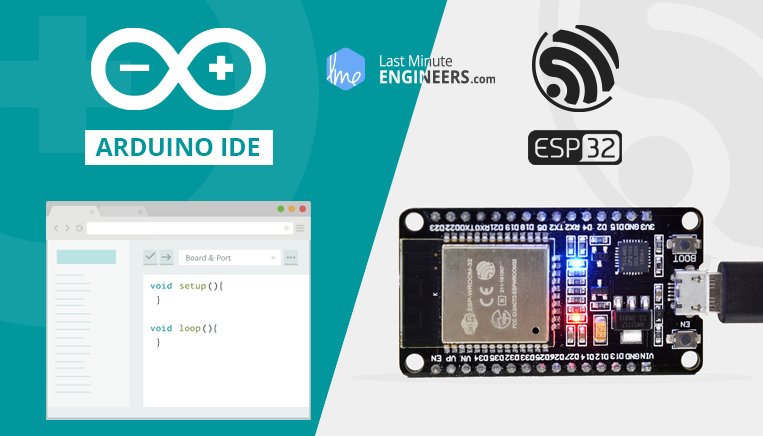
Insight Into ESP32 Features With Arduino IDE



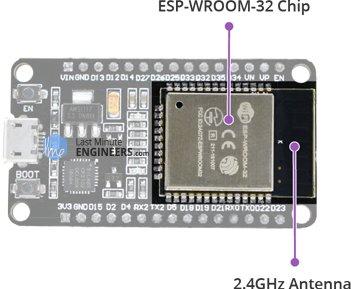
Few years back, ESP8266 took the embedded IoT world by storm. For less than $3, you could get a programmable, WiFi-enabled microcontroller being able to monitor and control things from anywhere in the world. Now [Espressif](https://www.espressif.com/) (The semiconductor company behind the ESP8266) has released a perfect super-charged upgrade: the ESP32. Being successor to ESP8266; not only does it have a WiFi support, but it also features Bluetooth 4.0 (BLE/Bluetooth Smart) – perfect for just about any IoT project.

ESP-WROOM-32 Module

The development board equips the ESP-WROOM-32 module containing Tensilica Xtensa® Dual-Core 32-bit LX6 microprocessor. This processor is similar to the ESP8266 but has two CPU cores (can be individually controlled), operates at 80 to 240 MHz adjustable clock frequency and performs at up to 600 DMIPS (Dhrystone Million Instructions Per Second).

ESP-WROOM-32 Chip

* Xtensa® Dual-Core 32-bit LX6
* Upto 240MHz Clock Freq.
* 520kB internal SRAM
* 4MB external flash
* 802.11b/g/n Wi-Fi transceiver
* Bluetooth 4.2/BLE



There’s also 448 KB of ROM, 520 KB of SRAM and 4MB of Flash memory (for program and data storage) just enough to cope with the large strings that make up web pages, JSON/XML data, and everything we throw at IoT devices nowadays.

The ESP32 Integrates 802.11b/g/n HT40 Wi-Fi transceiver, so it can not only connect to a WiFi network and interact with the Internet, but it can also set up a network of its own, allowing other devices to connect directly to it. The ESP32 supports [WiFi Direct](https://en.wikipedia.org/wiki/Wi-Fi_Direct) as well, which is a good option for peer-to-peer connection without the need of an access point. The WiFi Direct is easier to setup and the data transfer speeds are much better than Bluetooth.

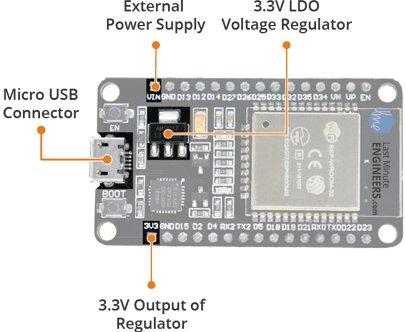
The chip also has dual mode Bluetooth capabilities, meaning it supports both Bluetooth 4.0 (BLE/Bluetooth Smart) and Bluetooth Classic (BT), making it even more versatile.

Power Requirement

As the operating voltage range of ESP32 is 2.2V to 3.6V, the board comes with a LDO voltage regulator to keep the voltage steady at 3.3V. It can reliably supply up to 600mA, which should be more than enough when ESP32 pulls as much as [250mA during RF transmissions](https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/). The output of the regulator is also broken out to one of the sides of the board and labeled as 3V3. This pin can be used to supply power to external components.

Power Requirement

* Operating Voltage: 2.2V to 3.6V
* On-board 3.3V 600mA regulator
* 5 µA during Sleep Mode
* 250mA during RF transmissions



Power to the ESP32 development board is supplied via the on-board MicroB USB connector. Alternatively, if you have a regulated 5V voltage source, the VIN pin can be used to directly supply the ESP32 and its peripherals.

Also the sleep current of the ESP32 chip is less than 5 µA, making it suitable for battery powered and wearable electronics applications.

Warning:

The ESP32 requires a 3.3V power supply and 3.3V logic levels for communication. The GPIO pins are not 5V-tolerant! If you want to interface the board with 5V (or higher) components, you’ll need to do some level shifting.

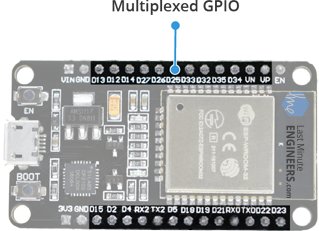
Peripherals and I/O

Although the ESP32 has total 48 GPIO pins, only 25 of them are broken out to the pin headers on both sides of the development board. These pins can be assigned to all sorts of peripheral duties, including:

* 15 ADC channels – 15 channels of 12-bit SAR ADC’s. The ADC range can be set, in firmware, to either 0-1V, 0-1.4V, 0-2V, or 0-4V
* 2 UART interfaces – 2 UART interfaces. One is used to load code serially. They feature flow control, and support IrDA too!
* 25 PWM outputs – 25 channels of PWM pins for dimming LEDs or controlling motors.
* 2 DAC channels – 8-bit DACs to produce true analog voltages.
* SPI, I2C & I2S interface – There are 3 SPI and 1 I2C interfaces to hook up all sorts of sensors and peripherals, plus two I2S interfaces if you want to add sound to your project.
* 9 Touch Pads – 9 GPIOs feature capacitive touch sensing.

Multiplexed I/Os

* 15 ADC channels
* 2 UART interfaces
* 25 PWM outputs
* 2 DAC channels
* SPI, I2C & I2S interface
* 9 Touch Pads



Thanks to the ESP32’s pin multiplexing feature (Multiple peripherals multiplexed on a single GPIO pin). Meaning a single GPIO pin can act as an ADC input/DAC output/Touch pad.

Input Only GPIOs

Pin D34, D35, VP and VN cannot be configured as outputs, but they can be used as either digital inputs, analog inputs, or for other unique purposes. Also note that they do not have internal pull-up or pull-down resistors, like the other GPIO pins.

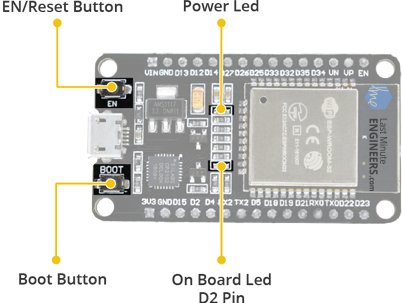
Also GPIO pins VP and VN are an integral part of the ultra-low-noise pre-amplifier for the ADC, which help to configure the sampling time and noise of the pre-amp.

On-board Switches & LED Indicators

The ESP32 development board features two buttons. One marked as EN located on the top left corner is the Reset button, used of course to reset the ESP32 chip. The other Boot button on the bottom left corner is the download button used while downloading the new sketch/programs.

Switches & Indicators

* EN – Reset the ESP32 chip
* Boot – Download new programs
* Red LED – Power Indicator
* Blue LED – User Programmable



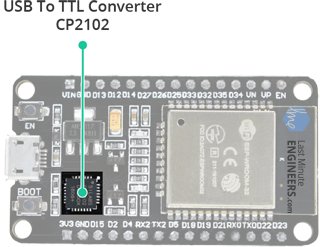
The board also has 2 LED indicators viz. Red LED & Blue LED. A Red LED indicates that the board is powered up and has 3.3V from the regulator. The Blue LED is user programmable and is connected to the D2 pin of the board.

Serial Communication

The board includes CP2102 USB-to-UART Bridge Controller from [Silicon Labs](http://www.silabs.com/), which converts USB signal to serial and allows your computer to program and communicate with the ESP32 chip.

Serial Communication

* CP2102 USB-to-UART converter
* 5 Mbps communication speed
* IrDA support

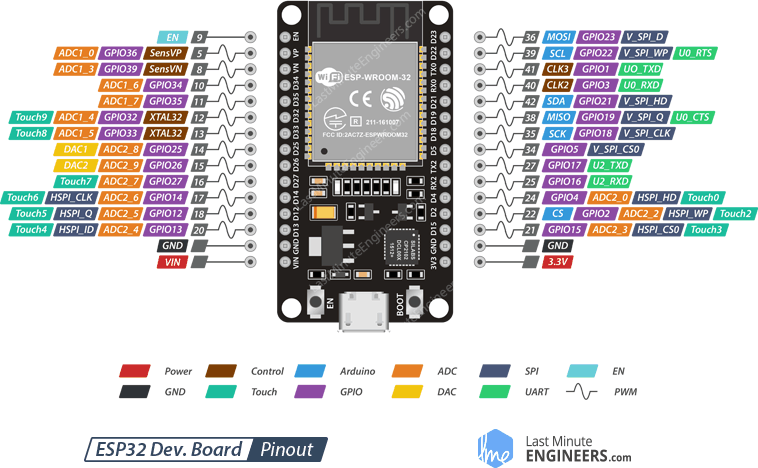


If you have an older version of CP2102 driver installed on your PC, we recommend upgrading now.

[CP2102 Driver](https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers)

ESP32 Development Board Pinout

The ESP32 development board has total 30 pins that interface it to the outside world. The connections are as follows:



For the sake of simplicity, we will make groups of pins with similar functionalities.

Power Pins There are two power pins viz. VIN pin & 3.3V pin. The VIN pin can be used to directly supply the ESP32 and its peripherals, if you have a regulated 5V voltage source. The 3.3V pin is the output of an on-board voltage regulator. This pin can be used to supply power to external components.

GND is a ground pin of ESP32 development board.

Arduino Pins are nothing but ESP32’s hardware I2C and SPI pins to hook up all sorts of sensors and peripherals in your project.

GPIO Pins ESP32 development board has 25 GPIO pins which can be assigned to various functions programmatically. Each digital enabled GPIO can be configured to internal pull-up or pull-down, or set to high impedance. When configured as an input, it can also be set to edge-trigger or level-trigger to generate CPU interrupts.

ADC Channels The board integrates 12-bit SAR ADCs and supports measurements on 15 channels (analog enabled pins). Some of these pins can be used to build a programmable gain amplifier which is used for the measurement of small analog signals. The ESP32 is also designed to measure the voltages while operating in the sleep mode.

DAC Channels The board features two 8-bit DAC channels to convert digital signals into true analog voltages. This dual DAC can drive other circuits.

Touch Pads The board offers 9 capacitive sensing GPIOs which detect capacitive variations introduced by the GPIO’s direct contact or close proximity with a finger or other objects.

UART Pins ESP32 development board has 2 UART interfaces, i.e. UART0 and UART2, which provide asynchronous communication (RS232 and RS485) and IrDA support, and communicate at up to 5 Mbps. UART provides hardware management of the CTS and RTS signals and software flow control (XON and XOFF) as well.

SPI Pins SPI Pins ESP32 features three SPIs (SPI, HSPI and VSPI) in slave and master modes. These SPIs also support the following general-purpose SPI features:

* 4 timing modes of the SPI format transfer
* Up to 80 MHz and the divided clocks of 80 MHz
* Up to 64-Byte FIFO

All SPIs can also be used to connect to the external Flash/SRAM and LCD.

~ PWM Pins The board has 25 channels (Nearly All GPIO pins) of PWM pins controlled by Pulse Width Modulation (PWM) controller. The PWM output can be used for driving digital motors and LEDs. The controller consists of PWM timers and the PWM operator. Each timer provides timing in synchronous or independent form, and each PWM operator generates the waveform for one PWM channel.

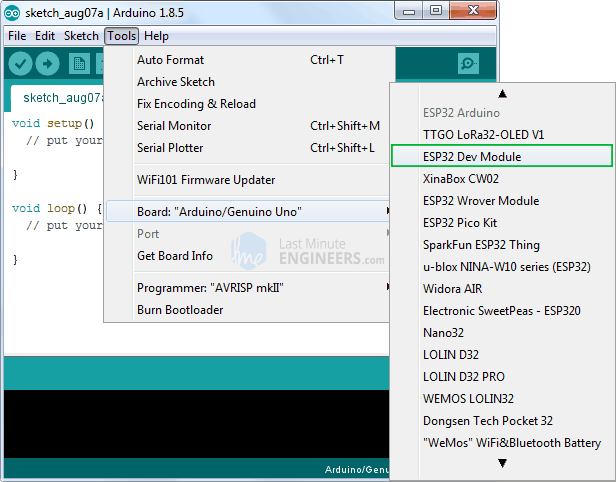
EN Pin is used to enable ESP32. The chip is enabled when pulled HIGH. When pulled LOW the chip works at minimum power.

Arduino Example: Blink

To make sure ESP32 Arduino core and the ESP32 development board are properly set up, we’ll upload the simplest sketch of all – The Blink!

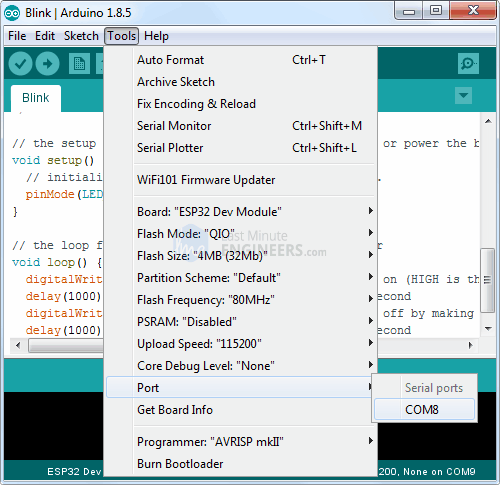
We will use the on-board LED for this test. As mentioned earlier in this tutorial, D2 pin of the board is connected to on-board Blue LED & is user programmable. Perfect!

Before we get to uploading sketch & playing with LED, we need to make sure that the board is selected properly in Arduino IDE. Open Arduino IDE and select ESP32 Dev Module option under your Arduino IDE > Tools > Board menu.



Now, plug your ESP32 development board into your computer via micro-B USB cable. Once the board is plugged in, it should be assigned a unique COM port. On Windows machines, this will be something like COM#, and on Mac/Linux computers it will come in the form of /dev/tty.usbserial-XXXXXX. Select this serial port under the Arduino IDE > Tools > Port menu.

Also the upload speed is selected to 921600 by default. Try lowering it to Upload Speed : 115200 as many users complained about getting espcomm\_sync failed error when trying to upload the sketch at 921600 speed.



Once you are done, try the example sketch below.

int ledPin = 2;

void setup()

{

pinMode(ledPin, OUTPUT);

}

void loop()

{

digitalWrite(ledPin, HIGH);

delay(500);

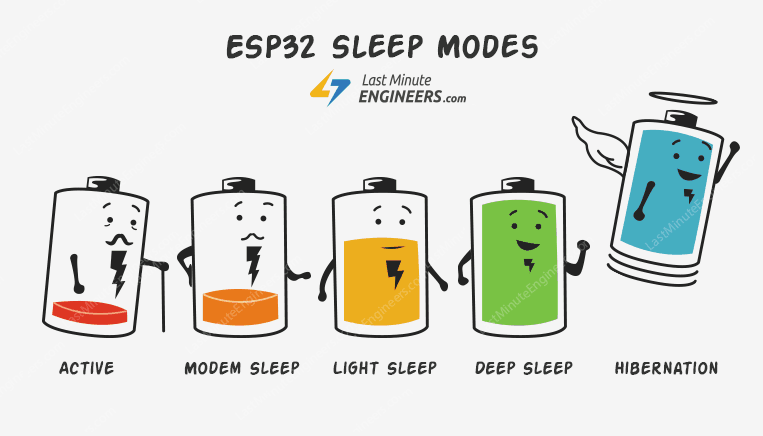
digitalWrite(ledPin, LOW);

delay(500);

}

Once the code is uploaded, LED will start blinking. You may need to tap the EN button to get your ESP32 to begin running the sketch.

Insight Into ESP32 Sleep Modes & Their Power Consumption



There is no question that ESP32 is a worthy competitor to many WiFi/MCU SoCs out there, often beating it on both performance and price. But, depending on which state it’s in, the ESP32 can be a relatively power-hungry device.

When your IoT project is powered by a plug in the wall, you tend not to care too much about power consumption. But if you are going to power your project by batteries, every mA counts.

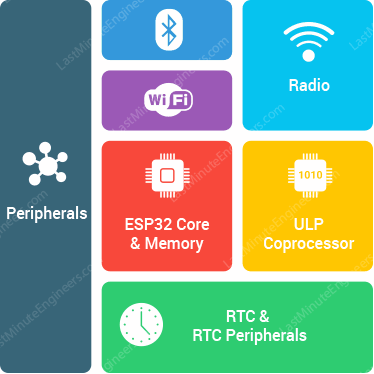
The solution here is to cut back ESP32’s power usage by leveraging one of its Sleep Modes. It’s really a great strategy for dramatically extending the battery life of a project that doesn’t need to be active all the time.

What is ESP32 sleep mode?

ESP32 Sleep mode is a power-saving state that ESP32 can enter when not in use. The ESP32’s state is maintained in RAM. When ESP32 enters sleep mode, power is cut to any unneeded digital peripherals, while RAM receives just enough power to enable it to retain its data.

Inside ESP32 chip

In order to understand how ESP32 achieves power saving, we need to know what’s inside the chip. The following illustration shows function block diagram of ESP32 chip.



At the heart of the ESP32 chip is a Dual-Core 32-bit microprocessor along with 448 KB of ROM, 520 KB of SRAM and 4MB of Flash memory.

It also contains WiFi module, Bluetooth Module, Cryptographic Accelerator (a co-processor designed specifically to perform cryptographic operations), the RTC module, and lot of peripherals.

ESP32 Power Modes

Thanks to the ESP32’s advanced power management, it offers 5 configurable power modes. As per the power requirement, the chip can switch between different power modes. The modes are:

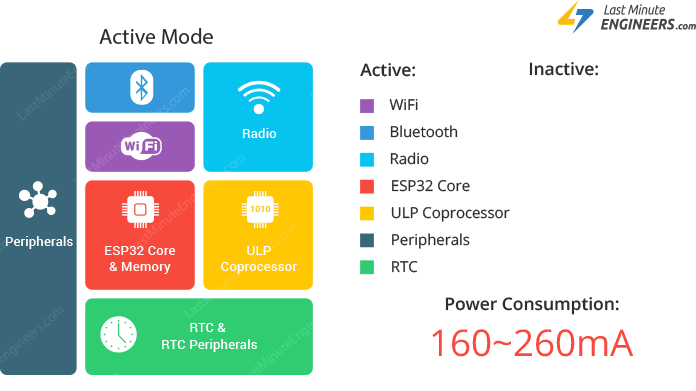
* Active Mode
* Modem Sleep Mode
* Light Sleep Mode
* Deep Sleep Mode
* Hibernation Mode

Each mode has its own distinct features and power saving capabilities. Let’s look in to them one by one.

ESP32 Active Mode

The normal mode is also known as Active Mode. In this mode all the features of the chip are active.

As the active mode keeps everything (especially the WiFi module, the Processing Cores and the Bluetooth module) ON at all times, the chip requires more than 240mA current to operate. Also we observed that if you use both WiFi and Bluetooth functions together, sometimes high power spikes appear (biggest was 790mA).



If you look at the [ESP32 datasheet](https://lastminuteengineers.com/datasheets/esp32-datasheet-en.pdf), power consumption during Active power mode, with RF working is as follows:

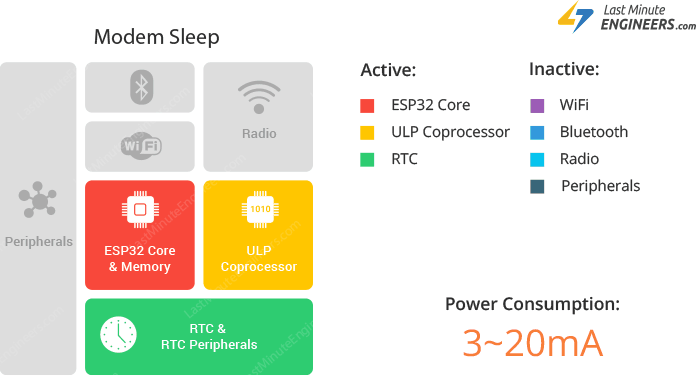
|  |  |
| --- | --- |
| Mode | Power Consumption |
| Wi-Fi Tx packet 13dBm~21dBm | 160~260mA |
| Wi-Fi/BT Tx packet 0dBm | 120mA |
| Wi-Fi/BT Rx and listening | 80~90mA |

Obviously, this is the most inefficient mode and will drain the most current. So, if we want to conserve power we have to disable them (by leveraging one of the other power modes) when not in use.

ESP32 Modem Sleep

In modem sleep mode everything is active while only WiFi, Bluetooth and radio are disabled. The CPU is also operational and the clock is configurable.

In this mode the chip consumes around 3mA at slow speed and 20mA at high speed.



To keep WiFi/Bluetooth connections alive, the CPU, Wi-Fi, Bluetooth, and radio are woken up at predefined intervals. It is known as Association sleep pattern.

During this sleep pattern, the power mode switches between the active mode and Modem sleep mode.

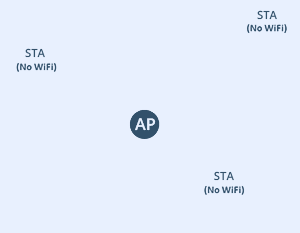
ESP32 can enter modem sleep mode only when it connects to the router in station mode. ESP32 stays connected to the router through the DTIM beacon mechanism.

In order to save power, ESP32 disables the Wi-Fi module between two DTIM Beacon intervals and wakes up automatically before the next Beacon arrival.

The sleep time is decided by the DTIM Beacon interval time of the router which is usually 100ms to 1000ms.

What is DTIM beacon mechanism?

DTIM is acronym for Delivery Traffic Indication Message.



In this mechanism, the access point(AP)/router transmits beacon frames periodically. Each frame contains all the information about the network. It is used to announce the presence of a wireless network and synchronize all the connected members.

ESP32 Light Sleep

The working mode of light sleep is similar to that of modem sleep. The chip also follows association sleep pattern.

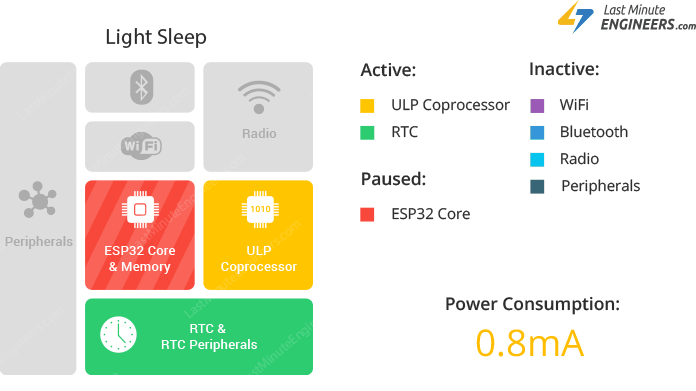
The difference is that, during light sleep mode, digital peripherals, most of the RAM and CPU are clock-gated.

What is Clock Gating?

Clock gating is a technique for reducing the dynamic power consumption.

It disables portions of the circuitry by powering off clock pulses, so that the flip-flops in them do not have to switch states. As switching states consumes power, when not being switched, the power consumption goes to zero.

During light sleep mode, the CPU is paused by powering off its clock pulses, while RTC and ULP-coprocessor are kept active. This results in less power consumption than in modem sleep mode which is around 0.8mA.



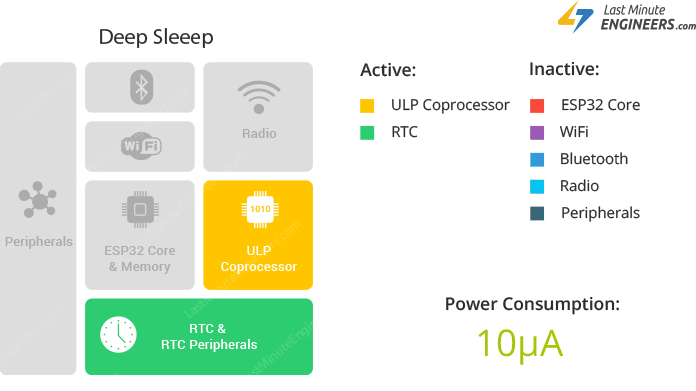
Before entering light sleep mode, ESP32 preserves its internal state and resumes operation upon exit from the sleep. It is known Full RAM Retention.

esp\_light\_sleep\_start() function can be used to enter light sleep once wake-up sources are configured.

ESP32 Deep Sleep

In deep sleep mode, the CPU, most of the RAM and all the digital peripherals are powered off. The only parts of the chip that remains powered on are: RTC controller, RTC peripherals (including ULP co-processor), and RTC memories (slow and fast).

The chip consumes around 0.15 mA(if ULP co-processor is powered on) to 10µA.



During deep sleep mode, the main CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system, based on the measured data from sensors. This sleep pattern is known as ULP sensor-monitored pattern.

Along with the CPU, the main memory of the chip is also disabled. So, everything stored in that memory is wiped out and cannot be accessed.

However, the RTC memory is kept powered on. So, its contents are preserved during deep sleep and can be retrieved after we wake the chip up. That’s the reason, the chip stores Wi-Fi and Bluetooth connection data in RTC memory before disabling them.

So, if you want to use the data over reboot, store it into the RTC memory by defining a global variable with RTC\_DATA\_ATTR attribute. For example, RTC\_DATA\_ATTR int bootCount = 0;

In Deep sleep mode, power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset. This means program execution starts from the beginning once again.

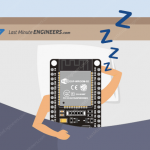
TIP

ESP32 supports running a [deep sleep wake stub](https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/deep-sleep-stub.html) when coming out of deep sleep. This function runs immediately as soon as the chip wakes up – before any normal initialization, bootloader code has run. After the wake stub runs, the chip can go back to sleep or continue to start normally.

Unlike the other sleep modes, the system cannot go into Deep-sleep mode automatically. esp\_deep\_sleep\_start() function can be used to immediately enter deep sleep once wake-up sources are configured.

By default, ESP32 will automatically power down the peripherals not needed by the wake-up source. But you can optionally decide what all peripherals to shut down/keep on. For more information, check out [API docs](http://esp-idf.readthedocs.io/en/latest/api-reference/system/deep_sleep.html).

To know more about ESP32 Deep Sleep & its wake-up sources, please visit below tutorial.

