A
PROJECT REPORT ON

"CAN Based Vehicle sensor Monitoring System with IOT"

SUBMITTED IN
PARTIAL FULFILLMENT OF

DIPLOMA IN EMBEDDED SYSTEM DESIGN (PG-DESD)

BY

PRASHANT TRIPATHI (56317)
PRADEEP KUMAR MAHOUR (56822)
JAGRUTI RATHOD (56748)
NAYANA PATIL (56436)

AT

SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY, HINJAWADI

**SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY,
HINJAWADI.**

CERTIFICATE

This is to certify that the project

“CAN Based Vehicle sensor Monitoring System with IOT”

Has been submitted by

PRASHANT TRIPATHI (56317)
PRADEEP KUMAR MAHOUR (56822)
JAGRUTI RATHOD (56748)
NAYANA PATIL (56436)

In partial fulfillment of the requirement for the Course of **PG Diploma in Embedded System Design (PG-DESD SEP 2021)** as prescribed by The **CDAC** ACTS, PUNE

Place: Hinjewadi

Date : 13/04/2022

Authorized Signature

ACKNOWLEDGEMENT

It is indeed a matter of great pleasure and proud privilege to be able to present this project on “CAN Based Vehicle sensor Monitoring System with IOT”. The completion of this project work is great experience in student’s life and its execution is inevitable in hands of guide. We are highly indebted to the Project Guide Mr. Devendra Dhande for his invaluable guidance and appreciation for giving form and substance to this project. It is due to her enduring efforts, patience and enthusiasm which has given sense of direction and purposefulness to this project and alternately made it a success.

Amongst the panorama of other people who supported us in this project, we are highly indebted to our technical staff of Sunbeam. It is there support, encouragement and guidance which motivated us to go ahead and make the project a successful venture.

Lastly, we would like to express our gratitude to all other helping hands which have been directly or indirectly associated with our project.

ABSTRACT

In this study, a driver behavior analysis tool and an in-vehicle sensor data monitoring system are presented. The study offers a system based on low-cost hardware and advanced software capabilities.

The embedded system utilizes the information provided by in-car sensors using the Controller Area Network (CAN). STM32 and Arduino works as CAN nodes in this project can communication happening between them.

By combining this information, the driving performance and driving characteristics are determined. Thanks to the CAN protocol for monitoring sensor of vehicle and display their variation on web server using thinger.io application. In addition, dozens of vehicle-related sensor data can be accessed by users.

INDEX

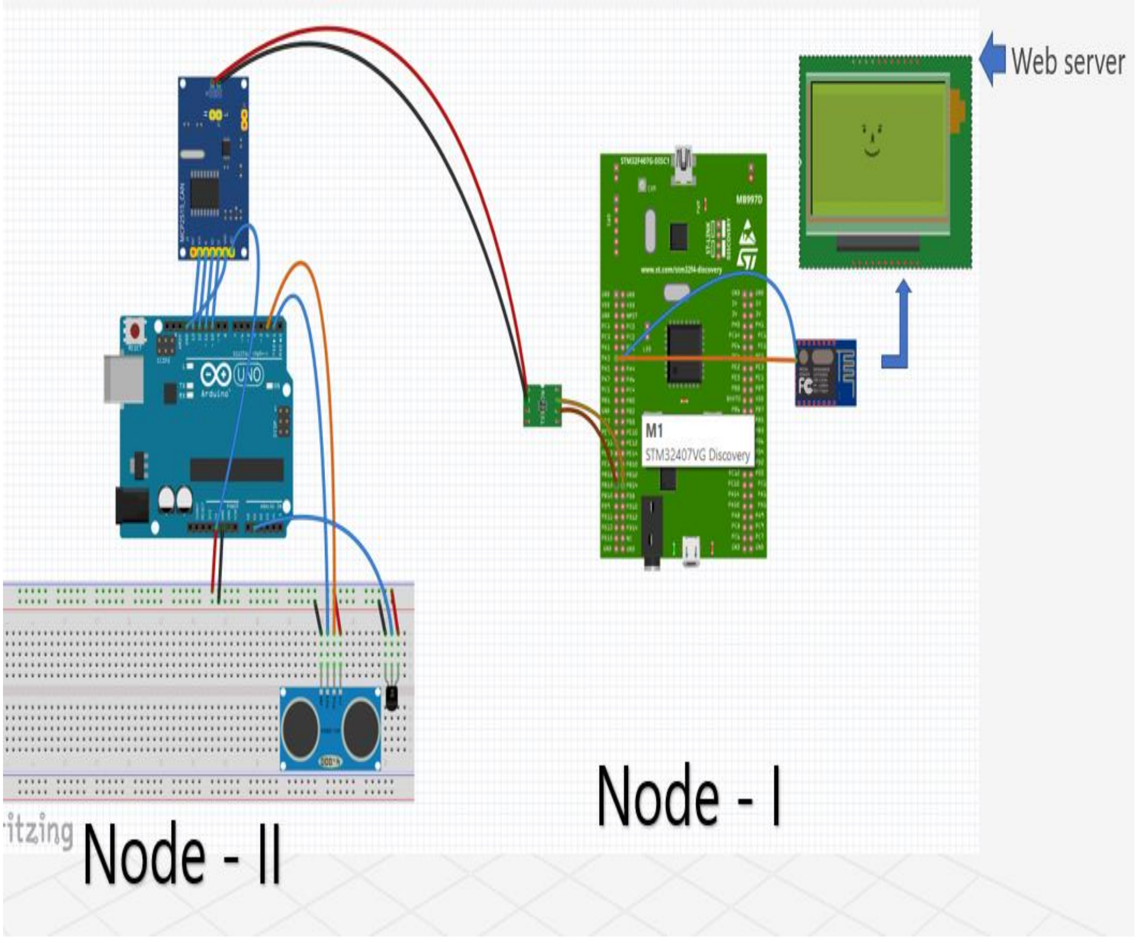
INTRODUCTION

In Today's Busy World Importance of smart devices is increasing. From Monitoring to security these devices are becoming part of our lives. ThisProject is designed to monitor sensors present in Automotive vehicles.

System has 2 can nodes. On 1st node we have use MCP2515 CAN module with inbuilt TJA1050 CAN Trans receiver where we interfaced temperature sensor (LM35) and ultrasonic sensor on 2nd node we have use STM32f407g Microcontroller with mcp2551 CAN Transreceiver which works as a Dashboard.

CAN communication is happening between node1 and node 2. For monitoring using IOT we have used ESP8266 module which takes sensor data from STM32f407g using Serial communication. And display sensor variation on web server using thinger.io application.

PROJECT ARCHITECTURE



BLOCK DIAGRAM DISCRIPTION

We have divided this block diagram in 2 nodes that is node 1 and node 2. On 1st node we have use MCP2515 CAN module with inbuilt TJA1050 CAN Trans receiver which convert digital signal of 2515 into differential can signal. where we interfaced temperature sensor (LM35) and ultrasonic sensor. on 2nd node we have use STM32f407g Microcontroller with mcp2551 CAN Transreceiver which works as a Dashboard. CAN communication is happening between node1 and node 2.
Suppose we have use temperature sensor connect data pin to Arduino A0 and other pin connected to ground and Vcc and see the variation of sensor on web server using thinger.io application.

- **Connection between CAN controller and Arduino**

CAN Controller Pin	Arduino Pin
SCK	D13
SI	D11
SO	D12
CS	D10
Ground	Ground
Vcc	Vcc

- **Connection between ultrasonic sensor and Arduino**

Ultrasonic Sensor pin	Arduino pin
Vcc	Vin
Trigger	3
Echo	2
Ground	Ground

- **Connection between Temperature sensor and Arduino**

Temperature sensor pin	Arduino
Ground	Ground
Data	A0
Vcc	Vcc

CAN (Controller Area Network)

CAN stands for controller area network. They are designed specially to meet the Automobile Industry needs. Before CAN was introduced, each electronic device is connected to other devices using many wires to enable communication. But when the functions in the automobile system increased, it was difficult to maintain because of the tedious wiring system. With the help of the CAN bus system, which allows ECUs to communicate with each other without much complexity by just connecting each ECU to the common serial bus. Hence when compared with the other protocols used in automotive systems i.e., CAN vs LIN, CAN is robust due to less complexity.

CAN Protocol can be defined as a set of rules for transmitting and receiving messages in a network of electronic devices connected through a serial bus. Each electronic device in a CAN network is called a node. Each node must have hardware and software embedded in them for data exchange. Every node of a CAN bus system has a host microcontroller unit, CAN controller and, CAN transceiver in it. CAN controller is a chip that can be embedded inside the host controller or added separately, which is needed to manage the data and sends data via transceiver over the serial bus and vice versa.

CAN Transceiver chip is used to adapt signals to CAN bus levels. CAN is a message-based protocol where every message is identified by a predefined unique ID. The transmitted data packet is received by all nodes in a CAN bus network, but depending on the ID, CAN node decides whether to accept it or not. CAN bus follows the arbitration process when multiple nodes try to send data at the same time.

Features of CAN

- Serial communication
- multi-master protocol
- compact – twisted pair line
- 1 megabits per second

Frame types of CAN:

Frame is a defined structure or format that carries meaningful data(bytes) within the network. CAN has four frame types:

- Data Frame
- Remote Frame
- Error Frame
- Overload frame

Data Frame:-

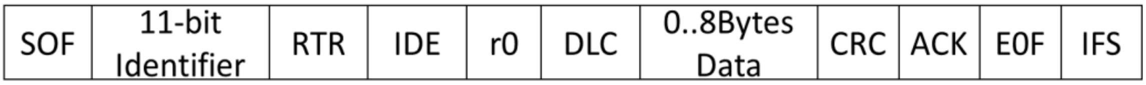
Data Frame contains the actual data for transmission. Data Frames consist of fields that provide additional information about the message i.e., Arbitration Field, Control Field, Data Field, CRC Field, a 2-bit Acknowledge Field, and an End of Frame.

There are two types of Data frames

1. Standard Frame or Base frame format
2. Extended Frame format

The only difference between the two formats is standard frame supports an 11-bit identifier, and the extended frame supports a 29-bit identifier made up of an 11-bit identifier and extended 18-bit identifier. IDE bit is dominant in a standard frame and recessive in an extended frame.

Base Frame Format: Standard CAN protocol is also known as the Base frame format. The standard frame is mainly used to send data.



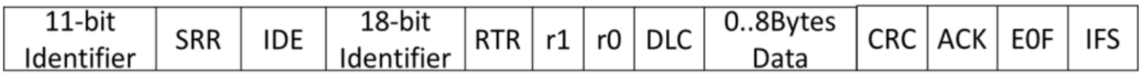
SOF	11-bit Identifier	RTR	IDE	r0	DLC	0..8Bytes Data	CRC	ACK	E0F	IFS

Figure 2. Standard Frame

Terminologies

- SOF - Start of Frame. Denotes the start of frame transmission.
- Identifier - 11-bit unique id and also represents message priority Lower the value, higher is the priority.
- RTR - Remote Transmission Request. It is dominant for data frames and recessive for remote frames.
- IDE - Single Identification Extension. It is dominant for standard frames and recessive for extended frames.
- R0 - reserved bit.
- DLC - Data Length Code. Defines the length of the data being sent. It is of 4-bit size.
- Data - Data to be transmitted and length is decided by DLC.
- CRC– Cyclic Redundancy Check. It contains the checksum of the preceding application data for error detection.
- ACK– Acknowledge. It is 2 bits in length. It is dominant if an accurate message is received.
- EOF– end of the frame and must be recessive.
- IFS– Inter Frame Space. It contains the time required by the controller to move a correctly received frame to its proper position.

Extended Frame:



11-bit Identifier	SRR	IDE	18-bit Identifier	RTR	r1	r0	DLC	0..8Bytes Data	CRC	ACK	E0F	IFS

Figure 3.

Extended Frame It is the same as the Standard Frame with some additional fields. SRR- Substitute Reverse Request. The SRR bit is always transmitted as a recessive bit to ensure that the standard Data frame has high priority when compared to the extended Data frame if both messages have the same 11-bit identifier. It also contains an 18-bit identifier other than an 11-bit identifier. r1- Reserved bit.

Remote Frame:

The remote frame is similar to a data frame with two differences. The remote frame is sent by the receiver to request data from the transmitter. The differences between the remote frame and data frame are remote frame does not contain any data field in it since it is not used for the data transfer. The second difference is RTR bit in the arbitration field is recessive for the remote frame. The data frame wins the arbitration if both are ready to be transmitted at the same time because of the dominant RTR bit in the data frame.

