**ECE 540 PROJECT PHASE II**

**REPORT**

**(WirelessHART Network TEMpErature Meter)**

By

Abel A. Adegbesote

Submitted to

Dr. Yao

cid:image001.png@01CDFEEE.BA68BBF0

Department of Electrical and Computer Engineering

St. Cloud State University

St. Cloud, MN

Fall 2020

Contents Table

[Abstract 3](#_Toc57712072)

[Introduction 4](#_Toc57712073)

[Background: 4](#_Toc57712074)

[Problem Description: 4](#_Toc57712075)

[Design Objectives: 5](#_Toc57712076)

[System design 6](#_Toc57712077)

[Design Requirements: 6](#_Toc57712078)

[Design Specifications: 6](#_Toc57712079)

[Detailed Design: 9](#_Toc57712080)

[Implementation: 11](#_Toc57712081)

[Result: 19](#_Toc57712082)

[Conclusion: 21](#_Toc57712083)

[Constraints: 22](#_Toc57712084)

[References 22](#_Toc57712085)

[Appendix A: 23](#_Toc57712086)

A1- Schematic Diagram………………………………………………………………………………………………………23

[A2- Hardware Assembling 23](#_Toc57712087)

[A3- Program codes 24](#_Toc57712088)

# Abstract

WirelessHART is a wireless standard, engaged by many industries to provide trustworthy and protected information transmission with high degree of reliability. The architecture of the network is a mesh topology which allows redundant paths for data to reach its destination. It also makes use of the AES 128 bit encryption for data integrity and creates a level of security that is hard to penetrate. WirelessHART technology is utilized in some area in industry where sensor data collection is in difficult to access or extremely dangerous. Sensor’s data is sent over a network via one access point and this involved the configuration of the system parameters such as network ID number and security Join key in order to join a network. This project focus on designing a wireless network temperature meter. This is a system device that sense data through a temperature sensor (LM35), process data sensed using LPC 1114 Micro Controller Unit (MCU), display the Temperature on LCD and further send the displayed Temperature to a wireless network using a smart mesh WirelessHART Mote (DC9003A-C). The result, method used and necessary challenges faced are documented in this report.

# Introduction

The purpose of this project is to design a Temperature sensor that senses the temperature of the room or body object, the temperature can then be displayed on LCD and also sent wirelessly from the mote.

# Background:

Temperature sensors measurement is essential daily in this 21st century (Information Age). These Sensed data can be monitor, processed and used to control any temperature sensitive equipment ranging from Home uses to Industrial uses. Communication also has evolved from the era of 2G through 5G and from wired connections to wireless. These new trend of technology has made it easy to get sensed data remotely from places that are difficult to access due to physical challenges or extremely dangerous condition. Wireless transmitting has a lot of advantages in terms of cost, flexibility and reliability

# Problem Description:

A WirelessHART is a technology used in the industry to transmit data that are sensed over a wireless network securely and reliably. The sensed data for this project is a temperature from LM35. The measured temperature will be displayed on LCD and sent to a wireless network through the WirelessHART mote. This system can be used to monitor environmental temperatures, Home Temperatures, Industrial or processes involving temperature controls such as dangerous conditions like nuclear plant or boiler plants.

# Design Objectives:

The aim of this project is to design and implement a system unit capable of digitalizing the output temperature of a Temperature sensor (LM-35), using Microcontroller LPC1114. The digitalized temperature is displayed on LCD and is further sent wirelessly to a wireless network via a WirelessHART Mote (DC9003A-C). Some of the system units consist of other peripherals such as Push button, LCD, ADC, LEDs, ST-LINK as debugger, SmartMesh APIExplorer, DC9003A-C Mote, and Keil uVision5 as the C compiler.

# System design

# Design Requirements:

The main requirements for this project are as follow;

1. Laptop/PC

2. Keil uVision5 compiler for C language

3. 2 Push buttons as External Interrupt control and Reset

4. 3.3 Volts supply from ST-LINK debugger via USB

5. 2 x 16 LCD

6. LEDs and Wires

7. LPC1114/102 Arm Microcontroller

8. LM-35 (Temperature Sensor)

9. 1 Potentiometer, 3(Three) 1k Ohm resistors and 2(two) 10k Ohm resistors

10. ST-LINK V2 debugger

11. WirelessHART (DC9003A-C) Mote

12. SmartMesh APIEXPLORER

# Design Specifications:

The device comprises of Four (4) main parts which are Input (which consist mainly sensing unit and other Hardware peripherals), Computing processing, Communication unit (Mote), and Output (LCD and API Explorer).

Below is the block diagram and flow of each part;

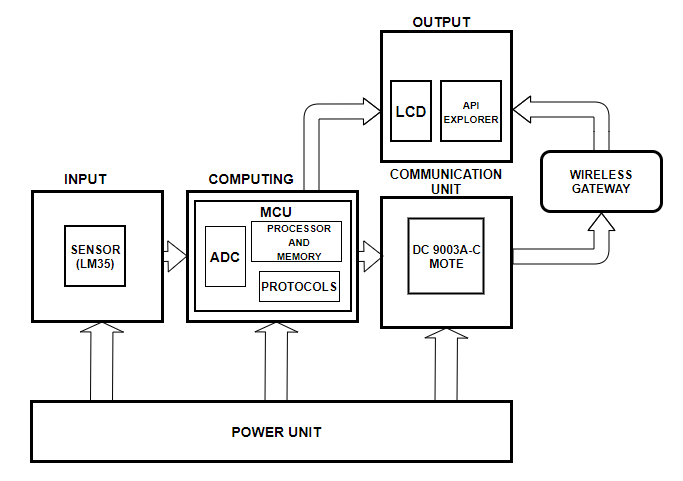


Figure 1: BLOCK DIAGRAM OF THE SYSTEM UNIT

INPUT: The input consists of the input devices (The 2 push buttons) and the analog temperature data from the LM-35 temperature sensor. The inputs are fed to the microcontroller for processing via IO pins and ADC pin of the LPC1114. The raw value of the ADC was sampled digitally then the voltage levels where converted to temperature. The temperature reading was calculated as using the below formula

value = ((LM35.read()\*3276)/10); // Temperature in Celsius

tempF = value \* 1.8 + 32; //Temperature in Fahrenheit

COMPUTING: This is the power house of the system where every logical and computing decision were made and the processed data or information is sent to the required output devices. At this part wireless protocols are also initiated to enable the MCU transmit data wirelessly via the UART pins of the MCU through the mote to a network.

COMMUNICATION UNIT: Communication between the mote and the wireless gateway are preconfigured into the two devices by the manufacturer. There are various features implemented including optimized routing of data and encryption of packets using AES-128 bit. The microcontroller gives the mote commands based on pre-determined packets that are sent to it by the microcontroller to initiate the many operations that the wireless network is capable of. These operations involve different states which are shown in the mote operation flowchart in figure 2 below. After transmission is complete, data is waiting at the network gateway for a computer to subscribe to it. This data then is read from the network gateway using the APIExplorer app.

OUTPUT: The output consists of the LCD for display of the digitalized data from the processing unit, APIExplorer app to read sent data from the network gateway and LEDs for the Digital Out

POWER UNIT: The power unit supplies a voltage of 3.3V to all parts of the unit that needed power to drive then. This supply source comes from the 3.3V pin of the ST-LINK debugger which is a USB powered source from the computer.

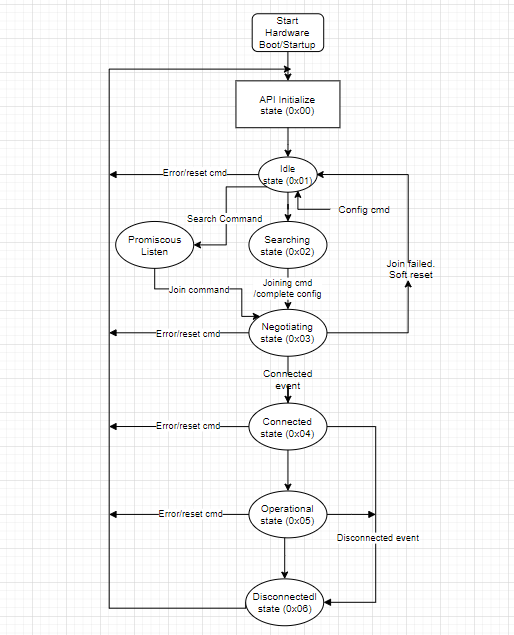


Figure 2: Flowchart of Mote State Machine

# Detailed Design:

The entire system relayed fully on the Mote to be said to be Smart. The Mote is based on the Dust wireless technology which uses the WirelessHART network protocol for transmission of packets over the network.