[](https://lastminuteengineers.com/esp32-deep-sleep-wakeup-sources/)

[ESP32 Deep Sleep & Its Wake-up Sources](https://lastminuteengineers.com/esp32-deep-sleep-wakeup-sources/)

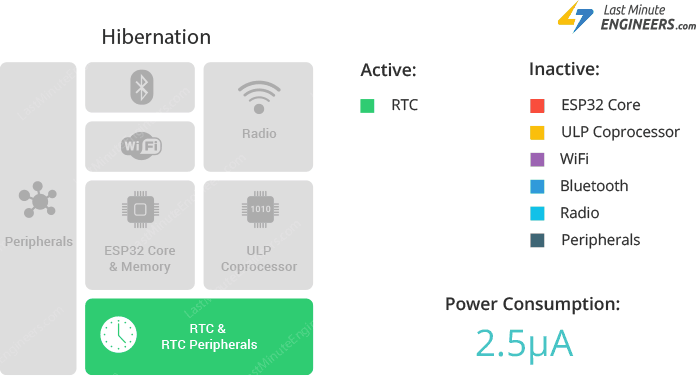
Have you ever wanted your IoT project to last on batteries for almost 5 years? Wait... What? 5 years? Yes. It might sound ridiculous, but...

ESP32 Hibernation mode

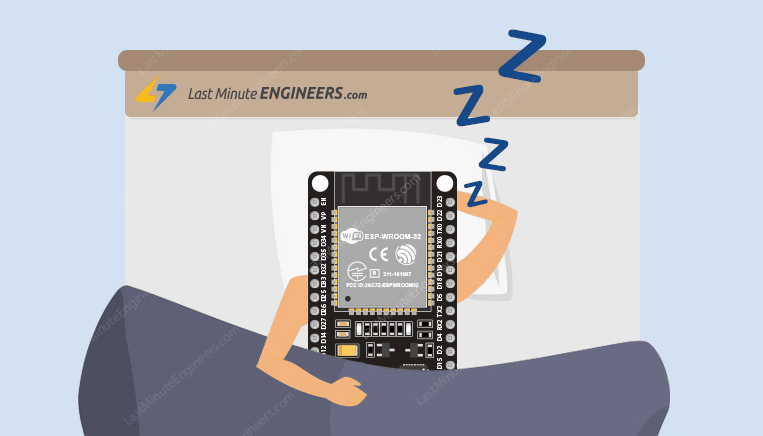
Unlike deep sleep mode, in hibernation mode the chip disables internal 8MHz oscillator and ULP-coprocessor as well. The RTC recovery memory is also powered down, meaning there’s no way we can preserve any data during hibernation mode.

Everything else is shut off except only one RTC timer on the slow clock and some RTC GPIOs are active. They are responsible for waking up the chip from the hibernation mode.

This reduces power consumption even further. The chip consumes around 2.5µA only in hibernation mode.



**ESP32 Deep Sleep & Its Wake-up Sources**



Have you ever wanted your IoT project to last on batteries for almost 5 years? Wait… What? 5 years? Yes. It might sound ridiculous, but this is possible with the ESP32’s deep sleep feature.

Why do ESP32 need deep sleep?

Depending on which state it’s in, the ESP32 can be a relatively power-hungry device. It usually pulls about 75mA in normal operation and hits about 240mA while transmitting data over WiFi.

When your IoT project is powered by a plug in the wall, you tend not to care too much about power consumption. But if you are going to power your project by batteries, every mA counts. The solution here is to cut back ESP32’s power usage by leveraging Deep Sleep Mode.

To know more about other sleep modes of ESP32 and their power consumption, please visit below tutorial.

[](https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/)

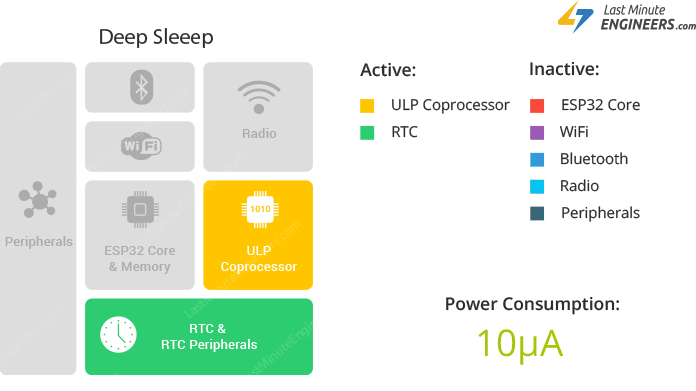
[Insight Into ESP32 Sleep Modes & Their Power Consumption](https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/)

There is no question that ESP32 is a worthy competitor to many WiFi/MCU SoCs out there, often beating it on both performance and price. But,...

ESP32 Deep Sleep

In deep sleep mode, the CPU, most of the RAM and all the digital peripherals are powered off. The only parts of the chip that remains powered on are: RTC controller, RTC peripherals (including ULP co-processor), and RTC memories (slow and fast).

The chip consumes around 0.15 mA(if ULP co-processor is powered on) to 10µA.



During deep sleep mode, the main CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system, based on the measured data from sensors. This sleep pattern is known as ULP sensor-monitored pattern.

Along with the CPU, the main memory of the chip is also disabled. So, everything stored in that memory is wiped out and cannot be accessed.

However, the RTC memory is kept powered on. So, its contents are preserved during deep sleep and can be retrieved after we wake the chip up. That’s the reason, the chip stores Wi-Fi and Bluetooth connection data in RTC memory before disabling them.

So, if you want to use the data over reboot, store it into the RTC memory by defining a global variable with RTC\_DATA\_ATTR attribute. For example, RTC\_DATA\_ATTR int bootCount = 0;

In Deep sleep mode, power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset. This means program execution starts from the beginning once again.

ESP32 supports running a [deep sleep wake stub](https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/deep-sleep-stub.html) when coming out of deep sleep. This function runs immediately as soon as the chip wakes up – before any normal initialization, bootloader code has run. After the wake stub runs, the chip can go back to sleep or continue to start normally.

Unlike the other sleep modes, the system cannot go into Deep-sleep mode automatically. esp\_deep\_sleep\_start() function can be used to immediately enter deep sleep once wake-up sources are configured.

By default, ESP32 will automatically power down the peripherals not needed by the wake-up source. But you can optionally decide what all peripherals to shut down/keep on. For more information, check out [API docs](http://esp-idf.readthedocs.io/en/latest/api-reference/system/deep_sleep.html).

ESP32 Deep Sleep Wake-up sources

Wake up from deep sleep mode can be done using several sources. These sources are:

* Timer
* Touch pad
* External wakeup(ext0 & ext1)

Wake-up sources can be combined, in this case the chip will wake up when any one of the sources is triggered.

These sources can be configured at any moment before entering in to sleep mode.

Warning:

It is also possible to go into deep sleep with no wake-up sources configured, in this case the chip will be in deep sleep mode indefinitely, until external reset is applied.

ESP32 Wake-up Source : Timer

RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time.

Time is specified at microsecond precision, but the actual resolution depends on the clock source selected.

esp\_sleep\_enable\_timer\_wakeup() function can be used to enable deep sleep wake up using a timer.

Here’s a code that demonstrates the most basic deep sleep example with a timer as wake-up source and how to store data in RTC memory to use it over reboots.

#define uS\_TO\_S\_FACTOR 1000000 //Conversion factor for micro seconds to seconds

#define TIME\_TO\_SLEEP 5 //Time ESP32 will go to sleep (in seconds)

RTC\_DATA\_ATTR int bootCount = 0;

void setup(){

Serial.begin(115200);

delay(1000); //Take some time to open up the Serial Monitor

//Increment boot number and print it every reboot

++bootCount;

Serial.println("Boot number: " + String(bootCount));

//Print the wakeup reason for ESP32

print\_wakeup\_reason();

//Set timer to 5 seconds

esp\_sleep\_enable\_timer\_wakeup(TIME\_TO\_SLEEP \* uS\_TO\_S\_FACTOR);

Serial.println("Setup ESP32 to sleep for every " + String(TIME\_TO\_SLEEP) +

" Seconds");

//Go to sleep now

esp\_deep\_sleep\_start();

}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep

void print\_wakeup\_reason(){

esp\_sleep\_wakeup\_cause\_t wakeup\_reason;

wakeup\_reason = esp\_sleep\_get\_wakeup\_cause();

switch(wakeup\_reason)

{

case 1 : Serial.println("Wakeup caused by external signal using RTC\_IO"); break;

case 2 : Serial.println("Wakeup caused by external signal using RTC\_CNTL"); break;

case 3 : Serial.println("Wakeup caused by timer"); break;

case 4 : Serial.println("Wakeup caused by touchpad"); break;

case 5 : Serial.println("Wakeup caused by ULP program"); break;

default : Serial.println("Wakeup was not caused by deep sleep"); break;

}

}

ESP32 Wake-up Source : Touch Pad

RTC IO module contains logic to trigger wake up when a touch sensor interrupt occurs.

You need to conﬁgure the touch pad interrupt before the chip starts deep sleep.

esp\_sleep\_enable\_touchpad\_wakeup() function can be used to enable this wake-up source.

Here’s a code that demonstrates the most basic deep sleep example with a touch as wake-up source and how to store data in RTC memory to use it over reboots.

//Define touch sensitivity. Greater the value, more the sensitivity.

#define Threshold 40

RTC\_DATA\_ATTR int bootCount = 0;

touch\_pad\_t touchPin;

void callback(){

//placeholder callback function

}

void setup(){

Serial.begin(115200);

delay(1000);

//Increment boot number and print it every reboot

++bootCount;

Serial.println("Boot number: " + String(bootCount));

//Print the wakeup reason for ESP32 and touchpad too

print\_wakeup\_reason();

print\_wakeup\_touchpad();

//Setup interrupt on Touch Pad 3 (GPIO15)

touchAttachInterrupt(T3, callback, Threshold);

//Configure Touchpad as wakeup source

esp\_sleep\_enable\_touchpad\_wakeup();

//Go to sleep now

esp\_deep\_sleep\_start();

}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep

void print\_wakeup\_reason(){

esp\_sleep\_wakeup\_cause\_t wakeup\_reason;

wakeup\_reason = esp\_sleep\_get\_wakeup\_cause();

switch(wakeup\_reason)

{

case 1 : Serial.println("Wakeup caused by external signal using RTC\_IO"); break;

case 2 : Serial.println("Wakeup caused by external signal using RTC\_CNTL"); break;

case 3 : Serial.println("Wakeup caused by timer"); break;

case 4 : Serial.println("Wakeup caused by touchpad"); break;

case 5 : Serial.println("Wakeup caused by ULP program"); break;

default : Serial.println("Wakeup was not caused by deep sleep"); break;

}

}

//Function that prints the touchpad by which ESP32 has been awaken from sleep

void print\_wakeup\_touchpad(){

touch\_pad\_t pin;

touchPin = esp\_sleep\_get\_touchpad\_wakeup\_status();

switch(touchPin)

{

case 0 : Serial.println("Touch detected on GPIO 4"); break;

case 1 : Serial.println("Touch detected on GPIO 0"); break;

case 2 : Serial.println("Touch detected on GPIO 2"); break;

case 3 : Serial.println("Touch detected on GPIO 15"); break;

case 4 : Serial.println("Touch detected on GPIO 13"); break;

case 5 : Serial.println("Touch detected on GPIO 12"); break;

case 6 : Serial.println("Touch detected on GPIO 14"); break;

case 7 : Serial.println("Touch detected on GPIO 27"); break;

case 8 : Serial.println("Touch detected on GPIO 33"); break;

case 9 : Serial.println("Touch detected on GPIO 32"); break;

default : Serial.println("Wakeup not by touchpad"); break;

}

}

ESP32 Wake-up Source : External Wake-up

There are two types of external triggers to wake ESP32 up from deep sleep.

* ext0 – Use it when you want to wake-up the chip by one particular pin only.
* ext1 – Use it when you have several buttons for the wake-up.

ext0 External Wake-up Source

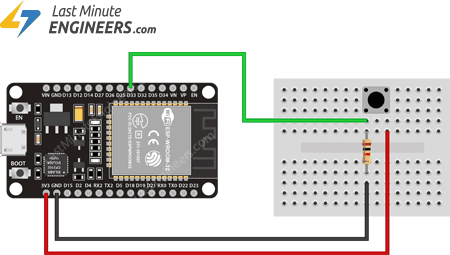
RTC controller contains logic to trigger wake-up when one particular pin is set to a predefined logic level. That pin can be one of RTC GPIOs 0,2,4,12-15,25-27,32-39.

esp\_sleep\_enable\_ext0\_wakeup(GPIO\_PIN,LOGIC\_LEVEL) function can be used to enable this wake-up source. The function takes two parameters. First one is a pin number to which button is connected and second one decides if we want to trigger the wake up by a LOW or a HIGH state of the pin.

ext0 uses RTC IO to wake up, so RTC peripherals will be kept powered ON during deep sleep if this wake-up source is requested.

Because RTC IO module is enabled in this mode, internal pullup or pulldown resistors can also be used. They need to be configured by the application using rtc\_gpio\_pullup\_en() and rtc\_gpio\_pulldown\_en() functions, before calling esp\_sleep\_start().

Below schematic shows how to connect a push button to ESP32 GPIO that serves as a ext0 external wake-up source.



Here’s a code that demonstrates the most basic deep sleep example with a ext0 as wake-up source.

RTC\_DATA\_ATTR int bootCount = 0;

void setup(){

Serial.begin(115200);

delay(1000);

//Increment boot number and print it every reboot

++bootCount;

Serial.println("Boot number: " + String(bootCount));

//Print the wakeup reason for ESP32

print\_wakeup\_reason();

//Configure GPIO33 as ext0 wake up source for HIGH logic level

esp\_sleep\_enable\_ext0\_wakeup(GPIO\_NUM\_33,1);

//Go to sleep now

esp\_deep\_sleep\_start();

}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep

void print\_wakeup\_reason(){

esp\_sleep\_wakeup\_cause\_t wakeup\_reason;

wakeup\_reason = esp\_sleep\_get\_wakeup\_cause();

switch(wakeup\_reason)

{

case 1 : Serial.println("Wakeup caused by external signal using RTC\_IO"); break;

case 2 : Serial.println("Wakeup caused by external signal using RTC\_CNTL"); break;

case 3 : Serial.println("Wakeup caused by timer"); break;

case 4 : Serial.println("Wakeup caused by touchpad"); break;

case 5 : Serial.println("Wakeup caused by ULP program"); break;

default : Serial.println("Wakeup was not caused by deep sleep"); break;

}

}

ext1 External Wake-up Source

ESP32 can be woken up from deep sleep using multiple GPIO pins. Those pins can be one of RTC GPIOs 32-39.

As ext1 wake-up source uses RTC controller, it does’t need RTC peripherals and RTC memories to be powered ON. In this case internal pullup and pulldown resistors will not be available.

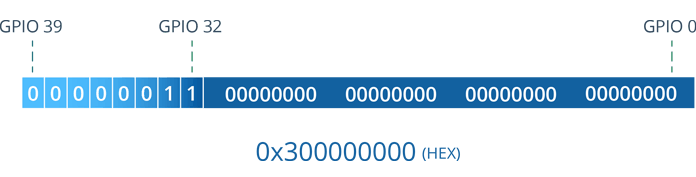
To use internal pullup or pulldown resistors, we need to request RTC peripherals to be kept on during sleep, and conﬁgure pullup/pulldown resistors using rtc\_gpio\_ functions, before entering sleep.

esp\_sleep\_enable\_ext1\_wakeup(BUTTON\_PIN\_MASK,LOGIC\_LEVEL) function can be used to enable this wake-up source. The function takes two parameters. First one is a pin mask to let ESP32 know which all pins we are going to use

The second parameter can be one of the two logic functions can be used to trigger wake-up:

* Wake up if any of the selected pins is HIGH ( ESP\_EXT1\_WAKEUP\_ANY\_HIGH )
* Wake up if all the selected pins are LOW ( ESP\_EXT1\_WAKEUP\_ALL\_LOW )

The easiest way to understand this pin masking technique is to write it in a binary format.



* 0 represents masked pins
* 1 represents pins that will be enabled as a wake-up source

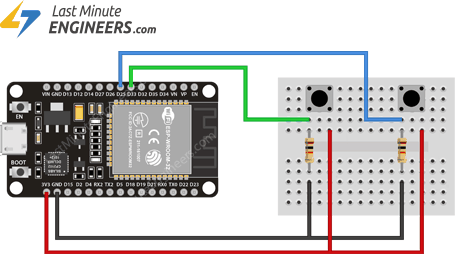
The bit numbering is so simple and based on a normal GPIO numbering. The least significant bit(LSB) represents GPIO0 and the most significant bit(MSB) represents GPIO39.

As the the first pin available is GPIO32, the mask contains 32 times 0 on the right. And then for each enabled pin we write a 1.

If you don’t want to enable any GPIO for wake up you have to write a 0 at its corresponding place.

Once you are done, you need to convert it into HEX before using it as a parameter.

Below schematic shows how to connect multiple push buttons to ESP32 GPIOs that serves as a ext1 external wake-up source.



Here’s a code that demonstrates the most basic deep sleep example with a ext1 as wake-up source.

//Pushbuttons connected to GPIO32 & GPIO33

#define BUTTON\_PIN\_BITMASK 0x300000000

RTC\_DATA\_ATTR int bootCount = 0;

void setup(){

Serial.begin(115200);

delay(1000);

//Increment boot number and print it every reboot

++bootCount;

Serial.println("Boot number: " + String(bootCount));

//Print the wakeup reason for ESP32

print\_wakeup\_reason();

//Configure GPIO32 & GPIO33 as ext1 wake up source for HIGH logic level

esp\_sleep\_enable\_ext1\_wakeup(BUTTON\_PIN\_BITMASK,ESP\_EXT1\_WAKEUP\_ANY\_HIGH);

//Go to sleep now

esp\_deep\_sleep\_start();

}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep

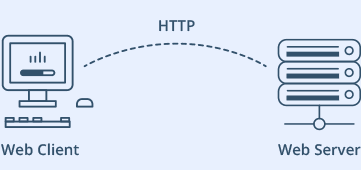
Create A Simple ESP32 Web Server In Arduino IDE



The newly launched successor of ESP8266 – the ESP32 has been a growing star among IoT or WiFi-related projects. It’s an extremely cost-effective WiFi module that – with a little extra effort – can be programmed to build a standalone web server.

What is a Web server and how it works?

Web server is a place which stores, processes and delivers web pages to Web clients. Web client is nothing but a web browser on our laptops and smartphones. The communication between client and server takes place using a special protocol called Hypertext Transfer Protocol (HTTP).



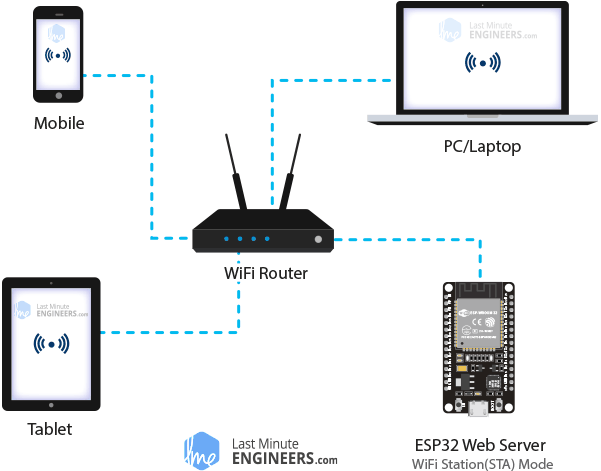
In this protocol, a client initiates communication by making a request for a specific web page using HTTP and the server responds with the content of that web page or an error message if unable to do so (like famous 404 Error). Pages delivered by a server are mostly HTML documents.

ESP32 Operating Modes

One of the greatest features ESP32 provides is that it cannot only connect to an existing WiFi network and act as a Web Server, but it can also set up a network of its own, allowing other devices to connect directly to it and access web pages. This is possible because ESP32 can operate in three different modes: Station mode, Soft Access Point mode, and both at the same time. This provides possibility of building [mesh networks](https://en.wikipedia.org/wiki/Mesh_networking).

Station (STA) Mode

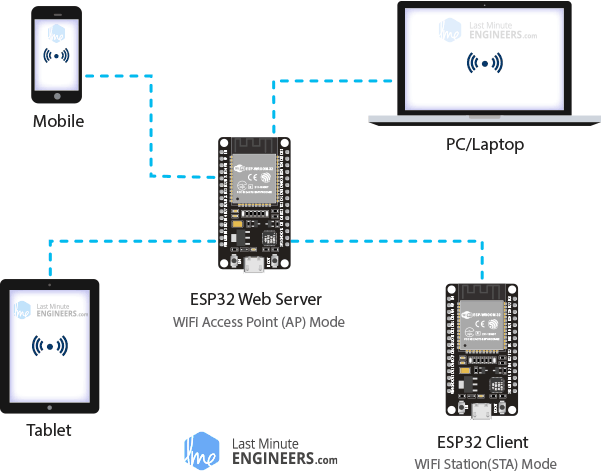
The ESP32 that connects to an existing WiFi network (one created by your wireless router) is called Station (STA)



In STA mode ESP32 gets IP from wireless router to which it is connected. With this IP address, it can set up a web server and deliver web pages to all connected devices under existing WiFi network.

Soft Access Point (AP) Mode

The ESP32 that creates its own WiFi network and acts as a hub (Just like WiFi router) for one or more stations is called Access Point (AP). Unlike WiFi router, it does not have interface to a wired network. So, such mode of operation is called Soft Access Point (soft-AP). Also the maximum number of stations that can connect to it is limited to five.



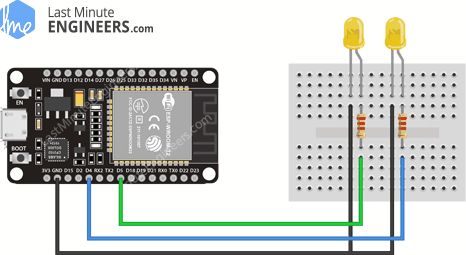
In AP mode ESP32 creates a new WiFi network and sets SSID (Name of the network) and IP address to it. With this IP address, it can deliver web pages to all connected devices under its own network.

Wiring – Connecting LEDs to ESP32

Now that we know the basics of how web server works, and in which modes ESP32 can create a web server, it’s time to connect some LEDs to ESP32 that we want to control over WiFi.

Start by placing the ESP32 on to your breadboard, ensuring each side of the board is on a separate side of the breadboard. Next, connect two LEDs to digital GPIO 4 and 5 through a 220Ω current limiting resistor.

When you’re done you should have something that looks similar to the illustration shown below.

Wiring LEDs to ESP32

Concept Behind Controlling Things From ESP32 Web Server

So, you might be thinking, “How am I going to control things from a web server that merely processes and delivers web pages?” Well, then you need to understand what’s going on behind the scene.

When you type a URL in a web browser and hit ENTER, the browser sends a HTTP request (a.k.a. GET request) to a web server. It’s a job of web server to handle this request by doing something. You might have figured it out by now that we are going to control things by accessing a specific URL. For example, suppose we entered a URL like http://192.168.1.1/ledon in a browser. The browser then sends a HTTP request to ESP32 to handle this request. When ESP32 reads this request, it knows that user wants to turn the LED ON. So, it turns the LED ON and sends a dynamic webpage to a browser showing LED status : ON. As easy as Pie!

ESP32 as HTTP Server using WiFi Access Point (AP) mode

Now let’s move on to the interesting stuff!

As the heading suggests, this example demonstrates how to turn the ESP32 into an access point (AP), and serve up web pages to any connected client. To start with, plug your ESP32 into your computer and Try the sketch out; and then we will dissect it in some detail.

#include <WiFi.h>

#include <WebServer.h>

/\* Put your SSID & Password \*/

const char\* ssid = "ESP32"; // Enter SSID here

const char\* password = "12345678"; //Enter Password here

/\* Put IP Address details \*/

IPAddress local\_ip(192,168,1,1);

IPAddress gateway(192,168,1,1);

IPAddress subnet(255,255,255,0);

WebServer server(80);

uint8\_t LED1pin = 4;

bool LED1status = LOW;

uint8\_t LED2pin = 5;

bool LED2status = LOW;

void setup() {

Serial.begin(115200);

pinMode(LED1pin, OUTPUT);

pinMode(LED2pin, OUTPUT);

WiFi.softAP(ssid, password);

WiFi.softAPConfig(local\_ip, gateway, subnet);

delay(100);

server.on("/", handle\_OnConnect);

server.on("/led1on", handle\_led1on);

server.on("/led1off", handle\_led1off);

server.on("/led2on", handle\_led2on);

server.on("/led2off", handle\_led2off);

server.onNotFound(handle\_NotFound);

server.begin();

Serial.println("HTTP server started");

}

void loop() {

server.handleClient();

if(LED1status)

{digitalWrite(LED1pin, HIGH);}

else

{digitalWrite(LED1pin, LOW);}

if(LED2status)

{digitalWrite(LED2pin, HIGH);}

else

{digitalWrite(LED2pin, LOW);}

}

void handle\_OnConnect() {

LED1status = LOW;

LED2status = LOW;

Serial.println("GPIO4 Status: OFF | GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,LED2status));

}

void handle\_led1on() {

LED1status = HIGH;

Serial.println("GPIO4 Status: ON");

server.send(200, "text/html", SendHTML(true,LED2status));

}

void handle\_led1off() {

LED1status = LOW;

Serial.println("GPIO4 Status: OFF");

server.send(200, "text/html", SendHTML(false,LED2status));

}

void handle\_led2on() {

LED2status = HIGH;

Serial.println("GPIO5 Status: ON");

server.send(200, "text/html", SendHTML(LED1status,true));

}

void handle\_led2off() {

LED2status = LOW;

Serial.println("GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,false));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

String SendHTML(uint8\_t led1stat,uint8\_t led2stat){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>LED Control</title>\n";

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;} h3 {color: #444444;margin-bottom: 50px;}\n";

ptr +=".button {display: block;width: 80px;background-color: #3498db;border: none;color: white;padding: 13px 30px;text-decoration: none;font-size: 25px;margin: 0px auto 35px;cursor: pointer;border-radius: 4px;}\n";

ptr +=".button-on {background-color: #3498db;}\n";

ptr +=".button-on:active {background-color: #2980b9;}\n";

ptr +=".button-off {background-color: #34495e;}\n";

ptr +=".button-off:active {background-color: #2c3e50;}\n";

ptr +="p {font-size: 14px;color: #888;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<h1>ESP32 Web Server</h1>\n";

ptr +="<h3>Using Access Point(AP) Mode</h3>\n";

if(led1stat)

{ptr +="<p>LED1 Status: ON</p><a class=\"button button-off\" href=\"/led1off\">OFF</a>\n";}

else

{ptr +="<p>LED1 Status: OFF</p><a class=\"button button-on\" href=\"/led1on\">ON</a>\n";}

if(led2stat)

{ptr +="<p>LED2 Status: ON</p><a class=\"button button-off\" href=\"/led2off\">OFF</a>\n";}

else

{ptr +="<p>LED2 Status: OFF</p><a class=\"button button-on\" href=\"/led2on\">ON</a>\n";}

ptr +="</body>\n";

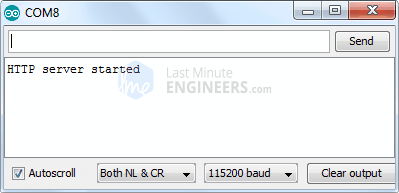
ptr +="</html>\n";

return ptr;

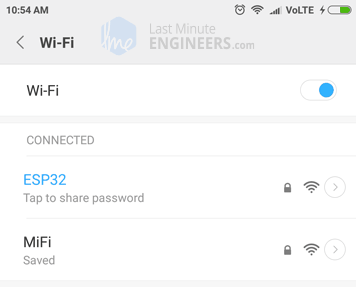
}

Accessing the Web Server in AP mode

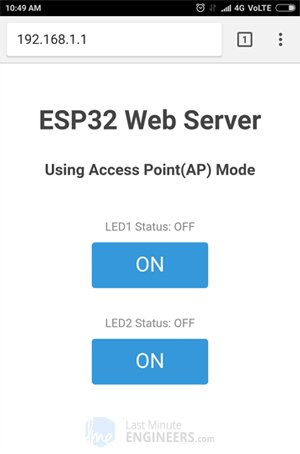
After uploading the sketch, open the Serial Monitor at a baud rate of 115200. And press the RESET button on ESP32. If everything is OK, it will show HTTP server started message.

