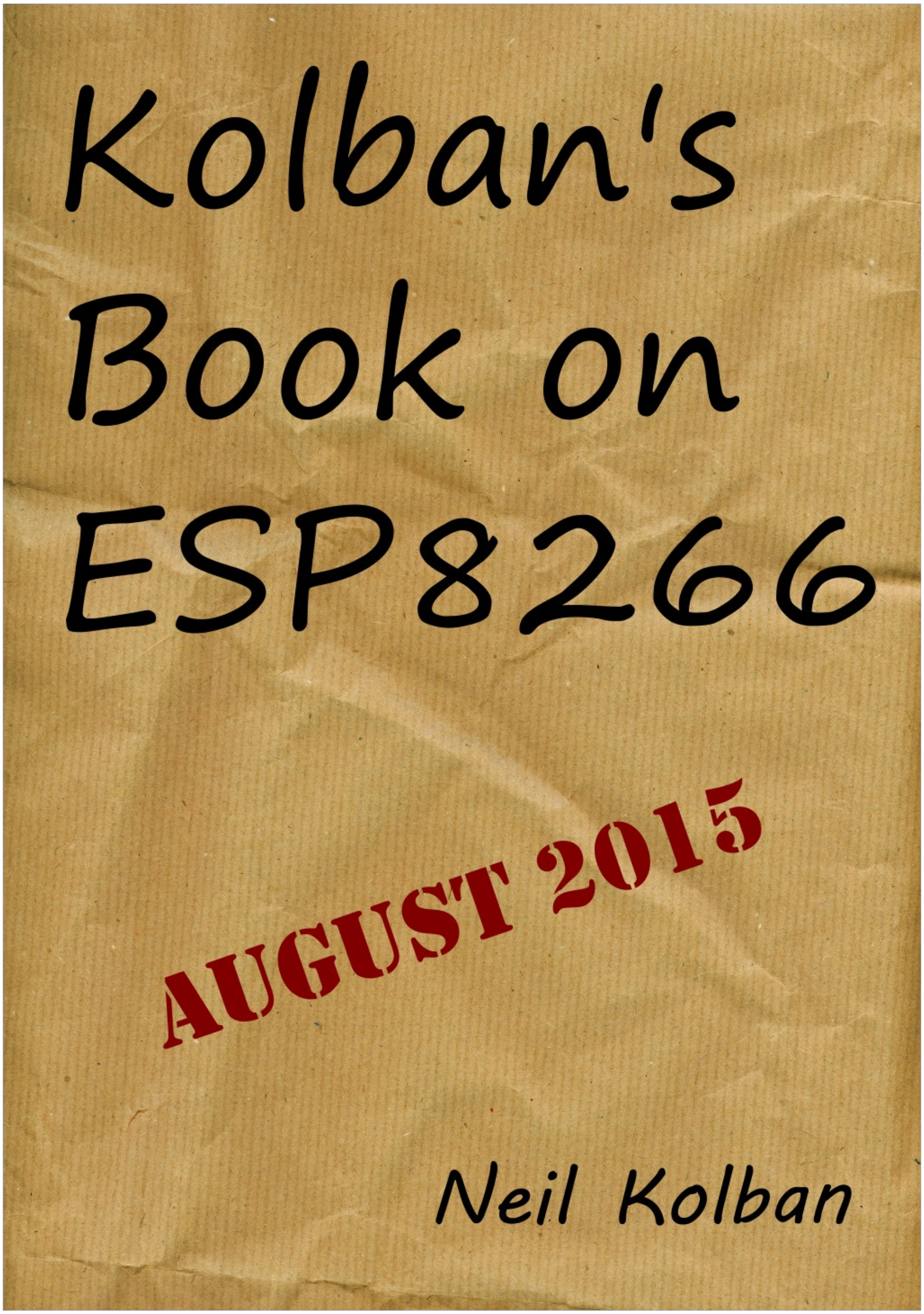

Kolban's
Book on
ESP8266
AUGUST 2015
Neil Kolban

Table of Contents

Introduction

Howdy Folks,

Ive been working in the software business for over 30 years but until recently, hadn't been playing directly with Micro Processors. When I bought a Raspberry PI and then an Arduino, I'm afraid I got hooked. In my house I am surrounded by computers of all shapes, sizes and capacities … any one of them with orders of magnitude more power than any of these small devices … however, I still found myself fascinated.

When I stumbled across the ESP8266 earlier this year, it peaked my interest. I hadn't touched C programming in decades (I'm a Java man these days). As I started to read what was available in the way of documentation from the excellent community surrounding the device, I found that there were only small pockets of knowledge. The best source of information was (and still is) the official PDFs for the SDK from Espressif (the makers of the ESP8266) but even that is quite "light" on examples and background. As I studied the device, I started to make notes and my pages of notes continued to grow and grow.

This book (if we want to call it that) is my collated and polished version of those notes. Rather than keep them to myself, I offer them to all of us in the ESP8266 community in the hope that they will be of some value. My plan is to continue to update this work as we all learn more and share what we find in the community forums. As such, I will re-release the work at regular intervals.

As you read, make sure that you fully understand that there are undoubtedly inaccuracies, errors in my understanding and errors in my writing. Only by feedback and time will we be able to correct those. Please forgive the grammatical errors and spelling mistakes that my spell checker hasn't caught.

Please don't email me directly with technical questions. Instead, let us use the forum and ask and answer the questions as a great community of ESP8266 minded enthusiasts, hobbyists and professionals.

Neil Kolban

Texas, USA

Overview

A microprocessor is an integrated circuit that is capable of running programs. There are many instances of those on the market today from a variety of manufacturers. The prices of these microprocessors keeps falling. In the hobbyist market, an open source architecture called "Arduino" that uses the Atmel range of processors has caught the imagination of countless folks. The boards containing these Atmel chips combined with a convention for connections and also a free set of development tools has lowered the entry point for playing with electronics to virtually nill. Unlike a PC, these processors are extremely low end with low amounts of ram and storage capabilities. They won't be replacing the desktop or laptop any time soon. For those who want more "oomph" in their processors, the folks over at Raspberry PI have developed a very cheap (~$45) board that is based on the ARM processors that has much more memory and uses micro SD for persistent data storage. These devices run a variant of the Linux operating system. I'm not going to talk further about the Raspberry PI as it is in the class of "computer" as opposed to microprocessor.

These microprocessors and architectures are great and there will always be a place for them. However, there is a catch … and that is networking. These devices have an amazing set of capabilities including direct electrical inputs and outputs (GPIOs) and support for a variety of protocols including SPI, I2C, UART and more, however, none of them so far come with wireless networking included.

No question (in my mind) that the Arduino has captured everyone’s attention. The Arduino is based on the Atmel chips and has a variety of physical sizes in its open hardware footprints. The primary microprocessor used is the ATmega328. One can find instances of these raw processors on ebay for under $2 with fully constructed boards containing them for under $3. This is 10-20 times cheaper than the Raspberry PI. Of course, one gets dramatically less than the Raspberry PI so comparison can become odd … however if what one wants to do is tinker with electronics or make some simple devices that connect to LEDs, switches or sensors, then the functional features needed become closer.

Between them, the Arduino and the Raspberry PI appear to have all the needs covered. If that were the case, this would be a very short book. Let us add the twist that we started with … wireless networking. To have a device move a robot chassis or flash LED patterns or make some noises or read data from a sensor and beep when the temperature gets too high … these are all great and worthy projects. However, we are all very much aware of the value of the Internet. Our computers are Internet connected, our phones are connected, we watch TV (Netflix) over the Internet, we play games over the Internet, we socialize (??) over the Internet … and so on. The Internet has become such a basic commodity that we would laugh if someone offered us a new computer or a phone that lacked the ability to go "on-line".

Now imagine what a microprocessor with native wireless Internet could do for us? This would be a processor which could run applications as well as or better than an Arduino, which would have GPIO and hardware protocol support, would have RAM and flash memory … but would have the killer new feature that it would also be able to form Internet connections. And that … simply put … is what the ESP8266 device is. It is an alternative microprocessor to the ones already mentioned but also has WiFi and TCP/IP (Transmission Control Protocol / Internet Protocol) support already built in. What is more, it is also not much more expensive than an Arduino. Searching ebay, we find ESP8266 boards under $3.

The ESP8266

The ESP8266 is the name of a microprocessor designed by Espressif Systems. Espressif is a Chinese company based out of Shanghai. The ESP8266 advertises itself as a self-contained WiFi networking solution offering itself as a bridge from existing microprocessors to WiFi … and … is also capable of running self contained applications.

Volume production of the ESP8266 didn't start until the beginning of 2014 which means that, in the scheme of things, this is a brand new entry in the line-up of processors. And … in our technology hungry world, new commonly equates to interesting. A couple of years after IC production, 3rd party OEMs are taking these chips and building "breakout boards" for them. If I were to hand you a raw ESP8266 straight from the factory, it is unlikely we would know what to do with one. They are very tiny and virtually impossible for hobbyists to attach wires to allow them to be plugged into breadboards. Thankfully, these OEMs bulk purchase the ICs, design basic circuits, design printed circuit boards and construct pre-made boards with the ICs pre-attached immediately ready for our use. It is these boards that capture our interest and that we can buy for a few dollars on ebay.

There are a variety of board styles available. The two that I am going to focus on have been given the names ESP-1 and ESP-12. It is important to note that there is only one ESP8266 processor and it is this processor that is found on ALL breakout boards. What distinguishes one board from another is the number of GPIO pins exposed, the amount of flash memory provided, the style of connector pins and various other considerations related to construction. From a programming perspective, they are all the same.

Maturity

It is my belief that the ESP8266 is immature. This is not a bad thing. Everybody and everything has to start somewhere. On the plus side, there is a whole new wealth of territory to be explored and new features and functions and usage patterns to be discovered. On the down side, it does not yet have the richness of tutorials, samples and videos that accompany other microprocessor systems. Its documentation is not brilliant and some of the core questions on its usage are still being examined. How this sits with you is a function of your intent of tinkering in this area. If you want to follow the paths that have been followed many times before, other processors will be more attractive. However if you like a sense of adventure and getting in on the "ground floor" of a new arrival, the challenges that we (the ESP8266 community) are trying to solve may actively excite you rather than dissuade you.

It is also a major reason that folks like myself spend many, many hours studying and documenting what we find … so others can hopefully build on what has been learned without re-inventing the wheel.

Could the excitement about ESP8266 processors fizzle? Yes … these devices may just be a flash in the pan and a few years from now, the hobbyist won't give a second thought about them. But what I ask you is to approach the device with an open mind.

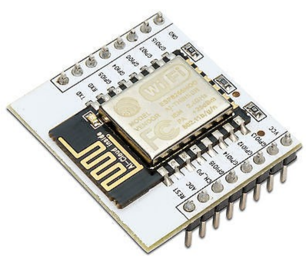
ESP8266 Modules

The ESP8266 integrated circuit comes in a small package, maybe five millimeters square. Obviously, unless you are a master solderer you aren't going to do much with that. The good news is that a number of vendors have created breakout boards that make the job much easier for you. Here we list some of the more common modules.

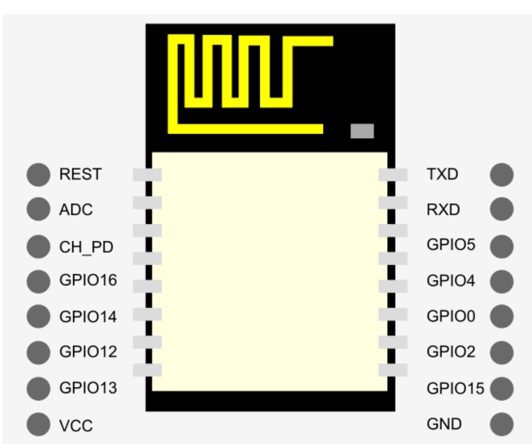
ESP-12

The current most popular and flexible configuration available today is called the ESP-12. It exposes the most GPIO pins for use. The basic ESP-12 module really needs its own expander module to make it breadboard and 0.1" strip board friendly.

Here is what an ESP-12 device looks like when mounted on a breadboard extender board:



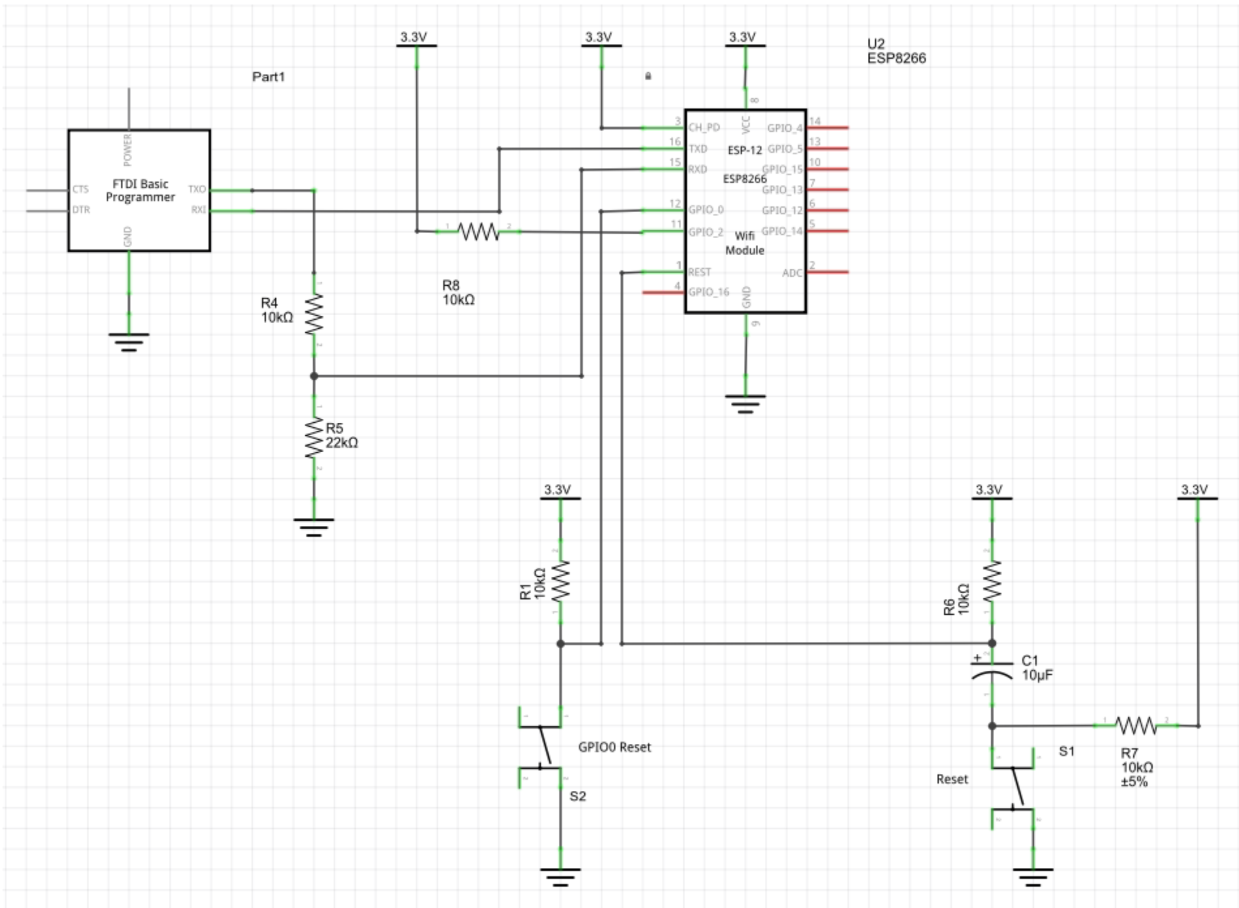
The pin out of the extender board looks as follows:



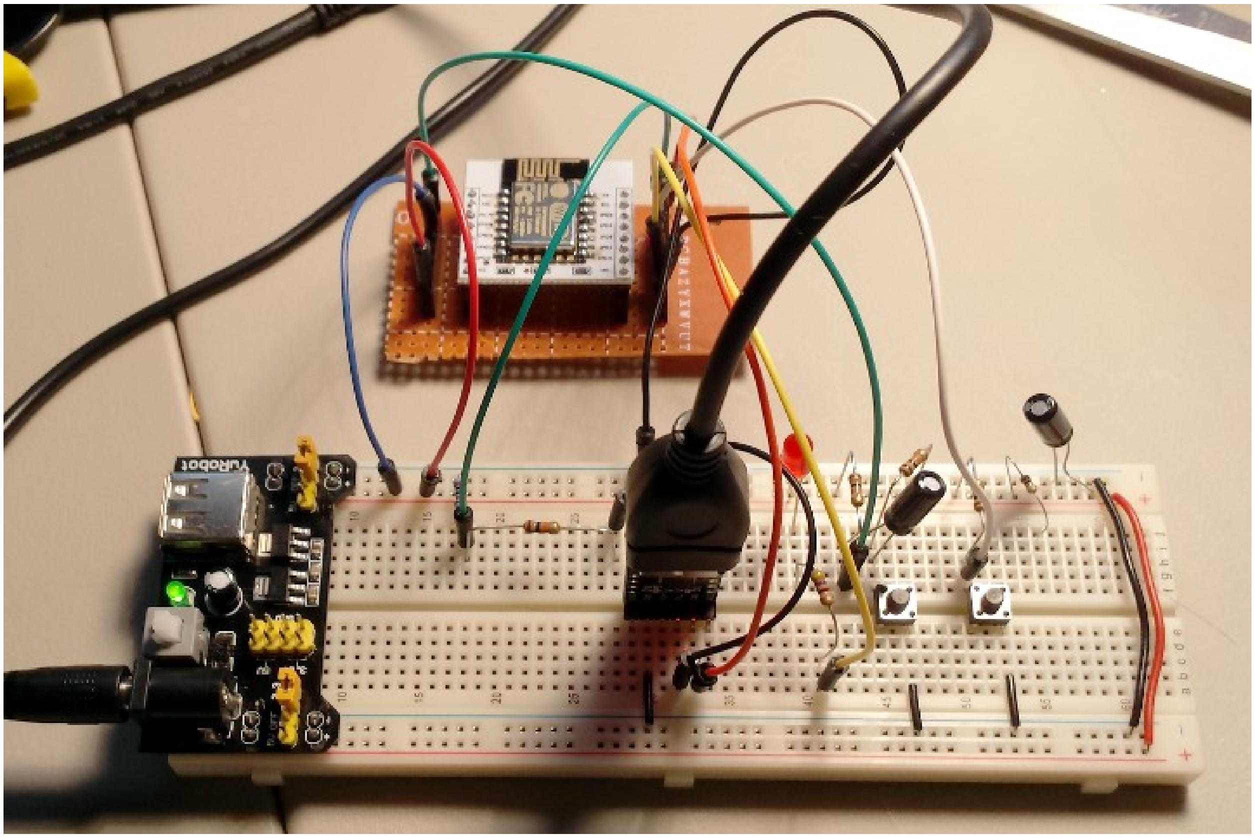
Here is a description of the various pins:

Name	Description
VCC	3.3V.
GPIO 13	Also used for SPI MOSI.
GPIO 12	Also used for SPI MISO.
GPIO 14	Also used for SPI Clock.
GPIO 16	
CH_PD	Chip enable. Should be high for normal operation. • 0 – Disabled • 1 – Enabled
ADC	
REST	External reset. • 0 – Reset • 1 – Normal
TXD	UART 0 transmit.
RXD	UART 0 Receive.
GPIO 4	Regular GPIO.
GPIO 5	Regular GPIO.
GPIO 0	Should be **high** on boot, low for flash update.
GPIO 2	Should be **high** on boot.
GND	Ground.

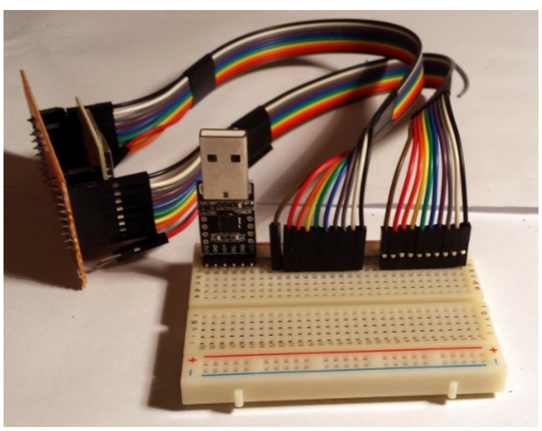
Here is a schematic for connecting an instance:



Next we see an image of this circuit built out on a breadboard.



If we just wish to use our breakout board, we have the following when mounted on a breadboard, we can have the following setup:



This gives us two sets of 8 pin connectors. The first set is:

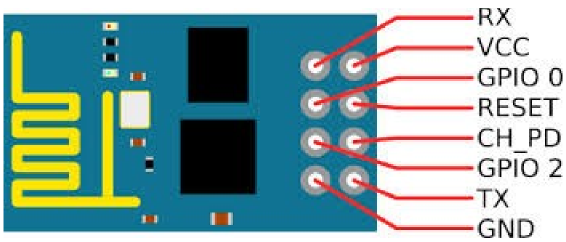
Set 1	
Pin	**Color**
GND	Orange
GPIO15	Yellow
GPIO2	Green
GPIO0	Blue
GPIO5	Purple
GPIO4	Grey
RXD	White
TXD	Black

The second set is:

Set 2	
Pin	**Color**
VCC	Orange
GPIO13	Yellow
GPIO12	Green
GPIO14	Blue
GPIO16	Purple
CH_PD	Grey
ADC	White
REST	Black

ESP-1

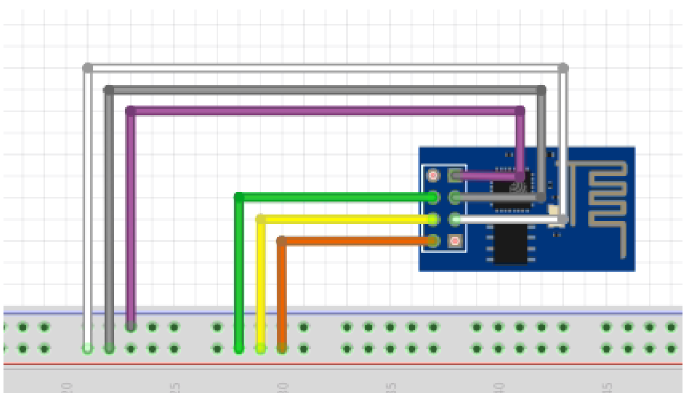
The ESP-1 board is an ESP8266 on an 8 pin board. It is not at all breadboard friendly but fortunately we can make adapters for it extremely easily.



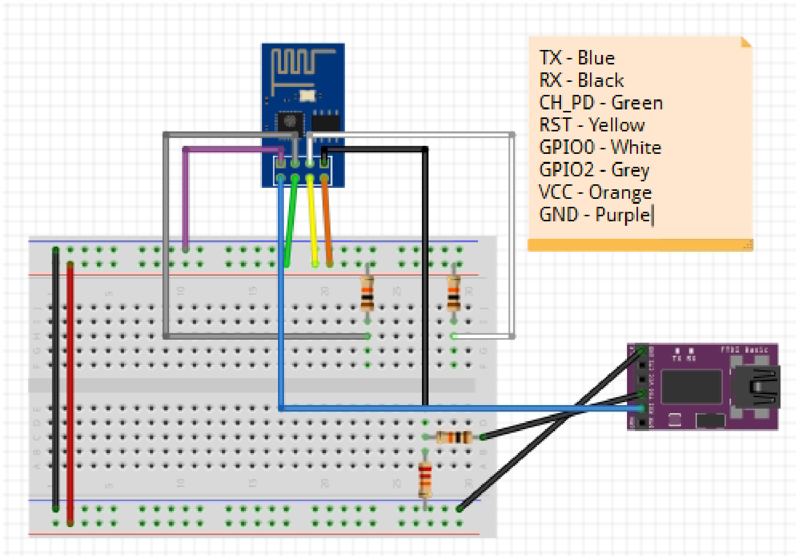
The pin out of the device is as follows:

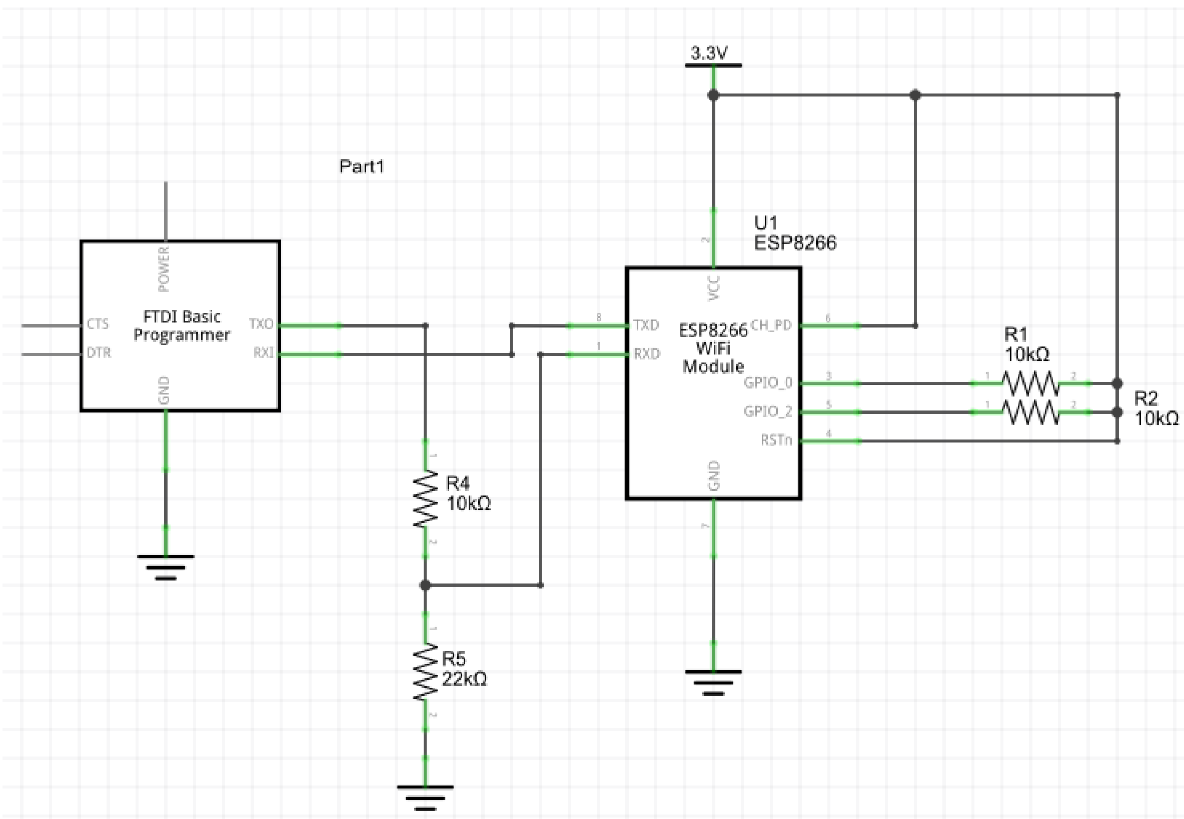
Function	Color	Description
TX		Transmit
RX		Receive. **Always** used a level converter for incoming data. This device is **not** 5V tolerant.
CH_PD		Chip enable. Should be high for normal operation. • 0 – Disabled • 1 – Enabled
RST		External reset. • 0 – Reset • 1 – Normal
GPIO 0		Should be **high** on boot, low for flash update.
GPIO 2		Should be **high** on boot.
VCC		3.3V
GND		Ground

A simple circuit is shown below. Note that the TX and RX pins are shown **not** connected. Remember to **always** use a level converter for the RX pin into the device as it is **not** 5V tolerant.

