

SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER

Submitted in partial fulfillment of the requirements for the award of
Bachelor of Engineering degree in Electronics and Communication
Engineering

By

BHUMIREDDY PAVAN KUMAR REDDY (37130057)
BANDARU VENKATA SAI ARUN KUMAR (37130044)



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

**(DEEMED TO BE UNIVERSITY)
Accredited with Grade “A” by NAAC**

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119

March – 2021



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited with “A” grade by NAAC

Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600 119

www.sathyabama.ac.in

DEPARTMENT OF ELECTRONICS AND

COMMUNICATION ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of BHUMIREDDY PAVAN KUMAR REDDY (37130057), BANDURU VENKATA SAI ARUN KUMAR (37130044), who carried out the project entitled “**SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER**” under my/our supervision from SEPTEMBER 2020 to MARCH 2021.

INTERNAL GUIDE

DR.G.D.ANBARASI JEBASELVI,M.E.,Ph.D

Associate Professor, Department of ECE

Head of the Department

Dr.T.Ravi,M.E,Ph.D

Submitted for Viva voce Examination held on APRIL 2021

Internal Examiner

External Examiner

DECLARATION

WE, BHUMIREDDY PAVAN KUMAR REDDY (37130057), BANDARU VENKATA SAI ARUN KUMAR (37130044). hereby declare that the Project Report entitled “**SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER**” done by us under the guidance of Dr..GD.ANBARASI JEBASELVI, M.E.,Ph.D is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in **ELECTRONICS AND COMMUNICATION ENGINEERING**

DATE:

PLACE:

1.

2.

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

We are pleased to acknowledge our sincere thanks to Board of Management of SATHYABAMA for their kind encouragement in doing this project and for completing it successfully. We are grateful to them.

We convey Our thanks to Dr.N.M.NANDHITHA,M.E.,Ph.D. Dean, School, Dept. of ELECTRONICS AND ELECTRICAL 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.

We would like to express our sincere and deep sense of gratitude to our Project Guide Dr.GD.ANBARASI JEBASELVI, M.E.Ph.D Associate Professor for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

We wish to express our 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.

We express our gratitude to our parents for their constant encouragement and support for the completion of the project.

Abstract:

Carbon monoxide, nitrogen oxides, and hydrocarbons are released when fuel is burned in an internal combustion engine and when air/fuel residuals are emitted through the vehicle tailpipe. Gasoline vapors also escape into the atmosphere during refueling and when fuel vaporizes from engines and fuel systems caused by vehicle operation or hot weather. The pollutants in vehicle emissions are known to damage lung tissue, and can lead to and aggravate respiratory diseases, such as asthma. Motor vehicle pollution also contributes to the formation of acid rain and adds to the greenhouse gases that cause climate change. Pollutants emitted directly from vehicles are not the only cause for concern. On warm, sunny days, hydrocarbons react with oxides of nitrogen to create a secondary pollutant, ozone. In many urban areas, motor vehicles are the single largest contributor to ground-level ozone which is a common component of smog. Ozone causes coughing, wheezing and shortness of Breath, and can bring on permanent lung damage, making it a cause of crucial public health problem. This problem can be rectified by using some kind of sensor called smoke sensor using this sensor connected to internet it directly Send the data to government.

Table of content

SL.NO	DESCRIPTION	PAGE NO
1	BONAFIDE CERTIFICATE	2
2	DECLARATION	3
3	ACKNOWLEDGEMENT	4
4	ABSTRACT	5
5	CHAPTER 1	7
	1.1-INRODUCTION	
	1.2-LITERATURE SURVEY	
	1.3-EXISTING SYSTEM	
	1.4-PROPOSED SYSTEM	
	1.5-BLOCK DIAGRAM	
	1.6-HARDWARE REQUIRED	
6	CHAPTER-2	12
	2.1-EMBEDDED SYSTEMS	
	2.2-OVERVIEW OF ENBEDDED SYSTEMS	
	2.3-CHARACTERISTICS OF ENBEDDED SYSTEMS	
	2.4-POWER SUPPLY	
	2.5-TRANSFORMER,RECTIFIER,FILTERS	
	2.6-REGISTERS,REGULATORS	
7	CHAPTER-3	27
	3.1-NODE MCU	
	3.2-ARDUINO CORE	
8	CHAPTER	32
	4.1-IOT	
	4.2-MOTIVATION	
	4.3-LAYERS IN IOT	
	4.4-CHALLENGES	
	4.5-IOT MIDDLEWARE	
9	CHAPTER-5	41
	5.1-CONCLUSION	
	5.2 SENSOR MODULES	
	5.3-SOFTWARE REQUIRED	
	5.4-UBIDOTS EXPLORER	
10	CHAPTER-6	100
	6.1 REFERENCE	

SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER

CHAPTER 1

1.1 Introduction:

- THE current advances in the fields of technology and economy are having a significant impact over the environment, and have led to serious concerns regarding pollution and climate change. In the report released in 2014, the Intergovernmental Panel on Climate Change confirms that human activities are having an unequivocal and continuously increasing influence on the climate system, with recent changes that are unprecedented over decades to millennia.
- In this context, environmental monitoring represents a fundamental instrument for gathering relevant information about the ecosystem, leading to new knowledge and understanding, and for ultimately implementing adaptation and mitigation actions that address the degradation of the biosphere.
- Another related field currently gaining attention is the monitoring of the ambient in indoor spaces. It has been shown that this is important not only for the comfort and health of building occupants, but also for the reduction of greenhouse gas emissions.
- The most common parameters being monitored include the temperature, humidity, rainfall, atmospheric pressure, light intensity, air quality, affected by pollutants such as CO₂, CO, SO_x, volatile organic compounds, and many others. One of the immediate benefits brought by the acquisition of such physical properties, like soil moisture, temperature, and salinity, can be seen in agriculture, where significant water resource savings can be achieved.

- The majority of monitoring applications rely on WSNs, motivated by the indisputable advantages they bring: lower costs due to the replacement of cables, variable network topologies, scalability, and lower maintenance and commissioning expenses.

1.2 Literature survey:

1. A Smart Air Pollution Monitoring System

Author: Kennedy Okokpuije , Etinosa Noma-Osaghae, Modupe Odusami, Samuel John

Air pollution affects our day to day activities and quality of life. It poses a threat to the ecosystem and the quality of life on the planet. The dire need to monitor air quality is very glaring, owing to increased industrial activities over the past years. People need to know the extent to which their activities affect air quality. This project proposes an air pollution monitoring system. The system was developed using the Arduino microcontroller.

2. Air Quality Monitoring System

Author: 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.

3. An air quality monitoring system for urban areas based on the technology of wireless sensor networks

Author: Jen-Hao Liu, Yu-Fan Chen, Tzu-Shiang Lin, Chia-Pang Chen.

Air pollution is one of environmental issues that cannot be ignored. The heavy transportation and urbanization result in the air pollutants concentrated in certain areas. Inhaling pollutants for a long time causes damages in human health. Traditional air quality monitoring methods, such as building air quality monitoring stations, are typically expensive. In addition, monitoring stations are generally less densely deployed and

provide low resolution sensing data. This paper proposes an urban air quality monitoring system based on the technology of wireless sensor networks (WSNs).

4. Iot Based Quality Monitoring System

Author: Ch.V.Saikumar, M.Reji,

P.C.Kishoreraja

- The main objective of this project is to monitor the air eminence in industrial and urban areas. The proposed outline includes a set of gas sensors (CO, and NO₂) that are positioned on masses and structure of a IOT (Internet of things) and a dominant server to support both short-range real-time incident management and a continuing deliberate planning. In this Arduino platform is used to communicate the data simply and quickly. WSN (Wireless sensor network) acts as the transreceiver. This provide a real-time low rate monitoring system over the use of low rate, low information rate, and little control wireless communication technology.

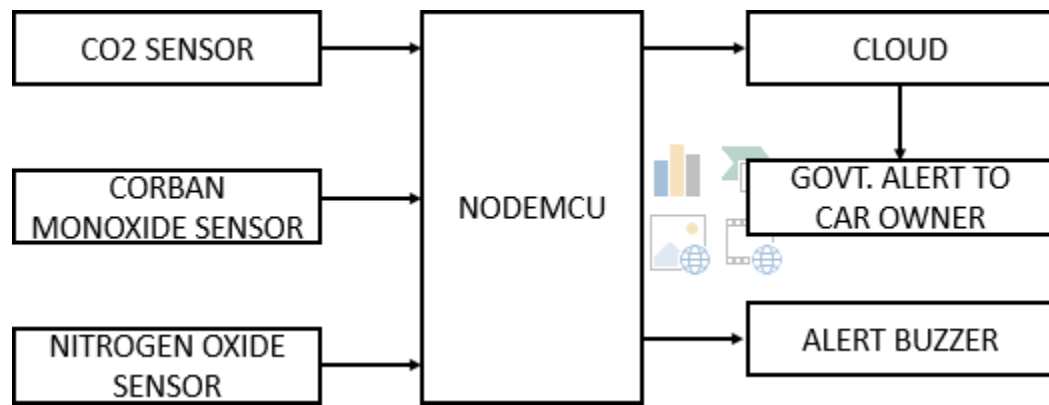
1.3 Existing system:

The microcontroller is programmed to do mainly three functions namely: Comparison, timer and triggering. When the smoke sensor output is more than the threshold value, the microcontroller triggers the timer circuit and an alarm is set on to inform the driver of the pollution in the vehicle and that it will come to the halt state. Semiconductor sensor MQ-2 is used to detect the smoke whose range is 300ppm to 10000ppm. CO Sensor having low conductivity in a clean environment is connected to a PIC microcontroller. When this sensor is exposed to the pollutants its conductivity increases generating a signal in the circuit which disables the motor. An automatic SMS is also generated by a gsm module which is connected in the circuit.

1.4 Proposed system:

- The given proposed system is continuously monitoring the vehicle pollution monitoring
- Carbon monoxide sensor, nitrogen oxides sensor are going to use in exhaust system in vehicle. It takes two inputs one from the smoke sensor and the other being the pre-defined threshold value specified by the government.
- If it reaches the high threshold value the vehicle location can be traced by GPS module and the location is sent to the government. The government may go to cancel the registration of the car and the message to the car owner.

1.5 Block diagram:



Hardware required:

- CO2 SENSOR
- NITROGEN OXIDE SENSOR
- NODEMCU
- CARBON MONOXIDE
- GPS

CHAPTER 2

2.1 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.

2.2 Overview of embedded systems

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

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

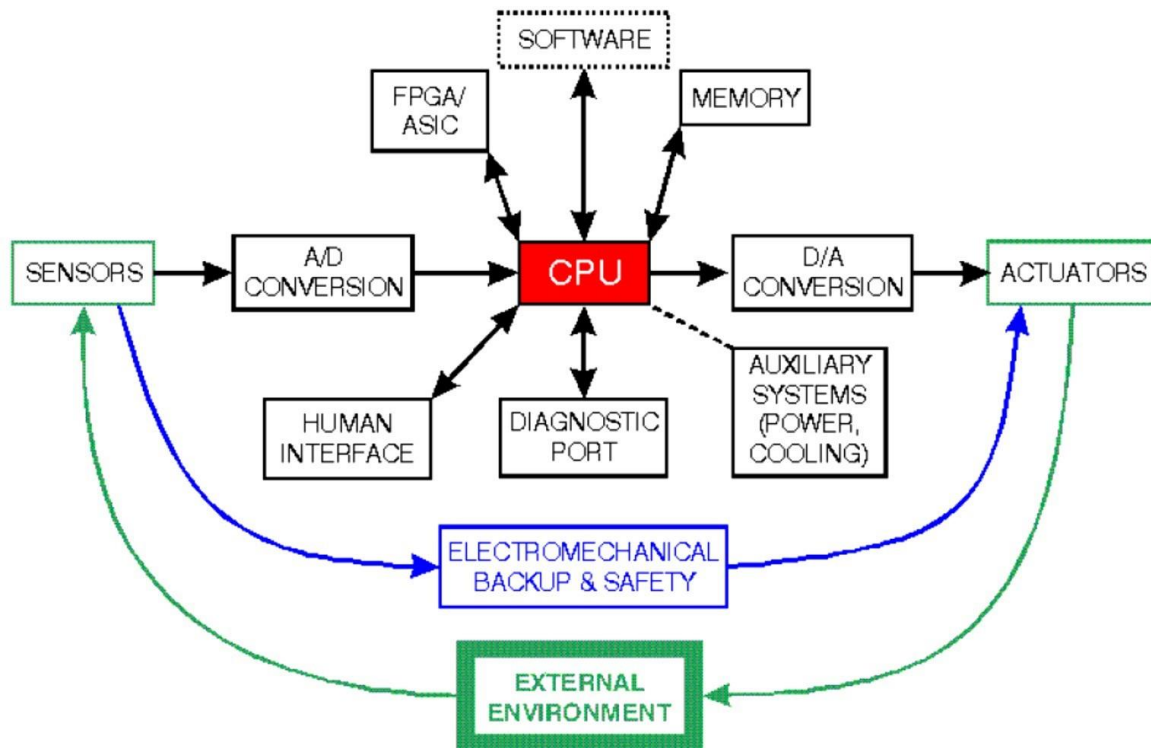


Fig 1.1 Block diagram of a typical embedded system

2.3 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.

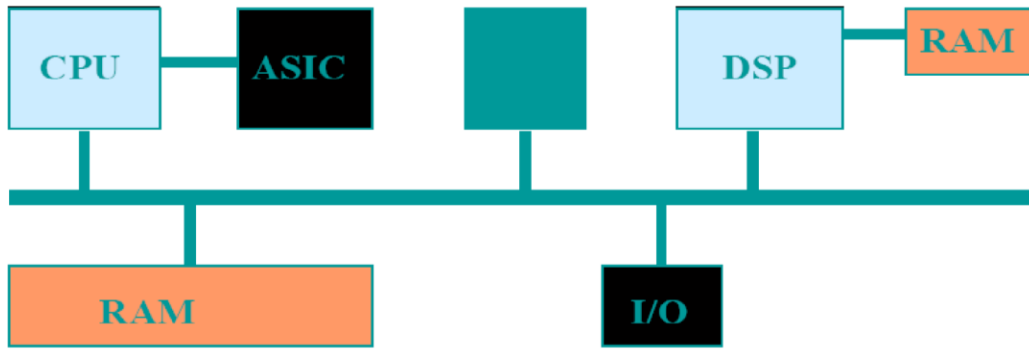


Fig 1.2 Embedded Systems having Heterogeneous

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.

2.4 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 some what, or the load connected to the output dc voltages changes.

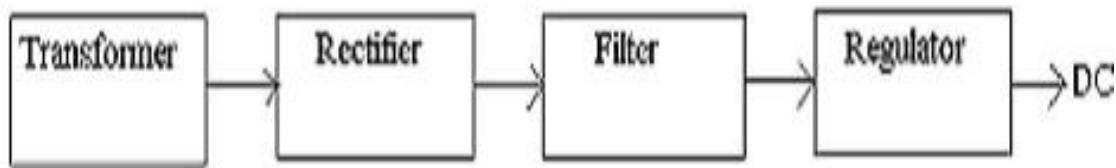


Fig 1.General Block Of Power Supply Unit

2.5 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.

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.

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.

2.6 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

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.

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

CHAPTER 3

3.1 NODE MCU

NodeMCU is an open source [IoT](#) platform. It includes [firmware](#) which runs on the [ESP8266 Wi-Fi SoC](#) from [Espressif Systems](#), and hardware which is based on the ESP-12 module. 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 and [SPIFFS](#)

NodeMCU was created shortly after the [ESP8266](#) came out. On December 30, 2013, [Espressif Systems](#) began production of the ESP8266. The ESP8266 is a Wi-Fi SoC integrated with a [Tensilica](#) Xtensa LX106 core, 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. 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. Later that month, Tuan PM ported [MQTT](#) client library from [Contiki](#) to the ESP8266 SoC platform, 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 to NodeMCU project, enabling NodeMCU to easily drive LCD, Screen, OLED, even VGA displays.

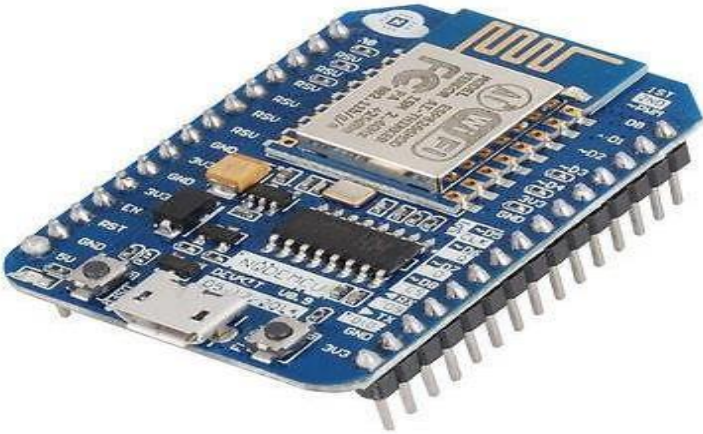
In summer 2015 the creators abandoned the firmware project and a group of independent contributors took over. By summer 2016 the NodeMCU included more than 40 different modules. Due to resource constraints users need to select the modules relevant for their project and build a firmware tailored to their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

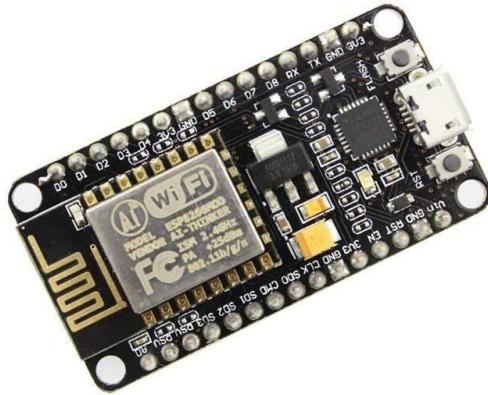


NodeMCU Development Board/kit v0.9 (Version1)

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](#)

3.2 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.

CHAPTER 4

4.1 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.

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 . 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 . 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 .

4.2 Motivation

IoT-based system is in charge of providing knowledge from an environment to an 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 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.

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-Architecture² (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]

4.3 Layers in IOT:

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.

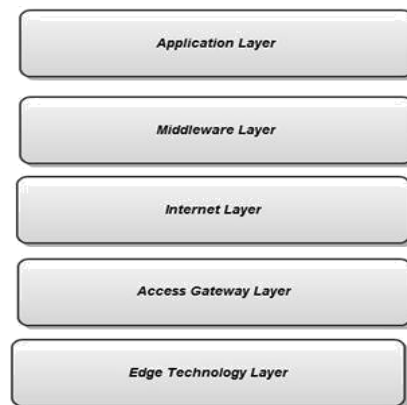


Figure 2-1 Layered architecture of the Internet of Things (IoT).

