Implementation of a Heterogeneous Sensor Network for Structural Health Monitoring

Gregory P. Jaman

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

April 30, 2007

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Implementation of a Heterogeneous Sensor Network for Structural Health Monitoring

BY

Gregory P. Jaman

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Master of Science

Gregory P. Jaman © 2007

Contents

List of Figures

Glossary

CAN	Controller Area Network
CAP	Contention Access Period
CCA	Clear Channel Assessment
CFP	Contention Free Period
CH	Cluster Head
CRC	Cyclic Redundancy Check
CSMA-CA	Carrier Sense Multiple Access with Collision Avoidance
DLC	Data Length Code
DSN	Data Sequence Number
DSSS	Direct Sequence Spread Spectrum
E	Energy
ED	Energy Detection
EID	Identifier Field
FIFO	First In First Out
GPS	Global Positioning System
GTS	Guaranteed Time Slot
GW	Gateway
HAA	Hardware Abstraction Architecture
HPL	Hardware Presentation Layer
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input / Output
IDE	Identifier Extension Bit
LQI	Link Quality Indication
MAC	Medium Access Control
MDT	Moisture Detection Tape

MFR	MAC Footer
MHR	MAC Header
MPDU	MAC Protocol Data Unit
MSDU	MAC Service Data Unit
PIC	Microcontroller developed by Microchip Technology Inc.
PHY	Physical Layer
QoS	Quality of Service
RF	Radio Frequency
RMU	Remote Measurement Unit
RTR	Remote Transmission Request Bit
RSSI	Receive Signal Strength Indicator
RX	Receive
SFD	Start of Frame Delimiter
SPI	Serial Peripheral Interface
SRR	Substitute Remote Request Bit.
TX	Transmit
WSN	Wireless Sensor Network

Acknowledgement

I would like to take this opportunity to thank all the people who have contributed towards this thesis.

I would first like to thank my advisors, Dr. Robert D. McLeod and Dr. Sajid Hussain, for their sincere guidance and encouragement through my academic years as well as their contributions and help with this thesis. In addition, I would like to express my appreciation for their kind and generous personalities.

I would also like to thank the members of my thesis committee, Dr. Peter Graham and Dr. Jun Cai for taking the time to be on my committee, and for having read my thesis.

Furthermore, I would like to thank my wife, Lisa Jaman, and our parents for their constant encouragement and support.

Abstract

This thesis investigates implementation and design issues for a heterogeneous network for structural health monitoring. A heterogeneous wireless/wired architecture is a practical if not near optimal solution to many problems amenable to wireless sensor monitoring. The specific application addressed in this thesis is the monitoring of moisture within a building or structure. The laborious installation of a wired network and interconnection of the sensors with the structure makes wireless a potential solution to the problem albeit constrained by energy management. The proposed application integrates wireless sensors using IEEE 802.15.4 and controller area network (CAN) to provide energy efficient monitoring. The contributions include: implementation of the PIC18f4680 platform and hardware modules for IEEE 802.15.4 and CAN in TinyOS, creating a message protocol and routing for IEEE 802.15.4 and CAN, and developing an integrated hardware prototype that includes the components mentioned. The thesis demonstrates the efficiency of a heterogeneous architecture for wireless sensor monitoring in a real environment with hard constraints.

Chapter 1

Introduction

Structural health monitoring is an on going topic of research in wireless sensor networks. To get accurate data about a structure, the sensors typically need to be embedded within the structure, in locations that are not easily accessed without demolition. Because of these constraints, the sensing lifetime of the network must be reasonable in comparison to building life. It would not be unreasonable for building developers to demand a sensing lifetime in the order of 10's of years.

Wired Sensor Networks in the past have been inexpensive and more reliable compared to Wireless Sensor Networks (WSNs). However, in recent years, WSNs are continually becoming less expensive. As well, advances in signal-processing techniques and new network architectures are making WSNs more reliable. In contrast, the cost of wired networks has remained constant or increased as there has been an increase in the cost of the wire, the connectors, and the labor to connect point A to point B. WSN lifetime is limited by the battery capacity and power consumption whereas a wired network has potentially infinite lifetime. Coverage and reliability is also limited in WSNs. Although techniques such as multi-hop routing can be deployed to increase coverage, it comes at the cost os increased drain on battery power. A heterogeneous network which mixes wired and wireless sensors can take advantage of the benefits of each network type[JJK04].

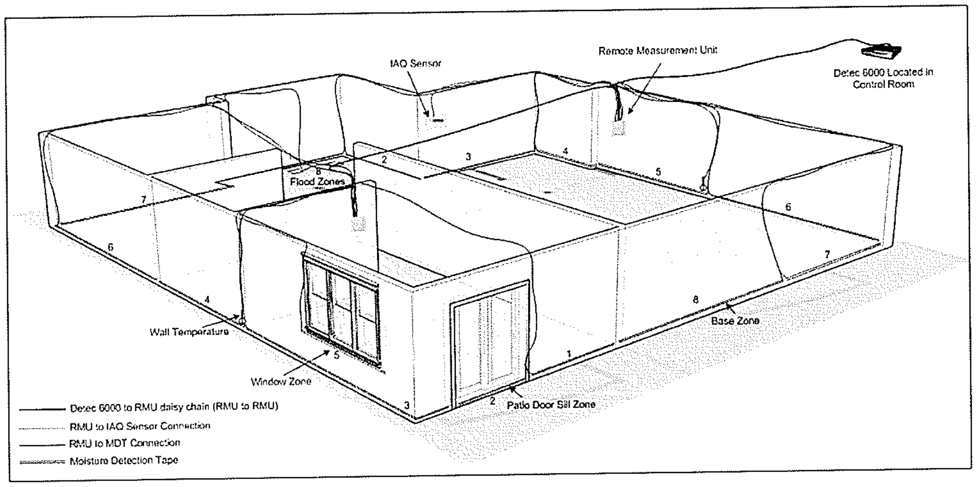


Figure 1.1: Wired Sensor Network for Moisture Detection

The term, civionics, has been recently introduced to describe the discipline of monitoring structures. Civionics is an emerging discipline describing the interaction between civil engineering and electronics. Integration of intelligent sensing will allow civil structural engineers to evaluate new concepts, materials, and understand aging buildings. The demand for this technology is evident in problems such as moisture intrusion in sealed building envelope systems. New buildings being constructed with efficiency in mind are sealed such that air does not easily pass through walls. Any intrusion of water is thus absorbed into the walls and not allowed to evaporate, resulting in the wood frame rotting or mold growing. The effects of moisture in building structures are out of the scope of this thesis, but are the motivation behind the sensor network and its application.

This thesis evaluates an existing wired sensing network for detecting moisture in building structures and proposes a method for integrating wireless nodes. The benefits of wireless nodes are seen in decreased installation and maintenance costs. A typical installation of a wired network for moisture detection in a building in shown in Figure 1.1. Segments of Moisture Detection Tape (MDT) are installed along the perimeter of the building. These segments of tape act as sensors providing data to the Remote Measurement Units (RMUs). Significant time and cost is required to route wires in stud cavities to sensors, between RMU's, and back to the gateway (Detec 6000) as seen in Figure 1.1. Installation costs have been valued at approximately \$50USD per segment of tape, overshadowing the cost of introducing wireless components to the hardware. In addition, because of the nature of the measurements, electrical isolation is extremely important. Resistive values measured on the tape range from $100\text{K}\Omega$to $100\text{M}\Omega$. By sharing common power and communication wiring between nodes, accidental grounding to building ground introduce errors over this large measurement range. Accidental grounding is typical in large commercial structures. Tradesmen not familiar with the electronics can unknowingly compromise the installation by faulty wiring or grounding of the tape. Figure 1.1 (left) makes clear the amount of labour required to route wires thought walls in a structure. Figure 1.2 (right) shows the connectivity between eights segments of tape. Depending on the size of a structure, it is not uncommon to have over 1000 segments.

It should be evident how accidental wiring errors can easily occur.

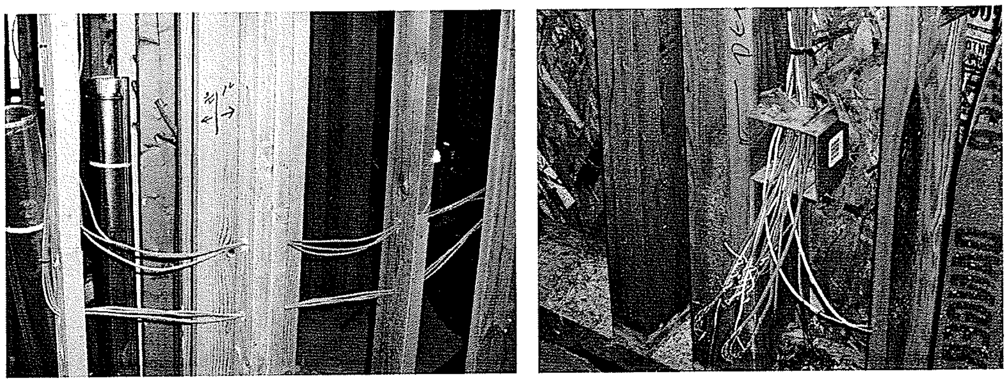


Figure 1.2: (Left) routing of wires in stud cavities, (right) wires terminating at a junction box

Using wireless sensors avoids many problems, however implementing a pure wireless network is not always optimal. In certain instances harsh environments can limit the communication range or prohibit devices from operating in the same frequency band. In some instances wireless communication may also fail when loss rates become unacceptable. To span large distances, multi-hop routing is required in a mesh network topology. Multi-hop routing requires complex algorithms to synchronize communication and increased power consumption due to the need to listen for neighbor's requests. In these cases an option for a node to communicate over a wired medium is necessary.

Wired networks suffer from high installation time and costs and limited installation options. On the other hand, wired networks provide power and more reliable communication to nodes. This thesis will discuss and implement a heterogeneous sensor network[1].

This thesis presents the design and evaluation of a heterogeneous network which is based on both wireless and wired networks. The contribution includes modifying TinyOS, an event based operating system designed for use with embedded network sensors, to include support for Controller Area Networks (CANs) and the PIC18f4680 platform [KWSH05, LO04], creating a message protocol over CAN. The thesis also includes the modification to 802.15.4 to support extremely low powered communication to maximizes sensing lifetime. An integrated hardware prototype that includes the components mentioned is also developed.

In the wireless sensor network community the term mote is used to refer to a wireless sensor. A mote is usually a single purpose sensor that is power constrained and physically very small. Within this thesis, the hardware developed can act as either an autonomous mote, which is power constrained, or a cluster head communicating over a wired medium with access to external power. In a star network topology, objects located around the root are considered nodes. Topology plays an important role throughout this thesis, therefore hardware operating as an autonomous mote is referred to as an end node and hardware with the ability to route messages from end nodes is referred to as a cluster head.

This chapter described the motivation and goals for this thesis. The following Chapters include the related work, problem, design, implementation details including the protocols, some results, and finally the conclusions.

Heterogeneous in this thesis implies a network with multiple communication mediums

[1]specifically wired and wireless

Chapter 2

Related Work

This chapter discuses the related work and the state of art research conducted in the field of WSNs. Section 2.1 discusses of TinyOS for the Microchip PIC microcontroller. Section 2.2 discusses the work done in heterogeneous networks to improve lifetime. Section 2.3 presents the existing research in managing energy consumption using various methods. Finally, section 2.4 explores the common hardware chosen for wireless sensor networks.

2.1 PIC and TinyOS

Lynch [LO04] evaluated the processor choices for wireless sensor networks to match the requirements of TinyOS. Various processors were compared, such as the AVR, PIC16, PIC18, MSP, and the 8051. Results showed the PIC18's performance was just shy of the MSP.

Lynch [LR05] describes the issues involved in running TinyOS on a PIC-based wireless sensor. It showed that the final performance is comparable to or better than existing sensor systems for a limited set of applications.

Kobert [KWSH05] showed it was feasible to implement TinyOS in a Microchip PIC18F series MCU designed for low-power devices. Due to the missing GCC tool chain to compile code for the PIC platform, NesC's C output files were modified by a perl script to conform to the Microchip C18 compiler. Future goals were to build the system around the PIC18F4620 which includes $nanoWattTechnology$ and greater memory resources. At the inception of this thesis, modules for the major components of TinyOS for the PIC18F4620 are committed to the CVS. This thesis used the committed code as a template and guide.

Li et al. [SFLR01] argues that event-driven OS architectures, such as TinyOS, are orders of magnitude better in terms of performance, instruction and data memory requirements, and power consumption than traditional general-purpose operating systems.

2.2 Heterogeneous Networks

Jae-Joon [JJK04] analyzed the coverage aging process of a heterogeneous deployment[1]. Lifetime sensing coverage was analyzed for both single and multi-hop communication. An optimal mixture of many inexpensive low-capability devices and more expensive high-capability devices was found mathematically and verified through simulations to extend the duration of the network's sensing performance.

[1]Heterogeneous in this context was defined as nodes with different capabilities.

2.3 Energy Consumption

To maximize network lifetime, managing energy consumption plays a major role. Techniques to reduce energy consumption include reducing the total number of transmit and receive messages, reducing the number of long range transmissions, reducing the duty cycle time, and processing queries using cross layer optimization. Significant research has been done on reducing the number of transmissions using methods such as cluster based techniques [HCB00, GSYS02, YLC$^+$02, BC03, YF04, LYCW05], solving maximum flow problems with linear programming [CT04, BC02, DKN03, KDN02, SK04], dynamic spanning trees based on energy metrics [CABM03, WTC03, SR02, dTdK03, HI], and genetic algorithms for intelligent routing [GKK$^+$89]. The work introduced in this thesis simplifies the routing by using single hop communications while taking advantage of cluster based communication where the network is divided into virtual clusters having cluster members transmit to cluster heads, and cluster heads transmit to base stations.

2.4 Hardware

As wireless sensor networks become increasing popular, new technologies have emerged for mote hardware. Current popular choices include MICAz[Tecc], TelosB[Tecd], Imote2[Tecb], BTNode[Teca], and TMote[TMo]. Many of thes motes an IEEE 802.15.4 compatible radio with a low power microprocessor such as the TI MSP430 or Atmel ATmega 128L. Hardware requirements for this thesis include a controller area network controller and transceiver as well the analog circuitry required for moisture detection. Consideration was taken as to whether to modify an existing wireless mote to include the required components or modify an existing wired sensor to include an IEEE 802.15.4 transceiver.

Most of the related work focuses on specific parts of wireless networks while this thesis, although influenced by such work, focuses instead on the system as a whole.

Chapter 3

Problem

3.1 Introduction

Several problems need to be considered in the overall design of a sensor network. First, the implementation issues of a heterogeneous network will be evaluated. These problems will establish the underlying decisions for the design and implementation in the following chapters.

3.2 Implementation Issues

Nodes in sensor networks need to interact with one another only infrequently. The motivation for one wireless node to interact with another wireless node would be to redirect/relay messages down to the root node. By designing a network to limit wireless/wireless node communication, significant savings can be achieved. The implementation problems are as follows:

- *Performance metrics:* acceptable estimates for system lifetime, network latency, capacity, throughput, fault tolerance, and scalability are required to make educated choices about network topology and design.

- *Protocols:* the protocols at every layer of the OSI model must be evaluated for the target application. To design efficient protocols, issues such as clustering, broadcasting, and sleep modes must be addressed.
- *Capacity/throughput:* the capacity and throughput requirements of the network affect network topology, software and hardware design, and the communication protocols used.
- *Routing:* the routing choices will affect the network topology. A many-to-one network involves all nodes reporting to a base station. Whereas, multi-hop routing can be implemented to span large areas wirelessly.
- *Channel Access and Scheduling:* helps energy and delay balancing. Because wireless nodes access a shared medium, problems such as collisions and spatial reuse must be considered.
- *Modeling:* before implementation, modeling could provide suggestions for design improvements. Modeling can help determine the number of motes and relative distribution, degree and type of mobility, characteristics of wireless links, volume of traffic injected by the source, life-span of nodes interaction, and detailed energy consumptions.
- *Connectivity:* is a very important issue for wireless sensor networks. Building monitoring requires that the network is not partitioned into disjoint parts.

- *Quality of Service(QoS):* QoS changes from application to application. Well defined quality of service constraints are required before proceeding with network design to ensure the network has the capacity to delivery reliable and timely data. Depending on the level of QoS, a network's performance may not be sufficient to satisfy the delay requirements. For example, because wireless nodes have power constraints and sleep for the majority of the time, a QoS constraints that require realtime data are not realistic.

Although a standard IEEE 802.15.4 implementation addresses many of the topics mentioned above, monitoring applications require these standards be re-evaluated to suit long term building monitoring. Many of the implementation problems are overlapping and require cross layer design.

3.3 Environment Deployment Issues

Network deployment is constrained by communication ranges. Human interaction is needed at every level of design. Therefore a set of rules need to be created to facilitate network design. Rule sets for the type of environment need to be established, which requires testing various environments and determining reasonable communication ranges. Also, tools need to be created to evaluate the environment before the network is deployed to map out "dead" spots. In special cases, such as extremely harsh environments, wired devices may be the only suitable device. Secondly, the co-existence of other devices in the 2.4GHz frequency band need to be considered. Evaluation of channel selection, not only for the establishment of subnetworks, but for interface from other devices is also necessary.

This chapter discussed some problems associated with using sensor networks. Many of these problems affect components at many levels of the system, therefore a cross level design approach needs to be taken. The results of this thesis will touch on every problem addressed in this chapter.

Chapter 4

Design

This chapter will discuss the design of a heterogeneous network for maximizing sensing lifetime. Section 4.1 presents the objectives of the software and hardware. Section 4.2 presents IEEE 802.15.4 as the wireless protocol and discusses the topology chosen. Section 4.3 discuses why CAN was chosen as the protocol for the wired medium. Section 4.4 presents the problems and solution for routing between the two protocols. Section 4.5 presents TinyOS and discusses why it was chosen. Finally, Section 4.6 presents the operation of the monitoring software by describing the autonomous tasks involved.

4.1 Software and Hardware Design

Hardware abstraction in system design allows for portability and simplified development by hiding hardware intricacies. This can conflict with the performance and energy-efficiency of WSN applications. An operating system, such as TinyOS, based on a component-based model, allows for component re-usability while maintaining energy-efficiency by allowing full access to the hardware capabilities. The Hardware Abstraction Architecture (HAA) in TinyOS provides balance between development convenience and efficiency.

All layers of the HAA, application design and implementation were explored. Because a new hardware target is introduced, many modules for the Hardware Presentation Layer (HPL) needed to be developed and tested. With respect to software design, the hardware interaction layer should be abstracted from the application layer so applications are unaware of the medium their messages are being transmitted on. A message protocol also needs to be created in CAN to encapsulate TinyOS messages and route them appropriately. Consideration of CAN's 8 byte message size needs to be addressed either by limiting the size of the TinyOS messages, or by segmenting a TinyOS message into serval CAN protocol messages.

