Real time MP3 transmission over bluetooth ACL link

Feng, Chen

2005

Feng, C. (2005). Real time MP3 transmission over bluetooth ACL link. Master' s thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/46878>

<https://doi.org/10.32657/10356/46878>

Nanyang Technological University

Downloaded on 26 Apr 2023 17:56:21 SGT

a356925 ✓

REAL TIME MP3 TRANSMISSION OVER BLUETOOTH ACL LINK

FENG CHEN

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2005

TK
7882
.C56
F332
2005

Table of Contents

Acknowledgements	i
Summary	ii
List of Figures	
List of Tables	
1 Introduction	
1.1 Motivation	1
1.2 Objectives	5
1.3 Major contribution of the Thesis	5
1.4 Organization of the Thesis	6
2 Bluetooth Protocol Architecture	
2.1 Introduction	8
2.2. Radio	9
2.3 Baseband	12
2.4 Link Manager Protocol	23
2.5 Host Controller Interface	25
2.6 Logical Link Control and Adaptation Protocol	30
2.7 Service Discovery Protocol	32

2.8 Rfcomm	34
2.9 Conclusions	35

3 The MP3 Standard

3.1 Introduction	37
3.2 The MP3 Encoding Process	38
3.2.1 Overview	38
3.2.2 Compression Stages	40
3.3 The MP3 Decoding Process	41
3.3.1 Bitstream Format	42
3.3.2 Frame Synchronization	46
3.3.3 Huffman Decoding	46
3.3.4 Descaling	48
3.3.5 Reordering	50
3.3.6 Stereo Processing	51
3.3.7 alias Reconstruction	52
3.3.8 Inverse Modified Dct (Imdct)	53
3.3.9 Frequency Inversion	55
3.3.10 Synthesis Polyphase Filter Bank	55
3.4 Conclusions	56

4 System Design and Implementation

4.1 System Design	58
-------------------	----

4.1.1 Development Platform and Programming Language	59
4.1.2 Flow Chart for Connection, Receiving, and Transmission	63
4.1.3 Design Considerations	69
4.2 System Implementation	73
4.2.1 Development Work in the First Stage	73
4.2.2 Development Work in the Second Stage	76
4.3 Modifications Made To the Stack	78
4.4 Conclusions	79

5 Experiment and Results

5.1 Experiment on Bluetooth ACL Link	80
5.1.1 Data Rate vs. Bluetooth ACL Packet Size	80
5.1.2 Data Rate Vs. Distance between Transmitter and Receiver (Without Interference)	84
5.1.3 Data Rate Vs. Distance between Transmitter and Receiver (With Interference)	85
5.1.4 Comparison between the Results in Section 5.1.2 and 5.1.3	86
5.1.5 Comparison with Result of Other Paper	89
5.2 Experiments on Memory Usage on Crd7309 Board	91
5.3 Conclusions	93

6 Conclusions and Recommendations

6.1 Conclusions	94
-----------------	----

References

Appendix A. An Overview of the Bluetooth Software Requirements for Users of the Bluecore[™] ICs

Appendix B. Sample Source Code for Bluecore2 Module on Upper Layer

Appendix C: Crd7309c Hardware Guide

Appendix D: Source Code on Crd7309c Board for Mp3 Decoding

Acknowledgements

First of all, I would like to thank all people who have contributed directly or indirectly in guiding or supporting me throughout the period of my candidature. I would like to extend my greatest gratitude to the following people:

- a) My sincere thanks to my project supervisors, Associate Professor Gan Woon Seng and Assistant Professor Farook Sattar. They have offered me a great opportunity and environment in pursuing my research. They have also given me advices, guidance, knowledge and encouragements during this research.
- b) Special thanks to my girl friend Lu Rui, who always gives me encourage and support, and brings a lot of fun to my life.
- c) Big thanks to all the colleagues and friends who have worked in the Digital Signal Processing Laboratory during my research, especially Mr. Furi Andi Karnapi, Mr. Lee Kong Aik, Mr. Tan Khim Sia and Mr. Zhang Hua, Mr. Kelvin Lee, and other friends. They have given many advices and made the place interesting for doing research.

Summary

Current Bluetooth specification does not support real time transmission of high quality audio signal due to the bandwidth limitation. There are two links exist in Bluetooth communication: SCO (synchronous connection oriented) link and ACL (asynchronous connectionless link) link. SCO link is for synchronous transmission with 64kbps data rate, and ACL link is for asynchronous data transmission, which has higher data rate (up to 723kbps).

This project makes use of Bluetooth ACL link to do the real time MP3 streaming. Specially, an MP3 server, which can stream MP3 to client, is implemented. A New ACL link scheme (connection, transmitting, and receiving) is implemented. And a client, which can real time decoding the incoming MP3 data is implemented.

The important contribution of the project is to establish a high speed, reliable and stable communication channel between the server and the mobile device so that the MP3 data can be real time streamed.

List of Figures

Figure 1.1 Point-to-point audio and user-controlled information transmission and playback	5
Figure 2.1 Bluetooth Protocol Stack	9
Figure 2.2 Frequency Hopping	11
Figure 2.3 Piconet and Scatternet	14
Figure 2.4 Standard Packet Format	16
Figure 2.5 Link Controller State Diagram	21
Figure 2.6 LMP Message Sequence For ACL Link Setup	24
Figure 2.7 Structure of a HCI command packet	27
Figure 2.8 HCI Data Packet Structure	29
Figure 2.9 Communication between Host and Bluetooth Module	30
Figure 2.10 Structure of a L2CAP packet	33
Figure 2.11 SDP Client/Server	34
Figure 2.12 RFCOMM Frame Structure	36
Figure 3.1 Perceptual masking of audio signals	39
Figure 3.2 Overview of the encoding process	39
Figure 3.3 Detailed encoding process	40
Figure 3.4 The MP3 decoding process	42
Figure 3.5 The bitstream format	43
Figure 3.6 Header information	43
Figure 3.7 The use of the bit reservoir	45

Figure 3.8 The frequency lines are divided into three partitions	48
Figure 3.9 The descaling process	49
Figure 3.10 Short and long windows used by the MDCT and the IMDCT	50
Figure 3.11 Reordering of short block types into continuous frequency lines	50
Figure 3.12 Alias reduction diagram	53
Figure 3.13 IMDCT with windowing and overlap add	54
Figure 3.14 Synthesis subband filter flow chart	56
Figure 4.1 Overview of System Design	59
Figure 4.2 Casira Bluetooth Development kit	60
Figure 4.3 Cirrus Logic CRD7309C board	61
Figure 4.4 Messages for Setting Up a Connection over HCI	64
Figure 4.5 Create ACL Connection	65
Figure 4.7 Functions For Transmitting Data	67
Figure 4.6 Receiving State Machine	68
Figure 4.8 Constructs in Double Buffer algorithm	71
Figure 4.9 HostController application interface	74
Figure 5.1 Experiment Setup	81
Figure 5.2 Experiment Setup	85
Figure 5.3 Data rate against distance between transmitter and receiver (without interference)	84
Figure 5.4 Data Rate against distance between transmitter and receiver (with interference)	86
Figure 5.5 Typical Wi-Fi (IEEE 802.11b) baseband Signal Power	87

Figure 5.6 Typical Bluetooth Signal Power Spectrum	88
Figure 5.7 Comparison between the data rate	88
Figure 5.8 Variation of the measured bitrate with the ACL packet type, the application packet length and the distance (no interference exist)	90
Figure 5.9 Variation of the measured bitrate with the distance for different application packet lengths (with the existence of interference and no interference)	91

List of Tables

Table 2.1: Summary of ACL packets	19
Table 4.1: Transmission Rates and Buffer Size Required for Different Bit Rates	70
Table 5.1. Data rate on Different Packet Size	82
Table 5.2. ROM & RAM Usage on CRD 7309 Board	92

Chapter1. INTRODUCTION

1.1 Motivation

In 1994 Ericsson started an investigation concerning how to design low cost radio modules that could transfer information wirelessly between mobile phone and their accessories. The Ericsson people soon realized that a short-range radio link solution was feasible but they needed companies to develop this technology. In 1998, some other established companies, such as Intel, IBM, Nokia Mobile Phone and Toshiba, joint with Ericsson and formed a Special Interest Group (SIG) to develop a new industrial standard for short-range wireless communication. And this new technology was named Bluetooth.

Bluetooth operates in the Industrial-Scientific-Medical (ISM) radio frequency band (around 2.4 GHz), [1] and the range for communication between two modules in an unobstructed space is about 10 meters. Bluetooth can be used to transfer data and voice, and the encryption and error correction functionalities it provided make the linkage secure and reliable.

Although Bluetooth is an attractive wireless specification for developing cable-free PAN (personal area network) products, the bandwidth limitation together with the nature of the permitted wireless links between the connected devices present some major drawbacks. More specifically: Time-bounded applications requiring a constant throughput must be

established through SCO links. The effective bandwidth in such a case is 64kbps (or 192kbps for 3 SCO links). [1]

Especially for wireless audio transmission, Bluetooth specification defines either a 64kbps logarithmic PCM format (A-law or μ -law) or a 64kbps Continuous Variable Slope Delta Modulation (CVSD) through SCO connections.

- Time-bounded applications requiring a constant throughput must be established through SCO links. The effective bandwidth in such a case is 64kbps (or 192kbps for 3 SCO links).
- Especially for wireless audio transmission, Bluetooth specification defines either a 64kbps logarithmic PCM format (A-law or μ -law) or a 64kbps Continuous Variable Slope Delta Modulation (CVSD) through SCO connections.
- The establishment of SCO links creates a bandwidth overhead, which dramatically decreases the transfer capabilities of any co-existing ACL link.
- The packet-switched nature of the ACL links with the limited bitrate is not suitable for real-time applications. Especially in noisy environments, packet retransmissions are applied to ensure data integrity, which further reduce the effective bandwidth.

On the other hand, it is well known that high quality digital audio reproduction is performed at bit rate in the range of Mbps (nearly 1.4Mbps for stereo CD quality). [2] Given the above practical limitations of the Bluetooth standard, the following

considerations are necessary for realizing high-quality, real-time audio reproduction through Bluetooth:

- The original audio data must be compressed prior to the transmission. As with many other applications, the choice of the compression ratio is a tradeoff between the desired audio quality and the available bandwidth. A number of ISO/MPEG standards (MPEG-1, AAC, etc.), as well as many propriety standards (Dolby Digital, ATRAC, WMA, DTS, etc.), provide a wide range of compression options for mono, stereo and multichannel applications. However, for all cases, acceptable quality is achieved for rates of 96kbps or higher.
- Apart from the transmitted audio data, many applications require the transmission of control information (e.g. volume control, timing info etc), which must be multiplexed with the compressed audio data.

Some possible application environments for real-time, compressed quality digital audio reproduction were implemented and are presented in [3]. The simulation and some experiment result show the feasibility of real time transmission of compressed audio over Bluetooth ACL link.

Some other papers related to the Bluetooth ACL transmission have been published in IEEE ([4] & [6]). In [4], the author raised some new SAR (segmentation and reassemble) mechanism, and some scheduling algorithm. However, the focus of the paper is to enhance the performance of ACL link over a Bluetooth Wireless network. Therefore,

what the paper concern is still data traffic, not the real time audio transmission. This project was carried on simulation level, and the software used is the Network simulator and Matlab package.

In [5], the author discusses the feasibilities of carrying voice over ACL links. The reason is only to save the bandwidth sharing in the piconet¹. Therefore, the TCP connection in the network will perform much better if SCO link is not used. The authors have done some hardware experiment, and collected some data for comparison.

The transfer of MP3 data to the decoder should be carried out over Bluetooth radio communication. During the work in this Master thesis the Advanced Audio Distribution Profile (A2DP) in the Bluetooth protocol was under development and only version 0.9 of the profile description was available. The payload format was not determined in this version of the profile. Because of this the decision to make a new format that suites the application was taken. This format communicates through the L2CAP layer and is described in section 4.

In [3], it only gives the simulation result and discussion of feasibility. This project is to implement the real time audio streaming over Bluetooth ACL link.

¹ A type of ad hoc network for Bluetooth device, will be introduced in chapter 2.

Figure 1.1 shows the basic system setup of this project.

Fig. 1.1. Point-to-point audio and user-controlled information transmission and playback.

1.2 Objectives

This project intends to develop a low cost gateway that can do real time streaming of high fidelity audio signal to the mobile devices via Bluetooth link.

1.3 Major Contribution of the Thesis

As mentioned above, although Bluetooth is an attractive wireless specification for developing cable-free PAN products, the bandwidth limitation is the major drawback, which make it impossible to transmit high quality audio signal in real time under current Bluetooth standard. [3] gives simulation results of using the Bluetooth ACL link for high quality audio signal for real time transmission. The project implemented the real Bluetooth system, which uses the Bluetooth ACL link do real time MP3 streaming.

The major contribution of the project:

- Established reliable and stable Bluetooth server;
- Established Bluetooth client, which can do real time decoding;
- Established a reliable, high speed Bluetooth ACL link between the server and the mobile client (connection, transmitting, receiving), so that the MP3 signal can be streamed real time;
- Established memory control at the receiving side. Double buffering technique is applied due to the limited resource at the receiving side.

The Bluetooth module is developed on the Casira Development Platform Kit designed by CSR. The transmitting side is a PC, and the receiving side, which should be mobile, is simulated by using a PC and a DSP board (CRD7309) respectively in two development stages respectively.

1.4 Organization of this Thesis

This thesis consists of 6 chapters. In Chapter 1, the introduction of this project is given, which includes the background, the objective of this project, the literature review and the report organization. Chapter 2 and Chapter 3 introduce the basic knowledge of Bluetooth protocol stack, and MP3 standard. Chapter 4 is the system design and implementation, detailed information of the development and problems encountered will be given. Chapter 5 presents the results obtained during the project, and some comparison with the

result from others and discussion is also included. Chapter 6 summarizes the report with conclusions and recommendations for future work.

Chapter 2. BLUETOOTH PROTOCOL ARCHITECTURE

In this chapter, the Basic Bluetooth protocol architecture is introduced. The functions of each Bluetooth stack layers, the flow of packets, and the comparison of different types of packets are all introduced in this chapter.

2.1 Introduction

The Bluetooth protocol stack can be grouped into three sections, the transport protocol group, the middleware protocol group and the application group. The transport protocol group is composed of protocols managing the physical and logical links between Bluetooth units. The middleware protocol group contains both special developed Bluetooth protocols and other adopted protocols such as PPP², IP, TCP, OBEX³, etc. Some of those adopted protocols are functional duplicates of the protocols in the transport protocol group but implemented since they are used by many applications. The application group consists of the applications that use Bluetooth links.

Figure 2.1 gives an overall picture of the Bluetooth protocol stack. Some protocols are discussed briefly in the following:

² PPP stands for Point-to-Point Protocol

³ OBEX stands for Object Exchange Protocol

Figure 2.1: Bluetooth Protocol Stack.

2.2 Radio

Radio is a part of the physical layer of the Bluetooth protocol stack. It is responsible for the reception of data from the baseband, perform modulation on them to achieve a certain radio frequency and transmission of the modulated signal. It is also responsible for the reception, demodulation, transmission to baseband.

Bluetooth devices operate in the ISM band, which is globally available and license-free, situated at 2.4 GHz. This band is reserved for general use by industrial, scientific and medical applications.

There are plenty of short-range wireless devices, such as the wireless local area network (WLAN) applications (IEEE802.11), operating in this band. Other electronic devices, such as microwave ovens, also emit interference in this band. As a result, the ISM band is not a reliable medium. In order to communicate properly in such a noisy environment, several special techniques, such as frequency hopping, adaptive power control and short data packets, are employed by Bluetooth.

2.2.1 Frequency Hopping

Frequency Hopping is one of the spread spectrum techniques [5]. The data signal is modulated with a carrier signal that hops to different frequencies as a function of time over a wide band of frequencies. This process is briefly shown in Figure 2.2 below.

Figure 2.2: Frequency Hopping.

With the frequency hopping technique, the data is transmitted in different frequency slots at different time slots. This characteristic significantly improves the robustness of data transmission. Usually, an interference source will only temporarily block a certain frequency slot; data that are transmitting in this particular frequency slot will be affected and become lost. The lost data will then be retransmitted in a different frequency slot, which is unlikely to affect by the interference source.

Bluetooth divides the ISM band into 79 slots. Each slot has 1 MHz bandwidth (except in France, where the ISM band is narrower and can only be divided into 23 slots) [6]. The hopping sequence is determined by the master and is needed to maximize the distance between two adjacent hop frequency slots.

2.2.2 Modulation

Bluetooth adopts the Gaussian Frequency Shift Keying (GFSK) scheme to perform the modulation [6]. A binary “1” gives rise to a positive frequency deviation from the nominal carrier frequency, while a binary “0” gives rise to a negative frequency deviation. With a 1 MHz bandwidth per frequency slot, the maximum raw transmission rate is 1 Mbits/sec.

2.2.3 Power Classes and Control

The *Bluetooth* specification defines three power classes (max output power) of 100mW, 2.5mW and 1mW, respectively [6]. The maximum operation distance between the two *Bluetooth* devices is up to 100m (for the 100mW output power).

Power control is achieved by monitoring the received signal strength indication (RSSI) at the receiver and sending Link Manager Protocol (LMP) control commands to the sender. If the RSSI value is too high, the LMP control commands will ask the sender to reduce the transmit power, while if the RSSI value is too low, the LMP control commands will request the transmit power to be increased. By using the power control, the power consumption can be minimized, and the transmission quality can be maintained.

2.3 Baseband and Link Controller

The Baseband and Link Controller layer enables physical link among *Bluetooth* units, controls the *Bluetooth* unit's synchronization, transmission frequency and hopping sequence.

2.3.1 Bluetooth Device Address

A 48-bit IEEE MAC⁴ address is assigned to each *Bluetooth* device and it is known as *Bluetooth* device address (BD_ADDR) [6]. The 48-bit *Bluetooth* device address is split

⁴ MAC stands for medium access control.

into 3 parts. They are: (i) Non-significant address part (NAP) (BD_ADDR [47:32]), which is used in the encryption; (ii) Upper address part (UAP) (BD_ADDR [31:24]), which is used to initialize the HEC and CRC calculations and for frequency hopping; and (iii) Lower address part (LAP) (BD_ADDR [23:0]), which is used for synchronization and frequency hopping.

2.3.2 Masters, Slaves, Piconets and Scatternets

Each *Bluetooth* device can be either configured as a master or a slave at any time. If it is configured to be a master, the *Bluetooth* device initializes the data exchange and determines the frequency hopping sequence. If it is a slave, the *Bluetooth* device responds to the master and synchronizes to the master's frequency hopping sequence.

Piconet is defined as a collection of slave devices operating together with one common master device, which is the owner of the piconet. Slave devices within the piconet only have links to the master. There is no direct links between slaves. Figure 2.3 (b) shows a piconet with three slaves. The number of slaves within a piconet is limited to seven by the specification. If the number of slaves is one, then that is equivalent to a point-to-point connection (Figure 2.3(a)).

Piconets can be linked together to form a scatternet. In order to link up the piconets, some devices will be members of more than one piconet. A scatternet can cover a larger area and have more network members. In Figure 2.3 (c), one of the devices is the Master of a

piconet with three Slaves, at the same time, it is also a Slave of another piconet. All these devices form a scatternet.

Figure 2.3: Piconet and Scatternet.

Bluetooth devices use time division multiplexing (TDM) technique to allow the master to communicate with several slaves. Piconet and scatternet operations are not supported by the Bluetooth devices developed.

2.3.3 Physical Links

Two types of links can be established between two Bluetooth devices (one configured as the master, and the other as slave). These two types of link are defined as **synchronous connection oriented (SCO)** link and **asynchronous connection less (ACL)** link [6] [7] [8].

2.3.3.1 SCO Link

The SCO Link is a symmetric, point-to-point link between the two *Bluetooth* devices (the master and the slave). This type of link is mainly used for transmitting voice, as in the PCM⁵ format.

2.3.3.2 ACL Link

The ACL link provides a packet-switched connection between the master and all active slaves in a piconet. It carries data to and from L2CAP or LMP layers

2.3.4 Packet Structure

Data transmission over the channel is conveyed in packets. Each packet consists of three parts, namely, the access code, the header and the payload. The length of the access code and the header are 68 or 72 bits, and 54 bits, respectively. The length of the payload can vary from 0 bit to 2745 bits for ACL link, and it is fixed to 240 bits for SCO link. The format of a general packet is shown in the figure 2.4 below.

Figure 2.4: Standard Packet Format.

⁵ PCM stands for pulse code modulation.

2.3.4.1 Access Code

Access code consists of three parts: preamble, synchronization word and trailer. It is derived from the master's *Bluetooth* address (BD_ADDR), therefore it is able to identify that the packet as being to or from a specific master. It is also used for synchronization and DC compensation.

2.3.4.2 Packet Header

Packet header consists of several fields, namely, AM_ADDR⁶, Packet Type, Flow, ARQN, SEQN and header error check (HEC).

AM_ADDR is a member address assigned to the slave by the master when the slave is participating in the piconet. Packet type identifies which type of traffic this packet carries (SCO, ACL, NULL, POLL). Flow, ARQN and SEQN are related to flow control and ARQ. HEC is the check sum for the whole header.

