U.S.N.A.—Trident Scholar project report; no. 310 (2003)

Designing a Bluetooth-Based Wireless Network for Distributed Shipboard Monitoring and Control Systems

by

Midshipman Kenneth J Hoover, Class of 2003
United States Naval Academy
Annapolis, Maryland

____
(signature)

Certification of Advisors Approval

Professor Antal A. Sarkady
Electrical Engineering Department

____
(signature)

____
(date)

Commander Charles B. Cameron
Electrical Engineering Department

____
(signature)

____
(date)

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

____
(signature)

____
(date)

USNA-1531-2

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burder for this collection of information is estibated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burder to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (FROM - TO)
14-05-2003		xx-xx-2003 to xx-xx-2003

4. TITLE AND SUBTITLE Designing a Bluetooth-Based Wireless Network for Distributed Shipboard Monitoring and Control Systems Unclassified	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME AND ADDRESS US Naval Academy Annapolis, MD21402	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME AND ADDRESS ,	10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APUBLIC RELEASE
,

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A Bluetooth based network has been developed for monitoring and controlling power systems on board US Navy Vessels. This network is intended for distributed measurement and control and is ideally suited for controlling individual slave nodes. The slave nodes are individual electronic systems which collect information from many sensors and make appropriate control decisions based on the occurrence of well-defined events. Bluetooth is a low-cost, low-power wireless standard, which boasts a theoretical maximum baud rate of approximately 750 kbps. The Bluetooth standard allows networking of several slave nodes. An important advantage of this system is that it can be configured for many shipboard applications. This wireless standard uses a spread-spectrum modulation scheme that allows reliable communication between devices within several sub-networks (piconets) in the same physical vicinity. A robust wireless network has been designed to maintain reliable connectivity among nodes even when the communication channels are adversely affected by the opening and closing of watertight doors. For testing the throughput of the Bluetooth network a low-level C++ program has been designed. The program establishes a Bluetooth piconet and records transmission times for several different size files. Experimental performance data have been collected using several different Bluetooth packet types and network topologies. Initial tests were completed in a long hallway to provide baseline control data. In addition, shipboard testing was done on board the ex-USS America. These tests have proved the practicability of using a Bluetooth network for shipboard control and monitoring systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19. NAME OF RESPONSIBLE PERSON
			Same as Report (SAR)	144	Cornell, Elizabeth ecornell@dtic.mil
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER International Area Code Area Code Telephone Number DSN

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39.18

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including g the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
14 May 2003

3. REPORT TYPE AND DATE COVERED

4. TITLE AND SUBTITLE
Designing a bluetooth-based wireless network for distributed shipboard monitoring and control systems

5. FUNDING NUMBERS

6. AUTHOR(S)
Hoover, Kenneth J. (Kenneth James), 1980-

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Naval Academy
Annapolis, MD 21402

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Trident Scholar project report no. 310 (2003)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
This document has been approved for public release; its distribution is UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT: A Bluetooth based network has been developed for monitoring and controlling power systems on board US Navy Vessels. This network is intended for distributed measurement and control and is ideally suited for controlling individual slave nodes. The slave nodes are individual electronic systems which collect information from many sensors and make appropriate control decisions based on the occurrence of well-defined events. Bluetooth is a low-cost, low-power wireless standard, which boasts a theoretical maximum baud rate of approximately 750 kbps. The Bluetooth standard allows networking of several slave nodes. An important advantage of this system is that it can be configured for many shipboard applications. This wireless standard uses a spread-spectrum modulation scheme that allows reliable communication between devices within several sub-networks (piconets) in the same physical vicinity. A robust wireless network has been designed to maintain reliable connectivity among nodes even when the communication channels are adversely affected by the opening and closing of watertight doors. For testing the throughput of the Bluetooth network a low-level C++ program has been designed. The program establishes a Bluetooth piconet and records transmission times for several different size files. Experimental performance data have been collected using several different Bluetooth packet types and network topologies. Initial tests were completed in a long hallway to provide baseline control data. In addition, shipboard testing was done on board the *ex-USS America*. These tests have proved the practicability of using a Bluetooth network for shipboard control and monitoring systems.

14. SUBJECT TERMS: shipboard uses of Bluetooth; frequency hopping spread spectrum (FHSS); piconet; scatternet; shipboard wireless network, fault tolerance; adhoc network.

15. NUMBER OF PAGES
141

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

U.S.N.A.—Trident Scholar project report; no. 310 (2003)

Designing a Bluetooth-Based Wireless Network for Distributed Shipboard Monitoring and Control Systems

Midshipman Kenneth J Hoover, Class of 2003
United States Naval Academy
Annapolis, Maryland

(signature)

Certification of Advisors Approval

Professor Antal A. Sarkady
Electrical Engineering Department

(signature)

(date)

Commander Charles B. Cameron
Electrical Engineering Department

(signature)

(date)

Acceptance for the Trident Scholar Committee

(signature)

(date)

Abstract

A Bluetooth based network has been developed for monitoring and controlling power systems on board US Navy Vessels. This network is intended for distributed measurement and control and is ideally suited for controlling individual slave nodes. The slave nodes are individual electronic systems which collect information from many sensors and make appropriate control decisions based on the occurrence of well-defined events. Bluetooth is a low-cost, low-power wireless standard, which boasts a theoretical maximum baud rate of approximately 750 kbps. The Bluetooth standard allows networking of several slave nodes. An important advantage of this system is that it can be configured for many shipboard applications. This wireless standard uses a spread-spectrum modulation scheme that allows reliable communication between devices within several sub-networks (piconets) in the same physical vicinity. A robust wireless network has been designed to maintain reliable connectivity among nodes even when the communication channels are adversely affected by the opening and closing of watertight doors. For testing the throughput of the Bluetooth network a low-level C++ program has been designed. The program establishes a Bluetooth piconet and records transmission times for several different size files. Experimental performance data have been collected using several different Bluetooth packet types and network topologies. Initial tests were completed in a long hallway to provide baseline control data. In addition, shipboard testing was done on board the *ex-USS America*. These tests have proved the practicability of using a Bluetooth network for shipboard control and monitoring systems.

Key Words: shipboard uses of Bluetooth; frequency hopping spread spectrum (FHSS); piconet; scatternet; shipboard wireless network, fault tolerance; adhoc network.

Acknowledgments

I would like to thank the United States Naval Academy Electrical Engineering Department for providing their support and equipment for this project. In particular, I would like to thank my advisors Professor Antal A. Sarkady and CDR Charles B. Cameron for their support, patience and instruction over the past year.

This project was suggested and sponsored by Henry Whitesel (Naval Surface Warfare Center, Carderock Division Code 853, Naval Business Center, Philadelphia, Pennsylvania 19112) and Jack Overby of the Machinery Research and Development Directorate at the Naval Surface Warfare Center Carderock Division (NSWCCD) in Philadelphia. Their support is greatly appreciated.

A special thanks to Dan Evans and Mic Robinson from the Naval Surface Warfare Center for helping to provide *ex-USS America* as a testing platform.

I would like to thank the electrical engineering technical staff at the Naval Academy for all their help and support, especially Bill Stanton and Daphi Jobe.

This project would not have been possible without the help of Professor Carl Wick from the Weapons and Systems Engineering Department and his crash course in *Borland C++ Builder*.

I would like to thank my family and friends for all of their support. I would especially like to thank my bride to be, Anna Wasienko, my mother and father, Kenneth and Susan Hoover, and my brother, Matthew Hoover, for patiently listening to me describe to them the intricate details of this project.

Finally, I would like to thank my Heavenly Father for the patience and endurance He has given me, the wisdom he has given to all those who have helped me along the way, and the comfort I have knowing that He has never left my side.

Contents

1 Project Description and Goals **8**

1.1 Introduction 8

1.2 Project Description 9

2 Bluetooth **13**

2.1 Bluetooth Basics 13

2.2 Bluetooth Networking 15

2.3 Interfacing a Bluetooth Device with a Host 17

2.4 Bluetooth Data Transmission 19

2.5 Bluetooth Transmission Rates 22

3 The Transmission Test Program **24**

3.1 Program Operation 27

4 Transmission Tests **32**

4.1 Hallway Tests 33

4.1.1 Test Setup 33

4.1.2 Test Results 35

4.2 Shipboard Tests 41

4.2.1 Test Setup 42

4.2.2 Test Results 43

4.3 Test Statistics 46

5 Conclusions **48**

6 Future Research **49**

A Bluetooth Include Functions **51**

A.1 bluetooth.h 51

A.2 BT_functions.h 56

A.3 BT_functions.cpp 62

B BTTest v. 1.1 **71**

B.1 BTTest1_1.h 71

B.2 BTTest1_1.cpp 75

C BTTest v. 3.1 **105**

C.1 BTTest3_1.h 105

C.2 BTTest3_1.cpp 109

List of Figures

1.1 Sinusoid and Pulse with Fourier Transforms 10
1.2 Intersymbol Interference 11

2.1 Bluetooth tranceiver module 14
2.2 Bluetooth Network 17
2.3 A Bluetooth Node, Composed of the Host and Host Controller. 18
2.4 Bluetooth HCI Event Packet[1] 19
2.5 Data encapsulation. 20
2.6 L2CAP Data Packet 20
2.7 ACL Data Packets 21
2.8 Illustration of baseband packet transmission 22

3.1 User Interface to the Transmission Test Program 25
3.2 Example of a test data file 26
3.3 Example of a test results output file 26
3.4 Bluetooth connection setup timing diagram 28
3.5 Flow Chart of the Test-data Transmission Routine 31

4.1 Bluetooth packet acknowledgement process [1] 32
4.2 Michelson hallway tests 33
4.3 Hallway test topology 34

4.4 Data Throughput vs. Door Position for DM3 Packet Type. (Error bars are placed at ± two standard deviations providing a certainty of 95%) 35
4.5 One-way Transmission Time vs. File Size 36
4.6 Data Throughput vs. Topology corrected, showing that throughput is largely independent of file size. 38
4.7 Standard transmission times 38
4.8 Properties of Baseband Packets [1] 39
4.9 File Size vs. Packet Retransmissions for Hallway Tests (Error bars are set at 95% certainty) 40
4.10 Hallway Piconet Test Topology 41
4.11 *ex-USS America* Tests 41
4.12 Shipboard Topology 42
4.13 Test Results for DM3 Packet Type. (Error bars are placed at ± two standard deviations providing a certainty of 95%) 43
4.14 ENS Estes' Powernode 44
4.15 File Size vs. Packet Retransmissions for Shipboard Tests (Error bars are set at 95% certainty) 45

List of Tables

3.1 Bluetooth Initialization Commands 27

4.1 Packet Retransmissions for the Hallway Test (DM1 Baseband Packet) 39

4.2 Packet Retransmissions for the Hallway Test (DM1 Baseband Packet) 44

4.3 Transmission Failures per Attempted Transmissions 46

4.4 Number of Failed Connections per Connection Attempts 47

Chapter 1

Project Description and Goals

1.1 Introduction

The Navy is always looking for efficient ways to improve its role in protecting our country. One method to accomplish this task is the reduction of ships' crew sizes. A military made up of an all-volunteer force is very expensive. A large amount of money is spent every year on training and salary for sailors and officers. Reducing crew sizes would greatly reduce the cost of the Navy as a whole, if this could be done without a loss of efficiency.

Increased automation on board ship is one way to make smaller crew sizes possible. A lot of man-hours are spent on board ships by sailors walking around the ship taking readings and recording the data into logbooks. Much of this time could be eliminated through the use of a computer network for centralized data logging and display. Such a network could also be employed in the area of damage control, thereby increasing the safety of the ship.

One proposed method of increasing the automation on board ship is the use of a wireless network. While a computer network such as the one described above could be implemented using a wired network, wires add a great deal of weight to the ship and decrease the structural and watertight integrity because of the holes which need to be cut into the bulkheads for the wires to pass from compartment to compartment. Along with the structural advantages

of a wireless network there are also financial advantages. Maintenance and upgrades both are much easier when there are fewer wires to deal with, because there are no installation costs. A wired network is not easily or inexpensively reconfigured or upgraded. However, a wireless network is easily and inexpensively reconfigured and upgraded; one simply needs to download new software to reconfigure an entire network. Therefore, the advantages of a wireless network are increased automation, increased structural integrity, decreased weight, decreased installation costs (in both time and money), and increased reconfigurability.

The purpose of this proposed research is to determine the feasibility of a Bluetooth network in a ship-board environment.

1.2 Project Description

This research project uses *radio frequency* (RF) attenuation data obtained by a previous Trident Scholar at the U.S. Naval Academy, ENS Daniel R. J. Estes, completed in May 2001. He performed RF transmission tests using sinusoidal sources ranging from $0.8 - 2.5$ GHz in frequency on the *ex-USS America*, *USS Ross, USS Carr, USS Leyte Gulf,* and *USS Oscar Austin*[4]. During his research ENS Estes measured the signal attenuation through closed doors using five different geometries for the transmitter and receiver: open door in the direct path from transmitter to receiver, open door not in the direct path from transmitter to receiver, closed doors almost in the direct path from transmitter to receiver, a bulkhead with no open doors, and closed doors not in the direct path from transmitter to receiver. He concluded that the maximum RF signal attenuation was approximately 25 dB[4]. Bluetooth devices currently have a receiver sensitivity of -75 dBm. This means that when using the lowest power Bluetooth transmitter (0 dBm) in an environment with no external noise, a signal with 75 dB of signal path loss is still receivable. This is the equivalent of passing through three closed doors on board a ship. It is because of this characteristic of Bluetooth that we believe Bluetooth is capable of operating in a shipboard environment.

ENS Estes concluded from his research that wireless transmission is possible on board ships. However, he used only unmodulated sinusoidal carrier frequencies in his tests. Modulated carriers require much greater bandwidth.

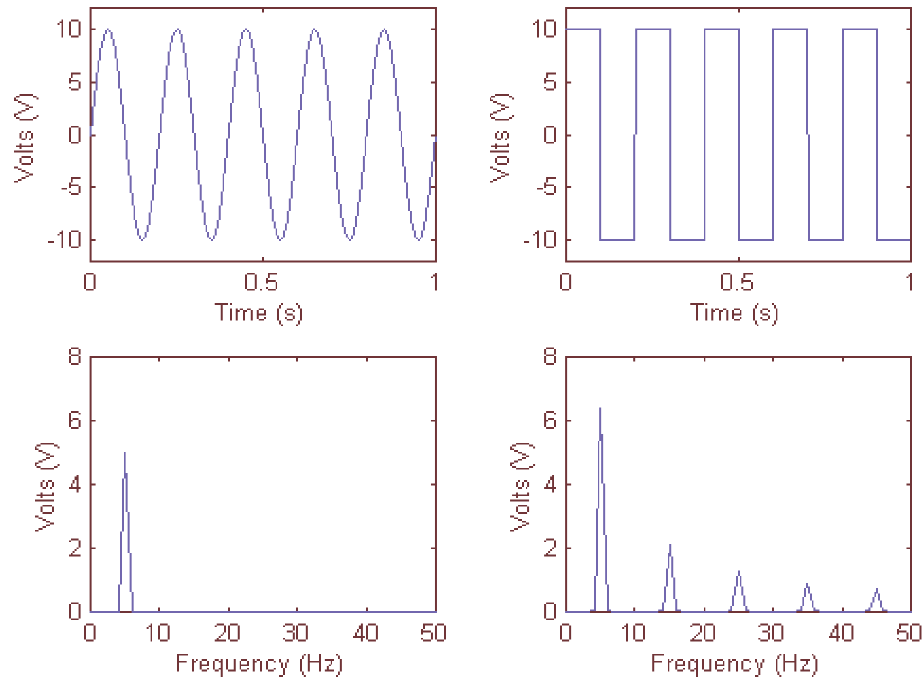

Figure 1.1: Sinusoid and Pulse with Fourier Transforms

For example, look at figure 1.1. This figure shows four plots. The top two plots are the time-domain representations of a sinusoidal signal and a square pulse. The bottom two pictures are the frequency-domain representations of these two signals. The sinusoidal signal occupies only one discrete frequency, whereas the square pulse requires considerably more bandwidth because it is made up of many harmonically related sinusoids (a *Fourier Series*). The sharp edges of the pulse are created by hundreds of *Fourier Series* components and therefore imply large bandwidth. Though actual digital transmission signals do not use square pulses, because of the large bandwidth associated with them, the signals do still occupy much more bandwidth than a single sinusoid. Our research entailed another series of shipboard wireless testing and therefore extended ENS Estes' work to a broader frequency

range.

Because signal interference may affect the phase component of a signal along with its magnitude, it is important to conduct a series of tests similar to those done by ENS Estes, but using actual data communication. When data are transmitted and attenuated through objects such as bulkheads and doors, a phenomenon known as *Intersymbol Interference* (ISI) may occur.

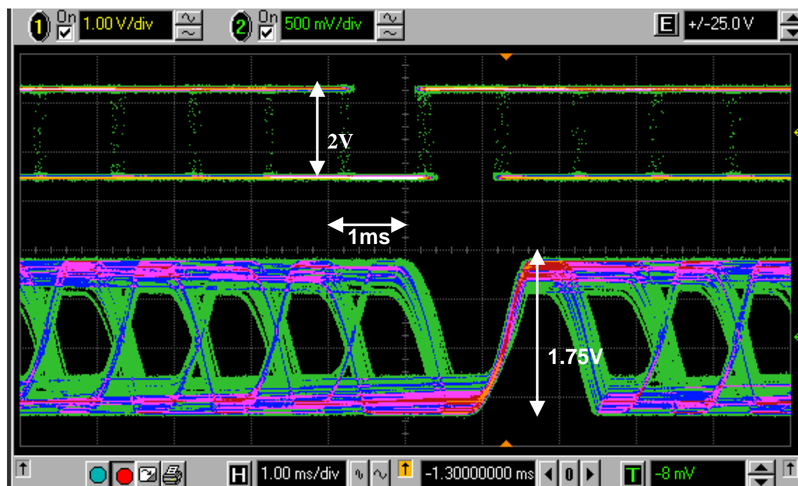

Figure 1.2: Intersymbol Interference

ISI occurs when a signal comprised of distinct time-domain symbols or pulses is distorted such that those pulses begin to “bleed” into one another. This “bleeding” leads to receiver errors in the reconstruction of binary digits (bits). Figure 1.2 shows an example of the effects of ISI. The figure, taken from an oscilloscope, shows the original signal and the signal after passing it through a filter which caused ISI to occur. The figure shows many traces of the same signals. The top signal is the original. It is a series of random square pulses two volts in amplitude. The bottom signal was created by passing the initial top signal through a filter that was used to model a transmission channel (*e.g.*, a wire or electromagnetic waves). The filter attenuated the signal (reduced its amplitude) and created ISI within the signal. From this figure you can see how the transitions from a high amplitude to a low amplitude are not as clear on the bottom signal as on the top. This is the problem with ISI. When the

transition of a signal from one state to another is not clear the signal may be misinterpreted by the receiver.

Having seen the effects of ISI on the signal itself what are the affects of ISI on the actual data sent? The most likely effect of ISI on digital data is *bit errors*. Bit errors are the alteration of a single bit or multiple bits of data during transmission. Since a bit is represented as a "0" or a "1", a bit error occurs when, through ISI or noise, a bit is altered either from a "1" to a "0", or *vice versa*. Small numbers of bit errors have little or no effect on some types of data (*e.g.*, voice and video), however a single bit error in other data (*e.g.*, files and control information) may require retransmission of that data.

Chapter 2

Bluetooth

This chapter provides some background information about the Bluetooth Wireless Standard. We will define terms relative to the Bluetooth standard, discuss the proposed uses for Bluetooth equipment, explain basic Bluetooth network operation, and discuss how to interface a Bluetooth transmitter with a host.

2.1 Bluetooth Basics

Engineers at Ericsson first conceived of a new standard for wireless ad-hoc networking in the mid 90's. The new standard was named after a king of Denmark who united Denmark and Norway and brought Christianity to Scandinavia.[2] That Bluetooth is a standard. It is not the name of a specific product. Different vendors design products which must conform to this standard.

The Bluetooth specification defines a short-range, low-power consumption, radio frequency (RF) technology for digital, wireless communication. Bluetooth is aimed at replacing the serial cables which are used on a day-to-day basis with many electronic devices, such as the cables connecting the mouse and keyboard to a personal computer. Bluetooth is also capable of coding, decoding, and transmitting voice data, making it applicable to cellular telephones and wireless headsets.

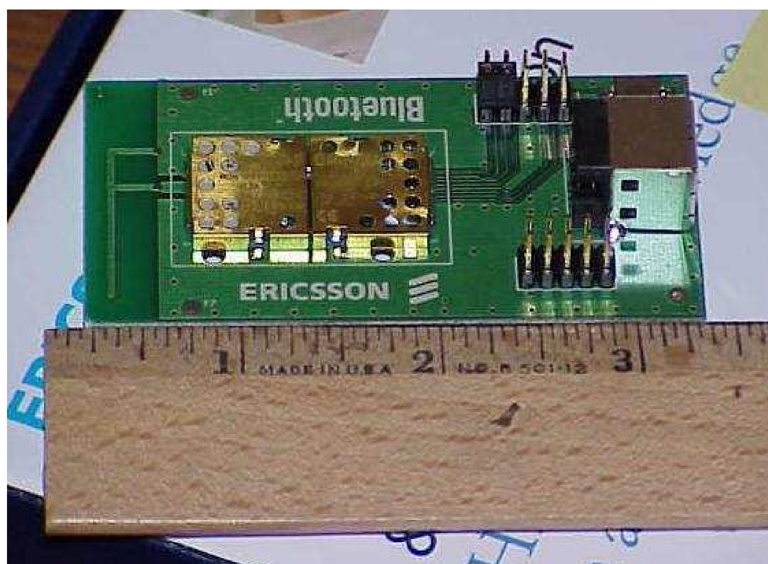

Figure 2.1: Bluetooth tranceiver module

Bluetooth transmits in the unlicensed frequency band (2.4–2.48 GHz). Because this frequency range is used by many other devices (*e.g.*, cordless telephones and microwave ovens), Bluetooth devices must address the problem of interference with these other devices. To accomplish this task, Bluetooth incorporates a *Frequency Hopping Spread Spectrum* (FHSS) scheme. The frequency hopping pattern is chosen by a *pseudonoise* (PN) code generator. Pseudonoise is defined as a seemingly random sequence or code. A pseudonoise code appears random to an uninformed observer; however, it can actually be seen to be predictable by a privileged user who knows the PN code generation system. A PN generator includes the circuitry together with a seed value (initial value fed to the circuitry), and a clock which produce PN codes. Spread spectrum refers to dividing transmissions among several frequency channels (79 channels for Bluetooth) as opposed to assigning them to a single channel for the entire transmission. Thus there are specific times during which transmissions may occur on a single frequency and these are called *time slots*. Bluetooth changes transmission channels at a rate of 1600 hops/s creating 625 µs time slots. For example, if a user wanted to transmit 1 kbyte of data, instead of sending all of the information on a single channel where a burst of noise could destroy the entire transmission, the data would be segmented and each segment transmitted on a different channel. Therefore, if a burst of RF noise on a specific channel

corrupts that information the majority of the data would be preserved. The fact that data were corrupted will be known at the receiver, so the corrupted data may be retransmitted. Frequency hopping refers to the order in which each transmission channel is used. In a frequency-hopping scheme, channels are chosen randomly. This builds security into the system. Only the transmitting and receiving device will know what the next transmission channel is because they will be synchronized before any transmissions take place. Along with FHSS, Bluetooth also implements several error-detection and -correction techniques. One of which is Cyclic Redundancy Codes (CRC). A CRC is a series of bits appended at the end of a data packet with which the receiver's hardware can determine if errors are present in the received data. If an error is present the receiver will request a retransmission from the originating device.

2.2 Bluetooth Networking

A Bluetooth network is made up of master and slave devices. The designation of a device as a master or slave occurs as the network is being established. This designation is in no way indicative of the capabilities of the device: it merely designates which device will control network operations. The master device determines the frequency-hopping pattern and synchronizes the PN generators of other devices, assigns network addresses to other devices, and handles all communication procedures between devices. The master controls the network.

The most basic Bluetooth network is a single point-to-point link. A point-to-point link is a communication link from one remote device to another remote device. We will use this type of network as an example of how to establish a Bluetooth network.

Bluetooth network setup is accomplished completely by the software and hardware within a given Bluetooth device. No input from the user is necessary. The first step to establishing a network is for one device to execute an *inquiry*. During an inquiry the device is sending

signals and waiting for replies to determine if there are any other Bluetooth devices within range. When a second Bluetooth device receives this inquiry signal it will respond with, among other information, its Bluetooth Device Address (BD_ADDR). Upon receipt of this response the inquiring device becomes master of the network and will use the BD_ADDR of the second device to contact that device until a communication link is established.

This leads to the next step: creating a link between devices. Bluetooth provides two types of communication links between devices: Asynchronous Connection-Less (ACL) links and Synchronous Connection-Oriented (SCO) links. SCO links are reserved for voice traffic so we will not go into detail about them. ACL links are used for data traffic. They can be point-to-point or point-to-multipoint (broadcast) transmissions and most ACL communications can be retransmitted in case of errors. In order to establish an ACL link the master will enter a *page mode* in which it will send connection requests to devices for which it has the BD_ADDR. In a two–device network, that would be the only device which responded to the master's inquiry. The slave device then has the option of accepting or rejecting the connection request. Assuming that the slave accepts the connection request it will then transmit more information about itself to the master. The master will use this information to establish rules for transmissions on this link, such as data transmission rate or encryption type. Once an ACL link is established, data transmission between master and slave can take place.

The basic multi-slave Bluetooth network is a *piconet* (Figure 2.2). A piconet is defined as a network consisting of a master and up to seven slaves. The master and slaves of a piconet form a star topology. All transmissions must go through the master. For example if Slave One wants to communicate with Slave Two, Slave One must first send the data to the master and then the master will send the data to Slave Two.

The procedures to establish a link are the same in a piconet as described above for a point-to-point connection. The only difference is that an inquiry may result in several devices responding instead of only one. When the master is ready to create a link it can

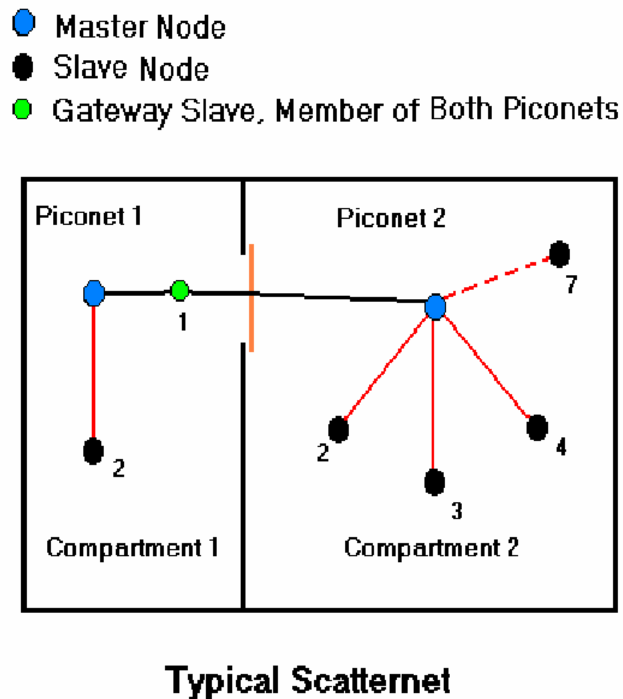

Figure 2.2: Bluetooth Network

choose with which devices it wants to create a connection.

2.3 Interfacing a Bluetooth Device with a Host

The first step to creating a Bluetooth network is to create an interface or method of communication from the Bluetooth device to a host. A host is the electronic device which will be running Bluetooth applications (*e.g.*, a computer, cell phone, or Palm Pilot). A host and Bluetooth transceiver interfaced together make up a Bluetooth master or slave (figure 2.3. The majority of the transmission procedures and link policy controls for Bluetooth are taken care of by the *firmware* included in the Ericsson Bluetooth module which we used. Firmware is code which is written in read-only memory (ROM), normally within a microcontroller. This firmware and the associated hardware on the Bluetooth transceiver which handle the interface with the host are collectively known as the host controller. The protocols and code used in dealing with this interface are known as the *Host Controller*

Interface (HCI) layer.

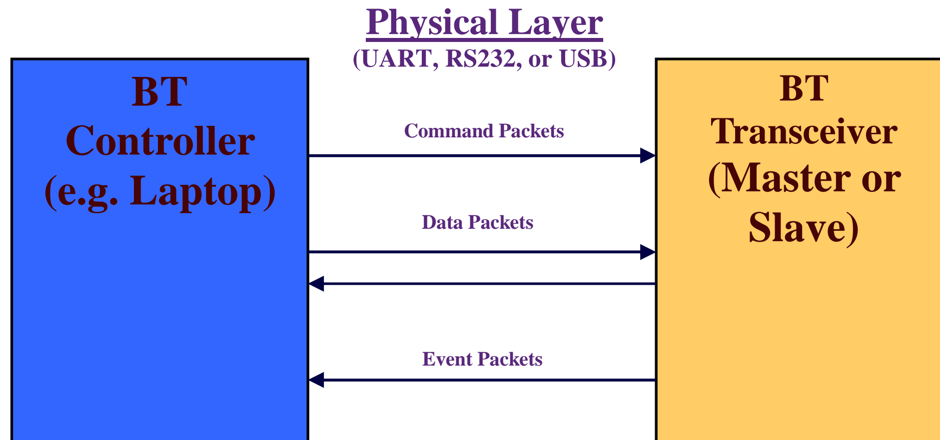

Figure 2.3: A Bluetooth Node, Composed of the Host and Host Controller.

The HCI is the Bluetooth networking layer that provides a standard interface for a host to communicate with a Bluetooth transceiver through a Universal Serial Bus (USB), RS-232 serial communication port, or a Universal Asynchronous Receiver-Transmitter (UART) serial connection. A host passes commands and data to the host controller, which then interprets the commands and passes instructions to the *Link Manager Protocol* (LMP) layer, which is the Bluetooth layer that controls the actual wireless link and data transmission.

All information that is passed from the host to the host controller (or *vice versa*) is passed in packets. A packet is basically a number of bytes arranged in a standard order so that they can be interpreted by a receiving device. Bluetooth uses four types of packets: command packets, event packets, and ACL and SCO data packets. ACL and SCO data packets are used for transmitting the ACL and SCO data, respectively. Command and event packets are used to transmit control or status information (figure 2.3). Command packets are sent only from the host to the host controller and are instructions to be executed by the Bluetooth device. Event packets are sent only from the host controller to the host and are used to inform the host of the status of requested commands or events that take place within the network or within the Bluetooth transceiver itself. Figure 2.4 shows the general layout of an

event packet.

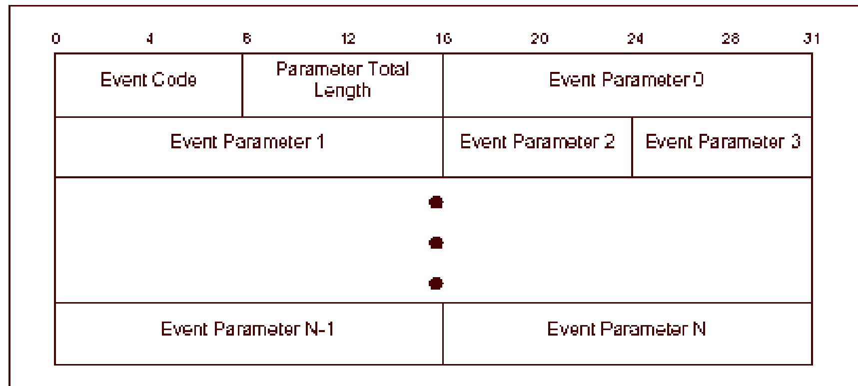

Figure 2.4: Bluetooth HCI Event Packet[1]

2.4 Bluetooth Data Transmission

Once a Bluetooth link is set up and, in the case of ACL data, once a connection has been successfully established, it is possible to transmit data. In order to prepare the data for transmission the host must first encapsulate the data (figure 2.5). Encapsulated data is referred to as a data packet. Data encapsulation is analogous to placing a letter inside an addressed envelope. When data is encapsulated, a header (and sometimes a trailer) is appended to the beginning (and end) of the data. The information contained in a header is similar to an address on a letter. The header contains information regarding the recipient of the packet, the size of the packet, and the type of data contained in the packet.