4.4 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.

4.5 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.

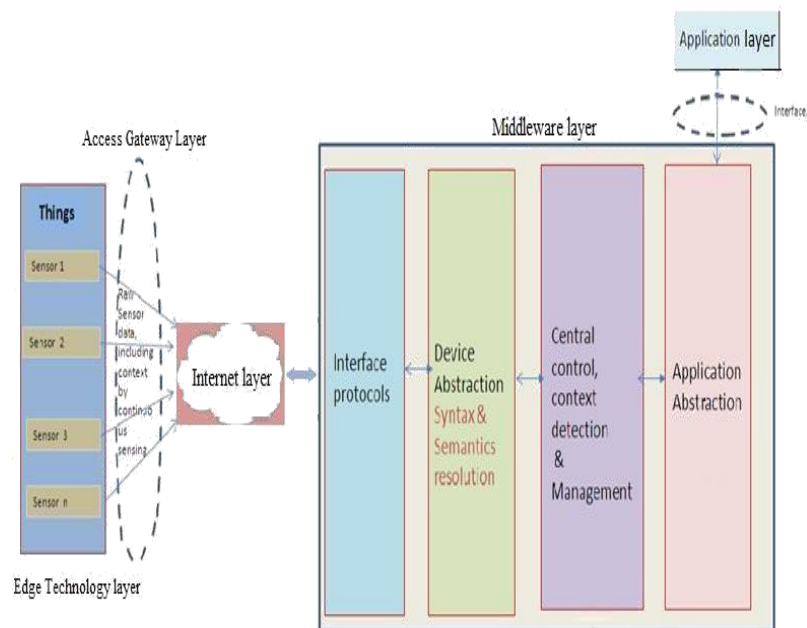


Figure 2.2 Functional components of a middleware for IoT-based systems

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.

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

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.

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
```

CHAPTER 5

5.1 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 .

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-anywhere 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.

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.

5.2 CO2 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. **CO2 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.

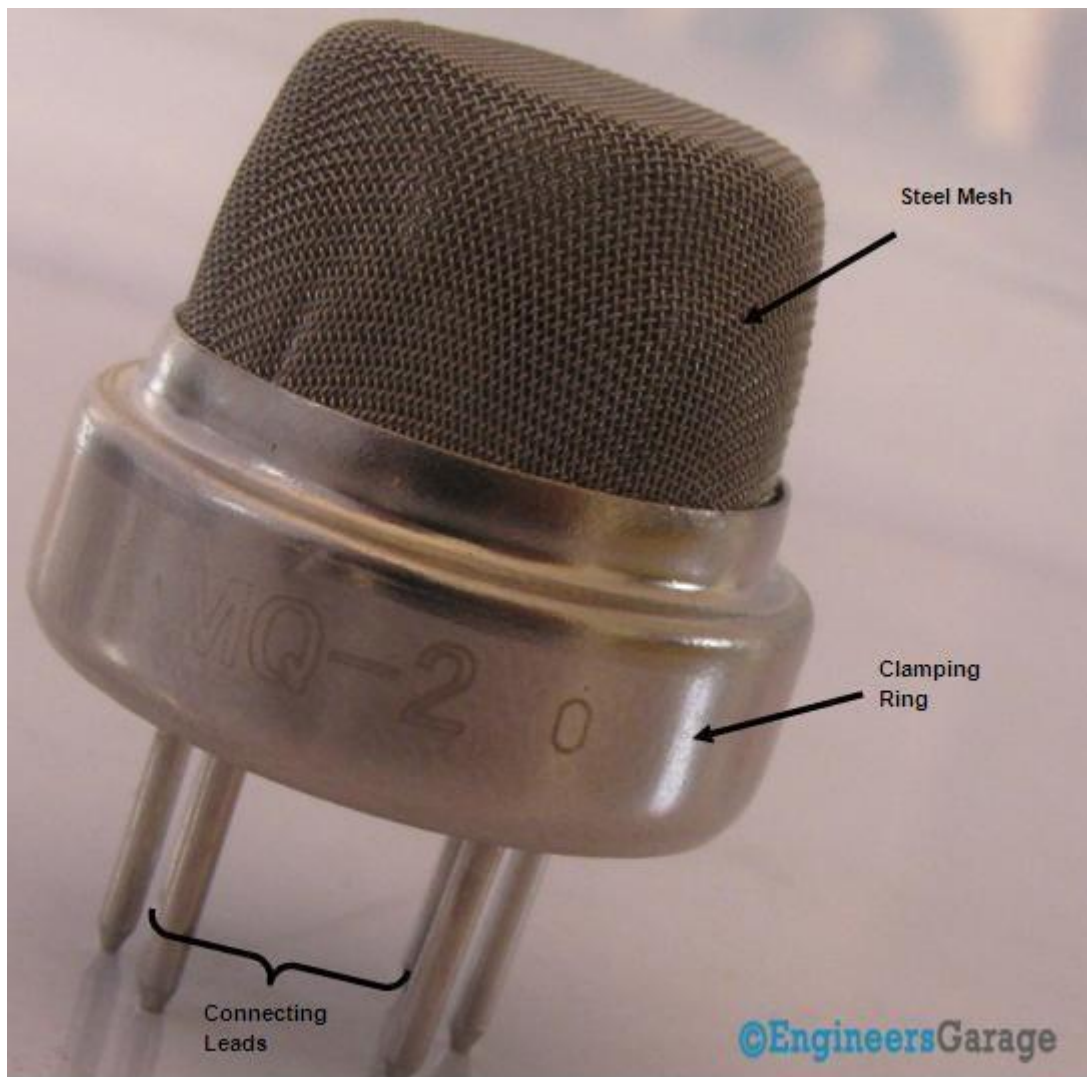


Image shows externals of a standard gas sensor module: a steel mesh, copper clamping ring and connecting leads. The top part is a stainless steel mesh which takes care of the following:

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

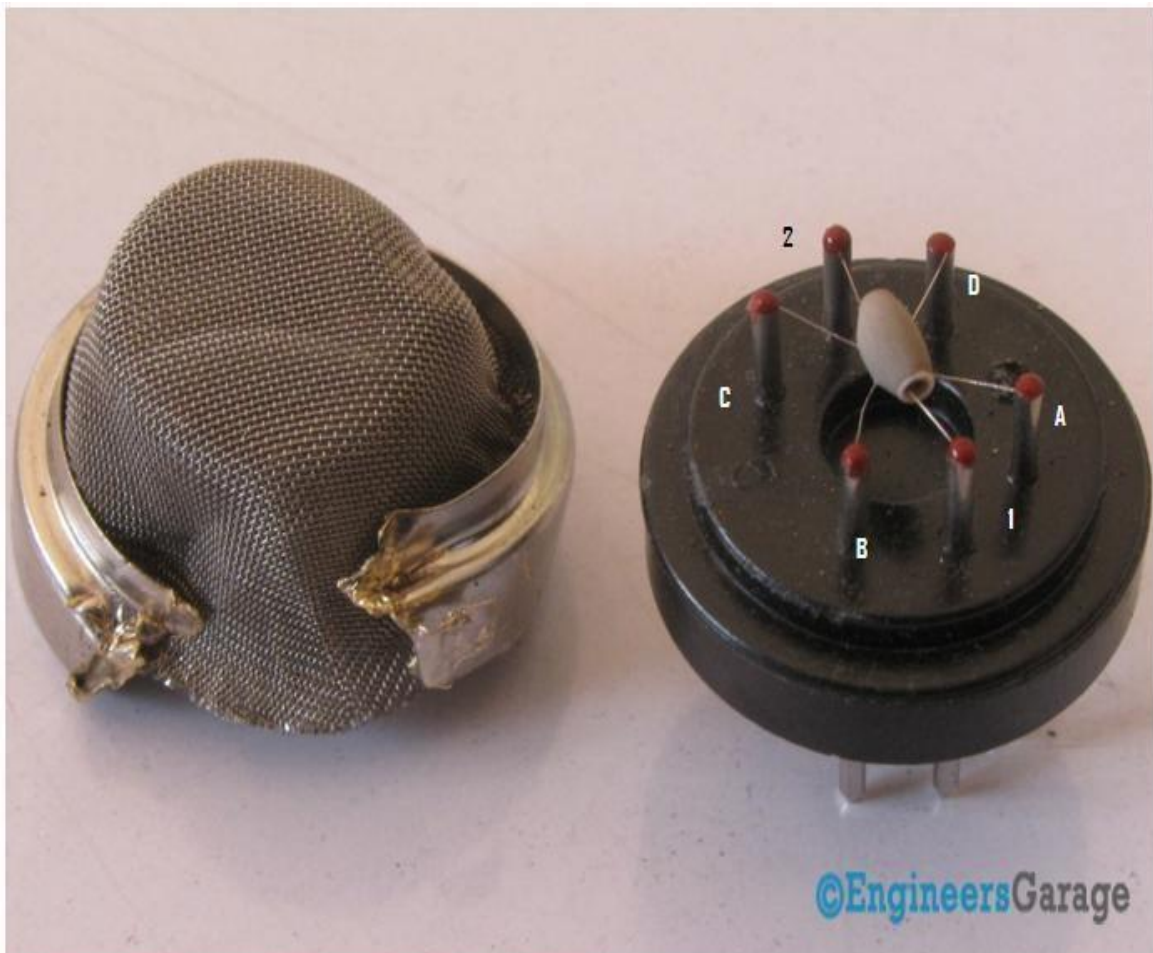
- ✓ Protecting the insides of the sensor.
- ✓ Exhibits an anti-explosion network that keeps the sensor module intact at high temperatures and gas pressures.

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

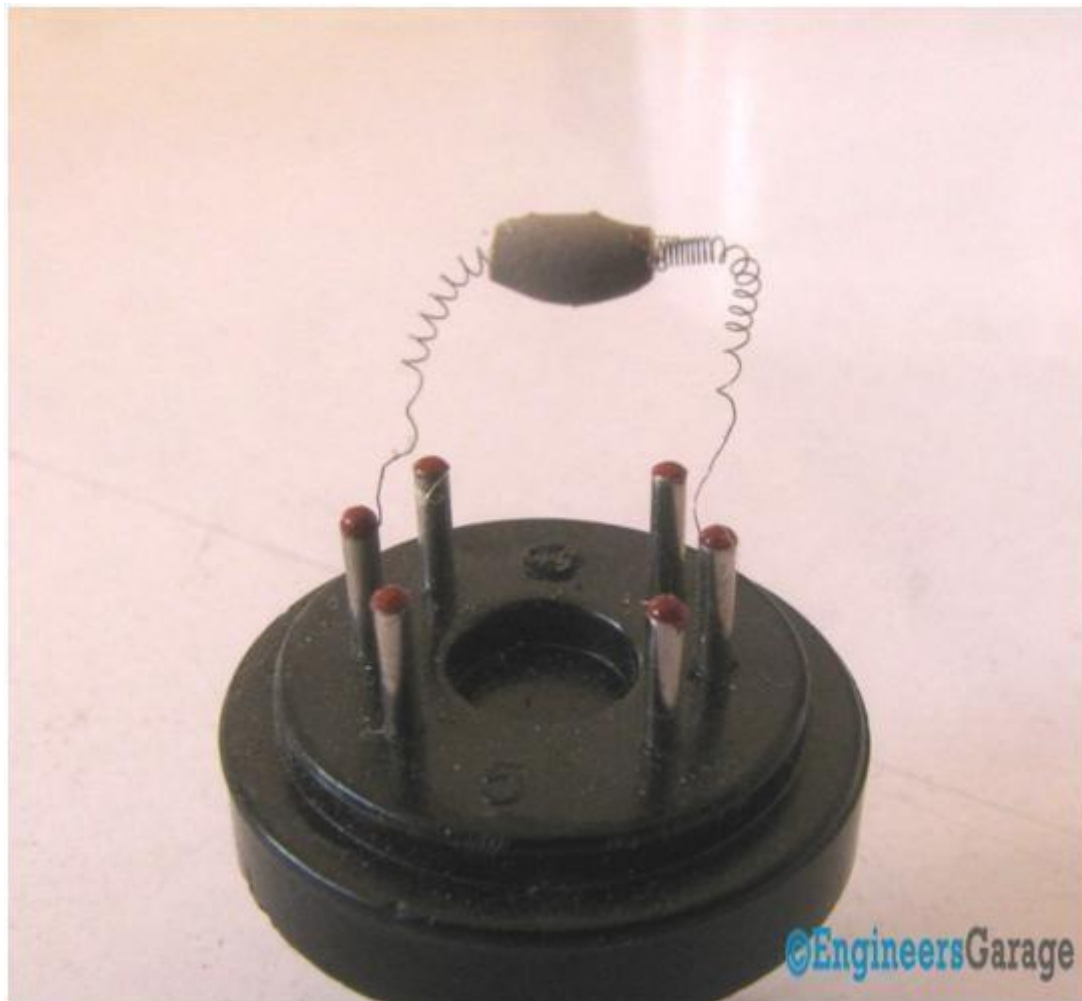
base.



Image shows the hollow sensing element which is made up from Aluminium 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 shows coiled part of the wire which is placed on the inside of the hollow ceramic.

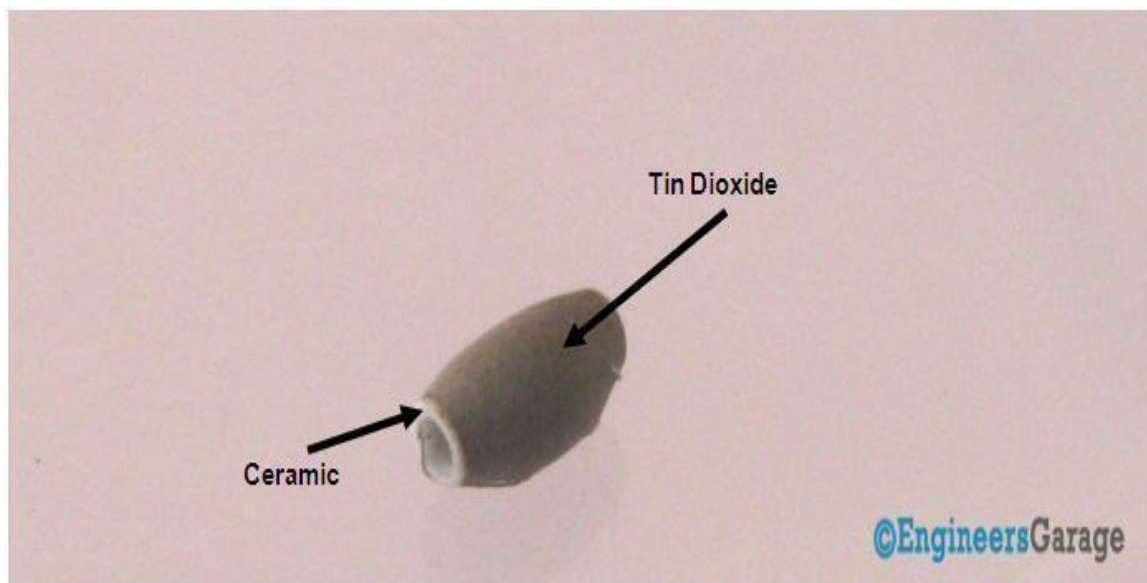


Image shows the ceramic with tin dioxide on the top coating that has good adsorbing property. Any gas to be

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.

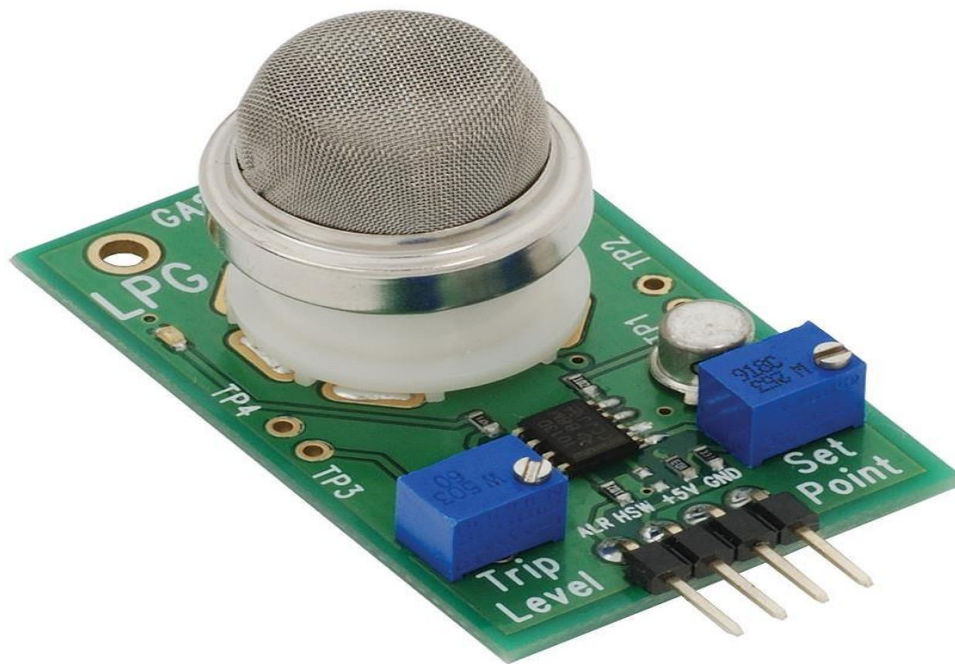
Adsorbed molecules change the resistance of the tin dioxide layer. This changes the current flowing through the sensing element and is conveyed through the output leads to the unit that controls the **working of the gas sensor**.

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.

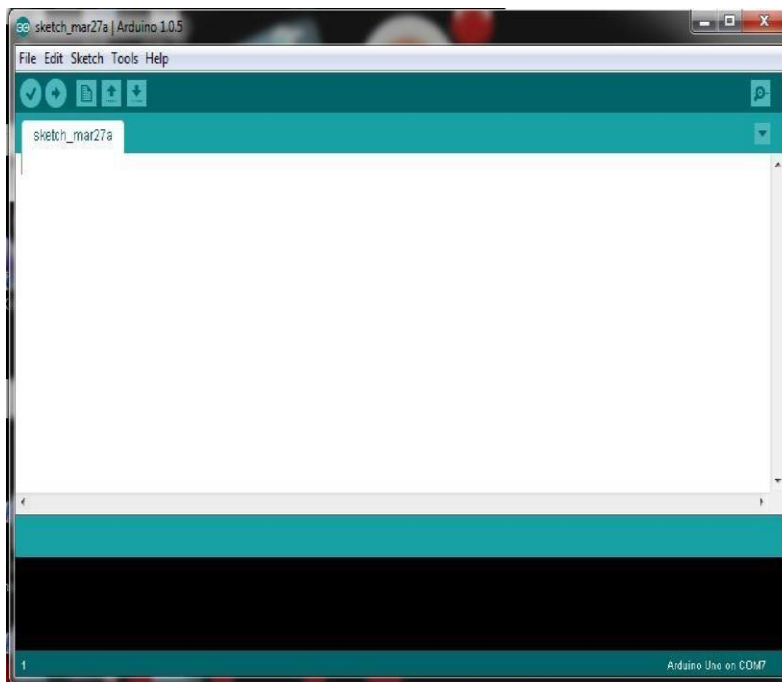
5.3 Software required:

Arduino ide

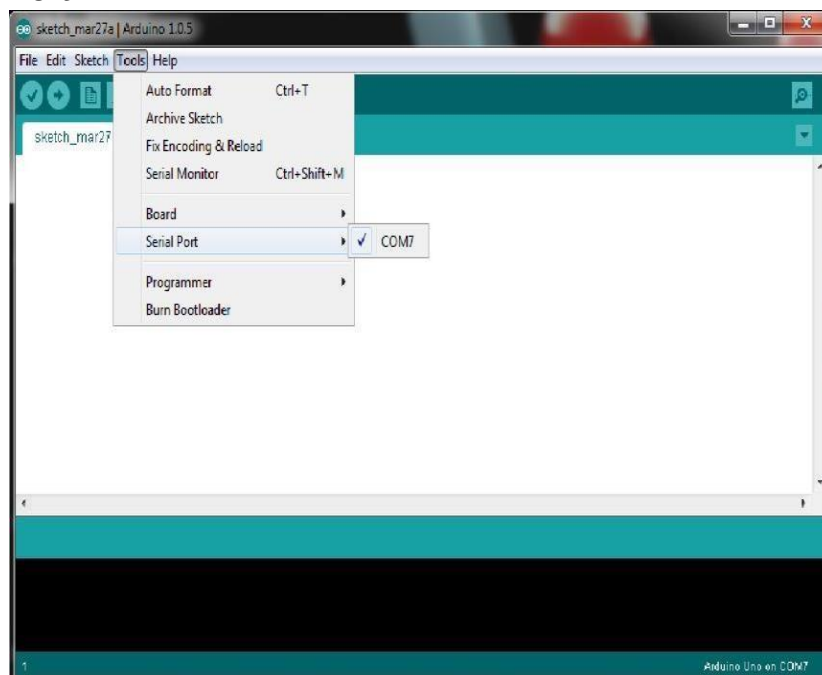
Arduino C

ARDUINO SOFTWARE:

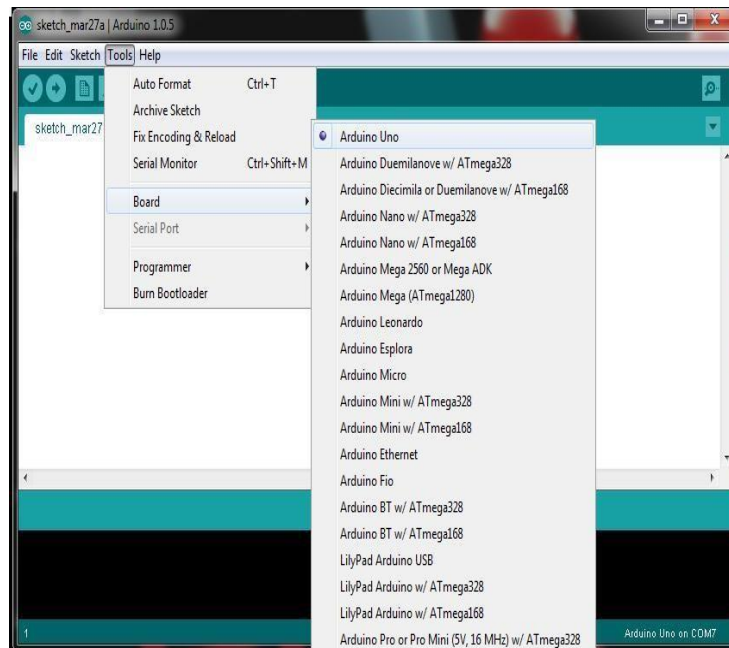
1. Open Arduino IDE as shown below
Open Arduino IDE



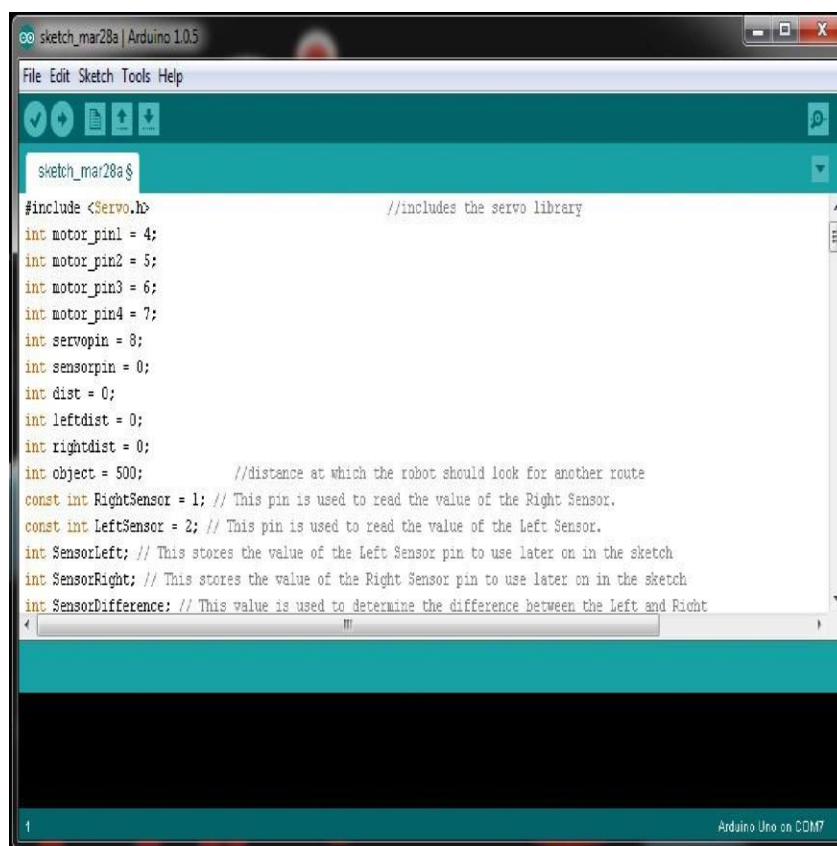
1. Select the COM Port from tool Select the COM Port



2. Select the required Arduino board from Tools as shown below
Select the required Arduino board



3. Write the sketch in Arduino IDE



1. Compile and upload the Sketch to Arduino board upload the Sketch to Arduino board



5.4 UBIDOTS EXPLORER:

Introduction

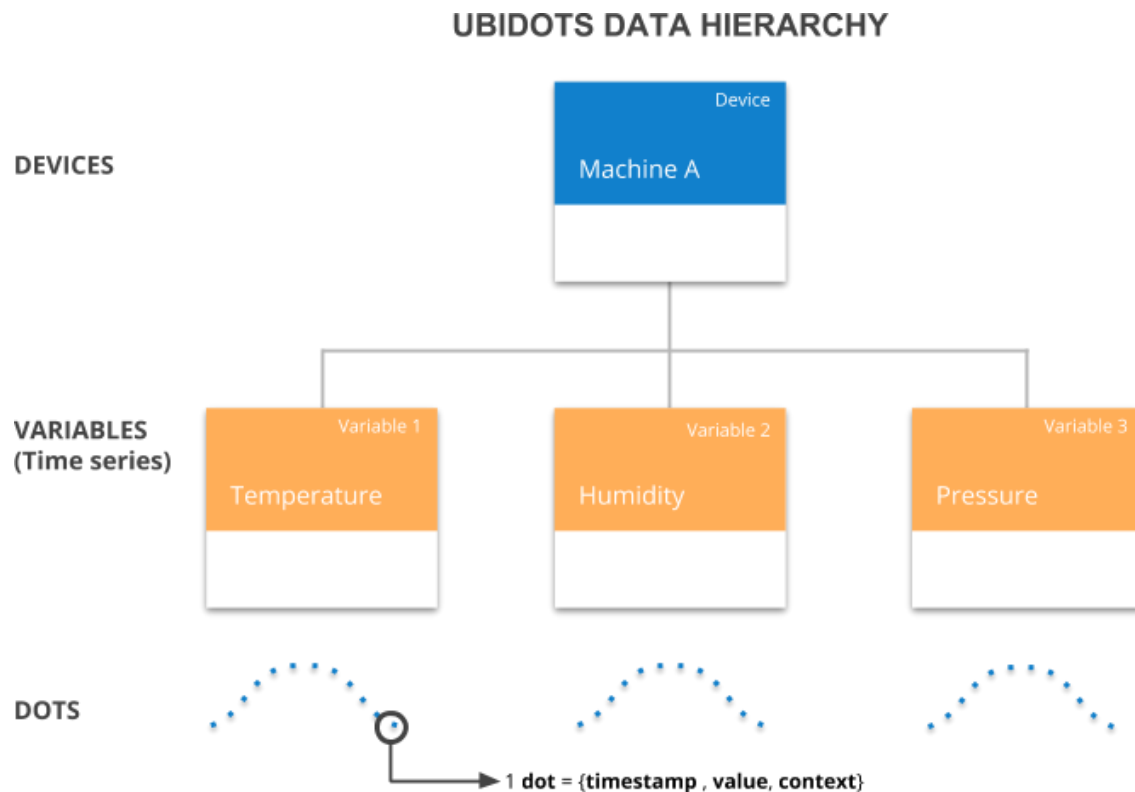
Welcome to the Ubidots REST API hardware docs. Here you would reference how to send or retrieve data from your hardware devices using our supported communication protocols: **HTTP**, **MQTT** and **TCP/UDP**.

The purpose of this section is to help you understand what happens in the

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

Item	Description	Mandatory
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:

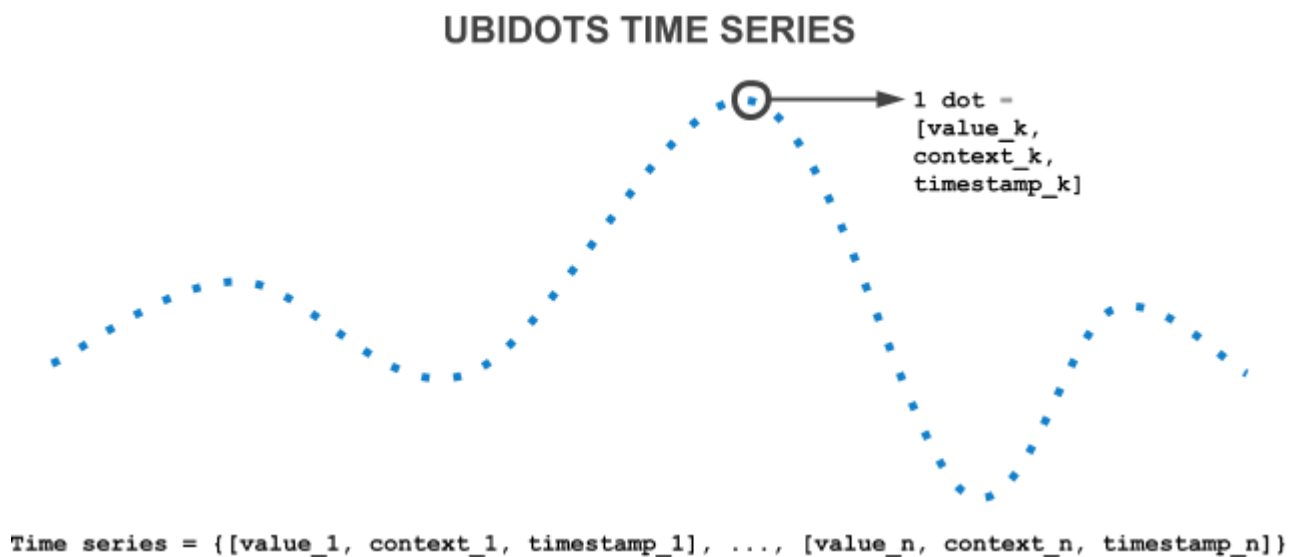
```
"context" : {"status" : "on", "weather" : "sunny"}
```

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
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

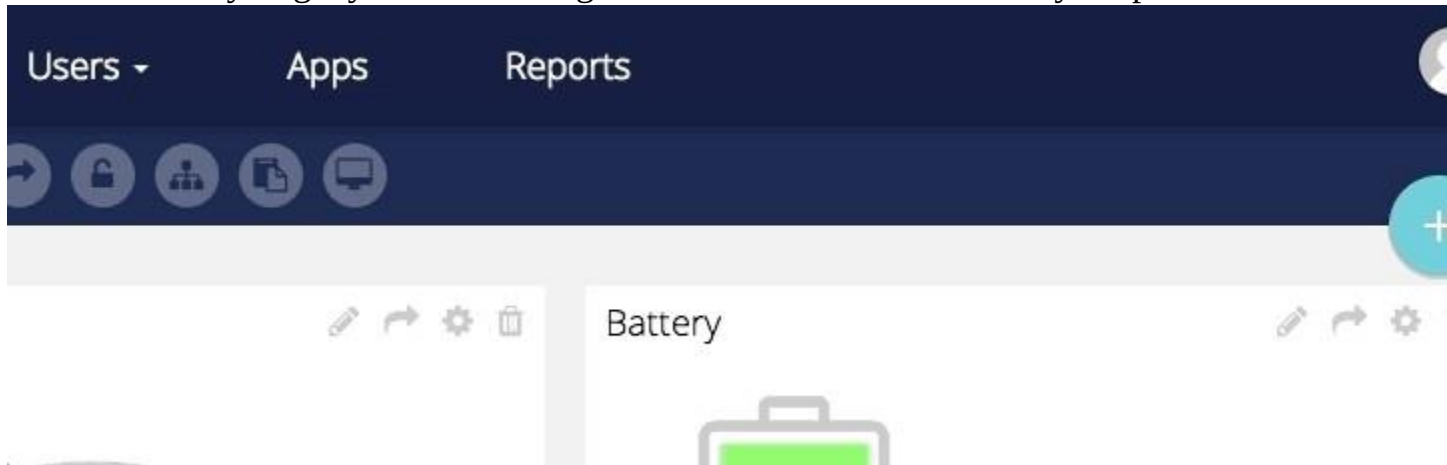
Ubidots also supports fixed IPs address for data ingestion, but we highly advise to our users to implement their firmware routines using the dns endpoints instead of fixed IPs, as these last ones may change in the future.

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° Variable s	JSON structure
One	{"variable-label": VALUE}

N° Variable s	JSON structure
	<pre>{ "key": VALUE }}, ... , "variable-label- n": { "value": VALUE, "timestamp": TIMESTAMP, "context": { "key": VALUE } }</pre>

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](#).

Examples

- Single dot: {"temperature": 27}
- Multiple dots: {"temperature": 27, "humidity": 55, ... , "pressure": 78}

Including additional keys like *context* and *timestamp*

- Single dot: {"position": { "value" : 1, "timestamp": 1514808000000, "context": { "lat": -6.2, "lng": 75.4, "name" : "John" } }}
- Multiple dots: {"temperature": { "value": 27, "timestamp": 1514808000000 }, "humidity": 55, "pressure": { "value": 78, "context": { "name" : "John" } }}

POST

https://industrial.api.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}

POST

https://things.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}

Send a single dot (Educational users)