Nodes will also be responsible for autonomous communication queries over a network. These tend to be repetitive in nature. The continuous listening for requests from the root node consumes a great deal of energy. This is even more evident in monitoring static civil structures over long periods of time. For example, in the monitoring of moisture in a building, the gateway polls each node individually. Nodes sit idle listing for a request in a passive nature. A *smart* node should anticipate such polling, and an even more advanced node should not need to be asked to produce the results, results should, instead, just be expected at regular intervals by the gateway. By providing nodes with information to produce their own results, a great deal of energy can be saved. The trade off in this design comes in the reduced ability to query a node at any point in time. Specialized queries need to be deferred until the next time the node checks in with values. This also requires cluster heads to queue messages.

The hardware/software interaction for the transceivers to reduce power consumption was also investigated. A cluster star network topology was selected for network deployment. Cluster heads will communicate to each other over the fast, reliable and very robust CAN network. A clustered star network topology suggests cluster heads be externally powered to support the continuous listening required when communication is central. Using a wired protocol to connect cluster heads removes the burden of multi-hop routing to span large distances between clusters, such as those methods found in mesh networks. By removing these constraints, the end nodes have few responsibilities and therefore can have lower duty cycles.

4.2 IEEE 802.15.4

For wireless communication, IEEE 802.15.4 [oEE] was chosen for the PHY and part of the MAC layer. The IEEE 802.15.4 protocol was chosen over other standards, such as IEEE 802.11 or Bluetooth because it is targeted toward low-rate, low-power consumption and low-cost. IEEE 802.15.4 defines three network topologies: star, peer-to-peer, and the cluster tree. The hybrid network defined in this thesis closely resemble the cluster tree, as shown in Figure 4.1. The cluster heads communicate using the CAN instead of routing messages through leaf nodes.

By choosing an IEEE 802.15.4 compatible transceiver, several design issues were resolved due to the availability of IEEE 802.15.4 data sniffers used

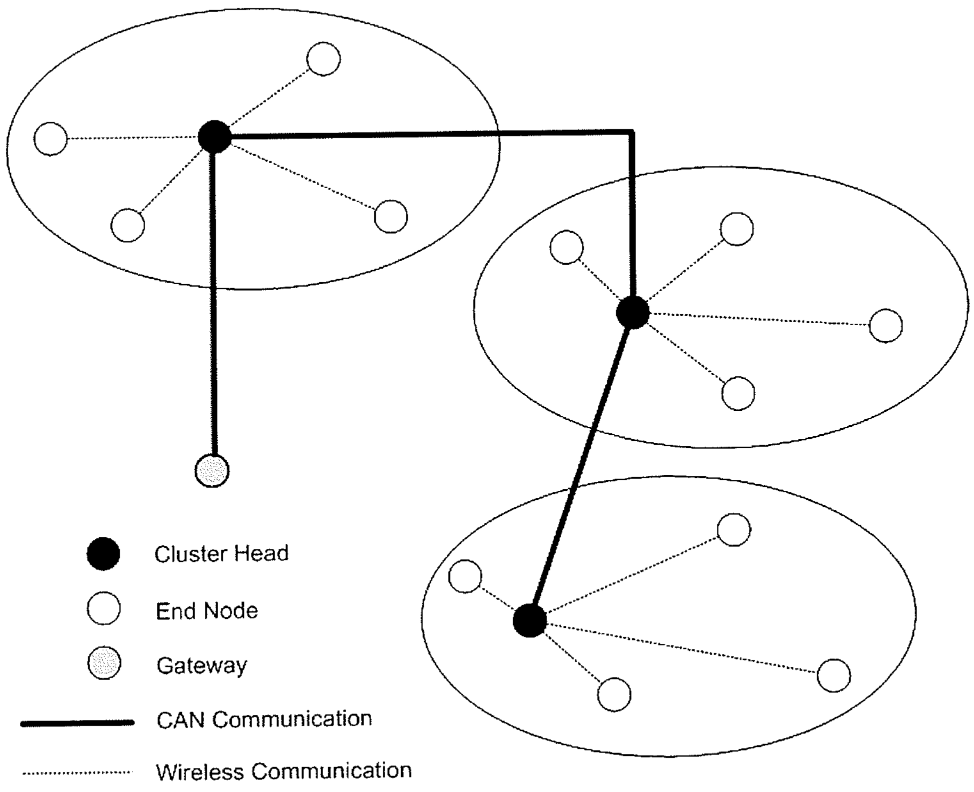


Figure 4.1: Cluster Tree Topology

for debugging network problems. Physical Layer management services include the following: activation and deactivation of the radio transceiver, Energy Detection (ED), Link Quality Indicator (LQI), and Clear Channel Assessment (CCA).

The MAC layer provided by IEEE 802.15.4 provides services such as beacon management, channel access, guaranteed time slots (GTS) management, frame validation, acknowledged frame delivery, association and disassociation, and security mechanisms. IEEE 802.15.4 can operate in two modes: beacon-enabled mode and non-beacon-enabled mode . The use of beacon-enabled mode promotes synchronized communication. Coordinators generate beacons, in a super frame structure, to manage communication. A version of slotted CSMA/CA is used. The frame is divided into 16 slots followed by a predefined inactive period inside a beacon interval as shown in Figure 4.2. The duration of the slotted communication is the Contention Access Period (CAP). General communication is restricted to the CAP by competing for a slot. A Contention Free Period (CFP) can be added after the CAP and can be used to guarantee QoS by dividing it into Guaranteed Time Slots (GTS). The inactive period allows nodes to enter sleep mode, fulfilling the need for low power.

The second mode, non-beacon-enabled, uses unslotted CSMA/CA. There is no requirement for nodes to synchronize. Nodes access the medium by evaluating energy on the channel and waiting a random back-off period before transmitting if the channel is busy. The proposed network will use a version of unslotted CSMA/CA which could be considered a reverse beacon-enabled mode. Instead of coordinators or cluster heads sending beacons, end nodes

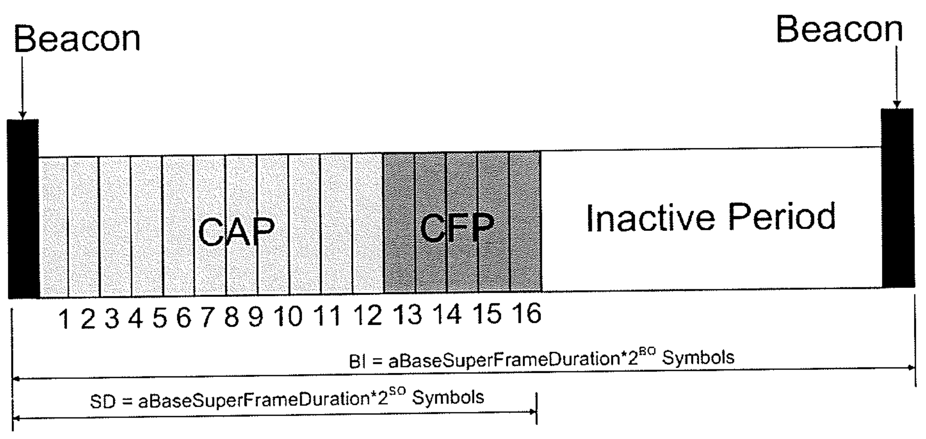

Beacon
Beacon
CAP
CFP
Inactive Period
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
BI = aBaseSuperFrameDuration*2^BO Symbols
SD = aBaseSuperFrameDuration*2^SO Symbols

Figure 4.2: Super Frame of IEEE 802.15.4 [oEE]

will issue a beacon to start communication. Beacons will be intercepted by coordinators. ACK RSSI values are used to determine which node is the closest. During this handshaking, power levels can be adjusted to the minimum acceptable value. Leaf nodes transmit their messages and obtain network variables stored by the coordinator. This process is explained more fully in Section 7.1.5 This operation only works in networks with low bandwidth and staggered communication. Otherwise, performance will be degraded by collisions. The beacon-enabled mode fits networks where synchronization is important while the non-beacon mode suits networks where power saving is more important.

4.3 Controller Area Network

A Controller Area Network (*CAN*) was chosen as the wired medium protocol. A CAN promotes efficient design due to its limited payload size. One design challenge is to relay messages from the wired medium to the wireless and vice versa. A CAN supports an 8 byte payload, therefore the application layer must structure messages to accommodate this limitation. Figure 6.1 illustrates the fields and their lengths in a CAN frame.

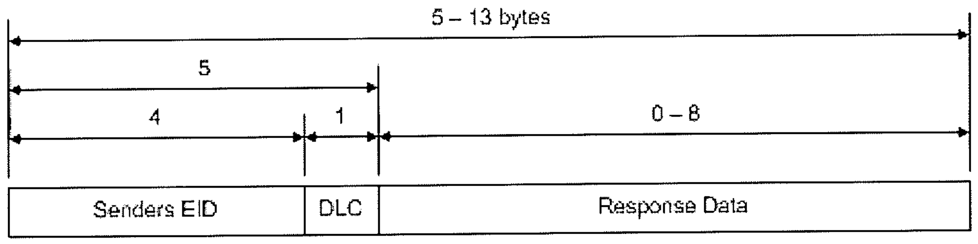


Figure 4.3: CAN Frame

The advantages of CANs over other wired network protocols are as follows:

- Fault tolerance
- Short messages, but high message frequency (more than 10,000/s)
- High bandwidth utilization
- Reasonable transmission speeds
- Several higher-layer protocols available such as DeviceNet, CANopen, and J1939
- Adjustable bit rates are configurable and can accommodate long distance communication

A CAN uses a message acknowledgment system that provides a transmitter with an acknowledgment of receipt within the message itself. One of the last bits in a CAN message is an acknowledgment bit that is not written by the transmitter. Instead, it is used by all receivers to send an indication back to the transmitter that this message was received successfully.

If any single node has a receive error, there is a mechanism that allows that node to destroy the message for every node resulting in an automatic retransmission of the message. As this is implemented in low-level hardware, every CAN node in the network participates in this error checking mechanism, even if it has no use for the message transmitted. As a result, a transmitter knows that if it received the acknowledgment, every node on the network has successfully received the message. In the event that multiple nodes try to access the network simultaneously, a bit-wise non-destructive arbitration mechanism resolves the potential conflict with no loss of data or bandwidth. During arbitration every transmitter compares the level of bit transmitted against the value monitored on the bus. Arbitration is lost when a transmitter sends a recessive bit while the value monitored on the bus is dominant. The transmitter that lost arbitration will withdraw without sending any more bits.

As compared to Ethernet, a CAN message does not specify a transmitting station or node. As a result, an Identifier Field (EID) is included in each message. In the CAN 2.0 A spec, this field is 11 bits. In the CAN 2.0 B spec, it is 29 bits. Such a message identifier has to be unique within the whole network and it defines not only the content but also the priority of the message.

4.4 Addressing

In the design implemented, the 29 bit CAN Message identifier is broken down into two fields, *Node Type* and *Node ID*. The first 5 bits will be used as the Node Type and the remaining 24 bits for the Node ID. Because the CAN protocol does not provide a destination address, only the source address (EID), the first 3 bytes of the message are reserved for the destination node ID. When relaying messages over wireless using IEEE 802.15.4, addressing changes are required. IEEE 802.15.4 supports a short addressing mode of 16 bits. For the initial design, all nodes are programmed with a 24 bit Node ID, but, while communicating wirelessly, each node will only use the 16 least significant bits of the Node ID as its address. Figure 4.4 illustrates the changing of addresses between CAN and IEEE 802.15.4 frames.

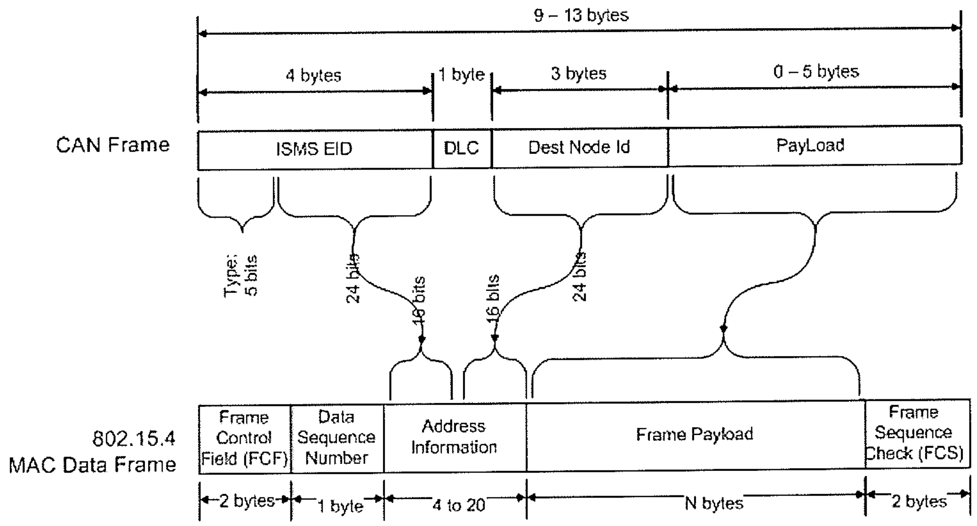


Figure 4.4: CAN-IEEE 802.15.4 Packet Transition

Messages being relayed from CAN-to-Wireless require packet restructuring. Along with the change in addressing noted above, the destination address must be extracted from the first 3 bytes of the payload and inserted into the destination address in the IEEE 802.15.4 MAC addressing fields. Figure 4.5 details the fields required for IEE 802.15.4. Similarly, Wireless-to-CAN requires the reverse packet re-organization. The destination Node ID from the wireless message must be placed in the first 3 bytes of the CAN message. Secondly, the CAN EID must be replaced with the sending Node ID's address.

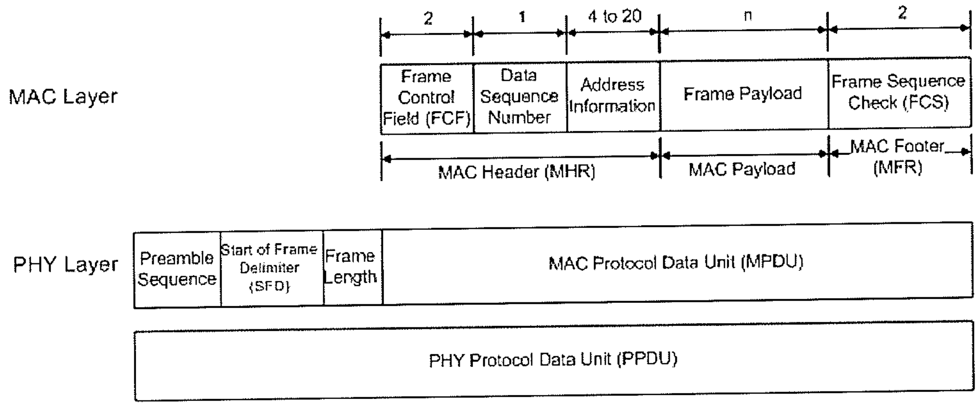


Figure 4.5: Data Frame [Chi]

4.5 TinyOS

Traditional embedded applications are bonded tightly to a particular hardware environment by integrating high-level logic with low level manipulation of bits. This is due to limited hardware resources, specialized applications, and tight development cycles. Operating systems promote development of reliable software by providing abstraction of the hardware resources. Sensor networks, although embedded, are general purpose enough to need such abstractions. TinyOS [Hil00], an operating system for sensor networks, provides an event-drvien execution model and a component-based software design that supports a high degree of concurrency for a small footprint, enhances robustness, and minimizes power consumption while facilitating implementation of relatively sophisticated protocols and algorithms. TinyOS-based applications consist of components with well-defined interfaces.

4.6 Autonomous Tasks

A majority of motes in sensor networks perform repetitive tasks. These tasks are either induced individually by the gateway or preformed autonomously by the mote. A third type of task, a combination of the above two, is injected by the root and allows the motes to provide results over time. This approach is similar to stream queries implemented in systems such as tinyDB [MFHH05]. Stream queries have an additional component, a window, which allows aggregation over a partition of the dataset. The Motes described in this thesis are seeded with an initial query to provide data back to the gateway. The complexity of data aggregation is left to the gateway or data collection center. Parameters for the query can be modified at the time results are pushed forward. When a wireless mote initiates communication with a gateway node to push its results, it listens for a small time period to become informed of any updates. This approach not only simplifies communication, but also satisfies the requirement for extremely low power consumption. First, by seeding repetitive tasks in motes, such as a streaming query, motes are not required to listen to individually injected singleton tasks. Secondly, by having motes retrieve query parameters while pushing data towards the gateway, motes are not required to synchronize with beacons to receive messages.

This chapter discussed the overall design of the proposed heterogeneous network. The following section will present the implementation of the system.

Chapter 5

Implementation

The implementation of this thesis included developing protocols, software and hardware for a heterogeneous network. Section 5.1 presents the hardware and interfacing required to build the first wireless/CAN prototype. Finally, Section 5.2 discusses the modules created in TinyOS to support the heterogeneous network.

5.1 Node Hardware

The Remote Measurement Unit (RMU) developed by SMT Research was chosen as the base platform. It includes a PIC18F MCU [Incb], that includes a CAN controller and circuitry to provide moisture measurements. Modifications to this base platform were done to support a radio transceiver and battery operation.

Figure 5.1 shows a block diagram of the hardware for the proposed RMU redesign. All the necessary sensors are included for structural health moni-

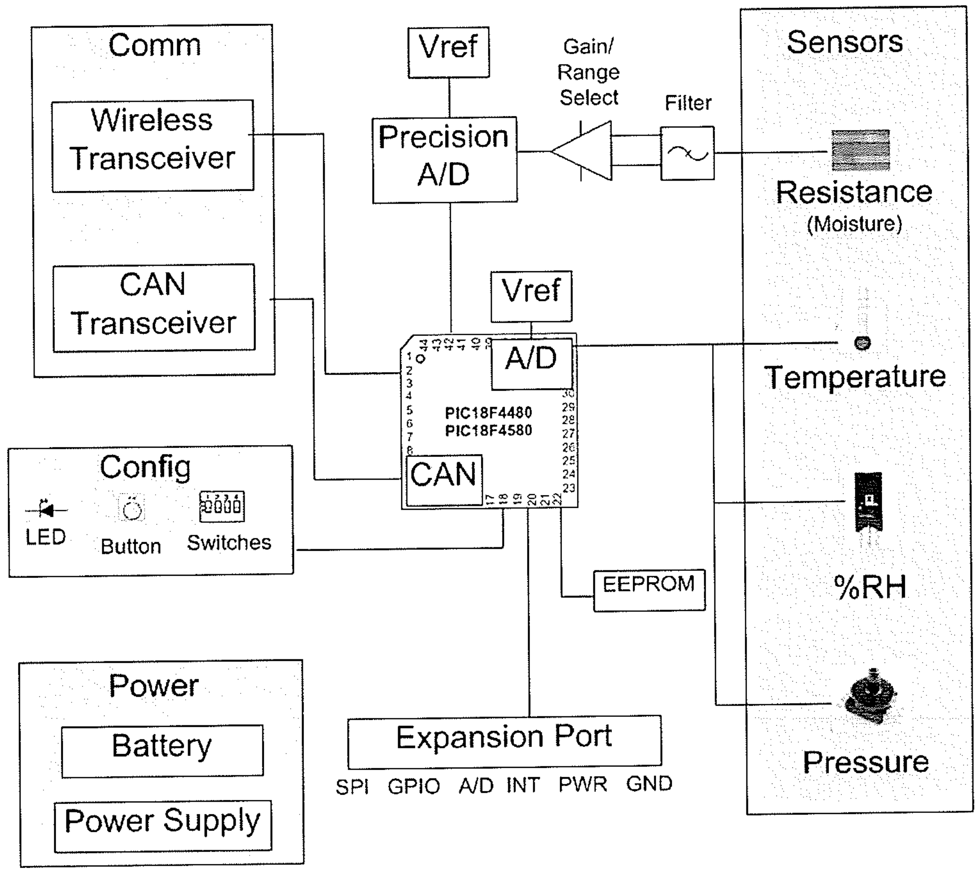


Figure 5.1: Hardware Block Diagram

toring as well as components for communication. The redesigned hardware will have the ability to operate in either cluster head or autonomous modes. While operating in cluster head mode, the RMU will communicate over the CAN and, as a result, have access to an external power source.

