

**IOT BASED LOW COST GAS LEAKAGE, FIRE AND TEMPERATURE DETECTION
SYSTEM WITH CALL FACILITIES**

Submitted in partial fulfilment of the requirements for the award of

**Bachelor of Engineering degree in
Electronics and Communication Engineering**

By

SAMEER SHAIK (38130205) & SYED RAJABUDEEN (38130218)



SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

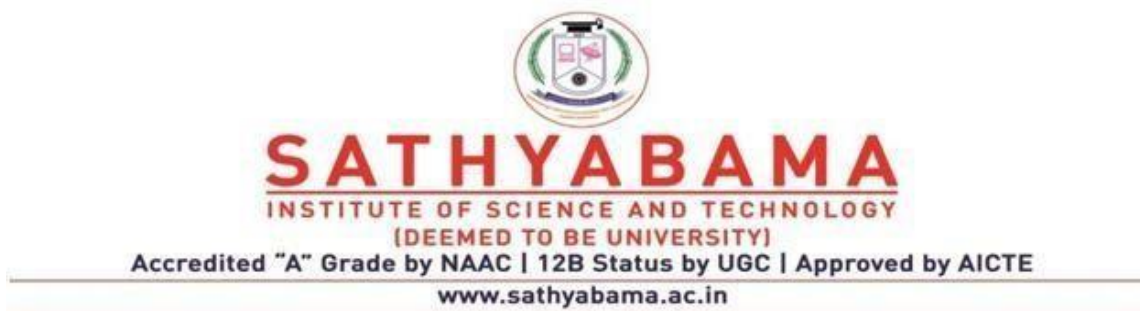
**DEPARTMENT OF ELECTRONICS & COMMUNICATION
ENGINEERING**

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

Accredited with Grade "A" by NAAC

JEPPIAAR NAGAR, RAJIV GANDHI SALAI CHENNAI -600119



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING
BONAFIDE CERTIFICATE**

This is to certify that Project Report is the bonafide work of SAMEER SHAIK-(38130205) & SYED RAJABUDEEN-(38130218) who carried out the Project entitled **"IOT BASED LOW COST GAS LEAKAGE, FIRE AND TEMPERATURE DETECTION SYSTEM WITH CALL FACILITIES"** under the guidance of **"MRS.F.V.JAYASUDHA"** during the period Jan 2022 to March 2022.

INTERNAL GUIDE

MRS.F.V.JAYASUDHA

HEAD OF THE DEPARTMENT

Dr. T. RAVI, M.E, PHD

Submitted for Viva Voce held on

Internal Examiner

Dr. Rekha Chakravarthi Asso.Prof.

External Examiner

Pandian Ramanathan.ME.Phd

DECLARATION

We , SAMEER SHAIK-(38130205) & SYED RAJABUDEEN-(38130218)

Hereby declare that the Project report entitled **IOT BASED LOW COST GAS LEAKAGE, FIRE AND TEMPERATURE DETECTION SYSTEM WITH CALL FACILITIES** is done by us under the guidance of **MRS.F.V.JAYASUDHA.**, is submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering degree in **ELECTRONICS AND COMMUNICATION ENGINEERING.**

DATE:

PLACE: CHENNAI

ACKNOWLEDGEMENT

I pleased to acknowledge my sincere thanks to board management of SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY for their kind encouragement in doing this internship and for completing it successfully. I am grateful to them.

I convey my thanks to MRS.F.V.JAYASUDHA Department of Electrical and Electronics engineering and Dr. T. RAVI, M.E., Ph.D., Head of the Department, Electronics and Communication engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **MRS.F.V.JAYASUDHA** for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my Project.

I wish to express my thanks to all Teaching and Non-teaching staff members of the Department of Electronics and Communication Engineering who were helpful in many ways for the completion of the project.

Abstract:

Gas pipes play very important roles for cities, industries and thus in growing economies. So gas leakages lead to losses as well as are a threat because they can also lead to fire accidents. Placing sensors at each section of pipe is very costly. So here we propose an innovative robot that clings on to the outer surface of the gas pipe and moves with the pipe to check for leakages.

The robot consists of gas sensor that is used to detect gas leakages. As the robot keeps moving along the metal pipe it keeps monitoring for any gas leakage, on detection it uses an interface gps sensor to transmit location of the leakage detected over to the IOT login system, here we use UBIDOTS to receive and display the gas leakage alert and location over IOT.

Thus we have a fully automated insect like robot that moves with the gas pipe and detects gas leakages instantly at a low budget.

TABLE OF CONTENTS

1.INTRODUCTION	7
2.LITERATURE SURVEY	8
3.HARDWARE REQUIRED	12
4.INTRODUCTION OF EMBEDDED SYSTEM	13
5.NODEMCU	22
6.BRIEF ABOUT SENSORS USED	30
7.BRIEF ABOUT IOT	44
8.SOFTWARE REQUIRED	57
9.BRIEF ABOUT UBIDOTS EXPLORER	72
10.STEPS FOR ASSEMBLING	82
11.CODE	87
12.CONCLUSION	90
13.REFERENCES	91

Introduction:

IoT (Internet of Things) is a widely used technology in the industrial revolution. Its growth has been exponential in this decade and is commonly used for measuring and reporting data.

The various applications include healthcare and framework systems with integration of artificial intelligence, industrial developments smart surveillance, smart home systems.

Due to the rise in advancement of technology in industries, there is an enlargement of IoT technology. Understanding the system in gas leakage from, the key idea is to propose and develop a simple gas leakage system using IoT and notifying the user using thing tweet which is an app to link a user's twitter account to the thing speak database. The notifications of the gas leakage are sent as an alert in the user's twitter handle.

Literature survey:

1. Arduino Based LPG gas Monitoring & Automatic Cylinder booking with Alert System

R.Naresh Naik,

Siva Nagendra Reddy,

S.Nanda Kishore

There is a rapid development in technology which influencing the human life in several aspects due to rapid development in different fields but we still need to adopt that technology such that we can make human life more easier to live. In our Country it is not possible to supply LPG through Pipes to each and every home as production of LPG is too short. At present we are having an system Advance LPG cylinder booking through IVRS or online which is most difficult for the illiterate and busy schedule people to book the LPG cylinder in advance. Another Major problem LPG cylinder users facing is “They don’t know exactly the status of LPG gas completion” makes even more delay in booking the cylinder which is uncomfortable most of the times.

Advantages: low cost and sustainable.

Disadvantages: Great problems in the domestic as well as working places.

2. Air Quality Monitoring System

Boyanka Nikolova, Marin B. Marinov, Georgi Todorov Nikolov

Detection of low concentration of air pollution, like cigarette smoke, cooking fumes, etc. is possible with the combination of an air quality sensor

and data acquisition system. In present paper is presented approach for design and implementation of air quality monitoring system based on tin dioxide gas sensor, integrated temperature and humidity sensors, portable modular data acquisition system and graphical programming language. An interpolation method reducing influence from temperature and humidity is suggested.

Advantages: The proposed system has multi-layer protection for gas leakage in the kitchen.

Disadvantages:

Implementation is very difficult, bcz the valve was directly fixed to gas cylinder.

3. LPG Gas Monitoring System

ARUN RAJ, ATHIRA VISWANATHAN.

The design of LPG leakage monitoring system is proposed for home safety. The accidents due to the explosion of LPG is increasing now days which became a threaten to human life. In this system, the gas sensor detects the leakage of the LPG and alert the owner about the leak by sending SMS to his personal mobile and activate the alarm. In additional to this, the system continuously monitors the level of the LPG in the cylinder using load sensor and if the level is below the threshold limit the system inform them by SMS and also by the LCD display. So that the user have an idea about the max time the LPG lasts. An automatically booking of the cylinder using a GSM module is also used in this proposed system. The device ensures safety and prevents suffocation and explosion due to gas leakage.

Advantages: to ensure the safety of people and surroundings

Disadvantages:

It is some what difficult to do in thing speak.

4. Smart Lpg Gas Monitoring And Automatic Cylinder Reservation System

M.Anitha, A.Reethika Shree, R.Manjula devi

Next main issue facing users of LPG cylinder is "They are unaware of the status of completing LPG gas" which makes booking the cylinder much more time-consuming, which is mostly inconvenient. For this reason, most of the analphabets are unable to complete the booking, and usually , these phone line telephones are moreover not reachable because of the overcrowded calls or else telephones that are not functioning because of about procedural problems. The advises a program that is automate full LPG cylinder reservation technique without hominoid interference. The device endlessly monitors the heaviness of the cylinder and will automatically send notification to the approved LPG Manager once it exceeds the minimum threshold so that they perhaps distribute the LPG cylinder in step. In conjunction through the Automatic Cylinder Reservation, we have developed a safety measures for the user in which it continually tracks LPG gas leak [1i]and warns consumer about it

Advantages: It is very useful for industries bcz we can control the valve from anywhere,

Disadvantages: If any gas detected there is no exhaust fan.

5. IoT Based Real Time Industrial Hazardous Gas Leakage Detection and Alerting System

Parijat Kundu¹, Vaibhav Mishra², Pradeep Kumar G M

Gas leaks can be hazardous to all sentient beings and the environment. Due to the lack of real time warning systems, people staying nearby the industries can lose their lives. In this proposed system the detector(s) senses the presence of hazardous gases where they should not be present and informs the nearby people using mobile app.

Advantages: we can easily detect any sort of hazardous gas leakage around the industry and alert them in real time using a dedicated mobile application

Disadvantages:

we can easily detect any sort of hazardous gas leakage around the industry and alert them in real time using a dedicated mobile application

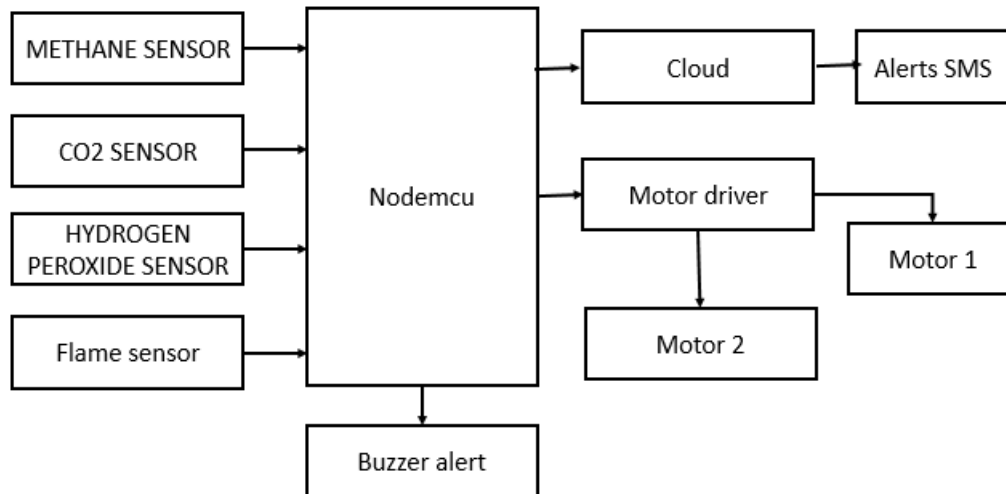
Existing system:

- The existing system is done by using the smoke sensor to detect the gas if it detect the gas means it automatically give message through the GSM module to the authenticated person
- In gas sensor it only detect the one poison gas and in normal gas pipe is release the co2 , methane , hydrogen peroxide to identify these gas is not possible
- And then main disadvantage is to place each and every node of pipe
- it is not possible for long distance

Proposed system:

- The given proposed system is done by using Nodemcu microcontroller with three sensor like CO2 gas sensor, LPG gas sensor, hydrogen peroxide sensor and flame sensor.
- The CO2, LPG gas, and hydrogen gas sensor is used to identify the CO2, LPG gas, hydrogen peroxide gas from the industry.
- If any gas is detected, it automatically send the alert message to the authorised persons and automatically turn on ventilation fan.
- Here any fire can identified by the flame sensor, it will shrinks water on the flame. And continuously update the sensor value to the cloud it is highly useful for future reference.

Block diagram:



HARDWARE USED:

Nodemcu

METHANE SENSOR

CO2 SENSOR

HYDROGEN PEROXIDE SENSOR

MOTOR DRIVER

DC MOTOR

1. EMBEDDED SYSTEMS

1.1 Overview of embedded systems

An embedded system is a special-purpose computer system designed to perform one or a few dedicated functions, often with real-time computing constraints. It is usually embedded as part of a complete device including hardware and mechanical parts. In contrast, a general-purpose computer, such as a personal computer, can do many different tasks depending on programming. Embedded systems have become very important today as they control many of the common devices we use.

Since the embedded system is dedicated to specific tasks, design engineers can optimize it, reducing the size and cost of the product, or increasing the reliability and performance. Some embedded systems are mass-produced, benefiting from economies of scale.

Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, or the systems controlling nuclear power plants. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

In general, "embedded system" is not an exactly defined term, as many systems have some element of programmability. For example, Handheld computers share some elements with embedded systems — such as the operating systems and microprocessors which power them — but are not truly embedded systems, because they allow different applications to be loaded and peripherals to be connected.

Embedded systems provide several functions

- Monitor the environment; embedded systems read data from input sensors. This data is then processed and the results displayed in some format to a user or users
- Control the environment; embedded systems generate and transmit commands for actuators.
- Transform the information; embedded systems transform the data collected in some meaningful way, such as data compression/decompression

Although interaction with the external world via sensors and actuators is an important aspect of embedded systems, these systems also provide functionality specific to their applications. Embedded systems typically execute applications such as control laws, finite state machines, and signal processing algorithms. These systems must also detect and react to faults in both the internal computing environment as well as the surrounding electromechanical systems.

There are many categories of embedded systems, from communication devices to home appliances to control systems. Examples include;

- Communication devices
 - e. g.: modems, cellular phones
- Home Appliances
 - e. g.: CD player, VCR, microwave oven
- Control Systems
 - e. g.: Automobile anti-lock braking systems, robotics, satellite control

1.2 Block diagram of an embedded system:

An embedded system usually contains an embedded processor. Many appliances that have a digital interface -- microwaves, VCRs, cars -- utilize embedded systems. Some embedded systems include an operating system. Others are very specialized resulting in the entire logic being implemented as a single program. These systems are embedded into some device for some

specific purpose other than to provide general purpose computing . A typical embedded system is shown in Fig 1.1

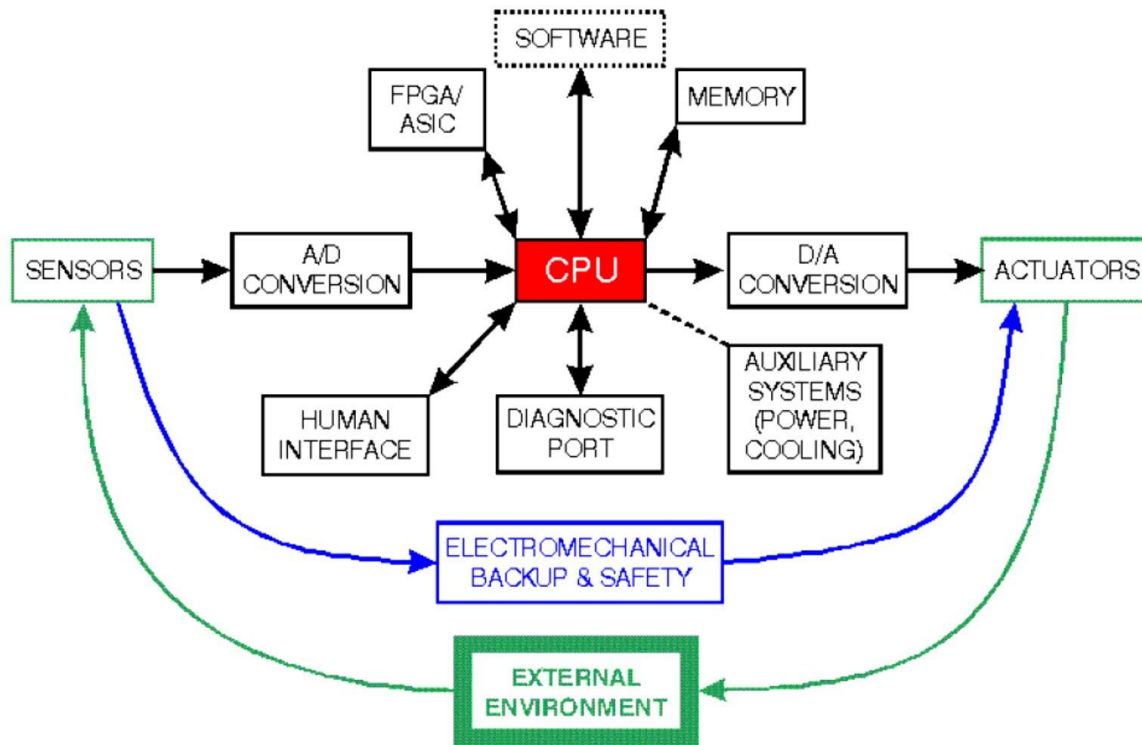


Fig 1.1 Block diagram of a typical embedded system

1.2.1 Characteristics of embedded systems

Embedded systems are characterized by a unique set of characteristics. Each of these characteristics imposed a specific set of design constraints on embedded systems designers. The challenge to designing embedded systems is to conform to the specific set of constraints for the application.

Application Specific Systems:

Embedded systems are not general-purpose computers. Embedded system designs are optimized for a specific application. Many of the job characteristics are known before the hardware is designed. This allows the designer to focus on the specific design constraints of a well-defined application. As such, there is limited user reprogram ability. Some embedded systems, however, require the flexibility of reprogram ability. Programmable DSPs are common for such applications.

Reactive Systems

As mentioned earlier, a typical embedded systems model responds to the environment via sensors and control the environment using actuators. This requires embedded systems to run at the speed of the environment. This characteristic of embedded system is called “reactive”. Reactive computation means that the system (primarily the software component) executes in response to external events. External events can be either periodic or aperiodic. Periodic events make it easier to schedule processing to guarantee performance. Aperiodic events are harder to schedule. The maximum event arrival rate must be estimated in order to accommodate worst case situations. Most embedded systems have a significant reactive component. One of the biggest challenges for embedded system designers is performing an accurate worst case design analysis on systems with statistical performance characteristics (e.g., cache memory on a DSP or other embedded processor). Real time system operation means that the correctness of a computation depends, in part, on the time at which it is delivered. Systems with this requirement must often design to worst case performance. But accurately predicting the worst case may be difficult on complicated architectures. This often leads to overly pessimistic estimates erring on the side of caution. Many embedded systems have a significant requirement for real time operation in order to meet external I/O and control stability requirements. Many real-time systems are also reactive systems.

Distributed Systems

A common characteristic of an embedded system is one that consists of communicating processes executing on several CPUs or ASICs which are connected by communication links. The reason for this is economy. Economical 4 8-bit microcontrollers may be cheaper than a 32-bit processors. Even after adding the cost of the communication links, this approach may be preferable. In this approach, multiple processors are usually required to handle multiple time-critical tasks. Devices under control of embedded systems may also be physically distributed.

Heterogeneous Architectures

Embedded systems often are composed of heterogeneous architectures (Fig 1.2). They may contain different processors in the same system solution. They

may also be mixed signal systems. The combination of I/O interfaces, local and remote memories, and sensors and actuators makes embedded system design truly unique. Embedded systems also have tight design constraints, and heterogeneity provides better design flexibility.