```
'{"temperature": 27}' https://things
```



Send a single dot (Industrial users)

```
Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d  
'{"temperature": 27}' https://industrial.api
```



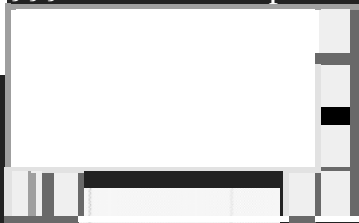
Send a dot with coordinates

```
'{"position": {"value" : 1, "context": {"lat": -6.2, "lng": 75.4}}}' https://industr
```



Send multiple dots

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d  
'{"temperature": {"value": 27, "timestamp": 1514808000000}, "humidity": 55,  
"pressure": {"value": 78, "context": {"name": "John"}},  
}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device
```



[Send data to a variable](#) You can also send one or multiple dots to a single variable

Request structure:

```
POST /api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values 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/1.1 201 Created<CR><LN>
Server: nginx<CR><LN>
Date: Tue, 04 Sep 2018 22:50:55 GMT<CR><LN>
Content-Type:
application/json<CR><LN>    Transfer-
Encoding: chunked<CR><LN>    Vary:
Accept, Cookie<CR><LN>
Location:
http://industrial.api.ubidots.com/api/v1.6/values/{VARIABLE_ID}<CR><LN>
Allow: GET, POST, HEAD, OPTIONS<CR><LN><CR><LN>
{PAYLOAD_LENGTH_IN_HEXADECIMAL}<CR><LN>
{"url": "http://things.ubidots.com/api/v1.6/values/{VARIABLE_ID}", "value":
27.0, "timestamp": 1536101455822, "context": {}, "created_at": "2018-09-
04T22:50:55.822035Z"}<CR><LN>
0<CR><LN>
```

In the *path* specified above, replace the {DEVICE_LABEL} and {VARIABLE_LABEL} keys with the unique labels of the Ubidots device and variable you wish to send data to. If the specified device and/or variable do not exist, they will be automatically created.

