A Self-configuring Wireless Sensor Network

Submitted to:
The Department of Electrical Engineering for
fulfillment of the requirements for the Degree of Master of Science
in Electrical Engineering

by:
Dirk de Jager

University of Cape Town, September 2004

Declaration

I declare that the following work is my own. The references refer to words and ideas of other people. This thesis has never before been submitted for any degree or examination to any other University in the world.

signature removed

Signature of Author

Cape Town
September 17, 2004

Abstract

This thesis covers the investigation and implementation of a self-configuring wireless sensor network.

The investigation covers early network systems to modern wireless networks. Several wireless networks were compared, with Bluetooth being selected as the implemented network technology and the Zigbee 802.15.4 network standard being selected for future implementations and developments.

The Bluetooth Protocol is described. An investigation into several different open source protocol stacks was made with the BlueZ protocol stack being selected as the implemented Bluetooth stack.

Wireless development platforms were also investigated and discussed, with the Axis 82 Development platform being selected as the implemented development platform. A Motornostix Canary was connected to the development platform and used as the sensor input for the network.

The Network was implemented and tests were performed and described. Conclusions are made and possible future work is given.

Acknowledgements

I would like to acknowledge the following people:

Prof J. Tapson
Prof G. de Jager
Assoc Prof K. de Jager
Mr S. Anand
Mr A. Rodzevski

for their help and support throughout this thesis.

Contents

List of Figures

List of Tables

Glossary

ACL: Asynchronous Connectionless Link

BNEP: Bluetooth Network Encapsulation Protocol

CVS: Concurrent Version System

DSSS: Direct Sequence Spread Spectrum

IEEE: The Institute of Electrical and Electronics Engineers

IPC: Inter Process Communication

FHSS: Frequency Hopping Spread Spectrum

GN: Group ad-hoc Network

HCI: Host to Controller Interface

IrDA: Infrared Data Association

ISM: Industrial, Scientific and Medical

ITU: International Telecommunication Union

L2CAP: Logical Link Control and Adaptation Protocol

LMP: Link Manager Protocol

MIPS: Million Instructions Per Second

MCM: Multi-Chip Module

MMU: Memory Management Unit

NAP: Network Access Point

OFDM: Orthogonal Frequency Division Multiplexing

PAN: Personal Area Networks

SDP: Service Discovery Protocol

RISC: Reduced Instruction Set Computer

USB: Universal Serial Bus

WPAN: Wireless Personal Area Network

WSN: Wireless Sensor Network

Chapter 1

Introduction

1.1 Subject

This thesis contains a study of presently available wireless technologies, along with future and upcoming wireless technologies that are being designed. It also describes the design and implementation of a self-configuring wireless sensor network using an off-the-shelf communications protocol.

1.2 Background

The concept of using an electronic device, a sensor, to extract information from a real-world environment is of great importance in modern day society. From receiving weather information to monitoring large electrical generators, to counting people entering a supermarket, modern society uses sensors in every modern trade.

1.2.1 Industrial Sensors

Industry is highly reliant on sensor technology. All machinery, as well as industry outputs, need to be closely monitored to ensure the stability and control of the overall system. Human monitoring and observation have become both inadequate

and expensive and therefore electronic sensors, either analog or digital, have become the common-place tool.

In the industrial environment, sensors are generally found under unusually harsh conditions. Such harsh industrial conditions can be due to a multitude of factors: radical temperature, high pressure, intense vibration.

Due to the nature of industry and safety standards, sensors are therefore required to operate over extended periods under minimal maintenance, requiring them to be robust.

Analysis of the vulnerable elements in the electronic sensor system reveals the sensor's communication systems between the physical sensors to be a weakness. Communication cables are prone to wear as they are often on continuously moving machinery.

Industrial environments need to adhere to extremely stringent safety regulations. These requirements also apply to positioning of cables and wires. This could require a sensor which is five meters from a base station or other sensor, to be wired with twenty-five meters of cable.

This extra expense could be extremely costly if one were to install many sensors. The expenses are amplified even more as the installations have to be carried out by safety qualified engineers to ensure compliance with the standards. This is where wireless sensors come into their own.

1.2.2 Wireless Sensors

Wireless communication systems seem a suitable solution to overcome the problems inherent to the traditional wired approach. Two major problems are associated with wireless solutions: cost and configuration.

Present wireless systems are extremely costly. Manufacture of wireless systems is still on a reasonably small scale.

Current wireless systems also require a large amount of configuration to ensure system functionality. This is mainly due to nodes in the network sharing the same bandwidth.

Recent and continuing advances in wireless systems have brought about many different standards in wireless communications. These advances anticipate that the cost problem associated with these systems will be solved in the near future as manufacture of units increases and technology develops.

1.3 Objectives

The objectives of this thesis are to investigate the present wireless systems available, to implement a self-configuring wireless sensor network and to establish a direction towards which future wireless technologies will move.

1.4 Scope and Limitations

This thesis was started in 2002, when many of the current(2004) technologies on the market were not available. Present technology could simplify some of the complications encountered in this thesis.

Wireless technologies in the 2.4Ghz band were only considered as this is one of the Industrial, Scientific and Medical Bands - frequency bands in which transceiver equipment is freely used throughout most of the world. This particular frequency band has many attributes which will be discussed in chapter 3.

A sparse sensor network was implemented, as the sensor units were costly and limited funds where available.

One of the major limitations in this thesis, is the majority of references being online references as opposed to journal written or paper-based references. This is due to most of the work being done on this project having a highly dynamic content. Published work on many of these topics had not appeared in archival journals at the time of this project.

1.5 Plan of Development

Chapter 2 shows the progression of networks towards the network model that we aspire to today. It covers the basic description of earlier network models including their strengths and weaknesses. This chapter will also cover the models that are currently used and the models that current technologies are pointing towards.

Chapter 3 shows the rationale behind the selection of the 2.4Ghz bandwidth. It will then proceed to an investigation into the available wireless communication standards using this bandwidth. The selection of Bluetooth as the implemented wireless standard is justified in this chapter.

Chapter 4 discusses the Bluetooth Protocol in more depth and gives an overview of the specification.

Chapter 5 describes the available wireless development platforms available in 2002 and the development platforms used. It also describes the benefits of each and the complications of using the different platforms.

Chapter 6 discusses the different protocol stacks available. Commercial and Open Source protocol stacks are both investigated. Selection of the implemented stack is described in this chapter.

Chapter 7 gives an overview of the Motornostix Canary. The Canary, a device developed for Web-enabled condition monitoring of heavy industrial machinery, will be used as the sensing unit for the system setup.

Chapter 8 describes the software configuration process and the installation of the software modules. Although the system used a Linux kernel, different modules needed to be compiled for the AXIS ETRAX100LX processor. This chapter describes the process and the different aspects of the steps that needed to be taken.

The implementation of the self-configuring wireless sensor network is described in chapter 9.

Results are presented in Chapter 10 and conclusions in Chapter 11.

Recommendations and future work are presented in Chapter 12, showing the directions towards which continuing projects could head.

Chapter 2

Evolution of Network Standards

2.1 Networks

2.1.1 Network Basics

Many different networks exist around the world. From a group of friends knowing each other, to a telephone system, to the Internet, to the functioning of the human brain, these systems can all be called networks. A network is simply an overall system consisting of two components: Nodes and Links.

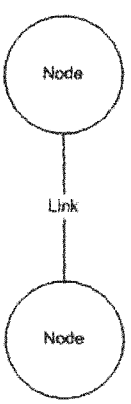

Figure 2.1: Two nodes connected with a link

A single node is a structure which can or cannot be joined to other nodes via a link. A link is also generally referred to as an edge or a connection. Networks consist of at least two or more nodes and at least one link. There is no upper

limit to the number of nodes, except that it must be a finite integer. When all the nodes are connected to all the other nodes by links, we have reached the maximum number of links allowed. This is called a globally coupled network [42].

Mathematically, the limits of a network are:

$$nodes \geq 2$$
$$links \geq 1$$
$$nodes < \infty$$
$$links \leq \frac{nodes(nodes-1)}{2}$$

Globally coupled networks allow for nodes to communicate directly with all the other nodes in the network. This is an optimal connection scheme in terms of speed, signal quality and network robustness.

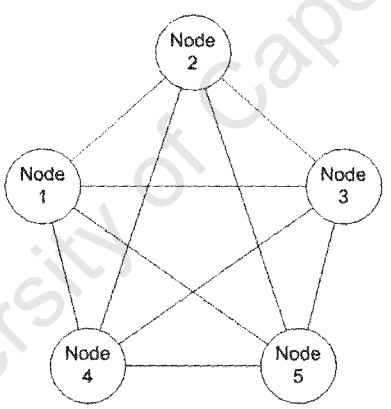

Figure 2.2: Order 5: Globally coupled network

The network is optimal in terms of speed, since assuming that all the links are of the same quality, it will take the shortest time to communicate from one node to another. Similarly, signal quality is ensured by error probability being kept at the minimum. The network is regarded as robust because if a link is removed between two nodes, the network has ample different pathways to link through to the node by taking a route which is connected via another node.

The disadvantage of a globally coupled network is the number of links. For any $O(n)$ nodes, one needs $O(n^2)$ links. Realistically, this is both irrational and requires a very large memory overhead for a large implementation of a network system.

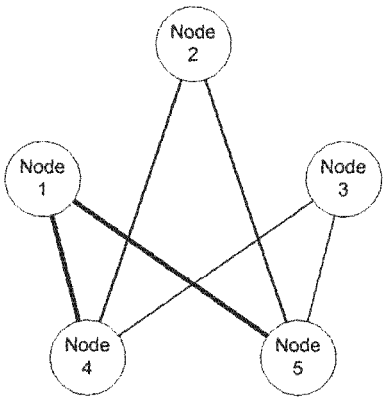

Figure 2.3: Robustness of globally coupled network: Three alternate routes between Nodes 4 and 5

This has lead to real-life networks being analysed to see how networks function and operate. From years of analysis, several forms of networks have become building blocks of networks and network theory. These can be seen in figure 2.4.

These network structures are self-explanatory ranging from ring networks, through tree networks, to star networks, to bus networks. These networks can describe real-world networks, such as a circles of friends, Family trees, Ethernet switches and many more.

Most of these networks have been implemented with wire, and the IEEE formed sub-groups of their IEEE 802 LAN/MAN network standard to represent their different topologies and structures.

The improvements in research and implementations of networks, have led to the improvements of nodes and links of networks. This has also led to the requirements for modern links to use much higher data-rates, more inexpensive materials, less deterioration in communications, and larger distances between nodes.

One of the most successful solutions to the links problem has been the optical fibre. Optical fibres can transmit and receive information at the speed of light, while degradation of the signal is minimal. The cost of an optical fibre is by material much less expensive then a conductive medium. Although optical fibre is an excellent solution to modern network link requirements, the actual medium still costs money whereas a technology using normal air as a medium would eventually be much less expensive.

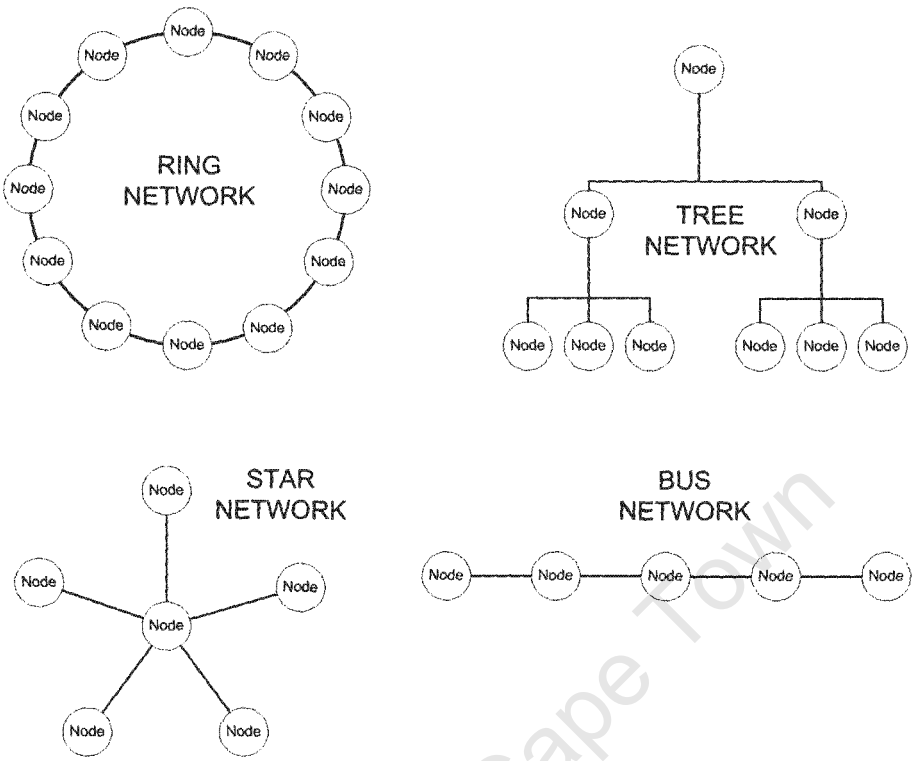

Figure 2.4: Common Network structures

2.2 Wireless Networks

The use of using radio-frequencies to transmit information through space has been at the fore-front of much recent research. Connecting nodes through space requires an understanding of the concepts and implications behind these wireless networks.

2.2.1 Describing Wireless Networks

Although much can be learnt from wire-based technology, wireless networks have very different requirements, properties and constraints.

Wireless network *nodes* are exactly the same as normal network nodes, but the *link*s should be modeled differently. A wireless network link's most predominant feature is that instead of being a solid wired link, it covers a mostly spherical area. Although spherical, a two-dimensional representation fulfills the majority of needs in describing these wireless networks and such a 2-dimensional repre-

sentation can be seen in figure 2.5.

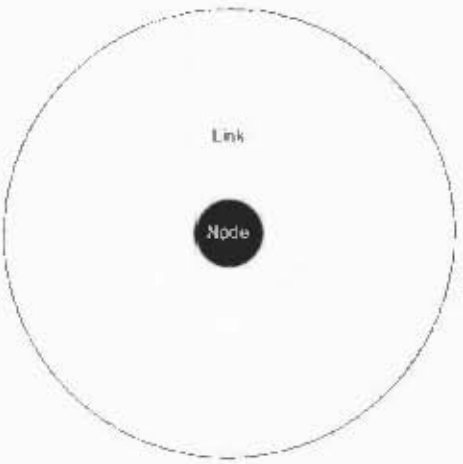

Figure 2.5: Wireless network Node and Link(disk)

Figure 2.5 shows that the links can be referred to as disks. A unit disk is a disk with a diameter of one. A link exists between two nodes if the maximum distance between the nodes is one. *Unit-disk graphs* [26] as referred to by Alzoubi, Wan and Frieder [33] spatially describe important properties of wireless networks - areas of signal strength and redundancy.

From a wireless network described using unit-disk graphs, like figure 2.6, one can see that a well connected area is darker than a lightly connected area. More connections allow for better scope of signal strength as well as more redundant nodes to use.

Because there is no physical link between two nodes, a wireless network's structure or topology must either be self-realised (or *self-configured*) by the network or manually coded in. From the world's experience in wired networks, manually coding networks proves to be highly complex and error prone for medium to large networks, and is therefore not a recommended option.

Assuming that a network comprises of equal nodes, for it to have the ability to configure itself requires intelligence. Since the nodes should all be equal, (as having one more intelligent controlling node requires manual placement to ensure coverage of all the other nodes) the individual nodes all require intelligence.

If the nodes are intelligent, they can self-organise themselves into a network topology allowing for quick and effective transmissions around the network. The

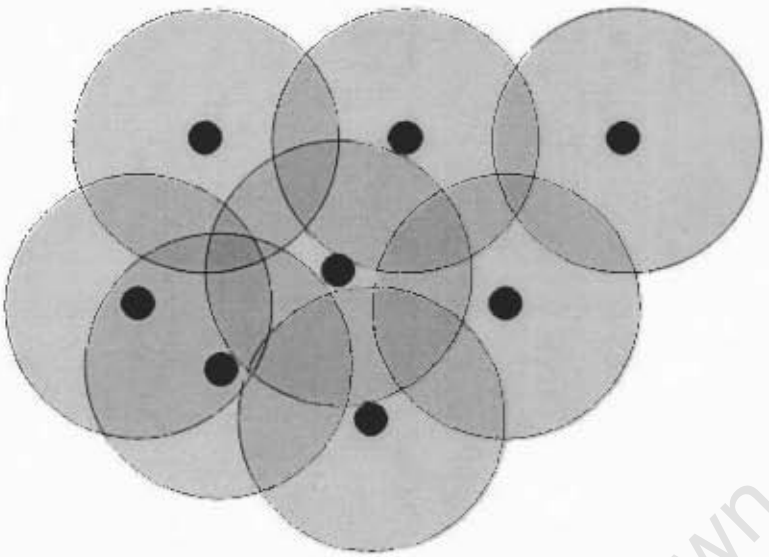

Figure 2.6: A Wireless Network using unit-disk graphs showing connections by overlapping disks

optimal self-organising network algorithm is still being researched. Current research involves using different routing algorithms including minimum spanning trees (MST) [3] and probabilistic routing [28].

2.2.2 Wireless Sensor Networks

Sensors are devices that convert physical variables into electrical variables. Sensors are used for numerous applications, from military battlefield target acquisition systems, to industrial monitoring of machines to home-surveillance.

Due to recent technological developments and advancements, embedded computational systems can be placed in sensors giving them intelligence. An Intelligent Sensor can process some of the electrical variables itself and transmit only important variables and relevant data.

One of the most important criteria of a wireless sensor, is the power source. Due to its wireless nature, the sensor requires either a form of a battery or a small power-source generation device to be able to communicate with the rest of the network. This requirement puts heavy restrictions on the rest of the sensor's capabilities.

Wireless sensors need intelligence. Not only for analysing and transmitting relevant data, but also for self-organisation and routing to find out where to send the information. If the information is sent in all directions, a network greater than around 5-sensor nodes, will be flooded with network traffic and most transmission of information will be lost due to network collisions.

If a new sensor-node is added to the network, the sensor network should allow for inclusion without manual configuration of the whole network as this can become highly complex and it may be difficult to trace problem areas. Removal of a node is also a highly important point to address, as this could easily happen from a node failure or the node falling into *sleep* mode for power conservation. These factors require the sensor-network to rely on a highly dynamic network structure.

Routing packets efficiently and effectively through a wireless sensor network is also an important criterion. For a large number of nodes, if position and transmission paths are not known, transmitting information along the shortest route can be highly complex.

With current development areas focusing on small sensor-nodes in dense distribution networks, most of the current research has gone into efficient routing algorithms in dynamic networks and efficient power sources for these networks.

The current model for a wireless sensor network is described in figure 2.7.

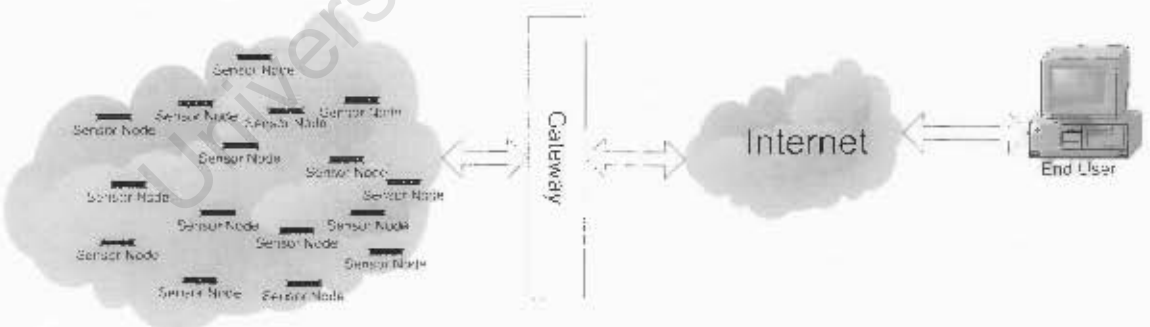

Figure 2.7: A Wireless Sensor Network in place with the Internet

Many wireless sensors are connected through some form of gateway to the Internet and can be visualised as either a whole network unit or individual nodes. The end-user or server then views the information across the Internet.