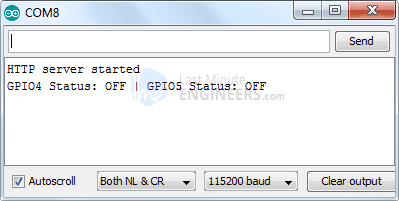


Next, find any device that you can connect to a WiFi network – phone, laptop, etc. And look for a network called ESP32. Join the network with password 123456789.

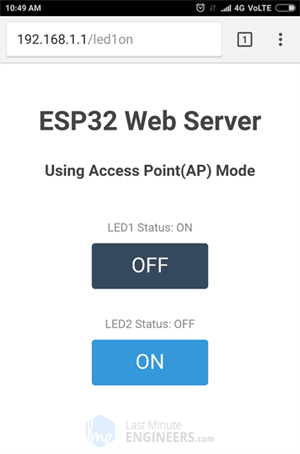


After connecting to your ESP32 AP network, load up a browser and point it to 192.168.1.1 The ESP32 should serve up a web page showing current status of LEDs and two buttons to control them. If take a look at the serial monitor at the same time, you can see status of ESP32’s GPIO pins.





Now, click the button to turn LED1 ON while keeping an eye on the URL. Once you click the button, the ESP32 receives a request for /led1on URL. It then turns the LED1 ON and serves a web page with status of LED updated. It also prints the status of GPIO pin on the serial monitor.





You can test LED2 button and check that it works in a similar way.

Now, let’s take a closer look at the code to see how it works, so that you are able to modify it to fulfill your needs.

Detailed Code Explanation

The sketch starts by including WiFi.h library. This library provides ESP32 specific WiFi methods we are calling to connect to network. Following that we also include the WebServer.h library, which has some methods available that will help us setting up a server and handle incoming HTTP requests without needing to worry about low level implementation details.

#include <WiFi.h>

#include <WebServer.h>

As we are setting the ESP32 in Access Point (AP) mode, it will create a WiFi network. Hence, we need to set its SSID, Password, IP address, IP subnet mask and IP gateway.

/\* Put your SSID & Password \*/

const char\* ssid = "ESP32"; // Enter SSID here

const char\* password = "12345678"; //Enter Password here

/\* Put IP Address details \*/

IPAddress local\_ip(192,168,1,1);

IPAddress gateway(192,168,1,1);

IPAddress subnet(255,255,255,0);

Next, we declare an object of WebServer library, so we can access its functions. The constructor of this object takes [port](https://en.wikipedia.org/wiki/Port_(computer_networking)) (where the server will be listening to) as a parameter. Since 80 is the default port for HTTP, we will use this value. Now you can access the server without needing to specify the port in the URL.

// declare an object of WebServer library

WebServer server(80);

Next, we declare the ESP32’s GPIO pins to which LEDs are connected and their initial state.

uint8\_t LED1pin = 4;

bool LED1status = LOW;

uint8\_t LED2pin = 5;

bool LED2status = LOW;

Inside Setup() Function

We configure our HTTP server before actually running it. First of all, we open a serial connection for debugging purpose and set GPIO ports to OUTPUT.

Serial.begin(115200);

pinMode(LED1pin, OUTPUT);

pinMode(LED2pin, OUTPUT);

Then, we set up a soft access point to establish a Wi-Fi network by proving SSID, Password, IP address, IP subnet mask and IP gateway.

WiFi.softAP(ssid, password);

WiFi.softAPConfig(local\_ip, gateway, subnet);

delay(100);

In order to handle incoming HTTP requests, we need to specify which code to execute when a particular URL is hit. To do so, we use on method. This method takes two parameters. First one is a URL path and second one is the name of function which we want to execute when that URL is hit.

For example, the first line of below code snippet indicates that when a server receives an HTTP request on the root (/) path, it will trigger the handle\_OnConnect() function. Note that the URL specified is a relative path.

Likewise, we need to specify 4 more URLs to handle two states of 2 LEDs.

server.on("/", handle\_OnConnect);

server.on("/led1on", handle\_led1on);

server.on("/led1off", handle\_led1off);

server.on("/led2on", handle\_led2on);

server.on("/led2off", handle\_led2off);

We haven’t specified what the server should do if the client requests any URL other than specified with server.on() . It should respond with an HTTP status 404 (Not Found) and a message for the user. We put this in a function as well, and use server.onNotFound() to tell it that it should execute it when it receives a request for a URI that wasn’t specified with server.on

server.onNotFound(handle\_NotFound);

Now, to start our server, we call the begin method on the server object.

server.begin();

Serial.println("HTTP server started");

Inside Loop() Function

To handle the actual incoming HTTP requests, we need to call the handleClient() method on the server object. We also change the state of LED as per the request.

void loop() {

server.handleClient();

if(LED1status)

{digitalWrite(LED1pin, HIGH);}

else

{digitalWrite(LED1pin, LOW);}

if(LED2status)

{digitalWrite(LED2pin, HIGH);}

else

{digitalWrite(LED2pin, LOW);}

}

Next, we need to create a function we attached to root (/) URL with server.on. Remember? At the start of this function, we set the status of both the LEDs to LOW (Initial state of LEDs) and print it on serial monitor. In order to respond to the HTTP request, we use the send method. Although the method can be called with a different set of arguments, its simplest form consists of the HTTP response code, the content type and the content.

In our case, we are sending the code 200 (one of the [HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)), which corresponds to the OK response. Then, we are specifying the content type as “text/html“, and finally we are calling SendHTML() custom function which creates a dynamic HTML page containing status of LEDs.

void handle\_OnConnect() {

LED1status = LOW;

LED2status = LOW;

Serial.println("GPIO4 Status: OFF | GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,LED2status));

}

Likewise, we need to create four functions to handle LED ON/OFF requests and 404 Error page.

void handle\_led1on() {

LED1status = HIGH;

Serial.println("GPIO4 Status: ON");

server.send(200, "text/html", SendHTML(true,LED2status));

}

void handle\_led1off() {

LED1status = LOW;

Serial.println("GPIO4 Status: OFF");

server.send(200, "text/html", SendHTML(false,LED2status));

}

void handle\_led2on() {

LED2status = HIGH;

Serial.println("GPIO5 Status: ON");

server.send(200, "text/html", SendHTML(LED1status,true));

}

void handle\_led2off() {

LED2status = LOW;

Serial.println("GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,false));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

Displaying the HTML Web Page

SendHTML() function is responsible for generating a web page whenever the ESP32 web server gets a request from a web client. It merely concatenates HTML code into a big string and returns to the server.send() function we discussed earlier. The function takes status of LEDs as a parameter to dynamically generate the HTML content.

The first text you should always send is the [<!DOCTYPE> declaration](https://www.w3schools.com/tags/tag_doctype.asp) that indicates that we’re sending HTML code.

String SendHTML(uint8\_t led1stat,uint8\_t led2stat){

String ptr = "<!DOCTYPE html> <html>\n";

Next, the [<meta> viewport element](https://www.w3schools.com/css/css_rwd_viewport.asp) makes the web page responsive in any web browser. While title tag sets the title of the page.

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>LED Control</title>\n";

Styling the Web Page

Next, we have some CSS to style the buttons and the web page appearance. We choose the Helvetica font, define the content to be displayed as an inline-block and aligned at the center.

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

Following code then sets color, font and margin around the body, H1, H3 and p tags.

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;} h3 {color: #444444;margin-bottom: 50px;}\n";

ptr +="p {font-size: 14px;color: #888;margin-bottom: 10px;}\n";

Some styling is applied to the buttons as well with properties like color, size, margin, etc. The ON and OFF button has different background color while [:active selector](https://www.w3schools.com/cssref/sel_active.asp) for buttons ensure button click effect.

ptr +=".button {display: block;width: 80px;background-color: #3498db;border: none;color: white;padding: 13px 30px;text-decoration: none;font-size: 25px;margin: 0px auto 35px;cursor: pointer;border-radius: 4px;}\n";

ptr +=".button-on {background-color: #3498db;}\n";

ptr +=".button-on:active {background-color: #2980b9;}\n";

ptr +=".button-off {background-color: #34495e;}\n";

ptr +=".button-off:active {background-color: #2c3e50;}\n";

Setting the Web Page Heading

Next, heading of the web page is set; you can change this text to anything that suits your application.

ptr +="<h1>ESP32 Web Server</h1>\n";

ptr +="<h3>Using Access Point(AP) Mode</h3>\n";

Displaying the Buttons and Corresponding State

To dynamically generate the buttons and LED status, we use if statement. So, depending upon the status of the GPIO pins, ON/OFF button is displayed.

if(led1stat)

{ptr +="<p>LED1 Status: ON</p><a class=\"button button-off\" href=\"/led1off\">OFF</a>\n";}

else

{ptr +="<p>LED1 Status: OFF</p><a class=\"button button-on\" href=\"/led1on\">ON</a>\n";}

if(led2stat)

{ptr +="<p>LED2 Status: ON</p><a class=\"button button-off\" href=\"/led2off\">OFF</a>\n";}

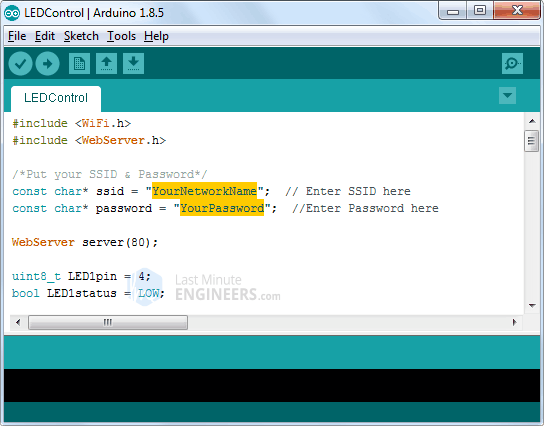
else

{ptr +="<p>LED2 Status: OFF</p><a class=\"button button-on\" href=\"/led2on\">ON</a>\n";}

ESP32 as HTTP Server using WiFi Station (STA) mode

Now let’s move on to our next example which demonstrates how to turn the ESP32 into Station (STA) mode, and serve up web pages to any connected client under existing network.

Before you head for uploading the sketch, you need to make some changes to make it work for you. You need to modify the following two variables with your network credentials, so that ESP32 can establish a connection with existing network.



Once you are done, go ahead and try the sketch out.

#include <WiFi.h>

#include <WebServer.h>

/\*Put your SSID & Password\*/

const char\* ssid = " YourNetworkName"; // Enter SSID here

const char\* password = " YourPassword"; //Enter Password here

WebServer server(80);

uint8\_t LED1pin = 4;

bool LED1status = LOW;

uint8\_t LED2pin = 5;

bool LED2status = LOW;

void setup() {

Serial.begin(115200);

delay(100);

pinMode(LED1pin, OUTPUT);

pinMode(LED2pin, OUTPUT);

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

server.on("/", handle\_OnConnect);

server.on("/led1on", handle\_led1on);

server.on("/led1off", handle\_led1off);

server.on("/led2on", handle\_led2on);

server.on("/led2off", handle\_led2off);

server.onNotFound(handle\_NotFound);

server.begin();

Serial.println("HTTP server started");

}

void loop() {

server.handleClient();

if(LED1status)

{digitalWrite(LED1pin, HIGH);}

else

{digitalWrite(LED1pin, LOW);}

if(LED2status)

{digitalWrite(LED2pin, HIGH);}

else

{digitalWrite(LED2pin, LOW);}

}

void handle\_OnConnect() {

LED1status = LOW;

LED2status = LOW;

Serial.println("GPIO4 Status: OFF | GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,LED2status));

}

void handle\_led1on() {

LED1status = HIGH;

Serial.println("GPIO4 Status: ON");

server.send(200, "text/html", SendHTML(true,LED2status));

}

void handle\_led1off() {

LED1status = LOW;

Serial.println("GPIO4 Status: OFF");

server.send(200, "text/html", SendHTML(false,LED2status));

}

void handle\_led2on() {

LED2status = HIGH;

Serial.println("GPIO5 Status: ON");

server.send(200, "text/html", SendHTML(LED1status,true));

}

void handle\_led2off() {

LED2status = LOW;

Serial.println("GPIO5 Status: OFF");

server.send(200, "text/html", SendHTML(LED1status,false));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

String SendHTML(uint8\_t led1stat,uint8\_t led2stat){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>LED Control</title>\n";

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;} h3 {color: #444444;margin-bottom: 50px;}\n";

ptr +=".button {display: block;width: 80px;background-color: #3498db;border: none;color: white;padding: 13px 30px;text-decoration: none;font-size: 25px;margin: 0px auto 35px;cursor: pointer;border-radius: 4px;}\n";

ptr +=".button-on {background-color: #3498db;}\n";

ptr +=".button-on:active {background-color: #2980b9;}\n";

ptr +=".button-off {background-color: #34495e;}\n";

ptr +=".button-off:active {background-color: #2c3e50;}\n";

ptr +="p {font-size: 14px;color: #888;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<h1>ESP32 Web Server</h1>\n";

ptr +="<h3>Using Station(STA) Mode</h3>\n";

if(led1stat)

{ptr +="<p>LED1 Status: ON</p><a class=\"button button-off\" href=\"/led1off\">OFF</a>\n";}

else

{ptr +="<p>LED1 Status: OFF</p><a class=\"button button-on\" href=\"/led1on\">ON</a>\n";}

if(led2stat)

{ptr +="<p>LED2 Status: ON</p><a class=\"button button-off\" href=\"/led2off\">OFF</a>\n";}

else

{ptr +="<p>LED2 Status: OFF</p><a class=\"button button-on\" href=\"/led2on\">ON</a>\n";}

ptr +="</body>\n";

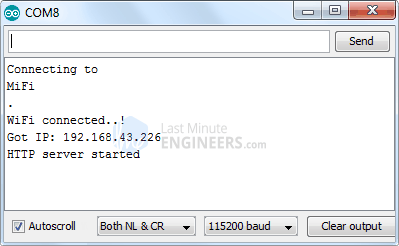
ptr +="</html>\n";

return ptr;

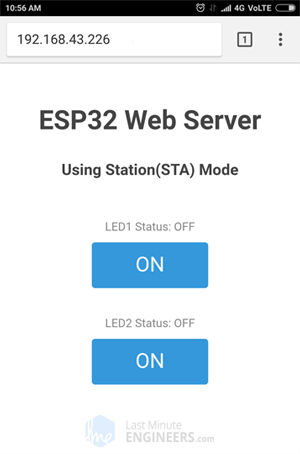
}

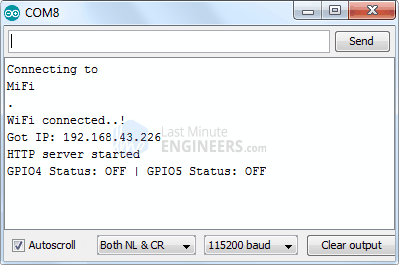
Accessing the Web Server in STA mode

After uploading the sketch, open the Serial Monitor at a baud rate of 115200. And press the RESET button on ESP32. If everything is OK, it will output the dynamic IP address obtained from your router and show HTTP server started message.



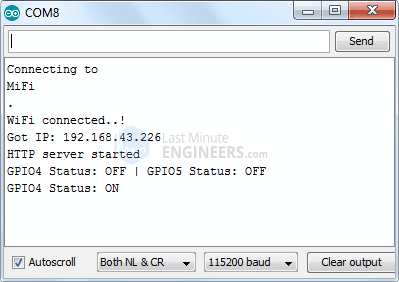
Next, load up a browser and point it to the IP address shown on the serial monitor. The ESP32 should serve up a web page showing current status of LEDs and two buttons to control them. If take a look at the serial monitor at the same time, you can see status of ESP32’s GPIO pins.





Now, click the button to turn LED1 ON while keeping an eye on the URL. Once you click the button, the ESP32 receives a request for /led1on URL. It then turns the LED1 ON and serves a web page with status of LED updated. It also prints the status of GPIO pin on the serial monitor.





You can test LED2 button and check that it works in a similar way.

Code Explanation

If you observe this code with the previous code, the only difference is that we are not setting the soft Access Point, Instead we are joining existing network using WiFi.begin() function.

//connect to your local wi-fi network

WiFi.begin(ssid, password);

While the ESP32 tries to connect to the network, we can check the connectivity status with WiFi.status() function.

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED)

{

delay(1000);

Serial.print(".");

}

Just for your information, this function returns the following statuses:

* WL\_CONNECTED: assigned when connected to a Wi-Fi network
* WL\_NO\_SHIELD: assigned when no Wi-Fi shield is present
* WL\_IDLE\_STATUS: a temporary status assigned when WiFi.begin() is called and remains active until the number of attempts expires (resulting in WL\_CONNECT\_FAILED) or a connection is established (resulting in WL\_CONNECTED)
* WL\_NO\_SSID\_AVAIL: assigned when no SSID are available
* WL\_SCAN\_COMPLETED: assigned when the scan networks is completed
* WL\_CONNECT\_FAILED: assigned when the connection fails for all the attempts
* WL\_CONNECTION\_LOST: assigned when the connection is lost
* WL\_DISCONNECTED: assigned when disconnected from a network

Once the ESP32 is connected to the network, the sketch prints the IP address assigned to ESP32 by displaying WiFi.localIP() value on serial monitor.

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

The only difference between AP & STA mode is one creates the network and other joins the existing network. So, rest of the code for handling HTTP requests and serving web page in STA mode is same as that of AP mode explained above. This includes:

* Declaring ESP32’s GPIO pins to which LEDs are connected
* Defining multiple server.on() methods to handle incoming HTTP requests
* Defining server.onNotFound() method to handle HTTP 404 error
* Creating custom functions that are executed when specific URL is hit
* Creating HTML page
* Styling the web page
* Creating buttons and displaying their status

Interface DHT11 DHT22 with ESP32 & Display Values Using Web Server



Have you ever wanted to have sensors scattered all around your house and garden reporting their temperature regularly to a central server? Then, this IoT project might be the solid launching point for you! This project uses ESP32 as the control device that easily connects to existing WiFi network & creates a Web Server. When any connected device accesses this web server, ESP32 reads in temperature & relative humidity from the DHT11, DHT22/AM2302 sensor & sends it to the web browser of that device with a nice interface. Excited? Let’s get started!

It may look intimidating, but there are a few concepts you should be familiar with, before venturing further into this tutorial. If any of the concepts below sound foreign to you, consider reading (at least skimming) through that tutorial first:

[](https://lastminuteengineers.com/dht11-dht22-arduino-tutorial/)

[How DHT11 DHT22 Sensors Work & Interface With Arduino](https://lastminuteengineers.com/dht11-dht22-arduino-tutorial/)

Give your next Arduino project the ability to sense the world around it with the inexpensive DHT11 or DHT22 Digital Temperature & Humidity Sensor from...

[](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

[Insight Into ESP32 Features & Using It With Arduino IDE](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

Few years back, ESP8266 took the embedded IoT world by storm. For less than $3, you could get a programmable, WiFi-enabled microcontroller being able to...

[](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

[Create A Simple ESP32 Web Server In Arduino IDE](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

The newly launched successor of ESP8266 - the ESP32 has been a growing star among IoT or WiFi-related projects. It’s an extremely cost-effective WiFi module...

Problem with ESP32’s Internal Temperature Sensor

If you didn’t know, ESP32 comes with an internal temperature sensor having a range of -40°C to 125°C. The temperature sensor generates a voltage proportional to the temperature which is converted into digital form via an internal analog-to-digital converter.

As per ESP32 datasheet, the problem with this temperature sensor is that, the offset of the sensor varies from chip to chip due to process variation. Also the heat generated by the Wi-Fi circuitry affects temperature measurements. So, the internal temperature sensor is only suitable for applications that detect temperature changes instead of absolute temperatures.

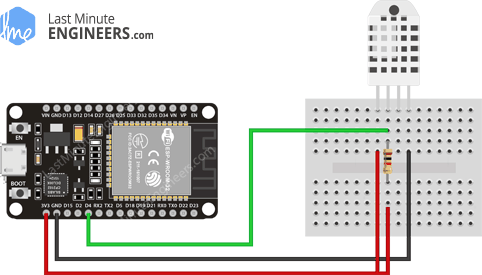
However, you can achieve accurate results by calibrating the temperature sensor and using ESP32 in a minimally powered-on application to reduce CPU temperature. But the results will never be as accurate as with inexpensive temperature sensors like DHT11, DHT22/AM2302.

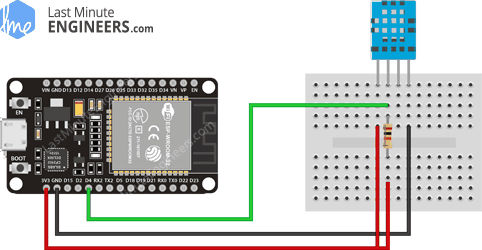
Wiring – Connecting DHT11, DHT22/AM2302 sensor to ESP32

Connecting DHT11/DHT22/AM2302 sensor to ESP32 is fairly simple. Start by placing the ESP32 on to your breadboard, ensuring each side of the board is on a separate side of the breadboard.

Now place the sensor on to your breadboard besides ESP32. Connect VCC pin on the sensor to the 3.3V pin on the ESP32 and ground to ground. Also connect Data pin of the sensor to D4 pin on the ESP32. Finally, we need to place a pull-up resistor of 10KΩ between VCC and data line to keep it HIGH for proper communication between sensor and MCU. If you happen to have a breakout board of the sensor, you need not add any external pull-up. It comes with a built-in pull-up resistor.

When you’re done you should have something that looks similar to the illustration shown below.

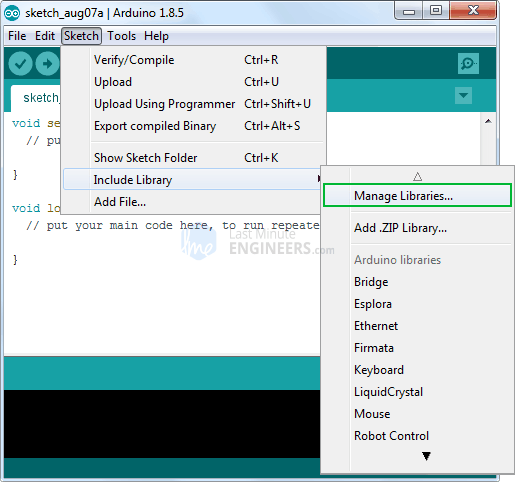
Wiring DHT22 Temperature Humidity Sensor with ESP32

Wiring DHT11 Temperature Humidity Sensor with ESP32

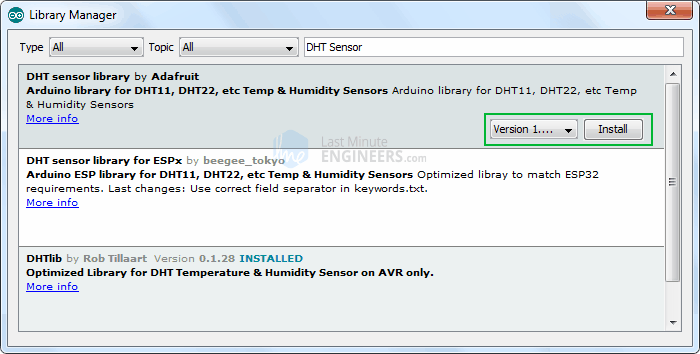
Installing DHT Sensor Library

Communicating with DHT11, DHT22/AM2302 sensors is a bunch of work, as they have their own single wire protocol for data transfer. And this protocol requires precise timing. Fortunately, we don’t have to worry much about this because we are going to use the DHT library from Adafruit which takes care of almost everything. The library is so powerful that it runs on both Arduino and ESP architecture.

To install the library navigate to the Sketch > Include Library > Manage Libraries… Wait for Library Manager to download libraries index and update list of installed libraries.



Filter your search by typing ‘DHT sensor’. There should be a couple entries. Look for DHT sensor library by Adafruit. Click on that entry, and then select Install.



The DHT sensor library uses the [Adafruit Sensor support backend](https://github.com/adafruit/Adafruit_Sensor). So, search the library manager for Adafruit Unified Sensor and install that too (you may have to scroll a bit)

Create ESP32 Web Server using WiFi Station (STA) mode

Now let’s move on to the interesting stuff!

As the heading suggests, we are going to configure our ESP32 into Station (STA) mode, and create a web server to serve up web pages to any connected client under existing network.

Before you head for uploading the sketch, you need to make one change to make it work for you. You need to modify the following two variables with your network credentials, so that ESP32 can establish a connection with existing network.

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

Once you are done, go ahead and try the sketch out and then we will dissect it in some detail.

#include <WiFi.h>

#include <WebServer.h>

#include "DHT.h"

// Uncomment one of the lines below for whatever DHT sensor type you're using!

//#define DHTTYPE DHT11 // DHT 11

//#define DHTTYPE DHT21 // DHT 21 (AM2301)

#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

WebServer server(80);

// DHT Sensor

uint8\_t DHTPin = 4;

// Initialize DHT sensor.

DHT dht(DHTPin, DHTTYPE);

float Temperature;

float Humidity;

void setup() {

Serial.begin(115200);

delay(100);

pinMode(DHTPin, INPUT);

dht.begin();

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

server.on("/", handle\_OnConnect);

server.onNotFound(handle\_NotFound);

server.begin();

Serial.println("HTTP server started");

}

void loop() {

server.handleClient();

}

void handle\_OnConnect() {

Temperature = dht.readTemperature(); // Gets the values of the temperature

Humidity = dht.readHumidity(); // Gets the values of the humidity

server.send(200, "text/html", SendHTML(Temperature,Humidity));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

String SendHTML(float Temperaturestat,float Humiditystat){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Weather Report</title>\n";

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Report</h1>\n";

ptr +="<p>Temperature: ";

ptr +=(int)Temperaturestat;

ptr +="°C</p>";

ptr +="<p>Humidity: ";

ptr +=(int)Humiditystat;

ptr +="%</p>";

ptr +="</div>\n";

ptr +="</body>\n";

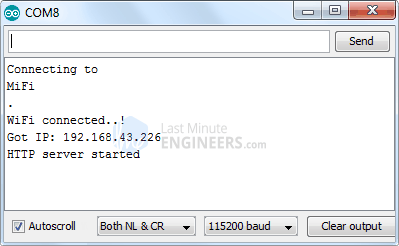
ptr +="</html>\n";

return ptr;

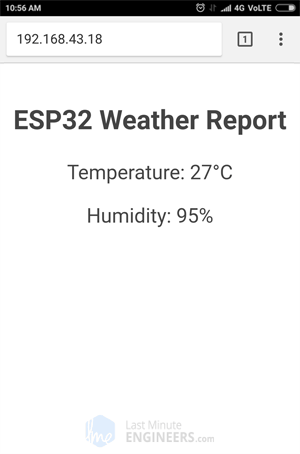
}

Accessing the Web Server

After uploading the sketch, open the Serial Monitor at a baud rate of 115200. And press the RESET button on ESP32. If everything is OK, it will output the dynamic IP address obtained from your router and show HTTP server started message.



