2001

Bluetooth software on Linux, wireless hand-held devices

Teck Khoon Low
Edith Cowan University

Edith Cowan University

Copyright Warning

BLUETOOTH SOFTWARE

ON LINUX, WIRELESS HAND-HELD DEVICES

Submitted by:

Low, Teck Khoon [redacted]

Submitted in partial fulfilment

of the Requirements for the Degree of Bachelor of Engineering

(Communication Systems) with Honours

School of Engineering and Mathematics

Faculty of Communications, Health & Science

Edith Cowan University

Perth, Western Australia

February 04, 2001

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

ABSTRACT

In order to enable existing computers (non-Bluetooth ready) to connect to a Bluetooth piconet, a Bluetooth hardware device comprising of the Radio antenna, the baseband and control circuit is used. The digital portion of this device is also known as a Host Controller, HC. In the traditional communication lingo, the Bluetooth Hardware functions, as the Data Communication Equipment (DCE) while the Host is the Data terminal Equipment (DTE).

This report discusses the theory and implementation of the communication protocol between the Host and the Host Controller, enabling communication between the computer and the Bluetooth hardware

DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

i incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

ii contain any material previously published or written by another person except where due reference is made in the text; or

iii contain any defamatory material.

Signature: _____

Date: 30/4/2001

ACKNOWLEDGEMENTS

Project Supervisor: Dr. Stefan Lachowicz.

Dr. Ma Zhongming

I would like to express my sincere gratitude to my project supervisors, Dr. Stefan Lachowicz at Edith Cowan University and Dr. Ma Zhongming in Singapore for their advice and guidance throughout the course of this project. Without which this project would not have been possible.

I would also like to thank Mr. Charlie Khoo Kai Hock who worked on a project closely related to my, for sharing with me with the vast amount of literature and ideas during the project.

My thanks are also due to Mr. Foo Chun Chong who was from Ericsson Telecommunications, who kindly and promptly clarified the many doubts I had on the Bluetooth protocol.

Lastly, I would like to thank my friends and family who had inspired me and given me unfaltering support throughout my education and career.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PROJECT DEFINITION

Aim

The aim of this project is to develop and simulate a Bluetooth Host Controller Interface link. The purpose of developing this software is to enable non-Bluetooth ready computers to connect to a Bluetooth network via Bluetooth Hardware (Radio). The objectives of this project are:

1. To explore one of the emerging wireless standard, Bluetooth.
2. To gain an understanding of the Bluetooth protocol stack.
3. To appreciate development using C programming on the Linux platform.

The software developed in this project would serve as a base on which the complete Bluetooth high level stack could be built. It could also be enhanced to act as a Bluetooth protocol analyser and tester as discussed in the future development section.

One of the original aims of this project is to develop a Graphic User Interface, GUI, for the wireless network protocol. However, after discussion with our project supervisor, the GUI was dropped from the project, as it does not aid in the study of the Bluetooth protocol.

Scope

The project comprises of both a research component and an implement component in the form of a null-modem simulation.

1. The first task is to familiarise with the development of software on the Linux platform. As the student learnt programming on the DOS platform, the different programming techniques need to be developed.
2. Explore the various wireless networking standards and to understand the various strengths and weakness.
3. Identify the existing Bluetooth resources available in the market.

4. In project 2, the initial plan was to implement a usage model using the LAN Access Profile. Due to the limited time in which to project was conducted (in the summer semester) and with no Bluetooth hardware device (that implements the radio and link management), only one portion of the usage model is simulated.

5. Understand how the computer communicates with the Bluetooth device through the Host Controller Interface (HCI).

6. Understanding the detail working of the HCI protocol.

CHAPTER 1
INTRODUCTION TO BLUETOOTH

Bluetooth History

Initially conceived by Ericsson in 1994, as a low power, low cost radio interface to communicate between mobile phones and their accessories. At the same time, several other companies such as Intel, IBM, Toshiba and Nokia Mobile Phones are also developing similar technology. These companies founded the Bluetooth Special Interest Group (SIG) in May 1998 to standardise and drive the development of Bluetooth. The Bluetooth SIG grew faster than any other wireless consortium, with 2164 members as of February 2001. (Information from the Official Bluetooth Web Site).

Figure 1 shows the SIG working structure. The Promoter companies (PM) includes 3Com, Ericsson, IBM, Intel, Lucent, Microsoft, Motorola, Nokia and Toshiba. These companies lead the effort of the SIG in promoting Bluetooth.

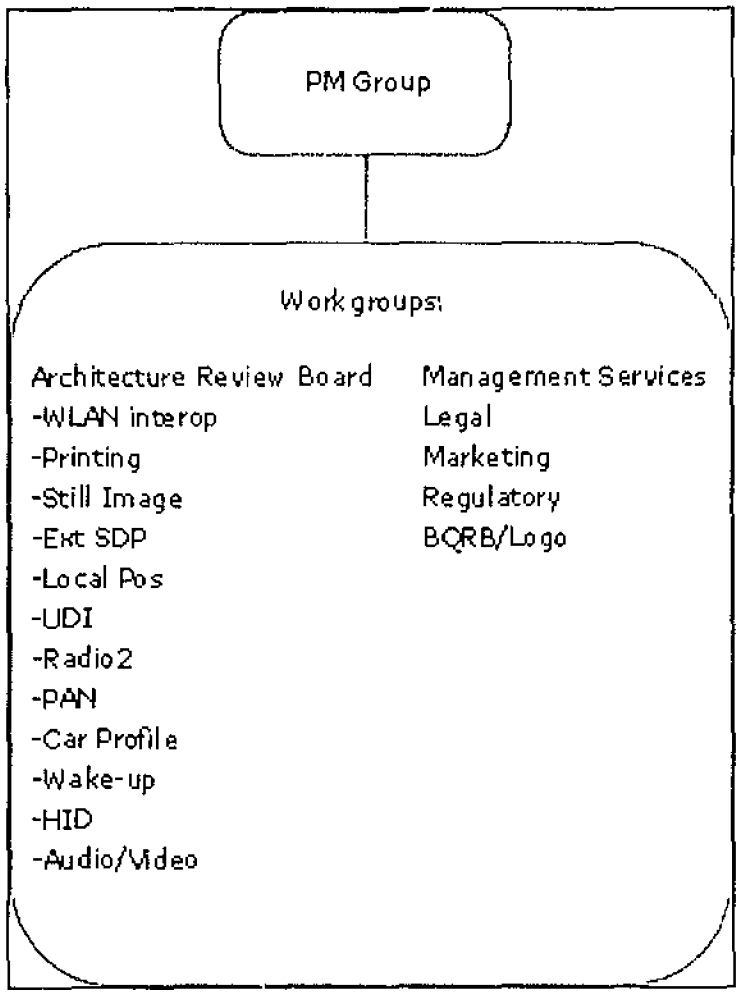


Figure 1 Bluetooth SIG working Groups

A Brief Overview of the Technology

Bluetooth is essentially a cable-replacement technology. The standard is based on Wireless LAN IEEE 802.11b (Canosa, J Nov 2000), however it differs from the Wireless LAN standard in that it calls for a small, cheap radio chip that can be plugged into computers, printers, mobile phones, etc. Bluetooth chip is designed to replace cables by taking the information normally carried by the cable, and transmitting it via the 2.4 GHZ ISM band to a receiver Bluetooth chip. The receiver will then give the information received to the computer, mobile phone or whatever it is attached to. The projected low cost of a Bluetooth chip at around US$5, it's small size and low power consumption, indicates that one could literally place anywhere.

Compared with other technology

Muller, N.J., (Sept 2000), compared Bluetooth to some of the present wireless standards. The following table is a summary of the features of each technology.

Table 1

Wireless technology comparison

Feature/ Function	Infrared	IEEE 802.11b
Connection Type	Infrared, narrow beam (30° angle or less)	Spread Spectrum (direct sequence DS, or Frequency Hopping FH)
Spectrum	Optical, 850nm	2.4 GHz ISM band
Transmission Power	20 dBm	20 dBm
Data Rate	Up to 16 Mbps using Very fast Infrared	1 to 2 Mbps using FH, 11Mbps using DS
Range	1 m	100m
Supported Stations	Two	Multiple devices per access point; Multiple access point per network
Voice Channels	One	None, Uses Voice over IP
Data Security	Short range and narrow angle of the IR provides a simple form of security. No other security at the link level	Authentication: challenge-response between access point and client via Wired Equivalent Policy Encryption: 40 bit standard

Feature / Function	Home RF	Bluetooth
Connection Type	Spread Spectrum Frequency Hopping FH	Spread Spectrum Frequency Hopping FH
Spectrum	2.4 GHz ISM band	2.4 GHz ISM band
Transmission Power	20 dBm	0dbm (normal) 20dBm (high power)
Data Rate	1 to 2 Mbps using FH	1 Mbps using FH
Range	100m	10m
Supported Stations	Up to 127 devices per network	Piconet support 1 master and 7 slaves. Multiple piconets are possible in the same area (i.e. Scatternet)
Voice Channels	Up to six	Up to three
Data Security	Blowfish encryption algorithm (over 1 trillion codes)	For authentication, a 128-bit key; For Encryption, key size is configurable between 8 and 128 bits

The requirements for a portable wireless device are reliable, convenience, ease to use and low power consumption. Compared with Infrared, Bluetooth do not require line of sight operation and has a longer range. Compared with the two other RF standards, Bluetooth devices consume less power and hence result in a longer battery life. The relative short distance of 10m is sufficient for most office spaces. In addition, the Bluetooth SIG is working on increasing the data rate to eventually allow multimedia support.

Existing Bluetooth Development

During the last two Quarters of the year 2000, several companies announced Bluetooth related products. Of these, there are 101 products qualified by the SIG (as listed in the official web site) (The Official Bluetooth web site). These include integrated circuits, software stacks, developer kits, host controllers, laptops with built in

Bluetooth radios, Mobile phone and accessories. Qualified Bluetooth hardware, (radio and Host Controller) includes devices from Digianswer, Ericsson, Xircomm, Cambridge Silicon Radio and others. While there are currently 13 qualified Bluetooth protocol stack, the only two stacks written for Linux, for connecting existing computers to the Bluetooth network are both still under development, these are from IBM and Axis communication.

Bluetooth Specifications

Bluetooth protocol structure comprises of seven defined layers, these are:

Radio

The Radio layer defines the requirements for a Bluetooth transceiver operating in the 2.4 GHz ISM band.

Baseband

The Baseband layer describes the specification of the Bluetooth Link Controller (LC) which carries out the baseband protocols and other low-level link routines.

LMP

The Link Manager Protocol (LMP) is used by the Link Managers for link set-up and control.

HCI

The Host Controller Interface (HCI) provides a command interface to the Baseband Link Controller and Link Manager, and access to hardware status and control registers. The HCI will be discussed in more detail in the next section.

L2CAP

Logical Link Control and Adaptation Protocol (L2CAP) supports higher level protocol multiplexing, packet segmentation and re-assembly, and the conveying of quality of service information.

RFCOMM

The RFCOMM protocol provides emulation of serial ports over the L2CAP protocol. The protocol is based on the ETSI standard TS 07.10, used for Global System for Mobile (GSM) communication devices.

SDP

The Service Discovery Protocol (SDP) provides a means for applications to discover which services are provided by or available through a Bluetooth device. It also allows applications to determine the characteristics of those available services.

Except for the Host Controller Interface, the other six protocols will not be covered, as they were discussed in relative detail in the Project One progress report. In addition to these seven protocols, Bluetooth also adopt several other protocols to handle higher layer data and voice as shown in the following figure.