In the case of ACL data, the Bluetooth standard defines two types of packets in which the data must be encapsulated prior to transmission. Each type of packet consists of a packet header followed by the data payload. The first packet type is the *Logical Link Control and Adaptation Protocol* (L2CAP) packet. The L2CAP basically determines the general type of data being received and, if several programs are using the same Bluetooth link, the L2CAP determines to which program the data belongs. The purpose then of the L2CAP packet is to route the data to whichever application needs it. Encapsulation of data in an L2CAP packet

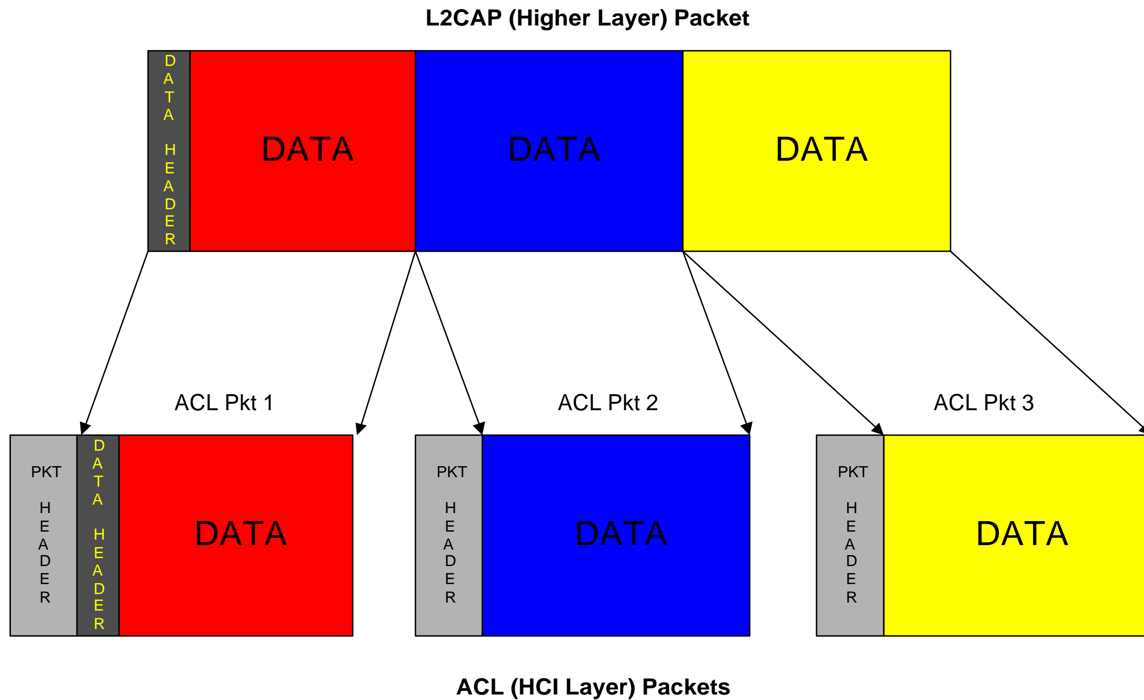

Figure 2.5: Data encapsulation.

occurs as shown in figure 2.6. The packet header is composed of a two-byte data-length parameter followed by a two-byte channel identifier (CID). The packet-length parameter informs the receiver of the size of the data payload in bytes (65,535 bytes maximum), while the CID is used to route the data to the correct application. Directly following the CID is the actual data (*i.e.*, the payload of the L2CAP packet). This L2CAP packet is then itself encapsulated in one or several ACL data packets.

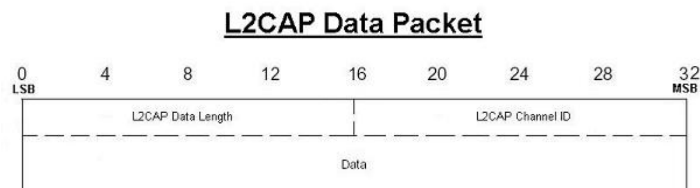

Figure 2.6: L2CAP Data Packet

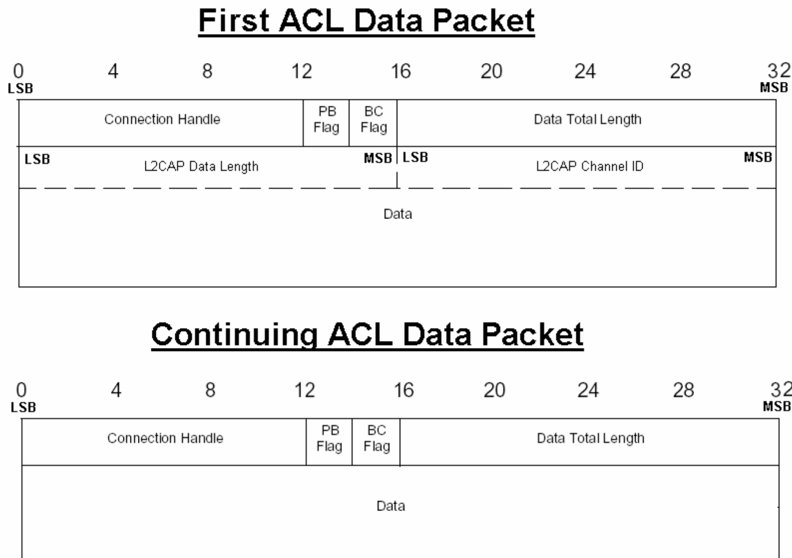

Figure 2.7: ACL Data Packets

An ACL packet header (figure 2.7) consists first of a twelve-bit connection handle. The connection handle designates the connection over which the packet is intended to be transmitted. Following the connection handle are two two-bit flags: the packet boundary (PB) flag and the broadcast (BC) flag. The PB flag is used to designate whether the given ACL packet is the first packet of an L2CAP message or a continuing fragment of an L2CAP message. The BC flag designates whether the ACL packet is intended for one specific slave or the whole piconet (multicasting). The last part of the ACL packet header is a two-byte data length. This parameter specifies the length of the data payload (in bytes) contained in the ACL packet. The maximum size of an ACL data packet is determined by the buffer size of the host controller. If an L2CAP packet is larger than the maximum size of an ACL packet then the L2CAP packet must be divided up amongst several ACL packets for transmission. For example the Ericsson Bluetooth transceiver we used has an ACL buffer size of 672 bytes. Therefore, if an L2CAP packet is larger than 672 bytes it will require several ACL data packets to transmit that one L2CAP packet.

Encapsulation of an L2CAP packet within ACL packets takes place as shown in figure 2.7. The first ACL packet contains (as ACL data payload) the packet header of the L2CAP packet followed by as much L2CAP data payload as needed to fill the ACL packet. The remaining L2CAP data payload is then encapsulated in as many ACL packets as needed. The ACL packets are then transmitted via UART or USB to the host controller on the Bluetooth transceiver where they are then sent over the RF link to the receiving device.

2.5 Bluetooth Transmission Rates

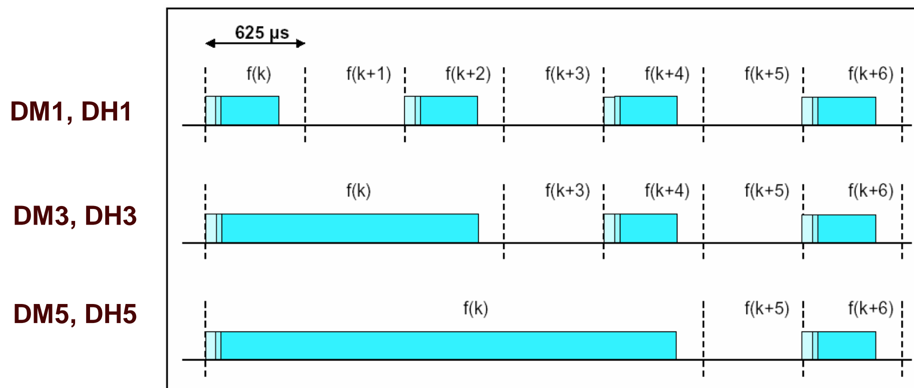

Figure 2.8: Illustration of baseband packet transmission

The Bluetooth standard provides for six different transmission rates made possible by six different types of baseband data packets. Baseband data packets are different from the packets previously discussed. When received by the Bluetooth transceiver the ACL packets are broken up and their data are placed in baseband packets which are the packets actually transmitted over the wireless channel. The six types of baseband packets are: DM1, DH1, DM3, DH3, DM5, and DH5 where the “D” signifies a data packet; “M” or “H” signifies a medium- or high-speed packet respectively; and the numbers 1, 3, and 5 signify the number of time slots utilized by the packet. The difference between medium- and high-speed packets is that high-speed packets have fewer error detection bytes and thus have a higher data rate. The other key to increasing data rate is the number of time slots used by each packet.

The more time slots used by a packet the faster the data rate of that packet. This is because as more time slots are being used for a single packet, more time is being allowed for that transmission to take place, as opposed to halting transmission and continuing during a subsequent time slot. The second example uses more header bytes (thus more overhead) slowing down transmission of actual data. This is why the DM1 and DH1 packets are the slowest. Figure 2.8 gives a graphical depiction of the baseband data packets.

Chapter 3

The Transmission Test Program

We have created two test programs along with the associated graphical user interfaces (GUI's) for running transmission tests and logging test data (figure 3.1). One program is designed to be used only for a two device (point-to-point) network. The second program is capable of conducting transmission tests between two or three devices. The program that will be discussed in detail here is the point-to-point program. The goal of the test program is to determine the performance of a Bluetooth link through a closed watertight door onboard a United States Naval vessel. The value measured by the test program is the round-trip transmission time for Bluetooth data transmissions. From the transmission time, a value for data throughput can be calculated using equation (3.1), where R is the data throughput in bits/s, n_{data} is the number of data bytes transfered, and $t_{\text{round trip}}$ is the round trip transmission time in seconds.

$$R = \frac{8n_{\text{data}}}{t_{\text{round trip}}/2} = \frac{16n_{\text{data}}}{t_{\text{round trip}}} \tag{3.1}$$

The test program is designed for measuring round trip transmission times of small (*i.e.*, $10 - 6500$ byte) text files across a Bluetooth link. To accomplish this the program reads in two specially created test data files, then times and records the round-trip transmission times for each file and repeats this for a number of trials specified by the user. The test

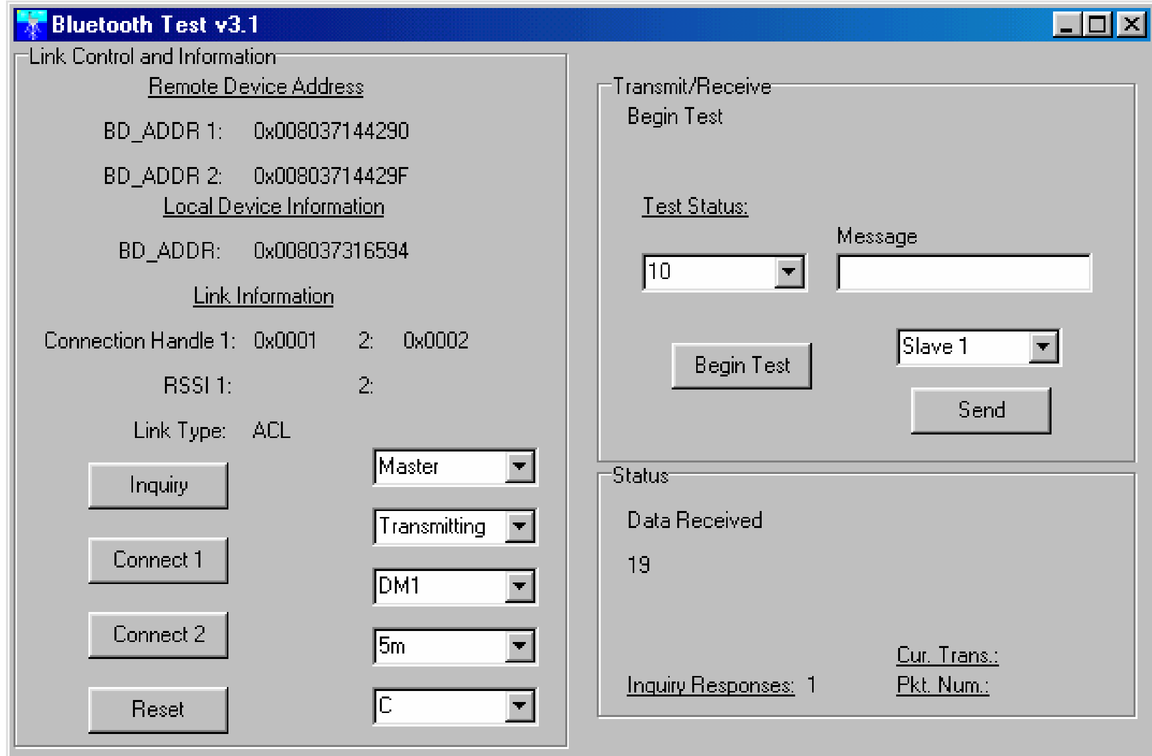

Figure 3.1: User Interface to the Transmission Test Program

program has been created such that it may be used as either a master or a slave, and may be used as either the testing platform (transmitting the files and recording the test data) or as the receiving platform (simply receiving and retransmitting the test data).

The test data files (figure 3.2) are specially formatted to be read into the transmission test program. The file begins with an integer representing the size of the file, followed by four characters to reserve buffer space for the L2CAP header, and finally the test data itself (*i.e.*, a simple character string).

Upon completion of each transmission the round-trip transmission time in seconds is recorded in a text file of comma-separated values which may be read into an Excel spreadsheet. In order to make data logging easier, information such as the time and date a given test was performed, distance between nodes, topology, master and slave Bluetooth addresses, receive signal strength (dB), baseband packet type, and test file sizes (in bytes) are recorded

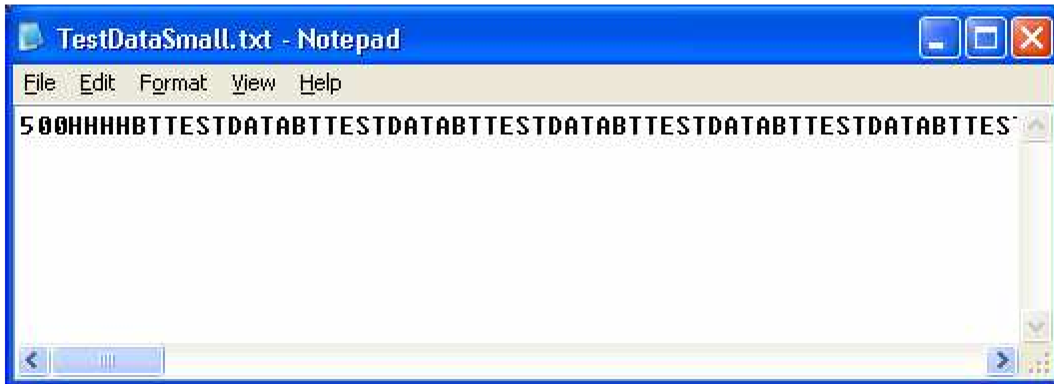

Figure 3.2: Example of a test data file

at the start of each test. Distance between nodes, topology, and baseband packet type are each selected by the user prior to beginning a test and written to the test data file. Time and date, master and slave Bluetooth addresses, receive signal strength, and test file sizes are each either read from the computer or from the Bluetooth transceiver automatically and logged in the output data file. Once a test has begun the transmission number, round trip transmission time (for the small and large file), processing time (from the remote computer), and number of errors is recorded for each transmission. An example output file is shown in figure 3.3.

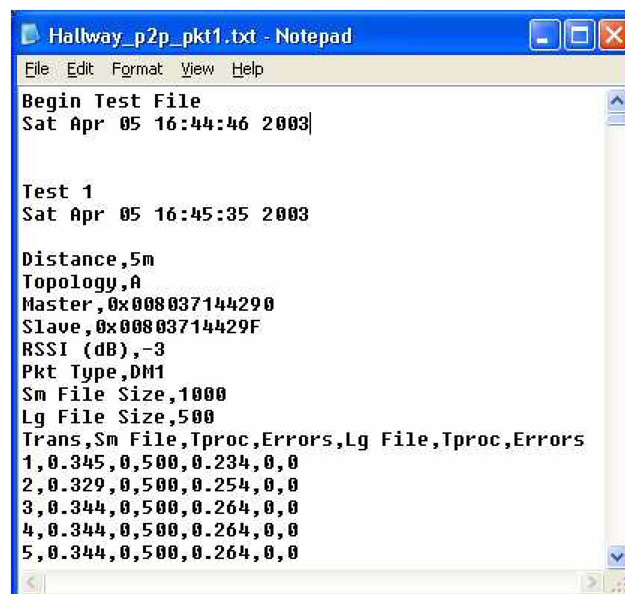

Figure 3.3: Example of a test results output file

3.1 Program Operation

Prior to the transmission of any data a Bluetooth link must be established. In order to begin establishing a network the Bluetooth transceiver must first be initialized. The initialization process consists of a series of commands which must be sent from the host to the host controller and summarized in table 3.1. These commands are issued to the host controller in the order shown at the start of the program.

HCI Command	Description
Reset	Resets the Bluetooth module.
Read Buffer Size	Informs the host of the size of the data buffer in the Bluetooth module.
Host Buffer Size	Informs the Bluetooth module of the size of the host's data buffer.
Write Scan Enable	Enable the Bluetooth module to scan for inquiries and connection requests from other Bluetooth devices.
Write Authentication Enable	Informs the Bluetooth module if link authentication will be used or not.
Set Event Filter	Used to filter unwanted events and automate the acceptance of a connection request.
Write Connection Accept Timeout	Sets the amount of time the Bluetooth module is allowed to wait for the remote device to accept a connection request event before automatically rejecting the request.
Write Page Timeout	Sets the amount of time the Bluetooth module is allowed to wait for a remote device to respond to a connection request.

Table 3.1: Bluetooth Initialization Commands

Once the transceiver is initialized, an inquiry is executed. The purpose of an inquiry is to determine if there are any Bluetooth -capable devices within range of the inquiring device. If a Bluetooth device is found within range, information about that device is passed from the host controller to the host. The user can then decide to create a connection with the discovered device. As stated earlier, after the connection is established it is possible to transmit data from one device to another. The connection process is summarized in figure 3.4.

Transmission test timing is accomplished by starting a timer just before transmission of a test file begins and stopping the timer after the last byte of the file has been received by the

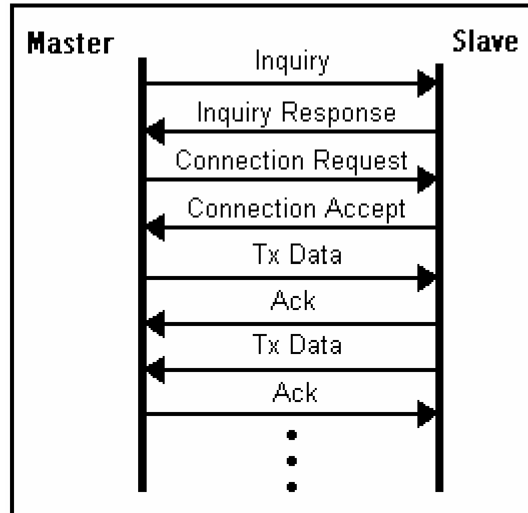

Figure 3.4: Bluetooth connection setup timing diagram

host that originally sent the file. During data transmission there are several factors which must be considered.

- What type of transmission is being requested? Small file? Large file? Or retransmission of a received file? (*e.g.*, A slave only re-transmits received data.)
- How many ACL data packets will be needed to transmit the file?
- Will the host controller's data buffer overflow?

A detailed flow chart of the test data transmission routine is shown in figure 3.5.

The Bluetooth test program receives data by checking the host computer's COMM port buffer at an interval of one millisecond. If data are present the data are then inspected and a decision is made on how to respond or process the received data. In the case that test data are received, the receive routine enters a loop which continues reading the COMM port until all test data are received. The reason for entering the loop is to reduce timing errors caused by the inherent delay in the polling scheme. When the entire data file has been received a timer is started to measure the computation time used by the remote computer preparing the data for retransmission to the originating computer. After the data have been

prepared for retransmission, the timer is stopped and the processing time is transmitted to the originating computer along with the test data file. Upon receipt of the entire test data file by the originating computer, the transmission timer is stopped and the results written to the output file along with the processing time used by the remote computer. After storing the information this whole process is repeated for a number of iterations defined by the user.

The test program has certain limitations worthy of note.

1. The test file sizes are limited to 6.7 kbytes by the buffers available on the Bluetooth transceiver. This problem could be overcome. However, there are bugs in this part of the code which have yet to be solved. A buffer overflow has been occurring in the host controller when files larger than 6.7 kbytes is used. HCI flow control has been implemented; however, the details have yet to be worked out.

2. The maximum possible data rate of the UART is 115 kbps. Bluetooth, however, allows six different data rates, five of which are greater than 115 kbps. This means that transmission tests using the faster Bluetooth data rates appear to transmit at the UART rate instead of the expected Bluetooth rate.

During testing of the program a few transmission errors seemed to escape detection by the error detection and correction methods implemented in the Bluetooth hardware. As these errors occurred, the program was thrown into an infinite loop, effectively crashing the program. Upon further inspection we realized that the error was a missing byte in the received data. Since a whole byte was missing from the data file, the program hung up waiting for a byte that would never be received. This problem was solved by adding a timeout.

The timeout prevents the program from staying in an infinite loop caused by a missing byte in a received file. When this occurs the program then throws out all of the data that had been received and makes a note of the error in the output file. For a program to eliminate this

process of discarding the data it would need to implement an algorithm capable of searching for the missing byte and adapting to this error.

The three-node test program works in much the same way. However, instead of a round-trip transmission being between just two computers, the data file must be transmitted four times to complete a round-trip. The first transmission is from one slave to the master, next from the master to the second slave, then back to the master, and finally back to the original slave. The transmit and receive routines are similar to those of the point-to-point program. One change is that in the three-node system the master decides to which slave to transmit the file based on which slave it is being received from. Secondly the processing times used by the master and second slave are summed, resulting in the total processing time used during the complete round-trip transmission.

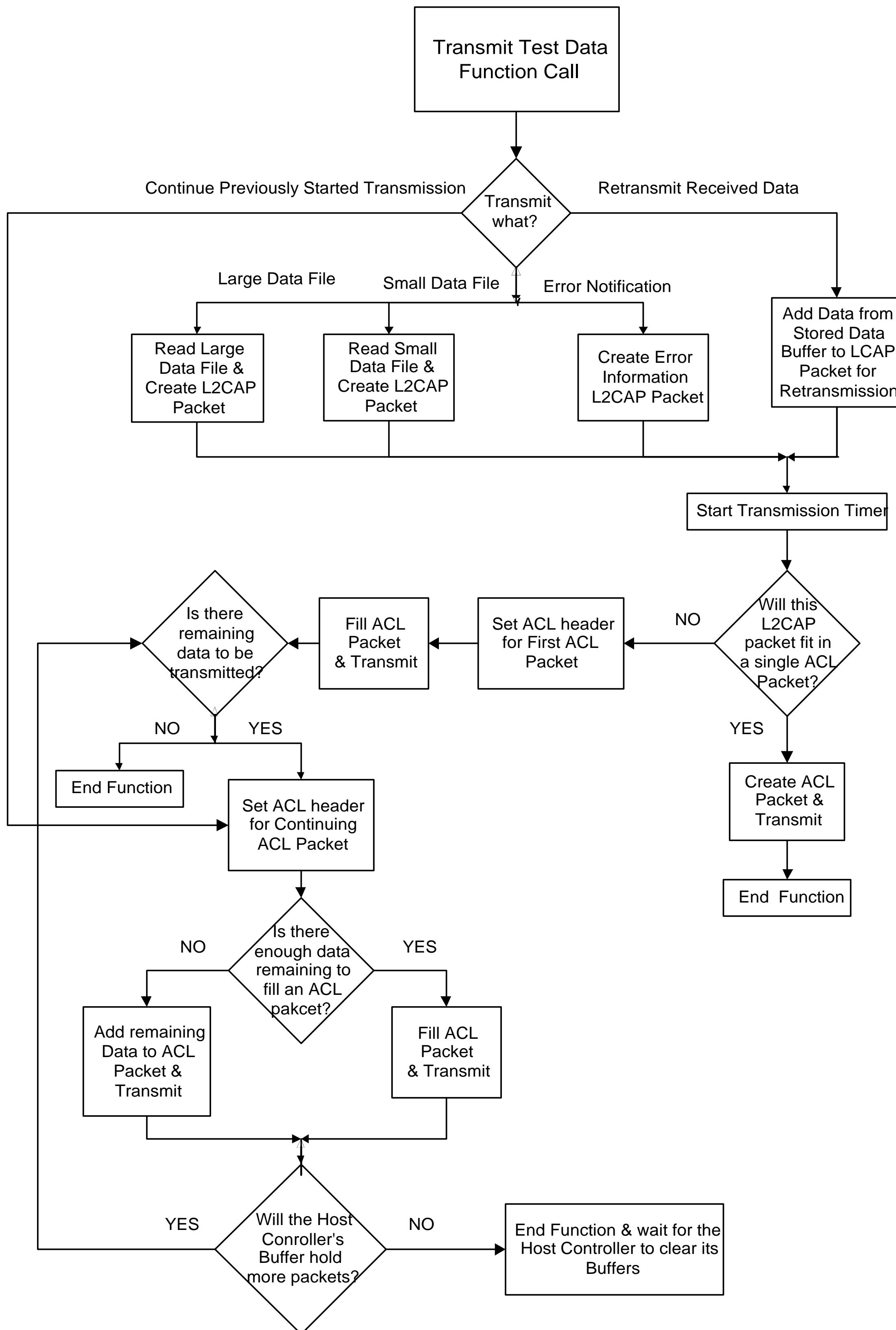

Figure 3.5: Flow Chart of the Test-data Transmission Routine

Chapter 4

Transmission Tests

We ran two sets of tests for this project. The first set of tests was run in the basement hallway of Michelson Hall at the United States Naval Academy. The purpose of this set of tests was to provide a baseline for evaluating the results of the second set of tests completed on board the *ex-USS America*. The data throughput for shipboard tests was expected to drop due to packet retransmissions caused by a decrease in signal to noise ratio, which in fact did occur.

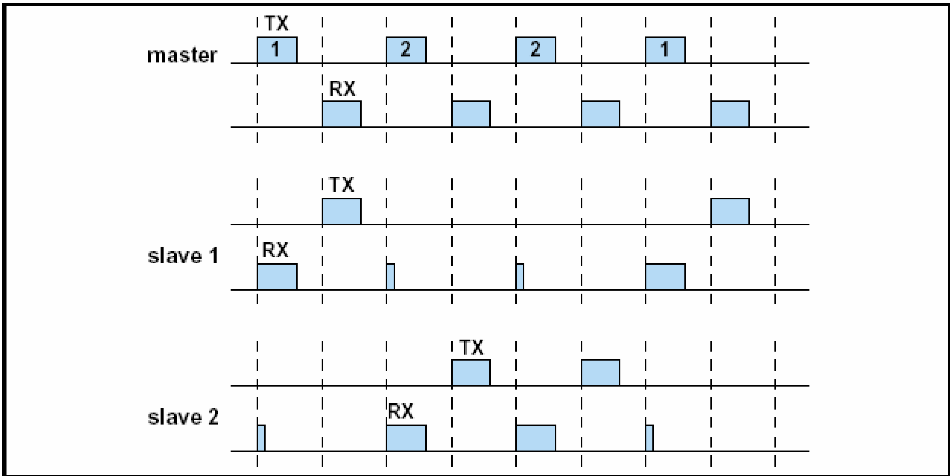

Figure 4.1: Bluetooth packet acknowledgement process [1]

As a way of ensuring reliable packet delivery Bluetooth implements an *automatic repeat request* (ARQ) packet retransmission scheme. Bluetooth ARQ works by having the recipient of a packet reply with an acknowledgment indicating the success or failure of the previous data transmission. The acknowledgement is transmitted in the recipient's next transmission time slot immediately following the data transmission. Figure 4.1 shows this process. An ACK (positive acknowledgement) is transmitted in the case of a successful transmission and a NACK (negative acknowledgment) is transmitted in the case of a packet arriving with errors. The default value for this acknowledgment is NACK so that if a packet is not received, the recipient's transmission time slot following the transmitter's unsuccessful attempt will contain a NACK. When a NACK is received by the originator, the last packet transmitted is retransmitted. This sequence will continue for a vendor specified number of attempts until the transmitting device finally times out and drops the packet.

4.1 Hallway Tests

Figure 4.2: Michelson hallway tests

4.1.1 Test Setup

The first set of transmission tests were completed in the hallway of the basement of Michelson just outside room ML1. One laptop set up outside ML1 on a lab cart (shown in the left half of figure 4.2) and another laptop was set on a chair in the middle of ML1 (shown

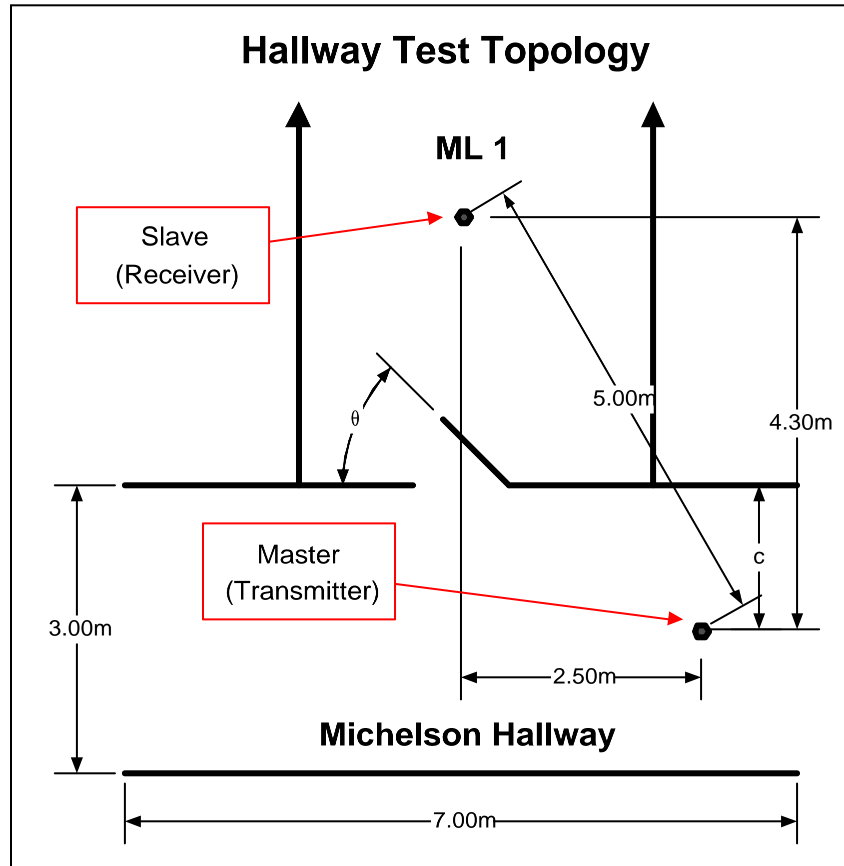

Figure 4.3: Hallway test topology

in the right half of figure 4.2). In order to reduce interference from metal objects near the Bluetooth transceivers they were attached to wooden dowels using velcro, with the dowel rods being placed in a retort stand. The topology was arranged such that the computers were 5 m apart and the computer in the hallway was offset from the door to ML1 (figure 4.3). The topology was changed for each test by changing the angle that the door was open (angle θ in figure 4.3). Tests were run with the door in five different positions: wide open (90°), 45° open, 15° open, cracked open (approx. 1°), and closed (0°).

Hallway testing was completed using four different file sizes: 500 bytes, 1 kbytes, 2.5 kbytes, and 5 kbytes. Each file was transmitted ten times to complete a test. Tests were completed using all six of the available Bluetooth baseband packets in each of the five topologies, resulting in a total of 1200 file transmissions.

4.1.2 Test Results

The results from the hallway tests revealed that the position of the door had essentially no effect on transmission rates and therefore no effect on the error rates within the Bluetooth transmissions. As is shown in figure 4.4 the data throughput rate remains fairly constant regardless of the position of the door. This was the case for all tests.

