ECE Senior Design Project Final Documentation

for

Remote Recording System

6 May 2004

Sponsor: Leighton R. Wall, Advanced Input Devices

Team Dialect

Justin M. Cassidy
cass0664@uidaho.edu

Ian D. Olson
olso4398@uidaho.edu

Thomas R. Stear
stea7935@uidaho.edu

Instructor: Dr. Richard Wall

Project Advisor: Dr. Richard Wall

I. INTRODUCTION 1

A. Background 1
B. Intended Use 1
C. Problem Solution 1
D. Functional and Non-functional Parts of the Project 1

II. OPERATIONAL SPECIFICATION 2

A. Operational Overview 2
B. Operational Metaphor 2

III. FUNCTIONAL SPECIFICATION 3

A. Functional Overview 3
B. Functional Metaphor 3
C. Functions 4

IV. MANUFACTURING 6

A. Product Lifecycle Plan 6
B. Failure Rate Calculations 9
C. Failure Modes and Effect Analysis 10

V. SOCIAL IMPACT 10

A. Health and Safety Issues 10
B. Environmental Issues 11

VI. SPECIFICATIONS NOT YET MET 11

VII. FUTURE MODIFICATIONS 11

A. Host 11
B. Remote 11

VIII. APPENDICES 2

A. USER'S MANUAL 3

Introduction 3
About this guide 3
Installation 3
Initial Setup 4
Using the System 5
Troubleshooting 5

B. SPECIFICATIONS 7

C. ORGANIZATION OF CODE AND DOCUMENTS ON ACCOMPANYING CD 8

D. BILL OF MATERIALS 9

E.1 WINDOWS DEVELOPMENT ENVIRONMENT DOCUMENTATION 10

E.1.1 Introduction 10
E.1.2 The DirectX Software Development Kit (SDK) 11
E.1.3 Host Development: Microsoft Visual C++ .NET 12
E.1.4 Remote Development: Microsoft eMbedded C++ 4.0 14
E.1.5 Windows CE Platform Builder and Windows CE Emulator 16

E.2 WINDOWS API DOCUMENTATION 17

E.2.1 Introduction 17

E.2.2 DIRECTSHOW 18
E.2.3 WINDOWS SOCKETS 27
E.2.4 AFX OBJECTS CSOCKET, CCESOCKET, CSOCKETFILE, AND CARCHIVE 34
E.2.5 DIRECTSOUND BUFFERS 38
E.2.6 MICROSOFT WAVEFORM API 42
E.2.7 BENEFITS AND DISADVANTAGES OF WINDOWS PROGRAMMING 49
F. TECHNICAL NOTES SHEETS 50
G. REFERENCES 64

Remote Recording System using Wireless Networking in Windows platforms (May 2004)

Justin M. Cassidy, Ian D. Olson, and Thomas R, Stear

***Abstract*—This paper describes in detail the process we undertook to develop a system centering around a handheld unit capable of transmitting voice and audio data over a wireless network to a host computer for further processing, including but not limited to voice recognition. Among the named processes are a few technologies that could not be made to function by us. They are included merely as reference.**

***Index Terms*—Wireless Networking (Wi-Fi), Windows CE, DirectX, Waveform API, Windows sockets.**

I. Introduction

A. Background

The Wi-Fi enabled Remote Recording System was designed to transmit audio data over Wi-Fi networks between a handheld Windows CE device and a Windows XP desktop host computer. The user can transmit audio over the Wi-Fi connection to the host computer that can perform any of various audio tasks with the data: archival, voice-recognition, or audio recording. The purpose of this project was to replace an existing system that is based on a USB connection, rather than a wireless network connection. The trouble users have with the USB system is that the user has to stay tied to a desk in order to use the system.

B. Intended Use

The intention for this project was to replace the existing system, which is being used in a hospital by doctors. The users of the existing system primarily take advantage of a voice recognition program to make notes in electronic charts for their patients. With a wireless network connection, users would be able to wander around a room where the host is located, or perhaps even use the system in an entirely faraway place, so long as the remote and host can communicate with each other over the wireless network.

C. Problem Solution

In solving this problem, we forged our way through several different methods and Programming Interfaces before finally settling on the Windows Waveform API. Initially, we had thought it would do us all a favor to use a simply interface for programming, such as DirectX. Microsoft made the DirectX suite of API's so that programmers could have access to the speed and power of hardware without worrying about manipulating the hardware on their own. As a result, games, Internet programs, and other multimedia rich applications use generic calls to DirectX libraries.

After several months of developing for the handheld unit our sponsor asked us to use, we discovered that the libraries we were using were not installed on the handheld, and there was no way for us to put those libraries on the handheld device, a critical step in launching our application. While our code worked very nicely on desktop computers, we were unable to make it run on the handheld device. At this point we looked to something much simpler.

The Waveform API is a low level interface that is purportedly the lowest level of programming available that will easily accommodate audio data transmission. With some generic buffering for data collection, a touch of Windows Sockets programming to make the connection, and Waveform API for audio capture and playback, we had constructed a system that accomplished our two major goals in this project: audio capture and transmission from a handheld device over Wi-Fi, and playback of the original recorded sound from the host back to the remote unit.

D. Functional and Non-functional Parts of the Project

There were a few other goals we had set our minds on in the planning stages of this project. We also wanted to tie our program into a third-party voice recognition program for dictation. Our sponsor recommended IBM's ViaVoice for this end, since that is what they use in their solutions.

We also had plans to control "tracks" or a history of recorded or transmitted sound clips. Since no data is stored on the remote, the archival or read-back would have to be managed on the host end. We learned that we could likely

This work was supported in part by the Electrical and Computer Engineering department at the University of Idaho, Moscow, Idaho, and by Advanced Input Devices, Coeur d'Alene, Idaho.

J. M. Cassidy is a graduating senior in Computer Engineering at the University of Idaho, Moscow, Idaho, 83843, USA. (phone: 208-883-4455; e-mail: cass0664@uidaho.edu).

I. D. Olson is a graduating senior in Computer Engineering at the University of Idaho, Moscow, Idaho, 83843, USA. (e-mail: olso4398@uidaho.edu).

T. R. Stear is a graduating senior in Computer Engineering at the University of Idaho, Moscow, Idaho, 83843, USA. (e-mail: stea7935@uidaho.edu).

control the host's playback and recording operations from the remote unit through message passing, a standard Windows procedure that allows an application to perform concurrent operations. To manage the tracks or sound clips, we had planned on creating a small table of file pointers every time the host application started. In this manner, skipping tracks from the remote would simply open a new sound file from the chronological listing and begin to play it back over the network. This would need a drastic modification to tie directly into the voice recognition program and hear what the recognizer thought you just said, so the pointer table would have been for the more musical user, rather than the dictation user.

At this point, the only operations that are correctly working are the recording of audio data and the playback of that same audio data. The host is not yet automatic in that someone has to also sit at the host computer and tell it to start listening or sending. Another shortcoming in our current system is that the operations have no graceful stop. In order to switch from recording to playback, the users (of both ends) must shut down the applications and restart them, then change the modes they use by answering the prompts differently.

We have not yet tried to attach voice recognition to the host. Our audio data is sampled such that the fidelity should be more than good enough to accommodate descent, accurate voice recognition, but we cannot say that is true with any certainty. This part of the project then is entirely untested.

II. Operational Specification

A. Operational Overview

The driver program and the wireless remote sound recording unit (SRU) work together to take audio recorded on the device and provide it to the host computer as a virtual microphone input. Together they create a Remote Recording System (RRS).

The RRS starts with the host driver running in the background on a host machine with a wireless network adapter, and a SRU that is powered down. The SRU is turned on, and it performs a quick diagnostic (battery life check, connecting to the host). The SRU can then record or play sound with the press of a button. When in record mode, the microphone picks up sound and the sound is transmitted via DirectShow over the network connection. When in play mode, the host will stream audio to the device for it to play through its internal speaker. Recent recorded audio streams are stored on the host, and the driver can track forward and backward through the stored streams during playback.

B. Operational Metaphor

1) First Use: The driver software is installed on a computer designated to listen to the SRU: a host computer. An IT professional will run the driver software and configure it with one or more SRU's network addresses and names. This will enable the SRU selected in the driver software to communicate with the host when the unit is powered on. While a host can be configured to communicate with multiple remote units, note that only one remote unit should be used at a given moment with the same host.

2) Powering On/Off: The SRU unit has an on/off switch for turning on the unit. A 3.7V rechargeable Lithium-Ion battery powers the unit. When it is turned on, it connects to the host computer via an ad hoc wireless network connection. If it cannot find a host to communicate with, the activity LED will flash red and the SRU will need to be powered off.

3) Recording Sound: The SRU uses an internal microphone to record audio. When the record button is pressed once, the SRU will verify the host is ready to receive audio. The activity LED will turn on, signaling the user to begin talking into the SRU. Either the record or stop buttons can be pressed to finish the record operation.

As the sound is recorded, it is streamed into the host driver as a virtual sound card input, and simultaneously saved as an audio file for playback by host programs or later by the SRU. Ten of these recorded "audio sessions" will be stored as files, numbered from 0 to 9, most recent to least recently recorded. For every hardware address/name pair the host driver is configured for, ten audio sessions will be stored.

4) Playing Sound: The SRU has an internal speaker used for playing back audio. When the play button is pressed once, the SRU will verify the host is ready to send the previously recorded audio session. The activity LED will turn on, signaling the audio has begun playing on the SRU. Either the play or stop buttons can be pressed to finish the play operation.

The previous/next track buttons cycle through the recorded sessions on the host session, selecting a different session to be streamed back to the SRU for playback. Pressing these buttons initiates the playback operations for this audio session. Either the play or stop buttons can be pressed to finish this play operation.

SRU Interface

The built-in speaker is on the back of the unit, and the microphone is on the top.

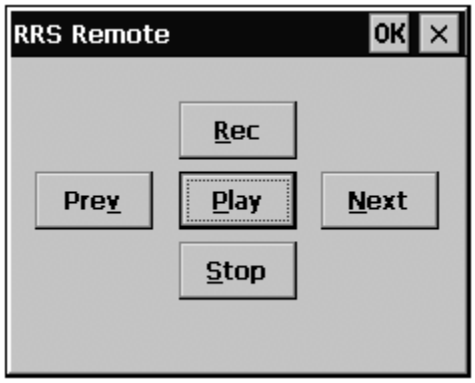


Fig. 1: Remote Application Interface This image is what the interface looks like for the remote application, which runs and the handheld device.

For our implementation, the buttons will be on the screen, and the user can use the pen to tap the buttons and control playback, recording, and track skipping.

Host Interface

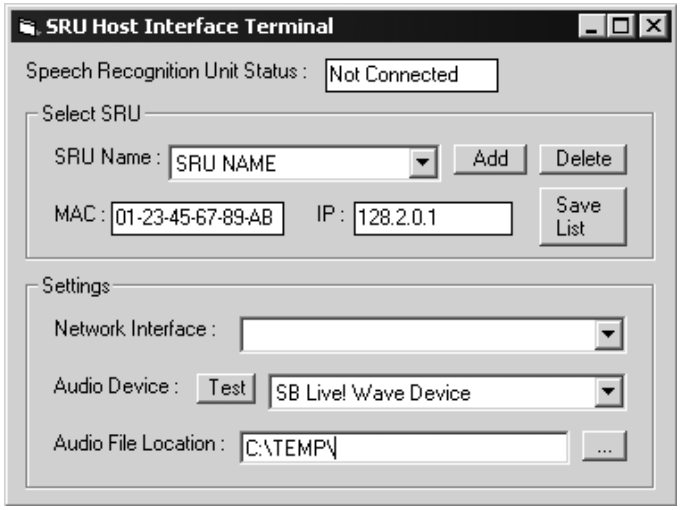


Fig. 2: Host's Remote Unit Management Application This is what the host's SRU management application looks like. There are fields for the nickname for a device, the remote's MAC and IP addresses, and the settings for the host's network and sound card.

The SRU Host Interface Terminal allows the user to add SRUs of any name of choice, and the corresponding network address information for that SRU. The user chooses which SRU to listen for on the network by selecting it from the SRU Name pulldown menu. The user can also select which network interface adapter the program will use to communicate to the SRU, and which sound device to send the decoded audio stream to.

III. Functional Specification

A. Functional Overview

The functionalities of the RRS system fall into three categories: control, recording, and playback. Control involves handshaking functions between SRU and host, recording involves streaming recorded sound over the network to the host, and playback involves streaming saved sound files back over the network to the SRU. Logic flowcharts can be seen below (see fig. 3: Host Flowchart, and fig. 4: Remote Flowchart).

B. Functional Metaphor

The RRS ties the SRU and the host intimately together—one cannot function without the other. Therefore, although functions can be tied to either the SRU or the host, the functions themselves fall into three categories: control, recording, and playback.

Control functions for the system consist of the handshaking signals sent between the SRU and Host in order to either establish the initial connection between the two units, or terminate the connection once one of the units powers off.

Recording functions begin with a handshake between the SRU and host, followed by initializing the audio stream on the SRU side. As the host receives the stream, it routes the data to both a file and a virtual sound input. A similar handshaking method closes the stream, as well as the file and input on the host side.

Playback functions begin with a handshake between the SRU and host, after which the host begins a stream audio to the SRU. The host receives the audio from either the virtual audio device's line input or a recorded file, and the audio is streamed to the SRU. On the SRU, the audio is sent to the on-board sound module, and played through the internal speaker. On the SRU, the user has the ability to cycle through 10 stored files on the host for streaming playback. When the user chooses to play a different recorded session, the SRU handshakes with the host, which skips to the desired file.

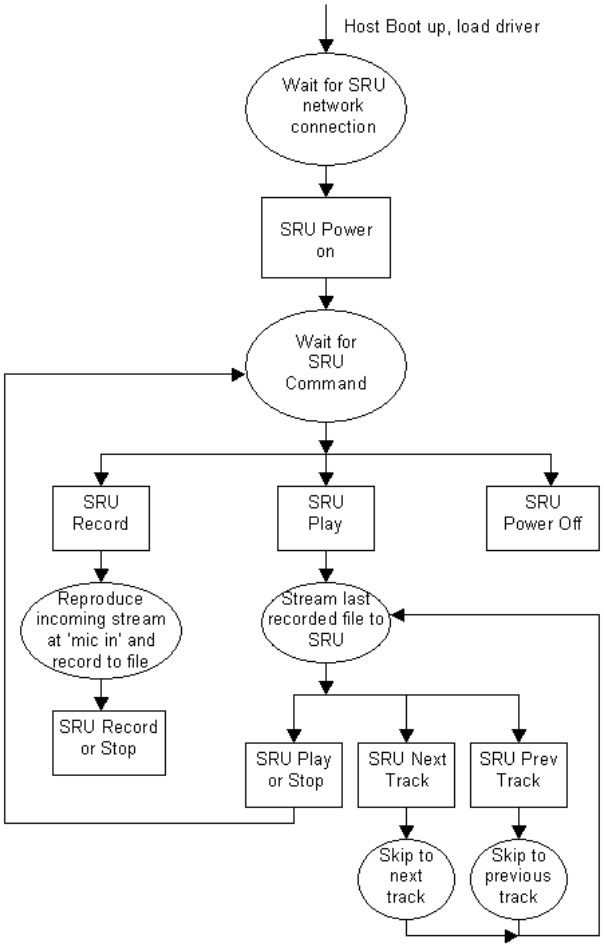


Fig. 3: Host Flowchart This chart shows the logic for the host application running in the background. This application would start with Windows, so that it is ready and waiting for a remote unit all the time.

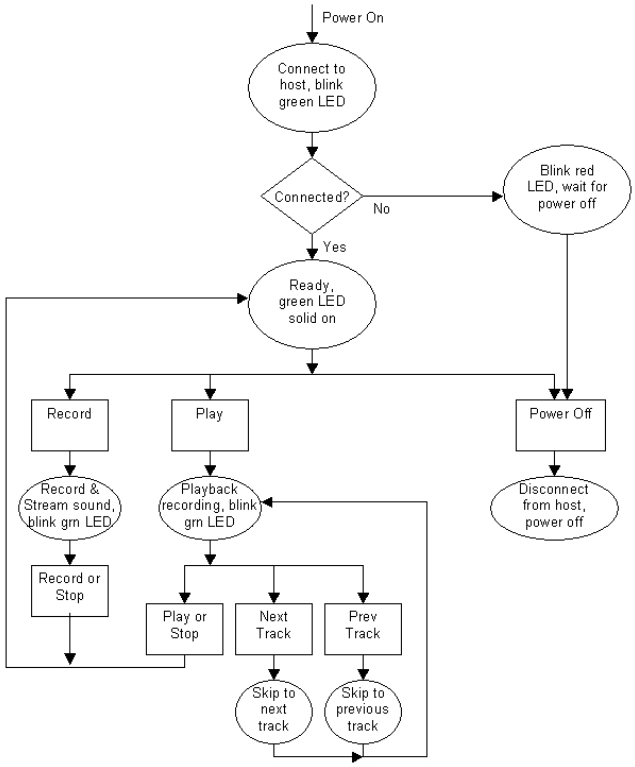


Fig. 4: Remote Flowchart This chart outlines the logic flow for the Handheld application. Through message passing, the remote asks the host to perform control procedures for recording or playback.

C. Functions

1) Control Functions

SRU:

Function: `Host_Connect`
Input: Host IP Address
Occurs: Upon SRU power up
Output: SRU is linked to a host machine
Description: Attempt to connect to the host at the given IP Address over the network, waiting for the listening host computer to respond. The unit can begin normal operation once a host has been found.

Function: `Host_Disconnect`
Input: none
Occurs: On SRU Power Off command
Output: none
Description: Sends a Disconnect control signal over the network to the Host and powers down.

Host:

Function: `Wait_for_SRU`
Input: SRU IP Address
Occurs: Any time a connection between host and SRU is not in place
Output: If the input packet is a host request from the SRU IP, send ACK
Description: Waits for a host request packet from the specified SRU (whose IP address is selected in the SRU Host Interface Terminal).

Function: `SRU_Power_Off`
Input: none
Occurs: when the SRU sends a power off command
Output: none
Description: Host software goes into a power off state where it waits for the SRU to reconnect.

2) Recording Functions

SRU:

Function: `Start_record`
Input: none
Occurs: When the user presses the record button.
Output: Stream creation functions
Description: Creates and sets up the record filter graph and sends a record command to the host and waits for acknowledge from host.

Function: `Begin_stream`
Input: Audio captured from microphone
Occurs: After `Start_Record`, when acknowledge is received from host.
Output: audio data streamed out of network device.
Description: Runs the record filter graph, which begins the audio capture, encode, and network stream process.

Function: `End_stream`
Input: none
Occurs: When the user presses stop or record.
Output: stop stream command to the host.
Description: The filter graph is stopped, and a stop stream command is sent to the host.

Host:

Function: `Host_ready`
Input: none
Occurs: When the host PC has connected to the SRU on the network
Output: acknowledges signal to the SRU that the host is ready
Description: Sends a packet to the SRU over the network that tells the SRU that the host connected and ready to function.

Function: `Begin_Record`
Input: record command from SRU.
Occurs: While the stream is being transmitted from the SRU
Output: Acknowledge message to SRU.
Description: The host sets up the record filter graph, sends an acknowledge message to the SRU, and then runs the filter graph to begin receiving the audio stream and decoding it to the virtual sound device input.

Function: `End_Record`
Input: none
Occurs: stop stream command received from SRU.
Output: none
Description: The record filter graph is stopped.

