when we got an
			 * IMAGE_READY signal.  We don't send any
			 * messages but IMAGE_READY_OK, so this
			 * works.  It's not optimal, but can
			 * easily be extended to support all
			 * outgoing unicasts.
			 */
			status = EM260_sendUnicast(irID, pcBuff+26, len-27, sq++, sq2++);
			if(status != 0x00)
			{
				snprintf(buff2, 11, "ERROR:%.2x\n\r",
					(int)status);
				toPC(buff2, 10);
				memset(buff2, 0, 11);
			}
			else
			{
				snprintf(buff2, 5, "OK\n\r");
				toPC(buff2, 4);
				memset(buff2, 0, 5);
				imageReady = 0;
			}

			memset(pcBuff, 0, len);
		}
		else if(strncmp(pcBuff, "AT+EN", 5) == 0)
		{
			memset(pcBuff, 0, len);

			/* If we're already joined to the network,
			 * let the PC know that everything is OK.
			 * Else, we must attempt to form the network
			 * and let the PC know the result of our
			 * attempt.
			 */
			if(netStat)
			{
				snprintf(buff2, 15, "JPAN:%.2x,%.2x%.2x\n\r",
					(int)channel, (int)panID[0],
					(int)panID[1]);
```

```
                toPC(buff2, 14);
                memset(buff2, 0, 15);
        }
        else
        {
                delay(0x0FFF);
                EM260_formNetwork(sq++);
                while(!nHOST_INT);
                P2OUT &= ~0x80; /* TODO: DEBUG */
                delay(0x0FFF);
                len = EM260_callback(buff1, sq);
                memset(buff1, 0, len);
                while(!nHOST_INT);
                P2OUT &= ~0x40; /* TODO: DEBUG */
                delay(0x0FFF);
                len = EM260_callback(buff1, sq);
                if(buff1[5] != 0x90)
                {
                        snprintf(buff2, 11,
                                "ERROR:%.2x\n\r",
                                (int)buff1[5]);
                        toPC(buff2, 10);
                        memset(buff2, 0, 11);
                        netStat = 0;
                        P6OUT |= 0x10; /* TODO: DEBUG */
                        memset(buff1, 0, len);
                        continue;
                }

                delay(0x0FFF);
                status = EM260_permitJoining(sq++);
                delay(0x0FFF);
                EM260_getNetworkParameters(\
                        buff2, sq++);
                channel = buff2[18];
                panID[0] = buff2[16];
                panID[1] = buff2[15];
                snprintf(buff2, 15, "JPAN:%.2x,%.2x%.2x\n\r",
                        (int)channel,
                        (int)panID[0],
                        (int)panID[1]);
                toPC(buff2, 14);
                memset(buff2, 0, 15);
                netStat = 1;
                P2OUT &= ~0x10; /* TODO: DEBUG */
                memset(buff1, 0, len);
        }
}
else if(strncmp(pcBuff, "AT+DASSL", 8) == 0)
{
        memset(pcBuff, 0, len);

        /* If we're not joined to a network, there
         * is no reason to try to leave one.  So
         * we just tell the PC that everything is
         * okay.
         */
        if(!netStat)
        {
                snprintf(buff1, 10,
                        "LeftPAN\r\n");
                toPC(buff1, 9);
                memset(buff1, 0, 10);
                continue;
        }

        /* Else, if we are joined to a network, we
         * must now leave it and let the PC know
         * if we were successful in doing so.
         */
```

```
                        delay(0x0FFF);
                        status = EM260_leaveNetwork(sq++);
                        P2OUT |= 0x10;
                        delay(0x0FFF);
                        if(status == 0x00)
                        {
                                while(!nHOST_INT);
                                len = EM260_callback(buff1, sq);
                                P2OUT |= 0x40;
                                if(buff1[5] == 0x91)
                                {       netStat = 0;
                                        channel = 0x00;
                                        panID[0] = 0x00;
                                        panID[1] = 0x00;
                                        //P2OUT |= 0x80; /* TODO: DEBUG */
                                        P2OUT |= 0x80;
                                        memset(buff1, 0, len);
                                        snprintf(buff1, 10,
                                                "LeftPAN\r\n");
                                        toPC(buff1, 9);
                                        memset(buff1, 0, 10);
                                }
                                else
                                {
                                        memset(buff1, 0, len);
                                        snprintf(buff1, 11,
                                                "ERROR:%.2x\r\n",
                                                (int)buff1[5]);
                                        toPC(buff1, 10);
                                        memset(buff1, 0, 11);
                                }
                        }
                        else
                        {
                                snprintf(buff1, 11,
                                        "ERROR:%.2x\r\n",
                                        (int)status);
                                toPC(buff1, 10);
                                memset(buff1, 0, 11);
                        }
                }
                else
                {
                        /* We ignore all other commands as they do
                         * not matter to us.
                         */
                        memset(pcBuff, 0, len);
                }
        }

        /* Send a callback to the EM260 to find out if it has
         * anything for us.  If so, we need to handle it.
         */
        if(potentialCallback)
        {
                delay(0x00FF);
                EM260_callback(buff1, sq++);
                delay(0x00FF);
                handleCallback(buff1, sq++);
                memset(buff1, 0, 136);
                potentialCallback--;
        }
    }
}
```

Appendix G: PC Base Station Code

em250.h:

```
PC Base Station
Copyright (C) 2007 Joseph Bosman, Steve Olivieri, Ipek Ozil, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <Windows.h>

#ifndef EM250_H
#define EM250_H

/* Return codes for main() */
#define MAIN_SUCCESS 0
#define MAIN_INPUT_ERROR 1
#define MAIN_SERIAL_PORT_FAIL 2
#define MAIN_LOG_FILE_FAIL 3
#define MAIN_OUTPUT_FILE_FAIL 4
#define MAIN_DATA_WRITE_ERROR 5
#define MAIN_READ_ERROR 6
#define MAIN_JOIN_FAIL 7
#define MAIN_COO_ERROR 8
#define MAIN_NO_CONNECTION 9
#define MAIN_REGISTER_FAIL 10

/* Return codes for compareTimeStructs() */
#define GREATER_THAN 1
#define LESS_THAN -1
#define EQUAL 0

/* Generic return codes for functions. */
#define SUCCESS 0
#define FAILURE 1

/* Sets the year, month, day, hour, minute, and second members of a SYSTIME
 * struct to 0.
 *      t: pointer to the SYSTIME to initialize
 */
void initTimeStruct(LPSYSTEMTIME t);

/* Sets two SYSTEMTIME structs equal to each other.  The first is filled with
 * the values from the second.
 *      t1: pointer to the SYSTEMTIME to be set
 *      t2: pointer to the SYSTEMTIME to get t1's new values from
 */
void setEqualTimeStructs(LPSYSTEMTIME t1, LPSYSTEMTIME t2);

/* Compares two SYSTIME structs to determine which has a more recent time.
```

```
 * If the first has a more recent time, GREATER_THAN is return.  If the second,
 * LESS_THAN.  If the two have the same time, return EQUAL.
 *      t1: pointer to the first SYSTIME to compare
 *      t2: pointer to the second SYSTIME to compare
 */
int compareTimeStructs(LPSYSTEMTIME t1, LPSYSTEMTIME t2);

/* Set up a COM port for communication.  This includes setting the baud rate, number
 * of bits and stop bits, etc.  We also set timeout values here and purge the input/output
 * buffers.
 *      port: the COM port to use, as a string (eg. COM1:)
 */
HANDLE setupCOM(LPCSTR port);

/* Writes data to the serial port.  Data should be formatted as a standard,
 * null-terminated string.  No LF or CR characters are necessary at the end.
 * If successful, just return 0.  Else, return non-zero.
 *      data: the data to send
 *      comPort: a pointer to the file handle for the serial port to write to
 *      len: the length of the data buffer
 *      command: true if this is an AT command, else false
 */
int writeSerial(LPCSTR data, PHANDLE comPort, UINT len, BOOL command);

/* Reads data from a file and stores it in the buffer pointed to by buff.
 * The value return is the number of bytes read.
 *      in: the file to read from
 *      buff: the buffer to store the read data in
 */
int readMessage(PHANDLE in, LPSTR buff);

/* Formats a message received from another ZigBee node so that it's more
 * useful to us.  The message is split into three buffers.  One contains the
 * actual message, another the hardware ID of the sender, and the third
 * contains both.  This function supports readable data (strings) as well
 * as binary data.  If binary data, binary should be 1, else 0.  This function
 * returns the length of the actual message (tBuff).
 *      tBuff: buffer to place the actual message in
 *      buff: buffer to place tBuff and hwID in
 *      hwID: buffer to place the hardware ID of the sender in
 *      binary: are we looking at binary data or readable data?
 *      l_tBuff: length of tBuff
 *      l_buff: length of buff
 *      l_hwID: length of hwID
 */
int formatMessage(LPSTR tBuff, LPSTR buff, LPSTR hwID, INT binary, INT l_tBuff, INT l_buff, INT
l_hwID);

/* Reads the input buffer of the COM port until it finds either the desired
 * message or an "ERROR:" string.  If either is found, a pointer to the first
 * character in the found message is returned.  This function will block until
 * either the desired message or "ERROR:" is found, so use it with caution.
 *      in: pointer to the handle of the input file (COM port)
 *      message:       the message to search for
 *      out:   the file to log all read data to (can be NULL)
 */
LPSTR getMessage(PHANDLE in, LPCSTR message, PHANDLE out);

/* Returns the ceiling of x.  This function exists because for some reason I
 * can't figure out how to use a built-in function in Windows.
 *      x: the number to get the ceiling of
 */
DWORD ceiling(double x);

#endif /* EM250_H */
```

em250.c:

```
PC Base Station
Copyright (C) 2007 Joseph Bosman, Steve Olivieri, Ipek Ozil, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include "em250.h"

void initTimeStruct(LPSYSTEMTIME t)
{
	/* Set the date to 00/00/0000, 00:00:00. */
	t->wYear = 0;
	t->wMonth = 0;
	t->wDay = 0;
	t->wHour = 0;
	t->wMinute = 0;
	t->wSecond = 0;

	return;
}

void setEqualTimeStructs(LPSYSTEMTIME t1, LPSYSTEMTIME t2)
{
	/* Set t1 equal to t2 to second precision. */
	t1->wYear = t2->wYear;
	t1->wMonth = t2->wMonth;
	t1->wDay = t2->wDay;
	t1->wHour = t2->wHour;
	t1->wMinute = t2->wMinute;
	t1->wSecond = t2->wSecond;

	return;
}

int compareTimeStructs(LPSYSTEMTIME t1, LPSYSTEMTIME t2)
{
	/* Compare the two dates from the largest unit to the smallest.
	 * If at any time one is larger than the other, stop comparing.
	 */
	if(t1->wYear > t2->wYear)
		return GREATER_THAN;
	if(t2->wYear > t1->wYear)
		return LESS_THAN;

	if(t1->wMonth > t2->wMonth)
		return GREATER_THAN;
	if(t2->wMonth > t1->wMonth)
		return LESS_THAN;

	if(t1->wDay > t2->wDay)
		return GREATER_THAN;
	if(t2->wDay > t1->wDay)
		return LESS_THAN;

	if(t1->wHour > t2->wHour)
```

```
                return GREATER_THAN;
        if(t2->wHour > t1->wHour)
                return LESS_THAN;

        if(t1->wMinute > t2->wMinute)
                return GREATER_THAN;
        if(t2->wMinute > t1->wMinute)
                return LESS_THAN;

        if(t1->wSecond > t2->wSecond)
                return GREATER_THAN;
        if(t2->wSecond > t1->wSecond)
                return LESS_THAN;

        /* The two dates are equal, to the precision of a second. */
        return EQUAL;
}

HANDLE setupCOM(LPCSTR port)
{
        /* Serial port configuration settings. */
        DCB comConfig;

        /* Serial port timeout settings. */
        COMMTIMEOUTS comTimeouts;

        /* The file handle to return to the caller. */
        HANDLE ret;

        /* Open the serial port. */
        ret = CreateFile((LPCSTR)port, GENERIC_READ|GENERIC_WRITE, 0, NULL,
                OPEN_EXISTING, 0, NULL);
        if(ret == INVALID_HANDLE_VALUE)
                return INVALID_HANDLE_VALUE;

        /* Get the port's configuration settings, then change them to better
         * match the ETRX2USB device.
         */
        if(GetCommState(ret, &comConfig))
        {
                #ifdef TELEGESIS
                  comConfig.BaudRate = 19200;
                #else
                  comConfig.BaudRate = 38400;
                #endif
                comConfig.ByteSize = 8;
                comConfig.Parity = NOPARITY;
                comConfig.StopBits = ONESTOPBIT;
                comConfig.fBinary = FALSE;
                comConfig.fParity = FALSE;
        }
        else
                return INVALID_HANDLE_VALUE;
        if(!SetCommState(ret, &comConfig))
                return INVALID_HANDLE_VALUE;

        /* Get the port's timeout settings, then set them all to 25 ms.  This
         * is probably not an ideal solution.
         */
        if(GetCommTimeouts(ret, &comTimeouts))
        {
                comTimeouts.ReadIntervalTimeout = 25;
                comTimeouts.ReadTotalTimeoutConstant = 25;
                comTimeouts.ReadTotalTimeoutMultiplier = 25;
                comTimeouts.WriteTotalTimeoutConstant = 25;
                comTimeouts.WriteTotalTimeoutMultiplier = 25;
        }
        else
                return INVALID_HANDLE_VALUE;
        if(!SetCommTimeouts(ret, &comTimeouts))
```

```
                return INVALID_HANDLE_VALUE;

        /* Empty the input and output buffers for the serial port. */
        PurgeComm(ret, PURGE_RXCLEAR|PURGE_TXCLEAR);