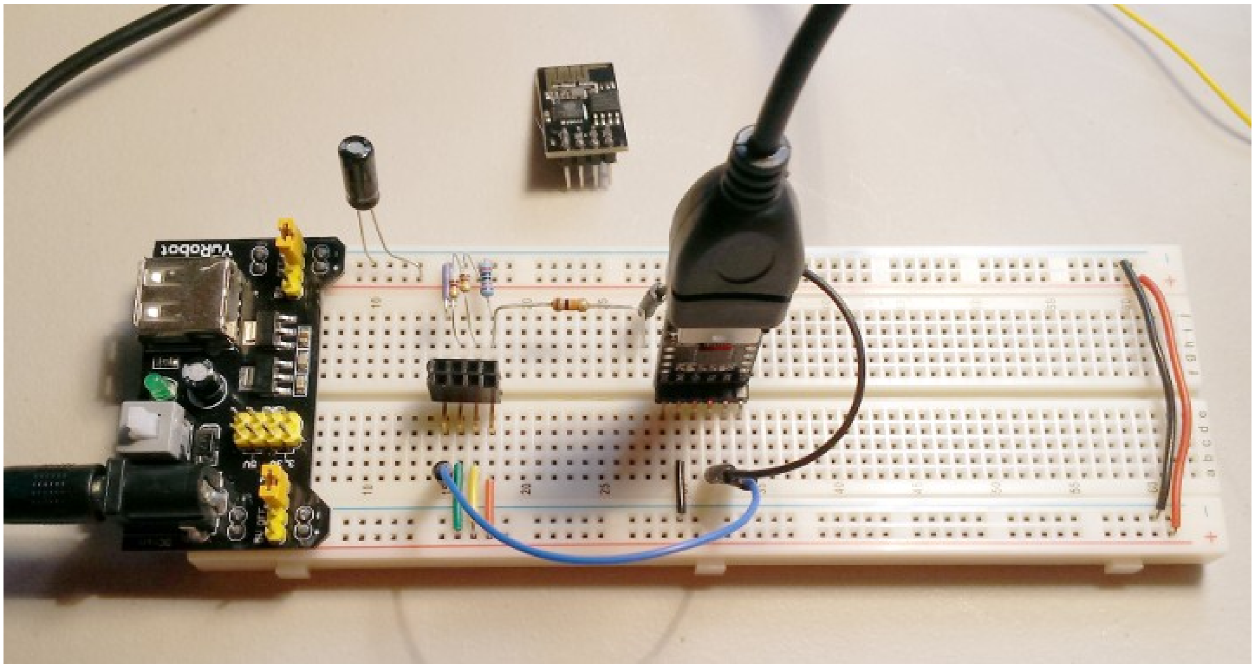


here is an alternate circuit:

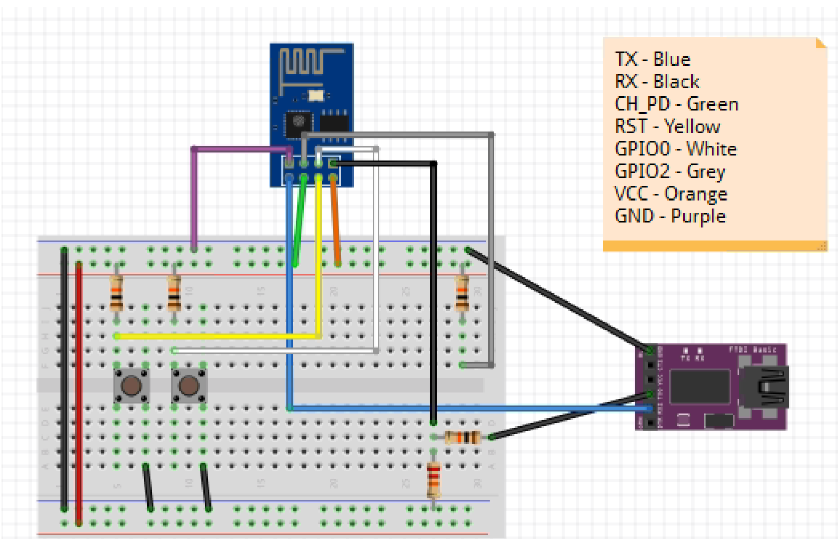


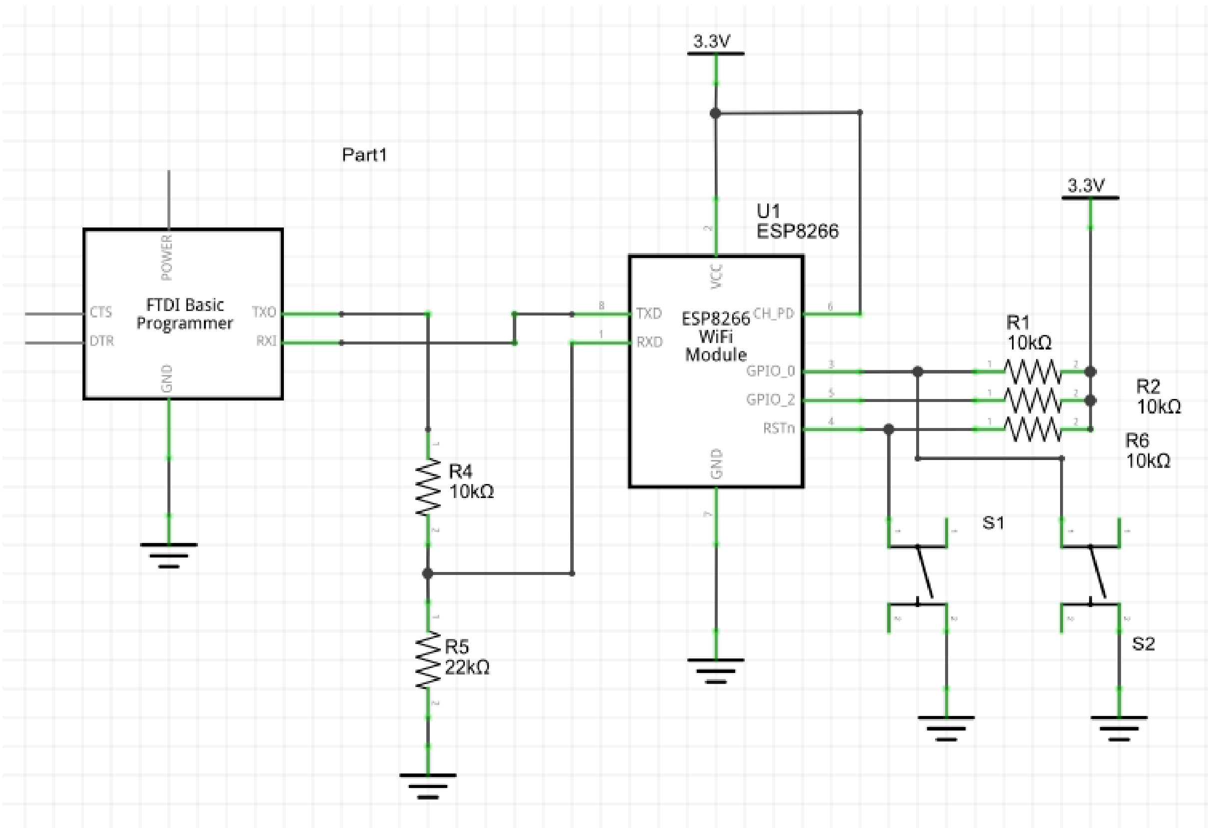


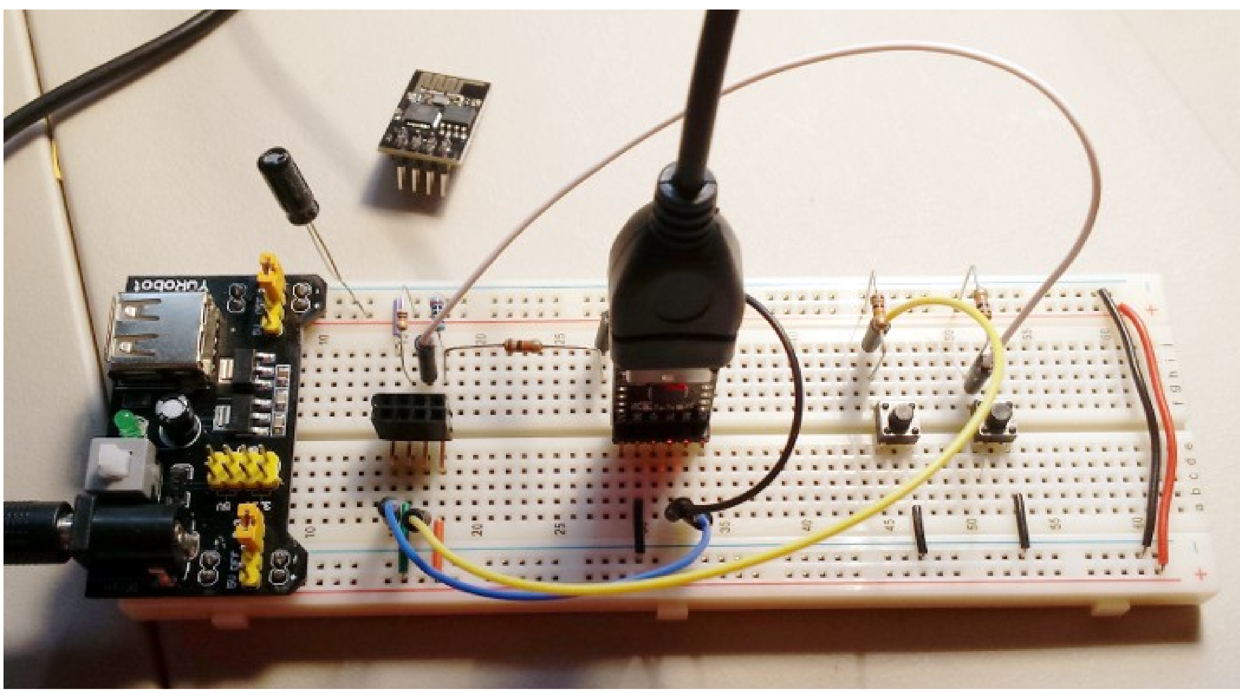
Here is the circuit on a breadboard that was demonstrated to work just fine.



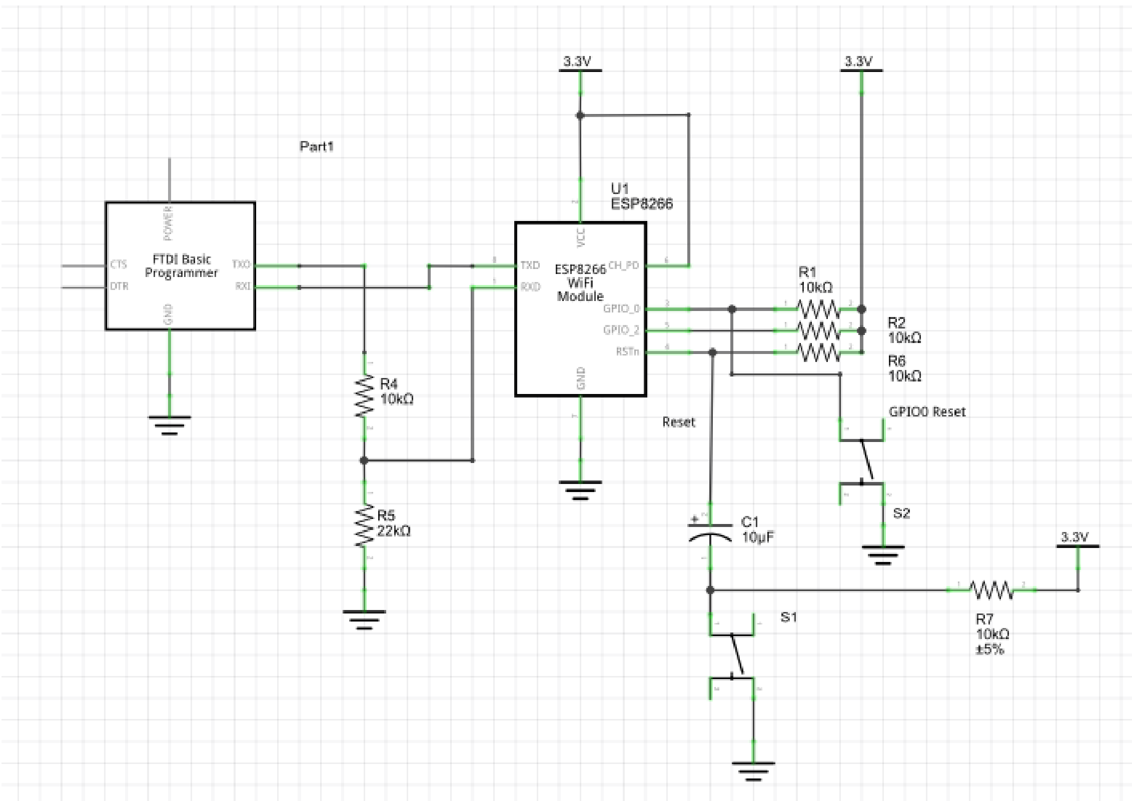
If we wish to add grounding buttons for RESET and GPIO 0, the following are some circuits:

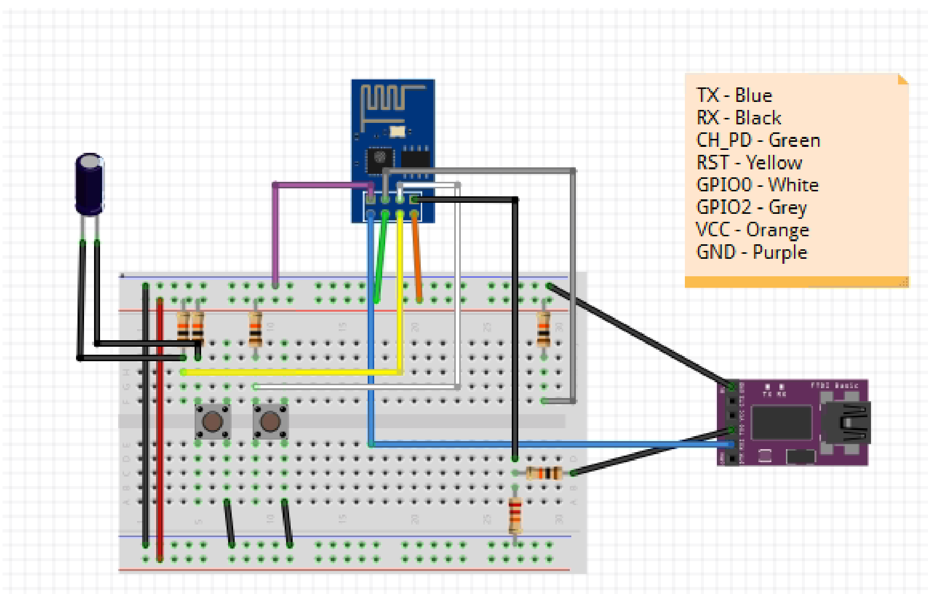






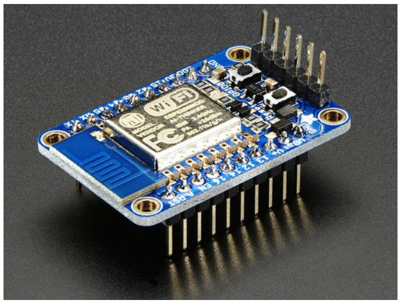
When we press the reset button, it makes sense for that just to be a momentary press. Here is a circuit for that:





The default serial connection speed seems to be 115200.

Adafruit HUZZAH

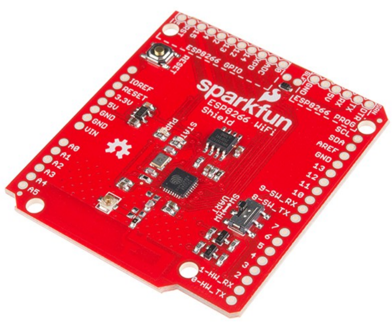


The Adafruit HAZZAH is a breakout board for the ESP8266. It is the most breadboard friendly of the solutions I have encountered so far.

See also:

- Adafruit HUZZAH

SparkFun WiFi Shield – ESP8266



SparkFun have produced a WiFi shield for the Arduino. This is an ESP8266 mounted on a well designed PCB that mates with the Arduino. This makes communicating with the ESP8266 via AT commands extremely easy with no wiring required. Simply push the shield board into the sockets of the Arduino and you are done.

See also:

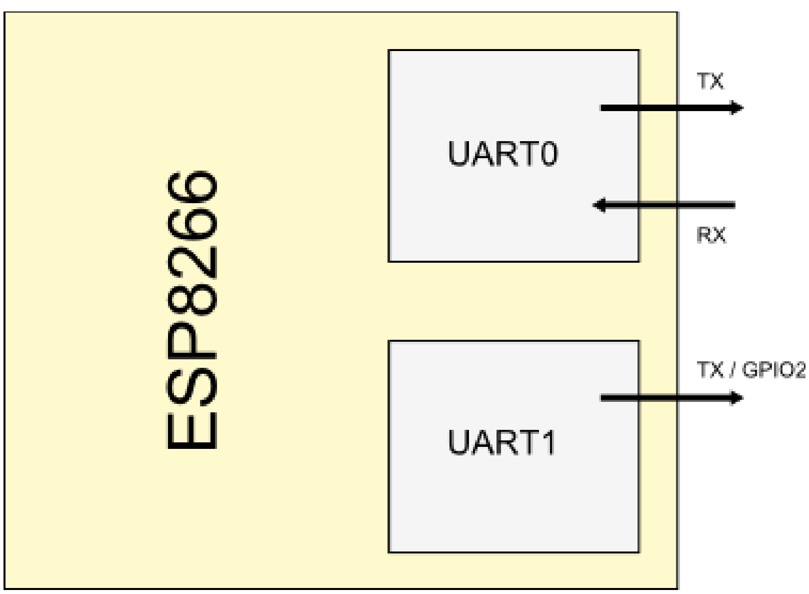
- SparkFun WiFi Shield – ESP8266

Connecting to the ESP8266

The ESP8266 is a WiFi device and hence we will eventually connect to it using WiFi protocols but some bootstrapping is required first. The device doesn't know what network to connect to, which password to use and other necessary parameters. This of course assumes we are connecting as a station, if we wish the device to be an access point or we wish to load our own applications into it, the story gets deeper. This implies that there is a some way to interact with the device other than WiFi and there is … the answer is UART (Serial). The ESP8266 has a dedicated UART interface with pins labeled TX and RX. The TX pin is the ESP8266 transmission (outbound from ESP8266) and the RX pin is used to receive data (inbound into the ESP8266). These pins can be connected to a UART partner. By far the easiest and most convenient partner for us is a USB → UART converter. These are discussed in detail later in the book. For now let us assume that we have set those up. Through the UART, we can attach a terminal emulator to send keystrokes and have received data displayed as characters on the screen. This is used extensively when working with the AT commands. A second purpose of the UART is to receive binary data used to "flash" the flash memory of the device to record new applications for execution. There are a variety of technical tools at our disposal to achieve that task.

When we use a UART, we need to consider the concept of a baud rate. This is the speed of communication of data between the ESP8266 and its partner. During boot, the ESP8266 attempts to automatically determine the baud rate of the partner and match it. It does this by sending some data down the serial line and looking for expected responses. It tries this at different baud rates until it finds a match or gives up. When it gives up, the default is 115200. A side effect of this is that during boot, if you have a terminal emulator attached, you will see what appear to be gibberish data before the normal text. As long as you understand that this is just protocol negotiations and that these are expected, there is nothing further to say. Ignore it.

The ESP8266 has a second UART associated with it that is output only. One of the primary purposes of this second UART is to output diagnostics and debugging information. This can be extremely useful during development and as such I recommend attaching **two** USB → UART converters to the device. The second UART is multiplexed with pin GPIO2.



See also:

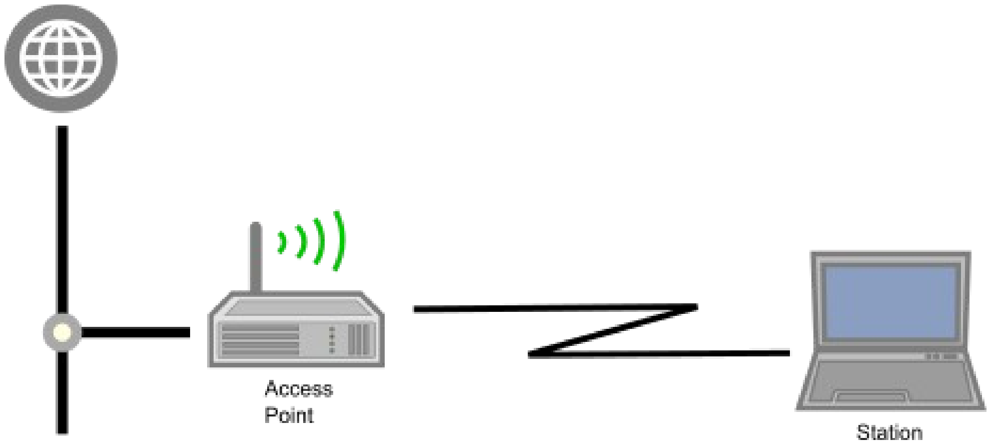
- USB to UART converters
- AT Command Programming
- Flashing the ESP8266
- esptool.py

WiFi Theory

When working with a WiFi oriented device, it is important that we have at least some understanding of the concepts related to WiFi. At a high level, WiFi is the ability to participate in TCP/IP connections over a wireless communication link. WiFi is specifically the set of protocols described in the IEEE 802.11 Wireless LAN architecture.

Within this story, a device called a Wireless Access Point (access point or AP) acts as the hub of all communications. Typically it is connected to (or acts as) as TCP/IP router to the rest of the TCP/IP network. For example, in your home, you are likely to have a WiFi access point connected to your modem (cable or DSL). WiFi connections are then formed to the access point (through devices called stations) and TCP/IP traffic flows through the access point to the Internet.

The devices that connect to the access points are called "stations":



An ESP8266 device can play the role of an Access Point, a Station or both at the same time.

Very commonly, the access point also has a network connection to the Internet and acts as a bridge between the wireless network and the broader TCP/IP network that is the Internet.

A collection of stations that wish to communicate with each other is termed a Basic Service Set (BSS). The common configuration is what is known as an Infrastructure BSS. In this mode, all communications inbound and outbound from an individual station are routed through the access point.

A station must associate itself with an access point in order to participate in the story. A station may only be associated with a single access point at any one time.

Each participant in the network has a unique identifier called the MAC address. This is a 48bit value.

When we have multiple access points within wireless range, the station needs to know with which one to connect. Each access point has a network identifier called the BSSID (or more commonly just SSID). SSID is **s**ervice **s**et **id**entifier. It is a 32 character value that represents the target of packets of information sent over the network.

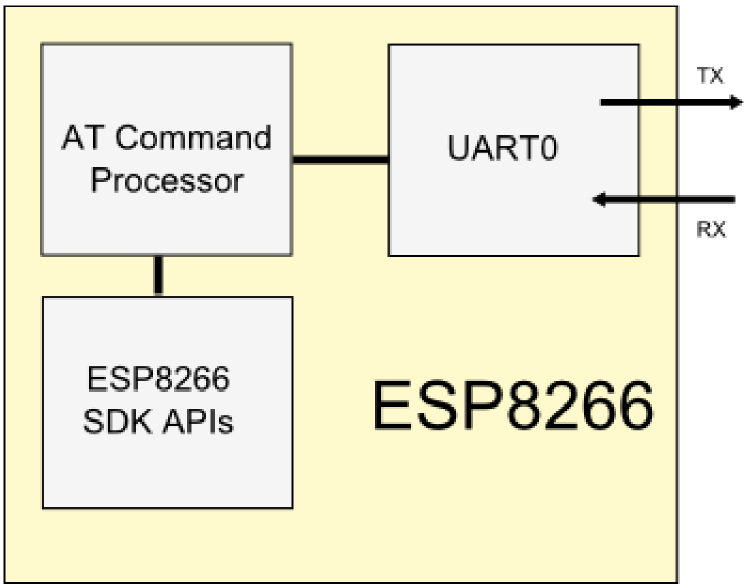
See also:

- Wikipedia – Wireless access point
- Wikipedia – IEEE 802.11
- Wikipedia – WiFi Protected Access
- Wikipedia – IEEE 802.11i-2004

AT Command Programming

The quickest and easiest way to get started with an ESP8266 is to access it via the AT command interface.