2.2.3 Selection Criteria of a Wireless Sensor Network Development Platform

As wireless networks are only recently becoming commercially viable solutions for many new applications, development platforms on which to develop new products are costly. As many international standards already exist for all parts of wireless sensor systems, creating one from scratch is therefore redundant. Chapter 3 discusses the different wireless *links* available, and the one which was selected, while Chapter 4 discusses the implemented link. Chapter 5 discusses the selection of a *sensor-node* development platform and the implemented platform.

Chapter 3

Investigation into Wireless Standards

3.1 Earlier Wireless Systems

Wireless connections have been around for many years. Early devices were connected with wireless modems implemented with RS-232 interfaces. These devices would transmit on a single band using narrow-band frequency modulation channels [43].

Unfortunately, using only a single band allows only two devices to communicate at a time. This creates a problem for multiple devices operating in a local environment as interference from one device will destroy packets from another device.

This led to the need of creating communication channels which "share" a bandwidth allocation. The solution to this is "Spread Spectrum" communication. Spread Spectrum techniques spread the required signal over a wider than signal required frequency band, thus creating a signal which resembles noise. This "noise-like" signal is hard to detect, intercept or demodulate. Because of these reasons, Spread Spectrum techniques allow many different devices to communicate with each other over the same bandwidth allocations. Spread spectrum signals must have the following characteristics: the bandwidth of the transmitted signal must be much greater then the data signal bandwidth, and the signal

being transmitted must have some function applied to it which does not contain any transmitted information.

The two most common Spread Spectrum Modulation techniques are Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping Spread Spectrum (FHSS) [15]. Each of these techniques have advantages and disadvantages depending on the application.

FHSS was first patented by Hedy Lamarr and George Antheil in the Lamarr-Antheil patent of 1942 [12]. It is still used today in many military applications. FHSS implements a system of narrow bandwidth signals which jump or *hop* around at a much higher bandwidth in a pattern. The receiver can then demodulate the signal by knowing the correct pattern to hop around in. The Lamarr-Antheil patent describes this technique much like piano players playing a duet.

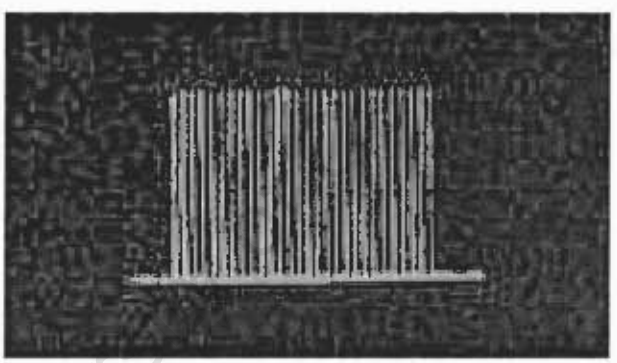

Figure 3.1: Constant Energy Distribution across Frequency Band of Frequency Hopping Spread Spectrum as seen from a spectrum analyser [15]

The spectrum analysis of FHSS shown in figure 3.1 shows the individual narrow band frequencies spread across the entire band being utilised. This pattern closely resembles white noise spread across the frequency spectrum.

Direct Sequence Spread Spectrum does not hop around the frequency band, but rather spreads or smears the signal over the whole band at the same time. This is performed by the use of a higher frequency pseudorandom noise signal multiplied to the signal which can be filtered out by multiplying the same pseudorandom noise signal on the receiver side [20].

DSSS is represented in figure 3.2:

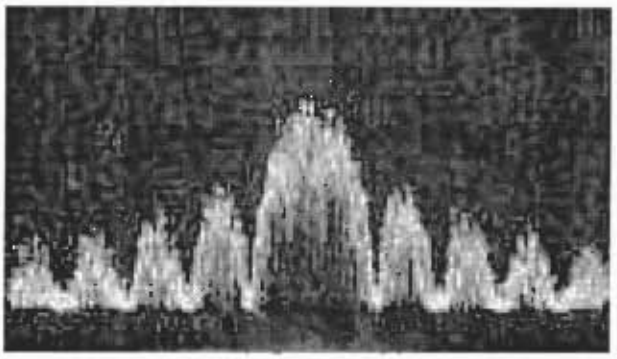

Figure 3.2: Non-Constant Energy Distribution of Direct Sequence Spread Spectrum as seen from a spectrum analyser [15]

As DSSS does not have to constantly re-tune the receiver to the different frequency channels, higher data-rates and shorter delays are generally obtained with DSSS. FHSS, on the other hand, can avoid radio "jamming" as it will hop to another frequency, whereas the DSSS can only dilute the jamming signal by the spread pseudorandom noise signal [4].

As FHSS and DSSS are highly immune to conventional sources of radio-frequency interference, when used in a close environment they can severely impede each other's performance. Initially tending to be more immune to noise, a DSSS signal will radically be destroyed with just a few FHSS transceivers within the same range and frequency band [4]. The FHSS signal will tend to degrade "gracefully" as the signal will slowly appear to worsen. Unfortunately, neither spreading algorithm works well when competing at very close range.

One of the major disadvantages of wireless technologies is the enforcement by governmental authorities to ensure that wireless transmitters of any type must abide by governmental rules and policies. Since these governmental rules are different and diverse in many different countries, a best world-wide standard appears theoretically impossible.

3.2 The Industrial, Scientific and Medical Bands

The Industrial, Scientific and Medical Bands are a set of frequency bands set aside by the International Telecommunication Union (ITU). These bands are

reserved internationally for non-commercial use and designed for radio-frequency equipment in the Industrial, Scientific and Medical field. These bands are set aside for Equipment which use radio frequencies for non-communication purposes - such as Microwave Heating. Transmission in these frequency bands does not require license fees. This has led to the use of these frequency bands for many local wireless networking technologies, even though they have been set aside for non-communication purposes.

Under Radio Regulation S5.150, the ITU recommends that in Region 1 the following frequency bands be reserved for ISM purposes [13].

- 13553 - 13567 kHz
- 26957 - 27283 kHz
- 40.66 - 40.70 MHz
- 433.05 - 434.79 MHz
- 2400 - 2500 MHz
- 5725 - 5875 MHz
- 24 - 24.25 GHz

Although these bandwidths are recommended to be reserved, the ITU does not have the authority to enforce these regulations on national governments. This means that the ITU can only recommend to governmental authorities to reserve these bands for ISM uses. Realistically, most governments do not fully comply with ITU Allocations.

South African radio frequency bandwidth allocations are set by The Independent Communications Authority of South Africa (ICASA) [5]. South African Allocations are the same as ITU region one, except they exclude the 26957 - 27283 kHz frequency band.

One of the more popular ISM frequency bands in Region 2 (Americas) of the ITU, is the 902-928 MHz band. This bandwidth is unfortunately a licensed band in South Africa for Vehicle Tracking systems and RFID tags [9]. Using this

bandwidth for wireless sensor transmissions is thus illegal. Region 2 areas are also restricted to only using the 2400-2483.5 MHz band.

Using a higher bandwidth allows for a higher data-rate. Unfortunately, higher bandwidth limits indoor transmission range as signal reflections off objects are much higher. A higher bandwidth also requires more transistors in the implementation of the transceiver, and is thus much more expensive. Therefore a compromise must be made for increasing frequency (for data-throughput) to decreasing frequency (for immunity to reflections, and therefore an increase in range).

The 2400 - 2500 MHz frequency spectrum seems ideal, as it occupies a reasonably low frequency band for transmission in a crowded environment as well as a high enough bandwidth for a reasonable data-rate. This band is also the lowest almost internationally recognised open frequency bands.

With the constant developments of wide-band radio transceivers, higher frequency band radios have become cheaper to implement and develop (Moore's Law [31]). As soon as the costs were sufficiently reduced, consumer products and standards rapidly developed and were implemented.

The first highly anticipated wireless networking standards arrived between 1997 and 1999. Of these, IEEE 802.11 and Bluetooth were the most predominant.

3.3 IEEE 802.11

The following brief overview is based on facts deriving from the Wikipedia [6].

IEEE 802.11 was started as a working group to develop Wireless Local Area Networks or W-LANs. The first 802.11 wireless standard was released in 1997. The 802.11 wireless networking standard allowed for many different modulation techniques and several different operating modes. The 802.11 standard was not a very set and defined standard, and therefore not widely implemented when it was released.

The 1997 802.11 standard, now referred to as 802.11 legacy, had a maximum data-rate of 2Mbits per second and a selection of either DSSS, FHSS or infrared

as a transmission method. Most 802.11 products used DSSS, with no products ever using the infrared method as the IrDA standard dominated the Infrared wireless market. The FHSS and DSSS transmission methods use the 2.4Ghz ISM band.

Later in 1999, the 802.11a and 802.11b standards were released as improvements on the 802.11 (now legacy) standard. 802.11a used the 5GHz ISM band with a maximum speed throughput of 54Mbits/s, while 802.11b used the 2.4GHz ISM band with a now improved maximum speed throughput of 11Mbits/s. The Wi-Fi Alliance was also formed in 1999 to certify interoperability of LAN devices based on the IEEE 802.11 standard.

Although the 802.11a standard was released in 1999, commercial products only started shipping in mid-2001. At 5GHz the 802.11a standard does not provide as great a distance of transmission and reception as the 2.4GHz 802.11b standard. The 802.11a standard was also only cleared for European use in around mid-2002 as the HIPERLAN standard was being investigated in using the frequency [6].

The 802.11b standard enjoyed much more commercial success with the first highly successful implementation of the standard in 1999 with the Apple Airport.

The 802.11g standard emerged in 2003 as an improvement on the 802.11b standard. 802.11g, which is backward compatible with 802.11b, uses *orthogonal frequency division multiplexing* (OFDM) to increase the speed of raw data transfer to 54Mbits/s. Apple was again first to implement the standard in their Airport Extreme product.

3.4 Bluetooth

Bluetooth technology started in 1994, as an investigation into a low-cost, low-power radio interface between mobile phones and their accessories [21]. The name Bluetooth was chosen from the Viking King Harald Bluetooth who united Denmark and Norway in 960 A.D. Similarly Bluetooth technology stands to unite the telecommunications industry with the computing industry.

The first Bluetooth Specification, 1.0 was released in July 1999. This specifi-

cation proved to be to quite erroneous and the specification 1.0B was released by the end of the year. The first products were released in early 2000 and were compliant with 1.0B. From these products, it was obvious that specification was ambiguous and various implementations noticeably different. In the first few months of 2001, the Bluetooth Specification 1.1 was released, which was much clearer and unambiguous. After Bluetooth was accepted as the IEEE 802.15.1 Protocol Standard, Bluetooth Specification 1.2 was released in November 2003. Most current Bluetooth products comply with Bluetooth Specification 1.1 [16], but are moving towards Bluetooth Specification 1.2.

The Bluetooth Specification is discussed more in detail in Chapter 4.

3.5 IEEE 802.15

Most of this section is derived from [31].

The Wireless Personal Area Network Study Group (WPAN) was started by the 802.11 working group when they realised the need for complementary standards of low-complexity, low-power consumption networks in the Personal Operating Space (POS) [25]. The Study Group was formed in March 1998. One year later on the 11 March 1999, the IEEE 802 LMSC (LAN and MAN Standards Committee) upgraded the WPAN study group into the IEEE 802.15 Working Group.

The IEEE 802.15 working group currently consists of four sub-task groups, namely: 802.15.1, 802.15.2, 802.15.3 and 802.15.4.

IEEE 802.15.1 is derived from and complies with the fundamental Bluetooth 1.1 specification [22]. The IEEE 802.15.1 was accepted by the IEEE standards board on the 15th of April 2002.

IEEE 802.15.2 describes recommended practices for Coexistence in Unlicensed Bands. Specifically, 802.15.2 was created to allow mechanisms in the network standards to facilitate 802.11 networks to exist with 802.15 networks without signal interference [7].

IEEE 802.15.3 was created for High-Rate WPAN Multimedia and Digital Imag-

ing equipment. The draft standard is already complete, and the maximum speed for this standard is 55Mbit/s.

IEEE 802.15.4 was created for Low-Rate WPANs. This standard is designed for use by automation applications, sensors and interactive toys. The IEEE 802.15.4-2003 Standard was completed in May 2003. The Standard only covers the lower Physical and Media Access Control layers. Higher layers for this standard have been developed by the Zigbee Alliance (An Organisation formed by a consortium of companies interested in manufacturing and implementing the Standard).

3.6 IEEE 802.15.4 and The Zigbee Alliance

One of the main niche markets that the IEEE 802.15.4 standard was designed for is Wireless Sensor Networks. The standard allows for multiple frequency band usage, specifically the 868.0 to 868.6 MHz Band (European Standard), the 902.0 to 928.0MHz band (American Standard) and the 2.40 to 2.48 GHz Band (International Standard).

Many of the problems associated with wireless sensor networks (see Chapter 2.2.2) have been corrected by the features of the IEEE 802.15.4 specification.

To minimise power consumption, IEEE 802.15.4 was designed to support very low duty cycles. This means that the transmitter and receiver can be inactive for over 99% of the time they are working. The standard also allows for extremely short network synchronisation packets (544 μs see [31]) called beacon packets. The time between these packet transmissions can also be adjusted to very long intervals (over 4 minutes). The standard has a mode where synchronisation is not required (in star network topologies) allowing for almost indefinite battery life.

The standard uses different modulation schemes in the different frequency bands ensuring low power consumption while maintaining relative implementation simplicity. The modulation schemes are further spread using DSSS. The standard also implements half-duplex communication, so that both receiver and transmitter do not have to be on for communications, thus reducing power consumption even more.

For security, the lower layers of 802.15.4 can use any of seven defined security suites all using different feature combinations of the Advanced Encryption Standard (AES) Algorithm. The seven different suites allow for different levels of encryption, integrity and freshness. Encryption is used for the encoding of the data information. Integrity is used for the data packets not being modified or manipulated by outside nodes, and freshness is a unique key to identify the current communication session.

The Zigbee Alliance is a non-profit industry consortium creating higher-level protocol layers for the IEEE 802.15.4 standard. Specifically they are focusing on different network topologies and data-security for multiple devices. The higher level protocol layers are referred to as application profiles [24].

The Zigbee Network layer makes use of several network topologies, namely Star, Cluster-Tree, and Mesh. This allows for very complex network structures. There are three different node types in the Zigbee Network. These are the *network Coordinator*, the *full-function device* (FFD) and the *reduced function device* (RFD) [41].

The network coordinator is the most powerful device and requires the most memory and processing power. It controls the whole network and therefore requires the most memory to locate the devices, and more processing power to route the addresses.

A full-function device implements a complete set of the lower level services of the IEEE 802.15.4 standard, allowing it to act as a network coordinator (if required) as well as a network device to simply send and receive information. The full-function device is ideally suited as a network router to route packets between RFD's and the network coordinator.

The reduced function device has a small set of the low level services, allowing it to typically be a simple information gathering point, connecting the network to a real world variable. The RFD's were designed to allow for exceedingly simple nodes, such as light switches.

Each IEEE 802.15.4 device has a 64-bit IEEE assigned identifier address, allowing for an almost infinite amount of devices participating in a network. This long device identifier can be swapped for a shorter 16-bit identifier at the start of the network setup to allow quicker communication.

3.7 Discussion and Selection of Wireless Standards

Wireless Technologies are currently rapidly changing and evolving to fill specific and unique markets. There will not be one wireless technology for all network systems for a very long time. Focusing in on the requirements and needs of the networking problem is of high importance in selecting the specific wireless networking standard.

Currently the best upcoming solution for developing a wireless sensor network is the IEEE 802.15.4 Standard. This is because the IEEE 802.15.4 standard has been designed specifically to solve the problems associated with wireless sensor networks. Unfortunately at the time of this work, the standard had not yet been ratified and no commercial products had yet been released.

The 802.11 Standard was not selected for an implementation because it required too many system resources, and was specifically aimed at high-speed local area networks (LAN's).

The Bluetooth standard, which aimed at both low-power and many types of implementation, proved to be the most viable standard, and was thus selected as the network link for the implementation of a self-configuring wireless sensor network.

Table 3.1 gives a brief overview of the current technologies available and the seemingly linear evolution from complexity to simplicity. A point to note is the IEEE 802.15 Working Group is not meant to be a competing standards group to IEEE 802.11, but rather a complementary wireless standards group to existing wireless technologies. The 802.11 and 802.15 standards should be seen as a suite of protocols designed for solving large-scale wireless networks to small scale wireless networks, with data-rates, range and complexity decreasing towards 802.15.4.

Table 3.1 shows an interesting anti-intuitive trend of decreasing complexity and bandwidth with an increase in time. With the abundant computing power available today, it is clear that there still exists a need for focus, and attention to simplicity for the requirements of a wireless network. The first 802.11 stan-

	Wi-Fi IEEE 802.11 WLAN	Bluetooth IEEE 802.15.1 WPAN	Zigbee IEEE 802.15.4 WPAN
First Standard released	1997	1999	2003
Range	100m	10m - 100m	10m - 100m +
Date Rate	2-11 Mb/s	1 Mb/s	$\leq$ 0.25Mb/s
Complexity	High	Medium	Very Low
System Resources	1MB+	250KB+	4KB - 32KB
Battery Life (days)	.5 - 5	1 - 7	100 - 1000+
Power Consumption	400+mA TX 20mA standby	40mA TX .2mA standby	30mA TX 3μA standby

Table 3.1: Comparison of different wireless technologies

dard which allowed for Infrared technology as well as two methods of Spread Spectrum techniques was very unsuccessful because the focus was too broad. Bluetooth, which initially tried to be a universal cable-replacement technology, was very unsuccessful in this attempt as it tried to be too much. As individual requirements are analysed and decided on, a better solution can be obtained and implemented. The Zigbee Standard will attempt to fill the Wireless Sensor Network requirements, and therefore the ideal choice for a wireless link technology. Unfortunately, at the time of this thesis, the Zigbee standard was not fully developed. Bluetooth was selected as the implemented wireless link, as it provided the closest solution to the requirements for a wireless sensor network.

The next chapter describes the details of the implemented wireless technology, Bluetooth.

Chapter 4

Bluetooth

The Bluetooth Specification is written and controlled by a group of companies which are referred to as the Bluetooth Special Interest Group or Bluetooth SIG. The latest version of the Specification is version 1.2 and was released in November 2003. As the majority of the work for this project was done before this, this version of the Bluetooth Specification will not be discussed in the following Chapter.

Most of the following Chapter is derived from the Nokia Bluetooth Technology Overview [16] and the Nokia Bluetooth Protocol Architecture document[39].

4.1 Specification Overview

The Bluetooth System is divided into several different system blocks. The blocks can be visualised in a protocol stack. The stack comprises of different layers, each of which each protocol adds functionality and abstraction, allowing the highest layer to send and receive very simple commands to achieve very complex tasks.