One of the challenges is to interface the RMU with a Chipcon CC2420 [Chi] daughter board to support wireless communication. Figure 5.2 shows the general approach to interfacing the transceiver to a processor. In the figure the *FIFO* data is connected to a general-purpose input/output(*GIO*) pin, *FIFOP*, the FIFO threshold interrupt, is connected to an interrupt pin, the Clear Channel Assessment (*CCA*) is connected to a second *GIO* pin, the Start of Frame Delimiter (*SFD*) is connected to a pin that supports a timer capture, the CC2420 Chip Select (CS_n) is connected to a *GIO* pin, SPI Slave Input (*SI*) is connected to Master SPI Output (*MOSI*), SPI Slave output (*SO*) is connected to Master SPI Input (*MISO*), and the SPI Clock Input (*SCLK*) is connected to the microprocessor SPI Clock output (*SCLK*). The design varied slightly because some *GIO* and interrupt pins were already in use. SFD is tied to an interrupt instead of timer capture and therefore must be polled in software. Another method evaluated was to interrupt on the *FIFO* pin instead of the *FIFOP* pin. The general practice to interrupt on the *FIFOP* pin was followed which ensures that data in the *FIFO* passes address recognition. Figure 5.3 shows the first prototype developed.

5.2 TinyOS

The prototype developed included TinyOS running on the PIC18F4680 MCU [KWSH05] as the base for the RMU platform code. Additional modules were required for the CAN controller, and CC2420. Figure 5.4 shows a visual representation of the module graph for the Wireless Structure Monitoring System (WISMS). Not shown in the application module graph are the hardware modules created. The standard `GenericComm` module was modified to support packets being sent and received over the CAN. The WISMS application operates in one of two modes: cluster head or autonomous agent.

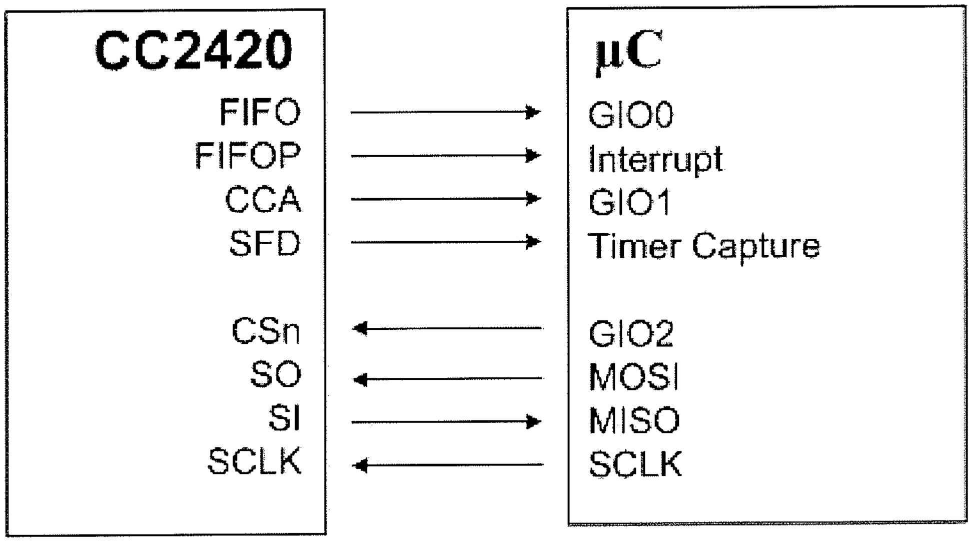


Figure 5.2: CC2420 Microprocessor Interface

While acting as a cluster head, the mote continuously listens for messages over the radio. Once a message is received via the `WISMSMessageHandler` module, the mote converts the radio packet to a CAN packet and forwards it to the gateway device. A mote operating in the autonomous agent mode uses the `AutonomousTimer` module to schedule wake up periods. When the `AutonomousTimer` expires, the mote executes the `CMDMeasure` command and forwards the result to the cluster head. The `WISMSMessageHandler` module can service requests over both the CAN and the Radio. The received message is processed against the `WISMSCommandC` command factory pattern module.

5.2.1 WISMS Command Configuration

Figure 5.5 illustrates the *WISMSCommand* configuration used for the implementation of commands. All commands implement a standard interface,

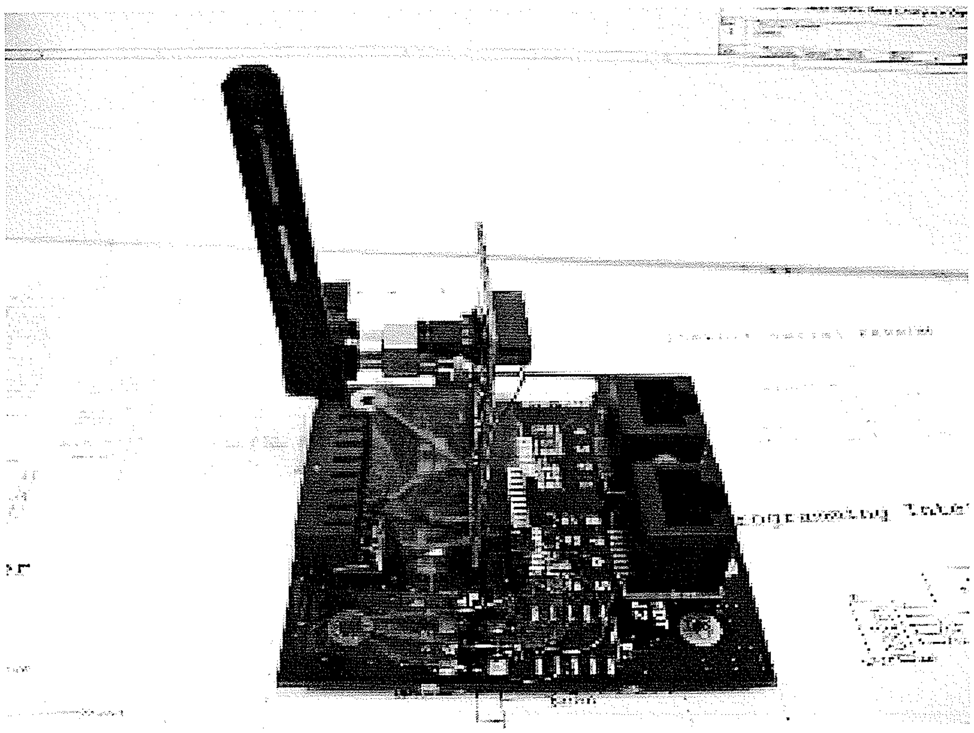


Figure 5.3: First Prototype of the Wireless RMU

WISMSCommand, which enforces structured command execution. Commands implemented include `CMDGetID`, `CMDChangeID`, *CMDAutonomous*, *CMDRest*, and `CMDMeasure`. Figure 5.5 shows the wiring of additional modules used to complete a command, such as a *Timer* for *CMDMeasure.*

5.2.2 CC2420 Component

Figure 5.6 shows the configuration for the CC2420 component. This configuration is responsible for "wiring" together many components including the *CC2420Control* module, *CC2420Radio* module, *HPLCC2420* module, and

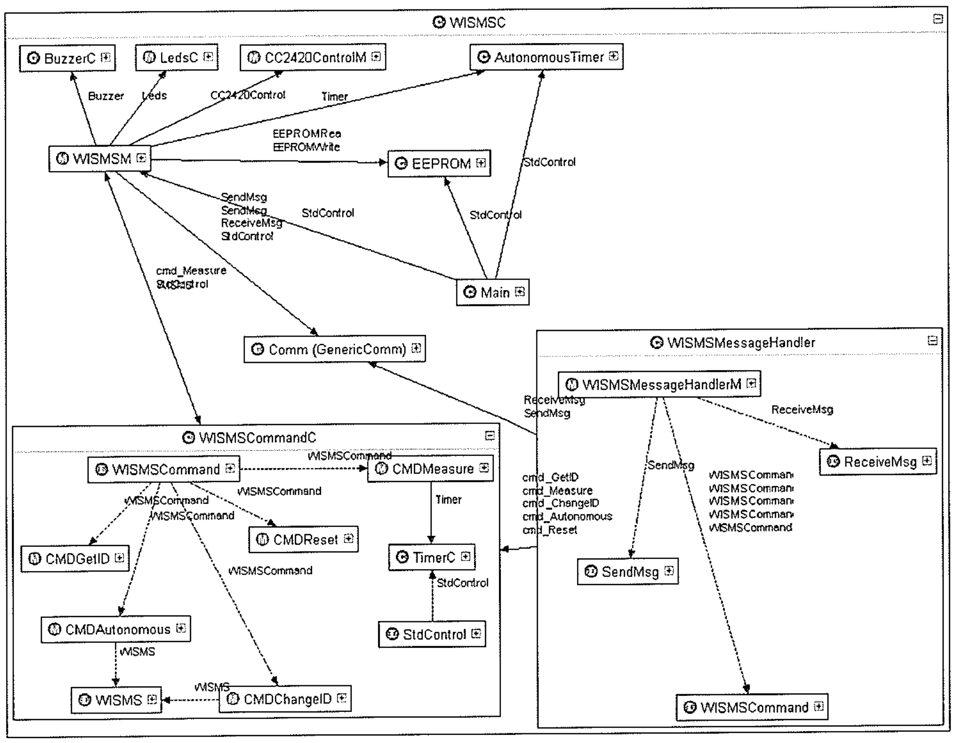


Figure 5.4: TinyOS component graph for the WISMS application

the *PIC18F2420Interrupt* module to build the WISMS application. This configuration provides interfaces for *ReceiveMsg* interface and *BareSendMsg* interface, making this configuration compatible with the *GenericComm* configuration. The *CC2420Control* module is responsible for configuration of the CC2420 including functions to set the channel and ID of the mote. The *HPLCC2420* module is responsible for implementing access to the data FIFO, low level access to the registers, and sending and receiving messages. Successful reception of a messages is signaled from the *PIC18F2420Interrupt* module.

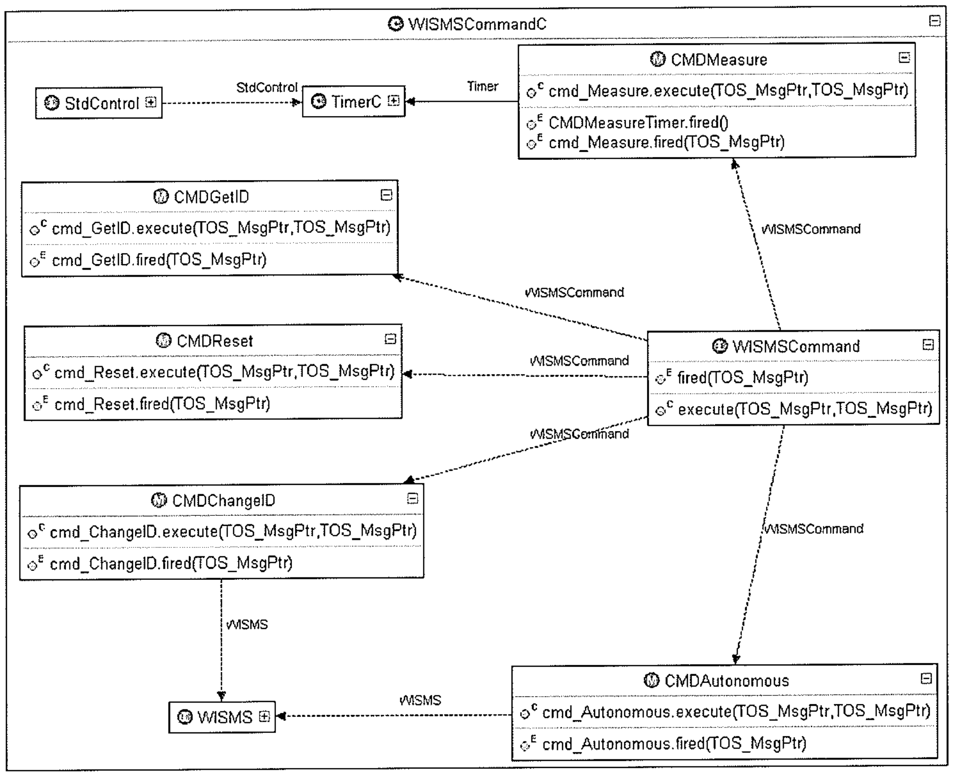

WISMSCommandC
StdControl
StdControl
TimerC
Timer
CMDMeasure
cmd_Measure.execute(TOS_MsgPtr,TOS_MsgPtr)
CMDMeasureTimer.fired()
cmd_Measure.fired(TOS_MsgPtr)
CMDGetID
cmd_GetID.execute(TOS_MsgPtr,TOS_MsgPtr)
cmd_GetID.fired(TOS_MsgPtr)
WISMSCommand
CMDReset
cmd_Reset.execute(TOS_MsgPtr,TOS_MsgPtr)
cmd_Reset.fired(TOS_MsgPtr)
WISMSCommand
fired(TOS_MsgPtr)
execute(TOS_MsgPtr,TOS_MsgPtr)
CMDChangeID
cmd_ChangeID.execute(TOS_MsgPtr,TOS_MsgPtr)
cmd_ChangeID.fired(TOS_MsgPtr)
WISMS
WISMS
CMDAutonomous
cmd_Autonomous.execute(TOS_MsgPtr,TOS_MsgPtr)
cmd_Autonomous.fired(TOS_MsgPtr)

Figure 5.5: WISMS command pattern

5.2.3 The CAN Component

Figure 5.7 show the configuration for the *CAN* component introduced into TinyOS. Analogous to the *CC2420Radio* configuration, the *CAN* configuration provides the *ReceiveMsg* and *BareSendMsg* making it compatible with *GenericComm.* Low level access to the CAN controller is handled by the *HPLCAN* module. This module is responsible for building TOS_msg structures from frames over the CAN bus by using the *PIC18F4620Interrupt* module to receive notification when frames have been received. Similarly, the

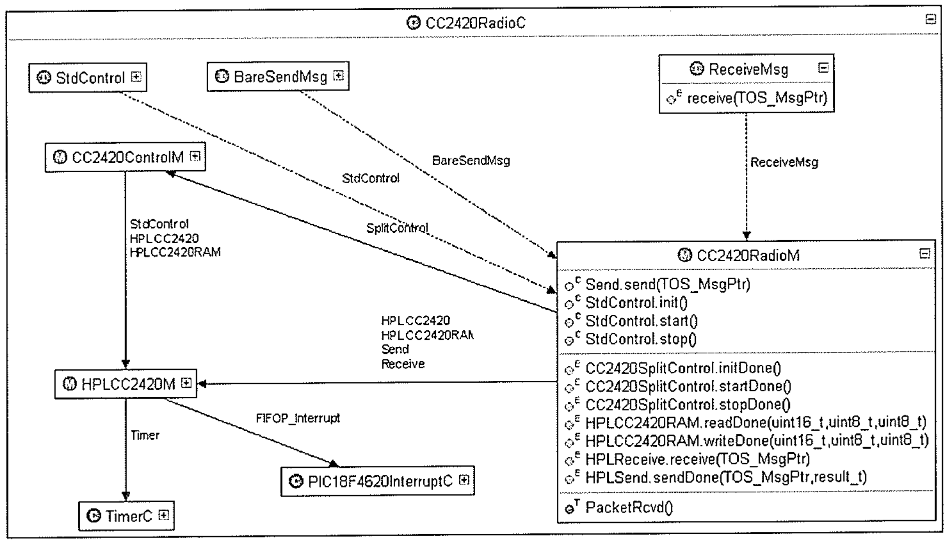


Figure 5.6: CC2420 Radio Configuration

HPLCAN module creates CAN frames from TOS_msg structures received by the *BareSendMsg* interface.

5.2.4 GenericComm Configuration

GenericComm is used in most cases to encapsulate the TinyOS network stack as shown in Figure 5.8. *GenericComm* provides *SendMsg* interface and *ReceiveMsg* interface by wiring a parameterized interface to the module. By parameterization of these interfaces, many different components can access a communication medium at the same time. Applications and the network stack exchange message buffers through pointers. Modules wishing to send data provide their own storage buffer while receiving components consume the buffer passed by the radio stack. *GenericComm* provides limited receive

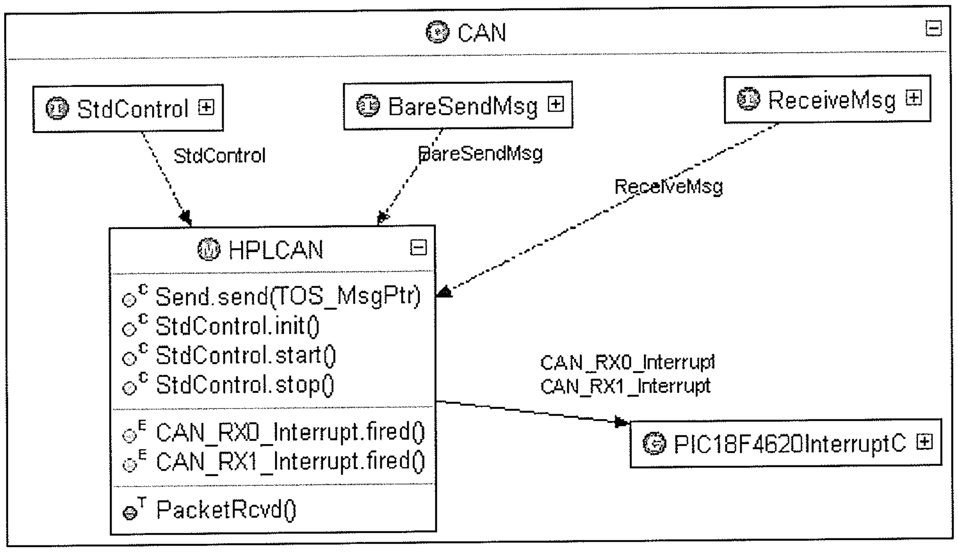


Figure 5.7: CAN Configuration

buffering. Typically, after consuming a buffer a component will pass it back to the radio stack, however, if the modules wishes to store that buffer it may instead return a pointer to another free buffer. For components to accumulate several packets before processing, the buffers must be statically allocated and locally stored, rather than relying on the network stack to provide a level of buffering.