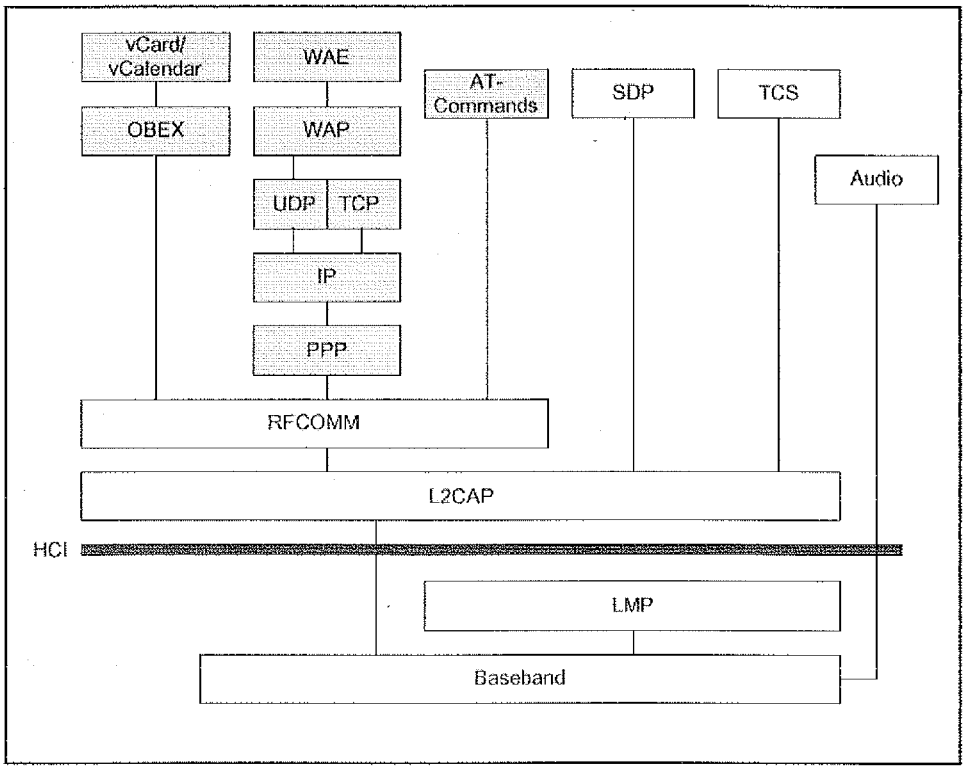


Figure 2 Bluetooth protocols

CHAPTER 2
THE HOST CONTROLLER INTERFACE

In order to enable existing computers (non-Bluetooth ready) to connect to a Bluetooth piconet, a Bluetooth hardware device comprising of the Radio antenna, the baseband and control circuit is used. The digital portion of this device is also known as a Host Controller, HC. In the traditional communication lingo, the Bluetooth Hardware functions, as the Data Communication Equipment (DCE) while the Host is the Data terminal Equipment (DTE). Figure 3 shows relation of the Host and the Bluetooth hardware.

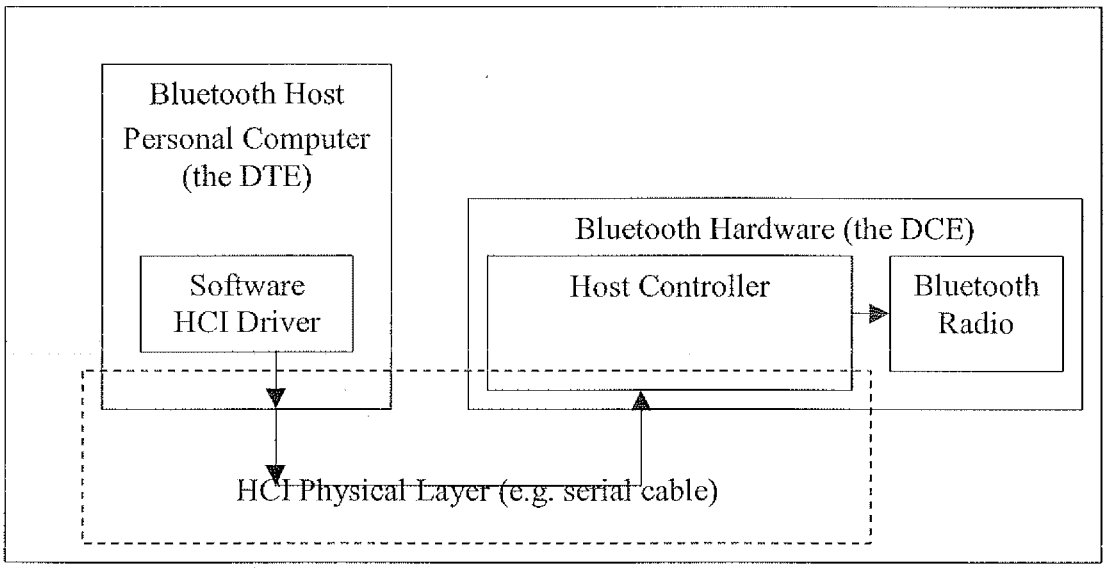


Figure 3 Host controller setup

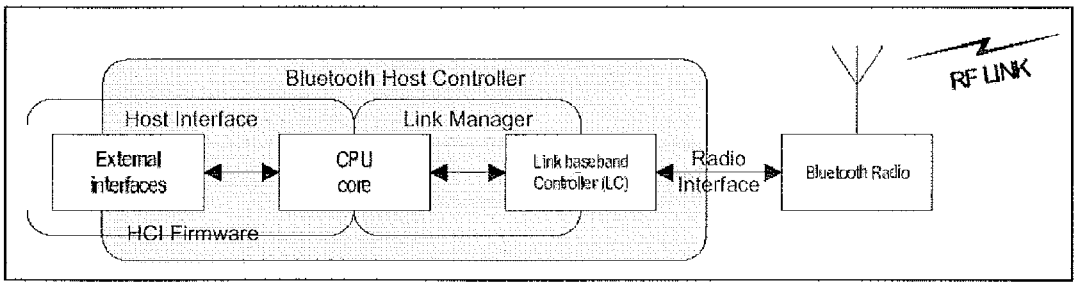


Figure 4 Bluetooth Device Breakdown

Figure 4 show the breakdown of the Bluetooth device by function. The Host Controller Interface serves as an interface between the (non-Bluetooth) DTE and the

Bluetooth Hardware (DCE). Essentially this interface provides a uniform method of accessing the Bluetooth baseband capabilities. The HCI exists across three sections, in the Host, Transport Layer, and Host Controller. Each of the sections has a different role to play in the HCI system.

Role of the HCI in an Embedded System

The HCI layer is not needed if the Bluetooth embedded solution is developed in the form of a chipset hosting the entire BT stack (that is, from Baseband to Application). This is because the HCI is used for applications having a clear divide (in terms of the different hosts for each of the parts) between the Host & the HC (Host Controller) parts of the stack.

However, it is still a good idea to implement HCI support. As it can be used to support some debug or test port to the chip. This debugging would require HCI to be implemented inside the chipset to interpret & respond to the debug commands from an external source. It is more convenient to perform testing and debugging of the upper layers (L2CAP, RFCOMM, SDP and the application) through the HCI than debugging via RF through the baseband.

The program described in section four can be used as a base to develop such a testing and debugging program.

HCI Driver

The term Host is used to refer to the HCI-enabled Software Unit. HCI Driver is located on the Host computer as a software entity. HCI link commands are used by the Host to communicate with the Host Controller; these commands provide the Host with the ability to control the link layer connections to other Bluetooth devices. In the reverse direction, the Host Controller sent HCI events to notify the Host when something occurs. When the Host discovers that an event has occurred it will then parse, the received event packet to determine which event occurred.

HCI Firmware

The term Host Controller is used to refer to the HCI-enabled Bluetooth device. HCI Firmware is located on the Host Controller, (that is, the actual Bluetooth hardware device). The HCI firmware implements the HCI Commands for the Bluetooth hardware by accessing baseband commands, link manager commands, hardware status registers, control registers, and event registers.

Host Controller Transport Layer

The HCI Driver and Firmware communicate via the Host Controller Transport Layer. This layer may comprise of several layers that exist between the HCI driver on the host system and the HCI firmware in the Bluetooth hardware. These intermediate layers, the Host Controller Transport Layer, provide the ability to transfer data without intimate knowledge of the data being transferred. Hence, the Host should receive asynchronous notifications of HCI events independent of which Host Controller Transport Layer is used. Several different Host Controller Transport protocols can be used. The three different Host Controller Transport protocols initially defined for Bluetooth are the USB, UART and RS232 Transport Layer. These three protocols are briefly discussed below although this project deals mainly with the RS232 Transport Layer.

UART Transport Layer

The objective of the HCI UART Transport Layer is to make it possible to use the Bluetooth HCI over a serial interface between two UART on the same PCB. The HCI UART Transport Layer assumes that the UART communication is free from line errors. Event and data packets flow through this layer, but the layer does not decode them.

RS232 Transport Layer

The objective of the HCI RS232 Transport Layer is to make it possible to use the Bluetooth HCI over one physical RS232 interface between the Bluetooth Host and the Bluetooth Host Controller. HCI Commands, Events and Data packets flow through this layer, but the layer does not decode them. The implementation of this protocol is discussed in further detail in section 3.

USB Transport Layer

The objective of the Universal Serial Bus (USB) Transport Layer is to the use a USB hardware interface for Bluetooth hardware. There are two ways in which this can be embodied: as an USB dongle (DCE) in the arrangement similar to figure 3, or integrated onto the motherboard of a notebook PC as shown in figure 5. A specific class code is assigned to USB Bluetooth devices. This will allow the proper driver stack to load, regardless of which vendor built the device. It also allows HCI commands to be differentiated from USB commands across the control endpoint

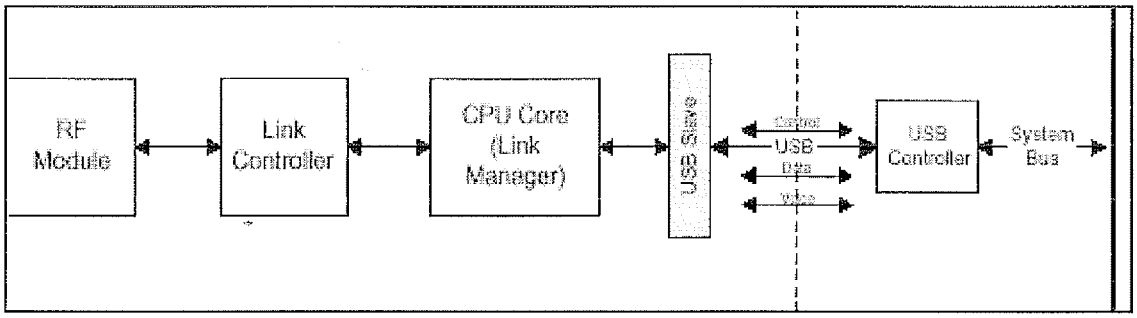


Figure 5 Integrated Bluetooth using the USB Bus

HCI Terminology

Four types of packets can be transferred between the Host and the host controller. They are the HCI Command Packet, HCI Event Packet, HCI ACL (Asynchronous Connectionless link) Data Packet and HCI SCO (Synchronous Connection Orientated) Data Packet. HCI Command Packets are only sent from the Host to the Bluetooth Host Controller. HCI Event Packets are only sent from the

Bluetooth Host Controller to the Host. HCI ACL/SCO Data Packets are sent both to and from the Bluetooth Host Controller. ACL carries packet data while SCO carries voice.

Before we proceed to describe the data transfer process, let us first familiarise with the terminology used to describe the Bluetooth protocol. The smallest unit of data that is transmitted by one device to another is the Baseband Packet. The HCI packet essentially carries the same quantity of data as the Baseband Packet, but baseband specific events such as data encryption and baseband error control are not included. The only higher-level protocol defined for Bluetooth now is the L2CAP packet. The L2CAP layer performs Segmentation and Reassembly of Higher Layer Protocol Data Units (PDU), hence; a single L2CAP packet can be segmented into several HCI data packets (either ACL or SCO).