When we think about an ESP8266 device we find that it has a built in UART (Serial) connection. This means that it can both send and receive data using the UART protocol. We also know that the device can communicate with WiFi. What if we had an application that ran on the ESP8266 that took "instructions" received over the serial link, executed them and then returned a response? This would then allow us to use the ESP8266 without ever having to know the programming languages that are native to the device. This is exactly what a program that has so far been found to be pre-installed on the ESP8266 does for us. The program is called the "AT command processor" named after the format of the commands sent through the serial link. These commands are all prefixed with "AT" and follow (roughly) the style known as the "Hayes command set".



If we think of an application wishing to use the services of the ESP8266 as a client and the ESP8266 as a server capable of servicing those commands as a server, then the client sends strings of characters through the UART connection to the server and server responds with the outcome.

Espressif publish a complete set of AT command documentation which can be found in their forum page at:

http://bbs.espressif.com/viewforum.php?f=5

There are two primary documents:

- ESP8266EX AT Instruction Set
- ESP8266EX AT Command Examples

Commands

When one has wired an ESP8266 to a serial converter, the next question will be "Is it working?". When we connect a serial monitor, the first command we can send is "AT" which should respond with a simple "OK".

An instruction passed to the device follows one of the following syntax options:

Type	Format	Description
Test	`AT+<x>=?`	Query the parameters and its range of values.
Query	`AT+<x>?`	Return the current value of the parameter.
Set	`AT+<x>=<...>`	Set the value of a parameter.
Execute	`AT+<x>`	Execute a command.

All "AT" instructions end with the "`\r\n`" pair.

Command	Description
`AT`	Returns OK
`AT+RST`	Restart the ESP8266.
`AT+GMR`	Returns firmware version for both the AT command processor and the SDK in use. Currently, the response returned looks like: AT version:0.21.0.0 SDK version:0.9.5
`AT+GSLP=<time>`	Put the device into a deep sleep for a time in milliseconds. It will wake up after this period.
`ATE[0|1]`	Echo AT commands. • ATE0 – Echo commands off • ATE1 – Echo commands on
`AT+RESTORE`	Restore the defaults of settings in flash memory.
`AT+UART_CUR=<baundrate>, <databits>, <stopbits>, <parity>, <flow control>`	
`AT+UART_DEF=<baundrate>, <databits>, <stopbits>, <parity>, <flow control>`	
`AT+SLEEP?`	
`AT+SLEEP=<sleep mode>`	
`AT+RFPOWER=<TX power>`	
`AT+RFVDD?`	
`AT+RFVDD=<VDD33>`	
`AT+RFVDD`	
WIFI	
`AT+CWMODE_CUR=<mode>`	Sets the current mode of operation. • 1 – Station mode • 2 – AP mode • 3 – AP + Station mode
`AT+CWMODE_CUR?`	Get the current mode of operation.

`AT+CWMODE_CUR=?`	Get the list of available modes.
`AT+CWMODE_DEF=<mode>`	Sets the current mode of operation. • 1 – Station mode • 2 – AP mode • 3 – AP + Station mode
`AT+CWMODE_DEF?`	Get the current mode of operation.
`AT+CWMODE_DEF=?`	Get the list of available modes.
`AT+CWJAP_CUR=<ssid>,<password>[,<bssid>]`	Join the WiFi network (JAP = Join Access Point).
`AT+CWJAP_CUR?`	Get the current connection info.
`AT+CWJAP_DEF=<ssid>,<password>[,<bssid>]`	Join the WiFi network (JAP = Join Access Point).
`AT+CWJAP_DEF?`	Get the current connection info.
`AT+CWLAP`	List the "List Access Points". The response is: + CWLAP: <ecn>, <ssid>, <rssi>, <mac>,<ch> where: • ecn 　○ 0 – OPEN 　○ 1 – WEP 　○ 2 – WPA_PSK 　○ 3 – WPA2_PSK 　○ 4 – WPA_WPA2_PSK • ssid – SSID of AP • rssi – Signal strength • mac – MAC address • ch – Channel
`AT+CWLAP=<ssid>,<mac>,<ch>`	List a filtered set of access points.
`AT+CWQAP`	Disconnect from AP.
`AT+CWSAP_CUR?`	Configuration of softAP mode
`AT+CWSAP_CUR=<ssid>, <pwd>, <chl>, <ecn>`	
`AT+CWSAP_DEF?`	Configuration of softAP mode
`AT+CWSAP_DEF=<ssid>, <pwd>, <chl>, <ecn>`	

AT+CWLIF	List of IPs connected in softAP mode
AT+CWDHCP_CUR?	
AT+CWDHCP_CUR=<mode><en>	Enable or disable DHCP. • mode ◦ 0 – softAP ◦ 1 – station ◦ 2 – softAP + station • en ◦ 0 – Enable ◦ 1 – Disable
AT+CWDHCP_DEF?	
AT+CWDHCP_DEF=<mode><en>	Enable or disable DHCP. • mode ◦ 0 – softAP ◦ 1 – station ◦ 2 – softAP + station • en ◦ 0 – Enable ◦ 1 – Disable
AP+CWAUTOCONN=<enable>	
AT+CIPSTAMAC_CUR?	Set/get MAC address of station.
AT+CIPSTAMAC_CUR=<mac>	Set/get MAC address of station.
AT+CIPSTAMAC_DEF?	Set/get MAC address of station.
AT+CIPSTAMAC_DEF=<mac>	Set/get MAC address of station.
AT+CIPAPMAC_CUR?	Set/get MAC address of softAP.
AT+CIPAPMAC_CUR=<mac>	Set/get MAC address of softAP.
AT+CIPAPMAC_DEF?	Set/get MAC address of softAP.
AT+CIPAPMAC_DEF=<mac>	Set/get MAC address of softAP.
AT+CIPSTA_CUR=<iP>	Set the ip address of station.
AT+CIPSTA_CUR?	Get the IP address of station. For example: +CIPSTA:"0.0.0.0"
AT+CIPSTA_DEF=<iP>	Set the ip address of station.

AT+CIPSTA_DEF?	Get the IP address of station. For example: +CIPSTA:"0.0.0.0"
AT+CIPAP_CUR?	Set the ip address of softAP.
AT+CIPAP_CUR=<IP>[,<gateway>, <netmask>]	Set the ip address of softAP.
AT+CIPAP_DEF?	Set the ip address of softAP.
AT+CIPAP_DEF=<IP>[,<gateway>, <netmask>]	Set the ip address of softAP.
AT+CIFSR	Returns the IP address and gateway IP address.
TCP/IP networking	

AT+CIPSTATUS	Information about connection. Response format is:
	STATUS: <stat>
	+ CIPSTATUS: <id>, <type>, <addr>, <port>, <tetype>
	• stat
	  ○ 2 – Got IP
	  ○ 3 – Connected
	  ○ 4 – Disconnected
	• id – Id of the connection
	• type – TCP or UDP
	• addr – IP address
	• port – Port number
	• tetype
	  ○ 0 – ESP8266 runs as client
	  ○ 1 – ESP8266 runs as server

AT+CIPSTART=<type>, <addr>, <port>[, <local port>, <mode>]	Start a connection when CIPMUX=0.
	• type – TCP or UDP
	• addr – Remote IP address
	• port – Remote port
	• local port – For UDP only
	• mode – For UDP only
	  ○ 0 – destination peer entity of UDP is fixed
	  ○ 1 – destination peer entity may change once
	  ○ 2 – destination peer entity may change

AT+CIPSTART=<id>, <type>, <addr>, <port>[,<local port>, <mode>]	Start a connection when CIPMUX=1. • id – 0-4 value of connection • type – TCP or UDP • addr – Remote IP address

	• port – Remote port
	• local port – For UDP only
	• mode – For UDP only
	  ○ 0 – destination peer entity of UDP is fixed
	  ○ 1 – destination peer entity may change once
	  ○ 2 – destination peer entity may change

`AT+CIPSTART=?`	???
`AT+CIPSEND=<length>`	Send length characters.
`AT+CIPCLOSE`	Close a connection.
`AT+CIFSR`	Get the local IP address.
`AT+CIPMUX=<mode>`	Enable multiple connections. • 0 – Single connection. • 1 – Multiple connections.
`AT+CIPMUX?`	Returns the current value for CIPMUX. • 0 – Single connection. • 1 – Multiple connections.

`AT+CIPSERVER=<mode>[,<port>]`	Configure as a TCP server. If no port is supplied, default is 333. A server may only be created when CIPMUX=1 (allow multiple connections).
	• mode
	  ○ 0 – Delete server (needs a restart after)
	  ○ 1 – Create server

`AT+CIPMODE=<mode>`	Set the transfer mode. • 0 – Normal mode. • 1 – Unvarnished mode.
`AT+CIPSTO=<time>`	Set server timeout. A value in the range of 0 – 7200 seconds.
`AT+CIUPDATE`	???

See also:

- YouTube – ESP8266 Tutorial AT Commands

Assembling circuits

Since the ESP8266 is an actual electronic component, some physical assembly is required. This book will not attempt to cover non-ESP8266 electronics as that is a very big and broad subject in its own right. However, what we will do is describe some of the components that we have found extremely useful while building ESP8266 solutions.

USB to UART converters

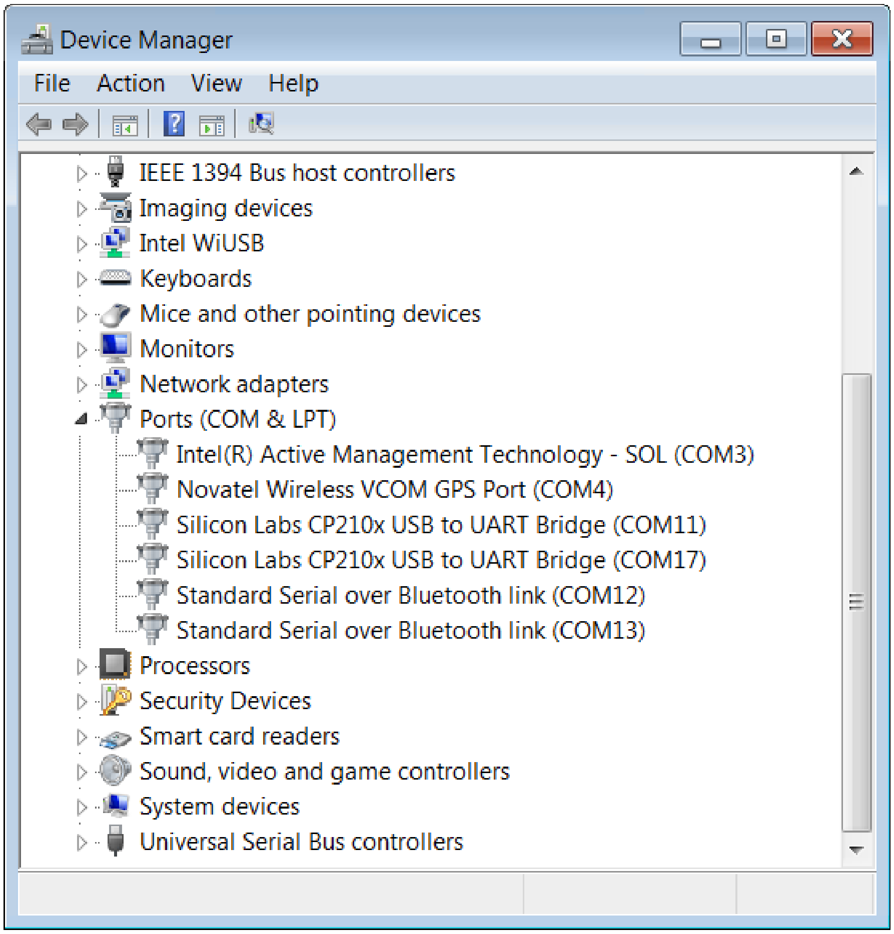
You can't program an ESP8266 without supplying it data through a UART. The easiest way to achieve this is through the use of a USB to UART converter. I use the devices that are based upon the CP2102 STC which can be found cheaply on ebay for under $2 each. Another popular brand are the devices from Future Technology Devices International (FTDI). You will want at least two. One for programming and one for debugging. I suggest buying more than two just in case …

When ordering, don't forget to get some male-female USB extender cables as it is unlikely you will be able to attach your USB devices to both a breadboard and the PC at the same time via direct connection and although connector cables will work, plugging into the breadboard is just so much easier. USB connector cables allow you to easily connect from the PC to the USB socket to the UART USB plug. Here is an image of the type of connector cable I recommend. Get them with as short a cable length as possible. 12-24 inches should be preferred.

When we plug in a USB → UART into a Windows machine, we can learn the COM port that the new serial port appears upon by opening the Windows Device Manager. There are a number of ways of doing this, one way is to launch it from the DOS command window with:

```
mmc devmgmt.msc
```

Under the section called Ports (COM & LPT) you will find entries for each of the COM ports. The COM ports don't provide you a mapping that a particular USB socket is hosting a particular COM port so my poor suggestion is to pull the USB from each socket one by one and make a note of which COM port disappears (or appears if you are inserting a USB).

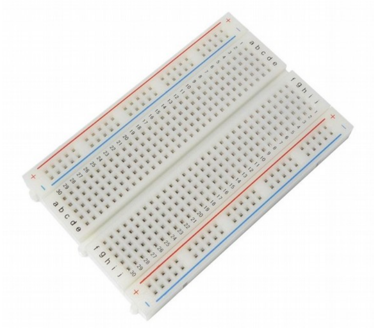


See also:

- Connecting to the ESP8266
- Working with serial

Breadboards

I find I can never have too many breadboards. I suggest getting a few full size and half size boards along with some 24 AWG connector wire and a good pair of wire strippers. Keep a trash bin close by otherwise you will find yourself knee deep in stripped insulation and cut wire parts before you know it. I also recommend some Dupont male-male pre-made wires. Ribbon cable can also be useful.



Power

We need electricity to get these devices working. I choose the MB102 breadboard attachable power adapters. These can be powered from an ordinary wall-wart (mains adapter) or from USB. The devices have a master on/off power switch plus a jumper to set 3.3V or 5V outputs. You can even have one breadboard rail be 3.3V and the other 5V … but take care not to apply 5V to your ESP8266. By having two power rails, one at 3.3V and the other at 5V, you can power both the ESP8266 and devices/circuits that require 5V.

When the ESP8266 starts to transmit over wireless, that can draw a lot of current which can cause ripples in your power supply. You may also have other sensors or devices connected to your supply as well. These fluctuations in the voltage can cause problems. It is strongly recommended that you place a 10 micro farad capacitor between +ve and -ve as close to your ESP8266 as you can. This will provide a reservoir of power to even out any transient ripples. This is one of those tips that you ignore at your peril. Everything may work just fine without the capacitor … until it doesn't or until you start getting intermittent problems and are at a loss to explain them. Let me put it this way, for the few cents it costs and the zero harm it does, why not?

Multi-meter / Logic probe / Logic Analyzer

When your circuit doesn't work and you are staring at it wondering what is wrong, you will be thankful if you have a multi-meter and a logic probe. If your budget will stretch, I also recommend a USB based logic analyzer such as those made by Saleae. These allow you to monitor the signals coming into or being produced by your ESP8266. Think of this as the best source of debugging available to you.

See also:

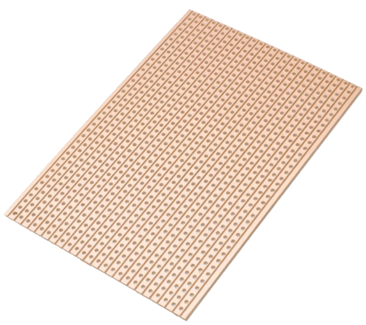
- Saleae logic analyzers

Sundry components

You will want the usual set of suspects for sundry components including LEDs, resistors, capacitors and more.

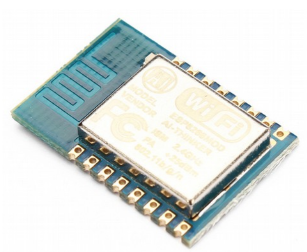
Physical construction

When you have breadboarded your circuit and written your application, there may come a time where you wish to make your solution permanent. At that point, you will need a soldering iron, solder and some strip-board. I also recommend some female header sockets so that you don't have to solder your ESP8266s directly into the circuits. Not only does this allow you to reuse the devices (should you desire) but in the unfortunate event that you fry one, it will be easier to replace.

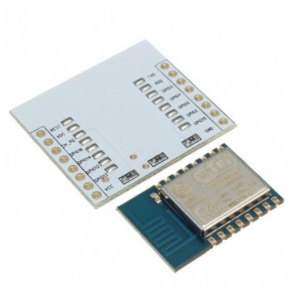


Recommended setup for programming ESP8266

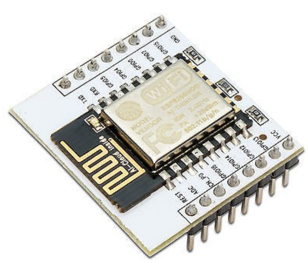
Obviously in order to program an ESP8266, you will actually need to obtain an ESP8266 but it isn't that easy. The actual ESP8266 itself is a tiny integrated circuit and you are unlikely to be able to use it directly. Instead, you will buy one of the many styles of breakout boards that already exist. The common ones are the ESP-1 which exposes 2 GPIO pins and the ESP-12 which exposes 9. I recommend the ESP-12 as it is only marginally more expensive for the extra pins exposed.



You will also need a mounting board as the ESP-12 by itself doesn't have connector pins. You can commonly buy both the ESP-12 and the mounting board together at the same time. However, check carefully, the mounting boards can be bought separately and you need to validate that when you order and assume you are getting both that you are not *just* buying the mounting boards without the ESP8266. You will be disappointed.



The ESP-12 is then soldered onto the mounting board so you will need a soldering iron and some fine grained hand control. The soldering is not the easiest in the world as the pins are extremely close together. For this reason and for others, I'd suggest buying multiple ESP-12s and mounting boards instead of just one. It is also not difficult to fry your ESP-12 if you get some wiring wrong. Once assembled, it should look as follows:



Mine never look this "clean" when build as my solder resin seems to discolor the original attractive white base of the mounting board. However, looks aren't important.

Assuming you now have a mounted ESP-12 with pins, your next question will be "now what"? This is where you will want a few breadboards and connector wire. You could use dupont connectors with female sockets attached to the ESP-12 and male pins on the other to attach to your breadboard but you will find that wires inevitably come loose at the worse possible times. You can mount the ESP-12 to a breadboard but I tend to find that there is not enough space for connector wires underneath it.

Once secured, I recommend **two** USB → UART connectors. Why two? One dedicated for flashing the device and one for debugging.

<insert diagrams here>

For power, I recommend using MB102 breadboard power supplies however, make sure that you set the jumper cables to be 3.3V. You will ruin your ESP8266 if you try and power it at 5V.

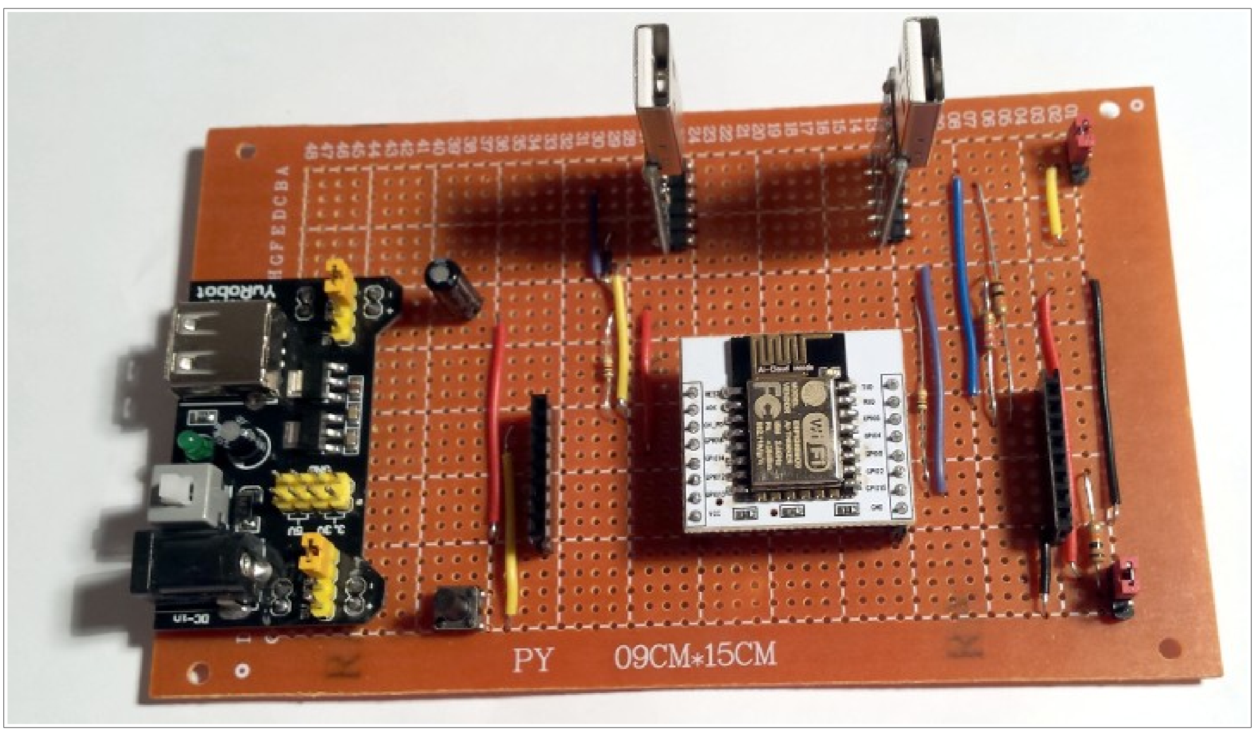
Once it is all wired up, you will need a PC with two open USB ports.

Parts list

- Breadboards – 2 half size – $3.50 for 2
- ESP-12 plus mounting boards – 3 sets – $3.80 each – $11.40
- CP2102 USB → UARTs – 2 pieces - $3.10
- USB male to female extenders – 2 pieces – $1.00 each – $2.00
- 24 AWG wire – 5 meters for $1.12
- 8pin 2.54mm stackable long legged female headers – 10 pieces for $3.95
- Red diffuse LEDs – A handful – $1.00
- Resistors – Some 10K, some 20K, some 330Ohm – A handful – $1.00
- Capacitors – Some 10 micro farad – $1.00

All told, it comes to about $30 + some shipping. I buy all my components through ebay from Chinese suppliers that give me the price/quality I am looking for. The name of the game though is patience. Once you order it usually takes 2-3 weeks for the parts to arrive so be patient and use the time to watch you-tube videos on electronics projects and the relevant community forums.

Eventually, you are likely going to want to build a permanent circuit for your development. On a strip board the circuit I built looks liked:



Configuration for flashing the device

Later on in the book you will find that when it comes time to flash the device with your new applications, you will have to set some of the GPIO pins to be low and then reboot. This is the signal that it is now ready to be flashed. Obviously, you can build a circuit that you use for flashing your firmware and then place the device in its final circuit but you will find that during development, you will want to flash and test pretty frequently. This means that you will want to use jumper wires and to allow you to move the links of pins on your breadboards from their "flash" position to their "normal use" position.

Programming

The ESP8266 allows you to write applications that can run natively on the device. You can compile C language code and deploy it to the device through a process known as flashing. In order for your applications to do something useful, they have to be able to interact with the environment. This could be making network connections or sending/receiving data from attached sensors, inputs and outputs. In order to make that happen, the ESP8266 contains a core set of functions that we can loosely think of as the operating system of the device. The services of the operating system are exposed to be called from your application providing a contract of services that you can leverage. These services are fully documented. In order to successfully write applications for deployment, you need to be aware of the existence of these services. They become indispensable tools in your tool chest. For example, if you need to connect to a WiFi access point, there is an API for that. To get your current IP address, there is an API for that and to get the time since the device was started, there is an API for that. In fact, there are a LOT of APIs available for us to use. The good news is that no-one is expecting us to memorize all the details of their use. Rather it is sufficient to broadly know that they exist and have somewhere to go when you want to look up the details of how to use them.

To sensibly manage the number and variety of these exposed APIs, we can collect sets of them together in meaningful groups of related functions. This gives us yet another and better way to manage our knowledge and learning of them.

The primary source of knowledge on programming the ESP8266 is the ESP8266 SDK API Guide. Direct links to all the relevant documents can be found at Reference documents.

See also:

- Espressif Systems – Manufacturers of the ESP8266
- Espressif Bulletin Board System – Place for SDKs, docs and forums

Boot mode

When the ESP8266 boots, the values of the pins known as MTDO, GPIO0 and GPIO2 are examined. The combination of the high or low values of these pins provide a 3 bit number with a total of 8 possible values from 000 to 111. Each value has a possible meaning interpreted by the device when it boots.

Value [15-0-2]	Value	Meaning
000	0	Remapping … details unknown.
001	1	Boot from the data received from UART0. Also includes flashing the flash memory for subsequent normal starts.
010	2	Jump start
011	3	Boot from flash
100	4	SDIO low speed V2
101	5	SDIO high speed V1
110	6	SDIO low speed V1
111	7	SDIO high speed V2

From a practical perspective, what this means is that if we wish the device to run normally, we want to boot from flash with the pins having values 011 while when we wish to flash the device with a new program, we want to supply 001 to boot from UART0.

Note that MTDO is also known as GPIO15.

The ESP8266 SDK

Include directories

The C programming language uses a text based pre-processor to include test in the compilation units. This is commonly called CPP. CPP has the ability to include addition C source files that, by convention, are called header files and end with the ".h" prefix. Within these files we commonly find definitions of data types and function prototypes that are used during compilation. The ESP8266 SDK provides a directory called "include" which contains the include files supplied by Espressif for use with the ESP8266. The list of header files that we may use are as described in the following table:

File	Notes
at_custom.h	Definitions for custom extensions to the AT command handler.
c_types.h	C language definitions.
eagle_soc.h	Low level definitions and macros. Heavily related to bit twiddling at the CPU level. No idea why the file is called "eagle".
espconn.h	TCP and UDP definitions.
espnow.h	Functions related to the esp now support.
ets_sys.h	Unknown.
gpio.h	Definitions for GPIO interactions.
ip_addr.h	IP address definitions and macros.

mem.h	Definitions for memory manipulation and access.
os_type.h	OS type definitions.
osapi.h	Includes a **user** supplied header called "user_config.h".
ping.h	Definitions for the ping capability.
pwm.h	Definitions for PWM.
queue.h	Queue and list definitions.
smartconfig.h	Definitions for smart config.
sntp.h	Definitions for SNTP.
spi_flash.h	Definitions for flash.
upgrade.h	Definitions for upgrades.
user_interface.h	Definitions for OS and WiFi. I have no explanation for why this file is named "user_interface" as there is obviously no UI involved with ESP8266s.