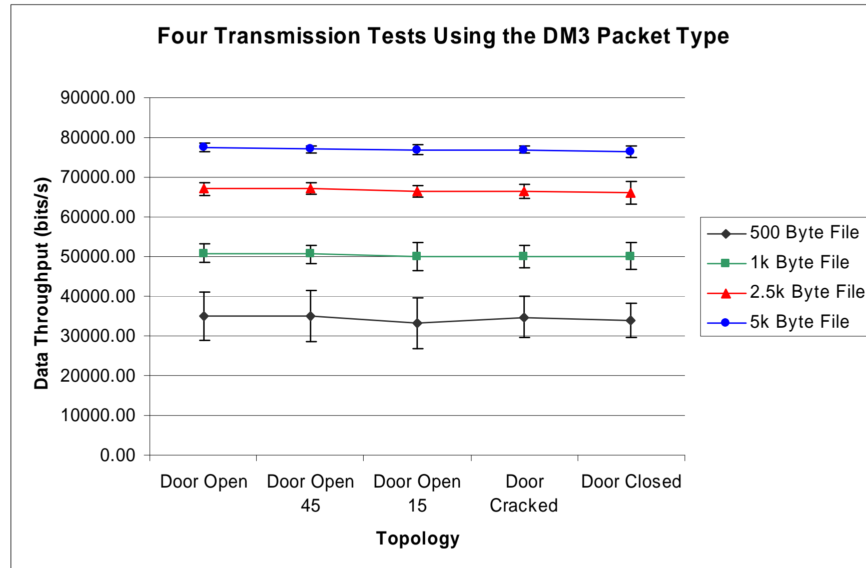

Figure 4.4: Data Throughput vs. Door Position for DM3 Packet Type. (Error bars are placed at ± two standard deviations providing a certainty of 95%)

These results are as expected because the hallway imposes minimal attenuation and few, if any, packet retransmissions are expected. Also of note is the fact that the walls in Michelson are sheet-rock which is transparent to RF energy, therefore the door position essentially had no effect on the Bluetooth transmissions. The graphs also seem to show that larger file sizes will have a higher transmission rate. However, after further analysis this conclusion is proved to be false.

The apparently higher transmission rates of the larger packets is in fact due to some overhead time introduced into the test times by the test program itself. This is shown by graphing data throughput vs. file size (figure 4.5). By performing a linear regression on this data we discovered that there is an offset in the y-direction indicating a small portion

of overhead common to all four file sizes. This offset also implies that it would take a finite amount of time to transmit a zero-byte file, which we know cannot be true, since in practice we simply would not do it.

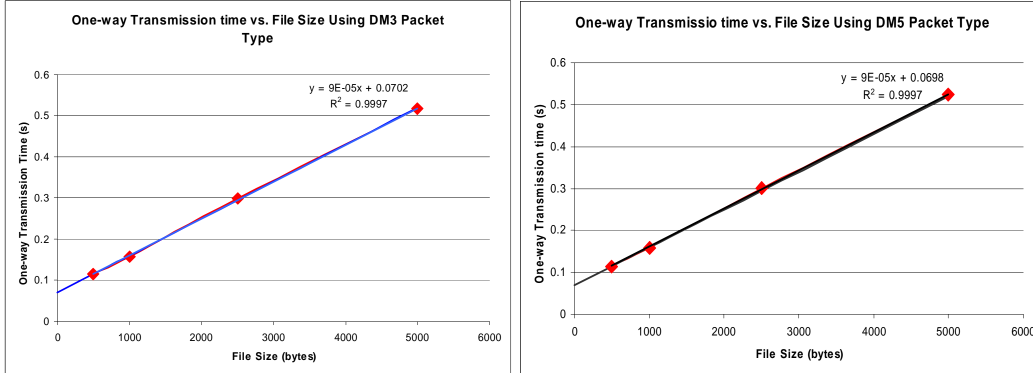

Figure 4.5: One-way Transmission Time vs. File Size

Also, calculations prove that the overhead time revealed by the graph is not the overhead one would expect from packet headers and UART start and stop bits. We will use the 500 byte file transmitted in using DM3 baseband packets as an example.

The graph in figure 4.5 shows a 70.2 ms overhead. Given a 114.6 ms one-way total transmission time we can compute the apparent overhead to be 61.3%.

Now we will compare this value to the expected overhead. First the file will receive a four-byte L2CAP header. Next, the 500 byte file will be split into two ACL packets based on a maximum buffer size of 400 bytes. Each ACL data packet has a five byte header. The total header size is now 14 bytes. The UART will add a start and a stop bit to each byte that is transmitted, therefore the total number of overhead bytes is given by:

$$n_{\mathrm{OH}} = 2(n_{\mathrm{data}} + n_{\mathrm{header}})/8 + n_{\mathrm{header}} \tag{4.1}$$

where n_{data} is the number of bytes of data and n_{header} is the number of bytes of header information. Applying equation (4.1) to the 500 byte file gives $n_{\mathrm{OH}} = 128.5$ bytes. This

gives us 20.4% expected overhead (%OH= $n_{\mathrm{OH}}/(n_{\mathrm{data}} + n_{\mathrm{OH}})$).

When this value is compared with the value obtained from the graph in figure 4.5 we can see that they are not even close. The final proof that the overhead value obtained from the graph represents program overhead is that when this value is subtracted from the total transmission time and the transmission bit rate is calculated using this corrected transmission time we find that the calculated bit rate is approximately equal to the bit rate of the UART. (The Bluetooth bit rate will not be obtained because the UART is slower, therefore it is the limiting factor for transmission time.)

$$R = \frac{8(n_{\mathrm{OH}} + n_{\mathrm{data}})}{t_{\mathrm{trans}}}, \text{ where } R \text{ is the bit rate.} \tag{4.2}$$

Applying equation (4.2) to the example and using the corrected transmission time as t_{trans} we get $R = 113.2$kbps. This value is within 1.5% error of the actual UART rate (155.0 kbps). Therefore the overhead time revealed by the graph in figure 4.5 is test program overhead. The source of this overhead is most likely the timeout values that are part of reading and writing data to the computer's serial data port.

When the program overhead is taken into consideration and the graph from figure 4.4 is regenerated (figure 4.6) it can be seen that file size does not affect data throughput rates. The apparent deviations of the 500 byte file may be due to the resolution of the timer in the test program. The resolution of the timer is one millisecond so any inaccuracy in timing will have a much greater effect on results for the smaller file sizes.

Once the existence of the test program overhead was established, several values for $t_{\mathrm{program\ OH}}$ were computed using linear regressions and a mean value of $t_{\mathrm{program\ OH}} = 70.0$ms was computed. Knowing this we calculated standard values for the expected transmission time of overhead bytes ($t_{\mathrm{OH\ bytes}}$), the expected transmission time of data bytes (t_{data}), and the expected total transmission time (t_{trans}) for each file size. These standard values are

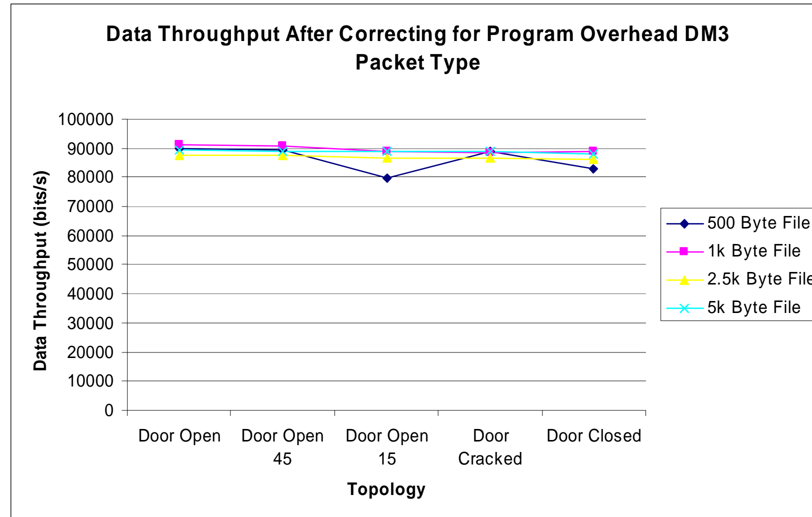

Figure 4.6: Data Throughput vs. Topology corrected, showing that throughput is largely independent of file size.

One Way Tran Times (s)

	500 Byte	1k Byte	2.5k Byte	5k Byte
DH1	0.1140	0.1626	0.3002	0.5201
DM3	0.1146	0.1575	0.2957	0.5164
DH3	0.1132	0.1595	0.3006	0.5227
DM5	0.1148	0.1584	0.3021	0.5231
DH5	0.1095	0.1579	0.3002	0.5228
Average	0.1132	0.1592	0.2997	0.5210
std	0.0022	0.0020	0.0024	0.0028
%Error*	3.81	2.56	1.61	1.09
$t_{OH\ bytes}$	0.0096	0.0192	0.0488	0.0951
t_{data}	0.0336	0.0700	0.1810	0.3559
t_{trans}	0.1132	0.1592	0.2997	0.5210

* Error based on two standard deviations (95% certainty)

Figure 4.7: Standard transmission times

shown in figure 4.7. The values were then used as a reference for determining the percent overhead due to packet retransmissions in the shipboard tests, assuming that the hallway tests experienced few or no packet retransmissions. This assumption is reasonable given that the calculated bit rate was within 1.5% of the actual bit rate for these tests.

Knowing the overhead associated with the transmission test program and the header bytes of each file size it is possible to calculate the number of retransmitted baseband packets. First

Type	Payload Header (bytes)	User Payload (bytes)	FEC	CRC	Symmetric Max. Rate (kb/s)	Asymmetric Max. Rate (kb/s)	
						Forward	Reverse
DM1	1	0-17	2/3	yes	108.8	108.8	108.8
DH1	1	0-27	no	yes	172.8	172.8	172.8
DM3	2	0-121	2/3	yes	258.1	387.2	54.4
DH3	2	0-183	no	yes	390.4	585.6	86.4
DM5	2	0-224	2/3	yes	286.7	477.8	36.3
DH5	2	0-339	no	yes	433.9	723.2	57.6
AUX1	1	0-29	no	no	185.6	185.6	185.6

Figure 4.8: Properties of Baseband Packets [1]

we must determine the number of packets a given file would be expected to occupy ($p_{\text{theoretical}}$) using equation (4.3).

$$p_{\text{theoretical}} = \left\lceil \frac{n_{\text{file}}}{n_{\text{payload}}} \right\rceil \tag{4.3}$$

where n_{file} equals the number of bytes in the file to be transmitted and n_{payload} equals the maximum number of payload bytes of the baseband packet type being used (figure 4.8). Then using the data from figure 4.7 we can compute the actual number of packets used to transmit the file (p_{actual}).

$$p_{\text{actual}} = \left\lceil \frac{t_{\text{trans}} - t_{\text{program OH}} - t_{\text{OH bytes}}}{2t_{\text{time slot}}} \right\rceil \tag{4.4}$$

Equation (4.4), however, is limited to use with only DM1 and DH1 baseband packet types. The number of packets retransmitted is then the difference between $p_{\text{theoretical}}$ and p_{actual}. The results of these calculations are shown in table 4.1.

File Size	$p_{\text{theoretical}}$	p_{actual}	$p_{\text{retransmissions}}$
500	30	40	10
1000	59	73	14
2500	148	182	34
5000	295	355	60

Table 4.1: Packet Retransmissions for the Hallway Test (DM1 Baseband Packet)

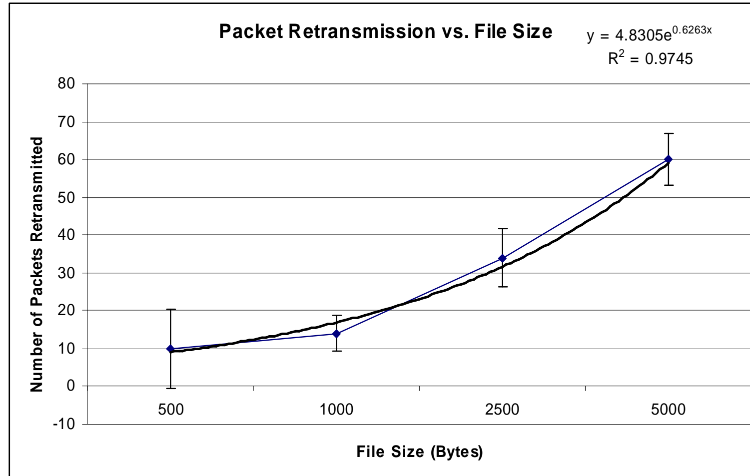

Figure 4.9: File Size vs. Packet Retransmissions for Hallway Tests (Error bars are set at 95% certainty)

It can be seen from this data that relatively few packet retransmissions were required in Michelson Hall. It is also interesting to note that the number of retransmitted packets seems to increase exponentially as the size of the transmitted file increases (figure 4.9). This figure shows an exponential regression (dark line) plotted on top of actual test data (light line). The equation for the regression function is shown in the upper right corner of the plot along with the R^2 value. The R^2 value represents how well the regression fit the actual data points. The closer R^2 is to 1 the better regression fits the actual data.

Hallway tests were also performed using a piconet topology as shown in figure 4.10. The only difference between this topology and the topology shown in figure 4.3 is the addition of a third node to the left of the door. In this case the data was transmitted from the slave-transmitter through the master to the slave-receiver and finally retransmitted back to the slave-transmitter through the master. These tests provided similar results as the point-to-point tests. Piconet testing, however, was not carried out onboard the *ex-USS America* due to bugs in the test program at the time the shipboard tests were being carried out.

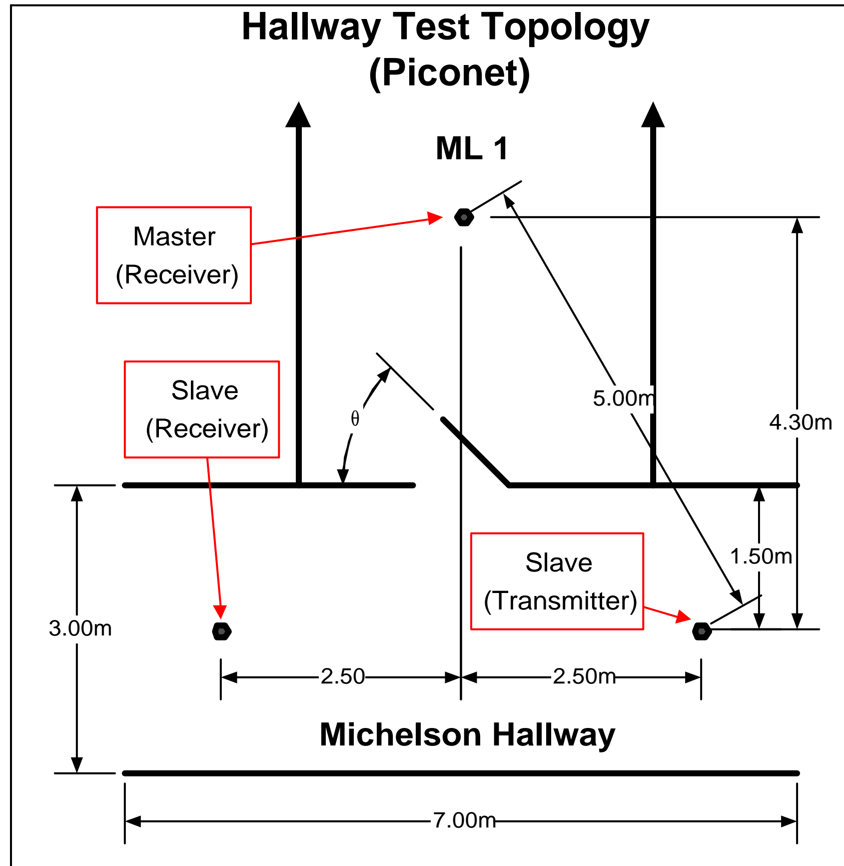

Figure 4.10: Hallway Piconet Test Topology

Figure 4.11: *ex-USS America* Tests

4.2 Shipboard Tests

The second set of tests was completed onboard the *ex-USS America*, a decommissioned aircraft carrier docked at the naval shipyard in Philadelphia. Tests were conducted in an

open aircraft hangar bay onboard the ship. The Philadelphia Naval Shipyard is near the Philadelphia International Airport, and it may well be that the radar from the airport interfered with some of the tests. During testing, the receive-signal-strength indicator in the hangar bay was very erratic. At times it showed receive-signal-strengths greater than the transmission power. Directly behind the area of the hangar bay in which we were testing was an open aircraft elevator in direct view of the airport radar. Therefore, we concluded the receive-signal-strength indicator was of no practical use in these tests.

4.2.1 Test Setup

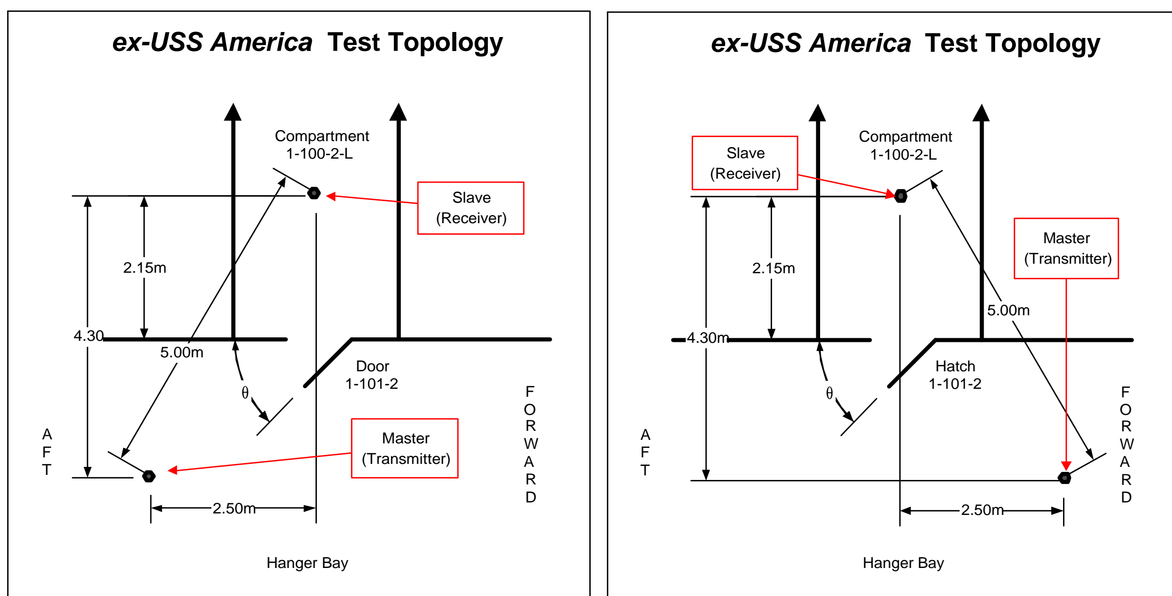

Figure 4.12: Shipboard Topology

Shipboard testing was conducted much like the hallway testing. Tests were run from the hangar bay through door 1-101-2 (left side of figure 4.11 and into compartment 1-100-2-L (right side of figure 4.11. One laptop was placed inside of the compartment and the other was placed in the hangar bay offset from the door. Tests were conducted with the laptop offset on both the right and left side of the door. The total distance between the transceivers was 5 m and the topology was changed by varying the angle of the door: wide open (90°),

45° open, 15° open, cracked open (approx. 1°), closed (0°), and door dogged (0°, A dogged door is closed tightly using levers.).

As with the hallway tests the Bluetooth transceivers were attached to wooden dowel rods and placed in retort stands to reduce interference from nearby metal objects. Also, as with the hallway tests, 10 transmissions of each file were completed for each test.

4.2.2 Test Results

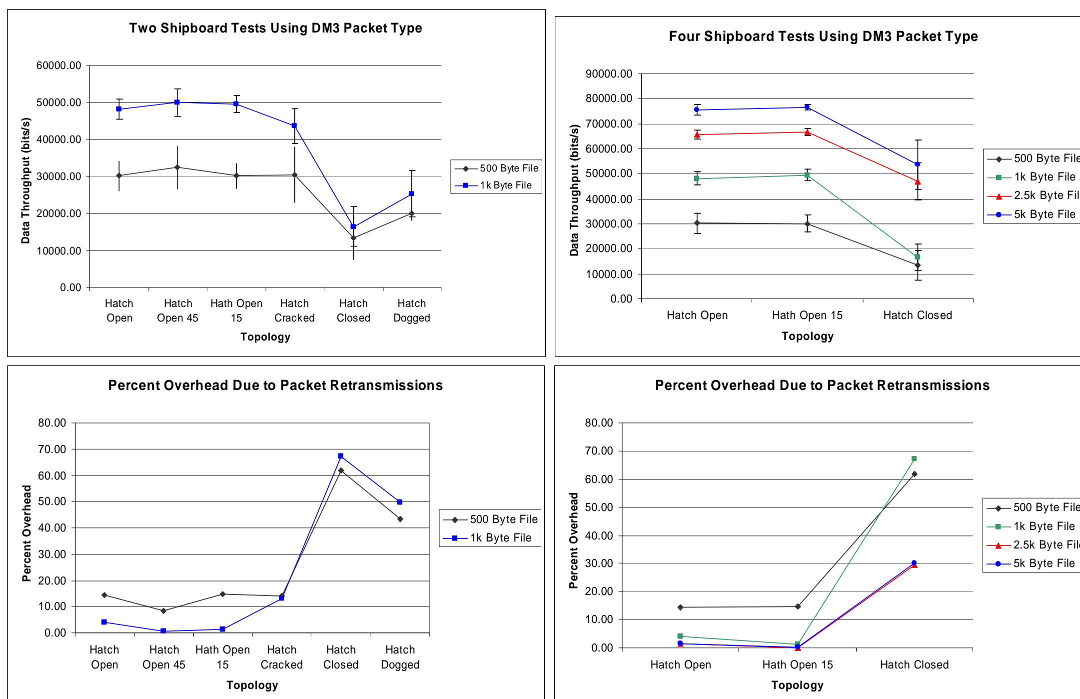

Figure 4.13: Test Results for DM3 Packet Type. (Error bars are placed at ± two standard deviations providing a certainty of 95%)

The shipboard tests prove that wireless data transmission through a closed and dogged door is possible, though a significant decrease in data throughput was experienced. A range of 30–70% overhead due to packet retransmissions was observed. As seen in figure 4.13, the overhead due to packet retransmissions was substantial but not completely overwhelming. Data were still successfully transmitted through the closed and dogged doors.

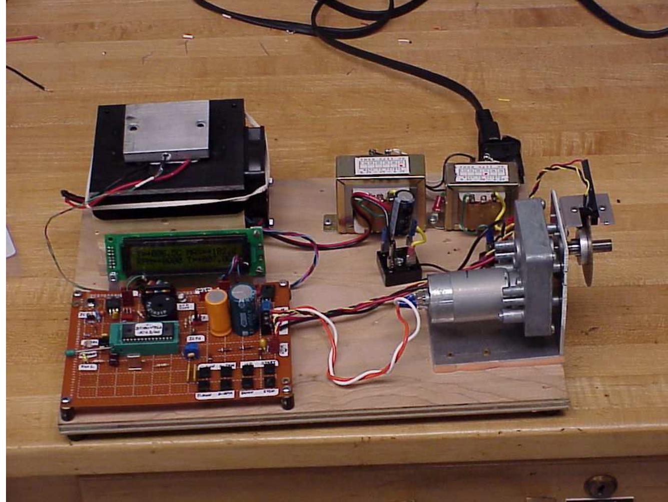

Figure 4.14: ENS Estes' Powernode

It is of note that the intended use for such a network is monitoring and control systems and not file transfers. These applications do not require much data to be transferred. The smallest file size used in tests was 500 bytes. Monitoring and control data do not usually require nearly that much data to be transferred. For example ENS Estes designed and built a *power node* (figure 4.14, a device used to simulate shipboard monitoring and controls). This power node requires only one byte to transfer control information and less than twenty bytes to transfer monitoring information. Therefore, the data throughput rates achieved in the shipboard tests would be acceptable for such a system.

File Size	$p_{\textbf{theoretical}}$	$p_{\textbf{actual}}$	$p_{\textbf{retransmissions}}$
500	30	152	122
1000	59	308	249
2500	148	401	253
5000	295	809	514

Table 4.2: Packet Retransmissions for the Hallway Test (DM1 Baseband Packet)

Packet retransmission data was calculated for the a test run using DM1 baseband packets and the closed door topology (table 4.2). As was the case with the hallway tests, the number of packet retransmissions seems to increase exponentially as file size is increased (figure 4.15). The plot on the right was obtained by excluding the 1000 byte file, which had a great deal

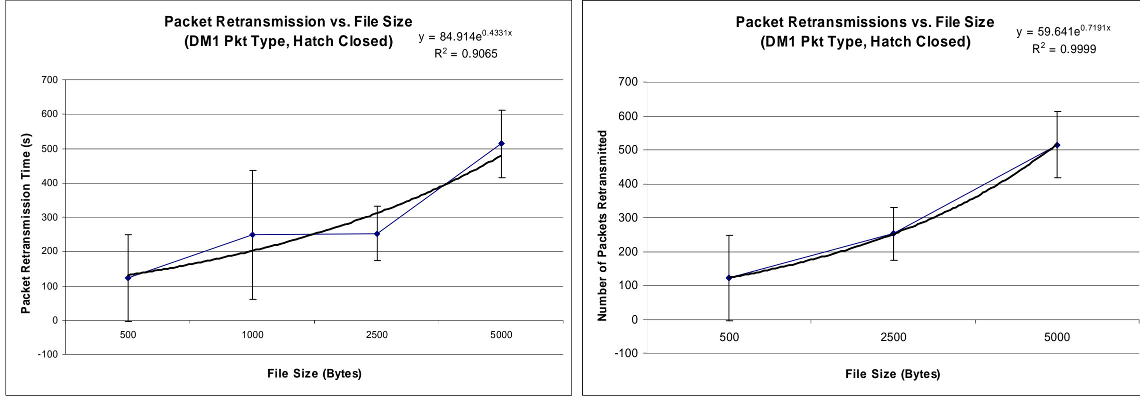

Figure 4.15: File Size vs. Packet Retransmissions for Shipboard Tests (Error bars are set at 95% certainty)

of error associated with it. By excluding the data from this file the exponential increase in packet retransmissions can be clearly seen. This data suggests that the probability of packet retransmission in a given environment is similar to the arrival process in queuing theory[8]. The arrival process is modeled as a Poisson process where the arrival of data is an independent, random process. Similarly in this case packet retransmission is an independent and random process and the time for a given transmission is proportional to the size of the file being transmitted. The probability distribution for such a process is given by

$$P_{\text{arrival}}(x) = \lambda e^{-\lambda x} \tag{4.5}$$

where λ is the average arrival rate. The corresponding cumulative distribution is given by[9]

$$F_{\text{X}}(x) = 1 - e^{-\lambda x} \tag{4.6}$$

We may then relate the expression found by the exponential regression of our test data ($f(x)$) in figure 4.15 to the cumulative distribution function for packet retransmissions ($F_{\text{X}}(x)$) by the equation

$$f(x) = \frac{1}{1 - F_{\text{X}}(x)} \tag{4.7}$$

Solving equation (4.7) for $F_X(x)$ gives

$$F_X(x) = 1 - \frac{1}{f(x)} \tag{4.8}$$

This then gives us the desired form for the cumulative distribution function

$$F_X(x) = 1 - \frac{1}{59.641} e^{-0.7191x} \tag{4.9}$$

However, in practical use this expression must be refined by further testing. In order to refine this expression the number of times a single baseband packet is retransmitted must be measured. Our testing was only able to measure the total number of baseband packet retransmissions in the transmission of an entire file. A more practical version of this expression would be to determine the cumulative distribution function for the number of times a packet is dropped due to the vendor specified timeout cause by excessive packet retransmission.

4.3 Test Statistics

Test	**500 Byte File**	**1k Byte File**	**2.5k Byte File**	**5k Byte File**
Hallway Tests	0/300	0/300	1/297	0/296
Shipboard Tests	0/422	3/422*	0/157	1/157

* Two of the three errors encountered here may have been due in part to partially discharged batteries in the laptop.

Table 4.3: Transmission Failures per Attempted Transmissions

Several of the tests completed onboard the *ex-USS America* and one hallway test resulted in the test program timing out. The cause of these timeouts is insufficient data transferred (i.e. a byte or several bytes missing from the transferred data). These are the only errors experienced which were not corrected by the Bluetooth error-detection-and-correction schemes. Based on the number of test transmissions attempted the number of failed transmission is relatively low (0.71% for the 1k byte file and 0.21% overall).

Test	Failed Connections
All Hallway Tests	0 failed of 60 attempts
All Shipboard Tests	13 failed of 73 attempts
Shipboard Tests w/ Door Open or Cracked	0 failed of 48 attempts
Shipboard Tests w/ Door Closed or Dogged	13 failed of 25 attempts

Table 4.4: Number of Failed Connections per Connection Attempts

Several times during the shipboard testing the paging process timed out while attempting to establish a connection between the two Bluetooth transceivers. The difficulty creating connections occurred only in the topologies with the door closed or dogged. This suggests that though data transfer through the door in the closed and dogged positions is possible, the signal-to-noise ratio may be at or near its limit in these two topologies. A connection success rate of 82.2% overall and only 48.0% with the door in the closed and dogged positions is not very encouraging, given that the network must be capable of operating in the extreme conditions experienced at battle stations, during which all watertight doors will be dogged. We must also keep in mind, however, that during the connection establishment process the master and slave are not using the same frequency hopping sequence. Therefore, the difficulties in connection establishment do not indicate problems with data transfer. After a connection is established both slave and master are using synchronized frequency hopping sequences and are always on the same transmission channel.

One possible solution to this problem is the use of a higher power Bluetooth tranceiver. Our tests were conducted using the low-power (1mW) Bluetooth transceiver. Presumably the higher power (100mW) Bluetooth transceiver will be able to overcome this obstacle.

Another possible solution to the connection problem is software that will continually attempt to establish a connection. The test program we created was not written to do this.

Chapter 5

Conclusions

The goal of this research project was to determine the feasibility of a Bluetooth wireless network in a shipboard environment. Based on the results obtained from the shipboard tests, Bluetooth wireless transmission is feasible through closed and dogged doors. However, a decrease in data throughput (30–70% packet retransmission overhead) was experienced. Taking into consideration the fact that the proposed uses for a Bluetooth network (shipboard montitoring and control systems) typically require very little data transfer the decrease in data throughput should not preclude the success of such a network. Problems may arise, however, due to the rate of failed connections when the door is in the closed or dogged position. This problem itself would be enough to disqualify Bluetooth from use onboard naval vessels. There does remain the possiblity that this problem could be solved by using a higher power Bluetooth transceiver and modifying the software which establishes a connection. Since higher power devices have been unavailable to us thus far, more tests will need to be conducted to demonstrate this.

Chapter 6

Future Research

The possibility and usefulness of a shipboard Bluetooth network still remains. A Bluetooth network would provide ease of use and an inexpensive solution to the implementation of a wireless shipboard monitoring and control network. This project answered the question of the possibility of data transfer through a closed door, but also gave rise to the question of reliability in establishing a connection through closed and dogged doors. Further research should be conducted using newer versions of Bluetooth hardware. A higher power transceiver will be available soon and newer versions of Bluetooth will implement adaptive frequency hopping schemes. [3] Both of these advances will have a favorable impact on the problem of connection establishment. It is also of note that Bluetooth is still in infant stages of development. The Bluetooth project was only begun by Ericsson in 1994, the Special Interest Group was not created until 2001,[2] and Bluetooth devices have only recently become available to the consumer. The possibility of a Bluetooth shipboard monitoring and control network should be pursued as the Bluetooth technology itself develops.

Bibliography

[1] Bluetooth SIG, *Specification of the Bluetooth System*, Vol. 1, 2001.

[2] Brent A. Miller, Chatschik Bisdikian, Ph.D., *Bluetooth Revealed: The Insider's Guide to an Open Specification for Global Wireless Communications*, Upper Saddle River, NJ: Prentice Hall, PTR, 2002.

[3] Robert Morrow, *Bluetooth Operation and Use*, New York: McGraw-Hill, 2002.

[4] Daniel R.J. Estes, ENS, *A Trident Scholar Project Report: Assessment of Radio Frequency Propagation in a Naval Shipboard Environment*, United States Naval Academy, Annapolis, MD, 2001.

[5] Ericsson, *ROK 101 008 Bluetooth PtP Module: Specification Sheet*, Kista-Stockholm, Sweden, 2000.