The Host Controller Transport Layer provides transparent exchange of HCI-specific information. These transporting mechanisms provide the ability for the Host to send HCI commands, ACL data, and SCO data to the Host Controller. These transport mechanisms also provide the ability for the Host to receive HCI events, ACL data, and SCO data from the Host Controller.

Since the Host Controller Transport Layer (explained in the next section) provides transparent exchange of HCI-specific information, the HCI specification specifies the format of the commands, events, and data exchange between the Host and the Host Controller. The following is a brief discussion of the packet formats.

HCI Command Packet Format

The HCI Command Packet is used to send commands to the Host Controller from the Host. The format of the HCI Command Packet is shown in the following figure followed by the definition of each field.

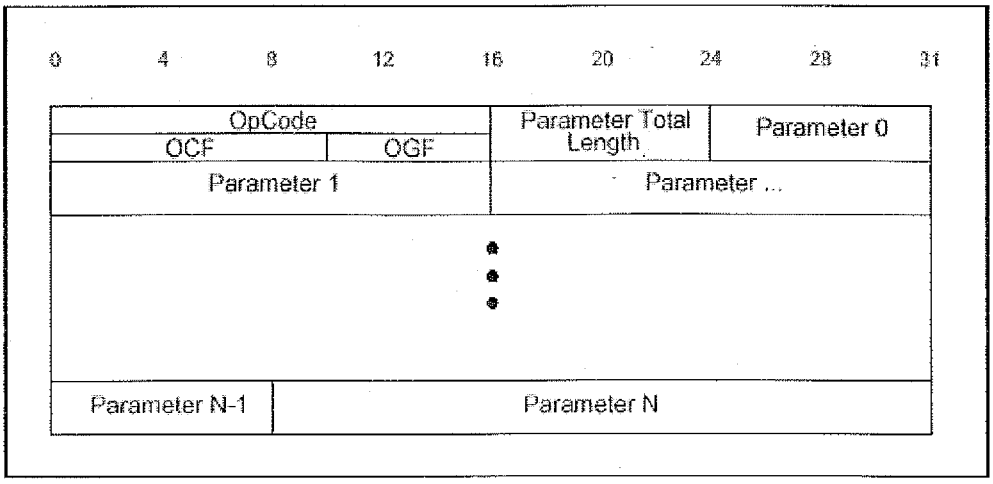


Figure 6 HCI Command Packet format

Op_Code: The Opcode parameter specifies the type of command sent. It is divided into two fields, called the OpCode Group Field (OGF) and OpCode Command Field (OCF). The OGF occupies the upper 6 bits of the Opcode, while the OCF occupies the remaining 10 bits. The OGF of 0x3F is reserved for vendor-specific debug commands. The OGF of 0x3E is reserved for Bluetooth Logo Testing. The organisation of the Opcodes allows additional information to be inferred without fully decoding the entire Opcode.

Parameter_Total_Length: This field specifies the lengths of all of the parameters contained in this packet measured in bytes (that is: total length of parameters, not number of parameters).

Parameter 0 - N: Each command has a specific number of parameters associated with it. These parameters and the size of each of the parameters are defined for each command. Each parameter is an integer number of bytes in size.

The HCI Event Packet

An event is a mechanism that the Host Controller uses to notify the Host when events occur. This includes for command completion, link layer status changes, etc. The Host must be able to accept HCI Event Packets with up to 255 bytes of data excluding the HCI Event Packet header. The format of the HCI Event Packet is shown in following figure followed by the definition of each field.

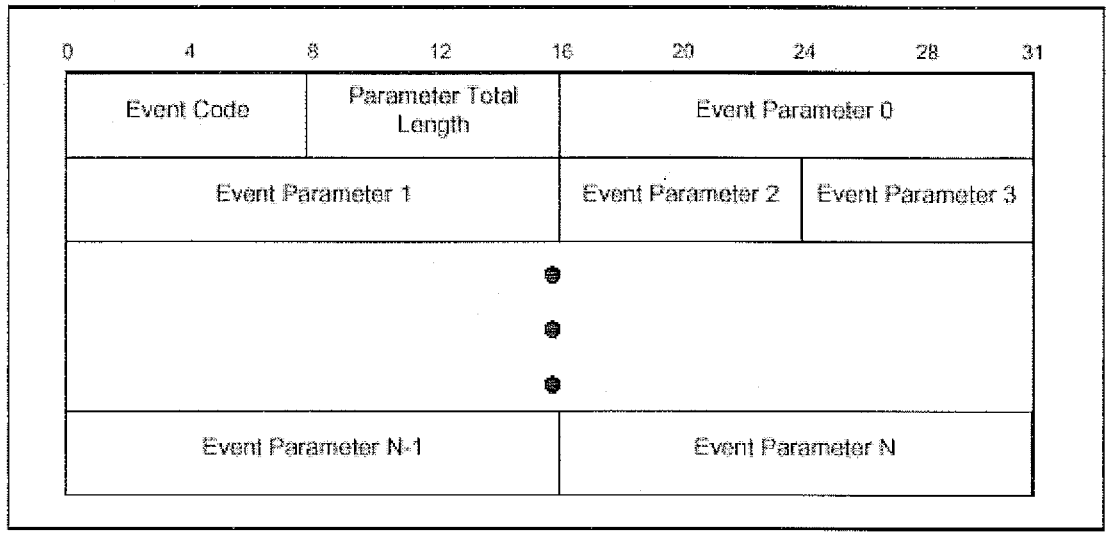


Figure 7 HCI Event Packet format

Event_Code: Each event is assigned a 1-Byte event code used to uniquely identify different types of events. Range: 0x00-0xFF (The event code 0xFF is reserved for the event code used for vendor-specific debug events. In addition, the event code 0xFE is also reserved for Bluetooth Logo Testing)

Parameter_Total_Length: Length of all of the parameters contained in this packet, measured in bytes.

Event_Parameter 0 – N: Each event has a specific number of parameters associated with it. These parameters and the size of each of the parameters are defined for each event. Each parameter is an integer number of bytes in size.

HCI Data Packets

HCI Data Packets are used to exchange data between the Host and the Host Controller. The data packets are defined for both ACL and SCO data types. The format of the HCI ACL Data Packet is shown in following Figure and the format of the SCO Data Packet is shown in Figure 9. The explanation for each of the fields in the data packets follows the packet diagram.

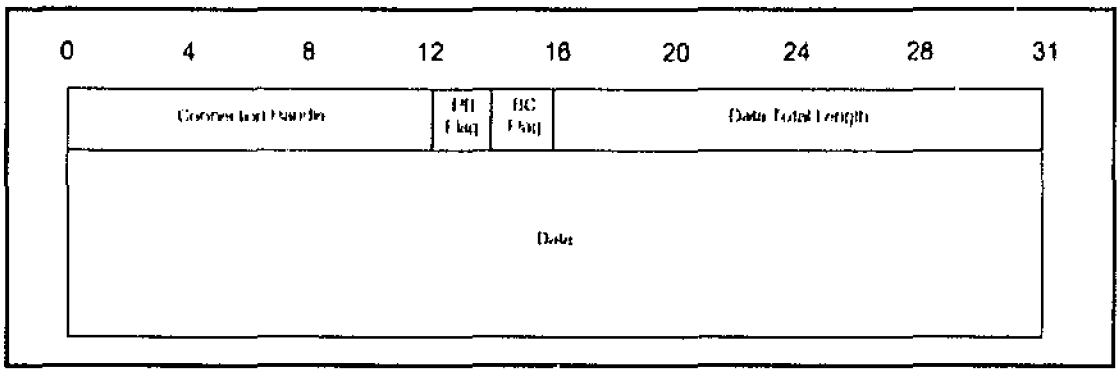


Figure 8 HCI ACL Data Packet format

Connection Handle: A connection handle is a 12-bit identifier, which is used to uniquely address a data/voice connection from one Bluetooth device to another. The connection handles can be visualised as identifying a unique data pipe that connects two Bluetooth devices. The connection handle is maintained for the lifetime of a connection, including when a device enters Park, Sniff, or Hold mode. The Connection Handle value has local scope between Host and Host Controller. There can be multiple connection handles for any given pair of Bluetooth devices but only one ACL connection.

Flags: The Flag Bits consist of the Packet_Boundary_Flag and Broadcast_Flag. The Packet_Boundary_Flag is located in bit 4 and bit 5, and the Broadcast_Flag is located in bit 6 and 7 in the second byte of the HCI ACL Data packet. Data_Total_Length is the length of data measured in bytes. The following table gives an explanation of the various flag settings.

Table 2

HCI ACL Data Packet fields

Packet_Boundary_Flag	
00	Reserved for future use
01	Continuing fragment packet of Higher Layer Message
10	First packet of Higher Layer Message (i.e. start of an L2CAP packet)
11	Reserved for future use
Broadcast_Flag -(in packet from Host to Host Controller):	
00	No broadcast. Only point-to-point.
01	Active Broadcast: packet is sent to all active slaves.
10	Piconet Broadcast: packet is sent to all slaves, including slaves in 'Park' mode.
11	Reserved for future use.

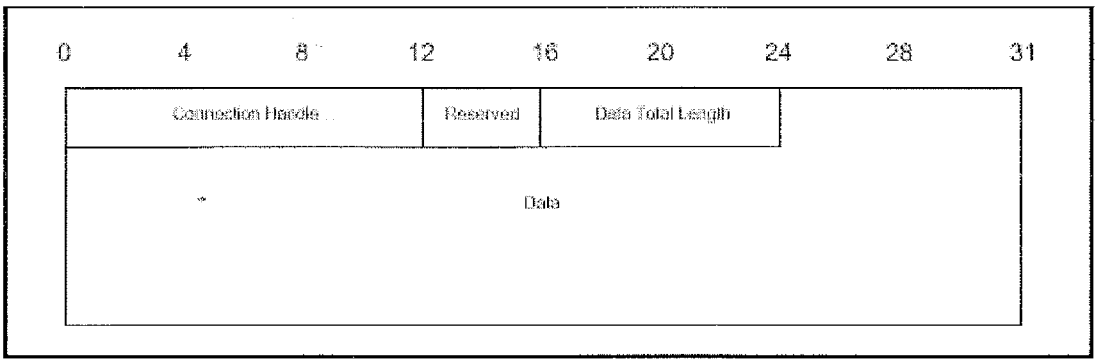


Figure 9 HCI SCO Data Packet format

Connection_Handle: Connection handle to be used to for transmitting a SCO data packet or segment.

The Reserved Bits consist of four bits which are located from bit 4 to bit 7 in the second byte of the HCI SCO Data packet. They are Reserved for future use

Data_Total_Length: Length of SCO data measured in bytes

CHAPTER 3
RS232 HOST CONTROLLER TRANSPORT LAYER

This section details the development of software to implement the RS232 HCI Transport for Linux.

Types of HCI Transport Data

There are four kinds of HCI packets that can be sent via the RS232 Transport Layer as described in the previous section. HCI driver does not provide the ability to differentiate the four HCI packet types. Therefore, if the HCI packets are sent via a common physical interface, a HCI packet indicator has to be added. In addition to those four HCI packet types, two additional packet types are introduced to support dynamic negotiation and error reporting. The Error Message Packet is used by the receiver to report the nature of error to the transmitting side. The Negotiation Packet is used to negotiate the communication settings and protocols. The table below shows the types of packet and the corresponding packet header.