2.3.5 Packet Types

Common Packets

These packets can be recognized by both the SCO and ACL connections [7] [8].

⁶ AM_ADDR stands for active member address.

ID Packet

This packet consists of the access code, and is used in paging, inquiry and response routines.

NULL Packet

The NULL Packet is used to return link information to the source regarding the success of the previous transmission. It does not have any payload.

POLL Packet

The POLL Packet is used by the master device to poll the Slaves within a piconet. The Slaves must respond to this type of packets.

FHS Packet

The FHS Packet is a special control packet, used to reveal the *Bluetooth* device address (BD_ADDR) and the clock of the sender. This information is used for hopping frequency synchronization.

DM (data medium) 1 Packet

DM1 Packet supports the control messages in both the SCO and ACL links. It can also be used to carry normal user data; therefore, it is regarded as ACL packet most of the time.

SCO Packets

SCO packets are used on the SCO link, and can not be retransmitted [7] [8].

HV (High quality Voice) 1 Packet

Typically used for voice transmission. One HV1 packet can carry 1.25ms speech at a rate of 64kbps.

HV2 Packet

Similar to the HV1 packet, but each of these packets is able to carry 2.5ms speech at a rate of 64kbps.

HV3 Packet

Similar to HV1 and HV2 packets, but each of these packets is able to carry 3.75 ms speech at a 64kbps rate.

DV Packet

The DV packet combines data and voice and transmits them together. But the data field and the voice field are treated separately.

ACL Packets

DM1/DM3/DM5 Packet

DM stands for data-medium rate. Information in the payload is Forward Error Correction (FEC) code protected. They may cover 1, 3 and 5 time slots respectively.

DH (Data High) 1/DH3/DH5 Packet

DH here stands for Data High-rate. They are similar to the DM packets, but the information in the payload is not Forward Error Correction (FEC) code protected.

AUX 1 Packet

Resembles of DH1 packet, but it does not have CRC code.

The various types of ACL packet and their characteristics are summarized in Table 2.1 below.

Table 2.1: Summary of ACL packets.

Packet type	Payload header (Byte)	User payload (Byte)	FEC	CRC	Symmetric Max. Rate (kbps)	Asymmetric Max Rate (kbps)	
						Forward	Reversed
DM1	1	0-17	2/3	Yes	108.8	108.8	108.8
DM3	2	0-121	2/3	Yes	258.1	387.2	54.4
DM5	2	0-224	2/3	Yes	286.7	477.8	36.3
DH1	1	0-27	No	Yes	172.8	172.8	172.8
DH3	2	0-183	No	Yes	390.4	585.6	86.4
DH5	2	0-339	No	Yes	433.9	723.2	57.6
AUX1	1	0-29	No	No	185.6	185.6	185.6

2.3.6 Flow Control and ARQ

The 1-bit Flow field in the packet header is used for the *Bluetooth* device to *Bluetooth* device flow control. This bit is set by a device when it cannot receive any more data

because the receive buffer is full. When a device receives a packet with this bit set, it should stop sending data to the device that sets this bit.

The ARQN and SEQN fields in the packet header are used for ARQ [6]. *Bluetooth* adopts an ARQ scheme similar to the stop-and-wait ARQ scheme. The SEQN bit will toggle whenever a new packet is sent. When the receiver receives the packet correctly, it sends back a packet with the ARQN bit set to 1 to indicate a positive acknowledgement (ACK). If the packet is not received correctly, the receiver sends back a packet with the ARQN bit set to 0 to indicate a negative acknowledgement (NAK). If the acknowledgement does not come back before the sender times out, the sender will assume that an NAK is received. When NAK is received, the packet must be retransmitted. In the retransmitted packet, the SEQN bit does not toggle, so that the receiver is able to distinguish the retransmitted packets when identical packets are received. The acknowledgement can be piggybacked in a data packet, or if the receiver has nothing to send back to the sender, a NULL packet is used for carrying the acknowledgement information.

2.3.7 Link Controller States

At any time instances, a Bluetooth device can be in one of many different states [6]. Figure 2.5 is a simplified stat diagram of the link controller.

Figure 2.5: Link Controller State Diagram.

The following explains the states:

(a) Standby state

When it is in the standby state, the device is not active and the radio is switched off.

Hence, no data can be transferred, and the device is not able to detect any access code.

(b) Inquiry state

A device enters the inquiry state to discover all the *Bluetooth* enabled devices within the range. During the inquiry procedure, the inquirer obtains essential information required to make a connection by receiving FHS packets from the device being inquired.

(c) Inquiry Scan and Inquiry Response state

In order to be discovered by other devices that are inquiring, a device enters the inquiry scan state periodically to listen to the inquiry messages. When a valid inquiry message is

received, it enters the inquiry response sub-state and responds to the inquirer with the FHS packet.

(d) Page and Master Response state

To establish a connection, the master enters the page state and carries out the paging procedure. In the page state, the master transmits page messages to the intended slave using information gathered during the inquiry procedure. Once the response from the slave is received, the master enters the master response sub-state and responds with its FHS packet.

(e) Page Scan and Slave Response state

In order to be able to establish a connection, a device enters the page scan state periodically. Once the paging message is received successfully, the device will enter the slave response sub-state where it acknowledges the paging message and waits for the FHS packet. When the FHS packet is received, the device uses it to update its own clock, synchronization word and access code.

(f) Connection state

In the connection state, the slave synchronizes to the master's clock. Hence, the master and the slave are synchronous in frequency hop and timing. After the link is verified, data can be exchanged between the master and the slave.

2.4 Link Manager Protocol (LMP)

LMP receives commands from the Host Controller Interface (HCI), and executes these commands at the baseband level. It is used for link set-up and control between the Bluetooth units. It controls and negotiates the packet types used for a connection with peer LMP, manages the power modes and power consumption. The generation, exchange and control of link and encryption keys for authentication and encryption are also handled by the LMP.

Link Managers communicate with their peers on other Bluetooth devices by sending the LMP protocol data units (PDUs). All LMP PDUs are carried by the single-slot packets. There is no flow control for the LMP PDUs, since the Flow bit in the Baseband packet that carries a LMP PDU is ignored.

2.4.1 Link Setup and Shutdown

The Link Manager is responsible for setting up, managing and shutting down the Baseband connections that link Bluetooth devices [6]. After the link controller finishes the page procedure, the link manager controls the Baseband to establish the ACL link. Figure 2.6 shows how the Link Manager messages are exchanged between the Master and the Slave by setting up an ACL link. If it refuses to accept the connection request, the Slave will reply the Master by an LMP_not_accepted instead.

Figure 2.6: LMP Message Sequence For ACL Link Setup.

On the other hand, the SCO link can be setup between the Master and the Slave once the ACL link is up. Either the master or the slave can shut down a Bluetooth connection at any time. The device that wants to shut down the link just simply sends an `LMP_detach` command to notify the other side.

2.4.2 Multi-slot Packet Control

Single-slot packets are in use by default when the connection is just setup. In order to have higher transmission rate, multi-slot packets are desired. Since it may have connections to multiple slaves, the master is the bottleneck when trying to use multi-slot packets. Therefore, the Link Manager on the Master device controls the use of multi-slot packets. This is done by sending an `LMP_max_slot` command to the Link Manager on

the Slaves. Slaves also can request for the use of multi-slot packet by sending the LMP_max_slot command to the Master. If the Master is able to cater for the requested multi-slot packet, an LMP_accepted will be sent back in reply. Otherwise, an LMP_not_accepted will be sent back instead.

2.5 Host Controller Interface (HCI)

2.5.1 Overview

In most of the cases, the *Bluetooth* stack is split into two parts: The lower part of the stack resides in the *Bluetooth* hardware, and the upper part of the stack resides in a host computer. These two parts can communicate through an uniform interface named as the Host Controller interface. The HCI driver in the host computer sends commands to the firmware resided in the *Bluetooth* hardware to control the *Bluetooth* Hardware. For example, the host computer can send an inquiry command through HCI to the *Bluetooth* hardware to initiate an inquiry procedure. The HCI driver also processes the events returned from the *Bluetooth* hardware, and notifies the upper protocols accordingly.

The HCI driver can operate over USB and UART, as long as a workable transport driver is provided.

2.5.2 HCI Commands

The *Bluetooth* hardware capabilities can be accessed by the HCI command packets sent by the host computer. The *Bluetooth* device reports the status of the command to the host computer by sending back the command status or command complete event.

The structure of these commands is shown in Figure 2.7. The first two bytes are the OPCODE, which is used to identify the type of the command. OGF stands for OPCODE Group Field, it identifies that the command belongs to which group. OCF stands for OPCODE Command Field, it identifies a command within a group. The Parameter Total Length comes next, which shows the length of the parameters in bytes. The final portion is the parameters, different commands will have different number of parameters.

Figure 2.7: Structure of a HCI command packet.

All HCI commands are grouped into 3 groups, which are Link Control Commands, Link Policy Commands, and Host Controller & Baseband Commands [7] [8]. The Link Control Commands allow the host controller to control the connection to other *Bluetooth* devices. The Link Policy Commands provide methods for the host to affect the way that

the Link Manager manages the piconet. The Host Controller & Baseband Commands allow the host to access and control various capabilities of the *Bluetooth* hardware.

2.5.3 HCI Events

Events are generated from the Bluetooth device to notify the host computer when events occur. For example, after the host computer sent an inquiry command, the host controller generates an *Inquiry_results* event to inform the host computer about the detected *Bluetooth* addresses of other Bluetooth devices within the range. And after the predefined inquiry time duration elapsed, the host controller generates an *Inquiry_complete* event to inform the host computer that the inquiry command has completed successfully.

The maximum parameter length of an event is 255 bytes and the parameters carry the information that can be translated by the host computer with the help of the *Bluetooth* specification [7] [8].

Some important events are command status event, command completed event and Num_Of_Completed_Packets event.

2.5.4 HCI Data

HCI data packets carry both SCO and ACL data. The packets used by SCO and ACL are slightly different.

Figure 2.8 (A) shows the HCI data packet structure used by SCO, and Figure 2.8 (B) shows the HCI data packet structure used by ACL. Their structures are very similar. The differences are as follows: the PB and BC fields in HCI ACL data packet become Reserved in HCI SCO data packet; the Data Total Length field is shorter in the SCO data packet, due to the limitation that the SCO packet can not be too long. The PB flag is for fragmenting the L2CAP packets and reassembling them in the peer HCI. The BC flag is for broadcasting.

Figure 2.8: HCI Data Packet Structure.

2.5.5 Flow Control

2.5.5.1 HCI Command Flow Control

The host computer controls the *Bluetooth* device through HCI commands, and the *Bluetooth* device reports what is happening inside itself to the host computer through HCI events. (Figure 2.9)

Figure 2.9: Communication Between Host and Bluetooth Module.

Each time a *Bluetooth* device receives a HCI command, and it responds with an HCI_Command_Complete event, if the command can be executed immediately. Otherwise, it responds with several HCI_Command_Status events during the execution of the command, followed by an HCI_Command_Complete event after it finishes executing the command. These two events contain a parameter called Num_HCI_Command_packets, which indicates that how many commands the device can accept. The host computer is able to send as many commands to the device as the number specified in this parameter. When the system is just powered up, the host computer has no idea about the value of the Num_HCI_Command_packets, therefore, the host computer may send only one command, and waits for the response [6] [7] [8].

2.5.5.2 ACL Data Flow Control: From Host Computer to Bluetooth Module

When the system is powered up, the host computer sends out an HCI command called HCI_Read_Buffer_Size to find out how many buffers the *Bluetooth* device has. Then the

host computer can send up to that number of packet to the *Bluetooth* device. As no more space is available in the device, the host computer has to stop sending and wait for further notification. If there is space available in the device, the device will notify the host computer by sending an `HCI_Number_Of_Completed_Packets` event, which indicates how many packets have been handled since last such an event was sent. Once the host computer knows that the device is able to receive more packets, it will continue to send packets to the device [6] [7] 8].

2.5.5.3 ACL Data Flow Control: From Bluetooth Module to Host Computer

In some cases when the processing capacity of the host computer is limited, the flow control from *Bluetooth* module to the host computer is needed. The command `HCI_Set_Host_Controller_To_Host_Flow_Control` is used to turn on the flow control in this direction. The host computer uses the `HCI_Host_Buffer_Size` command to notify the *Bluetooth* device of the host's buffer space and the `HCI_Host_Number_Of_Completed_Packets` command to inform the device how many packets have been processed. Therefore, the *Bluetooth* device is able to obtain the buffering information of the Host Computer and sends the ACL packets accordingly [6] [7] [8].

2.6 Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP provides connection-oriented and connectionless data services to upper layers. Since a number of protocols, such as RFCOMM, SDP etc, may operate over L2CAP

layer, L2CAP supports higher level protocol multiplexing. Packet segmentation and reassembly and the conveying of quality of service information are supported by L2CAP as well.

2.6.1 Multiplexing Using Channels

Multiplexing is provided in L2CAP to allow several higher layer links to communicate over a single ACL connection. Therefore, several different applications are able to share an ACL link with the higher layers of the Bluetooth stack.

L2CAP uses different channel numbers to identify packets from or to different entities of the higher layers, so that the data can always be routed to the correct place. Two special channel numbers are reserved for special purposes. One is for the L2CAP signaling packets that are used to control the connections. Another one is for receiving multicast packets. Valid channel numbers are assigned to higher layers as connections are setup.

2.6.2 L2CAP Packet

Figure 2.10 shows the structure of a L2CAP packet.

Figure 2.10: Structure of a L2CAP packet.

First two bytes of a L2CAP packet is the Length field, which indicates how many bytes of data the packet contains⁷. This is followed by the channel ID field, which occupies two bytes of space.

The rest of L2CAP packet carries the data from or to higher layers. The packet size is variable from 0 to 65535 bytes. But this figure is also limited by the L2CAP maximum transmission unit (MTU), which is usually 672 bytes.

2.7 Service Discovery Protocol (SDP)

Unlike a normal LAN where the network topology doesn't change a lot, a Bluetooth piconet can change quickly. As a result, services available on the piconet also change quickly. In order for *Bluetooth* devices to find out what services are available within its operating range, a protocol called service discovery protocol (SDP) is specified in the Bluetooth specification.

⁷ Since only 2 bytes are used to define the packet length, the max length is $2^{16}=65536$ bytes

SDP is for locating services provided by or available through a Bluetooth device. With the SDP, a Bluetooth device is able to discover the available services based on specific attributes within its operating range. The SDP also provides functionality for detecting whether a service is no longer available.

SDP adopts a client-server model as shown in Figure 2.11. A SDP server can be any Bluetooth device that offers a service or services to other Bluetooth devices. Every SDP server maintains a SDP database, which stores the information about the services that it can offer. The SDP database is modified when new services are added to the server, or services are removed from the server [6].

SDP clients use services provided by the servers. Client and Server exchange information about services using service records. A Bluetooth device can be a SDP server and client at the same time, because it can provide service to other Bluetooth devices and it also can use services provided by other Bluetooth devices [6].

Figure 2.11: SDP Client/Server.

2.8 RFCOMM Protocol

RFCOMM protocol emulates serial ports over the L2CAP protocol layer. It provides transport capabilities for upper level services. For example, the PPP can operate over the RFCOMM layer to establish a point-to-point connection.

2.8.1 RFCOMM Frame Types and Channels

RFCOMM is based on a protocol used by GSM cellular phones called as GSM TS 07.10. *Bluetooth* adapts some features of the TS07.10 protocol. The basic communication unit used in RFCOMM is the frame. There are five different frame types, such as:

- SABM--- Set Asynchronous Balanced Mode.
- UA--- Unnumbered Acknowledgment.
- DISC--- Disconnect.
- DM--- Disconnected Mode.
- UIH--- Unnumbered Information with Header check.

The usages of SABM, UA, DISC and DM frames are similar to the usages of the unnumbered frames in HDLC. SABM and DISC are the commands used to connect and disconnect. UA and DM are the responses to the SABM and DISC commands respectively. UIH frames are used to send data or control information.

RFCOMM uses channels to communicate with peer RFCOMM. Each channel is identified by a Data Link Connection Identifier (DLCI). Channel with DLCI=0 is reserved for sending control messages. Other Channels can be used to send data [6] [7] [8].

2.8.2 RFCOMM Frame Structure

The RFCOMM frame structure is shown in Figure 2.12.

Figure 2.12: RFCOMM Frame Structure.

A RFCOMM frame starts with the address field, which is used to identify the multiplexed channel that the frame belongs to. The next field is the control field. By looking at the different bit patterns of this field, one can tell the type of the frame. The length field indicates the total length of a RFCOMM frame. Data field only appears in UIH frame, the length of data field is limited by the RFCOMM MTU. A RFCOMM frame ends with the

FCS field, which is the check sum for the address and control fields only. The data field is not protected by this check sum.

2.9 Conclusions

To understand the Bluetooth protocol stack is quite important for the programming on both the host side and the Bluetooth module side. In this project, since the ACL link is used for the real time MP3 transmission, the ACL packet is defined to solve the problems, such as the packet loss, the synchronization at the receiving side, etc. Different packets are tested to achieve better performance, and detailed information is given in chapter 4.

CHAPTER 3. THE MP3 STANDARDS

In this Chapter, the MP3 encoding and decoding process is introduced. For the encoding process, the stages used in MP3 compression are described. For the decoding process, more details are introduced, including the bit stream, Huffman decoding, descaling, reordering, etc.

3.1 Introduction

MPEG-1 layer-III, also known as MP3, is a highly compressed digital audio format. It was standardized in 1991 by the Moving Pictures Expert Group, popularly referred to as MPEG. MPEG was formed in 1988 by the ISO/IEC organizations, to develop generic coding standards of moving pictures, audio and systems containing both. In 1992 the first phase of the work was completed. It is known as MPEG-1, which includes the generic audio coding standard ISO/IEC 11172-3 [9]. The standard defines three audio layers of increasing complexity, called Layer-I, Layer-II and Layer-III respectively. Layer-III uses the highest complexity and provides the highest quality of the three layers, when the same bitrate is used.

With the development of the more powerful MPEG-2 standard, two extensions of MPEG-1 audio were added. The first extension introduced multi-channel coding supporting up to six separate channels instead of normal stereo that uses two channels.

The second extension added support for lower sampling frequencies that improves the performance at low bitrates, i.e. when using high compression [10].

3.2 The MP3 Encoding Process

This section briefly describes the stages used in MP3 compression. A more profound description can be found in the ISO/IEC 11172-3 standard [10]. The sensitivity of the human ear is dependent both on the frequency as well as the energy in the signal. The MP3 compression process is based on a complex acoustic model that removes perceptual irrelevant information in the audio signal.

3.2.1 Overview

An audio signal is made up of frequencies with different amplitudes. If a dominating frequency is present, it suppresses the perception of other frequencies in the vicinity. This phenomenon is called frequency masking and can be used to reduce the amount of data needed to represent the original signal. The threshold that determines if a spectral component of the signal can be removed or not, is called a *masking threshold*.

The masking threshold is calculated from the strongest sound in the audio signal leading to that weak parts will be subdued by the quantization noise, and can, therefore, be removed [10].

Another perceptual limitation is that if a transient occurs in the audio signal, the human ear become less sensitive to frequencies lower than the transient for a short period. In this case, a coarser quantization can be used. This phenomenon is called temporal masking. Both masking types are shown in Figure 3.1.

Figure 3.1 Perceptual masking of audio signals.(from [10])

The MP3 encoder analyzes the audio signal using a perceptual model, also known as a psycho-acoustic model. The digital audio input is fed through an analysis filterbank that extracts the frequency information and arranges it in a number of frequency bands. The perceptual model determines which frequencies are to be omitted and how the remaining frequencies will be encoded. An overview of the encoding process can be found in Figure 3.2. After quantization, the information is coded into a bitstream of digitally compressed sound. Since the encoding technology did not applied in our project, other detailed processes (e.g. Huffman coding) will not be discussed here.

Figure 3.2 Overview of the encoding process.

3.2.2 Compression stages

Figure 3.3 shows a detailed flowchart of the encoding process. The encoder is fed with uncompressed audio information in the form of PCM (pulse code modulation) samples. The audio information is processed by a filterbank of bandpass filters, splitting the signal into different frequency bands. At the same time a Fast Fourier Transform (FFT) produces a frequency spectrum of the input samples. The psycho-acoustic model uses the frequency information to find and remove perceptual irrelevant information [11].

Figure 3.3 Detailed encoding process.

The filterbank arranges the samples into 32 subbands and the next step is to transform the samples to the frequency domain using a Modified Discrete Cosine Transform (MDCT).

The MDCT produces 576 frequency lines that are easier to manipulate by the perceptual model, than time domain samples. Before forwarding the 32 subbands from the filterbank to the MDCT, an alias reduction in the time domain is performed. This procedure compensates for the non-ideal bandpass filtering in the filterbank.

Prior to the MDCT, windowing is applied to the samples in each subband. This can be done using either short or long windows, depending on the dynamics in the subband. If short or long windows are used, the subbands are known as short or long blocks, respectively. In the decoding process, long and short blocks will be treated differently. After removing perceptual irrelevance, the audio information is compressed using Huffman encoding. Other important information, describing the encoded audio and how it should be properly decoded, is saved uncompressed in the side information. The Huffman encoded data and the side information are combined into a single bitstream.