5.2.5 TinyOS and the PIC Architecture

To compile machine code for the PIC architecture some additional steps were required than what would not be needed for platforms directly supported by TinyOS, such as the MSP430 and the Atmel processors.

NesC compiles all the modules required for an application into a single C

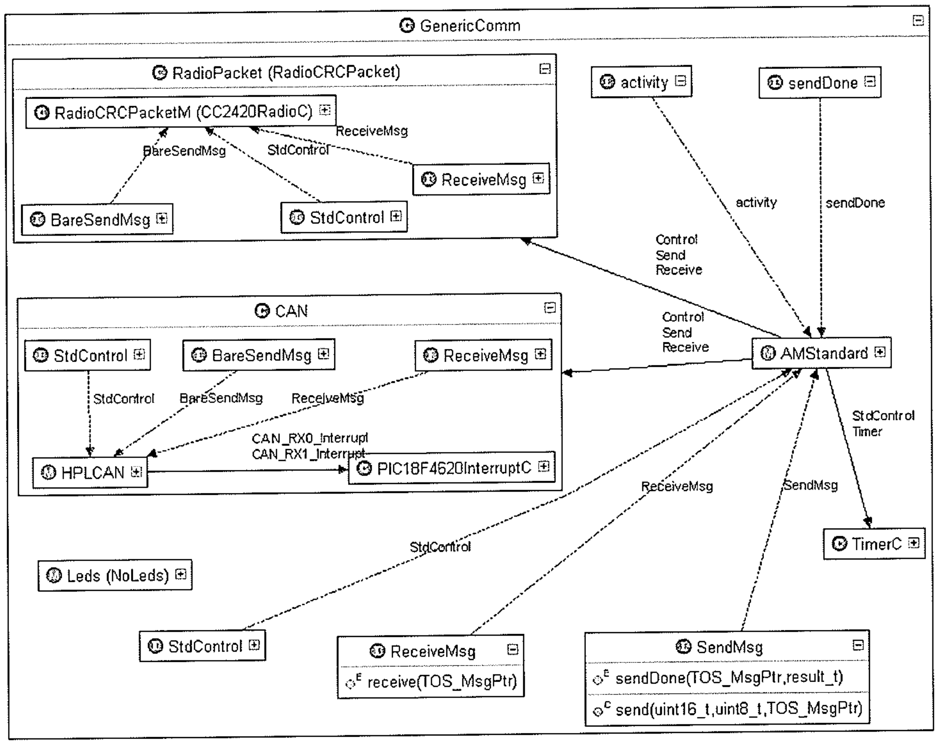


Figure 5.8: GenericComm Configuration

file. To allow private function calls and data, all function calls and variables inside the module are converted into fully qualified names starting with the module name. Because nesC follows ANSI C, a majority of the nesC code produced does not need to be changed.

The C file produced by nesC, however, is not compatible with the Microchip C18 [Inca] compiler. To solve this problem the output is filtered with a perl script to produce compatible code. A custom linker file was also created to allocate code and data in memory efficiently. The complexity of the application resulted in code that would not fit in the default sections created by the linker

Sections are portions of an application located at specific memory addresses. Sections can be located in either program or data memory. By default there are two types of sections for each type of memory. The program memory contains the *code* section, which holds the executable content, and *romdata* section, which holds the data in program memory. The data memory contains the *udata* section, which holds statically allocated uninitialized user variables, and the *idata* section, which contains the statically allocated initialized user variables.

The *pragma* language extension was used to change the section type. The perl script used to create code compatible with the C18 compile uses the *pragma* directive to allocated static data to a large portion of data memory defined in the custom linker file.

The PIC architecture is restricted to a pre-compiled stack. This means static resources are allocated for each function using a call graph at compile time. Because storage space for the function arguments is static, calling a function overwrites these arguments. This results in the lack of ability to allow recursive function calls and functions that are re-entrant. This restriction must be kept in mind while creating modules.

This chapter presented the hardware and software implementation for a heterogeneous sensor network.

Chapter 6

CAN Node Communication Protocol

This chapter describes the protocol used for communication between the gateway and various node types attached to a CAN interface. A CAN bus is used for the physical layer of this communication protocol. Some familiarity with CANs is assumed in this section. This section will focus on the higher level protocol layered on top of the CAN protocol for gateway/node communication, and will only discuss CAN details where they impact our protocol.

6.1 General Packet Format

Various fields of a CAN frame are used in this protocol, as shown in Table 6.1.

For our purposes, the EID field contains the identifier of the message

Table 6.1: CAN Frame Fields

CAN Field	Description
EID	A unique message identifier of the sending node. (Extended format is used containing 29 bits)
DLC	Data length code: the number of bytes in the data segment of the message. (4 bits)
Data	Arbitrary payload data sent with the message, of size determined by the DLC field.(0-8 bytes)

sender. The 29-bit EID is comprised of two subfields; a type field (upper 5 bits) and a node identifier field (lower 24 bits). Every CAN node, regardless of type, has a globally unique 24-bit node identifier (i.e. no two nodes will ever have the same node identifier across all node types), allowing for over 16 million nodes. Since the EID contains the sender's address, for gateway to node communications, the destination node identifier needs to be encoded in the data field of every packet. Only 24 bits are used for the node identifier so that messages sent from the gateway to a node, which must contain the destination of the message in the data field, uses only 3 of the available 8 data bytes.

The upper 5 bits of the EID field are used to determine the type of device sending the message, with all ones (0x1F) being reserved to indicate a communications protocol handling error. This allows for up to 31 different types of nodes/devices to be developed for attachment to the CAN interface.

Table 6.2: CAN Type and Description

Type	Description
0x00	Gateway
0x01	CAN Node
0x02..0x1E	Undefined - reserved for future use
0x1F	Error Packet

Currently defined node types are listed in Table 6.2.

Standard communication between the gateway and CAN nodes is initiated by the gateway in a master-slave relationship. The gateway to node communication is not type specific, but node to gateway communication is type qualified by the type field of the responding node reflecting the type of that node. As such, the gateway must manage node specific semantics in gateway to node communication by node identifier.

The CAN communication packets are defined as an integral package of 5 to 13 bytes, depending on the response data length, as per Figure 6.1.

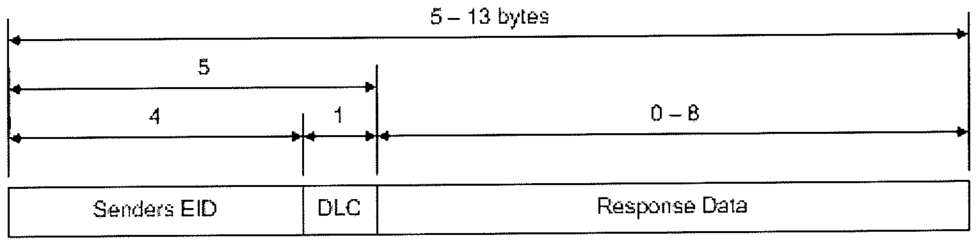


Figure 6.1: CAN Frame

Table 6.3 details the terms used in this section and their meanings. The

Table 6.3: CAN Definitions

Term	Description
EID	The 29-bit EID not including the SRR, IDE, and RTR bits, right justified in a 32-bit field.
CAN EID	The full 32-bit CAN extended ID presented on the wire on the CAN Bus, including SRR, IDE, and RTR bits. This is the EID transferred to/from the CAN Controller.
Node Id	The 24-bit node identifier of a node
Node Type	The 5-bit type identifier of a node

CAN controller is responsible for inserting the CAN SRR, IDE, and RTR bits into the EID as well using the least significant bits of the DLC for managing the data field. Conversely, the CAN controllers are responsible for removing those bits. The EID discussed throughout this document referrs to the CAN EID.

The gateway always uses an EID of 0x00000000 (node type = 0x00, node id = 0x000000). Since a transmission with an EID=0 has a higher priority on a CAN bus than an EID=1, this ensures that the gateway will always win arbitration if there is a collision on the CAN bus.

6.1.1 Byte Ordering

All multi-byte values are stored in the communications packets in big-endian representation (i.e. most significant byte first).

6.1.2 Gateway Command Packet (Gateway to Node)

A gateway to node transmission reserves the first four bytes of the data field for a target node identifier and a command identifier. The entire data field is defined in Figure 6.2

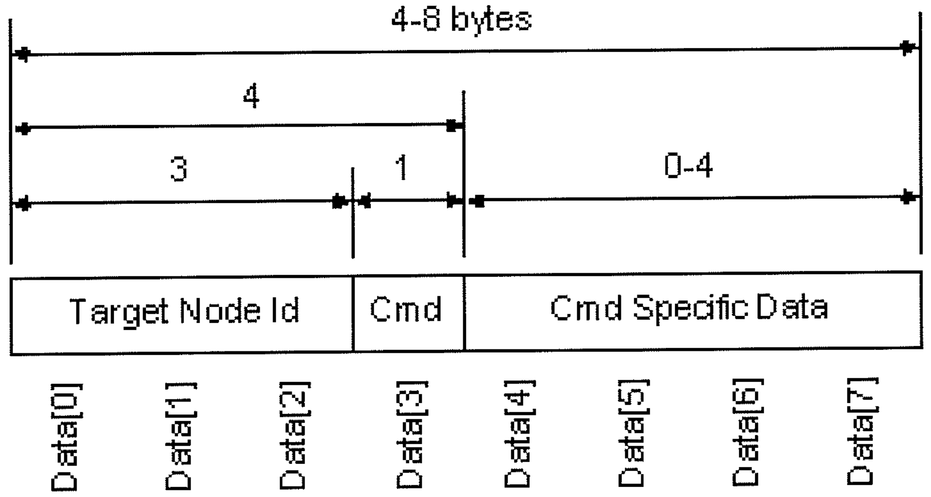


Figure 6.2: Gateway Command CAN Data Field

The Cmd field is a command identifier which is an unsigned 8-bit value that determines the action to be taken by the node based on this transmission. The Target Node Id field is the unique node identifier of the node, or the value 0xFFFFFF if the message is intended for all nodes (i.e. a broadcast message) as shown in Figure 6.4.

Complete packet structure is shown in Figures 6.3 and 6.4, where normal and broadcast packets are shown respectively.

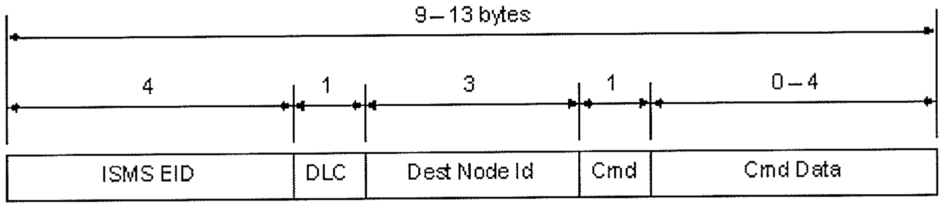


Figure 6.3: Gateway Command Packet

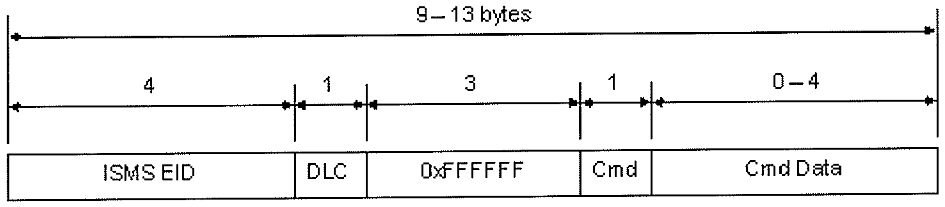


Figure 6.4: Gateway Broadcast Packet

6.1.3 Node Packet Reception Selection

A CAN node will use CAN receive filters (hardware filtering) for the following EIDs, translated appropriately to CAN EIDs:

- 0x00000000 (Regular packet from gateway)
- 0x1F000000 (Error packet from gateway)

This will result in a node only receiving gateway traffic from the CAN bus. A node's firmware does further packet selection based on the destination node identifier field (Data[0:2] from the data portion of the packet, selecting on the actual nodes node identifier or the node identifier of 0xFFFFFF).

A node will not accept, or respond to a broadcast packet for a command that does not have a defined behaviour for broadcast. Conversely, a node will respond with an error packet to command identifiers that are not defined for the specific node's type when addressed specifically to that node.

6.1.4 Node Response Packet (Node to Gateway)

In a response packet sent from a node to the gateway, the entire data field contains the node's response (0-8 bytes). For a node to gateway transmission which contains a data response, the first byte of the data field is reserved for a result code.

Note that the response packet does not indicate which command it is in response to. The gateway, on a per-node basis, will always wait for the response to one command, or timeout before sending another command to that node. The gateway may, however, send commands to other nodes while it is waiting for a response from a specific node. If the gateway is communicating with many nodes concurrently, then it must take precautions to wait for all outstanding communications to complete before commencing a broadcast transmission. Conversely, the gateway must wait for the actions of a broadcast transmission to be completed before commencing with per-node communications again.

The gateway does not use any CAN receive filters allowing it to receive traffic from all nodes. The gateway identifies the responding node from the EID field of the packet.

The gateway terminates the waiting-for-response state of a command-response transaction with a node upon the following conditions shown in Table 6.4.

For commands that do not require a response, the transaction is consid-

Table 6.4: CAN Transactions

Normal Response	EID = (Node type, Node id), (0 <= DLC <= 8) A normal response from the node, including responses with an unsuccessful result code.
Timeout	A programmed derived time period has elapsed since the command packet was sent. Responses that arrive after the timeout period will be discarded by the gateway.
Error Packet	EID = (Node type = 0x1F, Node id), DLC=0 An error does not include any data in the data portion of the packet.

ered complete once the packet has been sent.

6.1.5 Node Autonomous Packet (Node to Gateway)

Nodes will periodically provide data to the gateway that was not requested in the master/slave configuration. To distinguish such packets from requested packets, nodes change the type field in the EID to 0x02. The gateway filters autonomous packets so they do not appear in the input queue for requested commands. This alloww both types of communication to operate simultaneously. The gateway will need to know how to process autonomous packets. Autonomous packets will include a CMD, or command, field as the first byte in the data section, as shown in Figure 6.5.

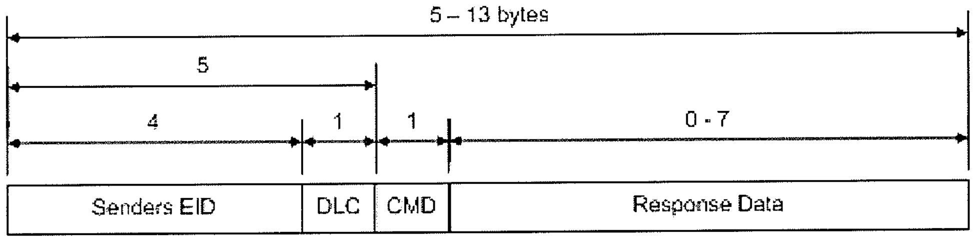


Figure 6.5: Autonomous CAN Packet

6.1.6 Node Error handling

A node will respond with an error packet for general errors in the communication protocol for packets addressed for a specific node. Protocol errors for broadcast packets are silently ignored to avoid an error storm on the CAN bus.

Generally, an error packet will be sent to the gateway from a node when one of these errors is detected at the node:

- DLC field incorrect for command (this will catch DLC > 8)
- command not defined for node

During command processing a node may detect additional errors. These errors are reported back to the gateway as a result code in a normal response packet as opposed to an error packet. For responses which normally include response data, dummy data values with a bit pattern of all 1s, will be populated in the response to maintain the defined packet size of the command response packet (e.g. 0xFF for a byte field).

For commands that provide response data (i.e. response packets with a DLC > 0), a result code is supplied in the first data byte (i.e. Data[0]) of the response data field. Result codes are defined across all node types. Responses to specific commands will use a subset of these result codes as defined by the command. Result codes are returned for commands regardless of the command addressing method (direct or broadcast) to the node. A list of result command codes is shown in Table 6.5

6.2 CAN Commands/Responses

This section presents each of the commands that the Gateway can send to any type of CAN node, as well as the format of the response expected from the node, if any. It is expected that all CAN nodes, regardless of type, will respond to each of these commands.

Table 6.6 contains a list of Master/Slave CAN Commands implemented in this project

6.3 Autonomous Node Response

Table 6.7 lists the autonomous commands that a gateway will process from any type of CAN node

6.4 Summary

This chapter presented the message protocol used on the CAN bus. In particular, addressing for sensor networks, message types, and message commands have been defined.

Table 6.5: CAN Normal Packet Response Result Code

Hex	Decimal (signed)	Result Code Definition
0x00	0	Success
0xFF	-1	Error, invalid command data. This error is used to respond to a situation where data for a command is outside of the range of acceptable values as specified in the command packet.
0xFE	-2	Error, feature not present. This error is used to respond to a situation where a command is directed towards an unimplemented feature of a node. E.g. Attempt to read External RAM, when no external RAM is present.
0xFD	-3	Error, non specific error. This error is used to respond to a situation where no other error code is appropriate.
0xFC	-4	Error, power not on RMU. Specific to command 0x09. This error is used to indicate that the selected external power line is not on.
0xFB	-5	Error, Non existent address. Only specific to command 0xFE and 0xFD. This error is used to respond to a request to operate on a memory location that does not exist in the node.
0xFA	-6	Error, CRC mismatch Only specific to command 0x0F (RU Program Image). This error is used to respond to a request to program an image from the external EEPROM that does not match the CRC included with the command.
0xF9	-7	Undefined
0x01		Reserved for future use.

Table 6.6: CAN Commands

Command	Command Name Name	Command DLC
0x01	Identify	4
0x02	Get Reading	6
0x03	Set Autonomous Mode	5
0x04	Get Version	4
0xFC	Set Node ID	7

Table 6.7: Autonomous CAN Commands

Command	Command Name Name	Command DLC
0x01	Autonomous Resistance Reading	8
0x02	Autonomous Voltage Reading	8
0x03	Battery Power	6
0x04	Routing Node	8

Chapter 7

Wireless Node Communication Protocol

This chapter describes the protocol used for communication between the cluster heads and various node types communicating over a wireless medium. IEEE 802.15.4 is used for the physical layer of this communication protocol. Nodes featuring only an 802.15.4 transceiver for communication are referred to as wireless nodes. Nodes containing an 802.15.4 and CAN transceiver are referred to as cluster heads because they will route messages from wireless nodes onto the CAN bus for the gateway to process. Communication between wireless nodes and cluster heads is limited to the IEEE 802.15.4 beacon interval.

Table 7.1: 802.15.4 Frame Fields

Frame Length	Frame length
FCF	Frame Control Field
DSN	Data Sequence Number
Addr. Info	Address Information
Payload	Frame Payload
FCS	Frame Sequence Check

7.1 General Packet Format

The fields of an 802.15.4 frame, shown in Table 7.1, are used in this protocol. Because packets from the radio are redirected over the CAN bus, payload sizes between CAN packets and 802.15.4 packets must be compatible. To accommodate this constraint wireless packets are limited to a payload of 8 bytes or less.