Error frames:

If transmitting or receiving nodes detect an error, it will immediately stop transmission and send an error frame consisting of an error flag made up of six dominant bits and error flag delimiter made up of eight recessive bits

Error flags:

1.Active Error flag
2.Passive Error flag

Active Error Flag: Transmitted by the node when an error is detected on a CAN network.
Passive Error Flag: Transmitted by the node when an active error is detected on a CAN network.

Error Counters: If an error is detected on a bus, then TEC or REC count increases.
1.Transmit Error Counter (TEC)
2.Receive Error Counter (REC)

- When TEC and REC is lesser than 128, an active error frame is transmitted
- When TEC or REC is greater than 127 and less than 255, a passive frame

is transmitted

- When TEC is greater than 255, the node enters into bus off state, then no frames can be transmitted

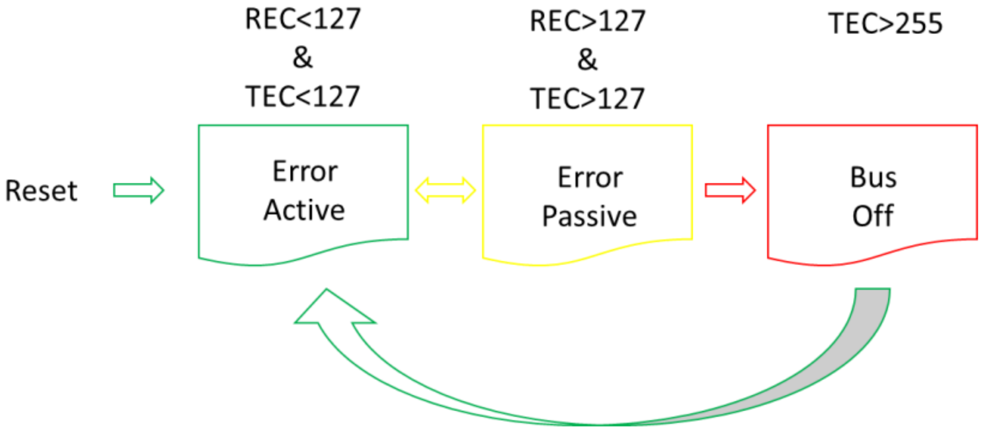


Figure 4.

Error Transition State Diagram Overload Frame The overload frame contains two fields Overload flag and Overload Delimiter. Overload flag consists of six dominant bits followed by overload flags generated by other nodes. Overload delimiter consists of eight recessive bits. Overload conditions that lead to the transmission of overload frame are:
1.when the receiver needs a delay of the next frame.
2.when a dominant bit is detected during intermission.

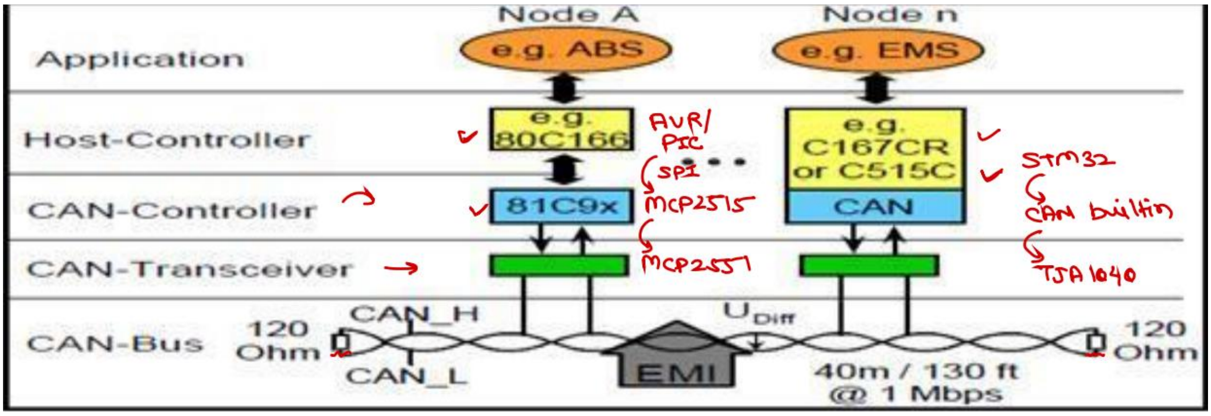
Overload Frame :
The overload frame contains two fields Overload flag and Overload Delimiter. Overload flag consists of six dominant bits followed by overload flags generated by other nodes. Overload delimiter consists of eight recessive bits. Overload conditions that lead to the transmission of overload frame are:
1.when the receiver needs a delay of the next frame.
2.when a dominant bit is detected during intermission.

Layers of CAN
It consists of three layers i.e. Application layer, Data link layer, and Physical layer.

- **Application Layer:** This layer interacts with the operating system or application of a CAN device.
- **Data Link Layer:** It connects actual data to the protocol in terms of sending, receiving, and validating data.
- **Physical Layer:** It represents the actual Hardware i.e. CAN Controller and Transceiver.

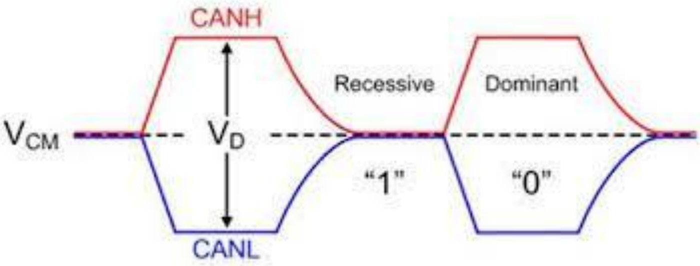
CAN Node

Each electronic device is called as Node. The CAN transceiver and controller make up the CAN node.
A CAN transceiver is **the interface between the controller and the CAN bus**.
The transceiver translates the logic level messages from the controller into the CAN differential scheme on the CANH and CANL pins of the CAN transceiver.



CAN Bus- CAN bus is a two twisted wire bus i.e. CANH & CANL.

- ` The passive voltage of each line is 2.5 V. (store)
- ` The active voltages are 3.5 V and 1.5 V. (produce)
- ` When both lines are 2.5 V, difference is 0 V. It represent logic 1 & called as “recessive bit”
- When both lines are pulled to 3.5 V and 1.5V respectively, then difference is 2.5v it represents a logic 0 so it is also called as “dominant bit”
- Note that dominant bit can always overwrite recessive bit.
- ` CAN bus is a linear bus terminated with 120 ohm. Also input impedance of each node is 120 ohm



Handling Bit Level Errors

Bit stuffing: CAN protocol follows NRZ encoding for transmission. The logic level does not change between bit interval. CAN requires a transition in logic level for re-synchronization. Hence 1 bit of opposite logic level is sent after 5 same consecutive bits. This is known as stuff bit and the receiver identifies it.

Bit error: A node that is sending the bit always monitors the bus. If the bit sent by the transmitter differs from the bit value on the bus then an error frame is generated.

Advantages of CAN bus Protocol

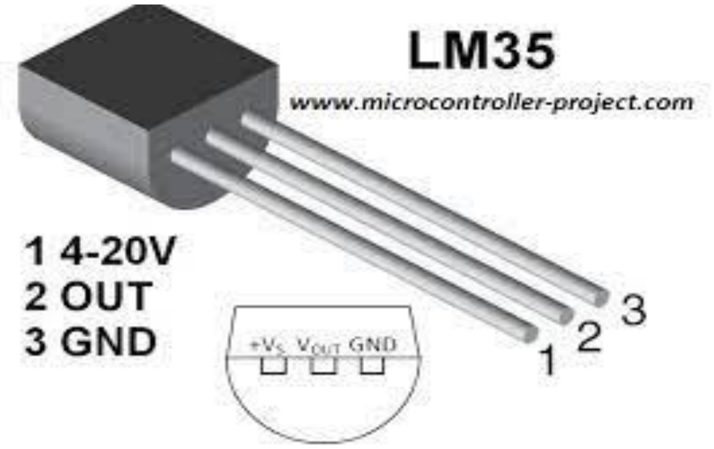
- Low cost because of reduced wiring
- Saves time due to simple wiring
- Auto retransmission of Lost messages
- Supports Error Detection
- Flexible Data transmission rate

Application of CAN

- The controller area network (CAN) is used for transmission airbags, antilock braking, electric power steering etc.
- It is used in audio video systems.
- The controller area network (CAN) is used in lifts and escalators.
- It is used in sport cameras.
- It is used in automatic doors.
- It is used in telescope and coffee machines.
- The controller area network (CAN) is used in aircraft with flight state sensors,
  navigation systems, flight data analysis to aircraft engine control systems such as
  fuel systems, linear actuators and pumps.
- It is used to windows, doors, and mirror adjustment.

HARDWARE DESCRIPTION

➢ LM35 TEMPERATURE SENSOR:-

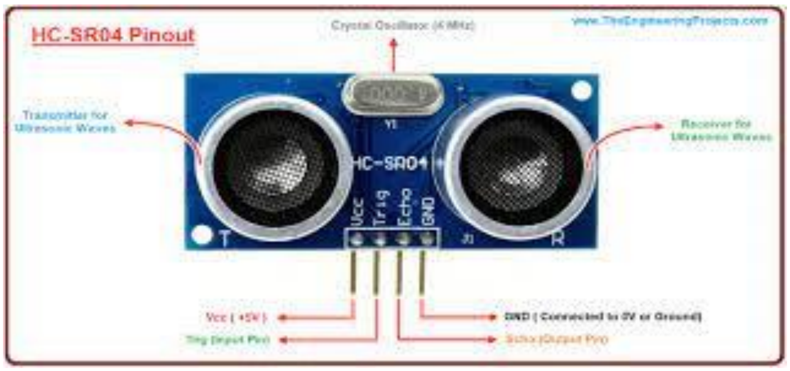


LM35 is a temperature sensor that outputs an analog signal which is proportional to the instantaneous temperature. The output voltage can easily be interpreted to obtain a temperature reading in Celsius. The advantage of lm35 over thermistor is it does not require any external calibration. The coating also protects it from self-heating. Low cost (approximately $0.95) and greater accuracy make it popular among hobbyists, DIY circuit makers, and students. Many low-end products take advantage of low cost, greater accuracy and used LM35 in their products. Its approximately 15+ years to its first release but the sensor is still surviving and is used in any products.

LM35 Temperature sensor Features

- Calibrated Directly in Celsius (Centigrade)
- Linear + 10-mV/°C Scale Factor
- 0.5°C Ensured Accuracy (at 25°C)
- Rated for Full −55°C to 150°C Range
- Suitable for Remote Applications
- Operates from 4 V to 30 V
- Less than 60-µA Current Drain
- Low Self-Heating, 0.08°C in Still Air
- Non-Linearity Only ±¼°C Typical
- Low-Impedance Output, 0.1 Ω for 1-mA Load

➢ HC-SR04 ULTRASONIC SENSOR :-

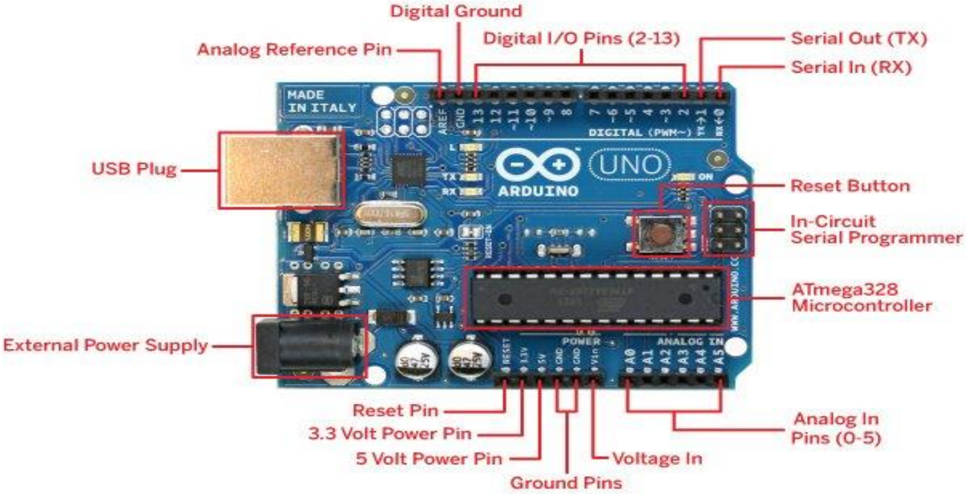

HC-SR04 Pinout
Crystal Oscillator (4 MHz)
www.TheEngineeringProjects.com
Transmitter for Ultrasonic Waves
Receiver for Ultrasonic Waves
HC-SR04
Y1
Vcc
Trig
Echo
GND
T
J1
R
Vcc (+5V)
Trig (Input Pin)
GND (Connected to 0V or Ground)
Echo (Output Pin)

An ultrasonic sensor is an electronic device that measures the distance of a target object by emitting ultrasonic sound waves, and converts the reflected sound into an electrical signal. Ultrasonic waves travel faster than the speed of audible sound (i.e. the sound that humans can hear). Ultrasonic sensors have two main components: the transmitter (which emits the sound using piezoelectric crystals) and the receiver (which encounters the sound after it has travelled to and from the target).

