



MODULE 2 BOOT WINDOWS 10 IOT

Q 1. Set-up Windows 10 IOT

Windows 10 IOT core OS was successfully set up on Raspberry Pi3 using windows 10 IoT Core dashboard as seen in figure(1). Figure(2) shows the windows 10 IoT Device Portal showing the connected device. Figure(3) shows the home page of windows 10 IoT core after successfully setting up on Raspberry Pi 3.

The screenshot shows the Windows 10 IoT Core Dashboard. On the left, there's a sidebar with options like 'My devices' (which is selected), 'Set up a new device', 'Connect to Azure', and 'Try some samples'. Below the sidebar, there are links for 'Sign in' and 'Settings', and a weather widget showing '46°F Clear'. The main area is titled 'My devices' and lists one device: 'MESA-rpi3' (Type: Raspberry Pi 3). It shows the IP address (10.0.17763.107), IPv4 Address (10.0.0.178), and IPv6 Address (fe80::dc2bcf2d:7534:27c8). A search bar is at the top right of the main table area.

Figure 1 : Windows 10 IoT Core Dashboard

The screenshot shows the Windows 10 IoT Core Device Portal. The URL is 10.0.0.178:8080/#Device%20Settings. The left sidebar has sections like 'Device Settings' (selected), 'Apps', 'Azure Clients', 'Processes', 'Debug', 'Devices', 'Connectivity', 'TPM Configuration', 'Windows Update', 'Remote', 'Scratch', and 'Tutorial'. The main content area is titled 'Device Settings' and includes fields for 'Change your device name' (New device name: 'New device name'), 'Change your password' (Old password, New password, Confirm password), and a 'Save' button. To the right, there's an 'Audio Control' section with an 'Audio Devices' list (refreshed) and a 'Screenshot' section with 'Capture' and 'Download' buttons. The device name 'MESA-rpi3' and 'Raspberry Pi 3' is displayed in the top right corner.

Figure 2 : Windows 10 IoT Core Device Portal

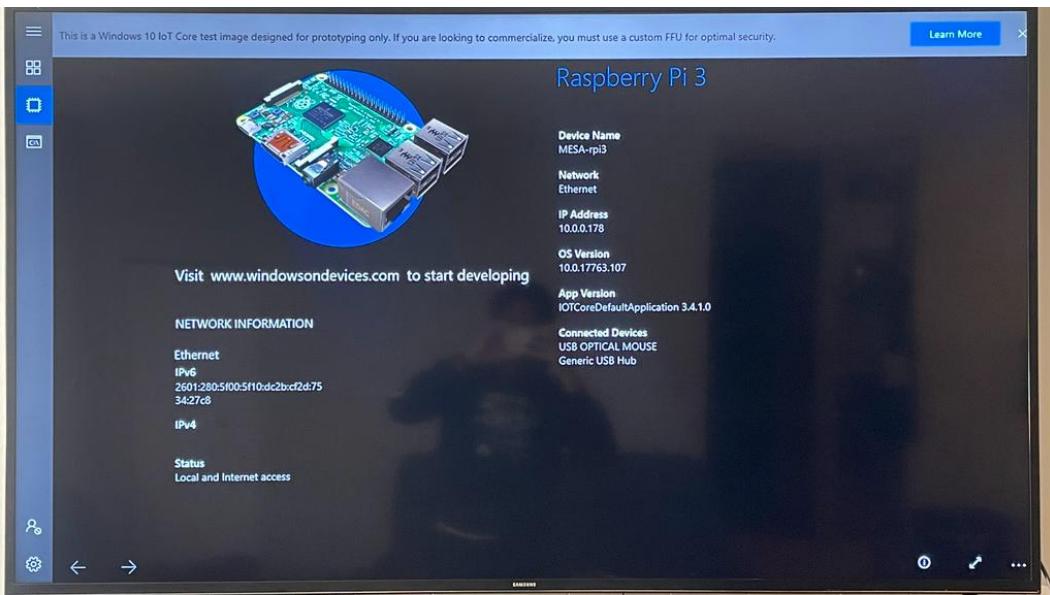


Figure 3 : Windows 10 IoT Core Home Window after Set-up

Q.2. Upon booting up, the serial port should be sending out debug messages. Open a terminal window to capture them. What do you see? Be aware that the port used to transmit the information may be different from what you expect?