Compiling

Application code for an ESP8266 can be written in C. Before we can deploy an application, we must compile the code into binary machine code instructions. To do this, we need a set of development tools. My personal preference is the package for Eclipse which has everything pre-built and ready for use. However, these tools can also be downloaded from the Internet as open source projects on a piece by piece basis.

The macro LOCAL is a synonym for the C language keyword "`static`".

From reading the docs, no published example of how to compile was found. However, when one uses the Eclipse open source project, one can see the Makefiles that are used and this exposes examples of compilation.

A typical compilation looks like:

```
17:57:16 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f
C:/Users/IBM_ADMIN/Documents/RaspberryPi/ESP8266/EclipseDevKit/WorkSpace/k_blinky/Make
file all
CC user/user_main.c
AR build/app_app.a
LD build/app.out
------------------------------------------------------------------------------
Section info:

build/app.out:     file format elf32-xtensa-le

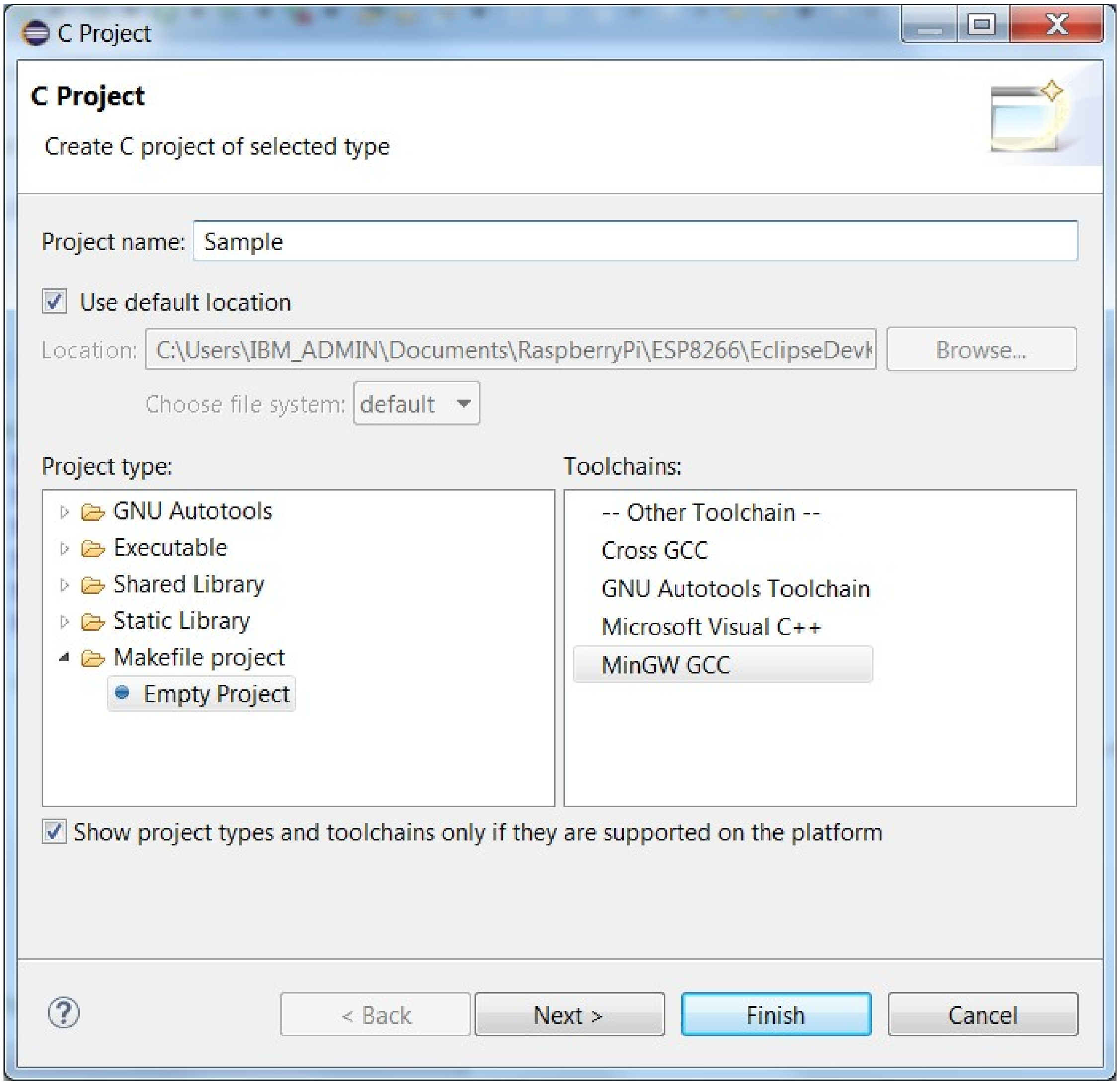
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000053c  3ffe8000  3ffe8000  000000e0  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  1 .rodata       00000878  3ffe8540  3ffe8540  00000620  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
  2 .bss          00009130  3ffe8db8  3ffe8db8  00000e98  2**4
                  ALLOC
  3 .text         00006f22  40100000  40100000  00000e98  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .irom0.text   00028058  40240000  40240000  00007dc0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
------------------------------------------------------------------------------
Section info:
   Section|                   Description| Start (hex)|   End (hex)|Used space
------------------------------------------------------------------------------
      data|        Initialized Data (RAM)|    3FFE8000|    3FFE853C|    1340
    rodata|           ReadOnly Data (RAM)|    3FFE8540|    3FFE8DB8|    2168
       bss|      Uninitialized Data (RAM)|    3FFE8DB8|    3FFF1EE8|   37168
      text|            Cached Code (IRAM)|    40100000|    40106F22|   28450
irom0_text|           Uncached Code (SPI)|    40240000|    40268058|  163928
Total Used RAM : 40676
Free RAM : 41244
Free IRam : 4336
------------------------------------------------------------------------------
Run objcopy, please wait...
objcopy done
Run gen_appbin.exe
No boot needed.
Generate eagle.flash.bin and eagle.irom0text.bin successully in folder firmware.
eagle.flash.bin-------->0x00000
eagle.irom0text.bin---->0x40000
Done

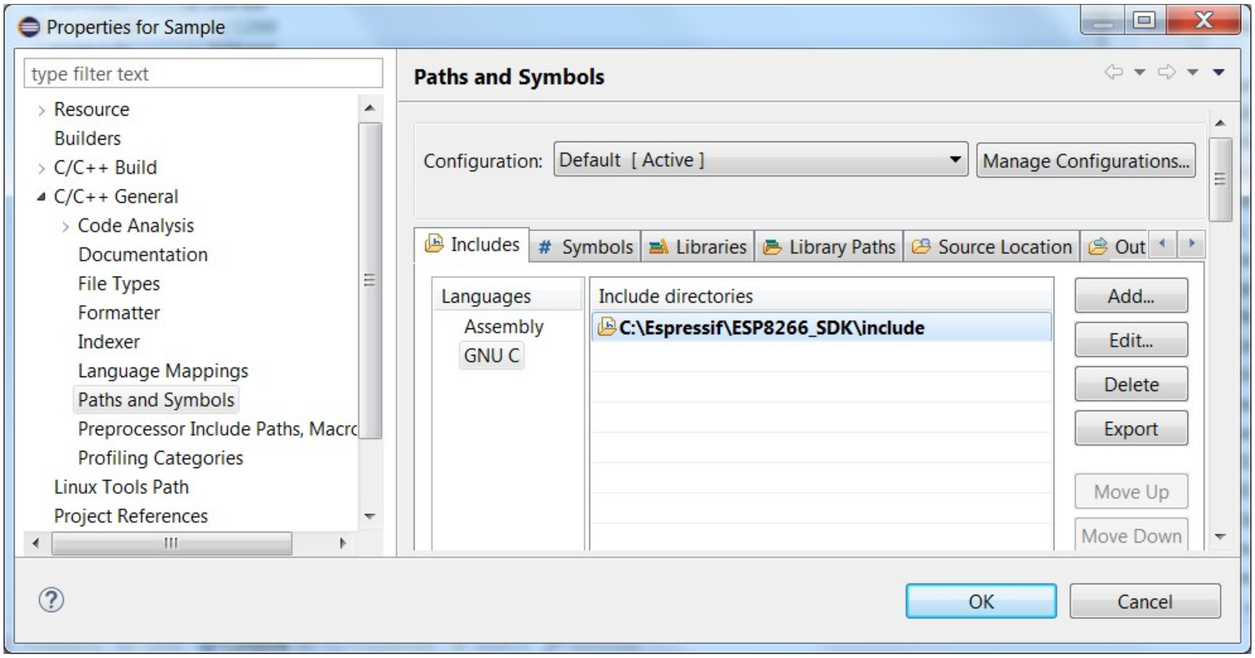
17:57:19 Build Finished (took 3s.141ms)
```

We can build solutions using the pre-supplied Makefiles but, personally, I don't like mystery so here is a recipe for building a solution from scratch.

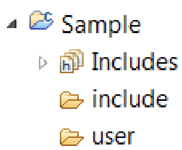
1. Create a new project from File > New > C Project
2. Select a Makefile project



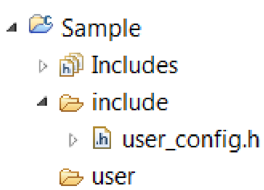
3. Add the ESP8266 include directory



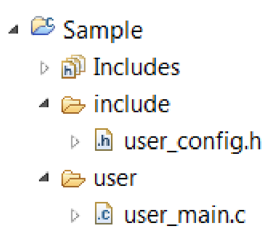
4. Create the folders called "user" and "include"



5. Create the file called "user_config.h" in include



6. Create the C file called "user_main.c" in user



7. Create a Makefile

```
# Base directory for the compiler
XTENSA_TOOLS_ROOT ?= c:/Espressif/xtensa-lx106-elf/bin
SDK_BASE       ?= c:/Espressif/ESP8266_SDK
SDK_TOOLS      ?= c:/Espressif/utils
ESPPORT        = COM18
#ESPBAUD       = 115200
```

```
ESPBAUD             = 230400

# select which tools to use as compiler, librarian and linker
CC           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
AR           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-ar
LD           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
OBJCOPY := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-objcopy
OBJDUMP := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-objdump
ESPTOOL      ?= $(SDK_TOOLS)/esptool.exe

# compiler flags using during compilation of source files
TARGET       = myApp
CFLAGS       = -Os -g -O2 -std=gnu90 -Wpointer-arith -Wundef -Werror -Wl,-EL -fno-
inline-functions -nostdlib -mlongcalls -mtext-section-literals -mno-serialize-volatile
-D__ets__ -DICACHE_FLASH
MODULES      = user
BUILD_BASE   = build
FW_BASE      = firmware
SDK_LIBDIR   = lib
SDK_LDDIR    = ld

#
# Nothing to configure south of here.
#
# linker flags used to generate the main object file
LDFLAGS      = -nostdlib -Wl,--no-check-sections -u call_user_start -Wl,-static
# libraries used in this project, mainly provided by the SDK
LIBS         = c gcc hal phy pp net80211 lwip wpa main


# linker script used for the above linkier step
LD_SCRIPT    = eagle.app.v6.ld
flashimageoptions = --flash_freq 40m --flash_mode qio --flash_size 4m
SDK_LIBDIR   := $(addprefix $(SDK_BASE)/, $(SDK_LIBDIR))
LD_SCRIPT    := $(addprefix -T$(SDK_BASE)/$(SDK_LDDIR)/, $(LD_SCRIPT))
LIBS         := $(addprefix -l, $(LIBS))
APP_AR       := $(addprefix $(BUILD_BASE)/, $(TARGET)_app.a)
TARGET_OUT   := $(addprefix $(BUILD_BASE)/, $(TARGET).out)
BUILD_DIRS   = $(addprefix $(BUILD_BASE)/, $(MODULES)) $(FW_BASE)
SRC                 = $(foreach moduleDir, $(MODULES), $(wildcard $(moduleDir)/*.c))
# Replace all x.c with x.o
OBJS         = $(patsubst %.c, $(BUILD_BASE)/%.o, $(SRC))

all: checkdirs $(TARGET_OUT)
	echo "Image file built!"

# Build the application archive.
# This is dependent on the compiled objects.
$(APP_AR): $(OBJS)
	$(AR) -cru $(APP_AR) $(OBJS)

# Build the objects from the C source files
$(BUILD_BASE)/%.o : %.c
	$(CC) $(CFLAGS) -I$(SDK_BASE)/include -Iinclude -c $< -o $@
```

```
# Check that the required directories are present
checkdirs: $(BUILD_DIRS)

# Create the directory structure which holds the builds (compiles)
$(BUILD_DIRS):
	mkdir --parents --verbose $@


$(TARGET_OUT): $(APP_AR)
	$(LD) -L$(SDK_LIBDIR) $(LD_SCRIPT) $(LDFLAGS) -Wl,--start-group $(LIBS) $
(APP_AR) -Wl,--end-group -o $@
	$(OBJDUMP) --headers --section=.data \
		--section=.rodata \
		--section=.bss \
		--section=.text \
		--section=.irom0.text $@
	$(OBJCOPY) --only-section .text --output-target binary $@ eagle.app.v6.text.bin
	$(OBJCOPY) --only-section .data --output-target binary $@ eagle.app.v6.data.bin
	$(OBJCOPY) --only-section .rodata --output-target binary $@
eagle.app.v6.rodata.bin
	$(OBJCOPY) --only-section .irom0.text --output-target binary $@
eagle.app.v6.irom0text.bin
	$(SDK_TOOLS)/gen_appbin.exe $@ 0 0 0 0
	mv eagle.app.flash.bin $(FW_BASE)/eagle.flash.bin
	mv eagle.app.v6.irom0text.bin $(FW_BASE)/eagle.irom0text.bin
	rm eagle.app.v6.*

#
# Flash the ESP8266
#
flash: all
	$(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) write_flash $(flashimageoptions)
0x00000 $(FW_BASE)/eagle.flash.bin 0x40000 $(FW_BASE)/eagle.irom0text.bin

#
# Clean any previous builds
#
clean:
# Remove forceably and recursively
	rm --recursive --force --verbose $(BUILD_BASE) $(FW_BASE)

flashId:
	$(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) flash_id

readMac:
	$(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) read_mac

imageInfo:
	$(ESPTOOL) image_info $(FW_BASE)/eagle.flash.bin
```

8. Add Make targets for at least all and flash

See also:

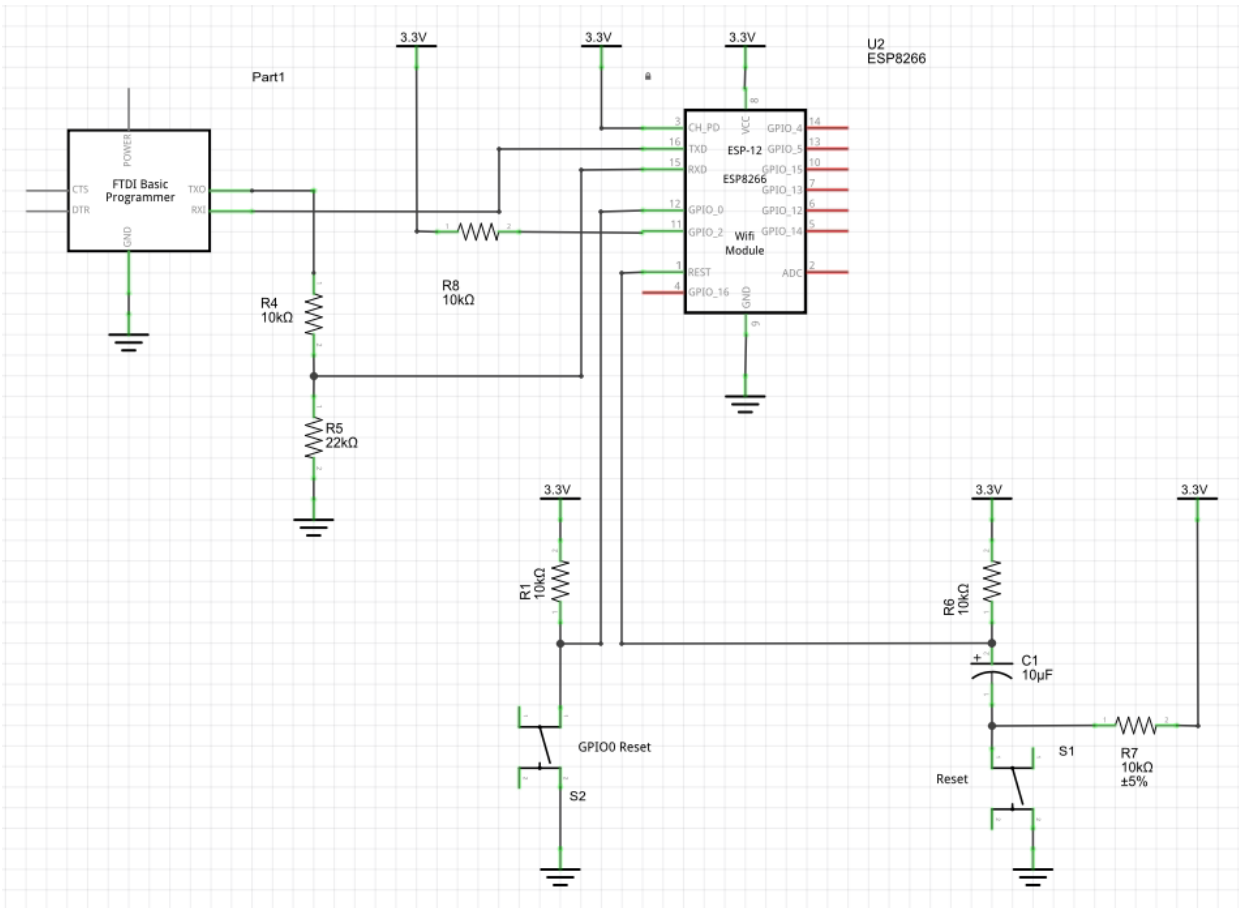
- Programming using Eclipse

Flashing the ESP8266

Once the program has been compiled, it needs to be loaded into the ESP8266. This task is called "flashing". In order to flash the ESP8266, it needs to be placed in a mode where it will accept the new incoming program to replace the old existing program. The way this is done is to reboot the ESP8266 either by removing and reapplying power or by bringing the REST pin low and then high again. However, just rebooting the device is not enough. During startup, the device examines the signal value found on GPIO0. If the signal is low, then this is the indication that a flash programming session is about to happen. If the signal on GPIO0 is high, it will enter its normal operation mode. Because of this, it is recommended not to let GPIO0 float. We don't want it to accidentally enter flashing mode when not desired. A pull-up resistor of 10k is perfect.

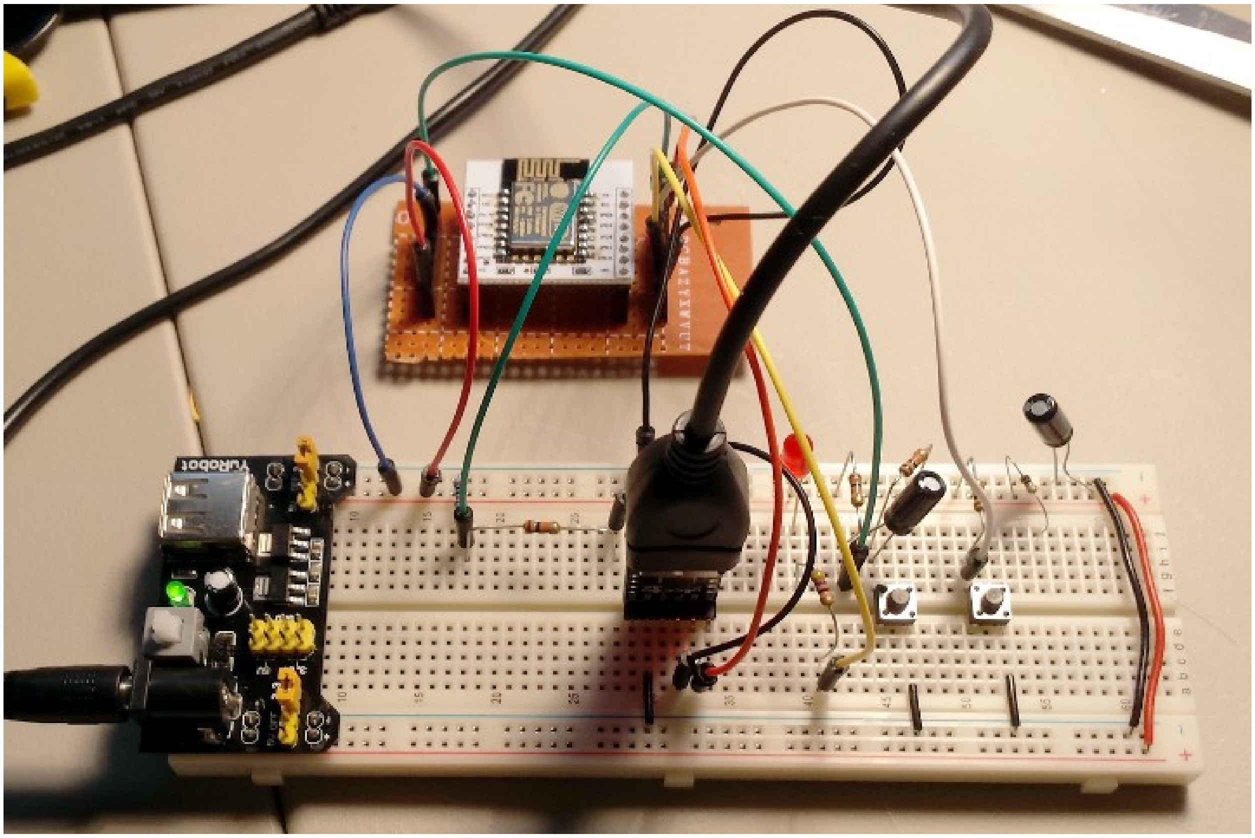
We can build a circuit which includes a couple of buttons. One for performing a reset and one for bringing GPIO0 low. Pressing the reset button by itself will reboot the device. This alone is already useful. However if we are holding the "GPIO0 low" button while we press reset, then we are placed in flash mode.

Here is an example schematic diagram illustrating an ESP-12 including the buttons:



Notice that there is a voltage divider from the output of the USB to UART converter TX pin. The thinking behind this is to handle the case where the output TX voltage is greater than the desired 3.3V wanted on the RX input of the ESP8266. Is this required? The belief is that it is **not** required if you are sure that the output TX voltage will be 3.3V. This appears to be the case for the CP2102 range of USB to UARTs however I am have no knowledge on other devices. What I can claim is that having a voltage divider that reduces 5V to 3.3V still results in a usable output level voltage to indicate a high signal when fed with a 3.3V actual output. I don't know how close I am coming to the minimum RX input voltage on the ESP8266 indicating a high.

When built out on a breadboard, it may look as follows:



This however suffers from the disadvantage that it requires us to manually press some buttons to load a new application. This is not a horrible situation but maybe we have alternatives?

When we are flashing our ESP8266s, we commonly connect them to USB->UART converters. These devices are able to supply UART used to program the ESP8266. We are familiar with the pins labeled RX and TX but what about the pins labeled RTS and DTR … what might those do for us?

RTS which is "Ready to Send" is an output from the UART to inform the downstream device that it may now send data. This is commonly connected to the partner input CTS which is "Clear to Send" which indicates that it is now acceptable to send data. Both RTS and CTS are active low.

DTR which is "Data Terminal Ready" is used in flow control.

When flashing the device using the Eclipse tools and recipes the following are the flash commands that are run (as an example) and the messages logged:

```
22:34:17 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f C:/Users/User1/WorkSpace/k_blinky/Makefile flash
c:/Espressif/utils/esptool.exe -p COM11 -b 115200 write_flash -ff 40m -fm qio -fs 4m
0x00000 firmware/eagle.flash.bin 0x40000 firmware/eagle.irom0text.bin
Connecting...
Erasing flash...
head: 8 ;total: 8
erase size : 16384
```

```
Writing at 0x00000000... (3 %)
Writing at 0x00000400... (6 %)
…
Writing at 0x00007000... (96 %)
Writing at 0x00007400... (100 %)
Written 30720 bytes in 3.01 seconds (81.62 kbit/s)...
Erasing flash...
head:  16 ;total: 41
erase size :  102400

Writing at 0x00040000... (0 %)
Writing at 0x00040400... (1 %)
…
Writing at 0x00067c00... (99 %)
Writing at 0x00068000... (100 %)
Written 164864 bytes in 16.18 seconds (81.53 kbit/s)...

Leaving...

22:34:40 Build Finished (took 23s.424ms)
```

As an example of what the messages look like if we fail to put the ESP8266 into flash mode, we have the following:

```
13:47:09 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f C:/Users/User1/WorkSpace/k_blinky/Makefile flash
c:/Espressif/utils/esptool.exe -p COM11 -b 115200 write_flash -ff 40m -fm qio -fs 4m
0x00000 firmware/eagle.flash.bin 0x40000 firmware/eagle.irom0text.bin
Connecting…
Traceback (most recent call last):
  File "esptool.py", line 558, in <module>
  File "esptool.py", line 160, in connect
Exception: Failed to connect
C:/Users/User1/WorkSpace/k_blinky/Makefile:313: recipe for target 'flash' failed
mingw32-make.exe: *** [flash] Error 255

13:47:14 Build Finished (took 5s.329ms)
```

The tool called `esptool.py` provides an excellent environment for flashing the device but it can also be used for "reading" what is currently on it. This can be used for making backups of the applications contained within before re-flashing them with a new program. This way, you can always return to what you had before over-writing. For example, on Unix:

```
esptool.py --port /dev/ttyUSB0 read_flash 0x00000 0xFFFF backup-0x00000.bin
esptool.py --port /dev/ttyUSB0 read_flash 0x10000 0x3FFFF backup-0x10000.bin
```

See also:

- USB to UART converters
- Recommended setup for programming ESP8266
- esptool.py
- [What is a UART?]