3.3 The MP3 decoding process

The decoding process generates output samples from an MP3 bitstream. MP3 is based on small packages, or frames, each producing a few milliseconds of sound output. A frame contains information on how the bitstream should be properly decoded, as well as the compressed audio data. [10]

The audio data is first Huffman decoded, producing symbols that represent 576 scaled frequency lines. The next step in the process is to descale the symbols to normal

frequency lines, using the scalefactors that are provided in the frame. If the frame contains more than one channel of audio information, the stereo decoder regenerates the frequencies for the left and right channels. The last decoder steps produce time samples from the frequency lines using complex calculations by running it through an Inverse Discrete Cosine Transform (IDCT) and a synthesis polyphase filter bank. An overview of the decoding process can be found in Figure 3.4.

Figure 3.4 The MP3 decoding process.(from [10])

3.3.1 Bitstream format

A frame consists of five parts, namely, Header, CRC, side information, main data and the ancillary data. In MP3, each frame contains information to produce 1152 output samples. The bitstream format can be shown in Figure 3.5.

Figure 3.5 The bitstream format.

3.3.1.1 Header

The header, which is a 32-bit field, contains information about the format of the frame. It starts with a synchronization word that is used to find the beginning of a frame. The header also contains information about the bitstream, including layer number, bitrate, sampling frequency and stereo encoding parameters. Some of the information is essential for the decoding process, like the bitrate index and sampling frequency. Other information, like the copyright and original bits, is not used by the decoder. This information should only be presented to the user.[11]

Figure 3.6 Header information.

3.3.1.2 Cyclic Redundancy Check (CRC)

It is possible to protect the header with a 16-bit CRC field. This is an optional feature, and can be omitted. When it is present, the decoder calculates a CRC on the header and compare it with the CRC in the frame. If they do not match, the decoder should start looking for a new syncword. It is not always a bad frame that causes the CRC error. If a non-valid synchronization word has been found, the CRC check will most likely fail.

3.3.1.3 Side information

The side information contains information for decoding and processing the main data. This information is global and is used by all different steps in the decoding process. When Huffman decoding the main data, the information about which Huffman table to use is stored in the side information. The dscaling of the output from the Huffman decoder uses the global scaling values that should be applied to all frequency bands. These parameters can also be found in the side information. The preceding are just two

examples of how the side information is used, however, the side information can be furthered to in the decoding process.

3.3.1.4 Main data

The main data consists of coded scalefactors and Huffman coded data. The scalefactors are used to reconstruct the original frequency lines from the information in the Huffman data. The size of the Huffman data is the only field in a frame that is variable, but the size of the frame is always the same (it can be calculated using the information in the header). This means that there is potentially unused space between the end of the main data and the end of the frame. This extra space is called a bit reservoir. The bit reservoir can be used for saving main data belonging to proceeding frames. When decoding a frame, the start of the main data must be found. There is a 9-bit pointer in the side information that is pointing to the beginning of the main data. Figure 3.7 shows the dynamic placement of the main data between frames. An important observation is that the main data pointer is always less than or equal to zero. In other words: When decoding a frame, the main data is either in the current frame or in the bit reservoir. [11]

Figure 3.7 The use of the bit reservoir.

3.3.1.5 Ancillary data

The ancillary data can hold user-defined information. The decoder does not process this auxiliary information, but sent to the application using the decoder. The ancillary data can hold the song title or optional song information. The drawback is that the ancillary data is placed in the frame by the encoding process and cannot be easily change at run-time. If it is being streamed over the network, live broadcasting information cannot be included in the ancillary data. Today, Internet radio stations often sends broadcasting information with a continuous, predefined interval. Streaming MP3 decoders must be able to handle this case correctly.

3.3.2 Frame synchronization

Before any decoding can take place, the start of the frame must be found. A synchronization word containing a bit pattern of 12 consecutive ones, '1111 1111 1111', is placed at the beginning of each frame header. It is possible that the same bit pattern exists in other parts of the frame as well, and the decoder cannot tell if it has found a synchronization word or just some arbitrary data. If a CRC is present, it can be used to confirm that a new valid header has been found. If the CRC is not present, the decoder can still make modest attempts to confirm that if it is a valid synchronization word by searching for invalid combinations of fields in the header.

The header contains enough information to calculate the size of the frame, which is also the start of the next frame, using the bitrate and frequency fields. If a frame is corrupted,

it is no longer possible to find the exact position of the next frame. This is the reason why the synchronization word is present in every frame.

3.3.3 Huffman decoding

Huffman encoding is a lossless coding scheme that produces Huffman codes from input symbols. The mapping of symbols to Huffman codes is based on the statistic contents of the input sequence. Symbols that occur more frequently are coded with a short code, while symbols that occur less frequently are coded with longer codes. On average, Huffman coding reduces redundancy, but is only possible if some combinations of input sequences are more likely to appear than others.

The decoding procedure is based on several Huffman tables that are used for mapping Huffman codes to symbols. The decoder compares the input sequence with the information in the Huffman table. If it finds a match, the corresponding output symbol is fetched. There is one and only one symbol for every possible input sequence. More information about Huffman coding can be found in [11].

In MP3, there are 32 different Huffman tables. The tables are predefined and based on statistics suitable for compressing audio information. The side information specifies which table to use for decoding the current frame. The output from the Huffman decoder is 576 scaled frequency lines, represented with integer values. The frequency lines are divided into three partitions: big-values, count1 and rzero, as can be seen in Figure 3.8.

Big-values contain the lowest frequency lines and are coded with the highest precision. The range of big-values is ± 15 . The next partition, count1, can only be coded with the values 0, 1 or -1. The last partition, rzero, represents the highest frequencies and is not a part of the bitstream. Instead, rzero represents the frequency lines that have been removed by the encoder, and it should be filled with zeros by the decoder.

The boundaries of the partitions are specified in the side information, and are selected by the psycho acoustic model during the encoding process.

Figure 3.8 The frequency lines are divided into three partitions.

When decoding big-values, the huffman table will produce two frequency lines for each input sequence and the count1 partition will produce four frequency lines for each input sequence.

It is possible to send larger big-values than 15 if extra precision is needed. It is done with a so-called escape sequence, and is only available for big-values. If the Huffman decoder finds the value 15, it assumes an escape sequence is present and reads out a number of bits from the stream. The escape value is directly added to the original value 15. The

number of bits to read for each escape sequence is specified by the Huffman table that is currently in use, and is called *linbits*.

3.3.4 Descaling

The symbols from the Huffman decoder can reconstruct the original frequency lines by using the scalefactors in the side information. In the encoder process, the frequency lines are divided into a scalefactor and a small integer number, normally in the range -15 to 15 . The scalefactors are saved separately and the integer number are Huffman encoded. Together, the scalefactors and the Huffman coded data forms the main data. An overview of the descaling process can be found in Figure 3.9.

Figure 3.9 The descaling process.

There is not one scalefactor for each frequency line. Instead the frequency lines are divided into 21 groups, called scalefactor bands, each using its own scalefactor. The number of frequency lines in each group depends on the frequency band that it is representing. A low frequency scalefactor band contains fewer values than a scalefactor band representing higher frequencies.

The descaling process uses global scaling information from the side information that should be applied to all frequency groups, as well as the 21 scalefactors that should be applied to each group independently.

3.3.5 Reordering

During the encoding process, the MDCT can arrange the output in two different ways. Normally the output from the MDCT is sorted by subbands in increasing frequency. When a short block is decoded, a short window will be used. Figure 3.10 shows short and long windows, applied on short and long blocks respectively. The output will in this case be sorted on subbands, then on windows and then on increasing frequency.[10]

Figure 3.10 Short and long windows used by the MDCT and the IMDCT.

Reordering is only applied on short blocks, sorting the frequency lines first by subbands and then by frequency. [10]

Figure 3.11 Reordering of short block types into continuous frequency lines.

Figure 3.11 shows the reordering of a short block into subbands with increasing frequency. The shading in the subband represents the frequencies. A darker color corresponds to a higher frequency.

3.3.6 Stereo processing

There are four different channel modes in MP3: Single channel, dual channel, MS stereo and Intensity stereo. When using single channel and dual channel, no stereo processing is required. Dual channel means that the two channels are coded independently.

3.3.6.1 MS stereo

Middle/side stereo, or MS stereo for short, is a method to remove redundancy between channels. This is achieved by coding the sum and difference of the signal instead of the left and right channels.

$$L_i = \frac{M_i + S_i}{\sqrt{2}} \quad \text{and} \quad R_i = \frac{M_i - S_i}{\sqrt{2}} \quad (3.1)$$

M_i is the sum of the two channels, and S_i is the difference. L_i and R_i are the output samples.

3.3.6.2 Intensity stereo

Intensity stereo is a more complex method of stereo coding that is divided in a magnitude part and a stereo position part. Intensity stereo only affects the coding of the higher

frequency lines, and the lower frequency lines can still be coded as left/right channel or in middle/side stereo. The calculation steps for producing the left and right channel is:

$$is_ratio = \tan\left(is_pos_{sb} \cdot \frac{\pi}{12}\right) \quad (3.2)$$

$$L_t = L_t \cdot \frac{is_ratio}{1 + is_ratio} \quad (3.3)$$

$$R_t = L_t \cdot \frac{1}{1 + is_ratio} \quad (3.4)$$

The intensity stereo position, is_pos_{sb} , is read from the scalefactor of the right channel.

Together these techniques of removing irrelevant difference between channels are called joint stereo coding.

3.3.7 Alias reconstruction

The encoder applies an alias reduction after the subband synthesis, because alias effects are always introduced after a non-ideal bandpass filtering. The filter merges the frequency lines using eight butterfly calculations for each subband. [10]

Figure 3.12 Alias reduction diagram.

The diagram of the alias reduction can be found in Figure 3.12. The adjacent subbands are weighted in all frequency lines, except for the two centermost frequencies in every subband. The coefficient in the butterfly filter is defined in the ISO/IEC 11172-3 standard [10].

3.3.8 Inverse modified DCT (IMDCT)

The Inverse Modified Discrete Cosine Transform (IMDCT) used in MP3 is an 18-point DCT that produces 36 output values from 18 input values. It does not produce more information than it is provided with, but it disperses the result from the calculation on a

larger number of values. This is why the term modified is being used. The DCT is inverse because it produces time samples from frequency lines.

The transformation from the frequency domain to the time domain is done in conjunction with the synthesis polyphase filter bank described in section 3.3.10. Instead of directly producing time samples, the IMDCT generates polyphase filter subband samples from the input frequency lines. These will later be used for creating time samples.

The 36 calculated values from the IMDCT must be multiplied with a 36-point window before it can be used by the next step in the decoding process. There are four different window types that can be applied to the output samples. The window to use is based on the block type, and the block type can be found in the side information.

The reason for generating 36 output values is that the IMDCT uses a 50% overlap. The lower 18 values are added with the higher 18 values from the previous frame, and used as output. The higher 18 values are then stored and used the same way when the next frame is being decoded.

Figure 3.13 IMDCT with windowing and overlap add.

A full view of IMDCT including windowing and overlap add can be found in Figure 3.13.

The IMDCT calculation is based on the following equation:

$$x_l = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left(\frac{\pi}{2n}\left(2l+1+\frac{n}{2}\right)(2k+1)\right) \quad 0 \leq l < n \quad (3.5)$$

One of four different windows (a normal, short, start or stop window) can be applied to the output from the IMDCT. Each input value is scaled with the corresponding value in the window. The window type is based on the behavior of the input sequence and can be any of the four types.

3.3.9 Frequency inversion

Before proceeding to the synthesis polyphase filter bank, all odd subsamples in all odd subbands must be negated. This is done to compensate for the frequency inversion in the synthesis polyphase filter bank.

3.3.10 Synthesis polyphase filter bank

The synthesis polyphase filter bank, or the subband synthesis, is the final step in the decoding process. It produces 32 PCM samples at a time using the input samples from the filter bank. A complete flow chart of the subband synthesis can be found in Figure 3.14.

Figure 3.14 Synthesis subband filter flow chart. (from [11])

3.4 Conclusions

The MP3 decoding process is a preliminary knowledge for the implementation of MP3 decoding on the Cirrus Logic CRD 7309 board in later stage. It helps to understand the

sample source, monitor the bitstream at different stage, and modify the I/O with the Bluetooth module.

Chapter 4. SYSTEM DESIGN AND IMPLEMENTATION

This project consists of two stages. The chapter introduces the idea of how the systems works, the hardware and the software used in this project, and implementation of this project.

Some details are discussed in this chapter, such as:

- The synchronization of ACL packets at the receiving side,
- The control link for controlling speed and packet retransmission,
- The memory usage on decoder board,
- The double buffering technique at the receiving side.

4.1 System Design

Figure 4.1 shows the overview of the system design. There are two parts of the system, which are the transmitting part and the receiving part. When the server (transmitting side) starts up, it will be waiting for connection from the client (receiving side). The client will initiate the connection. Once the connection between the server and client is up, the client can then indicate that it is ready for receiving MP3 songs. Upon receiving the request from the client, the server program reads the MP3 data from the hard disk and sends them to the Bluetooth stack. The data is then transmitted to the receiving side via a pair of Bluetooth modules. At the receiving side, the received data is passed to the client application program from the Bluetooth stack. The client application program then channels the data to the MP3 decoder, which starts to decode the MP3 song.

Figure 4.1. Overview of System Design.

This project consists of two stages, which include PC-to-PC transmission and PC to headset transmission. The transmitting side is always a PC, and the receiving side can be another PC or a headset, which was a MP3 decoder board (CRD7309C board). [14][15]

4.1.1 Development platform and programming language

The operating system used in this project is based on the Windows XP. The Bluetooth communication system is developed on the *Caisra* Development Kit (from CSR [16]),

which include Bluelab (software development kit) and a pair of development boards with *Bluetooth* module chips. The decoder chip at the receiver side is CRD7309C (from Cirrus Logic).

4.1.1.1 Host Platform

The upper layer of Bluetooth stack is developed in windows XP. The programming language employed is Visual C++ 6.0 enterprise version (with service pack 4).

4.1.1.2 The Casira Development Kit

The firmware of the Bluecore2 chip is developed in C language. The compiler used is GNU C compiler. Figure 4.2 shows the Casira development kit. [16]

Figure 4.2 Casira Bluetooth Development kit.

The CSR Casira Bluetooth Development kit is used in the development of host-side software and external radio circuits outside the BlueCore ICs. It will also be used to

evaluate the BlueCore chips. The Evaluation system consists of two identical modules that allow the user to send both data and voice over a Bluetooth link. Each unit is driven from a PC for data connections and from a headset (supplied with the kit) for voice connection. Each unit consists of a motherboard and a Bluetooth radio transceiver daughter card module. The first radio module shipped with the system uses the CSR BlueCore2 chip and also includes the front-end filter, an low noise amplifier (LNA) , a power amplifier (PA), a TX/RX switch, a flash memory, a crystal and an external (connectorised and detachable) antenna. The radio module plugs into the motherboard. The motherboard provides a serial port (RS232), SPI port, USB port, PCM port, power supply socket and audio socket [14].

4.1.1.3 CRD7309 Reference Kit

The CRD7309C Reference Kit is used at the receiving side to decode the MP3 data. It helps to develop solid-state digital audio player products based on the EP7312 processor. The EP7312 acts as a system-on-chip (SOC) ultra low power audio decoder.[14][15]

Figure 4.3 Cirrus Logic CRD7309C decoding board.

Some other important components on the boards are the memory, the USB interface, A/D and D/A converter, and the RS232 interface [14].

There are two types of memory (external to the processor) present on the development board: 8 MBytes of NOR flash and 8 MBytes of NAND flash. The NOR FLASH memory is provided in one bank, made up of a single device Intel 28F640 J3A-120 Strata Flash used on the development board. Each block in memory may be erased at least 100,000 times. The FLASH device has a 120 ns random access time and a 25 ns sequential access time. Program the controller for 3 random and 0 sequential wait states for optimal performance. There are 8 MBytes of NAND FLASH present on the board. The FLASH device installed is a Samsung™ KM29U64000T. In addition to the NAND FLASH devices(s), a SmartMedia connector is present. SmartMedia™ cards can be used to add up to an additional 128 Mbytes of storage capability.

The USB interface provided includes a Phillips™ PDIUSB12D chip for 8 bit wide accesses and conforms to the USB 1.1 specification. USB connection sense (pin nEXTFIQ) indicates presence of a cable and provides an interrupt to the processor.

The A/D and D/A Converters used here are the CS43L32 and the CS53L32. The CS43L32 is a 24 bit 96 kHz D/A which is synchronized to the DAI clocks SCLK, LRCLK, as well as the externally generated 11.286 MHz MCLK signal. Decoded music frames are clocked in and decoded. Programming is done via the two-wire bus interface using SCL and SDA for clocking and data respectively. All initialization, volume, and

tone control are adjusted using this interface. Left and right channel audio output is filtered and sent directly to the audio headphone jack. The CS53L32 digitizes incoming audio. Data is synchronized with the same clocks as used on the CS43L32 and programmed with the same interface. Volume and gain control for left and right channels are programmable.

A RS-232 interface is provided and serves as the download interface for the CRD board. The MAXIM3221EAE driver interfaces directly to the EP7312 microprocessor to achieve a maximum baud rate of 115200 (processor limited) between the board and the host PC. The port is bi-directional so as to serve as a supplemental debug interface for text messages or characters. The debug interface is JPI, which serves as the JTAG interface or in-circuit emulator devices. Processor pins nTEST1 and nTEST0 are pulled low to allow the user to connect the device without additional configuration.

More detailed information can be found in Appendix C.

4.1.2 Flow Chart for Connection, Transmitting, and Receiving

4.1.2.1 Establishing a Connection

Before the application programs can use Bluetooth device to transfer data, a connection (ACL) must be established. Various steps were involved in order to create a connection between two Bluetooth devices. The sequence to establish a connection is summarized in

the Figure 4.4 below.

Figure 4.4. Messages for Setting Up A Connection Over HCI.

The application program of party A passes the connect request and the Bluetooth device address (BDADD), which consists of six 2-digit hex-decimal numbers, to the RFCOMM layer. The RFCOMM, in turn, invokes the L2CA_connectreq (message 1) to request for a connection. Before the L2CAP layer is able to have a connection to its peer, an ACL link must be setup first. Therefore, the L2CAP invokes the LP_connectreq (message 2) to request for an ACL connection. HCI receives the request and then formulates a command packet corresponding to the information passed to it and send it to the peer HCI through the Bluetooth module (message 3).

At the other side, party B, when the HCI receives the connection request, it notifies the L2CAP through the LP_connectind (message 4) indication. Then the L2CAP notifies the RFCOMM through L2CA_connectind (message 5). It is up to the RFCOMM to determine whether to accept the connection or not. If the RFCOMM wants to reject the request, a negative response will be sent so that the lower layer and party A can react accordingly. Otherwise, a positive response will be sent down to L2CAP and HCI (message 6 and 7). Then the HCI replies the peer HCI about the acceptance of the request (message 8). At this point the ACL link is up.

The details of message 3 and message 8 are shown in Figure 4.5. This figure shows the sequence of message passing between host and Bluetooth module and between two Bluetooth modules.

Figure 4.5. Create ACL Connection.

After the ACL link is setup, L2CAP of party A is notified. Now, it is able to create a L2CAP channel to the peer L2CAP in party B (message 10, 11 & 12). Also it is able to negotiate with the peer L2CAP about the Maximum Transmission Unit, flush time out and quality of service.

Finally, an L2CA_connectcnf is sent to the RFCOMM of party to confirm that the connection is setup and ready for use.

If next time, more L2CAP connections are needed between two parties, it is only necessary to send requests directly to the peer L2CAP, since the ACL link has been connected.

4.1.2.2 Transmitting

Several functions of different layers involved in the sending procedure. The functions and the sequence that they are invoked are shown in Figure 4.7 below.

The application program sends data by calling the function `bt_write_top()` in the top glue layer. The glue layer invokes the `rfcomm_send_data()` function in the RFCOMM. The `rfcomm_send_data()` function segments the data into smaller packets according to the RFCOMM MTU and then pass the segmented data to the `send_uih()` function one by one. In the `send_uih()` function, the data is copied into a transmit buffer, and header is added to form a UIH packet. Because the current RFCOMM MTU is 127 bytes, which does not

exceed the maximum payload of a short UIH packet, the UIH packets constructed are all short UIH packets. The newly created UIH packet is then passed down to the L2CAP. The `l2cap_end_data()` function adds the L2CAP header in front of the UIH packet to form a L2CAP packet and passes it to the HCI. The `hci_send_data()` function takes the L2CAP packet and adds a HCI header to it. After so many manipulations, the packet is ready to be transmitted and is queued in a queue of transmit buffers. The `send_acl_data_task()` checks the queue, if it is not empty, it will call the `send_acl_data()` to send the HCI packets in the queue one by one. The `send_acl_data()` function actually invokes the `bt_write_lower_driver` to write the data to a descriptor, and releases the transmit buffer.

Figure 4.6. Functions For Transmitting Data.

4.1.2.3 Receiving

The HCI receive procedure of the Axis stack has a state machine, which is driven by the data received. Figure 4.6 shows the states and signals that drive the machine to change states.