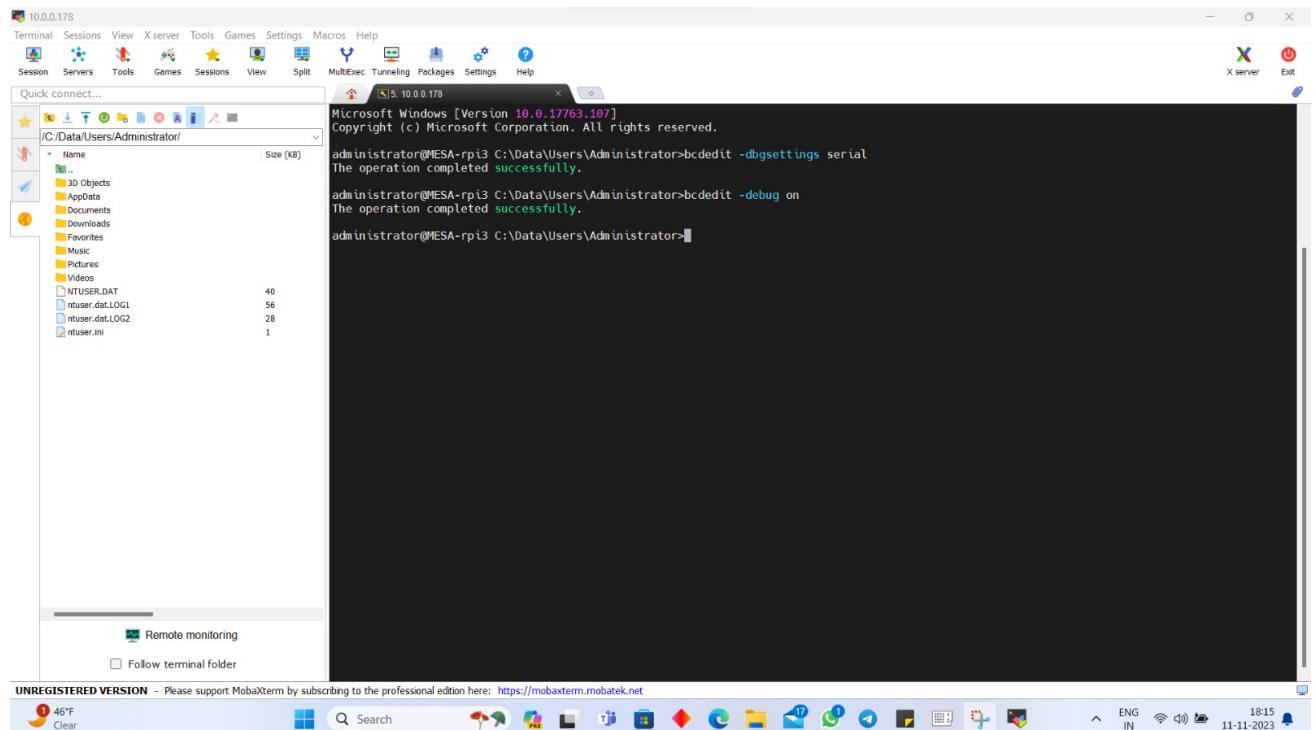
The FTDI USB to Serial adapter(3.3V) was connected to raspberry PI 3 as seen in figure(4). The connection was setup as follows:

[RPi3] Pin #6 (GND) <-> [FTDI] Black (GND)
[RPi3] Pin #8 (TX) <-> [FTDI] Yellow (RX)
[RPi3] Pin #10 (RX) <-> [FTDI] Orange (TX)