Programming environments

We can program the ESP8266 using the Espressif supplied SDK on Windows using Eclipse. A separate chapter on setting that up is supplied. We also have the ability to program the ESP8266 using the Arduino IDE. This is potentially a game changing story and it too been given its own important chapter.

See also:

- Programming using Eclipse
- Programming using the Arduino IDE

Compilation tools

There are a number of tools that are essential when building C based ESP8266 applications.

make

Make is a compilation engine used to track what has to be compiled in order to build your target application. Make is driven by a Makefile. Although powerful and simple enough for simple C projects, it can get complex pretty quickly. If you find yourself studying Makefiles written by others, grab the excellent GNU make documentation and study it deeply.

gcc

The open source GNU Compiler Collection includes compilers for C and C++. If we look carefully at the flags that are supplied for compiling and linking code for the ESP8266 we find the following:

Compiling

- `-Os` – Optimize code generation for size.
- `-O2` – Optimize even more.
- `-ggdb` – Generate debug code that can be used by GDB.
- `-std=gnu90` – Dialect of C supported.
- `-Werror` – Make all warnings errors.
- `-Wno-address` – Do not warn about suspicious use of memory addresses.
- `-Wpointer-arith` – Warn when pointer arithmetic is attempted that depends on `sizeof`.
- `-Wundef` – Warn when an identifier is found in a #if directive that is not a macro.
- `-fno-inline-functions` – Do not allow functions to be replaced with in-line code.
- `-mlongcalls` – Translate direct assembly language calls into indirect calls.
- `-mtext-section-literals` – Allow literals to be intermixed with the text section.

- `-mno-serialize-volatile` – Special instructions for volatile definitions.

Linking:

- `-nostdlib` – Don't use standard C or C++ system startup libraries

See also:

- GCC – The GNU Compiler Collection

ar

The archive tool is used to packaged together compiled object files into libraries. These libraries end with ".a" (archive). A library can be named when using a linker and the objects contained within will be used to resolve externals.

Some of the most common flags used with this tool include:

- `-c` – Create a library
- `-r` – Replace existing members in the library
- `-u` – Update existing members in the library

The syntax of the command is:

```
ar -cru libraryName member.o member.o ….
```

See also:

- GNU – ar

objcopy

See also:

- GNU – objcopy

objdump

The command is `xtensa-lx106-elf-objdump` located in

`C:\Espressif\xtensa-lx106-elf\bin`.

- Wikipedia – objdump
- GNU – objdump
- man page – objdump(1)

esptool.py

This tool is an open source implementation used to flash the ESP8266 through a serial port. It is written in Python. Versions have been seen to be available as windows executables that appear to have been generated "`.EXE`" files from the Python code suitable for running on Windows without a supporting Python runtime installation.

- `-p port` | `--port port` – The serial port to use
- `-b baud` | `--baud baud` – The baud rate to use for serial
- `-h` – Help
- `{command} -h` – Help for that command
- `load_ram {filename}` – Download an image to RAM and execute
- `dump_mem {address} {size} {filename}` – Dump arbitrary memory to disk
- `read_mem {address}` – Read arbitrary memory location
- `write_mem {address} {value} {mask}` – Read-modify-write to arbitrary memory location
- `write_flash` – Write a binary blob to flash
    - `--flash_freq {40m,26m,20m,80m} | -ff {40m,26m,20m,80m}` – SPI Flash frequency
    - `--flash_mode {qio,qout,dio,dout} | -fm {qio,qout,dio,dout}` – SPI Flash mode
    - `--flash_size {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2} | -fs {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2}` – SPI Flash size in Mbit
    - `{address} {fileName}` – Address to write, file to write … repeatable
- `run` – Run application code in flash
- `image_info {image file}` – Dump headers from an application image. Here is an example output:

Entry point: 40100004

3 segments

Segment 1: 25356 bytes at 40100000

Segment 2: 1344 bytes at 3ffe8000

Segment 3: 924 bytes at 3ffe8540

Checksum: 40 (valid)

- `make_image` – Create an application image from binary files

- `--segfile SEGFILE, -f SEGFILE` – Segment input file
- `--segaddr SEGADDR, -a SEGADDR` – Segment base address
- `--entrypoint ENTRYPOINT, -e ENTRYPOINT` – Address of entry point
- `output`

- `elf2image` – Create an application image from ELF file
  - `--output OUTPUT, -o OUTPUT` – Output filename prefix
  - `--flash_freq {40m,26m,20m,80m}, -ff {40m,26m,20m,80m}` – SPI Flash frequency
  - `--flash_mode {qio,qout,dio,dout}, -fm {qio,qout,dio,dout}` – SPI Flash mode
  - `--flash_size {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2}, -fs {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2}` – SPI Flash size in Mbit
  - `--entry-symbol ENTRY_SYMBOL, -es ENTRY_SYMBOL` – Entry point symbol name (default 'call_user_start')
- `read_mac` – Read MAC address from OTP ROM. Here is an example output:

MAC AP: 1A-FE-34-F9-43-22

MAC STA: 18-FE-34-F9-43-22

- `flash_id` – Read SPI flash manufacturer and device ID. Here is an example output:

head: 0 ;total: 0

erase size : 0

Manufacturer: c8

Device: 4014

- `read_flash` – Read SPI flash content
  - `address` – Start address
  - `size` – Size of region to dump
  - `filename` – Name of binary dump
- `erase_flash` – Perform Chip Erase on SPI flash

See also:

- Flashing the ESP8266
- Github: themadinventor/esptool

esptool-ck

Another tool that is also called esptool. The naming of these tools being so similar is starting to become uncomfortable.

- -eo <filename>
- -es <section> <filename>
- -ec
- -bo <filename>
- -bm <qio|qout|dio|dout>
- -bz <512K|256K|1M|2M|4M|8M|16M|32M>
- -bf <40|26|20|80>
- -bs <section>
- -bc
- -v
- -q
- -cp <device> - Serial device (eg. COM1)
- -cd <board>
- -cb <baudrate>
- -ca <address>
- -cf <filename>

See also:

- Github: https://github.com/igrr/esptool-ck

gen_appbin.py

The syntax of this tool is:

```
gen_appbin.py app.out boot_mode flash_mode flash_clk_div flash_size
```

- flash_mode
  - 0 – QIO
  - 1 – QOUT
  - 2 – DIO
  - 3 – DOUT
- flash_clk_div

- 0 – 80m / 2
- 1 – 80m / 3
- 2 – 80m / 4
- 0xf – 80m / 1

- flash_size_map
  - 0 – 512 KB (256 KB + 256 KB)
  - 1 – 256 KB
  - 2 – 1024 KB (512 KB + 512 KB)
  - 3 – 2048 KB (512 KB + 512 KB)
  - 4 – 4096 KB (512 KB + 512 KB)
  - 5 – 2048 KB (1024 KB + 1024 KB)
  - 6 – 4096 KB (1024 KB + 1024 KB)

The following files are expected to exist:

- `eagle.app.v6.irom0text.bin`
- `eagle.app.v6.text.bin`
- `eagle.app.v6.data.bin`
- `eagle.app.v6.rodata.bin`

The output of this command is a new file called `eagle.app.flash.bin`.

xxd

This is a deceptively useful tool. What it does is dump binary data from a file in a formatted form. One powerful use of it is to take a binary file and produce a C language data structure that represents the file. This means that you can take binary data and include it in your applications. A copy of `xxd.exe` is distributed with the SDK supplied by Espressif in the `tools` folder.

The following will read the content of `inFile` as binary data and produce a header file in the `outFile`.

```
xxd -include <inFile> <outFile>
```

Debugging

When writing programs, we may find that they don't always run as expected. Performing debugging on an SOC can be difficult since we have no readily available source level debuggers.

Logging to UART1

We can insert diagnostic statements using `os_printf()`. This causes the text and data associated with these functions to be written to UART1. If we attach a USB → UART device in the circuit, we can then look at the data logged. In my development environment I always have two USB → UART devices in play. One to flash new applications and one to use for diagnostic output.

The OS is also able to write debugging information. By default this is on but can be switched off with a call to `system_set_os_print()`.

See also:

- USB to UART converters
- Working with serial
- system_set_os_print

Run a Blinky

Physically looking at an ESP8266 there isn't much to see that tells you all is working well within it. There is a power light and a network transmission active light … but that's about it. A technique that I recommend is to always have your device perform execute a "blinking led" which is commonly known as a "Blinky". This can be achieved by connecting a GPIO pin to a current limiting resistor and then to an LED. When the GPIO signal goes high, the LED lights. When the GPIO signal goes low, the LED becomes dark. If we define a timer callback that is called (for example) once a second and toggles the GPIO pin signal value each invocation, we will have a simple blinking LED. You will be surprised how good a feeling it will give simply knowing that *something* is alive within the device each time you see it blink.

The cost of running the timer and changing the I/O value to achieve a blinking should not be a problem during development time so I wouldn't worry about side effects of doing this. Obviously for a published application, you may not desire this and can simply remove it.

However, although this is a trivial circuit, it has a lot of uses during development. First, you will always know that the device is operating. If the LED is blinking, you know the device has power and logic processing control. If the light stops blinking, you will know that something has locked up or you have entered an infinite loop.

Another useful purpose for including the Blinky is to validate that you have entered flash mode when programming the device. If we understand that the device can boot up in normal or flash mode and we boot it up in flash mode, then the Blinky will stop executing. This can be useful if you are using buttons or jumpers to toggle the boot mode as it will provide evidence that you are *not* in normal mode. On occasion I have mis-pressed some control buttons and was quickly able to realize that something was wrong before even attempting to flash it as the Blinky was still going.

Here is some simple code for setting up a Blinky. In this example we use GPIO4 as the LED driver. First, the code we place in `user_init`:

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO4_U, FUNC_GPIO4);
os_timer_disarm(&blink_timer);
os_timer_setfn(&blink_timer, (os_timer_func_t *)blink_cb, (void *)0);
os_timer_arm(&blink_timer, 1000, 1);
```

This assumes a global called `blink_timer` defined as:

```
LOCAL os_timer_t blink_timer;
```

The callback function in this example is called `blink_cb` and looks like:

```
LOCAL void ICACHE_FLASH_ATTR blink_cb(void *arg)
{
	led_state = !led_state;
	GPIO_OUTPUT_SET(4, led_state);
}
```

The global variable called `led_state` contains the current state of the LED (1=on, 0=off):

```
LOCAL uint8_t led_state=0;
```

Dumping IP Addresses

Being a WiFi and TCP/IP device, you would imagine that the ESP8266 works a lot with IP addresses and you would be right. We can generate a string representation of an IP address using:

```
os_printf(IPSTR, IP2STR(pIpAddrVar))
```

the IPSTR macro is "`%d.%d.%d.%d`" so the above is equivalent to:

```
os_printf("%d.%d.%d.%d", IP2STR(pIpAddrVar))
```

which may be more useful in certain situations.

See also:

- ipaddr_t

Exception handling

At runtime, things may not always work as expected and an exception can be thrown. For example, you might attempt to access storage at an invalid location or write to read only memory or perform a divide by zero.

When such an occurrence happens, the device will reboot itself but not before writing some diagnostics to UART1. Diagnostics may look like:

```
Fatal exception (28):
epc1=0x40243182, epc2=0x00000000, epc3=0x00000000, excvaddr=0x00000050,
depc=0x00000000
```

The codes are as follows:

- `epc1` – Exception program counter

- `excvaddr` – Virtual address that caused the most recent fetch, load or store exception. For example, if a write to memory occurs and that memory is not RAM an exception will be thrown and the value here will be the address that was attempted to be written.

The primary exception codes are:

Code	Cause name
0	IllegalInstructionCause
1	SyscallCause
2	InstructionFetchErrorCause
3	LoadStoreErrorCause
4	Level1InterruptCause
5	AllocaCause
6	IntegerDivideByZeroCause
7	Reserved
8	PrivilegedCause
9	LoadStoreAlignmentCause
10	Reserved
11	Reserved
12	InstrPIFDataErrorCause
13	LoadStorePIFDataErrorCause
14	InstrPIFAddrErrorCause
15	LoadStorePIFAddrErrorCause
16	InstTLBMissCause
17	InstTLBMultiHitCause
18	InstFetchPrivilegeCause
19	Reserved
20	InstFetchProhibitedCause
21	Reserved
22	Reserved
23	Reserved
24	LoadStoreTLBMissCause
25	LoadStoreTLBMultiHitCause
26	LoadStorePrivilegeCause
27	Reserved
28	LoadProhibitedCause
29	StoreProhibitedCause
30	Reserved
31	Reserved

32-39	CoprocessornDisabled
40-63	Reserved

If we know the location of the exception, we can analyze the executable (`app.out`) to figure out what piece of code caused the problem. For example:

```
xtensa-lx106-elf-objdump -x app.out -d
```

See also:

- system_get_rst_info
- struct rst_info

Debugging and testing TCP and UDP connections

When working with TCP/IP, you will likely want to have some applications that you can use to send and receive data so that you can be sure the ESP8266 is working. There are a number of excellent tools and utilities available and these vary by platform and function.

Android – Socket Protocol

The Socket Protocol is a free Android app available from the Google play app store. See:

https://play.google.com/store/apps/details?id=aprisco.app.android

Android – UDP Sender/Receiver

The UDP Sender/Receiver is another free Android app available from the Google play app store. What makes this one interesting is its ability to be a UDP (as opposed to TCP) sender and receiver. See:

https://play.google.com/store/apps/details?id=com.jca.udpsendreceive

Windows – Hercules

Hercules is an older app for Windows that still seems to work just fine on the latest releases. It looks a little old in the tooth now but still seems to get the job done just fine. See:

http://www.hw-group.com/products/hercules/index_en.html

Curl

Curl is powerful and comprehensive command line tool for performing any and all URL related commands. It can transmit HTTP requests of all different formats and receive their responses. It has a bewildering set of parameters available to it which is both a blessing and curse. You can be pretty sure that if it can be done, Curl can do it … however be prepared to wade through a lot of documentation.

See also:

- Curl

Architecture

To start thinking about writing applications for the ESP8266, we need to understand the high level architecture of the device.

Custom programs

Custom programs are applications that you can write and are the core focus of this book. These programs can be written in C or C++ and then compiled into the binary files. The programs are expected to have "well known" functions defined within that serve as architected entry points and callbacks.

Programmers write a C language file with a suggested name of "`user_main.c`". Contained within is a function with the signature:

```
void user_init(void)
```

This provides the initial entry into application code. It is called once during startup. While executing within this function, realize that not all of the environment is yet operational. If you need a fully functioning environment, register a callback function that will be invoked when the environment is 100% ready. This callback function can be registered with a call to `system_init_done_cb()`.

RF initialization must also be provided via:

```
void user_rf_pre_init(void)
```

When running in user code, we need to be sensitive that the primary purpose of the device is network communications. Since these are handled in the software, when user code gets control, that simply means that networking code doesn't. Since we only have one thread of control, we can't be in two places at once. The recommended duration to spend in user code at a single sitting is less than 10msecs.

See also:

- system_init_done_cb

WiFi at startup

The ESP8266 stores WiFi startup information in flash memory. This allows is to perform its functions at startup without having to do any special work. In my opinion, this is more trouble than it is worth. If I am going to write an ESP8266 application, I want to control when, how and to what it will connect or be an access point. Thankfully, there is a function called `wifi_station_set_auto_connect()` and its partner called `wifi_station_get_auto_connect()`. These allow us to override the auto connection functions when we are a station.

See also:

* wifi_station_get_auto_connect
* wifi_station_set_auto_connect

Working with WiFi

The ESP8266 can either be a station in the network, an access point for other devices or both. This is a fundamental consideration and we will want to choose how the device behaves early on in our design. Once we have chosen what we want, we set a global mode property which indicates which of the operational modes our device will perform (station, access point or station AND access point).

See also:

* wifi_set_opmode
* wifi_set_opmode_current

Scanning for access points

If the ESP8266 is performing the role of a station we will need to connect to an access point. We can request a list of the available access points against which we can attempt to connect. We do this using the `wifi_station_scan()` function. This function takes a callback function pointer as one of its parameters. This callback will be invoked when the scan has completed. The callback is necessary because it can take some time (a few seconds) for the scan to be performed and we can't afford to block operation until complete. The scan callback function receives a linked list of BSS structures. Contained within a BSS structure are:

* The SSID for the network
* The BSSID for the access point
* The channel
* The signal strength
* … others

For example:

```
LOCAL void scanCB(void *arg, STATUS status) {
	struct bss_info *bssInfo;
	bssInfo = (struct bss_info *)arg;
	// skip the first in the chain … it is invalid
	bssInfo = STAILQ_NEXT(bssInfo, next);
	while(bssInfo != NULL) {
		os_printf("ssid: %s\n", bssInfo->ssid);
		bssInfo = STAILQ_NEXT(bssInfo, next);
	}
}

//...
```

```
{
	// Ensure we are in station mode
	wifi_set_opmode_current(STATION);

	// Request a scan of the network calling "scanCB" on completion
	wifi_station_scan(NULL, scanCB);
}
```

Note the use of the `STAILQ_NEXT()` macro to navigate to the next entry in the list. The end of the list is indicated when this returns `NULL`.

See also:

- Sample – WiFi Scanner
- wifi_station_scan
- wifi_set_opmode
- struct bss_info
- STATUS

Defining the operating mode

The ESP8266 can execute as a WiFi Station, a WiFi access point or both a station and an access point. These are considered the three possible global operating modes. The operating mode that is used when the device boots is retained in flash memory but can be changed with a call to `wifi_set_opmode()`. This will change the current mode as well as record the mode to be used on next restart. To merely change the mode without changing the next boot mode, we can use `wifi_set_opmode_current()`. To retrieve the current mode, we can use `wifi_get_opmode()` and to retrieve the mode used on boot, we can use `wifi_get_opmode_default()`. Quite why we have the option to change the current mode without saving it in flash memory is a mystery. Presumably there is some occasion when such a feature was needed and thus exposed but what ever that reason may be is not obvious.

See also:

- wifi_get_opmode
- wifi_get_opmode_default
- wifi_set_opmode
- wifi_set_opmode_current

Handling WiFi events

During the course of operating as a WiFi device, certain events may occur that ESP8266 needs to know about. These may be of importance or interest to the applications running within it. Since we don't know when, or even if, any events will happen, we can't have our application block waiting for them to occur. Instead what we should do is define a callback function that will be invoked should an event actually occur. The function called `wifi_set_event_handler_cb()` does just that. It registers a function that will be called when the ESP8266 detects certain types of WiFi related events. The registered function is invoked and passed a rich data structure that includes the type of event and associated data corresponding to that event. The types of events that cause the callback to occur are:

- We connected to an access point
- We disconnected from an access point
- The authorization mode changed
- We got a DHCP issued IP address
- A station connected to us when we are in Access Point mode
- A station disconnected from us when we are in Access Point mode

Here is an example of an event handler function that simply logs the name of the event that was seen:

```
LOCAL void eventHandler(System_Event_t *event) {
	switch(event->event) {
	case EVENT_STAMODE_CONNECTED:
		os_printf("Event: EVENT_STAMODE_CONNECTED");
		break;
	case EVENT_STAMODE_DISCONNECTED:
		os_printf("Event: EVENT_STAMODE_DISCONNECTED");
		break;
	case EVENT_STAMODE_AUTHMODE_CHANGE:
		os_printf("Event: EVENT_STAMODE_AUTHMODE_CHANGE");
		break;
	case EVENT_STAMODE_GOT_IP:
		os_printf("Event: EVENT_STAMODE_CONNECTED");
		break;
	case EVENT_SOFTAPMODE_STACONNECTED:
		os_printf("Event: EVENT_SOFTAPMODE_STACONNECTED");
		break;
	case EVENT_SOFTAPMODE_STADISCONNECTED:
		os_printf("Event: EVENT_SOFTAPMODE_STADISCONNECTED");
		break;
	default:
		os_printf("Unexpected event: %d\r\n", event->event);
		break;
	}
}
```

The callback function can be registered in `user_init()` as follows:

```
wifi_set_event_handler_cb(eventHandler);
```

See also:

- wifi_set_event_handle_cb
- System_Event_t

Station configuration

When we think of an ESP8266 as a WiFi Station, we will realize that at any one time, it can only be connected to one access point. Putting it another way, there is no meaning in saying that the device is connected to **two** or more access points at the same time.

The identity of the access point to which we wish to be associated is known as the "`station_config`" and is modeled as the C structure called "`struct station_config`". Contained within that structure are two very important fields called "`ssid`" and "`password`". The `ssid` field is the SSID of the access point to which we will connect. The `password` field is the clear text value of the password that will be used to authenticate our device to the target access point to allow connection.

When booted, the ESP8266 remembers the last `station_config` we set. We can explicitly set the `station_config` data using the function `wifi_station_set_config()`. This will set the current configuration **and** save it for later retrieval after a reboot. If we only wish to set the current station config and **not** have the information persisted, we can use the `wifi_station_set_config_current()`.

We should not try and perform any WiFi operations until the device is fully initialized. We know we are initialized by registering a callback using the `system_init_done_cb()` function.

For example:

```
void initDone() {
	wifi_set_opmode_current(STATION_MODE);
	struct station_config stationConfig;
	strncpy(stationConfig.ssid, "myssid", 32);
	strncpy(stationConfig.password, "mypassword", 64);
	wifi_station_set_config(&stationConfig);
}
```

See also:

- system_init_done_cb
- wifi_station_get_config
- wifi_station_get_config_default
- wifi_station_set_config
- wifi_station_set_config_current
- wifi_set_opmode_current
- station_config

Connecting to an access point

Once the ESP8266 has been set up with the station configuration details which includes the SSID and password, we are ready to perform a connection to the target access point. The function `wifi_station_connect()` will form the connection. Realize that this is not instantaneous and you should not assume that immediately following this command you are connected. Nothing in the ESP8266 blocks and as such neither does the call to this function. Some time later, we will actually be connected. We will see two callback events fired. The first is

EVENT_STAMODE_CONNECTED indicating that we have connected to the access point. The second event is EVENT_STAMODE_GOT_IP which indicates that we have been assigned an IP address by the DHCP server. Only at that point can we truly participate in communications. If we are using static IP addresses for our device, then we will only see the connected event.

There is one further consideration associated with connecting to access points and that is the idea of automatic connection. There is a boolean flag that is stored in flash that indicates whether or not the ESP8266 should attempt to automatically connect to the last used access point. If set to true, then after the device is started and without you having to code any API calls, it will attempt to connect to the last used access point. This is a convenience that I prefer to switch off. Usually, I want control in my device to determine when I connect. We can enable or disable the auto connect feature by making a call to `wifi_station_set_auto_connect()`.

See also:

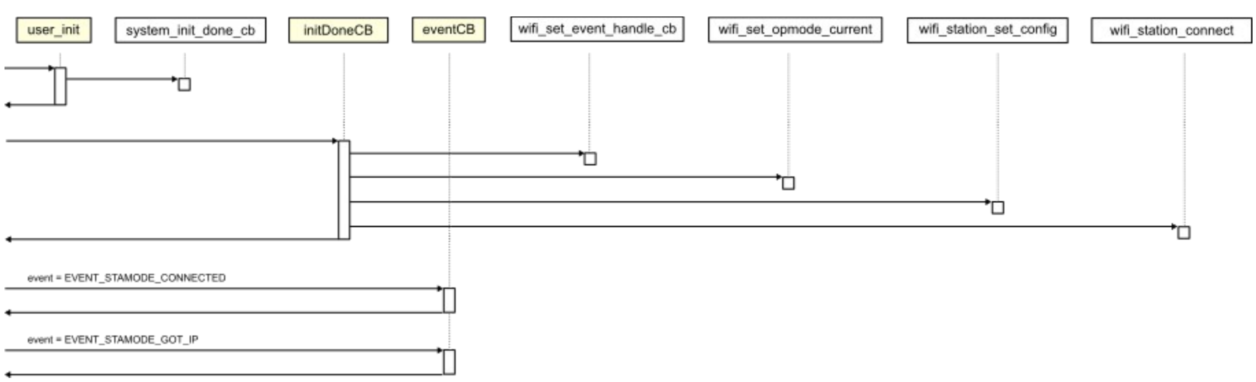
- Handling WiFi events
- wifi_station_set_auto_connect
- wifi_station_connect
- wifi_station_disconnect

Control and data flows when connecting as a station

We are now at the stage where we can draw a sequence flow of the parts. Some functions you are responsible and must supply including:

- `user_init` – Entry point into the application
- `initDoneCB` – Callback when initialization has been completed
- `eventCB` – Callback when a WiFi related event is detected

The other functions we are responsible for calling. We will consider this part of the sequence completed when we have an indication that we have a valid IP address.



Being an access point

So far we have only considered the ESP8266 as a WiFi station to an existing access point but it also has the ability to **be** an access point to other WiFi devices (stations) including other ESP8266s.

In order to be an access point, we need to define the SSID that that allows other devices to distinguish our network. This SSID can be flagged as hidden if we don't wish it to be scanned. In addition, we will also have to supply the authentication mode that will be used when a station wishes to connects with us. This is used to allow authorized stations and disallow non-authorized ones. Only stations that know our password will be allowed to connect. If we are using authentication, then we will also have to choose a password that the connecting stations will have to know and supply to successfully connect.

The first task in being an access point is to flag the ESP8266 as such using the `wifi_set_opmode()` or `wifi_set_opmode_current()` functions and pass in the flag that requests we be either a dedicate access point or an access point **and** a station.

Here is a snippet of code that can be used to setup and ESP8266 as an access point:

```
// Define our mode as an Access Point
wifi_set_opmode_current(SOFTAP_MODE);

// Build our Access Point configuration details
os_strcpy(config.ssid, "ESP8266");
os_strcpy(config.password, "password");
config.ssid_len = 0;
config.authmode = AUTH_OPEN;
config.ssid_hidden = 0;
config.max_connection = 4;
wifi_softap_set_config_current(&config);
```

When a remote station connects to the ESP8266 as an access point, we will see a debug message written to UART1 that may look similar to:

```
station: f0:25:b7:ff:12:c5 join, AID = 1
```

This contains the MAC address of the new station joining the network. When the station disconnects, we will see a corresponding debug log message that may be:

```
station: f0:25:b7:ff:12:c5 leave, AID = 1
```

From within the ESP8266, we can determine how many stations are currently connected with a call to `wifi_softap_get_station_num()`. If we wish to find the details of those stations, we can call `wifi_softap_get_station_info()` which will return a linked list of `struct station_info`. We have to explicitly release the storage allocated by this call with an invocation of `wifi_softap_free_station_info()`.

Here is an example of a snippet of code that lists the details of the connected stations:

```
uint8 stationCount = wifi_softap_get_station_num();
os_printf("stationCount = %d\n", stationCount);
```

```
struct station_info *stationInfo = wifi_softap_get_station_info();
if (stationInfo != NULL) {
	while (stationInfo != NULL) {
		os_printf("Station IP: %d.%d.%d.%d\n", IP2STR(&(stationInfo->ip)));
		stationInfo = STAILQ_NEXT(stationInfo, next);
	}
	wifi_softap_free_station_info();
}
```

See also:

- wifi_set_opmode
- wifi_set_opmode_current
- wifi_softap_get_station_num
- wifi_softap_get_station_info
- wifi_softap_free_station_info

The DHCP server

When the ESP8266 is performing the role of an access point, it is likely that you will want it to also behave as a DHCP server so that connecting stations will be able to be automatically assigned IP addresses and learn their subnet masks and gateways.

The DHCP server can be started and stopped within the device using the APIs called `wifi_softap_dhcps_start()` and `wifi_softap_dhcps_stop()`. The current status (started or stopped) of the DHCP server can be found with a call to `wifi_softap_dhcps_status()`.

The default range of IP addresses offered by the DHCP server is 192.168.4.1 upwards. The first address becomes assigned to the ESP8266 itself. It is important to realize that this address range is **not** the same address range as your LAN where you may be working. The ESP8266 has formed its own network address space and even though they may appear with the same sorts of numbers (192.168.x.x) they are isolated and independent networks. If you start an access point on the ESP8266 and connect to it from your phone, don't be surprised when you try and ping it from your Internet connected PC and don't get a response.

See also:

- wifi_softap_dhcps_start
- wifi_softap_dhcps_stop
- wifi_softap_set_dhcps_lease
- wifi_softap_dhcps_status

Current IP Address, netmask and gateway

Should we need it, we can query the OS environment for the current IP address, netmask and gateway. The values of these are commonly set for us by a DHCP server when we connect to an access point. The function called `wifi_get_ip_info()` returns our current information while the function called `wifi_set_ip_info()` allows us to set our addresses.

When we connect to an access point and have chosen to use DHCP, when we are allocated an IP address, an event is generated that can be used as an indication that we now have a valid IP address.

To correctly setup static IP addresses, in the `init_done callback`, call `wifi_station_dhcpc_stop()` to disable the DHCP client running in the ESP8266. After this call `wifi_station_connect()` to start the access point connection phase. When the event arrives that indicates we are connected to an access point as a station (`EVENT_STAMODE_CONNECTED`), we can call `wifi_set_ip_info()` and pass in the IP address, gateway and netmask that we wish to use. Note that when we use a static IP address, we will not receive the callback event that indicates we have received an IP address (`EVENT_STAMODE_GOT_IP`) as we already have it.

See also:

- Handling WiFi events
- wifi_get_ip_info
- wifi_set_ip_info
- wifi_station_dhcpc_stop
- struct ip_info

WiFi Protected Setup - WPS

The ESP8266 supports WiFi Protected Setup in station mode. This means that if the access point supports it, the ESP8266 can connect to the access point without presenting a password. Currently only the "push button mode" of connection is implemented. Using this mechanism, a physical button is pressed on the access point and, for a period of two minutes, any station in range can join the network using the WPS protocols. An example of use would be the access point WPS button being pressed and then the ESP8266 device calling `wifi_wps_enable()` and then `wifi_wps_start()`. The ESP8266 would then connect to the network.

See also:

- wifi_wps_enable
- wifi_wps_start
- wifi_set_wps_cb
- Simple Questions: What is WPS (WiFi Protected Setup)
- Wikipedia: WiFi Protected Setup

Working with TCP/IP

TCP/IP is the network protocol that is used on the Internet. It is the protocol that the ESP8266 natively understands and uses with WiFi as the transport. Books upon books have already been written about TCP/IP and our goal is not to attempt to reproduce a detailed discussion of how it works, however, there are some concepts that we will try and capture.

First, there is the IP address. This is a 32bit value and should be unique to every device connected to the Internet. A 32bit value can be thought of as four distinct 8bit values (4x8=32).

Since we can represent an 8bit number as a decimal value between 0 and 255, we commonly represent IP addresses with the notation <number>.<number>.<number>.<number> for example 173.194.64.102. These IP addresses are not commonly entered in applications. Instead a textual name is typed such as "`google.com`" … but don't be misled, these names are an illusion at the TCP/IP level. All work is performed with 32bit IP addresses. There is a mapping system that takes a name (such as "`google.com`") and retrieves its corresponding IP address. The technology that does this is called the "Domain Name System" or DNS.

When we think of TCP/IP, there are actually three distinct protocols at play here. The first is IP (Internet Protocol). This is the underlying transport layer datagram passing protocol. Above the IP layer is TCP (Transmission Control Protocol) which provides the illusion of a connection over the connectionless IP protocol. Finally there is UDP (User Datagram Protocol). This too lives above the IP protocol and provides datagram (connectionless) transmission between applications. When we say TCP/IP, we are **not** just talking about TCP running over IP but are in fact using this as a shorthand for the core protocols which are IP, TCP and UDP and additional related application level protocols such as DNS, HTTP, FTP, Telnet and more.

The ESPConn architecture

Because we are not allowed to block control in the ESP8266 for any length of time, we must register callback functions which will be invoked when some long duration action has completed or an asynchronous events occurs. For example, when we wish to receive an incoming network connection, we can't simply wait for that connection to arrive. Instead, we register a connection callback function and then return control back to the OS. When the connection eventually arrives in the future, the callback function that we previously registered is invoked on our behalf.

The following table lists the callback functions that the ESP8266 provides supporting TCP connections and events.

Register Function	Callback	Description
espconn_regist_connectcb	espconn_connect_callback	TCP connected successfully
espconn_regist_disconcb	espconn_disconnect_callback	TCP disconnected successfully
espconn_regist_reconcb	espconn_reconnect_callback	Error detected or TCP disconnected
espconn_regist_sentcb	espconn_sent_callback	Sent TCP or UDP data
espconn_regist_recvcb	espconn_recv_callback	Received TCP or UDP data
espconn_regist_write_finish	espconn_write_finish_callback	Write data into TCP-send-buffer

See also:

- espconn_regist_connectcb
- espconn_regist_disconcb
- espconn_regist_reconcb
- espconn_regist_sentcb
- espconn_regist_recvcb

- espconn_regist_write_finish

TCP

A TCP connection is a bi-directional pipe through which data can flow in both directions. Before the connection is established, one side is acting as a server. It is passively listening for incoming connection requests. It will simply sit there for as long as needed until a connection request arrives. The other side of the connection is responsible for initiating the connection and it actively asks for a connection to be formed. Once the connection has been constructed, both sides can send and receive data. In order for the "client" to request a connection, it must know the address information on which the server is listening. This address is composed of two distinct parts. The first part is the IP address of the server and the second part is the "port number" for the specific listener. If we think about a PC, you may have many applications on it, each of which can receive an incoming connection. Just knowing the IP address of your PC is not sufficient to address a connection to the correct application. The combination of IP address plus port number provides all the addressing necessary.

As an analogy to this, think of your cell phone. It is passively sitting there until someone calls it. In our story, it is the listener. The address that someone uses to form a connection is your phone number which is comprised of an area code plus the remainder. For example, a phone number of (817) 555-1234 will reach a particular phone. However the area code of 817 is for Fort Worth in Texas … calling that by itself is not sufficient to reach an individual … the full phone number is required.

No we will look at how an ESP8266 can set itself up as a listener for an incoming TCP/IP connection.

We start by introducing an absolutely vital data structure that is called "`struct espconn`". This data structure contains much of the "state" of our connection and is passed into most of our TCP APIs.

We initialize it by setting a number of its fields:

- `type` – This is the type of connection we are going to use. Since we want to use a TCP connection as opposed to a UDP connection, we supply `ESPCONN_TCP` as the value.
- `state` – The state of the connection will change over time but we initialize it to have an initial empty state by supplying `ESPCONN_NONE`.

For example:

```
LOCAL struct espconn conn1;

LOCAL void init() {
      conn1.type = ESPCONN_TCP;
      conn1.state = ESPCONN_NONE;
}
```

Now we introduce another structure called "`esp_tcp`". This structure contains TCP specific settings. For our story, this is where we supply the port number upon which our TCP connection will listen for client connections. This is supplied in the property called "`local_port`".

```
LOCAL esp_tcp tcp1;

LOCAL void init() {
       tcp1.local_port = 25867;
}
```

Within the `struct espconn` data type, there is a field called "`proto`" which is a pointer to a protocol specific data structure. For a TCP connection, this will be a pointer to an "`esp_tcp`" instance … and this is where we get to glue the story together. The full code becomes:

```
LOCAL struct espconn conn1;
LOCAL esp_tcp tcp1;

LOCAL void init() {
       tcp1.local_port = 25867;
       conn1.type = ESPCONN_TCP;
       conn1.state = ESPCONN_NONE;
       conn1.proto.tcp = &tcp1;
}
```

We can now start our server listening for incoming TCP connections using `espconn_accept()`. This takes the `struct espconn` as input which is used to indicate on what port we should listen (among other things). Here is an example:

```
espconn_accept(&conn1);
```

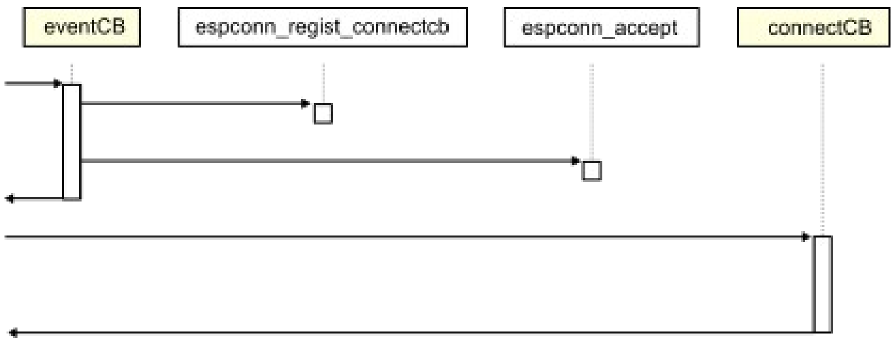
After calling this, the ESP8266 will now be passively listening for incoming TCP connections on the port specified in the `local_port` field. It is important to note that your API does not block waiting for an incoming request. Somewhere in the heart of the ESP8266 it now know to accept connections on that port. The next question is a simple one … what happens when a connection eventually arrives?

The answer to that is part of the core architecture of the device and revolves around the notion of callbacks. In your own application code, it is your responsibility to register a callback function that will be invoked when the connection arrives. This is where the `espconn_regist_connectb()` function comes into play. This function registers a user supplied callback function that will be called when a connection arrives.

```
void connectCB(void *arg) {
       struct espconn *pNewEspConn = (struct espconn *)arg;
       ...
}

{
       ...
       espconn_regist_connectcb(&conn1, connectCB);
       espconn_accept(&conn1);
}
```

Seen as a sequence flow diagram, we can see the relationships between some of the components. We assume that in the event callback when we have been allocated an IP address, we then register that we are interested in connections and that we are willing to accept incoming new connections. Then, at some time in the future, we receive a new connection request and the connection callback is invoked.



The content of the `struct espconn` passed into the callback will include the remote IP address of the partner that connected with us. We can use that information for logging or for authorization. For example, if the IP address is not one we wish to allow, we can disconnect at this point using `espconn_disconnect()`. Realize that this data structure represents the **new** connection with the partner that just invoked up and is **not** the same as `struct espconn` that was used to register that we wanted to accept new connections. A new `struct espconn` will be passed in for each new connection formed.

This covers the ESP8266 receiving incoming connection requests, but what if it should desire to form a connection outbound to a remote TCP application? To perform an outbound connection request we can use the `espconn_connect()` call. Just like the receiving an inbound connection, making an outbound connection will result in an invocation to the connection callback when the connection is established. Once the connection has been formed, once again, the two ends of the connection will be peers of each other.

If the partner in our conversation should close the connection, we will be informed of that through the function we register with `espconn_regist_disconcb()`. The state field of the `struct espconn` will contain `CLOSE`. Detection the graceful shutdown of a partner allows us to perform logic that we may need such as releasing resources or persisting data.

If a TCP connection is formed and no traffic flows over the connection for at least 10 seconds (default), then the connection is automatically closed from the ESP8266 end. The idle connection timeout property can be set with the `espconn_regist_time()` function.

See also:

- espconn_accept
- espconn_connect
- espconn_disconnect
- espconn_regist_connectcb
- espconn_regist_disconcb
- espconn_regist_time

- struct espconn
- esp_tcp

Sending and receiving TCP data

At this point, let us now assume that we have a connection between an ESP8266 and a partner application. Having a connection is great but now we need to have a conversation. Information and data needs to flow in one or both directions. There are two considerations… we may receive data from the partner or we may wish to send data to the parter. It is important to note that in TCP, a connection is bidirectional. Once the connection has been established, either party can send data at any time. There is no concept of one party having exclusive sending or receiving rights. The choice of who is the receiver and who is the transmitter is purely up to the design of the application.

For example, imagine we had a project to turn on an LED at an ESP8266 when it receives a "1" character and turn it off when it receives a "0" character. In that story, the ESP8266 would be exclusively a receiver and, simply by our choices, need not transmit data. The partner would be exclusively a transmitter.

Now let us consider a second example. In this case the ESP8266 is connected to a temperature sensor and every few seconds it sends the current temperature to the partner. In that story, the ESP8266 is exclusively a transmitter and the partner only a receiver.

Finally, we can image an ESP8266 connected to multiple sensors. It receives commands from the partner as input which it interprets. Based on the received data, the correct sensor is chosen, its value read and the results transmitted back. In this story, the ESP8266 is at first a receiver and then becomes a transmitter while the partner is the opposite.

To receive data from a partner, we register a callback function using `espconn_regist_recvcb()`. We pass in the `struct espconn` that was supplied in the connected callback that identifies our connection. This registered callback function is invoked when new data becomes available from the partner. The callback function is passed a buffer containing the data and an indicator of how much data was received.

The following is an example of logging data that is received over the network:

```
LOCAL void recvCB(void *arg, char *pData, unsigned short len) {
	struct espconn *pEspConn = (struct espconn *)arg;
	os_printf("Received data!! - length = %d\n", len);
	int i=0;
	for (i=0; i<len; i++) {
		os_printf("%c", pData[i]);
	}
	os_printf("\n");
} // End of recvCB
```

The function called `recvCB()` is registered as a callback when data is available for the connection. With this in mind, we can start running some experiments and the results will be interesting.

If we send data, we see the callback being invoked as expected. However, as the size of the data transmitted, which is received by the ESP8266, increases, at about 1460 bytes, a strange thing happens. Instead of `recvCB()` being called once, we see it being called twice. The first time it gets the first 1460 bytes and the second time it gets what remains. This is repeated for increments of 1460 byte transmission sizes. For example, if we send 5000 bytes, `recvCB()` is called 4 times. The first three times with 1460 bytes of data and the last with 620 bytes giving a total of 5000.

Why would this be? Part of the answer is that the ESP8266 has only a very small amount of RAM available to it and needs to be able to support parallel connections. As such, it can apparently throttle the data being sent from the sender until space is available to process it.

It can't be stressed enough the importance of this concept. Data sent from the server over a TCP connection is "streamed" to the ESP8266. There is no concept of a unit of data transmission. Instead data sent in the pipe at the sender will arrive at the ESP8266 but it may very well arrive at different rates. The order of the transmitted data is preserved (obviously). In principle, making two transmissions at the sender of 5 bytes each could result in one receive at the ESP8266 of 10 bytes. Don't make **any** assumptions about the bracketing of TCP data.

To transmit data to a partner we use the function called `espconn_sent()`.

> Note: Espressif is a Chinese company based out of Shanghai. My knowledge of the Chinese language is nill. I only speak one language, English and even that poorly and I am in awe of those who can juggle more than one. However, we are all human and we all have the opportunity to make mistakes.
>
> Take for example, the command "`espconn_sent()`". When called, its purpose is to "send" data. If you are a native English speaker, it is likely to be obvious the difference in "meaning" between "sent" and "send" … however, spare a thought for those whose native language is not English and also whose character set is not the Latin set you are reading now.
>
> There are a few places in the ESP8266 API documentation where there are items that either read oddly in English or are plain wrong from an English grammar perspective. What you should do if you find one of these is check the bug log at the Espressif BBS and, if not already reported, be a good citizen and report it. Try not to roll your eyes and ask why no-one caught it before now … instead … help everyone and report it and feel good that you helped push the ball forward.
>
> I anticipate that over time, the API will change to correct such potential defects … so check often. You may find that code you write today using "`espconn_sent()`" won't work when a new patch or release is applied because the function was renamed to "`espconn_send()`".

This command takes the `struct espconn` which identifies which connection to send data through. The function also takes a pointer to a buffer of data and the length of the data to send.

A vital consideration is that the data to be sent is not sent immediately. When we call `espconn_sent()` what we are doing is handing off a buffer of data to be transmitted at some time in the future. We anticipate this will be a few microseconds but it could be longer. We must honor the contract. When the ESP8266 does successfully transmit the data, a callback will be made to a function that was registered with the `espconn_regist_sentcb()`. Only after having seen a confirmation that the last transmission request has been completed should we execute another `espconn_sent()` request.

When we ask for data to be transmitted, we provide a pointer to a buffer that contains the data. It is important to realize that we must maintain that data until after we are sure its content has been sent. For example, we can't request a transmission and then immediately dispose off or change the buffer. What we hand off to the OS is a pointer to a buffer and until the OS tells us that it has finished consuming it, we must maintain its integrity.

See also:

- espconn_regist_recvcb
- espconn_sent

TCP Error handling

When a connection is formed between two partners it is essential that we realize that there isn't an actual dedicated underlying connection between them. Instead, there is only a logical connection that appears to be present over the datagram oriented protocol of IP. What this might mean is that if one end of the connection abnormally ends, the other end won't immediately know about it. As an example, if in the real world I make a phone call to you then your phone indicates to you that we have a connection. If the battery on my phone dies the telephone network detects that and drops the connection. Your phone also hangs up and you know we are no longer in communication. In the TCP world, that doesn't happen. If my "TCP" phone dies, your "TCP" phone isn't told that mine is gone. You may be left sitting there indefinitely listening to silence and waiting for me to say something.

To resolve that situation, TCP introduces a concept called "keep-alive". The notion is very simple. With keep-alive, the two partners periodically exchange a heartbeat communication with each other. As long as they each hear the heartbeat of the other, they are both still present. However, if one side of the connection is lost, the heartbeat request will be sent but no response will arrive at which point, the one sending the heartbeat will assume that the partner has gone and we can take appropriate cleanup and shutdown actions.

There is an API available to us to control the keep-alive settings. It is called `espconn_set_keepalive()`. It has a number of properties including:

- How long should we wait since the last time we heard from the partner before sending a heartbeat?
- If no response, how long between subsequent heartbeats?

- How many times should we send a heartbeat until we declare the partner dead?

It is recommended that if keep-alive processing is to be used then the keep-alive settings be made in the callback handler of the connect callback. The keep-alive option must also be explicitly enabled using the `espconn_set_opt()` call prior to setting the keep-alive properties.

If the partner connection is lost, we can detect that by registering a callback function with `espconn_reconnect_callback()`.

See also:

- espconn_set_keepalive
- espconn_get_keepalive
- espconn_set_opt
- espconn_clear_opt

UDP

If we think of TCP as forming a connection between two parties similar to a telephone call, then UDP is like sending a letter through the postal system. If I were to send you a letter, I would need to know your name and address. Your address is needed so that the letter can be delivered to the correct house while your name ensure that it ends up in your hands as opposed to someone else who may live with you. In TCP/IP terms, the address is the IP address and the name is the port number.

With a telephone conversation, we can exchange as much or as little information as we like. Sometimes I talk, sometimes you talk … but there is no maximum limit on how much information we can exchange in one conversation. With a letter however, there are only so many pages of paper that will fit in the envelopes I have at my disposal.

The notion of the mail analogy is how we might choose to think about UDP. The acronym stands for User Datagram Protocol and it is the notion of the datagram that is akin to the letter. A datagram is an array of bytes that are transmitted from the sender to the receiver as a unit. The maximum size of a datagram using UDP is 64KBytes. No connection need be setup between the two parties before data starts to flow. However, there is a down side. The sender of the data will not be made aware of a receiver's failure to retrieve the data. With TCP, we have handshaking between the two parties that lets the sender know that the data was received and, if not, can automatically retransmit until it has been received or we decide to give up. With UDP, and just like a letter, when we send a datagram, we lose sight of whether or not it actually arrives safely at the destination.

If we wish to receive incoming datagrams, we must register what port number we are interested in receiving them upon. We achieve that through the poorly named `espconn_create()` function. This function causes the ESP8266 to start listening for incoming datagrams on the local port defined in the `struct espconn`. After calling this function, you should then call `espconn_regist_recvcb()` to register a callback function that will be invoked when a datagram arrives.

Here is a high level example of setting up a UDP listener once an IP address has been allocated:

```
LOCAL struct espconn conn1;
LOCAL esp_udp udp1;

LOCAL void setupUDP() {
       sint8 err;
       conn1.type = ESPCONN_UDP;
       conn1.state = ESPCONN_NONE;
       udp1.local_port = 25867;
       conn1.proto.udp = &udp1;

       err = espconn_create(&conn1);
       err = espconn_regist_recvcb(&conn1, recvCB);
} // End of setupUDP
```

Should we wish to stop the ESP8266 from listening for datagrams, we can call the function called `espconn_delete()`.

Now is a good time to come back to IP addresses and port numbers. We should start to be aware that on a PC, only one application can be listening upon any given port. For example, if my application is listening on port 25867, then no other application can also be listening on that same port … not your application nor another copy/instance of mine. When an incoming connection or datagram arrives at a machine, it has arrived because the IP address of the sent data matches the IP address of the device at which it arrived. We then route within the device based on port numbers. And here is where I want to clarify a detail. We route within the machine based on the **pair** of both protocol and port number.

So for example, if a request arrives at a machine for port 25867 over a TCP connection, it is routed to the TCP application watching port 25867. If a request arrives at the same machine for port 25867 over UDP, it is routed to the UDP application watching port 25867. What this means is that we **can** have two applications listening on the same port but on different protocols. Putting this more formally, the allocation space for port numbers is a function of the protocol and it is not allowed for two applications to simultaneously reserve the same port within the same protocol allocation space. Although I used the story of a PC running multiple applications, in our ESP8266 the story is similar even though we just run one application on the device. If your single application should need to listen on multiple ports, don't try and use the same port with the same protocol as the second function call will find the first one has already allocated the port. This is a detail that I am happy for you to forget as you will rarely come across it but I wanted to catch it here for completeness.

Now let us look at what it takes to send a datagram. Similar to other functions, we need a `struct espconn` control block. This must be configured to use UDP and name the remote IP address and port. Once populated, we can then initialize the data structure with a call to `espconn_create()` and now we are ready to send data. We use the `espconn_sent()` function.

When we have sent all our data, we can conclude with an `espconn_delete()` to release the resources that the ESP8266 maintains for data sending.

Here is an example:

```
LOCAL struct espconn sendResponse;
LOCAL esp_udp udp;

void sendDatagram(char *datagram, uint16 size) {
       sendResponse.type = ESPCONN_UDP;
       sendResponse.state = ESPCONN_NONE;
       sendResponse.proto.udp = &udp;
       IP4_ADDR((ip_addr_t *)sendResponse.proto.udp->remote_ip, 192, 168, 1, 7);
       sendResponse.proto.udp->remote_port = 9876; // Remote port
       err = espconn_create(&sendResponse);
       err = espconn_sent(&sendResponse, "hi123", 5);
       err = espconn_delete(&sendResponse);
}
```

See also:

- espconn_create
- espconn_delete
- espconn_sent
- espconn_regist_recvcb
- espconn_regist_sentcb
- struct espconn

Broadcast with UDP

One of the features available to us with UDP is the concept of broadcast. This is the notion that a sender of data can build a datagram and transmit it such that all the devices on the same subnet can receive a copy of it. Receivers choose a UDP port and start listening upon it just as they normally would. A transmitting application transmits a message on the same port but with an IP address where the host part of the IP address is all binary ones. For example, if we have a netmask of `255.255.255.0` and our network is `192.168.1.x`, then transmitting on the IP address `192.168.1.255` will be a broadcast. A special IP address of `255.255.255.255` represents a broadcast on our local network.

For the ESP8266, there is an API called `wifi_set_broadcast_if()` which determines which interfaces will be available for broadcast. The choices are the station, the access point or both the station and access point. A corresponding API called `wifi_get_broadtcast_if()` can be used to retrieve the current broadcast configuration state.

See also:

- wifi_set_broadcast_if
- wifi_get_broadcast_if

Ping request

At the TCP/IP level, a device with an IP address can "ping" another device with an IP address. What this means is that messages are transmitted between them that allows them to know that they have a route through the network to each other. If the destination is either not running or no route is available, we will also be informed that there was a failure.

The ESP8266 provides a structure called `struct ping_option` that contains the details of a ping request. This is passed in as a parameter to the function called `ping_start()` which initiates the ping. Before calling this function, the target IP address and the number of ping requests should be set within the `struct ping_option`.

Two callback functions can be registered with `ping_regist_recv()` and `ping_regist_sent()`. The first is called when a ping response is received and the other is called when a ping request is sent.

See also:

- ping_start
- ping_regist_recv
- ping_regist_sent
- struct ping_option

Name Service

On the Internet, server machines can be located by their Domain Name Service (DNS) names. This is the service that resolves a human readable representation such as "`google.com`" into the necessary IP address value (eg. `216.58.217.206`). In order for this transformation to happen, the ESP8266 needs to know the IP address of one or more DNS servers that it will then use to perform the name to IP address mapping. If we are using DHCP then nothing else need be done as the DHCP server automatically provides the DNS server addresses. However, if we should not be using DHCP, then we need to instruct the ESP8266 of the locations of the DNS servers manually. We can do this using the `espconn_dns_setserver()` function. This takes an array of one or two IP addresses as input and from that point onwards, these servers will be used for DNS resolution. If two addresses are supplied and the first is unresponsive, the second will be used.

Google publicly makes available two name servers with the addresses of 8.8.8.8 and 8.8.4.4.

See also:

- espconn_dns_setserver
- espconn_gethostbyname
- Wikipedia: Domain Name System
- Google: Public DNS

Multicast Domain Name Systems

Using the Multicast Domain Name System (mDNS) an ESP8266 can attempt to resolve a hostname of a machine on the local network to its IP address. It does this by broadcasting a packet asking for the machine with that identity to respond.

See also:

- Wikipedia – Multicast DNS
- IETF RFC 6762: Multicast DNS

Working with SNTP

SNTP is the Simple Network Time Protocol and allows a device connected to the Internet to learn the current time. In order to use this, you must know of at least one time server located on the Internet. The US National Institute for Science and Technology (NIST) maintains a number of these which can be found here:

http://tf.nist.gov/tf-cgi/servers.cgi

Other time servers can be found all over the globe and I encourage you to Google search for your nearest or country specific server.

Once you know the identity of a server by its hostname or IP address, you can call either of the functions called `sntp_setservername()` or `sntp_setserver()` to declare that we wish to use that time server instance. The ESP8266 can be configured with up to three different time servers so that if one or two are not available, we might still get a result.

The ESP8266 must also be told the local timezone in which it is running. This is set with a call to `sntp_set_timezone()` which takes the number of hours offset from UTC. For example, I am in Texas and my timezone offset becomes "`-5`".

With these configured, we can start the SNTP service on the ESP8266 by calling `sntp_init()`. This will cause the device to determine its current time by sending packets over the network to the time servers and examining their responses. It is important to note that immediately after calling `sntp_init()`, you will not yet know what the current time may be. This is because it may take a few seconds for the ESP8266 to sends the time requests and get their responses and this will all happen asynchronously to your current commands and won't complete till sometime later.

When ready, we can retrieve the current time with a call to `sntp_get_current_timestamp()` which will return the number of seconds since the 1st of January 1970 UTC. We can also call the function called `sntp_get_real_time()` which will return a string representation of the time.

See also:

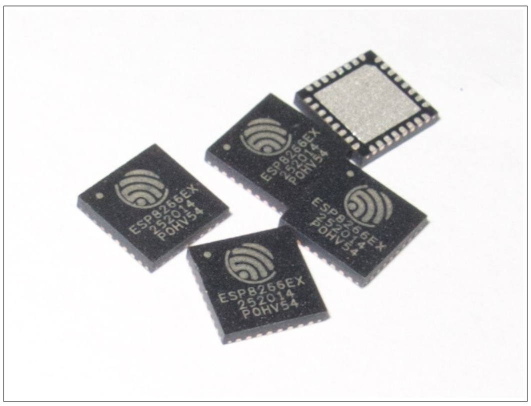
- sntp_setserver
- sntp_setservername
- sntp_init

- sntp_set_timezone
- sntp_get_current_timestamp
- sntp_get_real_time
- [IETF RFC5905: Network Time Protocol Version 4: Protocol and Algorithms Specification]

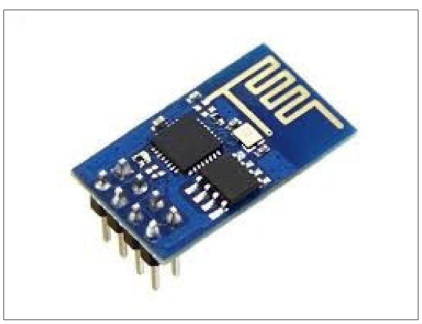
GPIOs

The ESP8266 has 17 GPIO pins. When we think of a GPIO we must realize that at any one time, each instance has two modes. It can either be an input or an output. When it is an input, we can read a value from it and determine the logic level of the signal present at the physical pin. When it is an output, we can write a logic level to it and that will appear as a physical output.

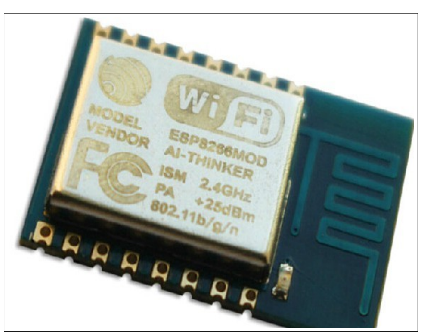
Remember to distinguish between the ESP8266 integrated circuit which is a tiny device:



which differs from the various models of breakout board such as the ESP-1:



which has 8 pins exposed, 4 of which are GPIO or the ESP-12:

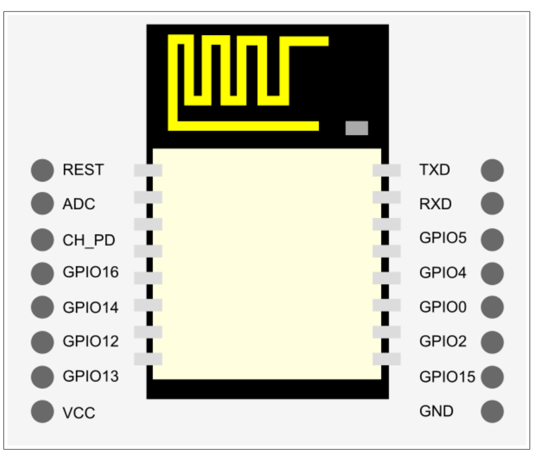


which has 16 pins exposed, 11 of which are GPIO.

For GPIO, here are the exposed mappings:

Pin	ESP-1	ESP-12
GPIO 0	●	●
GPIO 1	●	●
GPIO 2	●	●
GPIO 3	●	●
GPIO 4		●
GPIO 5		●
GPIO 6		
GPIO 7		
GPIO 8		
GPIO 9		
GPIO 10		
GPIO 11		
GPIO 12		●
GPIO 13		●
GPIO 14		●
GPIO 15		●
GPIO 16		●
Totals	**4**	**11**

It is also good to remind ourselves of the pin-outs of the device.



As you can see there is no obvious pattern to the layout of the pins and as such you must take great care when wiring up a circuit. It is easy to make a mistake.

Another vital consideration about working with GPIOs is voltage. The ESP8266 is a 3.3V device. You need to be extremely cautious of you are working with 5V (or above) partner MCUs or sensors. Unfortunately devices like the Arduino are typically 5V as are USB → UART converters and many sensors. This means you are as likely as not to be working in a mixed voltage environment. Under no circumstances think you can power the ESP8266 with a direct voltage of more than 3.3V. Obviously, you can convert higher voltages down to 3.3V but never try and connect a greater voltage directly. Another subtler consideration is when using GPIOs for signal input and supply greater than 3.3V as a high signal value. I strongly suggest not doing it. Some folks may claim you can "get away with it" and if you experiment it may (seem) to work but you are taking an unnecessary risk for no obviously good reason. If it works … then it will work till it doesn't at which point it will be too late and you may cook your device.

In my own experiments, I have accidentally over-powered ESP8266s, reverse voltage powered ESP8266s and applied too high a voltage as input. In each case the result was a dead chip and in a few cases, attempting to see if it still worked by applying normal voltage resulted in the device not only not working but getting so hot to the touch it burned my fingers.

Because accidents happen when building GPIO based circuits, I recommend buying more ESP8266 instances than you need. That way if you do happen to find yourself needing a second (or third or fourth) you will have them at your disposal.

The way that the ESP8266 thinks of GPIOs is as though each GPIO was a bit in a 16bit array.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(We will come back to how 17 GPIOs maps to 16 bits at a later time)

One array contains an indication of whether or not the GPIO is input or output. We will call this the direction array. A second array indicates the values of the GPIOs. For input GPIOs, the value is the value on the pin. For output GPIOs, the value is the value to be written to the pin. We will call this the value array.

A function is supplied by the ESP8266 called `gpio_output_set()`. This function takes **four** 16 bit values to be used as masks against the two 16 bit arrays.

The first mask is called the "`set_mask`". A 1 value in the set mask sets the corresponding bit value to be 1 in the value array.

The second mask is called the "`clear_mask`". A 1 value in the clear mask sets the corresponding bit value to be 0 in the value array.

Notice that in both cases, if the masks have a 0 value, the original values are unchanged.

The third mask is called the "`enable_output`" mask. A 1 value in the enable output mask sets the corresponding GPIO to be in output mode.

The fourth mask is called the "`enable_input`" mask. A 1 value in the enable input mask sets the corresponding GPIO to be in input mode.

Take care not to set a GPIO to be both input and output or to have a value of both 1 and 0. The results will be undefined.

Constants are defined for each of the bit positions. Those constants are:

- BIT0 – 2^0
- BIT1 – 2^1
- …
- BIT31 – 2^31

So, for example. If we want to set GPIO 5 to be input, we might code:

```
gpio_output_set(0, 0, 0, BIT5);
```

to set GPIO 4 to be output and have a high value, we might code:

```
gpio_output_set(BIT4, 0, BIT4, 0);
```

to set GPIO 0 and 1 to both be output and the first to be 1 and the second to be 0:

```
gpio_output_set(BIT0, BIT1, BIT0 | BIT1, 0);
```

If we wish to retrieve the values of the GPIOs, we can use the `gpio_input_get()` method. This returns a bit mask containing all the bits.

We have some helper macros that are available. These are useful wrappers around `gpio_output_set()` and `gpio_input_get()`.

- `GPIO_OUTPUT_SET(GPIO_NUMBER, value)` – Sets the corresponding GPIO to be output and sets its value.
- `GPIO_DIS_OUTPUT(GPIO_NUMBER)` – Sets the corresponding GPIO to be input (disabled output).
- `GPIO_INPUT_GET(GPIO_NUMBER)` – Gets the value of the input GPIO

Since pins on an ESP8266 can serve multiple purposes, we must first declare what function that pin will have. To do this, we use a macro which sets the function of the logical pin:

```
PIN_FUNC_SELECT(pinName, functionUsage)
```

For example, to define GPIO2 as a GPIO pin and set its value, we might code:

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO2_U, FUNC_GPIO2);
GPIO_OUTPUT_SET(2, 1);
```

Here is the complete table of mappings.

Pin Name	Function 1	Function 2	Function 3	Function 4	Physical pin	Devices
MTDI_U	MTDI	I2SI_DATA	HSPIQ MISO	GPIO12	10	12
MTCK_U	MTCK	I2SI_BCK	HSPID MOSI	GPIO13	12	12
MTMS_U	MTMS	I2SI_WS	HSPICLK	GPIO14	9	12
MTDO_U	MTDO	I2SO_BCK	HSPICS	GPIO15	13	12
U0RXD_U	U0RXD	I2SO_DATA		GPIO3	25	1, 12
U0TXD_U	U0TXD	SPICS1		GPIO1	26	1, 12
SD_CLK_U	SD_CLK	SPICLK		GPIO6	21	
SD_DATA0_U	SD_DATA0	SPIQ		GPIO7	22	
SD_DATA1_U	SD_DATA1	SPID		GPIO8	23	
SD_DATA2_U	SD_DATA2	SPIHD		GPIO9	18	
SD_DATA3_U	SD_DATA3	SPIWP		GPIO10	19	
SD_CMD_U	SD_CMD	SPICS0		GPIO11	20	
GPIO0_U	GPIO0	SPICS2			15	1, 12
GPIO2_U	GPIO2	I2SO_WS	U1TXD		14	1, 12
GPIO4_U	GPIO4	CLK_XTAL			16	12
GPIO5_U	GPIO5	CLK_RTC			24	12

The following are the keys to some of the values in the table:

- Devices column
    - 1=ESP-1
    - 12=ESP-12

Here are the GPIO pins by mapping:

GPIO	Pin Name	Notes	Risk
GPIO0	GPIO0_U	Pin controls state of ESP8266 at boot. Caution when used as an output pin.	
GPIO1	U0TXD_U	Pin is commonly used for flashing the device.	
GPIO2	GPIO2_U	Used for UART1 output and, as such, is likely to be used during development time for debugging. Written to when flashed with new firmware.	
GPIO3	U0RXD_U	Pin is commonly used for flashing the device.	
GPIO4	GPIO4_U	Only use is as a GPIO.	
GPIO5	GPIO5_U	Only use is as a GPIO.	
GPIO6	SD_CLK_U	Not exposed on current devices.	
GPIO7	SD_DATA0_U	Not exposed on current devices.	
GPIO8	SD_DATA1_U	Not exposed on current devices.	
GPIO9	SD_DATA2_U	Not exposed on current devices.	
GPIO10	SD_DATA3_U	Not exposed on current devices.	
GPIO11	SD_CMD_U	Not exposed on current devices.	
GPIO12	MTDI_U		
GPIO13	MTCK_U		
GPIO14	MTMS_U		
GPIO15	MTDO_U	Used to control UART0 RTS and hence may have an influence on firmware flashing since the firmware data arrives via UART0.	
GPIO16	???	???	

The maximum output current from a GPIO pin is only 12mA.

Given a choice, if you are using GPIO0, use it as an input pin as opposed to an output pin. The reason for this is that when you are developing solutions, you need to bring GPIO0 low to place the ESP8266 into flash mode where it reads new programs from the UART. This means that you will be changing the input signal to GPIO0. If you use the pin as an output, there is the possibility that when you change your wiring to bring it low or press a button to bring it low, if the signal is high at that time, you will short the circuit. However, if the pin is input then that won't be a problem. Ideally, avoid using GPIO0 altogether and leave it specifically for bootstrapping the device in different modes.

See also:

- PIN_FUNC_SELECT
- GPIO_OUTPUT_SET
- GPIO_DIS_OUTPUT
- GPIO_INPUT_GET
- gpio_output_set
- gpio_input_get

Working with serial

There are two UARTs in the system known as UART0 and UART1. UART0 has its own dedicated TX and RX pins while UART1 is multiplexed with GPIO2. UART1 is output only and hence only has a TX line.

The serial interface to the ESP8266 can be initialized with a call to the function `uart_init()`.

For example

```
uart_init(BIT_RATE_115200, BIT_RATE_115200);
```

To write a string to the serial port, we can then use `os_printf()`. This has the same format as a `printf` but writes to the serial port.

In order to work with UART, you must include the `uart.c`, `uart.h` and `uart_register.h` files from `examples/driver_lib`. In your application, you must then include "`driver/uart.h`".

To transmit data using UART0, we have the function called `uart0_tx_buffer()` which accepts a pointer to data and a length and transmits it.

See also:

- Connecting to the ESP8266
- USB to UART converters
- uart_init
- uart0_tx_buffer
- uart0_rx_intr_handler
- os_printf

Task handling

Imagine we wish to have a task performed for us asynchronously. What we might want to do is post that we wish this to happen and then go on with our business. When we are done and have relinquished control back to the OS, we assume that the task will eventually start executing. This is the function provided by the task functions of the ESP8266. There are two functions of interest to us. The first is called `system_os_task()` sets up a task processor.

When we wish to post that a task is eligible to start, we can use the second function called `system_os_post()` which posts a message.

The task function that we registered will then be "invoked" at some point after the post request and will be given the parameters supplied in the post. The priority identifies the relative priority of two posts that have been issued. The one with the highest priority will execute first.

It is important to note that only **three** priorities are allowed which are 0, 1 and 2 with 0 having the lowest priority. It is also important to note that there can only be **one** handler for each task registrations. So if we execute `system_os_task()` twice using the same priority in both cases, only the last one is remembered and will be executed when a task of that priority is posted.

See also:

- system_os_task
- system_os_post

Timers and time

Within our code, we may wish to delay for a period of time. We can use the `os_delay_us()` function to suspend processing for a given period measured in microseconds. There are 1000 microseconds in a millisecond and a 1000 milliseconds in a second.

We can configure a timer to be called on a periodic basis. A data structure called `os_timer_t` holds the state of the timer.

We can define the user function to be called when the timer fires using the `os_timer_setfn()` function. Note that we can only set the callback function when the timer is disarmed.

When ready, we can arm the timer so that it starts ticking and fires when ready. We do this using the `os_timer_arm()` function.

The repeat flag indicates whether the timer should restart after it has fired.

We can suspend or cancel the firing of the timer using `os_timer_disarm()`.

Here is an example:

```
os_timer_t myTimer;

void timerCallback(void *pArg) {
      os_printf("Tick!);
} // End of timerCallback

void user_init(void) {
      uart_init(BIT_RATE_115200, BIT_RATE_115200);
      os_timer_setfn(&myTimer, timerCallback, NULL);
      os_timer_arm(&myTimer, 1000, 1);
} // End of user_init
```

Another aspect of working with time is time calculations and measurement. The function `system_get_time()` returns a 32 bit unsigned (unit32) value which is the microseconds since the device booted. This value will roll over after 71 minutes.

We can also explicitly block execution for a period of time using `os_delay_us()`.

See also:

- system_get_time
- os_timer_arm
- os_timer_disarm
- os_timer_setfn

Working with memory

When working in C, you have to think in terms of computer memory. With great power comes great responsibility. The amount of available RAM is likely to be less than 45KBytes.

We can allocate memory using `os_malloc()` or `os_zalloc()`. The first function allocates and returns memory and the second does exactly the same but zeros the memory before returning. When your logic no longer needs the memory, it can return it back to the heap with `os_free()`. To determine how much heap size is available, we can call `system_get_free_heap_size()`. Once we have the memory pointer, we can start to manipulate it through a series of memory commands. The `os_memset()` command will set a block of memory to a specific value. The `os_memcpy()` will copy a block of memory to a different block. The `os_bzero()` function will set the values of a block of memory to zero.

Memory on the ESP8266 is made up of a number of components. We have:

- data
- rodata
- bss
- heap

The values of these can be found through the `system_print_meminfo()` function.

When the ESP8266 needs to read an instruction from memory in order to execute it, that instruction can come from one of two places. The instruction can in flash memory (also called `irom`) or it can be in RAM (also called `iram`). It takes less time for the processor to retrieve the instruction from RAM than it does from flash. It is believed that an instruction fetch from flash takes four times longer than the same instruction fetched from RAM. However, on the ESP8266 there is far less RAM than there is flash. What this means is that you are far more likely to run out of RAM way before you run out of flash. When writing normal applications, we shouldn't fixate on having instructions in RAM rather than flash for the performance benefit. The execution speeds of the ESP8266 are so fast that if the cost of retrieving an instruction from RAM is blindingly fast then retrieving an instruction from slower flash is **still** blindingly fast.

There are however certain classes of instructions that we might wish to place in RAM rather than flash. Examples of these are interrupt handlers where the time spent in these should always be as fast as possible and also function that write to flash.

When we define C functions, we can add an attribute by the name of `ICACHE_FLASH_ATTR`. What this does is place this function in the flash memory address space as opposed to RAM. Specifically, flagging a function with `ICACHE_FLASH_ATTR` tags it as being in the "`.text`" section of code.

See also:

- os_memset

- os_memcpy
- os_memcmp
- os_malloc
- os_zalloc
- os_free
- system_get_free_heap_size
- system_print_meminfo

Pulse Width Modulation – PWM

The idea behind pulse width modulation is that we can think of regular pulses of output signals encoding information as a function of how long the signal is kept high. Let us imagine that we have a period of 1HZ (one thing per second). Now let us assume that we raise the output voltage to a level of 1 for ½ of a second at the start of the period. This would give us a square wave which starts high, lasts for 500 milliseconds and then drops low for the next 500 milliseconds. This repeats on into the future. The duration that the pulse is high relative to the period allows us to encode an analog value onto digital signals. If the pulse is 100% high for the period then the encoded value would be 1.0. If the pulse is 100% low for the period, then the encoded value would be 0.0. If the pulse is on for "n" milliseconds (where n is less than 1000), then the encoded value would be n/1000.

Typically, the length of a period is not a second but much, much smaller allowing us to output many differing values very quickly. The ratio of the on signal to the period is called the "duty cycle". This encoding technique is called "Pulse Width Modulation" or "PWM".

There are a variety of purposes for PWM. Some are output data encoders. One commonly seen purpose is to control the brightness of an LED. If we apply maximum voltage to an LED, it is maximally bright. If we apply ½ the voltage, it is about ½ the brightness. By applying a fast period PWM signal to the input of an LED, the duty cycle becomes the brightness of the LED. The way this works is that either full voltage or no voltage is applied to the LED but because the period is so short, the "average" voltage over time follows the duty cycle and even though the LED is flickering on or off, it is so fast that our eyes can't detect it and all we see is the apparent brightness change.

For the ESP8266, the period of the PWM can range from 1 millisecond to 10 milliseconds. This is a frequency of 1KHz to 100Hz. The resolution of the duty cycle is down to 45 nanoseconds which is 14 bits of resolution data. The device provides support for up to 8 PWM channels where each channel can be associated with its own pin and duty cycle. The period is the same for all PWM channels.

To start using the ESP8266 PWM support, a call to `pwm_init()` is needed which sets up which pins are to be used for PWM and for which channels. A call to this function also sets up an initial period and duty cycle. A call to `pwm_start()` can then be made to start the PWM outputs. The period of PWM as a whole and duty cycles for each channel can be changed using the `pwm_set_period()` and `pwm_set_duty()` functions.

See also:

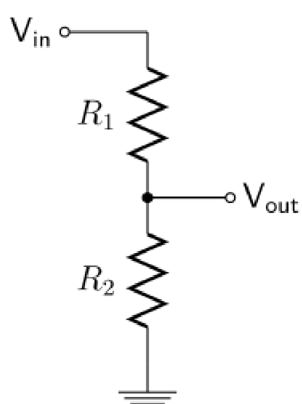
- Wikipedia: [Pulse-width modulation]
- pwm_init
- pwm_start
  pwm_set_duty
- pwm_get_duty
- pwm_set_period
- pwm_get_period

Analog to digital conversion

Analog to digital conversion is the ability to read a voltage level from a pin between 0 and some maximum value and convert that analog voltage into a digital representation. Varying the voltage applied to the pin will change the value read. The ESP8266 has an analog to digital converter built into it with a resolution of 1024 distinct values. What that means is that 0 volts will produce a digital value of 0 while the maximum voltage will produce a digital value of 1023 and voltage ranges between these will produce a correspondingly scaled digital value.

To read the digital value of the analog voltage, the function called `system_adc_read()` should be called. The pin on the physical ESP8266 from which the voltage is read is called `TOUT` and serves no other purpose.

The input range on the pin is from 0V to 1V. This implies that the input voltage to the ADC can not be the maximum voltage used to power the ESP8266 itself (3.3V). So we will need to use a voltage divider circuit.



The formula to map these out is:

$$Vout = \frac{R2}{R1+R2} \cdot Vin$$

Since we know `Vout` is going to 1V and `Vin` is 3V and we choose R2 to be 10K, we find:

$$R1 = \frac{R2 \cdot Vin}{Vout} - R2$$

and for our values:

$$R1 = \frac{10000 * 3.3}{1.0} - 10000 = 23000$$

A common 22K resistor will work well.

Here is an example. What this example does is print the value read from the ADC every second.

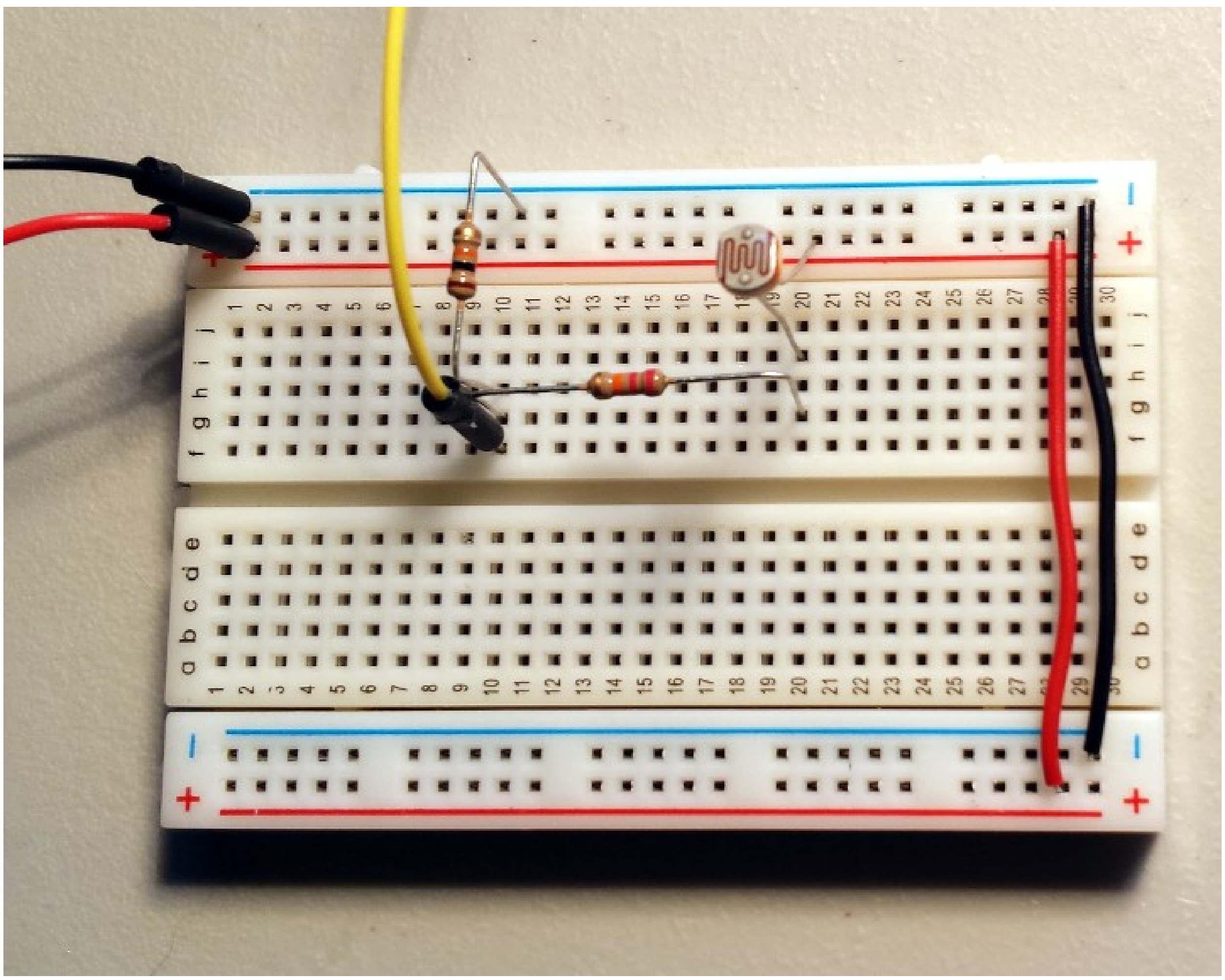
```
os_timer_t myTimer;

void timerCallback(void *pArg) {
	uint16 adcValue = system_adc_read();
	os_printf("adc = %d\n", adcValue);

} // End of timerCallback

void user_init(void) {
	uart_init(BIT_RATE_115200, BIT_RATE_115200);
	os_timer_setfn(&myTimer, timerCallback, NULL);
	os_timer_arm(&myTimer, 1000, 1);
} // End of user_init
```

If we build out on a breadboard a circuit which includes a light dependent resistor such as the following:



Then when we change the amount of light falling on the resistor, we can see the values change as data is written in the output log. This can be used to trigger an action (for example) when it becomes dark.