        /* Return the file handle for the serial port. */
        return ret;
}

int writeSerial(LPCSTR data, PHANDLE comPort, UINT len, BOOL command)
{
        /* Single-character buffer for writing strings to the serial port. */
        CHAR zBuff;

        /* How many bytes we wrote on the last write call. */
        DWORD bWritten;

        /* How many bytes we've read. */
        DWORD bRead;

        /* Loop variable. */
        UINT i;

        /* Write a command to the serial port. */
        for(i = 0; i < len; i++)
        {
                zBuff = *(data+i);
                if(!WriteFile(*comPort, (LPVOID)&zBuff, 1, &bWritten, NULL))
                        return FAILURE;
        }

        /* Send a carriage return to finish the command. */
        if(command)
        {
                zBuff = '\r';
                if(!WriteFile(*comPort, (LPVOID)&zBuff, 1, &bRead, NULL))
                        return FAILURE;
        }

        return SUCCESS;
}

int readMessage(PHANDLE in, LPSTR buff)
{
        /* Temporary buffer. */
        CHAR zBuff;

        /* Stores how many bytes the last ReadFile() read. */
        DWORD bRead;

        /* Stores how many total bytes we've read. */
        INT i;

        /* Initialize i.  Then, read in the data one byte at a
         * time and write it to buff.  Data is read this way for COM
         * support.  We finish when there is no more data to be read.
         */
        i = 0;
        do
        {
                ReadFile(*in, (LPVOID)&zBuff, 1, &bRead, NULL);
                if(bRead > 0)
                {
                        *(buff+i) = zBuff;
                        ++i;
                }
        } while (bRead > 0);
        *(buff+i) = '\0';

        return i;
```

```
}

LPSTR getMessage(PHANDLE in, LPCSTR message, PHANDLE out)
{
	/* Temporary buffer for storing data that we read from the COM port. */
	LPSTR buff;

	/* Temporary buffer that points to part of buff containing the desired
	 * message (if it exists) or nothing (otherwise).
	 */
	LPSTR tBuff;

	/* We copy tBuff into a separate buffer for returning to avoid memory
	 * leaks.
	 */
	LPSTR retBuff;

	/* How many bytes we wrote. */
	DWORD bWritten;

	/* How many bytes we read. */
	INT i;

	/* Since we're storing temporary data in buff, we need to allocate
	 * memory for it now.  Since we use tBuff as a BOOL, we leave it
	 * NULL for now and only give it a value when we find the message
	 * that we're looking for (or an ERROR).
	 */
	buff = NULL;
	buff = (LPSTR)calloc(150, sizeof(CHAR));
	tBuff = NULL;

	/* While we haven't found the desired message (or an ERROR), keep
	 * looking!
	 */
	while(!tBuff)
	{
		/* Read data from the COM port. */
		memset(buff, 0, 150);
		i = readMessage(in, buff);
		if(*(buff+i) != '\0')
			*(buff+i) = '\0';

		/* Search that data for our desired message. If we don't find it
		 * search for an ERROR message.
		 */
		tBuff = strstr(buff, message);
		if(!tBuff)
			tBuff = strstr(buff, "ERROR:");

		/* If out is not NULL, write any data that we've read to the file
		 * pointed to by out.  Usually, this is a log file.  And, we don't
		 * really care if it fails.
		 */
		if(out)
			WriteFile(*out, (LPVOID)buff, i, &bWritten, NULL);

	}

	/* Since we have something to return, let's allocate memory for retBuff
	 * and then copy the useful data into it.  The caller of getMessage()
	 * is responsible for freeing retBuff.
	 */
	retBuff = (LPSTR)strdup(tBuff);

	free(buff);
	free(tBuff);
	return retBuff;
}
```

```
int formatMessage(LPSTR tBuff, LPSTR buff, LPSTR hwID, INT binary, INT l_tBuff, INT l_buff, INT
l_hwID)
{
	/* How many bytes are in the real message. */
	UINT i;
	i = 0;

	/* The message contains two important things.  The hardware ID number
	 * of the node that it originated from, and the actual data of the
	 * message.  Messages begin with two useless characters, followed by
	 * a message type (NEWNODE, UCAST, etc.) and then a colon.  Next, the
	 * hardware ID number (16 hex digits), an equals sign (=), and finally
	 * the message.  Note that sytem commands (such as NEWNODE) do not
	 * contain anything after the hardware ID number.  Some commands have
	 * a space (' ') between the colon and the message, while others
	 * do not.  At the end, we have the following:
	 *	tBuff: the message
	 *	hwID: the hardware ID of the sender
	 *	buff: hwID=buff
	 */

	/* Remove the leading and trailing characters from the initial message. */
	memset(tBuff, 0, l_tBuff);
	memcpy(tBuff, buff+2, l_buff-5);

	/* Copy everything after the colon into buff. */
	memset(buff, 0, l_buff);
	memcpy(buff, (LPSTR)memchr(tBuff, (INT)':', l_buff)+1, l_buff);

	/* If this command places a space between the colon and the message,
	 * remove it.
	 */
	if(memchr(buff, (INT)' ', 1))
	{
		memset(tBuff, 0, l_tBuff);
		memcpy(tBuff, buff+1, l_buff);
		memset(buff, 0, l_buff);
		memcpy(buff, tBuff, l_tBuff);
	}

	/* If this is a binary message, we need to figure out how many bytes to
	 * read.  This number is stored in the last two characters of the
	 * message.  Between this number and the actual message is a comma.
	 * Also, this is a convenient time to get the hardware ID.
	 */
	if(binary)
	{
		memset(hwID, 0, l_hwID);
		memcpy(hwID, buff+17, 2);
		i = strtol(hwID, NULL, 16);
		memset(hwID, 0, l_hwID);
		memcpy(hwID, buff, 16);
	}
	else
	{
		/* Copy the hardware ID into hwID. */
		memset(hwID, 0, l_hwID);
		memcpy(hwID, buff, 16);
	}

	/* Finally, copy the message into tBuff. */
	if(binary)
	{
		memset(tBuff, 0, l_tBuff);
		memcpy(tBuff, buff+20, l_buff);
	}
	else
	{
		memset(tBuff, '\0', l_tBuff);
		memcpy(tBuff, buff+17, l_buff);
```

```
                tBuff[l_tBuff-1] = '\0';
        }

        /* For a binary string, return i (hex).  Else, return the length
         * of tBuff (decimal).  Both should be the length of the actual
         * message.
         */
        return (binary ? i : strlen(tBuff));
}

int main(int argc, char* argv[])
{
        /* Handle for the log file we're writing to */
        HANDLE log;

        /* Handle for the file we're reading from. */
        HANDLE serial;

        /* Handle for the image file we're writing. */
        HANDLE imageFile;

        /* The COM port to connect to. */
        LPSTR comPort;

        /* Directory to store files in. */
        LPSTR fileDir;

        /* Filename for the output file. */
        LPSTR fileName;

        /* Temporary filename for moving files. */
        /*LPSTR tmpName;*/

        /* Buffers for storing text to be written to a file. */
        LPSTR buff;
        LPSTR tBuff;

        /* The hardware ID number for the sender. */
        LPSTR hwID;
        LPSTR hwID2;

        /* Storage for how many bytes we wrote with WriteFile(). */
        DWORD bWritten;

        /* Storage for how many bytes we read with ReadFile(). */
        DWORD bRead;

        /* Event Mask for COM port. */
        DWORD eventMask;

        /* The number of packets that we expect an image to be. */
        DWORD numPackets;

        /* The number of packets that we've receieved so far. */
        DWORD recPackets;

        /* Stores how many files we've made today. */
        INT filesMade;

        /* Stores the current local time. */
        SYSTEMTIME currTime;

        /* Stores the last local time used. */
        SYSTEMTIME lastTime;

        /* Used as a timer for timeouts. */
        SYSTEMTIME timer;

        /* How long the last image transfer took in milliseconds. */
        DWORD tSec;
```

```
/* Loop variable. */
INT i;

/* Make sure that they specify the required command
 * arguments (COM port and directory).
 */
if(argc < 5)
	return MAIN_INPUT_ERROR;

/* Allocate memory for our strings. */
comPort = (LPSTR)calloc(6, sizeof(CHAR));
buff = (LPSTR)calloc(150, sizeof(CHAR));
hwID = (LPSTR)calloc(17, sizeof(CHAR));
hwID2 = (LPSTR)calloc(17, sizeof(CHAR));
fileDir = (LPSTR)calloc(999, sizeof(CHAR));
fileName = (LPSTR)calloc(1024, sizeof(CHAR));
/*tmpName = (LPSTR)calloc(1024, sizeof(CHAR));*/
tBuff = NULL;

/* Initialize variables. */
initTimeStruct(&currTime);
initTimeStruct(&lastTime);
initTimeStruct(&timer);
filesMade = 0;
bRead = 0;
bWritten = 0;
numPackets = 0x0;
recPackets = 0x0;

/* Parse input. */
for(i = 1; i < argc; i++)
{
	if(strcmp(argv[i], "-c") == 0)
		sprintf(comPort, "COM%s:", argv[++i]);
	else if(strcmp(argv[i], "-d") == 0)
		sprintf(fileDir, "%s", argv[++i]);
	else
		return MAIN_INPUT_ERROR;
}

/* Open the serial port. */
serial = setupCOM((LPCSTR)comPort);
if(serial == INVALID_HANDLE_VALUE)
{
	fprintf(stderr, "Could not open %s.\n", comPort);
	free(comPort);
	free(buff);
	free(hwID);
	free(hwID2);
	free(fileDir);
	free(fileName);
	return MAIN_SERIAL_PORT_FAIL;
}

/* Create the log file to write to. */
GetLocalTime((LPSYSTEMTIME)&currTime);
sprintf(fileName, "%sLOG_%.2d%.2d%.2d_%.2d%.2d%.2d.txt", fileDir,
	currTime.wDay, currTime.wMonth, currTime.wYear,
	currTime.wHour, currTime.wMinute, currTime.wSecond);
log = CreateFile((LPCSTR)fileName, GENERIC_WRITE, FILE_SHARE_READ,
	NULL, CREATE_ALWAYS, FILE_FLAG_SEQUENTIAL_SCAN, NULL);
if(log == INVALID_HANDLE_VALUE)
{
	fprintf(stderr, "Failed to open the log file: %s.\n", fileName);
	CloseHandle((PVOID)&serial);
	free(comPort);
	free(buff);
	free(tBuff);
	free(hwID);
```

```
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_LOG_FILE_FAIL;
}

/* Tell the EM250 to leave any PANs that it might be in. If we get an
 * ERROR here, it's because we weren't already in a network.  We can
 * ignore that error.
 */
if(writeSerial("AT+DASSL", &serial, strlen("AT+DASSL"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_DATA_WRITE_ERROR;
}

strncpy(buff, "LeftPAN", 8);
tBuff = getMessage(&serial, buff, &log);
free(tBuff);
memset(buff, 0, 8);

#ifdef TELEGESIS
/* Now, we need to turn on local echo, because it might have been off. */
if(writeSerial("ATS0B4=0", &serial, strlen("ATS0B4=0"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_DATA_WRITE_ERROR;
}

strncpy(buff, "OK", 3);
tBuff = getMessage(&serial, "OK", &log);
if(strstr(tBuff, "ERROR:"))
{
        fprintf(stderr, "Could not set register value 0B4, %s.\n", tBuff);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        free(tBuff);
        return MAIN_REGISTER_FAIL;
}
free(tBuff);
memset(buff, 0, 3);

/* We also need to turn off the ability to have multiple packets in
 * flight at once.  This makes it much easier to ensure proper message
 * delivery.
 */
if(writeSerial("ATS06D=1:TG-ETRX1", &serial, strlen("ATS06D=1:TG-ETRX1"), TRUE))
{
```

```
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_DATA_WRITE_ERROR;
}

strncpy(buff, "OK", 3);
tBuff = getMessage(&serial, "OK", &log);
if(strstr(tBuff, "ERROR:"))
{
        fprintf(stderr, "Could not set register value 06D, %s.\n", tBuff);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        free(tBuff);
        return MAIN_REGISTER_FAIL;
}
free(tBuff);
memset(buff, 0, 3);
#endif /* TELEGESIS */

/* Estabish a new PAN with this device as the coordinator.  We wait for
 * three responses because the ETRX2USB device sends the response in
 * three sections (and we'll time out between them).
 */
if(writeSerial("AT+EN", &serial, strlen("AT+EN"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_DATA_WRITE_ERROR;
}

strncpy(buff, "JPAN:", 6);
tBuff = getMessage(&serial, "JPAN:", &log);
if(!strstr(tBuff, "JPAN:"))
{
        fprintf(stderr, "Could not establish PAN.\n");
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        free(tBuff);
        return MAIN_JOIN_FAIL;
}
memset(buff, 0, 6);
strncpy(buff, tBuff+5, 7);
fprintf(stdout, "Established network %s on channel %c%c.\n",
        buff+3, *buff, *(buff+1));
```

```
free(tBuff);
memset(buff, 0, 150);

/* Now, we need to make sure that our buffers are completely clear. */
PurgeComm(&serial, PURGE_RXCLEAR);

/* From now on, we're only concerned with receiving images.  The software
 * on the ETRX2USB device should be handling any new nodes, routing
 * things, etc.
 */
while(1)
{
	/* Wait for data to arrive. */
	SetCommMask(serial, EV_RXCHAR);
	WaitCommEvent(serial, &eventMask, NULL);
	tBuff = NULL;

	/* Make sure our buffers are clear. */
	memset(buff, 0, 150);
	memset(hwID, 0, 17);

	/* Get the message and write it to the log file. */
	i = readMessage(&serial, buff);
	WriteFile(log, (LPVOID)buff, i, &bWritten, NULL);

	/* Format the message so we can use it. If it doesn't have at
	 * least 16 characters, it doesn't have a hardware ID and thus
	 * we don't want to actually do anything with it.
	 */
	if(i >= 16)
	{
		tBuff = (LPSTR)calloc(150, sizeof(CHAR));
		formatMessage(tBuff, buff, hwID, 0, 150, 150, 17);
	}
	else
	{
		continue;
	}