Table 3

HCI RS232 Packet Header

HCI packet type	HCI packet type indicator
HCI Command Packet	0x01
HCI ACL Data Packet	0x02
HCI SCO Data Packet	0x03
HCI Event Packet	0x04
Error Message Packet*	0x05
Negotiation Packet*	0x06

The HCI packet indicator is followed by an 8-bit sequence number that is incremented by one every time a packet is sent. The sequence numbers are not incremented in the case of a retransmission packet that is sent as a part of error recovery. The retransmitted packet uses the same sequence number as the original packet. The HCI packet immediately follows the sequence number field. All four types

of HCI packets have a length field, which is used to determine how many bytes are expected for the HCI packet. The Error Message Packet and Negotiation Packet are fixed-length packets, although the negotiation packet can be extended up to seven more bytes (as shown in part 3.2.6), based on the number in the extension field. The frame of the basic RS232 Transport Packet is shown below.

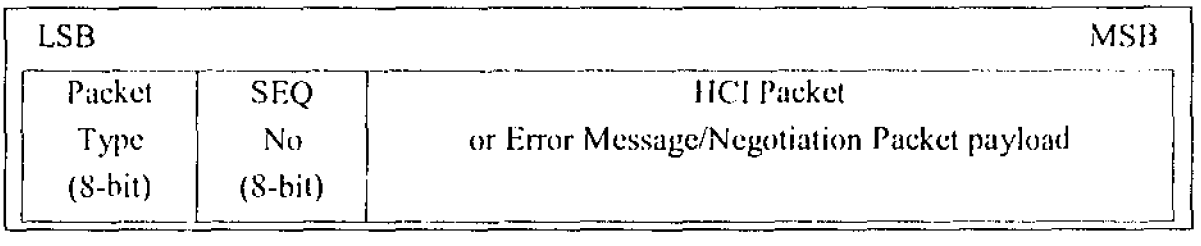
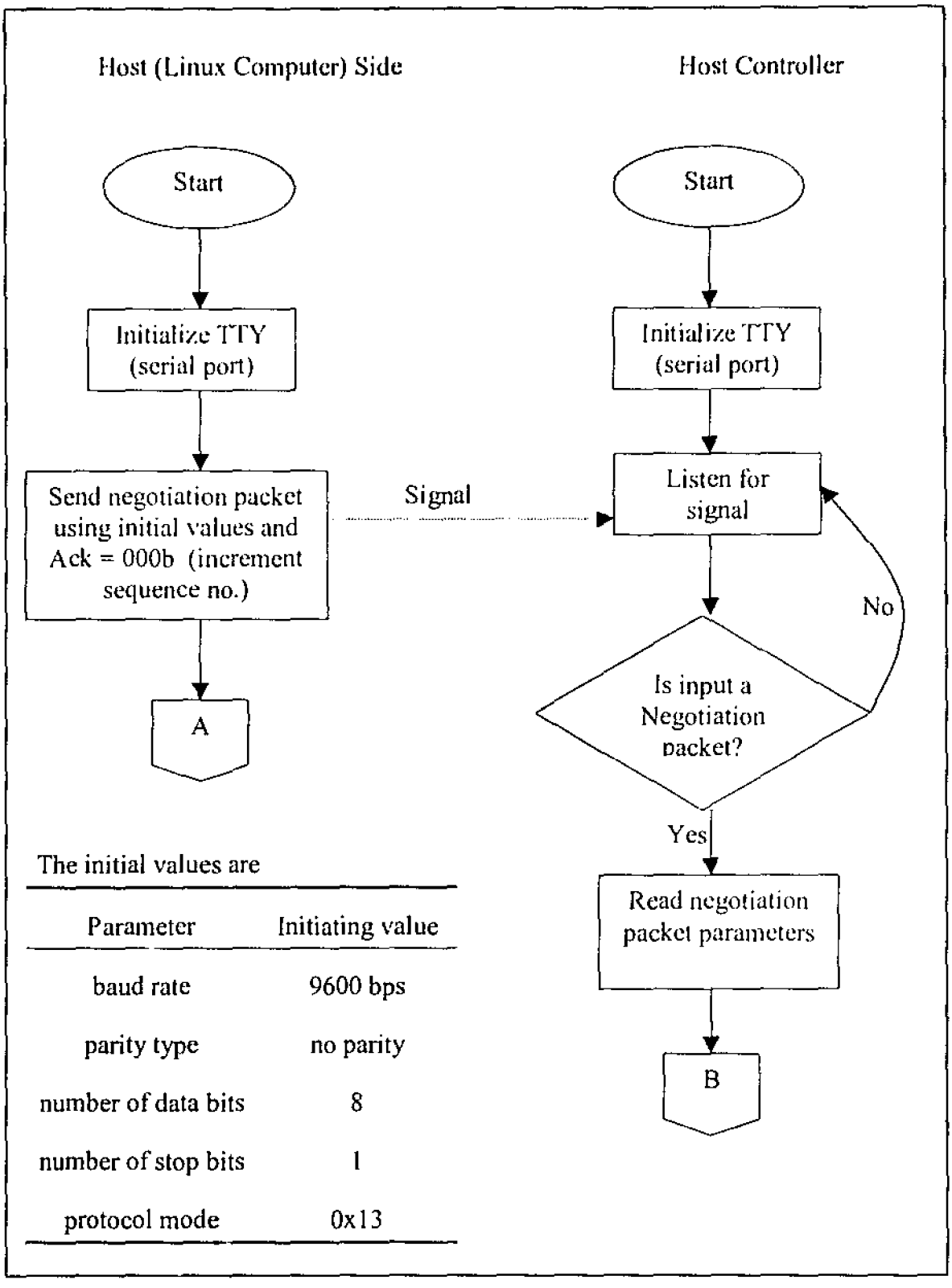


Figure 10 RS232 HCI Transport Packet

The least significant byte is transmitted first (unsigned Little Endian format).

The Negotiation Packet

During the establishment of the RS232 link, the link parameters should be negotiated between the Host Controller and the Host. The basic negotiation procedure is shown in the following pages (for simplicity, error control and recovery is not shown in the flow chart).



The initial values are

Parameter	Initiating value
baud rate	9600 bps
parity type	no parity
number of data bits	8
number of stop bits	1
protocol mode	0x13

Figure 11 Host Controller Interface negotiation process

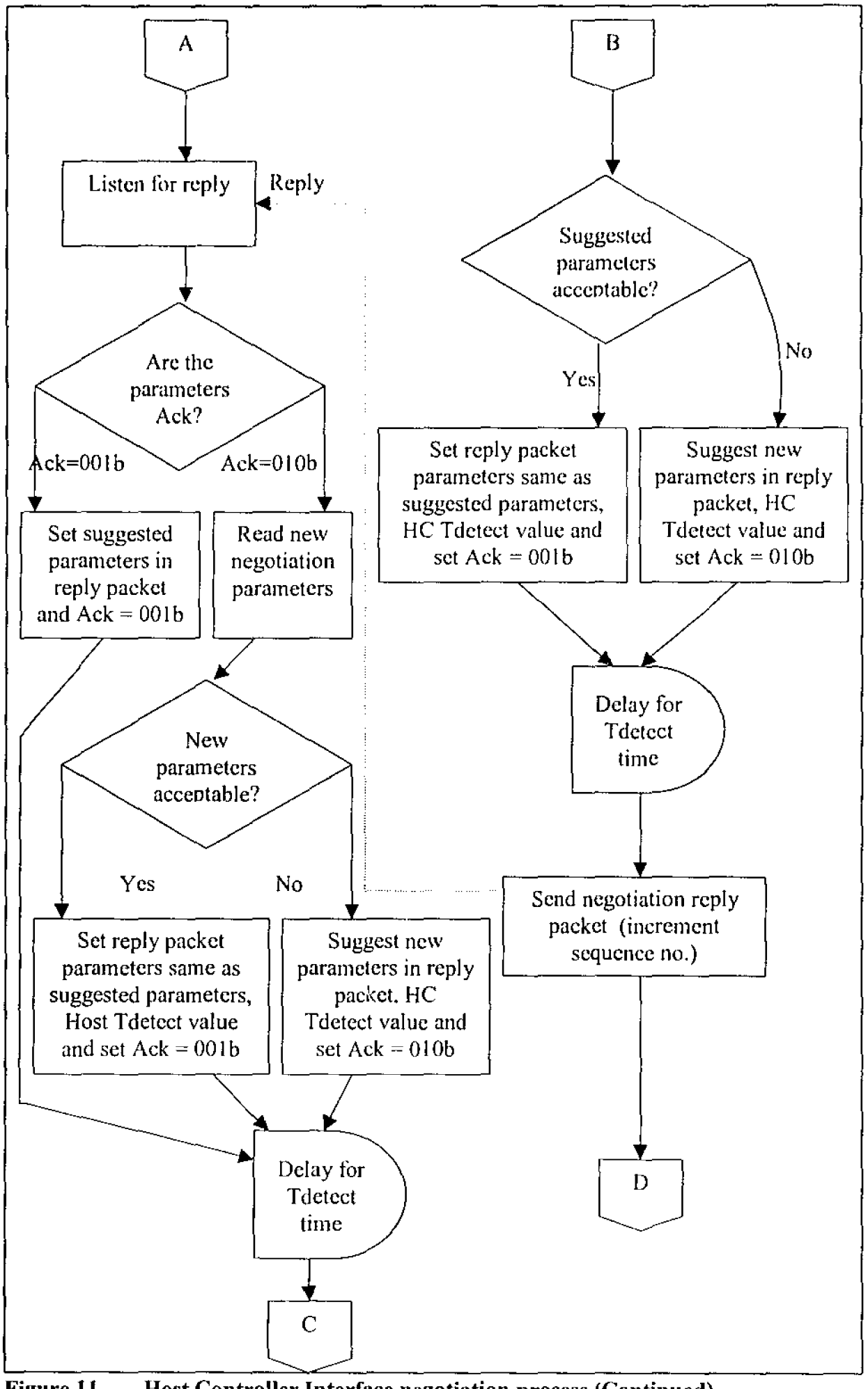


Figure 11 Host Controller Interface negotiation process (Continued)