In order to calculate the distance between the sensor and the object, the sensor measures the time it takes between the emission of the sound by the transmitter to its contact with the receiver. The formula for this calculation is D = ½ T x C (where D is the distance, T is the time, and C is the speed of sound ~ 343 meters/second). For example, if a scientist set up an ultrasonic sensor aimed at a box and it took 0.025 seconds for the sound to bounce back, the distance between the ultrasonic sensor and the box would be:

D = 0.5 x 0.025 x 343
or about 4.2875 meters.

- **ARDUINO UNO R3 :-**



Arduino Uno is a microcontroller board based on the ATmega328P (datasheet). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator (CSTCE16M0V53-R0), a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started.. You can tinker with your Uno without worrying too much about doing something wrong, worst case scenario you can replace the chip for a few dollars and start over again.

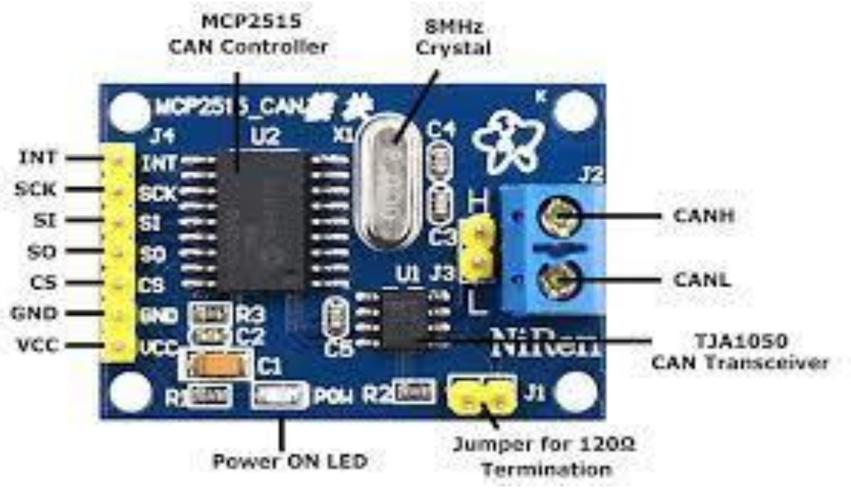
"Uno" means one in Italian and was chosen to mark the release of Arduino Software (IDE) 1.0. The Uno board and version 1.0 of Arduino Software (IDE) were the reference versions of Arduino, now evolved to newer releases. The Uno board is the first in a series of USB Arduino boards, and the reference model for the Arduino platform; for an extensive list of current, past or outdated boards see the Arduino index of boards.

Tech specs:-

MICROCONTROLLER	ATmega328P
OPERATING VOLTAGE	5V
INPUT VOLTAGE (RECOMMENDED)	7-12V
INPUT VOLTAGE (LIMIT)	6-20V
DIGITAL I/O PINS	14 (of which 6 provide PWM output)

PWM DIGITAL I/O PINS	6
ANALOG INPUT PINS	6
DC CURRENT PER I/O PIN	20 mA
DC CURRENT FOR 3.3V PIN	50 mA
FLASH MEMORY	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
CLOCK SPEED	16 MHz
LED_BUILTIN	13
LENGTH	68.6 mm
WIDTH	53.4 mm
WEIGHT	25 g

- **MCP 2515 CAN CONTROLLER:-**



Microchip Technology's MCP2515-I/SO is a stand alone Controller Area Network (CAN) controller. It is capable of transmitting and receiving both standard and extended data and remote frames. The MCP2515-I/SO has two acceptance masks and six acceptance filters that are used to filter out unwanted messages, thereby reducing the host MCU's overhead. The MCP2515-I/SO interfaces with microcontrollers (MCUs) via an industry standard Serial

Peripheral Interface (SPI).

- 0-8 byte length in the data field
- Standard and extended data and remote frames
- Two receive buffers with prioritised message storage
- Six 29 bit filters
- Two 29 bit masks
- Three transmit buffers with prioritisation and abort features
- High speed SPI interface (10 MHz)
- One-shot mode ensures message transmission is attempted only one time
- Clock out pin with programmable prescaler and can be used as a clock source for other devices
- Start-of-frame signal available for monitoring the SOF Signal
- Interrupt output pin with selectable enables
- Low power CMOS Technology

Schematic of MCP2515 CAN Bus Module

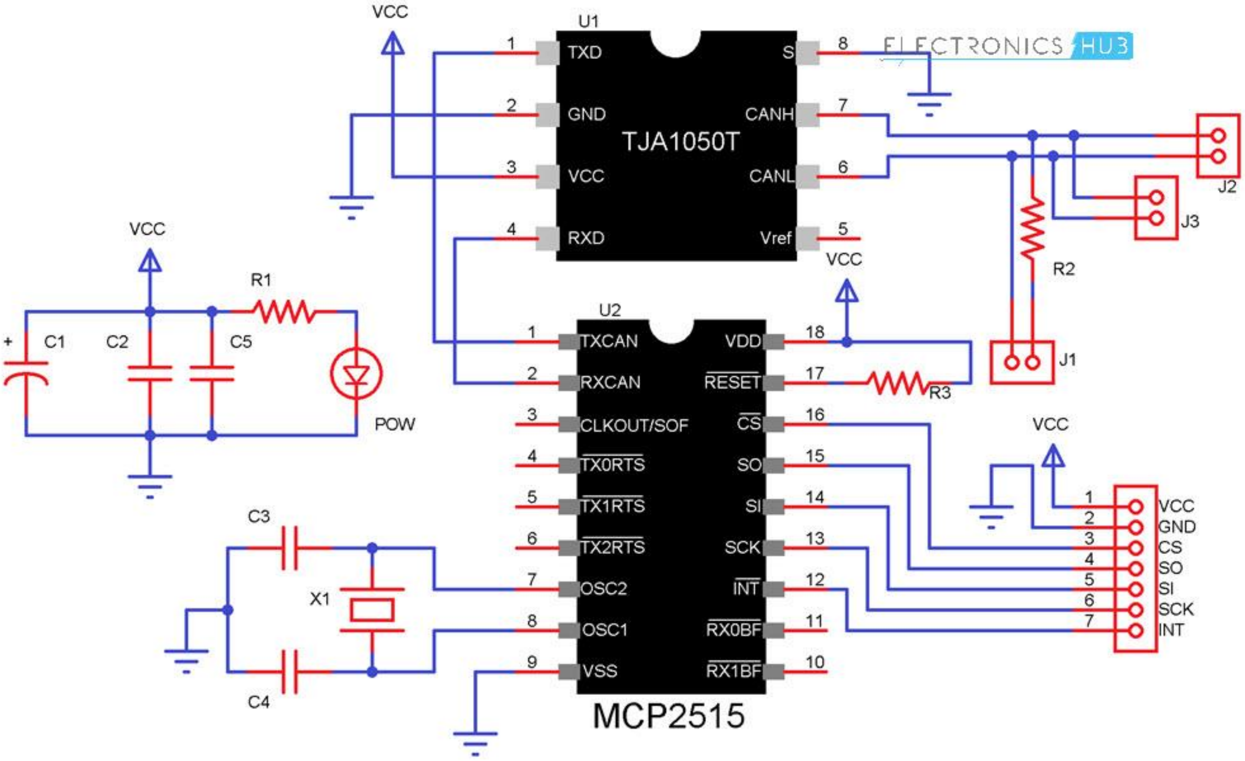
Before seeing the schematic of the module, you need to understand a couple of things about both the ICs i.e. MCP2515 and TJA1050.

MCP2515 IC is the main controller that internally consists of three main subcomponents: The CAN Module, the Control Logic and the SPI Block.

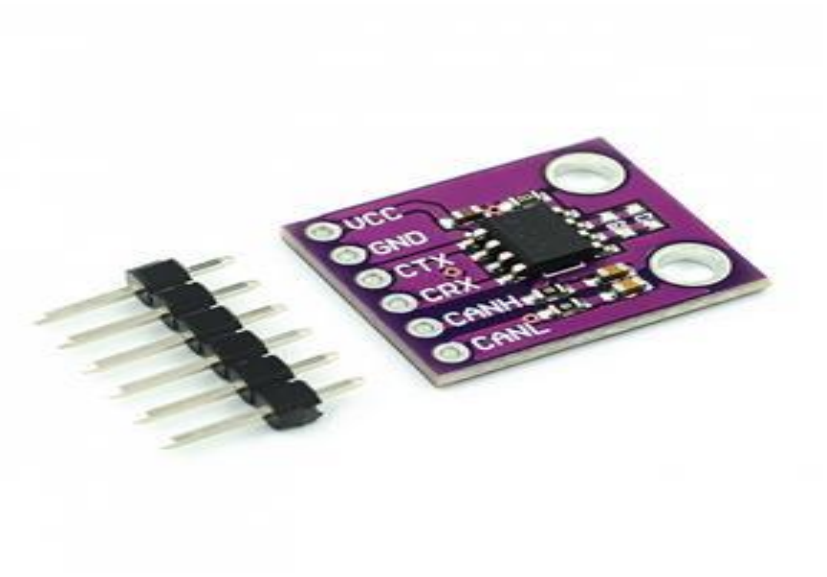
CAN Module is responsible for transmitting and receiving messages on the CAN Bus. Control Logic handles the setup and operation of the MCP2515 by interfacing all the blocks. The SPI Block is responsible for the SPI Communication interface.

Coming to the TJA1050 IC, since it acts as an interface between MCP2515 CAN Controller and the physical CAN Bus, this IC is responsible for taking the data from the controller and relaying it on to the bus.

The following image shows the schematic of the MCP2515 CAN Module and it shows how MCP2515 IC and TJA1050 IC are connected on the Module.



MCP2551 High-Speed CAN Transceiver:-



The MCP2551-E/P is a high speed CAN (Controller Area Network) transceiver in 8 pin DIP package. This CAN fault tolerant device serves as the interface between CAN protocol controller and physical bus. The MCP2551 provides differential transmit and receive capability for CAN protocol controller and is fully compatible with ISO-11898 standard including 24V requirements. It will operate at speed of up to 1Mb/s. Typically each node in CAN system must have device to convert digital signals generated by CAN controller to signals suitable for transmission over the bus cabling (differential output). It also provides buffer between CAN controller and high voltage spikes that can be generated on CAN bus by outside sources (EMI, ESD, electrical transients). The MCP2551 CAN outputs will drive a minimum load of 45ohm allowing a maximum of 112 nodes to be connected.

FEATURES:-

- Slope control input
- Supports 1 Mb/s operation
- Implements ISO-11898 standard physical layer requirements
- Suitable for 12V and 24V systems
- Externally-controlled slope for reduced RFI emissions
- Permanent dominant detect
- Low current standby operation
- High noise immunity due to differential bus implementation

Operating Modes

The RS pin allows three modes of operation to be selected:
• High-Speed
• Slope-Control
• Standby
These modes are summarized in Table 1-1.When in High-speed or Slope-control mode, the drivers for the CANH and CANL signals are internally regulated to provide controlled symmetry in order to minimize EMI emissions. Additionally, the slope of the signal transitions on CANH and CANL can be controlled with a resistor connected from pin 8 (RS) to ground, with the slope proportional to the current output at RS, further reducing EMI emissions.

HIGH-SPEED

High-speed mode is selected by connecting the RS pin to VSS. In this mode, the transmitter output drivers have fast output rise and fall times to support high-speed CAN bus rates.