3) Playback Functions

SRU:

Function: `Start_Playback`
Input: none
Occurs: when user presses the Play button
Output: request for audio stream sent to host
Description: When the user presses the Play button on the SRU, the playback filter graph is created, and a play command is sent to the host to request an audio stream of the last recorded track.

Function: `Play_Sound`
Input: streamed audio from host
Occurs: when the host responds to with a host ready play
Output: audio to the built in speaker or headset
Description: When the host confirms it is ready, the playback filter graph is run and the audio stream from the host begins. The audio is played back through the SRU's built in speaker.

Function: `Change_Track_Forward`
Input: none
Occurs: during playback, when user presses the Track Forward button
Output: skip to file request sent to host
Description: When the user presses the Track Forward button while an audio stream is playing back, a request is sent to the host to stop streaming of the current track and begin streaming of the next recorded track.

Function: `Change_Track_Back`
Input: none
Occurs: during playback, when user presses the Track Back button
Output: skip to file request sent to host
Description: When the user presses the Track Back button while an audio stream is playing back, a request is sent to the host to stop streaming of the current track and begin streaming of the previously recorded track.

Function: `Stop_Playback`
Input: none
Occurs: during playback, when user presses either the Play or Stop button
Output: end stream command sent to host
Description: The playback filter graph is stopped and an end stream command is sent to the host.

Host:

Function: `Host_Ready_Play`
Input: none
Occurs: when a playback stream request is received from the SRU
Output: host ready play packet sent to SRU
Description: When the host receives a playback stream request from the SRU, the host confirms to the SRU that the request was received.

Function: `Begin_Stream`
Input: audio file stored on host machine
Occurs: immediately following the `Host_Ready_Play` function
Output: audio stream to the SRU
Description: The host sets up and runs the playback filter graph, which begins streaming audio from an audio file to the SRU.

Function: `Skip_To_File`
Input: none
Occurs: when a skip to file request is received from the SRU
Output: none
Description: When the SRU sends a skip to file request, the host stops playback of the current file, opens the previous or next recorded file (specified in the request), and begins streaming that file to the SRU.

Function: `End_Stream`
Input: none
Occurs: when the SRU sends an end stream command
Output: none
Description: The playback filter graph is stopped, and streaming of the current audio file is terminated.

IV. Manufacturing

A. Product Lifecycle Plan

1) Introduction

The Remote Recording System (RRS) utilizes three main technologies: Windows CE, DirectX, and 802.11b wireless Internet. Our solution incorporates all three technologies to create a wireless microphone of sorts. With a remote unit, we intend to record and playback audio with help from a host computer that could be in the same room or across the building on a different floor. A doctor can easily make chart notes for a patient from the patient's room. A student could take notes in class on his dorm room computer with the single click of a button. A musician can record studio tracks for a new album from his bedroom at home.

1.1) Overview of Hardware and Software Technologies: The IEEE established 802.11 in 1997 to standardize wireless communications. Since it is not a new technology, it has been tested and revised since its inception for reliability and speed. Wi-Fi is often used implement large-building wide networks with hundreds of computers and printers, or simple, small peer-to-peer networks with just a few computers.

Windows CE is a version of Windows that is lightweight. It is designed to run on many different embedded architectures and RISC microprocessors. Handheld devices, cellular phones, and other miniaturized computers have been built on Windows CE. In spite of being lightweight, it has a broad assortment of applications, and libraries of all kinds have been developed to support those applications.

DirectX has been a part of Windows since Windows 95, when game developers asked for a faster set of tools for programming games in Windows. Since then, DirectX has found other uses and gained APIs for input devices, networking, graphics, audio, and other multimedia.

Since Wi-Fi, Windows CE, and DirectX are widely used, there is plenty of documentation available. Though our solution incorporates 802.11b, it never really directly uses or manages the Wi-Fi connection or hardware. That task is left to Windows. In that respect, we could use just about any type of network connection and, as long as Windows can support it, our solution will achieve similar results. DirectX has long been a part of Windows, and assuming it includes backwards compatibly with older programming interfaces as it has done in the past, our solution will still function with new versions of DirectX. The same goes for new versions of Windows CE. These benefits make the gauging the RRS system's end-of-life difficult (§6.1 and §6.2).

2) Design

2.1) Windows CE Platform Builder and DirectX: The Windows CE toolkit includes a Platform Builder for making custom versions of Windows CE that have only the libraries and headers we need for our system. We are also using eMbedded Visual C++ 4.0, a compiler that targets several different embedded architectures, including MIPS, x86, SH4, and ARM platforms. There is also an Emulator that we can use to load custom OS images, and run and test our application.

The DirectX toolkit includes plenty of sample applications and source code that demonstrates the unique features available to DirectX programmers, as well as the libraries required to implement those features. While this Software Development Kit is targeted at PC versions of Windows rather than the embedded versions, like Windows CE, we can still apply some of the principles demonstrated in the SDK to our project since Windows CE does support a limited set of DirectX tools and libraries, including DirectShow.

2.2) Learning Curve: Since we are new to all of the technologies in our solution, there is a fair amount of learning to do before we can program. The sample code helps quite a bit with the steepness of our learning curve, but it can only do so much. Apart from the Microsoft website, there is little help on the Internet for us, and there are only a few books that promise to offer us any assistance. Many classes and utilities are left for us to learn on our own.

The documentation available for some of the tools we are using is often difficult to understand. It seems that some documentation is out of date or just inaccurate, so when we read the help files, we have to then translate them to apply the directions they give to our build environments. For example, when we first started building Windows CE images, we followed the directions in the help file. Through frustrating experimentation, we learned that there are a few libraries that cannot be included in the image for emulators, along with other small discoveries that caused big problems. Whenever we discover a problem or quirk in the environment, the documentation, or the output of a build, we fully document it for educational purposes and to supply a complete list of instructions and "good" documentation at the end of our project.

2.3) Software Technology Costs: Since we are students, we were able to acquire Windows CE software and development tools from the University of Idaho Computer Science department at no cost.

However, this means we are using educationally licensed tools to build this solution. At some point near the end of the project, we will need to purchase licenses from Microsoft to make the solution we have built available for full use and for retail sale.

2.4) Networking: The 802.11b wireless standard we are using is supported by the handheld unit we are targeting for our solution. Our solution does not directly use Wi-Fi networking, but it asks Windows to handle the transfer of network data in that manner. In the future then, if the standard changes for Wi-Fi networking, our solution will still be able to ask Windows to handle the networking in the that manner. The only changes will be to the Windows CE image: we will have to build a new image that will support the new standards and include all the libraries we need for DirectX to function.

3) Implementation And Testing

Although this is a Computer Engineering Senior Design project, the software nature of the design dictates that most of the testing we will do relates to software. To this end our testing and implementation methodology exists not in hardware testing, but rather software validation and instrumentation. All our instrumentation/verification methods for the separate audio and networking routines will be clearly documented in simple *technical notes* sheets. These sheets also illustrate how the build environment is created, along with the exact software used.

3.1) SRU Playback and Recording Routines: Any time the SRU initiates a playback or a recording session, data will be streamed to and from the SRU and the host. Since the data stream will consist of a DirectShow audio over TCP/IP and 802.11b, we can abstract the network transfer apart from the actual sound functions. Whether sending from the host to the remote unit or vice versa, the transfer code will be the same. Network routine testing is discussed in §3.3. Ensuring audio fidelity for speech recognition is discussed in §3.4.

The playback and record routines need to be both functional and responsive, and this can be verified through step-by-step debugging in tandem with testing. Although functionality can be tested on the emulator, its speed is a limitation, so only the actual device will measure responsiveness properly.

3.2) Host Software and Virtual Audio Device: The host software consists of a front-end for storing SRU profiles, a network driver for receiving streaming audio from the host, and a Virtual Audio Driver. We are using a 3rd party VAD that hasn't been tested explicitly on Windows XP but appears to work for all our uses. However, part of the testing process will be making sure the received raw audio data finds its way to the virtual audio driver and that the software works as advertised.

The Host software will need to be tested initially on the OS specified in the requirements document, Windows XP, in an ad-hoc network configuration with the remote unit. Initial Host software testing will be with a hardware audio interface to eliminate possible problems related to the VAD itself. Every functionality of the remote unit will be tested, such as record, playback, track forward, track back, power on and off, to ensure the host software responds in the specified manner. After the Host software is verified, the VAD will be selected in place of the hardware sound device, and the same tests will be run on the Host software using the VAD, using various different 3rd party packages recording from the VAD audio interface, to ensure the functionalities we require of the VAD work correctly.

3.3) Networking: The job of the DirectX APIs is to abstract the hardware, and in this sense once we have verified the network code over Ethernet, we can take this code and merely change the data-link layer transport medium. Our emulator only has Ethernet capability, but if we can step-by-step debug our routines on that medium, we can simply change the network we're using once we adapt our routines for the PDA hardware. Our development suite has remote step-through debug capability also, for testing on the actual remote unit.

3.4) Fidelity and Latency Testing: DirectShow's Audio Compression Manager filter accepts the audio compression format as one of its arguments. For particular file formats and sample sizes, we can test transmission performance versus recognition quality by simply changing the argument to the ACM filter. Our goal is to get accurate speech-to-text synthesis in real time, but if this proves difficult we will optimize for fidelity rather than

performance. Having an accurate transcription is the highest priority at the hospital, more so than a quick performance.

Latency testing will occur in a number of different indoor and outdoor environments, in order to gauge the usefulness of the RRS in a variety of situations. Our goal for real-time operating distance is 100 feet, and we need to be sure this goal is met under common operating conditions.

3.5) System Testing: After the network and audio routines are completed, we will need to validate our final design. The RRS interfaces need to be easy to use for a doctor, easy to quickly maintain on the host end by trained IT staff, and intuitive enough such that someone who can operate a portable CD player or PDA won't have trouble using the SRU. Although the scope of our senior-year project dictates we may not be redesigning the GUI interface and the hard PDA interface, we will at least validate how well they suit the task.

3.6) Results of Testing: While presenting our project at the University of Idaho's Engineering Expo, we were able to test out system in a crowded room with other Wi-Fi projects also running in the same room. Mr. Cassidy carried the handheld around the auditorium and reached an approximate maximum distance of 150 feet from the host with minor obstructions, and the host computer and the handheld maintained constant connection and communication. However, when Mr. Cassidy carried the handheld unit through the auditorium entrance and began to walk down the brick corridor, the two computers lost communication. On the other hand, when the remote was brought back within range of the host, they began communicating again and transmitting data. We felt that we had satisfied the network test over distances and "noisy" network conditions.

We also tested latency of the system by snapping into the microphone on the handheld unit and holding the handheld close to the speakers on the host. In this manner, we recorded a snap and its respective echo through the system. We then looked at the waveform sample in a sound editing program and measured half a second of total system latency. We also calculated the total theoretical system latency.

We are recording at about 44Kilobytes per second. On the host computer, there is a 2Kilobyte buffer, and on the handheld unit, there is a 6Kilobyte buffer. The total latency due to audio buffering then is the total size of the buffers divided by the data rate. This comes out to 187 milliseconds. Theoretically, in order for the system to keep up with itself and remain in "Real-time" operation, the total latency due to network must me less than the latency due to audio buffering: it must take less time to send data over the network than it does to record it or play it back. By this calculation, our theoretical system latency is at most 374 milliseconds, which is 25% off the measured value. We believe this discrepancy may come from other processes running on either end of the system since Windows is made for multitasking, but we have not verified that idea. This delay may also come from the fact that the version of code we were running on both ends were the debug versions of the build, so they had the debugging symbols and extra information that makes the program run inefficiently.

To verify that the handheld unit would meet our audio specifications, we ran a few speaker and microphone frequency response tests. For an accurate test, Mr. Stear generated a sample of pink noise (the inverse of all frequencies) and a sample of white noise (all frequencies). To perform the speaker test, the noise samples were recorded from the handheld unit's speaker using a highly sensitive microphone. The microphone test used a similar technique. Using the speakers on the development computer as the source, the white and pink noise samples were recorded on the handheld unit's microphone. Then, using a Fast Fourier Transform program, the resulting waveforms were analyzed on a third octave scale to determine the range of active frequencies. Those frequencies that fell outside a 10 dB drop were ignored. The resulting frequency responses are given in the table below (see Table 1: Handheld Speaker and Microphone Frequency Response).

TABLE 1:

HANDHELD SPEAKER AND MICROPHONE FREQUENCY RESPONSE

	Pink Noise $V_o(f)=1/f$	White Noise $V_o(f)=f$
PDA Speaker	20Hz—16KHz	25Hz-16KHz
PDA Microphone	16Hz—16KHz	16Hz-16KHz

3.7) Instruction Manual: Our instruction manual will be based off our operational specifications and any usability guidelines documented in our *technical notes* sheets. The instruction manual will consist of two main sections: one for setting up the device on a wireless network and one for using the SRU and host software to record, play, and manage audio. The wireless network section will be written specifically for a trained IT professional, as this may be too complicated for easy installation by a doctor, especially for setting up the device for use over an entire WAN. It will describe how to set up the wireless communications between either a host and SRU, or a WAN and a SRU.

The usage manual will detail how to play, record, change tracks, and use the host, on a level understandable to someone who has never used a modern stereo. Additionally, we will write a section to prime users of previous USB units on the new SRU/host combination.

4) Release Plans

The release of our product will be a package that includes a PDA with remote software preinstalled; a disk with host software, remote image and remote image installation program; and a set of documentation, either on disk or in paper format. For use in hospitals, the IT department will likely perform the installation and address management setup. In the case of a private consumer, the

user will likely be the installer, so the documentation needs to be thorough enough to direct a person without formal computer training. Our system is not overly complicated, so the initial setup should be rather easy for any skill level.

To invite feedback on the new system, we could setup an email address or phone number to collect opinions and feedback from users of the system. Using the feedback we can develop upgrades or patches to improve and expand the operation of the system.

5) Maintenance

Once the Remote Recording System is released for distribution, the maintenance process will begin. As with any commercial software product, a large part of the maintenance process will be fixing any software bugs that were not found before product release. These bugs could arise from compatibility issues with certain host systems and OS versions that were not tested, incompatibility with other software that may be running simultaneously on some host systems, host OS updates that break compatibility with some features used in the RRS, or simply bugs in the release software that were not found during testing.

Product updates will also be a key part of the product maintenance. Any new technology or functionality added the Windows CE platform in the future that would be beneficial to the product can be integrated into the RRS. Some key technologies that may be integrated in the future would be newer wireless networking standards that allow for higher fidelity or longer operating range of the system, newer compression algorithms better suited to the system, or speech-to-text conversion directly on the remote unit.

6) End of Life

The end of life conditions for the Remote Recording System will occur when maintenance of the software becomes expensive or impossible. The main component of this RRS project is the software, which can be reused on other architectures with Windows and network support. Also, writing for DirectShow dictates that we are writing for general audio and network functions rather than specific sound or network hardware. This makes figuring out when this project's lifecycle will end difficult to say. The table in the next column gives estimates of end-of-support for the key technologies in our project solution (see Table 1: End-of-Life Estimates for Hardware and Software Technologies).

6.1) Hardware: From a hardware perspective, on the original PPT 8846 XScale architecture, a life cycle can be guessed based on how long previous generations of technology have lasted. If we consider the project's lifecycle to be tied to this architecture, then it would be the factor that determines the project lifespan of the RRS.

However, writing for Windows CE dictates that we are writing software for an operating system rather than an underlying architecture. So although AID will likely market actual screen-less devices with the SRU software on them, the true end-of-life of the RRS will occur when they cannot easily put the remote software on a standard newer Windows CE device or update the software for newer versions of Windows.

6.2) Software: The life of the system is also heavily tied to the software and support availability of Windows. We should consider the remote software's end of life as tied to the Windows CE .NET 4.2 platform, as we should consider the host software's end of life as tied to Windows XP. Microsoft has strict end-of-life policies on its software and operating systems. Below is a table of their product lifecycle support dates. The most important of these is probably the end of life date for the eMbedded C++ 4.0 IDE used to build this suite. We predict using a newer Microsoft IDE will fail to build the software using the same process we documented, and require copious amounts of tedious testing and debugging.

TABLE 2
END-OF-LIFE ESTIMATES FOR HARDWARE AND SOFTWARE TECHNOLOGIES[1]

Product	Description	End Support Date
Microsoft Windows CE .NET 4.2	Remote OS with DirectX Development Kit	30 June 2008
Microsoft Windows XP	Host Operating System	31 December 2006
Microsoft eMbedded C++	Remote Development Environment	30 June 2007
Platform Builder	Builds WinCE OS Images	30 June 2007
Intel Xscale Architecture	Remote Hardware platform	Estimated 2-3 years

B. Failure Rate Calculations

The following scores are the product of using the Relex program to compute hardware reliability scores. The use of Relex is outlined in the appendix section titled "Using Relex to Calculate Hardware Reliability Scores." Since we used a demonstration version of the program, there is a limit to the number of parts that could be added into a system. As a result, the host computer system is built of three separate systems: the power supply; the mainboard, CPU, RAM, I/O controllers, and chipsets; and the overall PC with video, sound, modem, and network cards, as well as input devices and a display. In a tree view, the PC system is a root, and the power supply and mainboard subsystems are the leaves.

The table below outlines the device tested, the Failure Rate of the device in failures per million hours, and the Mean Time Between Failure in hours (see Table 2: Reliability Scores for Hardware Systems).

TABLE 3:
RELIABILITY SCORES FOR HARDWARE SYSTEMS

Part	Failure Rate	MTBF (hours)
Host Computer	56.367	17,741
Power Supply	20.992	47,637
Mainboard/CPU	14.738	67,851
Handheld Unit	0.109	9,096,360

C. Failure Modes and Effect Analysis

1) Potential Failure Modes

Together, we brainstormed what we thought were potential failure modes, however likely or unlikely they might be. The list is below (see Table 3: List of Potential Failure Modes).

TABLE 4:
LIST OF POTENTIAL FAILURE MODES

1) Unit out of network range or obstruction
2) Low battery life
3) Physical damage
4) Other systems down (WAP)
5) HD full on host
6) Microphone malfunction
7) DirectShow or library failure
8) Host OS not working
9) DLL doesn't run at startup
10) No soundcard or network card detected
11) DirectX not installed
12) Other DirectX applications using resources, libraries occupied
13) Memory leak or memory full
14) Tracks unavailable (deleted or pointer corrupted)
15) Host OS not working

2) Assign a Severity rating for each effect

Here, we decided on a number to scale how severe the specified event would be if it were to occur. A higher number on an event means such an event occurring would have more severe or detrimental effects.

3) Assign an Occurrence rating for each failure mode

The following list outlines the occurrence rating we assigned to each failure mode, based on how often the failure mode would be likely to occur. In this case, higher numbers mean that the event is more likely to occur.

4) Assign a Detection rating for each failure mode

Here, we assigned a number to each failure mode to describe how easily a given failure mode would be to detect, or how obvious it would be that that event occurred. The higher the number, the more detectable the failure mode.

5) Calculate the Risk Priority Number for each effect

In the table below (Table 4: FMEA Table of Ratings), we calculated the initial Risk Priority Numbers. The RPN in the right-hand column gives a scale for judging which risks are the highest priority, meaning that they are the most severe, most likely to occur, and most detectable failure modes we brainstormed for the system. The failure modes with the highest RPN values are the most important failure modes, and need to be addressed with some form of safeguarding, or even eliminated.

6) Prioritize Failure modes for action.

