**Introduction**

ESP8266 is a popular chip in the IoT world these days, not least of all because of it’s cheapness (approx $5 USD), its small size, its built-in wifi capabilities, and its relative power for its small size.

In my last tutorial, I wrote about how to use Alexa and an ESP8266 (with attached IR transmitter) to allow us to turn our TV off and on by voice command. That tutorial required, among other things, for the code running on the ESP8266 chip to connect itself to the local wifi network. In order to allow it to connect to our wifi, we hard-coded the wifi network ssid and password on the chip. Each of us who went through that tutorial had to customize those settings for our own local networks.

Now, while that was all well and good for a personal experiment in our homes, it’s obviously not at all suitable for a production-ready solution. Imagine that our device is to become an actual consumer product. We could hardly expect each customer to install Arduino or RTOS, prepare their development environments, set their wifi credentials in a .h file, and then compile & flash the code to the chip. Quite clearly we need to provide an easy user-friendly way to set the wifi credentials. And that must be the first thing that the user does after hooking up the device to a power supply; because without wifi credentials, the rest of the program won’t work.

Probably the easiest way to allow the user to control the wifi credentials (and maybe any other settings that we want to expose to the user as configurables) on the chip would be to allow the user to somehow discover and connect to the chip using their mobile devices. Easy, so we just run a server on the chip that accepts http requests from a client… oh no, wait. How will they know the IP? Ok so we run a UDP listener on the server, and then… oh, we need to be on the wifi for that. So we just… um… no wait… hard-code the wifi credentials?

We quickly see the conundrum we’re in. Logically, we need to either make the ESP8266 connect to something constant and reliable already on the network from which to read the config, or we need to make the ESP8266 a constant findable entity, and push the config to it. The first option is not practical. There’s a way, however, to implement the second way.

**On-Chip Storage**

As a bonus, we’ll also introduce the use of an EEPROM database for on-chip storage of data. In the case of this tutorial, the data we’ll store is the wifi ssid and password. That data, once written, will be stored persistently on the chip. So even if we remove the chip’s power source, then power it up again later, that data will stay as written if not explicitly overwritten.

EEPROM; what’s EEPROM? EEPROM is a very old standard for persistent storage of data on an electronic medium. EEPROM isn’t our only choice for persistent storage. ESP8266 can support FAT, Posix, Spiffs, and other varying degrees of filesystem. We’re using EEPROM for this example because it’s the simplest, quickest solution (in my opinion, and I’m the one writing the code) that will do what we want. For the amount of data that we need to store, we won’t really require the benefits of using a more complex storage interface (again, in my opinion), and it’s reliable enough to do the job.

**EEPROM Structure**

Let me explain just for a minute, if you care, how our EEPROM storage will be structured. It’s ok to skip this section; it’s just for reference in case you care to know.

In our tiny ‘database’ we have three records that we want to store: 2 actual records with our data, and 1 meta-data record.

* Record 0: (meta-record) a unique GUID flag
* Record 1: wifi SSID
* Record 2: wifi password

The 0th record is a flag that we’ll use to determine whether or not data’s ever yet been written to the EEPROM on this chip. It’s a hard-coded unique value. Anytime we write data to EEPROM, we will write that one very specific value to the 0th record. When we read the EEPROM, if we see that it’s present, then we know that the EEPROM on the chip has been used before and likely has some values. If not, then we know that the EEPROM’s never been written to.

EEPROM is not by default divided into records, so that’s another structure that we have to impose ourselves. EEPROM is just a single fixed-size block of memory on the chip. We specify how big we want the EEPROM block to be, at startup. We’ve specified [TODO: EEPROM size], which is way more than we’ll need. I’ve delimited records by a 0-value byte, and then end of final record by 3 consecutive 0-value bytes. The data in each record is just a simple ASCII string. So once written, the very beginning of the EEPROM block will start off with our GUID meta-record value, followed by a single 0-value byte. The next byte will start the wifi SSID, and that full string will be followed by a 0-value byte. Next byte will start the wifi password string; at the end of that string will be a 0-value byte, followed by 2 more 0-value bytes to mark the end of storage. The rest of EEPROM will just be random bytes that we’ll ignore.

So for an example: let’s say that our GUID is 36 characters long. So the first 36 bytes of EEPROM will contain that GUID, and the 37th byte will be a 0. The wifi ssid is “mina” and the password is “1234”. So the 38th byte is “m”, and the 41st byte is “a” (ASCII). The 42nd is a 0. The 43rd byte is “1” (ASCII code), and the 47th, 48th, and 49th bytes are all 0’s. After that, we don’t care.

**Strategy**

The program on our ESP8266 will accept HTTP connections. Yes, we can easily do that! There are many handy libraries for running a tiny web server on the ESP, and for this tutorial we’ll just be using the default one for Arduino ESP8266.

