PROJECT 2

MODULE 2

BY

SHRUTHI THALLAPALLY

THARUNI GELLI

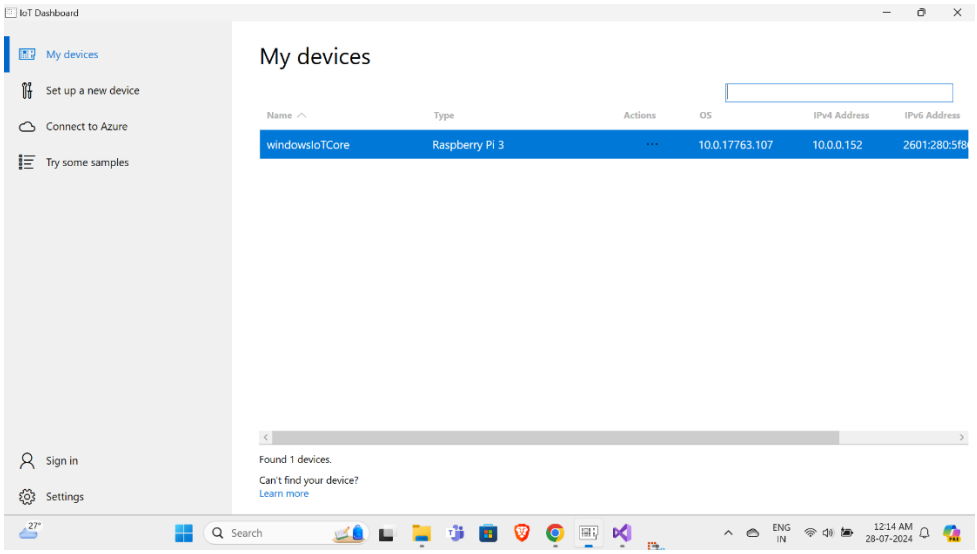
AYSVARYA GOPINATH

II. MODULE 2 BOOT WINDOWS 10 IOT, CREATE G.711 CODEC

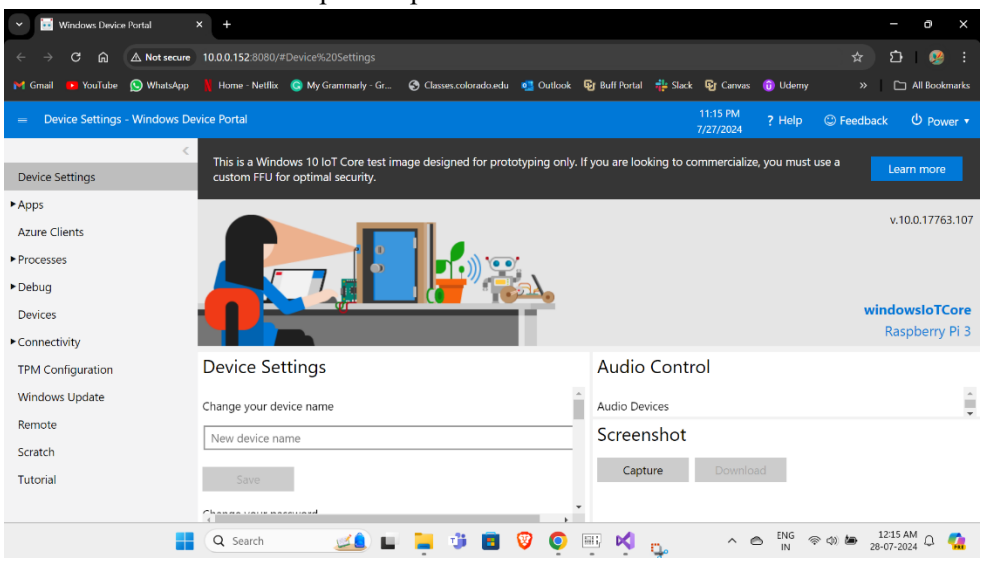
1. SETUP WINDOWS 10 IOT CORE

Windows 10 IoT core OS is successfully set up on Raspberry Pi 3 model B board using Windows IoT core Dashboard.

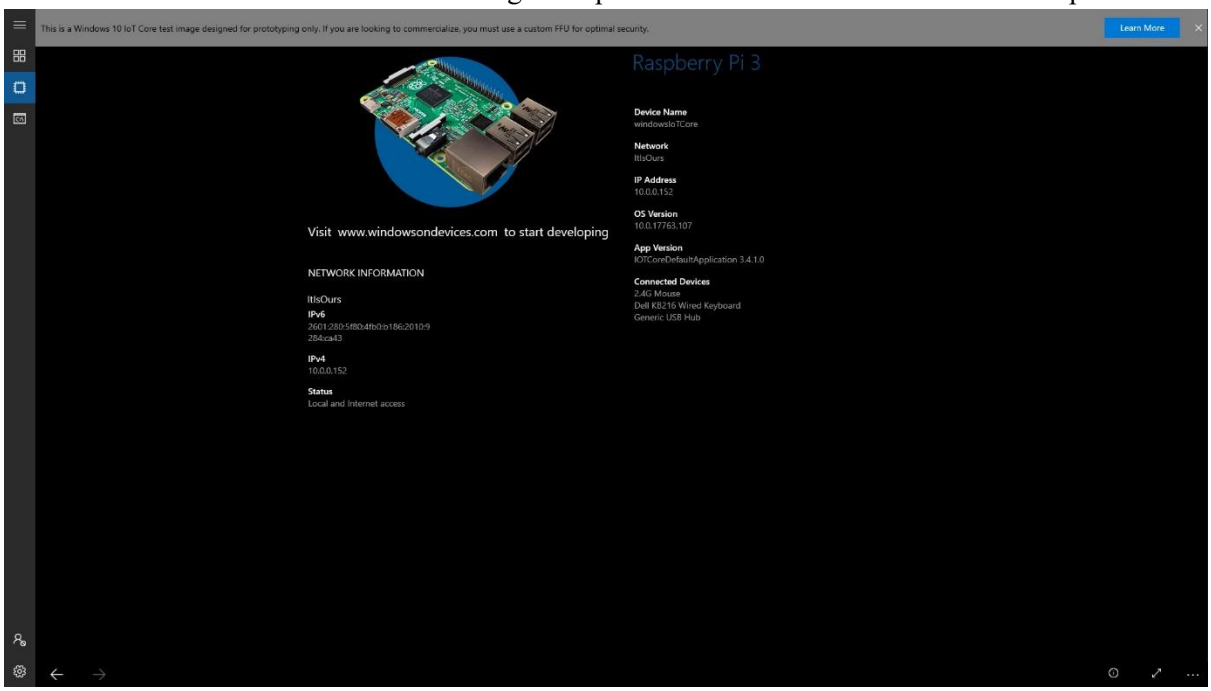
Windows IoT Dashboard:

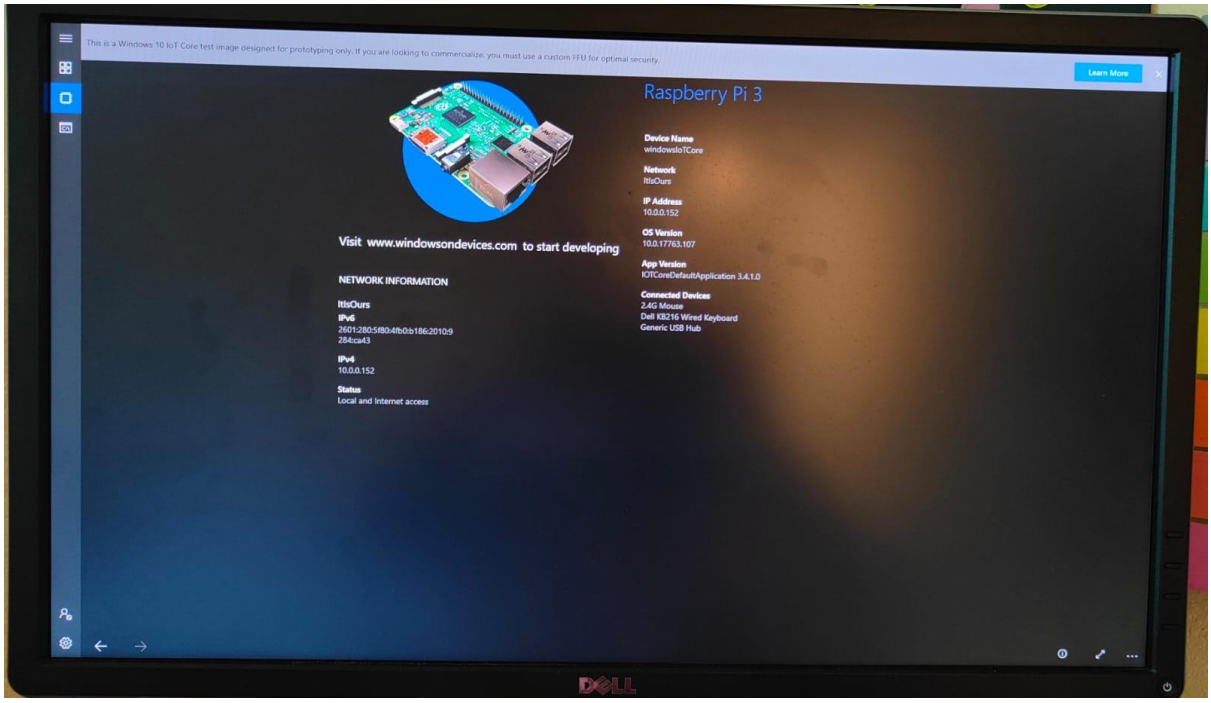


Windows IoT core Device portal opened from the dashboard:



Windows IoT core Home Screen: This image is captured and downloaded from the device portal.



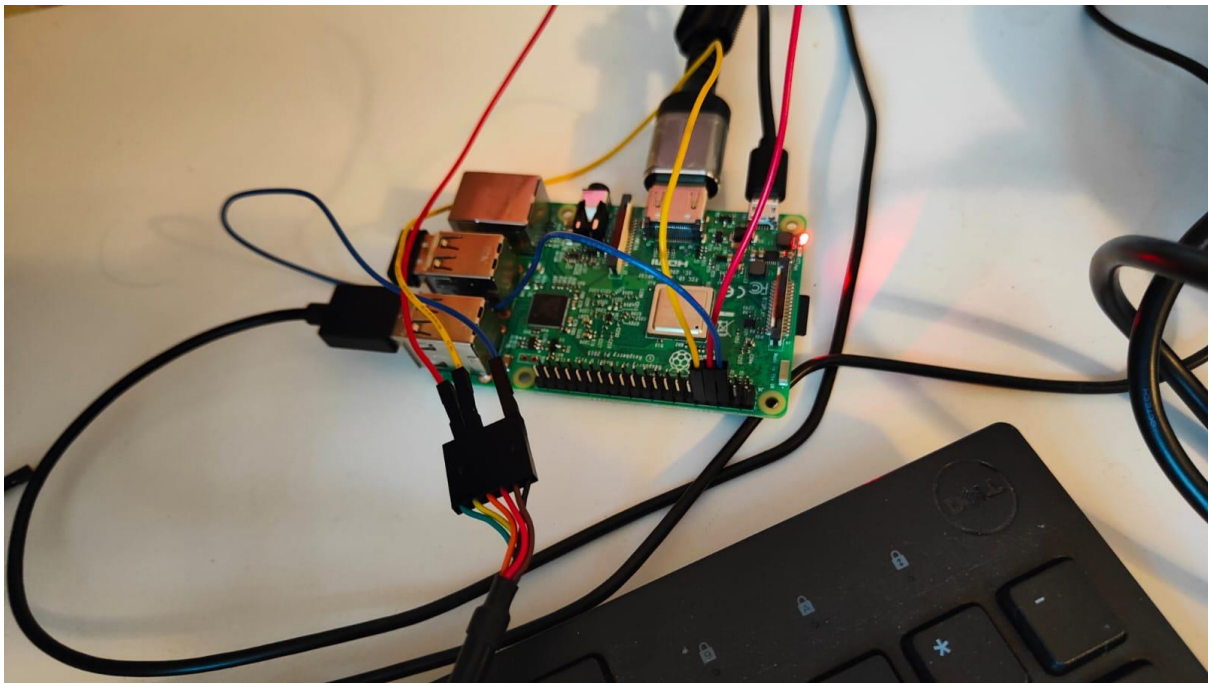


2. Upon booting up, the serial port should be sending out debug messages. Open a terminal window to capture them. What do you see? Be aware that the port used to transmit the information may be different from what you expect. You will need to do some research to determine how to connect to it.