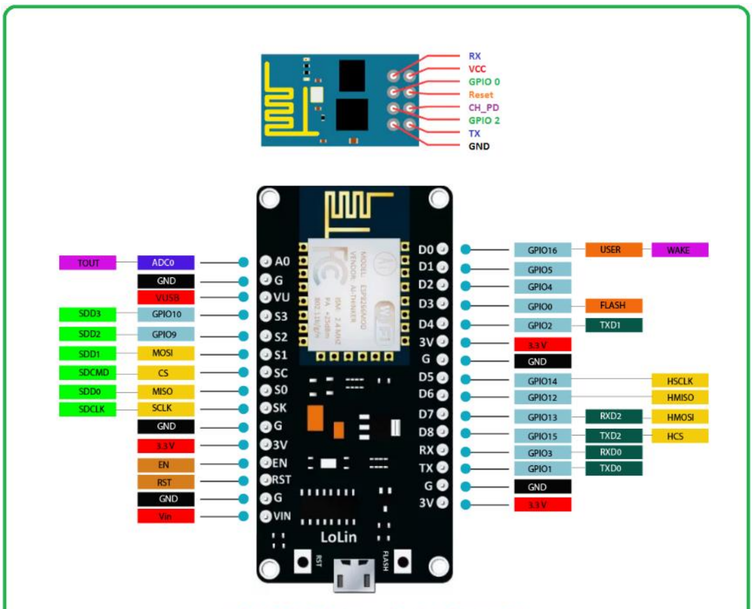
SLOPE-CONTROL

Slope-control mode further reduces EMI by limiting the rise and fall times of CANH and CANL. The slope, or slew rate (SR), is controlled by connecting an external resistor (REXT) between RS and VOL (usually ground). The slope is proportional to the current output at the RS pin. Since the current is primarily determined by the slope-control resistance value REXT, a certain slew rate is achieved by applying a respective resistance.

STANDBY MODE

The device may be placed in standby or “SLEEP” mode by applying a high-level to RS. In SLEEP mode, the transmitter is switched off and the receiver operates at a lower current. The receive pin on the controller side (RXD) is still functional but will operate at a slower rate. The attached microcontroller can monitor RXD for CAN bus activity and place the transceiver into normal operation via the RS pin (at higher bus rates, the first CAN message may be lost).

ESP8266 NODE MCU :-



ESP8266 NodeMCU V3

The ESP8266 WiFi Module is a self contained SOC with integrated TCP/IP protocol stack that can give any microcontroller access to your WiFi network. The ESP8266 is capable of either hosting an application or offloading all Wi-Fi networking functions from another application processor. Each ESP8266 module comes pre-programmed with an AT command set firmware, meaning, you can simply hook this up to your Arduino device and get about as much WiFi-ability as a WiFi Shield offers (and that's just out of the box)! The ESP8266 module is an extremely cost effective board with a huge, and ever growing, community.

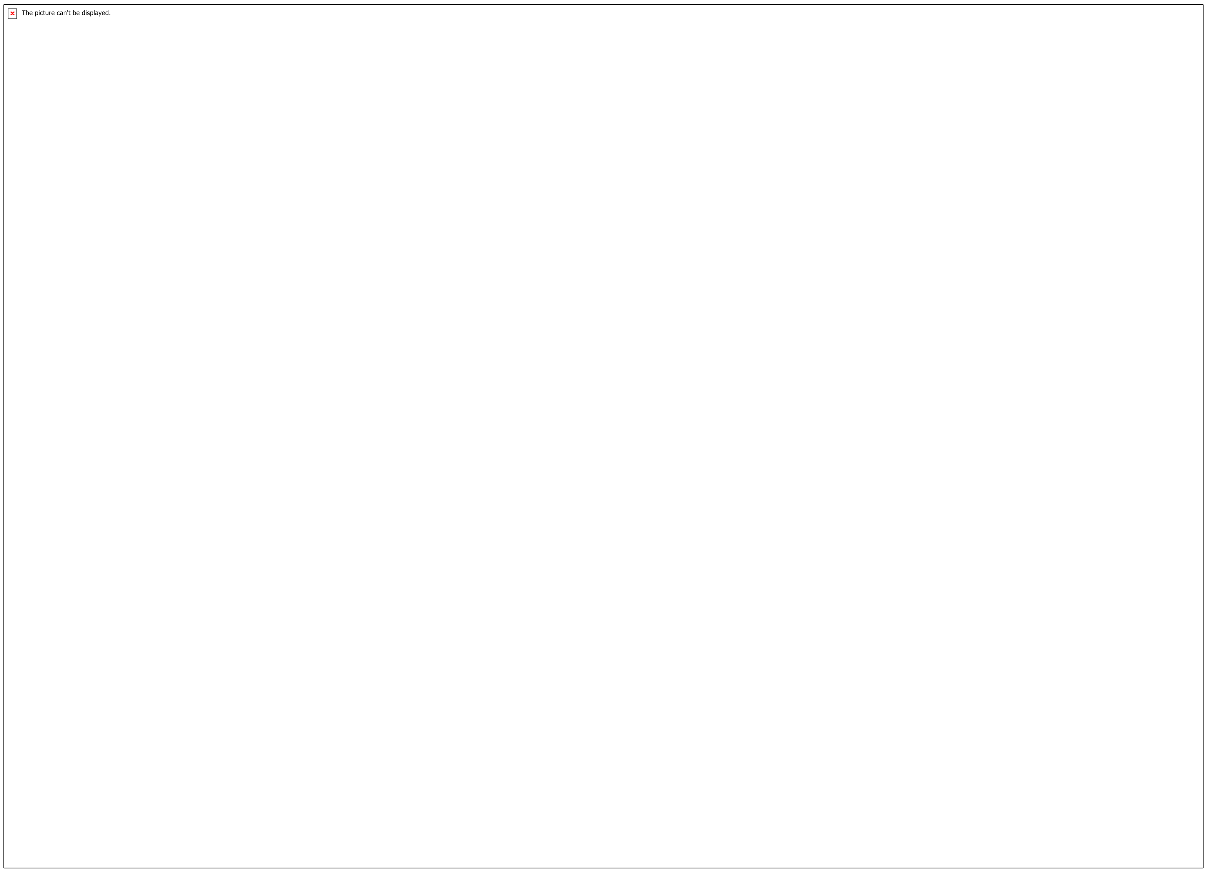
This module has a powerful enough on-board processing and storage capability that allows it to be integrated with the sensors and other application specific devices through its GPIOs with minimal development up-front and minimal loading during runtime. Its high degree of on-chip integration allows for minimal external circuitry, including the front-end module, is designed to occupy minimal PCB area. The ESP8266 supports APSD for VoIP applications and Bluetooth co-existance interfaces, it contains a self-calibrated RF allowing it to work under all operating conditions, and requires no external RF parts.

There is an almost limitless fountain of information available for the ESP8266, all of which has been provided by amazing community support. In the *Documents* section below you will find many resources to aid you in using the ESP8266, even instructions on how to transforming this module into an IoT (Internet of Things) solution

Features:-

- 802.11 b/g/n
- Wi-Fi Direct (P2P), soft-AP
- Integrated TCP/IP protocol stack
- Integrated TR switch, balun, LNA, power amplifier and matching network
- Integrated PLLs, regulators, DCXO and power management units
- +19.5dBm output power in 802.11b mode
- Power down leakage current of <10uA
- Integrated low power 32-bit CPU could be used as application processor
- SDIO 1.1 / 2.0, SPI, UART
- STBC, 1×1 MIMO, 2×1 MIMO
- A-MPDU & A-MSDU aggregation & 0.4ms guard interval
- Wake up and transmit packets in < 2ms
- Standby power consumption of < 1.0mW (DTIM3)
- Default baudrate: 115200

STM32F407G :-



The STM32F4DISCOVERY Discovery kit allows users to easily develop applications with the STM32F407VG high-performance microcontroller with the Arm® Cortex®-M4 32-bit core. It includes everything required either for beginners or experienced users to get started quickly.

Based on STM32F407VG, it includes an ST-LINK/V2-A embedded debug tool, one STMEMS digital accelerometer, one digital microphone, one audio DAC with integrated class D speaker driver, LEDs, push-buttons and a USB OTG Micro-AB connector. Specialized addon boards can be connected by means of the extension header connectors. The STM32F4DISCOVERY Discovery kit comes with the STM32 comprehensive free software libraries and examples available with the STM32CubeF4 MCU Package.

Features:-

The STM32F4DISCOVERY offers the following features:

- STM32F407VGT6 microcontroller featuring 32-bit Arm®(a) Cortex®-M4

with FPU core,
1-Mbyte Flash memory, 192-Kbyte RAM in an LQFP100 package
• USB OTG FS
• ST MEMS 3-axis accelerometer
• ST-MEMS audio sensor omni-directional digital microphone

• Audio DAC with integrated class D speaker driver
• User and reset push-buttons

• Eight LEDs:
  – LD1 (red/green) for USB communication
  – LD2 (red) for 3.3 V power on
  – Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
  – Two USB OTG LEDs, LD7 (green) VBUS and LD8 (red) over-current
• Board connectors:
  – USB with Micro-AB
  – Stereo headphone output jack
  – 2.54 mm pitch extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing

• Flexible power-supply options: ST-LINK, USB VBUS, or external sources
• External application power supply: 3 V and 5 V

• Comprehensive free software including a variety of examples, part of STM32CubeF4 MCU Package, or STSW-STM32068 for using legacy standard libraries

• On-board ST-LINK/V2-A debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port

• Support of a wide choice of Integrated Development Environments (IDEs) including IAREmbedded Workbench®, MDK-ARM, and STM32CubeIDE

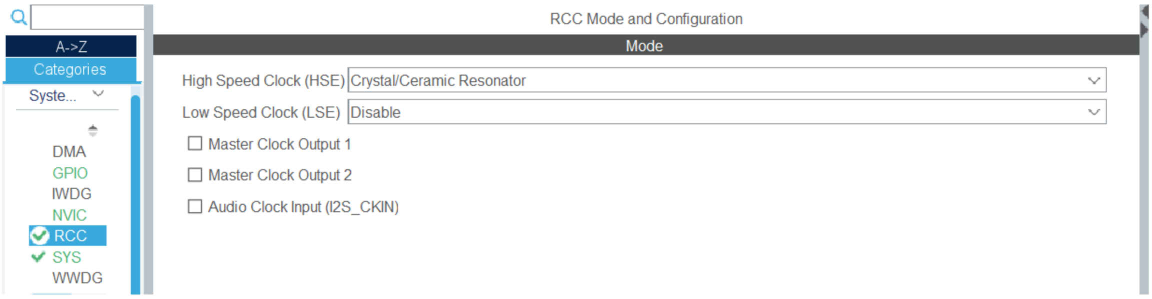
SOURCE CODE EXPLAINATION

Source code of this project is divided in 3 parts and each part is implemented on separate nodes.

A. STM32f407g Node

- **STM32CUBE IDE Settings**

1. We need a stable clock so we have to select HSE.



RCC Mode and Configuration

High Speed Clock (HSE) Crystal/Ceramic Resonator

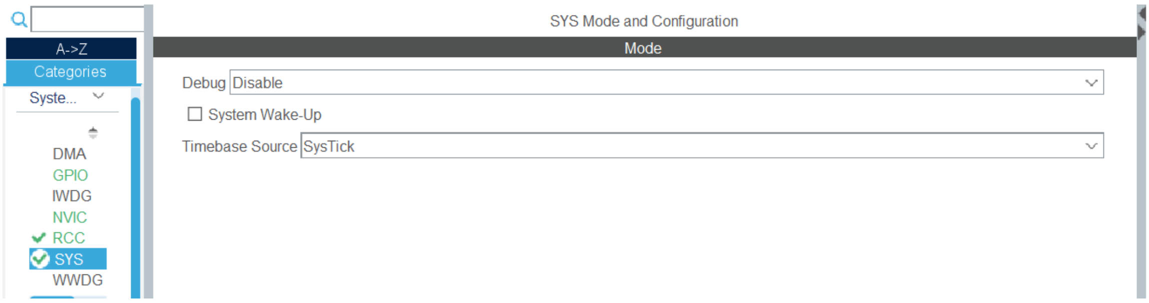
Low Speed Clock (LSE) Disable

☐ Master Clock Output 1

☐ Master Clock Output 2

☐ Audio Clock Input (I2S_CKIN)