Next, load up a browser and point it to the IP address shown on the serial monitor. The ESP32 should serve up a web page showing temperature and relative humidity.



Detailed Code Explanation

The sketch starts by including WiFi.h library. This library provides ESP32 specific WiFi methods we are calling to connect to network. Following that we also include the WebServer.h library, which has some methods available that will help us setting up a server and handle incoming HTTP requests without needing to worry about low level implementation details. Finally we include DHT.h library.

#include <WiFi.h>

#include <WebServer.h>

#include "DHT.h"

Next, we need to define the type of DHT sensor we are using. Uncomment one of the lines below accordingly!

//#define DHTTYPE DHT11 // DHT 11

//#define DHTTYPE DHT21 // DHT 21 (AM2301)

#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321

As we are configuring ESP32 in Station (STA) mode, it will join existing WiFi network. Hence, we need to provide it with your network’s SSID & Password.

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

Next, we declare an object of WebServer library, so we can access its functions. The constructor of this object takes [port](https://en.wikipedia.org/wiki/Port_(computer_networking)) (where the server will be listening to) as a parameter. Since 80 is the default port for HTTP, we will use this value. Now you can access the server without needing to specify the port in the URL.

// declare an object of WebServer library

WebServer server(80);

Next, we need to define the ESP32’s pin number to which our sensor’s Data pin is connected and create a DHT object. So, that we can access special functions related to the DHT library.

// DHT Sensor

uint8\_t DHTPin = 4;

// Initialize DHT sensor.

DHT dht(DHTPin, DHTTYPE);

Two float variables viz. Temperature & Humidity are declared to store respective values.

float Temperature;

float Humidity;

Inside Setup() Function

Inside Setup() Function we configure our HTTP server before actually running it. First of all, we open a serial connection for debugging purpose and set GPIO ports to INPUT. We also need to initialize the DHT object using begin() function.

Serial.begin(115200);

delay(100);

pinMode(DHTPin, INPUT);

dht.begin();

Now, we need to join the WiFi network using WiFi.begin() function. The function takes SSID (Network Name) and password as a parameter.

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

While the ESP32 tries to connect to the network, we can check the connectivity status with WiFi.status() function.

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Once the ESP32 is connected to the network, the sketch prints the IP address assigned to ESP32 by displaying WiFi.localIP() value on serial monitor.

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

In order to handle incoming HTTP requests, we need to specify which code to execute when a URL is hit. To do so, we use on method. This method takes two parameters. First one is a URL path and second one is the name of function which we want to execute when that URL is hit.

The code below indicates that when a server receives an HTTP request on the root (/) path, it will trigger the handle\_OnConnect function. Note that the URL specified is a relative path.

server.on("/", handle\_OnConnect);

We haven’t specified what the server should do if the client requests any URL other than specified with server.on. It should respond with an HTTP status 404 (Not Found) and a message for the user. We put this in a function as well, and use server.onNotFound to tell it that it should execute it when it receives a request for a URI that wasn’t specified with server.on

server.onNotFound(handle\_NotFound);

Now, to start our server, we call the begin method on the server object.

server.begin();

Serial.println("HTTP server started");

Inside Loop() Function

To handle the actual incoming HTTP requests, we need to call the handleClient() method on the server object.

server.handleClient();

Next, we need to create a function we attached to root (/) URL with server.on. Remember? At the start of this function, we get the values of temperature and humidity from the sensor. In order to respond to the HTTP request, we use the send method. Although the method can be called with a different set of arguments, its simplest form consists of the HTTP response code, the content type and the content.

In our case, we are sending the code 200 (one of the [HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)), which corresponds to the OK response. Then, we are specifying the content type as “text/html“, and finally we are calling SendHTML() custom function which creates a dynamic HTML page containing values of temperature and humidity.

void handle\_OnConnect()

{

Temperature = dht.readTemperature(); // Gets the values of the temperature

Humidity = dht.readHumidity(); // Gets the values of the humidity

server.send(200, "text/html", SendHTML(Temperature,Humidity));

}

Likewise, we need to create a function to handle 404 Error page.

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

Displaying the HTML Web Page

SendHTML() function is responsible for generating a web page whenever the ESP32 web server gets a request from a web client. It merely concatenates HTML code into a big string and returns to the server.send() function we discussed earlier. The function takes values of temperature and humidity as a parameter to dynamically generate the HTML content.

The first text you should always send is the <!DOCTYPE> declaration that indicates that we’re sending HTML code.

String SendHTML(float Temperaturestat,float Humiditystat){

String ptr = "<!DOCTYPE html> <html>\n";

Next, the <meta> viewport element makes the web page responsive in any web browser, while title tag sets the title of the page.

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Weather Report</title>\n";

Styling the Web Page

Next, we have some CSS to style the web page appearance. We choose the Helvetica font, define the content to be displayed as an inline-block and aligned at the center.

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

Following code then sets color, font and margin around the body, H1 and p tags.

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

Setting the Web Page Heading

Next, heading of the web page is set; you can change this text to anything that suits your application.

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Report</h1>\n";

Displaying Temperature and Humidity on Web Page

To dynamically display values of Temperature & humidity, we put those values in paragraph tag. These values are converted to integer by type casting. To display degree symbol, we use HTML entity &deg;

ptr +="<p>Temperature: ";

ptr +=(int)Temperaturestat;

ptr +="°C</p>";

ptr +="<p>Humidity: ";

ptr +=(int)Humiditystat;

ptr +="%</p>";

ptr +="</div>\n";

ptr +="</body>\n";

ptr +="</html>\n";

return ptr;

}

Styling Web Page to Look More Professional

Programmers like us are often intimidated by design – but a little effort can make your web page look more attractive and professional. Below screenshot will give you a basic idea of what we are going to do.



Pretty amazing, Right? Without further ado, let’s apply some style to our previous HTML page. To start with, copy-paste below code to replace SendHTML() function from the sketch above. Try the new sketch out and then we will do its detailed breakdown.

String SendHTML(float TempCstat,float TempFstat,float Humiditystat){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<link href=\"https://fonts.googleapis.com/css?family=Open+Sans:300,400,600\" rel=\"stylesheet\">\n";

ptr +="<title>ESP32 Weather Report</title>\n";

ptr +="<style>html { font-family: 'Open Sans', sans-serif; display: block; margin: 0px auto; text-align: center;color: #333333;}\n";

ptr +="body{margin-top: 50px;}\n";

ptr +="h1 {margin: 50px auto 30px;}\n";

ptr +=".side-by-side{display: inline-block;vertical-align: middle;position: relative;}\n";

ptr +=".humidity-icon{background-color: #3498db;width: 30px;height: 30px;border-radius: 50%;line-height: 36px;}\n";

ptr +=".humidity-text{font-weight: 600;padding-left: 15px;font-size: 19px;width: 160px;text-align: left;}\n";

ptr +=".humidity{font-weight: 300;font-size: 60px;color: #3498db;}\n";

ptr +=".temperature-icon{background-color: #f39c12;width: 30px;height: 30px;border-radius: 50%;line-height: 40px;}\n";

ptr +=".temperature-text{font-weight: 600;padding-left: 15px;font-size: 19px;width: 160px;text-align: left;}\n";

ptr +=".temperature{font-weight: 300;font-size: 60px;color: #f39c12;}\n";

ptr +=".superscript{font-size: 17px;font-weight: 600;position: absolute;right: -20px;top: 15px;}\n";

ptr +=".data{padding: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Report</h1>\n";

ptr +="<div class=\"data\">\n";

ptr +="<div class=\"side-by-side temperature-icon\">\n";

ptr +="<svg version=\"1.1\" id=\"Layer\_1\" xmlns=\"http://www.w3.org/2000/svg\" xmlns:xlink=\"http://www.w3.org/1999/xlink\" x=\"0px\" y=\"0px\"\n";

ptr +="width=\"9.915px\" height=\"22px\" viewBox=\"0 0 9.915 22\" enable-background=\"new 0 0 9.915 22\" xml:space=\"preserve\">\n";

ptr +="<path fill=\"#FFFFFF\" d=\"M3.498,0.53c0.377-0.331,0.877-0.501,1.374-0.527C5.697-0.04,6.522,0.421,6.924,1.142\n";

ptr +="c0.237,0.399,0.315,0.871,0.311,1.33C7.229,5.856,7.245,9.24,7.227,12.625c1.019,0.539,1.855,1.424,2.301,2.491\n";

ptr +="c0.491,1.163,0.518,2.514,0.062,3.693c-0.414,1.102-1.24,2.038-2.276,2.594c-1.056,0.583-2.331,0.743-3.501,0.463\n";

ptr +="c-1.417-0.323-2.659-1.314-3.3-2.617C0.014,18.26-0.115,17.104,0.1,16.022c0.296-1.443,1.274-2.717,2.58-3.394\n";

ptr +="c0.013-3.44,0-6.881,0.007-10.322C2.674,1.634,2.974,0.955,3.498,0.53z\"/>\n";

ptr +="</svg>\n";

ptr +="</div>\n";

ptr +="<div class=\"side-by-side temperature-text\">Temperature</div>\n";

ptr +="<div class=\"side-by-side temperature\">";

ptr +=(int)TempCstat;

ptr +="<span class=\"superscript\">°C</span></div>\n";

ptr +="</div>\n";

ptr +="<div class=\"data\">\n";

ptr +="<div class=\"side-by-side humidity-icon\">\n";

ptr +="<svg version=\"1.1\" id=\"Layer\_2\" xmlns=\"http://www.w3.org/2000/svg\" xmlns:xlink=\"http://www.w3.org/1999/xlink\" x=\"0px\" y=\"0px\"\n\"; width=\"12px\" height=\"17.955px\" viewBox=\"0 0 13 17.955\" enable-background=\"new 0 0 13 17.955\" xml:space=\"preserve\">\n";

ptr +="<path fill=\"#FFFFFF\" d=\"M1.819,6.217C3.139,4.064,6.5,0,6.5,0s3.363,4.064,4.681,6.217c1.793,2.926,2.133,5.05,1.571,7.057\n";

ptr +="c-0.438,1.574-2.264,4.681-6.252,4.681c-3.988,0-5.813-3.107-6.252-4.681C-0.313,11.267,0.026,9.143,1.819,6.217\"></path>\n";

ptr +="</svg>\n";

ptr +="</div>\n";

ptr +="<div class=\"side-by-side humidity-text\">Humidity</div>\n";

ptr +="<div class=\"side-by-side humidity\">";

ptr +=(int)Humiditystat;

ptr +="<span class=\"superscript\">%</span></div>\n";

ptr +="</div>\n";

ptr +="</div>\n";

ptr +="</body>\n";

ptr +="</html>\n";

return ptr;

}

We already know that <!DOCTYPE> declaration tells the browser we’re sending HTML code and the <meta> viewport element makes the web page responsive. The only addition here is we are going to use [google fonts](https://fonts.google.com/). Google has hundreds of web fonts and are free for commercial and personal use. Perfect!

We will use Google commissioned [Open Sans](https://fonts.google.com/specimen/Open+Sans) web font for our web page. Google font is embedded using link tag in <head> of your HTML document. We have selected 300 (Light), 400 (Regular) & 600 (bold) font weight for our page. You can select as many as you want but keep in mind that selecting unnecessary font weights hamper page load time. You can also add italic style by just adding i character at the end of font weight e.g. 400i will embed italic style.

Note that you cannot see Google font, without active internet connection on the device, you are going to access this page. Google fonts are loaded on the fly.

String SendHTML(float TempCstat,float TempFstat,float Humiditystat){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<link href=\"https://fonts.googleapis.com/css?family=Open+Sans:300,400,600\" rel=\"stylesheet\">\n";

Next, we are going to apply ‘Open Sans’ font for our whole HTML. We also need to specify sans-serif as our fallback font, to ensure maximum compatibility between browsers/operating systems. If the browser does not support the first font, it tries the next font.

ptr +="<title>ESP32 Weather Report</title>\n";

ptr +="<style>html { font-family: 'Open Sans', sans-serif; display: block; margin: 0px auto; text-align: center;color: #333333;}\n";

ptr +="body{margin-top: 50px;}\n";

ptr +="h1 {margin: 50px auto 30px;}\n";

Next, we need to apply CSS for Humidity & Temperature – icons, titles and actual values. All these three things are made inline and aligned vertical. Background of icons is made circle using 50% border radius and made 30px height and width.

ptr +=".side-by-side{display: inline-block;vertical-align: middle;position: relative;}\n";

ptr +=".humidity-icon{background-color: #3498db;width: 30px;height: 30px;border-radius: 50%;line-height: 36px;}\n";

ptr +=".humidity-text{font-weight: 600;padding-left: 15px;font-size: 19px;width: 160px;text-align: left;}\n";

ptr +=".humidity{font-weight: 300;font-size: 60px;color: #3498db;}\n";

ptr +=".temperature-icon{background-color: #f39c12;width: 30px;height: 30px;border-radius: 50%;line-height: 40px;}\n";

ptr +=".temperature-text{font-weight: 600;padding-left: 15px;font-size: 19px;width: 160px;text-align: left;}\n";

ptr +=".temperature{font-weight: 300;font-size: 60px;color: #f39c12;}\n";

ptr +=".superscript{font-size: 17px;font-weight: 600;position: absolute;right: -20px;top: 15px;}\n";

ptr +=".data{padding: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

Next, we will show temperature readings with this  nice little icon.

The temperature icon is actually a [Scalable Vector Graphics](https://en.wikipedia.org/wiki/Scalable_Vector_Graphics) (SVG) defined in <svg> tag. Creating SVG doesn’t require any special programming skills. You can use [Google SVG Editor](https://svg-edit.github.io/svgedit/releases/latest/editor/svg-editor.html) for creating graphics for your page. After the icon, we are going to show the actual value of temperature from the sensor.

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Report</h1>\n";

ptr +="<div class=\"data\">\n";

ptr +="<div class=\"side-by-side temperature-icon\">\n";

ptr +="<svg version=\"1.1\" id=\"Layer\_1\" xmlns=\"http://www.w3.org/2000/svg\" xmlns:xlink=\"http://www.w3.org/1999/xlink\" x=\"0px\" y=\"0px\"\n";

ptr +="width=\"9.915px\" height=\"22px\" viewBox=\"0 0 9.915 22\" enable-background=\"new 0 0 9.915 22\" xml:space=\"preserve\">\n";

ptr +="<path fill=\"#FFFFFF\" d=\"M3.498,0.53c0.377-0.331,0.877-0.501,1.374-0.527C5.697-0.04,6.522,0.421,6.924,1.142\n";

ptr +="c0.237,0.399,0.315,0.871,0.311,1.33C7.229,5.856,7.245,9.24,7.227,12.625c1.019,0.539,1.855,1.424,2.301,2.491\n";

ptr +="c0.491,1.163,0.518,2.514,0.062,3.693c-0.414,1.102-1.24,2.038-2.276,2.594c-1.056,0.583-2.331,0.743-3.501,0.463\n";

ptr +="c-1.417-0.323-2.659-1.314-3.3-2.617C0.014,18.26-0.115,17.104,0.1,16.022c0.296-1.443,1.274-2.717,2.58-3.394\n";

ptr +="c0.013-3.44,0-6.881,0.007-10.322C2.674,1.634,2.974,0.955,3.498,0.53z\"/>\n";

ptr +="</svg>\n";

ptr +="</div>\n";

ptr +="<div class=\"side-by-side temperature-text\">Temperature</div>\n";

ptr +="<div class=\"side-by-side temperature\">";

ptr +=(int)TempCstat;

ptr +="<span class=\"superscript\">°C</span></div>\n";

ptr +="</div>\n";

Next, we will show humidity readings with this  icon.

Again it’s a SVG. After printing humidity values, we are going to end all the open tags like body and html.

ptr +="<div class=\"data\">\n";

ptr +="<div class=\"side-by-side humidity-icon\">\n";

ptr +="<svg version=\"1.1\" id=\"Layer\_2\" xmlns=\"http://www.w3.org/2000/svg\" xmlns:xlink=\"http://www.w3.org/1999/xlink\" x=\"0px\" y=\"0px\"\n\"; width=\"12px\" height=\"17.955px\" viewBox=\"0 0 13 17.955\" enable-background=\"new 0 0 13 17.955\" xml:space=\"preserve\">\n";

ptr +="<path fill=\"#FFFFFF\" d=\"M1.819,6.217C3.139,4.064,6.5,0,6.5,0s3.363,4.064,4.681,6.217c1.793,2.926,2.133,5.05,1.571,7.057\n";

ptr +="c-0.438,1.574-2.264,4.681-6.252,4.681c-3.988,0-5.813-3.107-6.252-4.681C-0.313,11.267,0.026,9.143,1.819,6.217\"></path>\n";

ptr +="</svg>\n";

ptr +="</div>\n";

ptr +="<div class=\"side-by-side humidity-text\">Humidity</div>\n";

ptr +="<div class=\"side-by-side humidity\">";

ptr +=(int)Humiditystat;

ptr +="<span class=\"superscript\">%</span></div>\n";

ptr +="</div>\n";

ptr +="</div>\n";

ptr +="</body>\n";

ptr +="</html>\n";

return ptr;

}

Improvement to the Code – Auto Page Refresh

One of the improvements you can do with our code is refreshing the page automatically in order to update the sensor value.

With the addition of a single meta tag into your HTML document, you can instruct the browser to automatically reload the page at a provided interval.

<meta http-equiv="refresh" content="2" >

Place this code in the the <head> tag of your document, this meta tag will instruct the browser to refresh every two seconds. Pretty nifty!

Dynamically load Sensor Data with AJAX

Refreshing a web page isn’t too practical if you have a heavy web page. A better method is to use [Asynchronous Javascript And Xml](https://en.wikipedia.org/wiki/Ajax_(programming)) (AJAX) so that we can request data from the server asynchronously (in the background) without refreshing the page.

The [XMLHttpRequest](https://en.wikipedia.org/wiki/XMLHttpRequest) object within JavaScript is commonly used to execute AJAX on webpages. It performs the silent GET request on the server and updates the element on the page. AJAX is not a new technology, or different language, just existing technologies used in new ways. Besides this, AJAX also makes it possible to

* Request data from a server after the page has loaded
* Receive data from a server after the page has loaded
* Send data to a server in the background

Here is the AJAX script that we’ll be using. Place this script just before you close </head> tag.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,200);\n";

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.getElementById(\"webpage\").innerHTML =this.responseText}\n";

ptr +="};\n";

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

ptr +="</script>\n";

The script starts with <script> tag, as AJAX script is nothing but a javascript so, we need to write it in <script> tag. In order for this function to be repeatedly called, we will be using the javascript setInterval() function. It takes two parameters – a function to be executed and time interval (in milliseconds) on how often to execute the function.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,200);\n";

The heart of this script is a loadDoc() function. Inside this function, an XMLHttpRequest() object is created. This object is used to request data from a web server.

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

The xhttp.onreadystatechange() function is called every time the readyState changes. The readyState property holds the status of the XMLHttpRequest. It has one of the following values.

* 0: request not initialized
* 1: server connection established
* 2: request received
* 3: processing request
* 4: request finished and response is ready

The status property holds the status of the XMLHttpRequest object. It has one of the following values.

* 200: “OK”
* 403: “Forbidden”
* 404: “Page not found”

When readyState is 4 and status is 200, the response is ready. Now, the content of element with id webpage (div holding values of temperature & humidity) is updated.

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.getElementById(\"webpage\").innerHTML =this.responseText}\n";

ptr +="};\n";

The HTTP request is then initiated via the open() and send() functions.

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

**Create A Simple ESP32 Weather Station With BME280**



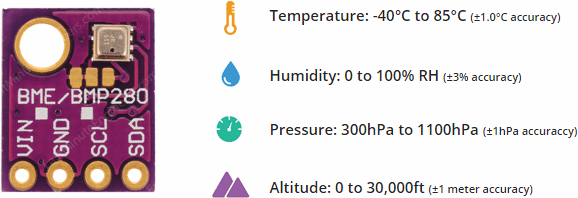
Don’t let the smartphone weather apps or commercial weather stations(that feeds you with data from stations based miles away) ruin your outdoor plans. With this IoT project you can be your own weatherman!

This project uses ESP32 as the control device that easily connects to existing WiFi network & creates a Web Server. When any connected device accesses this web server, ESP32 reads in temperature, humidity, barometric pressure & altitude from BME280 & sends it to the web browser of that device with a nice interface. Excited? Let’s get started!

BME280 Temperature, Humidity and Pressure Sensor

First, let’s take a quick look at the BME280 module.

BME280 is the next-generation digital temperature, humidity and pressure sensor manufactured by Bosch. It’s a successor to sensors like BMP180, BMP085 or BMP183.



The operating voltage of the BME280 module is from 3.3V to 5V – Perfect for interfacing with 3.3V microcontrollers like ESP32.

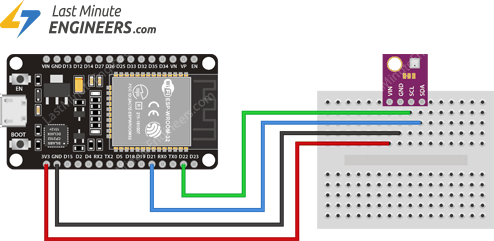
The module features a simple two-wire I2C interface for communication. The default I2C address of the BME280 module is 0x76 and can be changed to 0x77 easily with [this procedure](https://lastminuteengineers.com/bme280-arduino-tutorial/#procedure-to-change-i2c-address).

Wiring BME280 Sensor to ESP32

Connections are fairly simple. Start by connecting VIN pin to the 3.3V output on the ESP32 and connect GND to ground.

Next, Connect the SCL pin to the I2C clock D22 pin on your ESP32 and connect the SDA pin to the I2C data D21 pin on your ESP32.

The following diagram shows you how to wire everything.

Wiring ESP32 & BME280 Temperature Humidity Pressure Sensor

Preparing the Arduino IDE

There’s an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE. Follow below tutorial to prepare your Arduino IDE to work with the ESP32, if you haven’t already.

[](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

[Insight Into ESP32 Features & Using It With Arduino IDE](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

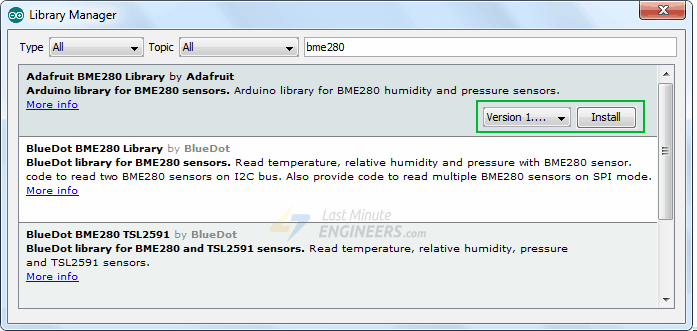
Few years back, ESP8266 took the embedded IoT world by storm. For less than $3, you could get a programmable, WiFi-enabled microcontroller being able to...

Installing Library For BME280

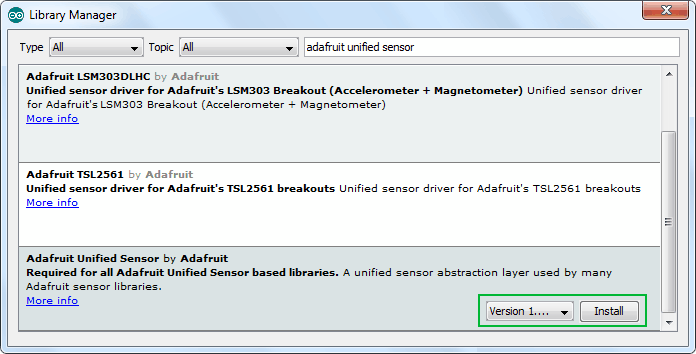
Communicating with a BME280 module is a bunch of work. Fortunately, [Adafruit BME280 Library](https://github.com/adafruit/Adafruit_BME280_Library) was written to hide away all the complexities so that we can issue simple commands to read the temperature, relative humidity & barometric pressure data.

To install the library navigate to the Arduino IDE > Sketch > Include Library > Manage Libraries… Wait for Library Manager to download libraries index and update list of installed libraries.

Filter your search by typing ‘bme280’. There should be a couple entries. Look for Adafruit BME280 Library by Adafruit. Click on that entry, and then select Install.



The BME280 sensor library uses the [Adafruit Sensor support backend](https://github.com/adafruit/Adafruit_Sensor). So, search the library manager for Adafruit Unified Sensor and install that too (you may have to scroll a bit)



Displaying Temperature, Humidity, Pressure & Altitude On ESP32 Web Server

Now, we are going to configure our ESP32 into Station (STA) mode, and create a web server to serve up web pages to any connected client under existing network.

If you want to learn about creating a web server with ESP32 in AP/STA mode, check this tutorial out.

[](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

[Create A Simple ESP32 Web Server In Arduino IDE](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

The newly launched successor of ESP8266 - the ESP32 has been a growing star among IoT or WiFi-related projects. It’s an extremely cost-effective WiFi module...

Before you head for uploading the sketch, you need to make one change to make it work for you. You need to modify the following two variables with your network credentials, so that ESP32 can establish a connection with existing network.

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

Once you are done, go ahead and try the sketch out.

#include <WiFi.h>

#include <WebServer.h>

#include <Wire.h>

#include <Adafruit\_Sensor.h>

#include <Adafruit\_BME280.h>

#define SEALEVELPRESSURE\_HPA (1013.25)

Adafruit\_BME280 bme;

float temperature, humidity, pressure, altitude;

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

WebServer server(80);

void setup() {

Serial.begin(115200);

delay(100);

bme.begin(0x76);

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

server.on("/", handle\_OnConnect);

server.onNotFound(handle\_NotFound);

server.begin();

Serial.println("HTTP server started");

}

void loop() {

server.handleClient();

}

void handle\_OnConnect() {

temperature = bme.readTemperature();

humidity = bme.readHumidity();

pressure = bme.readPressure() / 100.0F;

altitude = bme.readAltitude(SEALEVELPRESSURE\_HPA);

server.send(200, "text/html", SendHTML(temperature,humidity,pressure,altitude));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

String SendHTML(float temperature,float humidity,float pressure,float altitude){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Weather Station</title>\n";

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Station</h1>\n";

ptr +="<p>Temperature: ";

ptr +=temperature;

ptr +="&deg;C</p>";

ptr +="<p>Humidity: ";

ptr +=humidity;

ptr +="%</p>";

ptr +="<p>Pressure: ";

ptr +=pressure;

ptr +="hPa</p>";

ptr +="<p>Altitude: ";

ptr +=altitude;

ptr +="m</p>";

ptr +="</div>\n";

ptr +="</body>\n";

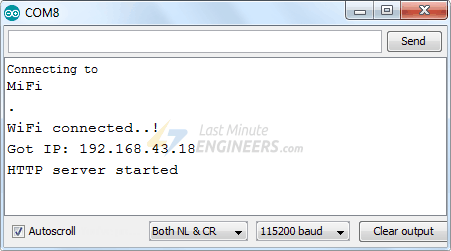
ptr +="</html>\n";

return ptr;

}

Accessing the Web Server

After uploading the sketch, open the Serial Monitor at a baud rate of 115200. And press the EN button on ESP32. If everything is OK, it will output the dynamic IP address obtained from your router and show HTTP server started message.