The totality of the project is a combination of a Dust WirelessHART transceiver, MCU (LPC1114), LCD, LEDs, Sensor (LM34), STLINK Debugger and other components. In order to keep the operation simple, the operation is illustrated by a flow diagram in figure 3 below. When the system is powered on, the microcontroller runs a program designed to search, Join and connect to a network based on certain parameters given which are the Net ID, Join Key, Bandwidth and the Port, the LCD displayed each state of the mote and transitions till it has connected to a network. The systems then further takes the Temperature from the LM35 sensor, Digitalized it by using the ADC pin on the MCU, sent it to be displayed on an LCD and further transmitting the data through the UART pins of the MCU to the Mote, the Mote then synchronizes the data to the connected network via wireless gateway. The states of the mote are showed in figure 2 above, and explanations are given as below;

Initial state (0x00): At this state serial port is not yet active and the mote will wait for the initialization command to so as to enable the serial port and enable the interrupts handler.

Idle state (0x01): this state accepts configuration commands. After receiving the join command, the mote moves into searching state

Promiscuous listen (0x0f): this state is exceptional search state that is invoked by the search command where the mote listens for advertisement from any network ID, and reports heard advertisements. The mote does not attempt to join the network and proceed to the searching state when given the join command.

Searching (0x02): states ensure the mote keeps its receiver on while searching for network.

Negotiating (0x03): The mote detected a network and has received a join request

Connected (0x04): The mote has joined the network and established communication with the network but has no links for sending or receiving data.

Operational: The mote has the links to the network and has sufficient bandwidth for basic communication with the control program.

Disconnected (0x05): The mote no longer has the links to the network and will reset after the disconnect timeout.

# Implementation:

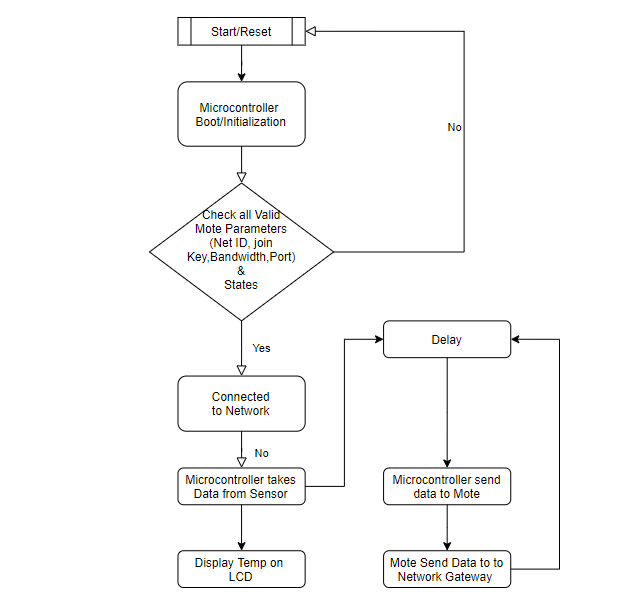


Figure 3: Simple Flow diagram operation of the system

There are 2 libraries used which are C libraries and QSL (Quick Start Libraries). A special subroutine (data\_send\_clib) was also included, without all these libraries the mote will not work.

The C Library puts together a set of functions that the application can call as commands. Every time the application calls a function this library handles formatting serial frames using HDLC framing and matching commands with their corresponding responses. Every time a notification is received on the mote, a callback function is invoked. This library handles parsing the serial notification frames and sending a serial acknowledgement when necessary.

The QSL handles most logic. Few functions called here will drive the mote through specific states without having to call each function in the C library to move from one state to another. It also handles timing issues where it has timeout constants to wait for a response from the network before it goes into the following state or reset.

Below are the functions that were called for the mote to search, join and connect to the network:

* dn\_qsl\_int(void): Whenever the program starts, this function is called to start the joining process, this function initializes all necessary structures and underlying modules and making sure that a serial connection to the mote is established. It is a Boolean function, when the serial connection between the mote and the external microcontroller is successfully established the function returns true.
* dn\_qsl\_isConnected(void): this function is also Boolean, it checks if the mote is currently connected and ready to send/receive data using a true or false question method.
* dn\_qsl\_connect (netID, joinkey, srcPort, service\_ms): This function returns a Boolean when the mote is successfully initialized, this function makes the mote to search and join a network with the given parameters (network ID and joinkey). Subsequently, a socket is opened and bound to the given port further making the mote ready to send/receive user data.

data\_send\_clib: This subroutine is also a C library which has some functions that is responsible for pushing data from the mote to the manager. The **send\_mote\_data ()** function in this subroutine is used as a replacement for the dn\_qsl\_send (payload, payload size, destport) in the QSL. If the mote is connected to the network, this function will be called to send a packet with the given port. The packet is addressed to the manager by default. The payload is a pointer to a byte array containing the payload.

After all physical connections were made and the codes were flashed. The mote goes through the various states as a result of the functions called in the QSL libraries, C library and the data\_send\_clib subroutine till it is ready to push data.