[6] Ferrel G. Stremler, *Introduction to Communication Systems*, Third Edition, New York: Addison–Wesley Publishing Company, 1992.

[7] Jorgen Ostlund, Infineon Technologies North America Corporation Sales Applications Specialist, Email Correspondence, 21 November 2002.

[8] Alberto Leon-Garcia, Indra Widjaja, *Communication Networks Fundamental Concepts and Key Architectures*, Boston: McGraw-Hill, 2000.

[9] Alberto Leon-Garcia, *Probability and Random Processes for Electrical Engineering*, New York: Addison-Wesley, 1989.

Appendix A

Bluetooth Include Functions

A.1 bluetooth.h

```
//bluetooth.h
//Written on 19OCT02 by MIDN Kenneth J. Hoover
//Last updated 02APR03

//This header file defines the Host Controller Interface (HCI)
//commands, events, error codes, and UART packet headers for
//bluetooth devices.

//---------------------------------------------------------------------------

#ifndef bluetoothH
#define bluetoothH

/*********************** COMMANDS *****************************/

//-------LINK CONTROL COMMANDS OGF 0x01-------

//Inquiry Commands
#define HCI_INQUIRY 0x0401
#define HCI_INQUIRY_CANCEL 0x0402
#define HCI_PERIODIC_INQUIRY_MODE 0x0403
#define HCI_EXIT_PERIODIC_INQUIRY_MODE 0x0404
//Connect and Disconnect Commands
#define HCI_CREATE_CONNECTION 0x0405
#define HCI_DISCONNECT 0x0406
#define HCI_ADD_SCO_CONNECTION 0x0407
#define HCI_ACCEPT_CONNECTION_REQUEST 0x0409
#define HCI_REJECT_CONNECTION_REQUEST 0x040A
#define HCI_CHANGE_CONNECTION_PACKET_TYPE 0x040F  //command not in order
//Security-related Commands
#define HCI_LINK_KEY_REQUEST_REPLY 0x040B
#define HCI_LINK_KEY_REQUEST_NEGATIVE_REPLY 0x040C
#define HCI_PIN_CODE_REQUEST_REPLY 0x040D
#define HCI_PIN_CODE_REQUEST_NEGATIVE_REPLY 0x040E
#define HCI_AUTHENTICATION_REQUEST 0x0411
#define HCI_SET_CONNECTION_ENCRYPTION 0x0413
#define HCI_CHANGE_CONNECTION_LINK_KEY 0x0415
#define HCI_MASTER_LINK_KEY 0x0417
//Remote Device Information Commands
#define HCI_REMOTE_NAME_REQUEST 0x0419
```

```
#define HCI_READ_REMOTE_SUPPORTED_FEATURES 0x041B
#define HCI_READ_REMOTE_VERSION_INFORMATION 0x041D
#define HCI_READ_CLOCK_OFFSET 0x041F


//-------LINK POLICY COMMANDS OGF 0x02-------

//Low-power Mode Commands
#define HCI_HOLD_MODE 0x0801
#define HCI_SNIFF_MODE 0x0803
#define HCI_EXIT_SNIFF_MODE 0x0804
#define HCI_PARK_MODE 0x0805
#define HCI_EXIT_PARK_MODE 0x0806
//Setup and Configuration Commands
#define HCI_QoS_SETUP 0x0807
#define HCI_ROLE_DISCOVERY 0x0809
#define HCI_SWITCH_ROLE 0x080B
#define HCI_READ_LINK_POLICY_SETTINGS 0x080C
#define HCI_WRITE_LINK_POLICY_SETTINGS 0x080D


//-------HOST CONTROLLER & BASBAND COMMANDS OGF 0x03-------

//Setup and Configuration Commands
#define HCI_SET_EVENT_MASK 0x0C01
#define HCI_RESET 0x0C03
#define HCI_SET_EVENT_FILTER 0x0C05
#define HCI_CHANGE_LOCAL_NAME 0x0C13 //command not in order
#define HCI_READ_LOCAL_NAME 0x0C14 //command not in order
#define HCI_READ_CLASS_OF_DEVICE 0x0C23 //command not in order
#define HCI_WRITE_CLASS_OF_DEVICE 0x0C24 //command not in order
#define HCI_READ_VOICE_SETTING 0x0C25 //command not in order
#define HCI_WRITE_VOICE_SETTING 0x0C26 //command not in order
#define HCI_READ_NUM_BROADCAST_RETRANSMISSIONS 0x0C29 //command not in order
#define HCI_WRITE_NUM_BROADCAST_RETRANSMISSIONS 0x0C2A //command not in order
//Security-related Commands
#define HCI_READ_PIN_TYPE 0x0C09
#define HCI_WRITE_PIN_TYPE 0x0C0A
#define HCI_CREATE_NEW_UNIT_KEY 0x0C0B
#define HCI_READ_STORED_LINK_KEY 0x0C0D
#define HCI_WRITE_STORED_LINK_KEY 0x0C11
#define HCI_DELETE_STORED_LINK_KEY 0x0C12
#define HCI_READ_AUTHENTICATION_ENABLE 0x0C1F //command not in order
#define HCI_WRITE_AUTHENTICATION_ENABLE 0x0C20 //command not in order
#define HCI_READ_ENCRYPTION_MODE 0x0C21 //command not in order
#define HCI_WRITE_ENCRYPTION_MODE 0x0C22 //command not in order
//Timeout Commands
#define HCI_READ_CONNECTION_ACCEPT_TIMEOUT 0x0C15
#define HCI_WRITE_CONNECTION_ACCEPT_TIMEOUT 0x0C16
#define HCI_READ_PAGE_TIMEOUT 0x0C17
#define HCI_WRITE_PAGE_TIMEOUT 0x0C18
#define HCI_READ_AUTOMATIC_FLUSH_TIMEOUT 0x0C27 //command not in order
#define HCI_WRTIE_AUTOMATIC_FLUSH_TIMEOUT 0x0C28 //command not in order
#define HCI_READ_LINK_SUPERVISION_TIMEOUT 0x0C36 //command not in order
#define HCI_WRITE_LINK_SUPERVISION_TIMEOUT 0x0C37 //command not in order
//Inquiry and Page Scan Commands
#define HCI_READ_SCAN_ENABLE 0x0C19
#define HCI_WRITE_SCAN_ENABLE 0x0C1A
#define HCI_READ_PAGE_SCAN_ACTIVITY 0x0C1B
```

```
#define HCI_WRITE_PAGE_SCAN_ACTIVITY 0x0C1C
#define HCI_READ_INQUIRY_SCAN_ACTIVITY 0x0C1D
#define HCI_WRITE_INQUIRY_SCAN_ACTIVITY 0x0C1E
#define HCI_READ_NUM_SUPPORTED_IAC 0x0C38 //command not in order
#define HCI_READ_CURRENT_IAC_LAP 0x0C39 //command not in order
#define HCI_WRITE_CURRENT_IAC_LAP 0x0C3A //command not in order
#define HCI_READ_PAGE_SCAN_PERIOD_MODE 0x0C3B //command not in order
#define HCI_WRITE_PAGE_SCAN_PERIOD_MODE 0x0C3C //command not in order
#define HCI_READ_PAGE_SCAN_MODE 0x0C3D //command not in order
#define HCI_WRITE_PAGE_SCAN_MODE 0x0C3E //command not in order
//Low-power Mode Commands
#define HCI_READ_HOLD_MODE_ACTIVITY 0x0C2B
#define HCI_WRITE_HOLD_MODE_ACTIVITY 0x0C2C
//Transmit Power Commands
#define HCI_READ_TRANSMIT_POWER_LEVEL 0x0C2D
//Flow Control Commands
#define HCI_FLUSH 0x0C08 //command not in order
#define HCI_READ_SCO_FLOW_CONTROL_ENABLE 0x0C2E
#define HCI_WRITE_SCO_FLOW_CONTROL_ENABLE 0x0C2F
#define HCI_SET_HOST_CONTROLLER_TO_HOST_FLOW_CONTROL 0x0C31
#define HCI_HOST_BUFFER_SIZE 0x0C33
#define HCI_HOST_NUM_COMPLETED_PACKETS 0x0C35


//-------INFORMATIONAL PARAMETERS OGF 0x04-------
#define HCI_READ_LOCAL_VERSION_INFORMATION 0x1001
#define HCI_READ_LOCAL_SUPPORTED_FEATURES 0x1003
#define HCI_READ_BUFFER_SIZE 0x1005
#define HCI_READ_COUNTRY_CODE 0x1007
#define HCI_READ_BD_ADDR 0x1009


//-------STATUS PARAMETERS OGF 0x05-------
#define HCI_READ_FAILED_CONTACT_COUNTER 0x1401
#define HCI_RESET_FAILED_CONTACT_COUNTER 0x1402
#define HCI_GET_LINK_QUALITY 0x1403
#define HCI_READ_RSSI 0x1405


//-------TESTING COMMANDS OGF 0x06-------
#define HCI_READ_LOOPBACK_MODE 0x1801
#define HCI_WRITE_LOOPBACK_MODE 0x1802
#define HCI_ENABLE_DEVICE_UNDER_TEST_MODE 0x1803

//-------ERICSSON SPECIFIC HCI COMMANDS (ROK 101 008)-------
#define HCI_SET_UART_BAUD_RATE 0xFC09
//This command has one parameter one byte long
//The default value is 57.6kbps
//9600bps -> 10100
//BAUD RATES
#define BT_BR_9600 0x14
#define BT_BR_14400 0x05
#define BT_BR_57600 0x03


/*********************** EVENTS ****************************/

#define EV_INQUIRY_COMPLETE 0x01
#define EV_INQUIRY_RESULT 0x02
```

```
#define EV_CONNECTION_COMPLETE 0x03
#define EV_CONNECTION_REQUEST 0x04
#define EV_DISCONNECTION_COMPLETE 0x05
#define EV_AUTHENTICATION_COMPLETE 0x06
#define EV_REMOTE_NAME_REQUEST_COMPLETE 0x07
#define EV_ENCRYPTION_CHANGE 0x08
#define EV_CHANGE_CONNECTION_LINK_KEY_COMPLETE 0x09
#define EV_MASTER_LINK_KEY_COMPLETE 0x0A
#define EV_READ_REMOTE_SUPPORTED_FEATURES_COMPLETE 0x0B
#define EV_READ_REMOTE_VERSION_INFORMATION_COMPLETE 0x0C
#define EV_QoS_SETUP_COMPLETE 0x0D
#define EV_COMMAND_COMPLETE 0x0E
#define EV_COMMAND_STATUS 0x0F
#define EV_HARDWARE_ERROR 0x10
#define EV_FLUSH_OCCURRED 0x11
#define EV_ROLE_CHANGE 0x12
#define EV_NUM_COMPLETED_PACKETS 0x13
#define EV_MODE_CHANGE 0x14
#define EV_RETURN_LINK_KEYS 0x15
#define EV_PIN_CODE_REQUEST 0x16
#define EV_LINK_KEY_REQUEST 0x17
#define EV_LINK_KEY_NOTIFICATION 0x18
#define EV_LOOPBACK_COMMAND 0x19
#define EV_DATA_BUFFER_OVERFLOW 0x1A
#define EV_MAX_SLOTS_CHANGE 0x1B
#define EV_READ_CLOCK_OFFSET_COMPLETE 0x1C
#define EV_CONNECTION_PAKCKET_TYPE_CHANGED 0x1D
#define EV_QoS_VIOLATION 0x1E
#define EV_PAGE_SCAN_MODE_CHANGE 0x1F
#define EV_PAGE_SCAN_REPETITION_MODE_CHANGE 0x20


/*********************** ERROR CODES ****************************/

#define ERR_UNKNOWN_HCI_COMMAND 0x01
#define ERR_NO_CONNECTION 0x02
#define ERR_HARDWARE_FAILURE 0x03
#define ERR_PAGE_TIMEOUT 0x04
#define ERR_AUTHENTICATION_FAILURE 0x05
#define ERR_KEY_MISSING 0x06
#define ERR_MEMORY_FULL 0x07
#define ERR_CONNECTION_TIMEOUT 0x08
#define ERR_MAX_NUM_CONNECTIONS 0x09
#define ERR_MAX_NUM_SCO_CONNECTIONS_TO_A_DEVICE 0x0A
#define ERR_ACL_CONNECTION_ALREADY_EXISTS 0x0B
#define ERR_COMMAND_DISALLOWED 0x0C
#define ERR_HOST_REJECTED_DUE_TO_LIMITED_RESOURCES 0x0D
#define ERR_HOST_REJECTED_DUE_TO_SECURITY_REASONS 0x0E
#define ERR_HOST_REJECTED_DUE_TO_REMOTE_DEVICE_IS_ONLY_PERSONAL_DEVICE 0x0F
#define ERR_HOST_TIMEOUT 0x10
#define ERR_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE 0x11
#define ERR_INVALID_HCI_COMMAND_PARAMETERS 0x12
#define ERR_OTHER_END_TERMINATED_CONNECTION_USER_ENDED 0x13
#define ERR_OTHER_END_TERMINATED_CONNECTION_LOW_RESOURCES 0x14
#define ERR_OTHER_END_TERMINATED_CONNECTION_POWER_OFF 0x15
#define ERR_CONNECTION_TERMINATED_BY_LOCAL_HOST 0x16
#define ERR_REPEATED_ATTEMPTS 0x17
#define ERR_PAIRING_NOT_ALLOWED 0x18
#define ERR_UNKNOWN_LMP_PDU 0x19
```

```
#define ERR_UNSUPPORTED_REMOTE_FEATURE 0x1A
#define ERR_SCO_OFFSET_REJECTED 0x1B
#define ERR_SCO_INTERVAL_REJECTED 0x1C
#define ERR_SCO_AIR_MODE_REJECTED 0x1D
#define ERR_INVALID_LMP_PARAMETERS 0x1E
#define ERR_UNSPECIFIED_ERROR 0x1F
#define ERR_UNSUPPORTED_LMP_PARAMETER_VALUE 0x20
#define ERR_ROLE_CHANGE_NOT_ALLOWED 0x21
#define ERR_LMP_RESPONSE_TIMEOUT 0x22
#define ERR_LMP_ERROR_TRANSACTION_COLLISION 0x23
#define ERR_LMP_PDU_NOT_ALLOWED 0x24
#define ERR_ENCRYPTION_MODE_NOT_ACCEPTABLE 0x25
#define ERR_UNIT_KEY_USED 0x26
#define ERR_QoS_NOT_SUPPORTED 0x27
#define ERR_INSTANT_PASSED 0x28
#define ERR_PAIRING_WITH_UNIT_KEY_NOT_SUPPORTED 0x29


/*********************** UART HEADERS *************************/

#define UART_HCI_COMMAND_PACKET 0x01
#define UART_ACL_DATA_PACKET 0x02
#define UART_SCO_DATA_PACKET 0x03
#define UART_EVENT_PACKET 0x04

/************************ PARAMETERS ********************************/

//------------ PACKET TYPES ------------

#define DM1 0x0008
#define DH1 0x0010
#define DM3 0x0400
#define DH3 0x0800
#define DM5 0x4000
#define DH5 0x8000

//------------ LINK TYPES ------------

#define SCO_LINK 0x00
#define ACL_LINK 0x01

//------------ GIAC LAP ------------
//(Generic Inquiry Access Code Lower Address Part) from BT Spec

#define GIAC_LAP  0x9E8B33

//------------ DATA PACKET BOUNDARY/P2P FLAGS -------------
//These flags representa a combination of a P2P BC Flag and
//either a First Packet PB Flag or a continuing packet PB Flag.

#define FIRST_PKT_P2P 0x20
#define CONT_PKT_P2P 0x10

//------------ DATA PACKET BROADCAST FLAGS ------------

#define P2P 0x00
#define ACTIVE_BC 0x40
#define PICONET_BC 0x80
```

```
//------------ EVENT FILTER TYPES ------------

#define CLR_ALL_FILTERS 0x00
#define INQUIRY_RESULT 0x01
#define CONNECTION_SETUP 0x02

//------------ AUTO ACCEPT FLAG VALUES ------------
//Only needed for CONNECTION SETUP FILTERS

//Auto Accept off
#define AA_OFF 0x01
//Auto Accept on
#define AA_ON_RS_DISABLED 0x02 //Role switch disabled
#define AA_ON_RS_ENABLED 0x03 //Role switch enabled

//------------ L2CAP CID (MSB) CODES ---------------
//These L2CAP CID codes are arbitrary codes that I chose to use
//for this project.  They are not defined in the BT Spec

#define MESSAGE_MSB 0x03
#define TEST_DATA_MSB 0xEE

//------------ TRANSMISSION TEST TIMEOUT CODES -------------

#define TIME_OUT_LOCAL_COMP 0xFFEE
#define TIME_OUT_REMOTE_COMP 0xFFFF

//------------ TRANSMISSION TEST FILE SIZE CODES -------------

#define RETRANSMIT -1
#define LARGE_FILE 0
#define SMALL_FILE 1
#define CONTINUE_TRANSMISSION 2

#endif
```

A.2 BT_functions.h

```
//BT_functions.h
//Written on 23OCT02 by MIDN Kenneth J. Hoover
//Last updated 03FEB03

//This header file contains the structures and function prototypes for
//functions used in creating a Bluetooth application or sending and recieving Bluetooth
//HCI commands and events from Bluetooth Host Controller.

//---------------------------------------------------------------------------

#ifndef BT_functionsH
#define BT_functionsH

#include "bluetooth.h"
#include "UART.h"

//---------------------------------------------------------------------------

struct BUFFERS {
int sizeACLBuff;
```

```
char sizeSCOBuff;
int numACLBuff;
int numSCOBuff;
};//end struct

struct BT_DEVICE {
char addrDisp[15];
unsigned char bd_addr[6];
char pgScnRepMode;
char pgScnPerMode;
char pgScnMode;
char deviceClass[2];
char clkOffset[2];
char connectHndl[2];
char linkType;
int pktType;
char encryptMode;
bool master;
bool conductingTest;
bool connection;
short rssi;
//Initialization and default values
BT_DEVICE(char c1='0', char c2='x', char c3=NULL){
addrDisp[0] = c1;
addrDisp[1] = c2;
addrDisp[14] = c3;
master = false;
conductingTest = false;
connection = false;
}
};//end struct


//---------------------------------------------------------------------------

//change2char():
//This function convert a nibble of data to the ASCII equivilent character
//of the hexidecimal digit represented by the nibble of data.
//
//Parameters n: should contain one nibble of data in the least significant
// nibble of a char variable type.
//
//Return: ASCII equivilent character of a hexidecimal digit

char change2char(char n);

//---------------------------------------------------------------------------

//nibbles():
//This function separates a 1 byte char variable into two char's each containing
//one nibble of the data contained in the original char variable.
//
//Parameters x: data to be separated into nibbles.
// msn: most significant nibble.
// lsn: least significant nibble.

void nibbles(char x, char &msn, char &lsn);

//---------------------------------------------------------------------------
```

```
//FormatCommand():
//This function sets the first 3 bytes of a Bluetooth HCI command packet interfaced.
//via UART.  [UART pkt header],[opcode lsb],[opcode msb]
//
//Parameters hciCmd: Some Bluetooth HCI command as defined in "bluetooth.h"
// pktHdr[]: The array in which the command packet will be stored
// until it is transmitted.

void FormatCommand(int hciCmd, char pktHdr[]);

//---------------------------------------------------------------------------

//Int2Char():
//This function converts an integer into four characters by saving each of the
//bytes of the integer in an array of four characters. This is usefull when an
//integer must be transmitted in a Bluetooth command or data packet.
//
//Parameters i: Integer to convert to characters.
// ch[]: 4 byte character array in which to store the 4 bytes
// of the integer. The bytes of the integer are stored in
// the array in little endian format. (i.e. ch[0] = lsb.
// ch[3] = msb)

void Int2Char(int i, char ch[]);

//---------------------------------------------------------------------------

//SetUartBaudRate():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Set_UART_Baud_Rate.  This command changes the baud rate of the UART
//connection between the host and the host controller
//
//**NOTE:
//
//Parameters BR: New UART baud rate.
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD SetUartBaudRate(char BR, char cmdPkt[]);

//---------------------------------------------------------------------------

//ReadBD_ADDR():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Read_BD_ADDR.  This command reads the Bluetooth Device Address of the
//local host contoller.
//
//Parameters cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD ReadBD_ADDR(char cmdPkt[]);

//---------------------------------------------------------------------------
```

```
//ReadBufferSize():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Read_Buffer_Size.  This command reads the buffer size of the
//local host contoller.
//
//Parameters cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD ReadBufferSize(char cmdPkt[]);

//---------------------------------------------------------------------------

//HostBufferSize():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Host_Buffer_Size.  This command sends the host controller the size of
//the host's buffer sizes.
//
//Parameters *pHostBuff: A pointer to a BUFFERS structure containing the size
// of the host's buffes.
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD HostBufferSize(BUFFERS *pHostBuff, char cmdPkt[]);

//---------------------------------------------------------------------------

//SetEventMask():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Set_Event_Mask.  This command allows the host to mask off certain HCI
//events.  If an event is masked the host controller will not create that event.
//
//Parameters evMask: Desired event mask.
// of the host's buffes.
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD SetEventMask(int evMask, char cmdPkt[]);

//---------------------------------------------------------------------------

//WriteConnAcceptTimeout():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Write_Connectin_Accept_Timeout.  This command sets the maximum period of
//time allowed for a remote device to respond to a connection request.
//
//
//Parameters TOInterval: Timeout interval in increments of 625ms. (i.e.
// timeout = TOInterval * 625ms.
// Range (0x0001 - 0xB540)
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.
```

```
DWORD WriteConnAcceptTimeout(int TOInterval, char cmdPkt[]);

//--------------------------------------------------------------------------

//WritePageTimeout():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Write_Page_Timeout.  This command sets the maximum period of
//time allowed for a remote device to respond to a page.
//
//
//Parameters TOInterval: Timeout interval in increments of 625ms. (i.e.
// timeout = TOInterval * 625ms.
// Range (0x0001 - 0xFFFF)
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD WritePageTimeout(int TOInterval, char cmdPkt[]);

//--------------------------------------------------------------------------

//WriteLinkSupTimeout():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Write_Link_Supervision_Timeout.  The link supervision timeout is used
//to determine link loss.  If no baseband packets are transmitted for the
//duration of the timeout the link is disconnected.
//
//
//Parameters connectHndl: Connection handle to an ACL or SCO link.
// TOInterval: Timeout interval in increments of 625ms. (i.e.
// timeout = TOInterval * 625ms.
// Range (0x0001 - 0xFFFF)
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD WriteLinkSupTimeout(int connectHndl, int TOInterval, char cmdPkt[]);

//--------------------------------------------------------------------------

//ReadTxPowLevel():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Read_Transmit_Power_Level.  Read the transmit power level of the local
//Bluetooth Device.
//
//
//Parameters connectHndl: Connection handle to an ACL or SCO link.
// type: 0x00 - Read current transmit power level.
// 0x01 - Read maximum transmit power level.
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD ReadTxPowLevel(char connectHndl[], char type, char cmdPkt[]);
```

```
//------------------------------------------------------------------------

//ReadRSSI():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Read_RSSI.  Read the receive signal strength indicator (RSSI) of the
//local Bluetooth device.
//
//
//Parameters connectHndl: Connection handle to an ACL or SCO link.
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD ReadRSSI(char connectHndl[], char cmdPkt[]);

//------------------------------------------------------------------------

//Reset():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Reset.  Reset the local Bluetooth device.
//
//
//Parameters cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD Reset(char cmdPkt[]);

//------------------------------------------------------------------------

//Inquiry():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Inquiry.  Conduct an inquiry for other Bluetooth devices.
//
//
//Parameters inquiryLength: Maximum amount of time before inquiry is halted
// Range 0x00-0x30. Time = inquiryLength * 1.28s
// maxNumRes: Maximum number of responses to inquiry before the
// inquiry is halted 0x00-unlimited responses
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD Inquiry(char inquiryLength, char maxNumRes, char cmdPkt[]);

//------------------------------------------------------------------------

//CreateConnection():
//This function creates a Bluetooth HCI command packet for the HCI command
//HCI_Create_Connection.  Create a Bluetooth connection.
//
//
//Parameters *device: Pointer to the BT_DEVICE structure containing
// information about the device witch which to create
// a connection.
// pktType: Designates the baseband packet type to be used within
```

```
// the connection (DM1 DH1 DM3 DH3 DM5 or DH5)
// cmdPkt[]: The array in which the command packet will be stored
// until it is transmitted.
//
//Return: Packet size in bytes.

DWORD CreateConnection(BT_DEVICE *device, int pktType, char cmdPkt[]);

//---------------------------------------------------------------------------

DWORD WriteScanEnable(char scanEnable, char cmdPkt[]);

//---------------------------------------------------------------------------

DWORD AcceptConnectRqst(BT_DEVICE *device, char cmdPk[]);

//---------------------------------------------------------------------------

DWORD SendACLPkt(char connectHndl[], DWORD dataLength, char flags, char dataPkt[]);

//---------------------------------------------------------------------------

DWORD WriteVoiceSetting(int voiceSetting, char cmdPkt[]);

//---------------------------------------------------------------------------

DWORD WriteAuthEnable(char enable, char cmdPkt[]);

//---------------------------------------------------------------------------

DWORD SetEventFilter(char filterType, char autoAcceptFlag, char cmdPkt[]);

//---------------------------------------------------------------------------

DWORD Disconnect(char connHndl[2], char reason, char cmdPkt[]);

//---------------------------------------------------------------------------

AnsiString Error(char err);

//---------------------------------------------------------------------------

#endif
```

A.3 BT_functions.cpp

```
//BT_functions.cpp
//Written on 23OCT02 by MIDN Kenneth J. Hoover
//Last updated 15NOV02

//This file contains the function definitions for functions used in
//creating a Bluetooth application or sending and recieving Bluetooth
//HCI commands and events from Bluetooth Host Controller.

//---------------------------------------------------------------------------

#include "BT_functions.h"
```

```
//--------------------------------------------------------------------------

char change2char(char n){
if (n >= 10) //if n is greater than 9 n is the hex digit A-F.
return (n+55); //ascii code 65-70 represents letters A-F.
else
return (n+48); //ascii code 48-57 represents numerals 0-9.
}

//--------------------------------------------------------------------------

void nibbles(char x, char &msn, char &lsn){
msn = x >> 4; //right shift the bits 4 places msn = most sig nibble
msn = msn & 0x0F; //mask off the unused bits
lsn = x & 0x0F;

msn = change2char(msn); //convert each nibble to a the ascii code
lsn = change2char(lsn); //hex digit for display
}

//--------------------------------------------------------------------------

void FormatCommand(int hciCmd, char pktHdr[]){
int temp = 0;
unsigned int mask = 0x000000FF;

pktHdr[0] = UART_HCI_COMMAND_PACKET; //add UART command packet header byte 0

temp = hciCmd & mask; //mask hci_cmd so only lsbyte is present
pktHdr[1] = static_cast<char>(temp); //convert to an 8 bit char

temp = (hciCmd>>8) & mask; //mask and shift hci_cmd so only msbyte is present
pktHdr[2] = static_cast<char>(temp); //convert to an 8 bit char
}

//--------------------------------------------------------------------------

void Int2Char(int i, char ch[]){
int temp = 0;
unsigned int mask = 0x000000FF;

for (int n=0; n<4; n++){ //loop through all 4 bytes of the int variable
temp = (i>>n*8) & mask; //shift int and mask as needed 8*byte number(n) bits
ch[n] = static_cast<char>(temp); //convert to char and return array of up to 4 chars
}//end for
}

//--------------------------------------------------------------------------

DWORD SetUartBaudRate(char BR, char cmdPkt[]){
DWORD pktSize = 5;

FormatCommand(HCI_SET_UART_BAUD_RATE, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x01; //lengeth of parameters 1 byte
cmdPkt[4] = BR; //Baud Rate parameter

return pktSize; //return total packet size (in bytes)
}
```

```
//---------------------------------------------------------------------------

DWORD ReadBD_ADDR(char cmdPkt[]){
DWORD pktSize = 4;

FormatCommand(HCI_READ_BD_ADDR, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x00; //lengeth of parameters 0 bytes

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD ReadBufferSize(char cmdPkt[]){
DWORD pktSize = 4;

FormatCommand(HCI_READ_BUFFER_SIZE, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x00; //lengeth of parameters 0 bytes

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD HostBufferSize(BUFFERS *pHostBuff, char cmdPkt[]){
DWORD pktSize = 11;
char temp[4];

FormatCommand(HCI_HOST_BUFFER_SIZE, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x07; //lengeth of parameters 7 bytes

//Add buffer size data to command packet array
Int2Char(pHostBuff->sizeACLBuff, temp);
cmdPkt[4] = temp[0];
cmdPkt[5] = temp[1];
cmdPkt[6] = pHostBuff->sizeSCOBuff;
Int2Char(pHostBuff->numACLBuff, temp);
cmdPkt[7] = temp[0];
cmdPkt[8] = temp[1];
Int2Char(pHostBuff->numSCOBuff, temp);
cmdPkt[9] = temp[0];
cmdPkt[10] = temp[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD SetEventMask(int evMask, char cmdPkt[]){
DWORD pktSize = 12;
char temp[4];

FormatCommand(HCI_SET_EVENT_MASK, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x08; //lengeth of parameters 8 bytes
Int2Char(evMask, temp);
for (int n=0; n<4; n++) //set remaining 4 bytes of event mask
cmdPkt[n+8] = temp[n];
        for (int n=0; n<4; n++) //the most sig 4 bytes of the event mask are reserved
cmdPkt[n+4] = 0x00; //so set those bytes to 0
```

```
return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WriteConnAcceptTimeout(int TOInterval, char cmdPkt[]){
DWORD pktSize = 6;
char temp[4];

FormatCommand(HCI_WRITE_CONNECTION_ACCEPT_TIMEOUT, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x02; //lengeth of parameters 2 bytes
Int2Char(TOInterval, temp);
cmdPkt[4] = temp[0]; //add timout interval parameters to cmd pkt array
cmdPkt[5] = temp[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WritePageTimeout(int TOInterval, char cmdPkt[]){
DWORD pktSize = 6;
char temp[4];

FormatCommand(HCI_WRITE_PAGE_TIMEOUT, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x02; //lengeth of parameters 2 bytes
Int2Char(TOInterval, temp);
cmdPkt[4] = temp[0]; //add timout interval parameters to cmd pkt array
cmdPkt[5] = temp[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WriteLinkSupTimeout(BT_DEVICE *pDevice, int TOInterval, char cmdPkt[]){
DWORD pktSize = 8;
char temp[4];

FormatCommand(HCI_WRITE_LINK_SUPERVISION_TIMEOUT, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x04; //lengeth of parameters 4 bytes

//Add connection handle and timout interval to cmd pkt array
cmdPkt[4] = pDevice->connectHndl[0];
cmdPkt[5] = pDevice->connectHndl[1];
Int2Char(TOInterval, temp);
cmdPkt[6] = temp[0];
cmdPkt[7] = temp[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD ReadTxPowLevel(char connectHndl[], char type, char cmdPkt[]){
DWORD pktSize = 7;

FormatCommand(HCI_READ_TRANSMIT_POWER_LEVEL, cmdPkt); //sets the first 3 bytes of the cmd pkt
```

```
cmdPkt[3] = 0x03; //lengeth of parameters 3 bytes

//Add parameter to cmd pkt array
cmdPkt[4] = connectHndl[0];
cmdPkt[5] = connectHndl[1];
cmdPkt[6] = type;

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD ReadRSSI(char connectHndl[], char cmdPkt[]){
DWORD pktSize = 6;

FormatCommand(HCI_READ_RSSI, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x02; //lengeth of parameters 2 bytes

//Add parameters to cmd pkt array
cmdPkt[4] = connectHndl[0];
cmdPkt[5] = connectHndl[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD Reset(char cmdPkt[]){
DWORD pktSize = 4;

FormatCommand(HCI_RESET, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x00; //lengeth of parameters 0 bytes

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD Inquiry(char inquiryLength, char maxNumRes, char cmdPkt[]){
DWORD pktSize = 9;

FormatCommand(HCI_INQUIRY, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x05; //lengeth of parameters 5 bytes

//LAP for GIAC is 0x9E8B33 so set that as LAP
cmdPkt[4] = 0x33;
cmdPkt[5] = 0x8B;
cmdPkt[6] = 0x9E;

//Add parameters to cmd pkt array
cmdPkt[7] = inquiryLength; //Durration of inquiry (inquiryLength*1.28s)
//inquiryLength ranges from 0x01 to 0x30 (0-61.44s)
cmdPkt[8] = maxNumRes; //maxNumRes ranges from 0x01 to 0xFF.

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------
```

```
DWORD CreateConnection(BT_DEVICE *pDevice, int pktType, char cmdPkt[]){
DWORD pktSize = 17;
char temp[4];

FormatCommand(HCI_CREATE_CONNECTION, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x0D;  //lengeth of parameters 13 bytes

//add BD_ADDR to cmd pkt
for (int n = 0; n<6; n++)
cmdPkt[n+4] = pDevice->bd_addr[n];

//add pkt type to cmd pkt
Int2Char(pktType, temp);
cmdPkt[10] = temp[0];
cmdPkt[11] = temp[1];

cmdPkt[12] = pDevice->pgScnRepMode;
cmdPkt[13] = pDevice->pgScnMode;

//add clock offeset to cmd pkt
cmdPkt[14] = pDevice->clkOffset[0];
cmdPkt[15] = pDevice->clkOffset[1];

cmdPkt[16] = 0x00; //do not allow role switch

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WriteScanEnable(char scanEnable, char cmdPkt[]){
DWORD pktSize = 5;

FormatCommand(HCI_WRITE_SCAN_ENABLE, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x01; //lengeth of parameters 0 bytes

cmdPkt[4] = scanEnable; //0x00 no scans; 0x01 inquiry scn only; 0x02 pg scn only;
//0x03 all scans enabled.

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD AcceptConnectRqst(BT_DEVICE *pDevice, char cmdPkt[]){
DWORD pktSize = 11;

FormatCommand(HCI_ACCEPT_CONNECTION_REQUEST, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x07; //lengeth of parameters 7 bytes

for (int n = 0; n<6; n++)
cmdPkt[n+4] = pDevice->bd_addr[n];
cmdPkt[10] = 0x01; //do not perform a master-slave switch

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------
```

```
DWORD SendACLPkt(char connectHndl[], DWORD dataLength, char flags, char dataPkt[]){
DWORD pktSize;
char tempChar, tempAry[4];
const int headerSize = 5;
const char cnctHndlMsk = 0x0F;

pktSize = headerSize + dataLength;

//Shift data back 5 spaces in the array to make room for the header
for (unsigned int n=1; n<=dataLength; n++){
tempChar = dataPkt[dataLength - n]; //Move all array elements beginning with the last element
dataPkt[dataLength + (headerSize - n)] = tempChar; //back 5 spaces.
}//end for

//Adding ACL header to data pkt
dataPkt[0] = UART_ACL_DATA_PACKET; //UART header
dataPkt[1] = connectHndl[0]; //Connection Handle is 12 bits + 2 bits for each
dataPkt[2] = (connectHndl[1] & cnctHndlMsk) + flags;//pkt boundary flag and broadcast flag


Int2Char(dataLength, tempAry); //Convert int into 2 separate characters
dataPkt[3] = tempAry[0]; //for data transmission
dataPkt[4] = tempAry[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WriteVoiceSetting(int voiceSetting, char cmdPkt[]){
DWORD pktSize = 6;
char temp[4];

FormatCommand(HCI_WRITE_VOICE_SETTING, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x02;  //lengeth of parameters 2 bytes

//Add parameters to cmd pkt array
Int2Char(voiceSetting, temp);
cmdPkt[4] = temp[0];
cmdPkt[5] = temp[1];

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD WriteAuthEnable(char enable, char cmdPkt[]){
DWORD pktSize = 5;

FormatCommand(HCI_WRITE_AUTHENTICATION_ENABLE, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x01; //lengeth of parameters 1 byte

//Add parameters to cmd pkt array
cmdPkt[4] = enable;

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------
```

```
DWORD SetEventFilter(char filterType, char autoAcceptFlag, char cmdPkt[]){
DWORD pktSize;

FormatCommand(HCI_SET_EVENT_FILTER, cmdPkt); //sets the first 3 bytes of the cmd pkt

//What filter is being set?
switch (filterType){

case CLR_ALL_FILTERS:
//Only filterType parameter needed.
pktSize = 5;
cmdPkt[3] = 0x01;  //lengeth of parameters 1 byte

//Add parameters to cmd pkt array
cmdPkt[4] = filterType;
break;

case INQUIRY_RESULT:
//2 parameters required: filterType and filter condition type
pktSize = 6;
cmdPkt[3] = 0x02;  //length of parameters 2 bytes

//Add parameters to cmd pkt array
cmdPkt[4] = filterType;
cmdPkt[5] = 0x00;  //Always set Inquiry filter condtion type to 0x00
   //(A new device responded to the Inquiry process.)
break;

case CONNECTION_SETUP:
//3 parameters required: filterType, filter condition type, and
//auto accept flag.
pktSize = 7;
cmdPkt[3] = 0x03; //length of parameters 3 bytes

//Add parameters to cmd pkt array
cmdPkt[4] = filterType;
cmdPkt[5] = 0x00;  //Always set Inquiry filter condtion type to 0x00
   //(Allow Connections from all devices.)
cmdPkt[6] = autoAcceptFlag;
break;
}//end switch

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

DWORD Disconnect(char connHndl[2], char reason, char cmdPkt[]){
DWORD pktSize = 7;

FormatCommand(HCI_DISCONNECT, cmdPkt); //sets the first 3 bytes of the cmd pkt
cmdPkt[3] = 0x03;  //lengeth of parameters 3 byte

//Add parameters to cmd pkt array
cmdPkt[4] = connHndl[0]; //Connection handle of
cmdPkt[5] = connHndl[1]; //the connection to disconnect

//Reason for disconnect 0x13 = user ended connection 0x15 = about to power off
```

```
cmdPkt[6] = reason;

return pktSize; //return total packet size (in bytes)
}

//---------------------------------------------------------------------------

AnsiString Error(char err){

//Search for the error code and return the
//corresponding error message to be displayed
switch(err){
case ERR_UNKNOWN_HCI_COMMAND:
return "ERROR: Unknown HCI Command";

case ERR_NO_CONNECTION:
return "ERROR: No Connection";

case ERR_HARDWARE_FAILURE:
return "ERROR: Hardware Failure";

case ERR_PAGE_TIMEOUT:
return "Page Timeout";

case ERR_MEMORY_FULL:
return "ERROR: Memory Full";

case ERR_ACL_CONNECTION_ALREADY_EXISTS:
return "ERROR: ACL Connection Already Exists";

case ERR_COMMAND_DISALLOWED:
return "ERROR: Command Disallowed";

case ERR_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE:
return "ERROR: Unsupported Feature or Parameter Value";

case ERR_INVALID_HCI_COMMAND_PARAMETERS:
return "ERROR: Invalid HCI Command Parameters";

default:
return "Unknown Error";

}//end swtich
}

//---------------------------------------------------------------------------
```