Figure 4.7: Receiving State Machine.

The receiving procedure is invoked when it is signal that data is available. Initially, the state machine is in the Wait_for_packet_type state. If it cannot determine from the data received what type of packet it is receiving and the time out signal is received, the state machine will change to the Wait_for_timeout state and discard all the received data, then return to the initial state to wait for another block of data.

If it can determine that an event packet is incoming, the machine will go into the Wait_for_event_type state, and then the Wait_for_event_length state. If there is any error in this state, the machine will reset to the initial state. Otherwise, it will go into the Wait_for_event_param state. After the whole event packet is received completely, the HCI layer will process this event according to its event type. At the completion of event processing, the state machine will reset to its initial state.

If an ACL or SCO data packet is incoming, the machine will first in the Wait_for_ACL_header or Wait_for_SCO_header state, and then the Wait_for_ACL_data or Wait_for_SCO_data state. When the data packet is received completely, the HCI layer will pass it to upper layer of stack. And the machine resets to the initial state.

4.1.3 Design considerations

4.1.3.1 Real-time decoding

The decoder has to decode the sound at the same speed or faster than the decoded sound

is being played. When the data is directly written to the RFCOMM layer, the overheads are RFCOMM header and checksum (4 bytes), L2CAP header (4 bytes), HCI header (4 bytes), Baseband header and CRC (20 bytes). Totally, the overheads are equal to 32 bytes. As a result, for 127-byte long RFCOMM data, the overhead in percentage is $32/127 = 25.2\%$.

Table 4.1 summarizes the required transmission rates and the buffer sizes required to buffer 10 seconds of data for different MP3 bit rates.

Table 4.1: Transmission Rates And Buffer Size Required for Different Bit Rates.

Bit Rate	Transmission Rate	Buffer Size
	(Bit Rate $\times$ 1.252)	(Bit Rate $\times$ 10 / 8)
128kbps	161kbps	160 Kbytes
96kbps	120.2kbps	120 Kbytes
64kbps	80.2kbps	80 Kbytes
40kbps	51kbps	50 Kbytes

4.1.3.2 Memory requirements

It is always good to keep the memory requirements low since memory is expensive and consumes power, especially when the target system is a small portable device. It is important to have this in mind when choosing algorithms. The table of the memory usage is listed in section 4.2.2.2

4.1.3.3 Bluetooth communication

The transfer of MP3 data to the decoder should be carried out over Bluetooth radio communication. During this project, the Advanced Audio Distribution Profile (A2DP) in the Bluetooth protocol was under development and only version 0.95 of the profile description was available. The payload format was not determined in this version of the profile. Because of this the original format of transferring DH2 packet format is taken.

4.1.3.3 Double buffering technique

The since the transmission speed is different with the decoding speed. In the first stage, the receiving side is the PC, and the resource is quite enough for the decoding. In the second, the DSP board is used at the receiver side; the RAM size is only 64 kbytes. The buffer technique is necessary to control the transmission speed. The double buffering is employed in this project.

The basic constructs of the Double Buffer algorithm are shown in Figure 4.8.

Figure 4.8 Constructs in the Double Buffer algorithm.

The Pseudo Code of Double Buffer algorithm is shown below:

```

int NReader;           #Number of readers
int NRows = NReader+1; #Number of rows with the latest message
int Latest;            #Index to the row with the latest message
Message Buff[NRows][2]; #Reader count
Int ReaderCnt[NRows];  #Reader count for each row
Boolean Cl[NRows];     #Column with more up-to-date message

Reader () {
1:   ridx = Latest;
2:   int ReaderCnt[ridx];
3:   cl=Cl[ridx];
4:   read Buff[ridx][cl];
5:   dec ReaderCnt[ridx];
}

Writer() {
6:   for (i = Latest; ; i++)
7:       if (ReaderCnt[i mod NRows] == 0) break;
8:   cl = not Cl[i];
9:   write Buff[i][cl];
10:  Cl[i] = cl;
11:  Latest = i;
}

```

A two-dimensional shared message buffer, $\text{Buff}[][]$, has $(P + 1)$ rows, where P is the number of reader tasks. Each row has two buffers. Associated with each row i is a usage count, $\text{ReaderCnt}[i]$, representing the number of readers currently using either buffer in the row, and a flag, $\text{Cl}[i]$, indicating which of the two buffers is more current. A variable, latest , points to the row containing the most recently written data. A reader task first reads Latest , and indicates it is using the row by incrementing the usage count. It then reads the buffer indicated by the row's Cl flag, and decrements the row's usage count when it finishes reading. The writer is fairly straightforward. It first scans $\text{ReaderCnt}[]$, and selects a row that is not being used by the readers. It then writes to the buffer that was least recently written in the selected row (i.e., opposite to the one indicated by the row's Cl flag). We will see why this is necessary shortly. Finally, it updates the row's Cl flag to point to the newly-written buffer, and sets Latest to the row that contains this buffer. In case each reader is concurrently reading from a unique row, this algorithm requires $(P + 1)$ rows for the writer to work correctly, where P is the number of readers. As each row has

2 buffers, the space required for the message buffer array is $2(P + 1)$.

4.2 SYSTEM IMPLEMENTATION

The project has two stages, which are PC-to-PC Transmission, and PC to Headset Transmission.

4.2.1 Development in the First Stage

4.2.1.1 Upper layer Development

The application is built on top of the open source: the “Bluesuite HCISource” version 1.18 from CSR [4-3] [4-4]. It is developed under visually C++ 6.0 (enterprise version) with service pack version 4.0. According to the License Agreement, the modification on the source can be applied freely, as long as the development work is on CSR product. This source code includes the application layer, the RFcomm layer, and the L2cap layer. The firmware required on the Bluecore2 chip contains the HCI and lower layers.

The server application program is responsible to

- Listen to the request from client;
- Read the MP3 file from hard disk into a buffer when request from client is received;
- Invoke appropriate functions to process the data in the buffer.

The client application program is responsible to

- Request MP3 files from server;
- Read MP3 data from the receive buffer;
- Send the received MP3 data to MP3 decoder via piping.

4.2.1.2 MP3 Decoder on the receiver side

For the convenience, the ActiveX controller is used for the PC-to-PC transmission. The MP3 decoder used is the Windows media player, which can be directly imported to the Visual C++ codes.

4.2.1.3 Graphical User Interface

Figure 4.9. HostController application interface.

The group box on the upper-left corner is to determine the role of the Bluetooth module. The group box right below determines the transfer type. The upper middle group box is

used to display the address of the Bluetooth device, and the address of the connected Bluetooth device. The lower group box shows the transfer process of the file transmission. The three buttons on the upper right part of the panel is to toggle the “Inquiry”, “Connect”, and “Disconnect” functions. The list box on the lower part of the panel is to show the host controller interface messages.

Here the data transfer type means the conventional ACL transmission. The audio transmission is using ACL link instead of the SCO link. This is where the modification will be applied.

The communication between the HostController application and the Bluetooth module has three ways, through the BCSP, H4, and USB. The data transfer rate is ranging from 4800kbit/s to 1382400kbit/s. The typical data rate during the testing is 115200kbit/s. BCSP is the BlueCore serial protocol, which can be regarded as the UART driver from CSR on the Host side. H4 refers to the UART here. For the ease of the development, the H4 and USB connection will not be enabled, only the BCSP connection is applied at current stage.

This simple GUI, which is a control panel, consists of one text entry, three buttons, as well as a text area. The text entry is for the user to key into the Bluetooth address of the Bluetooth module that he or she wants to connect to. The ‘Inquiry’ button is used to invoke the inquiry function to inquiry to enquire Bluetooth modules that are within the operating range. The ‘Connect’ button is used to invoke the connect functions to connect

to the Bluetooth module whose Bluetooth address is the same as specified in the text entry. The 'Receive' button notifies the server that the client is ready for receiving data. All system information is displayed in the text area at the bottom of the control panel.

4.2.2 Development in the Second Stage

The development work in this stage is mainly on the decoder board for decoding the incoming MP3 data. Another part of the programming work in this stage is to establish the Bluetooth communication between the transmitting side and the receiving side.

4.2.2.1 Modification of the Decoding Program

Since the Cirrus logic has provided some API, it is not necessary to write the decoding algorithm from scratch, such as Huffman decoding, Inverse Discrete Cosine Transform (IDCT), etc. The original sample source code only provides one function, which just read data from an mp3 file and decodes it. To realize our real time decoding purpose, several additional functions have been added. Those functions will check the status of the current playing mp3 file, and point to next decoding frame. Some of the variables and Functions of the decoding program has been listed:

A structure which defines the persistent state of the MP3 decoder

```
typedef struct { } tMP3;
```

MP3InitBitstream initializes the stream to the given byte offset in the file.

```
static void MP3InitBitstream(tMP3 *pMP3, unsigned long ulFilePos)
```

MP3FindNextFrame finds the next frame in the input MP3 bitstream.

*static unsigned long MP3FindNextFrame(tMP3 *pMP3)*

MP3DecodeVBRHeader determines if the current frame is a Xing VBR header frame and decodes it if it is.

*static unsigned long MP3DecodeVBRHeader(tMP3 *pMP3)*

MP3DecodeID3Tag sees if the MP3 stream has an ID3 tag. If found, the song title and artist is extracted from the tag.

*static void MP3DecodeID3Tag(tMP3 *pMP3)*

MP3AddID3Frame adds a frame to the ID3-V2 tag.

*static unsigned char * MP3AddID3Frame(unsigned char *pucBuffer, char *pcTag, unsigned short *pusValue)*

Determines if the given file is a MP3 file, and returns the offset of the first frame if so.

*static unsigned long MP3IsMP3File(tFile *pFile, unsigned char *pucScratch, unsigned long ulOffset)*

MP3SetupDecoder initializes the MP3 decoder.

*unsigned long MP3SetupDecoder(tMP3 *pMP3, unsigned long ulStart, tFile *pFile)*

The codec plug-in entry point for the MP3 decoder.

unsigned long MP3Ioctl(unsigned long ulIoctl, unsigned long ulParam1, unsigned long ulParam2, unsigned long ulParam3, unsigned long ulParam4)

More detailed source can be found in Appendix D.

4.2.3 Modifications Made to the Stack

Because the transmission rate and the decoding rate are not exactly matched (the decoding rate is slightly slower than the transmission rate), the host-to-host controller flow control at the receiver side must be turned to limit the transmission rate. At the transmitter side, when the transmission rate is limited, some packets cannot be sent out at the first attempt, these packets may have been copied to the transmit buffer, or, when the transmit buffer is full, have not been copied to the transmit buffer. For those packets that are already in the transmit buffer, it is not necessary to re-transmit them, but for those that are not in the transmit buffer, retransmission is required.

Unfortunately, the stack makes no distinction between the two situations, and always retransmits those packets that failed to transmit in the previous attempts. As a result, data packets that are already in the transmit buffer are written twice to the buffer. Hence, the transmit buffer contains corrupted data, which degrade the sound quality when the stream is played back.

A modification was made to the functions involved in the transmitting process in the stack, so that the stack knows whether a packet has been written to the transmit buffer, and only packets that are not in the transmit buffer need to be retransmitted.

After the modification was made, the sound quality improves significantly.

4.4 Conclusions

In the first stage, since the receiving side is a PC, the programming is simpler. The Bluetooth stack on both side are same. PC has large enough memory size, so the memory control is not necessary. However, in the second stage, all Bluetooth stack layers at the receiving side are running on the Bluecore2 chip, which is different with the transmitting side, it take long time for the debugging of stack mismatching. The memory control is also important due to the limited memory size on the decoder board.

Chapter 5. EXPERIMENTS AND DISCUSSIONS

A number of experiments are conducted in order to test the transmission rate and the reliability of the connection. Some of the problems faced the Bluestack are also being discovered and solutions are recommended to solve these problems.

There are mainly three types of experiments carried out in this project:

- The testing of Bluetooth link,
- Evaluate the Memory usage on the CRD7309 Board,
- Evaluate the sound quality on the receiving side.

5.1 Experiments on Bluetooth ACL Link

Three experiments are measured to the data transmission rate of the Bluetooth ACL link.

These measurements include:

- Data rate vs. Bluetooth ACL packet size,
- Data rate vs. distance between transmitter and receiver (without interference),
- Data rate vs. distance between transmitter and receiver (with interference).

5.1.1 Data rate vs. Bluetooth ACL Packet Size

At the transmitting side, a Pentium IV PC and the Casira development platform are used as the host computer for different experiments. At the receiving side, a Pentium IV PC is used as the host computer. The system setup is shown in Figure 5.1.

Figure 5.1: Experiment Setup.

There is no interference in this experiment, and the distance between the transmitter and receiver is fixed (< 1 meter).

An application program is written on top of the Bluetooth Stack in order to test the feasibility of audio streaming using Bluetooth.

At the transmitting side, the specified file is read from the disk in ASCII mode or binary mode into a buffer. Then the `bt_write_top()` function in the Bluestack is invoked to send the data in the buffer using the specified packet size. The packet size here is used to indicate how many bytes can be passed to the RFCOMM layer at each time. After the data is passed to the RFCOMM, the Bluestack invokes corresponding functions to handle the data transmission. The number of transmitted bytes is returned as a value to the application.

At the receiving side, the received data is saved into a file with a name specified by the user so that they can be compared with the original file later on.

Since the MTU of RFCOMM is 127 bytes [14], which means that each 127-byte data is packed into an HCI packet and transmitted, it is more efficient to pass multiples of 127-byte of data from the application to the RFCOMM for transmission. Therefore, in the experiments, the packet sizes used are multiples of 127-byte. Table 5.1: summarizes the transmission rates obtained using different packet sizes.

Table 5.1. Data rate on Different Packet Size.

Packet Size	Tx Rate (sample data, host to HC flow control off)	Tx Rate (MP3 file, host to HC flow control off)	Tx Rate (sample data, host to HC flow control on)	Tx Rate (MP3 file, host to HC flow control on)
127 bytes	630 kbps	600.1 kbps	630 kbps	600.1 kbps
254 bytes	713.5 kbps	709 kbps	713.5 kbps	709 kbps
381 bytes	722.8 kbps (with errors)	N/a	704 kbps	N/a

The sample data used are generated by the stack, while the MP3 file is read from the disk. Therefore, the transmission rate obtained using sample data is higher then that obtained using a MP3 file.

The data obtained shows that when the packet size getting bigger, the transmission rate is higher. This is because with larger packet size, for the same amount of data, the application invokes the functions in the stack fewer times. Consequently, the overhead is less and the transmission rate is higher.

The data obtained shows that the receiving side is not fast enough to match with the transmitting side when the packet size is 381-byte. Hence, the trash data errors result. In order to eliminate the trash data errors, the host-to-host controller flow control is turned on so that the transmission rate is limited by the receiving side.

When the packet size is 381-byte and the host-to-host controller flow control is on, the transmission rate significantly reduces to 704 kbps, which is even lower than the transmission rate using the 254-byte packet. The reason for this is that the transmission rate is limited by the receiving side, and some data are left in the transmit buffer. The higher layer at the transmitting side notices this situation and it delays for some time to wait for the data to be sent before the new data are passed to the transmit buffer. As a result, the transmission rate drops further.

When the transmission rate is getting higher (or, the packet size is larger), the system becomes less stable. When the 127-byte packet is in use, the link is quite reliable. When the 254-byte packet is in use, more failures that is the transfer of data cannot finish during the transmission are experienced. When the 381-byte packet is in use, the system cannot finish transmitting a MP3 file.

According to the experimental data obtained, packet size of 127-byte is chosen to use in the implementation of the MP3 sever in order to achieve the most stable performance.

5.1.2 Data Rate vs. Distance between Transmitter and Receiver (Without Interference)

At the transmitting side, a Pentium II laptop and the Casira development platform are used as host computer. At the receiving side, a Pentium II laptop is used as the host computer. The system setup is shown in Figure 5.1.

Figure 5.2. Experiment Setup

There is no interference in this experiment, and the distance between the transmitter and receiver varies from 0 to 15 meter. The normal operation range of the class 2 Bluetooth chip is 10 meters, which is within the experimental distance. The packet size used in this experiment is 127 bytes.

To keep the transmission period long enough, the size of the testing MP3 file is 27575 kbits. The data is collected every 50cm, and the curve of the data rate is shown in Figure 5.3.

Figure 5.3. Data rate against distance between transmitter and receiver (without interference).

The class 2 Bluetooth chip has 10 meter operation range. The lowest data rate within the range is 450 kbps.

5.1.3 Data Rate vs. Distance between the Transmitter and Receiver (With Interference)

Since Bluetooth devices hop over 79 MHz of the ISM band and IEEE 802.11b devices require approximately 16MHz of bandwidth to operate [15], it is not possible to have both Wi-Fi and Bluetooth products in the same area without the chance of interference, since both technologies uses the ISM frequency band. Due to the potential for interference, interference test has been carried out with actual Bluetooth and Wi-Fi products to determine their level of interference.

This experiment takes the same experiment setup as previous experiment (in section

5.1.2). The only difference is that a laptop transmitting Wi-Fi signals at 1 meter distance away.

The distance between the transmitter and receiver varies from 0 meter to 15 meter, and the packet size used in this project is 127bytes. The same MP3 file is used in this experiment.

Figure 5.4 Data Rate against distance between transmitter and receiver (with interference).

5.1.4 Comparison between the Results in Section 5.1.2 and 5.1.3

The IEEE 802.11b standard is a specification for Wireless Local Area Networks (WLAN). The Wireless Ethernet Compatibility Alliance (WECA) acts as a certification organization for products that interoperate with one another via the IEEE 802.11b standard [15]. Products that achieve certification are deemed Wi-Fi compliant. The Wi-Fi

systems transmit data in the unlicensed 2.4GHz ISM band. A relatively large excess bandwidth (or bandwidth expansion factor) is used for the pulse-shaping filter, which results in a typical spectral mask as shown in Figure 5.5 (the central frequency is 2.4 GHz) [15].

Figure 5.5. Typical Wi-Fi (IEEE 802.11b) baseband Signal Power. (from [15])

Wi-Fi products transmit at data rates up to 11Mbps. Typically, Wi-Fi devices operate at distances up to 100 meters, however, range varies as a function of transmit power and environment, e.g. indoors versus outdoors.

The Bluetooth standard is a specification for Wireless Personal Area Networks (WPAN). Although products based on the Bluetooth standard are often capable of operating at greater distances, the targeted operational area is the area around an individual, e.g. within 10 meters of the user. The spectral mask of a Bluetooth signal is 1 MHz wide at

the 20dB points, as is shown in Figure 5.6(the central frequency is 2.4 GHz).

Figure 5.6. Typical Bluetooth Signal Power Spectrum.(from [16])

The result from section 5.1.2 and 5.1.3 has been put together in figure 5.7.

Figure 5.7. Comparison between the data rate.

Due to the interference of the Wi-Fi device, the data rate of the Bluetooth link drops 200-300 kbits. In this project, the real time decoding is required, so the data rate should not be smaller than the decoding rate. For typical MP3 files, the decoding rate is 128kbits/sec, and 192kbits/sec. The operation range of the class 2 Bluetooth chip is 10 meters, so, as we can see from Figure 5.7, the data rate in the with interference case is around 300 kbits. Therefore, the data rate is enough for real time MP3 decoding.

5.1.5 Comparison with result of paper [1]

In our project, the DH5 packet is used for the experiment. The result obtained in our experiment is not as ideal as the result shown in figure 5.8. When the data transmission is within 2 meter range, the curves are quite similar. When the distance is beyond 3 meters, our transmission rate decreases very fast. The reason should be the environment of carrying out the experiment. Since the experiment is carried out in campus area, which is covered by WLAN, although there is not WLAN device working nearby, but there is some interference from the WLAN server. Therefore, the data rate drops, since there are more retransmission due to the WLAN interference.

Figure 5.8. Variation of the measured bitrate with the ACL packet type, the application packet length and the distance (no interference exists) (from [1]).

For small application packet lengths (20k, 30k), the delays introduced due to the blocking procedures of the L2CAP layer are longer than those caused by the retransmissions of the transmitted packets, thus the effective bitrate remains constant with the distance. On the other hand, when large application packets are considered (40k), and in rooms with interference, the retransmission delays are dominant and reduce the bitrate with the distance. Moreover, in rooms with interference, the measured bitrate is reduced with the distance between the Bluetooth devices and no connection can be performed for distances higher than 12m. On the other hand, when no interference is appeared, the transmission is successfully completed at distances greater than 20m. Figure 5.9 illustrates this issue.

Figure 5.9. Variation of the measured bitrate with the distance for different application packet lengths (with the existence of interference and no interference) (from [1]).

5.2 Experiments on Memory Usage on CRD7309 Board

During the development of the program on the CRD 7309 board, small application is written to check the memory usage of the board. Table 5.2 shows the memory usage of the decoder board.

Table 5.2. ROM & RAM Usage on CRD 7309 Board.