There are two cases or paths that we need to handle here. Let’s call these “normal mode” and “setup mode”. In the first case, “normal mode”, the chip has been set up before, and is able to connect to the local wifi already. In that case, our code is simple; we’ll just accept incoming http requests to *modify* or *read* the stored data off of the chip, and that’s it. (For security reasons we won’t return the password, that will be write-only). In this case the user’s allowed to view the wifi username already configured, and is able to write new values for both the username & password.

In the second case, the chip has either never been set up, or is unable to connect to the wifi. In this case, we’ll be entering “setup mode”. Setup mode is more complicated. In setup mode, the ESP8266 stops trying to connect to a wifi network, and broadcasts itself as an AP (wifi access point). At this point, if you open up the wifi settings on your mobile, and you’re within range, you’ll see this new network appear (we’ll give it a unique name). You’ll be able to connect directly to that AP. Once connected, the AP will configure DNS in such a way that *all* requests of any kind get routed to the web server on the chip. And that’s the same web server in “normal mode” described above; it allows you to write/read the stored data settings on the chip. Once written, the chip will try to connect to the specified wifi network. If successful, then we go into “normal mode”.

[TODO: flowchart]

Start

Do we have saved wifi ssid/passwd?

Can connect to wifi?

Enter setup mode; open AP

Can connect to wifi?

Enter setup mode; open AP

Enter setup mode; open AP

Enter setup mode; open AP

Got request?

Can connect to wifi?

Save wifi info

Send IP address in response

Run in normal mode

**Hardware:**

To complete this tutorial, you’re going to need to obtain some items on your own, all of which are easy to obtain. There are fewer pieces of equipment required than in my previous IoT tutorial, and the total bill is orders of magnitude cheaper.

* An ESP8266 chip. They cost about just a few USD at the time of this writing. <http://www.ebay.com/itm/ESP8266-ESP-12E-Serial-WIFI-Wireless-Transceiver-Wireless-Module-LWIP-AP-STA-/191607430420> (approx $5 US)
* A serial adapter which on one side is USB (to plug into your dev computer), and the other side fits into the ESP8266 chip. (a few $ US)
* One or more local wifi networks to which you know the username and password.

**Software:**

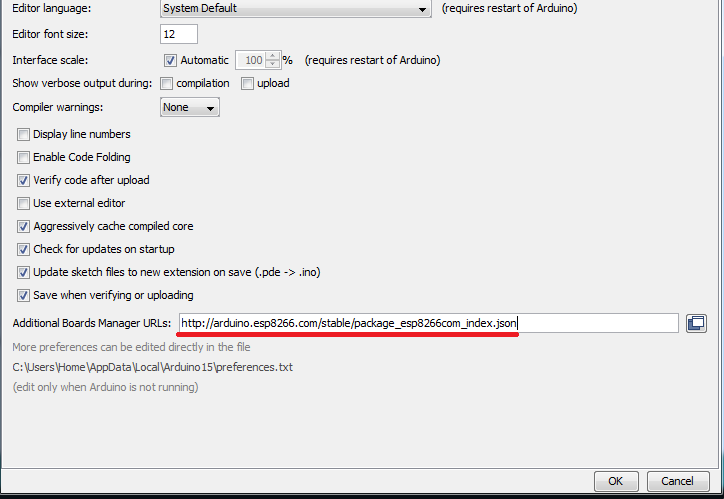
* Arduino IDE. There are versions for all major OSes, including Windows. This tutorial was developed on the Ubuntu version, but I’ve installed and used Arduino on Windows as well, no problems.
* The ESP8266 development library for Arduino. <https://github.com/esp8266/Arduino>
* Drivers: drivers for your adapter should most likely be plug & play, or no additional drivers required

**Setup:**

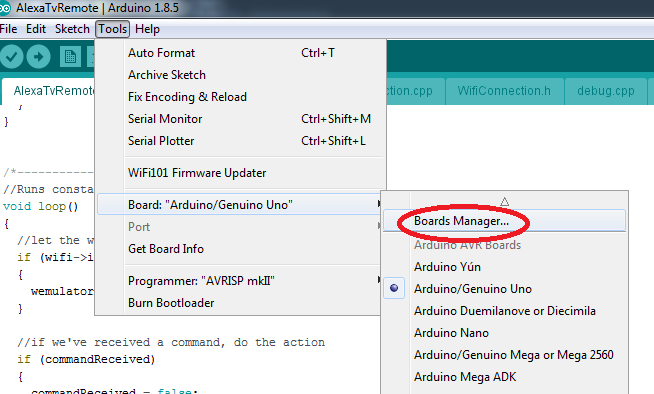
* Install Arduino IDE <https://www.arduino.cc/en/Guide/Windows>
* Install the ESP8266 library using Boards Manager