Our top three failure modes are: microphone malfunction, low battery life, and network devices out of range. Obviously we can do nothing to safeguard the operating system (without building a custom version of the operating system and fixing the bugs we might encounter), so there is nothing we can do to reduce or eliminate the risk of the Host Operating System failing. Also, there is little that can be done to reduce or eliminate the risk of physical damage to the remote device. The remote unit that we have been using for the development of this project is made to be very rugged, so it will likely withstand a short fall once in a while. An action we could take to help make the device more rugged would be to make or purchase a padded case for carrying the remote unit, but that might only protect it while it is not being used.

TABLE 5
FMEA TABLE OF RATINGS

	Severity	Occurrence	Detection	RPN
Phys. Damage	7	4	9	252
Host OS failure	5	3	10	150
Mic Malfunction	3	2	7	42
Low Battery Life	5	8	1	40
Out of Range	5	6	1	30
Library failure	5	2	2	20
External failure	5	2	1	10
No soundcard	5	2	1	10
Memory fail	5	2	1	10
Auto Start fail	5	1	1	5
DirectX absent	5	1	1	5
Tracks unavailable	5	1	1	5
HD full	4	1	1	4
Resources in Use	3	1	1	3

V. SOCIAL IMPACT

A. Health and Safety Issues

As this speech recognition unit will be used in hospitals as a transcription tool, there is the chance that it could be a vehicle for the spread of infection. Recently

cellular phones were found to spread dangerous strains of bacteria in hospitals [2]. Apart from good hygiene practices, our documentation will clearly outline effective methods of sterilizing the wireless unit.

Cellular phones that operate in the 900MHz band as well as other wireless devices have been known to disrupt pacemakers, defibrillators, and hearing aids [3]. However, higher frequency PCS phones operate on much lower power than their predecessors and have an insignificant effect on medical and life critical devices [4]. Devices that use 802.11b radios and technology are similar to PCS phones in this regard.

With any portable audio device, there is an issue of possible hearing loss when using headphones, particularly bud earphones. The simplest option for mitigating the problem of hearing loss with headphones is to set a safe upper limit on headphone volume. Many portable music devices employ an AVLS, or Automatic Volume Limiting System, technology to protect the listener's hearing. We could try to develop some method of AVLS in hardware, or simply set a reasonable limit on the volume dial to which the user will have access.

B. Environmental Issues

Two environmental factors are involved in the use of this device: speaker volume and radio interference. The external speaker should be limited to a reasonable volume that won't disrupt a moderately loud conversation.

Interference is another important consideration; the interference this device causes is discussed in relation to FCC rules in the Regulatory Issues subsection. The remote unit is also subject to environmental interference. Although the orientation of the internal antenna is primarily responsible for how well signals are received, factors such as weather, interference, walls, and other physical obstructions degrade wireless performance.

Furthermore, there is the issue of disposing or reusing components of the device. Lithium-Ion batteries can be recharged or recycled, and are safer to dispose of than other common battery types such as Nickel-Cadmium. Services exist to recycle and refurbish entire cell phones and PDAs as well [5].

C. Legal Issues

Since we developed our project using academically licensed software, we cannot release the applications for commercial use. In the cost analysis however, we account for the cost of the development tools such that a person could develop the project further and for commercial use.

VI. Specifications Not Yet Met

The specifications we did not meet in our implementation are listed below.

- Software command interface to control IBM's ViaVoice voice recognition software.
- Track management interface.

Given more time, and had we initially selected the method of solution we finally used, these specifications would likely be functional today. As it is, these may be somewhat simple to implement for another team of programmers.

VII. Future Modifications

Given more time for this project, this list of additional features would likely have been completed.

A. Host

- Support for multiple simultaneous remote hosting.
- Support for WLAN in addition to *ad hoc*.
- Support for WEP keys.
- Create custom docking software to configure remotes and their addresses.
- Advanced interface for controlling Voice Recognition programs other than ViaVoice.

B. Remote

- Change to a headless device implementation.
- Support for WLAN in addition to *ad hoc*.
- Support for WEP keys.
- Manufacture a new plastic shell and rubber buttons for the playback and record controls, as well as the volume adjustment.
- Add support for other handheld computing platforms (Axim, iPaq, Jornado, palmOS, etc.) that are equipped with speakers and microphones.
- Include volume controls for the recording and playback levels.
- Add voice command capabilities.
- Add headphone and microphone jacks for a headset.
- Add support for sound effects filters (reverb, Darth Vader, etc.).

VIII. APPENDICES

A. USER'S MANUAL 3

B. SPECIFICATIONS 7

C. ORGANIZATION OF CODE AND DOCUMENTS ON ACCOMPANYING CD 8

D. BILL OF MATERIALS 9

E. WINDOWS DEVELOPMENT AND API TUTORIALS 10

F. TECHNICAL NOTES SHEETS / BUILD INSTRUCTIONS

G. REFERENCES

A. USER'S MANUAL

Introduction

Congratulations on purchasing the Remote Recording System. This system is one the most advanced dictation system ever constructed. The features have been designed for ease of use and mobility to make for responsiveness, an easy to use interface, and portability. We hope you find the system powerful and useful.

About this guide

This guide describes how to install and use the Remote Recording System. The topics described herein include:

- Host software installation
- Host setup
- Remote Management Application configuration
- Recording tracks for dictation
- Playback
- Skipping tracks
- Troubleshooting

Installation

Host Software

System Requirements

The host software has been tested for use on IBM-compatible computers running Windows XP Professional. Follow these recommendations for your system:

Pentium III or Athlon 750 MHz
256 MB RAM
DirectX 9.0b Runtimes
Wireless Network Interface Card
Windows Compatible Sound Card
30 MB Free Hard Disk Space
Extra Hard Disk Space for Archival of Sound Files
Voice Recognition Program (For using Voice Recognition capabilities)

Step by Step Installation

1. Place the Installation CD into the CD-drive on your host computer. In the dialog box that appears, choose to install the Host Application.
2. If the installation does not start automatically, click "Start," "Run…," D:\host_setup.exe, where D:\ is the location of your CD-drive.
3. Choose a location on your computer's hard drive for the program files. After you have chosen a location, click "Next" to continue installing the host application.
4. Wait for the files to copy to your computer.
5. Click "Finish" once the files are copied.
6. Shut down your computer, and begin installing the cradle according to the Quick Start Guide supplied with your Remote Unit.
7. After the cradle is connected, restart your computer.
8. If it is not already in your computer's CD-drive, insert the Installation CD.
9. In the dialog box that appears, choose to install the ActiveSync remote software synchronization program. Follow the on screen directions to

install it. When you are finished installing ActiveSync, your host computer is ready to be configured for remote units.

Remote software

Before Using the Remote

Before using the remote, be sure to charge the batteries for the remote for the amount of time specified by the Quick Start Guide supplied with your remote unit. This may take up to 24 hours depending on the remote used for your system.

System Includes

The remote system should come with the proper operating system and remote unit software preinstalled. If it seems that the device does not have the proper software installed, see the Troubleshooting section of this guide. It is possible, though unlikely, that you will need to reinstall the remote unit software on the remote unit.

Step By Step Installation

To install the software on the remote device, first install the cradle onto the host computer, as described in the Quick Start Guide for your remote unit. The remote software is contained on the Installation CD. Installing it on the remote involves simply copying the executable file from the CD onto the remote unit's flash memory, then copying a shortcut to the device's startup folder.

Copying the Executable.

1. Insert the Installation CD into your host computer's CD-drive. Close the dialog box that appears for host software installation. Open the CD by double clicking on "My Computer." Right-click on the icon for your CD-drive, and choose "Open."
2. Be sure the remote unit is properly connected to the cradle and that ActiveSync is connected to the remote unit so that it can synchronize.
3. Click the "Explore" button on the toolbar at the top of the ActiveSync window.
4. A file browser will appear that shows the files on your remote unit.
5. In the file browser for your CD-drive, click on the file named "remoteapp.exe" and then choose "Copy" from the "Edit" menu.
6. In the file browser for your remote unit, double click on "My Computer" and "Paste" the file in that folder using the "Paste" command on the "Edit" menu.

Creating a Startup Shortcut

1. Right-click on the icon for "remoteapp.exe" and choose "Create Shortcut." Now, cut this shortcut using the "Cut" command on the "Edit" menu.
2. On the remote device, navigate to the "Windows" folder, then to the "Startup" folder, and "Paste" the shortcut using the command on the "Edit" menu.
3. Close the ActiveSync program.

Initial Setup

Configuring the Host for Remote Devices

On the Host computer, run the Host's Remote Management Application. The list of remote units should be empty at first, since no remotes have been configured yet. Click the "Add Remote" button next to the Remote Unit List drop-down. Give your remote unit a clever nickname, so that you can keep it straight from any other remote units (for example, "Rodger," "Kitten," "Shane," or anything easy to remember). Also enter the MAC address for your remote. The MAC address is usually on a sticker located on the back of the remote.

Hosting Multiple Remotes

It is possible to host multiple remote units from the same host computer. Using the Host's Remote Management Application, you can easily choose the remote unit you want to host from the drop-down list. Note that you can only host one remote at a time.

Using the System

Dictation

To use the dictation feature, simply press the "Rec" button once on the unit. When the Record indicator light stops flashing and stays on steady, the system is ready to receive audio data and transmit it to the host computer for processing. To stop recording, press the stop button on the remote. On the host computer, you should be able to see the effects of your recording, either in the voice recognition program, or in the audio application you are using, depending upon your situation.

By pressing record, you started recording a new audio track, and by pressing stop, you signal the end of the track. The number of tracks you can record and store at a time is controlled by the Host's Remote Management Application. The default number of tracks is set to ten, but this number can be changed easily by opening the settings in the Remote Management Application. The only limit on the size of audio data that can be stored is the size of your free hard disk space. Please note that the size of tracks you record increases with the amount of time stored in the track.

If your host computer is out of hard disk space to take dictation, you may be unable to record new audio data or translate voice into text. In this case, you will need to clear some disk space. Refer to your computer's help system for information on freeing some disk space.

Playback

To playback what you recorded, simply press the "Play" button once on the remote. You should hear the audio you just recorded play back over the speaker on the remote unit. In the case of a voice recognition program, you should hear the voice recognition program reading back the text you spoke into the remote.

If you have not yet recorded audio with the remote unit, you will not be able to playback any audio. This is also the case if you restarted your computer or closed the audio or voice recognition program you were using to record audio data.

The playback feature always plays back the last track you recorded, unless you have used the track skipping buttons to navigate among tracks you recorded. Note that if you record another track after skipping among the tracks, the playback feature will again play back the last track you recorded.

Skipping Tracks

To skip between tracks you have recorded, press the "Next" or "Prev" buttons on the remote unit. Each press will skip one track in the specified direction. If you have not recorded any tracks, then the skip buttons will not have any effect on the system.

The track skipping buttons work in a circular fashion. For example, if you recorded five tracks of audio data, and you try to skip backward past the first track in the list, you will effectively return to the top of the list and hear track five again. If you try to skip forward beyond the number of tracks you have recorded, you will arrive back at the first track in the list.

Troubleshooting

If you have problems with the system, try to troubleshoot the problems with the following table. Steps you take here will make it much easier to fix the system if you have to call Support for help

Symptoms	Possible Causes	Solution
Device will not record.	Dead Battery.	Charge the batteries according to the Quick Start Guide supplied with the remote.
	Host has no free disk space.	Free some disk space on the host computer.
	Host and Remote unable to communicate over network.	Ensure the host's wireless card is properly installed and configured, and the remote and host are within range of each other.
	Host's Remote Management Application not running.	Start the Remote Management Application, and ensure it is in the "Startup" folder, in "Programs" on the Start menu.
	Remote is not configured to communicate with Host.	Use the Host's Remote Management Application to add your Remote to the list of known Remotes.
	Remote unit software not installed properly.	Follow the directions in step ii. 3. to install the remote software on the remote unit.
Device will not playback.	No tracks have been recorded.	Record a track to playback.
	Dead Battery.	Charge the batteries according to the Quick Start Guide supplied with the remote.
	Host and Remote unable to communicate over network.	Ensure the host's wireless card is properly installed and configured.
	Host's Remote Management Application not running.	Start the Remote Management Application, and ensure it is in the "Startup" folder, in "Programs" on the Start menu.
	Remote is not configured to communicate with Host.	Use the Host's Remote Management Application to add your Remote to the list of known Remotes.

B. SPECIFICATIONS

1. Audio Characteristics
    a. Sample rate: 44100 Hz
    b. Bits per sample: 16
    c. Number of Channels: 1 (monaural)
    d. Compression: MPEG Layer 1 audio compression
2. Hardware Specifications
    a. Internal Microphone:
        i. Sensitivity: 65dB (0dB = 1V/0.1Pa @ 1kHz)
        ii. Frequency Response: 100Hz - 15kHz
    b. Internal Speaker:
        i. Sensitivity: 88dB / 1 W
        ii. Frequency Response: 600Hz - 10kHz 10dB
3. Remote Device Specifications:
    a. Processor: Intel XScale (ARM) Processor
    b. Operating System: Microsoft Windows CE .NET (Version 4.2)
    c. Device Size: 7" * 5" * 2"
    d. Weight: 2lbs
    e. Battery Life: 3 hrs active use, 12 hrs standby use
    f. Battery: 4.3V Li-Ion
4. Network Specifications:
    a. Wireless Protocol: 802.11b
    b. Frequency Band: 2.401 - 2.473GHz
    c. Reception Distance: 100ft.
    d. Network Buffering Time: 5s
5. Host PC Operating System: Windows NT Variant (XP, 2000 or newer)

C. Organization of Code and Documents on Accompanying CD

The disc is divided into a tree to separate documents, code for the remote device, and code for the host.

	Folder	Filename	File Format
\	[root]	Readme.txt	ASCII text file
	Documentation		
		Final Document.doc	Microsoft Word Document (Word 2000)
		Presentation.ppt	Microsoft PowerPoint
	Host\Sockets	TCPConnect.sln	Visual Studio Solution file
	Host\DirectSound	NetTalker.sln	Visual Studio Solution file
	Remote\WaveForm API	DialogApp.vcw	eMbedded Visual C++ 4.0 workspace file
	Remote\Sockets	TCPConnect.vcw	eMbedded Visual C++ 4.0 workspace file
	Remote\DirectShow	DialogApp.vcw	eMbedded Visual C++ 4.0 workspace file
	Remote\DirectSound	DialogApp.vcw	eMbedded Visual C++ 4.0 workspace file
	WavSamples	White.wav	Generated White noise waveform audio
	WavSamples	Pink.wav	Generated Pink noise waveform audio
	WavSamples	Whitespkr.wav	PDA Speaker White noise waveform audio
	WavSamples	Pinkspkr.wav	PDA Speaker Pink noise waveform audio
	WavSamples	Whitemic.wav	PDA mic White noise waveform audio
	WavSamples	Pinkmic.wav	PDA mic Pink noise waveform audio

Each directory for host or remote contains the source and the environment setup files for each of the methods we had coded to find a solution. The source for each project is handled by the project files listed in the table above. For each platform, the "WaveForm API" folder holds the solution that we finally developed for presentation.

The WavSamples folder on the root contains audio samples that were recorded or generated for frequency response testing. The Generated sounds were created using NCH tone generator, available for free download [6]. The "PDA spkr" samples were the result of playing the generated sound over the PDA's spkr, and the "PDA mic" samples were recorded on the PDA using the PDA's microphone.

D. Bill Of Materials

This project required few materials for solution. They are listed below, along with a table that details the total costs of this project, if the developer were to start from scratch and buy each component.

1. Microsoft Visual Studio .NET 2003
2. Microsoft Windows CE Platform Development Kit (includes eMbedded Visual C++ 4.0).
3. Microsoft eMbedded Visual C++ 4.0 SP3.
4. Microsoft DirectX Software Developer's Kit
5. Development computer:
    a. Windows XP Professional edition.
    b. Pentium 4 or Athlon processor
    c. 256 MB RAM
    d. 3000 MB free hard disk space for Visual Studio .NET 2003 and documentation
    e. 300 MB free hard disk space for Windows CE Platform Development Kit and service packs
    f. 100 MB free hard disk space for project files and builds
    g. 500 MB free hard disk space for DirectX SDK
    h. Wireless network card, capable of transmitting over 802.11b Wi-Fi standard.
6. Windows CE 4.2 .NET based handheld computer. For our project, we used the Symbol PPT8846 handheld.

	Description	Qty	Unit Cost	Extended Price
1.	Visual Studio	1	$1,079.00	$1,079.00
2.	WinCE Platform Kit	1	$995.00	$995.00
3.	eVC4 SP3	1	$0.00	$0.00
4.	DirectX SDK	1	$0.00	$0.00
5.	Development computer*	1	~$2000.00	~$2000.00
6.	Symbol PPT8846**	1	~$2000.00	~$2000.00
	Total Cost			~$6074.00

* We supplied our own personal computer to develop this project. For a developer just starting out, this would be an anticipated cost for a development computer.

** Our sponsor requested that we use this handheld computer, and they supplied it and covered the cost. In theory, this project could be developed on any handheld computer that is based on the same ARM architecture, and that includes an audio device (speaker and microphone).

E.1 WINDOWS DEVELOPMENT ENVIRONMENT DOCUMENTATION

E.1.1 Introduction

If you have never developed a Windows program before, buckle up and take notes! Windows development is a completely different experience from the Unix terminal-editor-compiler experience that lower-level college programming courses at colleges such as the University of Idaho teach. One must know object oriented programming quite well before they can use Microsoft's class-based tools effectively.

This section describes application development for Windows XP and Windows CE .NET using the Visual Studio .NET and eMbedded C++ 4.0 development environments, along with the entire Windows CE SDK and multiple versions of the DirectX SDK. This is a very broad overview of the processes used to develop this software. The technical notes section of the appendix contains how-to information for using specific Visual Studio and eMbedded C++ functionality.

All the software we used was obtained either from the University of Idaho CS department for educational use or from the Internet. Commercial and educational licenses are available for Visual Studio .NET. eMbedded C++ is a free download from the Microsoft Windows Embedded web site [source]. DirectX SDKs are available from Microsoft's DirectX web site for free as well [source].

The Windows CE SDK can either be obtained from Microsoft or a custom version can be created using their Platform Builder tool. Vendors often supply their own versions of the Windows CE SDK. For development with a Windows CE emulator we were required to build a custom Windows CE SDK with DirectX support. On the actual Symbol PPT 8846 device, we used the PPT88xx SDK available from the Symbol Developer Zone at http://devzone.symbol.com.

E.1.2 The DirectX Software Development Kit (SDK)

DirectX is Microsoft's multimedia platform that serves as both a toolkit and a hardware abstraction layer between a program and a video or sound devices. All modern versions of Windows after Windows 95 Service Pack 1 (SP1) include some form of DirectX runtime DLL. In order to develop DirectX applications, the entire DirectX software development kit is necessary in order to obtain the correct static compile-time libraries.

Platform	**Windows Version**	**SDK Used**	**DirectX Version**
Symbol PPT 8846	CE .NET 4.1	Symbol PPT 88xx	8.0 for CE
CE Emulator	CE .NET 4.2	Custom (SRU-R)	8.1 for CE
Host PC	XP	DirectX 9 SDK	9.0b full

Table E.1 DirectX versions used on each platform and where they came from

The above table shows the version of DirectX that corresponds with the OS on each hardware platform. On the host end, note that the DirectX 8 runtime (DLL) is installed with all versions of Windows XP, but DirectX 9 is the newest version as of this writing. For software development on the host, a version of the DirectX SDK was required. We decided to use the newer DirectX 9 SDK as it included DirectShow network sample code that was useful for us. Newer versions of DirectX are backwards compatible with the older versions, though, and we were careful to use methods and objects common to both DirectX 8 and 9 runtimes so our application would be backwards compatible.