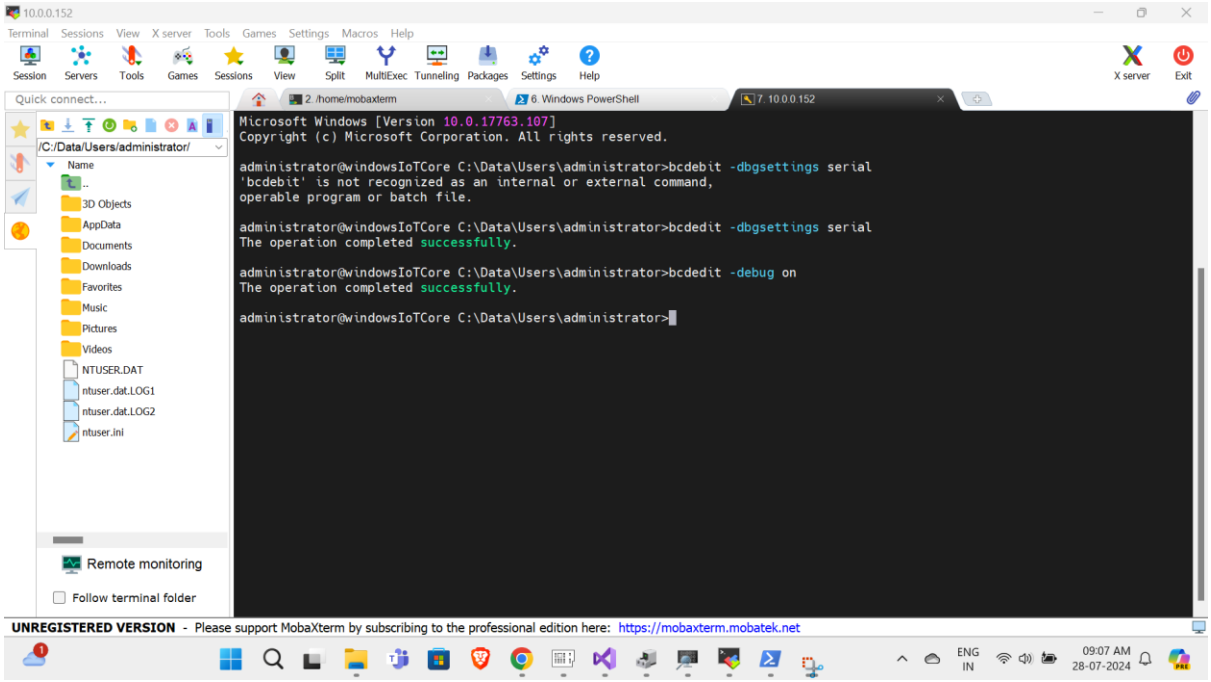
FTDI cable is used to capture the debug messages from rpi board. The connections hould be as follows:
FTDI GND (black) – RPI GND