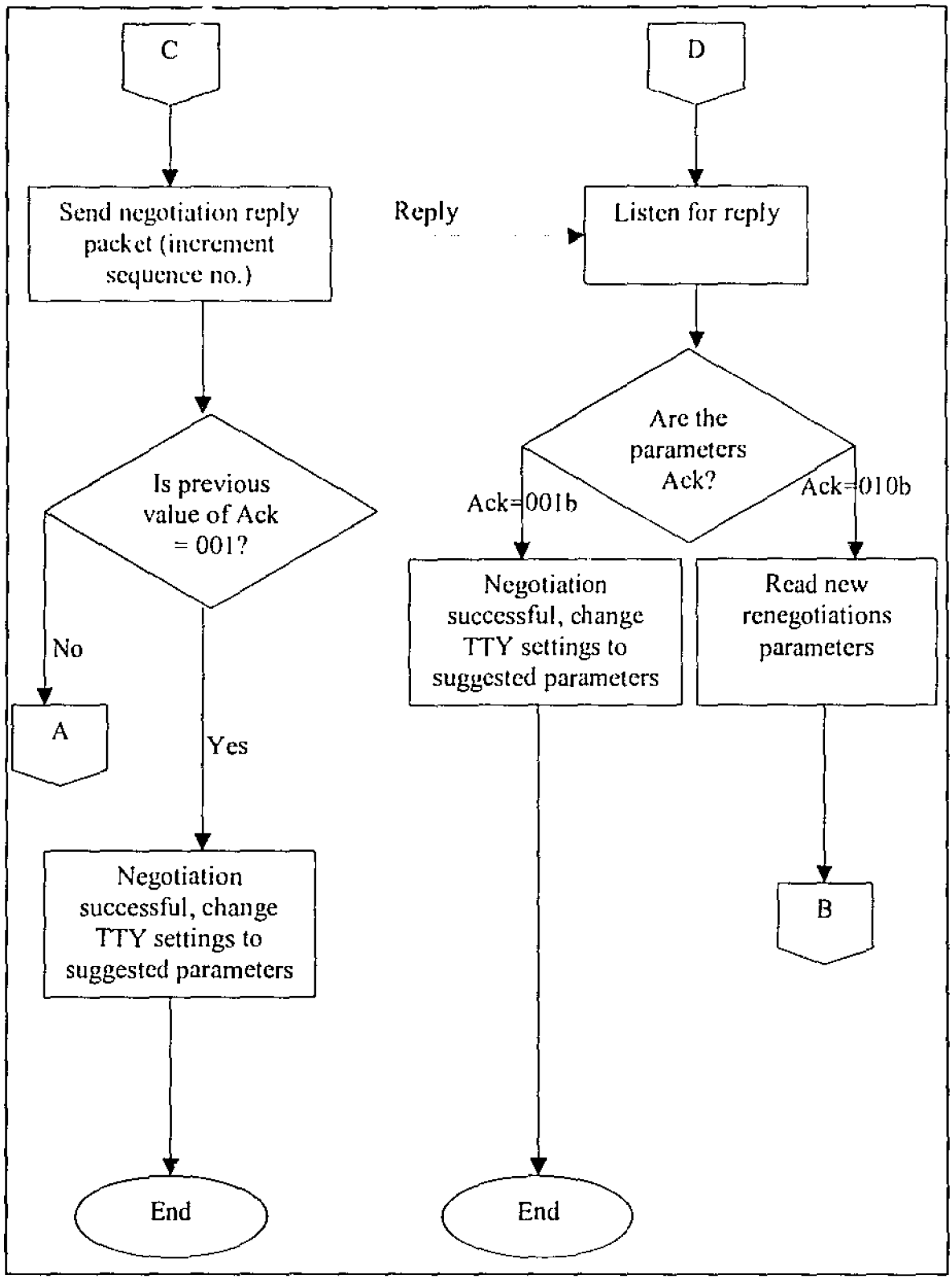


Figure 11 Host Controller Interface negotiation process (Continued)

LSB MSB

Packet Header 0x06 (8 bits)	SEQ No. (8-bit)	UART Settings and ACK (8 bits)	Baud Rate (16 bits)	Tdetect Time (16 bits)	Protocol Mode (8 bit)

Figure 12 Negotiation Packet Format

The negotiation packet parameters are shown in the above figure. The following is the description of each parameter.

Packet Header

The negotiation packet header type indicator is 0x06 (as shown in table 3)

SEQ No.

The packet sequence number is incremented by one each time a packet is transmitted, excluding the retransmission packets. The unsigned Little Endian format is used.

UART Settings and ACK Field

Bit 0-1	Bit 2	Bit 3	Bit 4	Bit 5-7
Reserved	Stop bit (1 bit)	Parity Enable (1 bit)	Parity Type (1 bit)	Ack Code (3 bits)

Figure 13 UART settings and Acknowledgement field

The following tables explain the meaning of each of the entry in the above diagram.

Table 4

UART settings and Acknowledgement

Bit value	Stop Bit	Parity Enable	Parity Type
0	1 stop bit	No parity	Odd Parity
1	2 stop bits	Parity Enable	Even Parity

Ack Code	Negotiation Acknowledgement
000b	Request
001b	Accepted
010b	Not accepted with new suggested values
011b-111b	Reserved

Baud Rate

In this context, the baud rate actually refers to the connection speed. The integer N should be entered in the field for the baud rate where:

The actual rate = 27,648,000 / N (where N=0 is invalid)

Therefore, the Maximum possible rate is 27.648Mbps, and the Minimum possible rate is 421.88bps

Tdetect Time

If RTS/CTS is used for error indication and re-synchronisation, Tdetect is the maximum time required for the transmitter to detect the CTS state change, plus the time it takes to flush the transmit buffer. Otherwise, Tdetect represents the local-side interrupt latency. The unit of time should be specified in 100 microseconds. (In the software developed, Tdetect is set to 1 millisecond).

Protocol Mode

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
CRC Used	Delimiter Used	RTS / CTS used	RTS / CTS Mode	Error Recovery	Ext0	Ext1	Ext2

Figure 14 Protocol mode error control and recovery field

The first five bits of the Protocol mode field describes manner in which error control should be implemented. The last three bits specify an increase in length in the negotiation packet length, and is there for future expansion purposes. The following tables explain the meaning of each bit. There are only two protocol modes described in the specification. They are 0x13 (LSB 11001000 MSB), the default, which uses CRC, Delimiters, and 0x14 (LSB 00101000 MSB), which uses only the RTS/CTS (Request/ Clear to sent) error control lines on the RS232 cable. Both modes use error recovery. The Host Controller may choose to support only one protocol mode. However, the Host (that is, the Linux computer) should be able to support any combination. The detail operation of each protocol mode is described in section 4.

Table 5

Protocol mode settings

Bit value	CRC Used	Delimiter Used
0	CRC-CCITT is not attached at the end of the packet.	Delimiter, 0x7E, is not used
1	CRC-CCITT is attached at the end of the packet. (Default)	Delimiter, 0x7E, is used with COBS (Default)

Bit value	RTS/CTS Used	RTS/CTS Mode
0	RTS/CTS is not used (Default)	RTS/CTS is used for Error indication and resynchronisation. (Default)
1	RTS/CTS is used	RTS/CTS is used for hardware flow control

Bit value	Error Recovery Used	Ext2,Ext1,Ext0
0	Error Recovery is not supported	These three bits indicate the number of extra bytes attached to the negotiation packet. Meant for future expansion
1	Error Recovery is supported. (Default)	

CRC can be used with either RTS/CTS or delimiters, as a synchronisation mechanism, although the specification only describes a case when it is used with delimiters. Usage of RTS/CTS reduces the computation time for COBS encoding, but requires two extra copper wires which may not be suitable in some applications. More details will be covered in section four.

Error Recovery retransmits the packet with error and all subsequent packets if RTS/CTS are used for synchronisation. On the other hand, if the delimiter 0x7E is used with COBS as a synchronisation mechanism, then the error recovery retransmits only the packet with error. Even if error recovery is disabled, the error message packet should still be sent to the transmitter side when the receiver side detects an error.

The Error Message Packet

The following figure shows the error-message packet format.

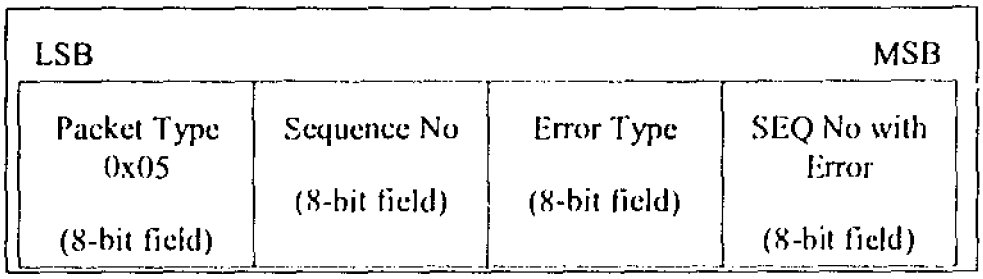


Figure 15 Error Message Packet

The table below describes the type of errors. The SEQ No with error holds the sequence number of the packet found with error.

Table 6

Error Types available

Error code	Type Description
0x00	Reserved
0x01	Overrun Error
0x02	Parity Error
0x03	Reserved
0x04	Framing Error
0x05	0x07 Reserved
0x08	CRC Error
0x09	Missing SEQ No
0x0A	0x80 Reserved
0x81	Missing Retransmission Packet
0x82	0xFF Reserved

CHAPTER 4
SOFTWARE DEVELOPMENT

Programming on Linux (Problems faced)

The problems faced during the development of the Host Controller Transport layer software lies mainly in the student's inexperience with the Linux programming environment. Problems include finding the correct tools to use (the program was developed using emacs and gcc), learning how to read, access, and control serial ports. Fortunately there is a vast amount of Linux programming resources available on the on the Internet. Particularly useful documents includes the "The Linux Serial Programming HOWTO" (Baumann, 1998), "Serial Programming Guide for POSIX Operating Systems" (Sweet 1999) and "Linux Programmer's Guide" (Goldt, Meer, Burkett, Welsh 1996). The following describes the operation of the Host and Host Controller emulation software and the software development process. The program source code is attached in the Appendix. The list of programs and their respective functions are listed in the following table.

Table 7

List of programs

Program	Function
myhost	Program which emulates the host.
myhc	Program which emulates the hc.
Stuff.c	Used to perform consistent overhead byte stuffing
Unstuff.c	Used to perform consistent overhead byte unstuffing
Set_port.c	Used to setup the serial port
Nego_ receive.c	Used to receive, process and unpack data
Nego_ sent.c	Used to sent, process and pack data
Host.c	Source file for *myhost*
Hc.c	Source file for *myhc*
Crc.c	Calculates and test crc

The Negotiation Routine

The negotiation procedure was carried out according to the flowchart in figure 11. The source code is attached in the Appendix. The programs *myhost.c* and *myhc.c* emulates the host-side and host-controller side of the RS232 HCI transport. The functions *nego_send.c* and *nego_receive.c* handles the sending and receiving of the data between the Host and Host Controller calling other sub-functions to handle error control such as CRC, and delimiters for Protocol Mode 0x13.

Both sides are initially set to 9600 bps baud rate using the function *set_port* and C function *tcsetattr*. The program starts when the host-side emulator, *myhost*, sends a negotiation packet using *nego_send.c* to the host-controller-side with the Acknowledge code (bits 21, 22, 23) set to 000b. The contents of this packet are the proposed parameters for the link, including the preferred baud rate, error control schemes (UART field) and the link latency (Tdetect field). If protocol mode 0x13 is used, a CRC sequence is attached to the end of the data and the data passes through the *StuffData*

function to remove any occurrence of the value 0x7E. 0x7E is then appended to the front and end of the packet to act as delimiters. On the receiver side, the *nego_receive.c* function polls the serial port until it receives the delimiter 0x7E (for Protocol Mode 0x13) which indicates a received packet. The *nego_receive* calls the *UnStuffData* function to restore the data to its proper format. It then does a check using the *crc* functions and sends an Error packet with error type 0x08 back to the sender if the CRC fails. For a packet without error, the *nego_receive* will read the suggested link parameters proposed by the Host. If it can accept the parameters, it sends a negotiation packet using *nego_sent* back to the host, with the host-controller's maximum latency (Tdetect) and the Acknowledgement code set to 001b. If it cannot accept the parameters, the Acknowledgement code is set to 010b instead and the Host-controller sent its required link parameter to the host using *nego_sent*.

Back at the Host end, *nego_receive* is started to read the serial port for a reply from the Host-Controller. On receiving a packet from the Host-Controller indicated by the 0x7E delimiter, the program *UnStuffData* the packet and checks the CRC for errors, sending an error packet back to the Host-Controller is an error is detected. If the received Acknowledgement code is 001b (that is, the Host's suggested parameters are accepted), the Host sends another negotiation packet similar to its first one but with the Acknowledgement code set to 001b.

If the Host-Controller had asked for re-negotiation (by setting the Ack code to 010b), the Host processes the new parameters. If the Host accepts the re-negotiated parameters, the Host sends another negotiation packet similar to its first one but with the Acknowledgement code set to 001b to the Host-Controller. If the Host cannot accept the renegotiated parameters, the Hosts sends another re-negotiation packet with new parameters to the Host-Controller with the Acknowledgement code is set to 010b using *nego_sent*. The re-negotiation process is repeated until both sides are satisfied with the link settings or one side runs out of negotiation parameters.