2. Select Timebase Source as Systick.



SYS Mode and Configuration

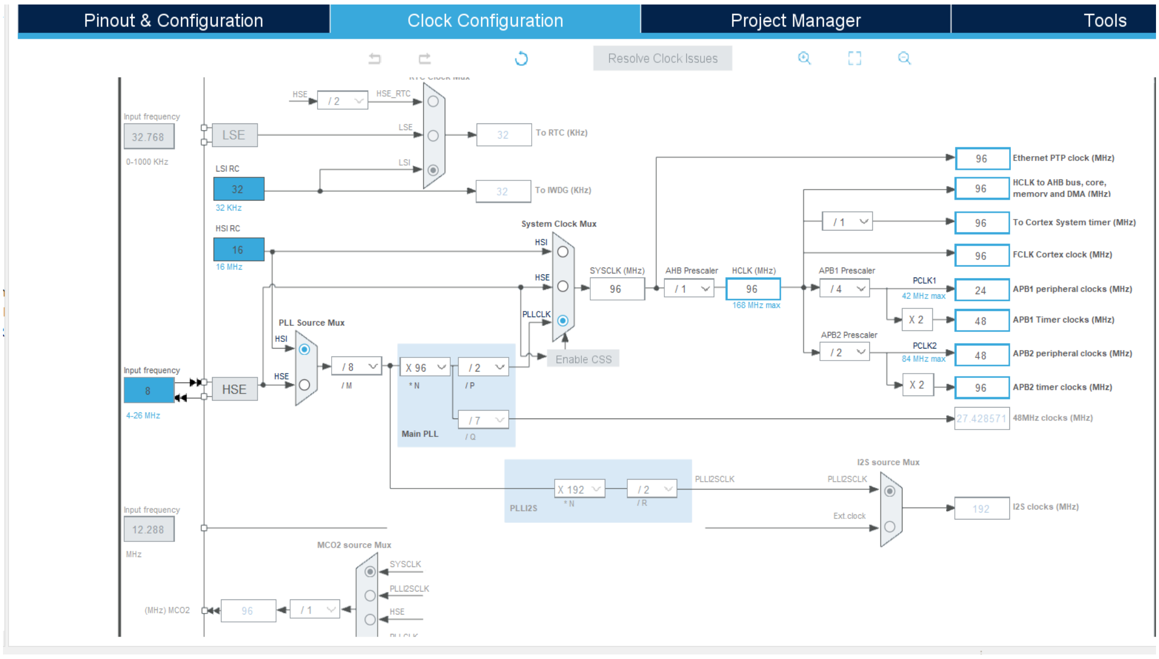
Debug Disable

☐ System Wake-Up

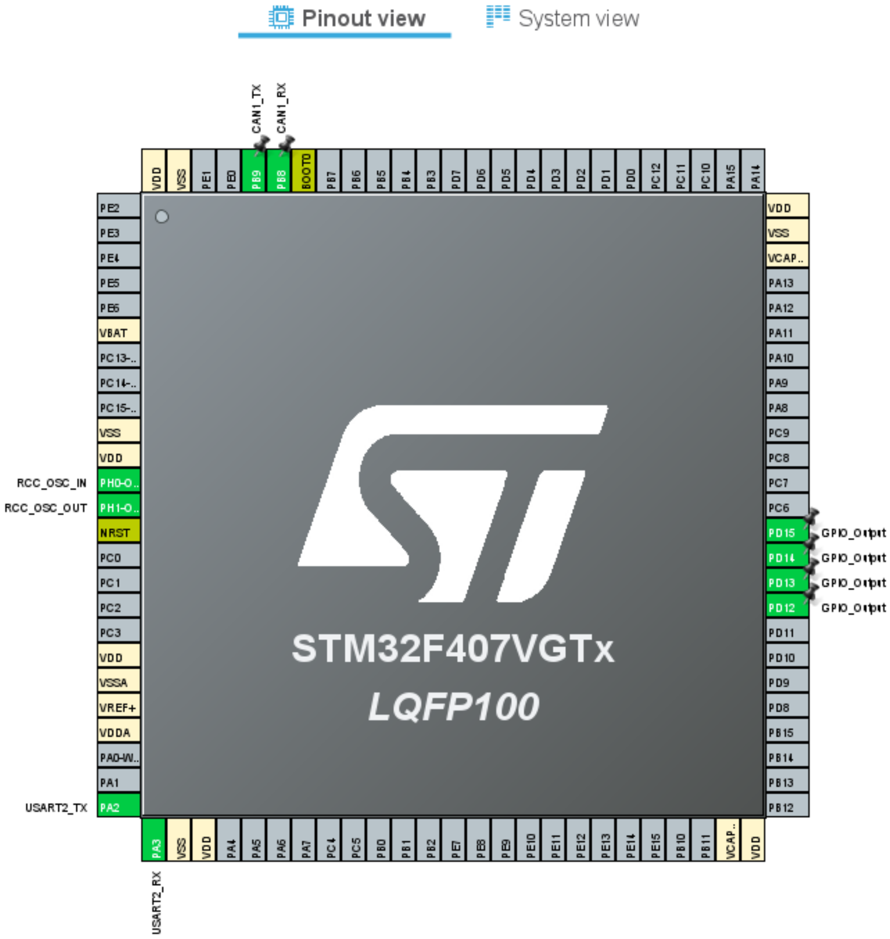
Timebase Source SysTick

3. Clock Settings

- This is the Clock Configuration to achieve **24 Mhz** at APB1.
- External Oscillator is on 8 MHz, APB1 for CAN operates on 24Mhz.



4. Pinout View



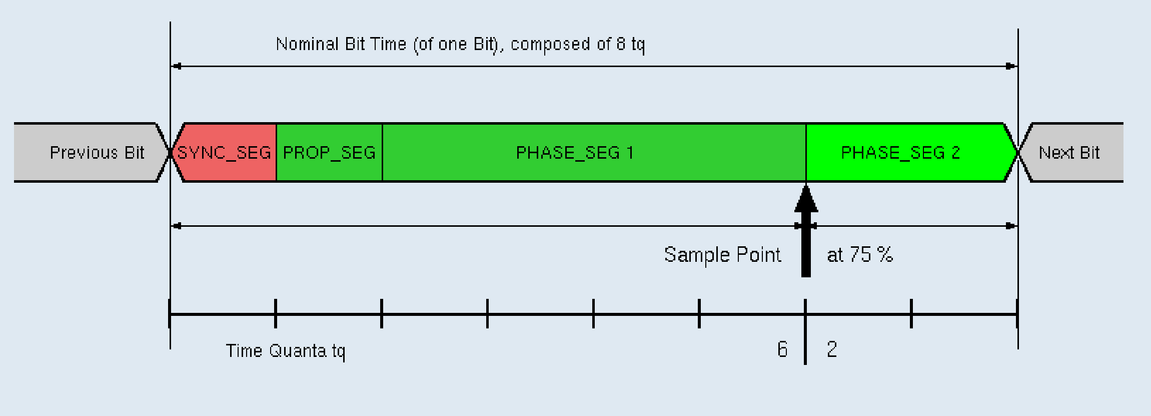
5. CAN Initialization (Prescaler, Time Qantas, Mode Selection)

Basically the CAN bit period can be subdivided into four time segments. Each time segment consists of a number of Time Quanta (tq).
The Time Quanta(Tq) is the smallest time unit for all configuration values.

#Configuring Pre Scalar for Tq
Example:
For Baud rate of 125Kbps if Pre-scalar = 12, Clock APB1=24Mhz -> 24Mhz/12 = 2Mhz=0.5us =500ns

- **SYNC_SEG** is 1 Time Quantum long. It is used to synchronize the various bus nodes.
- **PROP_SEG** is programmable to be 1, 2,... 8 Time Quanta long. It is used to compensate for signal delays across the network.
- **PHASE_SEG1** is programmable to be 1,2, ... 8 Time Quanta long. It is used to compensate for edge phase errors and may be lengthened during resynchronization.
- **PHASE_SEG2** (Seg 2) is the maximum of PHASE_SEG1 and the Information Processing Time long. It is also used to compensate for edge phase errors and may be shortened during resynchronization. For this the minimum value of PHASE_SEG2 is the value of SJW.

The picture shows you the time segments of a CAN-Bit as defined by ISO-11898.



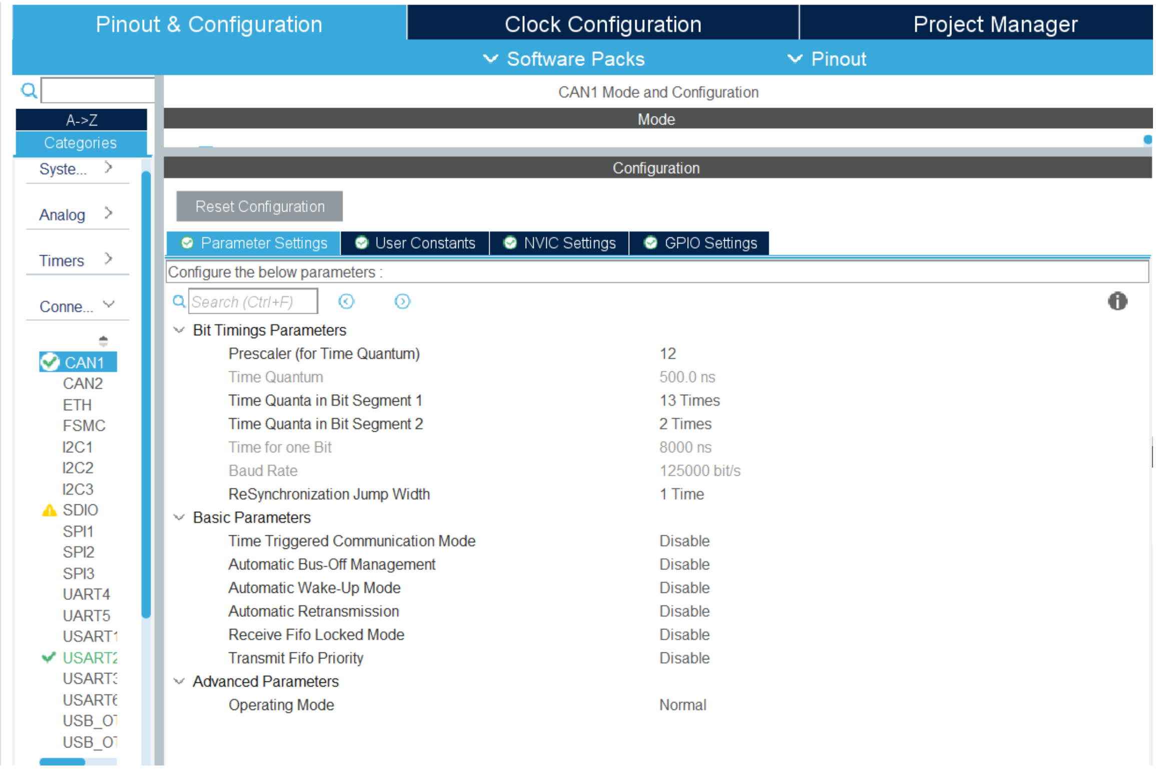
Sync_Seg:	1 tq
Prop_Seg + Phase_Seg1:	1 .. 16 tq

Phase_Seg2:	1 .. 8 tq
(Table calculation uses Prop_Seg = 0)	

Bit Rate Calculation Examples

Q. Calculate Required Time Quanta to achieve different baud rate if the system clock is 24mhz and CAN Pre Scalar is 24 or 12 or 3.

Input Freq. at APB1 Bus	24mhz	24mhz	24mhz
Prescaler	24	12	3
CAN Clock(fcan)	24mhz/24=1mhz	24/12=2mhz	24/3=8mhz
Time Quanta(tq)=1/fcan	1/1mhz=1000ns	1/2mhz=500ns	1/8mhz=125ns
Bit rate	**62.5kbps**	**125kbps**	**500kbps**
Bit time =1/Bit rate	1/62.5kbps=16000ns	1/125kbps=8000ns	1/500kbps=2000ns
No of tq= Bit time/tq	16000ns/1000ns=16tq	8000ns/500ns=16tq	2000ns/125ns=16tq
TqSEG1	13	13	13
TqSEG2	2	2	2
ReSync	1	1	1



CAN1 Mode and Configuration

Configure the below parameters :

Bit Timings Parameters

Prescaler (for Time Quantum)	12
Time Quantum	500.0 ns
Time Quanta in Bit Segment 1	13 Times
Time Quanta in Bit Segment 2	2 Times
Time for one Bit	8000 ns
Baud Rate	125000 bit/s
ReSynchronization Jump Width	1 Time

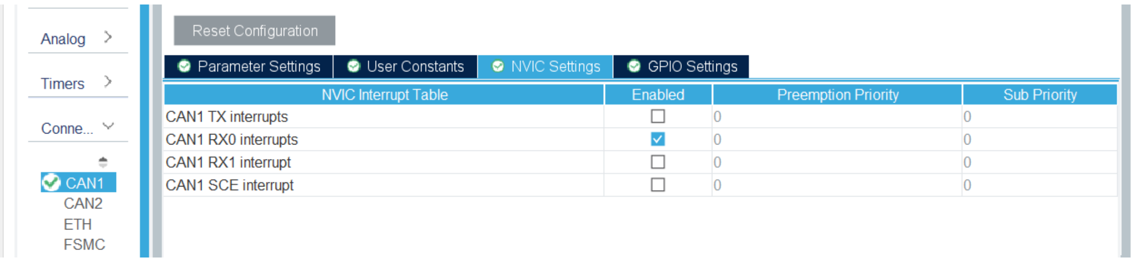
Basic Parameters

Time Triggered Communication Mode	Disable
Automatic Bus-Off Management	Disable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Disable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Disable