FTDI TX (orange) - RPI RX

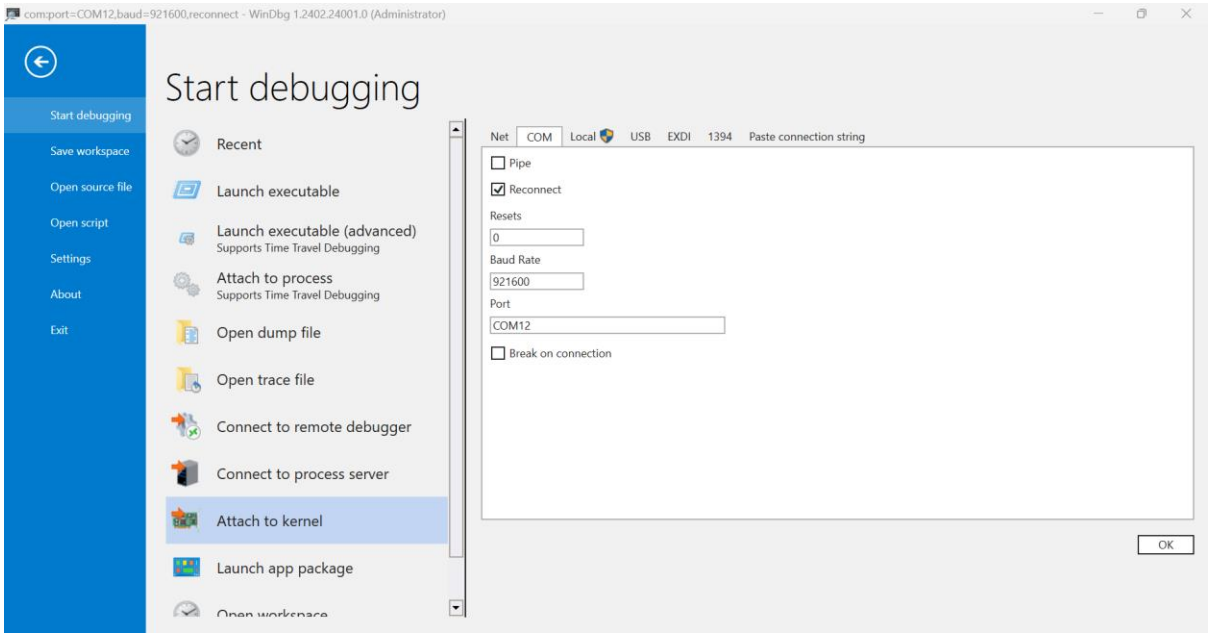
FTDI RX (yellow) - RPI TX

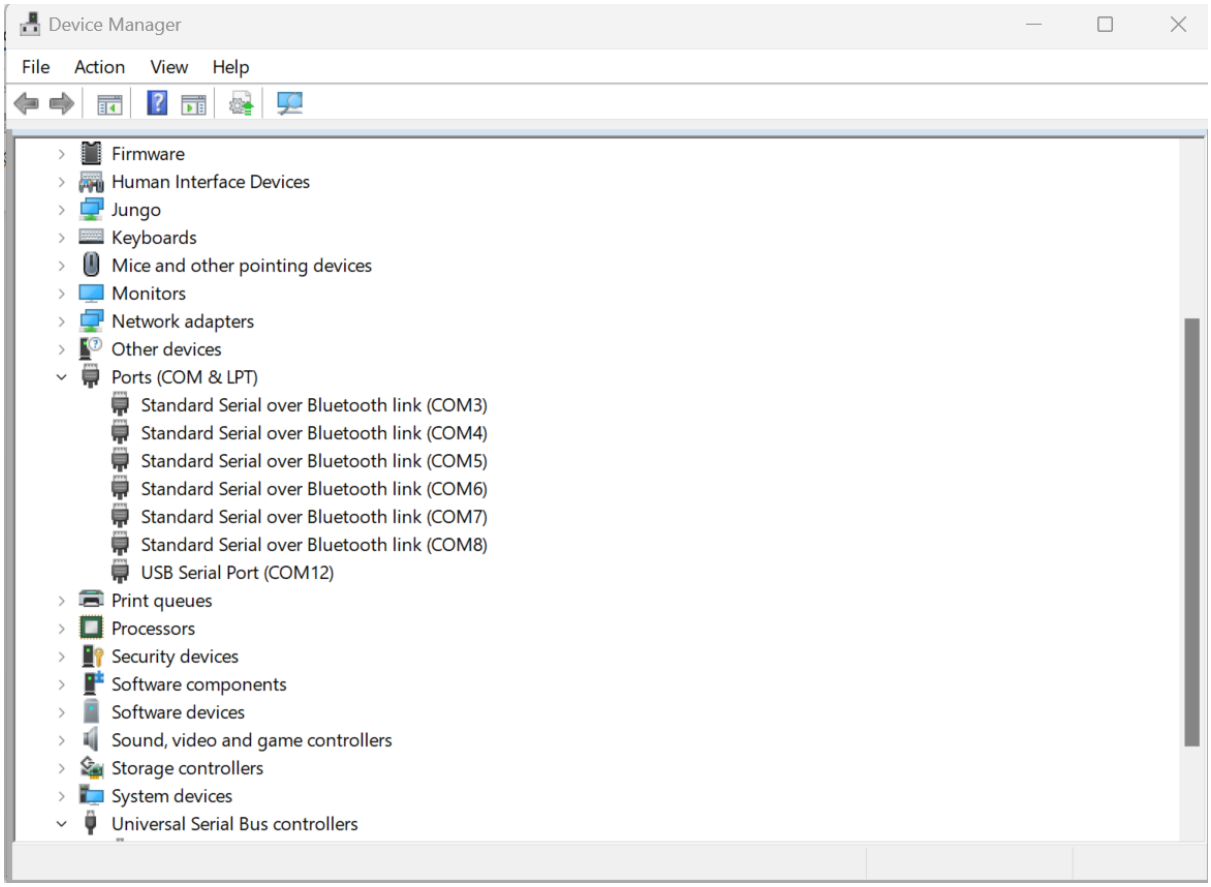


The serial debugging needs to be turned on the RPI. To do this, we used MobaXTerm. Using MobaXTerm, we can start an SSH session of RPI on our PC. The SSH session works remotely from the PC on Rpi as long as both devices are on the same network. Serial debugging is turned on the Rpi by using the below commands.