On the remote side, DirectX is included with the various versions of the Windows CE SDK. These versions lack some of the features of the desktop versions, and are listed accordingly in the table.

E.1.3 Host Development: Microsoft Visual C++ .NET

Visual Studio .NET is Microsoft's unified development environment for Windows software, supporting a myriad of compilers and tools under a single Integrated Development Environment (IDE). Visual C++ .NET is designed to work with object-oriented program design, and emphasizes looking at code not by files but by classes with its class selection menus and wizards. It is the only reasonable way to develop using Windows APIs.

1.3.1 Projects, Solutions, and Dependencies

In Visual Studio .NET terminology, a *project* is a .vcproj file containing a group of headers and source code files that creates a single static library, DLL, or executable. A *solution* is a .sln file that contains multiple projects. A solution and its component projects can be copied together between computers. To do so, you must copy all files referenced by the solution and keep all the relative pathnames between the solution, the project, and the source files intact.

Solutions are used because often times a project requires a library to be built by another project before it can compile. In this case you have a list of dependencies between your projects. Not only can developers specify which projects depend on one another, but they can also change the build order. Both these options are available when you right-click the solution name on the class or file view in the IDE.

1.3.2 Project Wizards and Windows GUI Projects

Like with Microsoft Office, in Visual Studio a *wizard* is a way to create an empty Windows interface that your program can be built off of. There are as many types of application wizards as there are Windows user interface libraries, but the three primary types are *document*, *dialog*, and *console*.

Document applications have a menu bar at the top and usually a user either creates or modifies files within the interface. They often call smaller *dialog* windows to set options – these dialogs can be full-featured applications in themselves. The RRS remote application is simply a dialog with buttons for playback, recording, and so on.

Console applications run within the Windows command prompt and resemble the Unix terminal applications of old. They can use `printf()` and `scanf()` calls. For those experienced with Unix programming as we were, a console app was the simplest way to test the Windows APIs we were learning.

1.3.3 Finding Variable/Function/Class/Method Declarations

Part of the power of Visual Studio is that developers are freed from their documentation and can rapidly find class and variable definitions. The class-list drop-down and context menus help to navigate through objects and their methods rapidly. Additionally, by placing your cursor on any function or method name and pressing F12, you will be taken to where that variable or function is defined, even if that definition is contained in a different file from the one you're viewing.

When you type in a class name, followed by two colons, both Visual Studio and eMbedded C++ will show a scrollable list of methods and variables for that class. This *auto-completion* list is very useful to help find variables that your program might need, such as a *window handle* for sending the window's memory address to other functions.

Another handy tip is to switch between the IDE's child editor windows by pressing CTRL+TAB or CTRL+SHIFT+TAB. This can be used in most document-style applications that handle more than one open document.

1.3.4 Static and Dynamic Libraries with Windows XP

Windows programs often work with both .lib format *static libraries* and .dll format runtime, or *dynamically linked libraries* (DLLs). When building a DLL project, it will build a DLL and an "exports" .lib file. The idea, without going into extreme detail, is that without the .lib file, the application won't know what the DLL file contains. Some programs solely use static libraries and build the final application with those, but

this results in very large executables. DLLs also have the handy property of being reusable by more than one program.

When working with Application Programming Interfaces such as Windows Sockets or DirectShow, you need to go to your Linker Input settings and add the required .lib files, as per the API documentation. For instance, when building sockets applications for XP, you need to add ws2_32.lib to the list of input libraries to include, in addition to including the winsock2.h header file in your source code.

If your solution builds an application that relies on a DLL that is also built by your solution, the application needs the output .lib file made by the DLL project, in addition to having the created DLL *registered* by the system. DLLs are registered using the regsvr32 utility included with Windows XP. In the Project Settings for a project, there is an option under Linker settings that will register your DLL automatically after it is successfully built.

Finally, before you start playing with the Linker input list and start including exact pathnames to each and every library file, note that Visual C++ .NET has directory lists it searches when it looks for everything from include files to libraries and binaries. Go to the Tools menu, click Options, and click the Directories tab. Sometimes library errors arise because the library search directories aren't in the correct order and you're obtaining the wrong version of a particular library.

To reiterate: Windows libraries in four steps:

- Include the proper header files
- Find where the libraries you need are located and add those paths to the directory list
- Add the required static libraries to the Linker input list
- Make sure DLLs are registered

1.3.5 Step-Through Debugging

The real strength of any Microsoft IDE is its debugger, which allows the developer to step through the code line-by-line as it is executing. In addition, there are tabs for watching local and user-specified variables as the program is running, eliminating the need for numerous `printf()` or `cout` debugging statements. *Breakpoints* are used to halt execution at specific points (F9), after which execution can be continued (F5) or code can be stepped through one line at a time (F10). F11 will step into the currently highlighted function.

Breakpoints may halt execution if a variable has a specific value, or alternatively every *n* times it is reached. Right-click a breakpoint to set its options. This is extremely useful for debugging threads or iterative algorithms that repeat many times.

1.3.6 Helper Tools

Two tools necessary for our Windows development with Visual C++ .NET are the Error Lookup tool and the Registry editor.

The Error Lookup tool, accessible from the Tools menu, is used to correlate hex values with error messages. When debugging an application, many API calls will return zero if successful, or one of potentially thousands of error messages if not. In the debugger, by copying the value from the variable watch tab into the Error Lookup tool, you can check what that error value means.

For debugging DLL files, it is necessary sometimes to check the registry to see if they were registered correctly with the operating system. Each DLL has Globally Unique Identifiers and Class Identifiers in their source code that will find their way into the Windows Registry if they were registered properly. Usually they show up under the `HKEY_CLASSES_ROOT\CLSID` key.

E.1.4 Remote Development: Microsoft eMbedded C++ 4.0

eMbedded C++ 4.0 is the standard IDE for developing C++ applications for Windows CE platforms. It would be nice if it shared all the same features as Visual Studio .NET but in reality eMbedded C++ is a offshoot of the Visual C++ 6.0 line, while Visual C++ .NET is another name for Visual C++ 7.0. The two IDEs have very different interfaces and feature sets, in particular because eMbedded C++ 4.0 has to worry about compiling programs for each architecture Windows CE runs on, including an emulator platform that runs on the host machine. That said, most of the function keys for debugging and searching (F5, F10, etc) are the same.

1.4.1 Projects and Workspaces

Visual C++ .NET projects are incompatible with eMbedded C++ projects and vice versa. Developers porting to Windows CE must create a new eMbedded C++ project and add all the files from the C++ .NET project by hand. Projects in eMbedded C++ follow the same rules and have the same properties as those in C++ .NET. Workspaces serve much the same function as solutions do in C++ .NET, grouping projects together, handling dependencies, and building projects in developer-defined orders.

1.4.2 Platforms

For all versions of the Windows CE SDK installed (PPT 88xx, custom, STANDARD_SDK to name a few) there will appear a platform *profile*. In addition, each profile will let the developer build programs for a specific hardware *architecture*, such as the ARMVI, MIPS, or WinCE Emulator. Finally, the developer can choose to send this code to any *target* Windows CE devices attached to the development machine. Usually this is just a single PDA device and the Emulator. If the target is incompatible with the profile and architecture combination, a warning message typically appears. Some combinations will allow the program to be sent, but then any attempt to run the program will fail because the binary format is incorrect for that architecture.

1.4.3 Configuring the Target Device using Platform Manager

All the Windows CE development tools use the Windows CE Platform Manager to configure the target devices. The Emulator comes pre-configured, but other CE device profiles might not work off the bat due to problems with serial or wireless communication. To access the platform manager, select the Tools menu and then select 'Configure Platform Manager'. Then select the profile you wish to modify – for the remote device, choose "PPT 800 Device".

In particular we had trouble communicating over the serial port, and the Symbol documentation was of no help whatsoever. The Platform Manager settings that worked for us are as follows:

Transport Server: TCP/IP Transport for Windows CE
Configure Connection over Serial, defaults elsewhere
Startup Server: ActiveSync

The target device can also be configured to use a wireless connection if the device is currently connected to your wireless network. Just put in the device's wireless address in the TCP/IP Transport Server options.

1.4.4 Static and Dynamic Libraries with Windows CE

As with a DLL project in Visual C++ .NET, a DLL project for eMbedded C++ creates a DLL file and a .lib file. Also, as with other Windows development, the .lib stub created by this DLL must be linked into applications that want to call the DLL. Note that .lib files and DLL for the various CE architectures are completely different from the ones compiled for Intel x86 architectures. Libraries used by eMbedded C++ must either be included with the Windows CE SDK or built from source code using the eMbedded C++ compiler.

1.4.5 Debugging, Helper Tools, and ActiveSync

Even if the IDE isn't the same, the eMbedded C++ debugger shares all the same major functionality with its Visual C++ counterpart. There is also an error lookup tool and a registry editor that work exactly like

the XP counterparts. One important difference is that the Windows CE registry editor runs from the development machine and must connect to the remote device using Microsoft ActiveSync.

Using ActiveSync with the PPT 8846 is a chore. It took random fumbling around with the COM port settings in the Windows Device Manager. The final settings that worked for us were as follows:

115200 baud
8 data bits
No Parity
1 Stop Bit
Xon/Xoff flow control
FIFO disabled (this was the main problem… at the port settings tab, click 'Advanced').

E.1.5 Windows CE Platform Builder and Windows CE Emulator

The purpose of the Windows CE platform builder is twofold: it is used to build Windows CE operating system images from a user-selectable catalogue of features. It also builds a specialized SDK for your platform that includes documentation and libraries for precisely the features you include in your platform build. Since the STANDARD_SDK default platform didn't include DirectX libraries, we needed to build a new one for the purpose of testing remote-end code outside of the actual handheld.

Although the Windows CE Emulator has been mentioned before, it is for all intents and purposes its own Windows CE Platform. It runs approximately 80% speed compared to a real device, and has no wireless Ethernet drivers. However, it accepts socket connections, has 802.3 Ethernet functionality, and it loads and debugs compiled code exactly like a real physical platform. Its major disadvantage outside of not being a real device is that building for real platforms requires additional setup work due to building and including different libraries for different architectures; and setting up linker input lists for each architecture can be a chore. In particular, if the linker directories list is in the wrong order you might not build with the right libraries for your architecture.

E.2 WINDOWS API DOCUMENTATION

E.2.1 Introduction

The Windows Application Programming Interfaces (APIs) used in the Remote Recording System (RRS) are just a few out of the many hundreds Microsoft has developed to make Windows development and deployment happen faster and sometimes easier. Although many Windows APIs support languages such as Visual Basic and the new .NET runtime framework, all of the primary Windows APIs are designed with C++ object-oriented development in mind.

This sections provides an overview of functions and class objects for each of the APIs we worked with during the duration of our project. These are designed to supplement the reference material included with the API documentation included with versions of Visual Studio. We assume readers have access to this documentation (also available on the MSDN website), as well as a basic understanding of Windows drivers, processes and threads, C++ class syntax, and related terminology. It will help you to read our Windows Development documentation as well as Table E.1 for the various versions of DirectX employed in the different versions of Windows.

E.2.2 DirectShow

Microsoft DirectShow is a media-transport API that we believed was capable of implementing all audio compression, network transfer and audio playback functionality on both the host side and the remote side. Although most of our development time was spent trying to develop DirectShow sample code into a form suitable for host and remote development, we ran into significant problems with how DirectShow transfers data between its component objects that made rapid development of our RRS system impossible.

Regardless of its absence from our final solution, since we spent most of our time working with DirectShow, we have gathered enough documentation to discuss how to develop DirectShow applications and describe how DirectShow is useful in practice, rather than in theory.

Headers, Static, and Dynamic Libraries (All Platforms):

Headers:

`dshow.h` – Main DirectShow header

`streams.h` – Used for streaming applications

Static Libraries:

`strmiids.lib` – CLSIDs and error messages for DirectShow objects

`strmbase.lib` – Base classes: used to develop filters. For release builds only

`strmbasd.lib` – Same as above, but for debug builds only

Runtime: Installed DirectX 8.0 or later

2.2.1 The Microsoft Common Object Model (COM)

Just about everything in DirectShow is a class object. Microsoft has a standard for designing these objects called the Common Object Model, or COM. The important things for application developers who use these objects to know are how to initialize them and how they're stored in memory. A quick and dirty summary: use `CoInitialize(NULL)` to initialize the COM subsystem DLLs, and then use `CoCreateInstance` to create instances of the objects you need. When you're done, use the object's `Release` method to deallocate object memory and `CoUninitalize(NULL)` to shutdown COM.

Once the DLL is registered with Windows, it creates global values in the Windows Registry so other programs know where to find it. This process will be described once we have described particular DirectShow objects in more detail.

2.2.2 Filters, Pins, and the Filter Graph

DirectShow is based around the idea of software *filters*. These filters are objects that obtain data from hardware, manipulate it in a vast variety of ways, and then write it to any sort of buffer such as a network or audio buffer, or a file. These filters are allocated under a *filter graph* object, and then the filter's *pins* are connected together. Remember that even with the hardware metaphors DirectShow employs, these objects are software code, and the process of using them doesn't necessarily work like it would real hardware.

After the filter graph is allocated and the filters are connected, the main program calls a `Run` method and the filter graph will run in its own thread, creating or collecting audio or video data, as the main program continues doing other things. The main program must use some form of event handling to listen to any messages the graph thread might send out in order to determine when it is done with its task.

2.2.2.a Filter Varieties

There are three main varieties of filter: transform filters, source filters, and sink filters. The transform filter is what one might associate with the actual hardware concept of a signal filter – it takes an input stream and manipulates each sample of that input stream in some way. Two sample transform filters included in the DirectShow SDK are the Gargle filter and the WavDest filter.

The Gargle filter manipulates sound samples to sound gargly, as one would expect. The WavDest filter does no modifications to the stream of bytes it receives, but it does keep track of the samples it obtains so that after all data has made it through the filter, it can append an appropriate wave file header to the beginning of the stream. It doesn't write to a file itself; it must be connected to a separate file writer filter. In this sense it is a transform filter as well.

Source filters are specific for the device drivers on a Windows XP system. A sound card will have an Audio Capture Filter and an Audio Renderer filter, a video capture card will have a Video Capture filter, and so on. Any Windows-supported audio or video hardware can have a source filter created for it by DirectShow. To use these sources in a program you must enumerate the available devices in a particular category, and then grab the device you want and include its filter. Source filters exist for capturing streamed network data as well; these require no device enumeration by the main program, as they enumerate the network card within the filter itself. Furthermore, a source filter could point to an input file, such as the File Renderer does.

Sink filters are the filters that take the input stream and write it to an output stream or buffer. These might be enumerated devices in the same sense as the source filters described previously, or they might be something as simple as a file writer. The DirectShow version of the RRS used a network sender filter that essentially was a 'sink' to the network.

2.2.2.b Pins and Media Types

As described earlier, all DirectShow filters have pins. These pins have methods that are used to connect themselves to other pins, or to other input or output data streams and buffers. Most transform filters have one input pin and one output pin, while sound card source filters have any number of inputs depending on what hardware functionality the device driver supports. If a filter doesn't have enough outputs, it is connected to an Infinite Tee filter which converts a single output for a filter into any number of equivalent output pins.

DirectShow pins are either output pin objects or input pin objects, and they all have a `MEDIA_TYPE`, a `MEDIA_SUBTYPE`, and a `MEDIA_FORMAT` variable. When filters connect to each other, the combination of `TYPE`, `SUBTYPE`, and `FORMAT` must be compatible between the output pin and the input pin. Alternatively, if the graph builder finds proper intermediary filters to add that will convert the output pin to agree with the input pin's type, it will add those in transparently. For local file operations, auto-connection usually makes filter graph creation extremely easy. Unfortunately, the same cannot be said of streaming operations such as those needed for the RRS.

2.2.2.c Data Flow in a Filter Graph

The final point to note about DirectShow pins is that they are either synchronous, deriving from the `IMemInput` class, or asynchronous, deriving from `IAsyncInput`. Most of the Microsoft documentation refers to these as push mode and pull mode, respectively. The difference between the two is that in pull mode the filter graph's progress is determined by the "downstream" sink filter, as it is the one that sends messages back up the graph to the "upstream" source to send data. In push mode, the samples are sent and the filters act synchronously on the data as it is moved down the graph.

2.2.3 A Poor Streaming Platform

Although DirectShow is designed for both synchronous streaming and asynchronous non-streaming applications, Microsoft later designed Windows Media APIs which are centered around streaming their proprietary formats, and as such DirectShow remains an immature platform for developing streaming applications.

The problem is that the filters required to implement streaming and negotiation between IAsync and IMem interfaces are non-existent. For example, it is relatively trivial to construct a graph that takes an input WAV file and converts it to MP3 with the MP3 compressor filter. However, if that WAV data is streaming

right off the Audio Capture filter, the developer cannot use the Microsoft MP3 compressor because the Audio Capture filter operates in push mode while the MP3 compressor can only handle asynchronous reads from a file (pull mode). To further complicate things, it is not readily apparent which data flow methods and pin types a filter employs without jumping headlong into documentation.

Our group might have chosen to develop a sink-to-source "parser" filter that makes a buffer that the downstream portion reads asynchronously. Unfortunately this would have taken more time than it would take to implement the RRS outside of DirectShow due to the large number of methods needed to instantiate the filter memory properly and set up all the component pins.

2.2.4 CLSIDs, GUIDs and the Windows Registry

When building a DirectShow filter for use in other applications, one aims to create a DLL file loaded by Windows that other applications can use. When Windows loads the DLL file that a project builds (remember to check the Register DLL option in your Build properties), it creates a new Class Identifier (CLSID) in the Windows Registry. The Windows Registry is a huge table of values useable by any program the OS is running, and the only way to read or write to it outside of the `regsvr32` application is with the `regedit` tool.

So when a DLL is registered with the operating system, the CLSID is stored in under the `HKEY_CLASSES_ROOT\CLSID` key in `regedit`. The CLSID is a specific type of Globally Unique Identifier (GUID) used by objects that are activated at run-time. Under the GUID key it might give the object name and other properties, but it always lists in what DLL the object is found. So when the main program wants to use that filter object, it instantiates it using the `CoCreateInstance` function (`malloc` for class objects), with the CLSID of that object as one of its arguments.

```
static const GUID CLSID_DSNetReceive = { 0x319f0815, 0xacef,
0x45fe,
{ 0xb4, 0x97, 0xa2, 0xe5, 0xd9, 0xa, 0x69, 0xd7} };

CoInitialize(NULL);
CoCreateInstance(CLSID_DSNetReceive, NULL, CLSCTX_INPROC_SERVER,
      IID_IBaseFilter, (void**)&g_pSend);
```

Fig. E.3 Creating a CLSID_DSNetReceive variable and using it to allocate a filter into memory

How does the host program find the CLSID for the filter so it can use it? The answer is that it doesn't—you hard-code the value into your program after discovering it in the registry or in the filter DLL source. The example above shows how a program stores the CLSID in a variable for use by `CoCreateInstance`.