Next, load up a browser and point it to the IP address shown on the serial monitor. The ESP32 should serve up a web page showing temperature, humidity, pressure and altitude from BME280.



Detailed Code Explanation

The sketch starts by including following libraries.

* WiFi.h library provides ESP32 specific WiFi methods we are calling to connect to network.
* WebServer.h library has some methods available that will help us setting up a server and handle incoming HTTP requests without needing to worry about low level implementation details.
* Wire.h library communicates with any I2C device not just BME280.
* Adafruit\_BME280.h & Adafruit\_Sensor.h libraries are hardware-specific libraries which handles lower-level functions.

#include <WiFi.h>

#include <WebServer.h>

#include <Wire.h>

#include <Adafruit\_Sensor.h>

#include <Adafruit\_BME280.h>

Next we create an object of the sensor and variables to store temperature, humidity, pressure and altitude.

#define SEALEVELPRESSURE\_HPA (1013.25)

Adafruit\_BME280 bme;

float temperature, humidity, pressure, altitude;

As we are configuring ESP32 in Station (STA) mode, it will join existing WiFi network. Hence, we need to provide it with your network’s SSID & Password. Next we start web server at port 80.

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

WebServer server(80);

Inside Setup() Function

Inside Setup() Function we configure our HTTP server before actually running it.

First of all, we initialize serial communication with PC and initialize BME object using begin() function. It initializes I2C interface with given I2C Address(0x76) and checks if the chip ID is correct. It then resets the chip using soft-reset & waits for the sensor for calibration after wake-up.

Serial.begin(115200);

delay(100);

bme.begin(0x76);

Now, we need to join the WiFi network using WiFi.begin() function. The function takes SSID (Network Name) and password as a parameter.

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

While the ESP32 tries to connect to the network, we can check the connectivity status with WiFi.status() function.

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Once the ESP32 is connected to the network, the sketch prints the IP address assigned to ESP32 by displaying WiFi.localIP() value on serial monitor.

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

In order to handle incoming HTTP requests, we need to specify which code to execute when a URL is hit. To do so, we use on method. This method takes two parameters. First one is a URL path and second one is the name of function which we want to execute when that URL is hit.

The code below indicates that when a server receives an HTTP request on the root (/) path, it will trigger the handle\_OnConnect function. Note that the URL specified is a relative path.

server.on("/", handle\_OnConnect);

We haven’t specified what the server should do if the client requests any URL other than specified with server.on. It should respond with an HTTP status 404 (Not Found) and a message for the user. We put this in a function as well, and use server.onNotFound to tell it that it should execute it when it receives a request for a URL that wasn’t specified with server.on

server.onNotFound(handle\_NotFound);

Now, to start our server, we call the begin method on the server object.

server.begin();

Serial.println("HTTP server started");

Inside Loop() Function

To handle the actual incoming HTTP requests, we need to call the handleClient() method on the server object.

server.handleClient();

Next, we need to create a function we attached to root (/) URL with server.on Remember?

At the start of this function, we get the temperature, humidity, pressure & altitude readings from the sensor. In order to respond to the HTTP request, we use the send method. Although the method can be called with a different set of arguments, its simplest form consists of the HTTP response code, the content type and the content.

In our case, we are sending the code 200 (one of the [HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)), which corresponds to the OK response. Then, we are specifying the content type as “text/html“, and finally we are calling SendHTML() custom function which creates a dynamic HTML page containing temperature, humidity, pressure & altitude readings.

void handle\_OnConnect() {

temperature = bme.readTemperature();

humidity = bme.readHumidity();

pressure = bme.readPressure() / 100.0F;

altitude = bme.readAltitude(SEALEVELPRESSURE\_HPA);

server.send(200, "text/html", SendHTML(temperature,humidity,pressure,altitude));

}

Likewise, we need to create a function to handle 404 Error page.

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

Displaying the HTML Web Page

SendHTML() function is responsible for generating a web page whenever the ESP32 web server gets a request from a web client. It merely concatenates HTML code into a big string and returns to the server.send() function we discussed earlier. The function takes temperature, humidity, pressure & altitude readings as a parameter to dynamically generate the HTML content.

The first text you should always send is the <!DOCTYPE> declaration that indicates that we’re sending HTML code.

String SendHTML(float temperature,float humidity,float pressure,float altitude){

String ptr = "<!DOCTYPE html> <html>\n";

Next, the <meta> viewport element makes the web page responsive in any web browser, while title tag sets the title of the page.

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Weather Station</title>\n";

Styling the Web Page

Next, we have some CSS to style the web page appearance. We choose the Helvetica font, define the content to be displayed as an inline-block and aligned at the center.

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

Following code sets color, font and margin around the body, H1 and p tags.

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

Setting the Web Page Heading

Next, heading of the web page is set; you can change this text to anything that suits your application.

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Weather Station</h1>\n";

Displaying Readings on Web Page

To dynamically display temperature, humidity, pressure & altitude readings, we put those values in paragraph tag. To display degree symbol, we use HTML entity &deg;

ptr +="<p>Temperature: ";

ptr +=temperature;

ptr +="&deg;C</p>";

ptr +="<p>Humidity: ";

ptr +=humidity;

ptr +="%</p>";

ptr +="<p>Pressure: ";

ptr +=pressure;

ptr +="hPa</p>";

ptr +="<p>Altitude: ";

ptr +=altitude;

ptr +="m</p>";

ptr +="</div>\n";

ptr +="</body>\n";

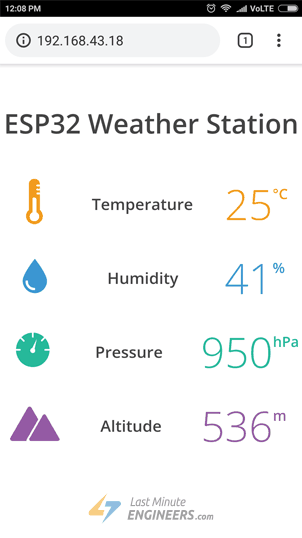
ptr +="</html>\n";

return ptr;

}

Styling Web Page to Look More Professional

Programmers like us are often intimidated by design – but a little effort can make your web page look more attractive and professional. Below screenshot will give you a basic idea of what we are going to do.



Pretty amazing, Right? Without further ado, let’s apply some style to our previous HTML page. To start with, copy-paste below code to replace SendHTML() function from the sketch above.

String SendHTML(float temperature,float humidity,float pressure,float altitude){

String ptr = "<!DOCTYPE html>";

ptr +="<html>";

ptr +="<head>";

ptr +="<title>ESP32 Weather Station</title>";

ptr +="<meta name='viewport' content='width=device-width, initial-scale=1.0'>";

ptr +="<link href='https://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>";

ptr +="<style>";

ptr +="html { font-family: 'Open Sans', sans-serif; display: block; margin: 0px auto; text-align: center;color: #444444;}";

ptr +="body{margin: 0px;} ";

ptr +="h1 {margin: 50px auto 30px;} ";

ptr +=".side-by-side{display: table-cell;vertical-align: middle;position: relative;}";

ptr +=".text{font-weight: 600;font-size: 19px;width: 200px;}";

ptr +=".reading{font-weight: 300;font-size: 50px;padding-right: 25px;}";

ptr +=".temperature .reading{color: #F29C1F;}";

ptr +=".humidity .reading{color: #3B97D3;}";

ptr +=".pressure .reading{color: #26B99A;}";

ptr +=".altitude .reading{color: #955BA5;}";

ptr +=".superscript{font-size: 17px;font-weight: 600;position: absolute;top: 10px;}";

ptr +=".data{padding: 10px;}";

ptr +=".container{display: table;margin: 0 auto;}";

ptr +=".icon{width:65px}";

ptr +="</style>";

ptr +="</head>";

ptr +="<body>";

ptr +="<h1>ESP32 Weather Station</h1>";

ptr +="<div class='container'>";

ptr +="<div class='data temperature'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 19.438 54.003'height=54.003px id=Layer\_1 version=1.1 viewBox='0 0 19.438 54.003'width=19.438px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><g><path d='M11.976,8.82v-2h4.084V6.063C16.06,2.715,13.345,0,9.996,0H9.313C5.965,0,3.252,2.715,3.252,6.063v30.982";

ptr +="C1.261,38.825,0,41.403,0,44.286c0,5.367,4.351,9.718,9.719,9.718c5.368,0,9.719-4.351,9.719-9.718";

ptr +="c0-2.943-1.312-5.574-3.378-7.355V18.436h-3.914v-2h3.914v-2.808h-4.084v-2h4.084V8.82H11.976z M15.302,44.833";

ptr +="c0,3.083-2.5,5.583-5.583,5.583s-5.583-2.5-5.583-5.583c0-2.279,1.368-4.236,3.326-5.104V24.257C7.462,23.01,8.472,22,9.719,22";

ptr +="s2.257,1.01,2.257,2.257V39.73C13.934,40.597,15.302,42.554,15.302,44.833z'fill=#F29C21 /></g></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Temperature</div>";

ptr +="<div class='side-by-side reading'>";

ptr +=(int)temperature;

ptr +="<span class='superscript'>&deg;C</span></div>";

ptr +="</div>";

ptr +="<div class='data humidity'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 29.235 40.64'height=40.64px id=Layer\_1 version=1.1 viewBox='0 0 29.235 40.64'width=29.235px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><path d='M14.618,0C14.618,0,0,17.95,0,26.022C0,34.096,6.544,40.64,14.618,40.64s14.617-6.544,14.617-14.617";

ptr +="C29.235,17.95,14.618,0,14.618,0z M13.667,37.135c-5.604,0-10.162-4.56-10.162-10.162c0-0.787,0.638-1.426,1.426-1.426";

ptr +="c0.787,0,1.425,0.639,1.425,1.426c0,4.031,3.28,7.312,7.311,7.312c0.787,0,1.425,0.638,1.425,1.425";

ptr +="C15.093,36.497,14.455,37.135,13.667,37.135z'fill=#3C97D3 /></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Humidity</div>";

ptr +="<div class='side-by-side reading'>";

ptr +=(int)humidity;

ptr +="<span class='superscript'>%</span></div>";

ptr +="</div>";

ptr +="<div class='data pressure'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 40.542 40.541'height=40.541px id=Layer\_1 version=1.1 viewBox='0 0 40.542 40.541'width=40.542px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><g><path d='M34.313,20.271c0-0.552,0.447-1,1-1h5.178c-0.236-4.841-2.163-9.228-5.214-12.593l-3.425,3.424";

ptr +="c-0.195,0.195-0.451,0.293-0.707,0.293s-0.512-0.098-0.707-0.293c-0.391-0.391-0.391-1.023,0-1.414l3.425-3.424";

ptr +="c-3.375-3.059-7.776-4.987-12.634-5.215c0.015,0.067,0.041,0.13,0.041,0.202v4.687c0,0.552-0.447,1-1,1s-1-0.448-1-1V0.25";

ptr +="c0-0.071,0.026-0.134,0.041-0.202C14.39,0.279,9.936,2.256,6.544,5.385l3.576,3.577c0.391,0.391,0.391,1.024,0,1.414";

ptr +="c-0.195,0.195-0.451,0.293-0.707,0.293s-0.512-0.098-0.707-0.293L5.142,6.812c-2.98,3.348-4.858,7.682-5.092,12.459h4.804";

ptr +="c0.552,0,1,0.448,1,1s-0.448,1-1,1H0.05c0.525,10.728,9.362,19.271,20.22,19.271c10.857,0,19.696-8.543,20.22-19.271h-5.178";

ptr +="C34.76,21.271,34.313,20.823,34.313,20.271z M23.084,22.037c-0.559,1.561-2.274,2.372-3.833,1.814";

ptr +="c-1.561-0.557-2.373-2.272-1.815-3.833c0.372-1.041,1.263-1.737,2.277-1.928L25.2,7.202L22.497,19.05";

ptr +="C23.196,19.843,23.464,20.973,23.084,22.037z'fill=#26B999 /></g></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Pressure</div>";

ptr +="<div class='side-by-side reading'>";

ptr +=(int)pressure;

ptr +="<span class='superscript'>hPa</span></div>";

ptr +="</div>";

ptr +="<div class='data altitude'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 58.422 40.639'height=40.639px id=Layer\_1 version=1.1 viewBox='0 0 58.422 40.639'width=58.422px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><g><path d='M58.203,37.754l0.007-0.004L42.09,9.935l-0.001,0.001c-0.356-0.543-0.969-0.902-1.667-0.902";

ptr +="c-0.655,0-1.231,0.32-1.595,0.808l-0.011-0.007l-0.039,0.067c-0.021,0.03-0.035,0.063-0.054,0.094L22.78,37.692l0.008,0.004";

ptr +="c-0.149,0.28-0.242,0.594-0.242,0.934c0,1.102,0.894,1.995,1.994,1.995v0.015h31.888c1.101,0,1.994-0.893,1.994-1.994";

ptr +="C58.422,38.323,58.339,38.024,58.203,37.754z'fill=#955BA5 /><path d='M19.704,38.674l-0.013-0.004l13.544-23.522L25.13,1.156l-0.002,0.001C24.671,0.459,23.885,0,22.985,0";

ptr +="c-0.84,0-1.582,0.41-2.051,1.038l-0.016-0.01L20.87,1.114c-0.025,0.039-0.046,0.082-0.068,0.124L0.299,36.851l0.013,0.004";

ptr +="C0.117,37.215,0,37.62,0,38.059c0,1.412,1.147,2.565,2.565,2.565v0.015h16.989c-0.091-0.256-0.149-0.526-0.149-0.813";

ptr +="C19.405,39.407,19.518,39.019,19.704,38.674z'fill=#955BA5 /></g></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Altitude</div>";

ptr +="<div class='side-by-side reading'>";

ptr +=(int)altitude;

ptr +="<span class='superscript'>m</span></div>";

ptr +="</div>";

ptr +="</div>";

ptr +="</body>";

ptr +="</html>";

return ptr;

}

If you try to compare this function with the previous one, you’ll come to know that they are similar except these changes.

* We have used Google commissioned [Open Sans](https://fonts.google.com/specimen/Open+Sans) web font for our web page. Note that you cannot see Google font, without active internet connection on the device. Google fonts are loaded on the fly.

ptr +="<link href='https://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>";

* The icons used to display temperature, humidity, pressure & altitude readings are actually a [Scalable Vector Graphics](https://en.wikipedia.org/wiki/Scalable_Vector_Graphics) (SVG) defined in <svg> tag. Creating SVG doesn’t require any special programming skills. You can use [Google SVG Editor](https://svg-edit.github.io/svgedit/releases/latest/editor/svg-editor.html) for creating graphics for your page. We have used these SVG icons.

Improvement to the Code – Auto Page Refresh

One of the improvements you can do with our code is refreshing the page automatically in order to update the sensor value.

With the addition of a single meta tag into your HTML document, you can instruct the browser to automatically reload the page at a provided interval.

<meta http-equiv="refresh" content="2" >

Place this code in the the <head> tag of your document, this meta tag will instruct the browser to refresh every two seconds. Pretty nifty!

Dynamically load Sensor Data with AJAX

Refreshing a web page isn’t too practical if you have a heavy web page. A better method is to use [Asynchronous Javascript And Xml](https://en.wikipedia.org/wiki/Ajax_(programming)) (AJAX) so that we can request data from the server asynchronously (in the background) without refreshing the page.

The [XMLHttpRequest](https://en.wikipedia.org/wiki/XMLHttpRequest) object within JavaScript is commonly used to execute AJAX on webpages. It performs the silent GET request on the server and updates the element on the page. AJAX is not a new technology, or different language, just existing technologies used in new ways. Besides this, AJAX also makes it possible to

* Request data from a server after the page has loaded
* Receive data from a server after the page has loaded
* Send data to a server in the background

Here is the AJAX script that we’ll be using. Place this script just before you close </head> tag.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,1000);\n";

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.body.innerHTML =this.responseText}\n";

ptr +="};\n";

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

ptr +="</script>\n";

The script starts with <script> tag. As AJAX script is nothing but a javascript, we need to write it in <script> tag. In order for this function to be repeatedly called, we will be using the javascript setInterval() function. It takes two parameters – a function to be executed and time interval (in milliseconds) on how often to execute the function.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,1000);\n";

The heart of this script is a loadDoc() function. Inside this function, an XMLHttpRequest() object is created. This object is used to request data from a web server.

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

The xhttp.onreadystatechange() function is called every time the readyState changes. The readyState property holds the status of the XMLHttpRequest. It has one of the following values.

* 0: request not initialized
* 1: server connection established
* 2: request received
* 3: processing request
* 4: request finished and response is ready

The status property holds the status of the XMLHttpRequest object. It has one of the following values.

* 200: “OK”
* 403: “Forbidden”
* 404: “Page not found”

When readyState is 4 and status is 200, the response is ready. Now, the content of body (holding temperature readings) is updated.

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.body.innerHTML =this.responseText}\n";

ptr +="};\n";

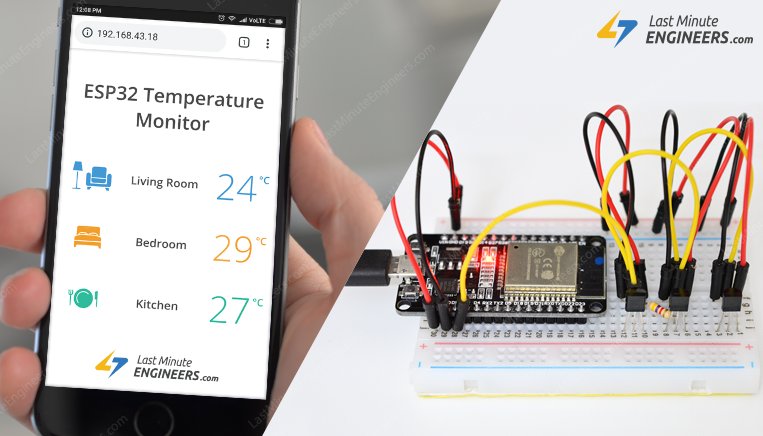
The HTTP request is then initiated via the open() and send() functions.

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

Interface Multiple DS18B20s with ESP32 & Display Values Using Web Server



Have you ever wanted to have sensors scattered all around your house and garden reporting their temperature regularly to a central server? Then, this IoT project might be the solid launching point for you!

This project uses ESP32 as the control device that easily connects to existing WiFi network & creates a Web Server. When any connected device accesses this web server, ESP32 reads in temperature from multiple DS18B20 Temperature sensors & sends it to the web browser of that device with a nice interface. Excited? Let’s get started!

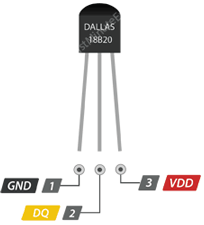
Multiple DS18B20s On Single Bus

One of the biggest features of DS18B20 is that multiple DS18B20s can coexist on the same 1-Wire bus. As each DS18B20 has a unique 64-bit serial code burned in at the factory, it’s easier to differentiate them from one another.

This feature can be a huge advantage when you want to control many DS18B20s distributed over a large area. In this tutorial we are going to do the same.

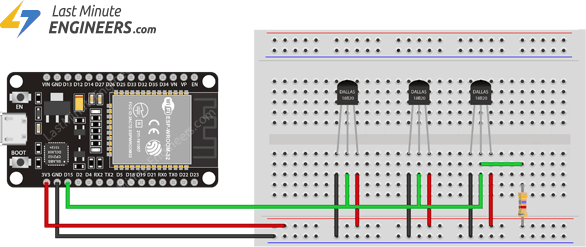
Wiring Multiple DS18B20 Sensors to ESP32

Connecting DS18B20 sensors to ESP32 is fairly simple.



Start by connecting all the DS18B20s in parallel i.e. common all the VDD pins, GND pins & signal pins. Then connect VDD to the 3.3V out on ESP32, GND to ground and connect signal pin to digital pin 15 on ESP32.

Next, you’ll need to add one 4.7k pull-up resistor for whole bus between the signal and power pin to keep the data transfer stable.

Wiring Multiple DS18B20 Temperature Sensors to ESP32

Preparing the Arduino IDE

There’s an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE. Follow below tutorial to prepare your Arduino IDE to work with the ESP32, if you haven’t already.

[](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

[Insight Into ESP32 Features & Using It With Arduino IDE](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

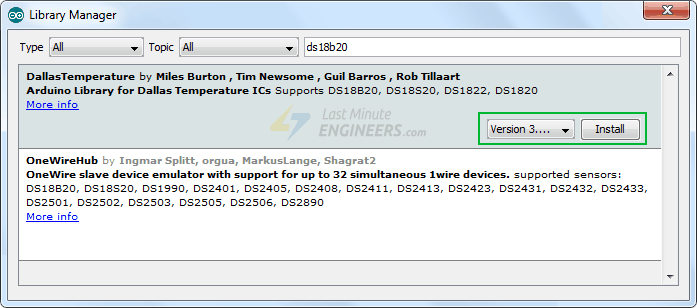
Few years back, ESP8266 took the embedded IoT world by storm. For less than $3, you could get a programmable, WiFi-enabled microcontroller being able to...

Installing Library For DS18B20

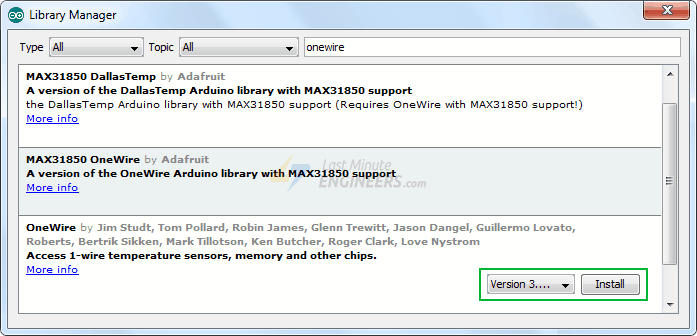
The Dallas 1-Wire protocol is somewhat complex, and requires a bunch of code to parse out the communication. To hide away this unnecessary complexity we will install [DallasTemperature.h](https://github.com/milesburton/Arduino-Temperature-Control-Library) library so that we can issue simple commands to get temperature readings from the sensor.

To install the library navigate to the Arduino IDE > Sketch > Include Library > Manage Libraries… Wait for Library Manager to download libraries index and update list of installed libraries.

Filter your search by typing ‘ds18b20’. There should be a couple entries. Look for DallasTemperature by Miles Burton. Click on that entry, and then select Install.



This Dallas Temperature library is a hardware-specific library which handles lower-level functions. It needs to be paired with [One Wire](https://playground.arduino.cc/Learning/OneWire) Library to communicate with any one-wire device not just DS18B20. Install this library as well.



Finding Addresses Of DS18B20s On Bus

We know that each DS18B20 has a unique 64-bit address assigned to it to differentiate them from one another. First, we’ll find that address to label each sensor accordingly. The address can then be used to read each sensor individually.

The following sketch detects all the DS18B20s present on the bus and prints their one-wire address on the serial monitor.

You can wire just one sensor at a time to find its address (or successively add a new sensor) so that you’re able to identify each one by its address. Then, you can label each sensor.

#include <OneWire.h>

#include <DallasTemperature.h>

// Data wire is plugged into port 15 on the ESP32

#define ONE\_WIRE\_BUS 15

// Setup a oneWire instance to communicate with any OneWire devices

OneWire oneWire(ONE\_WIRE\_BUS);

// Pass our oneWire reference to Dallas Temperature.

DallasTemperature sensors(&oneWire);

// variable to hold device addresses

DeviceAddress Thermometer;

int deviceCount = 0;

void setup(void)

{

// start serial port

Serial.begin(115200);

// Start up the library

sensors.begin();

// locate devices on the bus

Serial.println("Locating devices...");

Serial.print("Found ");

deviceCount = sensors.getDeviceCount();

Serial.print(deviceCount, DEC);

Serial.println(" devices.");

Serial.println("");

Serial.println("Printing addresses...");

for (int i = 0; i < deviceCount; i++)

{

Serial.print("Sensor ");

Serial.print(i+1);

Serial.print(" : ");

sensors.getAddress(Thermometer, i);

printAddress(Thermometer);

}

}

void loop(void)

{ }

void printAddress(DeviceAddress deviceAddress)

{

for (uint8\_t i = 0; i < 8; i++)

{

Serial.print("0x");

if (deviceAddress[i] < 0x10) Serial.print("0");

Serial.print(deviceAddress[i], HEX);

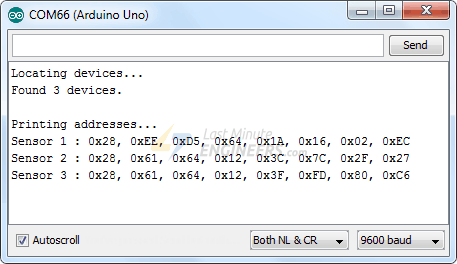
if (i < 7) Serial.print(", ");

}

Serial.println("");

}

Now, open the serial monitor. You should get something as follows.



Copy all the addresses as we need them in out next sketch.

Create ESP32 Web Server using WiFi Station (STA) mode

Now, we are going to configure our ESP32 into Station (STA) mode, and create a web server to serve up web pages to any connected client under existing network.

If you want to learn about creating a web server with ESP32 in AP/STA mode, check this tutorial out.

[](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

[Create A Simple ESP32 Web Server In Arduino IDE](https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/)

The newly launched successor of ESP8266 - the ESP32 has been a growing star among IoT or WiFi-related projects. It’s an extremely cost-effective WiFi module...

Before you head for uploading the sketch, you need to make some changes to make it work for you.

* You need to modify the following two variables with your network credentials, so that ESP32 can establish a connection with existing network.
* const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

* Before serving up a web page ESP32 reads the temperature from each DS18B20 by its address so, you need to change the addresses of DS18B20s with the one you’ve found in previous sketch.
* uint8\_t sensor1[8] = { 0x28, 0xEE, 0xD5, 0x64, 0x1A, 0x16, 0x02, 0xEC };
* uint8\_t sensor2[8] = { 0x28, 0x61, 0x64, 0x12, 0x3C, 0x7C, 0x2F, 0x27 };

uint8\_t sensor3[8] = { 0x28, 0x61, 0x64, 0x12, 0x3F, 0xFD, 0x80, 0xC6 };

Once you are done, go ahead and try the sketch out.

#include <WiFi.h>

#include <WebServer.h>

#include <OneWire.h>

#include <DallasTemperature.h>

// Data wire is plugged into port 15 on the ESP32

#define ONE\_WIRE\_BUS 15

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas temperature ICs)

OneWire oneWire(ONE\_WIRE\_BUS);

// Pass our oneWire reference to Dallas Temperature.