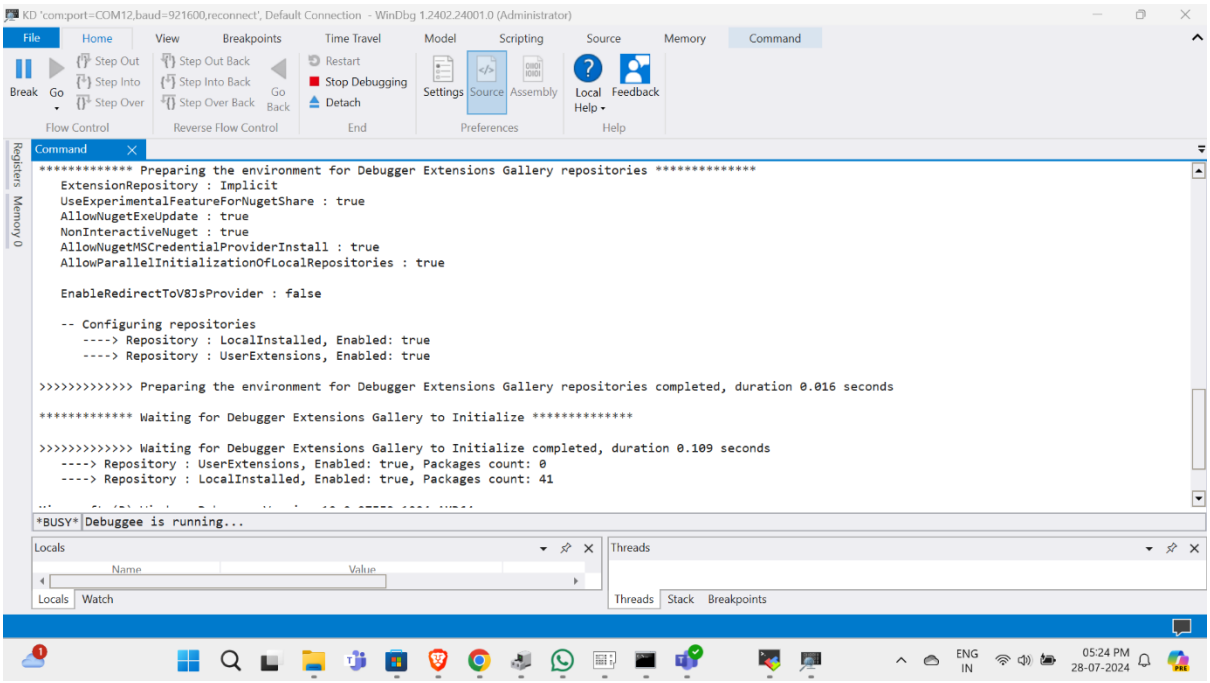


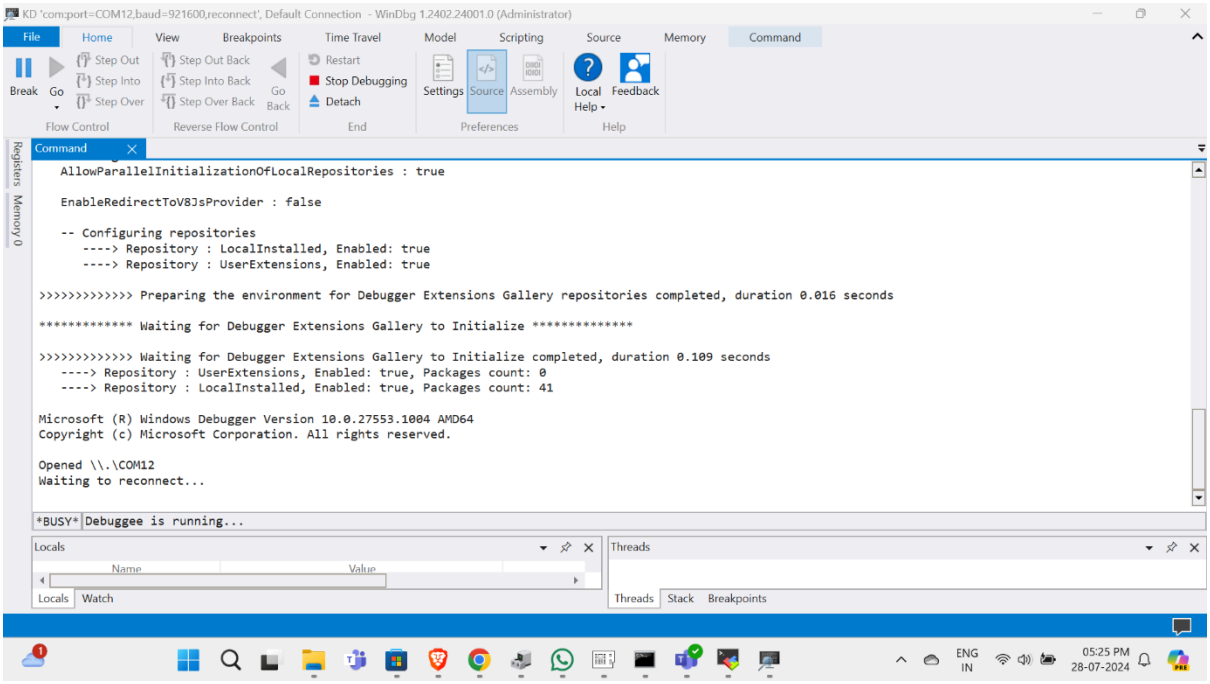
We can then see debug messages on Windbg (Windows Debugger) by selecting the port and baud rate.





After clicking on “OK”. The command terminal opens, and we should be able to see the debug messages.





We attempted multiple times and tried all the ways to get the debug messages but failed to get them. Debugging was done to check where we were going wrong and did a loopback test to check the functionality of the FTDI cable. TX of FTDI is connected to RX of FTDI. When a serial terminal is opened and the baud rate is set, we should be able to see the echo of characters typed on our PC. We didn’t get any echo on the terminal. We tried multiple baud rates, with no success. We are suspecting some issue with the FTDI cable behind this.

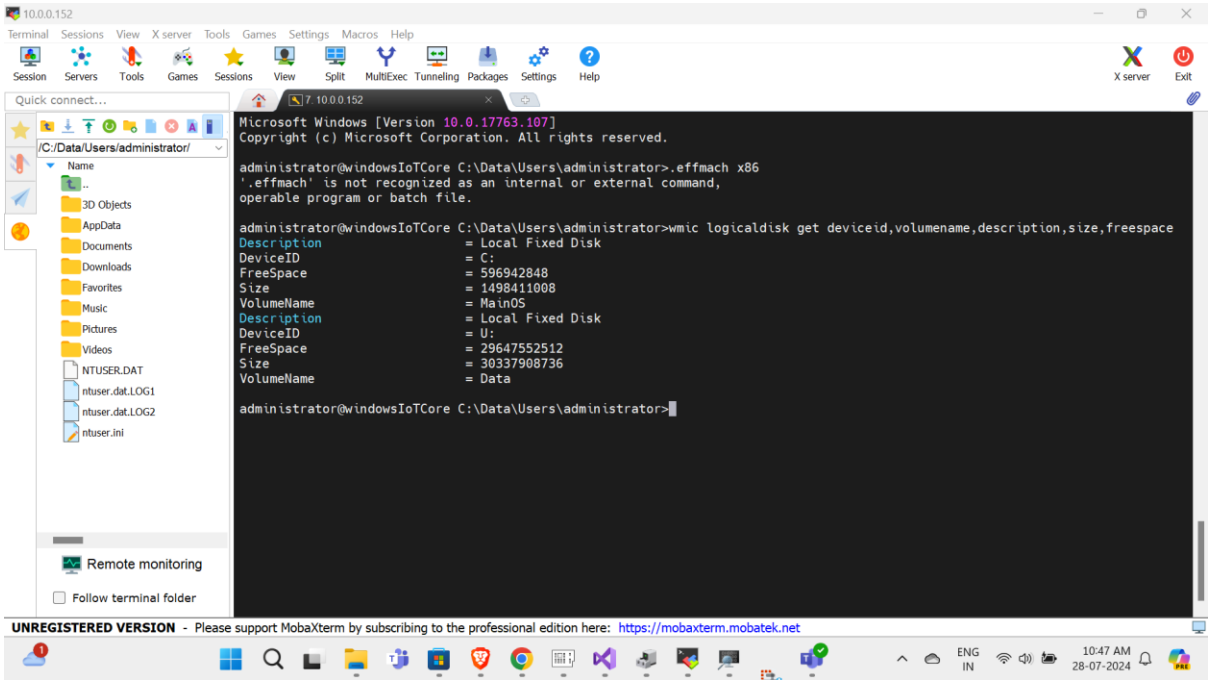
3. How much memory is used by the code? (What is the image size?)