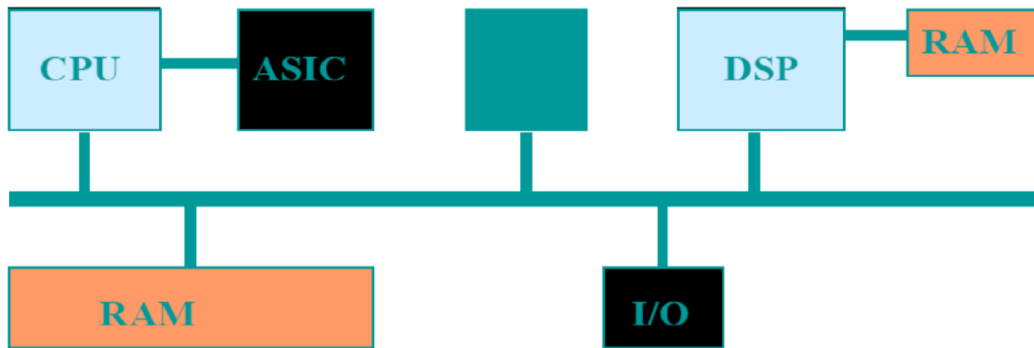


Fig 1.2 Embedded Systems having Heterogeneous Architectures

Harsh environment

Many embedded systems do not operate in a controlled environment. Excessive heat is often a problem, especially in applications involving combustion (e.g., many transportation applications). Additional problems can be caused for embedded computing by a need for protection from vibration, shock, lightning, power supply fluctuations, water, corrosion, fire, and general physical abuse.

System safety and reliability

As embedded system complexity and computing power continue to grow, they are starting to control more and more of the safety aspects of the overall system. These safety measures may be in the form of software as well as hardware control. Mechanical safety backups are normally activated when the computer system loses control in order to safely shut down system operation. Software safety and reliability is a bigger issue. Software doesn't normally "break" in the sense of hardware. However software may be so complex that a set of unexpected circumstances can cause software failures leading to unsafe situations. Discussion of this topic is outside the scope of this book, but the challenges for embedded designers include designing reliable software and building cheap, available systems using unreliable components. The main

challenge for embedded system designers is to obtain low-cost reliability with minimal redundancy.

Control of physical systems

One of the main reasons for embedding a computer is to interact with the environment. This is often done by monitoring and controlling external machinery. Embedded computers transform the analog signals from sensors into digital form for processing. Outputs must be transformed back to analog signal levels. When controlling physical equipment, large current loads may need to be switched in order to operate motors and other actuators. To meet these needs, embedded systems may need large computer circuit boards with many non-digital components. Embedded system designers must carefully balance system tradeoffs among analog components, power, mechanical, network, and digital hardware with corresponding software.

Small and low weight

Many embedded computers are physically located within some larger system. The form factor for the embedded system may be dictated by aesthetics. For example, the form factor for a missile may have to fit inside the nose of the missile. One of the challenges for embedded systems designers is to develop non-rectangular geometries for certain solutions. Weight can also be a critical constraint. Embedded automobile control systems, for example, must be light weight for fuel economy. Portable CD players must be light weight for portability purposes.

Cost sensitivity

Cost is an issue in most systems, but the sensitivity to cost changes can vary dramatically in embedded systems. This is mainly due to the effect of computer costs have on profitability and is more a function of the proportion of cost changes compared to the total system cost.

Power management

Embedded systems have strict constraints on power. Given the portability requirements of many embedded systems, the need to conserve power is

important to maintain battery life as long as possible. Minimization of heat production is another obvious concern for embedded systems.

1.3 POWER SUPPLY

All electronic circuits work only in low DC voltage, so we need a power supply unit to provide the appropriate voltage supply for their proper functioning. This unit consists of transformer, rectifier, filter & regulator. AC voltage of typically 230volts rms is connected to a transformer voltage down to the level to the desired ac voltage. A diode rectifier that provides the full wave rectified voltage that is initially filtered by a simple capacitor filter to produce a dc voltage. This resulting dc voltage usually has some ripple or ac voltage variation. A regulator circuit can use this dc input to provide dc voltage that not only has much less ripple voltage but also remains the same dc value even the dc voltage varies somewhat, or the load connected to the output dc voltages changes.

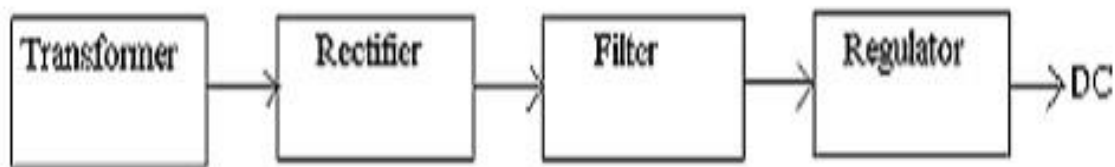


Fig 1.General Block Of Power Supply Unit

1.4 TRANSFORMER:

A transformer is a static piece of which electric power in one circuit is transformed into electric power of same frequency in another circuit. It can raise or lower the voltage in the circuit, but with a corresponding decrease or

increase in current. It works with the principle of mutual induction. In our project we are using a step down transformer to providing a necessary supply for the electronic circuits. Here we step down a 230volts ac into 12volts ac.

1.5 RECTIFIER:

A dc level obtained from a sinusoidal input can be improved 100% using a process called full wave rectification. Here in our project for full wave rectification we use bridge rectifier. From the basic bridge configuration we see that two diodes(say D2 & D3) are conducting while the other two diodes (D1 & D4) are in off state during the period $t = 0$ to $T/2$. Accordingly for the negative cycle of the input the conducting diodes are D1 & D4 .Thus the polarity across the load is the same.

In the bridge rectifier the diodes may be of variable types like 1N4001, 1N4003, 1N4004, 1N4005, 1N4007 etc.... can be used. But here we use 1N4007, because it can withstand up to 1000v.

1.6 FILTERS:

In order to obtain a dc voltage of 0 Hz, we have to use a low pass filter. So that a capacitive filter circuit is used where a capacitor is connected at the rectifier output & a dc is obtained across it. The filtered waveform is essentially a dc voltage with negligible ripples & it is ultimately fed to the load.

1.7 REGISTERS:

The controller IC has two 8 bit registers, an instruction register (IR) and a data register (DR). The IR stores the instruction codes and address information for display data RAM (DD RAM) and character generator RAM (CG RAM). The IR can be written, but not read by the MPU.

The DR temporally stores data to be written to /read from the DD RAM or CG RAM. The data written to DR by the MPU, is automatically written to the DD RAM or CG RAM as an internal operation.

When an address code is written to IR, the data is automatically transferred from the DD RAM or CG RAM to the DR. data transfer between the MPU is then completed when the MPU reads the DR. likewise, for the next MPU read of the DR, data in DD RAM or CG RAM at the address is sent to the DR automatically. Similarly, for the MPU write of the DR, the next DD RAM or CG RAM address is selected for the write operation.

The dot-matrix liquid crystal display controller and driver LSI displays alphanumeric, Japanese kana characters, and symbols. It can be configured to drive a dot-matrix liquid crystal display under the control of a 4- or 8-bit microprocessor. Since all the functions such as display RAM, character generator, and liquid crystal driver, required for driving a dot-matrix liquid crystal display are internally provided on one chip, a minimal system can be interfaced with this controller/driver

1.8 REGULATORS:

The output voltage from the capacitor is more filtered & finally regulated. The voltage regulator is a device, which maintains the output voltage constant irrespective of the change in supply variations, load variations & temperature changes. Here we use fixed voltage regulator namely LM7805. The IC LM7805 is a +5v regulator which is used for microcontroller.

1.9 FEATURES AND DESCRIPTION OF REGULATORS

- Output Current up to 1A
- Output Voltages of 5, 6, 8, 9, 10, 12, 15, 18, 24V
- Thermal Overload Protection

- Short Circuit Protection
- Output Transistor Safe Operating Area Protection

NODE MCU

NodeMCU is an open source [IoT](#) platform.^{[4][5]} It includes [firmware](#) which runs on the [ESP8266 Wi-Fi SoC](#) from [Espressif Systems](#), and hardware which is based on the ESP-12 module.^{[6][7]} The term "NodeMCU" by default refers to the firmware rather than the development kits. The firmware uses the [Lua](#) scripting language. It is based on the eLua project, and built on the Espressif Non-OS SDK for ESP8266. It uses many open source projects, such as lua-cjson^[8] and [SPIFFS](#)

NodeMCU was created shortly after the [ESP8266](#) came out. On December 30, 2013, [Espressif Systems](#)^[6] began production of the ESP8266.^[10] The ESP8266 is a Wi-Fi SoC integrated with a [Tensilica](#) Xtensa LX106 core,^[citation needed] widely used in IoT applications (see [related projects](#)). NodeMCU started on 13 Oct 2014, when Hong committed the first file of nodemcu-firmware to GitHub.^[11] Two months later, the project expanded to include an open-hardware platform when developer Huang R committed the [gerber](#) file of an ESP8266 board, named devkit v0.9.^[12] Later that month, Tuan PM ported [MQTT](#) client library from [Contiki](#) to the ESP8266 SoC platform,^[13] and committed to NodeMCU project, then NodeMCU was able to support the MQTT IoT protocol, using Lua to access the MQTT broker. Another important update was made on 30 Jan 2015, when Devsaurus ported the u8glib^[14] to NodeMCU project,^[15] enabling NodeMCU to easily drive LCD, Screen, OLED, even VGA displays.

In summer 2015 the creators abandoned the firmware project and a group of independent contributors took over. By summer 2016 the NodeMCU included more than 40 different modules. Due to resource constraints users

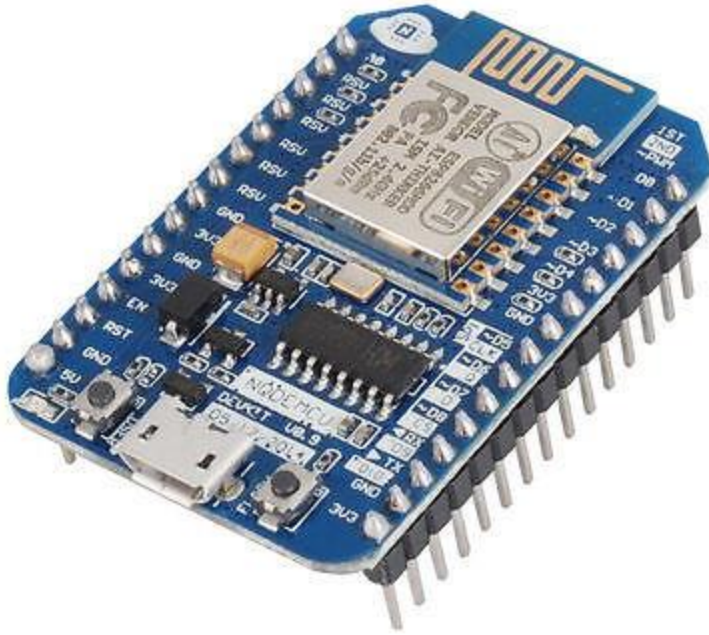
need to select the modules relevant for their project and build a firmware tailored to their needs.

ESP8266 Arduino Core[\[edit\]](#)

As Arduino.cc began developing new MCU boards based on non-AVR processors like the ARM/SAM MCU and used in the Arduino Due, they needed to modify the Arduino IDE so that it would be relatively easy to change the IDE to support alternate toolchains to allow Arduino C/C++ to be compiled for these new processors. They did this with the introduction of the Board Manager and the SAM Core. A "core" is the collection of software components required by the Board Manager and the Arduino IDE to compile an Arduino C/C++ source file for the target MCU's machine language. Some ESP8266 enthusiasts developed an Arduino core for the ESP8266 WiFi SoC, popularly called the "ESP8266 Core for the Arduino IDE".^[16] This has become a leading software development platform for the various ESP8266-based modules and development boards, including NodeMCUs.

introduction

NodeMCU is an open source [LUA](#) based firmware developed for ESP8266 wifi chip. By exploring functionality with ESP8266 chip, NodeMCU firmware comes with ESP8266 Development board/kit i.e. NodeMCU Development board.

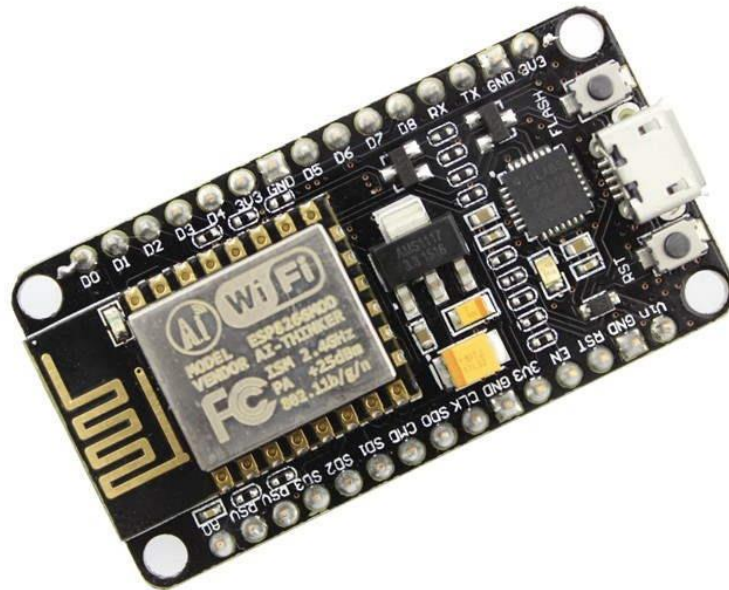


NodeMCU Development Board/kit v0.9 (Version1)

Since NodeMCU is open source platform, their hardware design is open for edit/modify/build.

NodeMCU Dev Kit/board consist of ESP8266 wifi enabled chip. The **ESP8266** is a low-cost [Wi-Fi](#) chip developed by Espressif Systems with TCP/IP protocol. For more information about ESP8266, you can refer [ESP8266 WiFi Module](#).

There is Version2 (V2) available for NodeMCU Dev Kit i.e. **NodeMCU Development Board v1.0 (Version2)**, which usually comes in black colored PCB.



NodeMCU Development Board/kit v1.0 (Version2)

For more information about NodeMCU Boards available in market refer [NodeMCU Development Boards](#)

NodeMCU Dev Kit has **Arduino like** Analog (i.e. A0) and Digital (D0-D8) pins on its board.

It supports serial communication protocols i.e. UART, SPI, I2C etc.

Using such serial protocols we can connect it with serial devices like I2C enabled LCD display, Magnetometer HMC5883, MPU-6050 Gyro meter + Accelerometer, RTC chips, GPS modules, touch screen displays, SD cards etc.

How to start with NodeMCU?

NodeMCU Development board is featured with wifi capability, analog pin, digital pins and serial communication protocols.

To get start with using NodeMCU for IoT applications first we need to know about how to write/download NodeMCU firmware in NodeMCU

Development Boards. And before that where this NodeMCU firmware will get as per our requirement.

There is online NodeMCU custom builds available using which we can easily get our custom NodeMCU firmware as per our requirement.

To know more about how to build custom NodeMCU firmware online and download it refer [Getting started with NodeMCU](#)

How to write codes for NodeMCU?

After setting up ESP8266 with Node-MCU firmware, let's see the IDE (Integrated Development Environment) required for development of NodeMCU.

NodeMCU with ESPlorer IDE

[Lua](#) scripts are generally used to code the NodeMCU. Lua is an open source, lightweight, embeddable scripting language built on top of C programming language.

For more information about how to write Lua script for NodeMCU refer [Getting started with NodeMCU using ESPlorerIDE](#)

NodeMCU with Arduino IDE

Here is another way of developing NodeMCU with a well-known IDE i.e. Arduino IDE. We can also develop applications on NodeMCU using Arduino development environment. This makes easy for Arduino developers than learning new language and IDE for NodeMCU.

For more information about how to write Arduino sketch for NodeMCU refer [Getting started with NodeMCU using ArduinoIDE](#)

Difference in using ESPlorer and Arduino IDE

Well, there is a programming language difference we can say while developing application for NodeMCU using ESPlorer IDE and Arduino IDE.

We need to code in C\C++ programming language if we are using Arduino IDE for developing NodeMCU applications and Lua language if we are using ESPlorer IDE.

Basically, NodeMCU is Lua Interpreter, so it can understand Lua script easily. When we write Lua scripts for NodeMCU and send/upload it to NodeMCU, then they will get executes sequentially. It will not build binary firmware file of code for NodeMCU to write. It will send Lua script as it is to NodeMCU to get execute.

In Arduino IDE when we write and compile code, ESP8266 toolchain in background creates binary firmware file of code we wrote. And when we upload it to NodeMCU then it will flash all NodeMCU firmware with newly generated binary firmware code. In fact, it writes the complete firmware.

That's the reason why NodeMCU not accept further Lua scripts/code after it is getting flashed by Arduino IDE. After getting flashed by Arduino sketch/code it will be no more Lua interpreter and we got error if we try to upload Lua scripts. To again start with Lua script, we need to flash it with NodeMCU firmware.

Since Arduino IDE compile and upload/writes complete firmware, it takes more time than ESPlorer IDE.

CURRENT TRANSFORMER

Current transformers are used in electric metering for large load situations to reduce the current level presented to the metering circuit in order to make it more manageable and safe. This paper presents the basics of operation of current transformers and discusses the in-service testing of current transformers to assure accurate performance.

BASICS OF OPERATION

Ratio:

A current transformer (henceforth called a CT) is designed to produce a secondary current, I_s , that is directly proportional to its primary current, I_p , which is also called the load current. In metering this proportionality is called the ratio of the CT and is represented by:

$$\text{Ratio} = \frac{I_p}{I_s} \times 5$$

Rated Ratio:

The ratio of a CT is referenced to a 5 amp secondary output for nominal full value input current such that a CT ratio of 200 to 5 means that the CT would produce a secondary current of 5 amps when exposed to a primary current of 200 amps. The CT manufacturer classifies its CT's by its ratio value called the rated ratio which is stamped on the nameplate of the CT and that is the certified operating value of the CT.

Accuracy Class:

The CT accuracy is determined by its certified accuracy class which is also stamped on its nameplate. For example, a CT accuracy class of 0.3 means that the CT is certified by the manufacturer to be accurate to within 0.3 percent of its rated ratio value for a primary current of 100 percent of rated ratio. For primary currents of 10 percent of rated ratio the specified accuracy for a given classification is double that of the 100 percent value or 0.6 percent for the above-stated case. The accuracy is relatively linear between these two points. Hence a CT with a rated ratio of 200 to 5 with accuracy class of 0.3 would operate within 0.45 percent of its rated ratio value for a primary current of 100 amps. To be more explicit, for a primary current of 100.00 amps it is certified to produce a secondary current between 2.489 amps and 2.511 amps.

Rating Factor:

Some CT's are certified to be accurate for primary current values greater than 100 percent of the rated ratio, called over-range. This over-range is designated by a rating factor representing the factor by which the upper limit of the primary current can be increased and still remain within certified accuracy. Hence a 200 to 5 rated ratio CT with a rating factor of 2 will remain within its accuracy certification up to 400 amps (200 amps times

2). The rating factor of the CT is stamped on the nameplate of the CT. Typical rating factors are 1, 1.5, 2, 3, and 4.