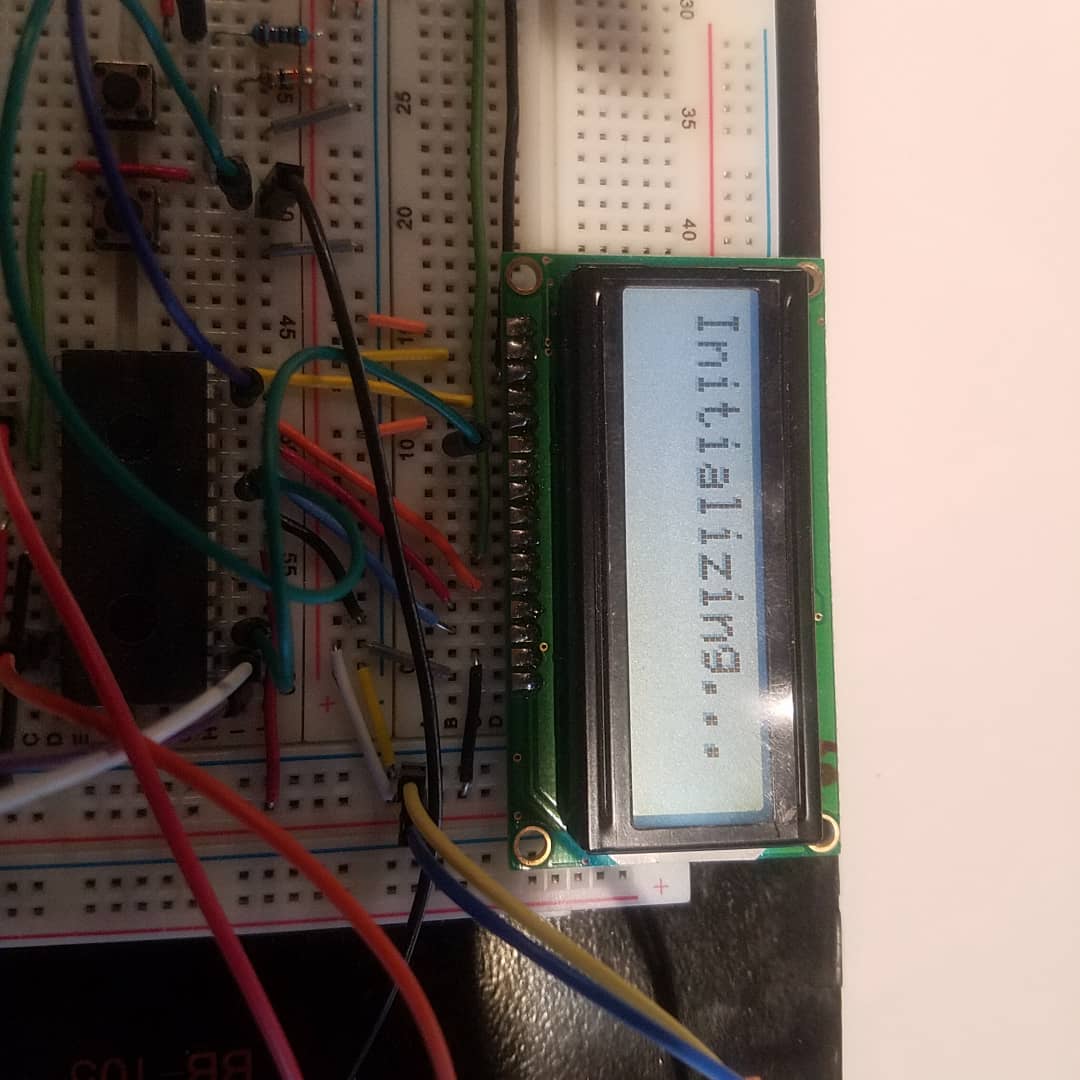


Figure 4: Showing the Initializing status of Mote

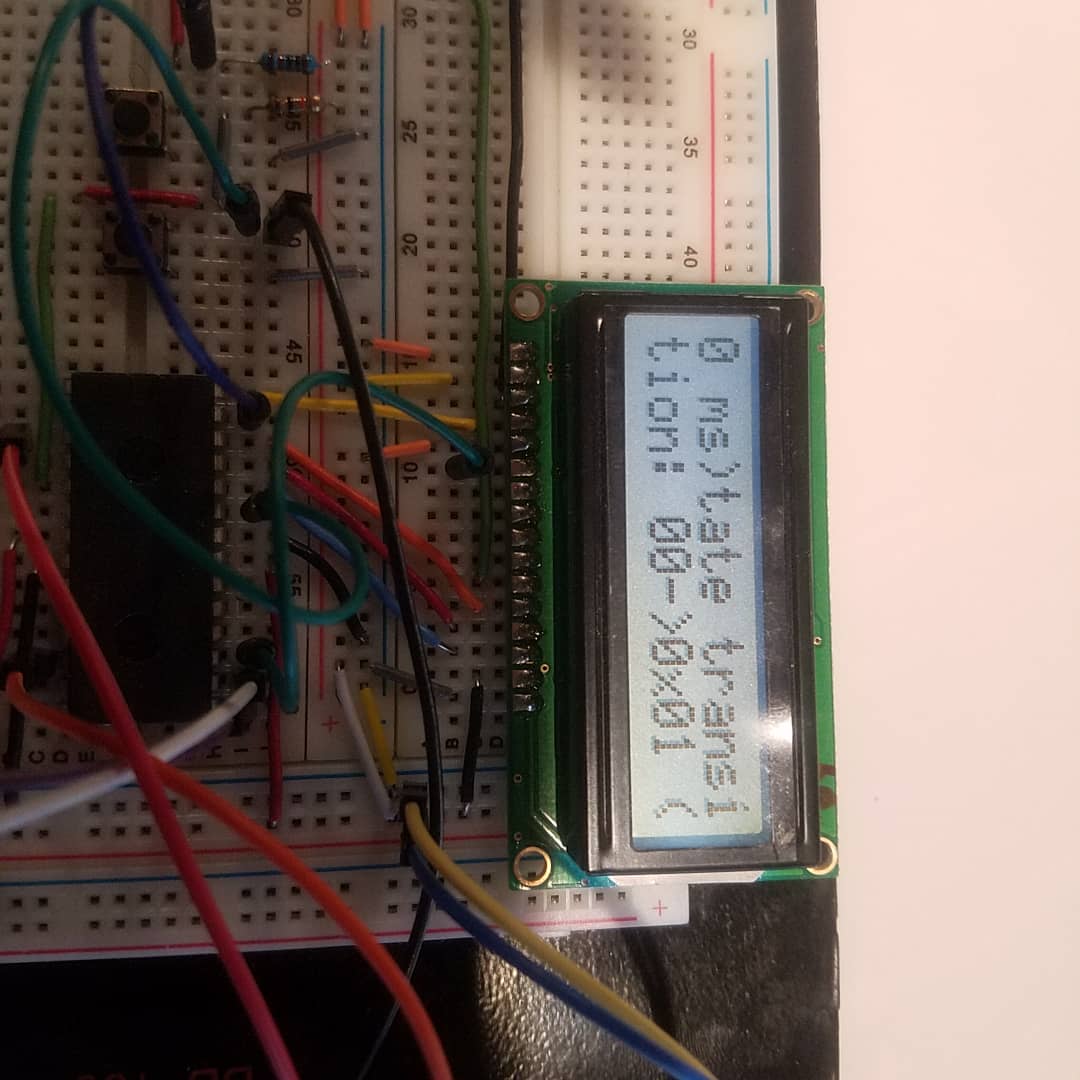


Figure 5: Showing the Transition State 0x01 of Mote

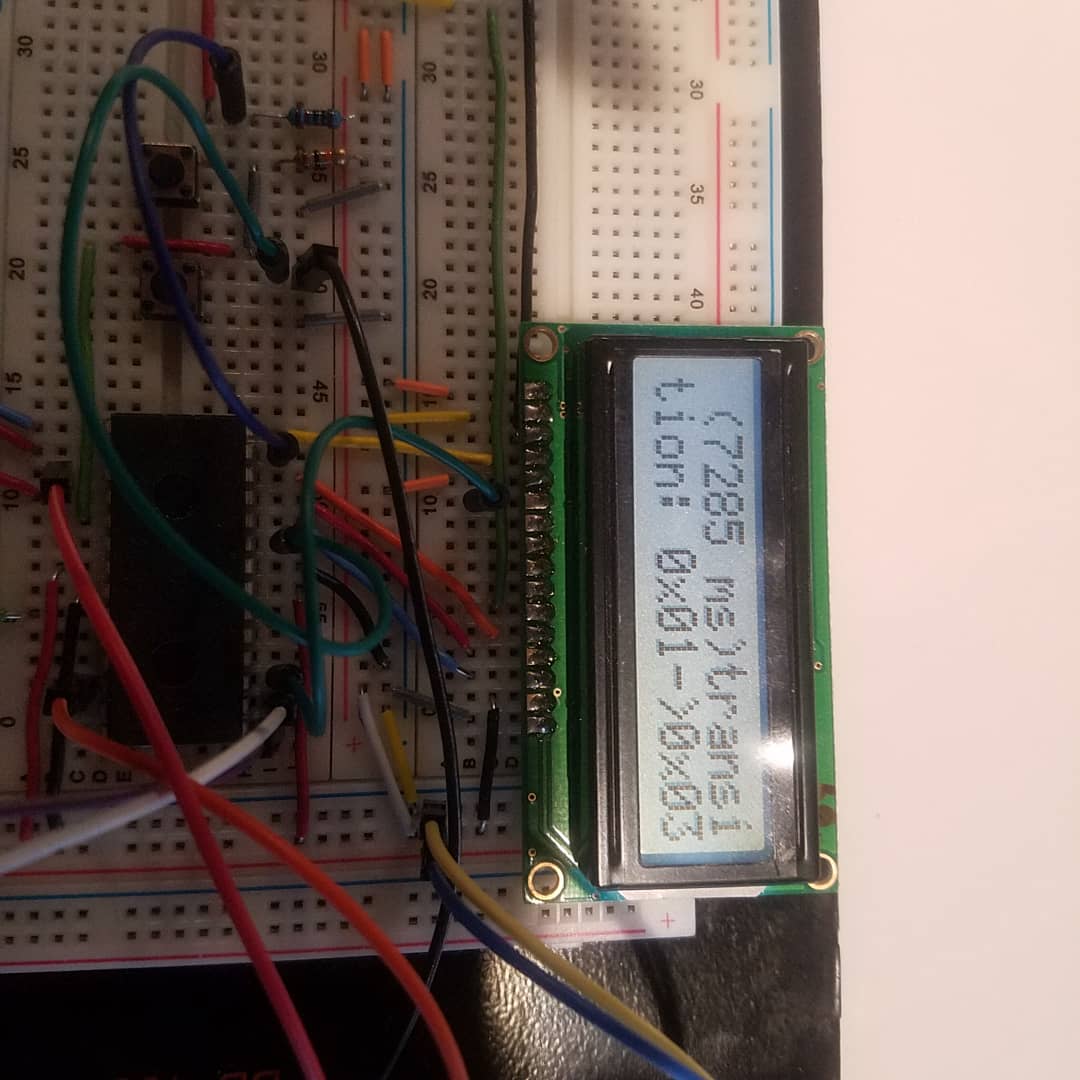


Figure 6: Showing the Transition State 0x03 of Mote

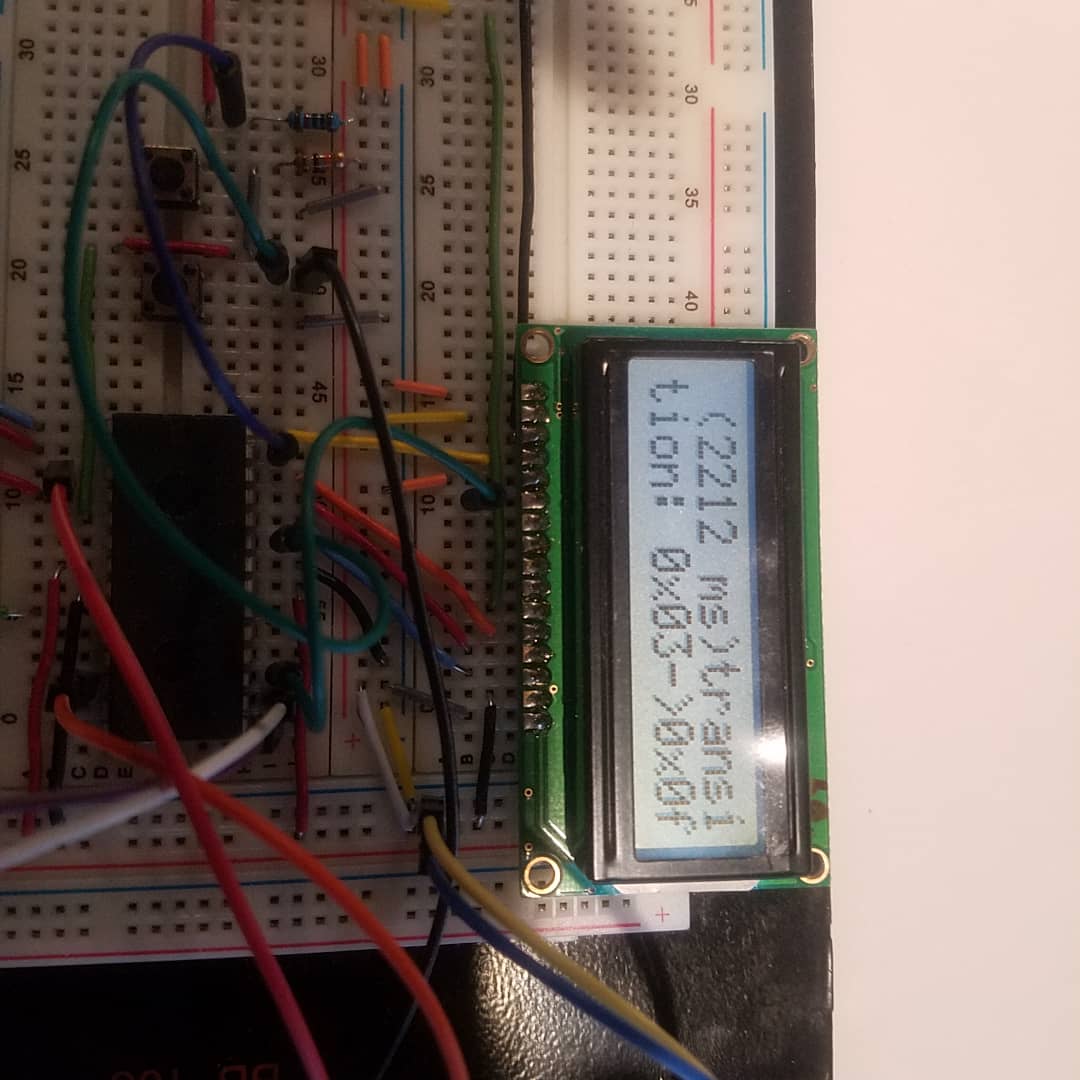


Figure 7: Showing the Transition State 0x0f of Mote

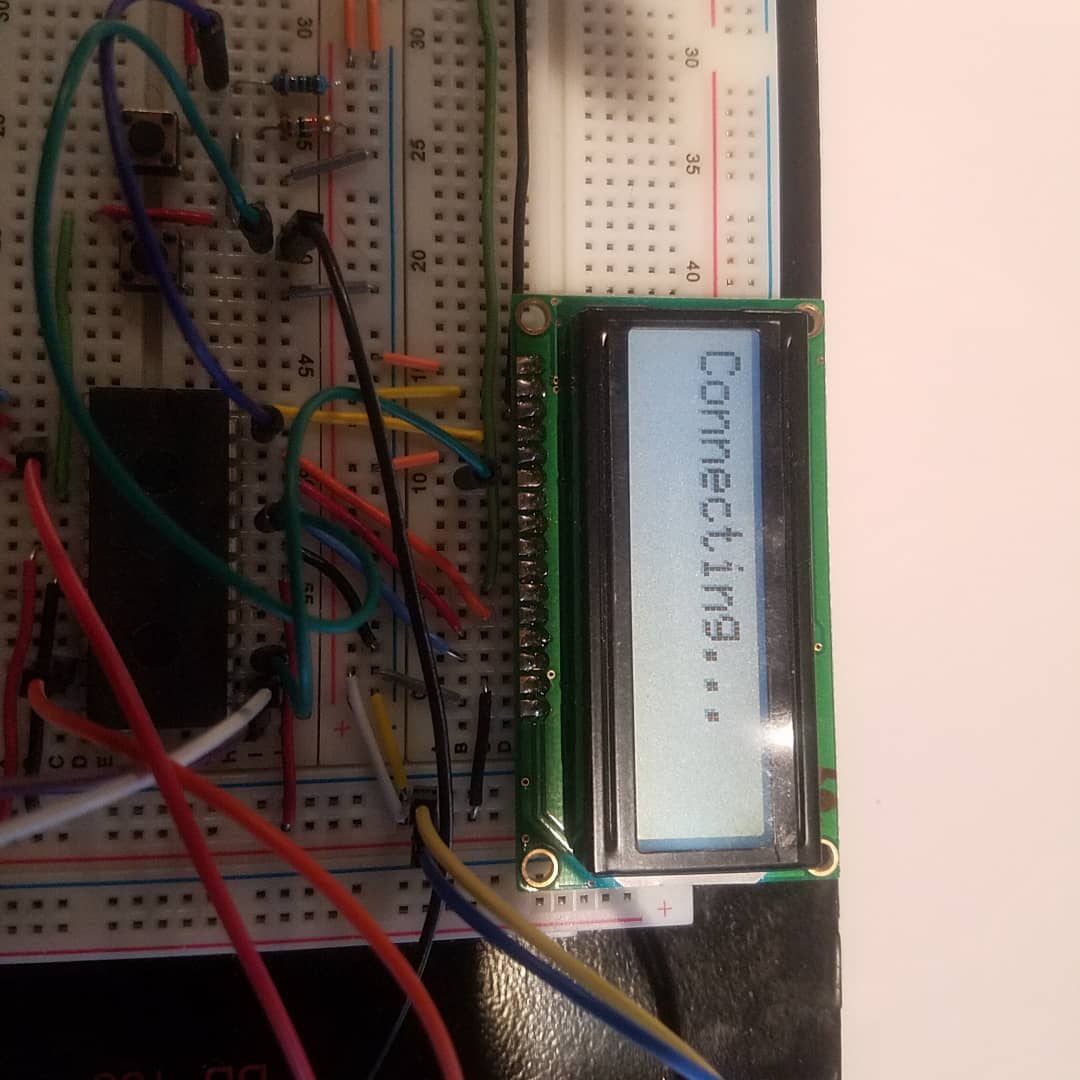