Figure 7.1 show the structure of a 802.15.4 frame.

7.1.1 Byte Ordering

All multi-byte values are stored in the communications packets in big-endian representation (i.e. most significant byte first).

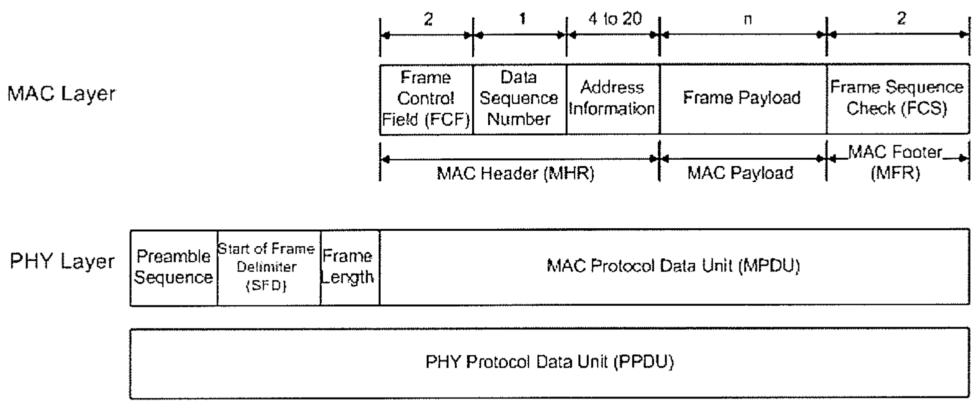


Figure 7.1: 802.15.4 Frame [Chi]

7.1.2 Wireless Node Packet (Wireless Node to Cluster Head)

A gateway to node transmission reserves the first four bytes of the data field for a target node identifier and a command identifier. The entire data field is defined as follows:

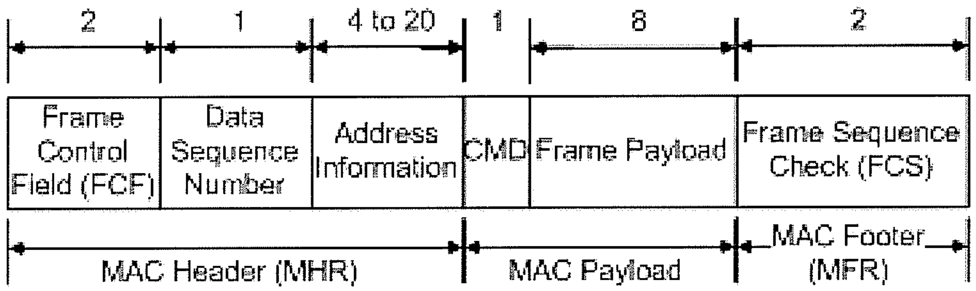


Figure 7.2: Autonomous Wireless Packet

The CMD field is a command identifier which is an unsigned 8-bit value that determines the action to be taken by the cluster head based on this transmission.

7.1.3 Cluster Head Packet Reception Selection

A cluster head node continuously listens to traffic on the radio. The address decoder and software filters select messages in the same personal area network (PAN) and addresses either to:

- 0xFF (Broadcast packet), or
- the cluster head address

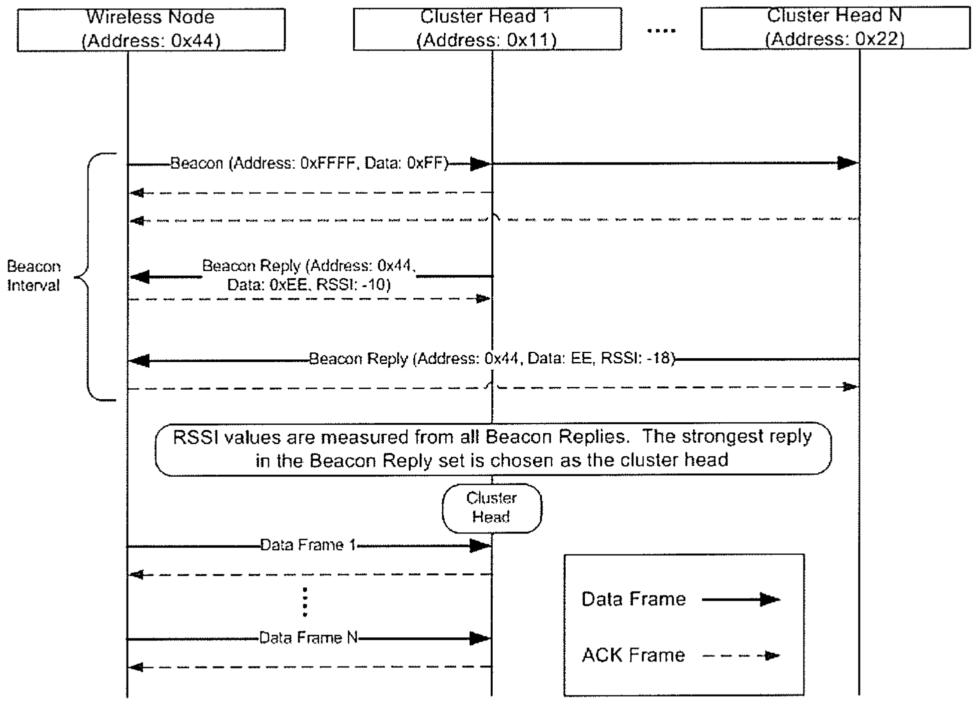
This will result in a cluster head only receiving traffic from nodes wishing to filter messages through it.

7.1.4 Wireless Node Packet Reception Selection

To conserve energy, wireless nodes do not listen passively on the radio unless they are in their beacon interval. During the beacon interval, wireless nodes will turn their radios on and only communicate to the cluster head. The only exception is during cluster head selection, where the wireless node listens for the beacon response.

7.1.5 Wireless Beacon

To initiate communication between cluster heads and wireless nodes, a beacon frame is sent from the wireless node to the cluster head. Figure 7.3 shows the communication during the beacon interval. The beacon frame contains the command 0xFF in data[0] and is broadcast across the channel. Cluster heads within the range of the wireless node respond to the beacon with the beacon reply frame containing the command 0xEE in data[0]. The wireless node waits until all cluster heads have had a chance to respond and continues communication with the cluster head with the strongest RSSI value.



Wireless Node (Address: 0x44)
Cluster Head 1 (Address: 0x11)
....
Cluster Head N (Address: 0x22)
Beacon (Address: 0xFFFF, Data: 0xFF)
Beacon Interval
Beacon Reply (Address: 0x44, Data: 0xEE, RSSI: -10)
Beacon Reply (Address: 0x44, Data: EE, RSSI: -18)
RSSI values are measured from all Beacon Replies. The strongest reply in the Beacon Reply set is chosen as the cluster head
Cluster Head
Data Frame 1
Data Frame N
Data Frame
ACK Frame

Figure 7.3: Beacon Interval Communication

7.1.6 Wireless Node-to-Cluster Head Commands/Node Responses

This section presents each of the commands that a wireless node can send to a cluster head, as well as the data length code (DLC). Table 7.2 contains the

Table 7.2: Wireless Commands

Command	Command Name Name	Command DLC
0x01	Autonomous Resistance Reading	8
0x02	Autonomous Voltage Reading	8
0x03	Battery Power	6
0x04	Routing Node	8
0xEE	Beacon Response	1
0xFF	Beacon	1

commands used in wireless communication. All commands sent to the cluster head are autonomous in origin, the only exception being beacon requests.

7.2 Summary

This chapter presented the message protocol used over IEEE 802.15.4. In particular, addressing for sensor networks, message types, and message commands have been defined. The distinction between cluster heads and autonomous nodes has been defined in terms of functionality and the data exchange protocol was shown in a message sequence diagram. Communication between autonomous wireless nodes and cluster heads is limited to the beacon interval initiated by the an autonomous wireless node to conserve energy by limiting idle listening.

Chapter 8

Results

All of the problems outlined in this thesis have been addressed in the design and implementation. This section summaries the work done and how it solved the identified problems. First, Section 8.1 describes the modifications made to IEEE 802.15.4 to suit long sensing lifetimes. Section 8.2 describes the benefits of using a CAN integrated with a wireless sensor network for structural health monitoring. Section 8.3 presents the results from analyzing co-existence with IEEE 802.11 networks and the importance of channel selection. Section 8.4 presents the tests conducted to verify the correctness of system operation and its robustness. Finally, Section 8.5 presents the results of a simulation used to predict the sensor lifetime under different scenarios.

8.1 Modified IEEE 802.15.4

This thesis uses the PHY layer and many of the services provided by the MAC layer in the IEEE 802.15.4 protocol. The use of these components helped develop a robust protocol that is better suited for extremely long term monitoring. By using the IEEE 802.15.4 protocol for wireless communication, issues such as channel access and scheduling were resolved. A higher level application protocol used the services provided by the IEEE 802.15.4 transceiver to allow for the beacon interval to be initiated at the wireless end node.

8.2 CAN

CAN was introduced in this thesis to relieve the burden of multi-hop routing by routing messages back to the gateway over a fast and reliable wired medium. The addition of the CAN network improved the connectivity of the systems and simplified the routing of messages.

8.3 Channel Selection

To maximize capacity and throughput, selection of appropriate channels is necessary. Proper channel selection will improve the coexistence with other devices in the 2.4GHz spectrum. One of the biggest contenders in buildings would be IEEE 802.11 networks. RF channels for IEEE 802.15.4 overlap channels in IEEE 802.11. Coexistence between these protocol was studied in [CW].

IEEE 802.11 is broken down into 11 channels in North America and 13 in Europe. These channels are staggered with center frequencies 5MHz apart with an overall channel bandwidth of 22 MHz. The channels are identical for

IEEE 802.11b and g, the differences lie in the modulation scheme. Channels are spread spectrum meaning they do not occupy a single channel, but operate over a band of frequencies within the 22 MHz channel. When selecting a channel you are actually selecting the center frequency. These channels, along with lower, center, and upper frequencies are shown in Table 8.1.

The IEEE 802.15.4 radio, similarly to IEEE 802.11b, uses direct-sequence spread spectrum coding which results in an overall bit rate of 250 kbits/s per channel in the 2.4GHz spectrum. There are 16 channels, each with a center channel separation of 5 MHz and a frequency range of 3 MHz. Unlike IEEE 802.11, IEEE 802.15.4 uses non-overlapping channels.

Co-channel interference is an important issue to consider to ensure that each wireless service maintains its desired performance requirements. Referring to table 8.1, it should be noted thatIEEE 802.15.4 has two channels, 25 and 26, that are outside the frequency range of IEEE 802.11. Visually, this can be seen in Figure 8.1 where the IEEE 802.15.4 channels are shown against the three most common IEEE 802.11 channels. Channels 1, 6, and 11 are used in typical IEEE 802.11 deployments because they are the only three non-overlapping independent channels in the North American domain. As a rule of thumb for IEEE 802.15.4 deployments, channels 25 and 26 should be chosen. If additional channels are required, channels 15 and 20 can be used, but care must be used to ensure that an 802.11 network is not deployed in the same environment using non-standard and overlapping channels.

During testing, one cluster head and one wireless autonomous node were setup to transmit messages every second for a duration of two minutes. Different channels were selected for IEEE 802.15.4 to test the effects of coex-

Table 8.1: IEEE 802.15.4 and IEEE 802.11 channels within the 2.4GHz ISM Band

IEEE 802.11		IEEE 802.15.4	
Channel	Freq. (GHz)	Channel	Freq. (GHz)
1	2.401-(2.412)-2.423	11	2.405
2	2.404-(2.417)–2.428	12	2.410
3	2.411-(2.422)–2.433	13	2.415
4	2.416-(2.427)-2.438	14	2.420
5	2.421-(2.432)-2.443	15	2.425
6	2.426-(2.437)-2.448	16	2.430
7	2.431-(2.442)-2.453	17	2.435
8	2.436-(2.447)-2.458	18	2.440
9	2.441-(2.452)-2.463	19	2.445
10	2.446-(2.457)-2.468	20	2.450
11	2.451-(2.462)-2.473	21	2.455
		22	2.460
		23	2.465
		24	2.470
		25	2.475
		26	2.480

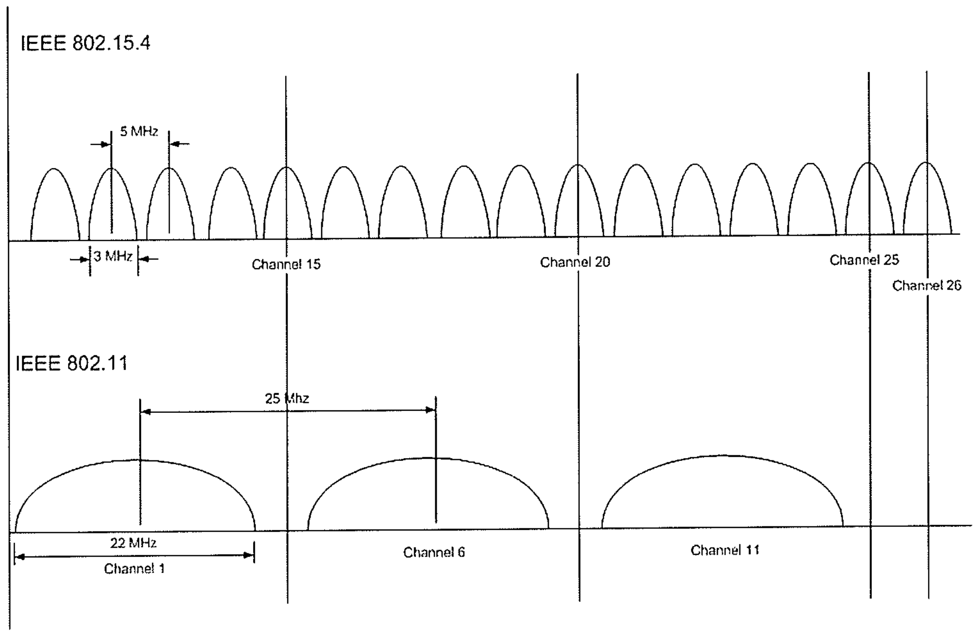


Figure 8.1: IEEE 802.15.4 and IEEE 802.11 Channels

istence of IEEE 802.11. IEEE 802.11 was setup to transmit on channel 11. This channel should influence communication of IEEE 802.15.4 over channels 20 through 24. Because of overhead to sample data, approximately 118 data points should be logged over this time. By choosing channel 26 for IEEE 802.15.4 we achieved a packet loss rate of less than 1% and logged between 118 and 120 samples. Choosing channel 22 for IEEE 802.15.4, which shared approximately the same center frequency as IEEE 802.11 channel 11, packets received varied from 25 to 155 out of an expected 118. This observation showed that more samples were being recorded than were taken. After observing the trace files from the wireless sniffer, it was concluded that the loss of ACK packets resulted in the retransmission of data. Although the data was being recorded by the cluster head, the acknowledgment was never being seen by the autonomous node resulting in up to 5 retransmissions. This observation resulted in a modification to the radio software.

When *AUTOACK* is enabled in the CC2420, all incoming frames accepted by the address recognition that have a valid CRC are acknowledged by transmitting an ACK frame that uses the data sequence number (DSN) from the received frame. The retransmit mechanism used by the transmitter retransmits frames with the same DSN until a corresponding ACK is received. To solve the problem of logging multiple data frames, the receiver was modified to log the last received DSN and only process new data records with a new DSN. This technique must be disabled for broadcast frames such as beacon frames to allow multiple beacon replies.

The modifications resulted in reasonable results. Packet reception typically was in the range of 20-40% in the tested environment. Results can vary drastically, however, depending on the signal propagation characteristics. A stochastic model, such as the one presented in [IH], could have provided a better understanding of the environment.

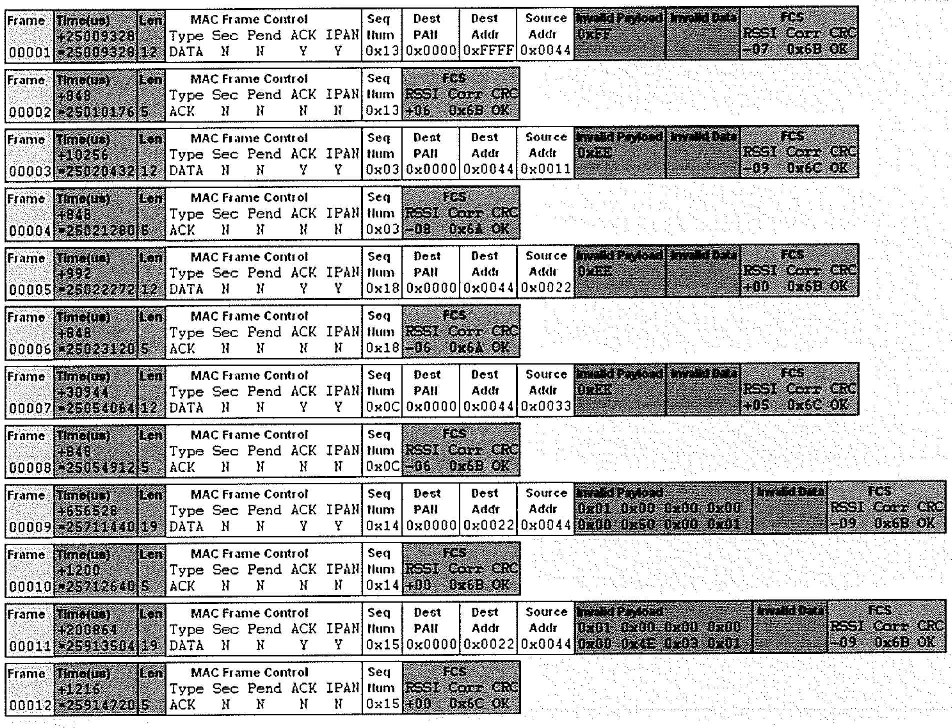
8.4 System Operation and Robustness

Multiple tests were conducted to verify the hardware and software worked as expected. Testing environments were setup to reflect the characteristics of the environments these sensors would be installed in. For initial testing, four prototype RMUs were developed.

8.4.1 Wireless Node Communicating to Multiple Cluster Heads

The first test involved one wireless node communicating to 3 cluster heads. Figure 8.2 shows a typical interaction of a wireless node with 3 cluster heads within transmission distance. Nodes with source addresses *0x0011*, *0x0022*, *0x003* were all configured as cluster heads, while the node with source address *0x0044* are configured as an autonomous wireless node. The wireless node was configured to perform a reading every 30 seconds. Communication begins when the wireless node's autonomous timer fires. At that time the wireless node sends out a beacon frame by sending a frame with payload *0xFF* to destination address *0xFFFF.* If the wireless node receives an ACK it will begin to listen for beacon replies. All three cluster heads received the beacon and replied with a beacon frame with payload *0xEE* to destination address *0x0044*, the wireless node's address. Cluster heads are notified that the beacon response frame was received by the notification of an ACK frame. The wireless node begins communication with the cluster head with the strongest RSSI value. Because RSSI values are calculated by the IEEE 802.15.4 transceiver at the time a packet is received, the RSSI values shown in Figure 8.2 are the values as seen by the sniffer and not the wireless node. The wireless node chose the cluster head *0x0022* and transmitted its reading to it.