Burden:

When a CT is used in a metering circuit it will have a secondary current loop connecting its output through a test block to a meter . This current loop will consist of a secondary wire from its X1 terminal to a test block and go through a shorting switch. From the shorting switch the current loop will have a wire connection over to the meter, go through the meter, and have a wire connection back to the test block. From there the current loop will go through the secondary current test switch and have a wire connection return to the X2 terminal of the CT with a connection to neutral either at the output of the current test switch or at the X2 terminal of the CT . This secondary current loop has an electrical impedance called its burden. This burden will have a resistance component and may have an inductance component. Usually the inductance component is small and is treated as negligible. The burden of the secondary circuit, as installed, is designated the natural burden of the circuit. If installed properly the natural burden should be less than a few milliohms. The CT is designed to operate accurately under certain burden conditions as specified. The accuracy class certification of a CT is presented for a stated upper value of burden, called the CT's rated burden. For example it may have an accuracy class of 0.3 for burden of 1, meaning that the CT's rated burden is 1 ohm and the CT will maintain accurate rated ratio performance within 0.3 percent for secondary circuit burdens up to 1 ohm. This rated burden classification is also stamped on the CT nameplate. If the natural burden of the secondary circuit is larger than the rated burden, then the CT is not certified to have accurate performance. For this condition, called over-burdened, the secondary current will not be accurate tending to be somewhat less to significantly less than the accurate value causing the metering to be in error in favor of the customer. Furthermore, secondary circuits may have burden build up as they age due to loose connections, corrosion, broken stranded wires, etc. CT's, as installed, need to have enough excess burden capacity over the

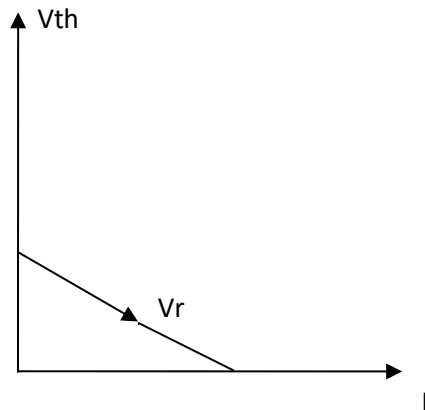
natural burden to allow accurate performance over time even if there is additional burden build up in the secondary circuit.

3.1.4 POTENTIAL TRANSFORMER

The principle of voltage transformer is identical to the conventional transformer. Hence, its equivalent circuit can be represented as follows; Typically, the secondary voltage of the VT is standardized to 110V (ac). Hence, as the primary voltage increases, the turns ratio $N_1:N_2$ increases and transformer becomes bulky. To cut down the VT size and cost, a capacitance potential divider is used (fig 2). This leads to development of coupling capacitor voltage transformers (CCVT). Thus, a reduced voltage is fed to primary of the transformer. This reduces the size of VT.

Assuming, the X_{eq} to be ideal, the Thevenin's equivalent circuit of CCVT. It is now obvious that Z_{th} due to the capacitance divider, affects the voltage received by the relay. To achieve high level of accuracy, it is therefore necessary to compensate for this voltage drop by connecting a tuning inductor. The tuning inductor's value is so chosen that it compensates for the 'net C' at power frequency (50Hz in India). From the corresponding equivalent circuit, it is apparent that, if $\omega L = 1/\omega(C_1+C_2)$, then voltage drop across C is neutralized and the relay burden sees the actual voltage to be measured.

The phasor diagram across resistive load,

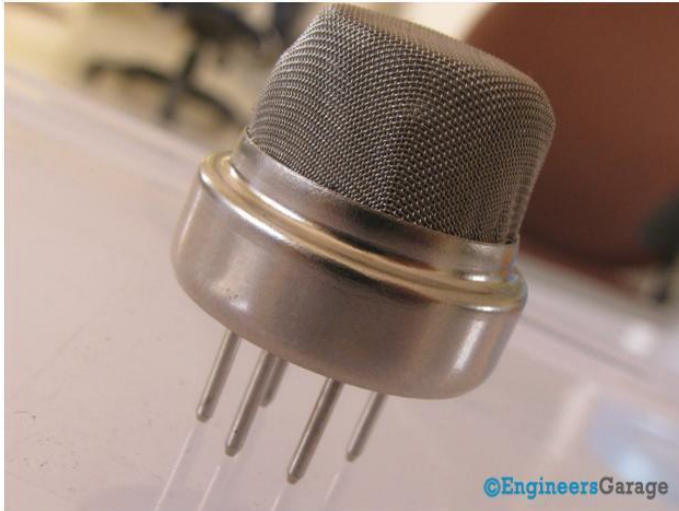


SENSORS:

GAS SENSORS

In current technology scenario, monitoring of gases produced is very important. From home appliances such as air conditioners to electric chimneys and safety systems at industries monitoring of gases is very crucial. **Gas sensors** are very important part of such systems. Small like a nose, gas sensors spontaneously react to the gas present, thus keeping the system updated about any alterations that occur in the concentration of molecules at gaseous state.

Gas sensors are available in wide specifications depending on the sensitivity levels, type of gas to be sensed, physical dimensions and numerous other factors. This Insight covers a **methane gas sensor** that can sense gases such as ammonia which might get produced from methane. When a gas interacts with this sensor, it is first ionized into its constituents and is then adsorbed by the sensing element. This adsorption creates a potential difference on the element which is conveyed to the processor unit through output pins in form of current. What is this sensing element? Is it kept in some chamber or is kept exposed? How does it get current and how it is taken out? Let's find out in this Insight!!!



The **gas sensor module** consists of a steel exoskeleton under which a sensing element is housed. This sensing element is subjected to current through connecting leads. This current is known as heating current through it, the gases coming close to the sensing element get ionized and are absorbed by the sensing element. This changes the resistance of the sensing element which alters the value of the current going out of it.



Image 01 shows externals of a standard gas sensor module: a steel mesh, copper clamping ring and connecting leads. The top part is a stainless steel mesh which takes care of the following:

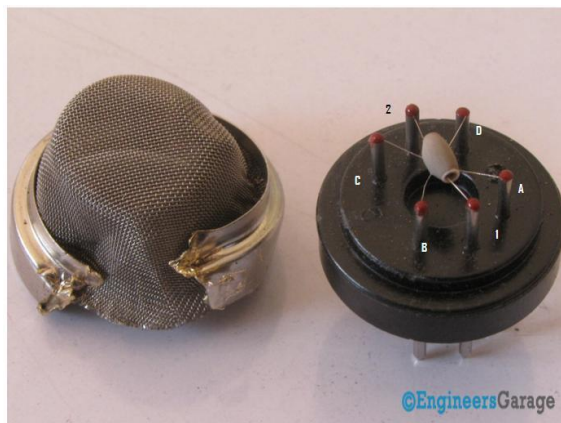
1. Filtering out the suspended particles so that only gaseous elements are able to pass to insides of the sensor.

2. Protecting the insides of the sensor.
3. Exhibits an anti explosion network that keeps the sensor module intact at high temperatures and gas pressures.
4. In order to manage above listed functions efficiently, the steel mesh is made into two layers. The mesh is bound to rest of the body via a copper plated clamping ring.



The connecting leads of the sensor are thick so that sensor can be connected firmly to the circuit and sufficient amount of heat gets conducted to the inside part. They are casted from copper and have tin plating over them. Four of the six leads (A, B, C, D) are for signal fetching while two (1,2) are used to provide sufficient heat to the sensing element.

The pins are placed on a Bakelite base which is a good insulator and provides firm gripping to the connecting leads of the sensor.

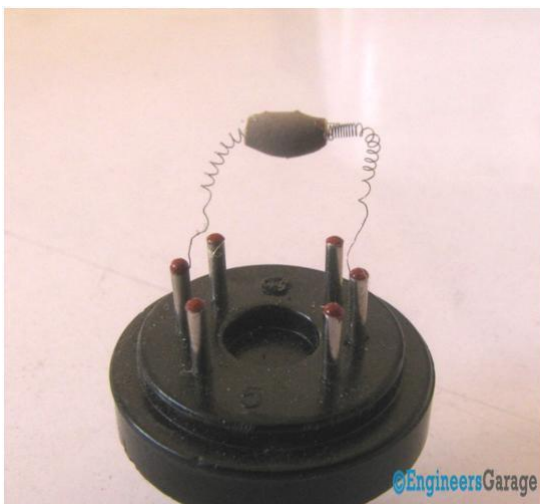


The top of the gas sensor is removed off to see the internal parts of the sensor: sensing element and connection wiring. The hexapod structure is constituted by the sensing element and six connecting legs that extend beyond the Bakelite base.



Image4 shows the hollow sensing element which is made up from Aluminum Oxide based ceramic and has a coating of tin oxide. Using a ceramic substrate increases the heating efficiency and tin oxide, being sensitive towards adsorbing desired gas' components (in this case methane and its products) suffices as sensing coating.

The leads responsible for heating the sensing element are connected through Nickel-Chromium, well known conductive alloy. Leads responsible for output signals are connected using platinum wires which convey small changes in the current that passes through the sensing element. The platinum wires are connected to the body of the sensing element while Nickel-Chromium wires pass through its hollow structure.



While other wires are attached to the outer body of the element, Nickel-Chromium wires are placed inside the element in a spring shaped. Image 5

shows coiled part of the wire which is placed on the inside of the hollow ceramic.

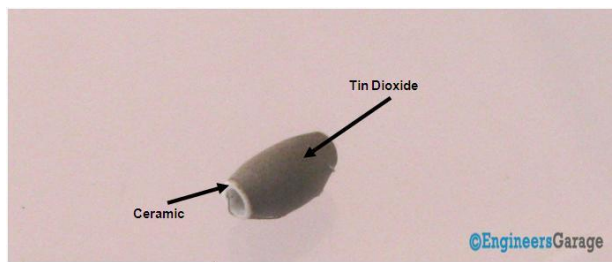


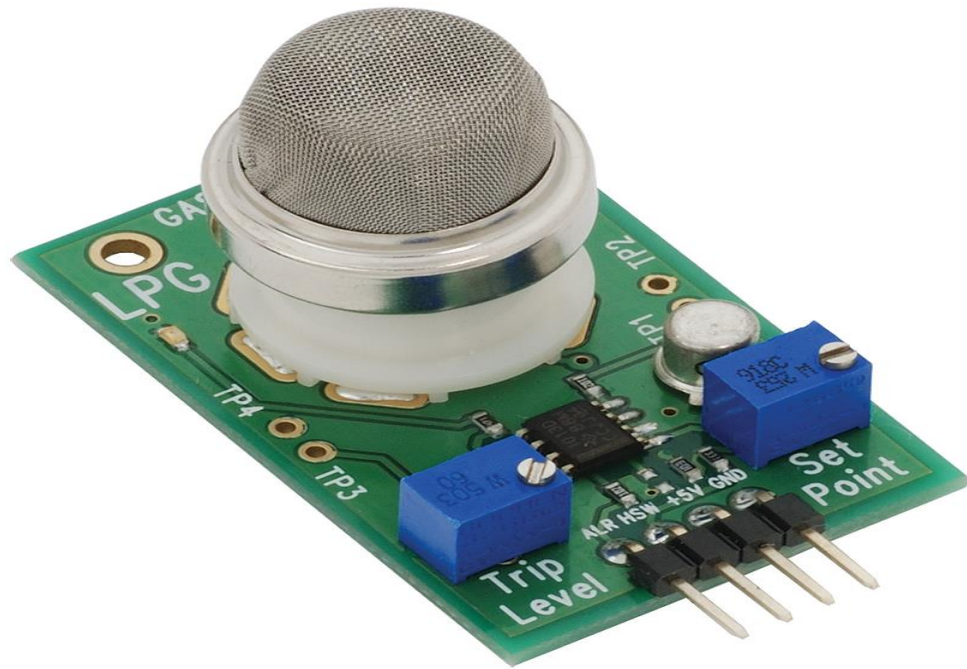
Image06 shows the ceramic with tin dioxide on the top coating that has good adsorbing property. Any gas to be monitored has specific temperature at which it ionizes. The task of the sensor is to work at the desired temperature so that gas molecules get ionized. Through Nickel-chromium wire, the ceramic region of the sensing element is subjected to heating current. The heat is radiated by the element in the nearby region where gases interact with it and get ionized. Once, ionized, they are absorbed by the tin dioxide. Adsorbed molecules change the resistance of the tin dioxide layer. This changes the current flowing through the sensing element and is conveyed through the output leads to the unit that controls the **working of the gas sensor**.

Gas sensor module

In current technology scenario, monitoring of gases produced is very important. From home appliances such as air conditioners to electric chimneys and safety systems at industries monitoring of gases is very crucial. **Gas sensors** are very important part of such systems. Small like a nose, gas sensors spontaneously react to the gas present, thus keeping the system updated about any alterations that occur in the concentration of molecules at gaseous state.

Gas sensors are available in wide specifications depending on the sensitivity levels, type of gas to be sensed, physical dimensions and numerous other factors. This Insight covers a **methane gas sensor** that can sense gases such as ammonia which might get produced from methane. When a gas interacts with this sensor, it is first ionized into its constituents and is then adsorbed by the sensing element. This adsorption creates a

potential difference on the element which is conveyed to the processor unit through output pins in form of current. What is this sensing element? Is it kept in some chamber or is kept exposed? How does it get current and how it is taken out? Let's find out in this Insight!!!



The **gas sensor module** consists of a steel exoskeleton under which a sensing element is housed. This sensing element is subjected to current through connecting leads. This current is known as heating current through it, the gases coming close to the sensing element get ionized and are absorbed by the sensing element. This changes the resistance of the sensing element which alters the value of the current going out of it.

Image shows externals of a standard gas sensor module: a steel mesh, copper clamping ring and connecting leads. The top part is a stainless steel mesh which takes care of the following:

Filtering out the suspended particles so that only gaseous elements are able to pass to insides of the sensor.

- ✓ Protecting the insides of the sensor.
- ✓ Exhibits an anti-explosion network that keeps the sensor module intact at high temperatures and gas pressures.

In order to manage above listed functions efficiently, the steel mesh is made into two layers. The mesh is bound to rest of the body via a copper plated clamping ring.

The connecting leads of the sensor are thick so that sensor can be connected firmly to the circuit and sufficient amount of heat gets conducted to the inside part. They are casted from copper and have tin plating over them. Four of the six leads (A, B, C, D) are for signal fetching while two (1, 2) are used to provide sufficient heat to the sensing element.

The pins are placed on a Bakelite base which is a good insulator and provides firm gripping to the connecting leads of the sensor.

The top of the gas sensor is removed off to see the internal parts of the sensor: sensing element and connection wiring. The hexapod structure is constituted by the sensing element and six connecting legs that extend beyond the Bakelite base.

Using a ceramic substrate increases the heating efficiency and tin oxide, being sensitive towards adsorbing desired gas' components (in this case methane and its products) suffices as sensing coating.

The leads responsible for heating the sensing element are connected through Nickel-Chromium, well known conductive alloy. Leads responsible for output signals are connected using platinum wires which convey small changes in the current that passes through the sensing element. The platinum wires are connected to the body of the sensing element while Nickel-Chromium wires pass through its hollow structure.

While other wires are attached to the outer body of the element, Nickel-Chromium wires are placed inside the element in a spring shaped. Image

shows coiled part of the wire which is placed on the inside of the hollow ceramic

Image shows the ceramic with tin dioxide on the top coating that has good adsorbing property. Any gas to be monitored has specific temperature at which it ionizes. The task of the sensor is to work at the desired temperature so that gas molecules get ionized. Through Nickel-chromium wire, the ceramic region of the sensing element is subjected to heating current. The heat is radiated by the element in the nearby region where gases interact with it and get ionized. Once, ionized, they are absorbed by the tin dioxide. Adsorbed molecules change the resistance of the tin dioxide layer. This changes the current flowing through the sensing element and is conveyed through the output leads to the unit that controls the working of the gas sensor.

Temperature sensor LM35:

General Description:

The LM35 series are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature. The LM35 thus has an advantage over linear temperature sensors calibrated in ° Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Centigrade scaling. The LM35 does not require any external calibration or trimming to provide typical accuracies of $\pm 1/4^\circ\text{C}$ at room temperature and $\pm 3/4^\circ\text{C}$ over a full -55 to $+150^\circ\text{C}$ temperature range. Low cost is assured by trimming and calibration at the wafer level. The LM35's low output impedance, linear output, and precise inherent calibration make interfacing to readout or control circuitry especially easy. It can be used with single power supplies, or with plus and minus supplies. As it draws only $60\text{ }\mu\text{A}$ from its supply, it has very low self-heating, less than 0.1°C in still air. The LM35 is rated to operate over a -55° to $+150^\circ\text{C}$ temperature range, while the LM35C is rated for a -40° to $+110^\circ\text{C}$ range (-10° with improved accuracy). The LM35 series is available packaged in hermetic TO-46 transistor packages, while the LM35C, LM35CA,

and LM35D are also available in the plastic TO-92 transistor package. The LM35D is also available in an 8-lead surface mount small outline package and a plastic TO-220 package.

Features

- Calibrated directly in ° Celsius (Centigrade)
- Linear + 10.0 mV/°C scale factor
- 0.5°C accuracy guaranteeable (at +25°C)
- Rated for full -55° to +150°C range
- Suitable for remote applications
- Low cost due to wafer-level trimming
- Operates from 4 to 30 volts
- Less than 60 µA current drain
- Low self-heating, 0.08°C in still air
- Nonlinearity only $\pm 1/4^\circ\text{C}$ typical
- Low impedance output, 0.1 Ω for 1 mA load

Typical Applications

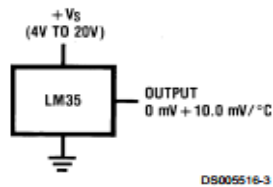
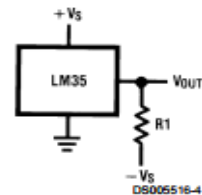


FIGURE 1. Basic Centigrade Temperature Sensor (+2°C to +150°C)

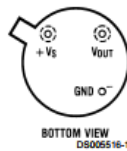


Choose $R_1 = -V_S/50 \mu\text{A}$
 $V_{OUT} = +1,500 \text{ mV at } +150^\circ\text{C}$
 $= +250 \text{ mV at } +25^\circ\text{C}$
 $= -550 \text{ mV at } -55^\circ\text{C}$

FIGURE 2. Full-Range Centigrade Temperature Sensor

Connection Diagrams

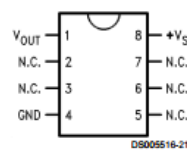
TO-46
Metal Can Package*



*Case is connected to negative pin (GND)

Order Number LM35H, LM35AH, LM35CH, LM35CAH or LM35DH
See NS Package Number H03H

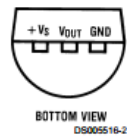
SO-8
Small Outline Molded Package



N.C. = No Connection

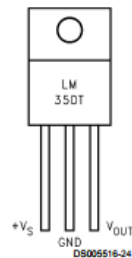
Top View
Order Number LM35DM
See NS Package Number M08A

TO-92
Plastic Package



Order Number LM35CZ, LM35CAZ or LM35DZ
See NS Package Number Z03A

TO-220
Plastic Package*



*Tab is connected to the negative pin (GND).

Note: The LM35DT pinout is different than the discontinued LM35DP.