Advanced Parameters

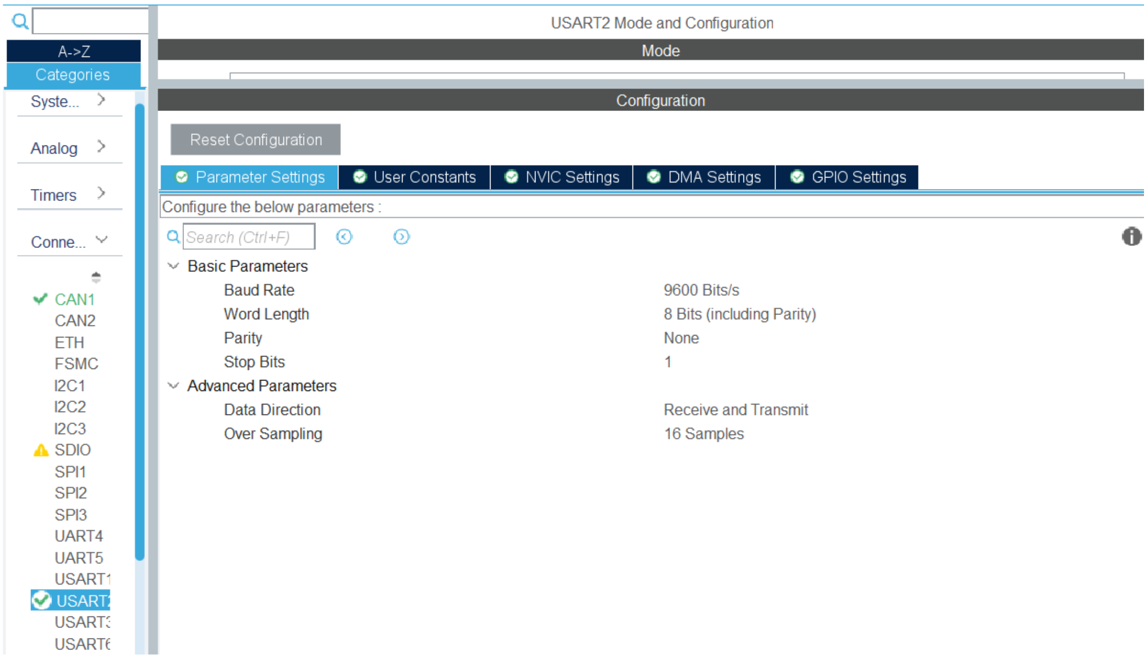
Operating Mode	Normal

6. Also Enable CANRx Interrupt & set subpriority



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
CAN1 TX interrupts	□	0	0
CAN1 RX0 interrupts	☑	0	0
CAN1 RX1 interrupt	□	0	0
CAN1 SCE interrupt	□	0	0

7. UART configuration



Configure the below parameters :

Basic Parameters

Baud Rate	9600 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Advanced Parameters

Data Direction	Receive and Transmit
Over Sampling	16 Samples

• Header files , Private Variables and function

- These header files used in code.

```
#include "main.h"
#include<stdio.h>
#include<string.h>
```

- These are the private variables which are for data and mailbox.

```
CAN_HandleTypeDef hcan1;
UART_HandleTypeDef huart2;
CAN_TxHeaderTypeDef TxHeader1;
CAN_RxHeaderTypeDef RxHeader;
uint8_t TxData1[8] , RxData[8];
uint32_t TxMailbox1;
```

- UartTask function is used for sending data to esp8266 which is received by CAN bus.

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN1_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */
void UartTask(uint8_t *ptr);
/* USER CODE END PFP */
```

- **main function**

we have to start can with HAL_CAN_Start(); and HAL_CAN_ActivateNotification() ;
which starts can and notify about pending interrupts.

```
int main(void)
{

  /* MCU Configuration--------------------------------------------------------
---------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_CAN1_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */
  HAL_CAN_Start(&hcan1);

  HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);

        TxHeader1.StdId = 0x123;
        TxHeader1.RTR = CAN_RTR_DATA;
        TxHeader1.IDE = CAN_ID_STD;
        TxHeader1.DLC = 2;
        TxHeader1.TransmitGlobalTime = DISABLE;
        TxData1[0] = 0x11;
        TxData1[1] = 0x00;
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  }

}
```

- **CAN Acceptance filter Configuration (filter bank, FilterIdHigh, FilterIdLow etc..)**

```
  /* USER CODE BEGIN CAN1_Init 2 */
  CAN_FilterTypeDef FilterConfig;

  FilterConfig.FilterActivation = CAN_FILTER_ENABLE;

  FilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;

  FilterConfig.SlaveStartFilterBank = 14;

  FilterConfig.FilterBank = 10;// Any number from 0 to
SlaveStartFilterBank

  FilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;

  FilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
      //CAN_FILTERMODE_IDLIST;
  FilterConfig.FilterMaskIdLow = 0x0000;

  FilterConfig.FilterMaskIdHigh = 0x07F8 << 5;//id1 = 0x07FA << 5

  FilterConfig.FilterIdLow = 0x0000;

  FilterConfig.FilterIdHigh = 0x00A8 << 5;  //id2 = 0x07FB << 5

  HAL_CAN_ConfigFilter(&hcan1, &FilterConfig);
  /* USER CODE END CAN1_Init 2 */
```

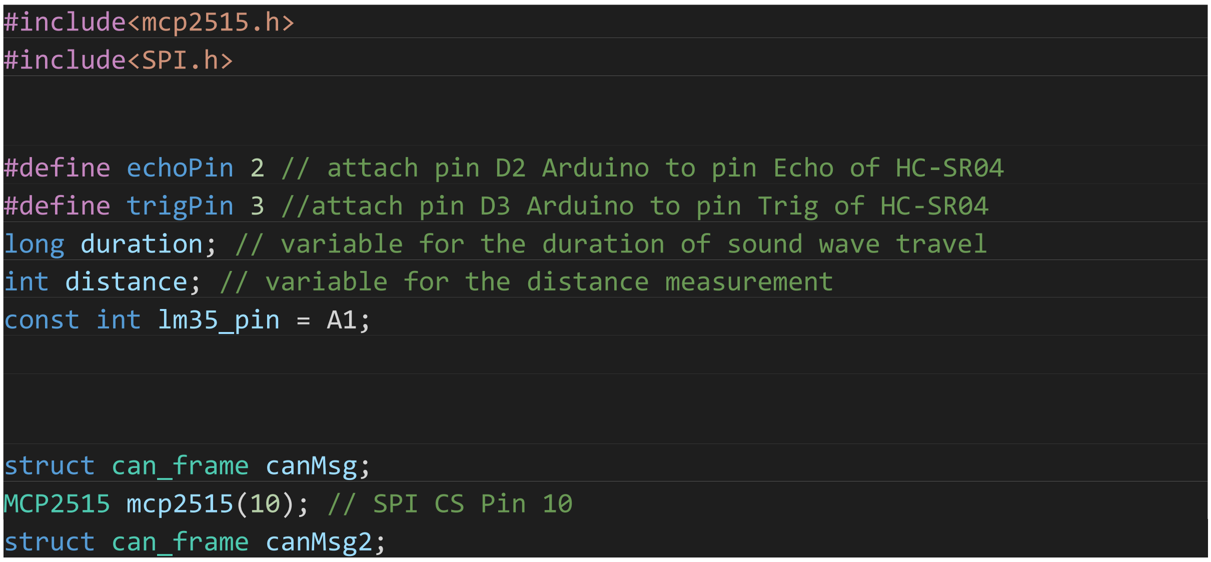
- **Callback for Interrupt and UartTask function Definition**

```
/* USER CODE BEGIN 4 */
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
	if( HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData)
!= HAL_OK ){
		Error_Handler();
	}
	if((RxHeader.StdId == 0x0A8) && (RxHeader.IDE == CAN_ID_STD) &&
(RxHeader.DLC == 8)){
		UartTask(RxData);
	}

}
void UartTask(uint8_t *ptr)
{
	char buffer[300];
	sprintf(buffer ,"%d ,%d , %d*", ptr[0] , ptr[1] , ptr[2]);
	HAL_UART_Transmit(&huart2, (uint8_t*)buffer, strlen(buffer),
HAL_MAX_DELAY);
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error
return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
```

B. Arduino Node

- In this node we have interfaced different sensors with Arduino and mcp2515 is also interfaced with Arduino.So we read sensor’s data from different nodes and we set that values in can data and send that data over CAN Bus to STM32f407g.

1. Here we have used Arduino-mcp2515-master Library and initialize some pins.



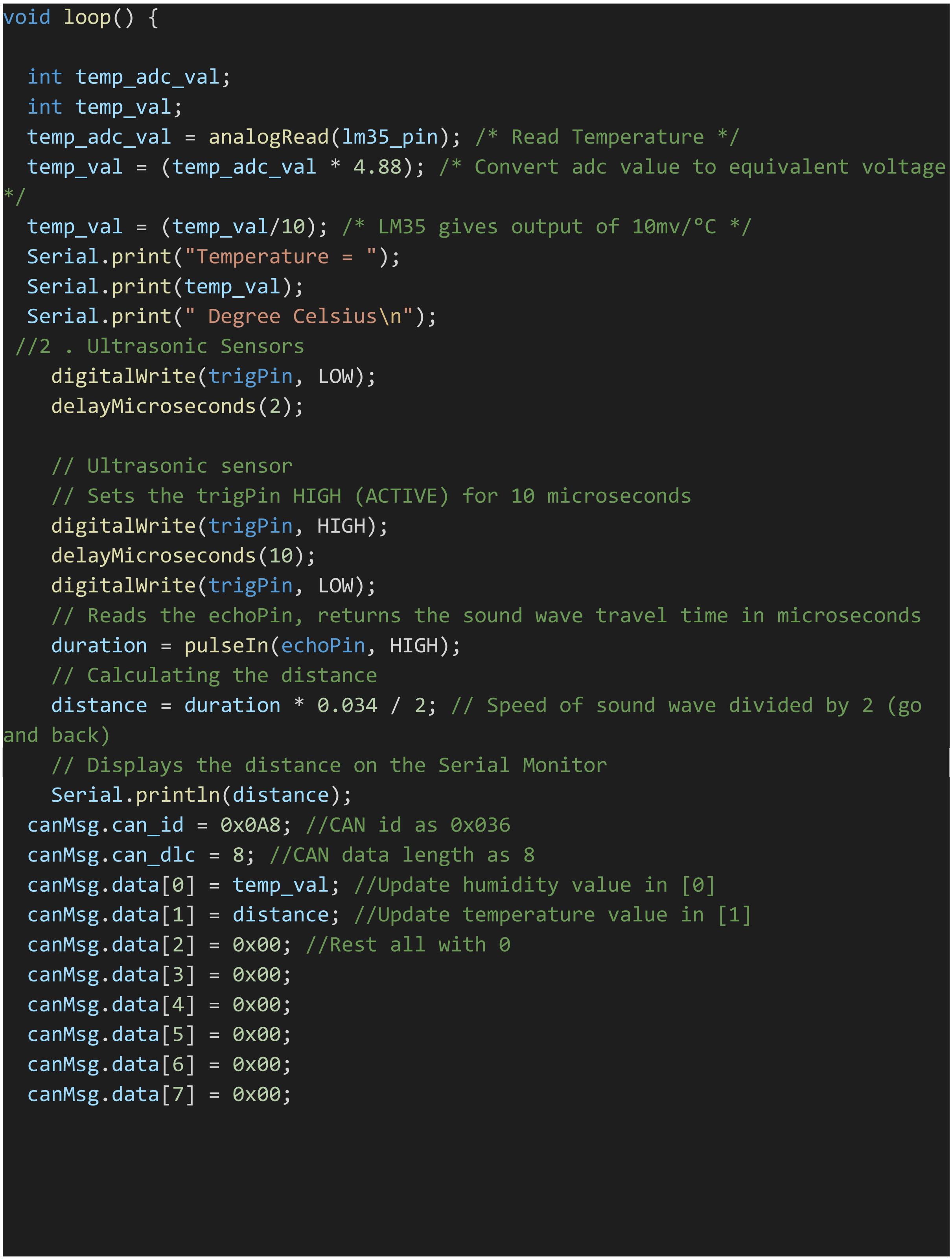
```
#include<mcp2515.h>
#include<SPI.h>