Examples

- Single dot: {"value":27}
- Single dot with context and timestamp: {"value":27, "timestamp": 1514808000000, "context":{"lat":37.773, "lng":-122.431}}
- Multiple dots (must include the timestamp of each value): [{"value": 27, "timestamp":1514808000000}, {"value": 12, "timestamp":1514808900000}]

Send a single dot (Educational users)

POST

`https://industrial.api.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values`

POST

`https://things.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values`

Send a single dot (Industrial users)

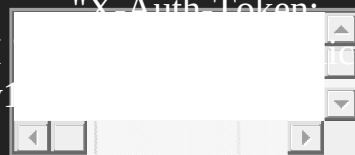
```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"value": 27}' https://industrial.api.ubidots.com
```



`/api/v1.6/devices/my-new-device/my-variable/values`

Send a single dot (Educational users)

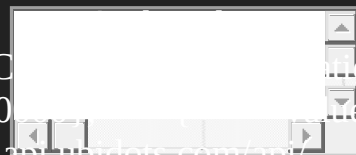
```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"value": 27}' https://things.ubidots.com/api/v1
```



`.6/devices/my-new-device/my-variable/values`

Send multiple dots

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '[{"value": 27, "timestamp": 1514808000}, {"value": 12, "timestamp": 1514808900000}]' https://industrial.api.ubidots.com/api/
```



Retrieve data

The Ubidots REST API allows you to retrieve data from the platform data base. A typical application for data retrieval is actuators like motors or lights control.

Retrieve mutiple values from a variable

Request structure:

```
GET /api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values
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><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>
{"count": true, "previous": null, "results": [{PAYLOAD_WITH_VALUES}], "next": {UR
L_WITH_ADDITIONAL_VALUES}}<CR><LN>
0<CR><LN>
```

In the *path* specified above, replace the `{DEVICELABEL}` and `{VARIABLE_LABEL}` keys with the unique labels of the Ubidots device and variable you desired to retrieve data from. This *_path* returns by default the last 50 values stored of your variable. You can manage the number of dots to retrieve using the **page_size** parameter in your path:

```
/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values/?page_size=2
```

The path above retrieves the last 2 dots of your variable.

Ubidots API also allows you to filter the values to be retrieved by date range, number of dots, and more. If you desire to go deep into this kind of requests, please

GET

```
https://industrial.api.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values
```

```
https://things.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/values
```

Get the last two values from a variable (Educational users)

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" https://things.ubidots.com/api/v1.6/devices/my-de
```

```
vice/my-variable/values/?page_size=2
```

Get the last two values from a variable (Industrial users)

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" https://industrial.api.ubidots.com/api/v1.6/devices/
```

```
my-device/my-variable/values/?page_size=2
```

[Retrieve the last value from a variable](#)

A typical control application in general only needs to ask for the last value of a variable, i.e, if you push a virtual switch at Ubidots, you just need to know what is the last value of the variable related with that switch to take actions on any actuators at your side.

Request structure:

```
GET /api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/lv
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><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-
```

```
Allow: GET, POST, HEAD, OPTIONS<CR><LN><CR><LN>
{PAYLOAD_LENGTH_IN_HEXADECIMAL}<CR><LN>
{LAST_VALUE}<CR>
<LN> 0<CR><LN>
```

In the *path* specified above, replace the {DEVICE_LABEL} and {VARIABLE_LABEL} keys with the unique labels of the Ubidots device and variable you desired to retrieve data from.

```
GET
https://industrial.api.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/lv
GET
https://things.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}/{VARIABLE_LABEL}/lv
```

Get the last value from a variable (Educational users)

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB"
-H "Content-Type: application/json"
https://things.ubidots.com/api/v1.6/devices/my-device/my-variable/lv
```

Get the last value from a variable (Industrial users)

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB"
-H "Content-Type: application/json"
https://industrial.api.ubidots.com/api/v1.6/devices/my-device/my-variable/lv
```

HTTP Examples

In this section, you will find different examples that show you how to handle data with Ubidots to start monitoring and controlling your IoT Devices.

At the right side, you can find each example provided in different programming languages. Please, refer to your preferred programming language by clicking the tabs located on the right upper side of the page.

[Sending a geopoint](#)

Ubidots provides multiples ways to start tracking your IoT devices, so here we are going to show you how to send coordinates to Ubidots in order to start with a GPS IoT Solution.

To learn more about the Ubidots Maps Features, please check [this guide](#).

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"location":{"value": 1, "context":{"lat":37.773, "lng":-122.431}}}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-devi
```

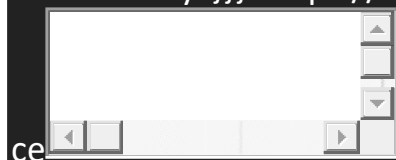


[Sending a string value](#)

The String values provide additional information to the dot which is being sent. As a basic example here, let's suppose that you are deploying a Weather Station which provides you the Weather Status as a String value. At this point, the Weather Status ought to be sent in the context of the dot.

To learn more of how to display the context of a dot in the Ubidots Dashboard, please check [this guide](#).

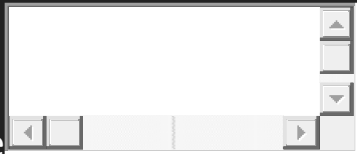
```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"weather":{"value": 1, "context":{"weather-status": "sunny"}}}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-devi
```



Sending values in bulk from a datalogger

In case you desire to send information to Ubidots from a data logger which is in charge of taking samples for long periods of time, you can simply do it by managing the values registered plus the time when the value was registered.

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"my-sensor":[{"value": 27, "timestamp":1514808000000}, {"value": 30, "timestamp":1514808900000}, {"value": 31, "timestamp":1514809800000}, {"value": 29, "timestamp":1514810700000}, {"value": 27, "timestamp":1514768400000}]}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device
```



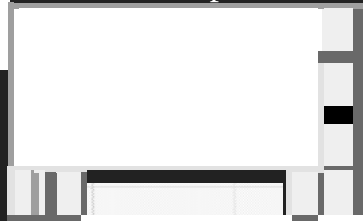
Retrieve the last value for control

This is typically used for remote control applications. Your devices would retrieve the last value of a variable, and then act based on that value. It is also used to remotely manage your devices, like for example changing a sampling rate or performing a remote reboot.

To learn more about the control features that Ubidots offer, please check [this guide](#).

Example Request

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" "https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device/control/lv"
```



Response codes The Ubidots REST API uses the following response codes when you make

Code	Meaning
200	Ok -- Successful request.
201	Created -- Successful request + an item (device or variable) was created.
202	Accepted -- The request has been accepted for processing, but the processing has not been completed.
204	One of the fields is incorrect and the request is not saved -- Please verify it's a valid JSON string and that the fields are the ones expected by the endpoint (string, object or float).
400	Bad Request -- Error due to an invalid body in your request. Please verify it's a valid JSON string and that the fields are the ones expected by the endpoint (string, object or float).
401	Invalid API key -- Please verify your API Key.
402	Payment required -- Please verify your balance.
403	Forbidden -- This token is not valid. Please verify your token.
404	Not Found -- We couldn't find the URL you're trying to access. This might be due to a wrong device label or ID, a wrong variable label or ID, or simply a typo in the URL of the request.
405	Method Not Allowed -- This API endpoint does not accept the method used. Check our API docs to see the allowed methods.
415	Unsupported media type -- The payload is in a format not supported by this method on the target resource.

Code	Meaning
420	You have exceeded your API limits -- To upgrade your API limits contacts the Ubidots team
423	Device does not receive data because it is disabled
429	Too many requests -- Many requests sent in a given amount of time ("rate limiting")
50*	Internal Error -- We're having issues with our servers. Please check our status page .

MQTT

[MQTT](#) is an "**Internet of Things**" connectivity protocol. It was designed as an extremely lightweight **publish/subscribe** messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium"

(Source: [MQTT.org](#)).

MQTT is specially useful to push data from the cloud to your devices. Imagine a cloud-controlled device to open/close a door remotely. In the case of HTTP, the device would have to continuously make GET requests to the Ubidots server to see if there's a change in a variable, say "*Door Control Variable*", and then take an action depending on the last reading. This takes a lot of requests and it's not entirely a real-time interaction since it depends of the polling frequency. With MQTT, the device can "*listen*" to the cloud and only get notified when there's a change in the variable. This way, the connection between the device and the cloud is left open but data only travels when is necessary, saving battery, network bandwidth and improving the real-time experience.

In this section you will find the documentation needed to handle your devices with Ubidots over MQTT.

For a detailed explation of the **Ubidots MQTT API Reference** [click here](#)

Quality of Service

In MQTT, the quality of service, QoS for short, guarantees the delivery of a specific message. There are three levels of QoS:

- 0, At most once: This level does not guarantee the message delivery, it would be labeled as *best-effort* server QoS.
- 1, At least once: This level guarantees that the message is delivered at least one time to the receiver. The receiver answers with a puback packet, if this puback packet is not received the sender sends again the message.
- 2, Exactly once: Guarantees that the message is received only once by the receptor. It is the slowest packet interchange QoS as it requires two request/response flows.

Ubidots supports QoS up to 1 (at most once).

Broker URLs

API access can be over plain MQTT or secure MQTT with TLS, using the following endpoints based on your Ubidots Account type:

Non-TLS

Ubidots Account	Endpoint	Port
Educational	things.ubidots.com	1883
Industrial	industrial.api.ubidots.com	1883

TLS

Ubidots Account	Endpoint	Port
Educational	things.ubidots.com	8883
Industrial	industrial.api.ubidots.com	8883

You can download the Ubidots PEM certificate for TLS [here](#)

Security Note: We strongly advise to use MQTT with TLS to make sure your data travels encrypted, avoiding the exposure of your API token and/or sensor data to third parties.

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

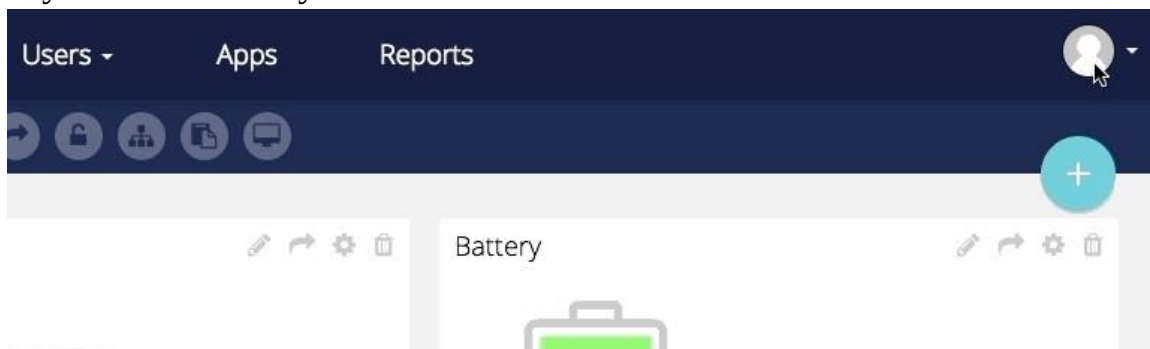
Fixed IPs

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	1883/8883
Industrial	169.55.61.243	2607:f0d0:3:b::2	1883/8883

MQTT Authentication

To interact with the Ubidots MQTT broker you'll need a *token*. A *token* is an unique key that authorizes your device to interact with Ubidots API.



To establish the connection with the MQTT broker you need to assign the **username** and **password** following the table below:

MQTT Credentials	Parameter
Username	Ubidots <i>token</i>
Password	any character - not required -

Send data (Publish)

To send data, set the unique [Device label](#) in the topic path.

Ubidots accepts MQTT data *JavaScript Object Notation* or *JSON*. *JSON* 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"}}}
```

The keys *context* and *timestamp* are optional.

NOTE 1:: 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. You may change the default location variable from Ubidots web user interface.

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](#).

If you wish to send **more than one value**, then you need to separate them using commas. Below you can reference a typical structure for MQTT payloads:

Variables	JSON structure
One	<pre>{"VARIABLE-LABEL": {"value":VALUE, "timestamp": TIMESTAMP, "context":{"KEY":VALUE}}}</pre>
Multiple	<pre>{"VARIABLE-LABEL_1": {"value":VALUE, "context":{"KEY":VALUE}}}, ... ,</pre>

Variables	JSON structure
	"VARIABLE-LABEL_n": {"value":VALUE, "context":{"KEY": VALUE}}

keys *context* and *timestamp* are optional.

Examples

Note: All the *bash* examples from this section makes use of the Eclipse Mosquitto library, if you wish to test them please install it using the [official instructions](#)

- Single dot: {"temperature": 27}
- Multiple dots: {"temperature": 27, "humidity": 55, ... ,"pressure": 78}

Including additional keys like *context* and *timestamp*

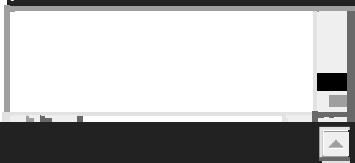
- Single dot: {"position": {"value" : 1, "timestamp": 1514808000000, "context":{"lat":-6.2, "lng":75.4, "name" : "John"}}
- Multiple dots:{"temperature": {"value": 27, "timestamp": 1514808000000}, "humidity": 55, "pressure": {"value": 78, "context":{"name" : "John"}}

PUB

/v1.6/devices/{DEVICE_LABEL}/

Send a single dot (Educational users)

```
mosquitto_pub -h "things.api.ubidots.com" -t "/v1.6/devices/weather-station" -u "p4uuT2O25AJwv7ncADLoVhQflmwRQW" -p 1883 -q 1 -m '{"temperature": 27}'
```





Send a dot with coordinates

```
mosquitto_pub -h "industrial.api.ubidots.com" -t "/v1.6/devices/weather-station"
-u "BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB" -p 1883 -q 1 -m '{"position":
{"val
ue" : 1, "timestamp": 1514808000000, "context":{"lat":-6.2,
"lng":75.4}}}'
```

Send multiple dots

```
78, "context":{"name" : "John"}}}
```

Retrieve data (Subscribe)

Similar to the GET function in HTTP, subscribe is the method to obtain values with a huge difference: you do not have to be continuously ask the server for each value in your custom script. For example, if a variable's value changes, Ubidots will automatically update you (the user) of any changes. Thus saving data requests and processing time for your device and overall project functionality and expenses. Because of this, MQTT is a communication protocol advised to implement firmware routines to control actuators.

[Retrieve the last value of a variable](#)

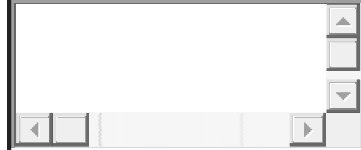
Replace the {DEVICE_LABEL} and {VARIABLE_LABEL} with the unique label of the device and the variable to subscribe for retrieving values update. If the specified Device or Variable do not exist, you will not able to establish the subscription with it.

SUB

```
/v1.6/devices/{DEVICE_LABEL}/{LABEL_VARIABLE}/lv
```

Subscribe to a variable (Educational users)

```
mosquitto_sub -h "things.ubidots.com" -t "/v1.6/devices/weather-station/temperature/lv" -u "p4uuT2O25AJwv7ncADLoVhQflmwRQW" -p 1883 -q 1
```



Subscribe to a variable (Industrial users)

```
mosquitto_sub -h "industrial.api.ubidots.com" -t "/v1.6/devices/weather-station/temperature/lv" -u "BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -p 1883 -q 1
```



MQTT Examples

In this section, you will find different examples that show you how use MQTT to implement several IoT applications.

At the right side, you can find each example provided in different programming languages. Please, refer to your preferred programming language by clicking the tabs located on the right upper side of the page.

GPS Tracker

Ubidots provides multiples ways to start tracking your IoT devices, so here we are going to show you how to send coordinates to Ubidots in order to start with a GPS IoT Solution.

To learn more about the Ubidots Maps Features, please check [this guide](#).

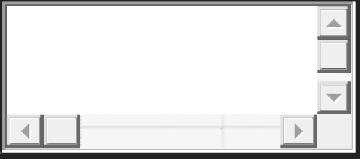
```
mosquitto_pub -p 1883 -h industrial.api.ubidots.com -t "/v1.6/devices/my-new-device" -u 'BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB' -m '{"location":{"value": 1, "context":{"lat":37.773, "lng":-122.431}}}' -d -q 1
```

Weather description

The String values provide additional information to the dot which is being sent. As a basic example here, let's suppose that you are deploying a Weather Station which provides you the Weather Status as a String value. At this point, the Weather Status ought to be sent in the context of the dot.

To learn more of how to display the context of a dot in the Ubidots Dashboard, please check [this guide](#).

```
mosquitto_pub -p 1883 -h industrial.api.ubidots.com -t "/v1.6/devices/my-new-device" -u 'BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB' -m '{"weather":{"value": 1, "context":{"weather-status": "sunny"}}}' -d -q 1
```

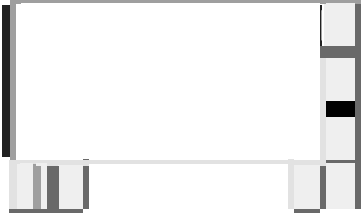


Retrieve the last value for control

This is the perfect solution for you in case you desire to control remotely the garage doors, home lights, alarm system, or any asset you desire.

To learn more about the control features that Ubidots offer, please check [this guide](#).
Example Request

```
mosquitto_sub -p 1883 -h industrial.api.ubidots.com -t "/v1.6/devices/my-new-device/control/lv" -u 'BBFF-Rfcgaxns6HlVb155WA0RhSY85xNDmB' -d -q 1
```



TCP / UDP

The Transmission Control Protocol, TCP, is one of the most popular data transport protocols, so much it is supported by default in most hardware devices. Its main advantage comes in the packet size, usually lower than packets sent using HTTP

(which in fact, uses TCP as transport layer). TCP is "connection oriented", which

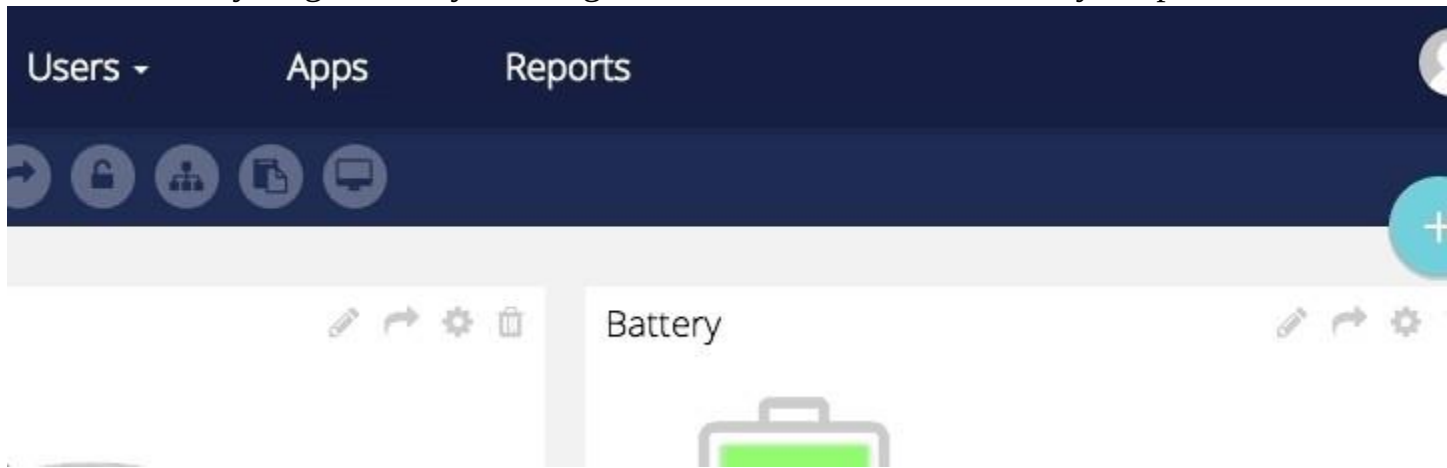
means it implements confirmation methods to guarantee that data is received by the destinationary.

The User Datagram Protocol, UDP, is also a very popular protocol that, unlike TCP, it is "connectionless". This means it does not implements any confirmation method to guarantee that data is received by the destinationary. This can be an advantage in applications where simplicity and speed are more important than reliability.

TCP and UDP are similar in the sense that they share the structure of **endpoint + port**.

TCP Authentication

To send data to Ubidots using TCP or UDP broker you need your **Ubidots TOKEN**. The easiest way to get it is by clicking on "**API Credentials**" under your profile



TCP URLs

API access can be over plain TCP or secure TCP with TLS, using the following endpoints based on your Ubidots Account type:

Non-TLS

Ubidots Account	Endpoint	Port
Educational	things.ubidots.com	9012
Industrial	industrial.api.ubidots.com	9012

TLS

Ubidots Account	Endpoint	Port
Educational	things.ubidots.com	Not Available
Industrial	industrial.api.ubidots.com	9812

You can download the Ubidots PEM certificate for TLS [here](#)

Security Note: We strongly advise to use TCP with TLS to make sure your data travels encrypted, avoiding the exposure of your API token and/or sensor data to third parties.

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

Send Data

Structure

Ubidots created a pre-defined structure to understand the content of your TCP/UDP packets. If you wish to implement a custom structure, please get in touch with our support channel.

Single device:

```
{USER_AGENT}|POST|{TOKEN}|DEV_LABEL(:DEV_NAME)(@GLOBAL_TIMESTAMP  
)=>[VAR_LABEL:VALUE([$CONTEXT_KEY=CONTEXT_VAL]...)](@VARIABLE_TIMESTAMP  
AMP)|end
```

Multiple devices:

```
{USER_AGENT}|POST|{TOKEN}|DEV_LABEL(:DEV_NAME)(@GLOBAL_TIMESTAMP  
)=>[VAR_LABEL:VALUE([$CONTEXT_KEY=CONTEXT_VAL]...)](@VARIABLE_TIMESTAMP  
AMP); ...
```

```
;DEV_LABEL(:DEV_NAME)(@GLOBAL_TIMESTAMP)=>[VAR_LABEL:VA  
LUE  
([$CONTEXT_KEY=CONTEXT_VAL...)](@VARIABLE_TIMESTAMP)|end
```

Where:

- {USER_AGENT}: *Mandatory*. Contains a characteristic string that allows to identify the application type, operating system, software vendor or software version of the requesting software user agent. Examples: ESP8266/1.0, Particle/1.2
- {DEVICE_LABEL}: *Mandatory*. The [device label](#) to publish values. If it does not exist, Ubidots will create it. Blank spaces are not allowed. Examples: Weather-station, living-room
- {DEVICE_Name}: *Optional*. The device name to publish values. If not set, the device name will be set using the device label. This parameter is useful when you have as device label your device MAC and you wish to show to final users a friendly name. Blank spaces are not allowed. Examples: Street-42-weather-station, living-room-floor-1
- {GLOBAL_TIMESTAMP}: *Optional*. Unix time stamp. If set, any variable value without timestamp will use this one to store values. Must be in milliseconds. If not set, Ubidots will add it automatically with the actual timestamp in UTC. Examples: 1534282544356
- {VARIABLE_LABEL}: *Mandatory*. The variable label that will store the dot. If it does not exist, Ubidots will create it. You may send multiple variables splitting them by commas. Examples: temperature, wind-speed
- {VALUE}: *Mandatory*. Dot value. You can use a dot to set float numbers. Examples: 1.0005, 2
- {CONTEXT_KEY}: *Optional*. If set, you must specify a context value. You can add as many context key as you wish splitting them with a '\$' char. Examples: lat, lng, name, surname.
- {CONTEXT_VALUE}: *Optional*. The context key value. Examples (based on keys above): -6.2, 75.4, "John", "Smith"
- {VARIABLE_TIMESTAMP}: *Optional*. Unix time stamp. If set, the variable will use this one to store dots. This parameter has precedence among the global timestamp. Must be in milliseconds. If not set, Ubidots will add it automatically with the actual timestamp in UTC. Examples: 1534282544356

ERROR|400

- Valid messages: Ubidots will answer Ok if the request is correct. If multiple values are sent, Ok|OK... strings (one per variable in the request) splitted by pipelines '|' will be returned.
- Non valid messages: Ubidots will answer ERROR|400 if the message is not valid.

Stream Examples

- Variable with multiple context values:

```
ubidots/1.0|POST|{TOKEN}|weather-station=>temperature:20$day-hour=11$day-minute=23|end
```

- Variable with same value to multiple devices:

```
ubidots/1.0|POST|{TOKEN}|weather-station-denver=>temperature:20;weather-station-london=>temperature:20|end
```

Examples

Send a single dot to a device with name *green-truck* and device label *5b7356ccbdbdd594df54555* stored inside the variable *speed*:
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|5b7356ccbdbdd594df54555:green-truck=>speed:2|end

Send a single dot to a device named as *green-truck* with device label *5b7356ccbdbdd594df54555* stored inside the variable *speed* at 2017/12/31 8:00 UTC:
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|5b7356ccbdbdd594df54555:green-truck=>speed:2@1514707200000|end

Send a single dot to a device labeled as *green-truck* with api label *5b7356ccbdbdd594df54555* stored inside the variable *speed* at 2017/12/31 8:00 UTC with latitude, longitude and other keys inside its context:
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|5b7356ccbdbdd594df54555:green-truck=>speed:2\$lat=-6.2\$lng=75.4\$name=John\$surname=Smith@1514707200000|end

Send multiple values

```
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|weather-station=>temperature:20,humidity:35|end
```

Send values to multiple devices

```
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|weather-station-tor  
onto=>temperature:20,humidity:35;weather-station-washington=>temperature:20,humidity:35|end
```

Send a single dot (Educational users)

```
echo -n "ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|5b7356ccb  
bddbd594df54555:green-truck=>speed:2|end" | telnet things.ubidots.com 9012
```

Send a single dot (Industrial users)

```
echo -n "ESP8266/1.0|POST|BBFF  
kiFdJvtV3OWTY56M0rKluUGm71lz6k6oWpbWp7hJFah7fL6p1kcks|green-
```

Send multiple dots (Educational users)

```
echo -n "ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQfImwRQW|weather-  
st  
ation=>temperature:20,humidity:35|end" | telnet things.ubidots.com 9012
```



Send multiple dots (Industrial users)

```
echo -n "ESP8266/1.0|POST|BBFF-kiFdJvtV3OWTY56M0rKluUGm71lzzn6kboWpb  
WpfhJFah7fL6p1kcks|weather-station=>temperature:20,humidity:35|end" | telne  
t industrial.api.ubidots.com 9012
```



Retrieve Data

Structure

Ubidots TCP/UDP supports the payload request structure below for retrieving dots values

```
{USER_AGENT}|LV|{TOKEN}|{DEVICE_LABEL}:{VARIABLE_LABEL}|end  
{USER_AGENT}|GET|{TOKEN}|{VARIABLE_ID}|end
```

where:

- {USER_AGENT}: *Mandatory*. Contains a characteristic string that allows to identify the application type, operating system, software vendor or software version of the requesting software user agent. Examples: ESP8266/1.0, Particle/1.2
 - {DEVICE_LABEL}: *Mandatory*. The [device label](#) to publish values. If it does not exist, Ubidots will create it. Blank spaces are not allowed. Examples: Weather-station, living-room
 - {VARIABLE_LABEL}: *Mandatory*. The variable label that stores the dot. Examples: temperature, wind-speed
-
- {VARIABLE_LABEL}: *Mandatory*. The variable unique identifier that stores the dot. Examples: 573dc1ee76254241c57a4616

Server Response

OK|Number

ERROR|400 , ERROR|404

- Valid message: Ubidots will answer Ok| followed by the dot value if the request is correct.

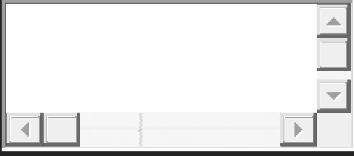
- Non valid message: Ubidots will answer ERROR|400 if the message is not valid or ERROR|400 if the message contains a variable or device that does not exist.

Examples

Get the last value from the variable *speed* inside a device labeled with api label *green-truck*:
 ESP8266/1.0|LV|p4uuT2O25AJwv7ncADLoVhQfImwRQW|green-truck:speed|end
 Get the last value from the variable *speed* inside a device labeled with api label *green-truck* using the *speed* variable unique id:
 ESP8266/1.0|GET|p4uuT2O25AJwv7ncADLoVhQfImwRQW|573dc1ee76254241c57a4616|end

Get last variable value (Educational users)

```
echo -n "ESP8266/1.0|LV|p4uuT2O25AJwv7ncADLoVhQfImwRQW|green-truck:speed|end" | telnet things.ubidots.com 9012
```



Get last variable value (Industrial users)