Module	ROM (byte)	RAM (byte)
Main Player	20,480	2,036
Filesystem	30,720	520
Internal NAND	3,072	1,080
SmartMedia	3,072	1,080
User Interface	16,384	172
Input Processing	1,024	76
Output Processing	18,432	1,644
USB	4,096	28
Software Tone Control	1,536	64
Software Volume Control	1,024	36
Spectrum Analyzer	2,048	224
MP3 decode	32,768	26,920
MP3 encode	78,848	37,084

The table is broken up into three groups; the first set of modules (main player, file system, internal NAND, User interface, Input processing, Output processing, USB) must always exist and always consume the specified amount of ROM and RAM. The remaining modules are individually selectable, are optional, and consume no memory (ROM or RAM) if not included in the player. The second sets of modules (Software Tone Control, Software volume Control, Spectrum Analyzer) always consume the specified amount of

RAM if included in the player. The third group of modules is for the codecs; they consume the specified RAM only when decoding a file.

5.3 Conclusions

The Bluetooth ACL link experiments carried out during the development of the project have tested the transmission rate and the reliability of the connection. Memory usage experiment helps to decide the allocation of memory. Sound quality experiment helps to decide the most suitable buffer size.

The experiments also reveal the bugs in the Bluestack (introduced in Chapter 4), and improvements can be made to the program (see in Chapter 6).

CHAPTER 6. CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

In this report, the wireless transmission of digital audio data using the Bluetooth protocol for real-time playback applications was examined, starting from the initial conclusion that reproduction can be achieved only after compressing the original audio material and transmitting it through Bluetooth ACL links. Two application scenarios were implemented and tested, realizing point-to-point transmission of both audio and control information.

From implementation, the data collected during the experiment and the assessment of the system performance, the followings can be concluded:

- The connection between PC to Bluetooth module and Bluetooth module to decoder board should use USB cable, since the serial connection cannot provide high enough data rate for the transmission.
- The transmission rate between two Bluetooth devices is sufficient for transmitting MP3 data. If hardware is stable when the bit rate is high, better quality MP3 songs can be transmitted.

- The client machine on one hand need to run a Bluetooth stack to receive MP3 data, on the other hand, it need to run a decoder to decode the received MP3 data. Hence, the client machine requires significant amount of processing power.
- The single chip solution of Bluecore2 chip makes it possible to use decoder board at the receiving side instead of PC. For the Bluetooth stack protocol, all layers, which are from the application layer to the radio layer, all run on the Bluecore2 module. The data send from the Bluetooth module to the decoder board is in byte format.

6.2 Recommendations for Future Works

Since the Bluetooth technology is still far from its maturity, there is certainly much room to improve this project in the future. Some key ideas are summarized as follows:

- The application program developed in this project is highly coupled with the Bluetooth stack, which makes it difficult to maintain. Also, other applications that are developed for normal network cannot be used directly on the Bluetooth network. A program that is able to simulate a LAN interface on top of the Bluetooth stack can be developed to solve this problem in this project.
- Service discovery is important for Ad-Hoc network like Bluetooth. However, due to time constraint, the service discovery feature of Bluetooth has not been

examined in-depth in this project. Also, the Service Discovery Protocol (SDP) has not been fully implemented in the Bluetooth stack that is currently being used. Future work can be done to complete the implementation of SDP in the stack. Once the SDP is available, some attractive functions can then be easily implemented. Some of these functions are: Before a client starts to request songs from the server, it is able to obtain a name list of songs available in the server, and then selectively request songs from the server; the server may also provide other services such as lyrics download, the client will be notified through the SDP.

- Due to time and hardware constraints, the piconet and scatternet operations have been ignored. However, a practical MP3 server must make use of the piconet operation. Hence, more effort can be put in to investigate the piconet and scatternet operations.
- As mentioned in chapter 4, the client machine requires significant amount of processing power to run the Bluetooth stack and the MP3 decoder. However, Bluetooth technology will be mainly used by mobile devices, and most of these devices only have limited processing power. Hardware implementation of Bluetooth stack may be necessary for mobile devices.
- Multiple client transmission is still to be implemented. The channel allocation, how to allocate channels to the multiple transmission at the same time, need to be take into consider action. The degradation of the transmission quality due to the

Interference between channels is another problem, which needs further investigation.

REFERENCES

1. *Specification of the Bluetooth System Core Version 1.1*. Available at URL:
<http://www.bluetooth.com>
2. ISO/IEC 11172-3, Information technology . Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s . Part 3: Audio, 1993.
3. "A Study of Wireless Compressed Digital-audio Transmission", A. Floros, M. Koutroubas, & et.al, *Wireless Audio Transmission*, 2003.
4. A. Das, & et.al., "*Enhancing performance of Asynchronous Data Traffic over the Bluetooth Wireless Ad-hoc Network*", 20th Annual Joint Conference of IEEE Computer and Communication Societies Proceedings, Inforcomm 2001.
5. R. Kapoor, &et.al., "Bluetooth: Carrying Voice over ACL links", *Mobile and Wireless Communications Network*, 2002. 4th International Workshop on 2002.
6. Brent A. M., and Chatschik B, *Bluetooth Revealed*, Prentice Hall PTR, 2001.
7. M. Albrecht, and et. al, "IP services over Bluetooth: Leading the way to new mobility", in *Proceedings of the 5th International Conference on Local Computer networks*, pp.2-11, 1999
8. D. Groton and J.R. Schmidt, "Bluetooth-based Mobile Ad Hoc networks: Opportunities and challenges for telecommunications operator," *Vehicular Technology Conference*, Spring 2001, vol. 2, pp. 1134-1138.
9. <http://www.palowireless.com/infotooth/tutorial/profiles.asp>

10. K. Brandenburg and H. Popp, "An Introduction to MPEG Layer-3", Fraunhofer Institute, EBU Technical Review, June 2000.
11. "A Hardware Accelerated MP3 Decoder with Bluetooth Streaming Capabilities", S. Gadd, T.Lenart. November 2001.
12. Specification of the Bluetooth System: Advanced Audio Distribution Profile, version 0.95b
13. Specification of the Bluetooth System: Audio/Video Distribution Transport Protocol, version 1.0.
14. Hardware Guide for Cirrus Logic development board CRD7309C, Cirrus Logic. Available at URL: <http://www.cirrus.com/support>
15. Quick Start User's Guide for Cirrus Logic development board CRD7309C, Cirrus Logic. Available at URL: <http://www.cirrus.com/support>
16. Specification of Casira Development Kit, CSR Ltd. Available at URL: <http://www.csr.com/support>
17. John Hyde. USB Design by Example: A Practical Guide to Building I/O Devices. John Wiley & Sons, Inc. 1999
18. <http://www.allegromicro.com/techpub21usb/episode1.htm>
19. Internet Audio Player Reference Software, Release 17, Cirrus Logic, Inc. Available at URL: <http://www.cirrus.com/support>

Appendix A: An Overview of the Bluetooth Software Requirements for Users of the BlueCoreTM ICs

A.1 INTRODUCTION

CSR's BlueCoreTM single chip systems are supplied with firmware that runs on the chip's 16-bit RISC microcontroller. The firmware for BlueCoreTM 01 is held in an off-chip ROM; for BlueCoreTM 02 the ROM will be on-chip. The early versions of the firmware implement the lower layers of the Bluetooth protocol stack. Later versions will include some upper layers of the Bluetooth protocol stack.

Initially, the Bluetooth developer will need to obtain and/or implement software in addition to the software supplied directly by CSR in order to complete the integration and certification of the Bluetooth system inside their systems.

This application note provides an overview of:

- . • The Bluetooth firmware initially supplied by CSR with the BlueCoreTM Evaluation System, and the requirements for software not supplied by CSR.
- . • Future upgrades to the CSR Firmware and the associated requirements for on-host software.
- . Future upgrades to the CSR Firmware for stand-alone (no-host) Bluetooth applications.

The Software Development tools supplied by CSR and its development partners today and how these will be extended in the future.

A.2. INITIAL CSR SOFTWARE/FIRMWARE

With the first shipment of CSR’s BlueCore™ Evaluation Systems in May 2000 (alpha software release), the following software model is supported:

Figure A-1. Typical software partition in non-PC based systems with both a host and host controller.

A.2.1 Bluetooth Protocol Stack Lower Layers

The Link Controller, Link Manager and Host Controller Interface–bottom (HCI-bottom) are supplied as executable code with BlueCore™ ICs, free of charge. For applications that will use BlueCore™ ICs, CSR can also discuss the terms of a licence to the source code for these software layers. The alpha release of CSR firmware implements all of the Bluetooth v1.0b specification below HCI except for those areas specified in the BlueCore firmware alpha release note.

This software is pre-installed in the BlueCoreTM 01 IC inside the BlueCoreTM Evaluation System. However firmware upgrades will be provided by CSR that will extend the functionality of the CSR firmware, as described in this document. The utility to upload firmware revisions onto the BlueCoreTM ICs is freely available from CSR.

A.2.2 BlueCoreTM Serial Link Protocol (BCSP)

CSR has developed a serial transport protocol for the communications link between the Bluetooth Host and Host Controller. The BlueCoreTM IC hardware is capable of supporting the Bluetooth SIG defined HCI transport protocols for UART based physical transport. These are 'RS232' transport protocol and the 'UART' transport protocol defined in Bluetooth Foundation Core Specification v1.0b parts H:3 and H:4 respectively. However, the BlueCoreTM ICs have hardware extensions for the efficient support of BCSP which give the following advantages over the other two protocols:

1. BCSP uses SLIP to form a packet orientated wire protocol. The BlueCoreTM hardware exploits this to generate hardware interrupts at the packet rate rather than at the byte rate, so easing the load on the chip's processor.
2. BCSP uses CRC integrity checking rather than the more heavyweight COBS used in the standard RS232 HCI transport. This is handled by the chip's hardware, so reducing the processor load. The CRC checking can be disabled to reduce the host processor's load.
3. BCSP uses windowing flow control mechanisms to ensure packet delivery and to

provide flow control. This helps the overall system tolerate short-term processor blockages. It also allows data to flow in one direction while the other is temporarily blocked.

4. BCSP provides a set of reliable datagram channels between the host and host controller. This allows extra flows of data between the processors beyond those specified by HCI. This permits a common transport protocol be used with different Bluetooth software partitions between the host and host controller.

The BCSP protocol is defined in detail in the CSR application notes and release documents. CSR has also implemented a DLL running under WindowsTM 98 and WindowsTM NT that is a real example of the BCSP protocol on a PC host. The DLL binary, 'C' source and build environment for BCSP.DLL are shipped with the BlueCoreTM Evaluation System.

A.2.3 Bluetooth Protocol Stack Upper Layers

Support for the L2CAP, RFCOMM and SDP protocols are needed for most of the early Bluetooth profiles. These profiles include the Serial Port Profile, the Headset Profile, the Dial-up Networking Profile, the LAN Access Profile, the Fax Profile, the File Transfer Profile, the Generic Object Exchange Profile, the Synchronisation Profile and the Object Push Profile. If software developers using BlueCoreTM plan to run these extra software layers on an external host, then they must either develop these layers from the Bluetooth specification themselves or buy this software from third parties. It is likely that several

competent suppliers of this software will emerge during the early roll-out of Bluetooth. Later, it is likely that these Bluetooth layers will be offered as standard within a range of industry standard operating systems.

CSR continues to work closely with its software partner Cambridge Consultants Ltd. (CCL) who is providing 'C' source licences for these upper layers running on a host, as well as host side software development tools. The CCL software stack is called BlueStack and is at alpha release at the same time as the BlueCoreTM firmware. Implementations of these upper layers and different Bluetooth Profiles will be supplied by other partners of CSR, including Socket Communications, Inc. (for WindowsTM CE applications)

The CCL upper layers run under the same simple (co-operative, non pre-emptive, multitasking) scheduler used in the BlueCoreTM firmware. It should be run as a single thread under the main host operating system

A.2.4 Interface Between Protocol Stack Upper Layers and the Application & O/S on the Host

This software will normally be developed by CSR's customers, however CSR's software development partners can also produce this software under contract.

A.3 SOFTWARE DEVELOPEMENT TOOLS

A.3.1 BCSP DLLs and BlueChat Example Application Provided by CSR

The BlueCoreTM Evaluation System is shipped with the binary and source files of the BCSP DLL as well as documentation describing the BCSP structure and API. This will allow software interfaces to be written for the BlueCoreTM UART/BCSP interface.

The BlueChat example application allows text and file transfers as well as the establishment and disconnection of audio links over the Bluetooth radio using two WindowsTM 98 or WindowsTM NT PCs.

A.3.2 CCL BlueStack Software Development Kit (SDK)

Software available separately from CCL includes host-side software development tools running on a WindowsTM PC.

The evaluation copy on the CDROM that is shipped with the first BlueCoreTM Evaluation System provides accessibility and visibility of the interfaces between the different layers within the host-side Bluetooth stack via a command and watch window interface. Macros can be constructed to string together sequences of commands at the L2CAP, RFCOMM or SDP level.

CCL will also ship demonstration applications of file and text transfers and audio connections that simultaneously give access to the command and watch windows.

The full version of the BlueStack SDK can be purchased directly from CCL as an upgrade. This opens the full API to the CCL host-side software stack and delivers this software stack in the form of linkable object files into a larger application running under

Windows NT. The source code of the demonstration application will also be shipped to provide an example of how to write an application that links in the BlueStack object code.

Appendix B. Sample source code for BlueCore2 module on upper layer

Only the header file is given here to show the functions. The real implementation of those functions is not in BlueCore2.cpp file.

```
// BlueChat2Dlg.h : header file
//

#ifndef __AFX_BLUECHAT2DLG_H__FBFB9FAD_233B_4B09_9E0C_DA841BF93AF9__INCLUDED_
#define __AFX_BLUECHAT2DLG_H__FBFB9FAD_233B_4B09_9E0C_DA841BF93AF9__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "BlueChat2Engine.h"
#include "BlueChat2HciDefs.h"
#include "console.h"

#define MAX_NO_SLAVE_CONNECTIONS
    3 // 3 for now!

class CBlueChat2DlgAutoProxy;

////////////////////////////////////
// CBlueChat2Dlg dialog

class CBlueChat2Dlg : public CDialog
{
    DECLARE_DYNAMIC(CBlueChat2Dlg);
    friend class CBlueChat2DlgAutoProxy;

// Construction
public:

    // Comapre two addresses
```

```

        static      bool      CompareDeviceAddresses(BTDEVICE_ADDRESS      *
BTDeviceAddress1,

BTDEVICE_ADDRESS * BTDeviceAddress2

);
// is the selected device connected at all? If so return the index of our connection
in our
// arrays, otherwise return -1
// If we are a slave, this simply returns 0 if we are connected and -1 if not
// (as 0 will be our "index" in the arrays, as we only have one connection)
int IsSelectedDeviceConnected();
// is there a slave to slave audio connection, anywhere on the piconet?
bool IsSlaveToSlaveAudio();

void InitializeSound (void);
void InitializeCapture (void);
void PlaySCO (uint8 *ptr, unsigned int length, int sco_connection_index);
// returns length of buffer played
int PlaytoSCO();
int PlayCapturetoSCO();
// play sco data passed to slave passed slave index is connected to
int PlayScoToConnectionSlave(uint8 * data, int length, int connection_index);
// play sco data passed to passed connection index
int PlayScoToSlave(uint8 * data, int length, int connection_index);
// broadcast the sco to all our slaves (save where it came from)
void BroadcastSco(int from_connection_index,
uint8 *data,
unsigned int length);

// get the broadcast state of slave at given index
bool get_broadcast(int i) { return broadcast[i]; }

// Checks the capture buffer
void PreCheck();
void CheckCaptureBuffer(void);

static void scoCallback (uint8 *data, unsigned int length, void *aThis);

void SetFactory(BlueCoreControllerFactory * Factory);
void setScoOverHCI(bool hci_sco) {mEngine->setScoOverHCI(hci_sco);}

CBlueChat2Dlg(CWnd* pParent = NULL); // standard constructor
virtual ~CBlueChat2Dlg();

// Dialog Data
//{{AFX_DATA(CBlueChat2Dlg)

```

```

enum { IDD = IDD_BCTEF1_DIALOG };
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CBlueChat2Dlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
//}}AFX_VIRTUAL

// Implementation
protected:
    CBlueChat2DlgAutoProxy* m_pAutoProxy;
    HICON m_hIcon;

    BOOL CanExit();

    // find connection index from handles
    int FindScoConnection(unsigned int conn_handle);
    int FindAclConnection(unsigned int conn_handle);
    int FindAclConnection(BTDEVICE_ADDRESS addr);

    // remove the connections from our stores
    void RemoveScoConnection(int index);
    void RemoveConnection(int index);

    LPDIRECTSOUNDBUFFER lpDsb;
    LPDIRECTSOUND lpDirectSound;
    uint32 buffer_size;
    uint32 frequency;

    LPDIRECTSOUNDCAPTUREBUFFER lpDscpb;
    LPDIRECTSOUNDCAPTURE lpDirectSoundCapture;
    uint32 capture_buffer_size;

    CBlueChat2Engine                * mEngine;
    BlueCoreControllerFactory        * mFactory;
    HCI_INQUIRY_RESULT_EVENT
    HCIInquiryResultEventData;

    HCI_CONNECTION_COMPLETE_EVENT
    AclConnectionCompleteEventData;

    HCI_CONNECTION_COMPLETE_EVENT
    ScoConnectionCompleteEventData;

    HCI_SEND_DATA
    HCISendData;

```

```

        HCI_SEND_DATA *
HCIRceivedData;
        uint8 *
HCISendDataMessage;
        HCI_CMD_COMPLETE_READ_BUFFER_SIZE_EVENT *
HCICmdCompReadBufferSizeEventData;
        HCI_CMD_COMPLETE_READ_BD_ADDR_EVENT *
HCICmdCompReadBdAddrEventData;

// and 16 bit flags to indicate we are forcing a pcm mapped sco connection at this
index
// these values need to be 0 or 1 only
bool pcm_mapped[MAX_NO_SLAVE_CONNECTIONS];

// The slave to slave connections of our slave connections (if we are a master(
HCI_CONNECTION_COMPLETE_EVENT
** SlaveToSlaveConnections;
// is this slave wanting to broadcast to all others?
bool
broadcast[MAX_NO_SLAVE_CONNECTIONS];

Console InputWindow;
Console HCIDisplay;

// Generated message map functions
//{{AFX_MSG(CBlueChat2Dlg)
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnDestroy();
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnClose();
virtual void OnOK();
virtual void OnCancel();
afx_msg void OnInquire();
afx_msg void OnMaster();
afx_msg void OnSlave();
afx_msg void OnConnect();
afx_msg void OnDisconnect();
afx_msg void OnAudio();
afx_msg void OnData();
afx_msg void OnSelchangeRemoteBdaddr();
afx_msg void OnTimer(UINT nIDEvent);
afx_msg void OnBrowse();
afx_msg void OnSend();
afx_msg void OnInquireCancel();

```



```

#ifdef BROADCAST_SCO_DEMO
    afx_msg void OnSelchangeSlaveToSlave();
    afx_msg void OnBroadcast();
    afx_msg void OnDisconnectAll();
    afx_msg void OnSlaveToSlaveConnect();
#endif

//}}AFX_MSG
afx_msg void OnHCIIquiryResultEvent( WPARAM Dummy,
void * InquiryEventData);
afx_msg void OnConnectionComplete( WPARAM
Dummy, void * ConnectionCompleteEventData);
afx_msg void OnDisconnectComplete( WPARAM
Dummy, void * DisconnectionCompleteEventData);
afx_msg void SendChar( WPARAM
nChar, LPARAM dummy);
afx_msg void ReceiveL2CAPDataPacket ( WPARAM Dummy,
void * L2CAPDataPacket);
afx_msg void OnInquiryComplete( WPARAM
dummy, void * HCIDataPacket);
afx_msg void OnCommandComplete( unsigned int Opcode,
void * HCIDataPacket);
afx_msg void OnHCIEvent( unsigned int EventCode, void
* HCIDataPacket);
afx_msg void OnHCIFileMore( WPARAM
dummy, LPARAM ldummy);
afx_msg void UpdateProgressBar( WPARAM dummy,
LPARAM ldummy);
// Msg to recieve a play to sco request
afx_msg LRESULT OnPlaytoSCOMsg(WPARAM wParam, LPARAM lParam);

DECLARE_MESSAGE_MAP()

void AppendHCIDisplay(CString Message);

private:
    void Disconnect(int connection_index);
    void
ProcessHciEvConnectionRequest(HCI_CONNECTION_REQUEST_EVENT *)
HCIEventDataPacket);

    void ProcessHciEvCommandStatus(HCI_COMMAND_STATUS_EVENT *)
HCICmdStatusEvent);
    void SendHCIMessage();

```

```

    void StoreConnectionEventData(HCI_CONNECTION_COMPLETE_EVENT *
LocalConnectionData, HCI_CONNECTION_COMPLETE_EVENT * ConnectionData);
    void DisconnectReset(unsigned short Reason);
    void ResetAddressListBox(int id);
    void ResetAddressListBoxes();

void SendInitialCommands ();

    void                ProcessHCICommandRequest(PARAMETER_ARRAY
HCICommandArray);

    // Send a character to a specific connection
    void SendCharToConnection(char nChar, int connection_index);

// pretend-L2CAP Packet handling
void checkStatusOfPacket ( L2CAP_PACKET_T * todo );
void SortOutFileName ( L2CAP_PACKET_T * todo );
int WriteToFile(char *Message, int Length);

bool DiscardFileData;

    bool Audio_flag;
    // flags to indicate we have set the write voice properly for the given sco
connection
    bool WriteVoiceSet[MAX_NO_SLAVE_CONNECTIONS];
    // array for our need to capture
    bool capture_need[MAX_NO_SLAVE_CONNECTIONS];
    bool need_to_capture()
    {
        bool retval = false;
        for (int i = 0; i < MAX_NO_SLAVE_CONNECTIONS; i++)
            retval |= capture_need[i];
        return retval;
    }
    bool CommsStarted;
    // no of connections we have
    int no_connected;

    bool m_in_play_to_sco;
    uint16 newFileNameLength;
    uint16 newFileNameRecvedLength;
    bool FileBeingSent;
    bool FileBeingReceived;
    // the index of the connection being sent.
    int FileSendConnIndex;
    bool FileSent;