#define echoPin 2 // attach pin D2 Arduino to pin Echo of HC-SR04
#define trigPin 3 //attach pin D3 Arduino to pin Trig of HC-SR04
long duration; // variable for the duration of sound wave travel
int distance; // variable for the distance measurement
const int lm35_pin = A1;



struct can_frame canMsg;
MCP2515 mcp2515(10); // SPI CS Pin 10
struct can_frame canMsg2;
```

2. In Void Setup() we have to set the pinmodes and set the baud rate in Serial.begin(). We have to set CAN bus bit rate which is 125 kbps and set mode to Normal mode.

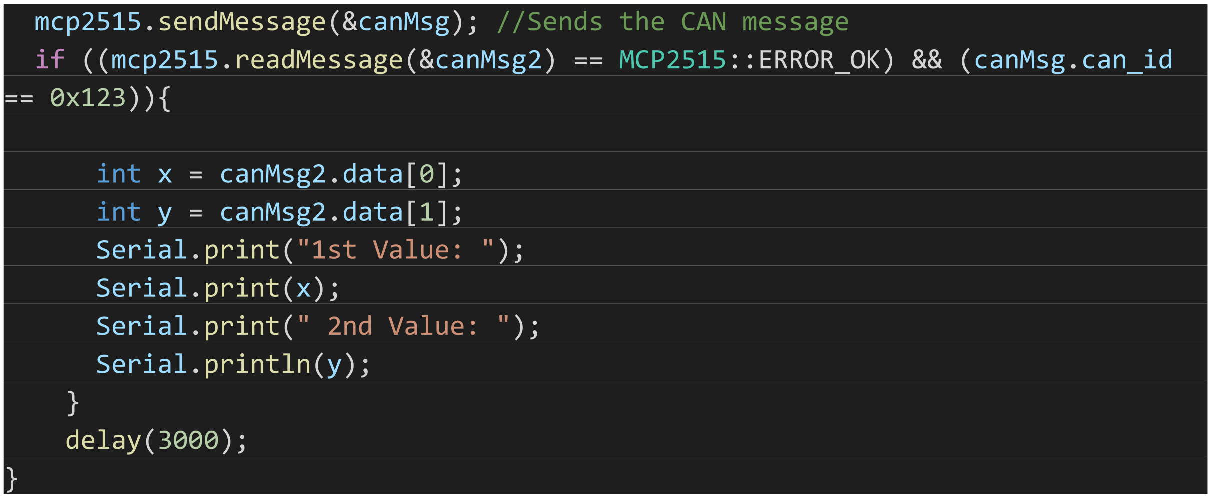
```
void setup() {
    Serial.begin(9600);


    pinMode(trigPin, OUTPUT); // Sets the trigPin as an OUTPUT
    pinMode(echoPin, INPUT); // Sets the echoPin as an INPUT

  SPI.begin();   //Begins SPI communication

    Serial.begin(9600); //Begins Serial Communication at 9600 baud rate
    mcp2515.reset();
    mcp2515.setBitrate(CAN_125KBPS,MCP_8MHZ); //Sets CAN at speed 500KBPS and
Clock 8MHz
    mcp2515.setNormalMode();  //Sets CAN at normal mode
    Serial.println("LM35");
    Serial.println("Ultrasonic Sensor HC-SR04 "); // print some text in Serial
Monitor

}
```

3. In Void Loop

In Void Loop() we Have to take sensor data in this and put that data in can frame and that can frame we have to sent on CAN Bus.



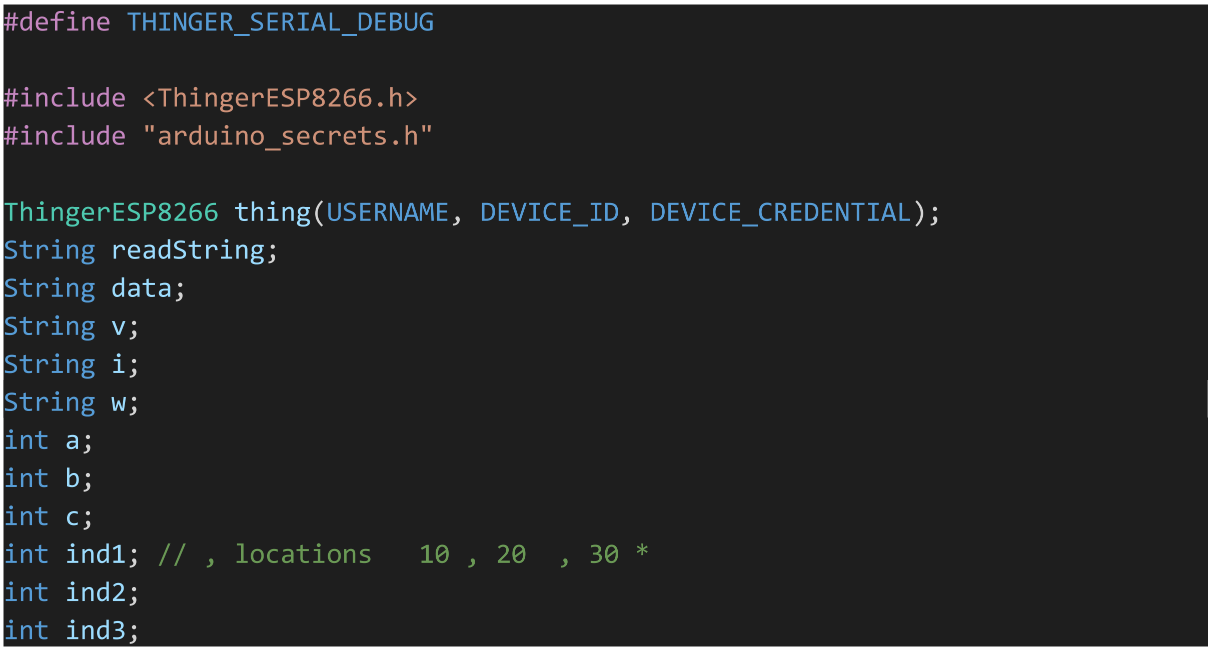
```
void loop() {

  int temp_adc_val;
  int temp_val;
  temp_adc_val = analogRead(lm35_pin); /* Read Temperature */
  temp_val = (temp_adc_val * 4.88); /* Convert adc value to equivalent voltage
*/
  temp_val = (temp_val/10); /* LM35 gives output of 10mv/°C */
  Serial.print("Temperature = ");
  Serial.print(temp_val);
  Serial.print(" Degree Celsius\n");
 //2 . Ultrasonic Sensors
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    // Ultrasonic sensor
    // Sets the trigPin HIGH (ACTIVE) for 10 microseconds
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
    // Reads the echoPin, returns the sound wave travel time in microseconds
    duration = pulseIn(echoPin, HIGH);
    // Calculating the distance
    distance = duration * 0.034 / 2; // Speed of sound wave divided by 2 (go
and back)
    // Displays the distance on the Serial Monitor
    Serial.println(distance);
  canMsg.can_id = 0x0A8; //CAN id as 0x036
  canMsg.can_dlc = 8; //CAN data length as 8
  canMsg.data[0] = temp_val; //Update humidity value in [0]
  canMsg.data[1] = distance; //Update temperature value in [1]
  canMsg.data[2] = 0x00; //Rest all with 0
  canMsg.data[3] = 0x00;
  canMsg.data[4] = 0x00;
  canMsg.data[5] = 0x00;
  canMsg.data[6] = 0x00;
  canMsg.data[7] = 0x00;
```



```
  mcp2515.sendMessage(&canMsg); //Sends the CAN message
  if ((mcp2515.readMessage(&canMsg2) == MCP2515::ERROR_OK) && (canMsg.can_id
== 0x123)){

      int x = canMsg2.data[0];
      int y = canMsg2.data[1];
      Serial.print("1st Value: ");
      Serial.print(x);
      Serial.print(" 2nd Value: ");
      Serial.println(y);
    }
    delay(3000);
}
```

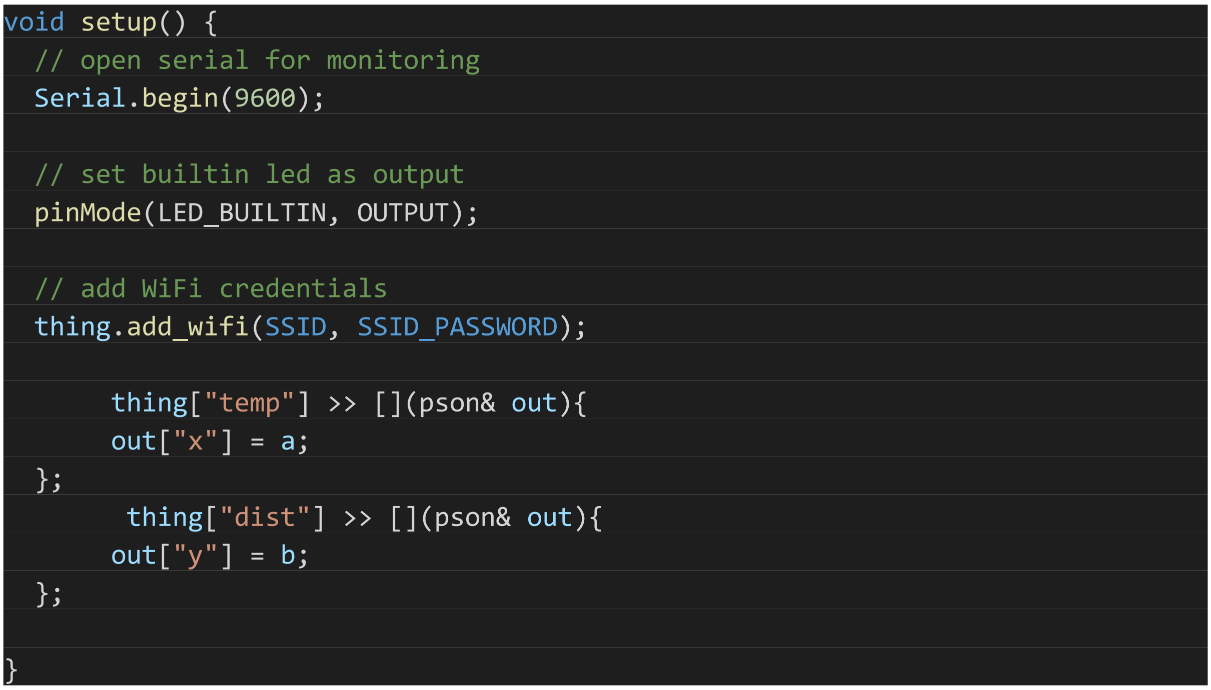
C . ESP8266 Node

1. On this node we recived data from Stm32 using Serial Communication and Post that Data to thinger.io.This node has 2 files. There is one .h file which has credentials for wifi and thinger.io.



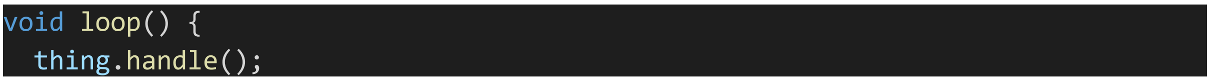
```
#define THINGER_SERIAL_DEBUG

#include <ThingerESP8266.h>
#include "arduino_secrets.h"

ThingerESP8266 thing(USERNAME, DEVICE_ID, DEVICE_CREDENTIAL);
String readString;
String data;
String v;
String i;
String w;
int a;
int b;
int c;
int ind1; // , locations   10 , 20   , 30 *
int ind2;
int ind3;
```

2. On void setup We have to set serial communication Buad rate and Also set resource for thinger.io.

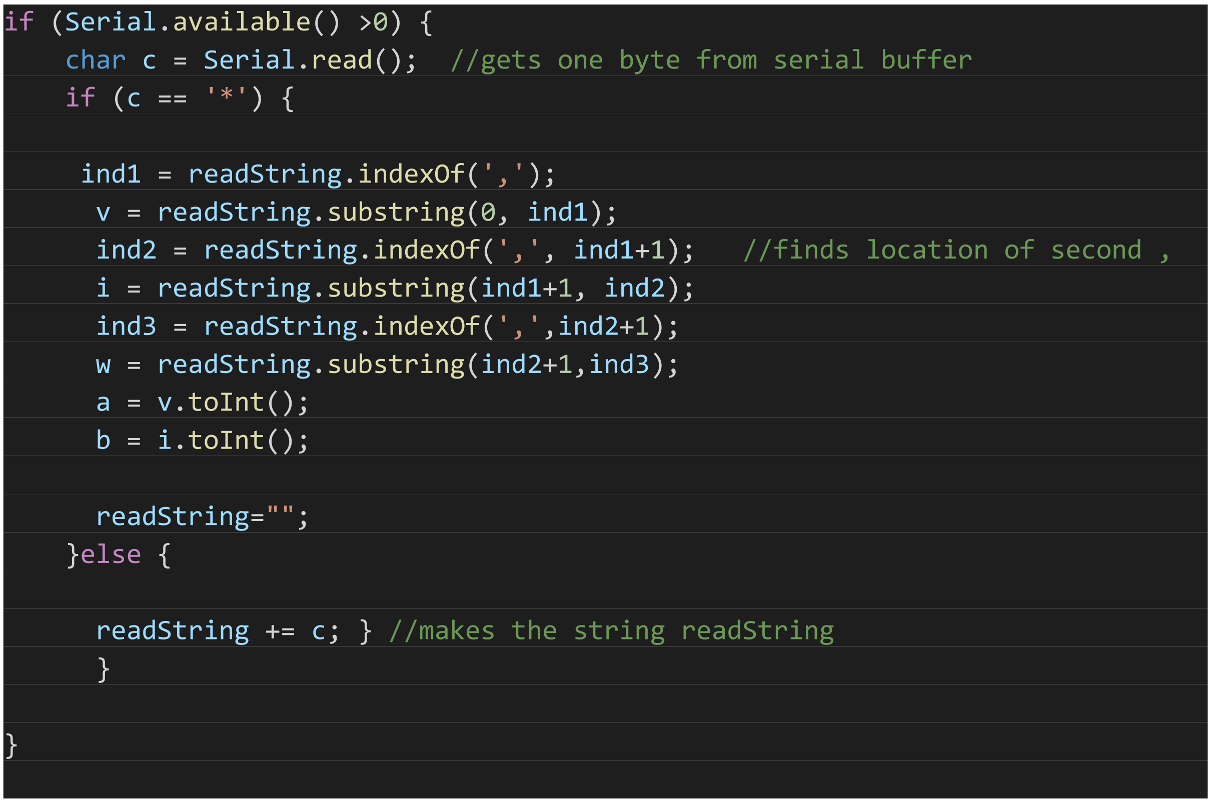