```
echo -n "ESP8266/1.0|LV|BBFF-kiFdJvtV3OWTY56M0rKluUGm71lzzn6kboWpbWp fhJFah7fL6p1kcks|green-truck:speed|end" | telnet industrial.api.ubidots.com 9012
```



Utils

Ubidots also supports to retrieve the actual UTC timestamp from our servers.

Structure

Ubidots TCP/UDP supports the payload request structure below for retrieving dots values

{USER_AGENT}|UTCNOW|||e
 nd where:

- {USER_AGENT}: *Mandatory*. Contains a characteristic string that allows to identify the application type, operating system, software vendor or software version of the requesting software user agent. Examples: ESP8266/1.0, Particle/1.2

Server

Response

OK|TIMESTAM

P ERROR|400

- Valid message: Ubidots will answer Ok| followed by the actual UTC timestamp in milliseconds.
- Not valid message: Ubidots will answer ERROR|400 if the message is not valid.

CHAPTER 6

6.1 References:

1. Madrazo, J.; Clappier, A.; Cuesta, O.; Belalcázar, L.C.; González, Y. Evidence of traffic-generated air pollution in Havana. *Atmósfera* 2019, 32, 15–24. [CrossRef]
2. Qu, H.; Lu, X.; Liu, L.; Ye, Y. Environmental Effects of traffic and urban parks on PM10 and PM2.5 mass concentrations. *Energy Sources Part A Recovery Util. Environ. Eff.* 2019, 1–13. [CrossRef]
3. Kendrick, C.M.; Koonce, P.; George, L.A. Diurnal and seasonal variations of NO, NO2 and PM2.5 mass as a function of traffic volumes alongside an urban arterial. *Atmos. Environ.* 2015, 122, 133–141. [CrossRef]
4. ISPRA Infographic on Air Quality and Environment. Available online: https://www.isprambiente.gov.it/files2020/pubblicazioni/stato-ambiente/annuario-2020/Infografiche_r.pdf (accessed on 3 September 2020).
5. World Health Organization. *Ambient Air Pollution: A Global Assessment of Exposure and Burden of Disease*; World Health Organization: Geneva, Switzerland, 2016.
6. United Nations. *The Sustainable Development Goals Report 2020*; United Nations: New York, NY, USA, 2020.
7. European Environment Agency. *Air Quality in Europe—2019 Report*; European Environment Agency: Copenhagen, Denmark, 2019.

report

by Report S

Submission date: 17-Apr-2021 12:12AM (UTC+0700)

Submission ID: 1561131400

File name: MONITORING_AND_CONTROL_OF_VEHICLE_USING_NODEMCU_MICROCONTROLLER.docx (1.54M)

Word count: 9152

Character count: 52732

SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER

B.PAVAN KUMAR REDDY -37130057

B.ARUN KUMAR-37130044

GUIDE:DR.GD.ANBARASIJEBASELVI,M.E.,Phd3,As
sociate Professor,Dept of ECE

SMART AIR POLLUTION MONITORING AND CONTROL OF VEHICLE USING NODEMCU MICROCONTROLLER

Abstract:

- As fuel is burned in an internal combustion engine and air/fuel residuals are generated through the vehicle tailpipe, carbon monoxide, nitrogen oxides, and hydrocarbons are released. During refuelling and when fuel vaporises from engines and fuel systems due to vehicle activity and hot weather, gasoline vapours often escape into the atmosphere.
- Contaminants in automobile emissions have been shown to damage lung tissue, as well as cause and exacerbate respiratory disorders including asthma. Motor vehicle emissions also leads to acid rain formation and increases the amount of greenhouse gases that cause global warming.
- Pollutants emitted directly from automobiles aren't the only thing to be concerned about. On hot, sunny days, hydrocarbons react with nitrogen oxides to form ozone, a secondary pollutant. Motor vehicles are the single largest contributor to ground-level ozone, a natural component of smog, in many urban areas. Coughing, wheezing, and shortness of breath are all symptoms of ozone, and it can lead to irreversible lung damage, making it a major public health problem.
- This problem can be solved by using a sensor called a smoke sensor that is connected to the internet. Send the information to the government.

Introduction:

- TECHNOLOGICAL AND ECONOMIC Developments ARE HAVING A Direct Effect ON THE ENVIRONMENT, CAUSING SERIOUS Questions ABOUT Emissions AND CLIMATE CHANGE. The Intergovernmental Panel on Climate Change (IPCC) confirmed in its 2014 report that human activities have an undeniable and steadily growing effect on the climate system, with recent changes that are unparalleled over decades to millennia.
- In this context, environmental monitoring is a critical tool for collecting accurate information about the ecosystem, resulting in new awareness and understanding, and eventually implementing adaptation and mitigation strategies.

- The most important parameters being tracked include temperature, humidity, rainfall, ambient pressure, light intensity, air quality, which is influenced by pollutants such as CO₂, CO, SO_x, volatile organic compounds, and many others. One of the most immediate benefits of acquiring physical properties such as soil moisture, temperature, and salinity can be seen in agriculture, where significant water resource savings can be made.
- WSNs are used in the majority of control applications due to their undeniable benefits.

Literature survey:

1. A Smart Monitoring System for Air Pollution

Kennedy Okokpujie, Etinosa Noma-Osaghae, Modupe Odusami, Samuel John Okokpujie, Etinosa Noma-Osaghae, Samuel John Okokpujie, Etinosa Noma-Osagha

Air pollution has an effect on our daily lives and quality of life. It poses a challenge to the planet's biodiversity and quality of life. Because of increased industrial activities in recent years, there is a clear need to regulate air quality. People must be aware of the degree to which their actions have an effect on air quality.

2. Monitoring System for Air Quality

Boyanka Nikolova, Marin B. Marinov, and Georgi Todorov Nikolov are the writers.

Low concentrations of air pollution, such as tobacco smoke and cooking fumes, can be detected using a combination of an air quality monitor and an air quality monitor.

3. An air quality control system for metropolitan areas focused on wireless sensor network technology.

Jen-Hao Liu, Yu-Fan Chen, Tzu-Shiang Lin, and Chia-Pang Chen are the writers.

One of the environmental problems that cannot be overlooked is air pollution. As a result of heavy traffic and urbanisation, air emissions are concentrated in specific areas. Long-term inhalation of toxins is harmful to human health. Building air quality monitoring stations, for example, is a costly form of air quality monitoring. Monitoring stations are often less densely distributed and provide low-resolution sensing data. This paper suggests a method for controlling urban air quality.

4. AIR QUALITY MONITORING SYSTEM BASED ON IOT

Ch.V.Saikumar, M.Reji, and P.C.Kishoreraja are the writers.

The key goal of this project is to keep track of air pollution in industrial and urban areas. A series of gas sensors (CO and NO₂) are located on the masses and structure of an IOT (Internet of things), along with a dominant server, to help both short-term real-time event management and long-term deliberate preparation. The Arduino framework is used to communicate data easily and quickly in this project. The transceiver is a WSN (Wireless Sensor Network). This provides a real-time low-rate monitoring device for the use of wireless networking technologies with limited power.

Existing system:

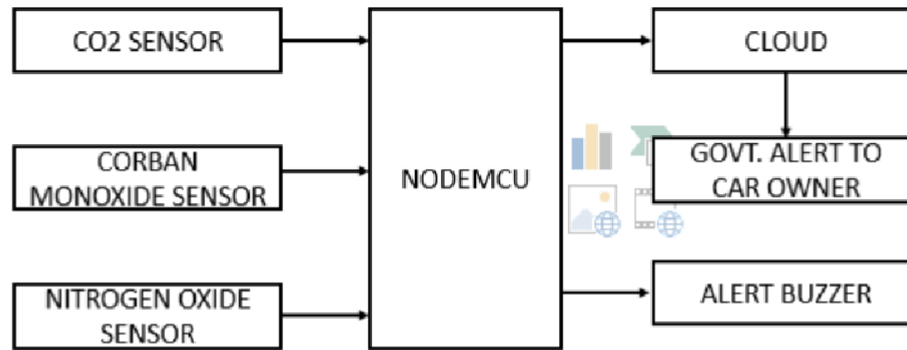
The microcontroller is primarily programmed to perform three tasks: comparison, timer, and triggering. When the smoke sensor output exceeds the threshold value, the microcontroller activates the timer circuit and an alarm is activated to warn the driver that the vehicle is polluted and will come to a halt. The MQ-2 semiconductor sensor is used to detect smoke with a detection range of 300ppm to 10000ppm. In a clean setting, a CO sensor with low conductivity is connected to a PIC microcontroller. As this sensor is exposed to contaminants, its conductivity rises, causing a signal to be generated in the circuit, disabling the motor. A gsm module can also send an automatic SMS.

Proposed system:

The proposed system constantly monitors the automotive emissions control.

- Carbon monoxide and nitrogen oxides sensors can be used in automotive exhaust systems. It receives two inputs: one from the smoke sensor and the other from the government's pre-defined threshold value.
- If the threshold value is exceeded, the vehicle's location can be tracked using a GPS module, and the information is submitted to the government. The government can decide to cancel the car's registration and send a letter to the owner.

Block diagram:



Hardware required:

- CO2 SENSOR
- NITROGEN OXIDE SENSOR
- NODEMCU
- CORBAN MONOXIDE
- GPS

1. EMBEDDED SYSTEMS

1.1 Overview of embedded systems

A special-purpose computer system designed to perform one or a few dedicated tasks, often with real-time computing constraints, is known as an embedded system. It's normally found as part of a larger device that includes both hardware and mechanical components. A general-purpose computer, such as a personal computer, on the other hand, can perform a wide range of tasks depending on the programming. Embedded systems have become increasingly relevant in today's world, as they power many of the devices we use on a daily basis.

Since the embedded device is dedicated to a particular mission, design engineers may customise it to reduce the product's size and cost while also improving its reliability and efficiency.

Embedded systems provide several functions

Keep an eye on the surroundings with embedded systems that read data from input sensors. The data is then analysed, and the results are shown to a user or users in some format.

- Maintain environmental control; embedded systems produce and send commands to actuators.
- Change the information; embedded systems change the data collected in some way, such as by compressing and decompressing it.

While embedded systems provide interaction with the outside world via sensors and actuators, they also provide functionality unique to their applications. Control rules, finite state machines, and signal processing are popular applications that are run on embedded 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 processor is typically found in a device. Embedded devices are used in many products with a digital interface, such as microwaves, VCRs, and automobiles. An operating system is used in some embedded systems. Others are highly specialised, resulting in the implementation of the entire logic as a single application. These systems are integrated into a computer for a particular use other than general-purpose computing. Figure 1.1 depicts a typical embedded device.

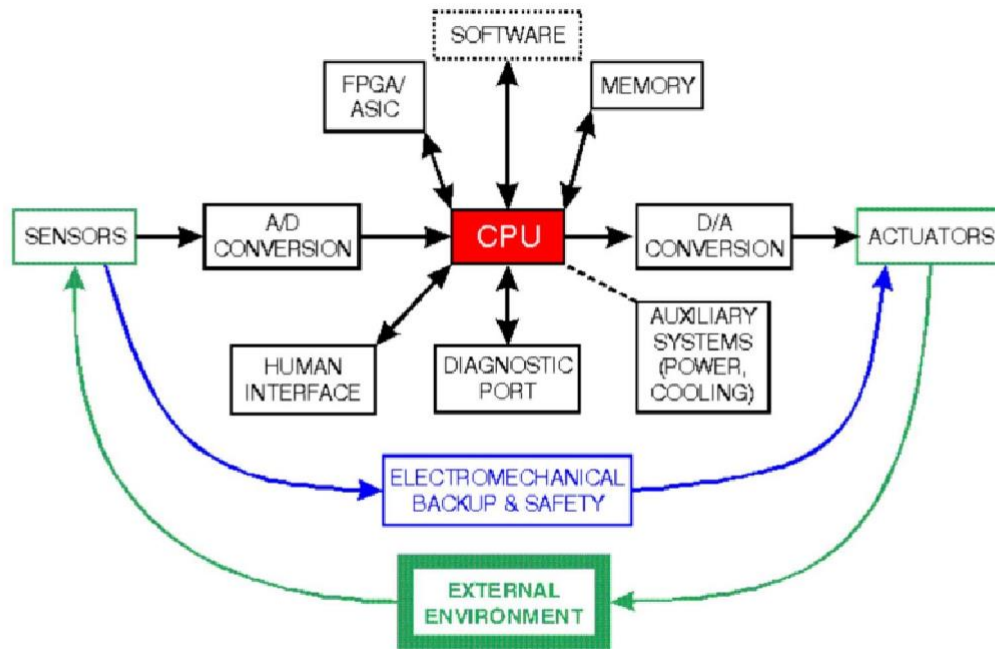


Fig 1.1 Block diagram of a typical embedded system

1.2.1 Characteristics of embedded systems

Embedded systems have a distinct collection of features that make them stand out. Embedded device designers were subjected to a unique set of design constraints as a result of each of these characteristics. The difficulty in designing embedded systems is adhering to the application's particular set of constraints.

Application Specific Systems:

Embedded devices aren't the same as regular machines. The designs of embedded systems are tailored to a particular application. Before the hardware is built, many of the job characteristics are understood. This helps the designer to concentrate on the application's precise design constraints. As a result, the user's ability to reprogram is minimal. Some embedded devices, on the other hand, necessitate the ability to reprogram. For such applications, programmable DSPs are popular.

Reactive Systems

As previously mentioned, a typical embedded systems model uses sensors to react to the environment and actuators to control the environment. This necessitates embedded systems running at the same rate as their surroundings. The term "reactive" describes this feature of

embedded systems. The machine (primarily the software component) executes in response to external events, which is known as reactive computation.

Distributed Systems

An embedded device that consists of communicating processes running on multiple CPUs or ASICs linked by communication links is a common feature. This is due to financial constraints. 4 8-bit microcontrollers can be less expensive than 32-bit processors. And when the cost of communication connections is factored in, this strategy could be preferable. Multiple processors are typically needed in this approach to handle multiple time-critical tasks. Physical distribution of devices regulated by embedded systems is also possible.

Heterogeneous Architectures

Different processors can be found in the same device solution. It's also possible that they're mixed signal systems. Embedded devices are mostly made up of heterogeneous architectures due to the combination of I/O interfaces, local and remote memories, and sensors and actuators (Fig 1.2). They could create a completely unique embedded device design. Embedded systems are therefore constrained in terms of configuration, and heterogeneity allows for more design versatility.

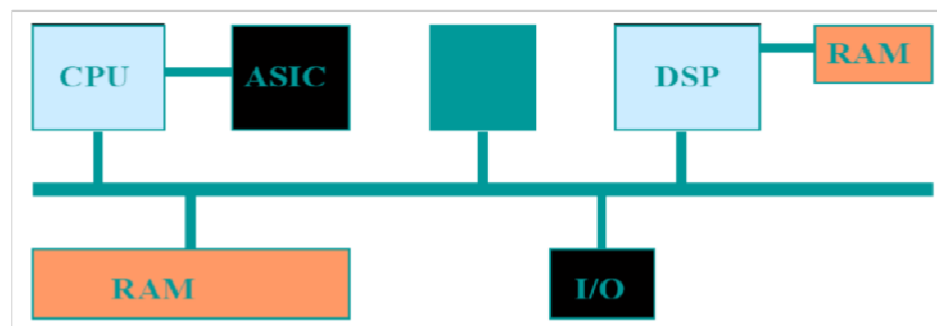


Fig 1.2 Embedded Systems having Heterogeneous Architectures

Harsh environment

Many embedded systems aren't designed to work in a controlled environment. Excessive heat is a common issue, particularly in combustion-related applications (e.g., many transportation applications). Defense from vibration, shock, lightning, power supply fluctuations, water, erosion, fire, and general physical abuse may all trigger problems for embedded computing.

System safety and reliability

As embedded device complexity and computational capacity increase, they are beginning to monitor more and more of the overall system's safety aspects. These safeguards could be implemented in the form of software or hardware. When a computer system loses power, mechanical safety backups are usually triggered to safely shut down the system. The problem of software security and reliability is becoming more severe. Normally, software does not "break" in the same way as hardware does. However, software can be so complex that a series of unanticipated events can result in software errors, resulting in dangerous situations. This subject is beyond the reach of this book, but the challenges for e-commerce are important.

Control of physical systems

The ability to communicate with the world is one of the key reasons for embedding a device. This is often accomplished by the use of external equipment to track and regulate. For processing, embedded computers convert analogue signals from sensors to digital signals. The outputs must be converted to analogue signal levels again. Large current loads can need to be transferred when controlling physical equipment in order to operate motors and other actuators. Embedded systems may require large computer circuit boards with a large number of non-digital components to meet these requirements. Device tradeoffs between analogue components, power, mechanical, network, and digital hardware must be carefully balanced by embedded system designers.

Small and low weight

Embedded computers are computers that are physically embedded in a larger system. The form factor of the embedded system can be determined by aesthetics. The form factor of a missile, for example, may need to fit within the missile's nose. One of the challenges faced by embedded system designers is developing non-rectangular geometries for such solutions. A person's weight can also be a stumbling block. To save fuel, embedded vehicle control systems, for example, must be lightweight. Portable CD players must be light in weight in order to be portable.

Cost sensitivity

Cost is a problem in most systems, but in embedded systems, the sensitivity to cost adjustments can differ drastically. This is mostly due to the impact of computing costs on

profitability, and it is more a function of the proportion of cost adjustments to overall device costs.

Power management

The power consumption of embedded systems is strictly controlled. Given the portability requirements of many embedded systems, power conservation is critical in order to maximise battery life. Another apparent problem for embedded systems is the reduction of heat generation.

1.3 POWER SUPPLY

Since all electronic circuits operate at low DC voltages, we need a power supply unit to provide the correct voltage supply for them to function properly. The transformer, rectifier, filter, and regulator are all part of this unit. A transformer voltage down to the level of the desired ac voltage is connected to an AC voltage of normally 230 volts rms. A full-wave rectified voltage is provided by a diode rectifier, which is then filtered by a simple capacitor filter to produce a dc voltage. There is typically some ripple or ac voltage difference in the resulting dc voltage. This dc input can be used by a regulator circuit to generate dc voltage

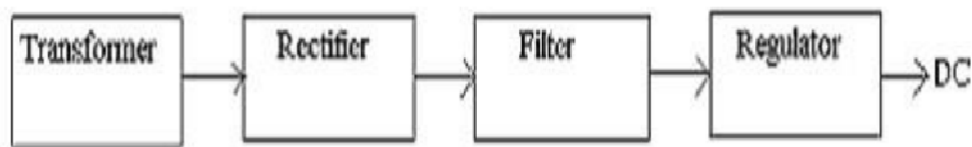


Fig 1. General Block Of Power Supply Unit

1.4 TRANSFORMER:

A transformer is a static component that converts electric power from one circuit to electric power from another circuit at the same frequency. It has the ability to increase or decrease

the voltage in the circuit, but it does so at the expense of a decrease or increase in current. It operates on the mutual induction principle. A step down transformer is used in our project to provide the required supply for the electronic circuits. We're converting 230 volts ac to 12 volts ac here.

1.5 RECTIFIER:

A process known as full wave rectification will increase a dc level obtained from a sinusoidal input by 100%. For full wave rectification, we use a bridge rectifier in our project. During the time $t = 0$ to $T/2$, two diodes (say D2 & D3) conduct while the other two diodes (D1 & D4) are in the off state, as seen in the simple bridge configuration. The conducting diodes for the negative cycle of the input are D1 and D4. As a result, the polarity is consistent in the load.

The diodes used in the bridge rectifier can be of various forms, such as 1N4001, 1N4003, 1N4004, 1N4005, 1N4007, and so on. However, we'll use 1N4007 because it can withstand 1000 volts.

1.6 FILTERS:

We'll need to use a low pass filter to get a dc voltage of 0 Hz. As a result, a capacitive filter circuit is used, with a capacitor connected to the rectifier output and a dc obtained across it. The filtered waveform is basically a dc voltage with minimal ripples, and it is fed to the load at the end.

1.7 REGISTERS:

Two 8-bit registers, an instruction register (IR), and a data register make up the controller IC (DR). For display data RAM (DD RAM) and character generator RAM, the IR stores the instruction codes and address information (CG RAM). The MPU will write to the IR but not read it.

Data to be written to/read from the DD RAM or CG RAM is temporarily stored in the DR. As an internal operation, the data written to DR by the MPU is automatically written to the DD RAM or CG RAM.

When an address code is written to IR, data from the DD RAM or CG RAM is automatically transferred to the DR. When the MPU reads the DR, the data transfer between the MPU is complete.

1.8 REGULATORS:

The capacitor's output voltage is filtered and finally regulated. The voltage regulator is a system that keeps the output voltage constant regardless of supply variations, load variations, or temperature variations. We're going to use the LM7805, which is a fixed voltage regulator. The IC LM7805 is a microcontroller regulator that operates at 5 volts.

1.9 FEATURES AND DESCRIPTION OF REGULATORS

- Up to 1A output current
- 5, 6, 8, 9, 10, 12, 15, 18, 24V output voltages
- Protection Against Thermal Overload
- Protection from short circuits
- Safe Operating Area Protection for Output Transistors

NODE MCU

The NodeMCU framework is a free and open source IoT platform. It includes firmware that runs on Espressif Systems' ESP8266 Wi-Fi SoC and hardware that is built on the ESP-12 board. By design, the name "NodeMCU" applies to the firmware, not the development kits. Lua is the scripting language used by the firmware. It was created using the Espressif Non-OS SDK for ESP8266 and is based on the eLua project. It makes use of a number of open source projects, including lua-cjson and SPIFFS.

Shortly after the ESP8266 was released, NodeMCU was developed. Espressif Systems began mass production of the ESP8266 on December 30, 2013. The ESP8266 is a Wi-Fi SoC with a Tensilica Xtensa LX106 core that is commonly used in Internet of Things (IoT) applications (see related).

The MQTT IoT protocol was then supported by NodeMCU, which used Lua to link to the MQTT broker. Another significant update occurred on January 30, 2015, when Devsaurus ported the u8glib library to the NodeMCU project, allowing NodeMCU to drive LCD, Screen, OLED, and even VGA displays with ease.

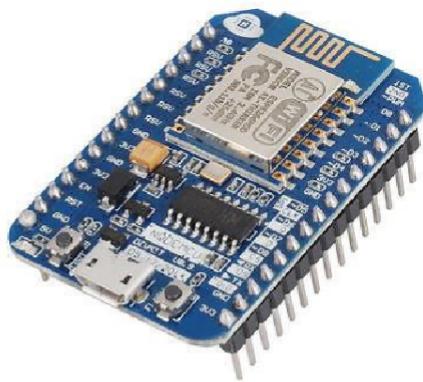
The developers left the firmware project in the summer of 2015, and it was taken over by a consortium of independent contributors. By the summer of 2016, the NodeMCU had grown to include over 40 separate modules. Users must select the modules applicable to their project and create a firmware customised to their needs due to resource constraints.

ESP8266 Arduino Core [edit]

As Arduino.cc began to develop new MCU boards based on non-AVR processors, such as the ARM/SAM MCU used in the Arduino Due, they needed to adapt the Arduino IDE to embrace alternative toolchains so that Arduino C/C++ could be compiled for these new processors. They were able to accomplish this after the arrival of the Board Manager and the SAM Core. To compile an Arduino C/C++ source file for the target MCU's machine language, both the Board Manager and the Arduino IDE require a "core" of software components. Some ESP8266 enthusiasts developed the "ESP8266 Core for the Arduino IDE" as an Arduino core for the ESP8266 WiFi SoC.

Introduction

NodeMCU is a LUA-based open source firmware for the ESP8266 wifi chip. NodeMCU firmware comes with ESP8266 Development board/kit, i.e. NodeMCU Development board, to explore functionality with ESP8266 chip.

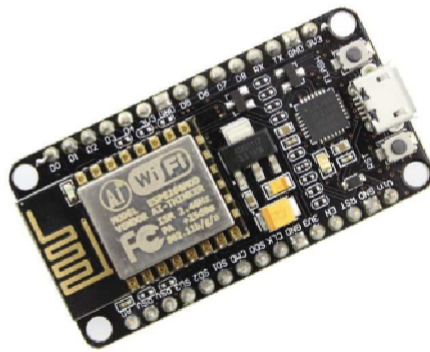


NodeMCU Development Board/kit v0.9 (Version1)

Since NodeMCU is an open source platform, their hardware design is available for everyone to edit, change, or create.

The ESP8266 wifi enabled chip is used in the NodeMCU Dev Kit/board. The ESP8266 is a low-cost Wi-Fi chip with TCP/IP protocol developed by Espressif Systems. Refer to the ESP8266 WiFi Module for more detail on the ESP8266.

Version 2 (V2) of the NodeMCU Dev Kit is available, namely NodeMCU Development Board v1.0 (Version2), which is usually black in colour.



NodeMCU Development Board/kit v1.0 (Version2)

NodeMCU Development Boards provides more detail about the NodeMCU Boards available on the market.

The Analog (A0) and Digital (D0-D8) pins on the NodeMCU Dev Kit's board are similar to those on an Arduino.

It is compatible with serial communication protocols such as UART, SPI, and I2C

We can link it to serial devices including I2C powered LCD displays, Magnetometer HMC5883, MPU-6050 Gyro metre + Accelerometer, RTC chips, GPS modules, touch screen displays, SD cards, and so on using these serial protocols.

How to start with NodeMCU?

The wifi functionality, analogue pins, digital pins, and serial communication protocols are all included on the NodeMCU Development board.

To begin using NodeMCU for IoT applications, we must first understand how to write and download NodeMCU firmware on NodeMCU Development Boards. And before that, where will this NodeMCU firmware be customised to meet our needs?

Online NodeMCU custom builds are available, allowing us to quickly create our own NodeMCU firmware to meet our specific needs.

Refer to this page for more information on [how to build custom NodeMCU firmware online and download it](#). NodeMCU: [Getting Started](#)

How to write codes for NodeMCU?

Let's look at the IDE (Integrated Development Environment) needed for NodeMCU development after we've set up the ESP8266 with Node-MCU firmware.

NodeMCU with ESPlorer IDE

The NodeMCU is usually coded using scripts. Lua is a [lightweight, embeddable scripting language](#) based on the [C programming language](#) that is open source.

Refer to this page for more detail on how to write Lua scripts for NodeMCU. Using ESPlorerIDE to get [started with NodeMCU](#)

NodeMCU with Arduino IDE

[Another](#) choice for [developing NodeMCU](#) is to use the Arduino IDE, which is a well-known IDE. We can also use the Arduino development environment to build NodeMCU applications. This makes learning a new language and IDE for NodeMCU much easier for Arduino developers.

Refer to this page for more detail on how to write an Arduino sketch for the NodeMCU. Using ArduinoIDE to get started with [NodeMCU](#)

Difference in using ESPlorer and Arduino IDE

When designing applications for NodeMCU using ESPlorer IDE and [Arduino IDE](#), we can assume that there is a programming language difference.

If we're using the [Arduino IDE](#) to build NodeMCU applications, we'll need to code in C++, and if we're using the ESPlorer IDE, we'll need to code in Lua.

NodeMCU is a [Lua interpreter](#), which means it can quickly understand Lua language. When we write Lua scripts for NodeMCU and send/upload them, they are executed in a sequential order. [It will not generate a binary](#)

firmware file for NodeMCU to read. It will submit the Lua script to NodeMCU in its current state for execution.

IOT:

The following is the chapter's structure: The motivation for this project is described in Section 1.1. The research objectives and questions are mentioned in Section 1.2. Section 1.3 outlines the measures necessary to achieve our goals. The report's structure is shown in Section 1.4.

1.1 Background

The aim of the Future Internet is to create an infrastructure that allows people to have instant access to knowledge about the real world and its artefacts. Physical artefacts can be used in a variety of applications, including e-health, warehouse management, and so on. Different types of physical devices may be used in different application domains. Each physical system has its own set of standards that must be followed in order to communicate with it. A layered vision that can promote data access is needed to achieve the future Internet target. The Internet of Things (IoT) is a vision that seeks to use a layered architecture to link the virtual world of knowledge to the real world of devices.

Middleware serves as a link between the hardware and application layers, and it is responsible for system interaction and data management [2]. A middleware's job is to provide a single programming model for interacting with devices. When communicating with computers, middleware is responsible for masking the heterogeneity and delivery issues that we encounter [3].

1.2 Motivation

The aim of an IoT-based system is to provide information from the environment to a non-expert user. Since an IoT-based framework may be used in a variety of settings, it must be able to address a large number of heterogeneous devices. As a result, a major concern when designing an IoT-based framework is how to manage non-expert users' contact with heterogeneous devices. A middleware layer between devices and non-expert users may solve this issue. This layer is responsible for hiding the variety of devices from the user's perspective and providing end users with system control transparency.

The concept of constructing interface abstractions has been discussed in the literature

2.1. IoT definition

We'll go through some IoT meanings in this section. We also go over IoT's layered architecture.

The Internet of Things (IoT) is gaining popularity in industry as a way to communicate with various types of devices. By incorporating physical devices into communication networks, the Internet of Things will have an effect on business and society [8]. IoT effects can be seen from a variety of angles, including private and business users. IoT has an effect on both professional and personal areas, such as smart homes and workplaces, e-health, and assisted living, from the viewpoint of a private individual. Impacts on business users will be in areas like robotics and industrial production, as well as logistics.

IoT is described by the International Telecommunication Union (ITU)¹ as “a global infrastructure for the Information Society, enabling advanced services by interconnecting (physical and virtual) stuff using existing and emerging interoperable information and communication technologies.” [nine]

“The idea of a globally interconnected continuum of computers, artefacts, and things in general originated with RFID technology, and this definition has been considerably expanded to the present vision that envisages a multitude of heterogeneous objects interacting with the physical environment,” according to the Internet of Things-Architecture 2

Edge Technology layer

This is the hardware layer, which includes embedded systems, RFID tags, sensor networks, and all other types of sensors. This hardware layer can perform a variety of tasks, including gathering data from a device or environment, processing data, and facilitating communication

Access Gateway layer

This layer is in charge of data handling and is in charge of publishing and subscribing to the services offered by the Stuff, as well as message routing and hovelling communication between platforms.

Middleware layer

This layer performs critical functions such as aggregating and filtering data obtained from hardware devices, performing knowledge discovery, and controlling device access for applications

Application layer

This layer is in charge of providing a variety of application services. These services are delivered to various applications and users in IoT-based systems via the middleware layer. The application services are applicable to a variety of sectors, including logistics, retail, and healthcare.

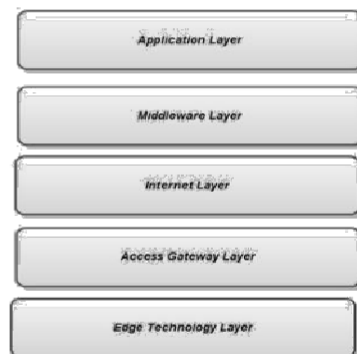


Figure 2-1 Layered architecture of the Internet of Things (IoT).

2.2 Challenges

Since IoT-based systems can be used for a variety of purposes and areas, we must face a variety of challenges. In this segment, we discuss some of the challenges that must be considered while conducting research [10]:

▣ Edge technology

Since an IoT-based system must be able to gather sufficient information about the real world by utilising a wide range of devices and environments, further research efforts are needed at the hardware level to improve the technology of embedded devices, sensors, actuators, and (passive and active) recognition. As a result, there is still work to be done in terms of connecting heterogeneous devices and deploying them in IoT applications, as well as providing support for new devices.

▣ Networking technologies

Things in the Internet of Things are linked via a variety of networks, including telephone, wired, and wireless networks. These networks allow bi-directional communication between real-world objects, devices, and services used by IoT applications at various levels. Although distributed data may cause privacy issues, this highly distributed structure should provide interconnection with low energy consumption.

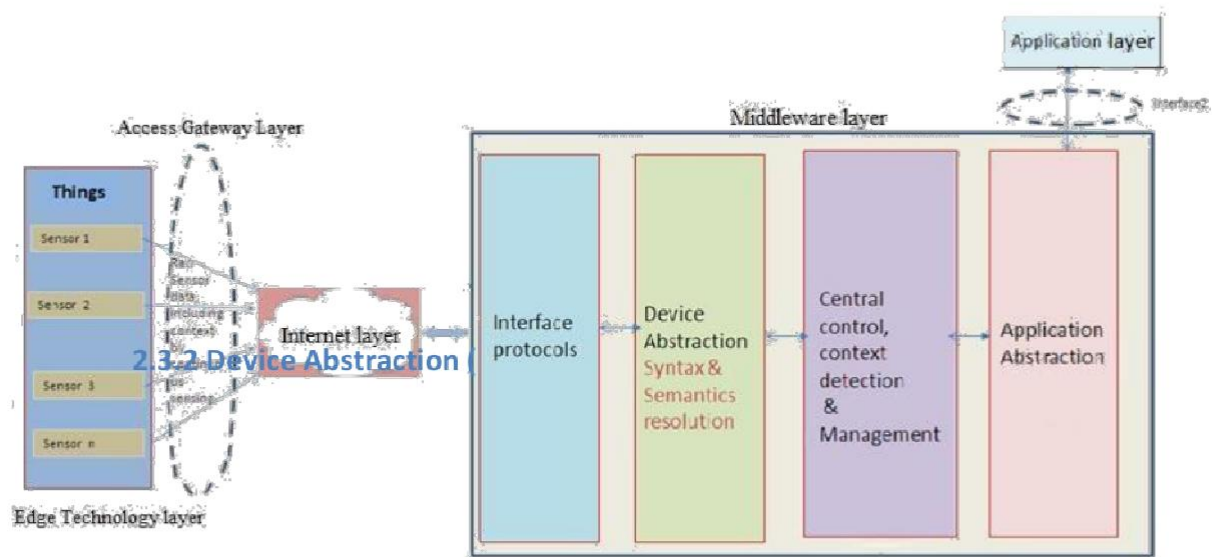
Middleware system

We have heterogeneous devices and networks in the Internet of Things. With new technologies, their heterogeneity can become even more pronounced. To make it easier for IoT applications to use these devices, we can hide their heterogeneity. To deal with the heterogeneity of devices, we need to build a stable, scalable, and semantically enriched service-oriented middleware. consumption of electricity

Platform services

They enable superior management of all involved devices in a unified fashion, ensuring scalability, high availability, and the safe and secure execution of system functionality. In the next section, we'll look at middleware problems, as we're looking for the features that a middleware can provide for the application layer in IoT-based systems.

2.3 IoT_based Middleware



2.3.3 Central control, Context detection & Management (CCM)

The situation of an individual, which can be a location, a person, or an object that is important to the user, applications, and their interactions, is referred to as context [1]. The CCM functional component is in charge of supporting context-aware computation, which is a method of computing that considers the context of the entities that communicate with the machine. To function in smart environments, middleware for IoT-based systems must be context-aware [1]. The term "smart environment" refers to a physical world that is richly and invisibly interwoven with sensors, actuators, screens, and computational elements, embedded seamlessly in everyday objects and linked through a continuous network [17].

- 1) Context detection, which entails gathering data from various sources and choosing information that may affect the computation.
- 2) Context processing, which involves using the information collected to complete a task or make a decision.