Figure 8.3 shows results from the same setup except with a packet being re-transmitted. There are two mechanisms for congestion control. First, nodes wait a small, random back-off period before accessing the channel. Secondly, before accessing the channel, nodes perform a clear channel assess-

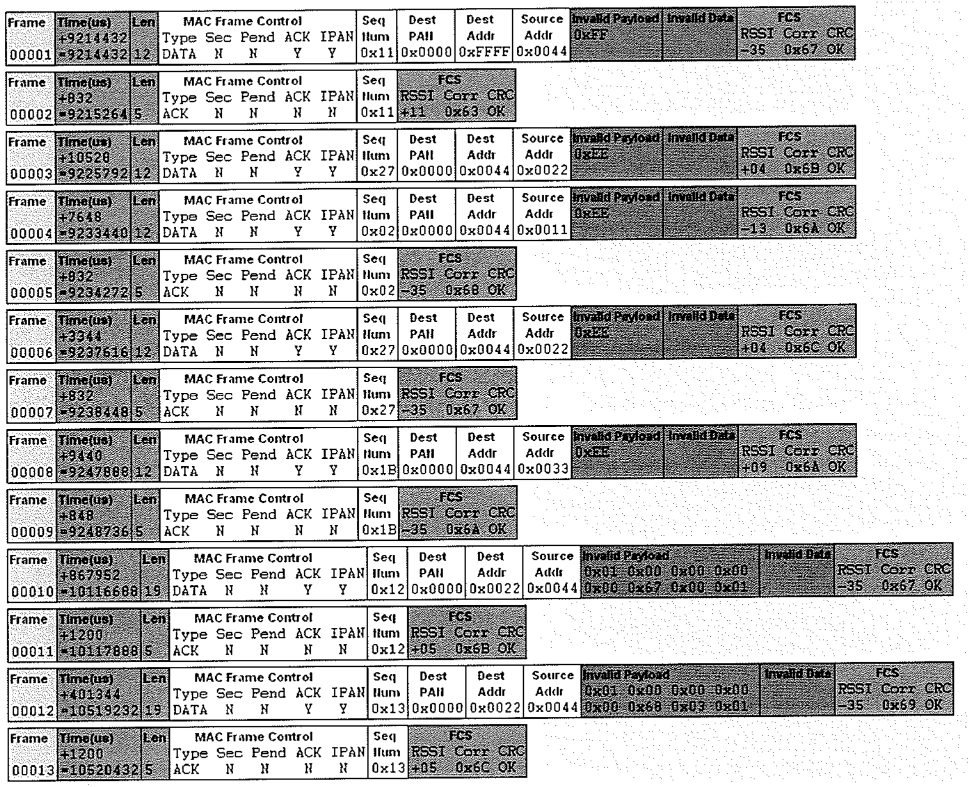


```
Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload Invalid Data    FCS
      +25009328     Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0xFF                            RSSI Corr CRC
00001 =25009328 12  DATA  N   N    Y   Y   0x13 0x0000 0xFFFF 0x0044                                 -07  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +848          Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00002 =25010176 5   ACK   N   N    N   N   0x13 +06  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload Invalid Data    FCS
      +10256        Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0xEE                            RSSI Corr CRC
00003 =25020432 12  DATA  N   N    Y   Y   0x03 0x0000 0x0044 0x0011                                 -09  0x6C OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +848          Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00004 =25021280 5   ACK   N   N    N   N   0x03 -08  0x6A OK

Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload Invalid Data    FCS
      +992          Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0xEE                            RSSI Corr CRC
00005 =25022272 12  DATA  N   N    Y   Y   0x18 0x0000 0x0044 0x0022                                 +00  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +848          Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00006 =25023120 5   ACK   N   N    N   N   0x18 -06  0x6A OK

Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload Invalid Data    FCS
      +30944        Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0xEE                            RSSI Corr CRC
00007 =25054064 12  DATA  N   N    Y   Y   0x0C 0x0000 0x0044 0x0033                                 +05  0x6C OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +848          Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00008 =25054912 5   ACK   N   N    N   N   0x0C -06  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload       Invalid Data    FCS
      +656528       Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0x01 0x00 0x00 0x00                 RSSI Corr CRC
00009 =25711440 19  DATA  N   N    Y   Y   0x14 0x0000 0x0022 0x0044 0x00 0x50 0x00 0x01                 -09  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +1200         Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00010 =25712640 5   ACK   N   N    N   N   0x14 +00  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq  Dest   Dest   Source Invalid Payload       Invalid Data    FCS
      +200864       Type Sec Pend ACK IPAN Num  PAN    Addr   Addr   0x01 0x00 0x00 0x00                 RSSI Corr CRC
00011 =25913504 19  DATA  N   N    Y   Y   0x15 0x0000 0x0022 0x0044 0x00 0x4E 0x03 0x01                 -09  0x6B OK

Frame Time(us)  Len     MAC Frame Control       Seq      FCS
      +1216         Type Sec Pend ACK IPAN Num  RSSI Corr CRC
00012 =25914720 5   ACK   N   N    N   N   0x15 +00  0x6C OK
```

Figure 8.2: One wireless node communication to three cluster heads

ment. In Figure 8.3 cluster head *0x0022* sends a beacon reply as seen in frame 3, but does not receive an ACK. In general, a node will try to retransmit a packet up to 5 times, but in this case its second attempt, frame 6, was ACK'ed.

These tests were conducted many times over a long period of time. By observing the output of the wireless sniffer we were able to verify the robustness of the protocol. Every instance of a dropped or unseen packet by either the gateway or cluster head was successfully re-transmitted.



```
Frame Time(us)  Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload Invalid Data  FCS
      +9214432      Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0xFF                          RSSI Corr CRC
00001 =9214432  12  DATA  N   N    Y   Y    0x11  0x0000 0xFFFF 0x0044                               -35  0x67 OK

Frame Time(us)  Len  MAC Frame Control        Seq    FCS
      +832          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00002 =9215264  5   ACK   N   N    N   N    0x11  +11  0x63 OK

Frame Time(us)  Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload Invalid Data  FCS
      +10528        Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0xEE                          RSSI Corr CRC
00003 =9225792  12  DATA  N   N    Y   Y    0x27  0x0000 0x0044 0x0022                               +04  0x6B OK

Frame Time(us)  Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload Invalid Data  FCS
      +7648         Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0xEE                          RSSI Corr CRC
00004 =9233440  12  DATA  N   N    Y   Y    0x02  0x0000 0x0044 0x0011                               -13  0x6A OK

Frame Time(us)  Len  MAC Frame Control        Seq    FCS
      +832          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00005 =9234272  5   ACK   N   N    N   N    0x02  -35  0x6B OK

Frame Time(us)  Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload Invalid Data  FCS
      +3344         Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0xEE                          RSSI Corr CRC
00006 =9237616  12  DATA  N   N    Y   Y    0x27  0x0000 0x0044 0x0022                               +04  0x6C OK

Frame Time(us)  Len  MAC Frame Control        Seq    FCS
      +832          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00007 =9238448  5   ACK   N   N    N   N    0x27  -35  0x67 OK

Frame Time(us)  Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload Invalid Data  FCS
      +9440         Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0xEE                          RSSI Corr CRC
00008 =9247888  12  DATA  N   N    Y   Y    0x1B  0x0000 0x0044 0x0033                               +09  0x6A OK

Frame Time(us)  Len  MAC Frame Control        Seq    FCS
      +848          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00009 =9248736  5   ACK   N   N    N   N    0x1B  -35  0x6A OK

Frame Time(us)   Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload      Invalid Data  FCS
      +867952        Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0x01 0x00 0x00 0x00                RSSI Corr CRC
00010 =10116688  19  DATA  N   N    Y   Y    0x12  0x0000 0x0022 0x0044 0x00 0x67 0x00 0x01                -35  0x67 OK

Frame Time(us)   Len  MAC Frame Control        Seq    FCS
      +1200          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00011 =10117888  5   ACK   N   N    N   N    0x12  +05  0x6B OK

Frame Time(us)   Len  MAC Frame Control        Seq   Dest   Dest   Source Invalid Payload      Invalid Data  FCS
      +401344        Type Sec Pend ACK IPAN  Num   PAN    Addr   Addr   0x01 0x00 0x00 0x00                RSSI Corr CRC
00012 =10519232  19  DATA  N   N    Y   Y    0x13  0x0000 0x0022 0x0044 0x00 0x68 0x03 0x01                -35  0x69 OK

Frame Time(us)   Len  MAC Frame Control        Seq    FCS
      +1200          Type Sec Pend ACK IPAN  Num   RSSI Corr CRC
00013 =10520432  5   ACK   N   N    N   N    0x13  +05  0x6C OK
```

Figure 8.3: One wireless node communication to three cluster heads with a retransmission

8.5 System Lifetime

One of the biggest issues addressed in this thesis is the system sensing lifetime[1]. All of the design and implementation choices kept this issue at the forefront. This can be seen in the cross layer design including choices for the MAC protocol and application design.

The initial hardware prototype design did not take into account power

[1]Sensing lifetime refers to the operation lifetime of a single mote

consumption, but some preliminary results for the second revision can be predicted based on the datasheet [Incb] using simulation. A simulation tool was written to provide insight into the effects and possible improvements to the suggested application. The simulation took into account the electrical characteristics of the MCU and radio, as well as battery characteristics to provide an estimation for battery life and hence sensor lifetime.

To provide a comparison between our protocol and IEEE 802.15.4, values for the MAC superframe order (SO) and MAC beacon order (BO) must to be chosen. SO determines the superframe duration (SD) whereas BO determines the beacon interval (BI), as shown in Figure 4.2. SO and BO are both integers between 0 and 14, requiring BO to be larger or equal to SO. The SD includes CAP and CFP, while BI includes SD and the inactive period. To achieve the best duty cycle for IEEE 802.15.4, values for BO and SO should be chosen to maximize the inactive period while minimizing the superframe duration. A 2.4GHz transceiver provides a symbol period of $16\mu s$ resulting in a maximum beacon interval of 245s. The minimum time for a SD is $16\mu s$, resulting in an extremely low duty cycle. It is important to note that while operating in a synchronous mode in very low duty cycles, motes require a very stable clock in order to not miss a beacon due to the addition of small clock drifts.

The expected number of hours of operation for a mote was estimated using three components. Equation 8.1 represents the power consumed while in PRI_RUN mode (P_{run}) where d, v_{dev} and i_{run} represent the duty cycle, device voltage and current in run mode, respectively.

$$P_{run} = d \times (v_{dev} i_{run}) \tag{8.1}$$

Equation 8.2 gives the power consumed while in either PRI_IDLE or SEC_IDLE modes, P_{idle}, where i_{idle} represents the current used in idle mode.

$$P_{idle} = (1 - d) \times (v_{dev} i_{idle}) \tag{8.2}$$

Equation 8.3 represents the power consumed by the radio (P_{radio}) each hour where t_h, P_{trans}, P_{recv}, and ρ represent the number of transmissions, per hour, the power consumed while transmitting, power consumed while receiving, and the probability of a collision, respectively.

$$P_{radio} = (1 + \rho) \sum_{i=1}^{t_h} (P_{trans} + P_{recv}) \tag{8.3}$$

Finally, Equation 8.4 provides the number of hours of operation (HO) by dividing the sum of P_{run}, P_{idle}, and P_{radio} by the battery's power, which is given by the battery voltage (B_v) and battery capacity (B_c).

$$HO = \frac{B_v B_c}{P_{run} + P_{idle} + P_{radio}} \tag{8.4}$$

While in PRI_IDLE mode, a PIC18F uses 0.85mA. Alternatively, operating in SEC_IDLE mode draws 13μA. Table 8.2 shows hours of operation, HO, for different scenarios. Using a standard Energizer CR2450 Battery providing 3V with a capacity of 575mAh and shelf life of over 10 years, Equation 8.4 with the current design operating with a duty cycle, $d = 0.1\%$ in PRI_RUN/PRI_IDLE modes results in 28 days of operation. By modifying the RMU to operate in PRI_RUN/SEC_IDLE the battery life is extended to 3.7 years by lowering the duty cycle even further. These calculations do not take into account age-related discharge, powering external peripherals, or analog measurements. Large capacity batteries, such as the Tadiran

Table 8.2: Hours of operation for different scenarios

Description	HO
CR2450 Battery (3V), $d = 0.1$, PRI_RUN/PRI_IDLE modes	28 days
CR2450 Battery (3V), PRI_RUN/SEC_IDLE	3.7 years
Tadiran Lithium Thionyl Chloride (3.6V), 2400mAh, PRI_RUN/SEC_IDLE	18.7 years
Tadiran Lithium Thionyl Chloride (3.6V), 2400mAh, 802.15.4 beacon-enabled mode	11.6 years

Lithium Thionyl Chloride cells providing 3.6V with a capacity of 2400mAh, can boost the potential life of the unit to 18.7 years, approaching the cell's 20 year operating life. However, using the same hardware but operating in IEEE 802.15.4 beacon enabled mode gives a maximum life of only 11.6 years.

Chapter 9

Conclusions and Future Work

This thesis presented the implementation and design issues for developing a heterogeneous structural health monitoring system based on TinyOS. Along with the identification of problems associated with the specific application of wireless sensors in buildings, a preliminary design was proposed. Alternatives ranging from a fully wired infrastructure to a wireless network were considered. From an engineering perspective, a heterogeneous network was proposed and its implementation undertaken.

9.1 Contributions

The main contributions of the thesis are as follows:

1. *TinyOS extensions:* Support for the PIC platform was continued by introducing modules for the PIC18F4680, CAN and an alternative module for ChipCon CC2420 to support the wireless IEEE 802.15.4 protocol.

2. *Use of a CAN:* A CAN was introduced as a routing medium between wireless sensors. Multi-hop routing which adds complexity to end sensors and requires additional power for radio listening was eliminated in favor of routing messages over a wired medium. As a side effect, the removal of multi-hop routing also eliminated redundant data.

3. *IEEE 802.15.4:* A simplified version of IEEE 802.15.4 was introduced to provide greater power saving. The modification switched the beaconing from the cluster heads to end nodes to initiate communication. This is justified as one of the main goals was energy conservation by the RMUs. Using the more traditional IEEE 802.15.4 beacon mode, the energy constraints could not have been met.

4. *Protocols:* Protocols were developed for the CAN and 802.15.4 as well as routing techniques. A beacon interval was introduced for end node to save power by eliminating listening periods and transmitting only when data is ready.

5. *Hardware:* A new hardware platform was created for use in wireless sensor networks. The new hardware platform contains components required for low powered operation with long sensing lifetime.

9.2 Future Work

In the future, the work presented in this thesis could be extended as follows:

1. Support in TinyOS for remote upgrading would allow firmware in the field to be upgraded to support new features and fix bugs. A boot

loader would be required to control the burning of a new image to program memory in the PIC.

2. For the first hardware production, a low frequency crystal connected to the timer inputs will allow the RMU to operate in SEC_IDLE instead of PRI_IDLE mode for sleeping.

3. After completion of the first hardware production, support for the new features will need to be included in TinyOS. This includes upgrades to timers and software support for new I/O and switches.

4. At the inception of this thesis, TinyOS 2.0 was only in beta and not mature enough for use in production. Since then, TinyOS 2.0 has been officially released out of beta. TinyOS 2.0 is a clean slate re-design and re-implementation of TinyOS resolving many of the limitations of TinyOS 1.1. Changes include redesigns of the hardware abstractions, scheduler, booting/initialization, virtualization, timers, communication, error codes, arbitration, power management, and network protocols.

5. Channel selection was designed to be done in either software or hardware by jumpers. The selection of a channel is based on a survey of the building to determine the best suited channel. Over time this selection may no longer be appropriate (e.g. due to addition of 802.11 access points) and, thus, may result in network performance degradation. An adaptive channel selection scheme similar to [CW] would prove to be valuable over time.

6. On top of adaptive channel selection, it would also be worthwhile exploring adaptive power selection for the CC2420 radio to produce the best power savings while maintaining reasonable loss rates.

7. Energy harvesting techniques might also be used to decrease the battery life constraint on sensing lifetime. The physical properties of the moisture detection tape allow induced voltages to occur. Voltages might also be induced by nearby AC power lines or RF energy. Induced voltages have been measured on the tape and, ironically, lead to the development of filters to remove it as the induced voltages on the moisture detection tape have been speculated to affect measurement circuitry.

9.3 Acknowledgement

I would like to thank SMT Research, http://www.smt-research.com, which is a Manitoba based company that performs research and development for structure monitoring technologies related to the field of Civionics and Building Science. Structure health monitoring technologies developed by SMT are targeted for the analysis of building envelope integrity and interior indoor air quality performance.

Bibliography

[BC02] M. Bhardwaj and A. P. Chandrakasan. Bounding the lifetime of sensor networks via optimal role assignments. In *Proceedings of IEEE INFOCOM*, June 2002.

[BC03] S. Bandyopadhyay and E. J. Coyle. An energy efficient hierarchical clustering algorithm for wireless sensor networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2003.

[CABM03] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of ACM MobiCom*, September 2003.

[Chi] Chipcon. CC2420 data sheet.

[CT04] J. H. Chang and L. Tassiulas. Maximum lifetime routing in wireless sensor networks. *IEEE/ACM Transactions on Networking*, 12(4):609 – 619, August 2004.

[CW] Jon-Hoon Youn Chulho Won. Adaptive radio channel allocation for supporting coexistence of 802.15.4 and 802.11b.

[DKN03] Koustuv Dasgupta, Konstantinos Kalpakis, and Parag Namjoshi. An efficient clustering-based heuristic for data gathering and aggregation in sensor networks. *In IEEE Wireless Communications and Networking Conference*, 2003.

[dTdK03] Hüseyin Özgür Tan and İbrahim Körpeoğlu. Power efficient data gathering and aggregation in wireless sensor networks. *SIGMOD Rec.*, 32(4):66–71, 2003.

[GKK+89] D. Goldberg, B. Karp, Y. Ke, S. Nath, and S. Seshan. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[GSYS02] S. Ghiasi, A. Srivastava, X. Yang, and M. Sarrafzadeh. Optimal energy aware clustering in sensor networks. *Sensors*, 2:258–259, 2002.

[HCB00] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the Hawaii International Conference on System Sciences*, January 2000.

[HI] Sajid Hussain and Obidul Islam. An energy efficient spanning tree based multi-hop routing in wireless sensor networks. *Technical Report, Jodrey School of Computer Science, Acadia University,* 2006(3):1–15.

[Hil00] Jason Hill. *A Software Architecture Supporting Networked Sensors,*. PhD thesis, University of California, Berkeley, 2000.

[IH] Jose A. Gutierrez Ivan Howitt. Ieee 802.15.4 low rate - wireless personal area network coexistence issues.

[Inca] Microchip Technology Inc. Microchip c18 compiler.

[Incb] Microchip Technology Inc. Pic18f2585/4585/2680/4680 data sheet.