After both sides are satisfied with the link parameters, they reset the serial ports to the negotiated settings using the function *set_port*.

Although the program does not show it, the negotiation process can be initiated again to re-negotiate new values or to update the Tdetect time. When the negotiation is

initiated again, the link should use the present link settings instead of the initial default settings.

Protocol Mode 0x13 Operation

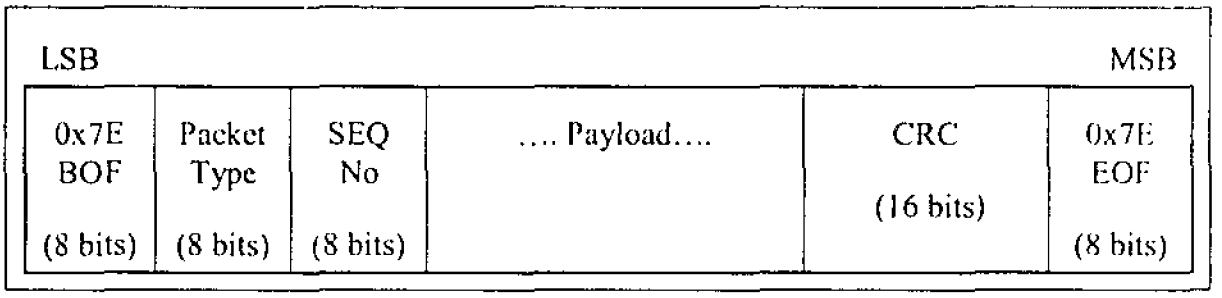


Figure 16 Protocol Mode 0x13 Packet Format

The above figure shows the format of a frame using Protocol Mode 0x13. This mode requires the use only three wires on the RS232 port (TxD, RxD and signal ground). It also frees the CTS/RTS so that they can be used for hardware flow control. This protocol mode uses cyclic redundancy checks and delimiter 0x7E to indicate the beginning and end of a frame. To prevent data bytes "0x7E" from being mistaken as the framing delimiter, as byte stuffing technique known as COBS is used to remove 0x7E from the data. The following describes the implementation of the error control procedures.

Cyclic Redundancy Check (CRC) Implementation

The CRC used is the 16-bit CCITT format with the Generator Polynomial = $x^{16} + x^{12} + x^5 + 1$. The CRC generation involves long division of the data (appended with 16 zeros) with the Generator polynomial (Halsall, 1996 p.134-137). The CRC code is the remainder from the division. At the receiver side, the data is passed through the long division process again. If the process yields 0x00 as the remainder, the code is deemed error free. The code segment for implementing CRC *crc.c* is attached in the Appendix.

Consistent Overhead Byte Stuffing (COBS)

COBS (Consistent Overhead Byte Stuffing) is a byte stuffing technique that is similar to HDLC-like framing. Compared with other older Byte Stuffing techniques, COBS yields significantly less overhead (>0.5%) regardless of the data pattern

(Cheshire, Baker, April 1999). It uses two steps to escape the delimiter, 0x7E. The first step is eliminating zeros and then replacing all 0x7E with 0x00 between the beginning and ending delimiters. The COBS code (Carlson, J., Cheshire, S. and Baker, M. Nov 1997) is attached in the Appendix.

Error Recovery

When the receiving end detects any error, it sends the error message packet with an error type back to the transmitting side. This error message packet contains a Sequence Number with Error field (SEQ No with Error) indicating in which packet the error was detected. The Sequence Number field that is on every packet is an 8-bit field that is incremented by one each time any type of packet is transmitted, except for the retransmission packets. The retransmitted packets should contain the original sequence number in the SEQ Number field.

The transmitting side should retransmit only the HCI packets that had an error. This is indicated by the SEQ No with Error field. It is the responsibility of the receiving end to reorder the packets in the right order. If the transmitting side does not have the packet with the correct sequence number in the retransmission holding buffer, it sends the error message packet with the missing sequence number for the retransmission packet, so that the receiving end can detect missing packets. This error message packet has Error Type equal to 0x81 and SEQ Number with the Error field. The missing packet is indicated at the receiver side and will be handled by the higher layers.

Protocol Mode 0x14 Operation

Although protocol mode 0x13 has been defined as the default protocol, some Bluetooth hardware actually uses mode 0x14 as the only mode of operation. This mode requires less processing by both the Host and CPU core in the Host Controller as the COBS code need not be calculated.

This mode does not use hardware flow control, which is protocol mode (LSB 00X1X000 MSB) and is used by the HCI UART interface. Hence, hardware flow control must be first disabled (that is, *...c_cflags &= ~CRTSCTS*).

The RTS/CTS lines are connected in NULL modem fashion (that is, the RTS of the Host to the CTS of the Host Controller and vice-versa). Packets can only be sent if the transmitter's CTS line is asserted (10V). Hence, the receiver controls the transmission by asserting and de-asserting the RTS line.

Unfortunately, at the time of writing, this mode of operation is not operating as anticipated. The source code is attached in the Appendix.

Error Recovery

When using protocol mode 0x14, the HCI packet is sent only when the transmitter's CTS bit is 1. If the CTS bit changes to 0 during the HCI packet transfer or after the last byte is transmitted, this indicates that there was some error on the receiver side. The receiving end will deassert RTS as soon as it detects any error, and send an error packet with an error type back to the transmission side. This error packet contains a Sequence Number with Error field that indicates the packet in which error was detected.

When the transmitting end detects CTS bit changing from 1 to 0 at any time, the transmitting end should hold the transmission and wait until the error packet is received before resuming the transmission. When the receiving end is ready to receive the new data, it should assert RTS after the minimum Tdetect time. Here, Tdetect time is the maximum time required for the transmitter side to detect the state change on CTS bit, plus the time it takes to flush the transmit buffer. The Tdetect value of each side should be informed to the other side during the negotiation phase. The local Tdetect value and the remote side Tdetect value together, along with the baud rate, can also be used to estimate the queue length required for the retransmission holding buffer. For an assumed baud rate of 115200 (the maximum for many older computers), the holding buffer size = 115200 x (total Tdetect) bits. Before the receiving side asserts RTS line again, it should flush the RX buffer.

The transmission side should retransmit all of the HCI packets from the packet that had an error, which is indicated by SEQ No with Error field. Before it retransmits, it should flush the transmit buffer that may hold the leftover data from the aborted previous packet. As it retransmits the packets from the transmission holding buffer, it should start transmitting the packet with the Sequence Number that matches the SEQ

No with Error. If the transmitting side does not have the packet with the correct sequence number in the retransmission holding buffer, the transmitter should send an error message packet with error type 0x81, and it should skip to the packet with the sequence number that is available in the buffer. The missing packets are indicated at the receiver side and will be handled by the higher layers.

CHAPTER 5
CONCLUSION

Project Achievements and Contributions

This project has allowed me to familiarise with the various wireless networking protocols and the concept of a personal area network (PAN). Bluetooth strength lies in the low power operation and low cost. Lately, it has rapidly gain momentum as the de-facto standard for the ad-hoc personal area network. This is a surprising achievement for a standard that was initially conceived to replace cables.

As there were no existing books or guides for Bluetooth at that time the project initially started, we had to read the core specifications and profile specifications. In the process, we gained a rather in depth understanding of the Bluetooth protocol as a whole. However, the short time and limitation in manpower does not allow for a more extensive exploration and implementation of the Bluetooth protocol.

Finally, I gain the hands-on knowledge of programming on the Linux platform which was something that I had always wanted to learn but never got around to do. Programming tools learned and used during the course of the project was mainly the GNU C compiler, debugger and emacs editor. The project has initially set out to develop a GUI for the Bluetooth protocol and in the process, thus I have also learned some techniques for GUI development on Linux using the Gimp toolkit (GTK+) and Glade.

Comments and Recommendations for Future Development

Due to the limitation in time, the software still has areas of functionality not in place. These include:

1. Implementing Error Control using CTS/RTS lines (Protocol Mode 0x14). This is not functioning properly as at the time of writing this report.
2. The software need to be tested for Real Time performance when used for synchronous connection, (SCO i.e. voice) data.

3. The code still needs some debugging (in case of memory overruns).

4. Allow full duplex communications between the host and the host controller emulators. Presently, they are set up in half-duplex mode.

Other areas for improvement include.

1. Adding auto detection of baud rate for the RS232 link.

2. Allow using another protocol mode other than 0x13 in case the receiver operates only on mode 0x14.

3. Implementation of HCI commands for controlling of Bluetooth Hardware.

4. Implementation of HCI Events handling for acting as a Bluetooth Host Controller Emulator.

The last two improvements allows 2 the possible uses for the software:

1. A completed Bluetooth stack can be developed bottom up from the *myhost* Host transport emulation program. Higher layers such as L2CAP, RFCOMM, SDP and the user application can directly access the Bluetooth network through the Host transport without detail knowledge of the actual physical connections.

2. The software can act as a Bluetooth protocol analyser and tester for an embedded Bluetooth chip as illustrated in figure 17 next page. A copy of the Host or Host emulation program, is stored on the testing computer, and communicates with the Bluetooth chip under test. Using this technique, it is more convenient to perform testing and debugging of the upper layers (L2CAP, RFCOMM, SDP and the application) than debugging via RF through the baseband. The lower layers (especially Baseband and Radio) can also be tested and analysed for performance using this software method.

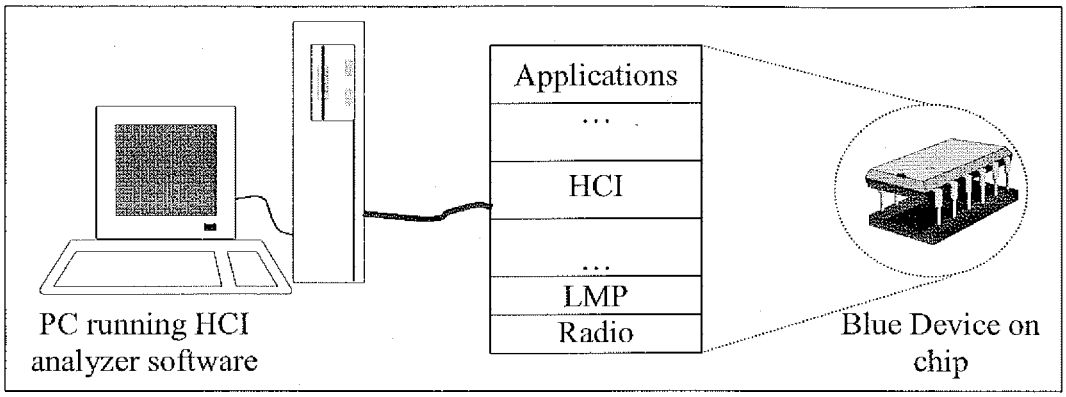


Figure 17 Possible uses for this Software

REFERENCES

The Official Bluetooth web site, [on-line]. Available WWW http://www.Bluetooth.com

RS232 HCI Transport layer: An addendum to the HCI document. [on-line]. Available WWW: www.bluetooth.com/link/spec/bluetooth_h3.pdf

Cheshire, S and Baker, M (April 1999) "Consistent Overhead Byte Stuffing", IEEE/ACM Transactions On Networking, Vol. 7, No. 2, April 1999

Carlson, J., Cheshire, S. and Baker, M. (Nov 1997). PPP Consistent Overhead Byte Stuffing (COBS), [on-line]. Available WWW: www.globecom.net/ietf/draft/draft-ietf-pppext-cobs-00.html

Goldt, S., Meer, S., Burkett, S., Welsh, M. (March 1996). Linux Programmer's Guide. [on-line]. Available WWW: http://www.ibiblio.org/pub/Linux/docs/linux-doc-project/programmers-guide/