Appendix B

BTTest v. 1.1

B.1 BTTest1_1.h

```
//BTTest1_1.h
//Written on 19OCT02 by MIDN Kenneth J. Hoover
//Last updated 02APR03
//
//This header file defines forms and variables and contains function
//prototyes for the BTTest program.

//---------------------------------------------------------------------------

#ifndef BTCommH
#define BTCommH

//---------------------------------------------------------------------------

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "BT_functions.h"
#include "UART.h"
#include <ExtCtrls.hpp>
#include <Menus.hpp>
#include "transTest.h"

DWORD BaudRate[] = {CBR_9600, CBR_14400, CBR_57600, CBR_115200};
const DWORD testDataBuffSize = 100100, inBuffSize = 100100, outBuffSize = 600;//595 bytes usable (5 byte header for
COMMTIMEOUTS CT1, CTinit;
BT_DEVICE slave[7], localDevice;
BUFFERS host, hostCtrl;
unsigned long pktSize;
bool readRssi = true, transmitInProgress = false;
DCB myDCB, initDCB;

//DATA VARIABLES
char inBuff[inBuffSize], outBuff[outBuffSize], testDataBuff[testDataBuffSize];
TransTest cTest;
unsigned int numTestTrans;

//---------------------------------------------------------------------------
class TBT_net : public TForm{
```

```
__published: // IDE-managed Components
        TTimer *Timer1;

        TGroupBox *GroupBox1;
        TGroupBox *GroupBox2;
        TGroupBox *GroupBox3;

//STATIC LABELS
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
TLabel *Label9;
TLabel *Label13;
TLabel *Label6;

//DYNAMIC LABELS
        TLabel *LBD_ADDR_rd; //BT_ADDR for remote device
        TLabel *LBD_ADDR_ld; //BT_ADDR for local device
TLabel *LLinkType;
        TLabel *LConnectHndl;
        TLabel *LInqRes;
        TLabel *LRSSI;
        TLabel *LRx;
        TLabel *LTestStatus;
TLabel *LStatus1;
        TLabel *LStatus2;

//INPUT BOX
TLabeledEdit *Message;

//BUTTONS
        TButton *BBeginTest;
        TButton *BSend;
        TButton *BReset;
        TButton *BConnect;
TButton *BInquiry;

//COMBOBOXES
        TComboBox *CBDist;
        TComboBox *CBTopology;
        TComboBox *CBControl;
        TComboBox *CBTestRole;
        TComboBox *CBPktType;
        TComboBox *CBNumTrans;
        TLabel *Label10;
        TLabel *Label11;
        TLabel *LCurTrans;
        TLabel *LPktNum;
        TLabel *LStatus3;

        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);

//TIMER INTERUPT
        void __fastcall Timer1Timer(TObject *Sender);
```

```
//BUTTON CLICKS
        void __fastcall BInquiryClick(TObject *Sender);
        void __fastcall BBeginTestClick(TObject *Sender);
        void __fastcall BSendClick(TObject *Sender);
        void __fastcall BResetClick(TObject *Sender);
        void __fastcall BConnectClick(TObject *Sender);

//COMBOBOXES CLOSE
        void __fastcall CBControlCloseUp(TObject *Sender);
        void __fastcall CBPktTypeCloseUp(TObject *Sender);
        void __fastcall CBTestRoleCloseUp(TObject *Sender);
        void __fastcall CBNumTransCloseUp(TObject *Sender);

private: // User declarations
public: // User declarations



__fastcall TBT_net(TComponent* Owner);
};

//---------------------------------------------------------------------------

//***** FUNCTIONS *****

//FindOpCode:
//This function looks up the OpCode returned in a Command Complete or
//Command Status event packet and processes the data contained within
//the event packet accordingly.
//
//Parameters *pBuff: Pointer to the first byte in the returned
// event packet

void FindOpCode(char *pBuff);

//---------------------------------------------------------------------------

//Find Event:
//This function looks up Event retruned in an event packet and processes the
//data contained within the event packet accordingly
//
//Parameters *pBuff: Pointer to the first byte in the returned
// event packet

void FindEvent(char *pBuff);

//---------------------------------------------------------------------------

//ResetBuffer:
//This function resets an array of characters to NULL
//
//Parameters buff[]: Buffer to reset.
// buffSize: Size of the buffer to reset.

void ResetBuffer(char buff[], DWORD buffSize);

//---------------------------------------------------------------------------

//NumDataPkts:
```

```
//This function is used to determine the number of Bluetooth HCI pakcets
//contained in a string of characters.  It also determines if all of the
//packets are complete or if a packet was segmented during the read.
//
//Parameters buff[]: Buffer in which the data is contained.
// numBytesRead: Number of bytes contained in the buffer
// &numPkts: Address of an integer which contains the
// number of data packets contained in the
// buffer.
// *pLastPkt: Pointer to the first byte of the last packet
// contained in the buffer.
// &dataError: Returns true if data is incomplete
//
//Return: (bool) True if a the buffer contains a segmented packet.

bool NumDataPkts(char buff[], DWORD numBytesRead, DWORD &numPkts, char *pLastPkt, bool &dataError);

//---------------------------------------------------------------------------

//TransmitTestData:
//This function prepares and transmits or retransmits the test data.
//
//Parameters fileSize: SMALL_FILE - Send small file
// LARGE_FILE - Send large file
// RETRANSMIT - Retransmit received data
// CONTINUE_TRANSMISSION - Continue with a
// transmission that is currently in progress
// tDataSize: Size of test data received. (only used when
// retransmitting data)

void TransmitTestData(int fileSize, unsigned int tDataSize);

//---------------------------------------------------------------------------

//ResetDisplay():
//This function resets the labels and controls on the Graphical User Interface.

void ResetDisplay();

//---------------------------------------------------------------------------

//ClearStatus():
//This function resets the Status labels on the Graphical User Interface.

void ClearStatus();

//---------------------------------------------------------------------------

//Disconnecting():
//This function resets the Status labels and other labels related to an ACL
//connection on the Graphical User Interface.

void Disconnecting();

//---------------------------------------------------------------------------


#define CLEAR 0x0000
```

```
extern PACKAGE TBT_net *BT_net;
//---------------------------------------------------------------------------
#endif
```