[JJK04] L. Jae-Joon and B. Krishnamachari. Impact of heterogeneous deployment on lifetime sensing coverage in sensor networks. *Sensor and Ad Hoc Communications and Networks (SECON)*, October 2004.

[KDN02] Konstantinos Kalpakis, Koustuv Dasgupta, and Parag Namjoshi. Maximum lifetime data gathering and aggregation in wireless sensor networks. *In IEEE International Conference on Networking*, pages 685–696, August 2002.

[KWSH05] H.-J. Krber, H. Wattar, G. Scholl, and W. Heller. Embedding a microchip PIC18F452 based commercial platform into TinyOS. *Proceedings of REALWSN*, June 2005. Stockholm, Sweden.

[LO04] C. Lynch and F. O'Reilly. PIC-based sensor operating system. *Irish Signals and Systems Conference*, June 2004. Belfast, Ireland.

[LR05] C. Lynch and F.O. Reilly. PIC-based TinyOS implementation. *Proceedings of the Second European Workshop on Wireless Sensor Networks*, February 2005.

[LYCW05] C. Li, M. Ye, G. Chen, and J. Wu. An energy efficient unequal clustering mechanism for wireless sensor networks. *In Proc. of MASS*, Nov 2005.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions of Database Systems (TODS)*, 30(1):122–173, 2005.

[oEE] Institute of Electrical and Electronics Engineers. IEEE 802.15.4.

[SFLR01] Roy Sutton Suet-Fei Li and Jan Rabaey. Low power operating system for heterogeneous wireless communication systems. In *Workshop on Compilers and Operating Systems for Low Power 2001*, September 2001.

[SK04] N. Sadagopan and B. Krishnamachari. Maximizing data extraction in energy-limited sensor networks. In *Proceedings of IEEE INFOCOM*, March 2004.

[SR02] R. C. Shah and J. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *Proceedings of IEEE WCNC*, March 2002.

[Teca] Crossbow Technology. Btnode.

[Tecb] Crossbow Technology. Imote2.

[Tecc] Crossbow Technology. Micaz.

[Tecd] Crossbow Technology. Telosb.

[TMo] Tmote.

[WTC03] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multi-hop routing in sensor networks. In *Proceedings of ACM SenSys*, November 2003.

[YF04] O. Younis and S. Fahmy. Distributed clustering in ad hoc sensor networks: A hybrid, energy-efficient approach. *IEEE Transactions on Mobile Computing*, 3(4):366–379, Dec 2004.

[YLC+02] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A two-tier data dissemination model for large-scale wireless sensor networks. In *Proceedings of ACM MobiCom*, September 2002.