Canosa, J (Nov 2000). Network Protocols for the Home. [on-line]. Available WWW: http://www.embedded.com/internet/0011/0011ia2.htm

Baumann, P. H., (Jan 1998). The Linux Serial Programming HOWTO. [on-line]. Available WWW: http://www.linuxdoc.org/HOWTO/Serial-Programming-HOWTO.html

Hallsall, F. (1996). Data Communications, Computer Networks and Open Systems. Harlow, Essex: Addison Wesley Longman Limited

Specification of the Bluetooth System-Core v1.0B. (2000) [on-line]. Available WWW: http://www.bluetooth.com/developer/specification/core_10_b.pdf

Specification of the Bluetooth System-Profiles v1.0B. (2000). [on-line]. Available WWW: http://www.bluetooth.com/developer/specification/profile_10_b.pdf

Palowireless Bluetooth Resource Centre. [on-line]. Available WWW: http://www.palowireless.com/bluetooth/

Muller N.J. (Sept 2000). Bluetooth Demystified. McGraw-Hill Telecom

Sweet, M.,(1999). Serial Programming Guide for POSIX Operating Systems. [on-line]. Available WWW: http://www.easysw.com/~mike/serial/serial.html

Wall, K., Watson, M., Whitis, M, (1999). Linux Programming Unleashed. Indianapolis: Sams

IBM BlueDrekar. [on-line]. Available WWW: http://www.alphaworks.ibm.com/tech/bluedrekar

The Bluetooth on Linux homepage. Available WWW: http://developer.axis.com/software/bluetooth/