The memory information can be obtained by executing the command “wmic logicaldisk”. The memory used by the code is 1.59 GB.

Obtained by subtracting the free space from size.

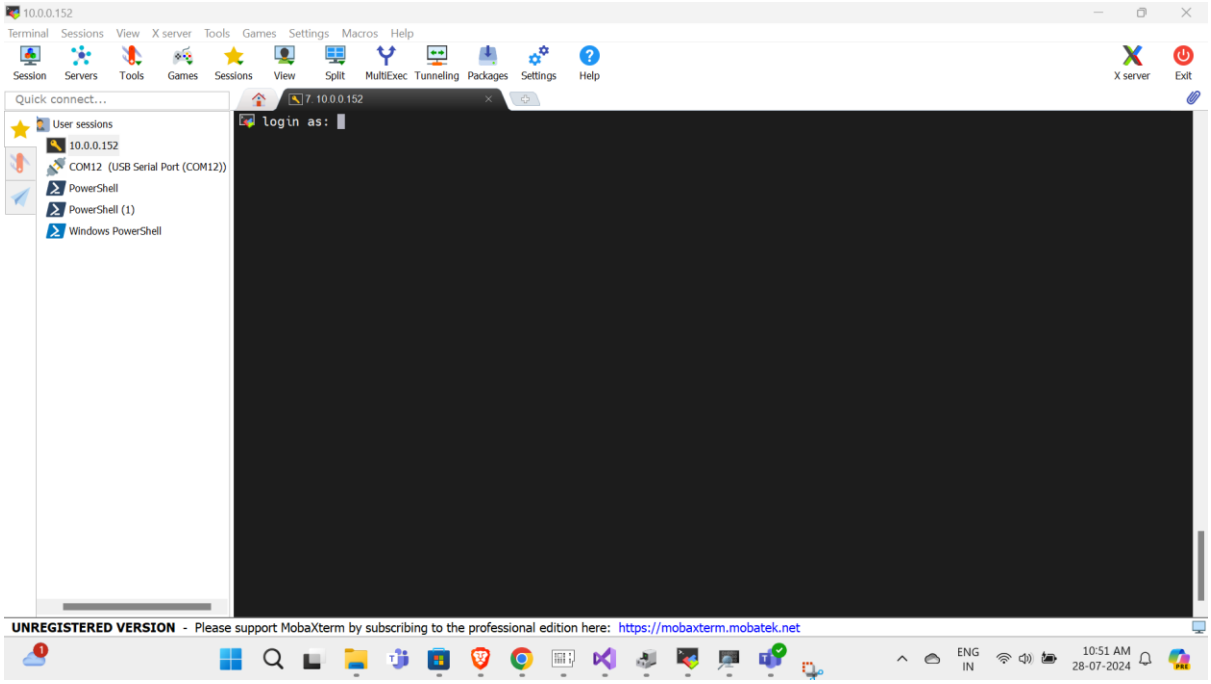
(1498411008-596942848) +(30337908736-29647552512)