The Bluetooth protocol stack is divided into two separate stacks: the Hardware Stack, known as the Bluetooth Controller, and the Software Stack, also referred to as the Bluetooth Host. The controller and the host communicate with each other through the Host Controller Interface (HCI) layer.

The next sections will describe the Hardware and Software Layers.

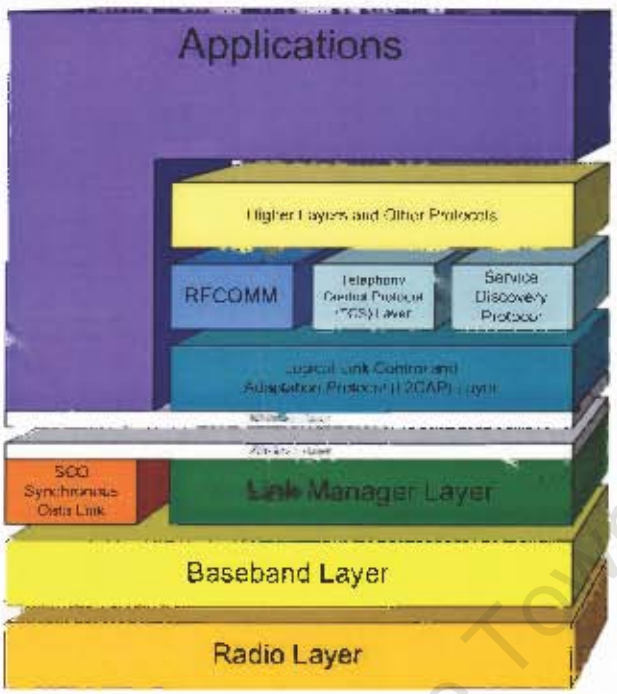

Figure 4.1: The Bluetooth Stack - System Blocks

4.2 The Bluetooth Controller

The Bluetooth Controller comprises of three main components. The Bluetooth Radio (Radio layer), the link controller (Baseband layer) and the Link manager (Link Manager layer). The Host to Controller Interface (HCI) sits on top of Link Manager layer allowing for communication to the Software Stack. The HCI allows for a command interface to the Link Manager, Baseband and Radio layers. The HCI layer can therefore also sit above higher layers, as it only provides a link between software and hardware and provides no real abstraction in terms of the software stack.

4.2.1 Radio Layer

The Bluetooth Radio uses FHSS in the 2400-2483.5 MHz ISM frequency band for its over-the-air communication. This bandwidth is divided into 79 different 1.0 MHz channels. The remaining frequency is used as a lower guard band of 2MHz and a upper guard band of 3.5 MHz. This is to comply with different countries out-of-band regulations.

The Bluetooth Radio can be of different power classes as described in Table 4.1. The different power classes can implement power control for minimum interference and device power consumption.

Power Class	Maximum Power Output	Nominal Power Output	Minimum Power Output
1	100mW (20dBm)	N/A	1mW (0dBm)
2	2.5mW (4dBm)	1mW (0dBm)	0.25mW (-6dBm)
3	1mW (0dBm)	N/A	N/A

Table 4.1: Power Class of Bluetooth Radios

4.2.2 Baseband Layer

The Baseband Layer is responsible for most of the Network architecture behind Bluetooth.

A Bluetooth Network connection is called a *piconet*. In a piconet, there must be at least one master, while all the other nodes act as slaves. One master can control up to a maximum of seven active slaves. The slaves cannot communicate with each other, they can only communicate with the Master. These connections are called point-to-point and point-to-multipoint connections. In a piconet the physical channel is divided up between all participants. This means that the maximum bandwidth is shared among all the nodes.

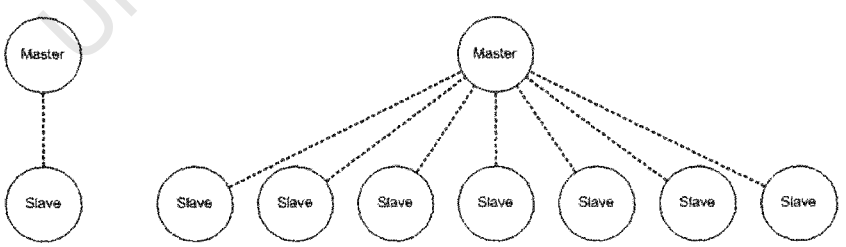

Figure 4.2: Two Piconet examples: a point to point connection and a point to multi-point connection

Although a master can only communicate with seven active slaves, many more *parked* slaves can remain synchronised to the master, but not partake in the communication. These parked slaves can join the network to become active at a

later stage without any connection establishment procedure to the piconet. The parked slaves are also controlled by the Master node.

A Master is defined during the connection establishment procedure. The device which initiates the connection is the Master while the other device(s) are the slaves.

The physical channel frequency-hopping algorithm used to communicate between a Master and its slaves is set by the 48-bit Master's Bluetooth Identity Address and the Master clock. This address is comparable to a Ethernet wired networks Media Access Control or MAC address, and is assigned by the IEEE. The Master will also allocate time-slots and data-rates for the slaves to transmit information to the Master.

A slave can be a slave to more then one piconet, and a slave can also be a Master of another piconet. This multi-piconet network is then called a *scatternet.* Scatternet support in modern Bluetooth devices is limited.

Baseband Protocol

All data transfers are packet based. A packet is made up of three parts: an access code, a header and a payload (Data). The Access code consists of 72 bits, and the header 54 bits, while the payload consists of a variable bit-length of 0 to 2745 bits.

Two types of communication links are defined in the Bluetooth Specification. These are Asynchronous Connectionless (ACL) Links, primarily for data transmission, and Synchronous Connection Orientated (SCO) links for point-to-point simultaneous voice communication. SCO links are mainly used for voice communications and therefore will not be discussed further.

ACL Links support data-rates of up to 723kbps in a single direction with 57.6kbps in the opposite direction. ACL links also have support for broadcast transmissions thus allowing for a Master to communicate the same information to all of the slaves.

ACL links also provide error correction and error checking schemes for operation in crowded and noisy environments. The three methods used are Forward Error

Correction (FEC), Automatic Repeat Request (ARQ) and Cyclic Redundancy Check (CRC).

ACL packets are sent in time slots of size either one, three or five slots. Using one time slot packets, the maximum transmission speed is 172.8kbps while using five slot packets the maximum one-way transmission speed is 723kbps, while the reverse speed is 57.6kbps. Most packets contain a 16-bit CRC code. Retransmission of packets is performed if there is no acknowledgement of packet reception.

4.2.3 Link Manager Layer

The link manager layer is responsible for establishing connections for the Bluetooth device. The Link Manager is controlled through the Link Manager Protocol (LMP). The connection establishment procedure is controlled by a state machine. The two major states are *standby* and *connection*. These two states describe the Bluetooth device when it is at rest waiting to make connections, and when the device is in a connected state (already in communication with a device).

In the standby state, the Bluetooth device wakes up and listens for two different types of connection request messages every 1.28 seconds. When the device wakes up and listens, it only listens on one of the 32 defined listening frequencies. In the connection state, the Bluetooth device can operate in several different modes.

To changeover from the standby state into the connection state two procedures have to be followed before a connection can be made. These are called *inquiry* and *page* procedures.

The inquiry procedure, performed by the master, finds devices which are available and returns the device address and system clock. This allows the master to page other devices and establish a connection with the slave device. Without the device address and system clock a page procedure can not be completed. Once a connection is terminated, the master can reconnect without a inquiry process.

As the device in standby mode wakes up only every 1.28 seconds and listens on a single channel, finding devices with an inquiry is time consuming at almost 40 seconds. When a master sends a page message to a specific slave device, it sends the page message once on 16 simultaneous frequencies and if there is no response

it sends the same message on the 16 other frequencies. This sets the maximum connection time to a relatively low 2.56 seconds for devices which know each other's addresses.

Once a device is in a connection mode, it operates in one of four modes: Active, Sniff, Hold and Park. Active mode is where the device is participating in communications. Sniff mode is where the slave device is put to listen very infrequently on a specific time slot. Hold mode allows for a device to participate in other piconets (thus allowing for scatternets). Park mode allows a slave device to remain synchronised to the master, but not participate in communications. This allows for more than seven devices to communicate to a Master in the same piconet.

4.2.4 Logical Link Control and Adaptation Protocol Layer

The *Logical Link Control and Adaptation Protocol* (L2CAP) layer is responsible for managing links from upper layers over the baseband. It multiplexes different data-streams over the physical ACL links and can provide connection-orientated or connectionless data services to higher layers. The L2CAP layer also provides segmentation and re-assembly of packets allowing higher level protocols to send and receive L2CAP packets of 64 kilobytes.

The L2CAP layer can almost be seen as the critical link between the higher layers and the Bluetooth Controller. L2CAP is the lowest link level a user should use when creating a Bluetooth connection to another device. Preferably the user should use the virtual serial port (RFCOMM) Protocol Layer to connect to another device.

4.2.5 RFCOMM Protocol Layer

The RFCOMM Protocol is regarded as the cable replacement protocol, as it emulates RS 232 signals over a wireless link (Radio Frequency Communications Port). The RFCOMM protocol can thus support legacy devices which previously used an RS-232 link to communicate with a device. The specific manner in which to create a link is defined in the Serial Port Profile (see Bluetooth Profiles - section

4.2.9).

4.2.6 Telephony Protocol Layer

The Telephony Control Protocol Layer (TCS-Binary or TCS) is s bit orientated protocol. This protocol allows for connections and the control of speech and data calls between different Bluetooth Devices. This would be used to start SCO links to other Bluetooth Devices.

4.2.7 Service Discovery Protocol Layer

The Service Discovery Protocol (SDP) is a very important Protocol. It allows for all devices to communicate to each other which services they have, and also the characteristics of the remote Bluetooth Devices. Devices have to negotiate a common service to both devices before a connection can be established

4.2.8 High-Level Bluetooth Protocols

Several higher level protocols have been included in the Bluetooth stack. These protocols are not mandatory in the stack, but allow for different *usage-models* of Bluetooth technology defined by the Bluetooth Profiles. These protocols are also not Bluetooth specific and have been adopted from other network standards.

As the Bluetooth Standard evolved many new adopted protocols have appeared and have been added to the specification in Errata. These can be seen in the Bluetooth Website [19]. The Bluetooth version 1.1 Protocols above the RF-COMM layer are described in Table 4.2.

4.2.9 Bluetooth Profiles

To allow many different vendors and manufacturers around the world to make their products successfully interact and correctly function together, the Bluetooth SIG developed several usage-models to describe how devices should perform specific functions. These usage-models resulted in a set of Profiles which

Abbreviation	Name	Description
PPP	Point to Point Protocol	Used for Internet communications over RS-232
IP	Internet Protocol	Adopted for Internet Connections
UDP	User Datagram Protocol	Adopted for Internet Connections
TCP	Transmission Control Protocol	Adopted for Internet Connections
AT - Commands	AT - Commands	A set of commands to allow communications with modems and faxes
OBEX	Object Exchange	Adopted from IrDA for communications with hand-held devices
vCard/vCal		Business Card and Calendar Information formats adopted from IrDA
WAP	Wireless Application Protocol	Used for connecting the Internet and telephony based services to cellphones and other wireless devices
WAE	WAP Application Environment	Environment using WAP to render wireless services transparent

Table 4.2: High-Level Bluetooth Protocols

describe how one device should behave when communicating to another device for a specific purpose. These profiles are included in the Specification thus enforcing the usage-models and allowing for seamless interoperability between different vendors.

The profiles are also written describing mandatory and optional feature sets of the profile. By using these profiles, vendors can release their products specifying exactly what profile the product supports.

The Profiles are also layered, thus allowing for a much higher level of device inter-operability. There are three main Profiles, namely [38]:

GAP The *Generic Access Profile* describes the process two devices have to perform to discover each other and establish a connection between each other.

All Bluetooth devices must have this Profile to be able to communicate to another device

SDAP The *Service Discovery Access Profile* describes how a device queries another device to find out which services and protocols the other device has to initiate the specific profile communication.

SPP The *Serial Port Profile* is the simplest communication profile which allows for a device connection which emulates a serial port over a Bluetooth link.

As new uses for Bluetooth develop, so will (and do) the profiles. Currently there are many new Profiles which have been added to the Specification in errata. By consulting the website, one can see which version of the specification support which profiles [19].

4.3 BNEP and the PAN Profile

One of the important profiles which was developed, was the *Personal Area Network* Profile (PAN) used for Internet over Bluetooth. Initially, there were two forms of this Profile, one over the RFCOMM protocol, and more recently a PAN profile over the *Bluetooth Network Encapsulation Protocol* (BNEP) protocol.

The BNEP protocol sits directly on top of the L2CAP layer and allows for common network packets (such as IPv4 and IPX) to be sent and received [14]. By communicating directly through the L2CAP layer, a much greater proportion of the bandwidth can be allocated to data throughput. By using BNEP over Bluetooth devices can be linked to the Internet transparently. BNEP also encapsulates common 802.3/Ethernet network packets.

By creating the PAN profile and the world-wide realisation of the importance of networks within the Personal Operating Space (POS), Bluetooth was adopted by the IEEE 802 LAN/MAN standards committee as the IEEE 802.15.1 Standard - The first IEEE Personal Area Network standard.

The PAN profile has three main components:

NAP The *Network Access Point* is the device which acts as a relay between a

group of Bluetooth devices and a higher level network infrastructure such as the Internet.

GN The *Group ad-hoc Network* is a device which relays network packets between a piconet. No higher-level network is included and the network consists of purely the Bluetooth nodes in the piconet.

PANU The *PAN User* is a client to either the GN or NAP device.

The PAN profile relies only on the GAP profile. Currently the PAN profile does not include automatic network formation, ad-hoc networking of multiple piconets or Quality of Services, but future versions of this Profile would most probably include these topics [34].

The PAN profile and the BNEP protocol was used in this thesis. Chapter 9 describes the use of a PAN profile implementation used in a wireless sensor network environment.

The next chapter describes the selected development platform on which the Bluetooth devices were developed.

Chapter 5

Selection of a Wireless Development Platform

As stated in the beginning of this thesis, a development platform was required to initiate the study and implementation.

The study was done with the Bluetooth Protocol as the favoured Wireless standard. One of the major requirements for the system was low cost, while the system had to remain highly flexible. This chapter describes the development platforms tested and investigated, while a discussion follows at the end.

5.1 The Teleca-Comtec Development Boards

Figure 5.1: The Teleca-Comtec Development Boards

5.1.1 Kit Contents

Two Ericsson development kits were acquired during the initial stages of the project to investigate the possibilities and requirements entailed. The development kits where manufactured by a Swedish company called Teleca-Comtec and licensed under Ericsson.

Each kit comprised of a development board with both a USB and a serial interface. The development board had a Bluetooth radio and a Bluetooth stack up to the HCI layer. No extra hardware was provided.

A software stack was made available with the development boards written by Ericsson. This stack was pre-compiled for the Windows Operating System and written for a Visual C working environment. No other Software was made available for the boards.

5.1.2 Tests

The chipsets used on the boards were the ROK 101 007 Ericsson chipsets. These chipsets were some of the earlier Bluetooth chipsets, and had serious problems and hardware issues. The kits where shown to be unreliable and the software stack provided by Ericsson not fully debugged. Of serious concern was the USB Interface which would not re-initialise after a board crash, limiting development use to the serial port.

We found that the software stack supplied with the board, was not well written and constant major updates where being provided over the Internet which required major modifications of user written software code. The stack was also not Open Source and provided support for only basic Bluetooth functionality.

Using these boards unfolded two main differences in Bluetooth implementations. Generally, commercially available Bluetooth stacks used the Windows Operating System as a basis for a development environment, while Open Source and freely available Bluetooth protocol stacks used Linux as a basis for a development environment. Since cost of hardware and software was a major limitation, Linux was more highly regarded for a basis of a development environment.

5.2 Crossbow Motes and TinyOS

In 1998, The University of California, Berkeley, started a project called "Smart Dust" to create sub-millimetre wireless sensors for military and commercial applications [36] [27]. From this project, they created macro-"motes" which were released commercially by a company called Crossbow.

The Mote kits consist of three different components, a data-acquisition board, a radio and processor board (motes) and an interface board (to a computer). The different components must be plugged and linked together to form a wireless sensor network, with each mote having a data-acquisition board linked to it.

There are many different kinds of data-acquisition boards which can capture several kinds of sensor information such as light, accelerometer readings, magnetometer readings and pressure readings. Some of the devices also have analog inputs to allow for other sensors to be attached to the data-acquisition boards.

There are many different motes as well. The different motes communicate on different frequency channels as well as having different processing power and size. The motes can be purchased for around four frequency bands sets. The sets are 433.1-434.8MHz, 868-870MHz, 902-928MHz, and 313.9-316.1MHz. Since the data throughput is very low, the devices require very little bandwidth. If the devices were required for larger data-throughput, the frequencies used for transmission and reception would severely impede the data throughput.

All the motes use an operating system called TinyOS, which is a very light Open Source operating system designed at the University of Berkeley for wireless sensor networks. TinyOS which was first released in May 2002, has become one of the leading wireless sensor network operating systems in the world. TinyOS, which uses basic C to program software, allows for rapid and efficient development.

The interface boards allow for programming the motes as well as interfacing the network to a single point. The interface board requires a mote radio attached to it to communicate with the rest of the network.

The mote kits are very popular in America and Europe and much research has been done using these kits. Unfortunately, the kits are relatively expensive, and a basic kit allowing one to set up an intelligent network costs $2000, while the

most basic kit which only allows for data-capture of pressure, light, temperature and sound costs $800.

5.3 The Axis Development Board with Bluetooth

Axis, a Swedish company specialising in Network solutions for computer peripherals, created a freely available Open Source Bluetooth Software Stack. The software stack, known as OpenBT conforms to the Bluetooth Protocol Standard 1.1 and was the first ever Open Source Software stack to be released.

To complement the OpenBT Software stack, Axis released a development board using an Axis ETRAX100LX processor and a generic Bluetooth module. This development kit, called the Axis Development board for Bluetooth, is configured as an Access Point(LAP Profile) for Bluetooth Networks and Devices on delivery.

The ETRAX100LX processor can run a full Linux kernel, and is therefore an excellent development tool for users comfortable with the Linux software environment. This also allows for inexpensive development as software licensing is unnecessary.

The board was selected as a development platform in 2002 due to its inexpensive price in comparison to other available solutions as well as its capabilities in terms of processing power and expansibility.

Limitations of the board included a maximum limit of four users connecting to the board and not seven as specified in the Specification.

5.3.1 Features of the Axis Board with Bluetooth

The board features:

- 1 ETRAX 100LX processor
- 1 RS-232 port
- 1 Ethernet 10/100MB port

Figure 5.2: The Axis Development board with Bluetooth

- 1 Class 1 Bluetooth Module
- 4 Megabytes RAM Memory
- 2 Megabytes FLASH Memory

The Axis ETRAX 100LX processor is a feature rich 100MIPS 32-bit RISC processor. The design goal behind the ETRAX 100LX was to create a high-performance, low-cost, easy to implement processor allowing one to put peripheral devices on a network. The ETRAX 100LX was specifically designed for use with Linux and therefore includes a Memory Management Unit or MMU. The MMU allows for native Linux support. This is very different to other embedded processors which require an entirely different version of Linux (uClinux) [11].

The single RS-232 port is a very useful interface as it is simple to use and implement a device to communicate with this port. Although the ETRAX 100LX has two general IO ports providing much more functionality, these pins are not accessible on this development platform. This is unfortunate and limits the extensibility of this development platform

The board has a 10/100MB ethernet port allowing for instantaneous internet connectivity. The ethernet port, which is built in internally in the processor can be configured in software and allows simple modification of the normally hardware set MAC address.