Order Number LM35DT
See NS Package Number TA03F

- Absolute Maximum Ratings (Note 10)
- Supply Voltage $+35V$ to $-0.2V$
- Output Voltage $+6V$ to $-1.0V$
- Output Current 10 mA Storage Temp.;
- TO-46 Package, $-60^{\circ}C$ to $+180^{\circ}C$
- TO-92 Package, $-60^{\circ}C$ to $+150^{\circ}C$
- SO-8 Package, $-65^{\circ}C$ to $+150^{\circ}C$
- TO-220 Package, $-65^{\circ}C$ to $+150^{\circ}C$ Lead Temp.:
- TO-46 Package, (Soldering, 10 seconds) $300^{\circ}C$
- TO-92 and TO-220 Package, (Soldering, 10 seconds) $260^{\circ}C$
- SO Package (Note 12)
- Vapor Phase (60 seconds) $215^{\circ}C$
- Infrared (15 seconds) $220^{\circ}C$

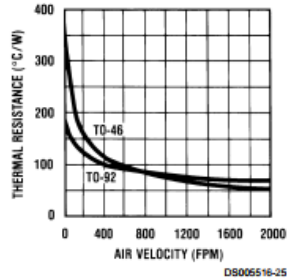
ESD Susceptibility (Note 11)

2500V Specified Operating Temperature Range: T_{MIN} to T_{MAX} (Note 2)

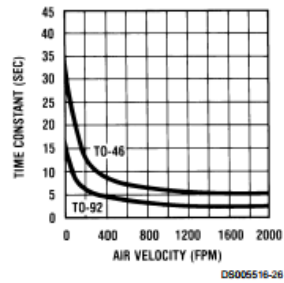
- LM35, LM35A -55°C to $+150^{\circ}\text{C}$
- LM35C, LM35CA -40°C to $+110^{\circ}\text{C}$
- LM35D 0°C to $+100^{\circ}\text{C}$

Typical Performance Characteristics

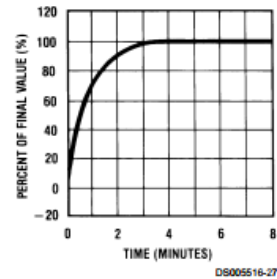
**Thermal Resistance
Junction to Air**



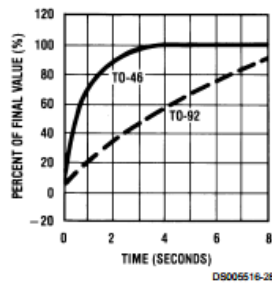
Thermal Time Constant



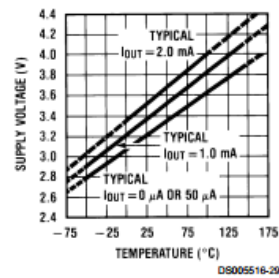
**Thermal Response
in Still Air**



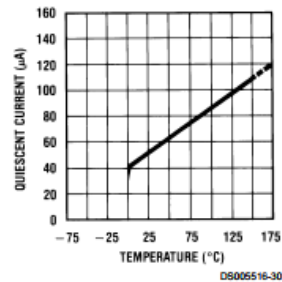
**Thermal Response in
Stirred Oil Bath**



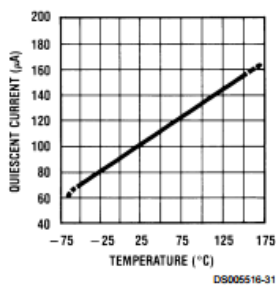
**Minimum Supply
Voltage vs. Temperature**



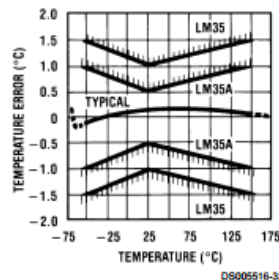
**Quiescent Current
vs. Temperature
(In Circuit of Figure 1.)**



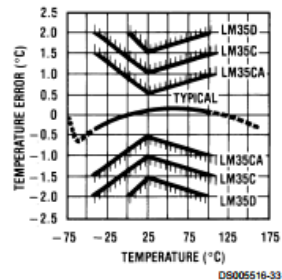
**Quiescent Current
vs. Temperature
(In Circuit of Figure 2.)**



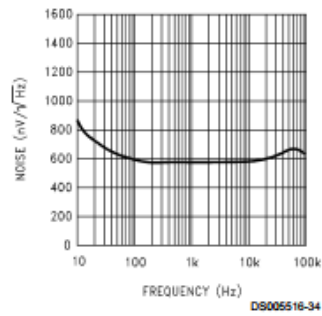
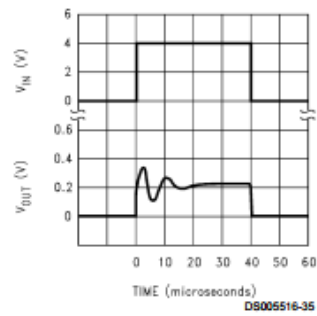
**Accuracy vs. Temperature
(Guaranteed)**



**Accuracy vs. Temperature
(Guaranteed)**



Typical Performance Characteristics

Noise Voltage**Start-Up Response**

Applications:

The LM35 can be applied easily in the same way as other integrated-circuit temperature sensors. It can be glued or cemented to a surface and its temperature will be within about 0.01°C of the surface temperature. This presumes that the ambient air temperature is almost the same as the surface temperature; if the air temperature were much higher or lower than the surface temperature, the actual temperature of the LM35 die would be at an intermediate temperature between the surface temperature and the air temperature. This is especially true for the TO-92 plastic package, where the copper leads are the principal thermal path to carry heat into the device, so its temperature might be closer to the air temperature than to the surface temperature. To minimize this problem, be sure that the wiring to the LM35, as it leaves the device, is held at the same temperature as the surface of interest. The easiest way to do this is to cover up these wires with a bead of epoxy which will insure that the leads and wires are all at the same temperature as the surface, and that the LM35 die's temperature will not be affected by the air temperature.

The TO-46 metal package can also be soldered to a metal surface or pipe without damage. Of course, in that case the V- terminal of the circuit will be grounded to that metal. Alternatively, the LM35 can be mounted inside a sealed-end metal tube, and can then be dipped into a bath or screwed into a threaded hole in a tank. As with any IC, the LM35 and accompanying wiring and circuits must be kept insulated and dry, to avoid leakage and corrosion. This is especially true if the circuit may operate at cold temperatures where condensation can occur. Printed-circuit coatings and varnishes such as

Humiseal and epoxy paints or dips are often used to insure that moisture cannot corrode the LM35 or its connections. These devices are sometimes soldered to a small light-weight heat fin, to decrease the thermal time constant and speed up the response in slowly-moving air. On the other hand, a small thermal mass may be added to the sensor, to give the steadiest reading despite small deviations in the air temperature.

Relay:

A relay is an electrically operated switch. Current flowing through the coil of the relay creates a magnetic field which attracts a lever and changes the switch contacts. The coil current can be on or off so relays have two switch positions and they are double throw (changeover) switches. Relays allow one circuit to switch a second circuit which can be completely separate from the first. For example a low voltage battery circuit can use a relay to switch a 230V AC mains circuit. There is no electrical connection inside the relay between the two circuits; the link is magnetic and mechanical.

The coil of a relay passes a relatively large current, typically 30mA for a 12V relay, but it can be as much as 100mA for relays designed to operate from lower voltages. Most ICs (chips) cannot provide this current and a transistor is usually used to amplify the small IC current to the larger value required for the relay coil. The maximum output current for the popular 555 timer IC is 200mA so these devices can supply relay coils directly without amplification.



Relays are usually SPDT or DPDT but they can have many more sets of switch contacts, for example relays with 4 sets of changeover contacts are readily available. Most relays are designed for PCB mounting but you can

solder wires directly to the pins providing you take care to avoid melting the plastic case of the relay. The animated picture shows a working relay with its coil and switch contacts. You can see a lever on the left being attracted by magnetism when the coil is switched on. This lever moves the switch contacts. There is one set of contacts (SPDT) in the foreground and another behind them, making the relay DPDT.



The relay's switch connections are usually labeled COM, NC and NO:

- **COM** = Common, always connect to this, it is the moving part of the switch.
- **NC** = Normally Closed, COM is connected to this when the relay coil is **off**.
- **NO** = Normally Open, COM is connected to this when the relay coil is **on**.

Chapter1

IOT:

The structure of this chapter is the following: Section 1.1 defines the motivation for this work. Section 1.2 shows the research objectives and questions. Section 1.3 presents the steps required to achieve our objectives. Section 1.4 shows the structure of the report.

1.1 Background

The Future Internet goal is to provide an infrastructure to have an immediate access to information about the physical world and its objects. Physical objects can be applicable to different application domains, such as e-health, warehouse management, etc. Each application domain may have different types of physical devices. Each physical device can have its own specifications, which is required to use in order to interact with it. To achieve the future Internet goal, a layered vision is required that can facilitate data access. Internet of Things (IoT) is a vision that aims to integrate the virtual world of information to the real world of devices through a layered architecture.

The term „Internet of Things“ consists of two words, namely *Internet* and *Things*. *Internet* refers to the global network infrastructure with scalable, configurable capabilities based on interoperable and standard communication protocols. *Things* are physical objects or devices, or virtual objects, devices or information, which have identities, physical attributes and virtual personalities, and use intelligent interfaces [1]. For instance, a virtual object can represent an abstract unit of sensor nodes that contains metadata to identify and discover its corresponding sensor nodes. Therefore, IoT refers to the *things* that can provide information from the physical environment through the Internet.

Middleware is as an interface between the hardware layer and the application layer, which is responsible for interacting with devices and

information management [2]. The role of a middleware is to present a unified programming model to interact with devices. A middleware is in charge of masking the heterogeneity and distribution problems that we face when interacting with devices [3].

1.2 Motivation

IoT-based system is in charge of providing knowledge from an environment to a non-expert user. IoT-based system can be used in different environments, so it needs to be able to address many heterogeneous devices. Thus, a major concern within developing an IoT-based system is how to handle the interaction with the heterogeneous devices for non-expert users. This concern can be addressed by a middleware layer between devices and non-expert users. This layer is responsible to hide the diversity of devices from the user perspective, and provides access transparency to the devices for the end users.

The idea of creating abstractions of devices has been addressed in the literature. The middleware we found in the literature can provide satisfaction by facilitating the interaction with devices, but they do not support low-level device configuration [4].

Chapter 2

2.1. IoT definition

In this section, we explain some of the IoT definitions. Also, we explain the layered architecture for IoT.

Internet of Things (IoT) has increasingly gained attention in industry to interact with different types of devices. IoT can have influence on industry and society by integrating physical devices into information networks [8]. IoT impacts can be on different perspectives, namely for private and business users. From the perspective of a private user, IoT has effect on both working and personal fields, such as smart homes and offices, e-health and assisted living. From the aspect of a business user the impacts would be in fields such as automation and industrial manufacturing, logistics, business process management, intelligent transportation of people and goods [9]

IoT integrates physical things into information networks. IoT covers the overall infrastructure, including software, hardware and services, which is used to support these information networks. The integrated physical things can exchange data about the physical properties and information that they sense in their environment. To identify devices, we can use identification technologies like for example RFID, which allow each device be uniquely identified [10].

International Telecommunication Union (ITU)¹ defines IoT as *“A global infrastructure for the Information Society, enabling advanced services by interconnecting (physical and virtual) things based on, existing and evolving, interoperable information and communication technologies”*[11]

Internet of Things-Architecture ² (IoT-A) defines it as *“The idea of a globally interconnected continuum of devices, objects and things in general emerged with the RFID technology, and this concept has considerably been extended to the current vision that envisages a plethora of heterogeneous objects*

interacting with the physical environment.”[12]

IoT has a layered architecture designed to answer the demands of various industries, enterprises and society. Fig. 2.1 shows a generic layered architecture for IoT that consist of five layers, which are discussed, in the following:[13]

❑ Edge Technology layer

This is a hardware layer that consists of embedded systems, RFID tags, sensor networks and all of the other sensors in different forms. This hardware layer can perform several functions, such as collecting information from a system or an environment, processing information and supporting communication.

❑ Access Gateway layer

This layer is concerned with data handling, and is responsible for publishing and subscribing the services that are provided by the *Things*, message routing, and hovelling the communication between platforms.

❑ Middleware layer

This layer has some critical functionalities, such as aggregating and filtering the received data from the hardware devices, performing information discovery and providing access control to the devices for applications.

Application layer

This layer is responsible for delivering various application services. These services are provided through the middleware layer to different applications and users in IoT-based systems. The application services can be used in different industries such as, logistics, retail,healthcare,etc.

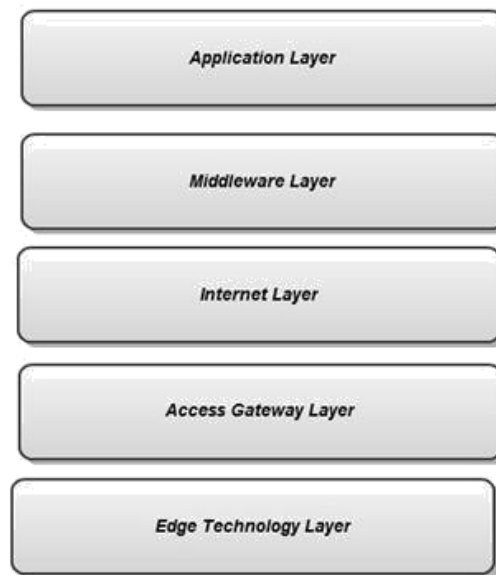


Figure 2-1 Layered architecture of the Internet of Things (IoT).

2.2 Challenges

IoT-based systems can be used for different purposes and areas, so that, we have to face different challenges. In this section, we explain some of the challenges that need to be considered in research activities [10]:

- Edge technology

At the hardware level, more research efforts are required to develop the technology of embedded devices, sensors, actuators and (passive and active) identification, since an IoT-based system must be able to gather sufficient information about the real world by employing a large variety of devices and environments. Thus, there more work is still required to connect heterogeneous devices and deploy them in IoT applications, and to provide support for new devices.

❏ Networking technologies

In IoT, *things* are connected through different kinds of networks, i.e. mobile, wired and wireless network. These networks support bi-directional communication at different levels among the real world objects, applications and services that are employed by the IoT applications. This highly distributed structure should provide interconnection with low energy consumption, while distributed data can cause privacy issues.

❏ Middleware system

In IoT, we have heterogeneous devices and networks. Their heterogeneity can potentially increase with new technologies. To facilitate the use of these devices by IoT applications, we should shield their heterogeneity. Therefore, we need to develop a secure, scalable and semantically enriched service-oriented middleware to cope with the heterogeneity of devices.

❏ Platform services

They support a superior management of all involved devices in an integrated way, ensuring scalability, high availability, and the safe and secure execution of the requested functionality from devices.

In continue, we focus on the middleware challenges, because we are

looking for the functionalities that a middleware can provide for the application layer in the IoT-based systems.

2.3 IoT_based Middleware

Bandyopadhyay, S. et. al. have studied the middleware systems that have been applied in IoT-based systems [1]. They classify the required functionality of middleware to manage interaction with a variety of devices in four functional components, namely (1) interface protocols, (2) device abstraction, (3) central control, context detection & management, and (4) application abstraction (shown in Fig. 2.2). In the following, we explain these components in details.

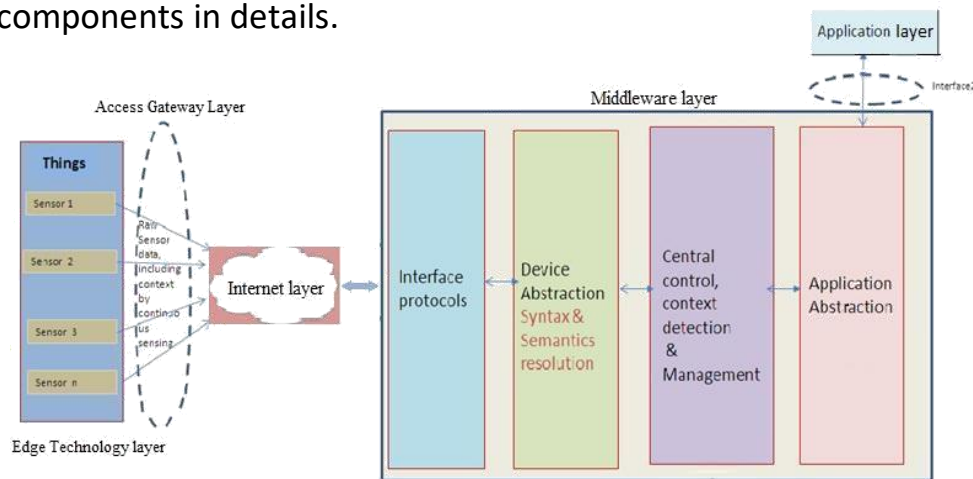


Figure 2.2 Functional components of a middleware for IoT-based systems

2.3.1 Interface protocols

This component is in charge of providing *technical interoperability*. Interoperability in the context of *Interface protocols* means: *the ability of two systems to interoperate by using the same communication protocols*. According to ETSI (European Telecommunications Standards Institute) [14] *technical interoperability* is defined as the *association of hardware or software components, systems and platforms that enable machine-to-machine communication to take place. This kind of interoperability is often centered on (communication) protocols and the infrastructure needed for those protocols to operate*[14].

The Interface Protocol component defines protocols for exchanging information among different networks that may work based on different communication protocols, in order to allow technical interoperability. This component is responsible for handling basic connectivity in the physical and data link, network, transport, and sometimes the application layer of the TCP/IP stack.

To cope with the heterogeneity of devices, we can use a wrapper for each device to translate the protocol supported by the device to a common protocol. This wrapper can be placed either on the device side or the middleware side. If we want to have direct interaction with devices, we should place the wrapper in the middleware side. Devices usually have limited capability of computational process, so this would be a reason to implement wrapper on the middleware side. In contrast, in case of indirect interaction with devices we can develop an intermediary wrapper between the middleware and the devices. The interface protocol component is responsible to allow the middleware to support both direct and indirect interactions.

2.3.2 Device Abstraction (DA)

This component is responsible for providing an abstract format to facilitate the interaction of the application components with devices. This abstraction provides *syntactic* and *semantic* interoperability, which are defined by ETSI [14] as follows:

- ❑ *Syntactic interoperability* is associated with data formats. The messages transferred by communication protocols must have a well-defined syntax and encoding format, which can be represented by using high-level transfer syntaxes such as, HTML and XML.
- ❑ *Semantic interoperability* is usually associated with the meaning of the content of message which is understandable for human. Thus, interoperability on this level means that there is a common understanding among people on the meaning of the content (information) being exchanged among them. However, since DA does not communicate directly with human, semantic interoperability in the context of DA is in charge of providing this common understanding for applications.

Semantic interoperability relies on semantic models which tends to be domain specific. For example, one way to provide semantic interoperability in Service Oriented (SOA) [15] based middleware is by using Devices Profile for Web Services (DPWS) [16] . In this context, each device type refers to a distinguished service type

DA component provide two general functionalities: (1) to ask devices to perform some functionality and (2) to define and configure devices DPWS uses the XML format that is shown in Code 2.1.

```

<dpws:ThisModel>
  <dpws:Manufacturer>ACME Manufacturing</dpws:Manufacturer>
  <dpws:ModelName xml:lang="en-GB" >
    ColourBeam 9
  </dpws:ModelName>
  <dpws:ModelName xml:lang="en-US" >
    ColorBeam 9
  </dpws:ModelName>
</dpws:ThisModel>
</wsx:MetadataSection>
<wsx:MetadataSection
  dialect="http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01/ThisDevice"
>
  <dpws:ThisDevice>
    <dpws:FriendlyName xml:lang="en-GB" >
      ACME ColourBeam Printer
    </dpws:FriendlyName>
    <dpws:FriendlyName xml:lang="en-US" >
      ACME ColorBeam Printer
    </dpws:FriendlyName>
  </dpws:ThisDevice>

```

Code 2.1 XML code to define a device

2.3.3 Central control, Context detection & Management (CCM)

Context characterizes the situation of an entity, which can be a place, a person or an object that is relevant to the user, applications and their interactions [1]. The CCM functional component is responsible to support context-aware computation that is a computational style that take to account the context of the entities that interact with the system. A middleware for IoT-based systems must be context-aware to work in smart environments [1]. Smart environments refer to a *physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network* [17]. Context-awareness includes two functionalities:

- 1) Context detection, which consists of collecting data from resources, and selecting the information that can have an impact on the computation.
- 2) Context processing, to use the gathered information to perform a task or make a decision.