DallasTemperature sensors(&oneWire);

uint8\_t sensor1[8] = { 0x28, 0xEE, 0xD5, 0x64, 0x1A, 0x16, 0x02, 0xEC };

uint8\_t sensor2[8] = { 0x28, 0x61, 0x64, 0x12, 0x3C, 0x7C, 0x2F, 0x27 };

uint8\_t sensor3[8] = { 0x28, 0x61, 0x64, 0x12, 0x3F, 0xFD, 0x80, 0xC6 };

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

WebServer server(80);

float tempSensor1, tempSensor2, tempSensor3;

void setup() {

Serial.begin(115200);

delay(100);

sensors.begin();

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

server.on("/", handle\_OnConnect);

server.onNotFound(handle\_NotFound);

server.begin();

Serial.println("HTTP server started");

}

void loop() {

server.handleClient();

}

void handle\_OnConnect() {

sensors.requestTemperatures();

tempSensor1 = sensors.getTempC(sensor1); // Gets the values of the temperature

tempSensor2 = sensors.getTempC(sensor2); // Gets the values of the temperature

tempSensor3 = sensors.getTempC(sensor3); // Gets the values of the temperature

server.send(200, "text/html", SendHTML(tempSensor1,tempSensor2,tempSensor3));

}

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

String SendHTML(float tempSensor1,float tempSensor2,float tempSensor3){

String ptr = "<!DOCTYPE html> <html>\n";

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Temperature Monitor</title>\n";

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Temperature Monitor</h1>\n";

ptr +="<p>Living Room: ";

ptr +=tempSensor1;

ptr +="&deg;C</p>";

ptr +="<p>Bedroom: ";

ptr +=tempSensor2;

ptr +="&deg;C</p>";

ptr +="<p>Kitchen: ";

ptr +=tempSensor3;

ptr +="&deg;C</p>";

ptr +="</div>\n";

ptr +="</body>\n";

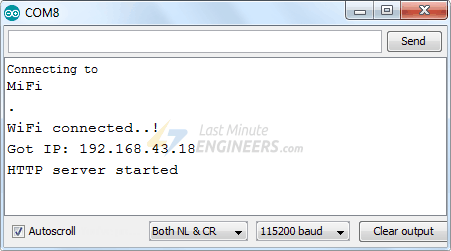
ptr +="</html>\n";

return ptr;

}

Accessing the Web Server

After uploading the sketch, open the Serial Monitor at a baud rate of 115200. And press the EN button on ESP32. If everything is OK, it will output the dynamic IP address obtained from your router and show HTTP server started message.



Next, load up a browser and point it to the IP address shown on the serial monitor. The ESP32 should serve up a web page showing temperatures from all the DS18B20s.



Detailed Code Explanation

The sketch starts by including following libraries.

* WiFi.h library provides ESP32 specific WiFi methods we are calling to connect to network.
* WebServer.h library has some methods available that will help us setting up a server and handle incoming HTTP requests without needing to worry about low level implementation details.
* DallasTemperature.h library is a hardware-specific library which handles lower-level functions. It needs to be paired with [One Wire](https://playground.arduino.cc/Learning/OneWire) Library, in order to make it work.
* OneWire.h library communicates with any one-wire device not just DS18B20.

#include <WiFi.h>

#include <WebServer.h>

#include <OneWire.h>

#include <DallasTemperature.h>

Next we create the instances needed for the temperature sensor and variables to store temperature readings. The temperature sensor is connected to GPIO15.

// Data wire is plugged into port 15 on the ESP32

#define ONE\_WIRE\_BUS 15

// Setup a oneWire instance to communicate with any OneWire devices

OneWire oneWire(ONE\_WIRE\_BUS);

// Pass our oneWire reference to Dallas Temperature.

DallasTemperature sensors(&oneWire);

float tempSensor1, tempSensor2, tempSensor3;

Next, we enter the addresses that are found previously for each temperature sensor. In our case, we have the following.

uint8\_t sensor1[8] = { 0x28, 0xEE, 0xD5, 0x64, 0x1A, 0x16, 0x02, 0xEC };

uint8\_t sensor2[8] = { 0x28, 0x61, 0x64, 0x12, 0x3C, 0x7C, 0x2F, 0x27 };

uint8\_t sensor3[8] = { 0x28, 0x61, 0x64, 0x12, 0x3F, 0xFD, 0x80, 0xC6 };

As we are configuring ESP32 in Station (STA) mode, it will join existing WiFi network. Hence, we need to provide it with your network’s SSID & Password. Next we start web server at port 80.

/\*Put your SSID & Password\*/

const char\* ssid = "YourNetworkName"; // Enter SSID here

const char\* password = "YourPassword"; //Enter Password here

WebServer server(80);

Inside Setup() Function

Inside Setup() Function we configure our HTTP server before actually running it. First of all, we initialize serial communication with PC and initialize DallasTemperature object using begin() function. It initializes the bus and detects all the DS18B20s present on it. Each sensor is then assigned with an index and set bit resolution to 12-bit.

Serial.begin(115200);

delay(100);

sensors.begin();

Now, we need to join the WiFi network using WiFi.begin() function. The function takes SSID (Network Name) and password as a parameter.

Serial.println("Connecting to ");

Serial.println(ssid);

//connect to your local wi-fi network

WiFi.begin(ssid, password);

While the ESP32 tries to connect to the network, we can check the connectivity status with WiFi.status() function.

//check wi-fi is connected to wi-fi network

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.print(".");

}

Once the ESP32 is connected to the network, the sketch prints the IP address assigned to ESP32 by displaying WiFi.localIP() value on serial monitor.

Serial.println("");

Serial.println("WiFi connected..!");

Serial.print("Got IP: "); Serial.println(WiFi.localIP());

In order to handle incoming HTTP requests, we need to specify which code to execute when a URL is hit. To do so, we use on method. This method takes two parameters. First one is a URL path and second one is the name of function which we want to execute when that URL is hit.

The code below indicates that when a server receives an HTTP request on the root (/) path, it will trigger the handle\_OnConnect function. Note that the URL specified is a relative path.

server.on("/", handle\_OnConnect);

We haven’t specified what the server should do if the client requests any URL other than specified with server.on. It should respond with an HTTP status 404 (Not Found) and a message for the user. We put this in a function as well, and use server.onNotFound to tell it that it should execute it when it receives a request for a URL that wasn’t specified with server.on

server.onNotFound(handle\_NotFound);

Now, to start our server, we call the begin method on the server object.

server.begin();

Serial.println("HTTP server started");

Inside Loop() Function

To handle the actual incoming HTTP requests, we need to call the handleClient() method on the server object.

server.handleClient();

Next, we need to create a function we attached to root (/) URL with server.on Remember?

At the start of this function, we get the temperature reading from each sensor. In order to respond to the HTTP request, we use the send method. Although the method can be called with a different set of arguments, its simplest form consists of the HTTP response code, the content type and the content.

In our case, we are sending the code 200 (one of the [HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)), which corresponds to the OK response. Then, we are specifying the content type as “text/html“, and finally we are calling SendHTML() custom function which creates a dynamic HTML page containing temperature readings.

void handle\_OnConnect() {

sensors.requestTemperatures();

tempSensor1 = sensors.getTempC(sensor1);

tempSensor2 = sensors.getTempC(sensor2);

tempSensor3 = sensors.getTempC(sensor3);

server.send(200, "text/html", SendHTML(tempSensor1,tempSensor2,tempSensor3));

}

Likewise, we need to create a function to handle 404 Error page.

void handle\_NotFound(){

server.send(404, "text/plain", "Not found");

}

Displaying the HTML Web Page

SendHTML() function is responsible for generating a web page whenever the ESP32 web server gets a request from a web client. It merely concatenates HTML code into a big string and returns to the server.send() function we discussed earlier. The function takes temperature readings as a parameter to dynamically generate the HTML content.

The first text you should always send is the <!DOCTYPE> declaration that indicates that we’re sending HTML code.

String SendHTML(float tempSensor1,float tempSensor2,float tempSensor3){

String ptr = "<!DOCTYPE html> <html>\n";

Next, the <meta> viewport element makes the web page responsive in any web browser, while title tag sets the title of the page.

ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-scalable=no\">\n";

ptr +="<title>ESP32 Temperature Monitor</title>\n";

Styling the Web Page

Next, we have some CSS to style the web page appearance. We choose the Helvetica font, define the content to be displayed as an inline-block and aligned at the center.

ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}\n";

Following code then sets color, font and margin around the body, H1 and p tags.

ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;}\n";

ptr +="p {font-size: 24px;color: #444444;margin-bottom: 10px;}\n";

ptr +="</style>\n";

ptr +="</head>\n";

ptr +="<body>\n";

Setting the Web Page Heading

Next, heading of the web page is set; you can change this text to anything that suits your application.

ptr +="<div id=\"webpage\">\n";

ptr +="<h1>ESP32 Temperature Monitor</h1>\n";

Displaying Temperature readings on Web Page

To dynamically display temperature readings, we put those values in paragraph tag. To display degree symbol, we use HTML entity &deg;

ptr +="<p>Living Room: ";

ptr +=tempSensor1;

ptr +="&deg;C</p>";

ptr +="<p>Bedroom: ";

ptr +=tempSensor2;

ptr +="&deg;C</p>";

ptr +="<p>Kitchen: ";

ptr +=tempSensor3;

ptr +="&deg;C</p>";

ptr +="</div>\n";

ptr +="</body>\n";

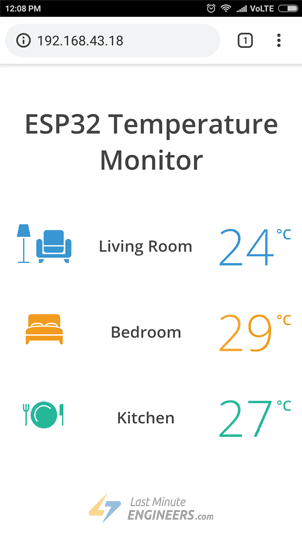
ptr +="</html>\n";

return ptr;

}

Styling Web Page to Look More Professional

Programmers like us are often intimidated by design – but a little effort can make your web page look more attractive and professional. Below screenshot will give you a basic idea of what we are going to do.



Pretty amazing, Right? Without further ado, let’s apply some style to our previous HTML page. To start with, copy-paste below code to replace SendHTML() function from the sketch above.

String SendHTML(float tempSensor1,float tempSensor2,float tempSensor3){

String ptr = "<!DOCTYPE html>";

ptr +="<html>";

ptr +="<head>";

ptr +="<title>ESP32 Temperature Monitor</title>";

ptr +="<meta name='viewport' content='width=device-width, initial-scale=1.0'>";

ptr +="<link href='https://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>";

ptr +="<style>";

ptr +="html { font-family: 'Open Sans', sans-serif; display: block; margin: 0px auto; text-align: center;color: #444444;}";

ptr +="body{margin-top: 50px;} ";

ptr +="h1 {margin: 50px auto 30px;} ";

ptr +=".side-by-side{display: table-cell;vertical-align: middle;position: relative;}";

ptr +=".text{font-weight: 600;font-size: 19px;width: 200px;}";

ptr +=".temperature{font-weight: 300;font-size: 50px;padding-right: 15px;}";

ptr +=".living-room .temperature{color: #3B97D3;}";

ptr +=".bedroom .temperature{color: #F29C1F;}";

ptr +=".kitchen .temperature{color: #26B99A;}";

ptr +=".superscript{font-size: 17px;font-weight: 600;position: absolute;right: -5px;top: 15px;}";

ptr +=".data{padding: 10px;}";

ptr +=".container{display: table;margin: 0 auto;}";

ptr +=".icon{width:82px}";

ptr +="</style>";

ptr +="</head>";

ptr +="<body>";

ptr +="<h1>ESP32 Temperature Monitor</h1>";

ptr +="<div class='container'>";

ptr +="<div class='data living-room'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 65.178 45.699'height=45.699px id=Layer\_1 version=1.1 viewBox='0 0 65.178 45.699'width=65.178px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><polygon fill=#3B97D3 points='8.969,44.261 8.969,16.469 7.469,16.469 7.469,44.261 1.469,44.261 1.469,45.699 14.906,45.699 ";

ptr +="14.906,44.261 '/><polygon fill=#3B97D3 points='13.438,0 3,0 0,14.938 16.438,14.938 '/><polygon fill=#3B97D3 points='29.927,45.699 26.261,45.699 26.261,41.156 32.927,41.156 '/><polygon fill=#3B97D3 points='58.572,45.699 62.239,45.699 62.239,41.156 55.572,41.156 '/><path d='M61.521,17.344c-2.021,0-3.656,1.637-3.656,3.656v14.199H30.594V21c0-2.02-1.638-3.656-3.656-3.656";

ptr +="c-2.02,0-3.657,1.636-3.657,3.656v14.938c0,2.021,1.637,3.655,3.656,3.655H61.52c2.02,0,3.655-1.637,3.655-3.655V21";

ptr +="C65.177,18.98,63.54,17.344,61.521,17.344z'fill=#3B97D3 /><g><path d='M32.052,30.042c0,2.02,1.637,3.656,3.656,3.656h16.688c2.019,0,3.656-1.638,3.656-3.656v-3.844h-24";

ptr +="L32.052,30.042L32.052,30.042z'fill=#3B97D3 /><path d='M52.396,6.781H35.709c-2.02,0-3.656,1.637-3.656,3.656v14.344h24V10.438";

ptr +="C56.053,8.418,54.415,6.781,52.396,6.781z'fill=#3B97D3 /></g></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Living Room</div>";

ptr +="<div class='side-by-side temperature'>";

ptr +=(int)tempSensor1;

ptr +="<span class='superscript'>°C</span></div>";

ptr +="</div>";

ptr +="<div class='data bedroom'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 43.438 35.75'height=35.75px id=Layer\_1 version=1.1 viewBox='0 0 43.438 35.75'width=43.438px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><g><path d='M25.489,14.909H17.95C13.007,14.908,0,15.245,0,20.188v3.688h43.438v-3.688";

ptr +="C43.438,15.245,30.431,14.909,25.489,14.909z'fill=#F29C1F /><polygon fill=#F29C1F points='0,31.25 0,35.75 2.5,35.75 4.5,31.25 38.938,31.25 40.938,35.75 43.438,35.75 43.438,31.25 ";

ptr +="43.438,25.375 0,25.375 '/><path d='M13.584,11.694c-3.332,0-6.033,0.973-6.033,2.175c0,0.134,0.041,0.264,0.105,0.391";

ptr +="c3.745-0.631,7.974-0.709,10.341-0.709h1.538C19.105,12.501,16.613,11.694,13.584,11.694z'fill=#F29C1F /><path d='M30.009,11.694c-3.03,0-5.522,0.807-5.951,1.856h1.425V13.55c2.389,0,6.674,0.081,10.444,0.728";

ptr +="c0.069-0.132,0.114-0.268,0.114-0.408C36.041,12.668,33.34,11.694,30.009,11.694z'fill=#F29C1F /><path d='M6.042,14.088c0-2.224,3.376-4.025,7.542-4.025c3.825,0,6.976,1.519,7.468,3.488h1.488";

ptr +="c0.49-1.97,3.644-3.489,7.469-3.489c4.166,0,7.542,1.801,7.542,4.025c0,0.17-0.029,0.337-0.067,0.502";

ptr +="c1.08,0.247,2.088,0.549,2.945,0.926V3.481C40.429,1.559,38.871,0,36.948,0H6.49C4.568,0,3.009,1.559,3.009,3.481v12.054";

ptr +="c0.895-0.398,1.956-0.713,3.095-0.968C6.069,14.41,6.042,14.251,6.042,14.088z'fill=#F29C1F /></g></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Bedroom</div>";

ptr +="<div class='side-by-side temperature'>";

ptr +=(int)tempSensor2;

ptr +="<span class='superscript'>°C</span></div>";

ptr +="</div>";

ptr +="<div class='data kitchen'>";

ptr +="<div class='side-by-side icon'>";

ptr +="<svg enable-background='new 0 0 48 31.5'height=31.5px id=Layer\_1 version=1.1 viewBox='0 0 48 31.5'width=48px x=0px xml:space=preserve xmlns=http://www.w3.org/2000/svg xmlns:xlink=http://www.w3.org/1999/xlink y=0px><circle cx=24.916 cy=15.75 fill=#26B99A r=15.75 /><path d='M14.917,15.75c0-5.522,4.478-10,10-10c2.92,0,5.541,1.26,7.369,3.257l1.088-1.031";

ptr +="c-2.103-2.285-5.106-3.726-8.457-3.726c-6.351,0-11.5,5.149-11.5,11.5c0,3.127,1.252,5.958,3.277,8.031l1.088-1.031";

ptr +="C16.011,20.945,14.917,18.477,14.917,15.75z'fill=#FFFFFF /><path d='M45.766,2.906c-1.232,0-2.232,1-2.232,2.234v11.203c0,0,2.76,0,3,0v12H48v-12V2.906";

ptr +="C48,2.906,46.035,2.906,45.766,2.906z'fill=#26B99A /><path d='M6.005,2.917v5.184c0,0.975-0.638,1.792-1.516,2.083V2.917H3.021v7.267c-0.878-0.29-1.516-1.107-1.516-2.083";

ptr +="V2.917H0v5.458c0,1.802,1.306,3.291,3.021,3.592v16.376H4.49v-16.38c1.695-0.318,2.979-1.8,2.979-3.588V2.917H6.005z'fill=#26B99A /></svg>";

ptr +="</div>";

ptr +="<div class='side-by-side text'>Kitchen</div>";

ptr +="<div class='side-by-side temperature'>";

ptr +=(int)tempSensor3;

ptr +="<span class='superscript'>°C</span></div>";

ptr +="</div>";

ptr +="</div>";

ptr +="</body>";

ptr +="</html>";

return ptr;

}

If you try to compare this function with the previous one, you’ll come to know that they are similar except these changes.

* We have used Google commissioned [Open Sans](https://fonts.google.com/specimen/Open+Sans) web font for our web page. Note that you cannot see Google font, without active internet connection on the device. Google fonts are loaded on the fly.

ptr +="<link href='https://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>";

* The icons used to display temperature readings are actually a [Scalable Vector Graphics](https://en.wikipedia.org/wiki/Scalable_Vector_Graphics) (SVG) defined in <svg> tag. Creating SVG doesn’t require any special programming skills. You can use [Google SVG Editor](https://svg-edit.github.io/svgedit/releases/latest/editor/svg-editor.html) for creating graphics for your page. We have used these SVG icons.

Improvement to the Code – Auto Page Refresh

One of the improvements you can do with our code is refreshing the page automatically in order to update the sensor value.

With the addition of a single meta tag into your HTML document, you can instruct the browser to automatically reload the page at a provided interval.

<meta http-equiv="refresh" content="2" >

Place this code in the the <head> tag of your document, this meta tag will instruct the browser to refresh every two seconds. Pretty nifty!

Dynamically load Sensor Data with AJAX

Refreshing a web page isn’t too practical if you have a heavy web page. A better method is to use [Asynchronous Javascript And Xml](https://en.wikipedia.org/wiki/Ajax_(programming)) (AJAX) so that we can request data from the server asynchronously (in the background) without refreshing the page.

The [XMLHttpRequest](https://en.wikipedia.org/wiki/XMLHttpRequest) object within JavaScript is commonly used to execute AJAX on webpages. It performs the silent GET request on the server and updates the element on the page. AJAX is not a new technology, or different language, just existing technologies used in new ways. Besides this, AJAX also makes it possible to

* Request data from a server after the page has loaded
* Receive data from a server after the page has loaded
* Send data to a server in the background

Here is the AJAX script that we’ll be using. Place this script just before you close </head> tag.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,1000);\n";

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.body.innerHTML =this.responseText}\n";

ptr +="};\n";

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

ptr +="</script>\n";

The script starts with <script> tag. As AJAX script is nothing but a javascript, we need to write it in <script> tag. In order for this function to be repeatedly called, we will be using the javascript setInterval() function. It takes two parameters – a function to be executed and time interval (in milliseconds) on how often to execute the function.

ptr +="<script>\n";

ptr +="setInterval(loadDoc,1000);\n";

The heart of this script is a loadDoc() function. Inside this function, an XMLHttpRequest() object is created. This object is used to request data from a web server.

ptr +="function loadDoc() {\n";

ptr +="var xhttp = new XMLHttpRequest();\n";

The xhttp.onreadystatechange() function is called every time the readyState changes. The readyState property holds the status of the XMLHttpRequest. It has one of the following values.

* 0: request not initialized
* 1: server connection established
* 2: request received
* 3: processing request
* 4: request finished and response is ready

The status property holds the status of the XMLHttpRequest object. It has one of the following values.

* 200: “OK”
* 403: “Forbidden”
* 404: “Page not found”

When readyState is 4 and status is 200, the response is ready. Now, the content of body (holding temperature readings) is updated.

ptr +="xhttp.onreadystatechange = function() {\n";

ptr +="if (this.readyState == 4 && this.status == 200) {\n";

ptr +="document.body.innerHTML =this.responseText}\n";

ptr +="};\n";

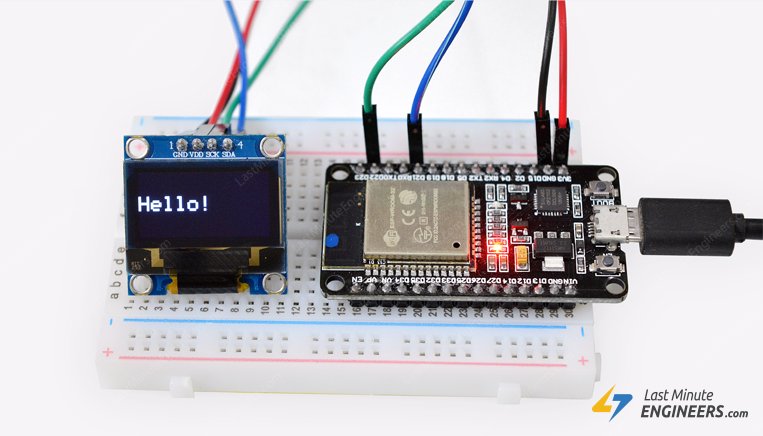
The HTTP request is then initiated via the open() and send() functions.

ptr +="xhttp.open(\"GET\", \"/\", true);\n";

ptr +="xhttp.send();\n";

ptr +="}\n";

Interface OLED Graphic Display Module with ESP32



Want to add little graphic pizzazz to your ESP32 IoT projects? Or maybe you want to display IP address of your ESP32 without resorting to serial output. These super-cool OLED (Organic Light-Emitting Diode) displays might be the perfect fit! They’re super-light, almost paper-thin, theoretically flexible, and produce a brighter and crisper picture.

OLED Display Module Overview

The OLED display module breaks out a small monochrome OLED display. It’s 128 pixels wide and 64 pixels tall, measuring 0.96″ across. It’s micro, but it still packs a punch – the OLED display is very readable due to the high contrast, and you can fit a deceivingly large amount of graphics on there.

As the display makes its own light, no backlight is required. This significantly reduces the power required to run the OLED and is why the display has such high contrast, extremely wide viewing angle and can display deep [black levels](https://en.wikipedia.org/wiki/Black_level).



At the heart of the module is a powerful single-chip CMOS OLED driver controller – SSD1306, which handles all the RAM buffering, so that very little work needs to be done by your ESP32. Also the operating voltage of the SSD1306 controller is from 1.65V to 3.3V – Perfect for interfacing with 3.3V microcontrollers like ESP32.

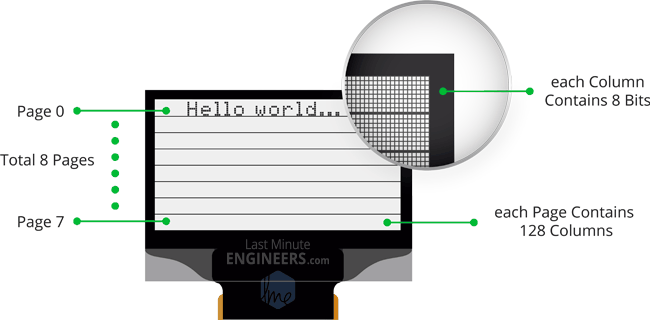
OLED Memory Map

To have absolute control over your OLED display module, it’s important to know about its memory map.

Regardless of the size of the OLED module, the SSD1306 driver has a built-in 1KB Graphic Display Data RAM (GDDRAM) for the screen which holds the bit pattern to be displayed. This 1K memory area is organized in 8 pages (from 0 to 7). Each page contains 128 columns/segments (block 0 to 127). And each column can store 8 bits of data (from 0 to 7). That surely tells us we have

8 pages x 128 segments x 8 bits of data = 8192 bits = 1024 bytes = 1KB memory

The whole 1K memory with pages, segments and data is highlighted below.



Each bit represents particular OLED pixel on the screen which can be turned ON or OFF programmatically.

The 128×64 OLED screen displays all the contents of RAM whereas 128×32 OLED screen displays only 4 pages (half content) of RAM.

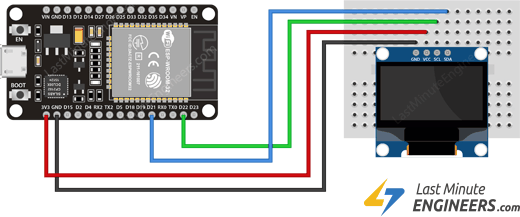
Wiring OLED display module to ESP32

Enough of the theory, Let’s Go Practical! Let’s hook the display up to the ESP32.

Connections are fairly simple. Start by connecting VCC pin to the 3.3V output on the ESP32 and connect GND to ground.

Next, Connect the SCL pin to the I2C clock D22 pin on your ESP32 and connect the SDA pin to the I2C data D21 pin on your ESP32. Refer to [ESP32 Pinout](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/#esp32-development-board-pinout).

The following diagram shows you how to wire everything.

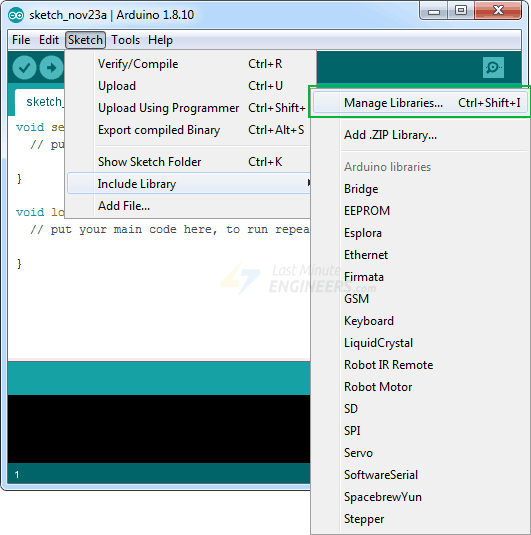
Wiring Connections for OLED Display Module with ESP32

With that, you’re now ready to upload some code and get the display printing.