The Bluetooth module included on the development board is connected to the processor via USB. The Bluetooth module is of power class 2 which means it has a nominal power output of 1mW (0dBm) and a range of around 10 metres.

Four megabytes of onboard volatile memory is provided on the board, while two megabytes of onboard FLASH memory is provided for information storage. Although this may seem like excess memory, much of it is used by the Linux kernel, and many useful applications need to be removed in order to make large applications fit on the board's memory.

Since the end of 2003 the Axis Development board with Bluetooth has been discontinued. Since then, the AXIS 82 Development platform with a Bluetooth USB device has been the recommended solution for continuation of similar research.

5.4 The Axis 82 Development Platform

In early 2003, Axis released the Axis ETRAX 100LX MCM or Multi-Chip Module. This Integrated Circuit included all the necessary components and stands by itself as a full computer. The MCM was designed as a full computer for embedding in hardware device peripherals used with modern computers.

Figure 5.3: The Axis 82 Development board

The MCM, which requires only a 3.3Volt power supply and a 20MHz crystal oscillator to function as a full computer, includes the following:

- ETRAX 100LX processor
- 2 Mbytes of Flash memory
- 8 Mbytes of SDRAM memory
- Ethernet Transceiver

This Integrated Circuit which comes in a 27mm by 27mm package allows for integration into almost any hardware system. As an ethernet transceiver resides on the MCM, connecting the MCM to an ethernet connection requires only a UTP connecter (with built in isolation magnetics).

The Axis 82 Development board which was released with the launch of the MCM, not only provided an excellent platform for development, it had the following features:

- 1 ETRAX 100LX MCM 2+8
- 4 Mbytes of external Flash memory
- 8 Mbytes of external SDRAM memory
- 2 RS-232 serial ports
- 1 RS-485/RS-422 serial port
- 2 Ethernet ports (1 Full 100MBit and 1 Realtek 8150 12MBit USB Ethernet)
- 1 USB port
- Full Access to the general IO pins

Using this board allows one to create a multitude of network services simply and efficiently. At a total cost of $295 the board is a very competitive development solution.

5.5 Bluetooth Modules

As a wireless interface for the Axis 82 Development platforms, two Abocom Class 2 Bluetooth Modules with CSR (Cambridge Silicon Radio) chipsets were used in this project. The Modules have a USB interface and are supplied with Windows WIDCOMM software. The modules were inexpensive at a cost of around R400 each.

A single Mitsumi WIF-0402C USB Bluetooth module was used as well. This device proved troublesome with the OpenBT stack, but worked successfully with the BlueZ stack. Several other minor problems were encountered with this module and this module is therefore not recommended for future use.

Figure 5.4: The Abocom Bluetooth Modules

5.6 Comparison of Development Platforms

Of the investigated and tested systems, the platforms described above proved to be the most viable solutions in the minimum price range (as much commercial development is being done in the field, many of the products available are very expensive). Each system had advantages and disadvantages, with the final system being chosen not only on the basis of system benefits, but also the experience of the developing team.

Table 5.1 shows a quick overview of the different systems, and their individual characteristics.

The Teleca-Comtec boards are plain simple Bluetooth Radios. The major advantage of using these boards as a Wireless Development Platform is the previous experience and knowledge gained by previous work having been done with the boards. The Windows-based software stack is simple to use, even though it has minor flaws. The major disadvantages with the boards are that they have no intelligence and therefore require additional hardware to run the software stack. As they are designed for Windows, the hardware required to implement the stack would be quite advanced and therefore also quite expensive.

The Crossbow mote kits seem a very good solution. One of the major advantages of the kits, is that they have been used by many institutions around the world and are regarded almost as the standard of wireless sensor networks development. The kits are also sold in a complete package and arrive as a fully functioning wireless sensor network. Unfortunately, the kits do not have the ability to interface with external sensor devices, except for a very simple analog signal input. This disadvantage, along with the expensive price-range at the beginning of the project rendered the Crossbow Mote Kits as a non-ideal solution.

The Axis development board with Bluetooth had many advantages when looked at as a development platform. It had a simple serial interface for data input from almost any device, along with a Bluetooth module for wireless communications. The Axis development team having also created the first open-source Bluetooth Software Stack made this development platform seem ideally suited for a wireless sensor network.

The Axis82 development board was used as a development platform later in the project as the Axis development board with Bluetooth had been removed from the market and replaced with the Axis82 development board. The Axis82 development platform has even more benefits than the Axis development board with Bluetooth.

	Teleca-Comtec Boards	Crossbow Mote Kit	Axis board with Bluetooth	Axis82 with Bluetooth Module
Cost (2002)	$500	$2000	$500	N/A
Cost (2004)	$500	$900	N/A	$300
Number of Nodes	1	3	N/A	1
Total System Cost (3 nodes)	$1500+	$900	$1500	$1000**
Radio	YES	YES	YES	NO*
Processor	N/A	Atmel ATMEGA 128L	ETRAX100LX	ETRAX100LX MCM
Speed	N/A	~8MIPS	100MIPS	100MIPS
Non-Volatile Memory	N/A	512KB	2 MBytes	6 MBytes
External Ports	RS232 USB	Custom	RS232 Ethernet	RS232 RS485 USB Ethernet
Development Environment	Windows	Windows Linux	Linux	Linux
Operating System	N/A	TinyOS	Linux kernel 2.4	Linux kernel 2.4

* Note: Separate USB Bluetooth Modules are required for the Axis82 Development Platform

** Note: This price does include the extra Bluetooth modules

Table 5.1: Comparison of Development Platforms

Chapter 6

Selecting a Bluetooth Protocol Stack

Many different Bluetooth Protocol Stacks exist. The selections are varied and the implementations are markedly different. This results in substantial incompatibility between different devices, which for the realisation of a standard, can be a serious concern.

Thus, selection of a good, solid Stack implementation is vital for a self-configuring wireless sensor network.

There are presently several commercially available software stacks. These are generally expensive and therefore not considered in this thesis. For an overview of the commercially available stacks, the Bluetooth Weblogs website [10] provides an excellent starting point.

Four open source software stacks are described in this chapter. Each stack was developed with different goals and architectures behind them. To choose a stack, different components are analysed and rating criteria extracted to establish which protocol stack provides the best solution for this thesis.

6.1 Protocol Selection Criteria

Selecting a protocol stack is not specifically just comparing advantages and disadvantages of what the stack has to offer. A comparison of the support and documentation of these stacks is vital as well as the community of support for these software modules. Selecting a poorly supported module leads to neglect in smaller problems, where a well supported stack can provide solutions to larger problems.

Since all the freely available stacks are open source, they are all continuous development projects, rampant with small errors and continuously updated through an open-source CVS repository - SourceForge [23]. Getting the latest version of the code through the SourceForge website and placing questions and troubles on the mailing list is the maintenance and support mechanism for these development projects.

6.2 OpenBT

The OpenBT software stack was developed by Axis Communications in Sweden [1]. The stack is renowned as one of the closest implementations of the Bluetooth Specification 1.1. Having started in April 1999, it was the first Open Source Software Stack available.

OpenBT supports the following Bluetooth protocols:

- SDP
- L2CAP
- RFCOMM

OpenBT also supports the following Bluetooth profiles:

SPP The Serial Port Profile is supported with OpenBT as described in chapter 4.

DUN *Dial up Networking* allows for devices to connect to the Internet via a modem or such device.

LAN The *Local Area Network Access Profile* allows for Internet over the RFCOMM layer. It has been removed from recent Specifications and replaced with the PAN profile.

PAN Limited PAN profile use over RFCOMM was developed for the Axis Developer board with Bluetooth.

Although the stack is well implemented and comprises of the most important Stack Layer implementations, this stack has not been very user friendly or greeted with much support from the worldwide community. The last stable version of this stack implementation was released in October 2001. This means that the implementation has fallen behind the other stacks and is now almost not supported at all.

A newer version of the OpenBT stack for the Axis Developer board for Bluetooth also supports the PAN profile for Personal Area Networks. The PAN profile which is described in chapter 3, is regarded as the new standard profile for Bluetooth network communications (see Chapter 3).

At the start of this thesis, the OpenBT Bluetooth Protocol Stack was selected as the simplest and most supported stack implementation. This proved over time to be a mistake as the BlueZ protocol stack was selected as the official stack of the Linux Operating System and thereby embraced by the worldwide development community as the stack of choice.

6.3 IBM BlueDrekar

IBM BlueDrekar, which was started in July 2000, was not a open source software stack, but it was a freely available software stack for the Linux platform. Due to it not being open source, the stack was not investigated and has subsequently been removed as a freely available stack from the Internet [2]. IBM BlueDrekar was never well supported.

6.4 Affix

The Affix Bluetooth Protocol Stack, which was developed by Nokia Research Group in Helsinki, Finland, is one of the newer freely available stacks [18]. Having been first started at the end of November, 2001 it has gained good support and is now the most powerful freely available software stack.

6.4.1 Features

The Affix stack has a modular implementation allowing for simplicity and re-use. The Affix stack is constantly being updated and continually improved. Having been developed by Nokia the Affix stack is also aimed for use in small devices such as microcontrollers.

The Affix stack also supports many profiles including the following of interest [18]:

- DialUp Networking Profile
- LAN Access Profile
- PAN Profile

Affix is currently the only stack which has some support for the Bluetooth Protocol Specification version 1.2.

The Affix stack comes in two parts: The Kernel based modules - affix-kernel, and the tools, libraries and daemons.

6.4.2 Implementation Details

Implementing the Affix Protocol Stack on Linux relies on a self-configuring installation program. This program does not support the Cris-Axis cross-compile tools and therefore was unsuccessfully ported to the Axis ETRAX100LX board. It has been noted that with modifications, the stack will run on the Axis 82 Development board.

6.5 BlueZ

Started in May 2001, BlueZ became the official Linux Stack implementation in June 2001 [40]. BlueZ is currently the most popular and widely used Open Source stack implementation.

6.5.1 Features

BlueZ supports many protocols including the BNEP protocol [40]. BlueZ also supports the following profiles of interest to this work:

- Dial Up Networking Profile
- Local Area Networking Profile
- Personal Area Network Profile
- Human Interface Device Profile

BlueZ Kernel drivers are supplied with all Linux Kernels from version 2.4.18. This means that implementation into any Linux compatible hardware requires little software configuration. The userspace libraries and tools are supplied off the Website as different tool collections. The current tools also allow the following extended functionality: passcode authentication helpers, *Packet Analysis* tools, *Emulation Tools* and *Firmware loaders* for certain chipsets.

The BlueZ software stack has excellent support for both kernel modules and userspace programs.

6.5.2 Discussion and Implementation

At the beginning of this work, the OpenBT protocol stack was selected as the best stack to use. The stack appeared as the most stable and well developed stack, and was relatively simple and easy to install and configure. At the time, the BlueZ stack consisted of many smaller modules which had to be individually installed and configured for support and overall system working.

The OpenBT stack was also originally developed by Axis, which had a cost-effective development platform available for implementation and testing (see chapter 5). The OpenBT stack was used effectively for simple Bluetooth communications and links on the RFCOMM protocol layer.

As the PAN Profile became more common place, the OpenBT stack could not provide a fully functional and easy to use solution to the changing environment. As the OpenBT stack had not been updated since 2001, another stack option was required.

The selection between Affix and BlueZ was decided upon the best support for the development environment used, and the support for the PAN Profile, as PAN usage was vital to this work. As BlueZ is the official Linux stack, and has implemented a fully functional BNEP protocol (for PAN use), The BlueZ protocol stack was selected as the best protocol stack and implemented in the wireless sensor network.

Implementing the BlueZ stack into the Axis 82 Development platform requires steps which are described in chapter 8.

Chapter 7

The Motornostix Canary

7.1 The Canary

The Motornostix Canary was developed in South Africa by a company called Motornostix [8]. The canary's main design goal was to implement a hardware device to be used in an Industrial environment as a heavy machine monitoring tool and report back device.

The canary connects several sensors and actuators to a central node and then communicates via a bus to a central receiving node. This bus uses a RS-485 communication protocol to communicate with the Master Scanner node which then packages the Information and transmits it over the Internet. The Master Scanner Node is generally a workstation, while the canaries are all microcontroller based sensor-nodes.

Although Canaries are deployed in industry using RS485 cables, the limitations due to cable laying policies and the expense of the cables, require alternative wireless solutions to be found. The RS485 Communication protocol, which uses point-to-multipoint connections, resembles the Bluetooth Architecture, and therefore Bluetooth is a very good wireless solution to implement with the Canary.

The Canary has many actuators and sensors on-board allowing for measurements of temperature, vibration, current, speed and flux.

Figure 7.1: The Motornostix Canary

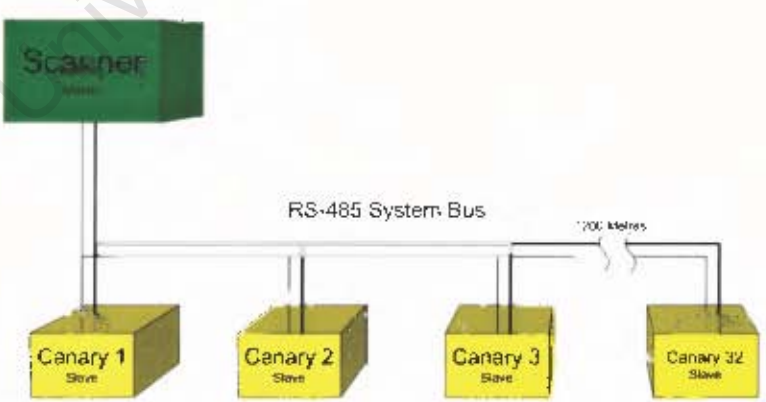

Figure 7.2: The Normal RS485 Canary Configuration

7.2 The Protocol

A Linux library toolkit was written in C for communication with the Canary in this work. The toolkit allows for basic communication with the Canary, including error correction and packet numbering for ordering of the information. The code can be seen in Appendix B.

The Library was created for advanced communications with the Canary. As device communication data retrieval details could not be obtained, the Library was not used in the final implementation, but is provided for future development.

The details of the protocol is proprietary and can be obtained from Motornostix.

Chapter 8

Configuring Software with the Axis Development Board

Setting up the development environment for a wireless sensor network involves initialising and configuring several different devices and software components. As the components were designed with different concepts and thinking behind them, bringing them together can be challenging and troublesome.

This chapter details the different components and modules which were used and modified for the setup of the development environment

8.1 Cross-compiler Tools

As the Axis ETRAX100LX does not run a i386 architecture, a set of tools is required to compile the source code for the processor. These tools are called cross-compilers as they compile software on one processor for another processor. The tools provided by Axis are called the Cris cross-compiler tools.

As the tools are for development purposes and constantly being changed, they are very poorly documented and features are not well described. Hands-on experience with the tool sets provides the best understanding and knowledge of the current tools.

To Compile the kernel for the Axis 82 board, the Cris-Cross Compiler tools ver-

sion 1.56 in conjunction with the Axis Development board utils version 1.92 was used. Following the rudimentary installation instructions allowed for installation into a system. The Linux Kernel 2.4.26 was also used and patched with an Axis patch downloaded from the website.

8.2 BlueZ

The Axis 82 Development board, uses a Linux kernel 2.4.26 which must be patched and modified. The BlueZ library drivers and utils version 2.7 were used in this project. To successfully add the BlueZ libraries and utils to this Linux kernel, several modifications are required to the code for it to successfully compile.

The first modification requires the user to add in the BlueZ kernel mode drivers supplied with the Linux Kernel. Modifying the Makefile in the Linux Kernel and uncommenting the BlueZ kernel mode drivers allows the kernel mode drivers to be installed and added to the Kernel. Uncommenting the Kernel Protocol drivers is also required for the BlueZ protocols to work.

A new makefile had to be written for the BlueZ library files. The Makefile can be seen in appendix C. The Makefile points the configuration scripts at the correct cross-compiling tools, and the output directories to the correct final target directories.

8.3 Linux Bridge Utils

To allow the nodes to relay IP Packets to and from the network, the Linux Bridge Utils had to be compiled and added into the Axis board kernel. By adding minor modifications to the individual Makefiles in all the directories of the source, the bridge utils were added to the kernel.

To get the Bridge Utils to function, Advanced IP Packet options must be selected in the Linux Kernel and IP Filtering should also be selected. Other Kernel level features enable extensive IP level packet control.

8.4 Scripts

To set the scripts to start automatically on the Axis board, the *inittab* file needs to be modified to include the automatic scripts. Commands can be supplied to set the scripts to run only once or respawn if killed. Different run levels are also required to check the state of the board startup.

Chapter 9

Implementation of Self-configuring wireless sensor network

A self-configuring wireless sensor was implemented using 3 AXIS 82 development boards with 3 USB Bluetooth Modules and a single Motornostix Canary. The overall description of the implementation is presented in this chapter.

9.1 Overall Implementation

The implementation of a self-configuring wireless sensor network, depends on several different components which have to interface with each other and work coherently together to function correctly. The major components are sensors, processing and data-capture boards, and a wireless network technology.

To realise the final implementation, several different off-the-shelf solutions were selected for use in this project. The wireless communications technology used was Bluetooth (see Chapter 4) and the sensor used was a Motornostix Canary (see Chapter 7). The processing and data-capture board used was a Axis 82 Development board (see Chapter 5). The wireless Interface used was a Bluetooth Module connected to the Axis board through the USB port.

9.2 System Set-up

The overall system setup was implemented with three nodes. A single Canary was used as the data-capturing tool. The Canary was switched between different nodes to test the configuration abilities of the network.

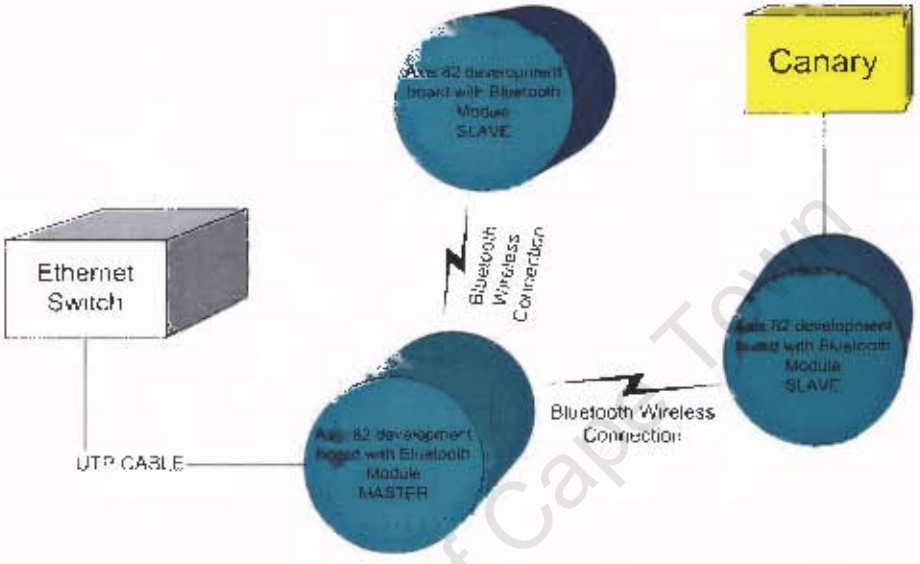

Figure 9.1: The overall system setup

The Network was accessed from computers attached outside of the setup connected to the Ethernet switch. Although the network was not tested from different international areas, this setup proved that the system worked on the Internet.