B.2 BTTest1_1.cpp

```
//BTTest1_1.cpp
//Written on 24OCT02 by MIDN Kenneth J. Hoover
//Last updated 02APR03
//
//This program sets up and creates a Bluetooth wireless link through which
//data is sent via a virtual serial link between two computers, each running
//this program.  It then runs a test program used to test data transmision times
//between a single master and slave using two differnt size text files.  This
//program is written for use with the Ericsson ROK 101 008 Bluetooth baseband
//controller and tranciever.

//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "BTTest1_1.h"

//---------------------------------------------------------------------------

#pragma package(smart_init)
#pragma resource "*.dfm"
TBT_net *BT_net;



//---------------------------------------------------------------------------

__fastcall TBT_net::TBT_net(TComponent* Owner)
        : TForm(Owner)
{
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::FormCreate(TObject *Sender){

        //INITIALIZATION OF THE SERIAL PORT (COMM 1)
        C1 = CreateFile(Port[0], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED, NULL);
        GetCommState(C1, &initDCB); //Save inital DCB in order to reset Comm 1 on exit of program
        myDCB = initDCB; //Initailize myDCB to current Comm Port settings
        myDCB.BaudRate = BaudRate[3];
        myDCB.fBinary = TRUE;
        myDCB.fParity = FALSE; //No parity check
        myDCB.fOutxCtsFlow = TRUE; //should be true ********
        myDCB.fOutxDsrFlow = FALSE;
        myDCB.fDtrControl = DTR_CONTROL_DISABLE;
        myDCB.fDsrSensitivity = FALSE;
        myDCB.fOutX = FALSE;
        myDCB.fInX = FALSE;
        myDCB.fNull = FALSE;
```

```
        myDCB.fRtsControl = RTS_CONTROL_TOGGLE;   //should be toggle *****
        myDCB.ByteSize = 8;
        myDCB.Parity = NOPARITY;
        myDCB.StopBits = ONESTOPBIT;
        SetCommState(C1, &myDCB);

        //COMM TIMEOUTS
        GetCommTimeouts(C1, &CT1);
        GetCommTimeouts(C1, &CTinit);
        CT1 = CTinit; //Initialize CT1 to current comm timeout settings.
        CT1.ReadIntervalTimeout = 10; //Timeout on read if delay is greater than 100ms
//between chars.
        CT1.ReadTotalTimeoutMultiplier = 0; //Timeout on read if total read time is greater
        CT1.ReadTotalTimeoutConstant = 50; //than 200ms
        SetCommTimeouts(C1,&CT1);

        //INITIALIZE BT MODULE
        pktSize = Reset(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//INITIALIZE DATA VARIABLES
ResetBuffer(inBuff, inBuffSize);
host.sizeACLBuff = 400;
host.sizeSCOBuff = 400;
host.numACLBuff = 1;
host.numSCOBuff = 1;

//GUI SETUP
ResetDisplay();

}

//---------------------------------------------------------------------------

void __fastcall TBT_net::FormDestroy(TObject *Sender){

        //RESET COMM STATES BEFORE CLOSING PROGRAM
        SetCommState(C1, &initDCB);
        SetCommTimeouts(C1, &CTinit);
        CloseHandle(&C1);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::Timer1Timer(TObject *Sender){

//****** VARIABLES ******
//-----Buffer Variables -----
    DWORD numBytesRead, numPkts = 0;
char *pPkt = inBuff, *pLastPkt = inBuff;
static char *pReadPt = inBuff;
static unsigned int buffAvail = inBuffSize;

//----- Packet and Data Control Variables
bool segmentedPkt = false;
unsigned int pktSize, mask = 0x000000FF, amtProcData = 0;
static unsigned int segSize = 0;

//----- RSSI Variables
```

```
static unsigned int cycle = 0;
const unsigned int rssiFreq = 12;

//----- Display Variables -----
char dataDisp[255] = {NULL};

//----- Temporary Variables -----
char tempAry[450] = {0};
unsigned int temp1, temp2;

//******** TRANSMISSION TEST VARIABLES *********
static unsigned int testDataReceived = 0, numTestsRan = 0;
static unsigned int l2capPktSize, storedL2capData = 0;
short fileType;
unsigned int n; //loop control variable
char aclFlags, flagMask = 0xF0;
Timer cTimer;
char bytesOfInt[4];
unsigned int processedPkts = 0, curProcPkts = 0, t_proc = 0, errors = 0;
DWORD newBytesRead = 0, loopLimit = 200, loopCount = 0;
bool timeOut = false;

//                  DETERMINE IF DATA HAS ARRIVED AND PROCESS IT
//*************************************************************************************

    Timer1->Enabled = false;//Disable timer while processing data

    ReadFile(C1, pReadPt, buffAvail, &numBytesRead, &OVL);//read data from COMM 1

    if(numBytesRead != 0 ){ //If data is present ...  If not end.
numBytesRead = numBytesRead + segSize;
segmentedPkt = NumDataPkts(inBuff, numBytesRead, numPkts, pLastPkt, timeOut); //Does a partial packet exist in the

for(unsigned int p=0; p<numPkts; p++){ //Process data from all available data packets

switch(*pPkt){ //switch UART header

case UART_EVENT_PACKET:
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
FindEvent(pPkt);
break;

case UART_ACL_DATA_PACKET:
ClearStatus();
LStatus1->Caption = "Data Received";

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;
LStatus2->Caption = pktSize;
```

```
// CHECK THE MOST SIGNIFICANT BYTE OF THE L2CAP CHANNEL ID
// THIS BYTE TELLS WHAT TYPE OF DATA IS BEING RECIEVED:
// TEST DATA OR MESSAGE DATA.  TEST DATA MUST BE PROCESSED
// AND RETRANSMITTED, BUT MESSAGE DATA IS DISPLAYED TO THE
// USER AND THEN DISCARDED.
//*****************************************************************

// IF THE ACL FLAGS INDICATE A CONTINUING PACKET THE PAKCET MUST
// TRANSMISSION TEST DATA.  SINCE CONTINUING PACKETS DO NOT
// CONTAIN L2CAP HEADER INFORMATION THE l2capCIDmsb VARIABLE
// MUST BE MANUALLY SET TO TEST_DATA_MSB.
//*****************************************************************

switch (*(pPkt+8)){ //Switch L2CAP CID msb

case MESSAGE_MSB:

//########### ACL Data Display for ONLY one BT Slave ############
for(unsigned int n=9; n<pktSize; n++)
dataDisp[n-9] = *(pPkt+n); //Move data from data pkt to a string for display
dataDisp[pktSize-9] = NULL; //Add a null char to end for string output
LRx->Caption = dataDisp;
break;

case TEST_DATA_MSB:
//IF EXECUTION REACHES THIS POINT AT LEAST ONE COMPLETE PKT EXISTS

/*******************************************************************************************
IF TEST DATA IS BEING RECIEVED EXECUTION WILL REMAIN IN THIS LOOP UNTIL THE WHOLE
TEST DATA FILE HAS BEEN READ INTO THE INPUT BUFFER.  DURING EACH ITERATION THE DATA
READ WILL BE COUNTED AND COMPARED TO THE TOTAL SIZE OF THE PAKCET.  IT WILL ALSO BE
FOR OTHER TYPES OF DATA WHICH MAY BE MIXED IN AND STORE THAT DATA SEPARATELY.  ONCE
THE COMPLETE FILE IS READ THE DATA WILL BE PROCESSED.  THIS WHOLE PROCESS WILL BE
TIMED AND RETURNED TO THE TESTING DEVICE AS Tproc (PROCESSING TIME).
*******************************************************************************************/

do{ // continue loop until the complete test file has been read
for(curProcPkts=processedPkts; curProcPkts<numPkts; curProcPkts++){//loop through all counted data pkts
ClearStatus();
LStatus1->Caption = "Receiving Test Data";

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask;//Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask;//into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//* IS THIS THE FIRST ACL PACKET OF A LARGE L2CAP PACKET? *
aclFlags = (*(pPkt+2) & flagMask);// Extract ACL flags

switch(aclFlags){
case FIRST_PKT_P2P:
//YES THIS IS THE FIRST OF MANY
```

```
//Subtract 9 bytes from the total pkt size
//5 bytes for the ACL pkt header and 4 bytes
//for the L2CAP pkt header.
testDataReceived = pktSize - 5 - 4;
//Find L2CAP Pkt Size
temp1 = (static_cast<unsigned int>(*(pPkt+5))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+6))) & mask; //into an int variable.
temp2 = temp2 << 8;
l2capPktSize = temp1 + temp2; //Size of exptected l2cap pkt

LStatus2->Caption = "L2CAP Packet Size:";
LStatus3->Caption = l2capPktSize;
break;

case CONT_PKT_P2P:
//NO THIS IS A CONTINUING PACKET TO ADD TO A PREVIOUSLY
//RECIEVED L2CAP FRAGMENT.

//Subtract only the 5 bytes for the ACL pkt header.
testDataReceived = testDataReceived + pktSize - 5;
break;

default:
LTestStatus->Caption = "Unknown ACL Data Packet Flags";
if(*pPkt == UART_EVENT_PACKET){//if an event packet was recieved store it for processing later
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
ClearStatus();
LStatus1->Caption = "Event Received During Test";
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
FindEvent(pPkt);
}//end if

}//end switch aclFlags
//Finished with present packet...move pointer to next data packet
pPkt = pPkt + pktSize;
amtProcData = amtProcData + pktSize;
pktSize = 0;
                                                       LPktNum->Caption = curProcPkts;
}//end for

processedPkts = numPkts; // update the number of processed packets

//WILL THERE BE MORE DATA COMING?
if(testDataReceived < l2capPktSize){//more data is coming
do{
//If there is a segmented packet compute the amount of data that does exist
if(segmentedPkt)
segSize = numBytesRead - amtProcData;
else{
segSize = 0;
LStatus2->Caption = " ";
}//end else

pReadPt = inBuff + amtProcData + segSize; //move the read point to the end of the data in the buffer
buffAvail = inBuffSize - amtProcData; //How much of the buffer is still free
ReadFile(C1, pReadPt, buffAvail, &newBytesRead, &OVL); //read data from COMM 1
```

```
//**** DEBUGING CODE ***
if(newBytesRead == 0){
loopCount++;
if(loopCount == loopLimit)
timeOut = true;
BT_net->LTestStatus->Caption = "Waiting for data. No Bytes Read";
}//end if
//**********************
else{
newBytesRead = newBytesRead + segSize; //Add segment size to newBytesRead so that a complete packet is sent to NumD
segmentedPkt = NumDataPkts(pPkt, newBytesRead, numPkts, pLastPkt, timeOut);     //Does a partial packet exist in th
numBytesRead = numBytesRead + newBytesRead - segSize; //add the new bytes read to the number of bytes already in th
numPkts = numPkts + processedPkts; //add the new number of packets to the number of packets already in the buffer
}//end else

}while(segmentedPkt && !timeOut); //continue loop until a segmented pkt no longer exists
}//end if more data is coming

}while((testDataReceived<l2capPktSize) && !timeOut); //Continue loop until the whole Test file (L2CAP
//pkt) is recieved or a timeout occurs.

//**************************** PROCESS RECIEVED TEST DATA ****************************************

if (testDataReceived  == l2capPktSize || timeOut){
//NO MORE DATA IS COMING OR A TIMEOUT HAS OCCURED
pPkt = inBuff;//reset the pkt pointer to point to the beginning of the data
while(*pPkt == UART_EVENT_PACKET){//make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

if(localDevice.conductingTest){
//LOCAL DEVICE IS CONDUCTING THE TEST

//Stop Timer. The LSB of the L2CAP CID contains a
//0 for a small file and a 1 for a large file.
fileType = static_cast<short>(*(pPkt+7));

//Read the processing time used by the other computer (1st and 2nd data bytes in the test data pkt)
temp1 = (static_cast<unsigned int>(*(pPkt+9))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+10))) & mask; //into an int variable.
temp2 = temp2 << 8;
t_proc = temp1 + temp2; //Processing time used on remote computer

if(t_proc == TIME_OUT_REMOTE_COMP){ //A timeout occured on remote computer
LTestStatus->Caption = "Timeout occured on remote computer";
cTest.endReceive(fileType, t_proc, errors); //end test
}//end if
else {
if(timeOut){ //A timeout occured on this computer
LTestStatus->Caption = "Timeout occured on this computer";
cTest.endReceive(fileType, TIME_OUT_LOCAL_COMP, errors); //end test
}//end if
else{
```

```
cTimer.startTimer();

//STORE RECIEVED DATA
storedL2capData = 0;
for(unsigned int b=0; b<numPkts; b++){ //loop through all packets and save them for retransmission
while(*pPkt == UART_EVENT_PACKET){ //make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//STORE THIS DATA
for (n=0; n<pktSize-5; n++)
testDataBuff[n+storedL2capData] = *(pPkt+n+5);
storedL2capData = storedL2capData + pktSize - 5;  //Account for ACL pkt header
pPkt = pPkt + pktSize;
}//end for

//ALL DATA HAS BEEN SAVED..RESET pPkt and pktSize
pPkt = inBuff;
pktSize = 0;

//CHECK FOR ERRORS
for(unsigned int k=6; k<l2capPktSize+4; k++){ //Begin k @ 6 to account for L2CAP pkt header and t_proc bytes
if(cTest.dataLong[k] != testDataBuff[k]) //If the saved test data does not
errors++; //equal the transmitted data then an
}//end for //error occured.

t_proc = t_proc + static_cast<unsigned int>(cTimer.stopTimer()*1000);
cTest.endReceive(fileType, static_cast<float>(t_proc)/1000, errors); //end this test
}//end else
}//end else

numTestsRan++;
LCurTrans->Caption = numTestsRan;

//IF MORE TESTS NEED TO BE RUN...
//TRANSMIT MORE DATA
if (numTestsRan < numTestTrans){
TransmitTestData(fileType, 0);
LTestStatus->Caption = "Transmit next pkt";
}//end if
else{
readRssi = true;
numTestsRan = 0;
LTestStatus->Caption = "Finished!";
MessageBox(0, "Test Finished", 0, MB_OK);
```

```
}//end else

}//end if local device conducting test
else{
//LOCAL DEVICE IS NOT CONDUCTING THE TEST
cTimer.startTimer();

storedL2capData = 0;
for(unsigned int b=0; b<numPkts; b++){ //loop through all packets and save them for retransmission
while(*pPkt == UART_EVENT_PACKET){ //make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//STORE THIS DATA
for (n=0; n<pktSize-5; n++)
testDataBuff[n+storedL2capData] = *(pPkt+n+5);
storedL2capData = storedL2capData + pktSize - 5;  //Account for ACL pkt header
pPkt = pPkt + pktSize;

if(timeOut) //If a timeout has occured only store 1st ACL pkt
b = numPkts;
}//end for

//ALL DATA HAS BEEN PROCESSED..RESET pPkt and pktSize
pPkt = inBuff;
pktSize = 0;

if(timeOut) //If a timeout has occured set t_proc to 0xFFFF
t_proc = TIME_OUT_REMOTE_COMP;
else{
//convert the float (seconds) value to an unsigned int (milliseconds) value for transmission
t_proc = static_cast<unsigned int>(cTimer.stopTimer()*1000);
}//end else

Int2Char(t_proc, bytesOfInt);
testDataBuff[4] = bytesOfInt[0]; //The first 4 bytes of the test data are the L2CAP header
testDataBuff[5] = bytesOfInt[1]; //store the value of t_proc in bytes 5 and 6.

//Retransmit
if(timeOut) //Let the transmit test function know if a timeout occured
TransmitTestData(RETRANSMIT, 2);
else
TransmitTestData(RETRANSMIT, l2capPktSize);
}//end else local device is not conducting the test
                                                //RESET VARIABLES
```

```
        testDataReceived = 0;
        l2capPktSize = 0;
        numPkts = 0;
        timeOut = false;
}//end if (testDataReceived == l2capPktSize)
else{
//SOMETHING SCREWED UP
LTestStatus->Caption = "SOMETHING SCREWED UP: Too much data received";
}//end else someting screwed up
break;
//**************************************************************************************************************


default:
LTestStatus->Caption = "Unknown L2CAP CID";
}//end switch L2CAP CID msb
}//end switch UART header

//Finished with present packet...move pointer to next data packet
pPkt = pPkt + pktSize;
amtProcData = amtProcData + pktSize;
pktSize = 0;
}//end for

//   IF A SEGMENTED DATA PACKET EXISTS MOVE EXISTING DATA
//   TO THE BEGINING OF THE BUFFER AND SET THE READ POINT
//   AT THE END OF THE PARTIAL DATA
//*********************************************************************************

if(segmentedPkt){ //The last packet read was not complete

segSize = numBytesRead - amtProcData;
for(unsigned int c=0; c<segSize; c++) //Temporarily store incomplete pkt
tempAry[c] = *(pLastPkt+c);
ResetBuffer(inBuff, inBuffSize); //Reset inBuff
for(unsigned int c=0; c<segSize; c++) //Place incomplete pkt in the front
inBuff[c] = tempAry[c]; //of inBuff
pReadPt = inBuff + segSize; //Set the Read Point pointer to the end of the incomplete pkt
buffAvail = inBuffSize - segSize; //Adjust the buffer size to reflect the portion used
//by the incomplete pkt.  (Needed for reading from COM 1)
}//end if segmentedPkt

else{ //Last packet was complete
ResetBuffer(inBuff, inBuffSize); //Reset inBuff
pReadPt = inBuff; //Reset Read Point pointer
buffAvail = inBuffSize; //Reset available buffer size.
segSize = 0;
}//end else

}//end if data is present


// IF A CONNECTION EXISTS READ AND RECEIVE SIGNAL
// STRENGTH (RSSI) FOR THE EXISTING LINKS.
//************************************************************************************
//############### THIS CODE MUST BE EDITED FOR USE WITH MORE THAN 2 DEVICES
//RSSI
if(readRssi&&(!segmentedPkt)){
if(slave[0].connection && localDevice.conductingTest){
```

```
if(cycle%rssiFreq == 0){
pktSize = ReadRSSI(slave[0].connectHndl, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
} //end if
cycle++;
}//end if
}//end if readRssi

Timer1->Enabled = true;//Data processing complete...re-enable timer
}


//---------------------------------------------------------------------------

void FindOpCode(char *pBuff){

//******* VARIABLES *******
//----- Control Variables
unsigned int opCode, mask = 0x000000FF;
static bool inquiryFilter; //Used to determine which filter needs to be set
   //(connection or inquiry) for each call to setEventFilter

//----- Display Variables -----
char opCodeDisp[7] = {'0','x'}, tempOpCode[4];
unsigned int x;

//----- Temporary Varialbes -----
unsigned int temp1, temp2;
//*************************

    // FIND THE OPCODE AND STORE IT IN AN INTEGER VARIABLE
// FOR USE IN A SWITCH STATEMENT.  THE OPCODE IS CONTAINED
// IN A DIFFERENT PART OF THE EVENT PACKET FOR A COMMAND COMPLETE
// EVENT AND A COMMAND STATUS EVENT.
//*************************************************************************************

    if (*(pBuff+1) == EV_COMMAND_COMPLETE){
    temp1 = static_cast<unsigned int>(*(pBuff+4)) & mask; //mask out all unwanted
    temp2 = static_cast<unsigned int>(*(pBuff+5)) & mask; //bytes.
    temp2 = temp2<<8; //shift the msbyte to its position
    opCode = temp1 + temp2; //combine the two bytes into one integer variable
    }//end if

    if (*(pBuff+1) == EV_COMMAND_STATUS){
        temp1 = static_cast<unsigned int>(*(pBuff+5)) & mask; //mask out all unwanted
    temp2 = static_cast<unsigned int>(*(pBuff+6)) & mask; //bytes.
    temp2 = temp2<<8; //shift the msbyte to its position
    opCode = temp1 + temp2; //combine the two bytes unsigned into one integer variable
    }//end if

// FIND THE RETURNED OPCODE AND EXEUTE ACCORDINGLY
//************************************************************************************

switch (opCode){

//####### NOTE: 1st Parameter in command complete event pkt is array index #6
//####### Status parameter for command status event is array index #3

// THE RESET COMMAND TRIGGERS A CHAIN OF SETUP COMMANDS
```

```
// TO PREPARE THE BT MODULE FOR USE.  EACH RESPECTIVE
// COMMAND COMPLETE EVENT LEADS TO THE EXECUTION OF THE
// NEXT COMMAND IN THE CHAIN ENDING WITH "WRITE PAGE
// TIMEOUT"  THE COMMAND COMPLETE EVENTS FOR ALL OF THE
// SETUP COMMMANDS ARE LISTED HERE IN THIER ORDER OF EXECUTION
// BEGINING WITH THE RESET COMMAND.
//*********************************************************************************

//----------- RESET -----------
case HCI_RESET:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If reset sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Device Reset";

//ISSUE NEXT SETUP COMMAND
pktSize = ReadBD_ADDR(outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called
inquiryFilter = false;
}//end if
        else { //If host buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Reset FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- READ BLUETOOTH DEVICE ADDRESS ----------
case HCI_READ_BD_ADDR:

//Extract the BD ADDR parameter and convert it to a string of
        //ASCII characters for display to the user.
        for (unsigned int n=0; n<6; n++)   //Extract 6 byte address from
localDevice.bd_addr[n] = *(pBuff+7+n);//the event packet

//Convert each nibble of the address to an ASCII
        //char representative of the hex digit contained in each nibble.
        for (unsigned int n=0; n<12; n=n+2){
x=n/2;
            nibbles(localDevice.bd_addr[5-x], localDevice.addrDisp[n+2], localDevice.addrDisp[n+3]);
        }//end for

        localDevice.addrDisp[14] = NULL; //Append a null for string display.
        BT_net->LBD_ADDR_ld->Caption = localDevice.addrDisp;

//ISSUE NEXT SETUP COMMAND
pktSize = ReadBufferSize(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

break;

//---------- READ BUFFER SIZE ----------
case HCI_READ_BUFFER_SIZE:
//WHAT IS THE STATUS OF THE COMMAND
```

```
if (*(pBuff+6) == 0){ //If read buffer size sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Read Buffer Size Complete";

//EXTRACT DATA FROM PACKET
//ACL buff size
temp1 = static_cast<unsigned int>(*(pBuff+7)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+8)) & mask;
temp2 = temp2<<8;
hostCtrl.sizeACLBuff = temp1 + temp2;
//SCO buff size
hostCtrl.sizeSCOBuff = *(pBuff+9);
//number of ACL data buffers
temp1 = static_cast<unsigned int>(*(pBuff+10)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+11)) & mask;
temp2 = temp2<<8;
hostCtrl.numACLBuff = temp1 + temp2;
//number of SCO data buffers
temp1 = static_cast<unsigned int>(*(pBuff+12)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+13)) & mask;
temp2 = temp2<<8;
hostCtrl.numSCOBuff = temp1 + temp2;

//ISSUE NEXT SETUP COMMAND
pktSize = HostBufferSize(&host, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

/*//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(INQUIRY_RESULT, 0, outBuff);//set inquiry result filter
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = true;*/

}//end if
        else { //If read buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Read Buffer Size FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- HOST BUFFER SIZE ----------
case HCI_HOST_BUFFER_SIZE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If host buffer size sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Host Buffer Size Sent";

//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(INQUIRY_RESULT, 0, outBuff); //set inquiry result filter
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
```

```
inquiryFilter = true;
}//end if
        else { //If host buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Host Buffer Size FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- SET EVENT FILTER ----------
case HCI_SET_EVENT_FILTER:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If set event filter sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Event Filter Set";
//ISSUE NEXT SETUP COMMAND (depends on which filter was last set)

if (inquiryFilter){ //was the inquiry filter the last one set?
pktSize = WriteScanEnable(3, outBuff); //Setting WSE to 3 enables all scans
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if inquiry filter

else{ //the connection filter was the last filter set
pktSize = WriteConnAcceptTimeout(0x2000, outBuff); //0x2000*625us = 5.12s
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end else inquiry filter
}//end if
        else { //If set event filter failed
ClearStatus();
BT_net->LStatus1->Caption = "Set Event Filter FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- WRITE SCAN ENABLE ----------
case HCI_WRITE_SCAN_ENABLE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write scan sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Write Scan Enable Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = WriteVoiceSetting(0x0060, outBuff); //0x0060 sets 16bit 2's complement input type
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if
        else { //If write scan failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Scan Enable FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- WRITE VOICE SETTING ----------
case HCI_WRITE_VOICE_SETTING:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write voice setting sucessful
```

```
ClearStatus();
BT_net->LStatus1->Caption = "Write Voice Setting Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = WriteAuthEnable(0x00, outBuff); //0x00 Disables authentication
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

}//end if
        else { //If write voice setting failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Voice Setting FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- WRITE AUTHENTICATIO ENABLE ----------
case HCI_WRITE_AUTHENTICATION_ENABLE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write authentication enable sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Write Authentication Enable Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(CONNECTION_SETUP, AA_ON_RS_DISABLED, outBuff);
//set connection setup filter and auto accept off
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = false;

}//end if
        else { //If write authentication enable failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Authentication Enable FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else
break;

        //---------- SET UART BAUD RATE ----------
case HCI_SET_UART_BAUD_RATE:
                //WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If set uart baud rate sucessful
ClearStatus();
BT_net->LStatus1->Caption = "UART Baud Rate Set to 460.8kbps";

//Set Baud Rate on Comm 1  to 460.8kbps
myDCB.BaudRate = CBR_115200;
SetCommState(C1, &myDCB);

//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(CONNECTION_SETUP, AA_ON_RS_DISABLED, outBuff);
//set connection setup filter and auto accept off
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
```

```
//each time this command is called.
inquiryFilter = false;
}//end if
else { //If set uart baud rate failed
ClearStatus();
BT_net->LStatus1->Caption = "Set UART Baud Rate FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
}//end else
break;

        //---------- WRITE CONNECTION ACCEPT TIMOUT ----------
case HCI_WRITE_CONNECTION_ACCEPT_TIMEOUT:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write connection accept timeout sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Connection Accept Timeout Written";

//ISSUE NEXT SETUP COMMAND
pktSize = WritePageTimeout(0x3000, outBuff); //0x3000*625us = 7.68s
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if
        else{ //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Connection Accept Timeout FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- WRITE PAGE TIMEOUT (END OF SETUP COMMANDS) ----------
case HCI_WRITE_PAGE_TIMEOUT:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write page timeout sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Page Timeout Written";
//If execution makes it to this point the setup is successful.
BT_net->LStatus2->Caption = "Setup Completed Successfully";
}//end if
        else { //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Page Timeout FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else
break;

//---------- INQUIRY STATUS ----------
case HCI_INQUIRY:

//Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Conducting Inquiry";
}//end if
        else{
BT_net->LStatus1->Caption = "Inquiry Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;
```

```
//---------- CREATE CONNECTION STATUS ---------
case HCI_CREATE_CONNECTION:

        //Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Creating Connection";
}//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Create Connection Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;

//---- ACCEPT CONNECTIO REQUEST STATUS (NOT NECCESSARY BECAUSE THE AUTO ACCEPT ----
// ---- FLAG IS CURRENTLY SET ON THE CONNECTION FILTER ----
case HCI_ACCEPT_CONNECTION_REQUEST:

//Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Accepting Connection Request";
}//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Accept Connection Request Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;

//---------- WRITE LINK SUPERVISION TIMOUT ----------
case HCI_WRITE_LINK_SUPERVISION_TIMEOUT:
/******INSERT CODE HERE*****/
break;

//---------- READ TRANSMIT POWER LEVEL ----------
case HCI_READ_TRANSMIT_POWER_LEVEL:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If read transmit power level sucessful

//Display Tx Power Level
//BT_net->LTxPower->Caption = (static_cast<unsigned int>(*(pBuff+9))) & mask;

//########### THIS CODE MUST BE EDITED FOR USE WTIH MORE THAN 2 DEVICES
//ISSUE READ RECIEVE SIGNAL STRENGTH INDICATOR (RSSI) COMMAND
pktSize = ReadRSSI(slave[0].connectHndl, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

}//end if
else{ //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Read Transmit Power Level FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
}//end else
break;
```

```
//---------- READ RECIEVE SIGNAL STRENGTH INDICATOR ----------
case HCI_READ_RSSI:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If read RSSI sucessful

//Display Receive signal strength
            localDevice.rssi = static_cast<short>(*(pBuff+9));
BT_net->LRSSI->Caption = localDevice.rssi;

}//end if
else { //If read RSSI failed
ClearStatus();
BT_net->LStatus1->Caption = "Read RSSI FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
}//end else
break;

//-------------------------------------------------------------

case CLEAR:
break;

default:
ClearStatus();
BT_net->LStatus1->Caption = "Unknown OpCode";

//Convert each nibble of the opcode to an ASCII
        //char representative of the hex digit contained in each nibble.
Int2Char(opCode, tempOpCode);
nibbles(tempOpCode[1], opCodeDisp[2], opCodeDisp[3]);
        nibbles(tempOpCode[0], opCodeDisp[4], opCodeDisp[5]);
BT_net->LStatus2->Caption = opCodeDisp;
}//end switch
}//end function

//----------------------------------------------------------------------------

void FindEvent(char *pBuff){

//******** VARIABLES ********
//------ Control Variables ------
char evt;
    unsigned int x = 0, numHndls = 0, numResp = 0;
    unsigned int mask = 0x000000FF;

//------ Display Variables ------
    //char addrDisp[15] = {'0','x'};
    char connectHndlDisp[7] = {'0','x'}, evtDisp[5]={'0','x'};
    unsigned int errorCode, numCompPkts = 0;
//***************************

evt=*(pBuff+1);

// FIND THE RECEIVED EVENT AND EXECUTE ACCORDINGLY
//****************************************************************************************

switch (evt){
```

```
//###### NOTE: 1st Parameter in event pkt is array index #3

case EV_COMMAND_COMPLETE:

FindOpCode(pBuff);
break;

case EV_COMMAND_STATUS:

FindOpCode(pBuff);
break;

case EV_INQUIRY_COMPLETE:
ClearStatus();
BT_net->LStatus1->Caption = "Inquiry Complete";
if (*(pBuff+3) == 0)
BT_net->LStatus2->Caption = "Completed Sucessfully";
else
BT_net->LStatus2->Caption = Error(*(pBuff+3));
break;

case EV_INQUIRY_RESULT:

//###### NOTE: MUST ADD COUNTER TO COUNT RESPONSES. ALL RESPONDING DEVICES MAY NOT BE
//######       CONTAINED IN ONE EVENT BUT MAY CREATE SEVERAL EVENTS THEREFORE YOU NEED
//######    TO KNOW HOW MANY DEVICES HAVE PREVIOUSLY RESPONED AND ARE ALREADY ACCOUNTED FOR.

numResp = (static_cast<unsigned int>(*(pBuff+3))) & mask;
BT_net->LInqRes->Caption = numResp;

//Load the Bluetooth device address for each device that responded
//into the bd_addr array for each of the slave structures (up to 7
//slaves possible) and other device specific data using nested for loops
for (unsigned int n=0; n<numResp; n++){
for (unsigned int c=0; c<6; c++)
slave[n].bd_addr[c] = *(pBuff+4+(6*n)+c);
}//end for

for (unsigned int n=0; n<numResp; n++){
slave[n].pgScnRepMode = *(pBuff+4+(6*numResp)+n);
}//end for

for (unsigned int n=0; n<numResp; n++){
slave[n].pgScnPerMode = *(pBuff+4+(7*numResp)+n);
}//end for

for (unsigned int n=0; n<numResp; n++){
slave[n].pgScnMode = *(pBuff+4+(8*numResp)+n);
}//end for

for (unsigned int n=0; n<numResp; n++){
for (unsigned int c=0; c<3; c++)
slave[n].deviceClass[c] = *(pBuff+4+(9*numResp)+(3*n)+c);
}//end for

for (unsigned int n=0; n<numResp; n++){
for (unsigned int c=0; c<3; c++)
slave[n].clkOffset[c] = *(pBuff+4+(12*numResp)+(2*n)+c);
}//end for
```

```
//**** These following lines of code are for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices


//Convert each nibble of the address to an ASCII
        //char representative of the hex digit contained in each nibble.
        for (unsigned int n=0; n<12; n=n+2){
x=n/2;
///##### Counter will affect this code
            nibbles(slave[0].bd_addr[5-x], slave[0].addrDisp[n+2], slave[0].addrDisp[n+3]);
        }//end for

        slave[0].addrDisp[14] = NULL; //Append a null for string display.
BT_net->LBD_ADDR_rd->Caption = slave[0].addrDisp;
break;

case EV_CONNECTION_COMPLETE:

//**** This Code is for connecting only two computers
//**** This must be modified before attempting to use with
//**** more than two computers in a piconet

if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Connection Completed";
BT_net->LStatus2->Caption = "Successfully";

//#### Counter will affect this code
//Load the information recieved about the
//remote device into a BT_DEVICE structure


slave[0].connection = true; //Yes a connection exists

slave[0].connectHndl[0] = *(pBuff+4);
slave[0].connectHndl[1] = *(pBuff+5);
slave[0].linkType = *(pBuff+12);
slave[0].encryptMode = *(pBuff+13);

//Inform user of the type of link established
if (slave[0].linkType == ACL_LINK)
BT_net->LLinkType->Caption = "ACL";
if (slave[0].linkType == SCO_LINK)
BT_net->LLinkType->Caption = "SCO";

            //Convert each nibble of the Connection Handle to an ASCII
//char representative of the hex digit contained in each nibble.
for (unsigned int n=0; n<4; n=n+2){
x=n/2;

///##### Counter will affect this code

nibbles(slave[0].connectHndl[1-x], connectHndlDisp[n+2], connectHndlDisp[n+3]);
}//end for

connectHndlDisp[6] = NULL; //Append a null for string display.
BT_net->LConnectHndl->Caption = connectHndlDisp;
```

```
        }//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Connection Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
        }//end else
break;

case EV_NUM_COMPLETED_PACKETS:

//#### NOTE: THIS CODE IS WRITTEN FOR A SINGLE POINT TO POINT CONNECTION WHEN EDITING
//####   CODE TO HANDLE SEVERAL DEVICES THE CONNECTION HANDLES RETURNED MAY NEED
//####   TO BE CHECKED AND COMPARED.

//BEGIN EXTRACTING DATA FROM PACKET
numHndls = *(pBuff+3); //number of connection handles returned

if(numHndls > 1){
//Because of the nature of this test program (transmit...wait for remote
//computer to process data then re-transmit...there should be no more than
//one connection handle returned from a Number of Completed Packets event.
MessageBox(0, "ERROR: More than 1 connection handle\nreturned in NUM_COMP_PKTS event",0 , MB_OK);
}//end if
else{
numCompPkts = *(pBuff+6); //Only need to read LSB of comp ACL pkts because the device buffer
//only holds 10 pkts
if(transmitInProgress){ //If there is a transmit in progress continue with it
TransmitTestData(CONTINUE_TRANSMISSION, numCompPkts);
}//end if
}//end else
break;

case EV_DATA_BUFFER_OVERFLOW:
ClearStatus();
MessageBox(0, "DATA BUFFER OVERFLOW", 0, MB_OK);
break;

case EV_DISCONNECTION_COMPLETE:
ClearStatus();
BT_net->LStatus1->Caption = "Connection Terminated";
slave[0].connection = false;
        break;

case EV_MAX_SLOTS_CHANGE:
BT_net->LStatus3->Caption = "Max LMP Slots Change";
break;

default:
ClearStatus();
BT_net->LStatus1->Caption = "Unknown Event";

//Convert each nibble of the event code to an ASCII
        //char representative of the hex digit contained in each nibble.
        nibbles(evt, evtDisp[2], evtDisp[3]);
BT_net->LStatus2->Caption = evtDisp;

}//end switch
}//end function
```

```
//------------------------------------------------------------------------

void ResetBuffer(char buff[], DWORD buffSize){
for (unsigned int n=0; n<buffSize; n++)
buff[n] = NULL;
}//end function

//------------------------------------------------------------------------

bool NumDataPkts(char buff[], DWORD numBytesRead, DWORD &numPkts, char *pLastPkt, bool &dataError){
DWORD paramLength = 0, availBuff = 0;
bool moreData = true, pktSeg = false;
unsigned int temp1, temp2, mask = 0x000000FF;
        unsigned int totalData = 0;

numPkts = 0;
pLastPkt = buff;
while(moreData){//while unaccounted for data exists do....
switch(*pLastPkt){//what type of data is this packet?

case UART_EVENT_PACKET:
//The third byte (2nd array element) in an event pkt gives
//the total length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//plus current amount of data to compute total data present.
paramLength = static_cast<unsigned int>(*(pLastPkt+2)) & mask;
totalData = totalData + 3 + paramLength;
numPkts++;
break;

case UART_ACL_DATA_PACKET:
//The 4th and 5th bytes (3rd and 4th array elemetnts) in an ACL pkt
//give the total length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//plus current amount of data to compute total data present.
temp1 = (static_cast<unsigned int>(*(pLastPkt+3))) & mask; //Convet 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pLastPkt+4)))& mask; //into an int variable.
temp2 = temp2 << 8;
paramLength = temp2 + temp1;
totalData = totalData + 5 + paramLength;
numPkts++;
break;

default: //Unrecognized pkt header...Error in received data
moreData = false;
dataError = true;
numPkts = numPkts+1;
BT_net->LStatus2->Caption = "Error in received data";
do{
//Clear the input buffer
//The data is useless since it contained an error
                                        availBuff = inBuffSize - numBytesRead;
ReadFile(C1, pLastPkt, availBuff, &numBytesRead, &OVL);//read data from COMM 1
}while(numBytesRead > 0);
                                return false;

}//end switch
```

```
if (numBytesRead == totalData){
moreData = false;//All data accounted for
pktSeg = false;
}//end if
else{//1st else
if (numBytesRead < totalData){//Some expected data is missing
moreData = false; //must read buffer again.
pktSeg = true;
numPkts = numPkts - 1; //Last pkt was a pkt segment...do not count it
BT_net->LStatus2->Caption = "SEGMENTED PACKET";
}//end if numBytesRead<totalData
else {//2nd else
pLastPkt = buff;
pLastPkt = pLastPkt + totalData; //More data remaining to be counted.
}//end 2nd else
}//end 1st else
}//end while
return pktSeg;
}//end function

//---------------------------------------------------------------------------

void __fastcall TBT_net::BInquiryClick(TObject *Sender){

char inqDuration = 24; //= 24*1.28s = 30.72s

    pktSize = Inquiry(inqDuration, 1, outBuff); //Returns after 1 inquiry is received.
    WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BBeginTestClick(TObject *Sender){
        //TEST SETUP
        AnsiString sDist;
short sDistSize;
char dist[15] = {NULL};

//Assign the value entered by the user into the Transmission Test class "cTest"
sDist = CBDist->Text;
sDistSize = sDist.Length();
for (short c=1; c<=sDistSize; c++)
dist[c-1] = sDist[c];
dist[sDistSize] = NULL;

        LStatus2->Caption = dist;

cTest.setDistance(dist, sDistSize);

        AnsiString sTop;

//Assign the value entered by the user into the Transmission Test class "cTest"
sTop = CBTopology->Text;
cTest.setTopology(sTop[1]);
        LStatus2->Caption = sTop[1];

//BEGIN TEST
cTest.newTest();
cTest.setLinkState(localDevice.addrDisp, slave[0].addrDisp, localDevice.rssi);
```

```
        readRssi = false;

//Begin by transmitting the small file
TransmitTestData(SMALL_FILE, 0);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BSendClick(TObject *Sender){
DWORD dataLength;
AnsiString input;
unsigned int inputSize;

input = Message->Text;
Message->Text = ""; //Reset message block
inputSize = input.Length();
dataLength = inputSize + 4;
for (unsigned int c = 1; c <= inputSize; c++)
outBuff[c+3] = input[c];

//***** Adding L2CAP 4 byte header *****
//The first two bytes of the header are the L2CAP Packet length
//(i.e. The length of data to be transmitted may be greater than the HCI
//packet lenght)
//The second two bytes of the header is the Channel ID (CID) and can
//range from 0x0040 to 0xFFFF. For message traffic use 0x0300.

//***NOTE: Message is assumed to be <= 255 bytes long

outBuff[0] = static_cast<char>(inputSize); //L2CAP pkt length lsb
outBuff[1] = 0x00; //L2CAP pkt length msb
outBuff[2] = 0x00; //CID lsb
outBuff[3] = 0x03; //CID msb

    //**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

pktSize = SendACLPkt(slave[0].connectHndl, dataLength, FIRST_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BResetClick(TObject *Sender){

        //RESET BT MODULE
        pktSize = Reset(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

        //RESET CONNECTION CONRTOL VAR
        for (unsigned int n=0; n<7; n++)
                slave[n].connection = false;

        //RESET DISPLAY
        ResetDisplay();
}

//---------------------------------------------------------------------------
```

```
void __fastcall TBT_net::BConnectClick(TObject *Sender){
    //**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

pktSize = CreateConnection(&slave[0], localDevice.pktType, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBControlCloseUp(TObject *Sender){

localDevice.master = CBControl->ItemIndex;

if(localDevice.master){//if the local device will be the master...
//buttons and comboboxes
BInquiry->Enabled = true;
BConnect->Enabled = true;
CBPktType->Enabled = true;
}//end if
else{
//buttons and comboboxes
BInquiry->Enabled = false;
BConnect->Enabled = false;
CBPktType->Enabled = false;
}//end else

}

//---------------------------------------------------------------------------



void __fastcall TBT_net::CBPktTypeCloseUp(TObject *Sender){
unsigned int pktRef;

pktRef = CBPktType->ItemIndex;

//Assign the value entered by the user into the Transmission Test class "cTest"
switch(pktRef){
case 0:
localDevice.pktType = DM1;
cTest.setPktType(DM1);
break;
case 1:
localDevice.pktType = DH1;
cTest.setPktType(DH1);
break;
case 2:
localDevice.pktType = DM3;
cTest.setPktType(DM3);
break;
case 3:
localDevice.pktType = DH3;
cTest.setPktType(DH3);
break;
case 4:
```

```
localDevice.pktType = DM5;
cTest.setPktType(DM5);
break;
case 5:
localDevice.pktType = DH5;
cTest.setPktType(DH5);

}//end switch

//Disconnect all current connections.
for (unsigned int c=0; c<7; c++){
if (slave[c].connection & localDevice.master){
slave[c].connection = false;
Disconnecting();
LStatus1->Caption = "Disconnecting";
pktSize = Disconnect(slave[c].connectHndl, 0x13, outBuff); //user ended connection
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if
else
c = 7; //If no connection exists end loop (connections are established in sequential order
   //therefore if slave[0].connection does not exist there is no reason to complete the loop)
}//end for
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBTestRoleCloseUp(TObject *Sender){

localDevice.conductingTest = CBTestRole->ItemIndex;

if(localDevice.conductingTest){//if the local device will be conducting the test...
//buttons and comboboxes
BBeginTest->Enabled = true;
CBPktType->Enabled = true;
CBDist->Enabled = true;
CBTopology->Enabled = true;
CBNumTrans->Enabled = true;

}//end if
else{
//buttons and comboboxes
BBeginTest->Enabled = false;
CBPktType->Enabled = false;
CBDist->Enabled = false;
CBTopology->Enabled = false;
CBNumTrans->Enabled = false;

}//end else

}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBNumTransCloseUp(TObject *Sender){

//Assign twice the number of requested test
//transmissions to "numTestTrans" because two
//tests are run for each round of tests
numTestTrans = 2*(CBNumTrans->ItemIndex);
```

```
}

//---------------------------------------------------------------------------

void TransmitTestData(int fileSize, unsigned int tDataSize){
//******* VARIABLES ******
static DWORD dataLength, dataLeft;
static char *pDataFile;
char pktLength[4], cidLsb;
bool firstACLpkt = false;
unsigned int numACLPktsSent = 0, n;  //Loop control variable
unsigned int buffersAvailable = hostCtrl.numACLBuff;
const unsigned int maxPktSize = hostCtrl.sizeACLBuff; //max size of ACL pkt alowable so as not to
//overflow the ACL Buffer of the local BT module.
static unsigned int nextData2Send = 0; //contains the array index of the next data byte to send
//if nextData2Send == dataLength then all data has been sent
//************************

switch (fileSize){
case SMALL_FILE:
//SEND SMALL FILE
dataLength = cTest.sizeDataShort + 4;
pDataFile = cTest.dataShort;
cidLsb = LARGE_FILE;
transmitInProgress = true;
firstACLpkt = true;

BT_net->LTestStatus->Caption = "Transmitting Small File";
break;

case LARGE_FILE:
//SEND LARGE FILE
dataLength = cTest.sizeDataLong + 4;
pDataFile = cTest.dataLong;
cidLsb = SMALL_FILE;
transmitInProgress = true;
firstACLpkt = true;

BT_net->LTestStatus->Caption = "Transmitting Large File";
break;

case RETRANSMIT:
//LOCAL DEVICE IS NOT CONDUCTING TEST
//RETRANSMIT THE RECEIVED DATA

dataLength = tDataSize + 4;
pDataFile = testDataBuff;
firstACLpkt = true;

if(tDataSize == 2){ //A timeout has occured
*(pDataFile) = 0x02; //L2CAP pkt length lsb
*(pDataFile+1) = 0x00; //L2CAP pkt length msb
} //end if

BT_net->LTestStatus->Caption = "Retransmitting Received Data";
break;

case CONTINUE_TRANSMISSION:
```

```
//The new number of buffers available is equal
//to the number of packets completed.  This number
//is passed to TransmitTestData() as tDataSize
buffersAvailable = tDataSize;
break;

default:
BT_net->LTestStatus->Caption = "Unknown File type to Transmit";
}//end switch

if (dataLength <= maxPktSize){//If test data is smaller that the local device's
for (n=0; n<dataLength; n++) //ACL Buffer Size send it
outBuff[n] = *(pDataFile+n);

//***** Adding L2CAP 4 byte header *****
//The first two bytes of the header are the L2CAP Packet length
//(i.e. The length of data to be transmitted may be greater than the HCI
//packet lenght)
//The second two bytes of the header is the Channel ID (CID) and can
//range from 0x0040 to 0xFFFF.  For test data use 0xEE00

if ((fileSize!=RETRANSMIT) && (fileSize!=CONTINUE_TRANSMISSION)){
Int2Char(dataLength-4, pktLength);

outBuff[0] = pktLength[0]; //L2CAP pkt length lsb
outBuff[1] = pktLength[1]; //L2CAP pkt length msb
outBuff[2] = cidLsb; //CID lsb
outBuff[3] = TEST_DATA_MSB; //CID msb
}

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Packet is < 400 bytes";

pktSize = SendACLPkt(slave[0].connectHndl, dataLength, FIRST_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
if (fileSize != -1)
cTest.beginTransmit();

nextData2Send = dataLength; //Used to inform Program that Transmission is complete
}//end if
else{ //The data is too large to send in one ACL Packet...it must be split up
while((nextData2Send != dataLength) && (numACLPktsSent != buffersAvailable)){

if (dataLength > (nextData2Send+maxPktSize)){ //enough data exists to send another "full" ACL pkt.
//Load up the output Buffer
for (n=0; n<maxPktSize; n++)
outBuff[n] = *(pDataFile+nextData2Send+n);

if (firstACLpkt){
//If this is the first ACL pkt an L2CAP header needs to be
//added and also the ACL PB flag must be set to first packet

firstACLpkt = false;
```

```
if ((fileSize!=RETRANSMIT) && (fileSize!=CONTINUE_TRANSMISSION)){
Int2Char(dataLength-4, pktLength);

outBuff[0] = pktLength[0]; //L2CAP pkt length lsb
outBuff[1] = pktLength[1]; //L2CAP pkt length msb
outBuff[2] = cidLsb; //CID lsb
outBuff[3] = TEST_DATA_MSB; //CID msb
}//end if

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "1st ACL pkt of large L2CAP pkt";

pktSize = SendACLPkt(slave[0].connectHndl, maxPktSize, FIRST_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
if (fileSize != -1)
cTest.beginTransmit();

}//end if
else{
//If this is not the fist ACL pkt no L2CAP header needs to be added
//but the ACL PB flag must be set to continuing packet

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Continuing ACL pkt of large L2CAP pkt";

pktSize = SendACLPkt(slave[0].connectHndl, maxPktSize, CONT_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end else

nextData2Send = nextData2Send + maxPktSize; //increment nextData2Send
}//end if
else{ //There is not enough data left to send a "full" ACL pkt.
dataLeft = dataLength - nextData2Send;
//Load up the output Buffer
for (n=0; n<dataLeft; n++)
outBuff[n] = *(pDataFile+nextData2Send+n);

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Partially full ACL pkt for large L2CAP pkt";

pktSize = SendACLPkt(slave[0].connectHndl, dataLeft, CONT_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

nextData2Send = dataLength; //end the while loop
```

```
}//end else

numACLPktsSent++; //Increment the number of ACL pkts sent
}//end while
}//end else

if(nextData2Send == dataLength){ //Transmission is complete
BT_net->LStatus2->Caption = "Finished Transmitting File";
transmitInProgress = false; //Transmission is finished

//Reset Static variables
nextData2Send = 0;
dataLength = 0;
dataLeft = 0;
}//end if
}

//---------------------------------------------------------------------------

void ResetDisplay(){

//Buttons
BT_net->BInquiry->Enabled = false;
BT_net->BConnect->Enabled = false;
BT_net->BBeginTest->Enabled = false;

//Combo Boxes
BT_net->CBPktType->Enabled = false;
BT_net->CBDist->Enabled = false;
BT_net->CBTopology->Enabled = false;
BT_net->CBNumTrans->Enabled = false;

//Display Labels
BT_net->LBD_ADDR_rd->Caption = " "; //BT_ADDR for remote device 1
BT_net->LBD_ADDR_ld->Caption = " "; //BT_ADDR for local device
BT_net->LConnectHndl->Caption = " ";
BT_net->LInqRes->Caption = " ";
BT_net->LRSSI->Caption = " ";
BT_net->LRx->Caption = " ";
BT_net->LTestStatus->Caption = " ";
BT_net->LStatus1->Caption = " ";
BT_net->LStatus2->Caption = " ";
BT_net->LStatus3->Caption = " ";
BT_net->LCurTrans->Caption = " ";
BT_net->LPktNum->Caption = " ";

}

//---------------------------------------------------------------------------

void ClearStatus(){

//Clear all of the status labels
BT_net->LStatus1->Caption = " ";
BT_net->LStatus2->Caption = " ";
BT_net->LStatus3->Caption = " ";
}

//---------------------------------------------------------------------------
```

```
void Disconnecting(){

ClearStatus();

BT_net->LConnectHndl->Caption = " ";
BT_net->LRSSI->Caption = " ";
BT_net->LTestStatus->Caption = " ";
BT_net->LCurTrans->Caption = " ";
BT_net->LPktNum->Caption = " ";
}
```