	/* Now, we need to see what the message was.  If a picture is ready,
	 * then we need to act on that signal.  Otherwise, we can just ignore
	 * the message (we already wrote it to the log file).
	 */
	if(strncmp(tBuff, "IMAGE_READY", 11) == 0)
	{
		/* Determine how many packets the image will be.  This number
		 * should be a four-digit hex value in the form 0xNNNN where
		 * N ranges from 0 to F.  Since we haven't received any packets
		 * yet, recPackets must be 0.
		 */
		numPackets = strtol(tBuff+11, NULL, 16);
		recPackets = 0x0;
		free(tBuff);

		/* Store our hardware ID for future reference. */
		memset(hwID2, 0, 17);
		strcpy(hwID2, hwID);

		/* Before we send the OK to the sender, we must first create
		 * the new image file.  To do so, we must obtain the current
		 * time.  Then, we compare it with the last time that we made
		 * an image.  If they match, we increment filesMade.  Else, we
		 * set filesMade to 0 because we haven't yet made any files with
		 * this timestamp.
		 *
		 * In both cases, we then set lastTime equal to currentTime for
		 * future comparisons.
		 */
		GetLocalTime((LPSYSTEMTIME)&currTime);
		if(compareTimeStructs(&currTime, &lastTime) == 0)
```

```
        ++filesMade;
else
        filesMade = 0;
setEqualTimeStructs(&lastTime, &currTime);

/* Next, we determine the filename for the new file and then
 * create it.  Filenames are of the format:
 *      ddmmyyyy_hhmmss_x.jpg
 * where dd is the two-digit day of the month, mm is the two-
 * digit month, yyyy is a four-digit representation of the
 * current year, hh, mm, and ss are the current hour, minute,
 * and second values (each two-digits), and x is filesMade.
 * We must always remember to prefix the filename with the
 * user-specified directory.
 */
memset(fileName, 0, 1024);
sprintf(fileName, "%s%.2d%.2d%.2d_%.2d%.2d%.2d_%.2d.jpg", fileDir,
        currTime.wDay, currTime.wMonth, currTime.wYear,
        currTime.wHour, currTime.wMinute, currTime.wSecond,
        filesMade);
imageFile = CreateFile((LPCSTR)fileName, GENERIC_WRITE|GENERIC_READ,
        FILE_SHARE_READ|FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
        FILE_FLAG_SEQUENTIAL_SCAN, NULL);
if(imageFile == INVALID_HANDLE_VALUE)
{
        fprintf(stderr, "Could not create new image files: %s.\n",
                fileName);
        CloseHandle((PVOID)&serial);
        CloseHandle((PVOID)&log);
        free(comPort);
        free(buff);
        free(hwID);
        free(hwID2);
        free(fileDir);
        free(fileName);
        return MAIN_OUTPUT_FILE_FAIL;
}

/* Now, send the sender a message that says it's okay to begin
 * sending us data.  This message must be of the form:
 *      IMAGE_READY_OK
 * We'll try this ten times, then quit if we don't get an ACK.
 */
memset(buff, 0, 150);
sprintf(buff, "AT+UCAST:%s=%s", hwID2, "IMAGE_READY_OK");
for(i = 0; i < 10; i++)
{
        writeSerial(buff, &serial, strlen(buff), TRUE);
        fprintf(stdout, "\tSent IMAGE_READY_OK signal.\n");
        strncpy(hwID, "OK", 3);
        tBuff = getMessage(&serial, "OK", &log);
        memset(hwID, 0, 3);
        if(strstr(tBuff, "ERROR:"))
        {
                fprintf(stdout, "\tReceived NACK from %s.\n", hwID2);
                free(tBuff);
                if(i == 9)
                {
                        ++i;
                        break;
                }
        }
        else
        {
                fprintf(stdout, "\tGot ACK from %s.\n", hwID2);
                free(tBuff);
                break;
        }
}
```

```
                        if(i == 10)
                        {
                                CloseHandle((PVOID)&imageFile);
                                fprintf(stdout, "\r\n*****No response from %s, time out.\r\n",
                                        hwID2);
                                continue;
                        }

                        /* It's possible that a node will be disconnected from the
                         * network after sending us the IMAGE_READY signal and before
                         * sending us all of the image data.  To prevent the system
                         * from blocking forever, we establish a timer.  If we don't
                         * receive the first packet within 10 seconds of the IMAGE_READY
                         * signal, we consider the transmission void and write an error
                         * to the log file.  Each time we receive a packet, we reset the
                         * timer.  This effectively gives us a 10s timeout on each
                         * transmission.
                         */
                        GetLocalTime((LPSYSTEMTIME)&currTime);

                        /* We consider this the start of the image transfer. */
                        tSec = currTime.wMilliseconds + (currTime.wSecond * 1000) +
(currTime.wMinute * 60000);

                        /* Now, we need to get the image data.  We'll get exactly the
                         * number of packets specified in the IMAGE_READY signal.  Each
                         * packet can contain 150 bytes of data, though the ETRX2USB
                         * actually limits them to 65 bytes of data.
                         */
                        while(recPackets < numPackets)
                        {
                                /* Get the current time and see if it has been 10 seconds
                                 * since we last received a packet.  If so, write the error
                                 * to the log file and break.
                                 */
                                GetLocalTime((LPSYSTEMTIME)&timer);
                                if((timer.wSecond >= currTime.wSecond-50
                                                && timer.wMinute > currTime.wMinute)
                                        || ((timer.wSecond)-10 >= currTime.wSecond
                                                && timer.wMinute == currTime.wMinute))
                                {
                                        memset(buff, 0, 150);
                                        sprintf(buff, "\r\n*****TIMEOUT: %s (%s)\r\n",
                                                hwID2, fileName);
                                        WriteFile(log, (LPVOID)buff, strlen(buff), &bWritten,
NULL);
                                        fprintf(stdout, "\tImage %s from %s timed out!\n",
                                                fileName, hwID2);
                                        break;
                                }

                                /* If a byte has arrived, we must read the message, format it,
                                 * and write the useful part to the image file.  We then reset
                                 * the timer, increment the number of packets received, and
                                 * continue.
                                 */
                                memset(buff, 0, 150);
                                if((i = readMessage(&serial, buff)) > 0)
                                {
                                        i = formatMessage(tBuff, buff, hwID, 1, 150, 150, 17);
                                        fprintf(stdout, "Got packet %li of %li (%i bytes).\n",
                                                recPackets+1, numPackets, i);

                                        if(strcmp(hwID, hwID2) != 0)
                                        {
                                                fprintf(stdout, "Got packet from %s, expecting
%s.\n",
                                                        hwID, hwID2);
                                                continue;
                                        }
```

```
                                WriteFile(imageFile, (LPVOID)tBuff, i, &bWritten, NULL);
                                GetLocalTime((LPSYSTEMTIME)&currTime);
                                ++recPackets;
                                continue;
                        }
                }

                /* We stop our timer now, since we've finished the image transfer. */
                GetLocalTime((LPSYSTEMTIME)&currTime);
                tSec = ((currTime.wMilliseconds + (currTime.wSecond * 1000) +
                                (currTime.wMinute * 60000)) - tSec);

                /* We're finished with this file, so we close the handle, write
                 * a success message, and return to the beginning of the loop.
                 */
                CloseHandle((PVOID)&imageFile);
                /*memset(tmpName, 0, 1024);
                sprintf(tmpName, "%s.jpg", fileName);
                MoveFile(fileName, tmpName);*/
                fprintf(stdout, "Finished writing image (%li of %li packets)",
                        recPackets, numPackets);
                fprintf(stdout, ", %li milliseconds.\n", tSec);
                WriteFile(log, "\r\nFinished writing image.\r\n",
                        strlen("\r\nFinished writing image.\r\n"), &bWritten, NULL);
                continue;
        }
        else
        {
                free(tBuff);
                continue;
        }
    }

    /* Close the files. */
    CloseHandle((PVOID)&log);
    CloseHandle((PVOID)&serial);

    /* Free any used memory and exit. */
    free(comPort);
    free(buff);
    free(hwID);
    free(hwID2);
    free(fileDir);
    free(fileName);

    return MAIN_SUCCESS;
}
```

Appendix H: Camera Node Code

node.h:

```
MSP430 Camera Node
Copyright (C) 2007 Joseph Bosman, Matthew Conway, Phong Dam, Steve Olivieri,
       Ipek Ozil, Janelle Tavares, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include <msp430x16x.h>
#include <string.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include "mmc.h"

#ifndef NODE_H
#define NODE_H

/* How many pictures to take when the PIR is triggered */
const unsigned short int num_pic_taken = 2;

/* Image resolution, can be:
 *      0x03 : 160x128
 *      0x05 : 320x240
 *      0x07 : 640x480
 */
const unsigned char resolution = 0x05;

/* Interrupt flag for timer A */
unsigned short int timeflag_a = 0;

/* Interrupt flag for timer B */
unsigned short int timeflag_b = 0;

/* How many 1/10 seconds we've waited */
unsigned short int tsec = 0;

/* Camera command buffers */
unsigned char cmdSYNC[6];
unsigned char cmdACK[6];
unsigned char cmdReset[6];
unsigned char cmdPowerOff[6];
unsigned char cmdInitial[6];
unsigned char cmdSetPackageSize[6];
unsigned char cmdSetBaudRate[6];
unsigned char cmdSnapShot[6];
unsigned char cmdGetPic[6];
unsigned char cmdData[6];
unsigned char cmdEndPic[6];

/* Size of a camera command */
```

```
const unsigned short int cmdSize = 6;

/* Size of a double camera command */
const unsigned short int cmdACKsize = 12;

/* Size of an image data packet */
const unsigned short int packageSize = 262;

/* TODO: Phong */
const unsigned short int maxSendSYNC = 60;

/* Temporary buffer to store data packets from the camera */
unsigned char dummybuffer[262];

/* TODO: Phong */
unsigned short int dummyLen = 0;

/* TODO: Phong */
unsigned short int dummyRecv = 0;

/* Buffer for receiving data from the camera */
static unsigned char recvbuffer[262];

/* Which half of the mmc_buffer to store the packet in, can be:
 *      0 : first half (256 bytes)
 *      1 : second half (256 bytes)
 */
unsigned short int bufFlag = 0;

/* Camera flags (TODO: Phong) */
unsigned short int numberOfpackage;
unsigned short int lastPackDataSize;
unsigned short int currentLen = 0;
unsigned short int finishSYNCflag = 0;
unsigned short int prepSYNCsuccess = 0;
unsigned short int initialACKflag = 0;
unsigned short int setPackACKflag = 0;
unsigned short int snapACKflag = 0;
unsigned short int getPicACKflag = 0;
unsigned short int dataPackflag = 0;

/* Signals the MSP430 that data is ready. */
unsigned short int dataReady = 0;

/* Duration of the SD card initialization */
const unsigned short int timeout = 30;

/* Flag for SD initialization timeout */
unsigned short int timeoutFlag = 0;

/* How many blocks we've written to the SD card (resetting this variable
 * will cause us to overwrite any existing data)
 */
unsigned short int block = 0;

/* State of the SD card, can be:
 *      0 : not found
 *      1 : found
 */
extern unsigned char card_state;
unsigned char card_state = 0;

/* Buffer for holding data prior to transferring it to the SD card */
extern unsigned char mmc_buffer[512];

/* Size of each image */
unsigned int picSize[100];

/* Number of pictures taken */
unsigned short int numOfPic = 0;
```

```
/* Number of memory blocks read */
unsigned int numBlocks;

/* nHOST_INT interrupt boolean for the EM260 */
volatile unsigned char nHOST_INT;

/* The EM60's node ID */
unsigned char nodeID[2];

/* The sequence number for EM260 EZSP frames */
volatile unsigned char sq;

/* The sequence number for EM260 unicast messages */
volatile unsigned char sq2;

/* Interrupt functions */
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector=PORT1_VECTOR
    __interrupt void em260(void);
  #pragma vector=PORT2_VECTOR
    __interrupt void port_2(void);
  #pragma vector=TIMERA0_VECTOR
    __interrupt void Timer_A0(void);
  #pragma vector=TIMERB0_VECTOR
    __interrupt void Timer_B0(void);
  #pragma vector=UART0RX_VECTOR
    __interrupt void usart0_rx(void);
#else
  interrupt (PORT1_VECTOR) em260(void);
  interrupt (PORT2_VECTOR) port_2(void);
  interrupt (TIMERA0_VECTOR) Timer_A0(void);
  interrupt (TIMERB0_VECTOR) Timer_B0(void);
  interrupt (UART0RX_VECTOR) usart0_rx(void);
#endif

/* Starts running timer A. */
void runtimera(void);

/* Stops running timer A. */
void stoptimera(void);

/* Starts running timer B. */
void runtimerb(void);

/* Stops running timer B. */
void stoptimerb(void);

/* Initializes the MSP430. */
void init_sys(void);

/* Resets the camera variables and arrays. */
void reset_sys(void);

/* Initializes the SD card. */
void init_SD(void);

/* Enables SPI on USART0 of the MSP430. */
void enableSPI0(void);

/* Disables SPI on USART0 of the MSP430. */
void disableSPI0(void);

/* Enables UART on USART0 of the MSP430. */
void enableUART0(void);

/* Disables UART on USART0 of the MSP430. */
void disableUART0(void);

/* Waits for 1/10 of a second. */
```

```
void wait(void);

/* Waits for 1/100 of a second. */
void wait_a(void);

/* Waits for (s * 1/100) seconds.  For example, if s is equal to 10, this
 * function will wait for 0.10 seconds.
 *      @param s: the number of 1/100 seconds to wait
 */
void wait_a_sec(unsigned short int s);