9.3 Node Setup

Each node's software was installed and configured exactly the same. Each node contained an address look-up table to configure the hostname, Bluetooth Address and IP Number. This lookup table was stored as a configuration file on the node so that it could be easily and quickly modified and updated from anywhere on the network.

The Software for each network node consists of three main components: The Canary Background Server, The Connection Manager/Role Switch and the Web Interface. These three software packages worked together to initialise and imple-

ment the sensor network.

9.3.1 The Canary Background Server

Due to the closed communication standard of the Motornostix Canary, communication with the Canary is complex and troublesome. Several different factors increased the complexity of the interface.

An ATEN IC-485S RS232 - RS485 converter was used to connect the Canary to the serial port of the Axis board. Two settings on the converter allowed for flexibility, but finding the correct setting along with errors in the Linux code for driving the serial port caused many difficulties.

Communication with the Motornostix Canary was complicated by the limited documentation of the protocol. Although enough information was acquired for sending and receiving commands to and from the Canary, no documentation was acquired for receiving captured data from the canary sensors.

The background server was written in C. The basic Flow diagram for the program can be seen in figure 9.2.

The background server core's functionality lay in a select statement waiting on several files. The select statement must first have several filedescriptors initialised and then waits on those file descriptors. As soon as there is any activity on the file descriptors the select statement continues returning the file descriptor and inquires whether the data is trying to be read or trying to be written.

The select statement is run at a kernel level and is therefore extremely efficient and wastes no processing power. Because the Canary is not polled at this level, data traffic through the serial port remains very low.

The file pipe is created for inter-process communication or IPC. IPC allows one to send and receive commands from within one program to another program. This allows other programs to communicate with the Canary through the Background Driver. Other programs can therefore send data to the Canary without having to be concerned with packet structure and error correction. This is all maintained by the background server.

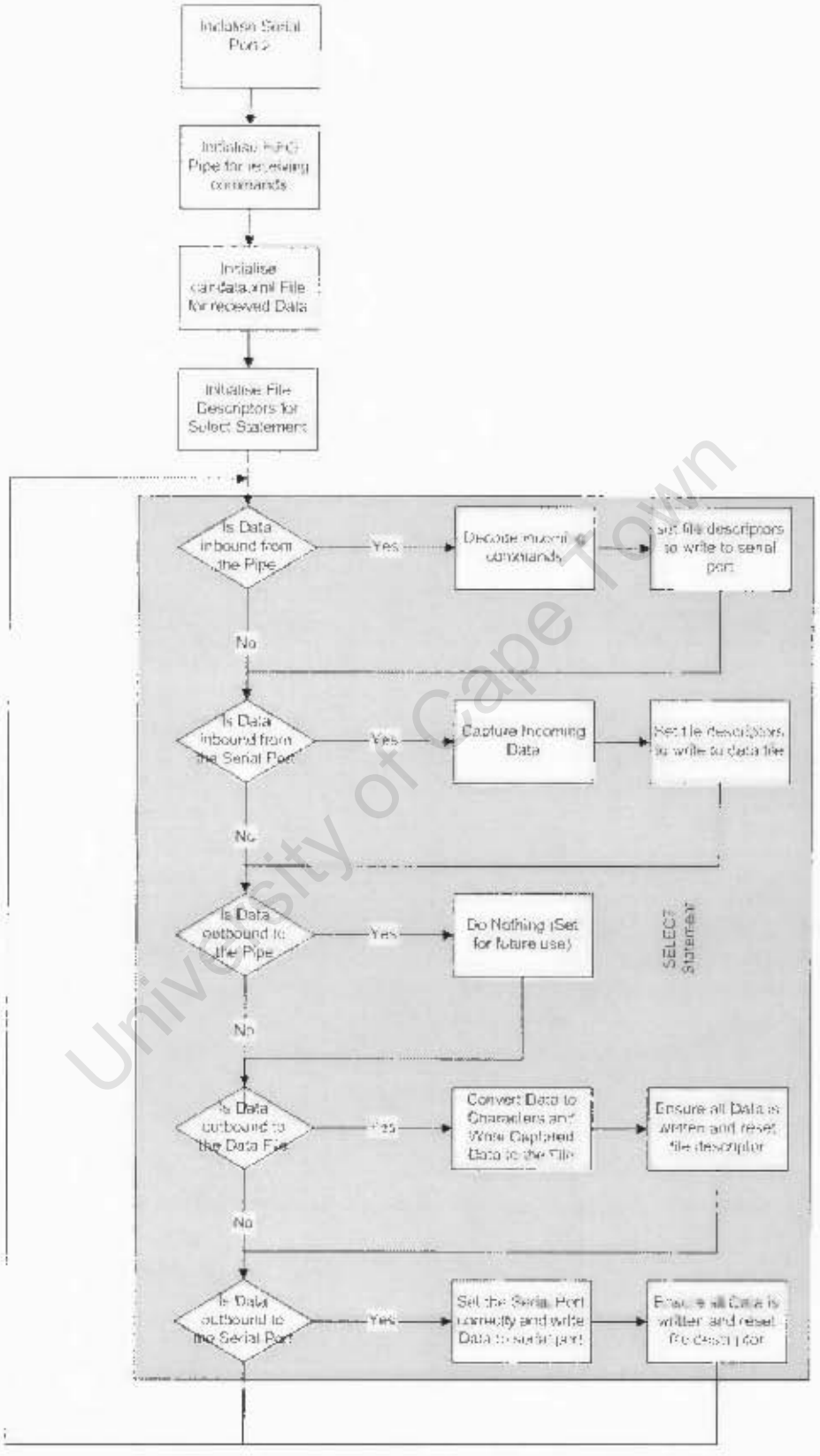

Figure 9.2: The Canary Background Server Program

9.3.2 The Connection Manager / Role Switch

The Connection Manager initialised the Nodes into the different roles depending on the hardware connected to the different Nodes. In this configuration, there were only two roles, namely Master and Slave. It then started the correct drivers and put the Canary into the different States.

The Connection Manager was written as a Linux Shell Script and the flow diagram can be seen in figure 9.3 with the code in Appendix A.2.1

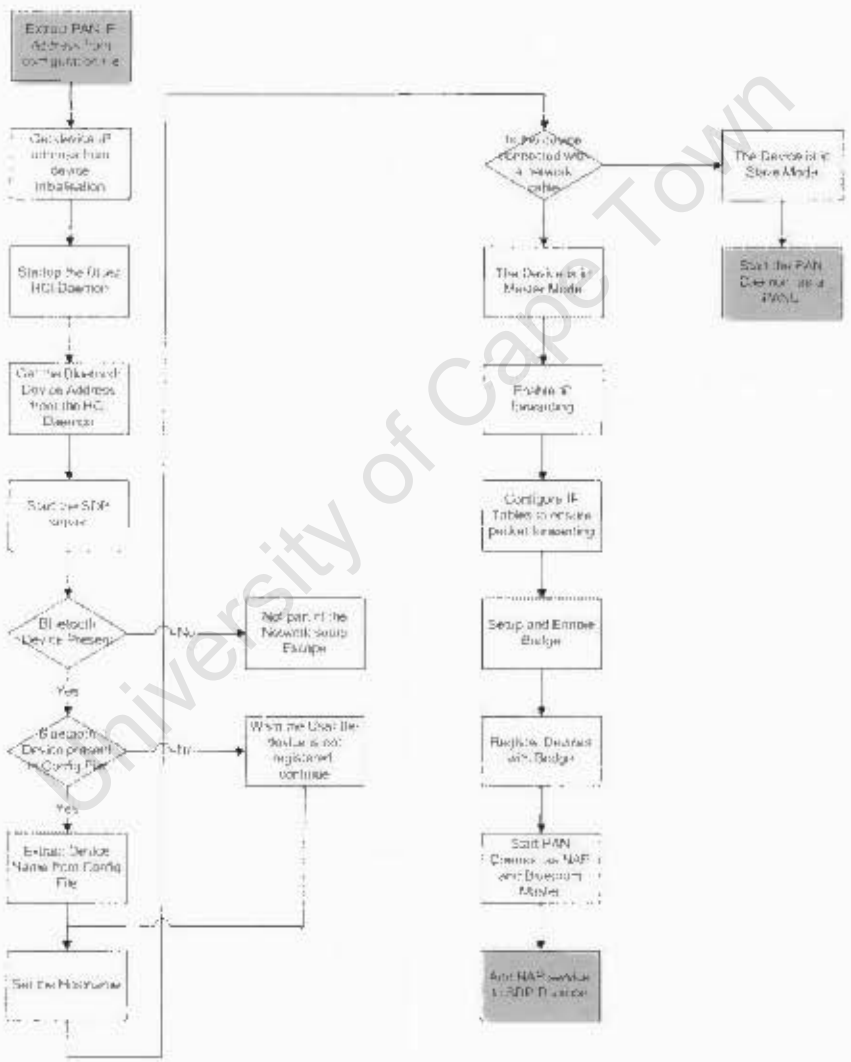

Figure 9.3: The Connection Manager / Role Switch

The connection manager starts the initial daemons and programs required for wireless communications. It then checks to see whether a Bluetooth device is

connected, and if so, set the hostname as configured to the device address. If no name exists for the device, the device can still participate in the wireless communications, but the user is notified that the device should be named. Only one device in the network can have no configured name.

Once the device name has been set, the connection manager checks to see whether the device has acquired an IP Number from a *Dynamic Host Configuration Protocol* (DHCP) request on the Ethernet port. The DHCP request was made during the initial startup of the device. If the device has an IP number, the device is switched into Master mode, whereas if it does not have an IP number, the node assumes that no ethernet connection is available and the device switches into Slave mode.

In Master Mode the device first enables IP forwarding. This allows IP packets to be transferred to different network adapters. It then sets up a bridge which allows for intelligent routing and then adds the adapters to the bridge. It then starts up the BlueZ PAN daemon with a NAP profile. The daemon is configured to listen for all devices to connect to it, but to ensure that this device controls the Bluetooth network and is set as Master of the piconet. The service is then registered with the SDP daemon which was started up at the beginning of the shell script.

When the device is initialised as a Slave, the PAN daemon is immediately started. The daemon is started with parameters to search throughout the network for a device running a NAP service. Once a NAP service is found a connection to the device is made.

Once two devices have created a Bluetooth connection, there is still some communication set-up to be done before the devices can send and receive IP Packets. For the devices to automate this procedure, help is acquired from the BlueZ Protocol stack which will automatically run a shell script in the BlueZ configuration directory when a new connection is made. The configuration script created for this system can be seen in Appendix A.2.2.

The script first ascertains whether the device is in Master or Slave mode. In Master mode, the new adapter is added to the bridge and the IP Number set to 0.0.0.0. In Slave mode the device acquires the IP number which it was assigned in the Configuration file and initialises itself with the IP. If there is no IP Number

assigned, the device will initialise itself with an IP assigned to all non-configured devices. It then sets the device to route all its packets through the Masters bridge at the location specified in the configuration file.

The devices can now communicate with the rest of the network. As the Master has been set into IP Forwarding mode, it can not be contacted on its IP Number.

9.3.3 The Web Interface

The Web Interface was created to view the data captured by the Canary and see which devices were connected to the Canary. To enable the device to display dynamic data, the Web Interface was written as a Common Gateway Interface (CGI) script. This meant that the file was very similar to a normal Linux shell script, except that it outputted information wrapped in HTML. This was implemented as a *here* file. The code for the Web Interface can be seen in Appendix A.4.1. The Web Interface also contained sub-scripts which actually retrieved information and sent data as required to the various device components.

The first component, called *cancheck*, simply polls the Canary while the user is viewing the page to see whether a Canary is connected to the development board.

All previous data is stored on the board under the previous data directory with a date stamp. Thus every time the board is restarted the backup script is run to save previous data. This previous data can be viewed with the two scripts named *showpreviousdata* and *previousdata*.

Chapter 10

Tests and Results

A self-configuring wireless sensor network was created using three Axis 82 Development boards. No external hardware was required for implementing the sensor network. The only external hardware used, was to view the information received from the Canaries.

The following chapter describes several tests and results obtained from the devices to describe the features and limitations of the devices. Tests described are broken up into different levels to illustrate the problem areas and functionality of Self-Configuring Wireless Sensor Networks. The tests are described starting from a single node and ending in tests performed on the full network.

10.1 Sensor Node Tests and Results

As only a single Canary was available for use in this work, It was important to ensure that multiple Canaries would be able to be used in the network. The approach used to test this was to connect and disconnect the Canary from multiple nodes in the network.

Canaries could successfully be connected and unconnected from the nodes via the Serial port. No problems in communication were found and the nodes were aware that the Canaries were connected or disconnected.

10.2 Network Tests and Results

The devices were set up in several different environments to test whether the devices could automatically detect each other and see the impacts of different environmental conditions. The devices were also restarted and stopped at different time-intervals to demonstrate real-world node failures to see if the network would be able to re-initialise itself.

Devices initialised as slaves could be stopped and started successfully, but unfortunately the node connected via RJ-45 would not successfully re-initialise itself to the fully functional previous state. If the nodes remained started and moved out of range with each other and were then brought back into range, the connection would successfully re-initialise.

Chapter 11

Conclusions

A Self-configuring Wireless Sensor network was decomposed into several different components and each component was investigated and analysed. From the analysis a self-configuring wireless sensor network was implemented.

The different components of the sensor network consist of a sensor, a node, a wireless link, node-intelligence for routing, and node-intelligence for self-configuration.

Several wireless technologies were investigated. The Bluetooth wireless technology was selected as the most practical to implement and use.

Different development platforms were investigated. The AXIS development board with Bluetooth was selected as the most affordable and easy to use development platform to develop the sensor network.

From the worldwide available Bluetooth Software Stacks, commercial and freely available stacks were investigated. The OpenBT software stack was first implemented and then changed over to the BlueZ software stack as it became necessary to implement newer and more advanced techniques during the course of this work.

A simple self-configuring wireless sensor network was created and implemented. The network was adaptive in terms of movement of the nodes, and addition of new slave nodes to the network. The network was self-configuring as it could decide what its role was in the network and have the ability to fulfill that role.

A self-configuring wireless sensor network requires some level of configuration for identifying individual nodes.

Chapter 12

Recommendations and Future Work

12.1 Further Investigations

As the need for Wireless Sensor Networks rapidly increases throughout the world, new technologies will arise that solve many of the current problems such as routing, sensor-intelligence and power consumption. Currently the technology that is leading the way in all of these fields is the Zigbee Alliance and IEEE 802.15.4. It is therefore of high importance to investigate the use of these technologies.

Later investigations should also be made into the IEEE 1451 Standards Working Group. This working group, which is especially aimed at smart sensors, has a sub-working group for intelligent wireless sensors called the IEEE 1451.5. Currently they are still in the process of creating a draft standard for wireless sensors, but it is expected to be complete soon.

12.2 Sensor Recommendations

Using the current system set-up, several further projects could be completed and carried out. Currently, the development boards are connected to the Canaries via the RS-232 port which in turn is connected to a RS232 to RS485 connector.

As the Canaries use RS485 as their communications protocol, and the development boards have an RS485 interface, connecting the Canary to the development board over this port would be a definite further step in development of this project. This would also immediately allow for multiple Canaries to be attached to the board, and thereby increasing the amount of sensor devices enormously.

As data from the Canaries sensors was never fully accessed, a project could be created on fully accessing the data, and graphically representing it on the Internet. This could be achieved by getting proper access to the development code of the Canaries as well as using the current network setup.

12.3 Wireless Communications Recommendations

As a wireless development platform, the Axis82 development board serves as an excellent starting point. Using the current setup an additional role could be added to the network. This "Relay" node could transmit packets between the Master Node and the Slave node, thereby increasing the distance and topology of the network system. This functionality, already inherent in newer technologies (such as Zigbee) would be interesting to investigate before the newer technologies are released.

Due to the abilities of the development platform used, other current wireless technologies apart from Bluetooth could also be implemented in a wireless sensor network. Of specific interest would be the 802.11b/g network standards. These standards, which have many commercial USB connectors available, could be implemented and tested and compared to the current network setup.

A multi-wireless standard node could also be created with this development board allowing for multiple network standards to connect to a single node. This could be a very interesting project, as a study could be done on both interference of frequencies of similar technologies as well as device interaction between devices of different wireless standards.

12.4 Interesting Projects

A few smaller interesting projects could be developed in terms of the importance of roles and role-based sensor networks [37] [32]. Similar nodes, such as in a Robot Soccer Team, could be developed to define a captain, defender and striker of the team. All players/nodes can take on any individual role, but much more benefit is gained by the team/network if individual roles are assigned from within the team/network.

A further extension of this project could be the implementation of a soccer pass. One node/player preparing the other node/player for data transmission/ball and sending the information, which is actually the ball across an unknown medium/-field.

As the Apple Computer Company has been one of the leaders of the IEEE 802.11 wireless networking standard, they have released several interesting open-source projects to help with wireless networking. One of these, called *Rendezvous* is a system for creating instant networking of devices and computers with dynamically discoverable services registered on the network. Rendezvous is based on IP networks.

Other interesting technologies include the IEEE 1451.5 *Draft Standard for a Smart Transducer Interface for Sensors and Actuators.* This working group which is part of the Wireless Sensor Working Group, appear to be close to having a new standard for Wireless Sensor Interfaces really soon.

Bibliography

[1] AXIS OpenBT Stack. [Online]. Available: http://developer.axis.com/software/bluetooth [Last accessed: 13 September, 2004].

[2] BlueDrekar has graduated! [Online]. Available: http://www.alphaworks.ibm.com/tech/bluedrekar [Last accessed: 13 September, 2004].

[3] CiteSeer - A distributed algorithm for minimum weight spanning trees. [Online]. Available: http://citeseer.ist.psu.edu/context/63827/0 [Last accessed: 13 September, 2004].

[4] DSSS and FHSS Spread Spectrum. [Online]. Available: http://www.arcelect.com/DSSS_FHSS-Spread_spectrum.htm [Last accessed: 31 August, 2004].

[5] ICASA Overview. [Online]. Available: http://www.icasa.org.za/default.aspx?page=1009 [Last accessed: 29 June, 2004].

[6] IEEE 802.11. [Online]. Available: http://en.wikipedia.org/wiki/IEEE_802.11 [Last accessed: 9 August, 2004].

[7] IEEE Wireless Zone - Overview. [Online]. Available: http://standards.ieee.org/wireless/overview.html [Last accessed: 14 August, 2004].

[8] Motornostix. [Online]. Available: http://www.motornostix.com [Last accessed: 13 September, 2004].

[9] South African Table of Frequency Allocations. [Online]. Available: http://www.icasa.org.za/Repository/resources/Whats%20New/South%20African%20Table%20of%20Allocations_pdf.pdf [Last accessed: 30 June, 2004].

[10] The Unofficial Bluetooth Weblog. [Online]. Available: http://bluetooth.weblogsinc.com [Last accessed: 19 August, 2004].

[11] uCLinux Embedded Linux / Microcontroller Project. [Online]. Available: http://www.uclinux.org/index.html [Last accessed: 13 September, 2004].

[12] Female Inventors: Hedy Lamarr. [Online]. Available: http://www.inventions.org/culture/female/lamarr.html [Last accessed: 25 May, 2004], 1999.

[13] Interference from Industrial, Scientific and Medical(ISM) Machines. [Online]. Available: http://www.ero.dk/documentation/docs/doc98/official/pdf/REP083.PDF [Last accessed: 29 June, 2004], June 2000.

[14] Bluetooth Network Encapsulation Protocol BNEP Specification. [Online]. Available: http://grouper.ieee.org/groups/802/15/Bluetooth/BNEP.pdf [Last accessed: 31 August, 2004], June 2001.