```

```

        bool Master;
        bool TryingToDisconnect;
    bool AboutToClose;
    bool FirstPacket;
    bool HiResClock;
    bool InquiryScanAllowed;
    bool PageScanAllowed;
    bool ReallyCloseComplete;
    bool Scanning;
    bool WaitingForFirstPacket;
    bool WaitOnInit;
    void FlushBuffers(bool playing);

    CStdioFile *NewFile;
    CStdioFile *ReadFile;

    unsigned long int    NewFileLength;
    unsigned long int    NewFileTotalLength;
    unsigned long int    FilePositionCounter;
    unsigned long int    ReadFileLength;
        unsigned int        HCIpacketsPerL2CAPpacket;
        unsigned int        PacketCounter;
    unsigned int        WaitCounter;

    DWORD                FileStartTime;
    CString               NewFileName;

    // Packet type to be used for SCO connection
    uint16 m_sco_packet_type[MAX_NO_SLAVE_CONNECTIONS];

    //Make an address string out of the passed address
    static CString MakeAddressString(BTDEVICE_ADDRESS* address);

#ifdef BROADCAST_SCO_DEMO
    // Add/Remove address to our slave to slave combo box
    void AddAddressToSlaveToSlave(HCI_CONNECTION_COMPLETE_EVENT*
conn_complete);
    void RemoveFromSlaveToSlave(HCI_CONNECTION_COMPLETE_EVENT*
conn_complete);

    // update the state of the slave to slave connection controls
    void UpdateS2SConnectButton();
    void UpdateS2SListBox();
#endif
    void UpdateDisconnectButton(int conn_index);

```

```
        // Make Sco connections associated with passed connection index, choosing to
ignore any
        // "connected" slaves or not. Also if we wish the connection to be mapped over
pcm
        // whatever the Pskey value is, we can pass true for the third fiels
void MakeScoConnection(int connection_index,
                        bool ignore_connected_slave,
                        bool force_pcm_mapping = false);
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif
#ifdef(AFX_BLUECHAT2DLG_H__FBFB9FAD_233B_4B09_9E0C_DA841BF93A
F9__INCLUDED_)
```

Appendix C: CRD7309C Hardware Guide

C.1 Description of the CRD7309C Reference Kit

Cirrus Logic®, Inc. offers the CRD7309C Reference Kit to systems designers who wish to develop solid-state digital audio player products based on the EP7312 processor. In this application, the EP7312 acts as a system-on-chip (SOC) ultra low power audio decoder. This document is intended to aid the developer in understanding the precepts of player development. Discussed are the resources found in the kit, their functions, and how to use them.

C.2 Hardware Details

Main feature set

Primary features of the development board are as follows:

- 1 Cirrus Logic EP7312 microprocessor
- 8 MBytes NOR flash
- 8 MBytes NAND flash
- 1 128x33 backlit LCD display
- 10 user input buttons
- 1 USB port
- 1 Smart Media™ Connector and Interface
- 1 stereo ADC
- 1 stereo DAC
- 1 headphone output

- 1 JTAG interface
- nPOR and nBROM buttons
- 1 serial port

Memory Map

Table 1-A: Memory Map

Physical Address Space	Width	Description
0x0000.0000 – 0x007F.FFFF	16	NOR FLASH bank, 8MB
0x1000.0000 – 0x1FFF.FFFF	8	NAND FLASH/SmartMedia
0x2000.0000 – 0x2FFF.FFFF	16	128x33 LCD Controller
0x4000.0000 – 0x4FFF.FFFF	8	USB Controller
0x6000.0000 – 0x6000.C000	32	Internal SRAM
0x8000.0000 – 0x8000.2FFF	32	Internal Registers

GPIO Assignment

Table 1-B: GPIO Assignment

Pin	I/O	Description
PA(0-7)	I	8 User Buttons
PB0	O	ALE signal to NAND FLASH
PB1	O	CLE signal to NAND FLASH
PB2	O	nWP signal to NAND FLASH
PB3	O	nCS signal to SmartMedia socket
PB4	O	nCS signal to first NAND FLASH
PB5	O	nCS signal to second NAND FLASH
PB6	O	Power Control for SmartMedia Interface
PB7	I	READY/nBUSY signal from NAND FLASH
PD0	O	Clock line to CS43L43 and CS53L32
PD1	I/O	Data line to CS43L43 and CS53L32
PD2	I	Write protect sense for SmartMedia Interface
PD3	O	nRESET signal to CS43L43
PD4	O	nRESET signal to CS53L32
PD5	O	nCS signal to USB controller

IRQ Assignment

Table 1-C: IRQ Assignment

EP7312 Pin	Description
nEXTFIQ	USB connection sense
nEINT1	USB Controller
nEINT2	SmartMedia removal interrupt
EINT3	SmartMedia Insert interrupt

Memory

There are two types of memory (external to the processor) present on the development board:

- 1. 8 MBytes of NOR FLASH
- 2. 8 MBytes of NAND FLASH

FLASH Memory

The FLASH memory (U22) is provided in one bank, made up of a single device Intel 28F640 J3A-120 Strata Flash used on the development board. Each block in memory may be erased at least 100,000 times. If frequent writes to FLASH are likely, an operating system to manage writes to FLASH sectors by distributing storage of data over the entire addressable space is recommended. The FLASH device has a 120 ns random access time and a 25 ns sequential access time. Program the controller for 3 random and 0 sequential wait states for optimal performance.

Details on erasing, writing, locking and unlocking the sectors of the FLASH devices can be found in the Intel datasheet for the J3 FLASH(see the datasheet section of the CD).

NAND FLASH

There 8 MBytes of NAND FLASH present on the board. The FLASH device installed is a Samsung™ KM29U64000T. In addition to the NAND FLASH devices(s), a SmartMedia connector is present (J2). SmartMedia™ cards can be used to add up to an additional 128 Mbytes of storage capability.

The details of reading, writing and erasing the NAND flash can be found in the data sheet for the KM29U64000T. The development board has a single NAND flash chip populated with a dual footprint for additional NAND flash chips. U6 and U8 can be populated with 16, 32, or 64 MB NAND flash while U7 is populated for 8MB. The removal of U7 will allow for expansion of the NAND flash.

The NAND flash devices are port mapped, so only one address is needed for operations. All accesses to both the on-board and SSFDC NAND FLASH devices are at physical address 0x1000:0000.

Battery Level

Table 1-D: Battery Levels

Pin Name	Voltage (V)
DSR	1.3
DCD	1.2
CTS	1.1
nEXTPWR	1.0

Note: Pin will be high if the battery voltage is greater than or equal to the given voltage.

Development Board Devices

Resetting the Board

The nPOR signal is the processor reset signal and is activated at power-up and when the nPOR button (S10) is pressed. The reference board implements this signal with a simple resistor and capacitor network with a time constant of approximately 10 ms at power-up. Once the EP7312 has been reset, it enters Standby. The board implements an automatic wakeup circuit that will cause the processor to enter operating state. The circuit centered around U18 provides an oscillator which has a duty cycle $> 125 \mu\text{s}$ required for waking up the chip. This is tied to the WAKEUP pin on the processor. Two seconds after the nPOR rise time, the processor will then sense the WAKEUP input and transition into operating state. Once in operating state, the software can tri-state the COL0 pin to halt the oscillator.

USB

The USB interface (U19) provided includes a Phillips™ PDIUSB12D chip for 8 bit wide accesses and conforms to the USB 1.1 specification. USB connection sense (pin nEXTFIQ) indicate presence of a cable and provide an interrupt to the processor. nMWE and nMOE pins connected to U19 are gated through CS4 to provide bus inhibit during off-page accesses.

A/D and D/A Converters

The CS43L32 (U2) is a 24 bit 96 kHz D/A which is synchronized to the DAI clocks SCLK, LRCLK, as well as the externally generated 11.286 MHz MCLK(Y1) signal. Decoded music frames are clocked in and decoded. Programming is done via the two-wire bus interface using SCL and SDA for clocking and data respectively. All initialization, volume, and tone control are adjusted using this interface. Left and right channel audio output is filtered and sent directly to the audio headphone jack (J1). The CS53L32 (U3) digitizes incoming audio. Data is synchronized with the same clocks as used on the CS43L32 and programmed with the same interface. Pads for a microphone are provided on the board; the user must populate these pads with a microphone in order for the record function to work. Volume and gain control for left and right channels are programmable.

LCD Interface

Part LTD43S338LD (J3) provides the interface to the Hitachi™ 66410TB0 for the 128x33 LCD display. LCD is controlled from the main bus. The on-chip LCD controller in the EP7312 is not used. LCD spec No. LM338-1 (see the datasheet section of the CD) contains all mechanical and electrical details for the LCD screen. Controller spec HD66410 (see the datasheet section of the CD) covers the programming of the display. The control registers internally are programmable to adjust brightness, turn LCD on /off etc and operates asynchronously with the EP7312 expansion bus. Display data is stored in the controller's internal RAM. The driver (U14) is a boost converter which provides the high voltage for the LCD backlight.

RS232 Interface

A single RS-232 interface is provided and serves as the download interface for the CRD board. The MAXIM3221ECAE driver (U20) interfaces directly to the EP7312 microprocessor to achieve a maximum baud rate of 115200 (processor limited) between the board and the host PC. The port is bi-directional so as to serve as a supplemental debug interface for text messages or characters.

Debug Interface

(JP1) serves as the JTAG interface or in-circuit emulator devices. Processor pins nTEST1 and nTEST0 are pulled low to allow the user to connect the device without additional configuration. R47 and R48 can be removed to permanently disable this port if desired, which will also result in minor power savings.

Features

Supports the following codecs:

- MP3 playback (decoder library included in this package).
- WMA playback (decoder library and DRM library must be separately obtained from Microsoft).
- AAC playback (decoder library must be separately obtained from Cirrus Logic).
- ACELP.net playback (decoder library must be separately obtained from Cirrus Logic).
- Microsoft™ ADPCM playback and record. Files recorded with MSADPCM can be played back on the PC.

- G.721 playback and record (decoder library must be separately obtained from Cirrus Logic). Files recorded with G.721 can not be played back on the PC.
- Supports InterTrust Technologies Corp.'s Rights/PD™ (decryption library must be separately obtained from InterTrust).

Supports the following third party audio processing effects:

- WOW™ from SRS Labs Inc. (audio processing library must be separately obtained from SRS).
- Q2X™ from QSound Labs, Inc. (audio processing library must be separately obtained from QSound).
- Spatializer™ from Spatializer Audio Laboratories, Inc. (audio processing library must be separately obtained from Spatialzier).

Appendix D: Source code on CRD7309C board for MP3 decoding

```

//*****
//
// MP3.C - Codec interface driver for the MP3 decoder.
//
//*****
#include "globals.h"
#ifdef SUPPORT_MP3
#endif
#include "mp3dec/mpgdata.h"
#include "mp3dec/mpgaudio.h"
#include "buffer/buffer.h"
#include "src/src.h"
#include "../hwdefs.h"

//*****
//
// The name of this codec.
//
//*****
static const unsigned short pusCodecName[] =
{
    'M', 'P', '3', ' ', 'M', 'P', 'E', 'G', ' ', 'L', 'a', 'y',
    'e', 'r', ' ', '3', '\0'
};

//*****
//
// The bit rates at which this codec can encode.
//
//*****
#ifdef SUPPORT_MP3_ENCODE
static const unsigned long pulEncodeBitRates[] =
{
    32000, 40000, 48000, 56000, 64000, 80000, 96000, 112000, 128000, 160000,
    192000, 224000, 256000, /*320000,*/ 0
};
#endif

//*****
//
// A structure which defines the persistent state of the MP3 decoder.
//
//*****
typedef struct
{
    //
    // A boolean that is TRUE if the MP3 codec is opened to encode a file.
    //
    unsigned long bIsEncoder;

    //
    // The following union contains the persistent state of either the MP3
    // encoder or MP3 decoder, depending upon the value of bIsEncoder.

```

```

//
union
{
    //
    // The following structure contains the persistent state of the MP3
    // decoder.
    //
    struct
    {
        //
        // The persistent internal state of the MP3 decoder library.
        //
        tMPEGInstance sMPEGInstance;

        //
        // The file from which we read data.
        //
        tFile *pFile;

        //
        // A buffer to contain the encoded MP3 audio.
        //
        char pcEncodedData[2048];

        //
        // Buffers to contain the decoded MP3 audio.
        //
        short psLeft[MP3_MAX_PCM_LENGTH + MP3_MAX_PCM_LENGTH +
                     NUMTAPS - 1];
        short psRight[MP3_MAX_PCM_LENGTH + MP3_MAX_PCM_LENGTH +
                     NUMTAPS - 1];

        //
        // The buffer to which we write decoded MP3 data.
        //
        BufferState *pOutput;

        //
        // The MP3 bitstream pointer.
        //
        tMPEGBitstream sBS;

        //
        // The decoded MP3 header information.
        //
        tMPEGHeader sHdr;

        //
        // The number of bytes in pcEncodedData which contain valid encoded
        // MP3 data.
        //
        unsigned short usValid;

        //
        // A boolean which is true if this file uses VBR.

```

```
//
unsigned char ucIsVBR;

//
// An unused byte to pad the next member to a word boundary.
//
unsigned char ucUnused;

//
// The offset of the first frame of the file.
//
unsigned long ulFirstFrame;

//
// The byte length of the file.
//
unsigned long ulLength;

//
// The time length of the file.
//
unsigned long ulTimeLength;

//
// The seek points for VBR files.
//
unsigned char pucVBRSeek[100];
} sDecoder;

//

//
// The sample rate of the PCM stream.
//
unsigned short usSampleRate;

//
// The number of channels in the file.
//
unsigned char ucChannels;

//
// An unused byte to pad the next member to a word boundary.
//
unsigned char ucUnused;

//
// The bit rate of the MP3 file.
//
unsigned long ulBitRate;

//
// The number of samples which have been decoded.
//
unsigned long ulTimePos;
```



```

//
// A buffer to contain the song title, if known.
//
unsigned short pusTitle[64];

//
// A buffer to contain the song author, if known.
//
unsigned short pusArtist[64];
} tMP3;

//*****
//
// The following is a mapping from the sample rate descriptor in the
// tMPEGHeader structure to the integer number of samples per second.
//
//*****
static const unsigned short usSRMap[] = { 11025, 12000, 8000, 0,
                                           22050, 24000, 16000, 0,
                                           44100, 48000, 32000, 0 };

//*****
//
// The following is a mapping from the bitrate_index and ID bit in the MPEG
// sync header to a the bitrate of the frame. The first 16 entries are for
// ID=0 (i.e. MPEG-2 or MPEG-2.5 half sample rate) and the second 16 entries
// are for ID=1 (i.e. MPEG-1 full sample rate).
//
//*****
static const unsigned short usBRMap[] = { 0, 8, 16, 24,
                                           32, 40, 48, 56,
                                           64, 80, 96, 112,
                                           128, 144, 160, 0,
                                           0, 32, 40, 48,
                                           56, 64, 80, 96,
                                           112, 128, 160, 192,
                                           224, 256, 320, 0 };

//*****
//
// MP3InitBitstream initializes the stream to the given byte offset in the
// file.
//
//*****
static void
MP3InitBitstream(tMP3 *pMP3, unsigned long ulFilePos)
{
    //
    // Seek to the correct block of data in the file.
    //
    FSeek(pMP3->uInfo.sDecoder.pFile, ulFilePos & ~511);

    //
    // Read the first four pages from this position.

```

```

//
pMP3->uInfo.sDecoder.usValid = FSReadBS(pMP3->uInfo.sDecoder.pFile,
                                         pMP3->uInfo.sDecoder.pcEncodedData,
                                         2048);

//
// Initialize the bitstream pointer structure.
//
pMP3->uInfo.sDecoder.sBS.bufptr =
    (unsigned int *) (pMP3->uInfo.sDecoder.pcEncodedData +
                     (ulFilePos & 508));
pMP3->uInfo.sDecoder.sBS.bitidx = 8 * (ulFilePos & 3);
}

//*****
//
// MP3FindNextFrame finds the next frame in the input MP3 bitstream.
//
//*****
static unsigned long
MP3FindNextFrame(tMP3 *pMP3)
{
    unsigned long ulRead;

    //
    // Make sure the bitstream pointer structure makes sense (i.e. it is
    // pointing to a valid bit in a word, not at the bit following the last
    // bit of the previous word).
    //
    if(pMP3->uInfo.sDecoder.sBS.bitidx == 32)
    {
        pMP3->uInfo.sDecoder.sBS.bufptr++;
        pMP3->uInfo.sDecoder.sBS.bitidx = 0;
    }

    //
    // If the bitstream pointer is not at the beginning of the local bitstream
    // buffer, then copy the remaining data back to the beginning.
    //
    if((int)pMP3->uInfo.sDecoder.sBS.bufptr !=
       (int)pMP3->uInfo.sDecoder.pcEncodedData)
    {
        //
        // Copy the remainder of the data from the end of the local bitstream
        // buffer to the beginning.
        //
        memcpy(pMP3->uInfo.sDecoder.pcEncodedData,
               pMP3->uInfo.sDecoder.sBS.bufptr,
               2048 - ((int)pMP3->uInfo.sDecoder.sBS.bufptr -
                      (int)pMP3->uInfo.sDecoder.pcEncodedData));

        //
        // Update the count of valid bytes in the local bitstream buffer.
        //
        ulRead = ((int)pMP3->uInfo.sDecoder.sBS.bufptr -

```

```

        (int)pMP3->uInfo.sDecoder.pcEncodedData);
    if (ulRead > pMP3->uInfo.sDecoder.usValid)
    {
        pMP3->uInfo.sDecoder.usValid = 0;
    }
    else
    {
        pMP3->uInfo.sDecoder.usValid -=
            ((int)pMP3->uInfo.sDecoder.sBS.bufptr -
             (int)pMP3->uInfo.sDecoder.pcEncodedData);
    }

    //
    // Update the bitstream pointer structure.
    //
    pMP3->uInfo.sDecoder.sBS.bufptr =
        (unsigned int *)pMP3->uInfo.sDecoder.pcEncodedData;
}

//
// See if there is any additional MP3 data that we can copy into the local
// bitstream buffer.
//
if (pMP3->uInfo.sDecoder.usValid <= (2048 - 512))
{
    //
    // Compute the number of bytes to read.
    //
    ulRead = (2048 - pMP3->uInfo.sDecoder.usValid) & ~511;

    //
    // Read MP3 data into the end of the local bitstream buffer. Since the
    // ARM MP3 decoder needs the bytes within each word swapped, perform
    // the byte swap as we read.
    //
    pMP3->uInfo.sDecoder.usValid +=
        FSReadBS(pMP3->uInfo.sDecoder.pFile,
                 (pMP3->uInfo.sDecoder.pcEncodedData +
                  pMP3->uInfo.sDecoder.usValid), ulRead);
}

//
// Find the next sync word.
//
if (MP3SearchForSyncword(&(pMP3->uInfo.sDecoder.sMPEGInstance),
                        &(pMP3->uInfo.sDecoder.sBS),
                        pMP3->uInfo.sDecoder.usValid * 8) != eNoErr)
{
    //
    // We could not find a sync word, so return an error.
    //
    return(0);
}

//

```

```

// Decode the header.
//
if (MP3DecodeInfo(&(pMP3->uInfo.sDecoder.sMPEGInstance),
                  &(pMP3->uInfo.sDecoder.sBS),
                  &(pMP3->uInfo.sDecoder.sHdr)) != eNoErr)
{
    //
    // Skip back past the word that was consumed by the attempt to decode
    // the header.
    //
    pMP3->uInfo.sDecoder.sBS.bufptr--;

    //
    // We could not decode the header, so return an error.
    //
    return(0);
}

//
// Success.
//
return(1);
}

//*****
//
// MP3DecodeVBRHeader determines if the current frame is a Xing VBR header
// frame and decodes it if it is.
//
//*****
static unsigned long
MP3DecodeVBRHeader(tmp3 *pMP3)
{
    unsigned char *pucPtr;
    unsigned long ulFlags, ulFrames, ulIdx, ulFrameSize;

    //
    // Get the offset to the beginning of the current frame.
    //
    pucPtr = ((unsigned char *)pMP3->uInfo.sDecoder.sBS.bufptr +
              (pMP3->uInfo.sDecoder.sBS.bitidx >> 3));

    //
    // Determine the offset of the header.
    //
    if (pMP3->usSampleRate >= 32000)
    {
        //
        // This is a MPEG-1 file. Is this a mono or stereo stream?
        //
        if (pMP3->ucChannels == 1)
        {
            //
            // This is a mono stream, so the header starts at the 17th byte of
            // the frame data.