Figure 4 : FTDI Connections

Serial debugging was enabled using command seen in figure(5) below.



```

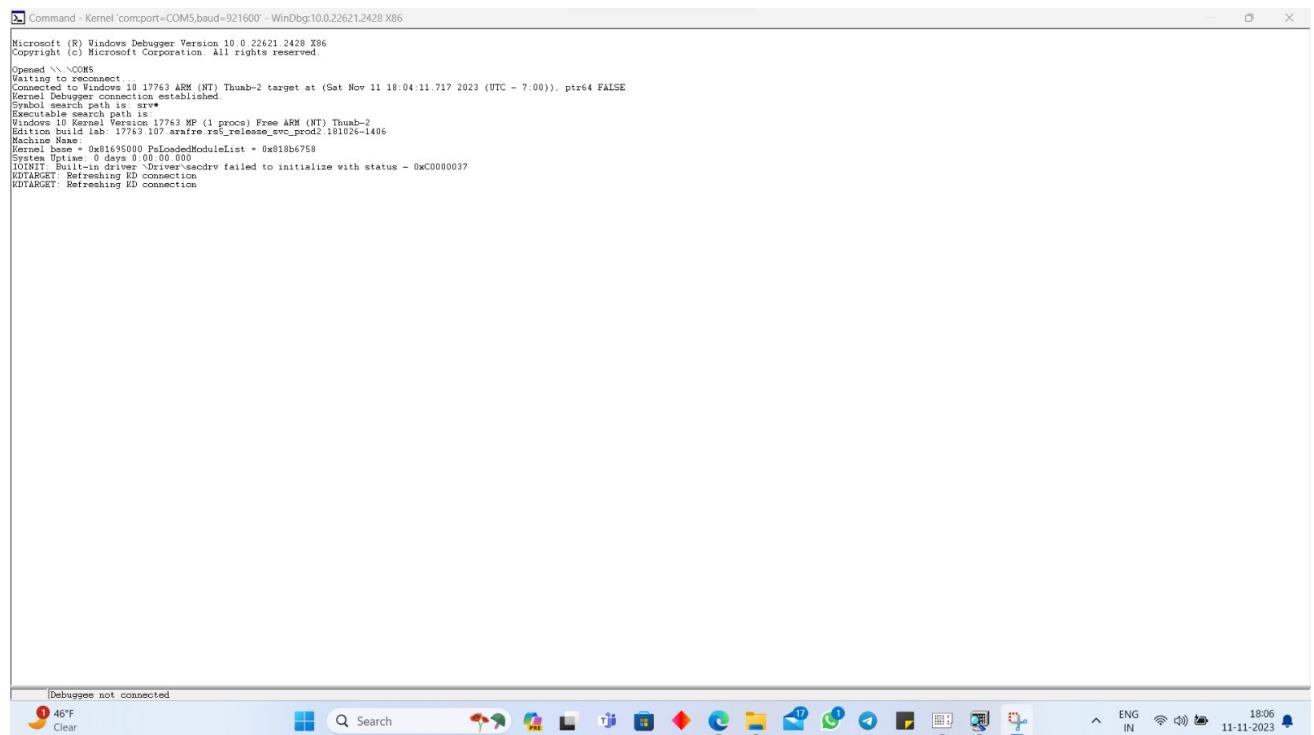
Administrator@MESA-rpi3:~$ bcdedit -dbgsettings serial
The operation completed successfully.

Administrator@MESA-rpi3:~$ bcdedit -debug on
The operation completed successfully.

Administrator@MESA-rpi3:~$ 
    
```

Figure 5 : Terminal Window Showing Commands to Enable Debugging

The figure(6) and (7) shows the screenshot from Windows Debugger(windbg) -> Kernal Debugger during initial boot and after reboot respectively.



```

Microsoft (R) Windows Debugger Version 10.0.22621.2428 X86
Copyright (C) Microsoft Corporation. All rights reserved.

Opened \\.\COM5
Waiting to reconnect
Connected to Windows 10 17763 ARM (NT) Thumb-2 target at (Sat Nov 11 18:04:11.717 2023 (UTC - 7:00)). ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is: srv*
Executable search path is:
Used module path is: 17763 MP (1 proc) Free ARM (NT) Thumb-2
Edition build lab: 17763.107 arafre.res_release_svc_prvcd2 181026-1406
Machine Name: 0xb1695000 PslLoadedModuleList = 0x818b6758
Kernel Name: 0xb1695000
System Uptime: 0 days 0:00:00.000
IOBUS: Main driver - \Device\saodrv failed to initialize with status - 0xC0000037
KDTARGET: Refreshing ED connection
EDTARGET: Refreshing ED connection
    
```