```
void setup() {
  // open serial for monitoring
  Serial.begin(9600);

  // set builtin led as output
  pinMode(LED_BUILTIN, OUTPUT);

  // add WiFi credentials
  thing.add_wifi(SSID, SSID_PASSWORD);

       thing["temp"] >> [](pson& out){
       out["x"] = a;
  };
        thing["dist"] >> [](pson& out){
       out["y"] = b;
  };

}
```

3. In Void Loop We have to Receive Data from stm32 in form of a string and then Split that string for getting sensor data for each sensor. Then data automatically get posted on thinger.io webserver. We have to call thing.handle() function for sending data to webserver.



```
void loop() {
  thing.handle();
```

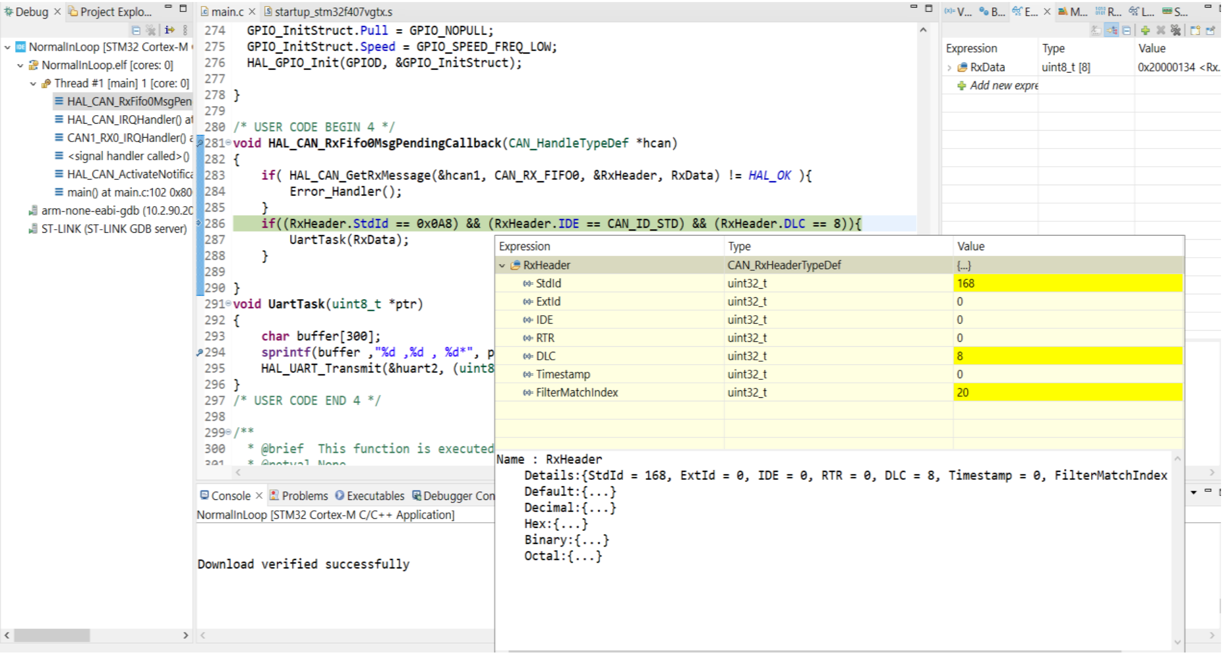


```
if (Serial.available() >0) {
    char c = Serial.read();  //gets one byte from serial buffer
    if (c == '*') {

     ind1 = readString.indexOf(',');
      v = readString.substring(0, ind1);
      ind2 = readString.indexOf(',', ind1+1);   //finds location of second ,
      i = readString.substring(ind1+1, ind2);
      ind3 = readString.indexOf(',',ind2+1);
      w = readString.substring(ind2+1,ind3);
      a = v.toInt();
      b = i.toInt();

      readString="";
    }else {

      readString += c; } //makes the string readString
      }

}
```

This was the Explanation for code on All three nodes.

TESTING

1. For testing the Can Bus on STM32f407g and we have to do debugging. For debugging we Have to set breakpoints in code. For example-



```
NormalInLoop [STM32 Cortex-M
  NormalInLoop.elf [cores: 0]
    Thread #1 [main] 1 [core: 0]
      HAL_CAN_RxFifo0MsgPen
      HAL_CAN_IRQHandler() at
      CAN1_RX0_IRQHandler() a
      <signal handler called>()
      HAL_CAN_ActivateNotifica
      main() at main.c:102 0x80
  arm-none-eabi-gdb (10.2.90.20
  ST-LINK (ST-LINK GDB server)
```

```c
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    if( HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData) != HAL_OK ){
        Error_Handler();
    }
    if((RxHeader.StdId == 0x0A8) && (RxHeader.IDE == CAN_ID_STD) && (RxHeader.DLC == 8)){
        UartTask(RxData);
    }

}
void UartTask(uint8_t *ptr)
{
    char buffer[300];
    sprintf(buffer ,"%d ,%d , %d*", p
    HAL_UART_Transmit(&huart2, (uint8
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed
  * @retval None
```

Expression	Type	Value
RxData	uint8_t [8]	0x20000134 <Rx.
Add new expre		

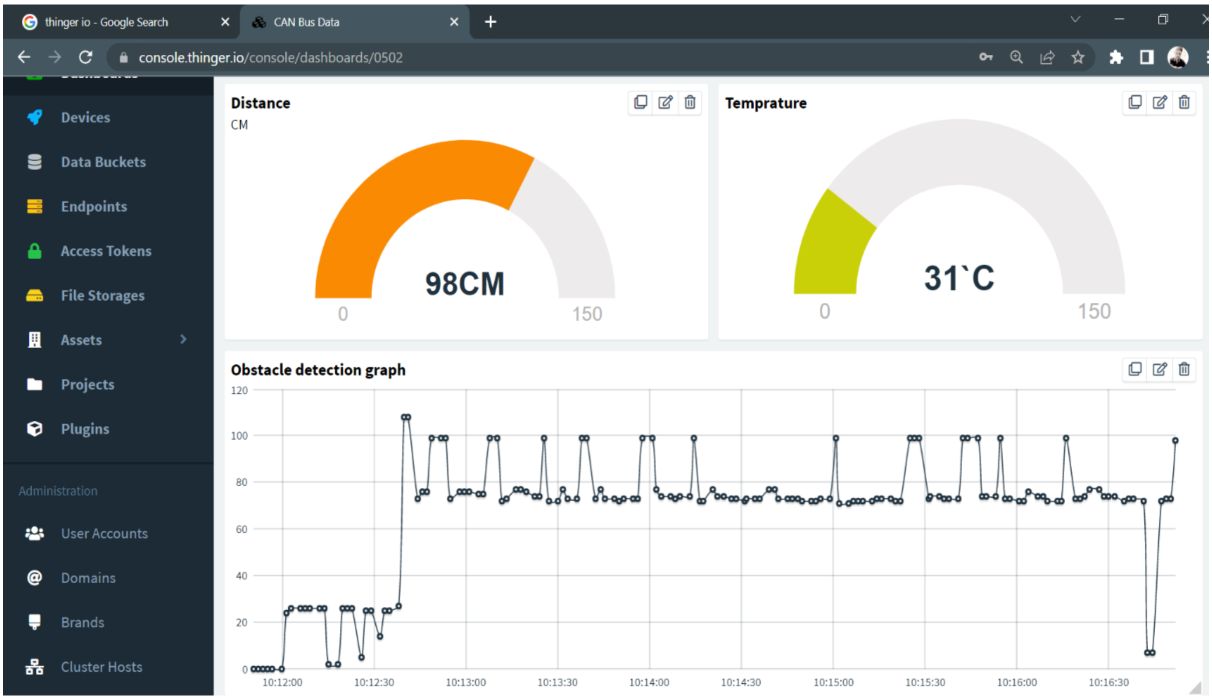
Expression	Type	Value
RxHeader	CAN_RxHeaderTypeDef	{...}
StdId	uint32_t	168
ExtId	uint32_t	0
IDE	uint32_t	0
RTR	uint32_t	0
DLC	uint32_t	8
Timestamp	uint32_t	0
FilterMatchIndex	uint32_t	20

```
Name : RxHeader
    Details:{StdId = 168, ExtId = 0, IDE = 0, RTR = 0, DLC = 8, Timestamp = 0, FilterMatchIndex
    Default:{...}
    Decimal:{...}
    Hex:{...}
    Binary:{...}
    Octal:{...}
```

```
NormalInLoop [STM32 Cortex-M C/C++ Application]

Download verified successfully
```

This is Screenshot of the debugging process in which are receiving sensor values in receiver fifo callback Function.

2. For testing the IOT part we have to set a web server in thinger.io. we have to set a dash board which has some data stream which are taking data from the ESP8266 nodemcu.



In above Screenshot we can see that there are 2 gauges which are connected to data streams and showing output.

There is also a time series chart which helps us to visualizing the sensor values.

FUTURE SCOPES

1. In future want to connect more than one CAN nodes in this setup and try to make it more versatile.

2. We also want to store sensor data into some stable database so that we can truly analysis the data by detecting it’s behavior for a longer period of time.

REFERNCES

[1] US Departement of Energy, How Hybrid Work, Energy Efficiency and Renewable Energy, last modified Tuesday August 19 2014 http://www.fueleconomy.gov/feg/hybridtech.shtml.

[2] Kristian Ismail, Aam Muharam, Amin, Sunarto Kaleg, Desain Test Vehicle untuk Sistem Manajemen Energi Kendaraan Hibrida Seri, Prosiding seminar nasional SMART 2010, hal D-84 - D89, UGM ISBN 978-602-97567-4-6.

[3] Advantech team, RS485 A Proud Legacy, Technicl White paper, advantech Enabling an intelligent, 2012,www.automation.com/pdf_articles/advantech/legacy.pdf.

[4] Kehai Rd, ABOUT CANBUS,FOSHAN SUNDREAM ELECTRONICS CO.,LTD Chancheng Economic Development Zone, Foshan, China, www.incarel.com.

[5] Guang Liu, Ai-ping Xiao, Hua Qian, Communication System Design Based on TMS320F2407 with CAN Bus, AASRI Procedia, Volume 3, 2012, Pages 463-467, ISSN 2212-6716, http://dx.doi.org/10.1016/j.aasri.2012.11.073.

[6] G. Nanda kumar, B. Raj Narain, Portable Embedded Data Display and Control Unit using CAN Bus, Procedia Engineering, Volume 38, 2012, Pages 791-798, ISSN 1877-7058, http://dx.doi.org/10.1016/j.proeng.2012.06.099.

[7] Marco Di Natale, Understanding and using the Controller Area Network, October 30, 2008, http://www6.in.tum.de/pub/Main/TeachingWs2013MSE/CANbus.pdf .

[8] Ewert Energy Systems, Orion BMS Operation Manual, Preliminary Version 0.5 Carol Stream, United States, http://www.orionbms.com/manuals/operational 050814.