Appendix C

BTTest v. 3.1

C.1 BTTest3_1.h

```
//BTTest3_1.h
//Written on 19OCT02 by MIDN Kenneth J. Hoover
//Last updated 22MAR03
//
//This header file defines forms and variables and contains function
//prototyes for the BTTest program.

//-------------------------------------------------------------------------------

#ifndef BTCommH
#define BTCommH

//-------------------------------------------------------------------------------

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "BT_functions.h"
#include "UART.h"
#include <ExtCtrls.hpp>
#include <Menus.hpp>
#include "transTest.h"

DWORD BaudRate[] = {CBR_9600, CBR_14400, CBR_57600, CBR_115200};
const DWORD testDataBuffSize = 100100, inBuffSize = 100100, outBuffSize = 600;//595 bytes usable (5 byte header for
COMMTIMEOUTS CT1, CTinit;
BT_DEVICE slave[7], localDevice;
BUFFERS host, hostCtrl;
unsigned long pktSize;
bool connection[7] = {0};
bool readRssi = true, transmitInProgress = false;
DCB myDCB, initDCB;

//DATA VARIABLES
char inBuff[inBuffSize], outBuff[outBuffSize], testDataBuff[testDataBuffSize];
TransTest cTest;
unsigned int numTestTrans;

//-------------------------------------------------------------------------------
```

```
class TBT_net : public TForm{

__published: // IDE-managed Components
        TTimer *Timer1;

        TGroupBox *GroupBox1;
        TGroupBox *GroupBox2;
        TGroupBox *GroupBox3;

//STATIC LABELS
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
TLabel *Label9;
TLabel *Label13;
TLabel *Label6;
TLabel *Label10;
        TLabel *Label11;
TLabel *Label12;
TLabel *Label14;
        TLabel *Label15;

//DISPLAY LABELS
        TLabel *LBD_ADDR_rd1; //BT_ADDR for remote device 1
TLabel *LBD_ADDR_rd2; //BT_ADDR for remote device 2
        TLabel *LBD_ADDR_ld; //BT_ADDR for local device
TLabel *LLinkType;
        TLabel *LConnectHndl1;
        TLabel *LInqRes;
        TLabel *LRSSI1;
        TLabel *LRx;
        TLabel *LTestStatus;
TLabel *LStatus1;
        TLabel *LStatus2;
TLabel *LStatus3;
TLabel *LCurTrans;
        TLabel *LPktNum;
TLabel *LConnectHndl2;
        TLabel *LRSSI2;

//INPUT BOX
TLabeledEdit *Message;

//BUTTONS
        TButton *BBeginTest;
        TButton *BSend;
        TButton *BReset;
        TButton *BConnect1;
TButton *BConnect2;
TButton *BInquiry;

//COMBOBOXES
        TComboBox *CBDist;
        TComboBox *CBTopology;
        TComboBox *CBControl;
        TComboBox *CBTestRole;
        TComboBox *CBPktType;
```

```
        TComboBox *CBNumTrans;
        TComboBox *CBSlaveSelect;


        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);

//TIMER INTERUPT
        void __fastcall Timer1Timer(TObject *Sender);

//BUTTON CLICKS
        void __fastcall BInquiryClick(TObject *Sender);
        void __fastcall BBeginTestClick(TObject *Sender);
        void __fastcall BSendClick(TObject *Sender);
        void __fastcall BResetClick(TObject *Sender);
        void __fastcall BConnect1Click(TObject *Sender);

//COMBOBOXES CLOSE
        void __fastcall CBControlCloseUp(TObject *Sender);
        void __fastcall CBPktTypeCloseUp(TObject *Sender);
        void __fastcall CBTestRoleCloseUp(TObject *Sender);
        void __fastcall CBNumTransCloseUp(TObject *Sender);
        void __fastcall BConnect2Click(TObject *Sender);

private: // User declarations
public: // User declarations



__fastcall TBT_net(TComponent* Owner);
};

//---------------------------------------------------------------------------

//***** FUNCTIONS *****

//FindOpCode():
//This function looks up the OpCode returned in a Command Complete or
//Command Status event packet and processes the data contained within
//the event packet accordingly.
//
//Parameters *pBuff: Pointer to the first byte in the returned
// event packet

void FindOpCode(char *pBuff);

//---------------------------------------------------------------------------

//Find Event():
//This function looks up Event retruned in an event packet and processes the
//data contained within the event packet accordingly
//
//Parameters *pBuff: Pointer to the first byte in the returned
// event packet

void FindEvent(char *pBuff);

//---------------------------------------------------------------------------
```

```
//ResetBuffer():
//This function resets an array of characters to NULL
//
//Parameters buff[]: Buffer to reset.
// buffSize: Size of the buffer to reset.

void ResetBuffer(char buff[], DWORD buffSize);

//---------------------------------------------------------------------------

//NumDataPkts():
//This function is used to determine the number of Bluetooth HCI pakcets
//contained in a string of characters.  It also determines if all of the
//packets are complete or if a packet was segmented during the read.
//
//Parameters buff[]: Buffer in which the data is contained.
// numBytesRead: Number of bytes contained in the buffer
// &numPkts: Address of an integer which contains the
// number of data packets contained in the
// buffer.
// *pLastPkt: Pointer to the first byte of the last packet
// contained in the buffer.
// &dataError: Returns true if data is incomplete
//
//Return: (bool) True if a the buffer contains a segmented packet.

bool NumDataPkts(char buff[], DWORD numBytesRead, DWORD &numPkts, char *pLastPkt, bool &dataError);

//---------------------------------------------------------------------------

//TransmitTestData():
//This function prepares and transmits or retransmits the test data.
//
//Parameters connHndl[]: Two Byte connection handle of the destination
// device.
// fileSize: SMALL_FILE - Send small file
// LARGE_FILE - Send large file
// RETRANSMIT - Retransmit received data
// CONTINUE_TRANSMISSION - Continue with a
// transmission that is currently in progress.
// tDataSize: Size of test data received. (only used when
// retransmitting data)

void TransmitTestData(char connHndl[], int fileSize, unsigned int tDataSize);

//---------------------------------------------------------------------------

//ResetDisplay():
//This function resets the labels and controls on the Graphical User Interface.

void ResetDisplay();

//---------------------------------------------------------------------------

//ClearStatus():
//This function resets the Status labels on the Graphical User Interface.

void ClearStatus();
```

```
//---------------------------------------------------------------------------


//Disconnecting():
//This function resets the Status labels and other labels related to an ACL
//connection on the Graphical User Interface.

void Disconnecting();

//---------------------------------------------------------------------------

#define ZERO_OPCODE 0x0000

extern PACKAGE TBT_net *BT_net;
//---------------------------------------------------------------------------
#endif
```