```

```

        //
        pucPtr += 17;
    }
    else
    {
        //
        // This is a stereo stream, so the header starts at the 32nd byte
        // of the frame data.
        //
        pucPtr += 32;
    }

    //
    // The frame size for MPEG-1 files is 1152.
    //
    ulFrameSize = 1152;
}
else
{
    //
    // This is a MPEG-2 file. Is this a mono or stereo stream?
    //
    if(pMP3->ucChannels == 1)
    {
        //
        // This is a mono stream, so the header starts at the 9th byte of
        // the frame data.
        //
        pucPtr += 9;
    }
    else
    {
        //
        // This is a stereo stream, so the header starts at the 17th byte
        // of the frame.
        //
        pucPtr += 17;
    }

    //
    // The frame size for MPEG-2 files is 576.
    //
    ulFrameSize = 576;
}

//
// See if the "Xing" signature appears in the frame.
//
if((pucPtr[3] != 'X') || (pucPtr[2] != 'i') ||
    (pucPtr[1] != 'n') || (pucPtr[0] != 'g'))
{
    //
    // This is not a Xing VBR header, so return a failure.
    //
    return(0);
}

```



```

}

//
// Skip past the signature.
//
pucPtr += 4;

//
// Get the header flags from the stream.
//
ulFlags = (pucPtr[3] << 24) | (pucPtr[2] << 16) |
          (pucPtr[1] << 8) | pucPtr[0];
pucPtr += 4;

//
// If the number of frames exists in the header, then extract it now.
//
if(ulFlags & 1)
{
    ulFrames = (pucPtr[3] << 24) | (pucPtr[2] << 16) |
              (pucPtr[1] << 8) | pucPtr[0];
    pucPtr += 4;
}
else
{
    //
    // The count of frames does not exist, so we will not be able to
    // properly support this file.
    //
    return(0);
}

//
// If the file size exists in the header, then skip it.
//
if(ulFlags & 2)
{
    pucPtr += 4;
}

//
// If the seek point table exists in the header, then read it in now.
//
if(ulFlags & 4)
{
    //
    // There are 100 entries in the table, so copy them one by one.
    //
    for(ulIdx = 0; ulIdx < 100; ulIdx++)
    {
        pMP3->uInfo.sDecoder.pucVBRSeek[ulIdx ^ 3] = *pucPtr++;
    }

    //
    // Since the seek point table exists in the header, indicate that this

```

```

        // is a VBR file.
        //
        pMP3->uInfo.sDecoder.ucIsVBR = 1;
    }
    else
    {
        //
        // Since there is no seek point table in the header, treat this file as
        // a non-VBR file.
        //
        pMP3->uInfo.sDecoder.ucIsVBR = 0;
    }

    //
    // Now, compute the bitrate from the file size and the count of frames in
    // the file.
    //
    pMP3->ulBitRate = ((FSLength(pMP3->uInfo.sDecoder.pFile) / ulFrames) *
                      pMP3->usSampleRate) / (ulFrameSize / 8);

    //
    // Success.
    //
    return(1);
}

//*****
//
// MP3DecodeID3Tag sees if the MP3 stream has an ID3 tag. If found, the
// song title and artist is extracted from the tag.
//
//*****
static void
MP3DecodeID3Tag(tMP3 *pMP3)
{
    unsigned long ulDataStart, ulLength;

    //
    // Save the original position and length of the MP3 data.
    //
    ulDataStart = pMP3->uInfo.sDecoder.ulFirstFrame;
    ulLength = (pMP3->uInfo.sDecoder.ulLength +
                pMP3->uInfo.sDecoder.ulFirstFrame);

    //
    // See if the file contains any ID3 tags. If it does, ulFirstFrame and
    // ulLength in the MP3 persistent data structure will be updated to reflect
    // the start and length of the actual MP3 data, without the ID3 tags.
    //
    ID3FindTags(pMP3->uInfo.sDecoder.pFile, &pMP3->uInfo.sDecoder.ulFirstFrame,
                &pMP3->uInfo.sDecoder.ulLength,
                (unsigned char *)pMP3->uInfo.sDecoder.pcEncodedData);

    //
    // Get the value of the "TPE1" tag, which contains the name of the artist.

```

```

//
if(ID3GetTagValue(pMP3->uInfo.sDecoder.pFile, ulDataStart, ulLength,
    (unsigned char *)pMP3->uInfo.sDecoder.pcEncodedData,
    "TPE1", pMP3->pusArtist, 64) == 0)
{
    pMP3->pusArtist[0] = 0;
}

//
// Get the value of the "TIT2" tag. which contains the name of the song.
//
if(ID3GetTagValue(pMP3->uInfo.sDecoder.pFile, ulDataStart, ulLength,
    (unsigned char *)pMP3->uInfo.sDecoder.pcEncodedData,
    "TIT2", pMP3->pusTitle, 64) == 0)
{
    pMP3->pusTitle[0] = 0;
}
}

//*****
//
// MP3AddID3Frame adds a frame to the ID3-V2 tag.
//
//*****
#ifdef SUPPORT_MP3_ENCODE
static unsigned char *
MP3AddID3Frame(unsigned char *pucBuffer, char *pcTag,
    unsigned short *pusValue)
{
    unsigned long ulLength, ulIdx;

    //
    // Add the name of the tag to the frame.
    //
    *pucBuffer++ = *pcTag++;
    *pucBuffer++ = *pcTag++;
    *pucBuffer++ = *pcTag++;
    *pucBuffer++ = *pcTag++;

    //
    // Get the length of the value string.
    //
    ulLength = wcslen(pusValue);

    //
    // Search the value string to see if there are any non-ASCII characters in
    // it.
    //
    for(ulIdx = 0; ulIdx < ulLength; ulIdx++)
    {
        if((pusValue[ulIdx] < 0x20) || (pusValue[ulIdx] > 0x7e))
        {
            break;
        }
    }
}

```

```

//
// See if any non-ASCII characters were found.
//
if(ulIdx == ulLength)
{
    //
    // The string is entirely ASCII, so add the length of the string after
    // conversion to ASCII to the frame.
    //
    *pucBuffer++ = ((ulLength + 2) >> 21) & 0x7f;
    *pucBuffer++ = ((ulLength + 2) >> 14) & 0x7f;
    *pucBuffer++ = ((ulLength + 2) >> 7) & 0x7f;
    *pucBuffer++ = (ulLength + 2) & 0x7f;

    //
    // The frame flags are all zero.
    //
    *pucBuffer++ = 0x00;
    *pucBuffer++ = 0x00;

    //
    // Set the string format to ASCII.
    //
    *pucBuffer++ = 0x00;

    //
    // Copy the string to the frame, converting each character to ASCII.
    //
    for(ulIdx = 0; ulIdx < ulLength; ulIdx++)
    {
        *pucBuffer++ = *pusValue++;
    }

    //
    // NULL terminate the string.
    //
    *pucBuffer++ = 0x00;
}
else
{
    //
    // There are non-ASCII characters in the string, so add the length of
    // the Unicode string to the frame.
    //
    *pucBuffer++ = (((ulLength * 2) + 5) >> 21) & 0x7f;
    *pucBuffer++ = (((ulLength * 2) + 5) >> 14) & 0x7f;
    *pucBuffer++ = (((ulLength * 2) + 5) >> 7) & 0x7f;
    *pucBuffer++ = ((ulLength * 2) + 5) & 0x7f;

    //
    // The frame flags are all zero.
    //
    *pucBuffer++ = 0x00;
    *pucBuffer++ = 0x00;
}

```

```

    //
    // Set the string format to Unicode.
    //
    *pucBuffer++ = 0x01;

    //
    // Indicate that the Unicode string is in little-endian format.
    //
    *pucBuffer++ = 0xff;
    *pucBuffer++ = 0xfe;

    //
    // Copy the string to the frame.
    //
    for(ulIdx = 0; ulIdx < ulLength; ulIdx++)
    {
        *pucBuffer++ = pusValue[ulIdx] & 0xff;
        *pucBuffer++ = pusValue[ulIdx] >> 8;
    }

    //
    // NULL terminate the string.
    //
    *pucBuffer++ = 0x00;
    *pucBuffer++ = 0x00;
}

//
// Return the byte in the tag to be used for the next frame.
//
return(pucBuffer);
}
#endif

//*****
//
// Determines if the given file is a MP3 file, and returns the offset of the
// first frame if so.
//
//*****
static unsigned long
MP3IsMP3File(tFile *pFile, unsigned char *pucScratch, unsigned long ulOffset)
{
    unsigned long ulSamplesPerFrame, ulSampleRate, ulBitRate, ulFrameLength;
    unsigned long ulPos, ulCount, ulLoop, ulHdr1, ulHdr2, ulNext;
    unsigned char pucBuffer[16];

    //
    // Seek to the start of the file data.
    //
    FSeek(pFile, ulOffset & ~511);

```



```

//
// Read 512 bytes from the file.
//
FSRead(pFile, pucScratch, 512);

//
// Get the starting offset of the data in the buffer.
//
ulPos = ulOffset & 511;

//
// See if there are 10 bytes of valid data in the buffer.
//
if((ulPos + 10) > 512)
{
    //
    // Get the count of valid bytes in the buffer.
    //
    ulCount = 512 - ulPos;

    //
    // Copy the valid bytes from the buffer into our temporary buffer.
    //
    memcpy(pucBuffer, pucScratch + ulPos, ulCount);

    //
    // Read the next 512 bytes from the file.
    //
    FSRead(pFile, pucScratch, 512);

    //
    // Copy the next set of valid bytes from the buffer into our temporary
    // buffer.
    //
    memcpy(pucBuffer + ulCount, pucScratch, 16 - ulCount);
}
else
{
    //
    // There are 10 bytes of valid data in the buffer, so simply copy them
    // into our temporary buffer.
    //
    memcpy(pucBuffer, pucScratch + ulPos, 16);
}

//
// See if this is a valid ID3 V2 tag.
//
ulCount = ID3IsV2Tag(pucBuffer);
if(ulCount != 0)
{
    //
    // There is an ID3 V2 tag, so skip past it.
    //
    ulOffset += ulCount + 10;
}

```

```

}

//
// Seek to the start of the MP3 stream.
//
FSeek(pFile, ulOffset & ~511);

//
// Read 512 bytes from the file.
//
FSRead(pFile, pucScratch, 512);

//
// Get the starting offset of the data in the buffer.
//
ulPos = ulOffset & 511;

//
// Get the offset of the start of data in the buffer.
//
ulOffset &= ~511;

//
// Examine the next 2048 bytes of the file (which is four 512 byte chunks)
// looking for a frame header.
//
for(ulLoop = 4; ; ulPos++)
{
    //
    // See if there are 4 bytes of valid data in the buffer.
    //
    if((ulPos + 4) > 512)
    {
        //
        // If we've examined 2048 bytes and not found a frame header then
        // we will not be able to decode this file.
        //
        if(ulLoop == 0)
        {
            return(-1);
        }

        //
        // Get the count of valid bytes in the buffer.
        //
        ulCount = 512 - ulPos;

        //
        // Copy the valid bytes from the buffer into our temporary buffer.
        //
        memcpy(pucBuffer, pucScratch + ulPos, ulCount);

        //
        // Read the next 512 bytes from the file.
        //

```

```

FSRead(pFile, pucScratch, 512);

//
// Indicate the number of bytes in the temporary buffer.
//
ulPos = 0 - ulCount;

//
// Update the offset of the start of data in the buffer.
//
ulOffset += 512;

//
// Decrement the count of pages to search.
//
ulLoop--;
}

//
// If there are residual bytes in the temporary buffer, then
// concatenate them with the bytes from the beginning of the current
// block.
//
if(ulPos > 512)
{
    //
    // Get the first byte from the temporary buffer.
    //
    ulHdr1 = pucBuffer[0 - ulPos] << 24;

    //
    // See if this is the last byte in the temporary buffer.
    //
    if((ulPos + 1) == 0)
    {
        //
        // This was the last byte in the temporary buffer, so get the
        // remaining three bytes from the beginning of the current
        // block.
        //
        ulHdr1 |= ((pucScratch[0] << 16) |
                   (pucScratch[1] << 8) |
                   pucScratch[2]);
    }
    else
    {
        //
        // There is at least one more byte in the temporary buffer, so
        // get the second byte from the temporary buffer.
        //
        ulHdr1 |= pucBuffer[1 - ulPos] << 16;

        //
        // See if this is the last byte in the temporary buffer.
        //

```

```

        if((ulPos + 2) == 0)
        {
            //
            // This was the last byte in the temporary buffer, so get
            // the remaining two bytes from the beginning of the
            // current block.
            //
            ulHdr1 |= ((pucScratch[0] << 8) | pucScratch[1]);
        }
        else
        {
            //
            // There is one more byte in the temporary buffer, so get
            // the last byte from the temporary buffer and the first
            // byte from the current block.
            //
            ulHdr1 |= ((pucBuffer[2 - ulPos] << 8) | pucScratch[0]);
        }
    }
}
else
{
    //
    // Get the next four bytes from the current block.
    //
    ulHdr1 = ((pucScratch[ulPos] << 24) |
              (pucScratch[ulPos + 1] << 16) |
              (pucScratch[ulPos + 2] << 8) |
              pucScratch[ulPos + 3]);
}

//
// See if this word starts with 0xFFE, which is the frame sync for MPEG
// audio.
//
if((ulHdr1 & 0xffe00000) != 0xffe00000)
{
    continue;
}

//
// See that a valid MPEG version was specified.
//
if((ulHdr1 & 0x00180000) == 0x00080000)
{
    continue;
}

//
// Make sure that the file is encoded with layer 3.
//
if((ulHdr1 & 0x00060000) != 0x00020000)
{
    continue;
}

```

```

//
// Make sure that a valid bit rate is specified.
//
if((ulHdr1 & 0x0000f000) == 0x0000f000)
{
    continue;
}

//
// Make sure that a valid sample rate is specified.
//
if((ulHdr1 & 0x00000c00) == 0x00000c00)
{
    continue;
}

//
// If the bit rate index is zero (indicating free-format bit rate), we
// simply assume that this is a valid frame header in a valid file.
//
if((ulHdr1 & 0x0000f000) == 0x00000000)
{
    return(ulOffset + ulPos);
}

//
// Determine if this frame is MPEG-1 or MPEG-2/2.5.
//
if((ulHdr1 & 0x00080000) == 0x00080000)
{
    //
    // This frame is MPEG-1, which has 1152 samples per frame.
    //
    ulSamplesPerFrame = 1152;

    //
    // Get the sample rate of this frame.
    //
    ulSampleRate = usSRMap[((ulHdr1 & 0x00000c00) >> 10) + 8];

    //
    // Get the bit rate of this frame.
    //
    ulBitRate = usBRMap[((ulHdr1 & 0x0000f000) >> 12) + 16];
}
else
{
    //
    // This frame is MPEG-2/2.5, which has 576 samples per frame.
    //
    ulSamplesPerFrame = 576;

    //
    // Determine if this frame is MPEG-2 or MPEG-2.5.

```



```

//
if((ulHdr1 & 0x00100000) == 0x00100000)
{
    //
    // This frame is MPEG-2. Get the sample rate of this frame.
    //
    ulSampleRate = usSRMap[(ulHdr1 & 0x00000c00) >> 10] + 4];
}
else
{
    //
    // This frame is MPEG-2.5. Get the sample rate of this frame.
    //
    ulSampleRate = usSRMap[ulHdr1 & 0x00000c00 >> 10];
}

//
// Get the bit rate of this frame.
//
ulBitRate = usBRMap[(ulHdr1 & 0x0000f000) >> 12];
}

//
// Determine the length of this frame.
//
ulFrameLength = (ulBitRate * 125 * ulSamplesPerFrame) / ulSampleRate;
if(ulHdr1 & 0x00000200)
{
    ulFrameLength++;
}

//
// Determine the offset of the next frame header.
//
ulNext = ulOffset + ulPos + ulFrameLength;

//
// Determine if the next frame header is in the currently read portion
// of the file.
//
if((ulNext - ulOffset) >= 512)
{
    //
    // Seek to the position within the file that contains the next
    // frame header.
    //
    FSSeek(pFile, ulNext & ~511);

    //
    // Read the 512 bytes from the file.
    //
    FSRead(pFile, pucScratch, 512);
}

//

```

```

// Get the offset into the buffer of the next frame header.
//
ulNext &= 511;

//
// Get the frame header from the buffer. Note that this could read
// past the end of the buffer by three bytes.
//
ulHdr2 = ((pucScratch[ulNext] << 24) |
          (pucScratch[ulNext + 1] << 16) |
          (pucScratch[ulNext + 2] << 8) |
          pucScratch[ulNext + 3]);

//
// See if the next frame header wraps into the next page of the file.
//
if(ulNext > 508)
{
    //
    // Read the next 512 bytes from the file.
    //
    FSRead(pFile, pucScratch, 512);

    //
    // Determine how many bytes to replace from the newly read page.
    //
    switch(ulNext)
    {
        //
        // We have three valid bytes from the previous page.
        //
        case 509:
        {
            //
            // Mask off the valid bytes from the previous page.
            //
            ulHdr2 &= 0xffffffff00;

            //
            // Add in the new byte from this page.
            //
            ulHdr2 |= pucScratch[0];

            //
            // We're done.
            //
            break;
        }

        //
        // We have two valid bytes from the previous page.
        //
        case 510:
        {
            //

```

```

        // Mask off the valid bytes from the previous page.
        //
        ulHdr2 &= 0xffff0000;

        //
        // Add in the new bytes from this page.
        //
        ulHdr2 |= (pucScratch[0] << 8) | pucScratch[1];

        //
        // We're done.
        //
        break;
    }

    //
    // We have one valid byte from the previous page.
    //
    case 511:
    {
        //
        // Mask off the valid byte from the previous page.
        //
        ulHdr2 &= 0xff000000;

        //
        // Add in the new bytes from this page.
        //
        ulHdr2 |= ((pucScratch[0] << 16) |
                    (pucScratch[1] << 8) | pucScratch[2]);

        //
        // We're done.
        //
        break;
    }
}

//
// See if the two headers match. We only compare the sync word, ID,
// layer, and sample rate fields (since the others can change from
// frame to frame).
//
if((ulHdr2 & 0xfffe0c00) == (ulHdr1 & 0xfffe0c00))
{
    //
    // The two headers match, so return the offset of the first header
    // as the starting position of MP3 data in the file.
    //
    return(ulOffset + ulPos);
}

//
// See if we performed a read to find the next frame header.

```

```

    //
    if(FSTell(pFile) != (ulOffset + 512))
    {
        //
        // Seek back to the original file position.
        //
        FSSeek(pFile, ulOffset);

        //
        // Read 512 bytes from the original file position.
        //
        FSRead(pFile, pucScratch, 512);
    }
}

//*****
//
// MP3SetupDecoder initializes the MP3 decoder.
//
//*****
unsigned long
MP3SetupDecoder(tMP3 *pMP3, unsigned long ulStart, tFile *pFile)
{
    //
    // Initialize the MP3 library.
    //
    InitMP3Audio(&(pMP3->uInfo.sDecoder.sMPEGInstance));

    //
    // Save the pointer to the file structure.
    //
    pMP3->uInfo.sDecoder.pFile = pFile;

    //
    // Save the position of the beginning of the file.
    //
    pMP3->uInfo.sDecoder.ulFirstFrame = ulStart;

    //
    // Get the length of the file.
    //
    pMP3->uInfo.sDecoder.ulLength =
        (FSLength(pMP3->uInfo.sDecoder.pFile) -
         pMP3->uInfo.sDecoder.ulFirstFrame);

    //
    // Decode the ID3 tag if it is present.
    //
    MP3DecodeID3Tag(pMP3);

    //
    // Determine the starting position of the first frame.
    //
    pMP3->uInfo.sDecoder.ulFirstFrame =

```

```

    MP3IsMP3File(pMP3->uInfo.sDecoder.pFile,
                 (unsigned char *)pMP3->uInfo.sDecoder.pcEncodedData,
                 pMP3->uInfo.sDecoder.ulFirstFrame);
if(pMP3->uInfo.sDecoder.ulFirstFrame == -1)
{
    //
    // We could not find a valid first frame, so return an error.
    //
    return(0);
}

//
// Initialize the bitstream.
//
MP3InitBitstream(pMP3, pMP3->uInfo.sDecoder.ulFirstFrame);

//
// Find the first frame of the file.
//
if(MP3FindNextFrame(pMP3) == 0)
{
    //
    // If we could not find the first frame of the file, then return an
    // error.
    //
    return(0);
}

//
// Get the sample rate of the file.
//
pMP3->usSampleRate = usSRMap[pMP3->uInfo.sDecoder.sHdr.sample_rate];

//
// Get the number of channels in the file.
//
pMP3->ucChannels = pMP3->uInfo.sDecoder.sHdr.numchans;

//
// Attempt to decode the VBR header.
//
if(MP3DecodeVBRHeader(pMP3) == 0)
{
    //
    // There is no VBR header on this file, so get the actual bitrate from
    // the sync header.
    //
    pMP3->ulBitRate =
        usBRMap[((pMP3->uInfo.sDecoder.sHdr.packed_info & 0x00080000) >>
                  15) |
                ((pMP3->uInfo.sDecoder.sHdr.packed_info & 0x0000f000) >>
                  12)] * 1000;

    //
    // If no bitrate is specified, then return an error.