Figure 6 : Kernel Debug During Initial Boot

```
Command - Kernel 'com:port=COM5,baud=921600' - WinDbg:10.0.22621.2428 X86

Microsoft (R) Windows Debugger Version 10.0.22621.2428 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\COM5
Waiting to reconnect...
Connected to Windows 10 17763 ARM (NT) Thub-2 target at (Sat Nov 11 18:07:22.940 2023 (UTC - 7:00)). ptr64 FALSE
Processor Depth: connection established
Symbol search path is: srv*
Windows 10 Kernel Version 17763 MP (4 procs) Free ARM (NT) Thub-2
Product: Windk_ suite: TerminalServer SingleSheetTS
Edition build lab: 17763.107.smrfe.rs5_release_svc_prod2.181026-1406
Machine Name:
Kernel base = 0x01600000 PsLoadedModuleList = 0x818b6758
System Uptime: 0 days 0:02:18.352
Shutdown occurred at: (Sat Nov 11 18:07:33.180 2023 (UTC - 7:00)).. unloading all symbol tables.

Connected to Windows 10 17763 ARM (NT) Thub-2 target at (Sat Nov 11 18:07:40.825 2023 (UTC - 7:00)). ptr64 FALSE
Kernel Debugger connection established

***** Path validation summary *****

Time (ms) Location
Described srv*
Symbol search path is: srv*
Symbol search path is: srv*
Windows 10 Kernel Version 17763 MP (1 procs) Free ARM (NT) Thub-2
Edition build lab: 17763.107.smrfe.rs5_release_svc_prod2.181026-1406
Machine Name:
Kernel base = 0x00c1e000 PsLoadedModuleList = 0x00e3f1758
System Uptime: 0 days 0:00:00.000
KDDIAGNOSTIC: 1-in driver - Driver\secdrv failed to initialize with status - 0xC0000037
KDTARGET: Refreshing RD connection
KDTARGET: Refreshing RD connection

Debuggee is running...
```

Figure 7 : Kernel Debug After Initial Boot

The image shown in figure(7) displays a session of the Windows Debugger (windbg) engaged with a Windows 10 IoT Core system on ARM architecture. The WinDbg version is 10.0.22621.24028 x86, and it shows multiple attempts to establish a connection, with a successful connection logged on November 11th. The system's kernel version is 17763 for an ARM processor with Free Build and Thumb-2 instruction set, indicating it's designed for non-commercial use with advanced CPU instruction capabilities. The debugger's symbol path is set to a server, suggesting it's using remote symbols for debugging. An error message reveals a driver initialization failure with a specific status code. The machine's name is not displayed, instead represented by a base memory address, and the system uptime is minimal, indicating a recent reboot or start-up. The output includes detailed timestamps, reflecting the debugger's real-time tracking of the session's progress.



Q.3. How much memory is used by the code? (What is the image size?)

The image size of 1.46 GB calculated from the command "**wmic logicaldisk**" as seen in the figure(8). Refer the calculation below for the image size calculated.

$$(1498411008 - 596844544) + (30337908736 - 29773643776) = 1,46,58,31,424 \text{ Bytes} = 1.46 \text{ GB}$$

```
Administrator@MESA-rl3:~ C:\Data\Users\Administrator> wmic logicaldisk get devicelid,volumename,description,size,freespace
Description
DeviceID
FreeSpace
Size
VolumeName
Administrator@MESA-rl3:~ C:\Data\Users\Administrator>
```

Figure 8 : Windows 10 IoT Core Image Size

Q.4. Capture a screen shot of the terminal window.

Refer the screenshot of terminal windows opened using SSH connection and HDMI as seen in figure (9) and (10) respectively.

```
Administrator@MESA-rl3:~ C:\Data\Users\Administrator> wmic logicaldisk get devicelid,volumename,description,size,freespace
Description
DeviceID
FreeSpace
Size
VolumeName
Administrator@MESA-rl3:~ C:\Data\Users\Administrator>
```