2.2.5 Creating The Filter Graph and its Filters in Memory

The filters we used in our DirectShow RRS include an Audio Network Receiver and an Audio Network Sender. Both of these were derived from Microsoft sample code included in the Summer 2003 Update of the DirectX 9 SDK, specifically the `DSNetwork` project. They originally used *multicasting* (see the Windows Sockets documentation), but we modified them to send over a simpler peer-to-peer scenario. Here is a step-by-step algorithm in order to do this:

1) Build the DLL project or set the solution's build order such that the DLL is built and registered before your main application project compiles. If this works it will send the appropriate CLSIDs into the registry.
2) Declare your DLL's CLSIDs somewhere in your main program, as was done in Figure E.3.
3) Initialize COM and create an instance of the Filter Graph object itself.

```
CoInitialize(NULL);
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
      IID_IGraphBuilder, (void **)&pFGraph);
```

Fig. E.4 Allocating space for a Graph Builder object in memory

It will make life much easier if you either make the pointer to the graph memory either global or in a readily accessible class. Do the same for all of the filters you create.

4) Next create any filters you need, using the same technique as in step 3. If your filters are not homemade, look up the `CLSID_FilterName` variable for that filter so you can call it with `CoCreateInstance`. Pay attention in the API documentation to what classes this filter derives from, i.e. `IBaseFilter` or `IFileSinkFilter` (the fourth argument of `CoCreateInstance`). This determines what methods are available to your filter, and what methods are available to its pins.

5) Now you'll need to set up each filter's special parameters. Most filters have unique methods that need to be called in order to find the right audio hardware, or to set the input or output file. For the network filters you need to set their destination or source IP address. These methods and parameters are described in the Microsoft DirectShow documentation under 'DirectShow Reference', as well as in our source code.

6) So you've created the filter graph and the filters but there's one step left. If you have created all your filters but have problems later this is a likely cause. Remember to associate these filters in memory with the filter graph object by adding them to the filter graph:

```
pFGraph->AddFilter(g_pSend, "Network Send Filter");
```

The second argument is just the name that your filter is given within this filter graph. If `g_pSend` is not pointing to a created filter in memory this statement will fail.

7) Remember to do error checking any time you allocate memory. In the IDE you can check the return values of `CoCreateInstance`, `AddFilter`, and a host of others by creating a test value and setting it equal to the return value of your function under test. Since these methods return a value of type `HRESULT`, just do something like this:

```
hr = CoCreateInstance(CLSID_FilterGraph, NULL,
CLSCTX_INPROC_SERVER,
      IID_IGraphBuilder, (void **) &pFGraph);
if(FAILED(hr))
      // Return an error
```

Fig. E.5 Error checking with the return value of a memory allocation function

`FAILED` is a macro that simply checks if hr is greater than zero. The convention is that a `HRESULT` of 0 (also defined as `S_OK`) is the non-error condition, while anything else is an error.

Just to reiterate: Build your DLL, initialize COM, create the filter graph and the filters, set up the each filter's unique parameters, add the filters to the graph, and do error checking along the way to make sure you didn't forget anything.

So you've got your filters in a graph. Now what do you do?

2.2.6 Connecting the Filters

You might have noticed we haven't connected any of the filters together yet. No data can flow until we have a source, a sink, and whatever transform filters connected together to form a graph. The basic idea of this process is to enumerate an output pin on an upstream filter, an input pin on a downstream filter, connect them together, and then deallocate the pin variables so we can use them to connect another pair of filters.

Here's a snippet of code that demonstrates the connection algorithm. The filters in this code must already been created in memory or their methods will fail.

```
IEnumPins* PinList;
IPin*      OutPin;
IPin*      InPin;
ULONG fetched;

hr = g_pAudioCapFilter->EnumPins(&PinList);
hr = PinList->Reset();

// Find the output pin on the Audio Capture filter
while(PinList->Next(1, &OutPin, &fetched)==S_OK)
{
      PIN_DIRECTION PinDirThis;
      OutPin->QueryDirection(&PinDirThis);
            if (PinDirThis == PINDIR_OUTPUT)
                  break;
            OutPin->Release();
}
hr = PinList->Release();

// Find input pin on Infinite Tee
hr = g_pInfTee->EnumPins(&PinList);
hr = PinList->Reset();
PinList->Next(1, &InPin, &fetched);
hr = PinList->Release();

// Connect the Audio Capture filter to the Infinite Tee
hr = g_pGraphBuilder->Connect(OutPin, InPin);
InPin->Release();
OutPin->Release();
```

Fig. E.6 Algorithm for Manually Connecting Filters

In order to get the correct pins from a filter you have to enumerate them all and do a quick search for the proper pin by checking the pin's direction as shown in the while loop above. Input pins are always the first pins in the enumerated list, so for the Infinite Tee we just chose the first pin using the `Next` method. After we connect the actual pins on the filter, they will remain connected even after we release the `InPin` and `OutPin` variables we were using.

One line that might confuse you is the `Reset` method used by the `PinList` list. This resets the list pointer to the head of the list just incase somehow `PinList` wasn't released properly, and is just a safety measure.

The `Connect` method won't work if you're trying to connect a filter that hasn't been added to the filter graph, so you might want to check for errors with that `hr` value in your code. Remember to call `AddFilters` to add the created filters to the graph in your allocation routines!
This process of connecting the filters is called *manual connection*. DirectShow has many methods for automatically connecting filters together when it's working with simpler tasks like pull-mode audio compression or local playback operations. However, these could not be used for our project because of our desire to use both push mode and pull mode filters. Look at the `RenderFile` method, which creates a playback graph specifically tailored for its file argument, for more information.

2.2.7 Running the Graph

In terms of the main program's logic the most important thing a programmer needs to plan on is *event handling*. As soon as the filters and graph are created and connected, you grab an `IMediaControl` pointer from the filter graph object and use its methods to `Run` and `Stop` the graph.

```
IMediaControl* g_pMediaControl;

hr = g_pGraphBuilder->QueryInterface(IID_IMediaControl,
     reinterpret_cast<void**>(&g_pMediaControl));
hr = g_pMediaControl->Run();
```

Fig. E.7 Obtaining the controller for the filter graph and using it to run the graph

`QueryInterface` is a very important method used in many DirectShow objects. It is used to allocate memory for a set of class methods used to operate on the calling object. Here we create a `IMediaControl` object named `g_pMediaControl` to control the `g_pGraphBuilder` filter graph object. As the variable name might suggest, it is useful to put the `IMediaControl` pointer into protected class memory or global memory so you don't have to worry about passing it during calls to the event handling function.

Up to this point if you've made a program that implemented the filter creation and filter connection techniques, created the media controller and called `Run`, you would have a working graph. It will simply run until the stream ends. Chances are an application will wish to have the option to stop this stream before it is over, however – this will require the main application to have an event handling strategy.

2.2.7 Detecting Events

The filter graph runs in its own execution thread, so your program can continue doing whatever else it needs to do, such as print status or diagnostic information, while a graph is running. Being that the filter graph is in its own thread, it is the main program's responsibility to know what that thread is up to. DirectShow has its own `IMediaEvent` interface that allows you to do precisely this. Using `IMediaEvent` methods, you can tell the filter graph to send event messages to the main application. Although very useful for sending events, `IMediaEvent` is a DirectShow-specific object, and cannot be used in event handling for programs with no filter graph.

```
// Constant representing the code for our filter graph message
#define WM_GRAPHNOTIFY WM_APP + 1

IMediaEvent* g_pMediaControl;

hr = pFGraph->QueryInterface(IID_IMediaEvent, (void **)&pMEvent);
hr = pMEvent->SetNotifyWindow((OAHWND)m_hWnd, WM_GRAPHNOTIFY, 0);
```

Fig. E.8 Obtaining the controller for the filter graph and using it to run the graph

Every Windows application has a *window handle*, named `hwnd` or something of that ilk. When building GUI applications such as a simple dialog application or a more typical document application, this variable is contained by one of the parent classes for your application's window painting or display routines. Even the visible windows themselves are described by class objects just like the filter graph.

If you build a dialog application as we did, the window handle variable will be `m_hWnd` but if you created a different type of application, your window handle may have a different name. To find it, type the name of your *application class* (ours was `CDialogappDlg`), followed by two colons, and watch the *auto-completion* come up with all the potential methods and variables for that class. Chances are the variable containing `hwnd` in it will be the one you're looking for.

Unfortunately we're not even close to done making an event handler. Our graph is set to send messages to our main application window, but the main application doesn't know how to deal with these events. The Windows way of handling messages is to set up the `WindowProc` function from one of your application-classes such as `CDialogappDlg`. `WindowProc` will handle any application level messages, and send the rest of them to a lower-level handling routine (also named `WindowProc`, but within a lower-level class).

You must create this `WindowProc` using the *Class Wizard*, selectable from the View menu or by pressing Ctrl+W. Double-click 'WindowProc' from the list of "messages" found in the Class Wizard window, under class name `CDialogappDlg` (replace with your application class name). It will create an empty WindowProc function with the appropriate return call to the lower-level handler. Adding a condition to check for WM_GRAPHNOTIFY messages will allow your application to handle the messages from the filter graph. The final function looks like this:

```
LRESULT CDialogappDlg::WindowProc
(UINT message, WPARAM wParam, LPARAM lParam)
{
   switch(message)
   {
      case WM_GRAPHNOTIFY:
           HandleEvent();
           break;
   }

   return CDialog::WindowProc(message, wParam, lParam);
}
```

Fig. E.9 Our application's WindowProc function

WM_GRAPHNOTIFY only tells us that something has happened with the filter graph, not what that something might be. So it calls a HandleEvent function we make to determine what really happened in our filter graph, and make the appropriate decision. HandleEvent calls IMediaEvent::GetEvent to obtain the filter graph's state. We then take action based on what this state is. All the following HandleEvent function acts upon is EC_COMPLETE, which determines if the entire stream went through the graph, or EC_USERABORT, which would be sent if the IMediaControl::Stop method was called somewhere.

```
HRESULT CDialogappDlg::HandleEvent()
{
   HRESULT hr;
   long evCode, param1, param2;

   hr = pMEvent->GetEvent(&evCode, &param1, &param2, 0);
   while(SUCCEEDED(hr))
   {
      hr = pMEvent->FreeEventParams(evCode, param1, param2);
      if ((evCode == EC_COMPLETE) || (evCode == EC_USERABORT))
      {
         CleanUp();
         break;
      }

      hr = pMEvent->GetEvent(&evCode, &param1, &param2, 0);
   }

   return hr;
}
```

Fig. E.10 Handling filter graph events

For simple event handling like we did, we didn't need to worry about the specifics of what the event parameters param1 and param2 were, other than they needed to be freed after every GetEvent call.

2.2.8 De-allocating Objects

After the graph is done running, and you're done with all your DirectShow objects, call the Release method on each one. The filter connection algorithm shown in Fig. E.6 demonstrates the PinList pointer being released every time it contained a new value. Every time the filter graph needs to do a new playback or recording session, it needs to be Released and recreated/connected in this same manner.

Failure to Release all the objects you create will result in memory leaks and runtime errors after your DirectShow graph is done doing its job.

2.2.9 Concluding Remarks About DirectShow

The hardest parts about using DirectShow are setting up the filter graph correctly and verifying your event handler works. It cannot be stressed enough that a programmer needs to do error checking on the return value of each and every DirectShow method call, especially if they are just starting out. Look at Fig. E.5 or the RRS source code for examples of error checking. If the program doesn't properly connect or create filters, step-through debugging will almost always reveal a bad return value for some DirectShow method.

Error checking makes this much easier to see, because you can watch the value of the `hr` variable as it is assigned.

For extra advice on event handling, deadlock might occur between two threads if any *system calls* are made in the `WindowProc` function. DirectShow methods don't make system calls to my knowledge, so this only comes into play if you try calls to lower-level APIs such as Windows Sockets or the Waveform API in your `WindowProc` function.

The Microsoft documentation states that DirectX shouldn't be used in high-security computing environments, and the deadlock issue above might be one reason why. For high-security sound programming, the Waveform API is more appropriate.

E.2.3 Windows Sockets

The concept of *sockets* was created by Unix network programmers in the 80s, and has since been adopted by every major operating system that uses networking in any form. A socket is a sort of network "jack" created by an operating system. For instance, a TCP/IP socket is given a port number and an IP address. Any data sent to this socket using the proper functions will be sent using the socket's protocol, port, and address information. The socket might be set to listen for incoming connections or to make a connection to a remote machine.

A good synopsis of a socket is that it contains all necessary addressing, connection, and state information for data transfer between two computers. Once it is set up, the two computers can use a standard set of functions such as `send()` and `recv()` to transmit data back and forth.

Windows Sockets, or *Winsock*, is the Microsoft adaptation of the earlier Berkeley Unix sockets program. Winsock is a low-level API for programming network transfers between two Windows computers, and is standard across all Windows platforms. Socket functions are actually OS system calls; in fact open sockets on ports within a certain range (0 to 5000) are handled directly by the operating system and will remain open even after a program is shut down.

There are two major versions of Winsock: version 1 and version 2. Version 2 is backwards-compatible with version 1, but has a number of new Microsoft additions to the original set of Berkeley Unix socket functions. We encountered build issues with Version 2 on the remote side, and most of the Microsoft additions didn't seem useful for our purposes. Therefore, we used solely version 1 functions on both platforms, and used version 1 libraries with the remote program. Most (but not all) version 2 functions act similar to version 1, except with more parameters and a "WSA" prefix: `socket()` and `WSASocket()`, `send()` and `WSASend()`, etc.

As this is a low-level API, the functions may be confusingly named but the method and code itself remains simpler than object-oriented code. Anyone who knows C should understand how to create network programs that use sockets after following the steps presented in this short overview of Winsock programming.

Headers, Static, and Dynamic Libraries (Windows XP):
Headers:
- `winsock2.h` – Main Winsock 2 header
- `ws2tcpip.h` – TCP/IP specific definitions and prototypes

Static Libraries:
- `Ws2_32.lib` – Main static library for Winsock version 2

Runtime: Part of Windows XP

Headers, Static, and Dynamic Libraries (Windows CE .NET 4.1 and 4.2):
Headers:
- `winsock.h` – Main Winsock 1.1 header

Static Libraries:
- `winsock.lib` – Winsock 1.1 static library

Runtime: Part of Windows CE

2.3.1 Using Connection-State TCP/IP Sockets

TCP/IP sockets work much like a telephone call: one socket attempts to connect to a remote machine, hopefully the remote machine answers, and then data can be moved back and forth in full-duplex fashion. If the remote machine does answer, the connection state guarantees reliable data transmission (unless a hardware catastrophe occurs). The following steps demonstrate what needs to be done for two sockets to talk to each other over TCP/IP.

To a beginner many of the socket functions are named confusingly. It is just best to know when they need to be called rather than understand precisely what their purpose is in order to create simple programs. Here's the recipe – after this we'll dive into more detail. If the steps for the client and server diverge, we'll describe these differences in clear detail.

	Description	*Client (Sender) Function*	*Server (Receiver) Function*
1	Initialize the runtime library	`WSAStartup`	`WSAStartup`
2	Create the socket	`socket`	`socket`
3	Set Socket Options	`setsockopt`	`setsockopt`
4	Bind address to socket	None *(implicit bind)*	`bind`
5	Connect to server/Listen to client	`connect`	`listen, accept`
6	Send/receive data	`send`	`recv`
7	Monitor the socket state	`select`	`Select`
8	Disconnect/Close socket	`shutdown`	`Shutdown`
9	Shutdown runtime library	`WSACleanup`	`WSACleanup`

Table E.2 Steps to create and use a connection-state TCP/IP socket

1) Initialize the Runtime Library
Call `WSAStartup` with no arguments and Windows knows you are going to be using sockets. This is specific to Windows Sockets programming, but it is not a version 2 function.

2) Create the Socket

```
// Create and initialize an actual socket and a return value holder
SOCKET sock, sockval;
sock = sockval = 0;

sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(sock == SOCKET_ERROR)
{
      sockval = WSAGetLastError();
      // Handle the error or return/exit
}
```

Fig. E.11 Creating a socket and checking for errors

SOCKET is just a glorified way to specify an unsigned integer that describes a socket value. If this or any other socket function fails, the version 1 function `WSAGetLastError` will give you a value to plug into the Error Lookup tool to see what was wrong.

3) Set Socket Options (SO_REUSEADDR)

```
char t = TRUE;
sock_val = setsockopt(
                              sock,
                              SOL_SOCKET,
                              SO_REUSEADDR,
                              (char *) & t,
                              sizeof t
                              ) ;
```

Fig. E.12 Setting socket options

The only socket option we care about is the one that lets us create multiple connecting sockets on the same port with the same source address. By default calls to the `bind` or `connect` functions will fail if there is another socket on this port with a connection already.

4) Bind Address to Socket

Binding is a strange thing. Some functions such as `connect` which we'll use on the client side later just automatically provide the socket with the client's own address. This is called *implicit binding*. For the server we need to bind the socket to the server's own address explicitly. This involves a conversion step for putting the IP address and port number into *network-byte order*:

```
struct sockaddr_in client;  // Stores address in form for connect
struct sockaddr_in server;  // Stores address for explicit bind

// Set up the Listener IP address and port
unsigned short usPort = htons(65074);
const char send_ip_address = "192.168.0.1"

// Translate these into network byte order
unsigned long myIP = INADDR_ANY;
unsigned long yourIP = inet_addr(send_ip_address);
if ((myIP == INADDR_NONE) || (yourIP == INADDR_NONE))
{
     shutdown(sock, SD_RECEIVE);
     shutdown(sock, SD_SEND);
     return -1;
}

client.sin_family            = AF_INET ;
client.sin_port              = usPort ;
client.sin_addr.S_un.S_addr  = yourIP ;

server.sin_family               = AF_INET ;
server.sin_port                 = usPort ;
server.sin_addr.S_un.S_addr     = ADDR_ANY ;

sockval = bind(sock, my_ip, sizeof(my_ip));
```

Fig. E.13 Address conversion to network-byte order and binding to that address

Network-byte order is the order in which the bits are sent over the network. The functions and structures above are standard across most sockets implementations, and they convert the input from whatever form provided to a big-endian form which is either an unsigned long (the IP address) or an unsigned short (the port number), depending on how many bits are used to store the field in the IP header. `bind` and other functions expect a structure containing the network-byte order form of the local address and port it is using.

5A) Connect to server (Client-Side)

```
struct sockaddr_in saddr;

// usPort and ulIP have been converted to network-byte
// order using htons and inet_addr respectively (OMITTED)

saddr.sin_family            = AF_INET ;
saddr.sin_port              = usPort ;
saddr.sin_addr.S_un.S_addr  = ulIP ;

// Attempt to connect to the remote machine
sock_val = connect (
                sock,
                (LPSOCKADDR) & saddr,
                sizeof saddr
                ) ;
```

Fig. E.14 Connecting a socket to a remote computer

We are making the telephone call on this end, being implicitly bound to the local address. Remember that `connect` also needs its port and address in network-byte order.

5B) Listen to client (Server-Side)

```
// The data transfer socket provided by accept()
SOCKET sock_trans = 0;

sock_val = listen(sock, SOMAXCONN) ;
// Check for errors (OMITTED)

sock_trans = accept(
                sock, // Socket to accept connections on
                NULL, // Pointer to contain client's address
                NULL  // Size of above
                ) ;
// Check for errors (OMITTED)
```

Fig. E.15 Listen and accept, and a new data socket

The receiver code is always trickier than the sending code in network programming. `listen` is analogous to plugging a phone into the telephone jack, and `accept` is like picking up the phone when you hear the phone start ringing. Notice that the `accept` function returns the value of a new socket which will be used to move data back

and forth as the original socket continues listening for other connections. This behavior isn't terribly useful for a simple test program but it's how sockets work so you just need to know it. Again, you must set up the addresses you send these functions in a network-byte order `struct` as done in step 4.