C.2 BTTest3_1.cpp

```
//BTTest3_1.cpp
//Written on 24OCT02 by MIDN Kenneth J. Hoover
//Last updated 31MAR03
//
//This program sets up and creates a Bluetooth wireless link through which
//data is sent via a virtual serial link between two computers, each running
//this program.  It then runs a test program used to test data transmision times
//between a single master and slave using two differnt size text files.  This
//program is written for use with the Ericsson ROK 101 008 Bluetooth baseband
//controller and tranciever.

//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "BTTest3_1.h"

//---------------------------------------------------------------------------

#pragma package(smart_init)
#pragma resource "*.dfm"
TBT_net *BT_net;



//---------------------------------------------------------------------------

__fastcall TBT_net::TBT_net(TComponent* Owner)
        : TForm(Owner)
{
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::FormCreate(TObject *Sender){

        //INITIALIZATION OF THE SERIAL PORT (COMM 1)
        C1 = CreateFile(Port[0], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
```

```
        FILE_FLAG_OVERLAPPED, NULL);
        GetCommState(C1, &initDCB); //Save inital DCB in order to reset Comm 1 on exit of program
        myDCB = initDCB; //Initailize myDCB to current Comm Port settings
        myDCB.BaudRate = BaudRate[3];
        myDCB.fBinary = TRUE;
        myDCB.fParity = FALSE; //No parity check
        myDCB.fOutxCtsFlow = TRUE; //should be true ********
        myDCB.fOutxDsrFlow = FALSE;
        myDCB.fDtrControl = DTR_CONTROL_DISABLE;
        myDCB.fDsrSensitivity = FALSE;
        myDCB.fOutX = FALSE;
        myDCB.fInX = FALSE;
        myDCB.fNull = FALSE;
        myDCB.fRtsControl = RTS_CONTROL_TOGGLE;   //should be toggle *****
        myDCB.ByteSize = 8;
        myDCB.Parity = NOPARITY;
        myDCB.StopBits = ONESTOPBIT;
        SetCommState(C1, &myDCB);

        //COMM TIMEOUTS
        GetCommTimeouts(C1, &CT1);
        GetCommTimeouts(C1, &CTinit);
        CT1 = CTinit; //Initialize CT1 to current comm timeout settings.
        CT1.ReadIntervalTimeout = 10; //Timeout on read if delay is greater than 100ms
//between chars.
        CT1.ReadTotalTimeoutMultiplier = 0; //Timeout on read if total read time is greater
        CT1.ReadTotalTimeoutConstant = 50; //than 200ms
        SetCommTimeouts(C1,&CT1);

        //INITIALIZE BT MODULE
        pktSize = Reset(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//INITIALIZE DATA VARIABLES
ResetBuffer(inBuff, inBuffSize);
host.sizeACLBuff = 400;
host.sizeSCOBuff = 400;
host.numACLBuff = 1;
host.numSCOBuff = 1;

//GUI SETUP
ResetDisplay();

}

//---------------------------------------------------------------------------

void __fastcall TBT_net::FormDestroy(TObject *Sender){

        //RESET COMM STATES BEFORE CLOSING PROGRAM
        SetCommState(C1, &initDCB);
        SetCommTimeouts(C1, &CTinit);
        CloseHandle(&C1);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::Timer1Timer(TObject *Sender){
```

```
//****** VARIABLES ******
//-----Buffer Variables -----
    DWORD numBytesRead, numPkts = 0;
char *pPkt = inBuff, *pLastPkt = inBuff;
static char *pReadPt = inBuff;
static unsigned int buffAvail = inBuffSize;

//----- Packet and Data Control Variables
bool segmentedPkt = false;
unsigned int pktSize, mask = 0x000000FF, amtProcData = 0;
static unsigned int segSize = 0;

//----- RSSI Variables
static unsigned int cycle = 0;
const unsigned int rssiFreq = 12;

//----- Display Variables -----
char dataDisp[255] = {NULL};

//----- Temporary Variables -----
char tempAry[450] = {0};
unsigned int temp1, temp2;

//******** TRANSMISSION TEST VARIABLES *********
static unsigned int testDataReceived = 0, numTestsRan = 0;
static unsigned int l2capPktSize, storedL2capData = 0;
short fileType;
unsigned int slaveIndex = 0, n; //loop control variable
char aclFlags, flagMask = 0xF0;
Timer cTimer;
char bytesOfInt[4];
unsigned int processedPkts = 0, curProcPkts = 0, t_proc = 0, errors = 0;
DWORD newBytesRead = 0, loopLimit = 200, loopCount = 0;
bool timeOut = false, slaveFound = false;
static char tempConnHndl;

//                  DETERMINE IF DATA HAS ARRIVED AND PROCESS IT
//*********************************************************************************

    Timer1->Enabled = false;//Disable timer while processing data

    ReadFile(C1, pReadPt, buffAvail, &numBytesRead, &OVL);//read data from COMM 1

    if(numBytesRead != 0 ){ //If data is present ...  If not end.
numBytesRead = numBytesRead + segSize;
segmentedPkt = NumDataPkts(inBuff, numBytesRead, numPkts, pLastPkt, timeOut); //Does a partial packet exist in the

for(unsigned int p=0; p<numPkts; p++){ //Process data from all available data packets

switch(*pPkt){ //switch UART header

case UART_EVENT_PACKET:
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
FindEvent(pPkt);
break;
```

```
case UART_ACL_DATA_PACKET:
ClearStatus();
LStatus1->Caption = "Data Received";

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;
LStatus2->Caption = pktSize;


// CHECK THE MOST SIGNIFICANT BYTE OF THE L2CAP CHANNEL ID
// THIS BYTE TELLS WHAT TYPE OF DATA IS BEING RECIEVED:
// TEST DATA OR MESSAGE DATA.  TEST DATA MUST BE PROCESSED
// AND RETRANSMITTED, BUT MESSAGE DATA IS DISPLAYED TO THE
// USER AND THEN DISCARDED.
//*****************************************************************

// IF THE ACL FLAGS INDICATE A CONTINUING PACKET THE PAKCET MUST
// TRANSMISSION TEST DATA.  SINCE CONTINUING PACKETS DO NOT
// CONTAIN L2CAP HEADER INFORMATION THE l2capCIDmsb VARIABLE
// MUST BE MANUALLY SET TO TEST_DATA_MSB.
//*****************************************************************

switch (*(pPkt+8)){ //Switch L2CAP CID msb

case MESSAGE_MSB:

//########### ACL Data Display for ONLY one BT Slave ############
for(unsigned int n=9; n<pktSize; n++)
dataDisp[n-9] = *(pPkt+n); //Move data from data pkt to a string for display
dataDisp[pktSize-9] = NULL; //Add a null char to end for string output
LRx->Caption = dataDisp;
break;

case TEST_DATA_MSB:
//IF EXECUTION REACHES THIS POINT AT LEAST ONE COMPLETE PKT EXISTS

/*****************************************************************************************
IF TEST DATA IS BEING RECIEVED EXECUTION WILL REMAIN IN THIS LOOP UNTIL THE WHOLE
TEST DATA FILE HAS BEEN READ INTO THE INPUT BUFFER.  DURING EACH ITERATION THE DATA
READ WILL BE COUNTED AND COMPARED TO THE TOTAL SIZE OF THE PAKCET.  IT WILL ALSO BE
FOR OTHER TYPES OF DATA WHICH MAY BE MIXED IN AND STORE THAT DATA SEPARATELY.  ONCE
THE COMPLETE FILE IS READ THE DATA WILL BE PROCESSED.  THIS WHOLE PROCESS WILL BE
TIMED AND RETURNED TO THE TESTING DEVICE AS Tproc (PROCESSING TIME).
*****************************************************************************************/

tempConnHndl = *(pPkt+1) & mask; //Store the connection handle so we know who sent the test data

do{ // continue loop until the complete test file has been read
for(curProcPkts=processedPkts; curProcPkts<numPkts; curProcPkts++){//loop through all counted data pkts
ClearStatus();
LStatus1->Caption = "Receiving Test Data";
```

```
//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask;//Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask;//into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//* IS THIS THE FIRST ACL PACKET OF A LARGE L2CAP PACKET? *
aclFlags = (*(pPkt+2) & flagMask);// Extract ACL flags

switch(aclFlags){
case FIRST_PKT_P2P:
//YES THIS IS THE FIRST OF MANY

//Subtract 9 bytes from the total pkt size
//5 bytes for the ACL pkt header and 4 bytes
//for the L2CAP pkt header.
testDataReceived = pktSize - 5 - 4;
//Find L2CAP Pkt Size
temp1 = (static_cast<unsigned int>(*(pPkt+5))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+6))) & mask; //into an int variable.
temp2 = temp2 << 8;
l2capPktSize = temp1 + temp2; //Size of exptected l2cap pkt

LStatus2->Caption = "L2CAP Packet Size:";
LStatus3->Caption = l2capPktSize;
break;

case CONT_PKT_P2P:
//NO THIS IS A CONTINUING PACKET TO ADD TO A PREVIOUSLY
//RECIEVED L2CAP FRAGMENT.

//Subtract only the 5 bytes for the ACL pkt header.
testDataReceived = testDataReceived + pktSize - 5;
break;

default:
LTestStatus->Caption = "Unknown ACL Data Packet Flags";
if(*pPkt == UART_EVENT_PACKET){//if an event packet was recieved store it for processing later
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
ClearStatus();
LStatus1->Caption = "Event Received During Test";
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
FindEvent(pPkt);
}//end if

}//end switch aclFlags
//Finished with present packet...move pointer to next data packet
pPkt = pPkt + pktSize;
amtProcData = amtProcData + pktSize;
pktSize = 0;
                                                        LPktNum->Caption = curProcPkts;
}//end for
```

```
processedPkts = numPkts; // update the number of processed packets

//WILL THERE BE MORE DATA COMING?
if(testDataReceived < l2capPktSize){//more data is coming
do{
//If there is a segmented packet compute the amount of data that does exist
if(segmentedPkt)
segSize = numBytesRead - amtProcData;
else{
segSize = 0;
LStatus2->Caption = " ";
}//end else

pReadPt = inBuff + amtProcData + segSize; //move the read point to the end of the data in the buffer
buffAvail = inBuffSize - amtProcData; //How much of the buffer is still free
ReadFile(C1, pReadPt, buffAvail, &newBytesRead, &OVL); //read data from COMM 1

//**** DEBUGING CODE ***
if(newBytesRead == 0){
loopCount++;
if(loopCount == loopLimit)
timeOut = true;
BT_net->LTestStatus->Caption = "Waiting for data. No Bytes Read";
}//end if
//**********************
else{
newBytesRead = newBytesRead + segSize; //Add segment size to newBytesRead so that a complete packet is sent to NumD
segmentedPkt = NumDataPkts(pPkt, newBytesRead, numPkts, pLastPkt, timeOut);
//Does a partial packet exist in the buffer
numBytesRead = numBytesRead + newBytesRead - segSize; //add the new bytes read to the number of bytes already in th
numPkts = numPkts + processedPkts; //add the new number of packets to the number of packets already in the buffer
}//end else

}while(segmentedPkt && !timeOut); //continue loop until a segmented pkt no longer exists
}//end if more data is coming

}while((testDataReceived<l2capPktSize) && !timeOut); //Continue loop until the whole Test file (L2CAP
//pkt) is recieved or a timeout occurs.

//*************************** PROCESS RECIEVED TEST DATA **************************************

if ((testDataReceived==l2capPktSize) || timeOut){
//NO MORE DATA IS COMING OR A TIMEOUT HAS OCCURED
pPkt = inBuff;//reset the pkt pointer to point to the beginning of the data
while(*pPkt == UART_EVENT_PACKET){//make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

if(localDevice.conductingTest){
//LOCAL DEVICE IS CONDUCTING THE TEST

//Stop Timer. The LSB of the L2CAP CID contains a
//0 for a small file and a 1 for a large file.
fileType = static_cast<short>(*(pPkt+7));
```

```
//Read the processing time used by the other computer (1st and 2nd data bytes in the test data pkt)
temp1 = (static_cast<unsigned int>(*(pPkt+9))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+10))) & mask; //into an int variable.
temp2 = temp2 << 8;
t_proc = temp1 + temp2; //Processing time used on remote computer

if(t_proc == TIME_OUT_REMOTE_COMP){ //A timeout occured on remote computer
LTestStatus->Caption = "Timeout occured on remote computer";
cTest.endReceive(fileType, t_proc, errors); //end test
}//end if
else {
if(timeOut){ //A timeout occured on this computer
LTestStatus->Caption = "Timeout occured on this computer";
cTest.endReceive(fileType, TIME_OUT_LOCAL_COMP, errors); //end test
}//end if
else{
cTimer.startTimer();

//STORE RECIEVED DATA
storedL2capData = 0;
for(unsigned int b=0; b<numPkts; b++){ //loop through all packets and save them for retransmission
while(*pPkt == UART_EVENT_PACKET){ //make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//STORE THIS DATA
for (n=0; n<pktSize-5; n++)
testDataBuff[n+storedL2capData] = *(pPkt+n+5);
storedL2capData = storedL2capData + pktSize - 5;  //Account for ACL pkt header
pPkt = pPkt + pktSize;
}//end for

//ALL DATA HAS BEEN SAVED..RESET pPkt and pktSize
pPkt = inBuff;
pktSize = 0;

//CHECK FOR ERRORS
for(unsigned int k=6; k<l2capPktSize+4; k++){
if(cTest.dataLong[k] != testDataBuff[k]) //If the saved test data does not
errors++; //equal the transmitted data then an
}//end for //error occured.

t_proc = t_proc + static_cast<unsigned int>(cTimer.stopTimer()*1000);
cTest.endReceive(fileType, static_cast<float>(t_proc)/1000, errors); //end this test
```

```
}//end else
}//end else

numTestsRan++;
LCurTrans->Caption = numTestsRan;

//IF MORE TESTS NEED TO BE RUN...
//TRANSMIT MORE DATA
if (numTestsRan < numTestTrans){
TransmitTestData(slave[0].connectHndl, fileType, 0);
LTestStatus->Caption = "Transmit next pkt";
}//end if
else{
readRssi = true;
numTestsRan = 0;
LTestStatus->Caption = "Finished!";
MessageBox(0, "Test Finished", 0, MB_OK);
}//end else

}//end if local device conducting test
else{
//LOCAL DEVICE IS NOT CONDUCTING THE TEST
cTimer.startTimer();

storedL2capData = 0;
for(unsigned int b=0; b<numPkts; b++){ //loop through all packets and save them for retransmission
while(*pPkt == UART_EVENT_PACKET){ //make sure that the pointer is pointing to the first ACL pkt
//The third byte in an event pkt gives the total
//length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//to compute total packet size.
pktSize = (static_cast<unsigned int>(*(pPkt+2)) & mask) + 3;
pPkt = pPkt + pktSize;
}//end while (pPkt == UART_EVENT_PACKET)

//The 4th and 5th bytes in an ACL pkt give the total
//length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//to compute total packet size.
temp1 = (static_cast<unsigned int>(*(pPkt+3))) & mask; //Convert 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pPkt+4))) & mask; //into an int variable.
temp2 = temp2 << 8;
pktSize = temp2 + temp1 + 5;

//STORE THIS DATA
for (n=0; n<pktSize-5; n++)
testDataBuff[n+storedL2capData] = *(pPkt+n+5);
storedL2capData = storedL2capData + pktSize - 5;  //Account for ACL pkt header
pPkt = pPkt + pktSize;

if(timeOut) //If a timeout has occured only store 1st ACL pkt
b = numPkts;
}//end for

//ALL DATA HAS BEEN PROCESSED..RESET pPkt and pktSize
pPkt = inBuff;
pktSize = 0;

if(t_proc == TIME_OUT_REMOTE_COMP){ //A timeout occured on remote computer
```

```
LTestStatus->Caption = "Timeout occured on remote computer";
cTest.endReceive(fileType, t_proc, errors); //end test
}//end if
else{
if(timeOut) //If a timeout has occured set t_proc to 0xFFFF
t_proc = TIME_OUT_REMOTE_COMP;
else{
if(t_proc == 0x5442){ //t_proc still contains "BT" this is the first transmission
//convert the float (seconds) value to an unsigned int (milliseconds) value for transmission
t_proc = static_cast<unsigned int>(cTimer.stopTimer()*1000);
}//end if
else{ //This file has been transmitted before add to the past t_proc
//convert the float (seconds) value to an unsigned int (milliseconds) value for transmission
t_proc = t_proc + static_cast<unsigned int>(cTimer.stopTimer()*1000);
}//end else
}//end else
}//end else

Int2Char(t_proc, bytesOfInt);
testDataBuff[4] = bytesOfInt[0]; //The first 4 bytes of the test data are the L2CAP header
testDataBuff[5] = bytesOfInt[1]; //store the value of t_proc in bytes 5 and 6.

//Retransmii
if(localDevice.master){ //Is the local device the master?
//Yes, must transfer data from one slave to the other
do{ //From which slave was data just received.
if (tempConnHndl == slave[slaveIndex].connectHndl[0])
slaveFound = true; //Connection Hndl matchers
else //Slave not found yet
slaveIndex++; //keep looking
}while(!slaveFound && (slaveIndex<2));

switch(slaveIndex){
case 0: //Slave 1 sent the test data
slaveIndex = 1; //Transmit data to slave 2

//Retransmit data
if(timeOut) //Let the transmit test function know if a timeout occured
TransmitTestData(slave[slaveIndex].connectHndl, RETRANSMIT, 2);
else
TransmitTestData(slave[slaveIndex].connectHndl, RETRANSMIT, l2capPktSize);
break;
case 1: //Slave 2 sent the test data
slaveIndex = 0; //Transmit data to slave 1

//Retransmit data
if(timeOut) //Let the transmit test function know if a timeout occured
TransmitTestData(slave[slaveIndex].connectHndl, RETRANSMIT, 2);
else
TransmitTestData(slave[slaveIndex].connectHndl, RETRANSMIT, l2capPktSize);
break;
default: //Don't know who sent the data
MessageBox(0, "Unknown Connection Handle", 0, MB_OK);
}//end switch
}//end if
else{ //Local device is another slave retransmit to master
if(timeOut) //Let the transmit test function know if a timeout occured
TransmitTestData(slave[0].connectHndl, RETRANSMIT, 2);
else
```

```
TransmitTestData(slave[0].connectHndl, RETRANSMIT, l2capPktSize);
}//end else
}//end else local device is not conducting the test

//RESET VARIABLES
testDataReceived = 0;
l2capPktSize = 0;
numPkts = 0;
timeOut = false;
}//end if (testDataReceived == l2capPktSize)
else{
//SOMETHING SCREWED UP
LTestStatus->Caption = "SOMETHING SCREWED UP: Too much data received";
}//end else someting screwed up
break;
//**************************************************************************************************************


default:
LTestStatus->Caption = "Unknown L2CAP CID";
}//end switch L2CAP CID msb
}//end switch UART header

//Finished with present packet...move pointer to next data packet
pPkt = pPkt + pktSize;
amtProcData = amtProcData + pktSize;
pktSize = 0;
}//end for

//   IF A SEGMENTED DATA PACKET EXISTS MOVE EXISTING DATA
//   TO THE BEGINING OF THE BUFFER AND SET THE READ POINT
//   AT THE END OF THE PARTIAL DATA
//********************************************************************************

if(segmentedPkt){ //The last packet read was not complete

segSize = numBytesRead - amtProcData;
for(unsigned int c=0; c<segSize; c++) //Temporarily store incomplete pkt
tempAry[c] = *(pLastPkt+c);
ResetBuffer(inBuff, inBuffSize); //Reset inBuff
for(unsigned int c=0; c<segSize; c++) //Place incomplete pkt in the front
inBuff[c] = tempAry[c]; //of inBuff
pReadPt = inBuff + segSize; //Set the Read Point pointer to the end of the incomplete pkt
buffAvail = inBuffSize - segSize; //Adjust the buffer size to reflect the portion used
//by the incomplete pkt.  (Needed for reading from COM 1)
}//end if segmentedPkt

else{ //Last packet was complete
ResetBuffer(inBuff, inBuffSize); //Reset inBuff
pReadPt = inBuff; //Reset Read Point pointer
buffAvail = inBuffSize; //Reset available buffer size.
segSize = 0;
}//end else

}//end if data is present


// IF A CONNECTION EXISTS READ AND RECEIVE SIGNAL
// STRENGTH (RSSI) FOR THE EXISTING LINKS.
```

```
//*************************************************************************************
//############### THIS CODE MUST BE EDITED FOR USE WITH MORE THAN 2 DEVICES
//RSSI
if(readRssi&&(!segmentedPkt)){
if(slave[0].connection && localDevice.conductingTest){
if(cycle%rssiFreq == 0){
pktSize = ReadRSSI(slave[0].connectHndl, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
} //end if
cycle++;
}//end if
}//end if readRssi

Timer1->Enabled = true;//Data processing complete...re-enable timer
}


//----------------------------------------------------------------------------

void FindOpCode(char *pBuff){

//******* VARIABLES *******
//----- Control Variables
unsigned int opCode, mask = 0x000000FF;
static bool inquiryFilter; //Used to determine which filter needs to be set
   //(connection or inquiry) for each call to setEventFilter

//----- Display Variables -----
char opCodeDisp[7] = {'0','x'}, tempOpCode[4];
unsigned int x;

//----- Temporary Varialbes -----
unsigned int temp1, temp2;
//*************************

    // FIND THE OPCODE AND STORE IT IN AN INTEGER VARIABLE
// FOR USE IN A SWITCH STATEMENT.  THE OPCODE IS CONTAINED
// IN A DIFFERENT PART OF THE EVENT PACKET FOR A COMMAND COMPLETE
// EVENT AND A COMMAND STATUS EVENT.
//*************************************************************************************

    if (*(pBuff+1) == EV_COMMAND_COMPLETE){
    temp1 = static_cast<unsigned int>(*(pBuff+4)) & mask; //mask out all unwanted
    temp2 = static_cast<unsigned int>(*(pBuff+5)) & mask; //bytes.
    temp2 = temp2<<8; //shift the msbyte to its position
    opCode = temp1 + temp2; //combine the two bytes into one integer variable
    }//end if

    if (*(pBuff+1) == EV_COMMAND_STATUS){
        temp1 = static_cast<unsigned int>(*(pBuff+5)) & mask; //mask out all unwanted
    temp2 = static_cast<unsigned int>(*(pBuff+6)) & mask; //bytes.
    temp2 = temp2<<8; //shift the msbyte to its position
    opCode = temp1 + temp2; //combine the two bytes unsigned into one integer variable
    }//end if

// FIND THE RETURNED OPCODE AND EXEUTE ACCORDINGLY
//************************************************************************************

switch (opCode){
```

```
//####### NOTE: 1st Parameter in command complete event pkt is array index #6
//####### Status parameter for command status event is array index #3

// THE RESET COMMAND TRIGGERS A CHAIN OF SETUP COMMANDS
// TO PREPARE THE BT MODULE FOR USE.  EACH RESPECTIVE
// COMMAND COMPLETE EVENT LEADS TO THE EXECUTION OF THE
// NEXT COMMAND IN THE CHAIN ENDING WITH "WRITE PAGE
// TIMEOUT"  THE COMMAND COMPLETE EVENTS FOR ALL OF THE
// SETUP COMMMANDS ARE LISTED HERE IN THIER ORDER OF EXECUTION
// BEGINING WITH THE RESET COMMAND.
//*******************************************************************************

//----------- RESET -----------
case HCI_RESET:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If reset sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Device Reset";

//ISSUE NEXT SETUP COMMAND
pktSize = ReadBD_ADDR(outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called
inquiryFilter = false;
}//end if
        else { //If host buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Reset FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- READ BLUETOOTH DEVICE ADDRESS ----------
case HCI_READ_BD_ADDR:

//Extract the BD ADDR parameter and convert it to a string of
        //ASCII characters for display to the user.
        for (unsigned int n=0; n<6; n++)   //Extract 6 byte address from
localDevice.bd_addr[n] = *(pBuff+7+n);//the event packet

//Convert each nibble of the address to an ASCII
        //char representative of the hex digit contained in each nibble.
        for (unsigned int n=0; n<12; n=n+2){
x=n/2;
            nibbles(localDevice.bd_addr[5-x], localDevice.addrDisp[n+2], localDevice.addrDisp[n+3]);
        }//end for

        localDevice.addrDisp[14] = NULL; //Append a null for string display.
        BT_net->LBD_ADDR_ld->Caption = localDevice.addrDisp;

//ISSUE NEXT SETUP COMMAND
pktSize = ReadBufferSize(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
```

```
break;

//---------- READ BUFFER SIZE ----------
case HCI_READ_BUFFER_SIZE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If read buffer size sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Read Buffer Size Complete";

//EXTRACT DATA FROM PACKET
//ACL buff size
temp1 = static_cast<unsigned int>(*(pBuff+7)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+8)) & mask;
temp2 = temp2<<8;
hostCtrl.sizeACLBuff = temp1 + temp2;
//SCO buff size
hostCtrl.sizeSCOBuff = *(pBuff+9);
//number of ACL data buffers
temp1 = static_cast<unsigned int>(*(pBuff+10)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+11)) & mask;
temp2 = temp2<<8;
hostCtrl.numACLBuff = temp1 + temp2;
//number of ACL data buffers
temp1 = static_cast<unsigned int>(*(pBuff+12)) & mask;
temp2 = static_cast<unsigned int>(*(pBuff+13)) & mask;
temp2 = temp2<<8;
hostCtrl.numSCOBuff = temp1 + temp2;

//ISSUE NEXT SETUP COMMAND
pktSize = HostBufferSize(&host, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

/*//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(INQUIRY_RESULT, 0, outBuff);//set inquiry result filter
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = true;*/

}//end if
        else { //If read buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Read Buffer Size FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- HOST BUFFER SIZE ----------
case HCI_HOST_BUFFER_SIZE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If host buffer size sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Host Buffer Size Sent";

//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(INQUIRY_RESULT, 0, outBuff); //set inquiry result filter
```

```
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = true;
}//end if
        else { //If host buffer size failed
ClearStatus();
BT_net->LStatus1->Caption = "Host Buffer Size FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- SET EVENT FILTER ----------
case HCI_SET_EVENT_FILTER:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If set event filter sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Event Filter Set";

//ISSUE NEXT SETUP COMMAND (depends on which filter was last set)

if (inquiryFilter){ //was the inquiry filter the last one set?
pktSize = WriteScanEnable(3, outBuff); //Setting WSE to 3 enables all scans
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if inquiry filter

else{ //the connection filter was the last filter set
pktSize = WriteConnAcceptTimeout(0x2000, outBuff); //0x2000*625us = 5.12s
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end else inquiry filter
}//end if
        else { //If set event filter failed
ClearStatus();
BT_net->LStatus1->Caption = "Set Event Filter FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
        }//end else

break;

//---------- WRITE SCAN ENABLE ----------
case HCI_WRITE_SCAN_ENABLE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write scan sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Write Scan Enable Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = WriteVoiceSetting(0x0060, outBuff); //0x0060 sets 16bit 2's complement input type
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if
        else { //If write scan failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Scan Enable FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else
```

```
break;

//---------- WRITE VOICE SETTING ----------
case HCI_WRITE_VOICE_SETTING:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write voice setting sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Write Voice Setting Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = WriteAuthEnable(0x00, outBuff); //0x00 Disables authentication
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

}//end if
        else { //If write voice setting failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Voice Setting FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- WRITE AUTHENTICATIO ENABLE ----------
case HCI_WRITE_AUTHENTICATION_ENABLE:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write authentication enable sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Write Authentication Enable Complete";

//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(CONNECTION_SETUP, AA_ON_RS_DISABLED, outBuff);
//set connection setup filter and auto accept off
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = false;



}//end if
        else { //If write authentication enable failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Authentication Enable FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

        //---------- SET UART BAUD RATE ----------
case HCI_SET_UART_BAUD_RATE:
                //WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If set uart baud rate sucessful
ClearStatus();
BT_net->LStatus1->Caption = "UART Baud Rate Set to 460.8kbps";

                        //Set Baud Rate on Comm 1  to 460.8kbps
                        myDCB.BaudRate = CBR_115200;
```

```
                    SetCommState(C1, &myDCB);


//ISSUE NEXT SETUP COMMAND
pktSize = SetEventFilter(CONNECTION_SETUP, AA_ON_RS_DISABLED, outBuff);
//set connection setup filter and auto accept off
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

//Because the SetEventFilter command will be called twice durring
//setup it is necessary to keep track of which filter is being set
//each time this command is called.
inquiryFilter = false;
}//end if
else { //If set uart baud rate failed
ClearStatus();
BT_net->LStatus1->Caption = "Set UART Baud Rate FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));
}//end else
break;

        //---------- WRITE CONNECTION ACCEPT TIMOUT ----------
case HCI_WRITE_CONNECTION_ACCEPT_TIMEOUT:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write connection accept timeout sucessful
ClearStatus();
BT_net->LStatus1->Caption = "Connection Accept Timeout Written";

//ISSUE NEXT SETUP COMMAND
pktSize = WritePageTimeout(0x3000, outBuff); //0x3000*625us = 7.68s
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end if
        else { //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Connection Accept Timeout FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- WRITE PAGE TIMEOUT (END OF SETUP COMMANDS) ----------
case HCI_WRITE_PAGE_TIMEOUT:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If write page timeout sucessfull
ClearStatus();
BT_net->LStatus1->Caption = "Page Timeout Written";
//If execution makes it to this point the setup is successful.
BT_net->LStatus2->Caption = "Setup Completed Successfully";
}//end if
        else { //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Write Page Timeout FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
        }//end else

break;

//---------- INQUIRY STATUS ----------
case HCI_INQUIRY:
```

```
//Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Conducting Inquiry";
}//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Inquiry Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;

//--------- CREATE CONNECTION STATUS ---------
case HCI_CREATE_CONNECTION:

        //Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Creating Connection";
}//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Create Connection Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;

//---- ACCEPT CONNECTIO REQUEST STATUS (NOT NECCESSARY BECAUSE THE AUTO ACCEPT ----
// ---- FLAG IS CURRENTLY SET ON THE CONNECTION FILTER ----
case HCI_ACCEPT_CONNECTION_REQUEST:

//Only a Command Status event will reach this block of code
        //so check status at array index #3
        if (*(pBuff+3) == 0){
ClearStatus();
BT_net->LStatus1->Caption = "Accepting Connection Request";
}//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Accept Connection Request Command Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
}//end else
break;

//---------- WRITE LINK SUPERVISION TIMOUT ----------
case HCI_WRITE_LINK_SUPERVISION_TIMEOUT:
/******INSERT CODE HERE*****/
break;

//---------- READ TRANSMIT POWER LEVEL ----------
case HCI_READ_TRANSMIT_POWER_LEVEL:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If read transmit power level sucessful

//Display Tx Power Level
//BT_net->LTxPower->Caption = (static_cast<unsigned int>(*(pBuff+9))) & mask;
```

```
//########### THIS CODE MUST BE EDITED FOR USE WTIH MORE THAN 2 DEVICES
//ISSUE READ RECIEVE SIGNAL STRENGTH INDICATOR (RSSI) COMMAND
pktSize = ReadRSSI(slave[0].connectHndl, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

}//end if
else { //If write connection accept timeout failed
ClearStatus();
BT_net->LStatus1->Caption = "Read Transmit Power Level FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
}//end else
break;

//---------- READ RECIEVE SIGNAL STRENGTH INDICATOR ----------
case HCI_READ_RSSI:
//WHAT IS THE STATUS OF THE COMMAND
if (*(pBuff+6) == 0){ //If read RSSI sucessful

//Display Receive signal strength
            localDevice.rssi = static_cast<short>(*(pBuff+9));
BT_net->LRSSI1->Caption = localDevice.rssi;

}//end if
else { //If read RSSI failed
ClearStatus();
BT_net->LStatus1->Caption = "Read RSSI FAILED";
BT_net->LStatus2->Caption = Error(*(pBuff+6));;
}//end else
break;

//-------------------------------------------------------------

case ZERO_OPCODE:
                //Do nothing
break;

default:
ClearStatus();
BT_net->LStatus1->Caption = "Unknown OpCode";

//Convert each nibble of the opcode to an ASCII
        //char representative of the hex digit contained in each nibble.
Int2Char(opCode, tempOpCode);
nibbles(tempOpCode[1], opCodeDisp[2], opCodeDisp[3]);
        nibbles(tempOpCode[0], opCodeDisp[4], opCodeDisp[5]);
BT_net->LStatus2->Caption = opCodeDisp;
}//end switch
}//end function

//----------------------------------------------------------------------------

void FindEvent(char *pBuff){

//******** VARIABLES ********
//------ Control Variables ------
char evt, tempConnHndl, curConnHndl[2];
unsigned char tempBD_ADDR[6];
    unsigned int x = 0, numHndls = 0, numResp = 0;
    unsigned int mask = 0x000000FF, matchingBytes = 0, slaveIndex = 0;
```

```
    bool slaveFound = false;
static unsigned int existingSlaves = 0;

//------ Display Variables ------
    //char addrDisp[15] = {'0','x'};
    char connectHndlDisp[7] = {'0','x'}, evtDisp[5]={'0','x'};
    unsigned int numCompPkts, errorCode;
//**************************

evt=*(pBuff+1);

// FIND THE RECEIVED EVENT AND EXECUTE ACCORDINGLY
//*****************************************************************************************

switch (evt){

//###### NOTE: 1st Parameter in event pkt is array index #3

case EV_COMMAND_COMPLETE:

FindOpCode(pBuff);
break;

case EV_COMMAND_STATUS:

FindOpCode(pBuff);
break;

case EV_INQUIRY_COMPLETE:
ClearStatus();
BT_net->LStatus1->Caption = "Inquiry Complete";
if (*(pBuff+3) == 0)
BT_net->LStatus2->Caption = "Completed Sucessfully";
else
BT_net->LStatus2->Caption = Error(*(pBuff+3));
break;

case EV_INQUIRY_RESULT:

//###### NOTE: MUST ADD COUNTER TO COUNT RESPONSES. ALL RESPONDING DEVICES MAY NOT BE
//######       CONTAINED IN ONE EVENT BUT MAY CREATE SEVERAL EVENTS THEREFORE YOU NEED
//######    TO KNOW HOW MANY DEVICES HAVE PREVIOUSLY RESPONED AND ARE ALREADY ACCOUNTED FOR.

numResp = (static_cast<unsigned int>(*(pBuff+3))) & mask;
BT_net->LInqRes->Caption = numResp;

//Read data for each of the slaves that responded
for (unsigned int n=0; n<numResp; n++){

//Read the BD_ADDR of the device with which
//a connection was just created.
for (unsigned int c=0; c<6; c++)
tempBD_ADDR[c] = *(pBuff+4+(6*n)+c);

do{ //With which slave was a connection just created.
for (unsigned int c=0; c<6; c++){
if (tempBD_ADDR[c] == slave[slaveIndex].bd_addr[c])
matchingBytes++;
}//end for
```

```
if (matchingBytes == 6) //All address bytes match
slaveFound = true;
else{ //Slave not found yet
slaveIndex++; //keep looking
matchingBytes = 0;
}//end else
}while (!slaveFound && (slaveIndex<7));

if (!slaveFound){ //A new slave responded

slaveIndex = existingSlaves; //Set the slave index to the number of existing slaves
//in order to add the new slave that responeded to the
//end of the array of slaves.

//Load the Bluetooth device address for each device that responded
//into the bd_addr array for each of the slave structures (up to 7
//slaves possible) and other device specific data using nested for loops
for (unsigned int c=0; c<6; c++)
slave[slaveIndex].bd_addr[c] = *(pBuff+4+(6*n)+c);

slave[slaveIndex].pgScnRepMode = *(pBuff+4+(6*numResp)+n);

slave[slaveIndex].pgScnPerMode = *(pBuff+4+(7*numResp)+n);

slave[slaveIndex].pgScnMode = *(pBuff+4+(8*numResp)+n);

for (unsigned int c=0; c<3; c++)
slave[slaveIndex].deviceClass[c] = *(pBuff+4+(9*numResp)+(3*n)+c);

for (unsigned int c=0; c<3; c++)
slave[slaveIndex].clkOffset[c] = *(pBuff+4+(12*numResp)+(2*n)+c);

//Convert each nibble of the address to an ASCII
//char representative of the hex digit contained in each nibble.
for (unsigned int c=0; c<12; c=c+2){
x=c/2;
nibbles(slave[slaveIndex].bd_addr[5-x], slave[slaveIndex].addrDisp[c+2], slave[slaveIndex].addrDisp[c+3]);
}//end for
slave[slaveIndex].addrDisp[14] = NULL; //Append a null for string display.

existingSlaves++; //increment the number of existing slaves
}//end if slave not found
}//end for

//**** These following lines of code are for only a three device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than three devices

BT_net->LBD_ADDR_rd1->Caption = slave[0].addrDisp;
BT_net->LBD_ADDR_rd2->Caption = slave[1].addrDisp;
break;

case EV_CONNECTION_COMPLETE:

//**** This Code is for connecting only two computers
//**** This must be modified before attempting to use with
//**** more than two computers in a piconet

if (*(pBuff+3) == 0){
```

```
ClearStatus();
BT_net->LStatus1->Caption = "Connection Completed";
BT_net->LStatus2->Caption = "Successfully";

if(localDevice.master){ //if the local device is the master there may be more than one
//remote slave
//Read the BD_ADDR of the device with which
//a connection was just created.
for (unsigned int n=0; n<6; n++)
tempBD_ADDR[n] = *(pBuff+6+n);

do{ //With which slave was a connection just created.
for (unsigned int n=0; n<6; n++){
if (tempBD_ADDR[n] == slave[slaveIndex].bd_addr[n])
matchingBytes++;
}//end for
if (matchingBytes == 6) //All address bytes match
slaveFound = true;
else{ //Slave not found yet
slaveIndex++; //keep looking
matchingBytes = 0;
}//end else
}while (!slaveFound && (slaveIndex<7));
}//end if
else
slaveIndex = 0;

//Load the information recieved about the
//remote device into a BT_DEVICE structure
slave[slaveIndex].connection = true; //Yes a connection exists

slave[slaveIndex].connectHndl[0] = *(pBuff+4);
slave[slaveIndex].connectHndl[1] = *(pBuff+5);
slave[slaveIndex].linkType = *(pBuff+12);
slave[slaveIndex].encryptMode = *(pBuff+13);

//Inform user of the type of link established
if (slave[slaveIndex].linkType == ACL_LINK)
BT_net->LLinkType->Caption = "ACL";
if (slave[slaveIndex].linkType == SCO_LINK)
BT_net->LLinkType->Caption = "SCO";

            //Convert each nibble of the Connection Handle to an ASCII
//char representative of the hex digit contained in each nibble.
for (unsigned int n=0; n<4; n=n+2){
x=n/2;
nibbles(slave[slaveIndex].connectHndl[1-x], connectHndlDisp[n+2], connectHndlDisp[n+3]);
}//end for
connectHndlDisp[6] = NULL; //Append a null for string display.

switch (slaveIndex){ //Display data in the correct box.
case 0:
BT_net->LConnectHndl1->Caption = connectHndlDisp;
break;
case 1:
BT_net->LConnectHndl2->Caption = connectHndlDisp;
break;
default:
ClearStatus();
```

```
BT_net->LStatus1->Caption = "Unknown Slave";
}//end switch

        }//end if
        else{
ClearStatus();
BT_net->LStatus1->Caption = "Connection Failed";
BT_net->LStatus2->Caption = Error(*(pBuff+3));
        }//end else
break;

case EV_NUM_COMPLETED_PACKETS:

//#### NOTE: THIS CODE IS WRITTEN FOR A SINGLE POINT TO POINT CONNECTION WHEN EDITING
//####   CODE TO HANDLE SEVERAL DEVICES THE CONNECTION HANDLES RETURNED MAY NEED
//####   TO BE CHECKED AND COMPARED.

//BEGIN EXTRACTING DATA FROM PACKET
numHndls = *(pBuff+3); //number of connection handles returned

if(numHndls > 1){
//Because of the nature of this test program (transmit...wait for remote
//computer to process data then re-transmit...there should be no more than
//one connection handle returned from a Number of Completed Packets event.
MessageBox(0, "ERROR: More than 1 connection handle\nreturned in NUM_COMP_PKTS event",0 , MB_OK);
}//end if
else{
curConnHndl[0] = *(pBuff+4); //Save the Connection handle of the recently completed pkts
curConnHndl[1] = *(pBuff+5); //and pass it to the TransmitTestData() function.
numCompPkts = *(pBuff+6); //Only need to read LSB of comp ACL pkts because the device buffer
//only holds 10 pkts
if(transmitInProgress){ //If there is a transmit in progress continue with it
TransmitTestData(curConnHndl, CONTINUE_TRANSMISSION, numCompPkts);
}//end if
}//end else
break;

case EV_DATA_BUFFER_OVERFLOW:
ClearStatus();
BT_net->LStatus1->Caption = "DATA BUFFER OVERFLOW";
break;

case EV_DISCONNECTION_COMPLETE:
tempConnHndl = *(pBuff+4);

do{ //From which slave was data just received.
if (tempConnHndl == slave[slaveIndex].connectHndl[0])
slaveFound = true; //Connection Hndl matchers
else //Slave not found yet
slaveIndex++; //keep looking
}while(!slaveFound && (slaveIndex<2));

ClearStatus();
BT_net->LStatus1->Caption = "Connection Terminated";
switch(slaveIndex){
case 0: //Slave 1 was disconnected
BT_net->LStatus2->Caption = "Slave 1";
BT_net->LConnectHndl1->Caption = " ";
break;
```

```
case 1: //Slave 2 was disconnected
BT_net->LStatus2->Caption = "Slave 2";
BT_net->LConnectHndl2->Caption = " ";
break;
default:
BT_net->LStatus2->Caption = "Unknown Connection Handle";
}//end switch

slave[slaveIndex].connection = false;
        break;

case EV_MAX_SLOTS_CHANGE:
BT_net->LStatus3->Caption = "Max LMP Slots Change";
break;

default:
ClearStatus();
BT_net->LStatus1->Caption = "Unknown Event";

//Convert each nibble of the event code to an ASCII
        //char representative of the hex digit contained in each nibble.
        nibbles(evt, evtDisp[2], evtDisp[3]);
BT_net->LStatus2->Caption = evtDisp;

}//end switch
}//end function

//---------------------------------------------------------------------------

void ResetBuffer(char buff[], DWORD buffSize){
for (unsigned int n=0; n<buffSize; n++)
buff[n] = NULL;
}//end function

//---------------------------------------------------------------------------

bool NumDataPkts(char buff[], DWORD numBytesRead, DWORD &numPkts, char *pLastPkt, bool &dataError){
DWORD paramLength = 0, availBuff = 0;
bool moreData = true, pktSeg = false;
unsigned int temp1, temp2, mask = 0x000000FF;
        unsigned int totalData = 0;

numPkts = 0;
pLastPkt = buff;
while(moreData){//while unaccounted for data exists do....
switch(*pLastPkt){//what type of data is this packet?

case UART_EVENT_PACKET:
//The third byte (2nd array element) in an event pkt gives
//the total length (in bytes of the parameters).  Add this
//number plus 3 (to account for the first 3 bytes)
//plus current amount of data to compute total data present.
paramLength = static_cast<unsigned int>(*(pLastPkt+2)) & mask;
totalData = totalData + 3 + paramLength;
numPkts++;
break;

case UART_ACL_DATA_PACKET:
//The 4th and 5th bytes (3rd and 4th array elemetnts) in an ACL pkt
```

```
//give the total length of the data (in bytes) contained in the pkt.
//Add this number plus 5 (to account for the first 5 bytes)
//plus current amount of data to compute total data present.
temp1 = (static_cast<unsigned int>(*(pLastPkt+3))) & mask; //Convet 2 separate bytes
temp2 = (static_cast<unsigned int>(*(pLastPkt+4)))& mask; //into an int variable.
temp2 = temp2 << 8;
paramLength = temp2 + temp1;
totalData = totalData + 5 + paramLength;
numPkts++;
break;

default: //Unrecognized pkt header...Error in received data
moreData = false;
dataError = true;
numPkts = numPkts+1;
BT_net->LStatus2->Caption = "Error in received data";
do{
//Clear the input buffer
//The data is useless since it contained an error
                                        availBuff = inBuffSize - numBytesRead;
ReadFile(C1, pLastPkt, availBuff, &numBytesRead, &OVL);//read data from COMM 1
}while(numBytesRead > 0);
                                return false;

}//end switch

if (numBytesRead == totalData){
moreData = false;//All data accounted for
pktSeg = false;
}//end if
else{//1st else
if (numBytesRead < totalData){//Some expected data is missing
moreData = false; //must read buffer again.
pktSeg = true;
numPkts = numPkts - 1; //Last pkt was a pkt segment...do not count it
BT_net->LStatus2->Caption = "SEGMENTED PACKET";
}//end if numBytesRead<totalData
else {//2nd else
pLastPkt = buff;
pLastPkt = pLastPkt + totalData; //More data remaining to be counted.
}//end 2nd else
}//end 1st else
}//end while
return pktSeg;
}//end function

//---------------------------------------------------------------------------

void __fastcall TBT_net::BInquiryClick(TObject *Sender){

char inqDuration = 48; //= 48*1.28s = 61.44s

    pktSize = Inquiry(inqDuration, 2, outBuff); //Returns after 2 inquiries are received.
    WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BBeginTestClick(TObject *Sender){
```

```
        //TEST SETUP
        AnsiString sDist;
short sDistSize;
char dist[15] = {NULL};

//Assign the value entered by the user into the Transmission Test class "cTest"
sDist = CBDist->Text;
sDistSize = sDist.Length();
for (short c=1; c<=sDistSize; c++)
dist[c-1] = sDist[c];
dist[sDistSize] = NULL;

        LStatus2->Caption = dist;

cTest.setDistance(dist, sDistSize);

        AnsiString sTop;

//Assign the value entered by the user into the Transmission Test class "cTest"
sTop = CBTopology->Text;
cTest.setTopology(sTop[1]);
        LStatus2->Caption = sTop[1];

//BEGIN TEST
cTest.newTest();
cTest.setLinkState(localDevice.addrDisp, slave[0].addrDisp, localDevice.rssi);
        readRssi = false;

//Begin by transmitting the small file
TransmitTestData(slave[0].connectHndl, SMALL_FILE, 0);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BSendClick(TObject *Sender){
DWORD dataLength;
AnsiString input;
unsigned int inputSize, messageRecipient;

if (!localDevice.master)
messageRecipient = 0;
else
messageRecipient = CBSlaveSelect->ItemIndex; //Who do you want to sent this message to?

if (messageRecipient == -1){
//No slave was selected to recieve the message
MessageBox(0, "Please select a slave to receive this message.", 0, MB_OK);
}//end if
else{
//Send the message to the selected slave
input = Message->Text;
Message->Text = ""; //Reset message block
inputSize = input.Length();
dataLength = inputSize + 4;
for (unsigned int c = 1; c <= inputSize; c++)
outBuff[c+3] = input[c];

//***** Adding L2CAP 4 byte header *****
//The first two bytes of the header are the L2CAP Packet length
```

```
//(i.e. The length of data to be transmitted may be greater than the HCI
//packet lenght)
//The second two bytes of the header is the Channel ID (CID) and can
//range from 0x0040 to 0xFFFF. For message traffic use 0x0300.

//***NOTE: Message is assumed to be <= 255 bytes long

outBuff[0] = static_cast<char>(inputSize); //L2CAP pkt length lsb
outBuff[1] = 0x00; //L2CAP pkt length msb
outBuff[2] = 0x00; //CID lsb
outBuff[3] = 0x03; //CID msb

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

pktSize = SendACLPkt(slave[messageRecipient].connectHndl, dataLength, FIRST_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end else
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BResetClick(TObject *Sender){

        //RESET BT MODULE
        pktSize = Reset(outBuff);
        WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

        //RESET CONNECTION CONRTOL VAR
        for (unsigned int n=0; n<7; n++)
                connection[n] = 0;

        //RESET DISPLAY
        ResetDisplay();
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BConnect1Click(TObject *Sender){
    //**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

pktSize = CreateConnection(&slave[0], localDevice.pktType, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBControlCloseUp(TObject *Sender){

localDevice.master = CBControl->ItemIndex;

if(localDevice.master){//if the local device will be the master...
//buttons and comboboxes
BInquiry->Enabled = true;
BConnect1->Enabled = true;
BConnect2->Enabled = true;
```

```
CBPktType->Enabled = true;
CBSlaveSelect->Enabled = true;
}//end if
else{
//buttons and comboboxes
BInquiry->Enabled = false;
BConnect1->Enabled = false;
BConnect2->Enabled = false;
CBPktType->Enabled = false;
CBSlaveSelect->Enabled = false;
}//end else

}

//---------------------------------------------------------------------------



void __fastcall TBT_net::CBPktTypeCloseUp(TObject *Sender){
unsigned int pktRef;

pktRef = CBPktType->ItemIndex;

//Assign the value entered by the user into the Transmission Test class "cTest"
switch(pktRef){
case 0:
localDevice.pktType = DM1;
cTest.setPktType(DM1);
break;
case 1:
localDevice.pktType = DH1;
cTest.setPktType(DH1);
break;
case 2:
localDevice.pktType = DM3;
cTest.setPktType(DM3);
break;
case 3:
localDevice.pktType = DH3;
cTest.setPktType(DH3);
break;
case 4:
localDevice.pktType = DM5;
cTest.setPktType(DM5);
break;
case 5:
localDevice.pktType = DH5;
cTest.setPktType(DH5);

}//end switch

//Disconnect all current connections.
for (unsigned int c=0; c<7; c++){
if (slave[c].connection && localDevice.master){
slave[c].connection = false;
Disconnecting();
LStatus1->Caption = "Disconnecting";
pktSize = Disconnect(slave[c].connectHndl, 0x13, outBuff); //user ended connection
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
```

```
}//end if
else
c = 7; //If no connection exists end loop (connections are established in sequential order
   //therefore if slave[0].connection does not exist there is no reason to complete the loop)
}//end for
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBTestRoleCloseUp(TObject *Sender){

localDevice.conductingTest = CBTestRole->ItemIndex;

if(localDevice.conductingTest){//if the local device will be conducting the test...
//buttons and comboboxes
BBeginTest->Enabled = true;
CBPktType->Enabled = true;
CBDist->Enabled = true;
CBTopology->Enabled = true;
CBNumTrans->Enabled = true;

}//end if
else{
if(!localDevice.master) //If the local device is the master the pkt type selection
//box is still needed.
CBPktType->Enabled = false;
//buttons and comboboxes
BBeginTest->Enabled = false;
CBDist->Enabled = false;
CBTopology->Enabled = false;
CBNumTrans->Enabled = false;

}//end else

}

//---------------------------------------------------------------------------

void __fastcall TBT_net::CBNumTransCloseUp(TObject *Sender){

//Assign twice the number of requested test
//transmissions to "numTestTrans" because two
//tests are run for each round of tests
numTestTrans = 2*(CBNumTrans->ItemIndex);
}

//---------------------------------------------------------------------------

void TransmitTestData(char connHndl[], int fileSize, unsigned int tDataSize){
//******* VARIABLES ******
static DWORD dataLength, dataLeft;
static char *pDataFile;
char pktLength[4], cidLsb;
bool firstACLpkt = false;
unsigned int numACLPktsSent = 0, n;  //Loop control variable
unsigned int buffersAvailable = hostCtrl.numACLBuff;
const unsigned int maxPktSize = hostCtrl.sizeACLBuff; //max size of ACL pkt alowable so as not to
//overflow the ACL Buffer of the local BT module.
unsigned int nextData2Send = 0; //contains the array index of the next data byte to send
```

```
//if nextData2Send == dataLength then all data has been sent
//************************

switch (fileSize){
case SMALL_FILE:
//SEND SMALL FILE
dataLength = cTest.sizeDataShort + 4;
pDataFile = cTest.dataShort;
cidLsb = LARGE_FILE;
transmitInProgress = true;
firstACLpkt = true;

BT_net->LTestStatus->Caption = "Transmitting Small File";
break;

case LARGE_FILE:
//SEND LARGE FILE
dataLength = cTest.sizeDataLong + 4;
pDataFile = cTest.dataLong;
cidLsb = SMALL_FILE;
transmitInProgress = true;
firstACLpkt = true;

BT_net->LTestStatus->Caption = "Transmitting Large File";
break;

case RETRANSMIT:
//LOCAL DEVICE IS NOT CONDUCTING TEST
//RETRANSMIT THE RECEIVED DATA

dataLength = tDataSize + 4;
pDataFile = testDataBuff;
firstACLpkt = true;

if(tDataSize == 2){ //A timeout has occured
*(pDataFile) = 0x02; //L2CAP pkt length lsb
*(pDataFile+1) = 0x00; //L2CAP pkt length msb
} //end if

BT_net->LTestStatus->Caption = "Retransmitting Received Data";
break;

case CONTINUE_TRANSMISSION:

//The new number of buffers available is equal
//to the number of packets completed.  This number
//is passed to TransmitTestData() as tDataSize
buffersAvailable = tDataSize;
break;

default:
BT_net->LTestStatus->Caption = "Unknown File type to Transmit";
}//end switch

if (dataLength <= maxPktSize){//If test data is smaller that the local device's
for (n=0; n<dataLength; n++) //ACL Buffer Size send it
outBuff[n] = *(pDataFile+n);

//***** Adding L2CAP 4 byte header *****
```

```
//The first two bytes of the header are the L2CAP Packet length
//(i.e. The length of data to be transmitted may be greater than the HCI
//packet lenght)
//The second two bytes of the header is the Channel ID (CID) and can
//range from 0x0040 to 0xFFFF.  For test data use 0xEE00

if ((fileSize!=RETRANSMIT) && (fileSize!=CONTINUE_TRANSMISSION)){
Int2Char(dataLength-4, pktLength);

outBuff[0] = pktLength[0]; //L2CAP pkt length lsb
outBuff[1] = pktLength[1]; //L2CAP pkt length msb
outBuff[2] = cidLsb; //CID lsb
outBuff[3] = TEST_DATA_MSB; //CID msb
}

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices
ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Packet is < 400 bytes";

pktSize = SendACLPkt(connHndl, dataLength, FIRST_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
if (fileSize != -1)
cTest.beginTransmit();

}//end if
else{ //The data is too large to send in one ACL Packet...it must be split up
while((nextData2Send != dataLength) && (numACLPktsSent != buffersAvailable)){

if (dataLength > (nextData2Send+maxPktSize)){ //enough data exists to send another "full" ACL pkt.
//Load up the output Buffer
for (n=0; n<maxPktSize; n++)
outBuff[n] = *(pDataFile+nextData2Send+n);

if (firstACLpkt){
//If this is the first ACL pkt an L2CAP header needs to be
//added and also the ACL PB flag must be set to first packet

firstACLpkt = false;

if ((fileSize!=RETRANSMIT) && (fileSize!=CONTINUE_TRANSMISSION)){
Int2Char(dataLength-4, pktLength);

outBuff[0] = pktLength[0]; //L2CAP pkt length lsb
outBuff[1] = pktLength[1]; //L2CAP pkt length msb
outBuff[2] = cidLsb; //CID lsb
outBuff[3] = TEST_DATA_MSB; //CID msb
}

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices
ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "1st ACL pkt of large L2CAP pkt";

pktSize = SendACLPkt(connHndl, maxPktSize, FIRST_PKT_P2P, outBuff);
```

```
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
if (fileSize != -1)
cTest.beginTransmit();

}//end if
else{
//If this is not the fist ACL pkt no L2CAP header needs to be added
//but the ACL PB flag must be set to continuing packet

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices
ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Continuing ACL pkt of large L2CAP pkt";

pktSize = SendACLPkt(connHndl, maxPktSize, CONT_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}//end else

nextData2Send = nextData2Send + maxPktSize; //increment nextData2Send
}//end if
else{ //There is not enough data left to send a "full" ACL pkt.
dataLeft = dataLength - nextData2Send;
//Load up the output Buffer
for (n=0; n<dataLeft; n++)
outBuff[n] = *(pDataFile+nextData2Send+n);

//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices
ClearStatus();
BT_net->LStatus1->Caption = "Transmitting Test Data";
BT_net->LStatus2->Caption = "Partially full ACL pkt for large L2CAP pkt";

pktSize = SendACLPkt(connHndl, dataLeft, CONT_PKT_P2P, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);

nextData2Send = dataLength; //end the while loop
}//end else

numACLPktsSent++; //Increment the number of ACL pkts sent
}//end while
}//end else

if(nextData2Send == dataLength){ //Transmission is complete
BT_net->LStatus2->Caption = "Finished Transmitting File";
transmitInProgress = false; //Transmission is finished

//Reset Static variables
nextData2Send = 0;
dataLength = 0;
dataLeft = 0;
}//end if
}

//---------------------------------------------------------------------------

void __fastcall TBT_net::BConnect2Click(TObject *Sender){
```

```
//**** These following lines of code for only a two device connection
//**** it must be deleted or edited for to be used in a piconet
//**** consisting of more than two devices

pktSize = CreateConnection(&slave[1], localDevice.pktType, outBuff);
WriteFile(C1, &outBuff, pktSize, &VAL, &OVL);
}

//---------------------------------------------------------------------------

void ResetDisplay(){

//Buttons
BT_net->BInquiry->Enabled = false;
BT_net->BConnect1->Enabled = false;
BT_net->BConnect2->Enabled = false;
BT_net->BBeginTest->Enabled = false;

//Combo Boxes
BT_net->CBPktType->Enabled = false;
BT_net->CBDist->Enabled = false;
BT_net->CBTopology->Enabled = false;
BT_net->CBNumTrans->Enabled = false;
BT_net->CBSlaveSelect->Enabled = false;

//Display Labels
BT_net->LBD_ADDR_rd1->Caption = " "; //BT_ADDR for remote device 1
BT_net->LBD_ADDR_rd2->Caption = " "; //BT_ADDR for remote device 2
BT_net->LBD_ADDR_ld->Caption = " "; //BT_ADDR for local device
BT_net->LConnectHndl1->Caption = " ";
BT_net->LConnectHndl2->Caption = " ";
BT_net->LInqRes->Caption = " ";
BT_net->LRSSI1->Caption = " ";
BT_net->LRSSI2->Caption = " ";
BT_net->LRx->Caption = " ";
BT_net->LTestStatus->Caption = " ";
BT_net->LStatus1->Caption = " ";
BT_net->LStatus2->Caption = " ";
BT_net->LStatus3->Caption = " ";
BT_net->LCurTrans->Caption = " ";
BT_net->LPktNum->Caption = " ";

}

//---------------------------------------------------------------------------

void ClearStatus(){

//Clear all of the status labels
BT_net->LStatus1->Caption = " ";
BT_net->LStatus2->Caption = " ";
BT_net->LStatus3->Caption = " ";
}

//---------------------------------------------------------------------------

void Disconnecting(){

ClearStatus();
```

```
BT_net->LConnectHndl1->Caption = " ";
BT_net->LConnectHndl2->Caption = " ";
BT_net->LRSSI1->Caption = " ";
BT_net->LRSSI2->Caption = " ";
BT_net->LTestStatus->Caption = " ";
BT_net->LCurTrans->Caption = " ";
BT_net->LPktNum->Caption = " ";
}
```