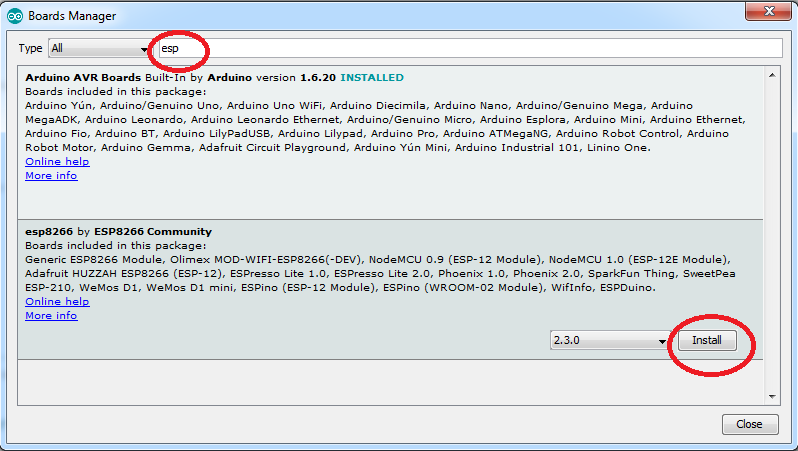
To install ESP8266 library using Boards Manager:

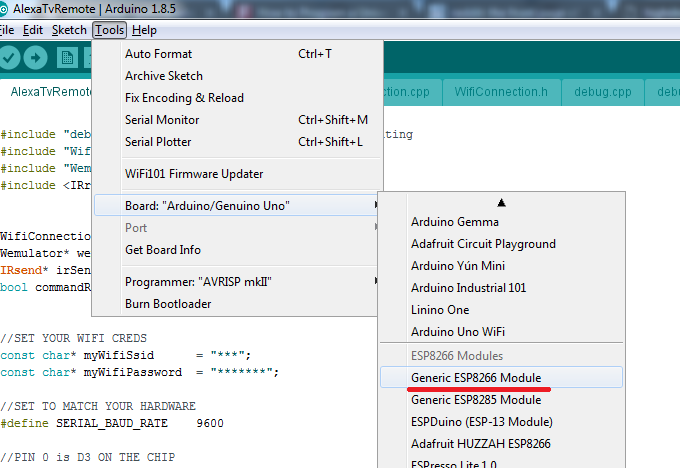
* In Arduino IDE, open File -> Preferences
* Enter this URL in “Additional Boards Manager URL”: <http://arduino.esp8266.com/stable/package_esp8266com_index.json>
* Click OK



* Go to Boards Manager (Tools -> Board: [current board] -> Boards Manager)
* In the ‘filter’ textbox, type “ESP8266”
* You should get an entry for “esp8266” now that you have the additional boards manager added. Choose it, and click “install”.
* Wait a while – it takes a while to download everything.
* Restart your Arduino IDE.
* Open Tools -> Board: -> this time scroll down to “Generic ESP8266 Module”, and select it.





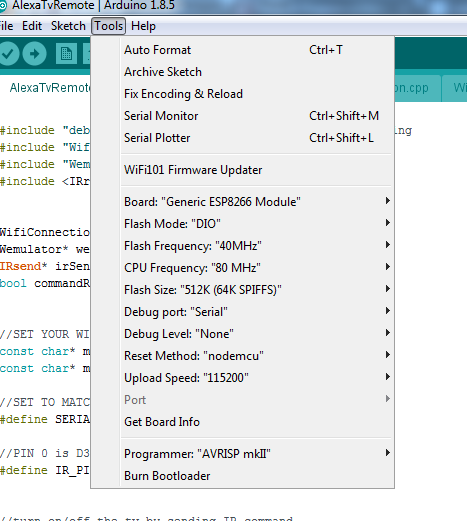


**Adding Third Party Libraries**

Adding third party libraries to Arduino is super easy; however, no third party libraries are required for this tutorial.

**Settings**

The image below shows typical settings - which work for me and my hardware - but may vary for each user. You can try the settings below, but there’s a chance that you may have to adjust them based on your particular chip and adapter. Mine is nodemcu, for example, so I had to change reset method from “ck” (the default) to “nodemcu”. Also, set “debug port” to “serial” so that you can use the serial debugger. Mine is a very typical setup, so you can use my settings as a base; I’m just saying don’t be surprised if you have to mess with them to get the compile & flash process to work.



**Prove your Setup:**

ESP8266 Hello World:

Arduino projects start with an .ino file. The .ino file defines two points of entry: setup and loop. For our “hello world”, we’re going to turn a little light on, on the ESP8266, just to verify that our code works.