Chapter 10

Appendix

```
/***************************************************************
 * File: WISMS.nc
 * Greg Jaman
 * WISMS Interface
 ***************************************************************/
interface WISMS
{
  command result_t setID(uint16_t *nodeid);
  command result_t getID(uint16_t *nodeid);
  command result_t setAutonomous(uint8_t set);
  command result_t StartClusterHead();

}
```

```
/***************************************************************
 * File: WISMSCommand.nc
 * Greg Jaman
 * WISMSCommand Interface
 ***************************************************************/
includes WISMSCommand;
includes AM;
interface WISMSCommand
{
  command result_t execute(TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send);
  event result_t fired(TOS_MsgPtr pmsg_send);
}
```

```
/*****************************************************************
 * File: CAN.nc
 * Greg Jaman
 * Configuration for CAN
 *****************************************************************/
includes CAN;
configuration CAN {
  provides{
    interface BareSendMsg as Send;
    interface ReceiveMsg as Receive;
    interface StdControl as Control;
   }
}
implementation
{
  components HPLCAN, PIC18F4620InterruptC;
  Control   = HPLCAN;
  Send      = HPLCAN.Send;
  Receive   = HPLCAN.Receive;
  HPLCAN.CAN_RX0_Interrupt -> PIC18F4620InterruptC.CAN_RX0_Interrupt;
  HPLCAN.CAN_RX1_Interrupt -> PIC18F4620InterruptC.CAN_RX1_Interrupt;

}
```

```
/****************************************************************
 * File: HPLCAN.nc
 * Greg Jaman
 * HPLCAN module
 ****************************************************************/
includes CAN;
includes AM;
module HPLCAN {
        provides{
        interface StdControl;
        interface BareSendMsg as Send;
        interface ReceiveMsg as Receive;
    }
    uses{
        interface PIC18F4620Interrupt as CAN_RX0_Interrupt;
        interface PIC18F4620Interrupt as CAN_RX1_Interrupt;
    }
}

implementation
{
  norace TOS_MsgPtr pBuf;                    // changed only in interrupt context
  TOS_Msg RxBuf;                             // define a TOS_Msg object

  command result_t StdControl.init() {
    CANCON_register = 0x80;                  // Request that the CAN module enter configuration mode
    while (CANSTAT_register & 0xE0 != 0x80);     // Wait to make sure we enter configuration mode
    //Bit Timing Configuration
    BRGCON1_register =   0x07;
    //BRGCON1 = 0b00000111; // (0x07)
            //00------  Synchronization Jump Width (SJW) = 1 x Tq
            //--000111  Tq = (2 x 8)/Fosc = 4us (@4MHz) (w/25 x Tq/bit = 10kbit/s)
    BRGCON2_register =   0xff;
    //BRGCON2 = 0b11111111; // (0xFF)
            //1-------  Phase Segment 2 is freely programmable
            //-1------  CAN bus line is sampled 3 times prior to the sample point
            //--111---  Phase Segment 1 = 8 x Tq
            //-----111  Propagation Time = 8 x Tq
    BRGCON3_register =  0x07;
    //BRGCON3 = 0b00000111; // (0x07)
            //X-XXX---  Don't care (unimplemented - set to '0')
            //-0------  CAN bus filter is not used to wake up the processor
            //-----111  Phase Segment 2 = 8 x Tq
    //CAN I/O Control Register
    CIOCON_register =   0x20;
    //CIOCON =  0b00100000; // (0x20)
            //XX--XXXX  Don't care (unimplemented - set to '0')
            //--1-----  CANTX pin will drive Vdd when recessive
            //---0----  CAN Capture disabled; RC2 is input to CCP1 module

    //Receive Buffer 0 Control Register
    RXB0CON_register =  0x40;
    //RXB0CON = 0b01000000; // (0x40)
            //1-------  Clear the message received flag
            //-10-----  Receive only valid messages with extended identifiers
            //---X----  Don't care (unimplemented - set to '0')
            //----0---  No Remote Transfer Request (RTR)
            //-----0--  Receive buffer 0 will not overflow to Receive buffer 1
            //------XX  Don't care

    // We only want to receive messages that match the ID (0x00000000) so we need to enable all of the
 bits in the
    // receiver acceptance mask
    RXM0SIDH_register = 0xFF;
    RXM0SIDL_register = 0xE3;
    RXM0EIDH_register = 0xFF;
    RXM0EIDL_register = 0xFF;
    //Filter 0
    RXF0EIDL_register = 0x00;
    RXF0EIDH_register = 0x00;
    RXF0SIDL_register = 0x08;
    RXF0SIDH_register = 0x00;
    // Filter 1
    RXF1EIDL_register = 0x00;
    RXF1EIDH_register = 0x00;
    RXF1SIDL_register = 0x08;
    RXF1SIDH_register = 0xF8;

    CANCON_register = 0x00;                         // Request that the CAN module return to normal
operating mode
    while (CANSTAT_register & 0xE0 != 0x00);        // Wait for the CAN module to return to normal
operating mode
    //get_identifier();
      return SUCCESS;
  }

  command result_t StdControl.start() {
      //Enable interrupt
      PIE3bits_RXB0IE = 1;  //Enable Receive Buffer 0 interrupt
      PIE3bits_RXB1IE = 1;  //Enable Receive Buffer 1 interrupt
      return SUCCESS;
  }

  command result_t StdControl.stop() {
      //Dissable interrupt
      PIE3bits_RXB0IE = 0; //Dissable Receive buffer 0 interrupt
      PIE3bits_RXB1IE = 0; //Dissable Receive buffer 1 interrupt
      return SUCCESS;
  }


task void PacketRcvd() {
     TOS_MsgPtr ptBuf;
     atomic {
       ptBuf = &RxBuf; //create a pointer to the message;
     }
     ptBuf = signal Receive.receive((TOS_MsgPtr)ptBuf);
}


  //Called from the Interrupt Handler Routine
  async event result_t CAN_RX0_Interrupt.fired(){
    atomic{
        pBuf =  &RxBuf; //Assign pBuf to the address;
          if(PIR3bits_IRXIF){
            PIR3bits_IRXIF = 0;
            RXB0CONbits_RXFUL=0; //Clear the flag for new message
            PIR3bits_RXB0IF = 0;
        }
        if(RXB0CONbits_RXFUL){ //Receive buffer contain a received message
            pBuf->addr          = ((unsigned long)RXB0D1_register << 8) | (unsigned long)
RXB0D2_register;
            pBuf->srcaddr       = 0x00;
            pBuf->type      = AM_CANMSG;
            pBuf->group     = TOS_AM_GROUP ;
            pBuf->length    = RXB0DLC_register;
            pBuf->data[0]   = RXB0D0_register;
            pBuf->data[1]   = RXB0D1_register;
        pBuf->data[2]   = RXB0D2_register;
        pBuf->data[3]   = RXB0D3_register;
        pBuf->data[4]   = RXB0D4_register;
        pBuf->data[5]   = RXB0D5_register;
        pBuf->data[6]   = RXB0D6_register;
        pBuf->data[7]   = RXB0D7_register;
        post PacketRcvd();
        RXB0CONbits_RXFUL=0; //Clear the flag for new message
        PIR3bits_RXB0IF = 0; //Clear the interrupt flag
      }
    }
    return SUCCESS;
  }

  command result_t Send.send(TOS_MsgPtr pMsg) {
    uint8_t templ, temph, tmpcanh;
    if(pMsg->usePacketaddr){
      templ = TXB0EIDL_register;
      temph = TXB0EIDH_register;
      TXB0EIDH_register =  (uint8_t)(pMsg->srcaddr>> 8 );
      TXB0EIDL_register =  (uint8_t)(pMsg->srcaddr & 0x00ff);
    }
    if(pMsg->autonomous){
      //change the wisms message type.   EID=29bits  (5bits=type, 14bits=id)
      tmpcanh = TXB0SIDH_register;
      TXB0SIDH_register =  ( 2<<3   ) | ( TXB0SIDH_register & 0x07);
    }
    TXB0DLC_register = pMsg->length;
    if(pMsg->length >=1) TXB0D0_register = pMsg->data[0];
    if(pMsg->length >=2) TXB0D1_register = pMsg->data[1];
    if(pMsg->length >=3) TXB0D2_register = pMsg->data[2];
    if(pMsg->length >=4) TXB0D3_register = pMsg->data[3];
    if(pMsg->length >=5) TXB0D4_register = pMsg->data[4];
    if(pMsg->length >=6) TXB0D5_register = pMsg->data[5];
    if(pMsg->length >=7) TXB0D6_register = pMsg->data[6];
    if(pMsg->length >=8) TXB0D7_register = pMsg->data[7];

    TXB0CONbits_TXREQ = 1;
    while (TXB0CONbits_TXREQ); //Send msg

    if(pMsg->autonomous){
      TXB0SIDH_register = tmpcanh;
    }

    if(pMsg->usePacketaddr){
      //restore the original address
      TXB0EIDL_register = templ;
      TXB0EIDH_register = temph;
    }
    return FAIL;            //Fail needs to be returned for the state to be cleared in AMStandard

  }
  async event result_t CAN_RX1_Interrupt.fired(){
    return SUCCESS;
  }

}
```

```
/*****************************************************************
 * File: AutonomousTimer.nc
 * Greg Jaman
 * Configuration for AutonomousTimer
 *****************************************************************/
 configuration AutonomousTimer {
  provides interface Timer;
  provides interface StdControl;
}

implementation {
  components TimerC;
  Timer         = TimerC.Timer[unique("Timer")];
  StdControl    = TimerC;
}
```

```
/****************************************************************
 * File: CAN.nc
 * Greg Jaman
 * CMDMeasure Module
 ****************************************************************/
module CMDMeasure
{
  provides {
    interface WISMSCommand as cmd_Measure[uint8_t id];
  }
  uses {
    interface A2D;
  }
}

implementation
{
    uint8_t cmdid = NULL;
    unsigned char channel;
    unsigned char polarity;
    TOS_MsgPtr workingMsgPtr;

  command result_t cmd_Measure.execute[uint8_t id](TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send)
  {
    polarity = pmsg->data[5];
    channel= pmsg->data[4];
    workingMsgPtr = pmsg_send;

    if(cmdid!=NULL){
       //call callback
       return FAIL;
    }else{
       cmdid = id;
    }
    if(!call A2D.measure(channel, polarity)){
            workingMsgPtr->data[0] = 0xFF;
            workingMsgPtr->length = 1;
            signal cmd_Measure.fired[cmdid](workingMsgPtr);
    }
    return SUCCESS;
  }

  default event result_t cmd_Measure.fired[uint8_t id](TOS_MsgPtr pmsg_send)
  {
      cmdid = NULL;
      return SUCCESS;
  }

  event result_t A2D.measureDone(uint32_t reading, uint8_t inchannel, uint8_t inpolarity)
  {
        workingMsgPtr->data[0] = (reading >> 24) & 0x000000FF;
        workingMsgPtr->data[1] = (reading >> 16) & 0x000000FF;
        workingMsgPtr->data[2] = (reading >> 8) & 0x000000FF;
        workingMsgPtr->data[3] = reading & 0x000000FF;
        workingMsgPtr->length=4;
       if(workingMsgPtr->autonomous == TRUE){
            workingMsgPtr->data[5] = channel;
            workingMsgPtr->data[6] = polarity;
            workingMsgPtr->length = 7;
        }else workingMsgPtr->length = 4;
        signal cmd_Measure.fired[cmdid](workingMsgPtr);
        cmdid = NULL;
        return SUCCESS;
    }
}
```

```
/***************************************************************
 * File: CAN.nc
 * Greg Jaman
 * CMDChangeID Module
 ***************************************************************/
module CMDChangeID
{
  provides {
    interface WISMSCommand as cmd_ChangeID[uint8_t id];
  }
  uses{
    interface WISMS;
  }
}

implementation
{
   uint16_t tempID;
   uint8_t  cmdid = NULL;

   command result_t cmd_ChangeID.execute[uint8_t id](TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send)
   {
        if(cmdid!=NULL){
            //call callback
            return FAIL;
        }else{
            cmdid = id;
        }
        pmsg_send->length = 0;
        if ((( (uint8_t)pmsg->data[4] == 0x00) && ( (uint8_t)pmsg->data[5] == 0x00) && ((uint8_t)pmsg-
>data[6] == 0x00)) ||
           (((uint8_t)pmsg->data[4] == 0xFF) && ((uint8_t)pmsg->data[5] == 0xFF) && ((uint8_t)pmsg->data
[6] == 0xFF)))
                pmsg->data[0] = 0xFF;                       // Invalid Command Data Error
        else
        {
            //  report success.
            pmsg_send->data[0] = 0;
            tempID = ((uint16_t)(pmsg->data[5]) << 8 ) | (0x00FF& (uint16_t )pmsg->data[6]);
            call WISMS.setID(&tempID);
        }
        pmsg_send->length = 1;              // 1-byte success flag
        //now send that message over the radio
        signal cmd_ChangeID.fired[cmdid](pmsg_send);
        cmdid = NULL;
        return SUCCESS;
   }

   default event result_t cmd_ChangeID.fired[uint8_t id](TOS_MsgPtr pmsg_send)
   {
      cmdid = NULL;
      return SUCCESS;
    }

}
```

```
/***************************************************************
 * File: CAN.nc
 * Greg Jaman
 * CMDAutonomous Module
 ***************************************************************/
module CMDAutonomous
{
  provides {
    interface WISMSCommand as cmd_Autonomous[uint8_t id];
  }
  uses {
    interface WISMS;
  }
}

implementation
{

  uint8_t   cmdid = NULL;
  command result_t cmd_Autonomous.execute[uint8_t id](TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send)
  {
    if(cmdid!=NULL){
       //call callback
       return FAIL;
    }else{
       cmdid = id;
    }
    pmsg_send->data[0] = 0;
    if (pmsg->data[4] & 0xFE)       // An unsupported relay state has been requested
        pmsg_send->data[0] = 0xFF;      // Invalid Command Data Error
    else
        if(pmsg->data[4]) call WISMS.setAutonomous(TRUE);
        else call WISMS.setAutonomous(FALSE);
    pmsg_send->length = 1;
    //call SendMsg.send(0x0000, pmsg_send->length, pmsg_send);

    signal cmd_Autonomous.fired[cmdid](pmsg_send);
    cmdid = NULL;
    return SUCCESS;
  }

  default event result_t cmd_Autonomous.fired[uint8_t id](TOS_MsgPtr pmsg_send)
  {
      cmdid = NULL;
      return SUCCESS;
  }
}
```

```
/**************************************************************
 * File: CAN.nc
 * Greg Jaman
 * CMDReset Module
 *************************************************************/
module CMDReset
{
  provides {
    interface WISMSCommand as cmd_Reset[uint8_t id];

  }
}

implementation
{
   uint8_t  cmdid = NULL;
   command result_t cmd_Reset.execute[uint8_t id](TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send)
   {
        if(cmdid!=NULL){
            //call callback
            return FAIL;
        }else{
            cmdid = id;
        }
        STKPTR_register = 0x00;             // STKPTR[7:6] are set to '0' on POR
        T3CON_register  = 0x00;             // T3CON[7:0] are set to '0' on POR
        // The PIC18 series supports resetting the PIC with a simple instruction which mimics a MCLR or
WDT reset
        TOSH_reset();
        pmsg_send->length = 0;
        pmsg_send->type = AM_CANMSG;
        signal cmd_Reset.fired[cmdid](pmsg_send);
        cmdid = NULL;
        return SUCCESS;
   }

  default event result_t cmd_Reset.fired[uint8_t id](TOS_MsgPtr pmsg_send)
  {
      return SUCCESS;
  }
}
```

```
/****************************************************************
 * File: CAN.nc
 * Greg Jaman
 * CMDGetID Module
 ***************************************************************/
module CMDGetID
{
  provides {
    interface WISMSCommand as cmd_GetID[uint8_t id];
  }
}

implementation
{
   uint8_t  cmdid = NULL;
   command result_t cmd_GetID.execute[uint8_t id](TOS_MsgPtr pmsg, TOS_MsgPtr pmsg_send)
   {
        if(cmdid!=NULL){
            return FAIL;
        }else{
            cmdid = id;
        }
        pmsg_send->length = 0;
        pmsg_send->type = AM_CANMSG;
        signal cmd_GetID.fired[cmdid](pmsg_send);
        cmdid = NULL;
        return SUCCESS;
   }


  default event result_t cmd_GetID.fired[uint8_t id](TOS_MsgPtr pmsg_send)
  {
      return SUCCESS;
  }
}
```

```
/**************************************************************
 * File: WISMSC.nc
 * Greg Jaman
 * WISMSC Configuration
 *************************************************************/
includes WISMS;
includes AM;
configuration WISMSC {
}
implementation {
  components Main, WISMSM,   LedsC, EEPROM,
        WISMSMessageHandler, AutonomousTimer,
        WISMSCommandC,
        CC2420ControlM,
        TimerC,
        HPLSpiM,
        A2DC,
        GenericComm as Comm;

  Main.StdControl -> AutonomousTimer.StdControl;
  Main.StdControl -> WISMSM.StdControl;
  Main.StdControl -> EEPROM.StdControl;

  WISMSM.AutonomousTimer    -> AutonomousTimer.Timer;
  WISMSM.waitTimer          -> TimerC.Timer[unique("Timer")];
  WISMSM.Leds               -> LedsC;
  WISMSM.RadioSend          -> Comm.SendMsg[AM_WIRELESSMSG];
  WISMSM.CANSend            -> Comm.SendMsg[AM_CANMSG];
  WISMSM.RadioReceive       -> Comm.ReceiveMsg[AM_WIRELESSMSG];
  WISMSM.CC2420Control      -> CC2420ControlM;
  WISMSM.A2D                -> A2DC.A2D[unique("A2D")];
  WISMSM.CommControl        -> Comm;

  WISMSMessageHandler.ReceiveMsg    -> Comm.ReceiveMsg[AM_CANMSG]; //how about wireless?
  WISMSMessageHandler.SendMsg       -> Comm.SendMsg[AM_CANMSG];

  WISMSCommandC.WISMS               -> WISMSM.WISMS;
  WISMSMessageHandler.CMDGetID      -> WISMSCommandC.cmd_GetID[unique("CMDGetID")];
  WISMSMessageHandler.CMDMeasure    -> WISMSCommandC.cmd_Measure[unique("CMDMeasure")];
  WISMSMessageHandler.CMDChangeID   -> WISMSCommandC.cmd_ChangeID[unique("CMDChangeID")];
  WISMSMessageHandler.CMDAutonomous -> WISMSCommandC.cmd_Autonomous[unique("CMDAutonomous")];
  WISMSMessageHandler.CMDReset      -> WISMSCommandC.cmd_Reset[unique("CMDReset")];

  WISMSM.EEPROMRead -> EEPROM.EEPROMRead;
  WISMSM.EEPROMWrite -> EEPROM.EEPROMWrite[unique("EEPROMWrite")];
}
```

```
/************************************************************
 * File: CAN.nc
 * Greg Jaman
 * WISMSMessageHandlerM Module
 ************************************************************/
module WISMSMessageHandlerM {
  provides {
    interface StdControl;
  }
  uses {
    interface WISMSCommand as cmd_GetID;
    interface WISMSCommand as cmd_ChangeID;
    interface WISMSCommand as cmd_Autonomous;
    interface WISMSCommand as cmd_Measure;
    interface WISMSCommand as cmd_Reset;

    interface ReceiveMsg    as ReceiveMsg;
    interface SendMsg       as SendMsg;
  }
}
implementation {
  TOS_Msg TxBuf; //Holder to output Msg

  command result_t StdControl.init()
  {
    TOS_MsgPtr  msgsp = &TxBuf;
    msgsp->usePacketaddr = FALSE;
    return SUCCESS;
  }

  command result_t StdControl.start()
  {
    return SUCCESS;
  }

  command result_t StdControl.stop()
  {
    return SUCCESS;
  }

  event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msgp)
  {
        TOS_MsgPtr  msgsp = &TxBuf;
        result_t tosend= FAIL;
        uint8_t cmd = msgp->data[3];
        msgsp->usePacketaddr = FALSE;
        msgsp->autonomous    = FALSE;
        switch(cmd){
            case (1) :
                tosend = call cmd_GetID.execute((TOS_MsgPtr)msgp, msgsp);
                break;
            case (2) :
                tosend = call cmd_Measure.execute((TOS_MsgPtr)msgp, msgsp);
                break;
            case (03) :
                tosend = call cmd_Autonomous.execute((TOS_MsgPtr)msgp, msgsp);
                break;
            case (0xF9) :
                tosend = call cmd_Reset.execute((TOS_MsgPtr)msgp, msgsp);
            case (0xFC) :
                tosend = call cmd_ChangeID.execute((TOS_MsgPtr)msgp, msgsp);
                break;
        }
        return msgp;
    }

  event result_t cmd_GetID.fired(TOS_MsgPtr pmsg_send)
  {
    call SendMsg.send(0x0000, pmsg_send->length, pmsg_send);
    return SUCCESS;
  }

  event result_t cmd_ChangeID.fired(TOS_MsgPtr pmsg_send)
  {
    call SendMsg.send(0x0000, pmsg_send->length, pmsg_send);
    return SUCCESS;
  }

  event result_t cmd_Measure.fired(TOS_MsgPtr pmsg_send)
  {
    call SendMsg.send(0x0000, pmsg_send->length, pmsg_send);
    return SUCCESS;
  }

  event result_t cmd_Autonomous.fired(TOS_MsgPtr pmsg_send)
  {
    call SendMsg.send(0x0000, pmsg_send->length, pmsg_send);
    return SUCCESS;
  }

  event result_t cmd_Reset.fired(TOS_MsgPtr pmsg_send)
  {
    return SUCCESS;
  }

  event result_t SendMsg.sendDone(TOS_MsgPtr param0, result_t param1)
  {
        return SUCCESS;
    }
}
```

```
/*************************************************************
 * File: WISMSM.nc
 * Greg Jaman
 * WISMS Module
 *************************************************************/
module WISMSM {
  provides {
    interface StdControl;
    interface WISMS;
  }
  uses {
    interface Timer as AutonomousTimer;
    interface Timer as waitTimer;
    interface Leds;
    interface StdControl as CommControl;
    interface EEPROMRead;
    interface EEPROMWrite;
    interface SendMsg    as RadioSend;
    interface SendMsg    as CANSend;
    interface ReceiveMsg as RadioReceive;
    interface CC2420Control;
    interface A2D;
  }
}
implementation {
/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
*  Members
*/
    TOS_Msg TestTxBuf; //Holder to output Msg
    uint8_t InputsToCheck;
    uint8_t CheckPosition;
    uint8_t CurrentChannel;
    uint8_t CurrentPolarity;
    uint8_t AutonomousState = 0;
    uint16_t ClusterHead_addr = 0x00;   //No node can have 00 as address
    uint8_t  ClusterHead_strength = 0x00;
    enum{
        autonomous_IDLE =0,
        autonomous_WORKING,
        autonomous_BEACON_WAIT,
        autonomous_SEND_DATA,       //state of sending data
        autonomous_NEXT_CHANNEL     //setup for the next channel
    };
    uint8_t ConfigOptions = 0;
    #define WISMS_autonomous    (ConfigOptions & 0x04)
    #define WISMS_medium        (ConfigOptions & 0x08)
                                //Medium:  0 CAN, 1 Wireless
    #define WISMS_Sleep_Period  60000

/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
*  Private prototype functions
*/
  void set_CAN_identifier();
  result_t get_hardware_options();
  result_t get_config_options();

  command result_t StdControl.init()
  {
    return rcombine(call CommControl.init(), call Leds.init());
  }

  command result_t StdControl.start()
  {
    uint16_t tempid;
    InputsToCheck = 0xff;
    CheckPosition = 0;
    get_hardware_options();
    get_config_options();
    call WISMS.getID(&tempid);
    call Leds.greenOn();  //Set green to indicate you are a receiver;
    if(WISMS_autonomous){
         call AutonomousTimer.start(TIMER_REPEAT, WISMS_Sleep_Period);
    }
    else{
         call Leds.redOn();
         call  WISMS.StartClusterHead();
    }
    return call CommControl.start();

  }

  command result_t StdControl.stop()
  {
    return rcombine(call AutonomousTimer.stop(), call CommControl.stop() );
  }

 void structureAutonomousMsg(TOS_MsgPtr pmsg_send, uint8_t type)
  {
    int8_t i;
    for(i=pmsg_send->length-1; i>=0; --i) pmsg_send->data[i+1] = pmsg_send->data[i];
    pmsg_send->data[0] = type;
    pmsg_send->length+=1;

  }

  task void TakeAutonomousReading()
  {
    call A2D.measure(CurrentChannel,CurrentPolarity);
  }

  task void startAutonomous()
  {
       TOS_MsgPtr  msgsp = &TestTxBuf;
       bool found = FALSE;
       if(ClusterHead_addr == 0x0000 && AutonomousState != autonomous_BEACON_WAIT){
            //send beacon...
            call CommControl.start();
            msgsp->length = 1;
            msgsp->data[0] = 0xFF;              //set the command to be FF
            call RadioSend.send(TOS_BCAST_ADDR, msgsp->length, msgsp);
            AutonomousState = autonomous_BEACON_WAIT;
            TOSH_radioSleep_disable();                      //keep radio on while sleepig
            call waitTimer.start(TIMER_ONE_SHOT, 2000);
            call Leds.redOn();
            return;                             //return so other posted tasks can execute
        }
        if(ClusterHead_addr!=0x0000 && AutonomousState!=autonomous_SEND_DATA){
            AutonomousState = autonomous_SEND_DATA;
            while(!found){
                if(CheckPosition<4){
                    if(  (InputsToCheck >> CheckPosition) & 0x01){
                        found=TRUE;
                        CurrentChannel = CheckPosition;
                        CurrentPolarity = 0;
                        post TakeAutonomousReading();
                        CheckPosition++;
                    }else CheckPosition++;
                }else{
                    found=TRUE;
                    CheckPosition = 0; //reset Checkpoisition
                    AutonomousState = autonomous_IDLE;
                }
            }
        }else AutonomousState = autonomous_IDLE;
  }

  event result_t waitTimer.fired()
  {
        call Leds.redOff();
        TOSH_radioSleep_enable();               //back to normal sleep
        if(AutonomousState == autonomous_BEACON_WAIT) post startAutonomous();
        return SUCCESS;
  }

  event result_t AutonomousTimer.fired()
  {
    if(AutonomousState == autonomous_IDLE ){
        AutonomousState = autonomous_WORKING;
        ClusterHead_addr = 0x0000;
        ClusterHead_strength = 0x00;
        post startAutonomous(); //need to check in case it changes
    }
    return SUCCESS;
  }

//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
  task void ClusterHeadListen()
  {
     if(!WISMS_autonomous) post ClusterHeadListen();
  }

  command result_t WISMS.StartClusterHead()
  {
        TOSH_radioSleep_disable();
        post ClusterHeadListen();
        return SUCCESS;
  }

  event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr msgp)
  {
       TOS_MsgPtr  msgsp = &TestTxBuf;
       uint8_t cmd = msgp->data[0];
       if(!WISMS_autonomous){
            switch(cmd){
                case (0xFF) :
                    msgsp->length = 1;              //only reply with 1 byte
                    msgsp->data[0] = 0xEE;          //reply wit FF
                    call RadioSend.send(msgp->srcaddr, msgsp->length, msgsp);
                    //Call command to send BeaconReply.
                    break;
                default:
                    msgp->usePacketaddr=TRUE;       //puts the src address in the CAN register
                    msgp->autonomous      =TRUE;                        //Changes the type of the EID
                    call CANSend.send(0x0000, msgp->length, msgp);
                    break;
            }
        }else{
            if(msgp->srcaddr != ClusterHead_addr){
                if(msgp->strength > ClusterHead_strength){
                    //chose a new cluster head...
                        ClusterHead_addr = msgp->srcaddr;
                        ClusterHead_strength = msgp->strength;
                    }
                }
            }
            return msgp;
    }

/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
*  Commands provided for WISMS interface
*
*/
    command result_t WISMS.setID(uint16_t *nodeid)
    {
          uint8_t eidl, eidh, temp;
          eidl = (uint8_t)( *nodeid & 0x00FF );
          eidh = (uint8_t)( (*nodeid >> 8) & 0x00ff);
          call EEPROMRead.read(CAN_ID_HIGH, &temp);
          call EEPROMWrite.write(CAN_ID_LOW, &eidl);
          call EEPROMWrite.write(CAN_ID_MID, &eidh);
          //SET the CAN nodeID / EID
          TXB0EIDL_register = eidl;
          TXB0EIDH_register = eidh;
          TXB0SIDL_register = ( (temp & 0x03) | 0x08 | ( (temp << 3) & 0xE0) );
          TXB0SIDH_register = 0x08 | (temp >> 5);
          TOS_LOCAL_ADDRESS = *nodeid;
          return SUCCESS;
    }

    command result_t WISMS.getID(uint16_t *nodeid)
    {
          uint8_t eidl, eidh,temp;
          call EEPROMRead.read(CAN_ID_HIGH, &temp);
          call EEPROMRead.read(CAN_ID_LOW, &eidl);
          call EEPROMRead.read(CAN_ID_MID, &eidh);
          TXB0EIDL_register = eidl;
          TXB0EIDH_register = eidh;
          TXB0SIDL_register = ((temp & 0x03) | 0x08 | ((temp << 3) & 0xE0));
          TXB0SIDH_register = 0x08 | (temp >> 5);
          *nodeid = ((uint16_t)eidh << 8 ) | (uint16_t )eidl;
          TOS_LOCAL_ADDRESS = *nodeid;
          return SUCCESS;
    }

    command result_t WISMS.setAutonomous(uint8_t set)
    {
        if(WISMS_autonomous){
          if(!set){
               ConfigOptions &= ~0x04;
               call EEPROMWrite.write(CONFIG_OPTIONS, &ConfigOptions);
               call AutonomousTimer.stop();
               TOSH_radioSleep_disable();
               //Start the Radio and Post the Listen task
               call CommControl.start();
               post ClusterHeadListen();
          }
        }else{
          if(set){
               ConfigOptions |= 0x04;
               TOSH_radioSleep_enable();
               call EEPROMWrite.write(CONFIG_OPTIONS, &ConfigOptions);
               call AutonomousTimer.start(TIMER_REPEAT, WISMS_Sleep_Period);
          }
        }
      return SUCCESS;
    }
```

```
/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
*  Private functions
*/
  void set_CAN_identifier()
  {
    unsigned char temp;
    uint8_t eidl, eidh;
    call EEPROMRead.read(CAN_ID_HIGH, &temp);
    call EEPROMRead.read(CAN_ID_LOW, &eidl);
    call EEPROMRead.read(CAN_ID_MID, &eidh);
    TXB0EIDL_register = eidl;
    TXB0EIDH_register = eidh;
    TXB0SIDL_register = ((temp & 0x03) | 0x08 | ((temp << 3) & 0xE0));
    TXB0SIDH_register = 0x08 | (temp >> 5);
  }

 result_t get_hardware_options()
 {
    call EEPROMRead.read(HARDWARE_OPTIONS, &Options);
    Options = Options & 0x1f;
    return SUCCESS;
 }


 result_t get_config_options()
 {
    call EEPROMRead.read(CONFIG_OPTIONS, &ConfigOptions);
    return SUCCESS;
 }


/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
*   Events
*/

  event result_t EEPROMRead.readDone(uint8_t* param0, result_t param1)
  {
        return SUCCESS;
  }

  event result_t EEPROMWrite.writeDone(uint8_t* param0)
  {
        return SUCCESS;
  }

  event result_t EEPROMWrite.endWriteDone(result_t param0)
  {
        return SUCCESS;
  }

  event result_t RadioSend.sendDone(TOS_MsgPtr param0, result_t param1)
  {
        uint8_t var = param0->ackReceived;
        if(WISMS_autonomous && AutonomousState == autonomous_SEND_DATA ){
            AutonomousState = autonomous_IDLE;
        }
        return SUCCESS;
  }

  event result_t CANSend.sendDone(TOS_MsgPtr param0, result_t param1)
  {
        return SUCCESS;
    }

    event result_t A2D.measureDone(uint32_t reading, uint8_t channel, uint8_t polarity){
          TOS_MsgPtr  msgsp = &TestTxBuf;
          msgsp->data[0] = (reading >> 24) & 0x000000FF;
          msgsp->data[1] = (reading >> 16) & 0x000000FF;
          msgsp->data[2] = (reading >> 8) & 0x000000FF;
          msgsp->data[3] = reading & 0x000000FF;
          msgsp->data[4] = channel;
          msgsp->data[5] = polarity;
          msgsp->length=6;
          msgsp->destpan = 0;
          msgsp->autonomous = TRUE;
          structureAutonomousMsg(msgsp,0x01);     //autonomous reading=0x01;
          call CommControl.start();
          call RadioSend.send(ClusterHead_addr, msgsp->length, msgsp);
          AutonomousState = autonomous_NEXT_CHANNEL;
          post startAutonomous();
          return SUCCESS;
    }
  }
```

```
/************************************************************
 * File: WISMSCommandC.nc
 * Greg Jaman
 * WISMSCommand Configuration
 ***********************************************************/
configuration WISMSCommandC
{
  provides {
     interface WISMSCommand as cmd_GetID[uint8_t id];
     interface WISMSCommand as cmd_ChangeID[uint8_t id];
     interface WISMSCommand as cmd_Measure[uint8_t id];
     interface WISMSCommand as cmd_Autonomous[uint8_t id];
     interface WISMSCommand as cmd_Reset[uint8_t id];
  }
  uses {
    interface WISMS;
  }
}

implementation
{
  components CMDGetID
            ,CMDMeasure
            ,CMDChangeID
            ,CMDAutonomous
            ,CMDReset
            ,A2DC
            ;

  cmd_GetID = CMDGetID.cmd_GetID;
  cmd_ChangeID = CMDChangeID.cmd_ChangeID;
  cmd_Measure = CMDMeasure.cmd_Measure;
  cmd_Autonomous = CMDAutonomous.cmd_Autonomous;
  cmd_Reset = CMDReset.cmd_Reset;

  CMDMeasure.A2D -> A2DC.A2D[unique("A2D")];
  CMDChangeID = WISMS;
  CMDAutonomous = WISMS;

}
```

```
/************************************************************
 * File: CAN.nc
 * Greg Jaman
 * WISMSMessageHandler Configuration
 ************************************************************/
configuration WISMSMessageHandler
{
  uses{
    interface WISMSCommand  as CMDGetID;
    interface WISMSCommand  as CMDChangeID;
    interface WISMSCommand  as CMDMeasure;
    interface WISMSCommand  as CMDAutonomous;
    interface WISMSCommand  as CMDReset;
    interface ReceiveMsg    as ReceiveMsg;
    interface SendMsg       as SendMsg;
  }
}
implementation
{
  components WISMSMessageHandlerM;
  WISMSMessageHandlerM.ReceiveMsg       = ReceiveMsg;
  WISMSMessageHandlerM.SendMsg          = SendMsg;
  WISMSMessageHandlerM.cmd_GetID        = CMDGetID;
  WISMSMessageHandlerM.cmd_ChangeID     = CMDChangeID;
  WISMSMessageHandlerM.cmd_Measure      = CMDMeasure;
  WISMSMessageHandlerM.cmd_Autonomous   = CMDAutonomous;
  WISMSMessageHandlerM.cmd_Reset        = CMDReset;
}
```