/* Waits for (s * 1/10) seconds.  For example, if s is equal to 10, this
 * function will wait for 1.00 seconds.
 */
void waitsec(unsigned short int s);

/* Turns on the MSP430 LED. */
void LEDOn(void);

/* Turns off the MSP430 LED. */
void LEDOff(void);

/* Turns on LED 1. */
void LED1On(void);

/* Turns off LED 1. */
void LED1Off(void);

/* Turns on LED 2. */
void LED2On(void);

/* Turns off LED 2. */
void LED2Off(void);

/* Turns on LED 3. */
void LED3On(void);

/* Turns off LED 3. */
void LED3Off(void);

/* Turns on LED 4. */
void LED4On(void);

/* Turns off LED 4. */
void LED4Off(void);

/* Blinks the specified LED.
 *      @param i: the LED to blink
 */
void blinkLED(unsigned short int i);

/* Configures the camera command buffers. */
void config_camera_cmd(void);

/* Sends a command to the camera.
 *      @param c: the command to send
 */
void send_cmd(unsigned char c[]);

/* Receives an ACK command from the camera. */
void recv_cmd(void);

/* Sends a sequence of SYNC commands to the camera. */
void sendSYNC(void);

/* Checks to see if the command received from the camera is NACK. */
void check(void);

/* Initializes the camera system.  This involves the following tasks:
 *      - Sending the "initialize" command to the camera
```

```
 *      - Sending the "set package size" command to the camera
 */
void cam_prep(void);

/* Takes a picture with the camera.  This involves the following tasks:
 *      - Sending the "snap shot" command to the camera
 *      - Sending the "get picture" command to the camera
 */
void take_pic(void);

/* Gets an image from the camera. */
void get_image(void);

/* Gets a specific data packet from the camera.
 *      @param id: the data packet to get
 */
void get_package(unsigned char id);

/* Calculates the number of packets for the last image taken. */
void cal_numbOfpack(void);

/* Calculates the size of the last image packet. */
void cal_lastPackSize(void);

/* Transfers image data packets to the SD card. */
void transfer_pack(void);

/* Prepares the mmc_buffer for writing data to the SD card. */
void prep_buffer(void);

/* Prepares the mmc_buffer with the last packet (TODO: Phong). */
void prep_last_buffer(void);

/* Clears the mmc_buffer. */
void clear_mmc_buffer(void);

/* Converts a character to an integer */
unsigned short int hex2int(unsigned char c1 ,unsigned char c2);

/* Calculates how many packets are going to be sent to the base
 * station by getting the total numer of bytes and sends the
 * "IMAGE_READY" signal to the base station specifying number
 * of packets. After it receives an "IMAGE_READY_OK" from
 * the base station, it reads from the memory and sends 64
 * bytes at a time until a whole image is sent.
 *      @param imgNum: the number of the image taken
 */
void transmitPicture(unsigned int imgNum);

/* Initializes the EM260 by performing the following tasks:
 *      - resets the EM260
 *      - performs a reset transaction over SPI
 *      - performs a "verion" transaction over SPI
 *      - performs a "SPI status" transaction over SPI
 */
unsigned int initEM260(void);

/* Sends a string to the EM260 over SPI1.
 *      @param buff: a pointer to the string to send
 *      @param len: the length of the string
 */
void toEM260(unsigned char* buff, unsigned int len);

/* Receives data from the EM260 over SPI1.
 *      @param buff: a pointer to the buffer to store the response in
 *      @return: the number of bytes received (and stored in buff)
 */
unsigned int fromEM260(unsigned char* buff);
```

```
/* Sets the configuration values for the EM260.
 *      @param params: the configuration to set and its new value
 *      @return: result of the command
 */
unsigned char EM260_setConfigurationValue(unsigned char* params);

/* Issues a callback() command to the EM260 and receives a response.
 * Typically, callbacks are made after calling other commands.  In the
 * case that the EM260 must send us a message asynchronously, the
 * nHOST_INT interrupt will be triggered and a callback must be made.
 *      @param buff: pointer to the buffer to store the respnse in
 */
void EM260_callback(unsigned char* buff);

/* Calls the EZSP networkInit() function.  The result of calling
 * this function depends on whether or not the EM260 was joined
 * to a network on last power off.  If so, a callbac must follow
 * this command.  In addition, the EM260 will once again be on
 * that network.  If not, no callback should be issued and the
 * EM260 will not be joined to any networks.
 *      @return: the network status
 */
void EM260_networkInit();

/* Calls the EZSP leaveNetwork() function.  The result of this
 * call is the return value for EM260_leaveNetwork().  If this
 * call is successful, the EM260 will leave any network that it
 * is joined to.  Else, the state of the network is unknown.
 *      @return: the status of the command
 */
unsigned char EM260_leaveNetwork();

/* Calls the EZSP networkState() function. The result of this
 * call is the return value for EM260_networkState() which
 * tells the state the network is in.
 *      @return: status of the network
 */
unsigned char EM260_networkState ();

/* Calls the EZSP startScan() function. The result of this
 * call is the return value for EM260_scanNetwork(). If this
 * call is successful, there will be a callback to the function
 * that either says scan is complete or network found.
 *      @return: result of the command
 */
unsigned char EM260_scanNetwork ();

/* Calls the EZSP joinNetwork() function. The result of this
 * call is the return value for EM260_justJoinNetwork(). If
 * this call is successful, EM260 will join the found network.
 * There will also be callback that says network is up.
 *      @param params: parameters from the callback of
 *      scanNetwork() that include the PAN ID, short PAN ID
 *      and the channel.
 *      @retur: result of the command
 */
unsigned char EM260_justJoinNetwork(unsigned char* params);

/* Calls the EZSP getNodeID() function. The result of this
 * call is the return values for EM260_getNodeID(). If this
 * call is successful, it will get the Node ID of the device.
 *      @param: pointer to a buffer to store the result of
 *      the call.
 */

void EM260_getNodeID(unsigned char* buff);

/* Sends a unicast message to another node on the network via the
 * EZSP sendUnicast() function.  If this function is successful,
 * a callback must follow.  This function always uses ProfileId 0,
```

```
 * ClusterId 0, and EndPoint 0x01 (source and destination).  Messages
 * are always tagged with 0x64, route discovery is enabled, retries
 * are enabled, and the source EUI64 is always sent before the message.
 *      @param addr: the two-byte NodeID to send the message to
 *      @param message: the message to send
 *      @param len: the length of message
 */

unsigned char EM260_sendUnicast(unsigned char* message, unsigned char* nodeID,
       unsigned char len);

/* Calls the EM260_networkState(), EM260_leaveNetwork(), EM260_networkScan()
 * and EM260_justJoinNetwork() functions. First, gets the state of the network
 * by calling EM260_networkState(). If the result says that the device is not
 * a part of a network, it calls EM260_scanNetwork()to find any available networks
 * and then calls EM260_justJoinNetwork(). If the result of EM260_networkState()
 * is joined to a network, it first calls EM260_leaveNetwork() to leave the
 * current network and then goes through the joinin procedure.
 */
void joinNetwork();

#endif /* NODE_H */
```

node.c:

```
MSP430 Camera Node
Copyright (C) 2007 Joseph Bosman, Matthew Conway, Phong Dam, Steve Olivieri,
       Ipek Ozil, Janelle Tavares, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include "node.h"

/************* Timer A ISR *****************/
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector = TIMERA0_VECTOR
  __interrupt void Timer_A0(void)
#else
  interrupt (TIMERA0_VECTOR) Timer_A0(void)
#endif
{
       timeflag_a = 1;
}

/************* Timer B ISR *****************/
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector = TIMERB0_VECTOR
  __interrupt void Timer_B0(void)
#else
  interrupt (TIMERB0_VECTOR) Timer_B0(void)
#endif
{
       timeoutFlag++;
       timeflag_b = 1;
}

/************* Port 2 ISR *****************/
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector = PORT2_VECTOR
  __interrupt void port_2(void)
#else
  interrupt (PORT2_VECTOR) port_2(void)
#endif
{
       _BIC_SR_IRQ(LPM1_bits); // Exit LPM1
       P2IFG = 0;      // clear Port 2 interrupt flag
}

/************* UART0 RX ISR *****************/
#ifdef __IAR_SYSTEMS_ICC__
  #pragma vector = UART0RX_VECTOR      // uart0 receive interrupt
  __interrupt void usart0_rx (void)
#else
  interrupt (UART0RX_VECTOR) usart0_rx(void)
#endif
{
       recvbuffer[currentLen++] = RXBUF0;
       if (dataReady == 1)
              dummybuffer[dummyLen++] = recvbuffer[dummyRecv++];
```

```
}

#ifdef __IAR_SYSTEMS_ICC__
 #pragma vector=PORT1_VECTOR
 __interrupt void em260 (void)
#else
 interrupt (PORT1_VECTOR) em260(void)
#endif
{
       nHOST_INT = 1;
       P1IFG &= ~0x08;
}


/******************* main () *************************
*  The system will first enter sleep mode. As soon as the
*  sensor triggers, the system will be activated and start
*  taking and storing 2 pictures onto SD card. It will
*  then go back to sleep mode. The routine repeats.
*****************************************************/
void main( void )
{
  /* Command buffer for the EM260 */
  unsigned char buff[3];

  /* Number of pictures taken */
  unsigned char pic_counter;

  /* Staitus for EM260 functions */
  unsigned char status;

  /* Loop variable */
  unsigned int i;

  sq = 0x01;
  sq2 = 0x00;

  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  _BIS_SR(GIE);  // Global interrupt enable
  P6OUT |= 0x04;
  init_sys();
  config_camera_cmd();
  LEDOn();
  init_SD();           // initialize SD card
  //waitsec(100);      // wait for sensor stablize

  /* Initialize the EM260 */
  enableSPI0();
  status = 0xFF;
  while(status != 0x00)
  {
       wait_a();
       status = initEM260();
  }
  blinkLED(1);

  /* Turn off ZigBee security. */
  buff[0] = 0x0D;
  buff[1] = 0x00;
  buff[2] = 0x00;
  do
  {
       wait_a();
       status = EM260_setConfigurationValue(buff);
  } while(status != 0x00);

  /* Enable message relaying. */
  buff[0] = 0x18;
  buff[1] = 0x00;
```

```
buff[2] = 0x00;
do
{
      wait_a();
      status = EM260_setConfigurationValue(buff);
} while(status != 0x00);

blinkLED(2);

/* Join a ZigBee network. */
EM260_networkInit();
blinkLED(3);

joinNetwork();
blinkLED(4);

/* Disable SPI and enable UART so that the camera can begin functioning. */
disableSPI0();
enableUART0();

/* From this point on, we wait for input from the PIR sensor.  If we get it,
 * we take pictures and send them over the ZigBee network.  Forever!
 */
while (1)
{
      /* First, we put the MSP430 into sleep mode.  We remain in sleep mode
       * unless the PIR sensor wakes us up.
       */
      pic_counter = 0; // keep track of how many pictures taken
      LEDOff();
      reset_sys();

      #ifdef __IAR_SYSTEMS_ICC__
             _BIS_SR(LPM1);   // LPM1 sleep mode enabled: MCLK and DCO are disabled
      #else                   // SMCLK and ACLK are active
             LPM1;
      #endif
      blinkLED(0);    //  indicate system is activated by sensor
      cam_prep();      // initialize camera by sending multiple SYNC commands
      blinkLED(1);    //  indicate camera is succesfully initialized

      /* Then, we take pictures. */
      while (pic_counter < num_pic_taken)
      {
             take_pic();      // tell camera to take snapshot
             get_image();    // get image data from camera

             reset_sys();
             pic_counter++;
             blinkLED(3);   // done taking a picture
             waitsec(3);
      }
      send_cmd(cmdReset);
      send_cmd(cmdPowerOff);  // turn off camera
      blinkLED(4);          // done taking user-specified number of pictures

       /* Finally, we transmit our pictures over the ZigBee network. */
      disableUART0();
      enableSPI0();
      for(i = 0; i < pic_counter; i++)
      {
             transmitPicture(i);
             waitsec(10);
      }
      disableSPI0();
      enableUART0();

      /* Now we reset the number of pictures taken and point block back to
       * block 0, as we no longer need the data on the SD card.
       */
```

```
        numOfPic = 0;
        block = 0;
        numBlocks = 0;
  }
}


/****************** init_sys() ************************/
void init_sys(void)
{
  volatile unsigned char dco_cnt = 255;
  BCSCTL1 &= ~XT2OFF;                           // XT2on
  do                                                 // wait for MCLK from quartz
  {
        IFG1 &= ~OFIFG;                                      // Clear OSCFault flag
        for (dco_cnt = 0xff; dco_cnt > 0; dco_cnt--); // Time for flag to set
  }
  while ((IFG1 & OFIFG) != 0);                       // OSCFault flag still set?
  BCSCTL2 |= SELM_2 + SELS + DIVS_1;                // MCLK = XT2  SMCLK= MCLK / 2 = 4MHz

  P6DIR |= BIT0 + BIT2 + BIT4 + BIT5 + BIT6 + BIT7;         // Set P6.0,  P6.4 - 6.7, to output
direction
  P6SEL &= ~(BIT0 + BIT2 + BIT4 + BIT5 + BIT6 + BIT7);      // Make P6.0, P.4 - 7, I/O option

  P2DIR &= ~BIT0;        // Set P2.0 to input direction
  P2SEL &= ~BIT0;        // Make P2.0 I/O option
  P2IES = 0x00;                 // Trigger mode: Low to high transition
  P2IE = 0x01; // Enable Port 2 interrupt

  /* P4.0 (nRESET) and P4.1 (nWAKE) are OUT. */
  P4DIR |= 0x03;
  P4OUT |= 0x03;

  /* P1.3 (nHOST_INT) is a H->L interrupt and IN. */
  P1DIR &= ~0x08;
  P1SEL &= ~0x08;
  P1IES |= 0x08;
  P1IE |= 0x08;