//SET TO MATCH YOUR HARDWARE

#define SERIAL\_BAUD\_RATE 9600

/\*---------------------------------------\*/

//Runs once, when device is powered on or code has just been flashed

void setup()

{

//if set wrong, your serial debugger will not be readable

Serial.begin(SERIAL\_BAUD\_RATE);

}

/\*---------------------------------------\*/

//Runs constantly

void loop()

{

pinMode(3, 1);

delay(1000);

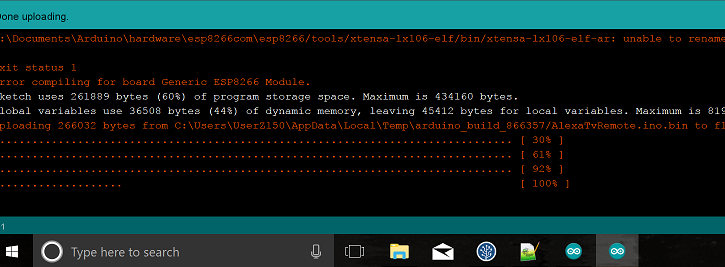
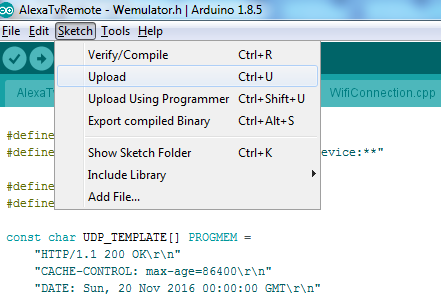
pinMode(3, 0);

delay(1000);

}

**Compile and Flash the Code**

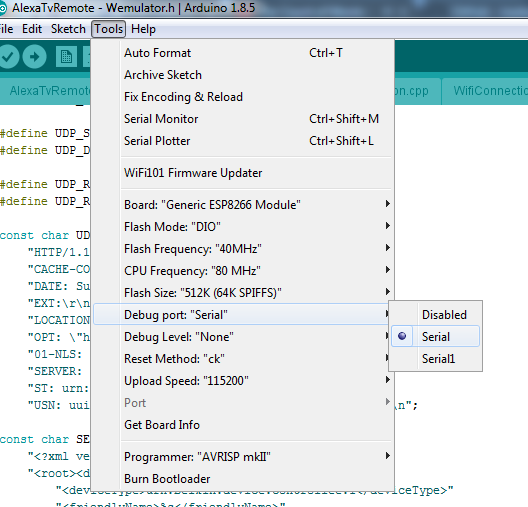
To compile and flash are easy steps, if your setup so far is correct. To compile without flashing, just go to Sketch -> Verify/Compile from the Arduino menu. To flash the code to the chip as well as compile, select Sketch -> Upload from the Arduino menu.



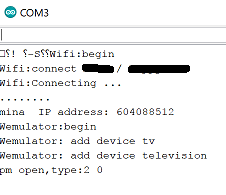
If the flash is successful, you will see a progress display go from 0% to 100%, during which time the LED on your chip will most likely actually blink or flash.

To test that serial debugging is working:

* First make sure that Debug Port is set to Serial (Tools -> Debug port)
* After your code has finished flashing to the chip, select Tools -> Serial Monitor



Output of serial debugger after a successful start:



[TODO: new image]

[TODO: go thru tutorial]

**Putting it Together:**

So, this is a proof of concept that can be used for a production app, but obviously it’s just a proof of concept. How to go from this to a full app?

Well, first off, we assume that the app does something useful. Our demo app here does literally nothing aside from allowing setup, except blink the LED on the chip. We must assume that your production-ready app actually does something!

Secondly, we need an admin app connecting to the chip, from which the user can do the setup. The instructions to the user would be as simple as possible. For example: “Just download the [myproduct] app, open it, and follow the instructions”. What I would do is, either create an IOS app and an Android app with identical functionality, or create a single solution using something like PhoneGap. The admin app(s) would either have the IP of the chip stored, or would not. If not, then it would send out a UDP request to discover the chip. If discovered, it would record the IP, and move on. If not discovered, it would search for the AP provided by the chip. If found, it would connect to that, and present the user with a form in which they can enter their wifi ssid/password. When the user submits the form, it would send the appropriate requests to the chip’s web server, wait for the chip to send its IP address, store it, and move on. At that point we’re connected to the chip, and can send it requests at will. If at any time the admin app loses connection to the chip, that discovery process is repeated.

Start, or user request

Great!

Enter normal mode; return to start

N

N

Y

Y

Y

N

N

Y

N

Success?

Connect to AP, set data via HTTP requests

Found it?

Look for ESP8266’s AP

Save the IP of the server

Send out UDP request

Found anything?

Setup failed

Do we have saved IP?

Does it work?