An important fact to remember is that by default `accept` will *block*. This means that the program will wait for incoming connections as long as it needs to or until a timeout value is reached rather than go on to the next instruction.

6A) Send data to the server (Client-Side)

```
// Buffer consists of a message and its length
WSABUF sendbuf;
sendbuf.buf = message;   // Some constant-length input string
sendbuf.len = strlen(message);

sock_val = sendto(
           sock,            // Socket identifier
           sendbuf.buf,     // Address of the list of buffers to send
           sendbuf.len,     // Size of the list of buffers
           NULL,
           (LPSOCKADDR) & saddr,
           sizeof saddr
           ) ;
```

Fig. E.16 Listen and accept, and a new data socket

The `send` function can also be used, but our code uses `sendto` so we have the added power of specifying the IP address we're sending to over this socket regardless of whether it is *bound* or not to a specific IP address with `bind`. Using the `WSABUF` object keeps the string and its length in one place, which is also convenient.

6B) Receive data from the client (Server-Side)

```
sock_val = recv(
                sock_trans,           // Data socket
                message,              // Some constant-length string
                strlen(message),      // Its length
                0
                ) ;
```

Fig. E.17 Listen and accept, and a new data socket

On the receiving end, either this or `recvfrom` can be used. `recvfrom` looks like `sendto` while this looks like send. Both are acceptable, but these figures show our code as it was originally written, in a form we've verified that works. So even if it is inconsistent, at least both sides have been tested to work correctly.

When using dynamic buffers for the input string, make sure they are set up correctly. For test programs I find it easier to make static arrays of the form `const char[]` than to use dynamic buffers, but your mileage may vary.

7) Monitor the Socket State

The `select` function is confusing, but it serves an important purpose – it will tell you the state of your socket. Say the client has been sending data for a little while and decides to close the socket on its side. The server will

continue listening and accepting data on this connection, oblivious to the fact that the client has closed its side of the socket. There's probably no reason to keep this socket open once either side has closed. So what you do is call `select` to figure out the state of the socket after every `send` or `recv` and if either side has closed down, you'll do the same.

```
sock_val = select(NULL, &sockets, NULL, NULL, NULL);
sock_val = select(NULL, NULL, &sockets, NULL, NULL);
```

The first select function is checking a list of sockets to see if they are readable, while the other list is checking to see if the sockets are writeable. Look up the FD_SET and FD_ZERO to see how to modify this special sockets list.

8) Disconnect/Close the Socket

```
sock_val = shutdown(sock, SD_RECEIVE);
sock_val = shutdown(sock, SD_SEND);
sock_val = closesocket(sock);
```

Fig. E.18 Shutdown procedure for closing the "sock" socket

9) Shutdown the Runtime Library
Just call `WSACleanup` with no arguments, and Windows will know you're done using the sockets runtime libraries for now. While you may start and shutdown sockets many times while your program runs, chances are you'll only initialize and shutdown the runtime libraries only when you first start your program and when your program closes or exits with an error.

2.3.2 Using Datagram UDP Sockets

We considered using UDP, or datagram, sockets because they're easier to set up than connection sockets. Furthermore, because they have no overhead from a connection state, they transmit data faster than a TCP socket. However, datagram sockets do not guarantee data gets from one computer to the next. It works more like the postal service – information may be sent but not necessarily received. For purposes of transcription, it is critical to receive all the data that was sent, so we chose to use TCP sockets rather than UDP.

While we won't go into details about the code required, the following table is provided as a reference for the order in which to call socket functions so that a simple program can send and receive data on these connectionless sockets. Note the lack of connection steps, as they aren't needed for datagram sockets.

	Description	*Client (Sender) Function*	*Server (Receiver) Function*
1	Initialize the runtime library	`WSAStartup`	`WSAStartup`
2	Create the socket	`socket`	`socket`
3	Set Socket Options	`setsockopt`	`setsockopt`
4	Send/Receive data	`sendto`	`recvfrom`
5	Shutdown runtime library	`WSACleanup`	`WSACleanup`

Table E.3 Steps to create and use a connectionless UDP socket

2.3.3 Practical Information

On our PPT 8846 PDA we were able to open a socket and do data transmission from server PC to the client PDA (in other words, playback functionality in the RRS system) even when the PDA was turned off. When the PDA was turned back on, the playback through the PDA speaker resumed as if nothing happened. However, the audio data sent while the device was off was lost.
We're not sure if the socket remains open and the PDA remains in a sleep state, or whether the host just sits and waits for the device to turn back on.

If we had more time we would have made an event handler for socket events such as turning off the PDA or host computer. Windows does have event handling tools (outside of DirectShow's special `IMediaEvent` interface) but we didn't have time to learn them. For a robust commercial application, event handling for sockets and media streaming is essential – otherwise the socket routines will run and nothing will be tending to updating the GUI as it paints or buttons that might be pressed.

As always, make sure error checking is done every time a socket function is called. Our strategy was to do this error checking in a separate function that passed the `sockval` to be checked for the SOCKET_ERROR case, as in Fig. E. 11.

E.2.4 MFC Objects CSocket, CCeSocket, CSocketFile, and CArchive

The CSockets interface is built upon Windows Sockets, and is a higher level of abstraction than normal Windows Sockets, making it simpler to use. CSockets uses a subset of the CAsycnSockets interface, and is a more programmer-friendly interface. The advantages of CSockets over CAsyncSockets is that it is easier to use, and guarantees all data sent will be received, and in the same order that it was sent. Using this type of socket connection, the sending and receiving of data between client and host can happen asynchronously, and all received data is stored in a buffer until the client or host reads the data out of it's receiving buffer.

Headers, Static, and Dynamic Libraries (All Platforms):

Headers:

`afxsock.h` – Main AFX sockets header

Static Libraries:

`none`

Runtime: Part of Windows

2.4.1 Using CSockets

The basic steps of using the CSocket are to fist establish a connection for the CSocket between the server and client applications. Then, bind a CSocketFile to the CSocket, then bind a CArchive(s) to the CSocketFile, and use the CArchive(s) for sending and/or receiving data to and from the socket connection. The CSocket, CSocketFile, and CArchive objects are all AFX based objects, which means that AFX must be initialized in the application before the objects can be used.

	Description	*Client (Sender) Function*	*Server (Receiver) Function*
1	Initialize runtime lib	`AFXWinInit`	`AFXWinInit`
2	Construct a Socket	`CSocket sockSrvr`	`CSocket sockClient`
3	Create the Socket	`sockSrvr.Create()`	`sockClient.Create()`
4	Start Listening	`sockSrvr.Listen()`	
5	Seek a Connection		`sockClient.Connect()`
6	Construct new socket	`CSocket sockRecv;`	
7	Accept Connection	`sockSrvr.Accept(sockRecv)`	
8	Construct file object	`CSocketFile file(&sockRecv)`	`CSocketFile file(&sockClient)`
9	Construct Archive	`CArchive arIn(&file, CArchive::load)`	`CArchive arOut(&file, CArchive::store)`
10	Use the Archive to pass data	`arIn >> dwValue`	`arOut << dwValue;`

Table E.4 Steps to create and use AFX CSocket interface

1) Initialize the Runtime Library

Call AFXWinInit, passing the HINSTANCE of the current module, and any command line parameters passed to the console app when the executable was called. This can be done in an If statement with an error handler, because AfxWinInit returns a FALSE if it fails.

```
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
{
      _tprintf(_T("Fatal Error:  MFC initialization failed\n"));
      nRetCode = 1;
}
```

Fig. E.19 Initializing Afx

2) Construct a Socket
Call the constructor CSocket and specify the name you wish to give the socket object. You can also construct CSocket pointers and use the "new" command to construct the socket.

```
CSocket* m_pSocket = NULL;

m_pSocket = new CSocket();
```

Fig. E.20 Constructing a new CSocket pointer variable

3) Create the Socket
Call the Create member of the new CSocket object and specify the port (nPort), type of socket (SOCK_STREAM), and IP address of the local computer's network adapter you wish to create the socket on. The Create function returns a zero on failure.

```
if (!m_pSocket->Create(nPort, SOCK_STREAM, myAddress))
{
      delete m_pSocket;
      m_pSocket = NULL;
      cout<<"Socket Create Failed\n";
      return 0;
}
```

Fig. E.21 Creating the socket on the specified network adapter and port

4) Start Listening
On the server application, call the Listen member of the CSocket to listen for incoming socket connection requests on that port. The Listen function takes one parameter, which is the number of connection requests it is allowed to queue up. Listen is a blocking function, meaning program execution halts on that call until a connection request is received.

5) Seek a Connection
Once the server is listening for a connection, the client can request a connection using the Connect member of the CSocket. The Connect member takes two passing arguments, the IP address of the server, and the port on which the server is listening for a connection. The Connect function returns a zero on failure.

```
if(!m_pSocket->Connect(ServerAddress, nPort))
{
      delete m_pSocket;
      m_pSocket = NULL;
      cout<<"Connect to server failed\n";
      return 0;
}
```

Fig. E.22 Connecting the Client socket to the Server

6) Construct new Socket
Once the server has received a connect request, a new socket needs to be constructed to accept the incoming request on. This way, the server always has one socket used for listening for new connection requests, and always creates a new socket to connect to any socket connection requests.

7) Accept Connection
Using the newly created socket (that has not yet been initialized or connected), the server calls the Accept member of the original listening CSocket, and passes the newly created socket as it's only parameter. Accept then connects the newly created socket to the Client socket that requested the connection. The Accept function returns a zero on failure.

```
if(!m_pSocket->Accept(*m_pConnectedSocket))
{
      delete m_pSocket;
      m_pSocket = NULL;
      cout<<"Server Accept Failed\n";
      return 0;
}
```

Fig. E.23 Accepting an incoming client connection request

8) Construct File Object
Once a CSockets connection is established, a new CSocketFile object needs to be created and attached to the connected CSocket. The CSocketFile object acts as both a buffer for storing sent and received data, and as a standardized File type interface, so that reading and writing data to the CSocket connection is as simple as reading and writing data to a file. You can construct the CSocketFile directly, or create a pointer and call the "new" command to construct the object.

```
CSocketFile* m_pFile = NULL;

m_pFile = new CSocketFile(m_pConnectedSocket);
```

Fig. E.24 Constructing a CSocketFile object and connecting it to the CSocket

9) Construct Archive
The CArchive object is used as the interface for sending or receiving data to the CSocketFile. If one-way communication over the socket is all that is necessary, then only one CArchive needs to be created, passing the CSocketFile and the type of CArchive (load or store) as parameters. If two way communication over the socket is necessary, then two CArchives must be created; one for storing, and the other for loading.

```
CArchive* m_pArchiveIn = NULL;
CArchive* m_pArchiveOut = NULL;

m_pArchiveIn = new CArchive(m_pFile,CArchive::load);
m_pArchiveOut = new CArchive(m_pFile,CArchive::store);
```

Fig. E.25 Constructing a CSocketFile object and connecting it to the CSocket

10) Use the Archive to pass data
The CArchive objects can now be used for sending or receiving data over the Socket connection. Data transmitting and receiving can happen asynchronously between client and host. The CArchive accept or receive data through the overloaded "<<" or ">>" operator, much like "cin" and "cout" are used. Or for larger chunks of data, the "write" and "read" operators can be used, which accept a pointer to the address of the data to be sent, and the number of

bytes to be sent. The “read” operator returns an integer value of the actual number of bytes read. This is useful if the end of the buffer was reached before the requested number of bytes was read.

```
m_pArchiveOut.Write(lpData, nBytes);

nReadBytes = m_pArchiveIn.Read(lpData, nBytes);
```

Fig. E.26 Constructing a CSocketFile object and connecting it to the CSocket

E.2.5 DirectSound Buffers

DirectSound is part of the Microsoft DirectX API, and is used for directly accessing audio playback and capture buffers on any of the system's hardware sound devices that support the DirectSound API. A sound buffer is a memory location that can reside in system memory or directly in the sound hardware's memory, that is used for storing raw audio data. In a playback buffer, the application writes the audio data into the buffer, and calls the Play function on the DirectSound device to playback the audio through the system's sound hardware. In a capture buffer, on the other hand, the sound hardware writes the audio data to the buffer, and the application pulls that data out of the buffer for it's own use. There are two types of playback buffers; Primary and Secondary. There is only one Primary buffer, into which multiple Secondary buffer's audio data can be combined (mixed) for playback. For most applications, Secondary buffers should always be used for audio playback, in order to allow other applications that are running simultaneously on the system to playback sound as well.

Headers, Static, and Dynamic Libraries (All Platforms):
Headers:
`dsound.h` – Main DirectSound header
`dsutil.h` – included in the DirectX SDK
Static Libraries:
`dsound.lib` – Main DirectSound library
Runtime: Installed DirectX 8.0 or later (not included on Symbol OS Image)

2.5.1 Using DirectSound Buffers

To use the DirectSound API, you must first initialize the COM library. Once this is done, you can create the DirectSound device, and use the device to create a DirectSound buffer in the format specified in a DirectSound Buffer Description structure. Once the buffer is created, you can then lock a portion of the buffer (or the entire buffer) and using a memcopy routine, copy data into or out of the buffer. Using a Playback buffer, you must first memcopy the data into the buffer, then "Play" the buffer. With a Capture buffer, you do the opposite, and first "Play" the buffer to allow the sound device to capture audio into the buffer, then lock the captured portion of the buffer and memcopy the data out of the buffer. Once the memcopy is complete, you Unlock the buffer which frees that part of the buffer back to the sound device.

	Description	*Secondary Playback Buffer*	*Capture Buffer*
1	Initialize COM library	`CoInitialize()`	`CoInitialize()`
2	Construct Sound Device	`LPDIRECTSOUND pDS8`	`LPDIRECTSOUNDCAPTURE pDSC`
3	Construct Buffer	`LPDIRECTSOUNDBUFFER pDSB`	`LPDIRECTSOUNDCAPTUREBUFFER pDSCB`
4	Construct WaveFormat	`WAVEFORMATEX wfx`	`WAVEFORMATEX wfx`
5	Construct Buffer Description	`DSBUFFERDESC dsbd`	`DSCBUFFERDESC dscbd`
6	Create Sound Device	`DirectSoundCreate`	`DirectSoundCaptureCreate`
7	Set Priority Level*	`pDS8->SetCooperativeLevel`	
8	Create Buffer	`pDS8->CreateSoundBuffer`	`pDSC->CreateCaptureBuffer`
9	Begin Capture*		`pDSCB->Start`
10	Lock Buffer Section	`pDSB->Lock`	`pDSCB->Lock`
11	Copy Buffer Data	`[memcopy function]`	`[memcopy function]`
12	Unlock Buffer	`pDSB->Unlock`	`pDSCB->Unlock`
13	Begin Playback*	`pDSB->Play`	
14	Stop Buffer	`pDSB->Stop`	`pDSCB->Stop`

Table E.5 Steps to create and use DirectSound buffers (* used only for capture or playback)

1) Initialize the COM Library
Call CoInitialize(NULL) before doing any DirectSound code in order to initialize COM. CoInitialize returns an HRESULT, which will indicate if the call completed successfully or failed.

2) Construct Sound Device
Call the constructor for the DirectSound or DirectSoundCapture device.

```
LPDIRECTSOUND pDS8 = NULL;
LPDIRECTSOUNDCAPTURE pDSC = NULL;
```

Fig. E.27 Constructing a DirectSound object

3) Construct Buffer
Call the constructor for the Secondary Buffer or Capture buffer.

```
LPDIRECTSOUNDBUFFER pDSB = NULL;
LPDIRECTSOUNDCAPTUREBUFFER pDSCB = NULL;
```

Fig. E.28 Constructing a DirectSound Buffer object

4) Construct WaveFormat Structure
The WAVEFORMATEX structure is used to specify the number of channels, sampling rate, and bit precision of the sound buffer. For our purposes we stored these values in global variables, so that they could be reassigned in order capture and playback and any supported wave audio format.

```
WAVEFORMATEX wfx =
           {WAVE_FORMAT_PCM, nChannels, SampleRate,
           BytesPerSec, BlockSize, SampleBits, 0};
```

Fig. E.29 Setting up the Wave Format structure

5) Construct Buffer Description Structure
The DirectSound Buffer Description structure is used in a similar way as the WaveFormat structure, but also includes additional flags and settings because of the additional features available in a DirectSound buffer that a normal WaveFormat buffer does not have. The Capture Buffer Description is slightly different than the Playback Buffer Description in the types of sound effects flags that can be added to the buffer.

```
Capture Buffer:
      dscbd.dwSize = sizeof(DSBUFFERDESC);
      dscbd.dwFlags = DSBCAPS_GLOBALFOCUS;
      dscbd.dwBufferBytes = CAPBUFFERSIZE;
      dscbd.dwReserved = 0;
      dscbd.lpwfxFormat = &wfx;
      dscbd.guid3DAlgorithm = GUID_NULL;

Playback Buffer:
      dsbd.dwSize = sizeof(DSCBUFFERDESC);
      dsbd.dwFlags = 0;
      dsbd.dwBufferBytes = CAPBUFFERSIZE;
      dsbd.dwReserved = 0;
      dsbd.lpwfxFormat = &wfx;
      dsbd.dwFXCount = 0;
```

```
dsbd.lpDSCFXDesc = NULL;
```

Fig. E.30 DirectSound Buffer Description Structures

---

6) Create Sound Device
The DirectSoundCreate and DirectSoundCaptureCreate system calls take a newly created DirectSound Device object as a parameter, and return it as a configured and ready to use object.

```
hr = DirectSoundCreate(NULL, &pDS8, NULL);
hr = DirectSoundCaptureCreate(NULL, &pDSC, NULL);
```

Fig. E.31 DirectSound Create system call

7) Set Priority Level
When using a DirectSound Playback buffer, the application must first call the SetCooprerativeLevel member of the DirectSound device. This function takes the handle to the current window, and a DirectSound defined flag to set the priority level of the application's sound playback. The most commonly used priority level is DSSCL_PRIORITY, which allows the application to playback sound whether the window is active or inactive, but also does not give the highest priority control over the sound device.

```
hr = pDS8->SetCooperativeLevel(hWnd,DSSCL_PRIORITY);
```

Fig. E.32 Setting the sound playback priority level

8) Create Buffer
Now that the sound device has been created and set up, the DirectSound Buffer now needs to be configured to use that sound device. This is done by passing the buffer and buffer description to the CreateSoundBuffer member of the DirectSound device.

```
hr = pDS8->CreateSoundBuffer(&dsbd, ppDSB, NULL);
hr = pDSC->CreateCaptureBuffer(&dscbd, ppDSCB, NULL);
```

Fig. E.33 Creating and configuring the DirectSound Buffer

9) Begin Capture
When using a DirectSound Capture buffer, once the buffer has been configured, the buffer contains no audio data, and is ready to be filled by the sound device. Audio Capture begins one the Play member of the Capture Buffer is called. The Play member takes one argument, which is a flag that can be set to DSCBSTART_LOOPING. This flag allows the sound device to treat the buffer as a circular buffer, and infinitely capture audio to the buffer until the Stop member of the buffer is called.

```
hr = pDSCB->Start(DSCBSTART_LOOPING);
```

Fig. E.34 Start audio capture to the Capture Buffer

10) Lock Buffer Section
In order to access any of the data contained in the DirectSound buffer, you must first call the Lock member of the buffer to lock that memory space to allow read/write access of the application. The Lock member of both the Capture and Playback buffers are identical. The first two parameters are integers that indicate the byte offset of where to begin the locked portion of the buffer, and the number of bytes to lock after that offset. The last four parameters are memory pointers and integers returned by the Lock command; a pointer to the first locked portion of the buffer, and the number of bytes in that portion, and also a pointer to the second portion of locked buffer, and the number of bytes in that portion. The reason two portions are returned is that if the buffer is circular and the number of bytes requested in the Lock exceeds the end of the buffer, the locked section wraps to the beginning of the buffer.