Figure 9 : Command Terminal Using SSH

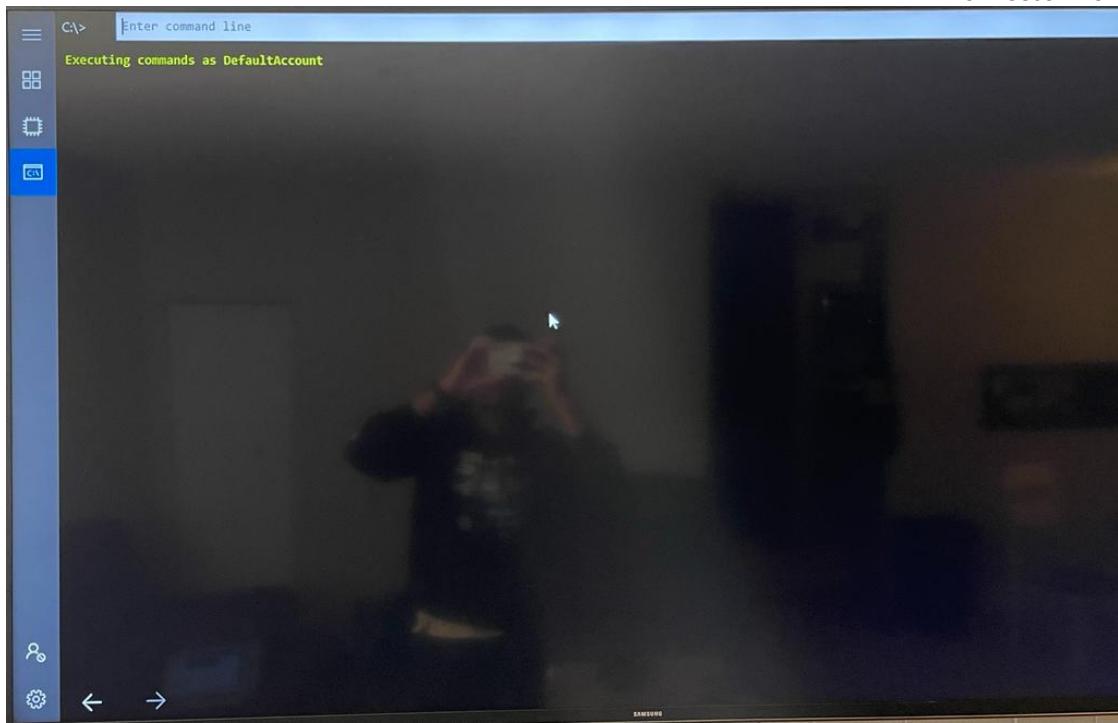


Figure 10 : Command Terminal On Raspberry PI without SSH using HDMI

Q.5. The Ethernet and HDMI ports should also be active. Connect the HDMI output to a monitor using a HDMI Cable and adapter if necessary, or connect using SSH to see the GUI window. Reboot the system – what do you see?

After initial boot setting using HDMI & Ethernet, we see the windows 10 IoT Core home page as seen in figure(11).