  /* Turn off the LEDs */
  LED1Off();
  LED2Off();
  LED3Off();
  LED4Off();
  //P6OUT |= 0x04;
}

/****************** init_SD () ************************/
void init_SD (void)
{
  int i;
  unsigned short int success = 0;

  while (!success)
  {
        runtimerb();
        while (timeoutFlag < timeout)
        {
              if (initMMC() == MMC_SUCCESS)   // card found
              {
       card_state |= 1;
       success = 1;
              }
              else
              {
       //Error card not detected or rejected during writing
       card_state = 0;                                  // no card
              }
        }
        stoptimerb();
```

```
        timeoutFlag = 0;
        if (card_state == 0)
              for (i = 0; i < 3; i++)
                    blinkLED(0);
  }
}

/****************** runtimera() ***********************/
void runtimera(void)
{

  TACTL = TASSEL_2 + CNTL_0 + MC_1 + ID_3; // SMCLK, 16bit, up mode, 8 divider
  TACCR0 = 0X1388;  // 5000 SMCLK tics = 0.01 second
  TACCTL0 = CCIE;      // TACCR0 interrupt enabled

  /*
  TACTL = TASSEL_1 + CNTL_0 + MC_1 + ID_0; // ACLK, 16bit, up mode, 1 divider
  TACCR0 = 0x8000;  // 32768 ACLK tics = 1 second
  TACCTL0 = CCIE;      // TACCR0 interrupt enabled
  */
}

/****************** stoptimera() ***********************/
void stoptimera(void)
{
  TACTL = MC_0; // stop timer
  TACCTL0 &= ~CCIE;  // TACCR0 interrupt disabled
}

/****************** runtimerb() ***********************/
void runtimerb(void)
{
  TBCTL = TBSSEL_2 + CNTL_0 + MC_1 + ID_3; // SMCLK, 16bit, up mode, 8 divider
  TBCCR0 = 0xC350;  // 50000 SMCLK tics = 0.1 second
  TBCCTL0 = CCIE;      // TBCCR0 interrupt enabled
}

/****************** stoptimerb() ***********************/
void stoptimerb(void)
{
  TBCTL = MC_0; // stop timer
  TBCCTL0 &= ~CCIE;  // TBCCR0 interrupt disabled
}

/****************** enableUART0() ********************/
void enableUART0(void)
{
        P3SEL |= 0x30;                // P3.4 , P3.5 = USART0 option select
        U0CTL |= SWRST;
        ME1 |= UTXE0 + URXE0;   // Enable USART0 TXD/RXD
        UCTL0 |= CHAR;                // 8-bit character , NO parity, 1 stop bit
        UCTL0 &= ~(SYNC + MM);
        UTCTL0 |= SSEL1;        // UCLK = SMCLK
        UBR00 = 0x68;           // 4Mhz/38400 ~ 104
        UBR10 = 0x00;           //
        UMCTL0 = 0x4;           // modulation
        UCTL0 &= ~SWRST;        // Initialize USART state machine
        IE1 |= URXIE0;                // Enable USART0 RX interru
}
/****************** enableUART0() ********************/
void disableUART0(void)
{
        ME1 &= ~(UTXE0 + URXE0);       // Disable USART0 TXD/RXD
}

/***************** enableSPI0() *********************/
void enableSPI0(void)
{
       /* Use pins 3.1-3.3 for USART1. */
       P3SEL |= 0x0E;
```

```
        /* P3.1 (MOSI) and P3.3 (SCLK) must be OUT, P3.2 (MISO) must be IN */
        P3DIR |= 0x0A;
        P3DIR &= ~0x04;

        /* Put USART0 into a reset state. */
        U0CTL |= SWRST;

        /* Enable SPI. */
        ME1 |= USPIE0;

        /* Set USART parameters to:
         *       8-bit character
         *       1 stop bit
         *       no parity
         *       SPI mode
         *       Master mode
         */
        U0CTL |= CHAR + SYNC + MM;

        /* BRCLK = SMCLK, 3-pin mode, phase shifting */
        U0TCTL |= SSEL1 + STC + CKPH;

        /* Set the baud rate to 500 Kbps. */
        UBR00 = 0x08;
        UBR10 = 0x00;

        /* Turn off modulation. */
        U0MCTL = 0x00;

        /* Set P3.0 to be OUT (nSSEL). */
        P3DIR |= 0x01;
        P3OUT |= 0x01;

        /* Initialize USART0 state machine. */
        U0CTL &= ~SWRST;

        /* Wait until the buffer is clear. */
        while(!(IFG1 & UTXIFG0));
}

void disableSPI0(void)
{
        /* Disables SPI on USART0 */
        ME1 &= ~USPIE0;
}

/****************** LEDOn() ************************/
void LEDOn(void)
{
         P6OUT &= ~BIT0; // P6.0 output = 0 (LED off)
}

/****************** LEDOff() ************************/
void LEDOff(void)
{
         P6OUT |= BIT0;  // P6.0 output = 1 (LED off)
}

/****************** LED1On() ************************/
void LED1On(void)
{
         P6OUT &= ~BIT6; // P6.6 output = 0 (LED off)
}

/****************** LED1Off() ************************/
void LED1Off(void)
{
         P6OUT |= BIT6;  // P6.6 output = 1 (LED off)
}
```

```
/****************** LED2On() ***********************/
void LED2On(void)
{
        P6OUT &= ~BIT7; // P6.7 output = 0 (LED off)
}

/****************** LED2Off() ***********************/
void LED2Off(void)
{
        P6OUT |= BIT7;  // P6.7 output = 1 (LED off)
}
/****************** LED3On() ***********************/
void LED3On(void)
{
        P6OUT &= ~BIT4; // P6.4 output = 0 (LED off)
}

/****************** LED3Off() ***********************/
void LED3Off(void)
{
        P6OUT |= BIT4;  // P6.4 output = 1 (LED off)
}

/****************** LED4On() ***********************/
void LED4On(void)
{
        P6OUT &= ~BIT5; // P6.5 output = 0 (LED off)
}

/****************** LED4Off() ***********************/
void LED4Off(void)
{
        P6OUT |= BIT5;  // P6.5 output = 1 (LED off)
}

/****************** wait_a() ********************/
void wait_a(void)
{
  runtimera();
  while (!timeflag_a);
  timeflag_a = 0;
  stoptimera();
}

/****************** wait() ********************/
void wait(void)
{
  runtimerb();
  while (!timeflag_b);
  timeflag_b = 0;
  stoptimerb();
}
/****************** wait_a_sec () ********************/
void wait_a_sec (unsigned short int s)
{
  int i;
  for (i = 0; i < s; i++)
        wait_a();
}
/****************** waitsec () ********************/
void waitsec (unsigned short int s)
{
  int i;
  for (i = 0; i < s; i++)
        wait();
}

/****************** config_camera_cmd () *********************

  * 2nd element of an array is the command's ID.
```

```
  * 3rd, 4th, 5th, and 6th elements of an array are
               Parameter 1, 2, 3, and 4 respectively.

************************************************************/
void config_camera_cmd(void)
{

  cmdSYNC[0] = 0xAA;
  cmdSYNC[1] = 0x0D;
  cmdSYNC[2] = 0x00;
  cmdSYNC[3] = 0x00;
  cmdSYNC[4] = 0x00;
  cmdSYNC[5] = 0x00;

  cmdACK[0] = 0xAA;
  cmdACK[1] = 0x0E;
  cmdACK[2] = 0x0D;
  cmdACK[3] = 0x00;
  cmdACK[4] = 0x00;
  cmdACK[5] = 0x00;

  cmdReset[0] = 0xAA;
  cmdReset[1] = 0x08;
  cmdReset[2] = 0x00;
  cmdReset[3] = 0x00;
  cmdReset[4] = 0x00;
  cmdReset[5] = 0xFF;

  cmdPowerOff[0] = 0xAA;
  cmdPowerOff[1] = 0x09;
  cmdPowerOff[2] = 0x00;
  cmdPowerOff[3] = 0x00;
  cmdPowerOff[4] = 0x00;
  cmdPowerOff[5] = 0x00;

  cmdInitial[0] = 0xAA;
  cmdInitial[1] = 0x01;
  cmdInitial[2] = 0x00;
  cmdInitial[3] = 0x07;       // JPEG file
  cmdInitial[4] = 0x00;
  cmdInitial[5] = resolution;


  // package size is 262 bytes = 0x0106h
  cmdSetPackageSize[0] = 0xAA;
  cmdSetPackageSize[1] = 0x06;
  cmdSetPackageSize[2] = 0x08;
  cmdSetPackageSize[3] = 0x06;        // Low Byte size
  cmdSetPackageSize[4] = 0x01;        // High Byte size
  cmdSetPackageSize[5] = 0x00;

  // the baud rate = 38400 bps
  cmdSetBaudRate[0] = 0xAA;
  cmdSetBaudRate[1] = 0x07;
  cmdSetBaudRate[2] = 0x2F;
  cmdSetBaudRate[3] = 0x01;
  cmdSetBaudRate[4] = 0x00;
  cmdSetBaudRate[5] = 0x00;

  cmdSnapShot[0] = 0xAA;
  cmdSnapShot[1] = 0x05;
  cmdSnapShot[2] = 0x00;
  cmdSnapShot[3] = 0x00;
  cmdSnapShot[4] = 0x00;
  cmdSnapShot[5] = 0x00;

  cmdGetPic[0] = 0xAA;
  cmdGetPic[1] = 0x04;
  cmdGetPic[2] = 0x01;  // snap mode
  cmdGetPic[3] = 0x00;
```

```
  cmdGetPic[4] = 0x00;
  cmdGetPic[5] = 0x00;

  cmdEndPic[0] = 0xAA;
  cmdEndPic[1] = 0x0E;
  cmdEndPic[2] = 0x00;
  cmdEndPic[3] = 0x00;
  cmdEndPic[4] = 0xF0;
  cmdEndPic[5] = 0xF0;
}

/****************** send_cmd() ***********************/
void send_cmd(unsigned char c[])
{
  int i;
  // send a command
  for(i = 0; i < cmdSize; i++)
  {
         while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
         TXBUF0 = c[i];
  }
}

/********************** sendSYNC () ********************/
void sendSYNC(void)
{
  // This loop will keep sending SYNC command to the camera
  // until the camera sends back ACK and SYNC command.
        while (!finishSYNCflag)
        {
               int i = 0;
               i++;
               if (i > maxSendSYNC)
               {
                        send_cmd(cmdReset);
                        send_cmd(cmdPowerOff);
               }
               send_cmd(cmdSYNC);
               wait_a_sec(5);
               if (currentLen == cmdACKsize) // receive complete message
               {
                      currentLen = 0;
                      finishSYNCflag = 1;
               }
        }
}

/********************** cam_prep () ********************/
void cam_prep (void)
{

       while(!prepSYNCsuccess)
       {
         sendSYNC();
         wait_a_sec(5);
         // The system sends back ACK command to camera
         send_cmd(cmdACK);

         wait();
         check();
       }
        // send initial command and wait for ACK
        send_cmd(cmdInitial);
        while (!initialACKflag)
        {
               if (currentLen == cmdSize)
               {
                      currentLen = 0;
                      initialACKflag = 1;
               }
```

```
        }
        wait_a_sec(5);

        // send set package size command and wait for ACK
        send_cmd(cmdSetPackageSize);
        while (!setPackACKflag)
        {
               if (currentLen == cmdSize)
               {
                      currentLen = 0;
                      setPackACKflag = 1;
               }
        }
         wait_a_sec(5);
         blinkLED(4);
}

/********************** take_pic () ********************/
void take_pic(void)
{
        // send snapshot command and wait for ACK
        send_cmd(cmdSnapShot);
        while (!snapACKflag)
        {
               if (currentLen == cmdSize)
               {
        currentLen = 0;
        snapACKflag = 1;
               }
        }
        wait_a_sec(5);

        // send get picture command and wait for ACK
        send_cmd(cmdGetPic);
        while (!getPicACKflag)
        {
               if (currentLen == cmdACKsize)
               {
        currentLen = 0;
        getPicACKflag = 1;
               }
        }
        cal_numbOfpack();
}

/********************** get_image () ********************/
void get_image(void)
{

  dataReady = 1;       // start receiving data and put them into dummybuffer[]
  currentLen = 0;

  unsigned short int packID = 0x00;
  // get each package from camera and dump it onto SD
  while (packID < numberOfpackage)
  {
        get_package(packID);
        wait();
        if (packID != numberOfpackage - 1)  // if packID is not the last packet
        {
              prep_buffer();
        }
        else
        {
              prep_last_buffer();
        }
        packID++;
        currentLen = 0;
        dummyLen = 0;
        dummyRecv = 0;
```

```
  }
  wait();
  send_cmd(cmdEndPic);
  dataReady = 0;        // stop storing into dummybuffer[]
}

/********************** get_package () ********************/
void get_package(unsigned char id)
{
  int i;
  unsigned char cmd[6];
  cmd[0] = 0xAA;
  cmd[1] = 0x0E;
  cmd[2] = 0X0A;
  cmd[3] = 0X00;
  cmd[4] = id;
  cmd[5] = 0x00;

  // send a command
  for(i = 0; i < cmdSize; i++)
  {
         while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
         TXBUF0 = cmd[i];
  }
}

/*************** prep_buffer() *************/
void prep_buffer(void)
{
  int i;
  if (bufFlag == 0)
  {
        clear_mmc_buffer();
        for (i = 4; i<260 ; i++)
              mmc_buffer[i-4] = dummybuffer[i];
        bufFlag = 1;
  }
  else
  {
        for (i = 4 ; i < 260; i++)
              mmc_buffer[i+252] = dummybuffer[i];
        transfer_pack();
        blinkLED(2);
        bufFlag = 0;
  }
}