Figure 8: Showing the Connecting Status of Mote

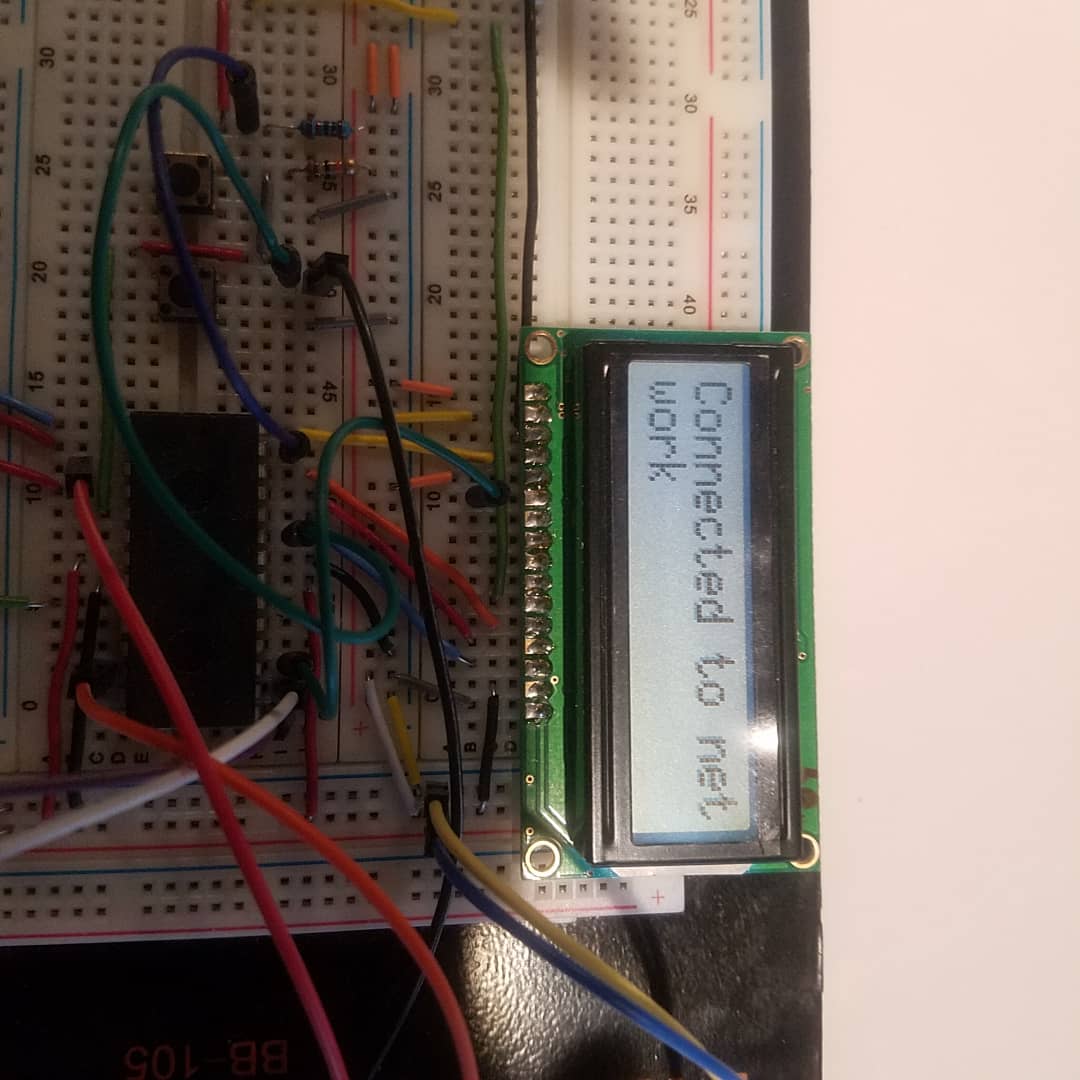


Figure 9: Showing the Connected to network Status of Mote

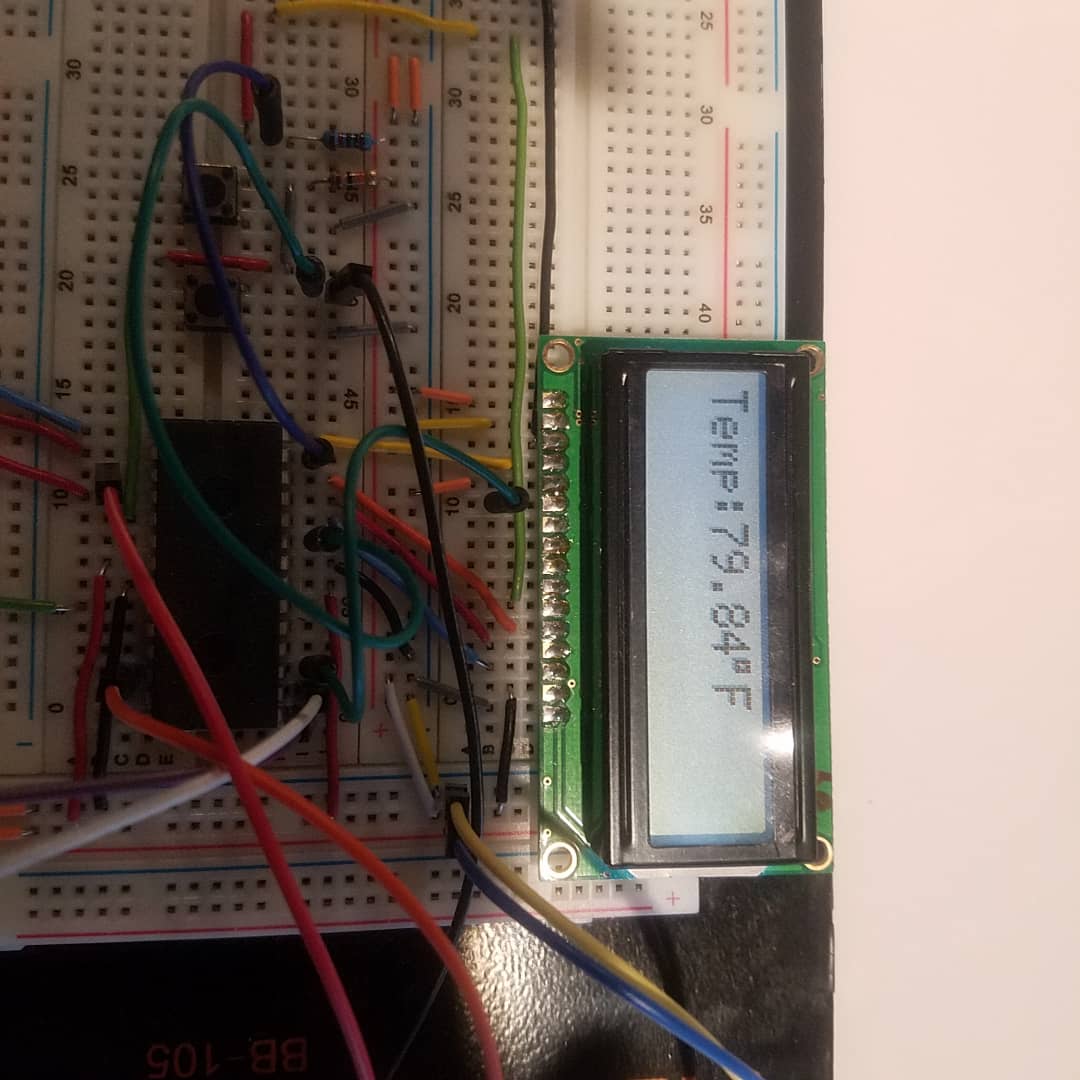


Figure 10: Showing the Temperature value from the sensor

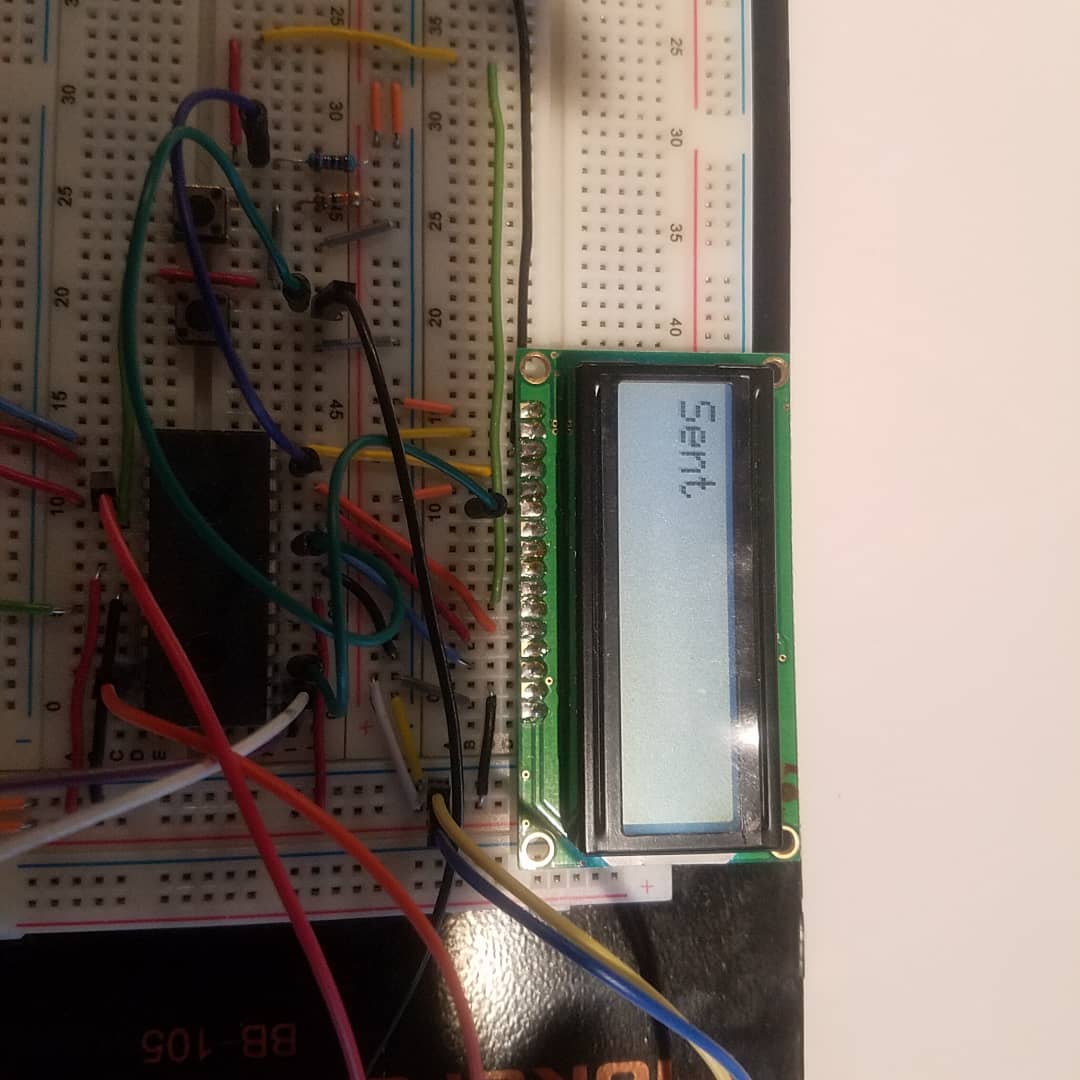


Figure 11: Showing the Sent to validate sensor data sent to mote