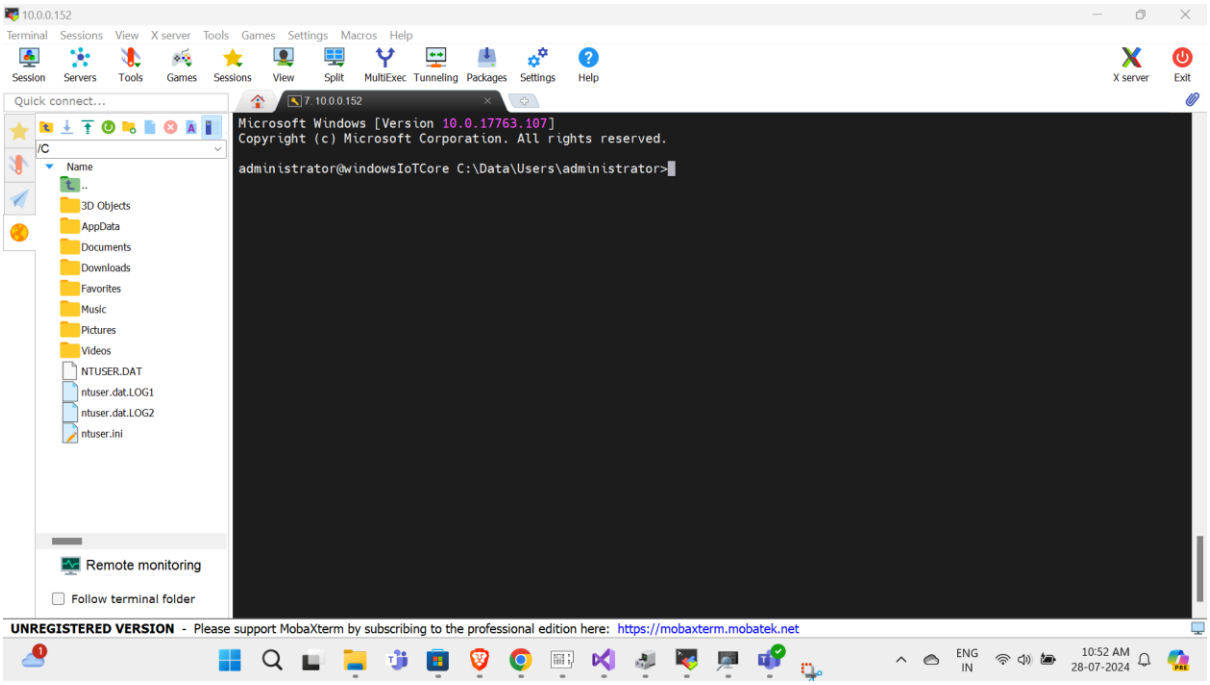
=1591824384 bytes = 1.59GB



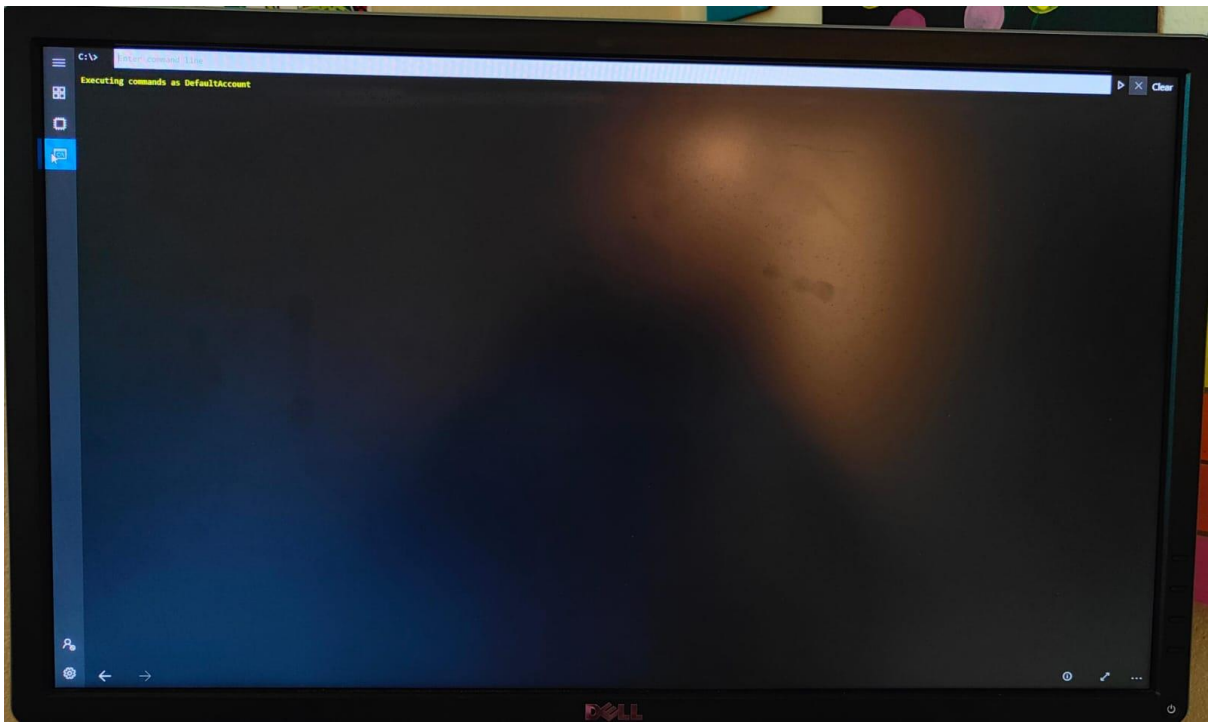
4. Capture a screenshot of the terminal window.

The SSH terminal window:



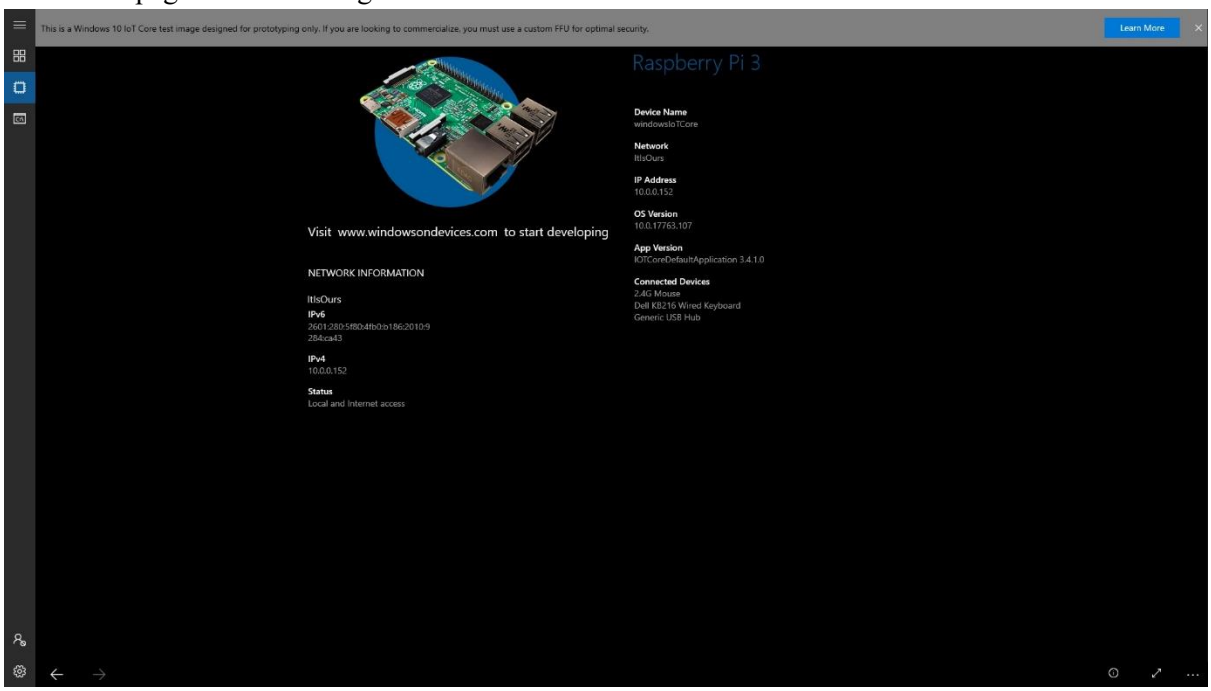


The command line on RPI board:



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?

The Homepage after rebooting the RPI board.



6. Write C code for a G.711 coder/decoder. See http://www.opensource.apple.com/source/tcl/tcl20/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.