```
SELECT nodeid, temp
FROM sensors
SAMPLE PERIOD
```

2.4 Conclusion

We specified IoT and described common IoT layers in this chapter. A reference middleware architecture for IoT-based systems was also addressed. Based on a review of current middleware frameworks for IoT-based systems, Bandyopadhyay, S. et al proposed this architecture. 1st. ONE FOR TENS

Chapter 3 Review of IoT-based Middleware

We checked the literatures that discuss both IoT and Pervasive systems since the vision of IoT is almost identical to that of Pervasive computing. We used the following methods to choose the middleware to discuss:

- 1) We examined popular IoT middleware including ISMP [21], ASPIRE [22], and GSN [23]. In comparison to the other middlewares, we chose HYDRA [24] to review because it is the most common and well-documented. Furthermore, according to Bandyopadhyay, S. et al. [3], such middleware do not help the functional components discussed in Chapter 3, while HYDRA does.
- ❑ TinyDB [19] is a database that collects data from smartphones. We looked at TinyDB, a common middleware, since we need to gather data from the environment through different devices in an IoT-based framework.
- ❑ WiseMID [29] is the only middleware in the study that is specifically designed for energy conservation. We looked at this middleware because conserving energy is an important problem for devices.

CO2 sensor module

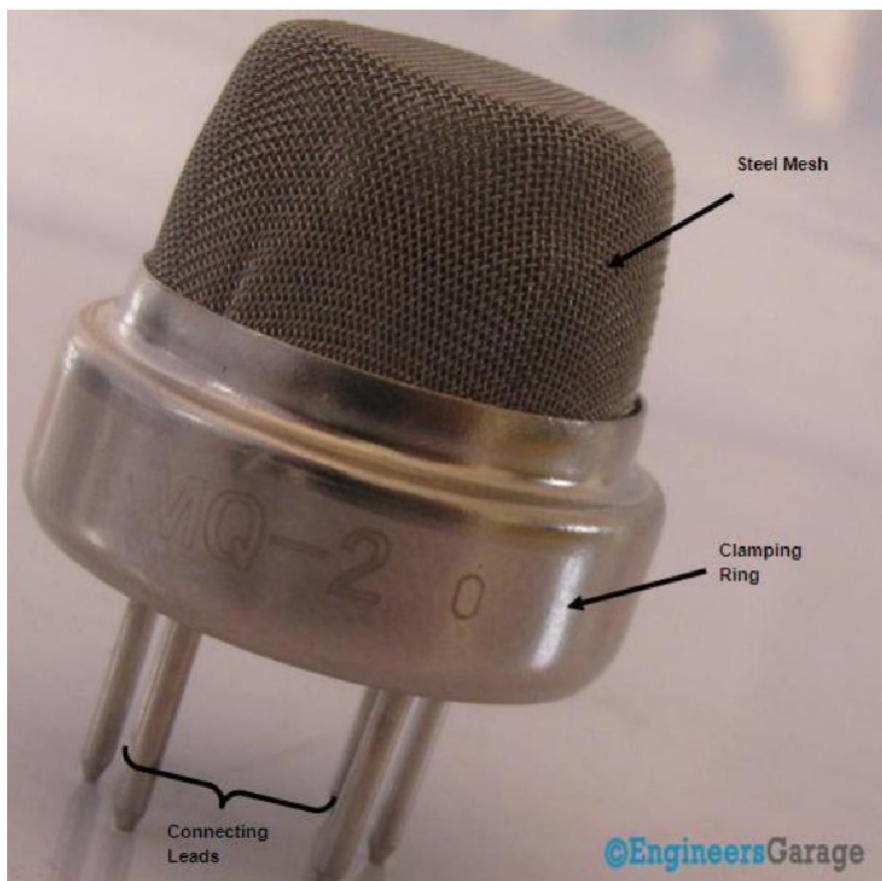
Monitoring of gases emitted is critical in today's technological setting. Monitoring of gases is important in everything from home appliances like air conditioners to electric chimneys and protection systems in industries. CO2 sensors are a critical component of such systems. Gas sensors, which are around the size of a nose, respond automatically to the gas present, keeping the device informed of any changes in the concentration of molecules in the gaseous state.

Gas sensors

Depending on the sensitivity levels, type of gas to be sensed, physical measurements, and a variety of other variables, are available in a variety of configurations. This Insight is about a methane gas sensor that can detect gases like ammonia that are emitted by methane. When a gas comes into

contact with this sensor, it is first ionised and then adsorbed by the sensing element. This adsorption produces a potential difference on the element, which is transmitted to the processor unit in the form of current through output pins.

The gas sensor module is made up of a steel exoskeleton that houses a sensing element. By connecting leads, current is applied to this sensing feature. The gases that come close to the sensing element become ionised and are absorbed by the sensing element as a result of this current, which is known as heating current. This alters the resistance of the sensing unit, resulting in a change in the value of the current leaving it.



A steel mesh, copper clamping ring, and connecting leads are shown on the outside of a standard gas sensor module. A stainless steel mesh covers the top section, which protects against the following:

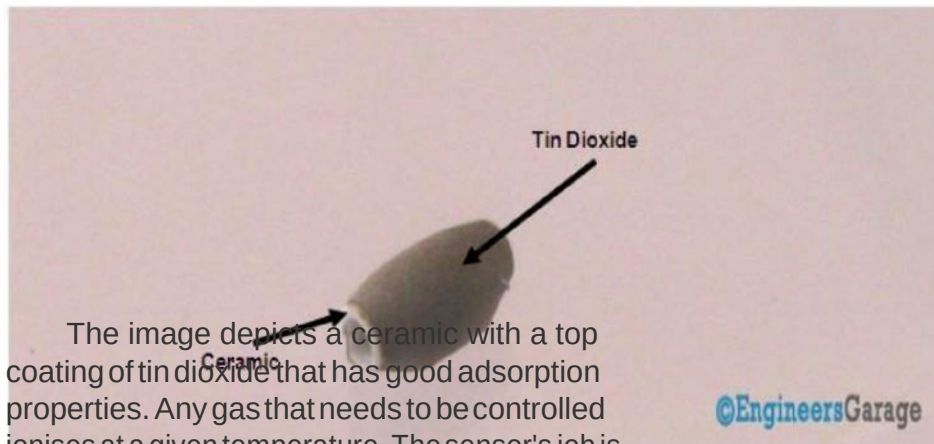
Filtering suspended particles out of the sensor so that only gaseous components can move through. Protects the sensor's internal components and features an anti-explosion network that holds the sensor module intact at high temperatures and gas pressures.

The steel mesh is divided into two layers to efficiently handle the functions mentioned above. A copper-plated clamping ring secures the mesh to the rest of the body.



The sensor's connecting leads are thick so that the sensor can be securely attached to the circuit and enough heat can be conducted to the inside portion. They're made of copper and have a tin plating on top. Four of the six leads (A, B, C, and D) are used to fetch signals, while two (1, 2) are used to keep the sensing factor warm.

The pins are mounted on a Bakelite frame, which is an excellent insulator and provides a tight grip on the sensor's connecting leads.



The image depicts a ceramic with a top coating of tin dioxide that has good adsorption properties. Any gas that needs to be controlled ionises at a given temperature. The sensor's job is to operate at the optimal temperature, causing gas molecules to ionise. Heating current is applied to the ceramic region of the sensing element via Nickel-chromium wire. The element radiates heat in the vicinity, where gases interact with it and become ionised. They are absorbed by tin dioxide after they have been ionised. Adsorbed molecules

alter the tin dioxide layer's resistance. This alters the current flowing through the sensing element, which is then transmitted to the uni through the output leads.

Gas sensor module

Monitoring of gases emitted is critical in today's technological setting. Monitoring of gases is important in everything from home appliances like air conditioners to electric chimneys and protection systems in industries. Gas sensors are an essential component of such systems. Gas sensors, which are around the size of a nose, respond automatically to the gas present, keeping the device informed of any changes in the concentration of molecules in the gaseous state.

Gas sensors come in a variety of configurations, depending on sensitivity levels, gas type, physical measurements, and a variety of other variables. This Insight is about a methane gas sensor that can detect gases like methane.



The gas sensor module is made up of a steel exoskeleton that houses a sensing element. By connecting leads, current is applied to this sensing feature. The gases that come close to the sensing element become ionised and are absorbed by the sensing element as a result of this current, which is known as heating current. This alters the resistance of the sensing unit, resulting in a change in the value of the current leaving it.

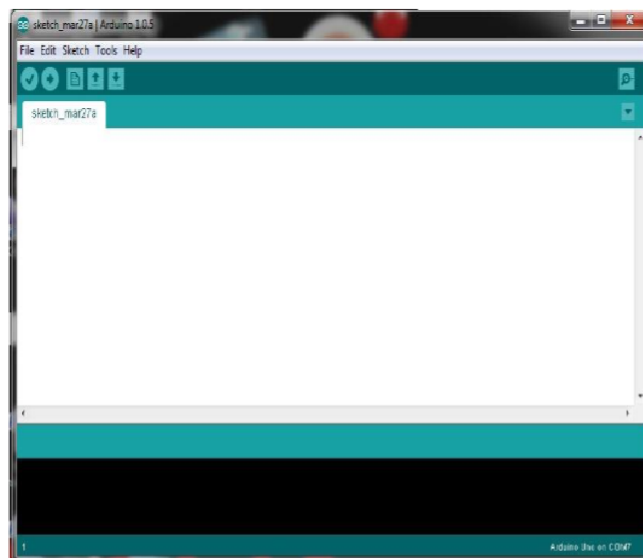
Software required:

Arduino ide

Arduino C

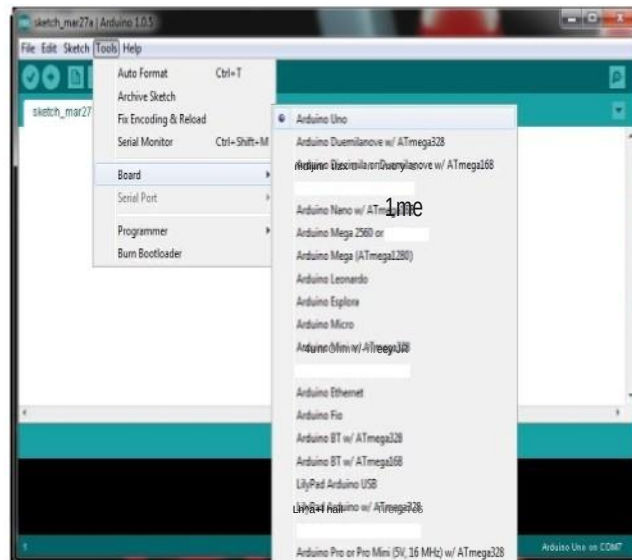
5 ARDUINO SOFTWARE:

1. As shown below, open the Arduino IDE.
Start the Arduino IDE.



2. From the tool, select the COM Port. Choose a COM Port.

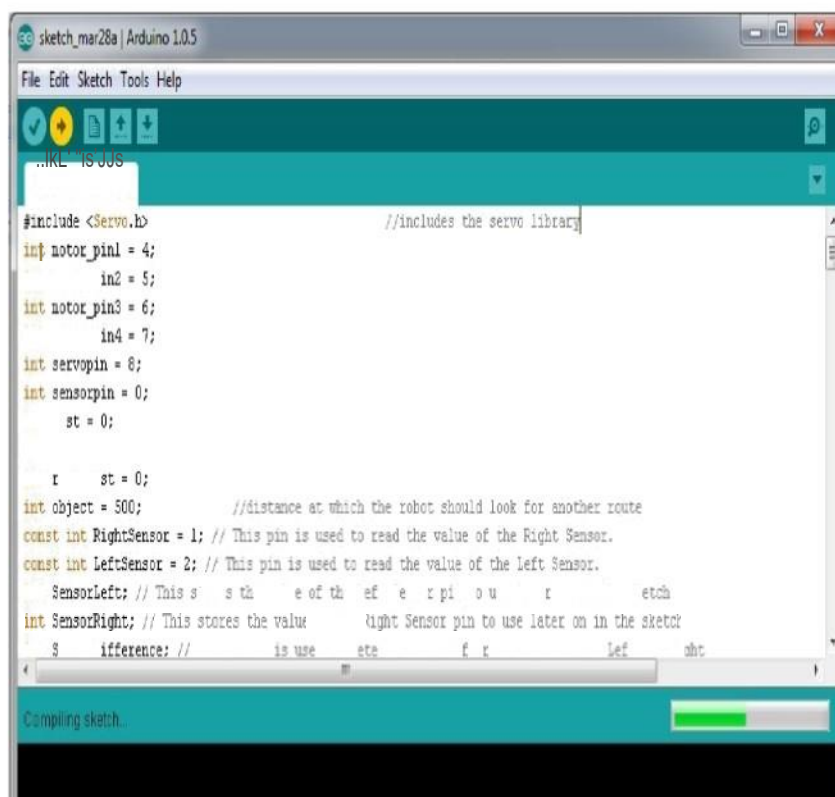
3. As shown below, go to Tools and select the appropriate Arduino board. Choose an Arduino board that you like.



4. In the Arduino IDE, build the sketch.

In the Arduino IDE, create a sketch

5. Compile the sketch and upload it to the Arduino board. uploading the sketch to the Arduino



UBIDOTS EXPLORER:

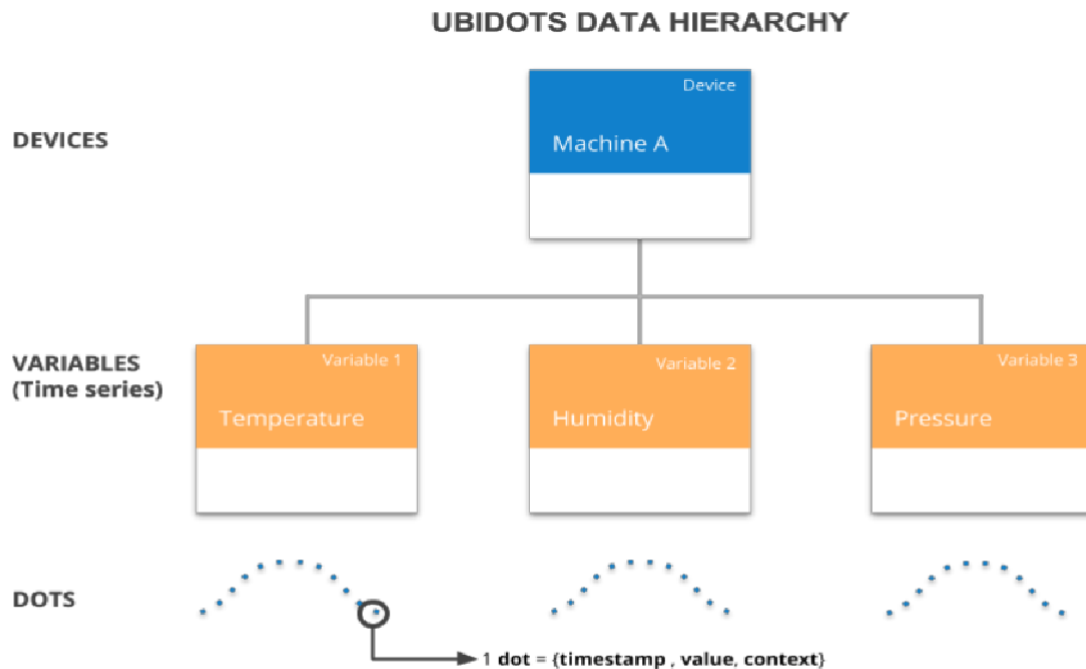
Introduction

Welcome to the hardware documentation for the Ubidots REST API. You'll learn how to send and receive data from your hardware devices using the HTTP, MQTT, and TCP/UDP communication protocols that we support.

The aim of this section is to explain what happens behind the scenes when interacting with Ubidots so that you can reproduce it in your firmware. As a result, we stop using examples that use libraries. Check out our compatible devices and API clients pages if you want to use one of our libraries (Python, C, and more).

How Ubidots Works?

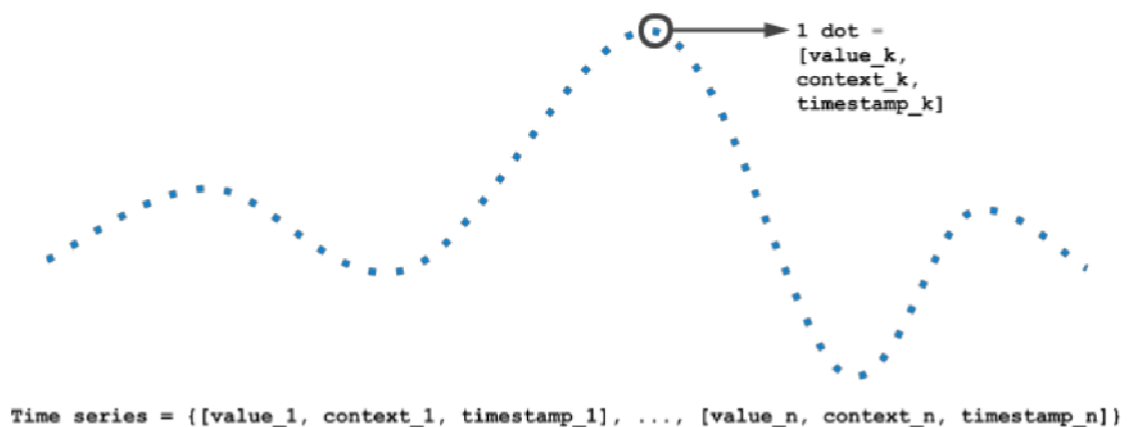
A data-point or "dot" is generated every time a system updates a sensor value in a variable. Ubidots stores dots from your devices within variables and assigns timestamps to them:



Each dot contains these items:

Item	Description	Mandatory
value	A monetary value. Ubidots accepts numbers of up to 16 floating-point length	YES
timestamp	Period in milliseconds after the Unix epoch. If you don't specify, our servers will allocate one to you when you arrive.	NO
context	An unrestricted set of key-value pairs. GPS devices' latitude and longitude coordinates are commonly stored in this format.	NO

UBIDOTS TIME SERIES



Ubidots is a platform agnostic, which means it doesn't care what hardware interface you're using as long as you can communicate with us using at least one of these protocols:

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

HTTP

HTTP is the most widely used internet protocol for data transfer. It also supports data encryption, also known as stable HTTP or HTTPS. You submit and receive packages using HTTP or HTTPS when you use a web browser (MS Edge, Firefox, Google Chrome, and so on). As a result, it's no surprise that HTTPS has become a common method of data sharing in the IoT.

When a developer wants to use HTTP, he should look for a RESTful Application Programming Interface, or REST API, that exposes all of the endpoints needed to interact with a third-party application (like Ubidots). You can use our REST API to learn how to send a message in this portion.

Within the HTTP norm, the following methods are defined:

HTTP Method	Description
GET	Information is retrieved using this method.
POST	It's used to make a new element.
PATCH	Used to make changes to existing elements
DELETE	To delete existing components, use this command.

HTTP is a request/response protocol, which means that the server responds to each request you make. A number (answer code) and a body are included in this response. A GET request is created when you make a request to download a file from a website (e.g. "Get me the file 'website.html'"). If the request is valid, the server will usually respond with a 200 response code and the requested file (body).

When a computer sends a GET request to the Ubidots server (i.e., requesting the last value of a variable), the server responds with the status of the request (response code) and a body, which is the value with its meaning and timestamp

The following parameters are also needed for an HTTP request:

- **Host:** This specifies the server to which HTTP requests will be sent.
- **Path:** The path determines the resource you want to consume, whether it's a vector or a device, and is usually the last part of the URL. The route will be `/api/v1.6/devices/my-device` if the API endpoint was `industrial.api.ubidots.com/api/v1.6/devices/my-device`.
- **Headers:** Define the HTTP request's operating parameters, such as authentication, Content-Type, Content-Length, and so on.
- **Body/payload:** This is the data submitted by your computer to the server in POST and PATCH requests. Since GET requests are used to request data rather than submit data, they usually don't have a body.

Ubidots accepts data in the form of JSON or JavaScript Object Notation. JSON is a list of name/value pairs and is a popular HTTP data form. This is referred to as an object, record, struct, dictionary, hash table, keyed list, or associative array in various programming languages. It's also readable by humans and language-independent. The following is an example of a JSON data form that Ubidots accepts:

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

The following is an example of an HTTP request to Ubidots:

```
HTTP/1.1CR>LN> POST PATH HTTP/1.1CR>LN>
```

```
HOSTCR>LN>HOSTCR>LN>HOSTCR>LN>HOSTCR>LN>HOSTCR>LN>HOST
```

```
USER AGENTCR>LN>USER AGENTCR>LN>USER AGENTCR>LN>USER  
AGENTCR>LN>USER AG
```

Where:

- **{PATH}:** Path to the resource to consume
Example: `/api/v1.6/variables/` to get user's variables information.

Send data

You can use our REST API to send data to the platform in two ways:

- Use a computer to send data to
- Use a variable to send data to

Please see our Ubidots Software REST API Reference for more advanced API features, such as editing a device name or listing all of your devices.

Send data to a device

Request structure:

```
HTTP/1.1CR>LN> POST /api/v1.6/devices/DEVICE LABEL
```

```
CR>LN>HostCR>LN>HostCR>LN>HostCR>LN>HostCR>LN>HostCR
```

```
USER AGENTCR>LN>USER AGENTCR>LN>USER AGENTCR>LN>USER  
AGENTCR>LN>USER AG
```

```
X-Auth-Token: TOKENCR>LN> X-Auth-Token: TOKENCR>LN> X-Auth-Token:  
TOKENCR>LN
```

```
LN>Content-Type: application/jsonCR>LN>Content-Type:  
application/jsonCR>LN>Content-Type: application
```

PAYLOAD

```
LENGTHCR>LN>CR>LN>CR>LN>CR>LN>CR>LN>CR>LN>CR>LN>CR>LN>CR>  
LN>CR>LN>CR>LN
```

PAYLOAD is a phrase that means "payload."

```
LN>CR>CR>LN>LN>LN>LN>LN>L
```

- **Multiple dots:** {"temperature": {"value": 27, "timestamp": 1514808000000}, "humidity": 55, "pressure": {"value": 78, "context":{"name": "John"}}}

POST

https://industrial.api.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}

POST

https://things.ubidots.com/api/v1.6/devices/{DEVICE_LABEL}

Send a single dot (Educational users)

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"temperature": 27}' https://things.ubidots.com
```

/api/v1.6/devices/my-new-device

Send a single dot (Industrial users)

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"temperature": 27}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device
```

ots.com/api/v1.6/devices/my-new-device

Send a dot with coordinates

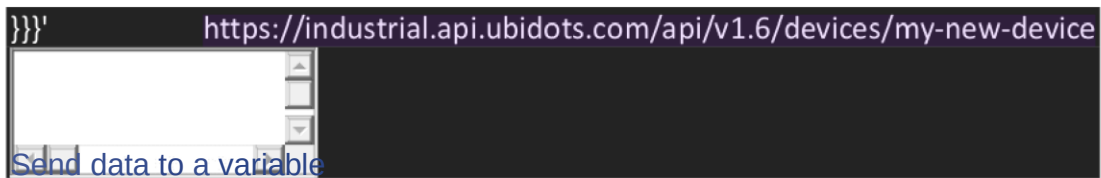
```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"position": {"value": 1, "timestamp": 1514808000000, "context":{"lat":-6.2, "lng":75.4}}}' https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device
```

v1.6/devices/my-new-device

Send multiple dots

```
curl -X POST -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" -H "Content-Type: application/json" -d '{"temperature": {"value": 27, "timestamp": 1514808000000}, "humidity": 55, "pressure": {"value": 78, "context":{"name": "John"}}

```



You can also send one or multiple dots to a single variable.

Request structure:

```
POST /api/v1.6/devices/(DEVICE_LABEL)/(VARIABLE_LABEL)/values 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/1.1 201 Created<CR><LN>
```

```
Server: nginx<CR><LN>
```

```
Date: Tue, 04 Sep 2018 22:50:55 GMT<CR><LN>
```

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

```
Transfer-Encoding: chunked<CR><LN>
```

```
Vary: Accept, Cookie<CR><LN>
```

```
Location: http://industrial.api.ubidots.com/api/v1.6/values/(VARIABLE_ID)<CR><LN>
```

```
Allow: GET, POST, HEAD, OPTIONS<CR><LN><CR><LN>
```

```
(PAYLOAD_LENGTH_IN_HEXADECIMAL)<CR><LN>
```

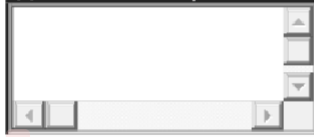
```
("url": "http://things.ubidots.com/api/v1.6/values/(VARIABLE_ID}", "value": 27.0,
  "timestamp": 1536101455822, "context": {}, "created_at": "2018-09-04T22:50:55.
822035Z"}<CR><LN>
```

```
0<CR><LN>
```

In the *path* specified above, replace the (DEVICE_LABEL) and (VARIABLE_LABEL) keys with the unique labels of the U bidots device and variable you wish to send data to. If the specified device and/or variable do not exist, they will be automatically created.

Example Request

```
curl -X GET -H "X-Auth-Token: BBFF-Rfcgaxns6HIVb155WA0RhSY85xNDmB" "https://industrial.api.ubidots.com/api/v1.6/devices/my-new-device/control/lv"
```



Response codes

When you make an HTTP request to the Ubidots REST API, you will receive the following answer codes, which will tell you the status of your request as well as any errors that may have occurred:

Code	Meaning
200	Okay, the request was successful.
201	Generated — The request was successful, and an object (device or variable) was created as a result.
202	Authorized — While the request has been approved for processing, it has not yet been completed.
204	The request is not saved because one of the fields is incorrect — Please double-check that it's a proper JSON string and that the fields are what the endpoint expects (string, object or float).
400	Bad Request — An invalid body in your request caused an error. Please double-check that it's a proper JSON string and that the fields are what the endpoint expects (string, object or float).
401	API key is invalid — Please double-check your API Key.
402	Please double-check your balance before making a payment.

Code	Meaning
403	Prohibited — This token is no longer in use. Please double-check your token
404	unable to locate — We were unable to locate the URL you requested. This may be caused by an incorrect system label or ID, an incorrect variable label or ID, or simply a typo in the request's URL.
405	Method Not Allowed — The method used isn't supported by this API endpoint. The methods that are enabled can be found in our API documentation.
415	Unsupported media type — The payload on the target resource is in a format that this tool does not support.
420	You've used up all of your API restrictions — Contact the Ubidots team to update your API limits.
423	The system is not receiving data because it is turned off.
429	Too many requests — A large number of requests are submitted in a short period of time ("rate limiting")
50*	. Error Internal — Our servers are having problems. Please visit our status page for more information.

MQTT

MQTT is a connectivity protocol for the "Internet of Things." It was created to be a very light-weight publish/subscribe messaging system. It's useful for remote connections where a small code footprint is needed and/or network bandwidth is limited."

ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQflmwRQW|weather-station=>
ESP8266/1.0|POST|p4uuT2O25AJwv7ncADLoVhQflmwRQW
end|temperature:20,humidity:35

References:

- 1.J. Madrazo; A. Clappier; O. Cuesta; L.C. Belalcázar; Y. González In Havana, proof of traffic-related air pollution. *Atmósfera (atmosphere)* 32, 15–24, 2019. [CrossRef] [CrossRef] [CrossRef]
2. H. Qu, X. Lu, L. Liu, and Y. Ye. Effects of traffic and urban parks on PM10 and PM2 levels in the environment There are five mass concentrations. Part A: Energy Sources Recuperation 1–13 in Util. Environ. Eff. [CrossRef] [CrossRef] [CrossRef]
3. C.M. Kendrick, P. Koonce, and L.A. George NO, NO2, and PM2 levels vary across the day and season. Along an urban arterial, 5 mass as a result of traffic volumes. *Atmos. Environ.*, 122, 133–141, 2015. [CrossRef]
4. ISPRA Air Quality and Environment Infographic http://www.isprambiente.gov.it/files2020/pubblicazioni/stato-ambiente/annuario-2020/Infografiche_r.pdf (accessed on 3 September 2020).
- 5.The World Health Organization is number five. World Health Organization, Geneva, Switzerland, 2016. *Ambient Air Pollution: A Global Assessment of Exposure and Disease Burden*.
- 6.The United Nations is number six. The United Nations Sustainable Development Goals Report 2020; New York, NY, USA, 2020.
- 7.The European Environment Agency is number seven. European Environment Agency, Copenhagen, Denmark, 2019. *Air Quality in Europe—2019 Report*; European Environment Agency, Copenhagen, Denmark, 2019.

report

ORIGINALITY REPORT

19%

SIMILARITY INDEX

10%

INTERNET SOURCES

0%

PUBLICATIONS

18%

STUDENT PAPERS

PRIMARY SOURCES

1	www.ubidots.com Internet Source	5%
2	essay.utwente.nl Internet Source	1%
3	ubidots.com Internet Source	1%
4	lamipc54.epfl.ch Internet Source	1%
5	www.electronicwings.com Internet Source	1%
6	en.wikipedia.org Internet Source	1%
7	Submitted to SVKM International School Student Paper	1%
8	Submitted to Engineers Australia Student Paper	1%
9	ijitech.org Internet Source	1%

10	Submitted to Higher Education Commission Pakistan Student Paper	1%
11	George Mois, Silviu Folea, Teodora Sanislav. "Analysis of Three IoT-Based Wireless Sensors for Environmental Monitoring", IEEE Transactions on Instrumentation and Measurement, 2017 Publication www.powershow.com Internet Source	1%
12	Submitted to University of Moratuwa Student Paper	1%
13	baadalsg.inflibnet.ac.in Internet Source	1%
14	Submitted to Universiti Teknologi MARA Student Paper	1%
15	Submitted to Visvesvaraya Technological University Student Paper	<1%
16	www.slideshare.net Internet Source	<1%
17	www.ijatir.org Internet Source	<1%
18		<1%

19	Submitted to Softwarica College Of IT & E-Commerce Student Paper	<1%
20	www.elprocus.com Internet Source	<1%
21	Submitted to University of Computer Studies Student Paper	<1%
22	Jean-Paul Kouma, Li Liu. "Internet of Food", 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing, 2011 Publication	<1%
23	Submitted to Republic Polytechnic Student Paper	<1%
24	Submitted to Institute of Research & Postgraduate Studies, Universiti Kuala Lumpur Student Paper	<1%
25	Submitted to Segi University College Student Paper	<1%
26	Submitted to Universiti Teknikal Malaysia Melaka Student Paper	<1%
	mapmet.com Internet Source	
27		<1%

28	www.oki.com Internet Source	<1%
29	Submitted to Siddaganga Institute of Technology Student Paper	<1%
30	Submitted to Auston Institute of Management and Technology Student Paper	<1%
31	Submitted to Management Development Institute Of Singapore Student Paper	<1%
32	Submitted to Manchester Metropolitan University Student Paper	<1%
33	ijset.com InternetSource	<1%
34	www.arcjournals.org Internet Source	<1%
35	Submitted to Asia Pacific University College of Technology and Innovation (UCTI) Student Paper	<1%
36	Submitted to Nelson Mandela Metropolitan University Student Paper	<1%
37	drrajivdesaimd.com Internet Source	<1%

Exclude quotes On

Exclude matches

< 5 words

Exclude bibliography On
