Audio Message App Summary

Purpose:

Simply put, the audio message app is intended to allow the user of one embedded device to send an audio message to the user of another embedded device.

Hardware:

- 2x Texas Instruments CC3200 LaunchPadXL embedded devices
- 2x Texas Instruments CC3200 Audio BoosterPack add-ons
- Speakers or headphones connected via 3.5mm audio jack

Development Tools:

- Texas Instruments Code Composer Studio IDE
- Texas Instruments Uniflash utility (for flashing binary executables to LP devices)
- Texas Instruments Smart Config app (for connecting LP devices to wireless access point)
- Amazon Web Services Elastic Cloud Compute (AWS EC2) instance
- Simple and Fast Multimedia Library (for EC2 UDP server/client)

High-level Overview and Use Case:

The user of LP1 wants to send a message to the user of LP2. User one presses the SW3 button on LP1. A green LED comes on to indicate the device is now recording audio. User of LP1 communicates their message and presses SW3 a second time, the green LED turns off, and LP1 stops recording audio.

User of LP2 then presses the SW2 button on LP2 and an orange LED comes on to indicate the beginning of audio message playback. User of LP2 should now hear the message. The audio message will stop playing when it has finished and the orange LED will automatically turn off.

EC2 Instance Audio App High-level Overview:

The UDP server/client running on the EC2 instance consists of a main program, an audio recording class, an audio playback class, and common message functions shared by all three.

The main function is the entry point of the program and is responsible for creating the UDP socket, audio player, and audio recorder. If it successfully creates the UDP socket, it enters the main loop which listens infinitely for UDP packets. When it receives a packet, it parses it and looks for the special control messages defined in network.h. If it finds a begin audio recording message it opens a file for the new message and delegates to the audio recorder object. Likewise, if it finds an audio playback message, it opens the audio message file and delegates to the audio player object.

The audio recorder continues to receive UDP packets on the socket and port passed to it by main and writes the data to file. It stops and returns once it encounters the end of audio message packet.

The audio player continues to read audio data from the file and send it to the LP device. It stops once it finds EOF and sends the end of message packet to the LP device.

Notable classes and functions:

- main.cc: contains the main program
  - main(): defines control flow of EC2 instance audio message app
- player.cc: contains audio player class implementation
  - playAudioMessage(): reads audio data from file and sends it to LP device

- recorder.cc: contains audio recorder class implementation
  - recordAudioMessage(): receives audio data and writes it to file
- common.cc: contains implementation of common functions used for special control messages
  - isSpecialPacket(): checks whether or not a packet is a special control message
  - sendSpecialPacket(): sends a special control message
- network.h: defines special messages, packet size, and audio port

LP Device Audio App High-level Overview:

The program running on the LP devices is a bit more complicated than the one running on the EC2 instance so I will focus on the components most relevant to this project that I modified. The LP app consists of a main program, a network task, a control task, a microphone task, and a speaker task.

The main program is responsible for setting up the circular buffers used for audio recording and playback. It registers the two buttons on the LP device with the Programmable Interrupt Controller and configures the Micro Direct Memory Access controller (uDMA) to transfer audio data from the audio in port to the record buffer in memory and to transfer audio data from the playback buffer in memory to the audio out port. Audio data is constantly being written to the record buffer and read from the playback buffer (when there is data in the playback buffer). After setting up the uDMA, the main program proceeds to create tasks using calls to the Real Time Operating System and finishes by starting the task scheduler. Once the task scheduler starts, the tasks mentioned above begin executing in parallel.

The network task establishes a connection to a wireless access point and creates and configures a UDP socket which is shared between tasks. The network task turns off loopback mode once it establishes a connection.

The control task is responsible for handling interrupts driven by pressing one of the two buttons. The microphone interrupt routine either starts the audio recording or stops the audio recording depending on the state of the microphone at the time the button is pressed. Similarly, the speaker interrupt routine either starts or stops audio playback depending on the state of audio playback when the button is pressed.

The microphone task consists of a loop that waits for audio recording mode to turn on. Once it is on, this routine begins reading audio data from the record buffer and sending it in UDP packets to the EC2 instance.

The speaker task also consists of a loop that waits for audio playback mode to turn on. Once it is on, the speaker task begins receiving UDP packets and writing their data to the audio playback buffer.

Notable files, classes, and functions:

- main.c: contains the main program
  - main(): defines control flow of audio app program
- network.c: contains the network task implementation
  - Network(): defines control flow for network task
  - ConnectToNetwork(): establishes a wireless connection
  - CreateUdpClientAndServer(): configures UDP socket
- control.c: contains the control task implementation
  - MicroPhoneControl(): turns on/off audio recording
  - SpeakerControl(): turns on/off audio playback
- microphone.c: contains the microphone task implementation

    - Microphone(): defines control flow of microphone task
    - sendMessage(): sends a special control message (i.e. begin audio playback, etc...)
- speaker.c: contains the speaker task implementation
    - Speaker(): defines control flow of microphone task
    - receiveMessage(): attempts to get a UDP packet from the UDP socket
    - isMessage(): tries to match a message with a target message
- network.h: defines special messages, packet size, and audio port

Challenges Encountered:

I encountered many challenges in developing this app despite the fact that I began with an example audio app that streams audio in real time from one launch pad (LP) device to another.

The first challenges were installing and getting familiar with the Code Composer Studio (CCS) IDE. There was a bit of a learning curve to figure out where all the compiler and linker options were located and to figure out how to add directories to the include and linking paths and add macro definitions to the compilation process.

Once I got the example programs to compile, I needed to upload the executable to the LP device so it would execute automatically each time the device turns on. This involved placing the flash jumper on the proper pins to put the device into flash mode. Then I could open the Uniflash utility, select the executable compiled by CCS, and write it to serial flash memory. This process is slightly longer than the typical desktop application development cycle where one can almost immediately compile and run small applications.

My original plan was to record the audio message on the LP device and send it to the EC2 instance after saving it to file. The LP device has only 1MB of serial flash memory, however, and that is shared with the executable program so I found the LP device quickly ran out of space in persistent memory. A message of ~8-10 seconds exceeds 1MB of disk storage so I decided to stream it directly to the server.

I ran into two unique problems streaming audio using UDP. First, I needed to create a message system to notify the UDP server when to begin recording/playing audio and when to stop. I created my own messages to accomplish this and created functions on the LP devices and EC2 instance to send and parse these control messages. I defined these messages in the network.h header file and stored a copy of that file on both the LP devices and the EC2 instance to ensure the messages are identical. The second problem I encountered occurred while streaming the audio message from the EC2 instance to the LP device for playback. It seems the LP device is unable to keep up with the pace of UDP packets coming from the EC2 instance and misses a lot of messages since there is no synchronization or receipt confirmation in the User Datagram Protocol. Ideally, this app should use TCP to stream audio data but Chris used TCP to send text messages so I chose to use UDP for audio messages. I found including a delay in the loop that sends audio data from the EC2 instance to the LP device solved the loss of audio problem. I had to experiment with the length of the delay to get it just right. If the delay is too short, the audio playback skips almost like it’s playing in fast-forward and if the delay is too long there are periodic audio gaps in the playback. Similarly, I had to experiment with the size of the UDP packet. If the packets are too small, the LP device cannot keep the play buffer full so the audio also has gaps in its playback and if the packets are too large it takes too long for the LP device to copy audio data from the incoming datagram buffer to the audio playback buffer. 1024 bytes per packet seems to work consistently well even for long audio messages.