The APIExplorer is used to check the status of the mote by following the steps below;

* Connecting to the manager through XML-RPC using ip address 10.0.35.95 and port 4445
* Select ***“getMotes”*** command and then press send to see all motes list.
* Check the mote state in the response area where all motes are listed.

The status can also be checked on the Admin Toolset by connecting to the 10.0.35.95 via the browser. Username and password are “*admin*” respectively.

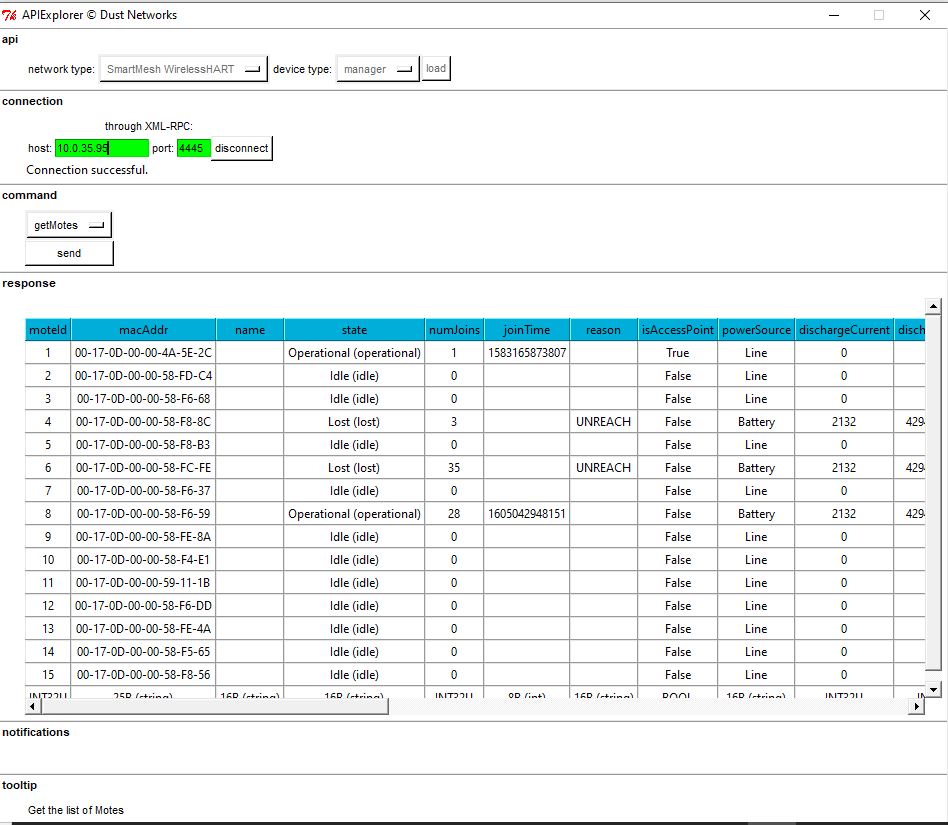


Figure 12: Showing the status of Mote using APIExplorer

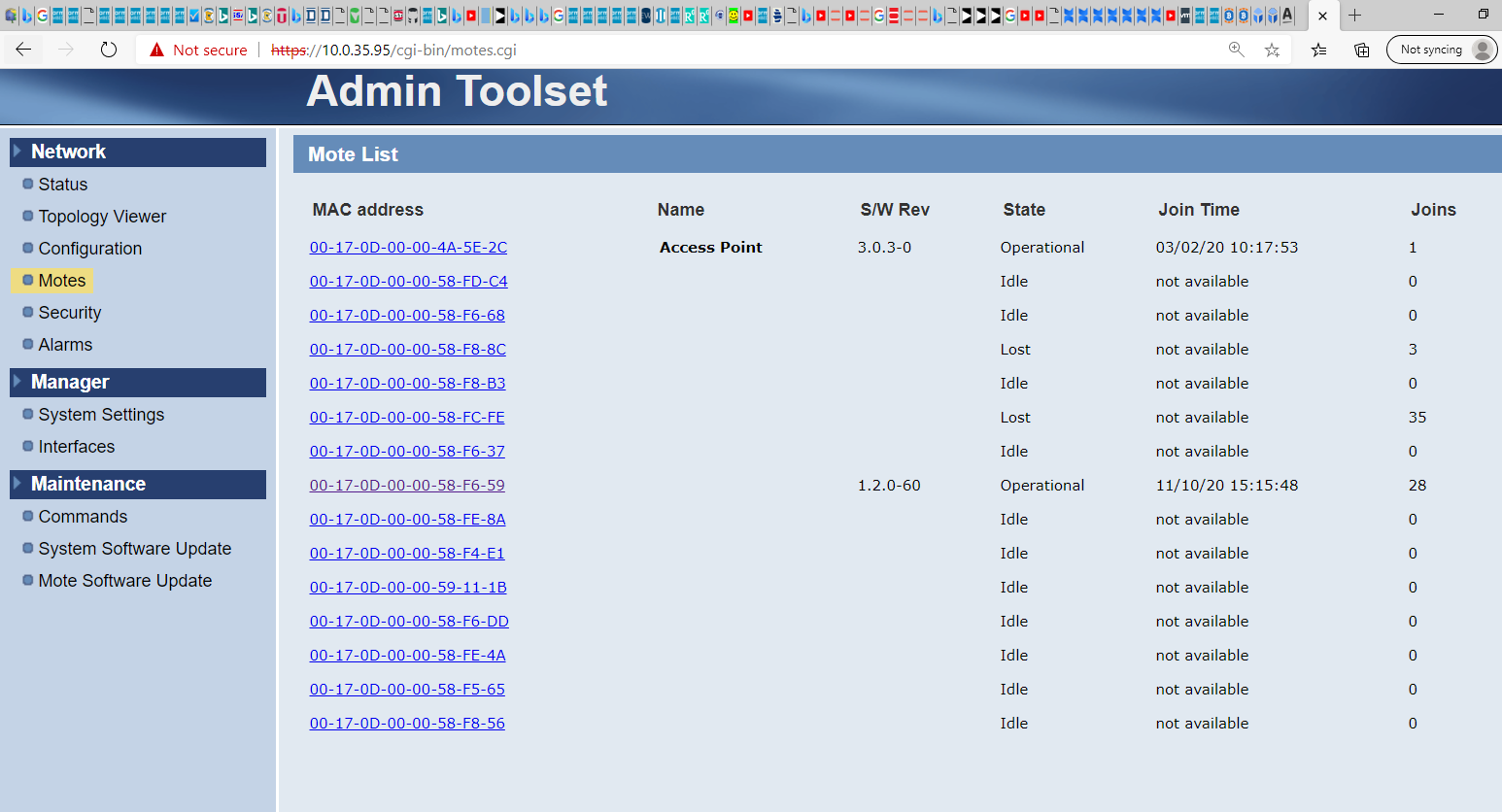


Figure 13: Showing the status of Mote using Admin Toolset

The status of the mote (Mac address 00-17-0D-00-00-58-F6-59) is operational which indicate that the mote is ready to accept any command to push data to the manager.