For instance, in a 1000 byte buffer, if I call Lock with an offset of 800 and a size of 300 bytes, the first pointer would point to the location of byte 800 and number of bytes would be 200, and the second pointer would point to zero (the beginning of the buffer) and the number of bytes in that portion would be 100. Special care should be taken to not lock a portion of the buffer that is currently being played or captured to by the sound device. One way to check for this condition is to poll the position of the playback or capture pointer using the GetCurrentPosition member of the DirectSound Buffer.

```
hr = pDSB->Lock(dwBeginRead, PACKETSIZE, &pvAudioPtr1,
                        &dwAudioBytes1, &pvAudioPtr2, &dwAudioBytes2, 0);
```

Fig. E.35 Locking a portion of the DirectSound Buffer

11) Copy Buffer Data
Once a section of the DirectSound buffer is locked, audio data can be written to the locked memory block for playback, or copied from the block after being captured for the application to use. This can be done with a simple memcopy command into an array in memory, or in the case of this project, the lock pointers and number of bytes can be passed to the Read or Write members of a CArchive object, to send that audio data out of the socket, or read audio data in from the socket and store it to the DirectSound Buffer.

```
m_pArchiveOut.Write(pvAudioPtr1, dwAudioBytes1);
if(dwAudioBytes2 > 0)
     m_pArchiveOut.Write(pvAudioPtr2, dwAudioBytes2);
```

Fig. E.36 Sending captured audio data out on a connected socket

12) Unlock Buffer
After copying the buffer data, the locked portion of the buffer needs to be given back to the sound device. This is done using the Unlock member of the DirectSound Buffer. Unlock takes four parameters, which should be the same two pointers and integers that the Lock command returned, so that the exact same portion of buffer is unlocked as was previously locked.

```
hr = pDSB->Unlock(pvAudioPtr1, dwAudioBytes1, pvAudioPtr2,
                        dwAudioBytes2);
```

Fig. E.37 Unlocking the locked portion of buffer

13) Begin Playback
Because the playback buffer starts out empty, it should not be played until sufficient audio data has first been written into the buffer. Once enough audio data has been written to the playback buffer, begin audio playback using the Play member of the DirectSound Buffer. The Play member takes three parameters. The first is reserved and should always be NULL, the second is the priority of the sound playback (must be zero unless the DSBCAPS_LOCDEFER flag was set in the buffer description), and a control flag.

```
hr = pDSB->Play(0,0,DSBPLAY_LOOPING);
```

Fig. E.38 Playing the playback buffer

14) Stop Buffer
The Playback and Capture buffers can be stopped at any time using the Stop member of the buffer.

E.2.6 Microsoft Waveform API

Microsoft's Waveform API is the lowest-level API that can be used to program I/O on a Windows-compatible soundcard. It is an integral part of any version of Windows. It has functions that send buffers to and from the sound device, and simple playback/recording functions. Unlike the DirectShow or DirectSound APIs, it doesn't use object oriented programming techniques. Rather, each function is an OS call to the sound hardware. It uses PCM uncompressed audio formats for sending and receiving audio to and from the audio device.

For our code here we use two Microsoft Foundation Classes (MFC) helper objects to store bytes received from the sound hardware. You'll want to be comfortable with these objects for the final step where you write from the buffers to a file or to the network.

Headers, Static, and Dynamic Libraries (All Platforms):
Headers:
`mmsystem.h` –The main Waveform API header, short for 'multimedia system'.
Static Libraries: None
Runtime: Part of Windows

2.6.1 Recording Using the Waveform API

As with other low-level APIs, writing a working program is a matter of knowing which functions to call and when, rather than understanding exactly what those functions do.

	Description	*Recording Function*	*Playback Function*
1	Check if Audio I/O devices exist	`waveInGetNumDevs`	`waveOutGetNumDevs`
2	Select a wave format**	`IsInputFormatSupported`	`IsOutputFormatSupported`
3	Open the Audio I/O device	`waveInOpen`, `waveInStart`	`waveOutOpen`
4*	Allocate and Prepare buffers	`waveInPrepareHeader`	`waveOutPrepareHeader`
5*	Send or receive audio data using the buffer	`waveInAddBuffer`	`waveOutWrite`
6*	Un-prepare a buffer	`waveInUnprepareHeader`	`waveOutUnprepareHeader`
7	Read or write buffer to/from a file or a socket interface**	`CFile::Write` or `CArchive::Write`	`CFile::Read` or `CArchive::Read`
8	Close the Audio I/O device	`waveInStop`, `waveInClose`	`waveOutStop`, `waveOutClose`

* Described together as part of a *buffering loop*
** Our own functions: full source provided

Table E.6 Audio Recording and Playback using the Waveform API

Not only do the names of the corresponding record and playback functions look similar; they have the exact same arguments! In fact, by changing the variable type names (i.e. `HWAVEIN` to `HWAVEOUT`) and function names (i.e. `waveInAddBuffer` to `waveOutWrite`) in the record routine snippets shown in the following steps, you'll have a playback routine as well.

1) Check if Audio I/O devices exist

```
if(waveInGetNumDevs() < 1)   // Are there audio input devices?
   // Error
```

2) Select a Wave Format
The tricky part here is setting up the structure correctly. In our program we store the parameters for the `WAVEFORMATEX` structure as global variables. The `IsInputFormatSupported` function is simply a wrapper to the special case of `waveInOpen` with the `WAVE_FORMAT_QUERY` argument, which checks to see if the device will support the given format.

```
UINT nChannels = 1;
UINT SampleRate = 22050;
// For 16-bit audio this will always be twice the sample rate
UINT BytesPerSec = 44100;
UINT SampleBits = 16;
UINT BlockSize = 2;
MMRESULT RetVal = 0;
DWORD dwBuffLength = 8192;   // was 6000

WAVEFORMATEX wfx = {
      WAVE_FORMAT_PCM,
      nChannels,
      SampleRate,
      BytesPerSec,
      BlockSize,
      SampleBits,
      0};

RetVal = IsInputFormatSupported(& wfx, WAVE_MAPPER);
if(RetVal != MMSYSERR_NOERROR)
      // Error handling

MMRESULT IsInputFormatSupported
(LPWAVEFORMATEX pwfx, UINT uDeviceID)
{
    return waveInOpen(NULL,               // ptr can be NULL for query
                      uDeviceID,          // The device identifier
                      pwfx,               // Defines the requested
                                          // format.
                      NULL,               // No callback
                      NULL,               // No instance data
                      WAVE_FORMAT_QUERY); // Query only, do not open.
}
```

Fig. E.39 Checking to see if the wave format is supported by the audio device

3) Open the Audio I/O device
For a playback routine, the `waveInStart` that tells the microphone or line input to start recording sound is unnecessary. The buffer allocation step is very important, and should be done before recording begins with `waveInStart`. However, buffer allocation is a detailed process that is related more to the next step in our algorithm, so it is omitted here.

```
HWAVEIN hwin;    // Stores the identifier of the open wave device
                 // after a call to waveInOpen
```

```
RetVal = waveInOpen(
           & hwin,          // Will point to open wave device
           WAVE_MAPPER,     // Device identifier
           & wfx,           // Format request
           NULL             // Callback function (for event handling)
           NULL,            // No instance data,
           CALLBACK_NULL    // Flags
           ) ;

// Allocate buffers to the sound device (NOT SHOWN)

// Start recording
RetVal = waveInStart(hwin);
```

Fig. E.40 Opening the sound device for recording

4, 5, and 6) Allocating, Preparing, and Unpreparing Buffers: The *Buffer-Loop*
This is by far the most important part of the audio I/O process. These buffers are where the audio data is stored as it is transported from memory to the audio device. Loops that read buffers from the audio device for recording follow the same process as loops that write buffers to the audio device for playback. Don't worry about what "preparing" and "unpreparing" buffers means. Just know that these steps must be done every time a buffer is to be used by the audio I/O device.

The following code snippet creates two buffers that wave data can be put into. Here's a quick rundown of the steps it follows after the `WAVEHDR` structs are created:

A) Initialize `first_buffer` and `second_buffer` for the first time.
- a) Prepare the buffer with `waveInPrepareHeader`
- b) Do a `waveInAddBuffer` (`waveOutWrite` if playing)

B) Start the recording process by calling `waveInStart`. `first_buffer` will start receiving wave data at this point.
C) Start a `while()` buffer loop for filling buffers with recorded data
- a) Wait for `first_buffer` to get done being filled (the nested `while()` loop)
- b) Unprepare this buffer with `waveInUnprepareHeader`
- c) Send the unprepared buffer to a file using a MFC object
- d) Wait for `second_buffer` to get done being filled (the nested `while()` loop)
- e) Unprepare this buffer with `waveInUnprepareHeader`
- f) Send the unprepared buffer to a file using a MFC object

D) Re-initialize the first and second buffers with `waveInPrepareHeader` and `waveInAddBuffer`.

```
DWORD dwBuffLength = 8192;   // Size of each sound buffer

WAVEHDR first_buffer;
first_buffer.dwBufferLength = dwBuffLength;
first_buffer.lpData = new char[dwBuffLength];
first_buffer.dwFlags = 0;

WAVEHDR second_buffer;
second_buffer.dwBufferLength = dwBuffLength;
second_buffer.lpData = new char[dwBuffLength];
```

```
second_buffer.dwFlags = 0;

// Call waveInOpen HERE (Omitted)

//////////////////////////////////////
// Setting up buffers for the first time

// Prepare the first buffer to receive audio
RetVal = waveInPrepareHeader(
                hwin,                   // Device handle
                & first_buffer,         // WAVEHDR structure
                sizeof(WAVEHDR)         // Size of the buffer structure
                ) ;

// Add buffer to the sound device
RetVal = waveInAddBuffer(
                hwin,                   // Device handle
                & first_buffer,         // WAVEHDR structure
                sizeof(WAVEHDR)         // Size of the buffer structure
                ) ;

RetVal = waveInPrepareHeader(
                hwin,                   // Device handle
                & second_buffer,        // WAVEHDR structure
                sizeof(WAVEHDR)         // Size of the buffer structure
                ) ;

RetVal = waveInAddBuffer(
                hwin,                   // Device handle
                & second_buffer,        // WAVEHDR structure
                sizeof(WAVEHDR)         // Size of the buffer structure
                ) ;

// Call waveInStart HERE (Omitted)

int cnt, nLoops;
cnt = 0;
nLoops = DESIRED_TIME_SECS * 3;

//////////////////
// The buffer loop

while(cnt < nLoops) // Capture a few seconds of audio
{
     // Poll to see when the first buffer is done being written to
     while((first_buffer.dwFlags & WHDR_DONE) != TRUE);
     waveInUnprepareHeader(hwin, &first_buffer, sizeof(WAVEHDR));

     // Call MFC method to write first buffer to a file or socket
     // right HERE (omitted)

     // Prepare the first buffer while the second one is filling up
```

```
      RetVal = waveInPrepareHeader(
                        hwin,                  // Device handle
                        & first_buffer,        // WAVEHDR structure
                        sizeof(WAVEHDR)        // Size of the buffer
                        ) ;
      RetVal = waveInAddBuffer(
                        hwin,                  // Device handle
                        & first_buffer,        // WAVEHDR structure
                        sizeof(WAVEHDR)        // Size of the buffer
                        ) ;

      // Poll to see when buffer is done being written to
      while((second_buffer.dwFlags & WHDR_DONE) != TRUE);
      waveInUnprepareHeader(hwin, &second_buffer, sizeof(WAVEHDR));

      // Call MFC method to write first buffer to a file or socket
      // right HERE (omitted)

      // Prepare the second buffer while the first one is filling up
      RetVal = waveInPrepareHeader(
                        hwin,                  // Device handle
                        & second_buffer,       // WAVEHDR structure
                        sizeof(WAVEHDR)        // Size of the buffer
                        ) ;
      RetVal = waveInAddBuffer(
                        hwin,                  // Device handle
                        & second_buffer,       // WAVEHDR structure
                        sizeof(WAVEHDR)        // Size of the buffer
                        ) ;
      cnt++;
}
```

Fig. E.41 The buffer loop

Hopefully you see that this is just a typical while loop setup where you initialize a variable (in this case the buffer) before entering the loop for the first time, and then you update the buffers every time the loop iterates. It really simplifies down to something like this:

```
// Initialize buffers for the first time HERE

cnt = 0;
nLoops = DESIRED_TIME_SECS * 3;
while(cnt < nLoops)
{
      // Update, wait on, and then reinitialize buffers HERE
      cnt++;
}
```

Fig. E.42 Our buffer loop, itty witty baby version

This easier loop makes the timing of our algorithm clearer. Each time we read in `first_buffer` and `second_buffer` with a buffer size of 8k, we've read in 16k. Experimentally this gives us about 0.32 seconds of audio every time we read in a buffer. This is where the `nLoops` value comes from, calculated from the number of seconds we want. If our loop were in a function, we would want this time in seconds to be an argument to that function.

For a smarter program this loop would run in its own thread and run indefinitely until the program received a message that it should stop – like, say, from the network computer sending or receiving data using this loop. You might be interested just how we would add file writing functionality or network functionality to a loop like this. That step is next on our list!

7) Read or write a buffer to an outside interface
This is the other tricky part about writing record or playback routines. There are countless ways to read in information from files, buffers, streams, or networks. The charm of using the MFC classes is that the objects we use are relatively simple, and make way for short and easy helper functions if they are needed at all. The following code initializes the objects and illustrates how they might be used to receive data from the network and write it to a file.

```
CFile myFile = NULL;
CSocket* m_pConnectedSocket = NULL;
CArchive* m_pArchiveIn = NULL;
void *pvAudioPtr1 = NULL;
UINT iTemp = 0;

m_pConnectedSocket = new CSocket();
// Create the socket as a sender or receiver with your own routine HERE
m_pFile = new CSocketFile(m_pConnectedSocket);
m_pArchiveIn = new CArchive(m_pFile,CArchive::load);

// Receive with this home-made function
iTemp = NetSendReceive(*m_pArchiveIn, pvAudioPtr1, dwAudioBytes1);

// Write void* wave memory to a file
myFile.Write( pvAudioPtr1, iTemp );
```

Fig. E.43 Receiving data over a socket with MFC objects

The example above initializes all the objects needed to create a MFC socket object, a MFC file object, and the helpers needed to send buffers over a network interface. Information on creating connected or listening `CSocket` objects is discussed in the MFC documentation in section E.2.4. Notice that our `NetSendReceive` can look at the socket mode and figure out whether to send or receive data, using the same Read and Write method as the `CFile` object does.

```
UINT NetSendRecieve(CArchive& m_pArData, void* Data, UINT nMax)
{
    if (m_pArData.IsStoring())            //send mode
    {
            m_pArData.Write(Data, nMax);
            m_pArData.Flush();
            return 1;
    }

    else    //receive mode
    {
```

```
            return m_pArData.Read(Data, nMax);
    }
}
```

Fig. E.44 Network data transfer code that checks the state of a `CArchive` object

Remember that the `CSocket` object is replaced in Windows CE by a `CCeSocket` object that has very minor differences in how it handles its internal state.

8) Close the Audio I/O device

```
waveInStop(hwin);
waveInUnprepareHeader(hwin, &first_buffer, sizeof(WAVEHDR));
waveInUnprepareHeader(hwin, &second_buffer, sizeof(WAVEHDR));
waveInReset(hwin);
waveInClose(hwin);
```

Fig. E.45 Audio I/O shutdown code

If you are closing an audio stream for good, make sure to call the functions in this order. `waveInReset` may not be necessary here as it just resets the audio device – this causes the audio device to go back to the head of whatever buffer it was playing.

2.6.2 Hints, Tips, and Closing Remarks

The buffer loop is the most important part of your audio I/O routines. Every time you allocate a buffer for the audio device, you should do error checking (which was omitted from the code samples above). Generally the I/O device has very limited resources so don't try to send 10 or so buffers with 8kB memory each to the device with `waveInAddBuffer` because resources will run out and eventually these calls will start failing.

After running these routines it is important to deallocate the MFC objects you created in memory with the new keyword. Objects left in memory are the source of memory leaks and they will cause your program to shut down with an error after your audio operations have completed.

For debugging, make sure your program has the chance to call `waveInClose` after it opens the audio device for I/O. If you fail to do so you might not be able to open the device later because it is still "in use". On the PDA this would require a soft reset to fix.

E.2.7 Benefits and Disadvantages of Windows Programming

After discussing no less than six different Windows APIs, you might be curious about the overall strengths and shortcomings of the "Windows way" of programming software. Even after months of reading through Microsoft documentation and becoming more proficient in C and strange data types, we still aren't experts in Windows programming. Still, we've seen the ropes and also have special insight on porting between different versions of Windows. So let's take a look at the good and the bad of Windows programming.

2.7.1 Programming Environments

With cross-platform Windows development, the biggest strengths and weaknesses lie with the development tools. The ability to debug your code step-by-step is crucial when developing GUI interfaces where it is impossible to use debugging output statements like `printf()`. With adjustable breakpoints and quick searching for variable and function definitions, it is the best debugging utility we have ever used.

On the flipside, the Visual Studio .NET project format is not compatible with the eMbedded C++ project format. This caused hours of copying files from one directory to another, and adding them to a completely separate eMbedded C++ project.

2.7.2 Microsoft APIs

Having the Windows APIs mostly supported across both Windows CE and Windows XP was very useful. For low-level code such as Windows Sockets and the Waveform API, the code was completely portable between platforms. The significance of this cannot be denied.

On the other hand, there were some irritating differences between platform-specific MFC objects such as `CSocket` and `CCeSocket`. DirectShow on CE was a terrible problem, and we spent months of time porting over sample code for network DirectShow filters. In that time we could have used the lower-level APIs to fully implement the functionality we needed and used the extra time for testing and implementing WAP selection into the RRS system.

This brings me to the biggest problem with Microsoft APIs. There are hundreds of them, and even though many of them claim to be for a certain task, often times this object-oriented approach just has too much overhead and isn't worth the time to develop with. None of the Microsoft documentation will tell you that a certain task will take less time using one API over another – unless you have help and supervision you'll be left trying to find the best tool for your job. If you're like us, the first few tools (and months of work) just won't cut it.

Of all the object-oriented work we did, we learned one thing – you can't debug an object until you know how the lower level functions and methods it calls works. Having a DirectShow network filter is wonderful, but you won't be able to debug it unless you know how it uses Winsock to create a socket and transfer network data. Objects can be powerful tools, but they certainly aren't crutches for the uninitiated like we originally presumed.

2.7.3 Object-Oriented Programming

As I mentioned previously, there are ways to take object-oriented programming to serious extremes as with DirectShow. However, not all of Microsoft's objects are bad – `CFile`, `CSocket`, and `CArchive` are wonderful tools for encapsulating the functionality of a lower-level class into an object that will do most of the tedious set-up for you. These objects are easier to work with in dynamic memory than many smaller variables, as they can be allocated and deallocated together, as necessary.