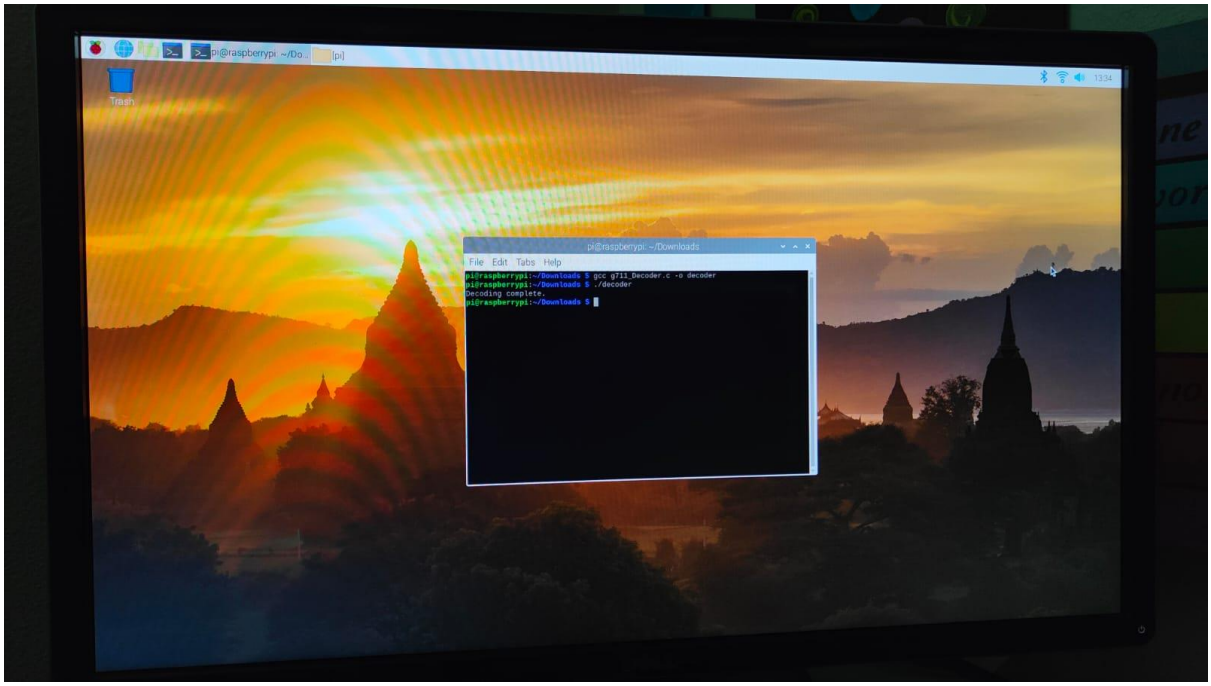
The decoder program was run on Linux (Raspbian) on the files obtained from the professor. The decoded .wav is saved as *decoder_output.wav.* The sentences on the decoded output file are

"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."



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

The major differences between Windows 10 IoT and Linux are

- Windows 10 IoT GUI resembles the PC having many similarities with Windows. Raspberry Pi OS, which is Linux, is specially customized for Raspberry Pi operations.
- The systems path is different in Windows IoT and Linux which indicates the difference in system hierarchy.
- Windows 10 IoT image size is smaller than the Linux version. This is because Windows 10 IoT is a condensed version of Windows for IoT applications whereas the Linux version of RPi has some pre-installed set of applications.
- Windows has a device portal feature for device management which is absent in the Linux version.

Appendix:

g711_Decoder.c

```
/* References:
   - https://github.com/dystopiancode/pcm-g711/tree/master
   - https://en.wikipedia.org/wiki/G.711
   - https://dystopiancode.blogspot.com/2012/02/pcm-law-and-u-law-companding-
algorithms.html
*/

// Include necessary libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
```

```
// Define constants used for μ-law encoding and decoding
#define SIGN_BIT    (0x80)    // Sign bit for μ-law byte, used for encoding
#define QUANT_MASK  (0x0F)    // 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: MuLawToLinearPCM
 * --------------------------
 * Converts a μ-law encoded byte to a 16-bit linear PCM value.
 *
 * Parameters:
 *   muLawValue - The μ-law encoded byte to be converted.
 *
 * Returns:
 *   A 16-bit linear PCM value.
 */
short MuLawToLinearPCM(unsigned char muLawValue) {
    short t;
    muLawValue = ~muLawValue;  // Complement the μ-law value to get the
original signal value
    t = ((muLawValue & QUANT_MASK) << 3) + BIAS;  // Extract and scale the
quantization bits
    t <<= ((unsigned)muLawValue & SEG_MASK) >> SEG_SHIFT;  // Shift the value
based on the segment
    return ((muLawValue & SIGN_BIT) ? (BIAS - t) : (t - BIAS));  // Return the
final PCM value
}

// Define a standard header for the μ-law WAV format
uint8_t wavHeader[] = {
    // 'RIFF' chunk descriptor
    'R', 'I', 'F', 'F',           // ChunkID
    0x00, 0x00, 0x00, 0x00,       // ChunkSize (placeholder, will be updated
later)
    'W', 'A', 'V', 'E',           // Format

    // '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)

    // 'data' sub-chunk (actual sound data)
    'd', 'a', 't', 'a',           // Subchunk2ID
    0x00, 0x00, 0x00, 0x00        // Subchunk2Size (placeholder, will be
updated later)
};

/**
 * Function: main
 * --------------
 * Main function that reads μ-law encoded audio data from an input file,
 * converts it to linear PCM, and writes the decoded data to an output file.
 *
 * Returns:
 *   An integer indicating the success (0) or failure (1) of the program.
 */
int main() {
    // Specify the paths to the input and output files
    const char* inputFilePath = "Au8A_eng_f2.wav";
    const char* outputFilePath = "decoded_output.wav";

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

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

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

    // Prepare to read and decode the μ-law data
    uint8_t buffer;
    uint16_t pcmOutput;
    size_t dataChunkSize = 0;
```

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

    // Decode each μ-law byte and write the output as PCM data
    while (fread(&buffer, sizeof(buffer), 1, inputFile) == 1) {
        pcmOutput = MuLawToLinearPCM(buffer);
        fwrite(&pcmOutput, sizeof(pcmOutput), 1, outputFile);
        dataChunkSize += sizeof(pcmOutput);
    }

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

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

    // Close the input and output files
    fclose(inputFile);
    fclose(outputFile);

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

    return 0;
}
```