```



```

    //
    if(pMP3->ulBitRate == 0)
    {
        return(0);
    }

    //
    // Indicate that this is not a VBR file.
    //
    pMP3->uInfo.sDecoder.ucIsVBR = 0;
}

//
// Get the length of the file.
//
pMP3->uInfo.sDecoder.ulTimeLength =
    (((pMP3->uInfo.sDecoder.ulLength * 8) / pMP3->ulBitRate) * 1000) +
    (((pMP3->uInfo.sDecoder.ulLength * 8) % pMP3->ulBitRate) * 1000) /
    pMP3->ulBitRate);

//
// The position is zero.
//
pMP3->ulTimePos = 0;

//
// There is no output buffer.
//
pMP3->uInfo.sDecoder.pOutput = 0;

//
// Indicate that the MP3 codec is configured as a decoder.
//
pMP3->bIsEncoder = 0;

//
// Success.
//
return(1);
}

//*****
//
// The codec plug-in entry point for the MP3 decoder.
//
//*****
unsigned long
MP3Ioctl(unsigned long ulIoctl, unsigned long ulParam1, unsigned long ulParam2,
          unsigned long ulParam3, unsigned long ulParam4)
{
    //
    // Determine what to do based on the specified IOCTL.
    //
    switch(ulIoctl)

```

```

{
    //
    // Return the name for this codec.
    //
    case IOCTL_CODEC_GETNAME:
    {
        const unsigned short **ppusName;

        //
        // The second parameter is a pointer for the name.
        //
        ppusName = (const unsigned short **)ulParam2;

        //
        // Return the name of this codec. The first four characters are
        // the short name and the string starting at the fifth character
        // is the long name.
        //
        *ppusName = pusCodecName;

        //
        // Success.
        //
        return(1);
    }

    //
    // Return the name of the artist.
    //
    case IOCTL_CODEC_GETARTIST:
    {
        unsigned short **ppusName;
        tMP3 *pMP3;

        //
        // The first parameter is a pointer to the MP3 persistent data.
        //
        pMP3 = (tMP3 *)ulParam1;

        //
        // The second parameter is a pointer for the name.
        //
        ppusName = (unsigned short **)ulParam2;

        //
        // Return the name of the artist.
        //
        *ppusName = pMP3->pusArtist;

        //
        // Success.
        //
        return(1);
    }
}

```

```

//
// Return the name of the song
//
case IOCTL_CODEC_GETTITLE:
{
    unsigned short **ppusName;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // The second parameter is a pointer for the name.
    //
    ppusName = (unsigned short **)ulParam2;

    //
    // Return the name of the song.
    //
    *ppusName = pMP3->pusTitle;

    //
    // Success.
    //
    return(1);
}

//
// Return the bitrate at which this file is encoded.
//
case IOCTL_CODEC_GETBITRATE:
{
    unsigned long *pulBitRate;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // The second parameter is a pointer for the bitrate.
    //
    pulBitRate = (unsigned long *)ulParam2;

    //
    // Return the average bitrate of the file.
    //
    *pulBitRate = pMP3->ulBitRate;

    //
    // Success.
    //

```

```

    return(1);
}

//
// Return the sample rate at which this file is encoded.
//
case IOCTL_CODEC_GETSAMPLERATE:
{
    unsigned long *pulSampleRate;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // The second parameter is a pointer for the sample rate.
    //
    pulSampleRate = (unsigned long *)ulParam2;

    //
    // Return the sample rate of the file.
    //
    *pulSampleRate = pMP3->usSampleRate;

    //
    // Success.
    //
    return(1);
}

//
// Return the number of channels in the file.
//
case IOCTL_CODEC_GETCHANNELS:
{
    unsigned long *pulChannels;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // The second parameter is a pointer for the number of channels.
    //
    pulChannels = (unsigned long *)ulParam2;

    //
    // Return the number of channels in the file.
    //
    *pulChannels = pMP3->ucChannels;
}

```

```

    //
    // Success.
    //
    return(1);
}

//
// Return the length (in milliseconds) of the file.
//
case IOCTL_CODEC_GETLENGTH:
{
    unsigned long *pullLength;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // The second parameter is a pointer for the number of
    // milliseconds.
    //
    pullLength = (unsigned long *)ulParam2;

    //
    // Return the length of the file.
    //
    if(pMP3->bIsEncoder)
    {
        *pullLength = (((pMP3->ulTimePos / pMP3->usSampleRate) * 1000) +
                        (((pMP3->ulTimePos % pMP3->usSampleRate) *
                          1000) / pMP3->usSampleRate));
    }
    else
    {
        *pullLength = pMP3->uInfo.sDecoder.ulTimeLength;
    }

    //
    // Success.
    //
    return(1);
}

//
// Return the current position (in milliseconds) within the file.
//
case IOCTL_CODEC_GETTIME:
{
    unsigned long *pulTime;
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent data.

```



```

//
pMP3 = (tMP3 *)ulParam1;

//
// The second parameter is a pointer for the number of seconds.
//
pulTime = (unsigned long *)ulParam2;

//
// Determine the time based on the sample rate.
//
*pulTime = (((pMP3->ulTimePos / pMP3->usSampleRate) * 1000) +
            (((pMP3->ulTimePos % pMP3->usSampleRate) * 1000) /
             pMP3->usSampleRate));

//
// Success.
//
return(1);
}

//
// Determine if the given file can be decoded.
//
case IOCTL_CODEC_QUERY:
{
    tFile *pFile;
    unsigned char *pucScratch;

    //
    // The second parameter is the file to be checked.
    //
    pFile = (tFile *)ulParam2;

    //
    // The third parameter is a 512 byte scratch buffer.
    //
    pucScratch = (unsigned char *)ulParam3;

    //
    // Determine if this is a MP3 file.
    //
    if(MP3IsMP3File(pFile, pucScratch, ulParam4) == -1)
    {
        return(0);
    }

    //
    // This is a MP3 file.
    //
    return(1);
}

//
// Determine if this codec can encode.

```

```

    //
    case IOCTL_CODEC_HASENCODER:
    {
#ifdef SUPPORT_MP3_ENCODE
        //
        // MP3 includes an encoder.
        //
        return(1);
#else
        //
        // There is no encoder.
        //
        return(0);
#endif
    }

    //
    // Prepare the codec to encode or decode a file.
    //
    case IOCTL_CODEC_OPEN:
    {
        tMP3 *pMP3;

        //
        // The first parameter is a pointer to the MP3 persistent state.
        //
        pMP3 = (tMP3 *)ulParam1;

        //
        // Make sure that we have enough memory for this decoder.
        //
        if((ulParam1 + sizeof(tMP3)) > ulExtentOfRAM)
        {
            return(0);
        }

        //
        // See if an encoder or decoder is being opened.
        //
        if(ulParam3 & CODEC_OPEN_DECODE)
        {
            //
            // Setup the MP3 decoder.
            //
            return(MP3SetupDecoder(pMP3, ulParam2, (tFile *)ulParam4));
        }
        else
        {
            //
            // If the MP3 encoder is supported, set it up now.
            //
#ifdef SUPPORT_MP3_ENCODE
                return(MP3SetupEncoder(pMP3, (tFile *)ulParam4));
#else
                return(0);
            #endif
        }
    }

```

```

#endif
    }
}

//
// Set the output buffer for the decoder.
//
case IOCTL_CODEC_SETBUFFER:
{
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent state.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // See if the MP3 codec was opened as an encoder or decoder.
    //
    if(pMP3->bIsEncoder)
    {
#ifdef SUPPORT_MP3_ENCODE
        //
        // The second parameter is a pointer to the input buffer.
        //
        pMP3->uInfo.sEncoder.pInput = (BufferState *)ulParam2;

        //
        // Provide the input buffer with our data buffers.
        //
        if(pMP3->ucChannels == 2)
        {
            BufferSetBuffer(pMP3->uInfo.sEncoder.pInput,
                           pMP3->uInfo.sEncoder.pDRAM->psLeft,
                           pMP3->uInfo.sEncoder.pDRAM->psRight,
                           MP3_MAX_PCM_LENGTH * 10);
        }
        else
        {
            BufferSetBuffer(pMP3->uInfo.sEncoder.pInput,
                           pMP3->uInfo.sEncoder.pDRAM->psLeft,
                           pMP3->uInfo.sEncoder.pDRAM->psLeft,
                           MP3_MAX_PCM_LENGTH * 10);
        }
    }
#endif
    }
else
    {
        //
        // The second parameter is a pointer to the output buffer.
        //
        pMP3->uInfo.sDecoder.pOutput = (BufferState *)ulParam2;

        //
        // Provide the output buffer with our data buffers.
    }
}

```

```

    //
    if(pMP3->ucChannels == 2)
    {
        BufferSetBuffer(pMP3->uInfo.sDecoder.pOutput,
                        pMP3->uInfo.sDecoder.psLeft,
                        pMP3->uInfo.sDecoder.psRight,
                        MP3_MAX_PCM_LENGTH + MP3_MAX_PCM_LENGTH);
    }
    else
    {
        BufferSetBuffer(pMP3->uInfo.sDecoder.pOutput,
                        pMP3->uInfo.sDecoder.psLeft,
                        pMP3->uInfo.sDecoder.psLeft,
                        MP3_MAX_PCM_LENGTH + MP3_MAX_PCM_LENGTH);
    }
}

//
// Success.
//
return(1);
}

//
// Determine if the next frame of data can be decoded.
//
case IOCTL_CODEC_CAN_DECODE:
{
    tMP3 *pMP3;

    //
    // The first parameter is a pointer to the MP3 persistent state.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // See if there is space available for decoding the next frame.
    //
    if(BufferSpaceAvailable(pMP3->uInfo.sDecoder.pOutput) <
        MP3_MAX_PCM_LENGTH)
    {
        //
        // There is not enough space available, so we can not decode
        // the next frame.
        //
        return(0);
    }

    //
    // We can decode the next frame.
    //
    return(1);
}

//
// Decode a frame of data.

```

```

//
case IOCTL_CODEC_DECODE:
{
    tMP3 *pMP3;
    short *psLeft, *psRight;
    long lLength;
    tMPEGStatus sStatus;

    //
    // The first parameter is a pointer to the MP3 persistent state.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // Wait until there is space in the output buffer. This tells us
    // when it is OK to write more data into the output buffer.
    //
    while(BufferSpaceAvailable(pMP3->uInfo.sDecoder.pOutput) <
        MP3_MAX_PCM_LENGTH)
    {
        //
        // Busy wait until space is available.
        //
    }

    //
    // Get a pointer to the available space in the output buffer.
    //
    BufferGetWritePointer(pMP3->uInfo.sDecoder.pOutput, &psLeft,
        &psRight, &lLength);

    //
    // Decode the data frame.
    //
    sStatus = MP3DecodeData(&(pMP3->uInfo.sDecoder.sMPEGInstance),
        psLeft, psRight,
        &(pMP3->uInfo.sDecoder.sBS));
    if((sStatus != eNoErr) && (sStatus != eFrameDiscarded))
    {
        //
        // We could not decode the frame, so see if there is more data
        // to be decoded.
        //
        if((FSTell(pMP3->uInfo.sDecoder.pFile) <
            pMP3->uInfo.sDecoder.ulLength) ||
            (((int)pMP3->uInfo.sDecoder.sBS.bufptr -
            (int)pMP3->uInfo.sDecoder.pcEncodedData) <
            pMP3->uInfo.sDecoder.usValid))
        {
            //
            // There is more data to be decoded, so re-initialize the
            // MP3 library (to try and minimize the artifacting).
            //
            InitMP3Audio(&(pMP3->uInfo.sDecoder.sMPEGInstance));
        }
    }
}

```



```

else
{
    //
    // There is no more data to be decoded, so return an error
    // to indicate that we are at the end of the file.
    //
    return(0);
}

//
// Make sure that this frame did not use garbage at the end of the
// encoded data buffer to complete the decode.
//
if(((int)pMP3->uInfo.sDecoder.sBS.bufptr -
    (int)pMP3->uInfo.sDecoder.pcEncodedData) >
    pMP3->uInfo.sDecoder.usValid)
{
    //
    // All the data for this frame was not present, so see if there
    // is more data in to be decoded.
    //
    if(FSTell(pMP3->uInfo.sDecoder.pFile) <
        pMP3->uInfo.sDecoder.ulLength)
    {
        //
        // There is more data to be decoded, so move the buffer
        // pointer back to the end of the valid data.
        //
        pMP3->uInfo.sDecoder.sBS.bufptr =
            (unsigned int *) (pMP3->uInfo.sDecoder.pcEncodedData +
                pMP3->uInfo.sDecoder.usValid);
        pMP3->uInfo.sDecoder.sBS.bitidx = 32;

        //
        // Force the frame status to discarded.
        //
        sStatus = eFrameDiscarded;

        //
        // Re-initialize the MP3 library (to try and minimize the
        // artifacting).
        //
        InitMP3Audio(&(pMP3->uInfo.sDecoder.sMPEGInstance));
    }
}
else
{
    //
    // There is no more data to be decoded, so return an error
    // to indicate that we are at the end of the file.
    //
    return(0);
}
}

```

```

//
// Determine the length of each frame, based on the sample rate of
// the file.
//
lLength = pMP3->usSampleRate >= 32000 ? 1152 : 576;

//
// If this frame was not discarded, then update the output buffer's
// write pointer.
//
if(sStatus != eFrameDiscarded)
{
    //
    // Update the output buffer's write pointer.
    //
    BufferUpdateWritePointer(pMP3->uInfo.sDecoder.pOutput,
                           lLength);

    //
    // Increment the time based on the number of samples.
    //
    pMP3->ulTimePos += lLength;
}

//
// Find the next frame in the bitstream.
//
while(MP3FindNextFrame(pMP3) == 0)
{
    //
    // We could not find another frame, so see if there is more
    // data to be decoded.
    //
    if(((FSTell(pMP3->uInfo.sDecoder.pFile) <
           pMP3->uInfo.sDecoder.ulLength) ||
        (((int)pMP3->uInfo.sDecoder.sBS.bufptr -
          (int)pMP3->uInfo.sDecoder.pcEncodedData) <
         pMP3->uInfo.sDecoder.usValid))
    {
        //
        // There is more data to be decoded, so advance the buffer
        // pointer by one byte.
        //
        if(pMP3->uInfo.sDecoder.sBS.bitidx == 32)
        {
            pMP3->uInfo.sDecoder.sBS.bufptr++;
            pMP3->uInfo.sDecoder.sBS.bitidx = 8;
        }
        else
        {
            pMP3->uInfo.sDecoder.sBS.bitidx += 8;
        }

        //
        // Re-initialize the MP3 library (to try and minimize the

```

```

        // artifacting).
        //
        InitMP3Audio(&(pMP3->uInfo.sDecoder.sMPEGInstance));
    }
    else
    {
        //
        // There is no more data to be decoded, so return an error
        // to indicate that we are at the end of the file.
        //
        return(0);
    }
}

//
// Success.
//
return(1);
}

//
// Seek to the specified time position.
//
case IOCTL_CODEC_SEEK:
{
    tMP3 *pMP3;
    unsigned long ulPos, ulPos2, ulFrameSize, ulPercent;

    //
    // The first parameter is a pointer to the MP3 persistent state.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // Make sure that the specified time is valid.
    //
    if(ulParam2 > pMP3->uInfo.sDecoder.ulTimeLength)
    {
        ulParam2 = pMP3->uInfo.sDecoder.ulTimeLength;
    }

    //
    // If this file uses VBR, then we use the seek point table to
    // compute the new position.
    //
    if(pMP3->uInfo.sDecoder.ucIsVBR)
    {
        //
        // Convert the seek position into a percentage through the
        // file.
        //
        ulPos = (ulParam2 * 100) / pMP3->uInfo.sDecoder.ulTimeLength;

        //
        // Compute the fractional percentage between this percentage
        // and the next.

```

```

//
ulPercent = (((ulParam2 * 100) %
              pMP3->uInfo.sDecoder.ulTimeLength) * 100) /
              pMP3->uInfo.sDecoder.ulTimeLength);

//
// Get the two seek points that surround the requested
// position.
//
if (ulPos >= 99)
{
    ulPos2 = 256;
}
else
{
    ulPos2 = pMP3->uInfo.sDecoder.pucVBRSeek[ulPos + 1];
}
if (ulPos > 99)
{
    ulPos = 256;
}
else
{
    ulPos = pMP3->uInfo.sDecoder.pucVBRSeek[ulPos];
}

//
// Interpolate between the two seek points.
//
ulPos = ((ulPos * (pMP3->uInfo.sDecoder.ulLength / 256)) +
         ((ulPos2 - ulPos) * ulPercent *
          (pMP3->uInfo.sDecoder.ulLength / 256)) / 100) +
         pMP3->uInfo.sDecoder.ulFirstFrame);

//
// The new time position is the requested seek position. This
// may be off by a little bit, but there's nothing that can be
// done about this (other than to decode the file from the
// beginning).
//
pMP3->ulTimePos = ((ulParam2 / 1000) * pMP3->usSampleRate) +
                  (((ulParam2 % 1000) * pMP3->usSampleRate) /
                   1000);
}
else
{
    //
    // Figure out the size of each frame based on the sample rate.
    //
    if (pMP3->usSampleRate >= 32000)
    {
        ulFrameSize = 1152;
    }
    else
    {

```

```

        ulFrameSize = 576;
    }

    //
    // Compute the number of frames that occur before the requested
    // time position.
    //
    ulPos = (((ulParam2 / 1000) * pMP3->usSampleRate) /
            ulFrameSize) +
            (((ulParam2 % 1000) * pMP3->usSampleRate) /
            (ulFrameSize * 1000));

    //
    // Compute the time for the computed frame number.
    //
    pMP3->ulTimePos = ulPos * ulFrameSize;

    //
    // Compute the file position based on the actual seek time.
    //
    ulPos = (((ulPos / pMP3->usSampleRate) *
            (pMP3->ulBitRate / 8)) +
            (((ulPos % pMP3->usSampleRate) *
            (pMP3->ulBitRate / 8)) / pMP3->usSampleRate) +
            pMP3->uInfo.sDecoder.ulFirstFrame);
    if(ulPos)
    {
        ulPos--;
    }
}

//
// Initialize the MP3 library.
//
InitMP3Audio(&(pMP3->uInfo.sDecoder.sMPEGInstance));

//
// Initialize the bitstream to the position.
//
MP3InitBitstream(pMP3, ulPos);

//
// Find the next frame in the input MP3 bitstream.
//
while(1)
{
    //
    // Look for the next frame.
    //
    if(MP3FindNextFrame(pMP3) == 1)
    {
        //
        // Make sure that the packed_info for the frame we found
        // matches (where appropriate) the packed_info for the
        // first frame in the file.

```



```

        //
        if((pMP3->usSampleRate ==
            usSRMap[pMP3->uInfo.sDecoder.sHdr.sample_rate]) &&
            (pMP3->ucChannels ==
            pMP3->uInfo.sDecoder.sHdr.numchans))
        {
            break;
        }
    }

    //
    // We could not find the next frame, so skip forward in the
    // input bitstream by a byte.
    //
    pMP3->uInfo.sDecoder.sBS.bitidx += 8;
    if(pMP3->uInfo.sDecoder.sBS.bitidx >= 32)
    {
        pMP3->uInfo.sDecoder.sBS.bufptr++;
        pMP3->uInfo.sDecoder.sBS.bitidx -= 32;
    }
}

//
// Success.
//
return(1);
}

//
// Cleanup after the codec.
//
case IOCTL_CODEC_CLOSE:
{
#ifdef SUPPORT_MP3_ENCODE
    tMP3 *pMP3;
    unsigned long ulNewBytes;

    //
    // The first parameter is a pointer to the MP3 persistent state.
    //
    pMP3 = (tMP3 *)ulParam1;

    //
    // See if the MP3 encoder is being closed.
    //
    if(pMP3->bIsEncoder)
    {
        //
        // See if any data was encoded.
        //
        if(pMP3->ulTimePos != 0)
        {
            //
            // Finish the MP3 stream.
            //

```

```

        MP3EncFinish(pMP3->uInfo.sEncoder.pvHandle,
                    pMP3->uInfo.sEncoder.pDRAM->pucEncodedData +
                    pMP3->uInfo.sEncoder.ulValid, &ulNewBytes);

        //
        // Increment the amount of data in the encoded data buffer.
        //
        pMP3->uInfo.sEncoder.ulValid += ulNewBytes;

        //
        // Write the remainder of the MP3 stream to the file.
        //
        FSWrite(pMP3->uInfo.sEncoder.pFile,
                pMP3->uInfo.sEncoder.pDRAM->pucEncodedData,
                pMP3->uInfo.sEncoder.ulValid);
    }

    //
    // Release the previously allocated DRAM.
    //
    DRAMRelease(pMP3->uInfo.sEncoder.pDRAM);
}

#endif

    //
    // Success.
    //
    return(1);
}

//
// We should never get here, but just in case...
//
default:
{
    //
    // Return a failure.
    //
    return(0);
}
}

#endif

```