API objects generally have a steep learning curve. This isn't helped by the dense, brief documentation in the MSDN library and the lack of sample source code demonstrating how the objects are used. Furthermore, the Microsoft way of assigning CLSIDs to objects and storing them in runtime libraries (with references in static libraries) is very confusing to work with at first. These libraries aren't portable like the source code, and must be rebuilt (and reorganized into new projects) when moving between C++ .NET and eMbedded C++.

F. TECHNICAL NOTES SHEETS

Technical Notes: An Example
Dialog Applications
Instantiating the Filter Graph
Making ActiveSync Work
Creating A MFC Console Application Project
Platform Builder: Windows CE OS Images and Platform SDKs
Configuring a Wireless Network Between a Host and a PDA

Our technical notes describe in detail important steps we took to set up a working environment between a PC and a PDA. These notes may not be necessary for you in setting up a build environment to duplicate our own results, but they will certainly save you time if you do read them! They are written from a practical step-by-step perspective as we were figuring out how to make our project work.

For those with the same amount of background experience and education as we have, these notes should prove useful.

Justin Cassidy
5/04/2004

Technical Notes: An Example
Justin Cassidy
Revision 0

Introduction

Over the course of our Senior Design project, we'll be working with a number of technologies that we've never studied before elsewhere. The purpose of a short technical note paper such as this is to compile the work and research you've done on a topic into a recipe-like form that is easy to refer to after the fact. I envision our final documentation will have a section devoted to categorically organized technical note sheets.

The goal of these papers is not to be a writing exercise; it is to put the information you are using and are likely to forget or need to look up later into a easily referenced source that is easy to send to others. I devised this format as an idea for describing the steps to making a Windows console application with certain features.

Description

Format

The two main sections are the Introduction and the Description section, plus whatever other main sections you feel are appropriate, such as a conclusion or results section. In order to keep all of these documents perfectly consistent, I propose sticking to Times New Roman 12pt font, with the exact same heading formatting for main sections and subsections as this document. The files should be in the MSWord .doc format, and use the PNG format for inlaid images.

Content

The introduction is one or two short paragraphs, which summarize the goal behind the note sheet and summarize the content of the paper.

The description should be concise and only contain what information is needed. Pictures, graphs, tables, and lists of instructions are the name of the game. Think of a recipe that is easy to refer to and communicate later. In terms of writing for an audience, write for people in the following order of importance:

1) Yourself
2) Other members of Team Dialect
3) Sponsors at AID
4) Other Computer Engineers or Programmers not used to the concepts you're describing

Dialog Window Applications: Microsoft Windows CE .NET 4.2
Justin Cassidy
Revision 2

Introduction

Since the Sound Recording Unit we are using has no screen, we don't need to make a full-fledged Windows application on the remote end. These are technical notes on Windows programming and event handling, targeted for someone with no prior Windows programming experience, using the eMbedded Visual C++ 4.2 development environment.

The notes describe the development of a simple button-based dialog window application. Although a console application was also considered, embedded Visual C++ does not document how these are created, and for accepting simple keyboard scancodes dialog applications seem to do just as well as the `stdio/scanf` solution.

Description

Project Creation
File -> New -> WCE Application -> Dialog -> Windows Sockets

The files created set up the dialog box with a default TODO message, which can be removed using the Resource Viewer/Editor tools.

User Interface
By clicking the Resource Viewer and clicking on the dialog resources, you have access to what the dialog window will look like and you can even add buttons or text simply by hand. These changes are stored, along with accelerators, in resource files that are called by the C code. The changes are not made in the C code itself, which can be confusing.

Double clicking on a created button will make a class method in the proper code file that can be edited. Every new button created has an incremented ID and this can be changed with a right-click… not sure if this matters. Each button and probably each GUI object has special events associated with it and once they are defined to exist, a proper class method will be created that can be lovingly coded.

Topics Related to Keypad Input

Accelerators: Windows Resource for Assigning Keystrokes to Actions
Since the final HW will have no screen, so all actions should be accessible with single keystrokes. Accelerators are keystrokes that accomplish tasks defined in the user interface. IBC_BUTTON1 seems to define a single button click to "Button1" and a key can be mapped to do that same action. Two types of keys can be assigned: ASCII keys and VirtKeys. VirtKeys is the naming system used internally by Windows CE. When you add a button and right-click it for its properties, press the 'Use next key pressed' button to press a key and assign the proper VK.

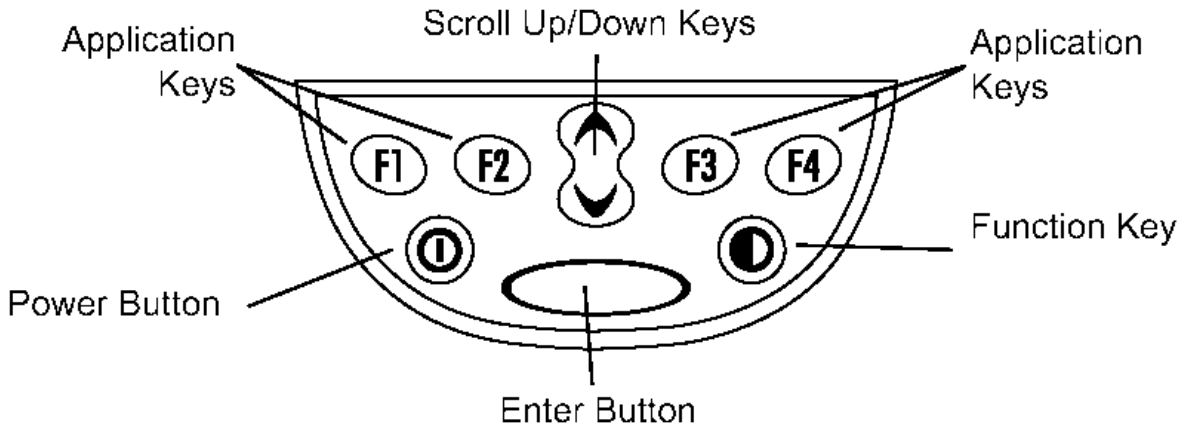
This doesn't seem to work when I test using breakpoints, although clicking on the buttons with the mouse in the emulator works. So we'll avoid accelerators for now.

Accelerator Alternatives

Using a button with a name of &Play or Pl&ay makes the 'P' and 'a' keys, activate those buttons, respectively, and this is a working alternative to accelerators.

(i) The PPT Keypad

On the PPT 8846, a key editor called "Keyboard Properties" will let you change the unit's buttons to correspond to different keyboard keys.



The default layout for the keys is as follows:

PDA Keypad	**Keyboard Scancode**
Enter	Enter
F1	Tab
F2	Left Cursor
F3	Right Cursor
F4	Escape

Working with a Windows button named &Play, the PDA 'Enter' key could be assigned to keyboard scancode 'P', but these keystroke values are reset every time the PDA does a hard reset. So reassigning PDA keys in CE isn't recommended. Because Windows buttons have a 'tab-order' by default, the two PDA buttons F1 and Enter could be used to cycle between all the functionality of the dialogue box. However, we want each button to have a function.

Since we wish to interrupt stopping and starting audio with these keys, our program will have to worry about event handling. Our technical notes on event handling are discussed in the DirectShow section of our final report.

Debugging Overview

I have not found a way to use `printf` or `scanf` or redirect `stdin` and `stdout` to a terminal within the eMbedded C++ IDE. Not for lack of trying… I can't find any documentation on how the little output window at the bottom of the IDE works, although it does work with a .vcl file (HTML-formatted) in the project top-level directory for its build log. I tried redirecting output to that file but it doesn't work.

Breakpoints are useful for seeing where you are in the code, and assert statements in the debugger can look at variable values as the program is running. This is the same sort of functionality debugging `printf` statements provide, although there is no analogue for `scanf`.

Instantiating The Filter Graph
Justin Cassidy
Revision 3

Introduction

DirectShow uses a Filter Graph paradigm, i.e. connect the decode filter to the playback filter to play an encoded media file. This document describes how to build the filter graph in a Windows CE application, the problems we faced, and the corrective measures we took to solve them. It is written towards a group which is inexperienced in both DirectX and Windows programming.

Nomenclature

Microsoft's sample programs use naming conventions that are sensible in a way. Their protected pointers are named m_pGB, m_pCapture, etc. and these point to memory locations for the graph builder, the capture filter graph, etc. "m" stands for *memory* while "p" probably stands for *protected*.

Generally, when a Windows function is called to create something or do an event, the function returns type HRESULT. This is an internally defined 32-bit value which Microsoft has defined constants for, such as the values E_NOINTERFACE and NOERROR. You can easily trace their definitions back to `windows.h` in the Microsoft IDEs by right-clicking -> Go To Definition.

So when this document mulls over code like the following (used to instantiate a DirectShow filter graph), we can analyze it. Get used to arguments that are reserved or meaningless to most programmers (such as that second NULL in the CoCreateInstance).

```
hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
                      IID_IGraphBuilder, (void **)&m_pGB);
if (FAILED(hr) || !m_pGB)
   return E_NOINTERFACE;
return NOERROR;
```

Figure 1. Calling the Graph Builder

We're trying to create a filter graph with the graph builder, allocating memory m_PGB and checking to see if the operation worked. If it did, m_PGB will point to a non-zero memory address and hr will be greater than zero, thus making the condition fail.

Instantiating the Filter Graph

Whether doing audio playback or capture, when using DirectShow the first step is to initiate the GraphBuilder. The sample code in Figure 1 was copied from the DirectShow audio sample program. When put into a method, it was found that hr was failing and the pointer m_pGB was NULL. The CoCreateInstance was failing in Figure 1. After putting the error code contained in the hr value into Tools -> Error Lookup, I learned I was getting a "Class not registered" error. What does this mean?

`REGDB_E_CLASSNOTREG`	A specified class is not registered in the registration database. Also can indicate that the type of server you requested in the **CLSCTX** enumeration is not registered or the values for the server types in the registry are corrupt.

Figure 2. Microsoft Help Description of the `CoCreateInstance` Registry error

I presumed this meant the default emulator image doesn't have all the DirectX libraries installed correctly. So we created a new one, a long and painful process. See the `Platform Builder` technical notes on how to do this.

Hopefully Helpful Notes

`m_pGB` and other filter graph pointer variables are protected in `DialogappDlg.h` so they can't be called unless the function is a method in the `DialogappDlg` class. So, don't use `HRESULT Record()` but rather `HRESULT DialogappDlg::Record()`.

Making ActiveSync Work
Tommy Stear
Revision 1

Introduction

This paper tells how to install Microsoft ActiveSync file and database synchronization software on the host computer, and to configure the host computer to connect with the remote unit.

What is it?

ActiveSync is a tool made by Microsoft that allows owners of handheld computing devices that are based on the Embedded Windows platforms to synchronize files with a desktop computer system running Windows. With this tool, users can synchronize their email and calendar databases, to-do lists, and phone and address books and make them mobile on the handheld device. The remote unit in the Remote Recording System is a device based on the Windows CE embedded computing platform.

---

1. Download the latest version of ActiveSync from the Microsoft website, in the "Downloads" area: http://download.microsoft.com.
2. Install ActiveSync on the host computer.
3. Install the cradle or cable (supplied with the remote unit) on the host computer.
4. Restart the host computer.
5. For USB cradles and cables,
    a. After Windows is finished loading, turn on the handheld device, and plug it into the cradle or cable firmly.
    b. Windows will automatically find the handheld device and ask you to establish a "Partnership" with the device. Here you can allow the device ability to connect to more than one host computer for synchronization, or to sync with only one computer, eliminating partnerships with other computers that are stored on the handheld device.
    c. Once you have a partnership established, you can connect to the remote device much like a disk drive that is attached to your computer.
6. For Serial (COM:) port cradles and cables,
    a. After Windows is finished loading, open the Properties Page for "My Computer." Make sure you have administrative privileges.
    b. On the "Hardware" tab, click "Device Manager."
    c. In the hardware list that appears, find the "Ports" section of the hardware category tree. Make sure you are viewing the devices by type, not by connection.
    d. Double click the COM: port that your cradle or cable is attached to open the properties for that port. Usually, this is COM1: or COM2: but your specific setup may vary.
    e. Now click the button for "Advanced" settings. In the dialog that appears, make sure the box is unchecked for "Use FIFO buffers…."
    f. Restart your computer. After Windows is finished reloading, turn on the handheld device, and connect it to the cradle or cable firmly.
    g. Click "Start," "All Programs," "ActiveSync." Windows should automatically find your device and ask you to establish a "partnership" between the host

computer and the handheld device. Here you can allow the device ability to connect to more than one host computer for synchronization, or to sync with only one computer, eliminating partnerships with other computers that are stored on the handheld device.

h. Once you have a partnership established, you can connect to the remote device much like a disk drive that is attached to your computer.

In Closing

While the hard connection is not necessary for everyday use of the remote unit, it will allow for testing and software repairs or updates later.

Creating a MFC Console Application Project
Tommy Stear and Justin Cassidy
Revision 1

Introduction

This paper describes how to setup and start a new project in Visual Studio that uses the Microsoft Foundation Classes.

What it is

The Host Application uses the Microsoft Foundation Classes (MFC) to set the basis for certain Windows-specific types and other global objects. The MFC establishes objects, global constants, and type definitions that programmers can use to make applications operate more efficiently under Windows environments. This includes useful wrappers to Windows Sockets (network connections) and writing files.

1. Open Microsoft Visual Studio.NET 2003.
2. Start a New Project from the "File" menu. Navigate through "Visual C++" to the "Win32" category, and choose the "Win32 Console Application."
3. Type a name for your project and click "Next." Then click "Application Settings" to define a few important properties for your project.
4. Click "Application Type," "Console Application," and choose to "Add Support for MFC."
5. Under "Additional Options," uncheck the box for "Precompiled Headers."
6. Click "Finish" to close and start the project.

You are now ready to start inserting code that uses the MFC base.

In Closing

The host application uses the Microsoft Foundation Classes to make functionality operate more smoothly and more efficiently under Windows environments. Using MFC makes the host application work well with the other common objects used by Windows.

Platform Builder: Windows CE OS Images and Platform SDKs
Justin Cassidy
Revision 3

Introduction

The Platform Builder is used to create a fine-tuned version of Windows CE with exactly the hardware support and protocols your embedded device needs. There are two important steps most developers need to take in order to test applications that use APIs such as DirectX:

1) Create a Windows CE platform with the required DirectX libraries
2) Create a Software Development Kit (SDK) so you can use the OS image with the remote device through the eMbedded C++ IDE.

Our team needed to make an image with DirectX support, as we were getting registry errors for classes not included on our default emulator image. The platform builder can automatically builds an OS image for the device, along with a SDK for the eMbedded IDE.

Steps

I. Build the Custom OS Image

1) Run Platform Builder, and select the New Platform Wizard from the File menu.
2) Choose the "Remote Media Server" build profile, and make sure 802.11 support is included in the network options.
3) Look through the Catalog bar, docked on the right hand side of the IDE, and select the features you wish to add to the operating system. The following is a list of features built into the OS image we used, not all of which may be necessary.

   DirectShow Core
   DirectSound
   ACM (Audio Compression Manager)

4) Build the image. This took 30 minutes on a 1.7GHz 512MB RAM system.

II. Build the SDK

1) Choose Platform -> Configure SDK, and make sure eMbedded C++ support is included, as well as all library options (ATL, MFC, etc) *except* for the .NET framework. Remember the name you give your new SDK (ours was SRU-R).
2) After this is done, choose Platform -> Build SDK. This takes 10-15 minutes on a 1.7GHz 512MB RAM system.

III. Set up the Emulator Image with eMbedded C++

In eMbedded C++, under the list of SDKs you should see an option for the SDK you created. In our case, we were able to select SRU-R for the platform and SRU-Emulator for the device from the drop-down menus. If you've added all the right features, you'll notice the 'Class Not Registered' errors disappear when you run and test your programs.

IV. Set up the OS on an actual device

This is a matter of using Tools -> Configure Platform Manager, and setting up the transport schemes properly. For our PPT 8846 PDA we believe Symbol makes it impossible to flash the OS through Platform Builder. They have their own tool for this task, requiring you to buy only supported binary images from Symbol. You may have more luck with a different PDA.

If your device is set up correctly, you can use your SDK to build to your actual target device in eMbedded C++, in exactly the same way described in part III above.

Configuring a Wireless Network Between a Host and a PDA
Justin Cassidy
Revision 2

Introduction

It would be nice if wireless peer-to-peer networks were as simple as two computers together in a room. Unfortunately it's slightly more complicated. This process describes how to set up a peer-to-peer wireless connection between a Windows XP host and an 802.11 wireless-equipped Symbol PPT 8800 series PDA using static IP addresses.

Process

Host

1) Turn on your wireless network card, or if using a laptop, insert into your PCMCIA slot.
2) Go to Control Panel -> Network Connections.
3) Right-click the 'Wireless Ethernet Connection' icon and click 'View Available Wireless Networks'.
4) At the list of available networks, click 'Advanced'.
5) Below the list of preferred networks, click 'Add'
6) In the box that appears, do the following:
    a. Give your network an arbitrary name. Ours was 'Locus'.
    b. Select WEP encryption and provide an arbitrary key. It will tell you how long your key can be. Remember the key you used for this step, as you'll need it when configuring the remote unit. Our key was a ten-character hex key, i.e. characters in the set {0-f}, which made it a 40-bit WEP key.
    c. Make sure you check the option for 'ad-hoc' network.
7) Usually neither computer on a peer-to-peer network knows how to self-configure an IP address using DHCP. So you'll need to set a static IP address:
    a. Right-click your 'Wireless Ethernet Connection' icon again, select 'Properties' and then find 'Internet Protocol (TCP/IP)' and click properties.
    b. In the next window, tell it to use the address you specify. We chose an address that wouldn't likely be on the other networks the host may be connected to, namely "192.168.0.108".

Remote

1) Double tap the network card icon on the Windows CE taskbar, and select 'WLAN Profiles'
2) Create a new profile, and do the following:
    a. Give the profile the same name you gave your host profile (Locus).
    b. Click the check-box to set the connection to ad-hoc mode.
    c. Click the encryption tab and provide the same key you did earlier. If you did a 10-character key earlier as described, select '40-bit Shared Key' as the encryption algorithm.
    d. Click the IP Config tab and give yourself a static address. Ours was "192.168.0.108".
    e. Back at the 'WLAN Profiles' window, connect to the new Locus profile.

Remarks

Without a key, you may create a profile that doesn't find the correct unprotected wireless connection. So make sure you set up a key with your profile. Also, the profile on remote and host ends must have the same name or they won't know to talk to each other.

DO NOT BRIDGE YOUR WIRELESS CARD TO OTHER CONNECTIONS or you will pay a huge performance penalty on your wireless network. We're talking 2kB/sec transfers versus 20kB/s transfers. To avoid using a bridge, you can disable other network cards on your computer, or set the wireless card as the preferred device in the settings for Windows Networking. See your computer's documentation for more details about preferred network devices.

G. REFERENCES

[1] "Microsoft End-Of-Life Support Cycle." Visited: 02/25/04. Avail.: http://support.microsoft.com/default.aspx?scid=fh;[ln];LifeEmbedded

[2]

[3]

[4]

[5]

[6] "NCH Tone Generator." Visited: 4/13/04. Avail.: http://www.afreego.com/Categories/Multimedia/Audio_and_Sound_Tools/016646.php