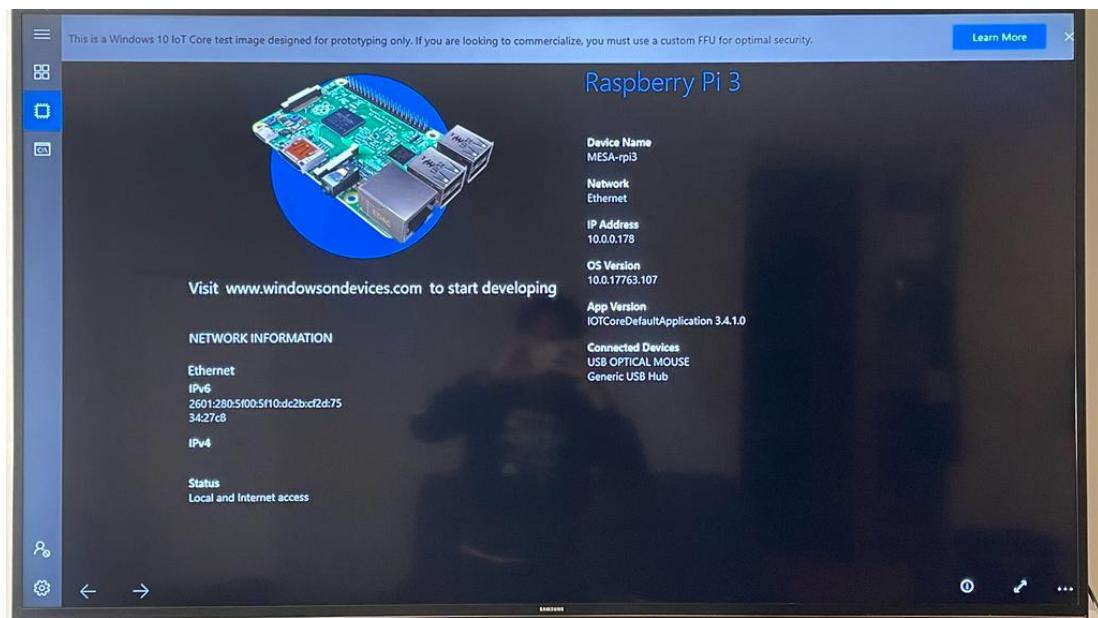


Figure 11 : Windows 10 IoT Core Home Page



Q.6. Write C code for a G.711 coder/decoder. See http://www.opensource.apple.com/source/tcl/tcl-20/tcl_ext/snack/snack/generic/g711.c or for an example. Use this decoder to decode a file given to you by your instructor. You will need to use Visual Studio 15 or later to compile code for this application and then run it on the target board. Alternatively, you could also do this in Linux using gcc on the target board?

In our case, we utilized the decoder to process two files: **1449183601-A_eng_f2.wav** and **Au8A_eng_f2.wav**. The decoding was successful, and we obtained **pcm_decoded_output_of_mulaw.wav** and **pcm_decoded_output_of_mulaw_2.wav** as the decoded outputs.

The textual content retrieved from the decoded audio is as follows:

"The ship was torn apart on the sharp reef."
"Sickness kept him home the third week."
"The box will hold seven gifts at once."
"Jazz and swing fans like fast music."

Each decoded file has a size of 185 KB, aligning with expectations considering G.711's encoding scheme. Since the original encoding uses 8 bits per sample and the decoding process yields 16 bits per sample, the decoded file size is naturally twice that of the encoded one. Note that G.711 is not lossless; therefore, a slight discrepancy in data size (approximately 184 KB) is noted, which is attributable to the nature of the compression algorithm.

Q.7. Record your observations. How is the behaviour of Windows 10 IoT different from Linux?

Based on our observations through the screenshots shared:

- We note that Windows 10 IoT presents a more graphical user interface that is similar to the traditional Windows desktop environment, whereas Raspberry Pi OS, which is a Linux distribution tailored for the Raspberry Pi, often emphasizes a command-line interface for operations, particularly in IoT or embedded systems.
- We observe that Windows 10 IoT uses Windows Management Instrumentation (WMI) for system management, a framework unique to Windows. In contrast, in Raspberry Pi OS, we would typically use various shell scripts and native Linux commands for this purpose.
- From the Windows Debugger output, we discern that Windows 10 IoT follows a different method for troubleshooting and system feedback compared to the hands-on, text-based diagnosis process we are accustomed to with Raspberry Pi OS.
- The file system structure and administrative paths shown in the screenshots indicate a significant difference in system organization between Windows 10 IoT and Raspberry Pi OS.
- We also note the smaller size of the Windows 10 IoT installation compared to Raspberry Pi OS. This is due to Windows 10 IoT Core being a more streamlined version of Windows with fewer pre-installed applications, as opposed to Raspberry Pi OS which typically comes with a set of applications included.

- Additionally, Windows 10 IoT Core features a device portal for device management, which we found to be absent in Raspberry Pi OS. Fedora's version for Raspberry Pi is an exception as it offers a similar feature, but generally, this is a distinctive aspect of Windows 10 IoT Core that enhances its manageability.

Code :