APPENDIX

```
/*** host.c ***/
#include "project.h"

/* Define Negotiation mode paramenters */

#define TTYUSED "/dev/ttyS1"
#define BUFLIMIT 255 /* 2^8 */

/*********************************************************************/
/****** Initial values for negotiation ***************************/
/*********************************************************************/
#define SUGGESTBAUD 115200
#define DEFAULT_UART 0x00
#define DEFAULT_TDETECT 10

/*********************************************************************/
/******* Start of Host Emulation Program *************************/
/*********************************************************************/
main()
{
  int fd, res, suggest_baud, suggest_tdetect, end_neg = 0, newbaud;
  struct termios oldtio, newtio;

  unsigned char stop_bit, parity, parity_type, ack_code,
    use_crc, use_delimit, use_rts_cts, rts_cts_type,
use_error_recovery;

  /* Initialise port */
  fd = open(TTYUSED, O_RDWR | O_NOCTTY);  /* optional | O_NDELAY */
  if (fd <0) {perror(TTYUSED); exit(-1); }
  tcgetattr(fd,&oldtio); /* save current port settings */

  newtio = set_port(DEFAULTBAUD); /* more settings to be added later
*/

  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);
  /*** initialise transmission buffer ***/

  negotiation_send(fd, DEFAULT_UART, SUGGESTBAUD, DEFAULT_TDETECT,
PROTOCOL_MODE(0,0,1,0,0,0), NEG_PKT_HDR, 0, 0);

  while(end_neg==0)
  end_neg = negotiation_receive(fd, HOST); /* start listening */
/*********************************************************************/
  /******** Start of actual link ***********************************/
  /*********************************************************************/
  newbaud = end_neg;
  printf("new baud %d", newbaud);
  newtio = set_port(newbaud); /* set baud rate to new settings */
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);
/* Clean up, reset port and close fd */
  tcsetattr(fd,TCSANOW,&oldtio);
  close(fd);
}
```

```
/**** File hc.c ************************/

#include "project.h"

#define TTYUSED "/dev/ttyS0"

/* Define Negotiation mode paramenters */


/*********************************************************************/
/******* Start of Host Controller Emulation Program ****************/
/*********************************************************************/


main()
{
  int fd, end_neg=0, newbaud;
  struct termios oldtio, newtio;

  fd = open(TTYUSED, O_RDWR | O_NOCTTY); /* | O_NDELAY dont wait for
host*/
  if (fd <0) {perror(TTYUSED); exit(-1); }
  tcgetattr(fd,&oldtio); /* save current port settings */

  newtio = set_port(DEFAULTBAUD); /* more settings to be added later
*/

  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);
  while(end_neg == 0){
    end_neg = negotiation_receive(fd, HC); /* start listening */
  }

  /*****************************************************************/
  /******** Start of actual link ***********************************/
  /*****************************************************************/
  newbaud = end_neg;
  printf("new baud %d", newbaud);
  newtio = set_port(newbaud); /* set baud rate to new settings */
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);
  /* Clean up, reset port and close fd */
  tcsetattr(fd,TCSANOW,&oldtio);
  close(fd);

}
```

```
/**** File set_port.c ************************/


#include "project.h"

struct termios set_port (int baud)
{
  struct termios newtio;
  /* We will try with protocol mode 0x14 first i.e. no CRC, delimiters
*/
  /* DEFAULTBAUD: Set bps rate to 9600
   * ~CRTSCTS : disable output hardware flow control
   * CS8      : 8n1 (8bit,no parity, 1 stop bit)
   * CLOCAL   : local connection, no modem contol
   * CREAD    : enable receiving characters */

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag &= ~CRTSCTS;
  newtio.c_cflag = baud | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;
  /* set input mode (non-canonical, no echo) */
  newtio.c_lflag = 0;
  newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 1;   /* blocking read until 8 chars received
*/
  return newtio;
}
```

```
/**** File crc.c ************************/

#include "common.h"

/*********************************************************************/
/*****Calculates the 16-bit CCITT checksum with polynomial *********/
/**** x^16+x^12+x^5+1 (i.e. 0x1021) for the indicated buffer *******/
/*********************************************************************/

#define CCITT_CRC_GEN 0x1021

unsigned short crc(unsigned char *buf, int length)
{
register unsigned short CRC = 0;
unsigned short databyte;
unsigned char i;
  while (length--) {
    databyte = *buf++ << 8;
    for (i=8; i>0; i--) {
      if ((databyte ^ CRC) & 0x8000)
        CRC = (CRC << 1) ^ CCITT_CRC_GEN;
      else
        CRC <<= 1;
      databyte <<= 1;
    }
  }
  return (CRC);
}
```

```
/**** File stuff.c ************************/

#include "cobs.h"


/*
* StuffData stuffs "length" bytes of data from the buffer "ptr",
* writing the output to "dst", and returning as the result the
* address of the next free byte in the output buffer.
* The size of the output buffer must be large enough to accommodate
* the encoded data, which in the worst case may expand by almost
* 0.5%.  The exact amount of safety margin required can be
* calculated using (length+1)/206, rounded *up* to the next whole
* number of bytes.  E.g. for a 1K packet, the output buffer needs to
* be 1K + 5 bytes to be certain of accommodating worst-case packets.
*/


unsigned char *StuffData(const unsigned char *ptr, unsigned int
length, unsigned char *dst, unsigned char **code_ptr_ptr)
{
     const unsigned char *end = ptr + length;
     unsigned char *code_ptr  = *code_ptr_ptr;
     unsigned char code       = DiffZero;

     /* Recover state from last call, if applicable */
     if (code_ptr) code = Rx(*code_ptr);
     else code_ptr = dst++;

     while (ptr < end)
       {
       unsigned char c = *ptr++;  /* Read the next character */
       if (c == 0)   /* If it's a zero, do one of these operations */
         {
         if    (isRunZero(code) && code < RunZeroMax) code++;
         else if (code == Diff2Zero) code = RunZero;
         else if (code <= MaxConvertible) code += ConvertZP;
         else  FinishBlock(code);
         }
       else    /* else, non-zero; do one of these operations */
         {
    if (isDiff2Zero(code)) FinishBlock(code - ConvertZP);
         else if (code == RunZero)   FinishBlock(Diff2Zero);
         else if (isRunZero(code))   FinishBlock(code-1);
         *dst++ = Tx(c);
         if  (++code == Diff)    FinishBlock(code);
         }
       }
     *code_ptr_ptr = code_ptr;
     FinishBlock(code);
     return(dst-1);
     }
```

```
/**** File unstuff.c ************************/


#include "cobs.h"


   /*
    * UnStuffData decodes "srclength" bytes of data from the buffer
    * "ptr", writing the output to "dst". If the decoded data does not
    * fit within "dstlength" bytes or any other error occurs, then
    * UnStuffData returns NULL.
    */
         unsigned  char  *UnStuffData(const  unsigned  char  *ptr,
unsigned int srclength, unsigned char *dst, unsigned int dstlength)
     {
     const unsigned char *end   = ptr + srclength;
     const unsigned char *limit = dst + dstlength;
     while (ptr < end)
       {
       int z, c = Rx(*ptr++);

       if (c == Error || c == Resume || c == Reserved) return(NULL);
       else if (c == Diff)        { z = 0;       c--;       }
       else if (isRunZero(c))     { z = c & 0xF; c = 0;     }
       else if (isDiff2Zero(c))   { z = 2;       c &= 0x1F; }
       else                       { z = 1;       c--;       }

       while (--c >= 0 && dst < limit) *dst++ = Rx(*ptr++);
       while (--z >= 0 && dst < limit) *dst++ = 0;
       }
     if (dst < limit) return(dst-1);
     else return(NULL);
     }
```

```
/******* nego_receive.c **************/

#include "project.h"
/*********************************************************/
/******* Data Reception Handling Alogorithm ***************/
/*********************************************************/
int negotiation_receive(int fd, unsigned char role)
{
  int res, suggest_tdetect, suggest_baud, length;
  static unsigned char  rec_seq_no = 1;
  unsigned char suggest_uart, stop_bit, parity, parity_type, ack_code,
use_crc, use_delimit, use_rts_cts, rts_cts_type, use_error_recovery,
ext_byte, suggest_protocol, ack=2, error_type, seq_with_error;
  unsigned char head, readdata, i, buf[MAXLENGTH], cobs[MAXLENGTH],
*ptr;

  STOP = FALSE;
  while (STOP==FALSE) {       /* loop for input */
    read(fd,&head,1);

    if(head==0x7e) {
      readdata = 1;
      i=0;
      while(readdata ==1) {
      read(fd,&head,1);
      if(head != 0x7e) {
      cobs[i]=head;
      i++;}
      else {
        readdata=0;
      }
      }
      length = i;
      /******** Prints out received data ***************/
      printf("Received data ");
      for(i=0;i<length;i++) printf("%x,", buf[i]);
      printf("\n");

      UnStuffData(&cobs[0], length, &buf[0], 10);
      printf("Decoded data ");
      for(i=0;i<10;i++) printf("%x,", buf[i]);
      printf("\n");

    if (buf[0]==NEG_PKT_HDR) {
      STOP=TRUE;
      /*** Handles Negotiation packet ***/
      if(buf[1] != rec_seq_no) {
      printf("Error! packet out of sequence\n");
      /* code to call error */
      negotiation_send(fd, 0, 0, 0, 0, ERR_PKT_HDR, MISS_SEQ_ERROR,
rec_seq_no);
      }

      if(crc(buf, BUFSIZE)) {
      printf("Error! CRC error\n");
      /* code to call error */
      negotiation_send(fd,  0,  0,  0,  0,  ERR_PKT_HDR,  CRC_ERROR,
rec_seq_no);
      }

      else {
```

```
      rec_seq_no++;

      /*** read in values ***/
      use_crc = (buf[7] & 0x01) ? TRUE : FALSE;
      use_delimit = (buf[7] & 0x02) ? TRUE : FALSE;
      use_rts_cts = (buf[7] & 0x04) ? TRUE : FALSE;
      rts_cts_type = (buf[7] & 0x08) ? TRUE : FALSE;
      use_error_recovery = (buf[7] & 0x10) ? TRUE : FALSE;

        stop_bit = (buf[2] & 0x04) ? 2 : 1;
        parity = (buf[2] & 0x08) ? TRUE : FALSE;
        parity_type = (buf[2] & 0x10) ? EVEN : ODD;
      ack_code = ((buf[2] & 0xE0) >> 5);
      ext_byte = (buf[7] & 0xE0) >> 5;
        if (ext_byte) {
        printf("extended bytes %d \n", ext_byte);}
      /*** futher code added in the future ***/
      /*** for extended negotiation parameters ***/

      /*** assuming all parameters are acceptable ***/
      printf("ack_code %x\n", ack_code);
      if (ack_code == 0) {
         if (role==HC) {
             ack = 1;}
         /* accept */
        else printf("error HC cannot request!\n");
      }
      else if(ack_code == 1) {
           if (role==HC) return suggest_baud;
         else ack = 1; /* accept */
      }
      else if(ack_code == 2) {
        /* recheck code for compatability */
        ack = 1;
      }

        /* accept */
      else {printf("Reserved ack code\n"); exit(-1);}

        suggest_uart = SET_UART(0, stop_bit-1, parity, parity_type,
ack);
      suggest_baud = 27648000/(buf[3]+(buf[4]<<8));
      suggest_tdetect = buf[5]+(buf[6]<<8);
      suggest_protocol     =     PROTOCOL_MODE(use_crc,     use_delimit,
use_rts_cts, rts_cts_type, use_error_recovery, ext_byte);

       sleep(suggest_tdetect/1000); /* wait after Tdetect time */

        negotiation_send(rd,         suggest_uart,         suggest_baud,
suggest_tdetect, suggest_protocol, NEG_PKT_HDR, 0, 0);

/* checks can be included here to simulate rejected connection */

      }

    //    else printf("Not HCI negotiation packet\n");


    }
     if (buf[0]==ERR_PKT_HDR) {
       /*** Handles Error Packet ***/
```

```
      if(buf[1] != rec_seq_no) {
      printf("Error! packet out of sequence\n");
      negotiation_send(fd, 0, 0, 0, 0, ERR_PKT_HDR, MISS_SEQ_ERROR,
rec_seq_no);
      }
      else {
      error_type = buf[2];
      if(error_type == MISS_RTX_PKT_ERROR) {
        /*** Missing retransmission packet ***/
        /*** Do nothing, let higher layer settle ***/
        printf("Retransmission packet not available.\n");
      }
      else
        {
        seq_with_error = buf[3];
            negotiation_send(fd,   0,   0,   0,   0,   RETRANSMIT,   0,
seq_with_error);
        rec_seq_no++;
        }
      }
     }
     if (buf[0]==CMD_PKT_HDR) {
      STOP=TRUE;
       /*** Handles Command Packet ***/
       /*** implemented in the future ***/
     }
     if (buf[0]==ACL_PKT_HDR) {
      STOP=TRUE;
       /*** Handles ACL Data Packet ***/
       /*** implemented in the future ***/
     }
     if (buf[0]==SCO_PKT_HDR) {
      STOP=TRUE;
       /*** Handles SCO Data Packet ***/
       /*** implemented in the future ***/
     }
     if (buf[0]==EVN_PKT_HDR) {
       STOP=TRUE;
       /*** Handles Events Packet ***/
       /*** implemented in the future ***/
     }

    }
  }
   STOP = FALSE; /* reset stop */
  /* the following was used for debugging */
/*       printf("Buffer has \n %x \n %x \n %x \n %x \n %x \n %x \n %x
\n %x \n", buf[0], buf[1], buf[2],  buf[3], buf[4], buf[5], buf[6],
buf[7]); */
  if((ack==1) && (role == HOST)) return  suggest_baud;
  else return 0;
}
```

```
/****** nego_sent.c *******/
#include "project.h"

void negotiation_send(int fd, unsigned char suggest_uart, int
suggest_baud, int suggest_tdetect, unsigned char suggest_protocol,
unsigned char pkt_type, unsigned char err_type, unsigned char
seq_with_error)
{
  static unsigned char sequence_no = 0; /* limit of 2^8=255 packets in
buffer*/
  unsigned char i, neg_byte[BUFSIZE], cobs[MAXLENGTH], *stuff=NULL,
*op;
  int fcs, length;
  unsigned char tx_buf[256];

  if(pkt_type==NEG_PKT_HDR) {
    /*** Pack and sent Negotiation Packet ***/
    neg_byte[0] = NEG_PKT_HDR;
    sequence_no++;
    neg_byte[1] = sequence_no;
    neg_byte[2] = suggest_uart; /*resv,stop,parity,type, Ack*/
    neg_byte[3] = (BAUD_TO_N(suggest_baud) & 0xff);
    neg_byte[4] = ((BAUD_TO_N(suggest_baud) >> 8) & 0xff);
    neg_byte[5] = suggest_tdetect; /* Tdetect value LSB */
    neg_byte[6] = (suggest_tdetect >> 8); /* Tdetect value MSB */
    neg_byte[7] = suggest_protocol; /*CRC, delimiter ... ext */
    length = 8;

  }
  if(pkt_type == ERR_PKT_HDR) {
    neg_byte[0] = ERR_PKT_HDR;
    sequence_no++;
    neg_byte[1] = sequence_no;
    neg_byte[2] = err_type;
    neg_byte[3] = seq_with_error;
    length = 4;
  }
  if(pkt_type==RETRANSMIT) {
    seq_buf = &tx_buf[0];
    seq_buf += ((seq_with_error - 1)*8);
    neg_byte[0] =  *seq_buf;
    seq_buf++;
      switch (neg_byte[0]) {
      case ERR_PKT_HDR:
      length = 4;
      break;
      case NEG_PKT_HDR:
      length = 8;
      break;
      default:
      printf("HCI higher layer packet, read the packet length\n");
        /*** for future development ***/
      break;
      }
      for(i=1; i<length; i++) {
      neg_byte[i] = *seq_buf;
      seq_buf++;
      }
  }

  /*** Calculate and append CRC ***/
```

```
    fcs = crc(neg_byte, length);
    neg_byte[length]=((fcs >> 8) & 0xff);
    length++;
    neg_byte[length]=(fcs & 0xff);
    length++;
    if(pkt_type != RETRANSMIT) {
      /*** store in Tx buffer ***/
    seq_buf = &tx_buf[0];
    seq_buf += ((sequence_no - 1)*8);

     for(i=1; i<length; i++) {
        *seq_buf = neg_byte[i];
      seq_buf++;
      }
    }

    /*** BOF delimiter ***/
    cobs[0]=0x7e;
    /*** COBS coding ***/
    op = StuffData(neg_byte, length, &cobs[1], &stuff);
    /*** EOF delimiter ***/
    *op = 0x7e;
    length = ++op  - &cobs[0];

    printf("sent data:");
    for(i=0; i != 10; i++) {
      printf(",%x",neg_byte[i]);
    }
    printf("\n");

    printf("coded data:");
    for(i=0; i != length; i++) {
      printf(",%x",cobs[i]);
    }
    printf("\n");

  write(fd, cobs, length);

}
```

```
/******* Header file for stuff.c and unstuff.c ***********/
#include "common.h"

/*****************************************************************/
/*** Consistent Overhead Byte Stuffing and Unstuffing Algorithm ***/
/*** Carlson J, et al 1997 ******************/
/*****************************************************************/

//typedef unsigned char u_char;    /* 8 bit quantity */
typedef enum
    {
    Unused       = 0x00,   /* Unused (framing character placeholder)
*/
    DiffZero     = 0x01,   /* Range 0x01 - 0xCE:
*/
    DiffZeroMax  = 0xCF,   /* n-1 explicit characters plus a zero
*/
    Diff         = 0xD0,   /* 207 explicit characters, no added zero
*/
    Resume       = 0xD1,   /* Unused (resume preempted packet)
*/
    Reserved     = 0xD2,   /* Unused (reserved for future use)
*/
    RunZero      = 0xD3,   /* Range 0xD3 - 0xDF:
*/
    RunZeroMax   = 0xDF,   /* 3-15 zeroes
*/
    Diff2Zero    = 0xE0,   /* Range 0xE0 - 0xFE:
*/
    Diff2ZeroMax = 0xFE,   /* 0-30 explicit characters plus 2 zeroes
*/
    Error        = 0xFF    /* Unused (PPP LCP renegotiation)
*/
  } StuffingCode;

   /* These macros examine just the top 3/4 bits of the code byte */
#define isDiff2Zero(X) (((X) & 0xE0) == (Diff2Zero & 0xE0))
#define isRunZero(X)   (((X) & 0xF0) == (RunZero   & 0xF0))

   /* Convert from single-zero code to corresponding double-zero code
*/
#define ConvertZP (Diff2Zero - DiffZero)


   /* Highest single-zero code with a corresponding double-zero code
*/
#define MaxConvertible (1 ? Diff2ZeroMax - ConvertZP : 0)

   /* Convert to/from 0x7E-free data for sending over PPP link */
static unsigned char Tx(unsigned char x) { return(x == 0x7E ? 0 : x);
}
static unsigned char Rx(unsigned char x) { return(x == 0 ? 0x7E : x);
}


#define FinishBlock(X) (*code_ptr = Tx(X), code_ptr = dst++, code =
DiffZero)
```

```
/***** common.h *******/
/***** common libraries ********/
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>

/****** project.h **********/
/*************************************************************************/
/******* Header files for Communication Emulation ******************/
/*************************************************************************/

#include "common.h"

#include <sys/ioctl.h>

#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define FALSE 0
#define TRUE 1
#define ODD 0
#define EVEN 1
#define HOST 0
#define HC 1
/*************************************************************************/
/******* Packet types: ********************************************/
/*************************************************************************/
#define RETRANSMIT 0x00 /*** not part of the specs ***/
#define CMD_PKT_HDR 0x01
#define ACL_PKT_HDR 0x02
#define SCO_PKT_HDR 0x03
#define EVN_PKT_HDR 0x04
#define ERR_PKT_HDR 0x05
#define NEG_PKT_HDR 0x06

/*************************************************************************/
/******* initial negotiation settings *****************************/
/*************************************************************************/
#define DEFAULTBAUD B9600
#define BAUD_TO_N(x) (27648000/(x))
#define  PROTOCOL_MODE(a,b,c,d,e,f)  ((((f)&7)<<5)  +  (((e)&1)<<4)  +
(((d)&1)<<3) \
                                    +  (((c)&1)<<2)  +  (((b)&1)<<1)  +
((a)&1))
#define   SET_UART(a,b,c,d,e)    ((((e)&7)<<5)    +    (((d)&1)<<4)    +
(((c)&1)<<3) \
                        +  (((b)&1)<<2) +  ((a)&3))

#define BUFSIZE 10
#define MAXLENGTH 16 /* max length of negotiation in bytes */

/*************************************************************************/
/******* Error types **********************************************/
/*************************************************************************/
#define OVER_RUN_ERROR 0x09
#define PARITY_ERROR 0x09
#define FRAME_ERROR 0x09
#define CRC_ERROR 0x09
#define MISS_SEQ_ERROR 0x09
#define MISS_RTX_PKT_ERROR 0x09
```

```
volatile int STOP;

unsigned char *seq_buf; /* tx buffer for error recovery */

struct termios set_port (int baud);

void negotiation_send(int fd, unsigned char suggest_uart, int
suggest_baud, int suggest_tdetect, unsigned char suggest_protocol,
unsigned char pkt_type, unsigned char err_type, unsigned char
seq_with_error);

int negotiation_receive(int fd, unsigned char role);

unsigned short crc(unsigned char *buf, int len);

 unsigned char *StuffData(const unsigned char *ptr, unsigned int
length, unsigned char *dst, unsigned char **code_ptr_ptr);

 unsigned char *UnStuffData(const unsigned char *ptr, unsigned int
srclength, unsigned char *dst, unsigned int dstlength);
```