/*************** prep_last_buffer() *************/
void prep_last_buffer(void)
{
  int i;
  cal_lastPackSize();    // get last package size
  if (bufFlag == 0)
  {
        clear_mmc_buffer();
        for (i = 4; i < (lastPackDataSize + 4); i++)
              mmc_buffer[i-4] = dummybuffer[i];
        transfer_pack();
        blinkLED(2);
  }
  else
  {
        for (i = 4 ; i < (lastPackDataSize + 4); i++)
              mmc_buffer[i+252] = dummybuffer[i];
        transfer_pack();
        blinkLED(2);
  }
}
/*************** transfer_pack() *************/
void transfer_pack(void)
```

```
{
        wait_a();
        mmcWriteBlock(512*(block++));        // write to SD card
        wait_a();
}

/*************** clear_mmc_buffer() *************/
void clear_mmc_buffer(void)
{
  int i;
  for (i = 0 ; i < 512 ; i++)
        mmc_buffer[i] = 0x00;
}

/*************** check() ******************/
void check(void)
{
  if (recvbuffer[1] == 0x0F || recvbuffer[7] == 0x0F)
  {
        send_cmd(cmdReset);
        send_cmd(cmdPowerOff);
  }
  else prepSYNCsuccess = 1;
}

/*************** cal_numbOfpack() ******************/
void cal_numbOfpack(void)
{
  unsigned short int totalBytes  = 0 ;
  totalBytes = hex2int (recvbuffer[9], recvbuffer[10]);
  picSize[numOfPic++] = totalBytes;
  numberOfpackage = (totalBytes / (packageSize - 6)) + 1;
}

/*************** cal_lastPackSize() *************/
void cal_lastPackSize(void)
{
  lastPackDataSize = hex2int (recvbuffer[2], recvbuffer[3]);
}

/*************** hex2int() **********************/
unsigned short int hex2int(unsigned char c1, unsigned char c2)
{

  unsigned short int i1,i2,i3,i4;
//  unsigned short int i5,i6;
  unsigned short int total  = 0 ;

  i1 = (c1 & 0x0F);
  i2 = ((c1 & 0xF0) >> 4) * 16;
  i3 = (c2 & 0X0F) * 256;
  i4 = ((c2 & 0xF0) >> 4) * 4096;

//  i5 = (c3 & 0X0F) * 65536;
//  i6 = ((c3 & 0xF0) >> 4) * 1048576;
//  totalBytes = i1+i2+i3+i4+i5+i6;
  total = i1+i2+i3+i4;
  return total;
}
/*************** blinkLED() ******************/
void blinkLED(unsigned short int i)
{
  switch (i)
  {
  case 0:
        LEDOn();
        waitsec(2);
        LEDOff();
        waitsec(2);
        break;
```

```
  case 1:
        LED1On();
        waitsec(2);
        LED1Off();
        waitsec(2);
        break;
  case 2:
        LED2On();
        waitsec(2);
        LED2Off();
        waitsec(2);
        break;
  case 3:
        LED3On();
        waitsec(2);
        LED3Off();
        waitsec(2);
        break;
  case 4:
        LED4On();
        waitsec(2);
        LED4Off();
        waitsec(2);
        break;
  default:
        break;
  }
}

/*************** reset_sys() ******************/
void reset_sys(void)
{
  int i;

  timeflag_a = 0;
  timeflag_b = 0;
  tsec = 0;
  dataReady = 0;
  bufFlag = 0;
  timeoutFlag = 0;
  dummyLen = 0;
  dummyRecv = 0;

  finishSYNCflag = 0;
  prepSYNCsuccess = 0;
  initialACKflag = 0;
  setPackACKflag = 0;
  snapACKflag = 0;
  getPicACKflag = 0;
  dataPackflag = 0;
  currentLen = 0;
  numberOfpackage = 0;
  lastPackDataSize = 0;

  for (i = 0; i < 262; i++) {
       dummybuffer[i] = 0x00;
       recvbuffer[i] = 0x00;
  }

  for (i = 0; i < 512; i++)
        mmc_buffer[i] = 0x00;
}

unsigned int initEM260(void)
{
       /* Buffer for storing commands and responses */
       unsigned char buff[10];

       /* Reset the EM260 (P4.0 = 0). */
       P4OUT &= ~0x01;
```

```
/* Wait for a while. */
waitsec(10);

/* Release the reset pin. */
P4OUT |= 0x01;

/* Wait for EM260 to assert nHOST_INT */
while(!nHOST_INT);
nHOST_INT = 0x00;

/* Now, we must transmit a specific series of commands to finish
 * initializing the EM260.            The first command is the "version"
 * command.      The first step is to load the "version" command (0x0A)
 * and the frame terminator (0xA7) into the buffer.
 */
buff[0] = 0x0A;
buff[1] = 0xA7;

/* Assert nSSEL. */
P3OUT &= ~0x01;

/* Send the command to the EM260. */
toEM260(buff, 2);

/* Clear enough buffer space to receive the response. */
memset(buff, 0, 3);

/* Now, receive the response. */
fromEM260(buff);

/* Deassert the nSSEL signal. */
P3OUT |= 0x01;

/* If the first byte is not 0x00 and the second byte is not 0x02,
 * there's a problem.
 */
if(buff[0] != 0x00 || buff[1] != 0x02)
        return 1;

/* We must wait at least 1ms between commands. */
wait_a();

/* We must again transmit the "version" command. */
buff[0] = 0x0A;
buff[1] = 0xA7;
P3OUT &= ~0x01;
toEM260(buff, 2);
memset(buff, 0, 3);
fromEM260(buff);
P3OUT |= 0x01;

/* If the first byte of our response is not 0x82, there is
 * a problem.
 */
if(buff[0] != 0x82)
        return 2;

/* Again, we must delay for at least 1ms between commands. */
wait_a();

/* Now, transmit the "SPI status" command. */
buff[0] = 0x0B;
buff[1] = 0xA7;
P3OUT &= ~0x01;
toEM260(buff, 2);
memset(buff, 0, 3);
fromEM260(buff);
P3OUT |= 0x01;
```

```
        /* If the first byte of the response is not 0xC1, there is
         * a problem.
         */
        if(buff[0] != 0xC1)
                return 3;

        /* Wait for at least 1ms. */
        wait_a();

        /* Finally, we must send the EZSP "version" command. */
        buff[0] = 0xFE;
        buff[1] = 0x04;
        buff[2] = 0x00;
        buff[3] = 0x00;
        buff[4] = 0x00;
        buff[5] = 0x02;
        buff[6] = 0xA7;
        P3OUT &= ~0x01;
        toEM260(buff, 7);
        memset(buff, 0, 10);
        fromEM260(buff);
        P3OUT |= 0x01;

        /* If bytes 5 and 6 are not each 0x02, something is wrong. */
        if(buff[5] != 0x02 || buff[6] != 0x02)
                return 4;

        /* Return successfully. */
        return 0;
}

void toEM260(unsigned char* buff, unsigned int len)
{
        /* Temporary receive buffer. */
        unsigned char tmp;

        /* Loop variable */
        unsigned int i;

        for(i = 0 ; i < len; i++)
        {
                /* Wait until the Tx buffer is empty. */
                while(!(IFG1 & UTXIFG0));

                /* Transmit the next byte in the buffer. */
                U0TXBUF = *(buff+i);

                /* Receive the idle response. */
                while(!(IFG1 & URXIFG0));
                tmp = U0RXBUF;
        }
}

unsigned int fromEM260(unsigned char* buff)
{
        /* Whether we have an EZSP frame or not */
        unsigned char ezsp;

        /* Length of the message we're receiving */
        unsigned char len;

        /* Loop variable */
        unsigned char i;

        /* We initialize i to 2 to account for the
         * first two bytes that we receive
         */
        i = 0x02;
        ezsp = 0;
        len = 0;
```

```
	/* Transmit 0xFF and receive 0xFF to clock the SPI.  Do so until
	 * we receive a byte that is not 0xFF.  This byte is the first
	 * byte of the response.
	 */
	*buff = 0xFF;
	while(*buff == 0xFF)
	{
		while(!(IFG1 & UTXIFG0));
		U0TXBUF = 0xFF;

		while(!(IFG1 & URXIFG0));
		*buff = U0RXBUF;
	}

	/* If the first bytes is 0xFE, we're receiving an EZSP frame. */
	if (*buff == 0xFE)
	{
		ezsp = 1;
	}

	/* Again, transmit 0xFF and receive the next byte of the
	 * message.
	 */
	while(!(IFG1 & UTXIFG0));
	U0TXBUF = 0xFF;
	while(!(IFG1 & URXIFG0));
	*(buff+1) = U0RXBUF;

	/* Some messages are only two bytes.  If the second byte is
	 * 0xA7 (the frame terminator), then we have such a message
	 * and there is no need to continue.
	 */
	if(*(buff+1) == 0xA7)
	{
		return i;
	}

	/* If we're receiving an EZSP frame, the length of the message
	 * is the second byte that we received.  Otherwise, the length
	 * of the message must be three bytes.  We always add three to
	 * the length of the message to account for the initial byte,
	 * the second byte, and the frame terminator.
	 */
	len = ezsp ? *(buff+1) : 0;
	len += 3;

	/* Now, transmit 0xFF to clock the SPI and receive the next
	 * byte of the response.      Do so until we have the entire
	 * response.
	 */
	do
	{
		/* Transmit 0xFF to the EM260. */
		while(!(IFG1 & UTXIFG0));
		U0TXBUF = 0xFF;

		/* Receive a response. */
		while(!(IFG1 & URXIFG0));
		*(buff+i) = U0RXBUF;

		++i;
	} while(i < len);

	/* Return the number of bytes that we received. */
	return i;
}

unsigned char EM260_setConfigurationValue(unsigned char* params)
{
```

```
        /* Command buffer */
        unsigned char buff[9];

        /* Set up the command buffer. */
        buff[0] = 0xFE;              /* EZSP frame */
        buff[1] = 0x06;              /* Length of the EZSP frame */
        buff[2] = sq++;              /* EZSP: sequence number */
        buff[3] = 0x00;              /* EZSP: command with no sleeping */
        buff[4] = 0x53;              /* EZSP: setConfigurationValue() command */
        buff[5] = params[0];   /* EZSP: Param: EZSP ConfigID */
        buff[6] = params[1];   /* EZSP: Param: new value (LSB) */
        buff[7] = params[2];   /* EZSP: Param: new value (MSB) */
        buff[8] = 0xA7;              /* Frame terminator */

        /* Transmit the command and receive a response. */
        P3OUT &= ~0x01;
        toEM260(buff, 9);
        memset(buff, 0, 9);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

void EM260_callback(unsigned char* buff)
{
        /* Set up the command buffer */
        buff[0] = 0xFE;              /* EZSP frame */
        buff[1] = 0x03;              /* Length of EZSP frame */
        buff[2] = sq++;              /* EZSP: sequence number */
        buff[3] = 0x00;              /* EZSP: command with no sleeping */
        buff[4] = 0x06;              /* EZSP: callback() command */
        buff[5] = 0xA7;              /* Frame terminator */

        /* Send the command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 6);
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;
}

void EM260_networkInit()
{
        /* Command buffer */
        unsigned char buff[7];

        /*Set up command buffer */
        buff[0] = 0xFE;              /* EZSP frame */
        buff[1] = 0x03;              /* Length of the EZSP frame */
        buff[2] = sq++;              /* EZSP: sequence number */
        buff[3] = 0x00;              /* EZSP: command with no sleeping */
        buff[4] = 0x17;              /* EZSP: networkInit() command */
        buff[5] = 0xA7;              /* Frame terminator */

        /* Send a command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 6);
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0x00;
        P3OUT |= 0x01;

        /* If we're on a network, there will be a callback. */
        if(buff[5] != 0x93)
        {
                waitsec(10);
```

```
                while(!nHOST_INT);
                EM260_callback(buff);

                waitsec(10);
                EM260_leaveNetwork();
                waitsec(10);
                while(!nHOST_INT);
                EM260_callback(buff);
        }
}

unsigned char EM260_leaveNetwork()
{
        /* Command buffer */
        unsigned char buff[7];

        /*Set up command buffer */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x03;               /* Length of the EZSP frame */
        buff[2] = sq++;               /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x20;               /* EZSP: leaveNetwork() command */
        buff[5] = 0xA7;               /* Frame terminator */

        /* Send a command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 6);
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

unsigned char EM260_scanNetwork()
{
        /* Command buffer */
        unsigned char buff[12];

        /* Set up the command buffer */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x09;               /* Length of EZSP frame */
        buff[2] = sq++;               /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x1A;               /* EZSP: startScan() function */
        buff[5] = 0x01;               /* EZSP: Param: EmberNetworkScanType */
        buff[6] = 0x00;               /* EZSP: Param: channel mask (LSB) */
        buff[7] = 0xF8;               /* EZSP: Param: channel mask */
        buff[8] = 0xFF;               /* EZSP: Param: channel mask */
        buff[9] = 0x07;               /* EZSP: Param: channel mask (MSB) */
        buff[10]= 0x02;               /* EZSP: Param: scan duration exponent */
        buff[11] = 0xA7;      /* Frame terminator */

        /* Send the command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 12);
        memset(buff, 0, 12);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

unsigned char EM260_justJoinNetwork(unsigned char* param)
{
        /* Command buffer */
```

```
        unsigned char buff[19];