2.3.4 Application Abstraction

This functional component provides an interface for both high-level applications and end-users to interact with devices. For instance, this interface can be a RESTful interface or can be implemented with some query-based language.

A. Malatras. et. al. propose a SOA-based middleware to perform data management and data monitoring services [18]. This middleware uses a RESTful interface to facilitate the interaction of high-level applications with sensors, which can communicate with Wireless Sensor Network (WSN) through the HTTP operations: (1) GET to issue a query on an existing resource, (2) DELETE to remove an existing resource, (3) POST to create a new resource, and (4) PUT to update an existing resource. For instance, a client application gets the result of a domain task by sending a GET request through the URL: „http :/{

hostname}/REST/ {version}/DomaintaskResult/id’. In this URI that reference to an „id“ leads to a unique result.

In the TinyDB middleware [19], an end-user can use a query language to interact with devices. For instance, a user can send the following query to get the id of a sensor

(nodeid) nearby and temperatures that are sensed by this sensor during the past 10 seconds before executing the query:

```
SELECT nodeid, temp
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

2.4 Conclusion

In this chapter, we defined IoT and identified common IoT layers. Furthermore, we discussed a reference middleware architecture for IoT-based systems. This architecture has been proposed by Bandyopadhyay, S. et. al based on a study on the existing middleware frameworks for IoT-based systems [1].

Chapter 3 Review of IoT-based Middleware

The term *Internet of Things (IoT)* refers to a wide set of applications and research areas such as, distributed computing and knowledge management. Much of the IoT research we found in the literature is related to the field of *pervasive computing*. World Wide Web Consortium (W3C)³ defines *pervasive computing* as a *vision about our future computing life style. In this vision, computer systems will be seamlessly integrated into our everyday life, anticipating our needs and providing relevant services and information for us in an anytime-and-any-where fashion. Pervasive Computing is all about making the human's life easier by exploiting the available computing technologies* [20]. This motivates scientists to develop new technologies for the everyday people's lives.

Since the vision of IoT is almost similar to Pervasive computing, we reviewed the literatures that address both IoT and Pervasive systems. To select the middleware to discuss we had the following approaches:

- 1) We looked at typical IoT middleware such as ISMP [21], ASPIRE [22] GSN [23]. We selected HYDRA [24] to review, because it is the most popular and well-documented middleware in comparer to the mentioned middleware. Also, based on the study of Bandyopadhyay, S. et. al [3], those middleware does not support the discussed functional components in Chapter 3, while HYDRA

supports them.

- 2) We looked at typical Pervasive system middleware such as [25], [26], [27]. We selected the following middleware to review:
- ❑ AURA [28] because this middleware focus on elaboration and manipulation of the gathered data from devices. we aim to provide ease of configuration and deployment for end user and developer. Since to meet this goal, we require to facilitate the gathered data manipulation, we reviewed AURA that was the only middleware among the reviewed paper that considered the data manipulation.
 - ❑ TinyDB [19] focus on gathering data from devices. Since in an IoT-based system we need to gather data from environment through different devices, we reviewed TinyDB that is a popular middleware.
 - ❑ WiseMID [29] is the only middleware among the reviewed middleware that is specific for energy saving purpose. As saving the energy of devices is important issue, we reviewed this middleware.

In this chapter, we the mentioned middleware in more detail. Also, we analyze the discussed middleware with respect to the functional components mentioned in Chapter 2.

Chapter 4

Software required:

Arduino ide

Embedded C

SOFTWARE DESIGN

ARDUINO UNO AND ITS PROGRAMMING

Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board.

Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. Arduino projects can be stand-alone, or they can be communicate with software running on your computer. The boards can be assembled by hand or purchased preassembled; the open-source IDE can be downloaded for free.

The Arduino programming language is an implementation of Wiring, a similar physical computing platform, which is based on the Processing multimedia programming environment.

2.1 Overview

The Arduino microcontroller is an easy to use yet powerful single board computer that has gained considerable traction in the hobby and professional market. The Arduino is open-source, which means hardware is reasonably priced and development software is free. This guide is for students in ME 2011, or students anywhere who are confronting the Arduino for the first time. For advanced Arduino users, prowl the web; there are lots of resources.

This guide covers the Arduino Uno board (Spark fun DEV-09950, \$29.95), a good choice for students and educators. With the Arduino board, you can write programs and create interface circuits to read switches and other sensors, and to control motors and lights with very little effort. Many of the pictures and drawings in this guide were taken from the documentation on the

This is what the Arduino ide looks like.



The Duemilanove board features an Atmel ATmega328 microcontroller operating at 5 V with 2 Kb of RAM, 32 Kb of flash memory for storing programs and 1 Kb of EEPROM for storing parameters. The clock speed is 16 MHz, which translates to about executing about 300,000 lines of C source code per second. The board has 14 digital I/O pins and 6 analog input pins. There is a USB connector for talking to the host computer and a DC power jack for connecting an external 6-20 V power source, for example a 9 V battery, when running a program while not connected to the host computer. Headers are provided for interfacing to the I/O pins using 22 g solid wire or header connectors.

The Arduino programming language is a simplified version of C/C++. If you know C, programming the Arduino will be familiar. If you do not know C, no need to worry as only a few commands are needed to perform useful functions.

An important feature of the Arduino is that you can create a control program on the host PC, download it to the Arduino and it will run automatically. Remove the USB cable connection to the PC, and the program will still run from the top each time you push the reset button. Remove the battery and put the Arduino board in a closet for six months. When you reconnect the battery, the last program you stored will run. This means that you connect the board to the host PC to develop and debug your program, but once that is done, you no longer need the PC to run the program.

The Arduino Uno is a microcontroller board based on the ATmega328. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, 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.

The Uno differs from all preceding boards in that it does not use the FTDI USB-to-serial driver chip. Instead, it features the Atmega16U2 (Atmega8U2 up to version R2) programmed as a USB-to-serial converter.

Arduino Software (IDE)

The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus. It connects to the Arduino and Genuino hardware to upload programs and communicate with them.

Writing Sketches

Programs written using Arduino Software (IDE) are called **sketches**. These sketches are written in the text editor and are saved with the file extension .ino. The editor has features for cutting/pasting and for searching/replacing text. The message area gives feedback while saving and exporting and also displays errors. The console displays text output by the Arduino Software (IDE), including complete error messages and other information. The bottom right hand corner of the window

displays the configured board and serial port. The toolbar buttons allow you to verify and upload programs, create, open, and save sketches, and open the serial monitor.

NB: Versions of the Arduino Software (IDE) prior to 1.0 saved sketches with the extension .pde. It is possible to open these files with version 1.0, you will be prompted to save the sketch with the .ino extension on save.



Verify

Checks your code for errors compiling it.



Upload

Compiles your code and uploads it to the configured board. See [uploading](#) below for details.

Note: If you are using an external programmer with your board, you can hold down the "shift" key on your computer when using this icon. The text will change to "Upload using Programmer"



New

Creates a new sketch.



Open

Presents a menu of all the sketches in your sketchbook. Clicking one will open it within the current window overwriting its content.

Note: due to a bug in Java, this menu doesn't scroll; if you need to open a sketch late in the list, use the **File | Sketchbook** menu instead.



Save

Saves your sketch.



Serial

Opens the [serial monitor](#).

Monitor

Additional commands are found within the five menus: **File**, **Edit**, **Sketch**, **Tools**, **Help**. The menus are context sensitive, which means only those items relevant to the work currently being carried out are available.

File

- *New*
Creates a new instance of the editor, with the bare minimum structure of a sketch already in place.
- *Open*
Allows to load a sketch file browsing through the computer drives and folders.
- *Open* *Recent*
Provides a short list of the most recent sketches, ready to be opened.
- *Sketchbook*
Shows the current sketches within the sketchbook folder structure; clicking on any name opens the corresponding sketch in a new editor instance.
- *Examples*
Any example provided by the Arduino Software (IDE) or library shows up in this menu item. All the examples are structured in a tree that allows easy access by topic or library.
- *Close*
Closes the instance of the Arduino Software from which it is clicked.
- *Save*
Saves the sketch with the current name. If the file hasn't been named before, a name will be provided in a "Save as.." window.
- *Save* *as...*
Allows to save the current sketch with a different name.
- *Page* *Setup*
It shows the Page Setup window for printing.
- *Print*
Sends the current sketch to the printer according to the settings defined in Page Setup.
- *Preferences*
Opens the Preferences window where some settings of the IDE may be customized, as the language of the IDE interface.

- *Quit*
Closes all IDE windows. The same sketches open when Quit was chosen will be automatically reopened the next time you start the IDE.

Edit

- *Undo/Redo*
Goes back of one or more steps you did while editing; when you go back, you may go forward with Redo.
- *Cut*
Removes the selected text from the editor and places it into the clipboard.
- *Copy*
Duplicates the selected text in the editor and places it into the clipboard.
- *Copy* *for* *Forum*
Copies the code of your sketch to the clipboard in a form suitable for posting to the forum, complete with syntax coloring.
- *Copy* *as* *HTML*
Copies the code of your sketch to the clipboard as HTML, suitable for embedding in web pages.
- *Paste*
Puts the contents of the clipboard at the cursor position, in the editor.
- *Select* *All*
Selects and highlights the whole content of the editor.
- *Comment/Uncomment*
Puts or removes the // comment marker at the beginning of each selected line.
- *Increase/Decrease* *Indent*
Adds or subtracts a space at the beginning of each selected line, moving the text one space on the right or eliminating a space at the beginning.
- *Find*
Opens the Find and Replace window where you can specify text to search inside the current sketch according to several options.
- *Find* *Next*
Highlights the next occurrence - if any - of the string specified as the search item in the Find window, relative to the cursor position.

- *Find* *Previous*
Highlights the previous occurrence - if any - of the string specified as the search item in the Find window relative to the cursor position.

Sketch

- *Verify/Compile*
Checks your sketch for errors compiling it; it will report memory usage for code and variables in the console area.
- *Upload*
Compiles and loads the binary file onto the configured board through the configured Port.
- *Upload* *Using* *Programmer*
This will overwrite the bootloader on the board; you will need to use Tools > Burn Bootloader to restore it and be able to Upload to USB serial port again. However, it allows you to use the full capacity of the Flash memory for your sketch. Please note that this command will NOT burn the fuses. To do so a *Tools -> Burn Bootloader* command must be executed.
- *Export* *Compiled* *Binary*
Saves a .hex file that may be kept as archive or sent to the board using other tools.
- *Show* *Sketch* *Folder*
Opens the current sketch folder.
- *Include* *Library*
Adds a library to your sketch by inserting #include statements at the start of your code. For more details, see [libraries](#) below. Additionally, from this menu item you can access the Library Manager and import new libraries from .zip files.
- *Add* *File...*
Adds a source file to the sketch (it will be copied from its current location). The new file appears in a new tab in the sketch window. Files can be removed from the sketch using the tab menu accessible clicking on the small triangle icon below the serial monitor one on the right side of the toolbar.

Tools

- *Auto* *Format*
This formats your code nicely: i.e. indents it so that opening and closing curly braces line up, and that the statements inside curly braces are indented more.
- *Archive* *Sketch*
Archives a copy of the current sketch in .zip format. The archive is placed in the same directory as the sketch.
- *Fix* *Encoding* *&* *Reload*
Fixes possible discrepancies between the editor char map encoding and other operating systems char maps.
- *Serial* *Monitor*
Opens the serial monitor window and initiates the exchange of data with any connected board on the currently selected Port. This usually resets the board, if the board supports Reset over serial port opening.
- *Board*
Select the board that you're using. See below for [descriptions of the various boards](#).
- *Port*
This menu contains all the serial devices (real or virtual) on your machine. It should automatically refresh every time you open the top-level tools menu.
- *Programmer*
For selecting a hardware programmer when programming a board or chip and not using the onboard USB-serial connection. Normally you won't need this, but if you're [burning a bootloader](#) to a new microcontroller, you will use this.
- *Burn* *Bootloader*
The items in this menu allow you to burn a [bootloader](#) onto the microcontroller on an Arduino board. This is not required for normal use of an Arduino or Genuino board but is useful if you purchase a new ATmega microcontroller (which normally come without a bootloader). Ensure that you've selected the correct board from the **Boards** menu before burning the bootloader on the target board. This command also set the right fuses.

Help

Here you find easy access to a number of documents that come with the Arduino Software (IDE). You have access to Getting Started, Reference, this guide to the IDE

and other documents locally, without an internet connection. The documents are a local copy of the online ones and may link back to our online website.

- *Find* *in* *Reference*
This is the only interactive function of the Help menu: it directly selects the relevant page in the local copy of the Reference for the function or command under the cursor.

Sketchbook

The Arduino Software (IDE) uses the concept of a sketchbook: a standard place to store your programs (or sketches). The sketches in your sketchbook can be opened from the **File > Sketchbook** menu or from the **Open** button on the toolbar. The first time you run the Arduino software, it will automatically create a directory for your sketchbook. You can view or change the location of the sketchbook location from with the **Preferences** dialog.

Beginning with version 1.0, files are saved with a .ino file extension. Previous versions use the .pde extension. You may still open .pde named files in version 1.0 and later, the software will automatically rename the extension to .ino.

Tabs, Multiple Files, and Compilation

Allows you to manage sketches with more than one file (each of which appears in its own tab). These can be normal Arduino code files (no visible extension), C files (.c extension), C++ files (.cpp), or header files (.h).

Uploading

Before uploading your sketch, you need to select the correct items from the **Tools > Board** and **Tools > Port** menus. The [boards](#) are described below. On the Mac, the serial port is probably something like **/dev/tty.usbmodem241** (for an Uno or Mega2560 or Leonardo) or **/dev/tty.usbserial-1B1** (for a Duemilanove or earlier USB board), or **/dev/tty.USA19QW1b1P1.1** (for a serial board connected with a Keyspan USB-to-Serial adapter). On Windows, it's probably COM1 or COM2 (for a serial board) or COM4, COM5, COM7, or higher (for a USB board) - to find out, you look for USB serial device in the ports section of the Windows Device Manager. On Linux, it should be **/dev/ttyACMx** , **/dev/ttyUSBx** or similar. Once you've

selected the correct serial port and board, press the upload button in the toolbar or select the **Upload** item from the **Sketch** menu. Current Arduino boards will reset automatically and begin the upload. With older boards (pre-Diecimila) that lack auto-reset, you'll need to press the reset button on the board just before starting the upload. On most boards, you'll see the RX and TX LEDs blink as the sketch is uploaded. The Arduino Software (IDE) will display a message when the upload is complete, or show an error.

When you upload a sketch, you're using the Arduino **bootloader**, a small program that has been loaded on to the microcontroller on your board. It allows you to upload code without using any additional hardware. The bootloader is active for a few seconds when the board resets; then it starts whichever sketch was most recently uploaded to the microcontroller. The bootloader will blink the on-board (pin 13) LED when it starts (i.e. when the board resets).

Libraries

Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data. To use a library in a sketch, select it from the **Sketch > Import Library** menu. This will insert one or more **#include** statements at the top of the sketch and compile the library with your sketch. Because libraries are uploaded to the board with your sketch, they increase the amount of space it takes up. If a sketch no longer needs a library, simply delete its **#include** statements from the top of your code.

There is a [list of libraries](#) in the reference. Some libraries are included with the Arduino software. Others can be downloaded from a variety of sources or through the Library Manager. Starting with version 1.0.5 of the IDE, you can import a library from a zip file and use it in an open sketch. See these [instructions for installing a third-party library](#).

To write your own library, see [this tutorial](#).

Third-Party Hardware

Support for third-party hardware can be added to the **hardware** directory of your sketchbook directory. Platforms installed there may include board definitions (which

appear in the board menu), core libraries, bootloaders, and programmer definitions. To install, create the **hardware** directory, then unzip the third-party platform into its own sub-directory. (Don't use "arduino" as the sub-directory name or you'll override the built-in Arduino platform.) To uninstall, simply delete its directory.

For details on creating packages for third-party hardware, see the [Arduino IDE 1.5 3rd party Hardware specification](#).

Serial Monitor

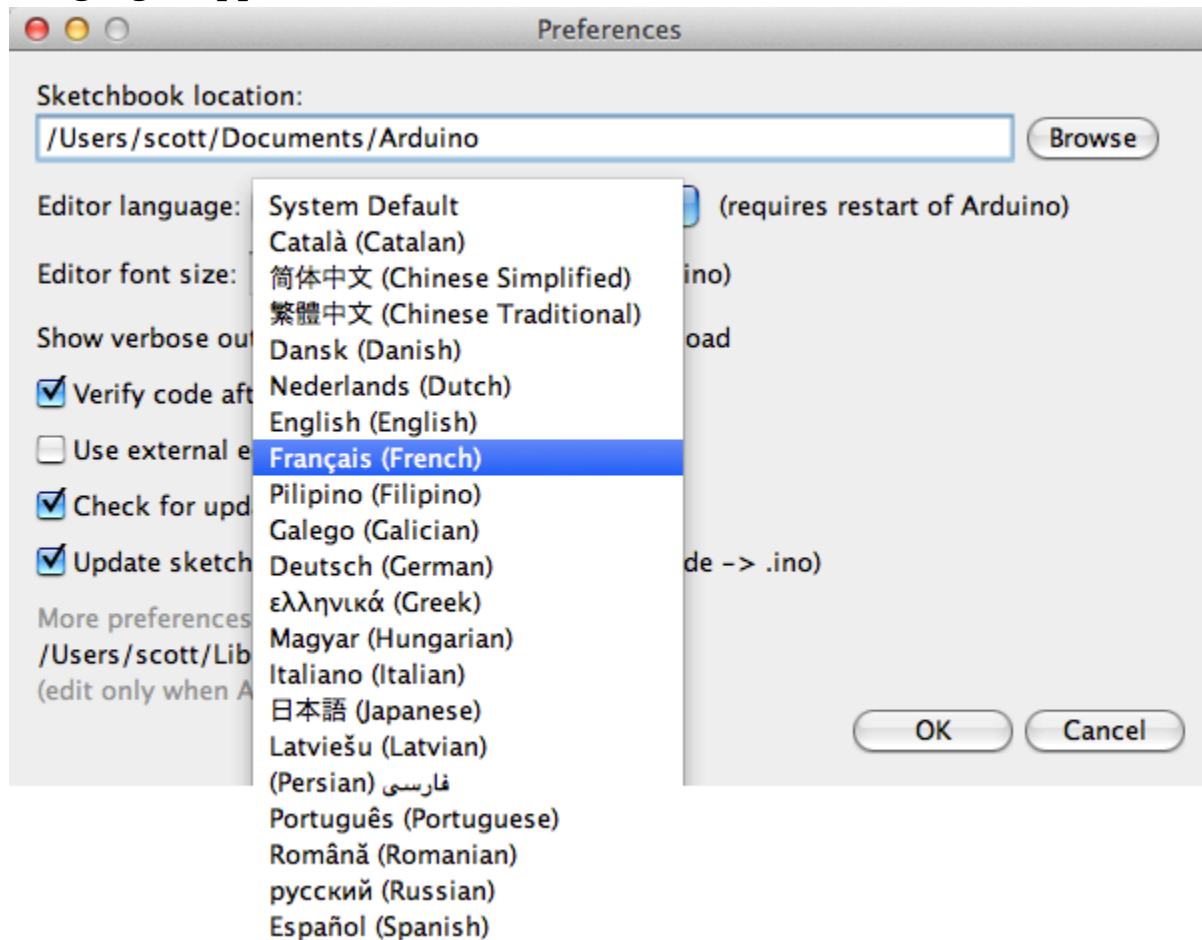
This displays serial sent from the Arduino or Genuino board over USB or serial connector. To send data to the board, enter text and click on the "send" button or press enter. Choose the baud rate from the drop-down menu that matches the rate passed to **Serial.begin** in your sketch. Note that on Windows, Mac or Linux the board will reset (it will rerun your sketch) when you connect with the serial monitor. Please note that the Serial Monitor does not process control characters; if your sketch needs a complete management of the serial communication with control characters, you can use an external terminal program and connect it to the COM port assigned to your Arduino board.

You can also talk to the board from Processing, Flash, MaxMSP, etc (see the [interfacing page](#) for details).

Preferences

Some preferences can be set in the preferences dialog (found under the **Arduino** menu on the Mac, or **File** on Windows and Linux). The rest can be found in the preferences file, whose location is shown in the preference dialog.

Language Support



Since version 1.0.1 , the Arduino Software (IDE) has been translated into 30+ different languages. By default, the IDE loads in the language selected by your operating system. (Note: on Windows and possibly Linux, this is determined by the locale setting which controls currency and date formats, not by the language the operating system is displayed in.)

If you would like to change the language manually, start the Arduino Software (IDE) and open the **Preferences** window. Next to the **Editor Language** there is a dropdown menu of currently supported languages. Select your preferred language from the menu, and restart the software to use the selected language. If your operating system language is not supported, the Arduino Software (IDE) will default to English.

You can return the software to its default setting of selecting its language based on your operating system by selecting **System Default** from the **Editor Language** drop-down. This setting will take effect when you restart the Arduino Software (IDE). Similarly, after changing your operating system's settings, you must restart the Arduino Software (IDE) to update it to the new default language.

Boards

The board selection has two effects: it sets the parameters (e.g. CPU speed and baud rate) used when compiling and uploading sketches; and sets the file and fuse settings used by the burn bootloader command. Some of the board definitions differ only in the latter, so even if you've been uploading successfully with a particular selection you'll want to check it before burning the bootloader. You can find a comparison table between the various boards [here](#).

Arduino Software (IDE) includes the built in support for the boards in the following list, all based on the AVR Core. The [Boards Manager](#) included in the standard installation allows to add support for the growing number of new boards based on different cores like Arduino Due, Arduino Zero, Edison, Galileo and so on.

- *Arduino* *Yùn*
An ATmega32u4 running at 16 MHz with auto-reset, 12 Analog In, 20 Digital I/O and 7 PWM.
- *Arduino/Genuino* *Uno*
An ATmega328P running at 16 MHz with auto-reset, 6 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino Diecimila or Duemilanove w/ ATmega168*
An ATmega168 running at 16 MHz with auto-reset.
- *Arduino Nano w/ ATmega328P*
An ATmega328P running at 16 MHz with auto-reset. Has eight analog inputs.
- *Arduino/Genuino Mega 2560*
An ATmega2560 running at 16 MHz with auto-reset, 16 Analog In, 54 Digital I/O and 15 PWM.
- *Arduino Mega*
An ATmega1280 running at 16 MHz with auto-reset, 16 Analog In, 54 Digital I/O and 15 PWM.

- *Arduino* *Mega* *ADK*
An ATmega2560 running at 16 MHz with auto-reset, 16 Analog In, 54 Digital I/O and 15 PWM.
- *Arduino* *Leonardo*
An ATmega32u4 running at 16 MHz with auto-reset, 12 Analog In, 20 Digital I/O and 7 PWM.
- *Arduino/Genuino* *Micro*
An ATmega32u4 running at 16 MHz with auto-reset, 12 Analog In, 20 Digital I/O and 7 PWM.
- *Arduino* *Esplora*
An ATmega32u4 running at 16 MHz with auto-reset.
- *Arduino* *Mini* *w/ ATmega328P*
An ATmega328P running at 16 MHz with auto-reset, 8 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino* *Ethernet*
Equivalent to Arduino UNO with an Ethernet shield: An ATmega328P running at 16 MHz with auto-reset, 6 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino* *Fio*
An ATmega328P running at 8 MHz with auto-reset. Equivalent to Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega328P, 6 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino* *BT* *w/ ATmega328P*
ATmega328P running at 16 MHz. The bootloader burned (4 KB) includes codes to initialize the on-board bluetooth module, 6 Analog In, 14 Digital I/O and 6 PWM..
- *LilyPad Arduino* *USB*
An ATmega32u4 running at 8 MHz with auto-reset, 4 Analog In, 9 Digital I/O and 4 PWM.
- *LilyPad Arduino*
An ATmega168 or ATmega132 running at 8 MHz with auto-reset, 6 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328P*
An ATmega328P running at 16 MHz with auto-reset. Equivalent to Arduino Duemilanove or Nano w/ ATmega328P; 6 Analog In, 14 Digital I/O and 6 PWM.
- *Arduino NG or older w/ ATmega168*
An ATmega168 running at 16 MHz *without* auto-reset. Compilation and upload is equivalent to Arduino Diecimila or Duemilanove w/ ATmega168, but the bootloader

burned has a slower timeout (and blinks the pin 13 LED three times on reset); 6 Analog In, 14 Digital I/O and 6 PWM.

- *Arduino* *Robot* *Control*
An ATmega328P running at 16 MHz with auto-reset.
- *Arduino* *Robot* *Motor*
An ATmega328P running at 16 MHz with auto-reset.
- *Arduino* *Gemma*
An ATtiny85 running at 8 MHz with auto-reset, 1 Analog In, 3 Digital I/O and 2 PWM.

For instructions on installing support for other boards, see [third-party hardware](#) above.

Last revision 2015/09/07 by SM

The text of the Arduino getting started guide is licensed under a [Creative Commons Attribution-ShareAlike 3.0 License](#). Code samples in the guide are released into the public domain.

UBIDOTS EXPLORER:

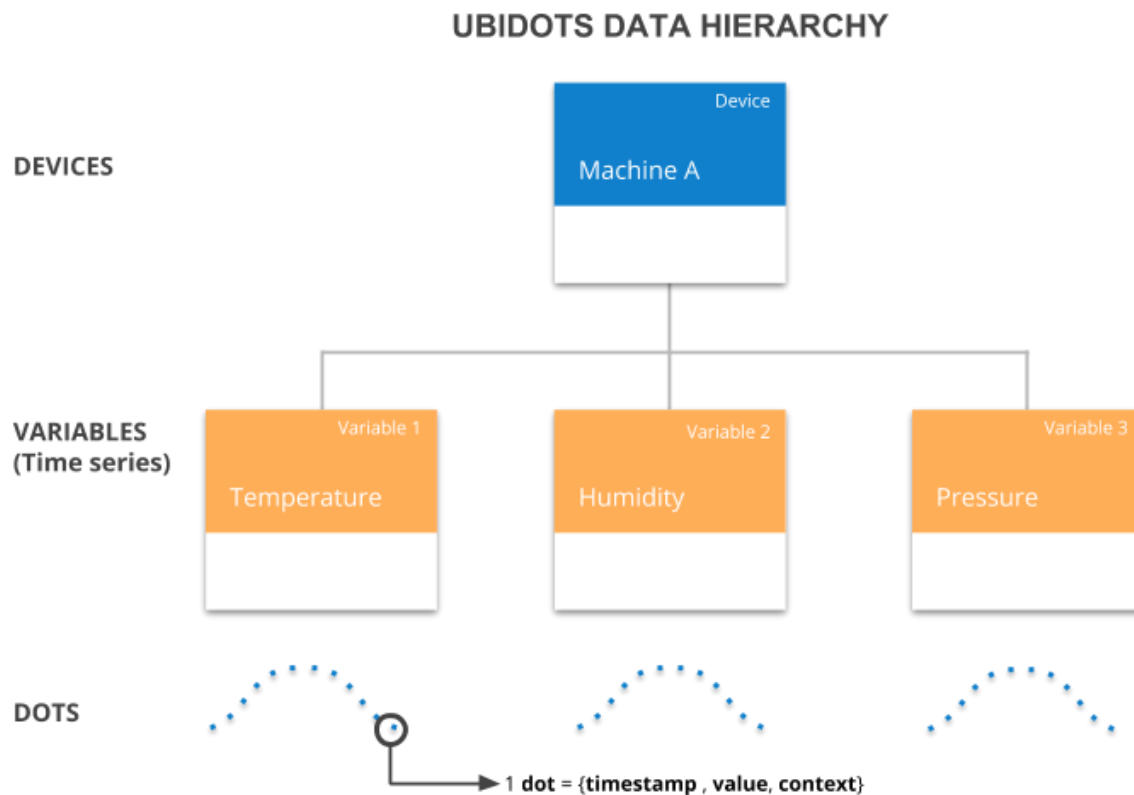
Introduction

Welcome to the Ubidots REST API hardware docs. Here you would reference how to send or retrieve data from your hardware devices using our supported communication protocols: **HTTP**, **MQTT** and **TCP/UDP**.

The purpose of this section is to help you understand what happens in the backstage when communicating with Ubidots, so you can replicate this in your firmware. For this reason, we avoid the use of examples using libraries. However, if you'd like to use one of our libraries (Python, C, and more), check out our [compatible devices](#) and [API clients](#) sections.

How Ubidots Works?

Every time a device updates a sensor value in a variable, a data-point or "**dot**" is created. Ubidots stores dots that come from your devices inside variables, and these stored dots have corresponding **timestamps**:



Each dot contains these items:

Item	Description	Mandatory
value	A numerical value. Ubidots accepts up to 16 floating-point length numbers.	YES
timestamp	Unix Epoch time, in milliseconds. If not specified, then our servers will assign one upon reception.	NO
context	An arbitrary collection of key-value pairs. Mostly used to store the latitude and longitude coordinates of GPS devices.	NO

Values

A numerical value. Ubidots accepts up to 16 floating-point length numbers.