[15] ABCs of Spread Spectrum - A Technology Introduction and Tutorial. [Online]. Available: http://www.sss-mag.com/ss.html [Last accessed: 30 June, 2004], November 2003.

[16] Bluetooth Technology Overview. [Online]. Available: http://ncsp.forum.nokia.com/downloads/nokia /documents/Bluetooth_Technology_ Overview_v1_0.pdf [Last accessed: 1 July, 2004], April 2003.

[17] IEEE 802. [Online]. Available: http://en.wikipedia.org/wiki/IEEE_802 [Last accessed: 25 May, 2004], Dec 2003.

[18] Affix Bluetooth Protocol Stack For Linux. [Online]. Available: http://affix.sourceforge.net [Last accessed: 22 July, 2004], July 2004.

[19] Bluetooth Specification Documents. [Online]. Available: https://www.bluetooth.org/spec [Last accessed: 31 August, 2004], 2004.

[20] Direct-sequence Spread Spectrum. [Online]. Available: http://en.wikipedia.org/wiki/Direct-sequence_spread_spectrum [Last accessed: 31 August, 2004], July 2004.

[21] Ericsson technology licensing. [Online]. Available: http://www.ericsson.com/bluetooth/companyove/history-bl [Last accessed: 30 June, 2004], March 2004.

[22] IEEE 802.15 WPAN Task Group 1 (TG1). [Online]. Available: http://www.ieee802.org/15/pub/TG1.html [Last accessed: 25 May, 2004], Feb 2004.

[23] SourceForge.net. [Online]. Available: http://sourceforge.net/index.php [Last accessed: 13 September, 2004], 2004.

[24] Zigbee Alliance Frequently Asked Questions. [Online]. Available: http://www.zigbee.org/about/faqs/index.asp [Last accessed: 14 August, 2004], May 2004.

[25] Rick Alfvin. IEEE P802.15 Working Group for Wireless Personal Area Networks - Frequently Asked Questions. [Online]. Available: http://grouper.ieee.org/groups/802/15/pub/WPAN-FAQ.html [Last accessed: 14 August, 2004].

[26] B.N Clark, C.J. Colbourn and D.S. Johnson. *Unit Disk Graphs*, volume 86, pages 165–177. 1990.

[27] Brett Warneke. SMART DUST. [Online]. Available: http://www-bsac.eecs.berkeley.edu/archive/users/warneke-brett/SmartDust [Last accessed: 13 September, 2004], April 2004.

[28] C. L. Barret, S. J. Eidenbenz, L. Kroc, M. Marathe and J. P. Smith. Parametric Probabilistic Sensor Network Routing. In *Wireless Sensor Networks & Applications*, pages 122–131, PO Box 11405, September 2003. ACM.

[29] C.Hughes and T. Hughes. *Linux Rapid Application Development.* M & T Books, 2000.

[30] Dan Sonnerstam. Specification of the Bluetooth System. [Online]. Available: https://www.bluetooth.org/foundry/adopters/document/ Bluetooth_Core_Specification_v1.2 [Last accessed: 31 August, 2004], November 2003.

[31] José A. Gutiérrez, Edgar H. Callaway Jr. and Raymond L. Barret Jr. *Low-Rate Wireless Personal Area Networks.* IEEE Standards Information Network, 2003.

[32] K. Römer, C. Frank, P. J. Marrón and C. Becker. Generic Role Assignment for Wireless Sensor Networks. [Online]. Available: http://www.inf.ethz.ch/~chfrank/papers_eth/sigops.roleassignment.pdf [Last accessed: 13 September, 2004].

[33] Khaled M. Alzoubi, Peng-Jun Wan and Ophir Frieder. Message-Optimal Connected Dominating Sets in Mobile Ad Hoc Networks. In *MobiHoc*, pages 157–164, PO Box 11405 New York, NY 10286-1405, June 2002. Association for Computing Machinery.

[34] Kjell Jorgen Hole. Bluetooth Part 11: The Personal Area Networking Profile. [Online]. Available: http://www.kjhole.com/Standards/BT/BT-PDF/Bluetooth11.pdf [Last accessed: 31 August, 2004].

[35] K.N. King. *C Programming A Modern Approach.* W. W. Norton & Company Inc., 1996.

[36] Kris Pister. Smart Dust Autonomous sensing and communication in a cubic millimeter. [Online]. Available: http://robotics.eecs.berkeley.edu/ pister/SmartDust [Last accessed: 13 September, 2004].

[37] M. Kochhal, L. Schwiebert and S. Gupta. Role-based Hierarchical Self Organization for Wireless Ad hoc Sensor Networks. In *Wireless Sensor Networks & Applications*, pages 98–107, PO Box 11405, September 2003. ACM.

[38] Magnus Unemyr. A bluetooth protocol stack for embedded systems. [Online]. Available: http://www.iar.com/FilesPublic/ARTICLES/001183/EiN_ArticleEng.pdf [Last accessed: 31 August, 2004].

[39] Riku Mettala. Bluetooth protocol architecture. [Online]. Available: http://redwood.snu.ac.kr/nrl/Nrl/FILE/Bluetooth-wp-1C12000.pdf [Last accessed: 31 August, 2004], August 1999.

[40] M.Holtmann and A. Vedral. Bluetooth programming for Linux. [Online]. Available: http://www.holtmann.org/papers/bluetooth/wtc2003_slides.pdf [Last accessed: 22 July, 2004], 2003.

[41] Mikhail Galeev. Home networking with Zigbee. [Online]. Available: http://www.embedded.com/showArticle.jhtml?articleID=18902431 [Last accessed: 15 August, 2004], April 2004.

[42] Xiao Fan Wang and Guanrong Chen. Complex networks: Small-world, scale-free and beyond. *IEEE Circuits and Systems Magazine*, 3(1):6–20, 2003.

[43] Wayne W. Manges, Glenn O. Allgood and Stephen F. Smith. It's Time for Sensors to Go Wireless. *Sensors*, April 1999.

Appendix A

Program Listing

A.1 Canary Background Server

```
1  #include <sys/time.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <assert.h>
7  #include <stdio.h>
8  #include <asm/termios.h>
9
10
11
12 #define FALSE 0
13 #define TRUE 1
14 /*
15 This Program runs communication between the Canary and the SERVERFIFO File
16 This allows for communication to the canary via the SERVERFIFO file stored in
       the mnt directory
17 This files outputs all captured data into and XML stored file called candata.
       xml
18 */
19
20
21
22 /*********Configuration Settings for Canary Server*******/
23 /*****************************************************/
24
25 //The Serialport Settings where the Canary is connected to:
26 #define MODEM_DEVICE    "/dev/ttyS2"
27 #define BAUDRATE B115200
28
29 //The FIFO Named pipe name
30 #define FIFO_FILE       "/mnt/flash/SERVERFIFO" //The Pipe We will be using
       for IPC
31
32 //The Data File used to dump the canary data
33 #define DATA_FILE       "/mnt/flash/candata.xml"
34
35
36 #define WELCOMENOTE1    "Back Ground Canary Server Started- Built 22 June 04
       -"
37 #define WELCOMENOTE2    "Written by D de Jager - 2004"
38
```

```
39  //Debugging Info on or off?
40  int debug = FALSE;
41
42  /*****END*Configuration Settings for Canary Server*******/
43  /*********************************************************/
44
45  void dump_fds(char *name, fd_set *set, int max_fd)
46  {
47          int i;
48          if(!debug) return;
49
50          fprintf(stderr, "%s:",name);
51          for(i=0;i<max_fd;i++)
52          {
53                  if (FD_ISSET(i,set))
54                  {
55                          fprintf(stderr,"%d,", i);
56                  }
57          }
58          fprintf(stderr,"\n");
59  }
60
61  int main()
62  {
63          long BAUD;                      // derived baud rate from command
                line
64          long DATABITS;
65          long STOPBITS;
66          long PARITYON;
67          long PARITY;
68
69          //File Descriptor Values
70          int serial;
71          int Pipe;
72          int OutputData;
73
74
75          //Useul Variables
76          int j;  //Counter for While loop in Pipe to ensure reads are correct
77          int rc; //rc is used for number of reads/writes returned
78          int hex;        //used to convert decimal to chars
79
80          //File Descriptor Sets to Use
81          fd_set readfds;
82          fd_set writefds;
83          fd_set exceptfds;
84          struct timeval tv;
85          int max_fd;
86
87          /*inbound and outbound keep track of whether
88            we have a character already read which needs
89            to be sent in that direction the _char and _data
90            variables are the data buffers*/
91
92          int outbound;
93          char outbound_char;
94          int inbound;
95          int inbound_data[1024];
96
97          //pipe_outbound keeps track of whether we have data bound for Canary
                from the Pipe
98          //while readbuf stores the message from the Pipe
99          int pipe_outbound;
100         char readbuf[80];
101
102         //Setup Serial port information
103         struct termios oldtio, newtio;  //Old and new port settings for
                serial port
104
105         /*Test command to send the Canary for communications retrieval*/
106         char buf[14];   //Canary Command Buffer
107
108         //General Variables
```

```
109         int serial_read_count;  //Counter to say how many bytes read from
                Serial Port
110         int modem_bits; //Used to set the RTS Signals for the 232-485
                Convertor
111
112
113         //After Declaring Variables Set their values
114         BAUD = BAUDRATE; //BAUDRATE declared in Config section
115         DATABITS = CS8; //CS8 declared in termios.h
116         STOPBITS = 0;
117         PARITYON = 0;
118         PARITY = 0;
119
120
121         //Default Canary Command to recieve a response from the canary
122         buf[0] = 0x08;
123         buf[1] = 0x01;
124         buf[2] = 0x00;
125         buf[3] = 0x00;
126         buf[4] = 0x01;
127         buf[5] = 0x02;
128         buf[6] = 0x00;
129         buf[7] = 0x00;
130         buf[8] = 0x00;
131         buf[9] = 0x02;
132         buf[10] = 0x38;
133         buf[11] = 0xbe;
134         buf[12] = 0xa0;
135         buf[13] = 0xff;
136         buf[14] = '\0';
137
138
139
140
141         /****************Open Controlling Devices****************/
142         /********************************************************/
143
144
145         //Open Serial Port on MODEM_DEVICE and set settings reffered to in '
                man termios'
146         serial = open(MODEM_DEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
147         assert(serial>=0);
148         tcgetattr(serial,&oldtio); // save current port settings
149         newtio.c_cflag = BAUD | CRTSCTS | DATABITS | STOPBITS | PARITYON |
                PARITY | CLOCAL | CREAD;
150         newtio.c_iflag = IGNPAR;
151         newtio.c_oflag = 0;
152         newtio.c_lflag = 0;        //ICANON;
153         newtio.c_cc[VMIN]=1;
154         newtio.c_cc[VTIME]=0;
155         tcflush(serial, TCIFLUSH);
156         tcsetattr(serial,TCSANOW,&newtio);
157
158
159         //Open Pipe for IPC Get the File Descriptor for the select call
160         mkfifo(FIFO_FILE,0666);
161         Pipe = open(FIFO_FILE,O_RDONLY |O_NONBLOCK| O_SYNC);
162         assert(Pipe>=0);
163         fcntl(Pipe,F_SETFL, O_RDONLY | O_SYNC);
164
165
166         //Open OutputData File to Write Data from the Canary to
167         OutputData = open(DATA_FILE, O_CREAT|O_WRONLY |O_APPEND, 0666);
168         assert(OutputData>=0);
169
170
171         /***************END Open Controlling Devices*************/
172         /********************************************************/
173         //Display Welcome Note
174         fprintf(stderr, WELCOMENOTE1);
175         fprintf(stderr, WELCOMENOTE2);
176
177
```

```
178         if(debug)    //Show Debug File descriptors
179         {
180                 fprintf(stderr, "serial=%d\n", serial);
181                 fprintf(stderr, "Pipe=%d\n", Pipe);
182         }
183
184         //Initialising Variables
185         serial_read_count=0;
186         outbound = inbound =0;
187         pipe_outbound =0;
188
189         //Start main while Loop
190         while(1)
191         {
192
193                 //writefds Checking the Write File Descriptor for anything
                        ready to write
194                 FD_ZERO(&writefds);
195                 if (inbound) FD_SET(OutputData, &writefds); //if data inbound
                        from serial get ready to write to file
196                 if (outbound) FD_SET(serial, &writefds); //if data outbound
                        to the serial write to serial
197                 if (pipe_outbound)              //if data outbound to Pipe
                        set it to go to the Pipe
198                 {
199                         if (debug) fprintf(stderr,"\nReseting pipe_outbound
                                for next pipe read\n");
200                         close(Pipe);
201                         Pipe = open(FIFO_FILE,O_RDONLY |O_NONBLOCK| O_SYNC);
202                         assert(Pipe>=0);
203                         fcntl(Pipe,F_SETFL, O_RDONLY | O_SYNC);
204                         pipe_outbound =0;
205                 }
206
207                 //readfds Checking the Read File Descriptor for anything
                        ready to be Read
208                 FD_ZERO(&readfds);
209                 if(!outbound) FD_SET(Pipe, &readfds); /*If we have no data in
                        outbound get ready to read from Pipe*/
210                 if(!inbound) FD_SET(serial, &readfds); //If we have no Data
                        inbound from Serial carryon waiting
211                 //if(!pipe_outbound) FD_SET(Pipe, &readfds);
212
213                 //Check to see what is the Maximum file descriptor for the
                        Select Statement and add one
214                 max_fd =0;
215                 if(serial > max_fd) max_fd = serial;
216                 if(Pipe > max_fd) max_fd = Pipe;
217                 if(OutputData > max_fd) max_fd = OutputData;
218                 max_fd++;
219
220
221                 if(debug) fprintf(stderr, "max_fd=%d\n", max_fd);
222                 tv.tv_sec = 10; //Resets the Select Call every 10 seconds
223                 tv.tv_usec = 0;
224
225                 //Dump file descriptors with appropriate values - Only showed
                        if
226                 //Debug is set to 1
227                 dump_fds("read in",&readfds, max_fd);
228                 dump_fds("write in", &writefds, max_fd);
229
230                 //Wait on this
231                 rc = select(max_fd, &readfds, &writefds, NULL, &tv);
232
233                 dump_fds("read out", &readfds, max_fd);
234                 dump_fds("write out", &writefds, max_fd);
235
236                 //Different Sets of Information sending
237
238
239                 //We can read Data from the Pipe
240                 if(FD_ISSET(Pipe, &readfds))
```

```
241                     {
242                             if (debug) fprintf(stderr, "\nReading Outbound Pipe\n
                                    ");
243                             rc = read(Pipe,&readbuf,80);
244                             read(Pipe, NULL,1); //Just added to Close PipeReads
                                    !!!!
245                             readbuf[rc] = '\0';
246                             pipe_outbound=1;
247
248                             //If Incoming PIPE command is "SendPacket" Write to
                                    serial port
249                             if(strcmp(readbuf, "SendPacket") == 0) outbound =1;
250                             if(strcmp(readbuf, "Quit") == 0) break;
251
252                     }
253
254
255                     //We can read Data from the Serial Port
256                     if(FD_ISSET(serial, &readfds))
257                     {
258                             if (debug) fprintf(stderr, "\nreading inbound\n");
259                             while(rc>0)
260                             {
261                                     rc = read(serial, &inbound_data[
                                            serial_read_count],1);
262                                     serial_read_count++;
263                             }
264                             if (serial_read_count>0) inbound=1;
265                     }
266
267
268                     //We can write Data to the Pipe
269                     if(FD_ISSET(Pipe, &writefds))
270                     {
271                         if(debug) fprintf(stderr, "\nwriting inbound to Pipe\n");
272
273                     }
274
275
276                     //We can write Data to the File
277                     //This just basically writes all the data recieved into
                            OutputData File
278                     //Declared in the Config file as DATA_FILE
279                     if(FD_ISSET(OutputData, &writefds))
280                     {
281                         for(j=0;j<serial_read_count;j++)
282                         {
283                             hex = inbound_data[j] >>4;
284
285                             if (hex> 9)
286                                 hex +=87;
287                             else
288                                 hex+=48;
289
290                             write(OutputData, &hex,1);
291                             hex = inbound_data[j] & 0xF;
292
293                             if (hex> 9)
294                                 hex +=87;
295                             else
296                                 hex+=48;
297
298                             write(OutputData, &hex,1);
299                         }
300                         hex = '\n';
301                         write(OutputData, &hex,1);
302
303                         //Make sure there is no-more DATA in
304                         if (rc >0) inbound=0;
305                         //Re-Initialise Serial_read_count for Serial Input
306                         serial_read_count =0;
307                     }
308
```

```
309
310                     //We can write Data to the Serial Port
311                     if(FD_ISSET(serial, &writefds))
312                     {
313                             if(debug) fprintf(stderr, "\nwriting outbound\n");
314                             //Set the RTS Line correctly to ensure Transmission
315
316
317                             //Set RTS High
318                             ioctl(serial, TIOCMGET, &modem_bits);
319                             modem_bits |= TIOCM_RTS;
320                             ioctl(serial, TIOCMSET, &modem_bits);
321
322                             //Get the Data from the Port
323                             rc = write(serial,buf,14);
324
325                             //Set RTS Low
326                             tcsetattr(serial, TCSADRAIN, &newtio);
327                             modem_bits &= ~TIOCM_RTS;
328                             ioctl(serial, TIOCMSET, &modem_bits);
329                             if(rc>0) outbound=0;
330                     }
331
332
333
334             }       //End main while loop
335
336             fprintf(stderr,"\nThank you for running Canary Server\n");
337
338
339             tcsetattr(serial,TCSANOW,&oldtio);
340             //Close the File Descriptors
341
342             close(serial);
343             close(Pipe);
344             close(OutputData);
345
346
347     }       //End main() Function
```

A.1.1 Pipe communication Tool

```
1   /*
        ************************************************************************

2    Excerpt from "Linux Programmer's Guide - Chapter 6"
3    (C)opyright 1994-1995, Scott Burkett
4    ************************************************************************

5    MODULE: fifoclient.c
6    ************************************************************************
         */
7
8   #include <stdio.h>
9   #include <stdlib.h>
10
11  #define FIFO_FILE       "/mnt/flash/SERVERFIFO"
12
13  int main(int argc, char *argv[])
14  {
15          FILE *fp;
16
17          if ( argc != 2 ) {
18                  printf("USAGE: cchat [string]\n");
19                  printf("       To send a Packet to Canary: cchat SendPacket\n
                        ");
20                  exit(1);
21          }
```

```
22
23          if((fp = fopen(FIFO_FILE, "w")) == NULL) {
24                  perror("fopen");
25                  exit(1);
26          }
27
28          fputs(argv[1], fp);
29
30          fclose(fp);
31          return(0);
32  }
```