# Result:

When the **send\_mote\_data ()** function is called the digitalized temperature from the ADC is sent to the mote which further pushes it to the manager. The data pushed can be checked using the APIExplorer App, the data is in the HEX format and need to be converted to Float Point value to get the actual temperature data sent. The HEX to Float Point converter allows an input for HEX and converts it to Floating Point Value by a series of calculations. The procedure to use APIExplorer to check the data sent to the manager is shown below;

* Connecting to the manager through XML-RPC using IP address 10.0.35.95 and port 4445
* Select **“*subscribe*”** command and set the ***filter*** by typing **“*data”*** and then press send to subscribe to data notifications.
* Check the data sent in HEX format which can be seen at the payload column of notifications area.
* The Data is then converted from HEX to float point using the Hex to float converter at https://gregstoll.dyndns.org/~gregstoll/floattohex/

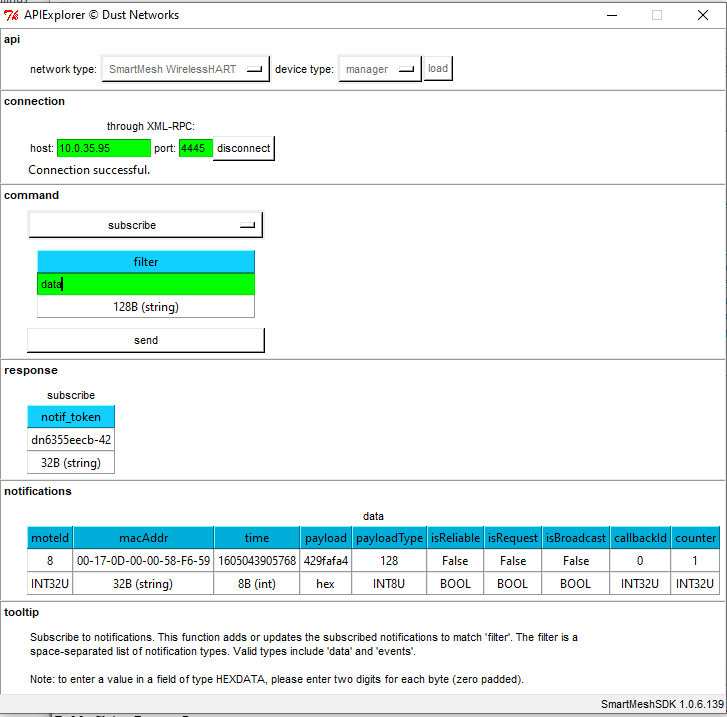


Figure 14: Showing the data sent to Manager using APIExplorer

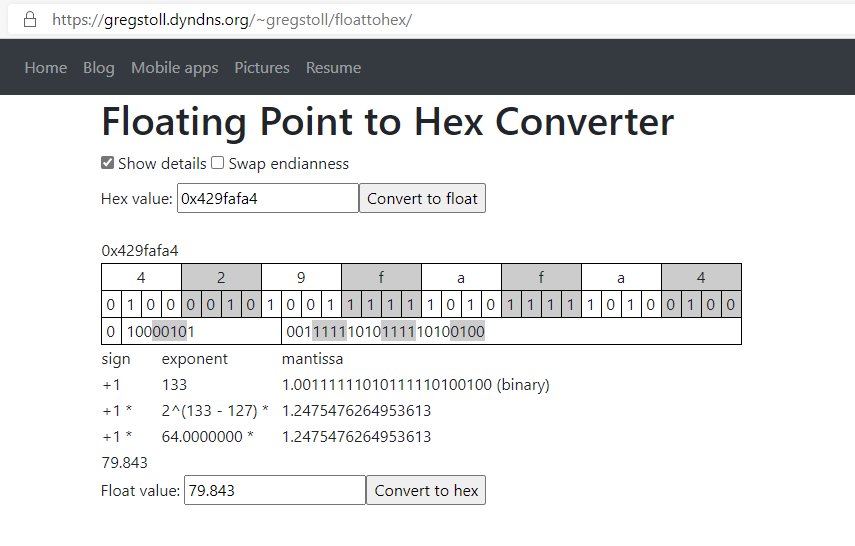


Figure 15: Showing the result of data from HEX to Float Point Converter

# Conclusion:

This WirelessHart project has really helped to develop prototype that can help collect data from sensors in remote and extremely dangerous locations, positions where human presence is difficult to reach. Various coding techniques and skills were used such as C language and mbed.

My Involvement in this project helps to develop some abilities to debug and troubleshoot codes. I became proficient in software languages like C++ and mbed techniques. I was able to learn more about how embedded systems are put together and how each works together to produce a good system. This project still needs to be developed further. I have confident now that I have completed this task. I will translate this experience into my future classes as well as my professional life.

# Constraints:

The constraints where not limited to the distance between the mote and the manager the further the mote is from the manager the weaker the signal receptions which can affect the reliability and availability thereby making it impossible for the mote to connect to the network and this can hinder the data to be sent. Another Constraint faced during the project was the accessibility to the manager due to the shutdown of the labs as a result of the COVID-19 pandemic.

# References

[1]Liu Chang; Zhang Guoguang, "The Design of Intelligent Temperature Transmitter Based on HART Protocol," *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012 Second International Conference* on , vol., no., pp.1499,1502, 8-10 Dec. 2012

[2] Kim, A.N.; Hekland, F.; Petersen, S.; Doyle, P., "When HART goes wireless: Understanding and implementing the WirelessHART standard," *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference* on , vol., no., pp.899,907, 15-18 Sept. 2008

[3]Raza, S.; Voigt, T., "Interconnecting WirelessHART and legacy HART networks," *Distributed Computing in Sensor Systems Workshops (DCOSSW), 2010 6th IEEE International Conference* on , vol., no., pp.1,8, 21-23 June 2010

[4] WirelessHART Device Specification,HCF\_SPEC-290, Revision 1.1.*HART Communication Foundation*, May 2008.

[5] <https://dustcloud.atlassian.net/wiki/spaces/QSL/overview?homepageId=80609282>

[6] <https://dustcloud.atlassian.net/wiki/spaces/QSL/pages/80609288/Installation>

# Appendix A:

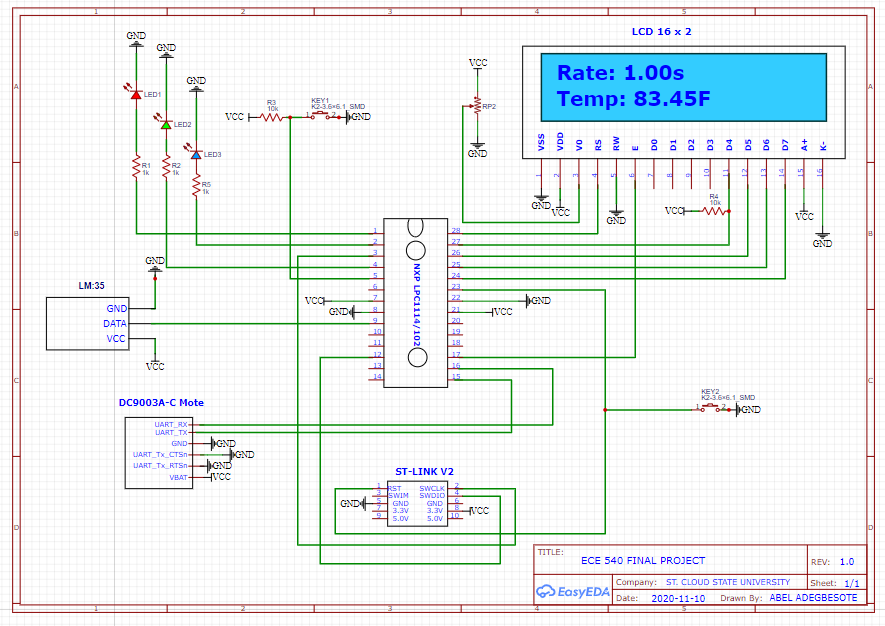
**A1- Schematic Diagram**

Figure 16: Schematics for the final WirelessHART Sensor system

# A2- Hardware Assembling

|  |  |
| --- | --- |
| **LPC1114** | **DC9003** |
| GND | GND |
| 3.3V | VBAT |
| TX1 | RX |
| RX1 | TX |
| GND | Tx CTSn |
| 3.3V  Table 1: Showing connection between LPC1114 and the Mote | RX RTSn |