g711_Decoder.c

```
/*
% -----
% Title: G.711 E-law Audio Decoder
% Brief: This file contains the implementation of a G.711 E-law audio decoder.
%         It reads a E-law encoded WAV file, decodes it to linear PCM format,
%         and writes the PCM data to a new WAV file.
% Author: Kiran Jojare & Viraj Patel
% Course: ECEN 5803 Mastering Embedded System Architecture
% Project: Project 2 - Module 2
% Date: Nov 8 2023
%
% References:
% - G.711 E-law and A-law implementations in C:
%   https://opensource.apple.com/source/tcl/tcl-20/tcl_ext/snack/snack/generic/g711.c
% - G.711 Wikipedia Article:
%   https://en.wikipedia.org/wiki/G.711
% - WAV file format tutorial:
%   http://www.topherlee.com/software/pcm-tut-wavformat.html
% -----
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

// Define constants used for E-law encoding and decoding
#define SIGN_BIT      (0x80)      // Sign bit for a A-law byte, used for encoding.
#define QUANT_MASK    (0xf)       // Mask to extract the quantization bits.
#define SEG_SHIFT     (4)         // Number of bits to left shift for segment number.
#define BIAS          (0x84)       // Bias for linear code.
#define SEG_MASK      (0x70)       // Mask to extract the segment number.
#define CLIP          32635        // Clipping value for the magnitude of the signal.

// Function to convert a E-law value to 16-bit linear PCM
short Snack_Mulaw2Lin(unsigned char u_val) {
    short t;
    u_val = ~u_val; // Complement the E-law value to get the original signal value.
    t = ((u_val & QUANT_MASK) << 3) + BIAS; // Extract and scale the quantization bits.
    t <= ((unsigned)u_val & SEG_MASK) >> SEG_SHIFT; // Shift the value based on the segment.
    return ((u_val & SIGN_BIT) ? (BIAS - t) : (t - BIAS)); // Return the final PCM value.
}

// Define a standard header for the E-law WAV format
uint8_t header[] = {
    // The 'RIFF' chunk descriptor
    'R', 'I', 'F', 'F', // ChunkID
    0x00, 0x00, 0x00, 0x00, // ChunkSize (placeholder, will be updated later)
    'W', 'A', 'V', 'E', // Format
    // The 'fmt' sub-chunk (format information)
    'f', 'm', 't', ' ', // Subchunk1ID
    0x10, 0x00, 0x00, 0x00, // Subchunk1Size (16 for PCM)
    0x01, 0x00, // AudioFormat (PCM)
    0x01, 0x00, // NumChannels (1 channel)
    0x40, 0x1f, 0x00, 0x00, // SampleRate (8000 Hz)
    0x80, 0x3e, 0x00, 0x00, // ByteRate (SampleRate * NumChannels * BitsPerSample/8)
    0x02, 0x00, // BlockAlign (NumChannels * BitsPerSample/8)
    0x10, 0x00, // BitsPerSample (16 bits)
    // The 'data' sub-chunk (actual sound data)
}
```

```

'd', 'a', 't', 'a',      // Subchunk2ID
0x00, 0x00, 0x00, 0x00 // Subchunk2Size (placeholder, will be updated later)
};

int main() {
    // Specify the path to the input and output files
    const char* inputFilePath = "Au8A_eng_f2.wav";
    const char* outputFilePath = "pcm_decoded_output_of_mulaw.wav";

    // Attempt to open the input file for reading
    FILE *inputFilePointer = fopen(inputFilePath, "rb");
    if (inputFilePointer == NULL) {
        fprintf(stderr, "Could not open input file %s\n", inputFilePath);
        return 1;
    }

    // Attempt to open the output file for writing
    FILE *outputFilePointer = fopen(outputFilePath, "wb+");
    if (outputFilePointer == NULL) {
        fprintf(stderr, "Could not open output file %s\n", outputPath);
        fclose(inputFilePointer);
        return 1;
    }

    // Write the placeholder header to the output file
    fwrite(header, sizeof(header), 1, outputFilePointer);

    // Prepare to read and decode the G^-law data
    uint8_t buffer;
    uint16_t output;
    size_t dataChunkSize = 0;

    // Skip the header of the input file as it's not needed for decoding
    fseek(inputFilePointer, 44, SEEK_SET);

    // Decode each G^-law byte and write the output as PCM data
    while (fread(&buffer, sizeof(buffer), 1, inputFilePointer) == 1) {
        output = Snack_Mulaw2Lin(buffer);
        fwrite(&output, sizeof(output), 1, outputFilePointer);
        dataChunkSize += sizeof(output);
    }

    // Update the sizes in the header with actual data size
    uint32_t riffChunkSize = dataChunkSize + 36; // Calculate RIFF chunk size
    fseek(outputFilePointer, 4, SEEK_SET); // Move to the RIFF chunk size position
    fwrite(&riffChunkSize, sizeof(riffChunkSize), 1, outputFilePointer);

    // Move to the 'data' subchunk size position and write it
    fseek(outputFilePointer, 40, SEEK_SET);
    fwrite(&dataChunkSize, sizeof(dataChunkSize), 1, outputFilePointer);

    // Close the input and output files
    fclose(inputFilePointer);
    fclose(outputFilePointer);

    // Indicate successful decoding
    printf("Decoding complete.\n");

    return 0;
}

```