```
{"value" : 34.87654974}
```

Timestamps

A **timestamp**, as best described [here](#), is a way to track time as a running total of seconds. This count starts at the Unix Epoch on January 1st, 1970 at UTC. Therefore, the unix time stamp is merely the number of seconds between a particular date and the Unix Epoch. Please keep in mind that when you send data to Ubidots, you must set the timestamp in milliseconds; also, if you retrieve a dot's timestamp, it will be in milliseconds.

```
"timestamp" : 1537453824000
```

The above timestamp corresponds to Thursday, September 20, 2018 2:30:24 PM.

PRO-TIP: A useful tool to convert between Unix timestamps and human-readable dates is [Epoch Converter](#).

Context

Numerical values are not the only data type supported; you can also store *string* or *char* data types inside what we call **context**. The **context** is a key-value object that allows you to store not only numerical but also string values. An example use of the **context** could be:

```
"context" : {"status" : "on", "weather" : "sunny"}
```

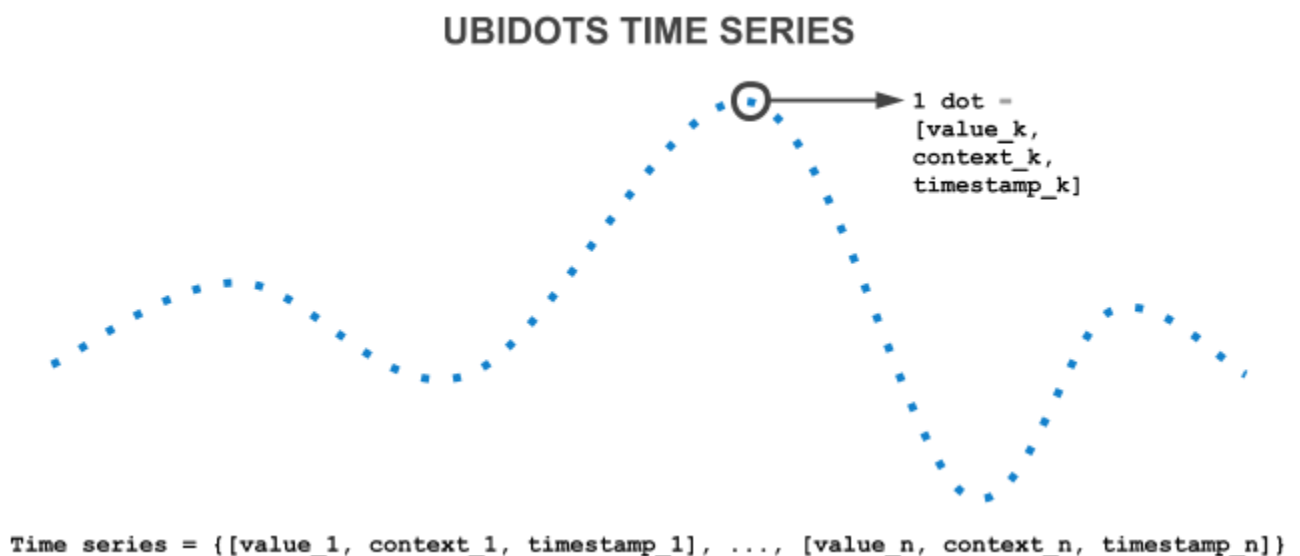
A **context** is commonly used to store the latitude and longitude coordinates of your device for GPS/tracking application use cases. All Ubidots maps uses the *lat* and *lng* keys from a dot's **context** to extract the coordinates of your device, in that way you just need to send a single dot with the coordinates values in the variable context to plot a map instead of sending separately both latitude and longitude in two different variables. Below you can find a typical **context** with coordinates values:

```
"context" : {"lat":-6.2, "lng":75.4, "weather" : "sunny"}
```

Please note that you can mix both string and numerical values in the **context**. If your application is for geo-localization purposes, make sure that the coordinates are set in decimal degrees.

Time Series

Based on the above, we can illustrate an Ubidots time series like this:



Ubidots is an agnostic platform, this means that it does not really care what hardware device you are using, as long as you're able to interact with us through at least one of these protocols:

- [HTTP](#)
- [MQTT](#)
- [TCP/UDP](#)

HTTP

HTTP is the main internet protocol for information exchange in the Internet. It also supports data encryption, which is usually called *secure HTTP* or *HTTPS*. When you use a web browser (MS Edge, Firefox, Google Chrome, etc) you are sending and receiving packages using *HTTP* or *HTTPS*. Because of this, it was not surprising to see *HTTPS* become a popular option for data exchange within IoT.

When a developer is to implement *HTTP* communication, then he should look for a *RESTful Application Programming Interface*, or *REST API*, which exposes all the endpoints required to interact with a third party application (like Ubidots). In this section, you can reference our REST API to understand how to send and retrieve data from our servers.

Note: Our goal with this section is to help you understand what's happening behind when communicating with Ubidots, enabling you to replicate it within your own firmware. Because of this, we avoid using our custom libraries in all the examples.

HTTP requests

The following methods are specified within the *HTTP* standard:

HTTP Method	Description
GET	Used to retrieve information
POST	Used to create a new element
PATCH	Used to update existing elements
DELETE	Used to delete existing elements

HTTP is a **request/response** protocol, this means that every request that you make is answered by the server. This response includes a number (response code) and a

body. For example, when you make a request to retrieve a file on a webpage "(e.g. "Get me the file 'webside.html'")", you build a GET request. If the request is correct, the server will typically return a 200 response code, along with the file requested (body).

Under the same logic, when a device sends a GET request to the Ubidots' server (i.e. requesting the last value of a variable), then the server sends back a response with the status of the request (response code), and a body, which would be the a value with its context and timestamp.

An *HTTP* request also needs the parameters below:

- **Host:** Specifies the server you will be making HTTP requests to.
- **Path:** This is typically the remaining portion of the URL that specifies the resource you want to consume, be it a variable or a device. For example, if an API endpoint is: `industrial.api.ubidots.com/api/v1.6/devices/my-device` then the path would be `/api/v1.6/devices/my-device`
- **Headers:** Define the operating parameters of the *HTTP* request such as authentication, Content-Type, Content-Length, etc.
- **Body/payload:** In the case of POST and PATCH requests, this is the data sent by your device to the server. GET requests typically do not have a body because they are meant to request data, not to send data.

Ubidots accepts data as *JavaScript Object Notation* or *JSON*. *JSON* is a typical *HTTP* data type, it is a collection of name/value pairs. In various programming languages, this is treated as an object, record, struct, dictionary, hash table, keyed list, or associative array. It is also human readable and language independent. An example of a *JSON* data type that Ubidots accepts can be referenced below:

```
{"temperature": {"value":10, "timestamp": 1534881387000, "context": {"machine": "1st floor"}}}
```

A typical *HTTP* request to Ubidots should be set as below:

```
POST {PATH} HTTP/1.1<CR><LN>
```

```
Host: {HOST}<CR><LN>
```

```
User-Agent: {USER_AGENT}<CR><LN>
```

```
X-Auth-Token: {TOKEN}<CR><LN>
```

```
Content-Type: application/json<CR><LN>
```

Content-Length: {PAYLOAD_LENGTH}<CR><LN><CR><LN>
{PAYLOAD}
<CR><LN>

Where:

- {PATH}: Path to the resource to consume
Example: /api/v1.6/variables/ to get user's variables information.
- {HOST}: Host URL.
Example: industrial.api.ubidots.com
- {USER_AGENT}: An optional string used to identify the type of client, be it by application type, operating system, software vendor or software version of the requesting user agent.
Examples: ESP8266/1.0, Particle/1.2
- {TOKEN}: Unique key that authorizes your device to ingest data inside your Ubidots account.
- {PAYLOAD_LENGTH}: The number of characters of your payload.
Example: The payload {"temperature": 20} will have a content-length of 19.
- {PAYLOAD}: Data to send.
Example: {"temperature": 20}

PRO TIP: The easy way of handle *HTTP* and *HTTPs* requests is by using **cURL**, a command line tool. To learn how to install cURL on Windows, MacOSX or Linux please refer to this [guide](#).

API URLs

API access can be over *HTTP* or secure *HTTP*, *HTTPs*, using the following endpoints based on your Ubidots Account type.

Security Note: We strongly advise to use *HTTPs* to make sure your data travels encrypted, avoiding the exposure of your API token and/or sensor data.

HTTP

Ubidots Account	Endpoint	Port
Educational	http://things.ubidots.com	80

Ubidots Account	Endpoint	Port
Industrial	http://industrial.api.ubidots.com	80

HTTPs

Ubidots Account	Endpoint	Port
Educational	https://things.ubidots.com	443
Industrial	https://industrial.api.ubidots.com	443

Ubidots supports SSL v1.1, TLS v1.2 and v1.3. You can download our root certificates in different formats:

- [PEM file](#): A certificate chain with two root certificates from our certificate authorities (CAs).
- [DER file](#): Same as the PEM file, with an alternative encoding.
- [CRT file](#): Same as the PEM file, with a different extension. Often referred to as .crt, .cert or .cer.

NOTE: To avoid future inconveniences, we advise to **use both certificates** in your device's auth routines. While the second is the current valid certificate, we will use the first after 2021.

Fixed IPs

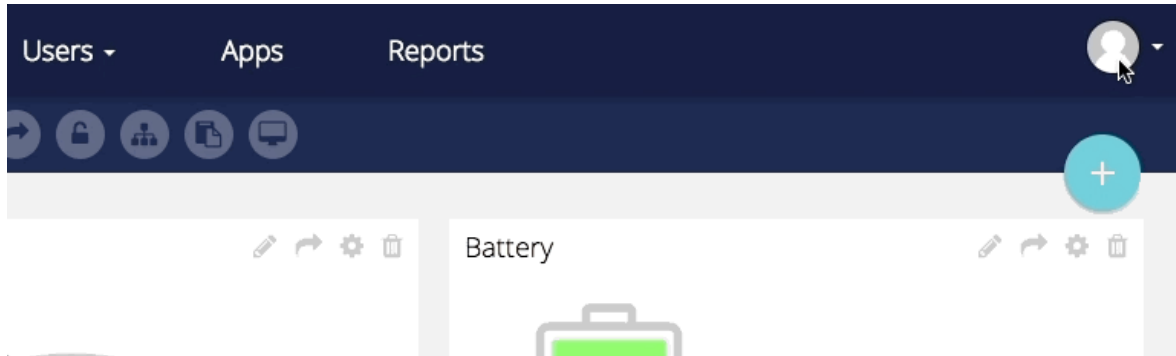
Ubidots also supports fixed IPs address for data ingestion, but we highly advise to our users to implement their firmware routines using the dns endpoints instead of fixed IPs, as these last ones may change in the future.

Ubidots Account	ipv4	ipv6	Port
Educational	50.23.124.68	2607:f0d0:2101:39::2	80/443
Industrial	169.55.61.243	2607:f0d0:3:b::2	80/443

HTTP Authentication

Every request requires a *token*. A *token* is an unique key that authorizes your device to ingest data inside your Ubidots account.

The easiest way to get yours is clicking on "**API Credentials**" under your profile tab:



Also, 6 hour valid *tokens* can be generated using your account *API-KEY*. If you desire to learn how to do it, please refer the [Ubidots REST Software API reference](#).

There are two ways to send a *token* in a request:

1. Send the *token* as parameter in the endpoint, setting it with the `?token=` expression. This is the easiest one to implement but the most unsecure method to send your authentication credentials.
2. Send the *token* as a request header. For this, use the `X-Auth-Token` header.

Security Note: While sending your *token* as a URL parameter can be quite comfortable in some devices, we only advise to do this for prototyping stages. For production stages, we strongly advise sending the token using the `X-Auth-Token` header.