A.2 Connection Manager / Role Switch

A.2.1 Node Initialisation

```
1   #!/bin/sh
2
3
4   PANIP="$(cat /etc/canServ.conf | grep PANIP | cut -d= -f2)"
5   IPADDRESS="$(cat /var/log/messages |grep eth0 |grep IP|cut -d'"' -f2)"
6   CONFIGFILE=$(cat /etc/canServ.conf | grep BTDEVICE | cut -d= -f2)
7
8
9   #Start the same services for both Masters and Slaves
10  hcid
11  sleep 1
12  #Get the BD Address now that HCI is up
13  BDADDRESS="$(hciconfig |grep 'BD Address' | cut -d ' ' -f3 )"
14  sdpd
15
16  #Check to See which device is up and running
17  #BDADDRESS="TEST TO SEE IF IT WORKS"
18  #BDADDRESS="00:E0:98:9F:AD:3B"
19
20  if [ -z "$BDADDRESS" ] ;
21      then
22          echo "No Bluetooth Device Present"
23          echo "Not enrolling onto Bluetooth Network"
24          exit 0
25  else
26      echo "Device:$BDADDRESS - Found"
27      echo "Searching Config for device $NAME"
28      DEVICEENTRY="$(cat /etc/canServ.conf | grep BTDEVICE | grep "$BDADDRESS")
            "
29      if [ -z "$DEVICEENTRY" ] ;
30      then
31          echo "Device Not found!"
32          echo "Setting Device to Wildcard entry"
33          UNKNOWNDEVICE="$(cat /etc/canServ.conf | grep "*")"
34          NAME=$(echo $UNKNOWNDEVICE | cut -d= -f3)
35          echo "Unknown Device $BDADDRESS set with name $NAME"
36      else
37          NAME=$(echo $DEVICEENTRY | cut -d= -f3)
38          echo "Device $BDADDRESS found with name $NAME"
39      fi
40
41
42
43  fi
44
45
46  #Set the HOSTNAME as defined in previous section
47  hostname $NAME
```

```
48  echo "HOSTNAME=\"$NAME\" " > /etc/conf.d/hostname
49
50
51  #Check if there was an IP Address Found from the DHCP Server
52  #This will set whether the device is the Master or a Slave
53
54  if [ -z "$IPADDRESS"]; then
55      echo "No IP Address Found - Setting Role as PAN User"
56      echo "Starting up PAN Driver as PANU"
57      pand --role PANU --search --service NAP --nodetach --persist
58
59  else
60      echo "IP Address - $IPADDRESS Found - Setting Role as PAN Master"
61      echo "Enabling IP Forwading"
62      echo "1" > /proc/sys/net/ipv4/ip_forward
63      echo "Enabling Rogue Packet Filtering on all interfaces"
64      for f in /proc/sys/net/ipv4/conf/*/rp_filter;do
65      echo 1 >> $f
66      echo $f
67      done
68
69      iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
70      iptables -A FORWARD -i pan0 -j ACCEPT
71      iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
72
73
74      #Set up the Bridge and show it out
75      echo "Setting up Canary Bridge"
76      echo "Creating Bridge pan0"
77      brctl addbr pan0
78      sleep 1
79      echo "Configuring Bridge to IP Address $PANIP"
80      ifconfig pan0 $PANIP
81      sleep 1
82      #echo "Turning off listening and learning on Canary Bridge"
83      #brctl setfd pan0 0
84      #sleep 1
85      #echo "Turning Minimum Spanning Tree Off on Bridge"
86      #brctl stp pan0 off
87      #sleep 1
88      echo "Adding Interface eth0 to Canary Bridge"
89      brctl addif pan0 eth0
90      sleep 1
91      echo "Starting up PAN Driver as NAP"
92      pand --listen --role NAP --master
93      sleep 1
94      echo "Adding NAP service to SDP"
95      sdptool add NAP
96      sleep 1
97      brctl show
98      brctl showmacs pan0
99      #Once the bridge has made a connection the script /etc/bluetooth/pan/dev-
            up is
100     #run and the device is brought up
101 fi
```


A.2.2 dev-up: Device Initialisation

```
1  #!/bin/sh
2
3  #Complete the Connection
4  PANIP=$(cat /etc/canServ.conf | grep PANIP | cut -d= -f2)
5  IPADDRESS="$(brctl show |grep pan0)"
6  LOGFILE=/var/log/messages
7  BDADDRESS="$(hciconfig |grep 'BD Address' | cut -d ' ' -f3 )"
8
9
10
```

```
11 if [ -n "$IPADDRESS" ] ;
12     then
13         echo "Role Set as Master - dev-up Script connected device $1 on $2"
               1>>$LOGFILE
14         brctl addif pan0 $1
15         sleep 1
16         ifconfig $1 0.0.0.0
17
18 else
19         echo "Role Set as Slave - dev-up Script connected device $1 on $2"
               1>>$LOGFILE
20         DEVICEENTRY="$(cat /etc/canServ.conf | grep BTDEVICE | grep "
               $BDADDRESS")"
21         if [ -z "$DEVICEENTRY" ] ;
22             then
23                 UNKNOWNDEVICE="$(cat /etc/canServ.conf | grep "*")"
24                 IP=$(echo $UNKNOWNDEVICE | cut -d= -f4 | cut -d '"' -f2)
25                 echo "IP Address $IP assigned to UNKNOWN $1 Device" 1>>
                       $LOGFILE
26                 ifconfig $1 $IP
27
28
29         else
30                 IP=$(echo $DEVICEENTRY | cut -d= -f4 | cut -d '"' -f2)
31                 echo "IP Address $IP assigned to $1 " 1>>$LOGFILE
32                 ifconfig $1 $IP
33         fi
34         sleep 1
35         route add 0.0.0.0 gw $PANIP
36 fi
```

A.3 canServ.conf: Device Configuration

```
1  #Configuration File for Canaries,
2  #Add an entry for each new Bluetooth
3  #Device
4
5  PANIP=192.168.0.10
6
7  BTDEVICE="00:A0:96:20:42:A0"="Blue"="192.168.0.21"
8  BTDEVICE="00:E0:98:85:8E:17"="Aqua"="192.168.0.22"
9  BTDEVICE="00:E0:98:9F:AD:3B"="Marine"="192.168.0.23"
10 BTDEVICE="*"="Billy"="192.168.99"
```

A.4 Web Interface

A.4.1 Web CGI Scripts

Main Web *here* file

```
1  #!/bin/sh
2
3
4  # Canary Configuration Page
5  ##################CONSTANTS
6  HOSTNAME=$(hostname)
```

```
title="Welcome to Canary Server $HOSTNAME"

#IPADDRESS="$(cat /var/log/messages |grep eth0 |grep IP |cut -d'"' -f2)"
#BLUETOOTHADDRESS=$(hciconfig | grep 'BD Address' | cut -d ' ' -f3)
PANCONNECTIONS="$(pand --show)"
CANCHECK="$(cancheck)"
CANDATA="$(cat /mnt/flash/candata.xml)"
PREDATALIST="$(previousdata)"

################## Web Page
cat <<- _EOF_
    <HTML>
    <HEAD>
    <META HTTP-EQUIV="Refresh" CONTENT="5; URL=/admin-bin/config.cgi">
        <TITLE>
        $title
        </TITLE>
    </HEAD>

    <BODY>
    <H1>$title</H1>
        <TABLE>
                <TR>
                        <TH>Current Date</TH>
                        <TD>$(date +"%x %r %Z")</TD>
                </TR>
<!--
                <TR>
                        <TH>IP Address</TH>
                        <TD>$IPADDRESS</TD>
                </TR>
                <TR>
                        <TH>Bluetooth Address</TH>
                        <TD> $BLUETOOTHADDRESS </TD>
                </TR>
-->
                <TR>
                        <TH>Personal Area Connections</TH>
                        <TD>$PANCONNECTIONS</TD>
                </TR>
                <TR>
                        <TH>Canary Connected</TH>
                        <TD>$CANCHECK</TD>
                </TR>
                <TR>
                        <TH>Canary Data</TH>
                        <TD>$CANDATA</TD>
                </TR>
                <TR>
                        <TH>Previous Canary Data</TH>
                        <TD>$PREDATALIST</TD>
                </TR>

        </TABLE>
    </BODY>
    </HTML>
_EOF_
```

cancheck.sh

```
#!/bin/sh


#This program polls the Canary and checks for any output
#If output is detected, reply "Canary Present" otherwise reply "No Canary
    Detected"

BEFORE=0
```

```
AFTER=0

TEST1=$(cat /mnt/flash/candata.xml)

for i in $TEST1

    do
        BEFORE=$(($BEFORE + 1))
done



#echo "$BEFORE lines before Canary Test"
cchat.cgi SendPacket
sleep 1
TEST2=$(cat /mnt/flash/candata.xml)

for i in $TEST2

    do
        AFTER=$(($AFTER + 1))
done

#echo "$AFTER lines after Canary Test"

if $(test $AFTER -gt $BEFORE); then
    echo "Canary Present"
else
    echo "No Canary Detected"
fi
```

A.4.2 Previous Data Scripts

previousdata.sh

```
#!/bin/sh


#This allows one to view a list of the previous Canary Data that has been
    recieved

DATA=$(ls /mnt/flash/candata |uniq |cut -d' ' -f1)

for i in $DATA
do echo "<A HREF=\"/admin-bin/show_predata.cgi?/mnt/flash/candata/$i/candata.
    xml\">$i</A> "
done
```

show_predata.sh

```
#!/bin/sh
#This allows one to view the previous Data

FILE="$(cat $1)"

cat <<- __EOFD__


<HTML>
    <HEAD>
        <TITLE>FILE LISTING OF $1 </TITLE>
```

```
12      </HEAD>
13
14
15      <BODY>
16          <H1>File listing of $1</H1>
17          <BR>
18          $FILE
19      </BODY>
20
21  </HTML>
22
23  __EOFD__
```

backupdata.sh

```
1   #!/bin/sh
2
3
4   #This program saves the data into a backup directory and clears candata.xml
5
6   DATE=$(date +%F)
7   chmod 666 /mnt/flash/candata.xml
8   mkdir /mnt/flash/candata
9   mkdir /mnt/flash/candata/$DATE
10  cp /mnt/flash/candata.xml /mnt/flash/candata/$DATE
11  rm /mnt/flash/candata.xml
12  "" > /mnt/flash/candata.xml
```

Appendix B

Canary Communication Protocol Code

B.1 General Commands

B.1.1 mtnx.h

```
#define DATALENGTH 1023

struct DATA_PACKET {
int Source_Device_Address;
int Destination_Device_Address;
int Protocol_Identifier;
int Command;
char DATA[DATALENGTH+1];
int INTDATA[DATALENGTH];
int data_length;
int Packet_Number;
int CRC_CHECK;
int ETx;
};



void Display_Packet(struct DATA_PACKET Data_Packet_to_display); //Displays
    the Data Packet in Human Readable Form

struct DATA_PACKET build_packet(int Source_Device_Address,
    int Destination_Device_Address, int Protocol_Identifier, int Command,
    int data, int Packet_Number, int CRC_CHECK, int ETx);       //Constructs
        a Data Packet

int Length(struct DATA_PACKET Data_Packet_to_measure);  //Length of Whole
    Data Packet excluding ETx characters
```

B.1.2 mtnx.c

```
1  /*New Program to take data only when there is data on the line*/
2
3  #include <stdlib.h>  /* Dirk added Standard Library Header */
4  #include <stdio.h>   /* Standard input/output definitions */
5  #include <string.h>  /* String function definitions */
6  //#include <unistd.h>  /* UNIX standard function definitions */
7  #include <fcntl.h>   /* File control definitions */
8  #include <errno.h>   /* Error number definitions */
9  //#include <termios.h> /* POSIX terminal control definitions */
10
11 #include <asm/unistd.h>  /* UNIX standard function definitions */
12 #include <asm/termios.h> /* POSIX terminal control definitions */
13
14 #include <sys/types.h>
15 #include <sys/stat.h>
16
17 #include "mtnx.h" /*Our Header File with Packet Structures*/
18 #include "mtnx_recv.h"
19
20 #define _POSIX_SOURCE 1 /* POSIX compliant source */
21 #define FALSE 0
22 #define TRUE 1
23
24 #define CRCPOLY 0x9005
25
26
27
28
29
30
31 /***********END Function Definitions***************/
32
33
34 //Read data stream until a key is pressed and then send packet
35 //Continue reading data stream
36
37
38 int Length(struct DATA_PACKET Data_Packet_to_measure)
39 {
40     if (Data_Packet_to_measure.CRC_CHECK == -1)
41         {
42             return strlen(Data_Packet_to_measure.DATA)+16;
43         }
44     else
45         {
46             return strlen(Data_Packet_to_measure.DATA)+20;
47         }
48
49
50 }
51
52 void Display_Packet(struct DATA_PACKET Data_Packet_to_display)
53 {
54     int i=0;
55     printf("\r\n*****************Human Readable Packet Information
           *****************");
56     printf("\r\nSource Device Address:%4.4x\r\n", Data_Packet_to_display.
           Source_Device_Address);
57     printf("\tSource Device Type:%2.2x\r\n",  Data_Packet_to_display.
           Source_Device_Address >>10);
58     printf("\tSource Device Identifier:%3.3x\r\n",  Data_Packet_to_display.
           Source_Device_Address & 1023);
59
60     printf("Destination Device Address:%4.4x\r\n", Data_Packet_to_display.
           Destination_Device_Address);
61     printf("\tDestination Device Type:%2.2x\r\n",  Data_Packet_to_display.
           Destination_Device_Address >>10);
62     printf("\tDestination Device Identifier:%3.3x\r\n",
           Data_Packet_to_display.Destination_Device_Address & 1023);
63
```

```
64      printf("Protocol Identifier:%2.2x\r\n", Data_Packet_to_display.
            Protocol_Identifier);
65      printf("Command Code:%2.2x\r\n", Data_Packet_to_display.Command);
66      printf("Data to be Transmitted:");
67      for (i=0;i<Data_Packet_to_display.data_length;i++)
68      {
69          printf("%2.2x",Data_Packet_to_display.INTDATA[i]);
70      }
71      printf("\r\nPacket Number:%4.4x\r\n", Data_Packet_to_display.
            Packet_Number);
72      printf("CRC Check Code:%4.4x\r\n", Data_Packet_to_display.CRC_CHECK);
73      printf("End-of-Transmission Characters:%4.4x\r\n", Data_Packet_to_display
            .ETx);
74      printf("*************END Human Readable Packet Information
            ****************\r\n\n");
75  }
76
77  struct DATA_PACKET build_packet(int Source_Device_Address,
78      int Destination_Device_Address, int Protocol_Identifier, int Command,
79      int data, int Packet_Number, int CRC_CHECK, int ETx)
80  {
81      struct DATA_PACKET d;
82      d.Source_Device_Address = Source_Device_Address;
83      d.Destination_Device_Address = Destination_Device_Address;
84      d.Protocol_Identifier = Protocol_Identifier;
85      d.Command = Command;
86      d.INTDATA[0] = data;
87      d.INTDATA[1] = 0;
88      d.Packet_Number = Packet_Number;
89      d.CRC_CHECK = CRC_CHECK;
90      d.ETx = ETx;
91      d.data_length = 2;
92      return d;
93
94  }
```

B.2 Receive Commands

B.2.1 mtnx_recv.h

```
1
2   /***************Function Definitions***************/
3
4   struct DATA_PACKET Get_New_Packet(int fd);
5
6   int *Remove_ETx_Sequencing(int *incoming_data);
7
8   int Check_Incoming_CRC(int *incoming_data, int length);
9
10  int Incoming_Length(int *incoming_data);
11  /************END Function Definitions***************/
```

B.2.2 mtnx_recv.c

```
1   /*New Program to take data only when there is data on the line*/
2
3   #include <stdlib.h>   /* Dirk added Standard Library Header */
4   #include <stdio.h>    /* Standard input/output definitions */
5   #include <string.h>   /* String function definitions */
```

```
6  //#include <unistd.h>  /* UNIX standard function definitions */
7  #include <fcntl.h>    /* File control definitions */
8  #include <errno.h>    /* Error number definitions */
9  //#include <termios.h> /* POSIX terminal control definitions */
10
11 #include <asm/unistd.h>  /* UNIX standard function definitions */
12 #include <asm/termios.h> /* POSIX terminal control definitions */
13
14 #include <sys/types.h>
15 #include <sys/stat.h>
16
17 #include "mtnx.h" /*Our Header File with Packet Structures*/
18 #include "mtnx_recv.h"
19
20 #define _POSIX_SOURCE 1 /* POSIX compliant source */
21 #define FALSE 0
22 #define TRUE 1
23
24 #define CRCPOLY 0x9005
25
26
27 struct DATA_PACKET Get_New_Packet(int fd)
28 {
29     struct DATA_PACKET tempPacket;
30     int *tempints = malloc(1024);
31     char buf[1024];                          //buffer for where data is put
32     int dirty, nbytes;
33     int length_of_packet;
34     int total_ints;
35     int i,j;
36     int *begin_ints;
37     int H_Value;
38     int res=0;
39     unsigned char In1;
40
41     begin_ints = tempints;
42     dirty=0;
43
44     printf("Getting Data!");
45     while(dirty !=10)
46     {
47
48         res = read(fd,buf,1024);
49         printf("\n\r***%d Bytes Read: ",res);
50
51         if (res>0)
52         {
53             for (i=0; i<res; i++)  //for all chars in string
54             {
55                 In1 = buf[i];
56                 *tempints = In1;
57                 tempints++;
58             }  //end of for all chars in string
59         }  //end if res?
60
61         tempints--;
62         if (*tempints == 0xff)
63         {
64             tempints--;
65             if(*tempints == 0xa0)
66             {
67                 printf("Correct Packet Recieved\r\n");
68                 dirty=10;
69             }
70             tempints++;
71         }
72
73
74         else
75         {
76             printf("Packet not Completed\r\n");
77             dirty++;
78         }
```

```
79          tempints++;
80
81
82          //printf("\r\n%d Dirty Bits\r",dirty);
83          /*
84          printf("%x ", *tempints);
85          if(*tempints == 0xff)
86          {
87               tempints--;
88               if(*tempints == 0xa0)
89               {
90                    printf("Correct Packet Recieved\r\n");
91               }
92               else
93               {
94                    printf("Packet not Completed\r\n");
95
96               }
97               tempints++;
98
99          }
100         else
101         {
102              printf("Packet not Completed\r\n");
103         }
104         */
105
106     }
107
108     total_ints = tempints - begin_ints;
109     tempints = begin_ints;
110
111     for(i=0;i<total_ints;i++)
112     {
113         printf("%2.2x ",tempints[i]);
114     }
115     printf("\r\n");
116
117
118
119     tempints = Remove_ETx_Sequencing(tempints);
120
121     //Calculate how long the packet is in bytes
122     length_of_packet = Incoming_Length(tempints);
123     //length_of_packet = total_ints;
124     //printf("The Packet is %d Bytes long",length_of_packet);
125
126     //Check the CRC
127     Check_Incoming_CRC(tempints, (length_of_packet-2));
128
129           //length_of_packet +=2;
130           //If the length of the Packet has enough bytes
131     if (length_of_packet > 9)
132         {
133
134              //Start with Source ID
135              tempPacket.Source_Device_Address = (tempints[0] << 8) + (tempints
                     [1]);
136
137              //Continue with Destination Device ID
138              tempPacket.Destination_Device_Address = (tempints[2] << 8) + (
                     tempints[3]);
139
140              //Protocol Identifier
141              tempPacket.Protocol_Identifier = (tempints[4]);
142
143              //Command
144              tempPacket.Command = (tempints[5]);
145
146
147              //Integer Data
148              for(i=0;i< length_of_packet-6;i++)
149              {
```

```
150                 tempPacket.INTDATA[i] = tempints[i+6];
151             }
152
153             //Integer Data Length
154             tempPacket.data_length = total_ints-2-6-4;
155             //printf("Total Ints: %d\n", total_ints);
156
157             //The Packet Number
158             tempPacket.Packet_Number = ((tempints[length_of_packet-6] << 8) +
                    (tempints[length_of_packet-5]));
159             //printf("The Packet Number is:%4.4x\n", tempPacket.Packet_Number
                    );
160
161             tempPacket.CRC_CHECK = ((tempints[length_of_packet-4] << 8) + (
                    tempints[length_of_packet-3]));
162             tempPacket.ETx = 0xa0ff;
163             //Display_Packet(tempPacket);       //Display's the Packet Data
                    in a Human friendly Form
164
165         }
166
167     return(tempPacket);
168
169 }
170
171
172 int Incoming_Length(int *incoming_data)
173 {
174     int *begin;
175     int dirty=0;
176
177     begin = incoming_data;
178     while(dirty != 1)
179     {
180         if (*incoming_data == 255)
181         {
182             incoming_data--;
183             if (*incoming_data == 160)
184                 dirty=1;
185             incoming_data++;
186         }
187         incoming_data++;
188     }
189
190     dirty = incoming_data - begin;
191     return(dirty);
192
193 }
194
195 int Check_Incoming_CRC(int *incoming_data, int length)
196 {
197
198     int CRCH = CRCPOLY >>8;
199     int CRCL = CRCPOLY & 0xFF;
200     int i,j, Carry, CRCDat1, CRCHigh, CRCLow;
201     int Returned_Value;
202
203     CRCHigh = incoming_data[0];
204
205     if (length > 1)
206         {
207             CRCLow = incoming_data[1];
208         }
209     else
210         CRCLow = 0;
211
212     for (i=1;i < length+1;i++)
213         {
214             if (length >2 && (i < length -1))
215                 {
216                     CRCDat1 = incoming_data[i+1];
217                 }
218             else
```

```
                {
                    CRCDat1 = 0;
                }
            for (j=1; j < 9;j++)
                {
                    Carry = (CRCHigh & 0x80) >> 7;
                    if (Carry > 0)
                        {
                            CRCHigh = CRCHigh ^ CRCH;
                            CRCLow = CRCLow ^ CRCL;
                        }
                    CRCHigh = (CRCHigh << 1) & 0xFF;
                    CRCHigh = CRCHigh + ((CRCLow & 0x80) >> 7);
                    CRCLow = (CRCLow << 1) & 0xFF;
                    CRCLow = CRCLow + ((CRCDat1 & 0x80) >> 7);
                    CRCDat1 = (CRCDat1 << 1)& 0xFF;
                }
        }

    Returned_Value = (CRCHigh << 8) + CRCLow;
    if (Returned_Value != 0)
        printf ("\r\nCRC Calculation Error:%4.4x!!!!!\r\n\n",Returned_Value);
    return Returned_Value;



}

int *Remove_ETx_Sequencing(int *incoming_data)
{

    int dirty;
    int i=0, j=0;
    int *temp = malloc(1024);
    int Bad_Packet=0;
    int Escapes=0;
    dirty = FALSE;
    while(dirty != TRUE)
        {

            if (incoming_data[i] == 0x10)
                {

                    switch(incoming_data[i+1])
                        {
                            case 0x10:
                                temp[j] = incoming_data[i+1];
                                i +=2;
                                Escapes++;
                                break;
                            case 0xa0:
                                temp[j] = incoming_data[i+1];
                                i +=2;
                                Escapes++;
                                break;
                            case 0xff:
                                temp[j] = incoming_data[i+1];
                                i +=2;
                                Escapes++;
                                break;
                            default:
                                //Singularity Escape Character found in
                                    Packet!
                                printf("Singularity in Escape Character
                                    Sequence - Bad Packet\r\n");
                                temp[j] = incoming_data[i];
                                Bad_Packet++;
                                i++;
                                break;
                        }

                }
            else
```

```
290                     {
291                         temp[j] = incoming_data[i];
292                         i++;
293                     }
294
295
296
297                 j++;
298                 if (incoming_data[j] == 255)
299                 {
300                     if (incoming_data[j-1] == 160)
301                     {
302                         dirty = TRUE;
303                     }
304                 }
305             }
306         temp[j] = 0xff;
307         j++;
308
309         /*printf("Packet with removed Escape Character bytes is:\n\t");
310         for (i=0;i<j;i++)
311             {
312                 printf("%2.2x",temp[i]);
313             }
314         */
315         if (Escapes>0)
316         {
317             printf("Escapes:%d\r\n",Escapes);
318             printf("Packet with removed Escape Character bytes is:\r\n");
319                 for (i=0;i<(j-Escapes);i++)
320                 {
321                     printf("%2.2x ",temp[i]);
322                 }
323             printf("\r\n");
324
325
326         }
327         if (Bad_Packet != 0)
328             temp = NULL;
329
330         //printf("\nEnd\n");
331         return temp;
332         free(temp);
333
334 }
```