        /* Set up the command buffer */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x10;               /* Length of EZSP frame */
        buff[2] = sq++;               /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x1F;               /* EZSP: joinNetwork() command */
        buff[5] = 0x02;               /* EZSP: Param: node type (router) */
        buff[6] = param[8];    /* EZSP: Param: extended PAN ID (LSB) */
        buff[7] = param[9];    /* EZSP: Param: extended PAN ID */
        buff[8] = param[10];   /* EZSP: Param: extended PAN ID */
        buff[9] = param[11];   /* EZSP: Param: extended PAN ID */
        buff[10] = param[12];  /* EZSP: Param: extended PAN ID */
        buff[11] = param[13];  /* EZSP: Param: extended PAN ID */
        buff[12] = param[14];  /* EZSP: Param: extended PAN ID */
        buff[13] = param[15];  /* EZSP: Param: extended PAN ID (MSB) */
        buff[14] = param[6];   /* EZSP: Param: short PAN ID (LSB) */
        buff[15] = param[7];   /* EZSP: Param: short PAN ID (MSB) */
        buff[16] = 0x00;       /* EZSP: Param: radio Tx power (0 dBm) */
        buff[17] = param[5];   /* EZSP: Param: radio channel */
        buff[18] = 0xA7;       /* Frame terminator */

        /* Send the command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 19);
        memset(buff, 0, 19);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

void EM260_getNodeID(unsigned char* buff)
{
        /* Set up the command buffer */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x03;               /* Length of EZSP frame */
        buff[2] = sq++;               /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x27;               /* EZSP: getNodeID() command */
        buff[5] = 0xA7;               /* Frame terminator */

        /* Send the command and receive a response */
        P3OUT &= ~0x01;
        toEM260(buff, 6);
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;
}

unsigned char EM260_sendUnicast(unsigned char* message, unsigned char* nodeID,
        unsigned char len)
{
        /* Command buffer */
        unsigned char command[24+len];

        /* Loop variable */
        unsigned int i;
        i = 0;

        /* Set up the command buffer */
        command[0] = 0xFE;     /* EZSP frame */
        command[1] = 21 + len; /* Length of the EZSP frame */
        command[2] = sq++;     /* EZSP: sequence number */
        command[3] = 0x00;     /* EZSP: command with no sleeping */
        command[4] = 0x34;     /* EZSP: sendUnicast() command */
```

```
        command[5] = 0x00;     /* EZSP: Param: MessageType (direct) */
        command[6] = 0x00;     /* EZSP: Param: destination NodeID (LSB) */
        command[7] = 0x00;     /* EZSP: Param: destination NodeID (MSB) */
        command[8] = 0x00;     /* EZSP: Param: profileID (LSB) */
        command[9] = 0x00;     /* EZSP: Param: profileID (MSB) */
        command[10] = 0x00;    /* EZSP: Param: clusterID (LSB) */
        command[11] = 0x00;    /* EZSP: Param: clusterID (MSB) */
        command[12] = 0x01;    /* EZSP: Param: source EndPoint */
        command[13] = 0x01;    /* EZSP: Param: destination EndPoint */
        command[14] = 0x40;    /* EZSP: Param: APS options (LSB) */
        command[15] = 0x05;    /* EZSP: Param: APS options (MSB) */
        command[16] = 0x00;    /* EZSP: Param: groupID (LSB) */
        command[17] = 0x00;    /* EZSP: Param: groupID (MSB) */
        command[18] = sq2++;   /* EZSP: Param: message sequence number */
        command[19] = 0x64;    /* EZSP: Param: message tag */
        command[20] = len + 2; /* EZSP: Param: length of the message */

        /* Now we include the NodeID, followed by the actual message.
         * Including the NodeID makes it easier for the base station
         * to reply to us.
         */
        command[21] = nodeID[0];
        command[22] = nodeID[1];
        for(i = 0; i < len; i++)
        {
                command[i+23] = message[i];
        }
        command[23+len] = 0xA7;       /* Frame terminator */

        /* Send the command and receive a response. */
        P3OUT &= ~0x01;
        toEM260(command, 24+len);
        memset(command, 0, 24+len);
        fromEM260(command);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return command[5];
}

unsigned char EM260_networkState()
{
        /* Command buffer */
        unsigned char buff[7];

        /* Set up the command buffer */
        buff[0] = 0xFE;               /* EZSP frame */
        buff[1] = 0x03;               /* Length of the EZSP frame */
        buff[2] = sq++;               /* EZSP: sequence number */
        buff[3] = 0x00;               /* EZSP: command with no sleeping */
        buff[4] = 0x18;               /* EZSP: networkState() command */
        buff[5] = 0xA7;               /* Frame terminator */

        /* Send the command and receive a response. */
        P3OUT &= ~0x01;
        toEM260(buff, 6);
        memset(buff, 0, 6);
        fromEM260(buff);
        nHOST_INT = 0;
        P3OUT |= 0x01;

        /* Return the result of the command. */
        return buff[5];
}

void joinNetwork()
{
        /* Command buffers */
        unsigned char buff1[136];
```

```
/* Status */
unsigned char stat;

/* We must first call getNetworkState() */
stat = EM260_networkState();

/* If we are already on a network, we must leave it. */
if(stat == 0x02 || stat == 0x03)
{
	//waitsec(10);
	//while(!nHOST_INT);
	//EM260_callback(buff1);
	//memset(buff1, 0, 136);
	//waitsec(10);
	EM260_leaveNetwork();
	waitsec(10);
	while(!nHOST_INT);
	EM260_callback(buff1);
	memset(buff1, 0, 136);
}

/* Turn off the network indicator LED. */
P6OUT |= 0x04;

/* Find a new network to join. */
buff1[4] = 0x00;
while (buff1[4] != 0x1B)
{
	memset(buff1, 0, 136);
	stat = 0xFF;
	while (stat != 0x00)
	{
		waitsec(10);
		stat = EM260_scanNetwork();
		blinkLED(3);
	}

	waitsec(10);
	EM260_callback(buff1);
}

/* Use the parameters found in scanNetwork() to join
 * a new network.
 */
stat = 0xFF;
while (stat != 0x00)
{
	waitsec(10);
	stat = EM260_justJoinNetwork(buff1);
}

/* Because the order of callbacks is not always predictable,
 * we might not get the joinNetwork() callback first.  So,
 * we'll keep sending callbacks until we do get it.
 */
memset(buff1, 0, 136);
while (buff1[5] != 0x90)
{
	waitsec(10);
	memset(buff1, 0, 136);
	EM260_callback(buff1);
}

waitsec(10);
memset(buff1, 0, 136);

/* Get our new NodeID. */
EM260_getNodeID(buff1);
nodeID[0] = buff1[5];
```

```
	nodeID[1] = buff1[6];

	/* Turn on the network indicator LED. */
	P6OUT &= ~0x04;
}

void transmitPicture(unsigned int imgNum)
{
	/* Message buffer */
	unsigned char buff1[136];

	/* Number of packets to send */
	unsigned int packets;

	/* Result of EM260 commands */
	unsigned char status;

	/* Number of bytes not yet sent */
	unsigned int bytesRemaining;

	/* Used in packet calculations */
	unsigned int tmp;

	/* Loop variables */
	unsigned int i;
	unsigned int j;
	unsigned int k;

	/* First, we must calculate the number of packets to send. */
	packets = picSize[imgNum] / 64;
	if(packets * 64 < picSize[imgNum])
		++packets;

	/* Now, we must convert the number of packets into readable hex. */
	buff1[13] = packets / 4096;
	tmp = packets % 4096;
	buff1[14] = tmp / 256;
	packets = tmp % 256;
	buff1[15] = packets / 16;
	buff1[16] = packets % 16;

	/* We must now send the IMAGE_READY0xNNNN signal. */
	buff1[0] = 'I';
	buff1[1] = 'M';
	buff1[2] = 'A';
	buff1[3] = 'G';
	buff1[4] = 'E';
	buff1[5] = '_';
	buff1[6] = 'R';
	buff1[7] = 'E';
	buff1[8] = 'A';
	buff1[9] = 'D';
	buff1[10] = 'Y';
	buff1[11] = '0';
	buff1[12] = 'x';
	buff1[13] += (buff1[13] > 9 ? 55 : 48);
	buff1[14] += (buff1[14] > 9 ? 55 : 48);
	buff1[15] += (buff1[15] > 9 ? 55 : 48);
	buff1[16] += (buff1[16] > 9 ? 55 : 48);

	status = EM260_sendUnicast(buff1, nodeID, 19);
	while(status != 0x00)
	{
		wait_a();
		joinNetwork();
		wait_a();
		status = EM260_sendUnicast(buff1, nodeID, 19);
	}

	/* Wait for the message sent callback. */
```

```
	memset(buff1, 0, 136);
	wait_a();
	while(!nHOST_INT);
	EM260_callback(buff1);
	memset(buff1, 0, 136);

	/* Now, wait for the base station to send the IMAGE_READY_OK
	 * signal.  The first callback will get a generic ACK from the
	 * base station and the second will get the IMAGE_READY_OK signal.
	 */
	while(!nHOST_INT);
	while(buff1[4] != 0x45)
	{
		memset(buff1, 0, 136);
		EM260_callback(buff1);
		wait_a();
	}
	memset(buff1, 0, 136);
	while(buff1[4] != 0x45)
	{
		memset(buff1, 0, 136);
		EM260_callback(buff1);
		wait_a();
	}

	/* Now, we can actually send the image to the base station.  We do
	 * so by reading the image data from the SD card in blocks of 512
	 * bytes and sending it in packets of 64 bytes.
	 */
	bytesRemaining = picSize[imgNum];
	while(bytesRemaining)
	{
		mmcReadBlock((512*numBlocks++), 512);
		for(i = 0; i < 512; i++)
		{
			memset(buff1, 0, 136);
			k = (bytesRemaining >= 64 ? 64 : bytesRemaining);
			buff1[0] = k / 16 + ((k / 16) > 9 ? 55 : 48);
			buff1[1] = k % 16 + ((k % 16) > 9 ? 55 : 48);
			buff1[2] = ',';
			for(j = 0; j < k; j++)
			{
				buff1[j+3] = mmc_buffer[i++];
			}
			--i;
			do
			{
				status = EM260_sendUnicast(buff1, nodeID, k+5);
				wait_a();
			} while(status != 0x00);

			memset(buff1, 0, 136);
			EM260_callback(buff1);
			memset(buff1, 0, 136);
			wait_a();
			bytesRemaining -= k;
			do
			{
				memset(buff1, 0, 136);
				EM260_callback(buff1);
				wait_a();
			} while(buff1[4] != 0x45);

			if(bytesRemaining == 0)
				break;
		}
	}
}
```

Appendix I: ETRX2USB Sender Code

This program allows one to use an ETRX2USB device to send a file to another ETRX2USB device. As mentioned, the 2.5.x version of Ember's ZigBee stack software running on the Telegesis ETRX2USB devices is not compatible with the 3.x stack running on the EM260 RCMs. However, Telegesis is currently in the process of updating their software to be compatible with the 3.x stack. As such, this code may be useful in a later project.

em250_sender.c:

```
ETRX2USB Sender
Copyright (C) 2007 Joseph Bosman, Steve Olivieri, Ipek Ozil, Brandon Steacy

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA


#include "em250.h"

HANDLE setupCOM(LPCSTR port)
{
	/* Serial port configuration settings. */
	DCB comConfig;

	/* Serial port timeout settings. */
	COMMTIMEOUTS comTimeouts;

	/* The file handle to return to the caller. */
	HANDLE ret;

	/* Open the serial port. */
	ret = CreateFile((LPCSTR)port, GENERIC_READ|GENERIC_WRITE, 0, NULL,
		OPEN_EXISTING, 0, NULL);
	if(ret == INVALID_HANDLE_VALUE)
		return INVALID_HANDLE_VALUE;

	/* Get the port's configuration settings, then change them to better
	 * match the ETRX2USB device.
	 */
	if(GetCommState(ret, &comConfig))
	{
		comConfig.BaudRate = 19200;
		comConfig.ByteSize = 8;
		comConfig.Parity = NOPARITY;
		comConfig.StopBits = ONESTOPBIT;
		comConfig.fBinary = FALSE;
		comConfig.fParity = FALSE;
	}
	else
		return INVALID_HANDLE_VALUE;
	if(!SetCommState(ret, &comConfig))
		return INVALID_HANDLE_VALUE;
```

```
	/* Get the port's timeout settings, then set them all to 25 ms.  This
	 * is probably not an ideal solution.
	 */
	if(GetCommTimeouts(ret, &comTimeouts))
	{
		comTimeouts.ReadIntervalTimeout = 25;
		comTimeouts.ReadTotalTimeoutConstant = 25;
		comTimeouts.ReadTotalTimeoutMultiplier = 25;
		comTimeouts.WriteTotalTimeoutConstant = 25;
		comTimeouts.WriteTotalTimeoutMultiplier = 25;
	}
	else
		return INVALID_HANDLE_VALUE;
	if(!SetCommTimeouts(ret, &comTimeouts))
		return INVALID_HANDLE_VALUE;

	/* Empty the input and output buffers for the serial port. */
	PurgeComm(ret, PURGE_RXCLEAR|PURGE_TXCLEAR);

	/* Return the file handle for the serial port. */
	return ret;
}

int writeSerial(LPCSTR data, PHANDLE comPort, UINT len, BOOL command)
{
	/* Single-character buffer for writing strings to the serial port. */
	CHAR zBuff;

	/* How many bytes we wrote on the last write call. */
	DWORD bWritten;

	/* How many bytes we've read. */
	DWORD bRead;

	/* Loop variable. */
	UINT i;

	/* Write a command to the serial port. */
	for(i = 0; i < len; i++)
	{
		zBuff = *(data+i);
		if(!WriteFile(*comPort, (LPVOID)&zBuff, 1, &bWritten, NULL))
			return FAILURE;
	}

	/* Send a carriage return to finish the command. */
	if(command)
	{
		zBuff = '\r';
		if(!WriteFile(*comPort, (LPVOID)&zBuff, 1, &bRead, NULL))
			return FAILURE;
	}

	return SUCCESS;
}

int readMessage(PHANDLE in, LPSTR buff)
{
	/* Temporary buffer. */
	CHAR zBuff;

	/* Stores how many bytes the last ReadFile() read. */
	DWORD bRead;

	/* Stores how many total bytes we've read. */
	INT i;