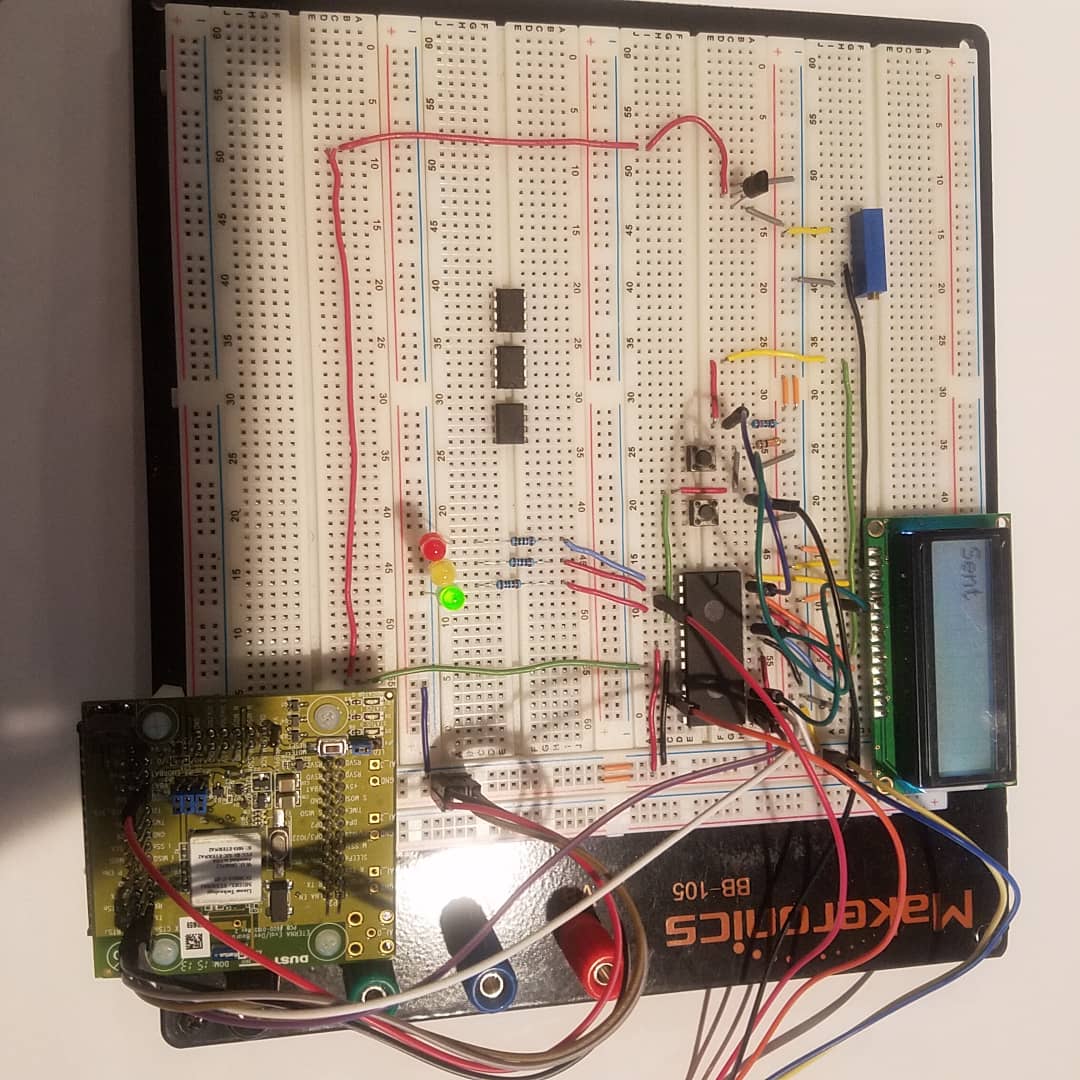


Figure 17: Hardware Implementation

# A3- Program codes

**Main.cpp**

#include "mbed.h"

#include "TextLCD.h"

#include "millis.h"

#include "dn\_qsl\_api.h" // Only really need this include from QSL

#include "dn\_debug.h" // Included to borrow debug macros

#include "dn\_endianness.h" // Included to borrow array copying

#include "dn\_time.h" // Included to borrow sleep function

#include "arm\_const\_structs.h"

#include <stdio.h>

#include <stdlib.h>

#include "data\_send\_clib.h"

char dS = 0xDF;

//External Interrupt

//InterruptIn button(dp5);

//Digital Out

DigitalOut myled(dp1); //led on pin 1

DigitalOut myled2(dp2); //led on pin 2

DigitalOut myled3(dp4); //led on pin 4

// ADC

AnalogIn LM35(dp9); // ADC pin float analog value=adc

// Mote Parameters

#define NETID 0 // Factory default value used if zero (1229)

#define JOINKEY 00000000000000000000000000000000 // Factory default value used if NULL (44 55 53 54 4E 45 54 57 4F 52 4B 53 52 4F 43 4B)

#define BANDWIDTH\_MS 5000 // Not changed if zero (default base bandwidth given by manager is 9 s)

#define SRC\_PORT 0 // Default port used if zero (0xf0b8)

#define DEST\_PORT 0 // Default port used if zero (0xf0b8)

#define DATA\_PERIOD\_MS 9000 // Should be longer than (or equal to) bandwidth

//Create LCD object

TextLCD lcd(dp28, dp17, dp27, dp26 ,dp25, dp24);// CREATE LCD rs, e, d4-d7

float value;

float32\_t tempF;

int main()

{

myled=0, myled2=0, myled3=0;

myled=1;

millisStart();

lcd.cls();

lcd.printf("Initializing...");

wait(5.0);

if (!dn\_qsl\_init())

{

return FALSE;// Initialization failed

lcd.printf("Not initialized");

myled=0;

dn\_qsl\_init();

}

while(1)

{

if (dn\_qsl\_isConnected())

{

lcd.cls();

lcd.printf("Connected to network");

dn\_sleep\_ms(150000);

lcd.cls(); //clear screen

value = ((LM35.read()\*3276)/10); // Temperature in Celsius

tempF = value \* 1.8 + 32; //Temperature in Farenheit

lcd.printf("Temp:%2.2f%cF",tempF,dS);

wait(5);

send\_mote\_data(tempF);

lcd.cls(); //clear screen

lcd.printf("Sent");

myled3 = 1; //turn on led

wait(4);

myled3 = 0; //turn off led

}else

{

lcd.cls();

lcd.printf("Connecting...");

myled = 0;

wait(5.0);

if (dn\_qsl\_connect(NETID, JOINKEY, SRC\_PORT, BANDWIDTH\_MS))

{

myled2 = 1; // Connected LED turned on

lcd.printf("Connected to network");

wait(4);

myled2 = 0;

}else

{

lcd.printf("Failed to connect");

}

}

}

}

**Send\_data\_Clib.cpp**

void send\_mote\_data(float data)

{

uint8\_t \*ptr = &payload[0];

unsigned char \*payload = (unsigned char \*)&data;

\*(ptr) = 0x05; //Command (send)

\*(ptr + 1) = 0x0D; //Length (Starting from DestAddr to end of Payload = 10)

\*(ptr + 2) = 0x04; //Flags: Ignore Packet ID(bit2)=1

\*(ptr + 3) = 0xF9; //destAddr (MSB)

\*(ptr + 4) = 0x81; //destAddr (LSB)

\*(ptr + 5) = 0x00; //serviceId

\*(ptr + 6) = 0x02; //appDomain (maintenance)

\*(ptr + 7) = 0x02; //priority (high)

\*(ptr + 8) = 0xFF; //reserved

\*(ptr + 9) = 0xFF; //reserved

\*(ptr + 10) = 0x00; //seqNum

\*(ptr + 11) = 0x04; //payloadLength (1 byte)

\*(ptr + 12) = \*(payload + 3); //payload byte 1 (Reverse Byte Order)

\*(ptr + 13) = \*(payload + 2); //payload byte 2 (Reverse Byte Order)

\*(ptr + 14) = \*(payload + 1); //payload byte 3 (Reverse Byte Order)

\*(ptr + 15) = \*(payload); //payload byte 4 (Reverse Byte Order)

send\_data(ptr, 16);

}