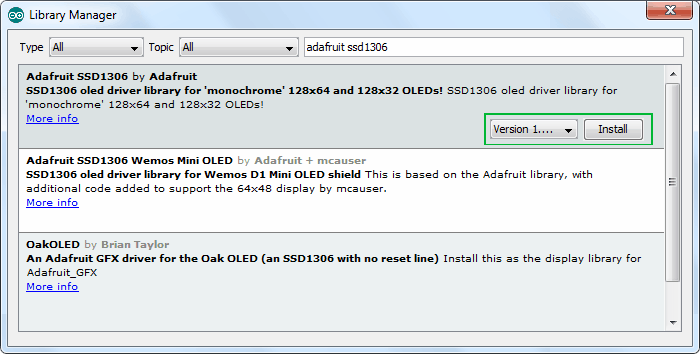
Installing Library for OLED Display Module

The SSD1306 controller of the OLED display has flexible yet complex drivers. Vast knowledge on memory addressing is required in order to use the SSD1306 controller. Fortunately, [Adafruit’s SSD1306 library](https://github.com/adafruit/Adafruit_SSD1306) was written to hide away the complexities of the SSD1306 controller so that we can issue simple commands to control the display.

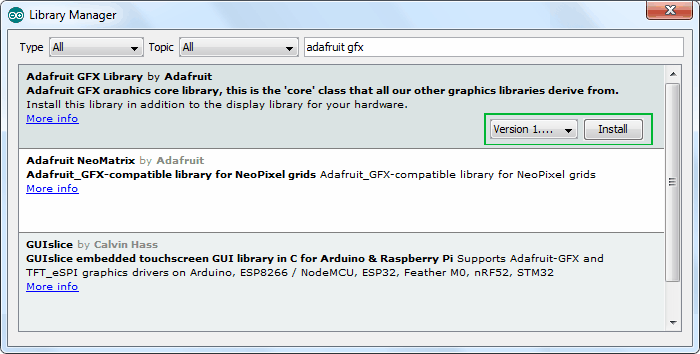
To install the library navigate to the Sketch > Include Library > Manage Libraries… Wait for Library Manager to download libraries index and update list of installed libraries.



Filter your search by typing ‘adafruit ssd1306’. There should be a couple entries. Look for Adafruit SSD1306 by Adafruit. Click on that entry, and then select Install.



This Adafruit SSD1306 library is a hardware-specific library which handles lower-level functions. It needs to be paired with [Adafruit GFX Library](https://github.com/adafruit/Adafruit-GFX-Library) to display graphics primitives like points, lines, circles, rectangles etc. Install this library as well.

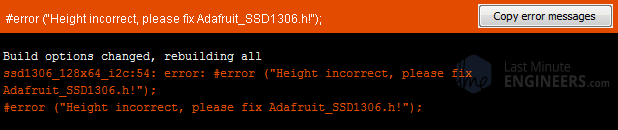


NOTE

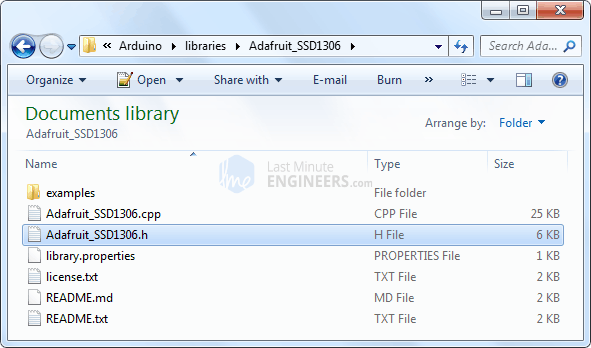
The library allocates 1KB (128×64)/8 bits) of memory from ESP32 as buffer. So, it can manipulate the screen buffer and then perform a bulk transfer from the ESP32’s memory to the internal memory of the SSD1306 controller.

Modifying Adafruit SSD1306 Library

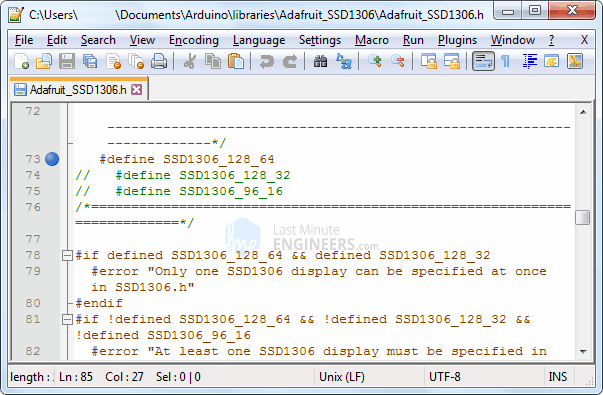
Adafruit’s SSD1306 Library isn’t set up for the 128×64 OLED displays (the one we are using right now). The display size must be changed in the Adafruit\_SSD1306.h header file to make it work for us. If it is not changed, an error message saying #error (“Height incorrect, please fix Adafruit\_SSD1306.h!”); may appear when attempting to verify the example sketch in the Arduino IDE:



In order to change the Adafruit\_SSD1306.h header file, open your sketchbook location. It’s generally My Documents > Arduino. Now go to libraries > Adafruit\_SSD1306



Open Adafruit\_SSD1306.h file in a text editor. Scroll down the file to find the section with the SSD1306 Displays or directly go to line no. 73. Comment out #define SSD1306\_128\_32 and uncomment #define SSD1306\_128\_64 so that the code in this section looks like this:



That’s it. Now save the file and restart your Arduino IDE.

ESP32 Code – Displaying Text

Now comes the interesting stuff!

The following test sketch will print ‘Hello World!’ message on the display. It also includes

* Displaying Inverted text
* Displaying Numbers
* Displaying Numbers with base (Hex, Dec)
* Displaying ASCII symbols
* Scrolling Text Horizontally & Vertically
* Scrolling part of the display

This will give you complete understanding about how to use the OLED display and can serve as the basis for more practical experiments and projects.

#include <SPI.h>

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

Adafruit\_SSD1306 display(-1);

void setup()

{

// initialize with the I2C addr 0x3C

display.begin(SSD1306\_SWITCHCAPVCC, 0x3C);

// Clear the buffer.

display.clearDisplay();

// Display Text

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,28);

display.println("Hello world!");

display.display();

delay(2000);

display.clearDisplay();

// Display Inverted Text

display.setTextColor(BLACK, WHITE); // 'inverted' text

display.setCursor(0,28);

display.println("Hello world!");

display.display();

delay(2000);

display.clearDisplay();

// Changing Font Size

display.setTextColor(WHITE);

display.setCursor(0,24);

display.setTextSize(2);

display.println("Hello!");

display.display();

delay(2000);

display.clearDisplay();

// Display Numbers

display.setTextSize(1);

display.setCursor(0,28);

display.println(123456789);

display.display();

delay(2000);

display.clearDisplay();

// Specifying Base For Numbers

display.setCursor(0,28);

display.print("0x"); display.print(0xFF, HEX);

display.print("(HEX) = ");

display.print(0xFF, DEC);

display.println("(DEC)");

display.display();

delay(2000);

display.clearDisplay();

// Display ASCII Characters

display.setCursor(0,24);

display.setTextSize(2);

display.write(3);

display.display();

delay(2000);

display.clearDisplay();

// Scroll full screen

display.setCursor(0,0);

display.setTextSize(1);

display.println("Full");

display.println("screen");

display.println("scrolling!");

display.display();

display.startscrollright(0x00, 0x07);

delay(2000);

display.stopscroll();

delay(1000);

display.startscrollleft(0x00, 0x07);

delay(2000);

display.stopscroll();

delay(1000);

display.startscrolldiagright(0x00, 0x07);

delay(2000);

display.startscrolldiagleft(0x00, 0x07);

delay(2000);

display.stopscroll();

display.clearDisplay();

// Scroll part of the screen

display.setCursor(0,0);

display.setTextSize(1);

display.println("Scroll");

display.println("some part");

display.println("of the screen.");

display.display();

display.startscrollright(0x00, 0x00);

}

void loop() {}

The sketch starts by including four libraries viz. SPI.h, Wire.h, Adafruit\_GFX.h and Adafruit\_SSD1306.h. Although SPI.h library is not required for I2C OLED displays, we need to add it for the sake of compiling our program.

#include <SPI.h>

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

Next, we need to create an object of Adafruit\_SSD1306.h. The Adafruit\_SSD1306 constructor accepts ESP32 pin number to which reset pin of the display is connected. As the OLED display we are using doesn’t have a RESET pin, we will send –1 to the constructor so that none of the ESP32 pins is used as a reset for the display.

Adafruit\_SSD1306 display(-1);

In setup function: we need to initialize the OLED object using begin() function. The function takes two parameters. First parameter SSD1306\_SWITCHCAPVCC turns the internal charge pump circuitry ON while second parameter provides I2C address of the OLED display. I2C address of such OLED display module is generally 0x3C. It’s fixed and cannot be changed.

Next, we clear the buffer before printing our first message on the screen.

// initialize with the I2C addr 0x3C

display.begin(SSD1306\_SWITCHCAPVCC, 0x3C);

// Clear the buffer.

display.clearDisplay();

Displaying simple Text (Hello World)



// Display Text

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,28);

display.println("Hello world!");

display.display();

delay(2000);

For displaying text on the screen, we need to set the font size. This can be done by calling setTextSize(font-size) and passing font size (starting from 1) as a parameter.

Next, we need to set the font color by calling function setTextColor(color) . Pass parameter WHITE for the dark background and pass BLACK for bright background. Now before printing the message we need to set the cursor position by calling function setCursor(X,Y). Pixels on the screen are addressed by their horizontal (X) and vertical (Y) coordinates. The coordinate system places the origin (0,0) at the top left corner, with positive X increasing to the right and positive Y increasing downward.

We can use simple print(“ ”) or println(“ ”) function to print the message on the screen just like we print data on serial monitor. Remember, println() will move the cursor to the new line.

In order for the library to perform extremely fast mathematical operations on the screen buffer (more than 100 frames per second), calls to the print functions do not immediately transfer the contents of screen buffer to the SSD1306 controller. A display() command is required to instruct the library to perform the bulk transfer from the screen buffer in the ESP32 to the internal memory of the SSD1306 controller. As soon as the memory is being transferred, the pixels corresponding to the screen buffer will show up on the OLED display.

Displaying Inverted Text



// Display Inverted Text

display.clearDisplay();

display.setTextColor(BLACK, WHITE); // 'inverted' text

display.setCursor(0,28);

display.println("Hello world!");

display.display();

delay(2000);

For displaying inverted text, we will call setTextColor(FontColor,BackgroundColor) function again. If you are paying attention, you know we passed only one parameter to this function earlier, but now we are passing two parameters. This is possible because of something called [function overloading](https://en.wikipedia.org/wiki/Function_overloading). Function overloading is the ability to create multiple [functions](https://en.wikipedia.org/wiki/Subprogram) of the same name but with different set of parameters. Calls to an overloaded function will run a specific implementation of that function depending upon the parameters passed.

In our case passing setTextColor(BLACK, WHITE) will render black text on filled background.

Scaling Font Size



// Changing Font Size

display.clearDisplay();

display.setTextColor(WHITE);

display.setCursor(0,24);

display.setTextSize(2);

display.println("Hello!");

display.display();

delay(2000);

Earlier in this tutorial, we called setTextSize(font-size) function to set font size and passed 1 as parameter. You can use this function to scale the font by passing any non-negative integer.

Characters are rendered in the ratio of 7:10. Meaning, passing font size 1 will render the text at 7×10 pixels per character, passing 2 will render the text at 14×20 pixels per character and so on.

The Adafruit\_GFX library is responsible for rendering font. By default the mono-spaced font is selected. However, more recent versions of the Adafruit GFX library offer the ability to use alternate fonts. Several alternate fonts come with the library, plus there’s the ability to add new ones.

Displaying Numbers



// Display Numbers

display.clearDisplay();

display.setTextSize(1);

display.setCursor(0,28);

display.println(123456789);

display.display();

delay(2000);

Numbers can be displayed on the OLED display by just calling print() or println() function. An overloaded implementation of these functions accepts 32-bit unsigned int, so you can only display numbers from 0 to 4,294,967,295.

Specifying Base For Numbers



// Specifying Base For Numbers

display.clearDisplay();

display.setCursor(0,28);

display.print("0x"); display.print(0xFF, HEX);

display.print("(HEX) = ");

display.print(0xFF, DEC);

display.println("(DEC)");

display.display();

delay(2000);

The print() & println() functions has optional second parameter that specifies the base (format) to use; permitted values are BIN (binary, or base 2), OCT (octal, or base 8), DEC (decimal, or base 10), HEX (hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example:

* print(78, BIN) gives “1001110”
* print(78, OCT) gives “116”
* print(78, DEC) gives “78”
* print(78, HEX) gives “4E”
* println(1.23456, 0) gives “1”
* println(1.23456, 2) gives “1.23”
* println(1.23456, 4) gives “1.2346”

Displaying ASCII Symbols



// Display ASCII Characters

display.clearDisplay();

display.setCursor(0,24);

display.setTextSize(2);

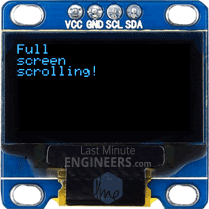
display.write(3);

display.display();

delay(2000);

The print() & println() functions send data to the display as human-readable ASCII text while write() function sends binary data to the display. So, you can use this function to display ASCII symbols. In our example sending number 3 will display heart symbol.

Full Screen Scrolling



// Scroll full screen

display.clearDisplay();

display.setCursor(0,0);

display.setTextSize(1);

display.println("Full");

display.println("screen");

display.println("scrolling!");

display.display();

display.startscrollright(0x00, 0x07);

delay(2000);

display.stopscroll();

delay(1000);

display.startscrollleft(0x00, 0x07);

delay(2000);

display.stopscroll();

delay(1000);

display.startscrolldiagright(0x00, 0x07);

delay(2000);

display.startscrolldiagleft(0x00, 0x07);

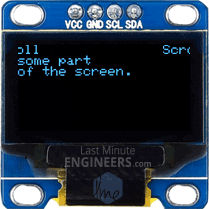
delay(2000);

display.stopscroll();

You can scroll the display horizontally by calling startscrollright(start page, stop page) & startscrollleft(start page, stop page) functions and diagonally by calling startscrolldiagright(start page, stop page) & startscrolldiagleft(start page, stop page). All these functions accept two parameters viz. start page and stop page. Refer to OLED Memory Map section for explanation of the pages. As there are eight pages in the display from 0 to 7, you can scroll entire screen by scrolling all the pages i.e. passing parameters 0x00 and 0x07.

To stop display from scrolling you can use stopscroll() function.

Scrolling Specific Part



// Scroll part of the screen

display.setCursor(0,0);

display.setTextSize(1);

display.println("Scroll");

display.println("some part");

display.println("of the screen.");

display.display();

display.startscrollright(0x00, 0x00);

Sometimes we don’t want to scroll entire display. You can do that by passing proper start page and stop page information to scrolling functions. Refer to OLED Memory Map section for explanation of the pages. As there are eight pages in the display from 0 to 7, you can scroll some part of the screen by passing specific page numbers as parameters.

In our example, we passed both the parameters as 0x00. This will scroll only first page (first 8 rows) of the display.

ESP32 Code – Basic Drawings

In this example, we’re going to try some basic drawings. This sketch demonstrates many drawing functions, including rectangles, round rectangles, circles and triangles. Try the sketch out and then we will dissect it in some detail.

#include <SPI.h>

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

Adafruit\_SSD1306 display(-1);

void setup()

{

// initialize with the I2C addr 0x3C

display.begin(SSD1306\_SWITCHCAPVCC, 0x3C);

// Clear the buffer.

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Rectangle");

display.drawRect(0, 15, 60, 40, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Rectangle");

display.fillRect(0, 15, 60, 40, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Round Rectangle");

display.drawRoundRect(0, 15, 60, 40, 8, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Round Rectangl");

display.fillRoundRect(0, 15, 60, 40, 8, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Circle");

display.drawCircle(20, 35, 20, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Circle");

display.fillCircle(20, 35, 20, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Triangle");

display.drawTriangle(30, 15, 0, 60, 60, 60, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Triangle");

display.fillTriangle(30, 15, 0, 60, 60, 60, WHITE);

display.display();

delay(2000);

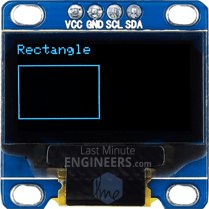
display.clearDisplay();

}

void loop() {}

Most of the code (Including libraries and initializing display) is same as above code example, except following code snippets to draw basic drawings.

Drawing Rectangle



display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Rectangle");

display.drawRect(0, 15, 60, 40, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Rectangle");

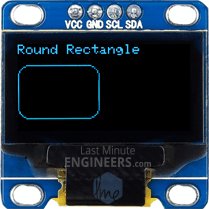
display.fillRect(0, 15, 60, 40, WHITE);

display.display();

delay(2000);

You can draw rectangle on the display by using drawRect(X-coordinate, Y-coordinate, Width, Height, color) function. Actually this function draws hollow rectangle with 1 pixel border. You can draw filled rectangle using fillRect() function.

Drawing Round Rectangle



display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Round Rectangle");

display.drawRoundRect(0, 15, 60, 40, 8, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Round Rectangl");

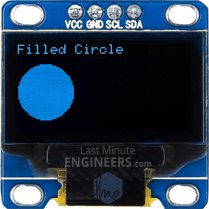
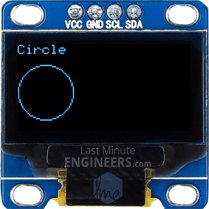
display.fillRoundRect(0, 15, 60, 40, 8, WHITE);

display.display();

delay(2000);

You can draw round rectangle on the display by using drawRoundRect(X-coordinate, Y-coordinate, Width, Height, color) function. This function takes same parameters as drawRect() function except one additional parameter – Radius of corner rounding. Actually this function draws hollow round rectangle with 1 pixel border. You can draw filled round rectangle using fillRoundRect() function.

Drawing Circle



display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Circle");

display.drawCircle(20, 35, 20, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Circle");

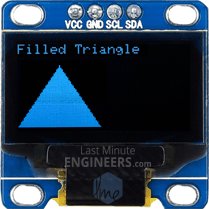
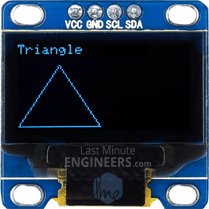
display.fillCircle(20, 35, 20, WHITE);

display.display();

delay(2000);

You can draw circle on the display by using drawCircle(X-coordinate of center, Y-coordinate of center, radius, color) function. This function draws hollow circle with 1 pixel border. You can draw filled circle using fillCircle() function.

Drawing Triangle



display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Triangle");

display.drawTriangle(30, 15, 0, 60, 60, 60, WHITE);

display.display();

delay(2000);

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Filled Triangle");

display.fillTriangle(30, 15, 0, 60, 60, 60, WHITE);

display.display();

delay(2000);

You can draw triangle on the display by using drawTriangle(x0, y0, x1, y1, x2, y2, color) function. The function takes seven parameters viz. 3 X & Y coordinates of vertices of triangle and color. (X0,y0) represents top vertex, (x1,y1) represents left vertex and (x2,y2) represents right vertex.

This function draws hollow triangle with 1 pixel border. You can draw filled triangle using fillTriangle() function.

ESP32 Code – Displaying Bitmap

This last example shows how to draw bitmap images to the OLED Display. This is useful for creating splash screens of company logos, making sprites or just creating fun graphics for displaying information. Copy the following code, paste it into the Arduino IDE and click upload.

#include <SPI.h>

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

Adafruit\_SSD1306 display(-1);

// Bitmap of MarilynMonroe Image

const unsigned char MarilynMonroe [] PROGMEM = {

0xff, 0xff, 0xff, 0xff, 0xff, 0xf8, 0x1f, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xc0, 0x1f, 0xff, 0xff, 0xf0, 0x41, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0x80, 0x7f, 0xff, 0xff, 0xf8, 0x03, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xf9, 0xff, 0xff, 0xff, 0xe0, 0x07, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0x87, 0xff, 0xff, 0xff, 0xf8, 0x03, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0x07, 0xff, 0xff, 0xff, 0xf8, 0x01, 0xf1, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0x9f, 0xff, 0xff, 0xff, 0xf8, 0x00, 0xf8, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xbf, 0xff, 0xff, 0xff, 0xfc, 0x02, 0x78, 0x7f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xfc, 0x3f, 0xff, 0xff, 0xfe, 0x03, 0x7c, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xf0, 0x07, 0xff, 0xff, 0xfe, 0x01, 0xfe, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xfd, 0xe0, 0x03, 0xff, 0xff, 0xfc, 0x00, 0xfe, 0x0f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xfe, 0x87, 0xe0, 0xff, 0xff, 0xfc, 0x00, 0x06, 0x07, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xfc, 0x1f, 0xf9, 0xff, 0xff, 0xfc, 0x00, 0x02, 0x07, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xf8, 0x1f, 0xff, 0xff, 0xff, 0xfc, 0x00, 0xc3, 0xc3, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xf0, 0x3f, 0xff, 0xff, 0xe0, 0x0c, 0x00, 0xe7, 0x81, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xf0, 0x0f, 0xff, 0xff, 0xe0, 0x02, 0x00, 0x02, 0x00, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xf0, 0x0f, 0xff, 0xff, 0xe0, 0x01, 0x00, 0x00, 0x00, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0x80, 0x00, 0x3f, 0xff, 0xff, 0xe0, 0x00, 0x00, 0x1e, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xfc, 0x00, 0x00, 0x0f, 0xff, 0x3f, 0xf8, 0x00, 0x18, 0x7f, 0x1f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xf8, 0x01, 0x80, 0x03, 0xfc, 0x3f, 0xfc, 0x00, 0x70, 0xfe, 0x1f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xf0, 0x43, 0xff, 0xff, 0xf8, 0x7f, 0xf8, 0x00, 0x00, 0x7e, 0x1f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe0, 0x07, 0xff, 0xff, 0xf0, 0xff, 0xfc, 0x00, 0x00, 0x7c, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe0, 0x0f, 0xff, 0xff, 0xf1, 0xef, 0xf8, 0x00, 0x01, 0xfc, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe4, 0xff, 0xff, 0xff, 0xf3, 0x80, 0xa0, 0x00, 0x07, 0xfc, 0xaf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xec, 0x5f, 0xff, 0xff, 0xe7, 0xf0, 0x00, 0x00, 0x03, 0xfe, 0xdf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xee, 0x7f, 0xff, 0xff, 0xc7, 0xf8, 0x00, 0x00, 0x03, 0xff, 0xdf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xf7, 0xc7, 0xff, 0x06, 0x00, 0x03, 0xff, 0xbf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xfe, 0x5f, 0xff, 0xc7, 0x07, 0xff, 0x80, 0x00, 0x07, 0xdb, 0xbf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xee, 0xff, 0xff, 0x80, 0x03, 0xff, 0xc0, 0x00, 0x03, 0xc3, 0x0f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xfe, 0xff, 0xff, 0x98, 0x03, 0xff, 0xf8, 0x00, 0x07, 0xe0, 0x0f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xef, 0xff, 0xff, 0xf8, 0x01, 0xff, 0xfc, 0x01, 0x07, 0xfc, 0x1f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xcf, 0xef, 0xff, 0xff, 0xe1, 0xff, 0xfc, 0x01, 0x07, 0xf8, 0x1f, 0xff, 0xff,

0xff, 0xff, 0xff, 0x9f, 0xff, 0xff, 0x7f, 0xf1, 0xff, 0xf8, 0x02, 0x07, 0x88, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xcf, 0xef, 0xf8, 0x0f, 0xff, 0xff, 0xe0, 0x00, 0x07, 0x84, 0x3f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe7, 0xef, 0xf0, 0x04, 0x7f, 0xff, 0xc0, 0x00, 0x07, 0x84, 0x7f, 0xff, 0xff,

0xff, 0xff, 0xff, 0x3f, 0xff, 0xe0, 0x00, 0x1f, 0xff, 0x80, 0x00, 0x06, 0x04, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x3f, 0x7f, 0xe1, 0xf0, 0x07, 0xff, 0x80, 0x00, 0x07, 0x06, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xc3, 0xfe, 0x03, 0xff, 0x00, 0x00, 0x03, 0x80, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xf2, 0x3f, 0xc6, 0x7f, 0x81, 0xce, 0x00, 0x00, 0x01, 0xc1, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe0, 0x3f, 0xc0, 0x07, 0xc1, 0xfe, 0x00, 0x00, 0x0d, 0xc0, 0x7f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xe0, 0x3f, 0xc0, 0x01, 0xe0, 0xfc, 0x00, 0x00, 0x0f, 0xc0, 0x7f, 0xff, 0xff,

0xff, 0xff, 0xff, 0xc0, 0x3f, 0xc0, 0x00, 0x50, 0xfc, 0x00, 0x00, 0x0e, 0xc0, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xc0, 0x3f, 0xc0, 0x00, 0x18, 0xf8, 0x00, 0x00, 0x0e, 0xc1, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xc0, 0x3f, 0xc0, 0x00, 0x00, 0xf8, 0x00, 0x00, 0x66, 0x81, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xc0, 0x1f, 0xc7, 0x80, 0x00, 0xf8, 0x00, 0x01, 0xe0, 0x00, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xc0, 0x1f, 0xc1, 0xe0, 0x01, 0xf8, 0x00, 0x03, 0xf0, 0x01, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x80, 0x1f, 0xc0, 0x3e, 0x03, 0xf0, 0x00, 0x00, 0xe0, 0x03, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x00, 0x1f, 0xe0, 0xe0, 0x03, 0xf2, 0x00, 0x00, 0xc0, 0x03, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x80, 0x1f, 0xf0, 0x00, 0x07, 0xe6, 0x00, 0x00, 0xc0, 0x03, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x80, 0x1f, 0xff, 0x00, 0x1f, 0xee, 0x00, 0x00, 0x80, 0x07, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xb8, 0x0f, 0xff, 0xf0, 0x3f, 0xdc, 0x00, 0x00, 0x00, 0x0f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xbc, 0x0f, 0xff, 0xff, 0xff, 0xdc, 0x00, 0x00, 0x00, 0x0f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x9e, 0x0f, 0xff, 0xff, 0xff, 0xf8, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x08, 0x0f, 0xff, 0xff, 0xff, 0x70, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x00, 0x0b, 0xff, 0xff, 0xfe, 0xe0, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x00, 0x0b, 0xff, 0xff, 0xf9, 0xc0, 0x00, 0x00, 0x00, 0x3f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x3c, 0x09, 0xff, 0xff, 0xf1, 0x80, 0x00, 0x00, 0x00, 0x7f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x1e, 0x08, 0x3f, 0xff, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x7f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x1f, 0x08, 0x03, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x80, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xce, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xfe, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0x7e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7f, 0xff, 0xff, 0xff

};

void setup()

{

// initialize with the I2C addr 0x3C

display.begin(SSD1306\_SWITCHCAPVCC, 0x3C);

// Clear the buffer.

display.clearDisplay();

// Display bitmap

display.drawBitmap(0, 0, MarilynMonroe, 128, 64, WHITE);

display.display();

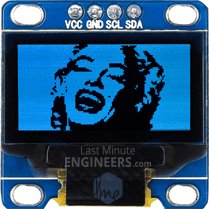
// Invert Display

//display.invertDisplay(1);

}

void loop() {}

This is how the output looks like.



To show bitmap image on the OLED display we need to call drawBitmap(X-coordinate, Y-coordinate, bitmap array, width, height, color) function. It takes six parameters viz. Top left corner X coordinate, top left corner Y coordinate, byte array of monochrome bitmap, width of bitmap in pixels, height of bitmap in pixels and Color.