	/* Initialize i.  Then, read in the data one byte at a
	 * time and write it to buff.  Data is read this way for COM
	 * support.  We finish when there is no more data to be read.
```

```
         */
        i = 0;
        do
        {
                ReadFile(*in, (LPVOID)&zBuff, 1, &bRead, NULL);
                if(bRead > 0)
                {
                        *(buff+i) = zBuff;
                        ++i;
                }
        } while (bRead > 0);

        return i;
}

LPSTR getMessage(PHANDLE in, LPCSTR message, PHANDLE out)
{
        /* Temporary buffer for storing data that we read from the COM port. */
        LPSTR buff;

        /* Temporary buffer that points to part of buff containing the desired
         * message (if it exists) or nothing (otherwise).
         */
        LPSTR tBuff;

        /* We copy tBuff into a separate buffer for returning to avoid memory
         * leaks.
         */
        LPSTR retBuff;

        /* How many bytes we wrote. */
        DWORD bWritten;

        /* How many bytes we read. */
        INT i;

        /* Since we're storing temporary data in buff, we need to allocate
         * memory for it now.  Since we use tBuff is a BOOL, we leave it
         * NULL for now and only give it a value when we find the message
         * that we're looking for (or an ERROR).
         */
        buff = (LPSTR)calloc(150, sizeof(CHAR));
        tBuff = NULL;

        /* While we haven't found the desired message (or an ERROR), keep
         * looking!
         */
        while(!tBuff)
        {
                /* Read data from the COM port. */
                memset(buff, 0, 150);
                i = readMessage(in, buff);

                /* Search that data for our desired message. If we don't find it
                 * search for an ERROR message.
                 */
                tBuff = strstr(buff, message);
                if(!tBuff)
                        tBuff = strstr(buff, "ERROR:");

                /* If out is not NULL, write any data that we've read to the file
                 * pointed to by out.  Usually, this is a log file.  And, we don't
                 * really care if it fails.
                 */
                if(out)
                        WriteFile(*out, (LPVOID)buff, i, &bWritten, NULL);
        }

        /* Since we have something to return, let's allocate memory for retBuff
         * and then copy the useful data into it.  The caller of getMessage()
```

```
         * is responsible for freeing retBuff.
         */
        retBuff = (LPSTR)calloc(150, sizeof(CHAR));
        strcpy(retBuff, tBuff);

        free(buff);
        free(tBuff);
        return retBuff;
}

DWORD ceiling(double x)
{
        /* Return value. */
        int ret;

        /* This basically abuses casting in C.   If I cast a float (or a double)
         * to an integer, anything after the decimal point is simply dropped.
         */
        ret = (DWORD)x;

        /* Now, if the chopped version is equal to the original, then we were
         * looking at an integer to begin with.  Otherwise, there was a decimal
         * component and we need to round up.
         */
        if(ret == x)
                return ret;
        else
                return (ret + 1);
}

int main(int argc, char* argv[])
{
        /* Handle for the COM port we're writing to. */
        HANDLE serial;

        /* Handle for the file we're sending. */
        HANDLE imageFile;

        /* The COM port to connect to. */
        LPSTR comPort;

        /* Filename for the input file. */
        LPSTR fileName;

        /* Buffers for storing data. */
        LPSTR buff;
        LPSTR tBuff;

        /* The hardware ID number for the sender. */
        LPSTR hwID;

        /* How many bytes we read. */
        DWORD bRead;

        /* The size of the file.  Note that we don't support files
         * larger then 2GB.
         */
        DWORD fileSize;

        /* The number of packets that we need to send. */
        DWORD numPackets;

        /* Loop variables. */
        INT i;
        UINT j;

        /* Make sure that they specify the required command
         * arguments (COM port and directory).
         */
        if(argc < 5)
```

```
        return MAIN_INPUT_ERROR;

/* Allocate memory for our strings. */
comPort = (LPSTR)calloc(6, sizeof(CHAR));
fileName = (LPSTR)calloc(1024, sizeof(CHAR));
buff = (LPSTR)calloc(150, sizeof(CHAR));
hwID = (LPSTR)calloc(17, sizeof(CHAR));

/* Initialize variables. */
tBuff = NULL;
bRead = 0;
numPackets = 0;
fileSize = 0;

/* Parse input. */
for(i = 1; i < argc; i++)
{
        if(strcmp(argv[i], "-c") == 0)
                sprintf(comPort, "COM%s:", argv[++i]);
        else if(strcmp(argv[i], "-f") == 0)
                sprintf(fileName, "%s", argv[++i]);
        else
                return MAIN_INPUT_ERROR;
}

/* Open the serial port. */
serial = setupCOM((LPCSTR)comPort);
if(serial == INVALID_HANDLE_VALUE)
{
        fprintf(stderr, "Could not open %s.\n", comPort);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_SERIAL_PORT_FAIL;
}

/* Tell the EM250 to leave any PANs that it might be in.   If we get an
 * ERROR here, that just means that we weren't already in a PAN.  So, we
 * can ignore it.
 */
if(writeSerial("AT+DASSL", &serial, strlen("AT+DASSL"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_DATA_WRITE_ERROR;
}
tBuff = getMessage(&serial, "LeftPAN", NULL);
free(tBuff);

/* Turn off the annoying echo when we're sending binary data.
 * Failure to do so will break ACKs and other fun things.
 */
if(writeSerial("ATS0B4=1", &serial, strlen("ATS0B4=1"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_DATA_WRITE_ERROR;
}
tBuff = getMessage(&serial, "OK", NULL);
if(strstr(tBuff, "ERROR:"))
{
```

```
        fprintf(stderr, "Could not set register value 0B4, %s.\n", tBuff);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        free(tBuff);
        return MAIN_REGISTER_FAIL;
}
free(tBuff);

/* We also need to turn off the ability to have multiple packets in
 * flight at once.  That way, we're certain that the receiver will
 * never be overloaded.
 */
if(writeSerial("ATS06D=1:TG-ETRX1", &serial, strlen("ATS06D=1:TG-ETRX1"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_DATA_WRITE_ERROR;
}
tBuff = getMessage(&serial, "OK", NULL);
if(strstr(tBuff, "ERROR:"))
{
        fprintf(stderr, "Could not set register value 06D, %s.\n", tBuff);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        free(tBuff);
        return MAIN_REGISTER_FAIL;
}
free(tBuff);

/* Join the next best ZigBee network. */
if(writeSerial("AT+JN", &serial, strlen("AT+JN"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_DATA_WRITE_ERROR;
}
tBuff = getMessage(&serial, "JPAN:", NULL);
if(!strstr(tBuff, "JPAN:"))
{
        fprintf(stderr, "Could not join a PAN.\n");
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        free(tBuff);
        return MAIN_JOIN_FAIL;
}
strncpy(buff, tBuff+5, 7);
fprintf(stdout, "Joined network %s.\n", buff);
free(tBuff);
memset(buff, 0, 150);

/* There's almost certainly a bunch of data in the buffers
 * now, but we don't really care what it says.  So, we'll
 * purge it.
```

```
 */
PurgeComm(&serial, PURGE_RXCLEAR);

/* Now, we need to get the hardware ID of our coordinator.  The
 * coordinator always acts as the base station, so this is where
 * we want to send our files.
 */
if(writeSerial("AT+SN:00", &serial, strlen("AT+SN:00"), TRUE))
{
        fprintf(stderr, "Could not write to %s.\n", comPort);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_DATA_WRITE_ERROR;
}
tBuff = getMessage(&serial, "COO:", NULL);
if(!strstr(tBuff, "COO:"))
{
        fprintf(stderr, "Could not find PAN coordinator.\n");
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        free(tBuff);
        return MAIN_COO_ERROR;
}
strncpy(hwID, tBuff+4, 16);
free(tBuff);
fprintf(stdout, "PAN Coordinator: %s\n", hwID);

/* Open the image that we plan to send.  We do this before sending
 * the IMAGE_READY signal because we first need to calculate how
 * many packets we'll be sending.
 */
imageFile = CreateFile((LPCSTR)fileName, GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);
if(imageFile == INVALID_HANDLE_VALUE)
{
        fprintf(stderr, "Could not open file: %s\n", fileName);
        CloseHandle((PVOID)&serial);
        free(comPort);
        free(fileName);
        free(buff);
        free(hwID);
        return MAIN_READ_ERROR;
}
fprintf(stdout, "Opened file: %s\n", fileName);

/* Get the size of the file.  Note that we're limited to files of 2GB
 * or less because we're using GetFileSize() instead of GetFileSizeEx().
 * This should be okay, as I don't expect a 640x480 JPEG to ever be more
 * than 2GB.  In fact, it can't be!
 */
fileSize = GetFileSize(imageFile, NULL);

/* Our packets will each contain 50 bytes of data.  So, determine
 * how many packets we'll need to send by diving the file size by 50
 * and taking the ceiling of that number.
 */
numPackets = ceiling((double)((fileSize)/65.0f));
fprintf(stdout, "File is %li bytes, need to send %li packets.\n",
        fileSize, numPackets);

/* Now, we'll let the base station know that we have a file to send.  If
 * we get the IMAGE_READY signal through, we'll wait for IMAGE_READY_OK.
 * If we don't get it, we'll send IMAGE_READY again and wait again.  We'll
 * do this at most 10 times before giving up entirely.
```

```
 */
for(j = 0; j < 10; j++)
{
	/* Send the IMAGE_READY signal to the base station.  The format for
	 * this command is IMAGE_READY0xNNNN where NNNN is numPackets in
	 * hex.  We'll try this 10 times before giving up.
	 */
	memset(buff, 0, 150);
	sprintf(buff, "AT+UCAST:%s=IMAGE_READY0x%.4x",
		hwID, (unsigned int)numPackets);
	for(i = 0; i < 10; i++)
	{
		writeSerial(buff, &serial, strlen(buff), TRUE);
		fprintf(stdout, "Sent IMAGE_READY0x%.4x to %s.\n",
			(unsigned int)numPackets, hwID);

		/* Since we set register S06D, we'll get either "OK" or "ERROR"
		 * in response to our transmission (rather than ACK/NACK).
		 */
		tBuff = getMessage(&serial, "OK", NULL);
		if(strstr(tBuff, "OK"))
		{
			fprintf(stdout, "\tGot an ACK from %s.\n", hwID);
			break;
		}
		else
		{
			fprintf(stdout, "\tNo ACK from %s.\n", hwID);

			/* If we've already tried ten times (i == 9), then set
			 * i = 10 for later error checking.
			 */
			if(i == 9)
				++i;
			free(tBuff);
		}
	}

	/* If none of our ten attempts worked, we'll have to quit. */
	if(i == 10)
	{
		fprintf(stderr, "Failed to send IMAGE_READY signal to %s.\n",
			hwID);
		CloseHandle((PVOID)&serial);
		CloseHandle((PVOID)&imageFile);
		free(comPort);
		free(fileName);
		free(buff);
		free(hwID);
		return MAIN_NO_CONNECTION;
	}

	/* Now, we need to wait for the IMAGE_READY_OK signal. Sometimes it
	 * gets read in with the ACK.  So, check for it first.  If we don't
	 * already have it, wait for it.  If we *still* don't get it, try
	 * again.
	 */
	if(!strstr(tBuff, "IMAGE_READY_OK"))
	{
		free(tBuff);
		tBuff = getMessage(&serial, "IMAGE_READY_OK", NULL);
	}
	if(!strstr(tBuff, "IMAGE_READY_OK"))
	{
		fprintf(stdout, "Did not get the OK from %s\n", hwID);
		continue;
	}
	else
		break;
}
```

```
	/* If we made it here, we've got the IMAGE_READY_OK signal. */
	fprintf(stdout, "Got the IMAGE_READY_OK signal from %s.\n", hwID);

	/* No, we can begin sending packets. */
	for(i = 0; i < numPackets; i++)
	{
		/* Read the next 50 bytes in the file. */
		memset(buff, 0, 150);
		ReadFile(imageFile, (LPVOID)buff, 65, &bRead, NULL);

		/* Now, try to send it to the base station.  We'll try each
		 * packet at most 10 times before aborting.
		 */
		for(j = 0; j < 10; j++)
		{
			/* Send the command. */
			tBuff = (LPSTR)calloc(150, sizeof(CHAR));
			sprintf(tBuff, "AT+UCASTB:%.2x,%s", (unsigned int)bRead, hwID);
			writeSerial(tBuff, &serial, strlen(tBuff), TRUE);

			/* Just to be safe. */
			Sleep(10);

			/* Send the data. */
			writeSerial(buff, &serial, bRead, FALSE);
			free(tBuff);
			fprintf(stderr, "Sent packet %i of %li (%li bytes).\n", i+1, numPackets,
bRead);

			/* Wait for OK/ERROR. */
			tBuff = getMessage(&serial, "OK", NULL);
			if(strstr(tBuff, "OK"))
			{
				fprintf(stdout, "\tGot ACK.\n");
				free(tBuff);
				PurgeComm(&serial, PURGE_TXCLEAR|PURGE_RXCLEAR);
				break;
			}
			else
			{
				free(tBuff);
				PurgeComm(&serial, PURGE_TXCLEAR|PURGE_RXCLEAR);

				/* If we've already tried 10 times, give up. */
				if(j == 9)
				{
					++j;
					break;
				}
			}
		}

		/* We failed 10 times, so we're quitting. */
		if(j == 10)
		{
			fprintf(stderr, "Lost connection to %s, aborting.\n", hwID);
			break;
		}
	}

	fprintf(stdout, "Sent %i of %li packets to %s.\n", i, numPackets, hwID);

	/* Close the files. */
	CloseHandle((PVOID)&serial);
	CloseHandle((PVOID)&imageFile);

	/* Free any used memory and exit. */
	free(comPort);
	free(fileName);
```

```
        free(buff);
        free(hwID);

        return MAIN_SUCCESS;
}
```