B.3 Send Commands

B.3.1 mtnx_send.h

```
1
2  /***************Function Definitions***************/
3  int Send_New_Packet(FILE *fd, struct DATA_PACKET Data_Packet_to_send);  //New
       Prototype - Working
4
5
6  //Upper Level Packet Functions
7  //int Send_Packet(int fd, struct DATA_PACKET Data_Packet_to_send);
8
9  //struct DATA_PACKET Get_Packet(int fd);          //Returns a packet from the
       Data_line
10
11 struct DATA_PACKET Append_CRC(struct DATA_PACKET Data_Packet_to_Append);
               //Appends CRC to Packet!
```

```
12
13  int *Add_Ints_ETx_Sequencing(int *tempintarray, struct DATA_PACKET
        Data_Packet_to_include_ETx_seqn);    //Adds prototype escape Character
        sequence
14
15  int Calculate_CRC(struct DATA_PACKET Data_Packet_to_check);     //NEEDS TO BE
         MODIFIED
16
17  int * ByteArray(struct DATA_PACKET Data_Packet_to_Convert);              //
        Converts the values of the DataPacket to a Byte Array
18
19  int Byte_Length(struct DATA_PACKET Data_Packet_to_measure);              //
        Returns the number of bytes in the Packet
20
21  int Check_CRC(struct DATA_PACKET Data_Packet_to_check);                  //
        Returns True if Correct else returns False
22
23
24  /***********END Function Definitions***************/
```

B.3.2 mtnx_send.c

```
1   /*New Program to take data only when there is data on the line*/
2
3   #include <stdlib.h>  /* Dirk added Standard Library Header */
4   #include <stdio.h>   /* Standard input/output definitions */
5   #include <string.h>  /* String function definitions */
6   //#include <unistd.h>  /* UNIX standard function definitions */
7   #include <fcntl.h>   /* File control definitions */
8   #include <errno.h>   /* Error number definitions */
9   //#include <termios.h> /* POSIX terminal control definitions */
10
11  #include <asm/unistd.h>  /* UNIX standard function definitions */
12  #include <asm/termios.h> /* POSIX terminal control definitions */
13
14  #include <sys/types.h>
15  #include <sys/stat.h>
16
17  #include "mtnx.h" /*Our Header File with Packet Structures*/
18  #include "mtnx_send.h"
19  #include <time.h> /*New Header File for Pauses */
20
21  #define _POSIX_SOURCE 1 /* POSIX compliant source */
22  #define FALSE 0
23  #define TRUE 1
24
25  #define CRCPOLY 0x9005
26
27
28  int Send_New_Packet(FILE *fd, struct DATA_PACKET Data_Packet_to_send)
29  {
30      int Packet_Length, check_write, i , dirty, *begin;
31      int *temp = malloc(1024);
32
33      dirty= FALSE;
34      begin = temp;
35      Packet_Length = 0;
36
37      if (Data_Packet_to_send.CRC_CHECK == -1)
38      {
39          printf("Appending CRC Value to packet\r\n");
40          Data_Packet_to_send = Append_CRC(Data_Packet_to_send);
41      }
42
43      Check_CRC(Data_Packet_to_send);
44
45      temp = Add_Ints_ETx_Sequencing(temp, Data_Packet_to_send);
46
```

```
47      //Now see how long the packet is with the Escape Character Sequences by
            counting
48      while(dirty != TRUE)
49      {
50          if (*temp == 0xff)
51          {
52              temp--;
53              if (*temp == 0xa0)
54              {
55                  dirty = TRUE;
56                  temp = begin;
57              }
58              else
59              {
60                  temp+=2;
61
62              }
63              Packet_Length++;
64          }
65          else
66          {
67              temp++;
68              Packet_Length++;
69          }
70      }
71      dirty = FALSE;
72      printf("Sending: %d Bytes\n", Packet_Length);
73
74      for(i=0;i<Packet_Length;i++)
75      {
76          printf("%2.2x",*temp);
77          check_write = fwrite(temp, 1 , 1, fd);
78          temp++;
79      }
80
81
82      fflush(fd);
83
84      //temp = ByteArray(Data_Packet_to_send);
85      /*
86      for (i=0; i < Byte_Length(Data_Packet_to_send)+2 ; i++)
87      {
88          printf("%2.2x",*temp);
89          temp++;
90      }
91      printf("\n");
92      */
93
94      //Packet_Length = Incoming_Length(temp);
95
96      //check_write = fwrite(temp, 1 , Packet_Length, fd);
97      check_write = Packet_Length;
98
99      //Error Messages
100     if (check_write < Packet_Length)
101     {
102         if (check_write == -1)
103         {
104             printf("ERROR - Could not write Packet!\n");
105         }
106         else
107         {
108             printf("Write Error - Only %d out of %d Bytes Written\n",
                        check_write, Packet_Length);
109         }
110     }
111     else
112     {
113         printf("Wrote Packet to Port\n");
114     }
115
116
117 }
```

```
118
119
120
121 int *Add_Ints_ETx_Sequencing(int *tempintarray, struct DATA_PACKET
        Data_Packet_to_include_ETx_seqn)
122 {
123     int *begin_int_array;
124     int i,k,Escapes;
125     k=0;
126     Escapes=0;
127
128     //tempintarray = malloc(1024);
129     begin_int_array = tempintarray;
130
131
132     for(i=0; i < Byte_Length(Data_Packet_to_include_ETx_seqn);i++)
133         {
134             if (ByteArray(Data_Packet_to_include_ETx_seqn)[i] == 0xFF ||
                    ByteArray(Data_Packet_to_include_ETx_seqn)[i] == 0x10)
135                 {
136                     *tempintarray = 0x10;
137                     tempintarray++;
138                     *tempintarray = ByteArray(Data_Packet_to_include_ETx_seqn
                            )[i];
139                     tempintarray++;
140                     printf("Escape Characters 0x10 inserted in Byte[%d]\n", i
                            );
141                     Escapes++;
142                 }
143             else
144                 {
145                     *tempintarray = ByteArray(Data_Packet_to_include_ETx_seqn
                            )[i];
146                     tempintarray++;
147                 }
148         }
149
150     if (Escapes >0)
151     {
152         printf("Total Number of Escape Characters in Packet = %d\n", Escapes)
                ;
153     }
154
155
156     //Add end of Packet Characters
157     *tempintarray= 0xa0;
158     tempintarray++;
159     *tempintarray= 0xff;
160
161
162     //For Debugging Purposes
163
164     k = tempintarray - begin_int_array + 1;
165     tempintarray = begin_int_array;
166     /*
167     for (i=0;i<k;i++)
168     {
169         printf("%2.2x",*tempintarray);
170         tempintarray++;
171
172     }
173     printf("\n");
174     */
175
176     return(begin_int_array);
177
178 }
179
180
181
182
183
184 int Check_CRC(struct DATA_PACKET Data_Packet_to_check)
```

```
{
    int temp;
    if (Data_Packet_to_check.CRC_CHECK != -1)
        {
            printf("The CRC value is:%4.4x - ",Data_Packet_to_check.CRC_CHECK
                );
            temp = Calculate_CRC(Data_Packet_to_check);
            if (temp == 0)
                {
                    printf("CRC Check has returned zero - correct packet\n");
                    return TRUE;
                }
            else
                {
                    printf("ERROR CRC Check has returned a non-zero value
                        %4.4x \n",temp);
                    return FALSE;
                }
        }
}



int Byte_Length(struct DATA_PACKET Data_Packet_to_measure)
{
    int temp;
    temp = Data_Packet_to_measure.data_length + 8;
    if (Data_Packet_to_measure.CRC_CHECK != -1)
    {
        temp +=2;
    }
    //printf("The Packet has %d bytes in total\n",temp);
    return temp;

}


int *ByteArray(struct DATA_PACKET Data_Packet_to_Convert)
{
    int *tempPacket = malloc(1024);
    int temp;

    int upperByte, lowerByte;
    int Length_of_Data_Integers;
    int i = 0;
    int counter=0;
    int dirty=0;

    temp = Data_Packet_to_Convert.Source_Device_Address;
    tempPacket[0] = temp >>8;
    tempPacket[1] = temp & 0xFF;
    //printf("Byte 1: %x\tByte 2: %x\n",tempPacket[0],tempPacket[1]);

    temp = Data_Packet_to_Convert.Destination_Device_Address;
    tempPacket[2] = temp >>8;
    tempPacket[3] = temp & 0xFF;
    tempPacket[4] = Data_Packet_to_Convert.Protocol_Identifier & 0xFF; //
        Ensure one Byte;
    tempPacket[5] = Data_Packet_to_Convert.Command & 0xFF; //Ensure one Byte;
    counter = 6;
    for (i = 0;i < Data_Packet_to_Convert.data_length;i++)
        {
            tempPacket[counter] = Data_Packet_to_Convert.INTDATA[i];
            counter++;
        }

    //counter++;
    tempPacket[counter] = Data_Packet_to_Convert.Packet_Number >>8;
    counter++;
    tempPacket[counter] = Data_Packet_to_Convert.Packet_Number & 0xFF;

    if (Data_Packet_to_Convert.CRC_CHECK != -1)
```

```
255         {
256             temp = Data_Packet_to_Convert.CRC_CHECK;
257             counter++;
258             tempPacket[counter] = Data_Packet_to_Convert.CRC_CHECK >>8;
259             counter++;
260             tempPacket[counter] = Data_Packet_to_Convert.CRC_CHECK & 0xFF;
261         }
262
263
264
265
266 //    for (i=0; i< counter+1;i++)
267 //      {
268 //          printf("TempPacket[%d] Byte Value:%2.2x\n",i,tempPacket[i]);
269 //      }
270
271     return tempPacket;
272     free(tempPacket);
273
274 }
275
276
277
278
279
280
281
282
283 struct DATA_PACKET Append_CRC(struct DATA_PACKET Data_Packet_to_Append)
284 {
285     if (Data_Packet_to_Append.CRC_CHECK != -1)
286         {
287             printf("\nCRC Already calculated and Appended!\n");
288         }
289     else
290         {
291             Data_Packet_to_Append.CRC_CHECK = Calculate_CRC(
                    Data_Packet_to_Append);
292         }
293     return Data_Packet_to_Append;
294 }
295
296
297 int Calculate_CRC(struct DATA_PACKET Data_Packet_to_check)
298 {
299     int CRCH = CRCPOLY >>8;
300     int CRCL = CRCPOLY & 0xFF;
301     int i,j, Carry, CRCDat1, CRCHigh, CRCLow;
302     int Returned_Value;
303
304     CRCHigh = ByteArray(Data_Packet_to_check)[0];
305
306     if (Byte_Length(Data_Packet_to_check) > 1)
307         {
308             CRCLow = ByteArray(Data_Packet_to_check)[1];
309         }
310     else
311         CRCLow = 0;
312
313     for (i=1;i < (Byte_Length(Data_Packet_to_check)+1);i++)
314         {
315             if (Byte_Length(Data_Packet_to_check) >2 && (i < Byte_Length(
                    Data_Packet_to_check) -1))
316                 {
317                     CRCDat1 = ByteArray(Data_Packet_to_check)[i+1];
318                 }
319             else
320                 {
321                     CRCDat1 = 0;
322                 }
323             for (j=1; j < 9;j++)
324                 {
325                     Carry = (CRCHigh & 0x80) >> 7;
```

```
326                         if (Carry > 0)
327                             {
328                                 CRCHigh = CRCHigh ^ CRCH;
329                                 CRCLow = CRCLow ^ CRCL;
330                             }
331                         CRCHigh = (CRCHigh << 1) & 0xFF;
332                         CRCHigh = CRCHigh + ((CRCLow & 0x80) >> 7);
333                         CRCLow = (CRCLow << 1) & 0xFF;
334                         CRCLow = CRCLow + ((CRCDat1 & 0x80) >> 7);
335                         CRCDat1 = (CRCDat1 << 1)& 0xFF;
336                     }
337             }
338
339         Returned_Value = (CRCHigh << 8) + CRCLow;
340         printf ("\nCRC Calculation:%4.4x\n\n",Returned_Value);
341         return Returned_Value;
342
343 }
```

Appendix C

Makefile listing

```
1  AXIS_USABLE_LIBS = GLIBC
2  include $(AXIS_TOP_DIR)/tools/build/Rules.axis
3
4  PROGS = bluez-libs bluez-utils
5
6  INSTDIR    = $(prefix)/bin
7  INSTMODE   = 0755
8  INSTOWNER = root
9  INSTGROUP = root
10
11 BLUEZLIBDIR = $(AXIS_TOP_DIR)/apps/bluetooth/bluez-libs-2.7
12 BLUEZUTILSDIR = $(AXIS_TOP_DIR)/apps/bluetooth/bluez-utils-2.7
13
14 .PHONY: $(PROGS)
15
16 all: $(PROGS)
17
18 install: $(PROGS) bluez-utils-install
19
20 clean: bluez-libs-clean bluez-utils-clean
21
22 bluez-libs:
23         @if [ ! -f $(BLUEZLIBDIR)/Makefile ]; then \
24                 cd $(BLUEZLIBDIR); \
25                 CC="$(CC)" LD="$(LD)" LDFLAGS="$(LDFLAGS)" LDLIBS="$(LDLIBS)"
                        ./configure \
26                 --prefix=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu \
27                 --sysconfdir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu/etc \
28                 --host=cris-axis-linux-gnu \
29                 --libdir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu \
30                 --includedir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu/
                        include \
31                 --with-kernel=$(AXIS_TOP_DIR)/os/linux ; \
32         fi
33         @$(MAKE) -C $(BLUEZLIBDIR); \
34
35
36
37 bluez-utils: bluez-libs-install
38         if [ ! -f $(BLUEZUTILSDIR)/Makefile ]; then \
39                 cd $(BLUEZUTILSDIR); \
40                 CC="$(CC)" LD="$(LD)" LDFLAGS="$(LDFLAGS)" ./configure \
41                 --prefix=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu \
42                 --sysconfdir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu/etc \
43                 --host=cris-axis-linux-gnu \
44                 --libdir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu \
45                 --includedir=$(AXIS_TOP_DIR)/target/cris-axis-linux-gnu/
                        include \
```

```
46                  --with-kernel=$(AXIS_TOP_DIR)/os/linux \
47                  --cache-file=/dev/null --without-glib ; \
48          fi
49          @$(MAKE) -C $(BLUEZUTILSDIR); \
50
51
52  bluez-libs-install: bluez-utils
53          @$(MAKE) -C $(BLUEZLIBDIR) prefix=$(prefix) install
54
55  bluez-utils-install: bluez-libs-install
56          @$(MAKE) -C $(BLUEZUTILSDIR) prefix=$(prefix) install
57
58  bluez-libs-clean:
59          @if [ -f $(BLUEZLIBDIR)/Makefile ]; then \
60                  $(MAKE) -C $(BLUEZLIBDIR) clean; \
61          fi
62
63  bluez-utils-clean:
64          @if [ -f $(BLUEZUTILSDIR)/Makefile ]; then \
65                  $(MAKE) -C $(BLUEZUTILSDIR) clean; \
66          fi
67
68  bluetooth:
69          @if [ -f bluetooth/Makefile ]; then \
70                  $(MAKE) -C bluetooth/ hci_uart.o; \
71          fi
```

Appendix D

CD-Rom Insert