In our example, the bitmap image is 128×64 in size. So, X & Y coordinates are set to 0 while width & height is set to 128 & 64.

// Display bitmap

display.drawBitmap(0, 0, MarilynMonroe, 128, 64, WHITE);

display.display();

But, before we can call the drawBitmap() function, we first need an image to draw. Remember, the screen resolution of the OLED display is 128×64 pixels, so images larger than that will not display correctly. To get a correctly sized image, you can use your favorite drawing programs like Inkscape, Photoshop, Paint, etc., setting the canvas size to 128×64 pixels.

We took laughing Marilyn Monroe image as an example and converted into 128×64 pixels using Paint and saved as .bmp.





Once you have a bitmap, it’s time to convert it into an array that the SSD1306 OLED controller can understand. This can be done using two ways: Online method using image2cpp and Offline method using LCD Assistant.

Online Bitmap Array Generator – image2cpp

There’s an online application called image2cpp – <http://javl.github.io/image2cpp/> which can convert your image into an array. Image2cpp is newer and much more powerful than LCD Assistant (later solution). It will allow you to:

* Convert multiple images simultaneously.
* Scale your image file – Stretch/Scale to fit/Original
* Adjust the Brightness threshold between black and white.
* Re-center the image vertically and / or horizontally.
* Reverse image colors

This tool is so powerful that it can work offline as well. Simply save the page to your PC and open it in your browser. Thanks to [Jasper van Loenen](https://github.com/javl) for his excellent contribution.

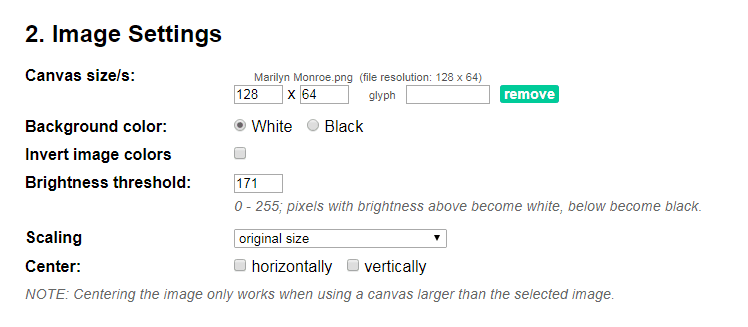
To begin with, open [image2cpp](http://javl.github.io/image2cpp/) in your browser and select any image you want to display on OLED screen.



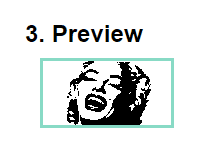
The dimensions of your image will populate in Canvas size option under Image settings. If you have selected bigger image than 128×64, change it to 128×64 and select proper Scaling option. You can view the output in Preview section.

You can change the Background color or Invert image colors if necessary.

Finally, change the most important option – Brightness threshold as per your requirement. Setting threshold will make pixels above this level white and below black. In our case we have set it to 171 to get nice details of Marilyn Monroe.

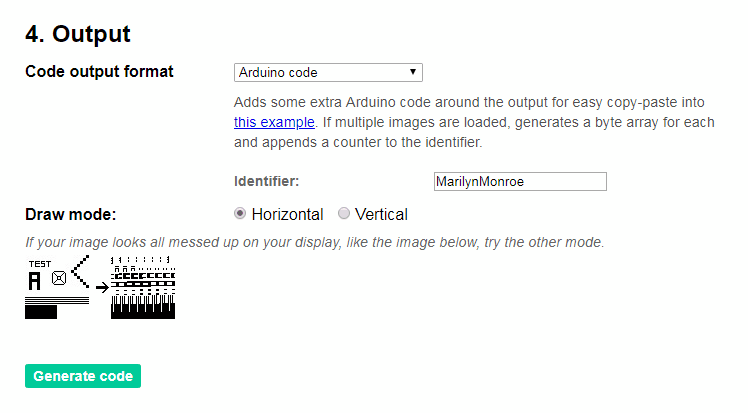


This little preview reflects whatever changes you make in your settings. You can change settings while keeping eye on it.

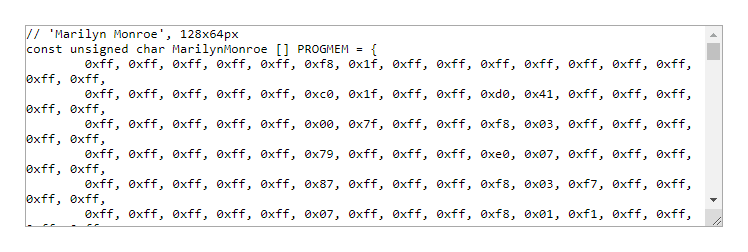


Once you are satisfied with the outcome, you can proceed generating the data array. Simply select Code output format as Arduino Code and click on Generate code button.

Just for your information, there’s an option called Draw mode. It actually creates image according to the scanning patter of the display. If your image looks all messed up on your display, try changing the mode.



That’s it. The byte array of your bitmap will be generated. You can use the output directly with our example code. Just be sure to name it appropriately. Then call your array inside the drawBitmap() function.

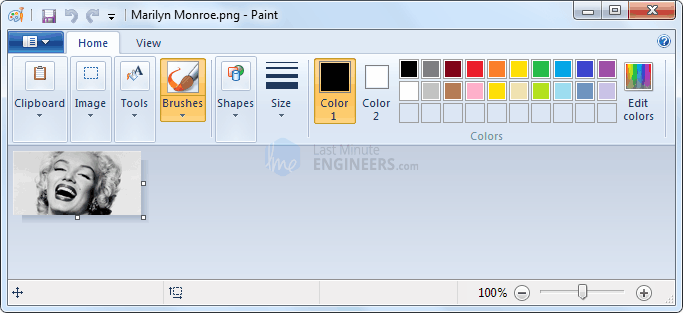


Offline Bitmap Array Generator – LCD Assistant

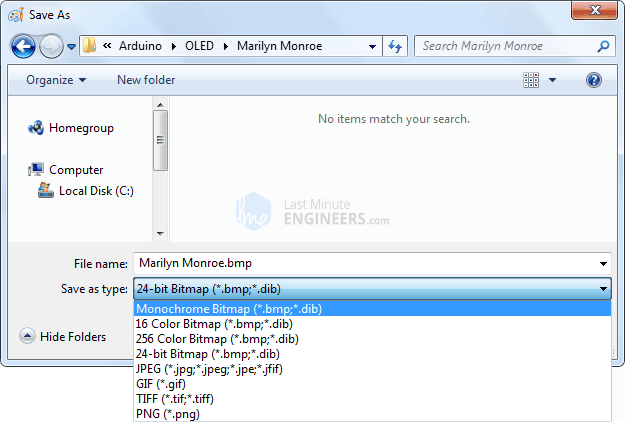
There’s another application called LCD assistant – <http://en.radzio.dxp.pl/bitmap_converter/> which can convert your bitmap image into data array. It’s not as powerful as image2cpp but still popular among hobbyists.

To start with, you need to convert you image into 128×64 1-bit monochrome bitmap. You can use your favorite drawing programs like Inkscape, Photoshop, Paint, etc. to do it, just like we did in MS paint.

Open your file in MS Paint and resize it to 128×64.

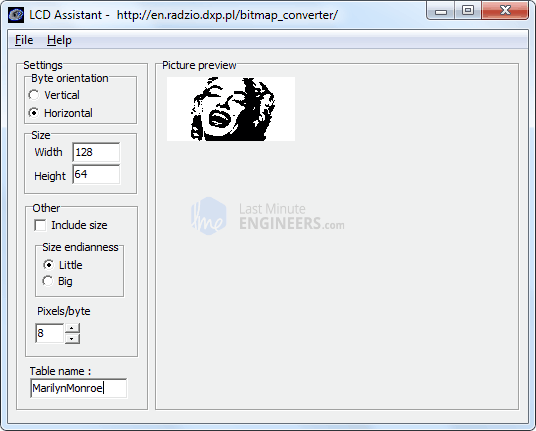


Now, save your file as bitmap. While saving the file choose Save as type : Monochrome Bitmap(\*.bmp;\*.dib). This will generate 1-bit/binary bitmap image that has only two possible values for each pixel i.e. 0 (black) or 1 (white).



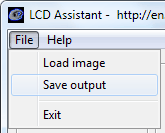
The only downside here is that you cannot set brightness threshold level. It is set to 50% by default and cannot be changed.

Anyways now, download LCD assistant program. Open the executable and load your bitmap from File menu.

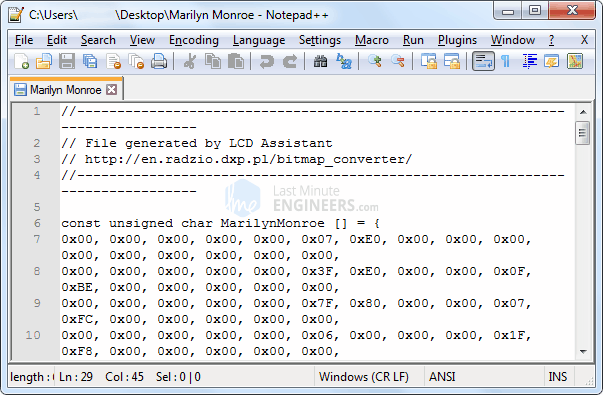


There’s nothing much you can do with this tool. So, just go to File menu and click on Save output option. Save the file as text file.

Just for your information, there’s an option called Byte Orientation. It actually creates image according to the scanning patter of the display. If your image looks all messed up on your display, try changing the mode.

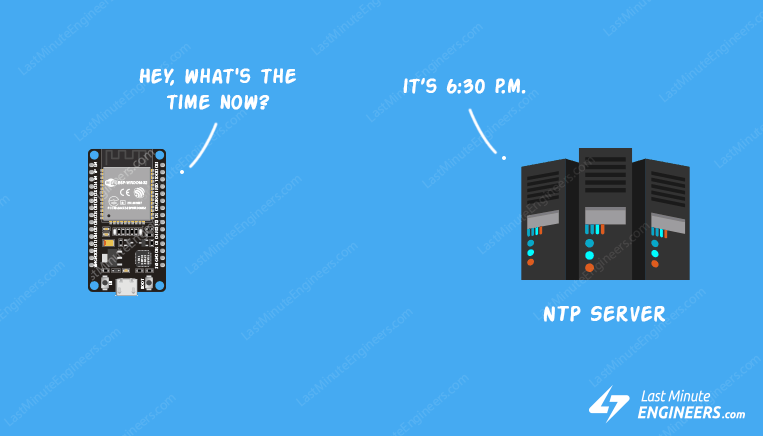


That’s it. With your array created, paste it into your code.



Just be sure to name it appropriately. Then call your array inside the drawBitmap() function.

**Getting Date & Time From NTP Server With ESP32**



Every once in a while you’ll come across an idea where keeping time a prime concern. For example, imagine a relay that has to be activated at a certain time or a data logger that has to store values at precise intervals.

The first thing that comes in your mind is to use an RTC (Real Time Clock) chip. But these chips are not perfectly accurate so, you need to do manual adjustments over and over again to keep them synchronized.

The solution here is to use Network Time Protocol (NTP). If your ESP32 project has access to the Internet, you can get date and time (with a precision within a few milliseconds of UTC) for FREE. You don’t need any additional hardware.

What is an NTP?

An NTP stands for [Network Time Protocol](http://www.ntp.org/ntpfaq/NTP-s-def.htm). It’s a standard Internet Protocol (IP) for synchronizing the computer clocks to some reference over a network.

The protocol can be used to synchronize all networked devices to Coordinated Universal Time (UTC) within a few milliseconds ( 50 milliseconds over the public Internet and under 5 milliseconds in a LAN environment).

Coordinated Universal Time (UTC) is a world-wide time standard, closely related to GMT (Greenwich Mean Time). UTC does not vary, it is the same world wide.

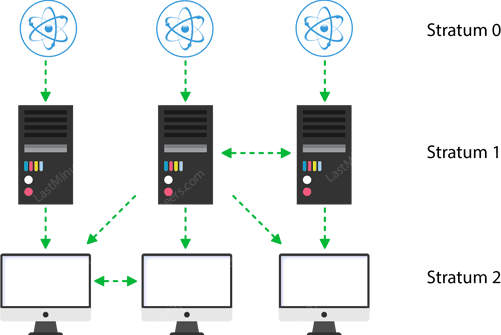
NTP sets the clocks of computers to UTC, any local time zone offset or day light saving time offset is applied by the client. In this manner clients can synchronize to servers regardless of location and time zone differences.

NTP Architecture

NTP uses a hierarchical architecture. Each level in the hierarchy is known as a stratum.

At the very top are high-precision timekeeping devices, such as atomic clocks, GPS or radio clocks, known as stratum 0 hardware clocks.

Stratum 1 servers have a direct connection to a stratum 0 hardware clock and therefore have the most accurate time.

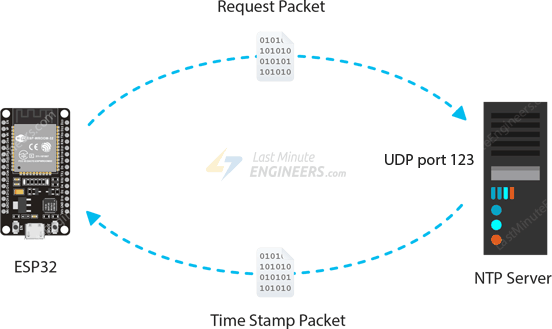


Each stratum in the hierarchy synchronizes to the stratum above and act as servers for lower stratum computers.

How NTP Works?

NTP can operate in a number of ways. The most common configuration is to operate in client-server mode. The basic working principle is as follows:

1. The client device such as ESP32 connects to the server using the User Datagram Protocol (UDP) on port 123.
2. A client then transmits a request packet to a NTP server.
3. In response to this request the NTP server sends a time stamp packet.
4. A time stamp packet contains multiple information like UNIX timestamp, accuracy, delay or timezone.
5. A client can then parse out current date & time values.



Preparing the Arduino IDE

Enough of the theory, Let’s Go Practical!

But before venturing further into this tutorial, you should have the ESP32 add-on installed in your Arduino IDE. Follow below tutorial to prepare your Arduino IDE to work with the ESP32, if you haven’t already.

[](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

[Insight Into ESP32 Features & Using It With Arduino IDE](https://lastminuteengineers.com/esp32-arduino-ide-tutorial/)

Few years back, ESP8266 took the embedded IoT world by storm. For less than $3, you could get a programmable, WiFi-enabled microcontroller being able to...

Getting Date and Time from NTP Server

The following sketch will give you complete understanding on how to get date and time from the NTP Server.

Before you head for uploading the sketch, you need to make some changes to make it work for you.

* You need to modify the following two variables with your network credentials, so that ESP32 can establish a connection with existing network.
* const char\* ssid = "YOUR\_SSID";
* const char\* password = "YOUR\_PASS";
* You need to adjust the UTC offset for your timezone in milliseconds. Refer the [list of UTC time offsets](https://en.wikipedia.org/wiki/List_of_UTC_time_offsets). Here are some examples for different timezones:
  + For UTC -5.00 : -5 \* 60 \* 60 : -18000
  + For UTC +1.00 : 1 \* 60 \* 60 : 3600
  + For UTC +0.00 : 0 \* 60 \* 60 : 0
  + const long gmtOffset\_sec = 3600;
  + Change the Daylight offset in milliseconds. If your country observes [Daylight saving time](https://en.wikipedia.org/wiki/Daylight_saving_time) set it to 3600. Otherwise, set it to 0.
  + const int daylightOffset\_sec = 3600;

Once you are done, go ahead and try the sketch out.

#include <WiFi.h>

#include "time.h"

const char\* ssid = "YOUR\_SSID";

const char\* password = "YOUR\_PASS";

const char\* ntpServer = "pool.ntp.org";

const long gmtOffset\_sec = 3600;

const int daylightOffset\_sec = 3600;

void printLocalTime()

{

struct tm timeinfo;

if(!getLocalTime(&timeinfo)){

Serial.println("Failed to obtain time");

return;

}

Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");

}

void setup()

{

Serial.begin(115200);

//connect to WiFi

Serial.printf("Connecting to %s ", ssid);

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println(" CONNECTED");

//init and get the time

configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer);

printLocalTime();

//disconnect WiFi as it's no longer needed

WiFi.disconnect(true);

WiFi.mode(WIFI\_OFF);

}

void loop()

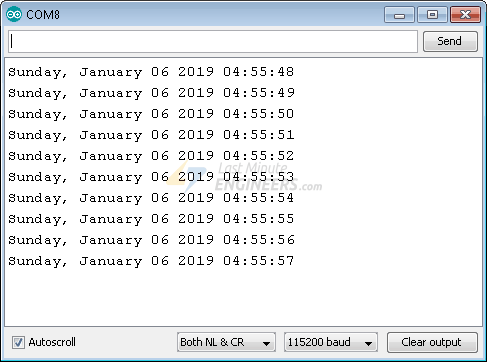
{

delay(1000);

printLocalTime();

}

After uploading the sketch, press the EN button on your ESP32, and you should get the date and time every second as shown below.



Code Explanation

Let’s take a quick look at the code to see how it works. First, we include the libraries needed for this project.

* WiFi.h library provides ESP32 specific WiFi methods we are calling to connect to network.
* time.h is the ESP32 native time library which does graceful NTP server synchronization.

#include <WiFi.h>

#include "time.h"

Next, we set up a few constants like SSID, WiFi password, UTC Offset & Daylight offset that you are already aware of.  
Along with that we need to specify the address of the NTP Server we wish to use. pool.ntp.org is an open NTP project great for things like this.

const char\* ntpServer = "pool.ntp.org";

The pool.ntp.org automatically picks time servers which are geographically close for you. But if you want to choose explicitly, use one of the sub-zones of pool.ntp.org.

|  |  |
| --- | --- |
| Area | HostName |
| Worldwide | pool.ntp.org |
| Asia | asia.pool.ntp.org |
| Europe | europe.pool.ntp.org |
| North America | north-america.pool.ntp.org |
| Oceania | oceania.pool.ntp.org |
| South America | south-america.pool.ntp.org |

In setup section, we first initialize serial communication with PC and join the WiFi network using WiFi.begin() function.

Serial.begin(115200);

//connect to WiFi

Serial.printf("Connecting to %s ", ssid);

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println(" CONNECTED");

Once ESP32 is connected to the network, we initialize the NTP client using configTime() function to get date and time from an NTP server.

//init and get the time

configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer);

Now we can simply call the printLocalTime() custom function whenever we want to print current date & time.

getLocalTime() function is used to transmit a request packet to a NTP server and parse the received time stamp packet into to a readable format. It takes time structure as a parameter.

You can access the date & time information by accessing members of this time structure.

|  |  |
| --- | --- |
| %A | returns day of week |
| %B | returns month of year |
| %d | returns day of month |
| %Y | returns year |
| %H | returns hour |
| %M | returns minutes |
| %S | returns seconds |

void printLocalTime()

{

struct tm timeinfo;

if(!getLocalTime(&timeinfo)){

Serial.println("Failed to obtain time");

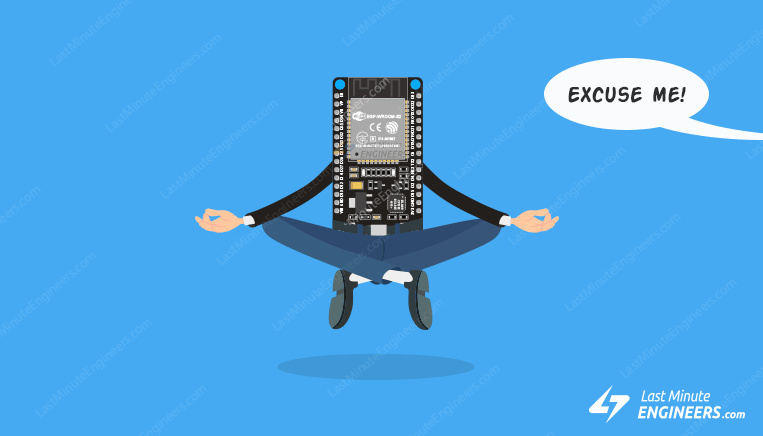
return;

}

Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");

}

**Configuring & Handling ESP32 GPIO Interrupts In Arduino IDE**



Often in a project you need the ESP32 to execute its normal program while continuously monitoring for some kind of event. A solution widely adopted is the use of an Interrupt.

Interrupts In ESP32

The ESP32 offers up to 32 interrupt slots for each core. Each interrupt has a certain priority level and can be categorized into two types.

Hardware Interrupts – These occur in response to an external event. For example, GPIO Interrupt(when a key is pressed down) or a Touch Interrupt(when touch is detected)

Software Interrupts – These occur in response to a software instruction. For example, a Simple Timer Interrupt or Watchdog Timer Interrupt(when timer times out)

ESP32 GPIO Interrupt

In ESP32, we can define an interrupt service routine function that will be called when a GPIO pin changes its signal value.

With an ESP32 board, all the GPIO pins can be configured to function as interrupt request inputs.



Attaching Interrupt to a GPIO Pin

In Arduino IDE, we use a function called attachInterrupt() to set an interrupt on a pin by pin basis. The recommended syntax looks like below.

attachInterrupt(GPIOPin, ISR, Mode);

This function takes three parameters:

GPIOPin – Sets the GPIO pin as an interrupt pin, which tells the ESP32 which pin to monitor.

ISR – Is the name of the function that will be called every time the interrupt is triggered.

Mode – Defines when the interrupt should be triggered. Five constants are predefined as valid values:

|  |  |
| --- | --- |
| LOW | Triggers interrupt whenever the pin is LOW |
| HIGH | Triggers interrupt whenever the pin is HIGH |
| CHANGE | Triggers interrupt whenever the pin changes value, from HIGH to LOW or LOW to HIGH |
| FALLING | Triggers interrupt when the pin goes from HIGH to LOW |
| RISING | Triggers interrupt when the pin goes from LOW to HIGH |

Detaching Interrupt from a GPIO Pin

You can optionally call detachInterrupt() function when you no longer want ESP32 to monitor a pin. The recommended syntax looks like below.

detachInterrupt(GPIOPin);

Interrupt Service Routine

Interrupt Service Routine is invoked when an interrupt occurs on any GPIO pin. Its syntax looks like below.

void IRAM\_ATTR ISR() {

Statements;

}

ISRs in ESP32 are special kinds of functions that have some unique rules most other functions do not have.

* The interrupt service routine must have an execution time as short as possible, because it blocks the normal program execution.
* Interrupt service routines should have the IRAM\_ATTR attribute, according to the [ESP32 documentation](https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/general-notes.html#iram-instruction-ram)

What is IRAM\_ATTR?

By flagging a piece of code with the IRAM\_ATTR attribute we are declaring that the compiled code will be placed in the Internal RAM (IRAM) of the ESP32.

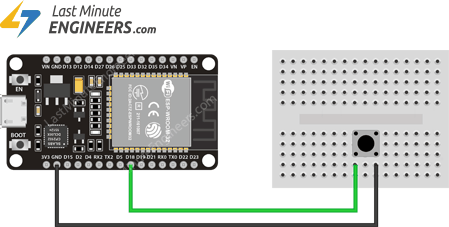
Otherwise the code is placed in the Flash. And flash on the ESP32 is much slower than internal RAM.

If the code we want to run is an interrupt service routine (ISR), we generally want to execute it as quickly as possible. If we had to ‘wait’ for an ISR to load from flash, things would go horribly wrong.

Hardware Hookup

Enough of theory! let’s see practical example.

Let’s hook one push button to GPIO pin 18 (D18) on the ESP32. You don’t require any pullup to this pin as we will pull the pin up internally.

Wiring Push Buttons to ESP32 For GPIO Interrupt

Example: Simple Interrupt

The following sketch demonstrates the use of the interrupts and the correct way to write an interrupt service routine.

struct Button {

const uint8\_t PIN;

uint32\_t numberKeyPresses;

bool pressed;

};

Button button1 = {18, 0, false};

void IRAM\_ATTR isr() {

button1.numberKeyPresses += 1;

button1.pressed = true;

}

void setup() {

Serial.begin(115200);

pinMode(button1.PIN, INPUT\_PULLUP);

attachInterrupt(button1.PIN, isr, FALLING);

}

void loop() {

if (button1.pressed) {

Serial.printf("Button 1 has been pressed %u times\n", button1.numberKeyPresses);

button1.pressed = false;

}

//Detach Interrupt after 1 Minute

static uint32\_t lastMillis = 0;

if (millis() - lastMillis > 60000) {

lastMillis = millis();

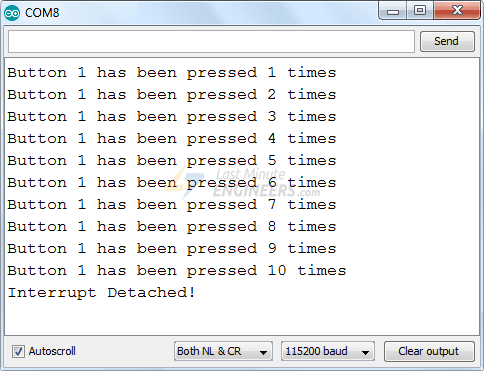
detachInterrupt(button1.PIN);

Serial.println("Interrupt Detached!");

}

}

Once you upload the sketch, press EN button on the ESP32 and open serial monitor at baud rate 115200. Now we will get the output as shown below, when you press the button.



Code Explanation

At the very start of the sketch we create a structure named Button. It has three members viz. pin number, number of key presses and pressed state. If you don’t know, Structure is the collection of variables of different types (but logically related to each other) under a single name.

struct Button {

const uint8\_t PIN;

uint32\_t numberKeyPresses;

bool pressed;

};

Next we create an instance of the Button structure and initialize pin number to 18, number of key presses to 0 and default pressed state to false.

Button button1 = {18, 0, false};

The following piece of code is an Interrupt Service Routine. As mentioned earlier, ISR in ESP32 should have the IRAM\_ATTR attribute.

In ISR we simply increment the KeyPresses counter by 1 & set button pressed state to True.

void IRAM\_ATTR isr() {

button1.numberKeyPresses += 1;

button1.pressed = true;

}

In Setup section of code, we first initialize the serial communication with PC. Then set input pullup the D18 pin.

Next we tell the ESP32 to monitor the D18 pin and call the interrupt service routine isr when the pin goes from HIGH to LOW i.e. FALLING state.

Serial.begin(115200);

pinMode(button1.PIN, INPUT\_PULLUP);

attachInterrupt(button1.PIN, isr, FALLING);

In Loop section of the code, we simply check if the button pressed state returns to be true. When it does, we simply print the number of key pressed till now and set the button pressed state LOW so that we can continue getting next interrupts.

if (button1.pressed) {

Serial.printf("Button 1 has been pressed %u times\n", button1.numberKeyPresses);

button1.pressed = false;

}

In loop section we also check the number of milliseconds that have passed since the program first started using millis() function. When this time is more than 60,000 milliseconds or 1 Minute, we simply tell ESP32 to not to monitor D18 pin using detachInterrupt() function.

//Detach Interrupt after 1 Minute

static uint32\_t lastMillis = 0;

if (millis() - lastMillis > 60000) {

lastMillis = millis();

detachInterrupt(button1.PIN);

Serial.println("Interrupt Detached!");

}