```
GET
https://industrial.api.ubidots.com/api/
```

Send data

Our REST API allows you send data to the platform in two different ways:

- Send data to a **device**
- Send data to a **variable**

If you're looking for more advanced API capabilities, such as editing a device name, or listing all of your devices, please refer to our Ubidots Software REST API Reference.

Send data to a device

Request structure:

```
POST /api/v1.6/devices/{DEVICE_LABEL} HTTP/1.1<CR><LN>
Host: {Host}<CR><LN>
User-Agent: {USER_AGENT}<CR><LN>
X-Auth-Token: {TOKEN}<CR><LN>
Content-Type: application/json<CR><LN>
Content-Length: {PAYLOAD_LENGTH}<CR><LN><CR><LN>
{PAYLOAD}
<CR><LN>
```

Expected Response:

```
HTTP/1.1 200 OK<CR><LN>
Server: nginx<CR><LN>
Date: Tue, 04 Sep 2018 22:35:06 GMT<CR><LN>
Content-Type: application/json<CR><LN>
Transfer-Encoding: chunked<CR><LN>
Vary: Cookie<CR><LN>
Allow: GET, POST, HEAD, OPTIONS<CR><LN><CR><LN>
{PAYLOAD_LENGTH_IN_HEXADECIMAL}<CR><LN>
{"{VARIABLE_LABEL}": [{"status_code": 201}]}<CR><LN>
0<CR><LN>
```

The easiest way to **send values to Ubidots** is by specifying your Device label in the request *path* and making a POST request to it.

Replace the {DEVICE_LABEL} with the unique label of the desired device to send data to. If the device does not exist in your Ubidots account, it will be automatically created. Device labels can contain any alphanumerical character, without blank spaces, i.e *weather-station*, *my-home*, *esp8266*.

To set the variables values, just create a *JSON* data payload as below

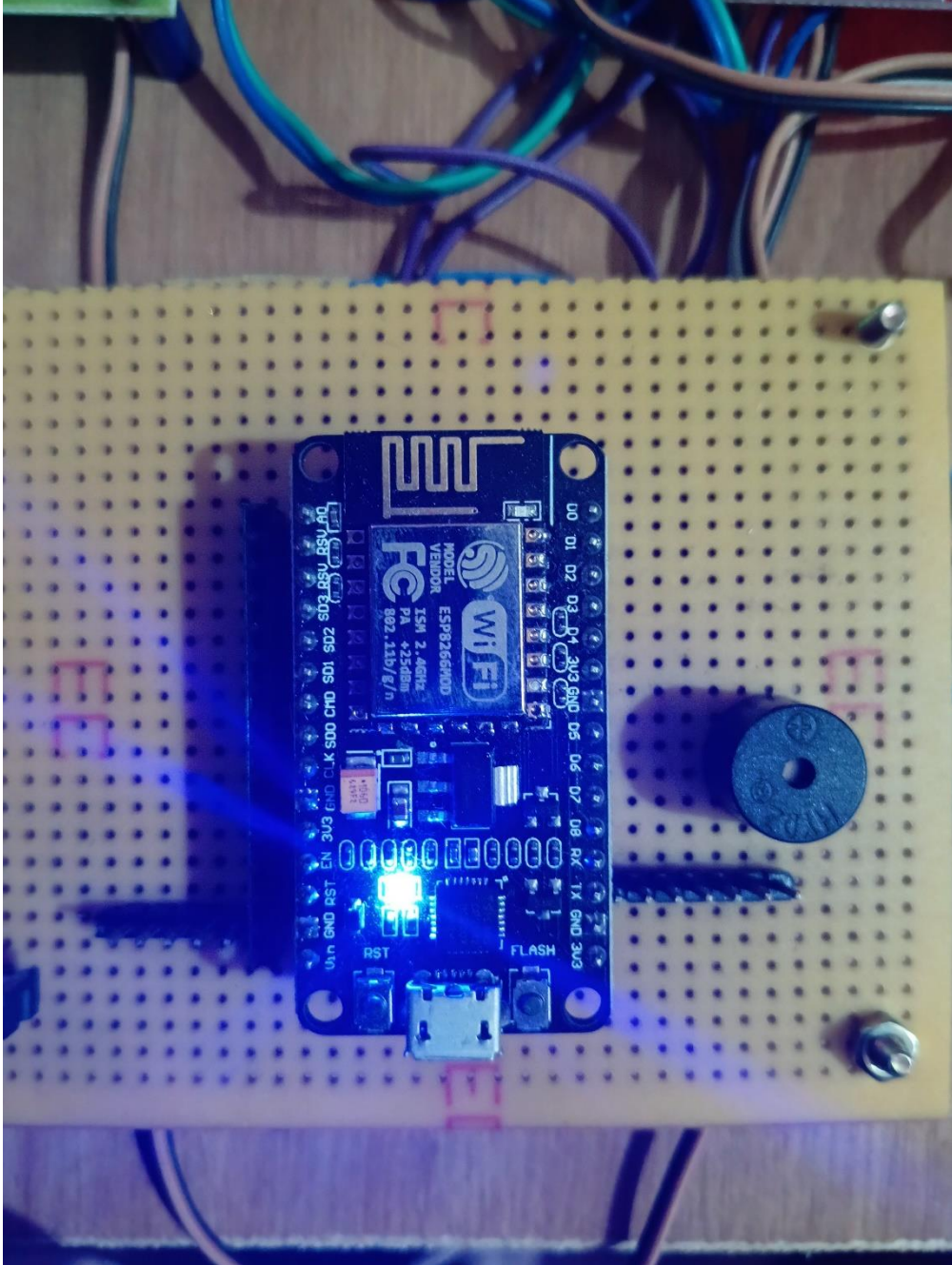
N° Variables	JSON structure
One	<code>{"variable-label": VALUE}</code>
Multiple	<code>{"variable-label-1": {"value": VALUE, "timestamp": TIMESTAMP, "context": {"key": VALUE}}, ... , "variable-label-n": {"value": VALUE, "timestamp": TIMESTAMP, "context": {"key": VALUE}}}</code>

timestamp and *context* parameters are optional.

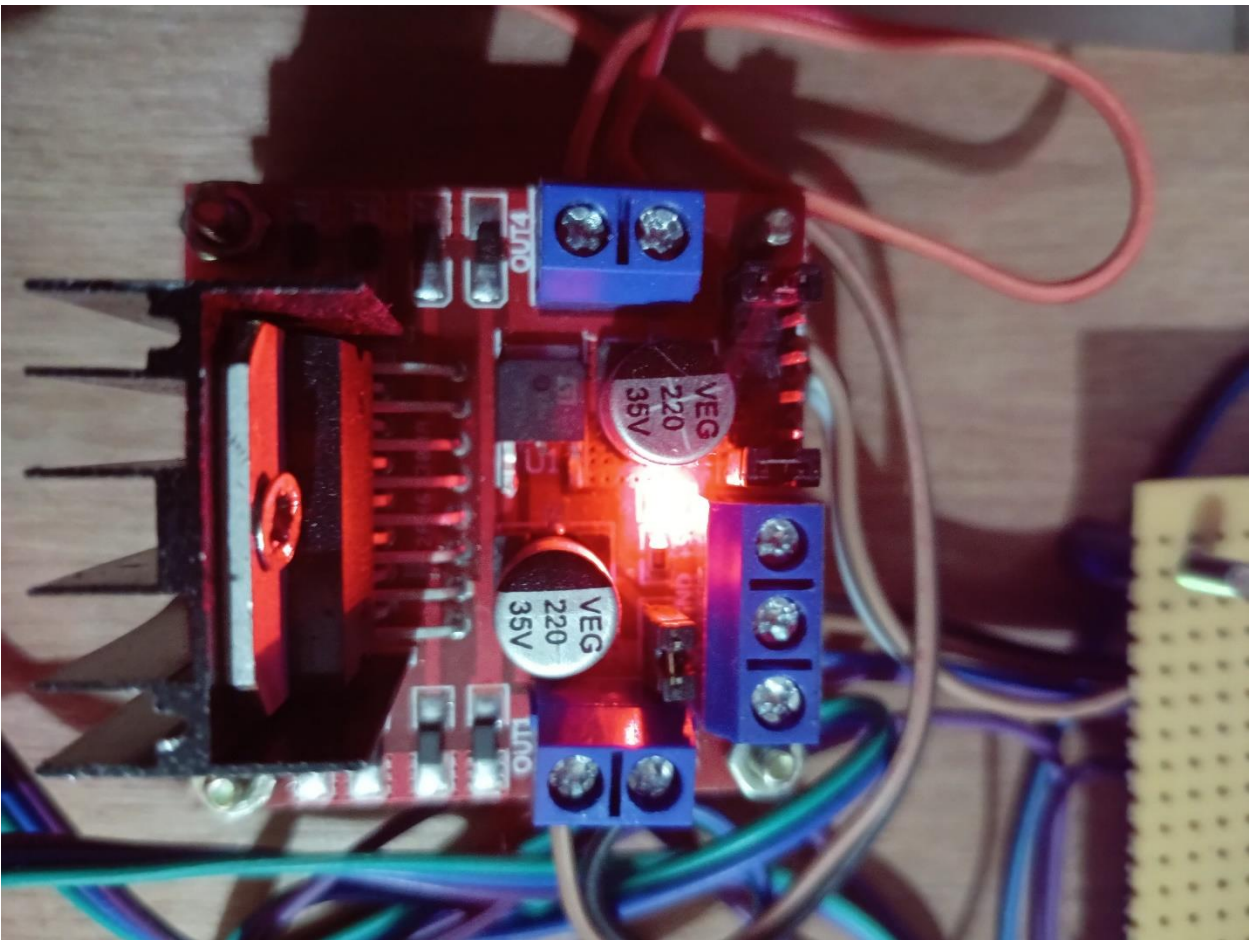
NOTE 1: By default, the Ubidots backend will take the latitude and longitude values inside any variable labeled as **"gps"**, **"location"**, **position** or **"coordinates"** automatically to get your device coordinates in all of the map widgets. Alternatively, you change the default location variable.

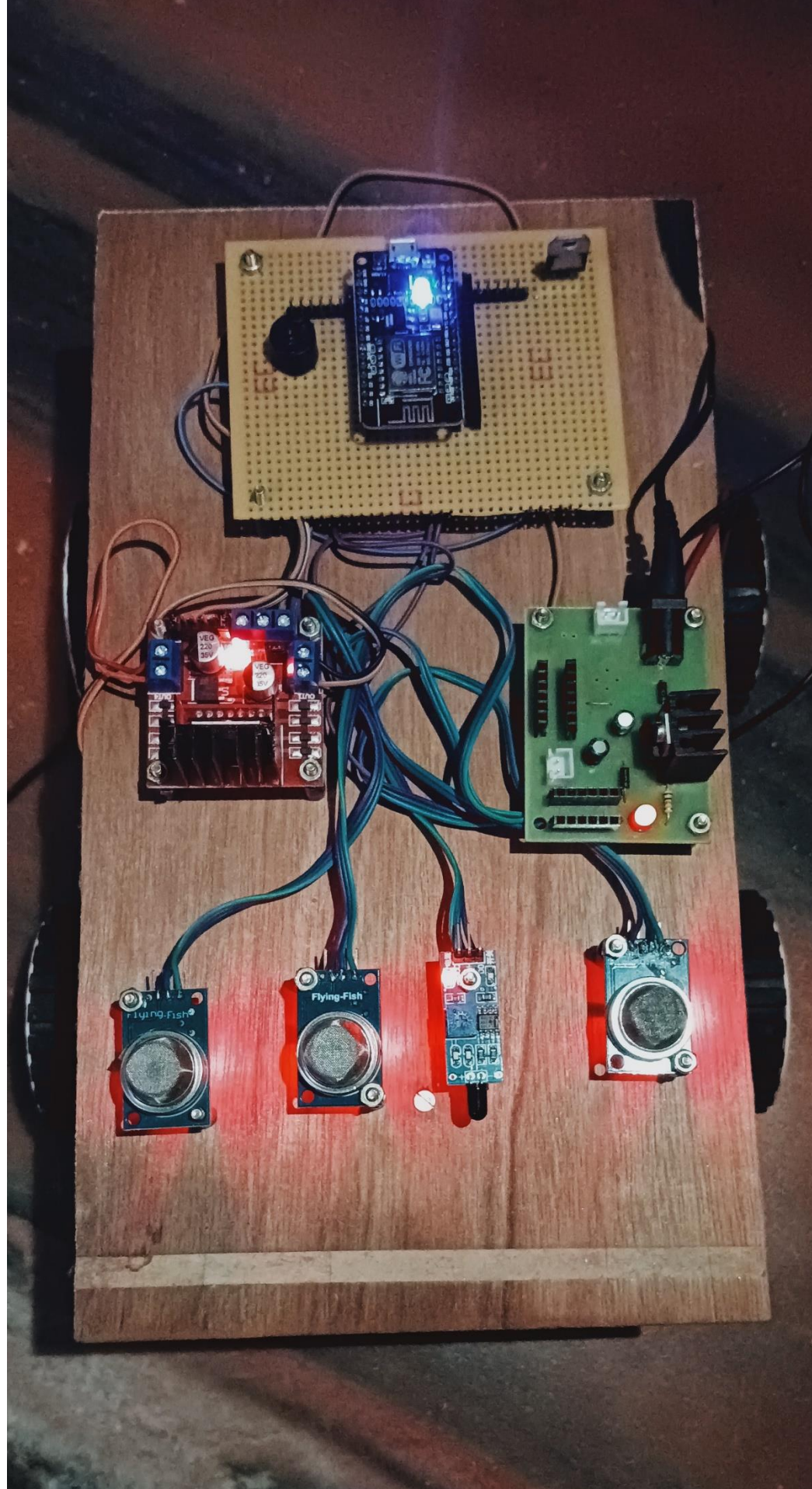
NOTE 2: All timestamps must be in milliseconds. If you set a timestamp for a variable value, it will be stored with the date represented by it. You can easily convert dates to timestamps values [here](#).

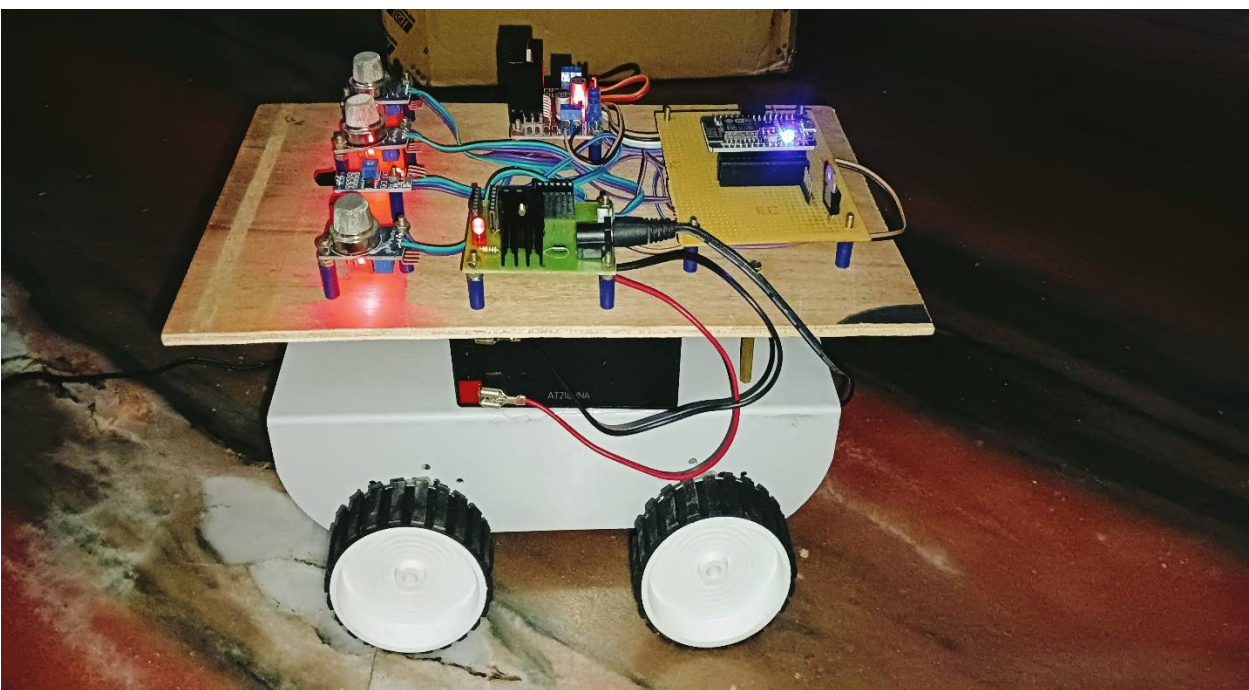
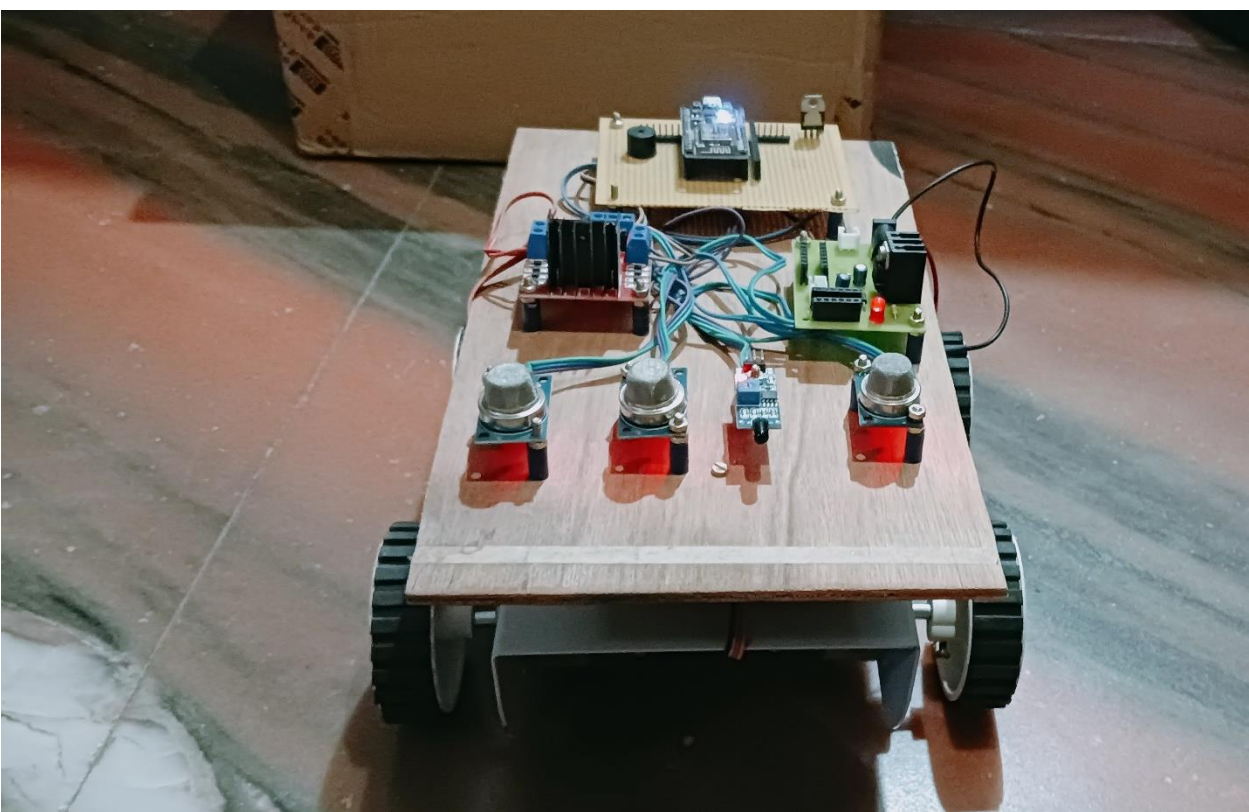
SETUP:











CODE:

```
014_fire | Arduino 1.8.19 (Windows Store 1.8.57.0)
File Edit Sketch Tools Help

#include "Ubidots.h"

const char* UBIDOTS_TOKEN = "BBFF-ytVYWSrs9f00k4BC8FhJ7nzQuNTTSR"; // Put here your Ubidots TOKEN
const char* WIFI_SSID = "fire"; // Put here your Wi-Fi SSID
const char* WIFI_PASS = "12345678"; // Put here your Wi-Fi password
const char* DEVICE_LABEL_TO_RETRIEVE_VALUES_FROM = "c82b962f4353"; // Replace with your device label
const char* VARIABLE_LABEL_TO_RETRIEVE_VALUES_FROM = "control";
Ubidots ubidots(UBIDOTS_TOKEN, UBI_HTTP);

int rightMotor1 = D0; // D7 - right Motor -
int rightMotor2 = D1; // D8 - right Motor +
int leftMotor2 = D2; // D3 - left Motor -
int leftMotor1 = D3;

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

  pinMode(D0, INPUT);
  pinMode(D1, INPUT);
  pinMode(D2, INPUT);
  pinMode(D3, INPUT);

  pinMode(10, OUTPUT);
  digitalWrite(10, LOW);

  pinMode(leftMotor1, OUTPUT);
  pinMode(leftMotor2, OUTPUT);
  pinMode(rightMotor1, OUTPUT);
}

Done compiling.
BSS : 25072 - zeroed variables (global, static) in RAM/HEAP
Sketch uses 401920 bytes (38%) of program storage space. Maximum is 1044464 bytes.
Global variables use 28860 bytes (35%) of dynamic memory, leaving 53060 bytes for local variables. Maximum is 81920 bytes.

NodeMCU 1.0 (ESP-12E Module), 80 MHz Flash, Legacy (new can return multi), All SSL ciphers (most compatible), 4MB (FS:2MB OTA:~1019KB), 2 v2 Lower Memory, Disabled, None, Only Sketch, 115200
ENG IN 11:22 PM 10-03-2022
```

```
014_fire | Arduino 1.8.19 (Windows Store 1.8.57.0)
File Edit Sketch Tools Help

pinMode(rightMotor1, OUTPUT);
pinMode(rightMotor2, OUTPUT);
digitalWrite(leftMotor1, LOW);
digitalWrite(leftMotor2, LOW);
digitalWrite(rightMotor1, LOW);
digitalWrite(rightMotor2, LOW);

ubidots.wifiConnect(WIFI_SSID, WIFI_PASS);
}

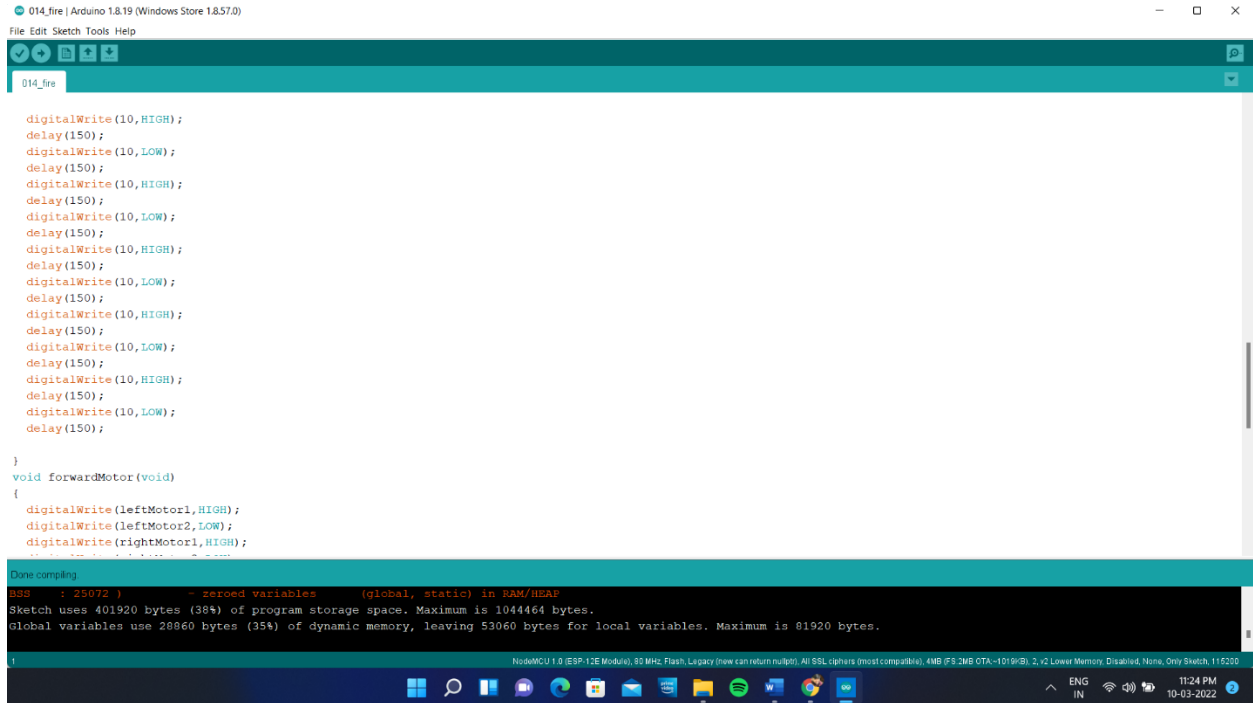
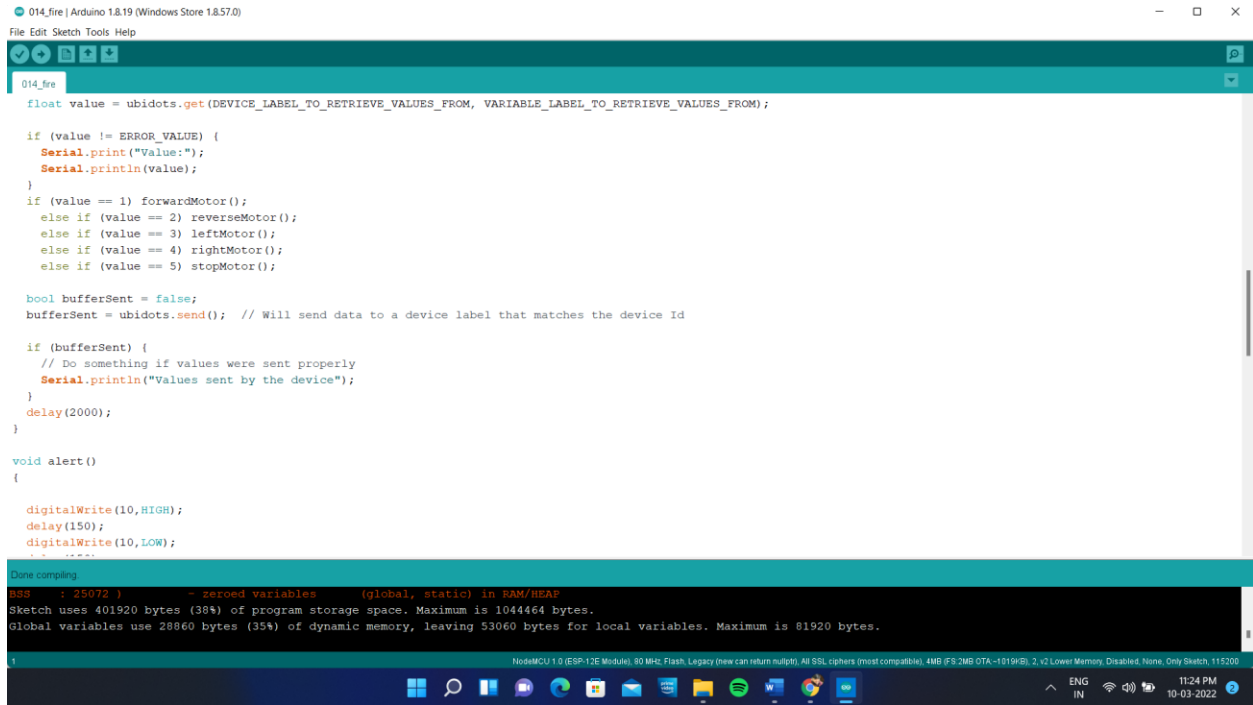
void loop() {

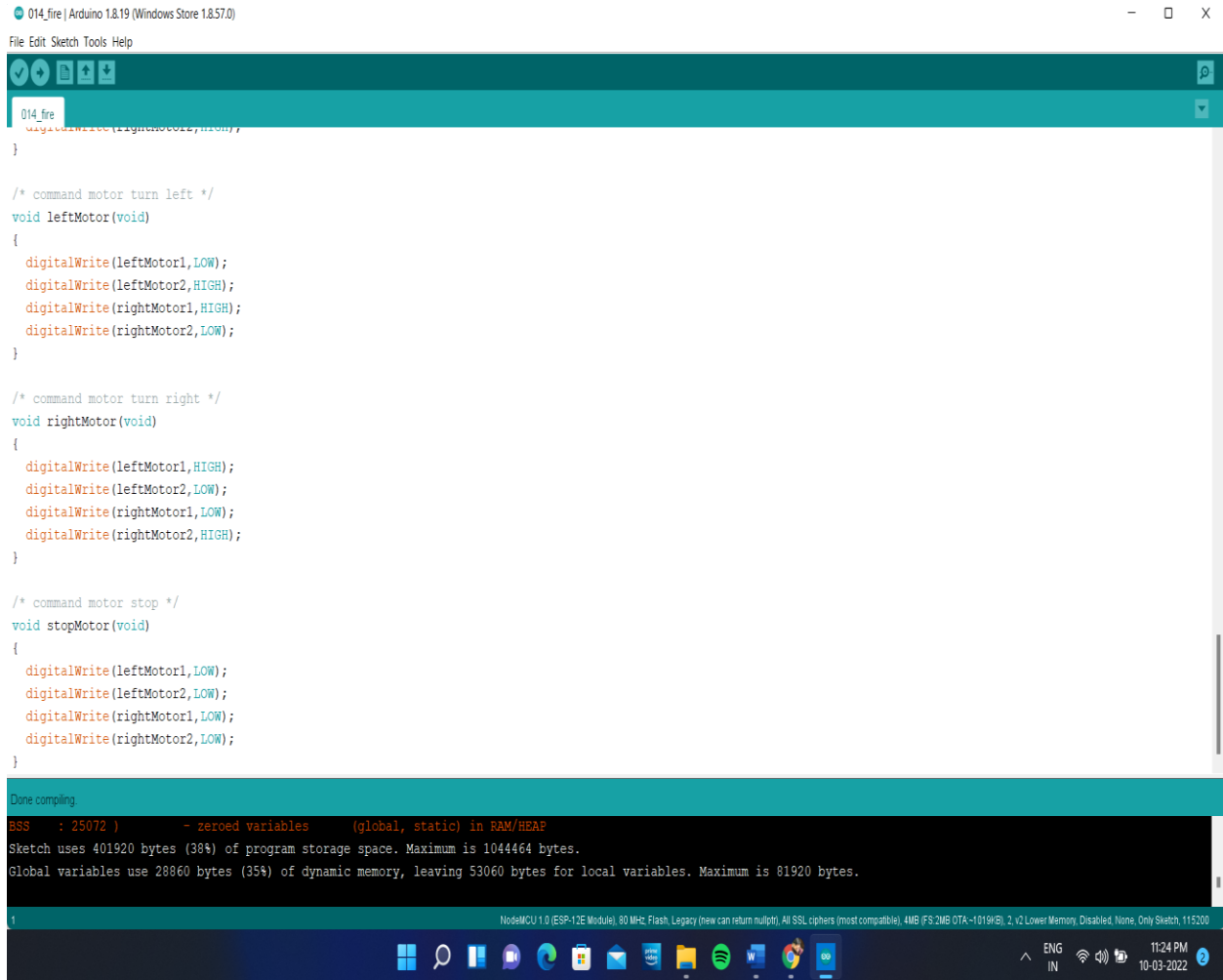
  int Methane = analogRead(A0);
  int co2 = digitalRead(D5);
  int flame = digitalRead(D6);
  int smoke = digitalRead(D7);
  if (co2 == 0 || flame == 0 || smoke == 0 || Methane > 800)
  {
    Serial.print(co2); Serial.print(","); Serial.print(flame); Serial.print(",");
    Serial.print(smoke); Serial.print(","); Serial.print(Methane); Serial.print(",");
    alert();
  }
  ubidots.add("Methane", Methane);
  ubidots.add("co2", co2);
  ubidots.add("flame", flame);
  ubidots.add("smoke", smoke);

  float value = ubidots.get(DEVICE_LABEL_TO_RETRIEVE_VALUES_FROM, VARIABLE_LABEL_TO_RETRIEVE_VALUES_FROM);

Done compiling.
BSS : 25072 - zeroed variables (global, static) in RAM/HEAP
Sketch uses 401920 bytes (38%) of program storage space. Maximum is 1044464 bytes.
Global variables use 28860 bytes (35%) of dynamic memory, leaving 53060 bytes for local variables. Maximum is 81920 bytes.

NodeMCU 1.0 (ESP-12E Module), 80 MHz Flash, Legacy (new can return multi), All SSL ciphers (most compatible), 4MB (FS:2MB OTA:~1019KB), 2 v2 Lower Memory, Disabled, None, Only Sketch, 115200
ENG IN 11:23 PM 10-03-2022
```





CONCLUSION:

This research's main objective is to build a low-cost and straightforward system that can be operated at three modes: gas and fire leakage detection mode and temperature detection mode. In many research papers, it almost uses the message alert system. When the gas, smoke, or fire leaks, it will send a message alert to the user phone. But we use a call alert system, which gives some advantages to users. Moreover, we deployed a system with different sensors. Overall it is user friendly and low cost. Besides, it doesn't give not only the call alert but also a graphical alert on the webserver. In previous works, the raspberry pi uses for getting the accurate output. We tried to do it by NodeMCU and applied machine learning to get sensors' accuracy or prediction ability in critical situations. Future work of this research will include data analytics that will apply to the cloud side to improve the system's services and accuracy. Also, use multiple gases and fire sensor for detecting gas and fire leakage in industries and prepare a database for gathering sensors output. Besides, somebody will develop it for calling or messaging multiple users by creating a database in the future. GSM data will be stored in the database and transmit to multiple users. Furthermore, a gas and fire sensing robot can be constructed to sense Gas Leakage through pipelines.

References:

- [1] Anusha, O., & Rajendra prasad, C. H. Experimental investigation on road safety system at crossings. *International Journal of Engineering and Advanced Technology*, 2019; 8(2):214–218.
- [2] Pravalika, V., & Rajendra Prasad, C. Internet of things based home monitoring and device control using Esp32. *International Journal of Recent Technology and Engineering*, 2019; 8(1 Special Issue 4):58–62.
- [3] Sanjay Kumar, S., Ramchandrar Rao, P., & Rajendra Prasad, C. Internet of things based pollution tracking and alerting system. *International Journal of Innovative Technology and Exploring Engineering*, 2019; 8(8):2242– 2245
- [4] Deepak, N., Rajendra Prasad, C., & Sanjay Kumar, S. Patient health monitoring using IOT. *International Journal of Innovative Technology and Exploring Engineering*, 2018; 8(2):454–457. <https://doi.org/10.4018/978-1-5225-8021-8.ch002>.
- [5] Murugam, K. (2020). Intelligent gas booking and leakage system using wireless sensor networks. *3C Tecnología. Glosas de innovación aplicadas a la pyme. Edición Especial, Marzo 2020*, 273-285.
<http://doi.org/10.17993/3ctecno.2020.specialissue4.273-285>
- [6] Padma P. K., Surekha M., Preethi R., Devika T., & Dhivya N. (2014). Smart gas cylinder using embedded system. *International journal of innovative research in electrical, electronics, instrumentation and control engineering*. Vol. 2, www.ijireeice.com.
- [7] Olubusola O. N., Kamoli A. A., and Ayorinde J. O. (2017). GSM-Based Gas Leakage Detection and Alert System. *ABUAD Journal of Engineering Research and Development (AJERD) Volume 1, Issue 1*, 100-104.
<http://ajerd.abuad.edu.ng/>.
- [8] Folorunso C.O., Raheem W.A., Akinyemi L.A., & Raji A.A. (2019). Proposed

and development of a Liquidified Petroleum Gas Leakage Detector, Level Indicator and Automatic Shutdown System. Covenant Journal of Engineering Technology (CJET) Vol.3 No.2, ISSN: p. 2682-5317 e. 2682-5325.

[9] Rahul G., Shubham M., Salman K., Raghunandan G., and Rahul J. (2018). GSM based gas leakage detection system. International Journal of Advance Research in science and engineering. Volume No.07, ISSN:2319-8354. www.ijarse.com.