Open question: What is the sample rate of the ADC?

See also:

- system_adc_read
- Wikipedia: Voltage divider

Watchdog timer

The ESP8266 is a single threaded processor. This means it can only do one thing at a time as there are no parallel threads that can be executed. An implication of this is that when the OS gives control to your application, it doesn't get control back until you explicitly relinquish it. However, this can cause problems. The ESP8266 is primarily a WiFi and TCP/IP device that expects to be able to receive and transmit data as well as respond to asynchronous events within a timely manner. As an example, if your ESP8266 device is connected to an access point and the access point wants to validate that you are still connected, it may transmit a packet to you and expect a response. You have no control over when that will happen. If your own application program has control over the execution at the time when the request arrives, that request will not be responded to until after you return control back to the OS. Meanwhile, the access point may be expecting a response within some predetermined time period and, if does not receive a reply within that interval, may assume that you have disconnected. This means that your application code has to return control back to the OS in a timely manner. It is recommended that your code return control within 50 milliseconds of gaining control. If you take longer, you run the risk of requests to your device timing out.

If your own code fails to return control back to the OS, the OS must assume that things are going wrong. As such, it has a timer that we call the "watchdog". When control is given to your own code, the watchdog timer starts ticking. If you have not returned control back to the OS by the time the watchdog timer reaches zero, it takes matters into its own hands. Explicitly what it does is reboot the device. This may sound like a pretty drastic action but the thinking is that it is better to do this and hope that whatever was blocked is now unblocked than just sit there "dead".

Reports claim that the watchdog timer may be about 1 second (1000 milliseconds). However, in my tests, I find that the timer fires at about 3.2 seconds (3200 msecs).

A function called `system_soft_wdt_stop()` stops the watchdog timer … or at least one of them. There appears to be **two** timers. One is in software, the other in hardware. This function stops the software timer. It can be restarted with `system_soft_wdt_restart()` … however, a second timer called the hardware watchdog timer will fire after about 8 seconds and doesn't appear able to be trapped.

See also:

- system_soft_wdt_stop
- system_soft_wdt_restart

Mapping from Arduino

Without argument, the Arduino has become the most successful microprocessor programming environment to-date. There are tons and tons of existing sketches in existence and let us not forget about the wealth of libraries. Tools and utilities exist to compile and run Arduino sketches on ESP8266s. What if instead we wanted to port those Arduino sketches to native ESP8266 code? Can we find mappings between the Arduino APIs and the corresponding ESP8266 APIs?

Arduino	ESP8266
digitalWrite(pin, value)	GPIO_OUTPUT_SET(pin, value)
digitalRead(pin)	GPIO_INPUT_GET(pin)
delay(ms)	os_delay_us(ms * 1000) Note: ms <= 65535
delayMicroseconds(us)	os_delay_us(us)
millis()	system_get_time() / 1000

From a functional perspective, here are some comparisons between an Arduino and an ESP8266:

	ESP8266	Arduino (Uno)
GPIOs	17 (Fewer typically exposed)	14 (20 including analog)
Analog input	1	6
PWM channels	8	6
Clock speed	80MHz	16MHz
Processor	Tensilica	Atmel
SRAM	45KBytes	2KBytes
Flash	512Kb or more (separate)	32KB (on chip)
Operating Voltage	3.3V	5V
Max current per I/O	12mA	40mA
UART (hardware)	1 ½	1
Networking	Built-in	Separate
Documentation	Poor	Excellent
Maturity	Early	Mature

Note: Because the Arduino has no native networking, no further comparisons of network capability were included above. Do remember that, at this time, when one is using an ESP8266, the chances are high it is because you **need** network access.

Partner TCP/IP APIs

If the ESP8266 can act as one end of a TCP/IP connection, something else has to act as the other (of course, there is nothing to prevent two ESP8266s from communicating between themselves). Here we look into some technologies that allow partners to interact with the ESP8266 over the TCP/IP protocol.

For the TCP/IP protocol, the programming API originally developed for the Unix platform and written in C was called "sockets". The notion of a socket is that it logically represents an endpoint of a network connection. A sender of data sends data through the socket and the receiver of data receives data through the socket. The implementation of the "socket" itself is provided by the libraries but the logical notion of the socket remains. You will find yourself working with an "instance" of a socket and you should think of it as an opaque data type that refers to a communication link.

Sockets remains the primary API and is present in the majority of languages. Here we discuss some of the variants for some of the more common languages.

Java Sockets

The sockets API is the defacto standard API for programming against TCP/IP. My programming language of choice is Java and it has full support for sockets. What this means is that I can write a Java based application that leverages sockets to communication with the ESP8266. I can send and receive data through quite easily.

In Java, there are two primary classes that represents sockets, those are `java.net.Socket` which represents a client application which will form a connection and the second class is `java.net.ServerSocket` which represents a server that is listening on a socket awaiting a client connection. Since the ESP8266 can be either a client or a server, both of these Java classes will come into play.

To connect to an ESP8266 running as a server, we need to know the IP address of the device and the port number on which it is listening. Once we know those, we can create an instance of the Java client with:

```
Socket clientSocket = new Socket(ipAddress, port);
```

This will form a connection to the ESP8266. Now we can ask for both an `InputStream` from which to receive partner data and an `OutputStream` to which we can write data.

```
InputStream is = clientSocket.getInputStream();
OutputStream os = clientSocket.getOutputStream();
```

When we are finished with the connection, we should call `close()` to close the Java side of the connection:

```
clientSocket.close();
```

It really is as simple as that. Here is an example application:

```
package kolban;

import java.io.OutputStream;
import java.net.Socket;

import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.DefaultParser;
import org.apache.commons.cli.Options;

public class SocketClient {
	private String hostname;
	private int port;

	public static void main(String[] args) {
		Options options = new Options();
		options.addOption("h", true, "hostname");
		options.addOption("p", true, "port");
		CommandLineParser parser = new DefaultParser();
		try {
			CommandLine cmd = parser.parse(options, args);

			SocketClient client = new SocketClient();
			client.hostname = cmd.getOptionValue("h");
			client.port = Integer.parseInt(cmd.getOptionValue("p"));
			client.run();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void run() {
		try {
			int SIZE = 65000;
			byte data[] = new byte[SIZE];
			for (int i = 0; i < SIZE; i++) {
				data[i] = 'X';
			}
			Socket s1 = new Socket(hostname, port);
			OutputStream os = s1.getOutputStream();
			os.write(data);
			s1.close();
			System.out.println("Data sent!");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
} // End of class
// End of file
```

To configure a Java application as a socket server is just as easy. This time we create an instance of the `SocketServer` class using:

```
SocketServer serverSocket = new SocketServer(port)
```

The port supplied is the port number on the machine on which the JVM is running that will be the endpoint of remote client connection requests. Once we have a `ServerSocket` instance, we need to wait for an incoming client connection. We do this using the blocking API method called `accept()`.

```
Socket partnerSocket = serverSocket.accept();
```

This call blocks until a client connect arrives. The returned `partnerSocket` is the connected socket to the partner which can used in the same fashion as we previously discussed for client connections. This means that we can request the `InputStream` and `OutputStream` objects to read and write to and from the partner. Since Java is a multi-threaded language, once we wake up from `accept()` we can pass off the received partner socket to a new thread and repeat the `accept()` call for other parallel connections. Remember to `close()` any partner socket connections you receive when you are done with them.

So far, we have been talking about TCP oriented connections where once a connection is opened it stays open until closed during which time either end can send or receive independently from the other. Now we look at datagrams that use the UDP protocol.

The core class behind this is called `DatagramSocket`. Unlike TCP, the `DatagramSocket` class is used both for clients and servers.

First, let us look at a client. If we wish to write a Java UDP client, we will create an instance of a `DatagramSocket` using:

```
DatagramSocket clientSocket = new DatagramSocket();
```

Next we will "connect" to the remote UDP partner. We will need to know the IP address and port that the partner is listening upon. Although the API is called "connect", we need to realize that no connection is formed. Datagrams are connectionless so what we are actually doing is associating our client socket with the partner socket on the other end so that **when** we actually wish to send data, we will know where to send it to.

```
clientSocket.connect(ipAddress, port);
```

Now we are ready to send a datagram using the send() method:

```
DatagramPacket data = new DatagramPacket(new byte[100], 100);
clientSocket.send(data);
```

To write a UDP listener that listens for incoming datagrams, we can use the following:

```
DatagramSocket serverSocket = new DatagramSocket(port);
```

The port here is the port number on the same machine as the JVM that will be used to listen for incoming UDP connections.

To wait for an incoming datagram, call `receive()`.

```
DatagramPacket data = new DatagramPacket(new byte[100], 100);
clientSocket.receive(data);
```

If you are going to use the Java Socket APIs, read the JavaDoc thoroughly for these classes are there are many features and options that were not listed here.

See also:

- [Java tutorial: All About Sockets](#)
- [JDK 8 JavaDoc](#)

WebSockets

WebSockets is both an API and a protocol introduced in HTML5. Simply put, if we imagine an HTTP server sitting waiting for incoming HTTP requests, we can convert a current request into a socket connection between the server and the browser such that either end can send data to be received by its partner.

Programming using Eclipse

Eclipse is a popular open source framework primarily used for hosting application development tools. Although primarily geared for building Java applications, it also has first class C and C++ support.

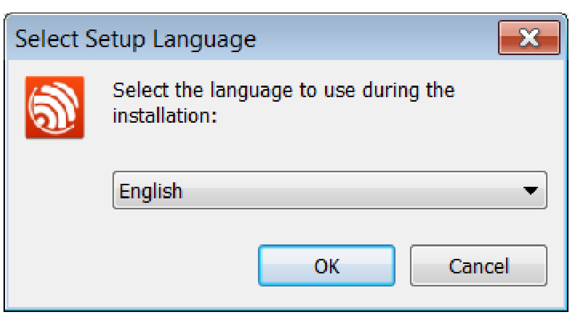
A project for building ESP8266 applications using Eclipse can be found here:

http://www.esp8266.com/viewtopic.php?f=9&t=820

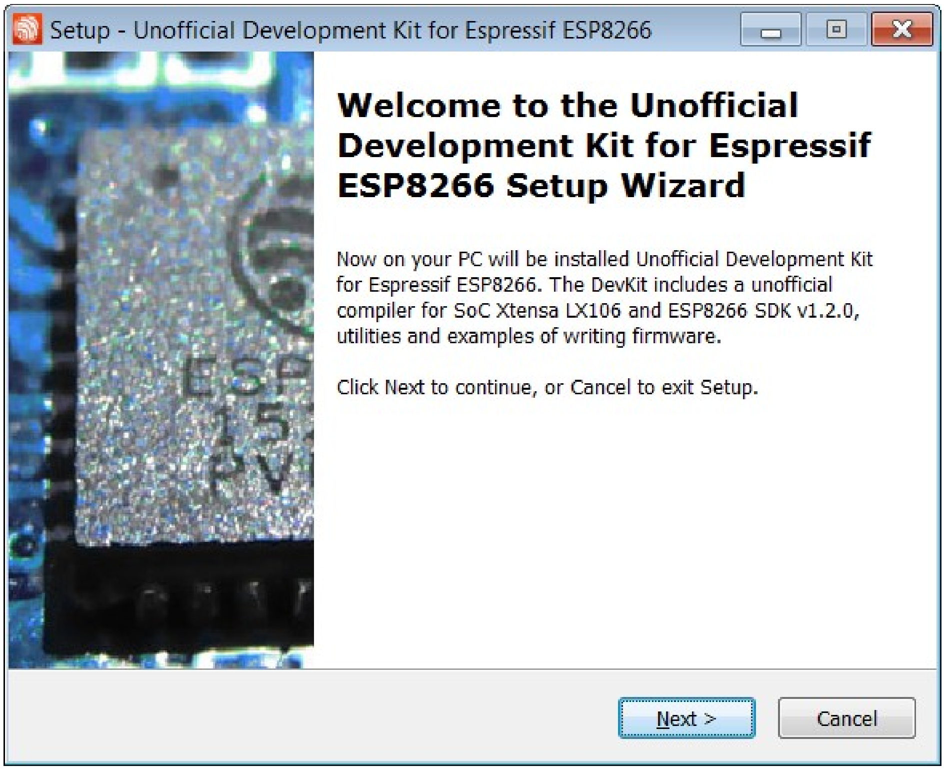
Do not include spaces in any of the path parts pointing to the workspace. Here are some notes on installing this project … however, always read the documentation accompanying the project.

Download the `Espressif-ESP8266-DevKit-vxxx-x86`. This is a large download of approx 125MBytes.

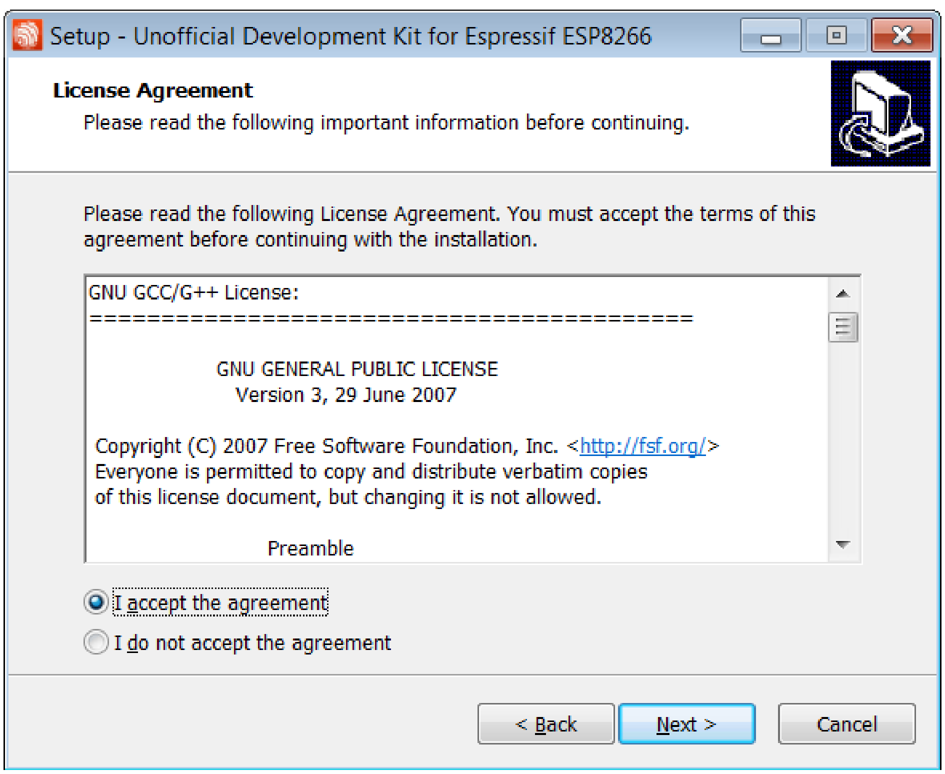
Run the installer. It will ask you for your choice of installation language.



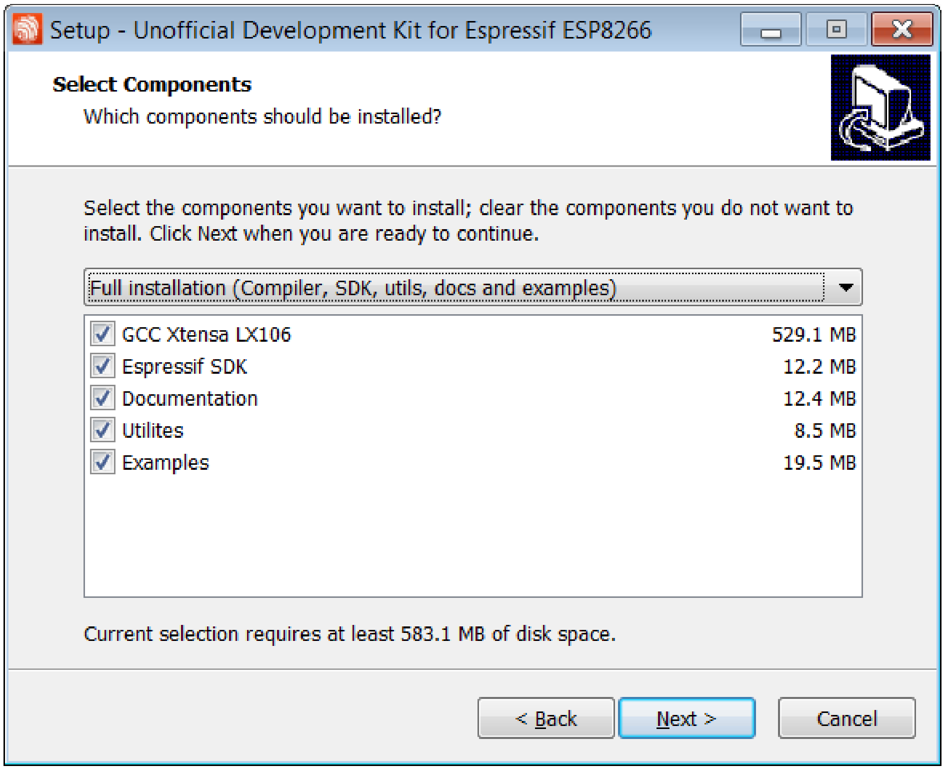
Next comes the splash screen:



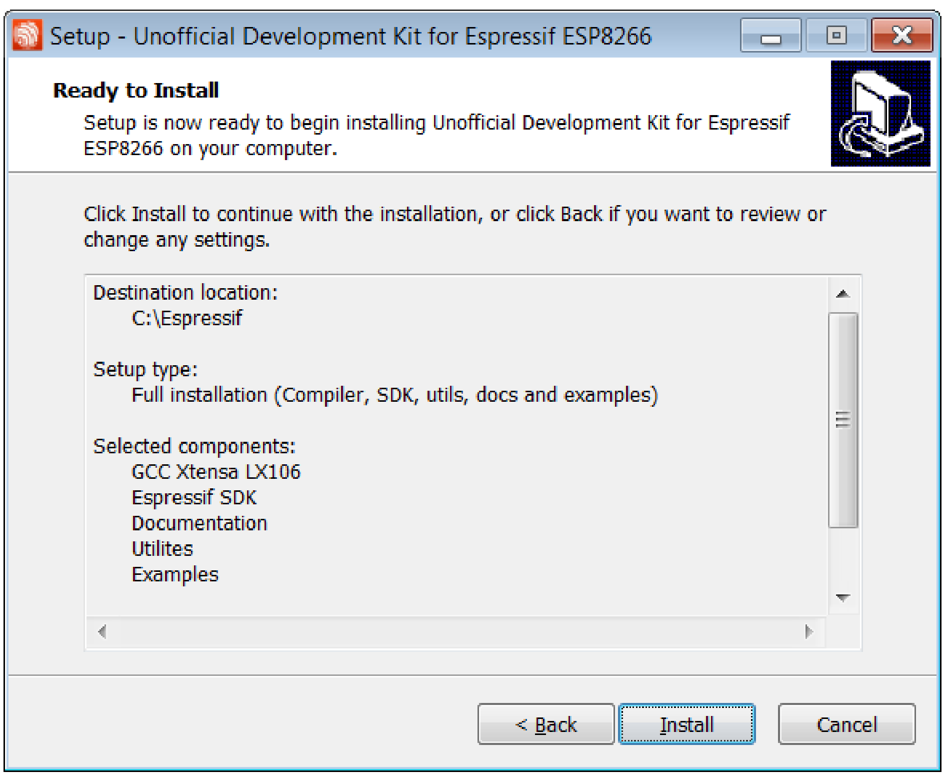
Next comes the license agreement:



Now the selection of which components to install:



Finally a confirmation dialog to review what you have selected.



The result of this will be a new directory structure at `C:\Espressif\`.

There are other dependencies that you will need which are listed at the link above. These include:

- A Java runtime environment. I use the latest Java 8 from Oracle.
- Eclipse environment with C/C++ developer tools. I use the latest "Mars" release.
- `MinGW` – Unix tools and utilities that execute on Windows.
- `MinGW installation helper` – A cache and list of the MinGW packages that need to be installed for correct operation.

The Makefiles supplied with the package are key. They have been crafted to provide the easiest compiles. The targets contained within the Makefiles include:

- all – Compile all the code but do not flash.
- clean – Clean any previous builds.
- flash – Compile the code if needed and then flash.
- flashboot
- flashinit
- flashonefile

There are some flags that are used with the Makefile that you can edit. These include:

- `VERBOSE=1` – Enable verbosity which includes debug information. Specifically the compilation commands are shown.

See also:

- Eclipse.org
- Eclipse C/C++ Development Tooling (CDT)
- Primary forum thread

Installing the Eclipse Serial terminal

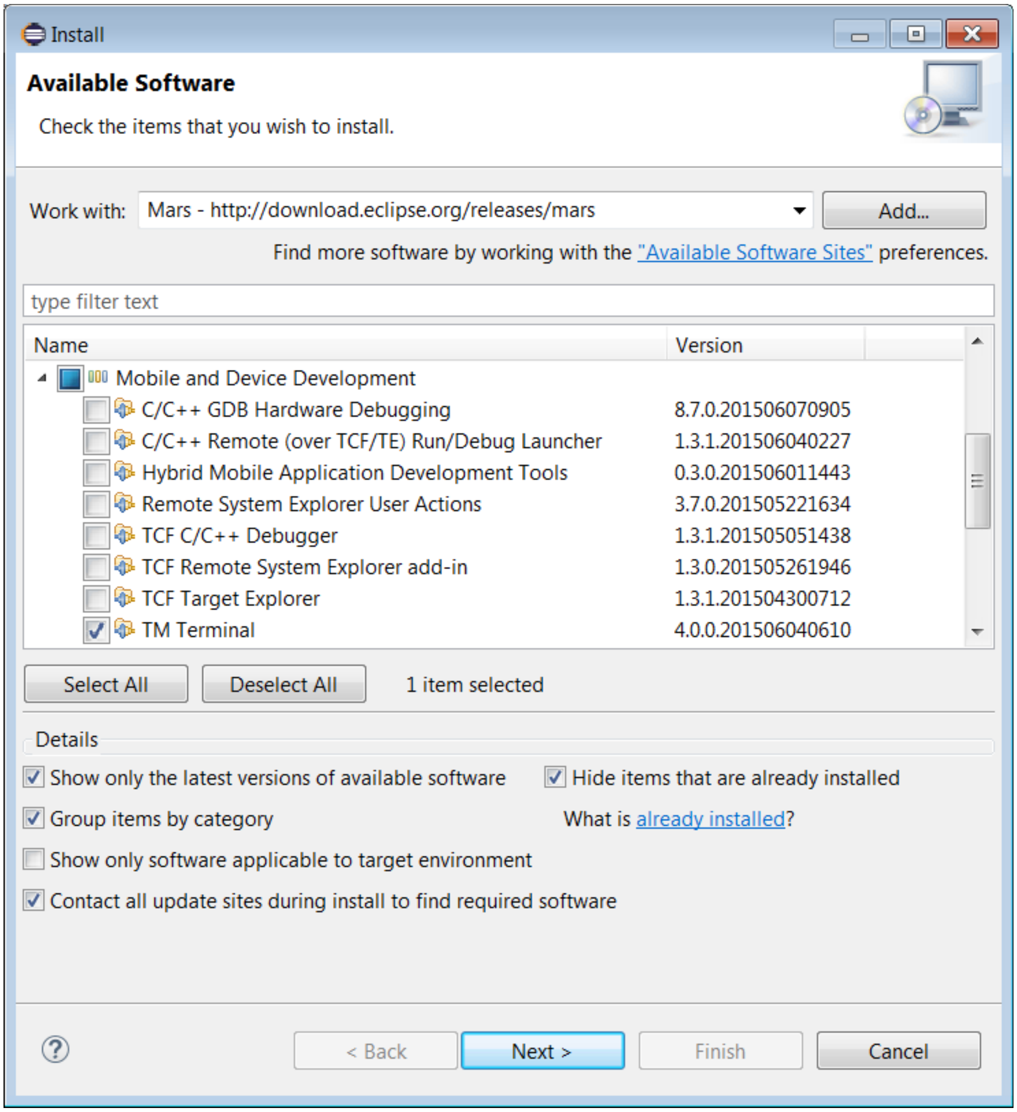
Although there are many excellent serial terminals available as stand-alone Windows applications, an alternative is the Eclipse Terminal which also has serial support. This allows a serial terminal to appear as a view within the Eclipse IDE. It does not come installed by default but the steps to add are not complex.

First start Eclipse (I use the Mars release).

Go to `Help > Install new software`.

Select the eclipse download repository.

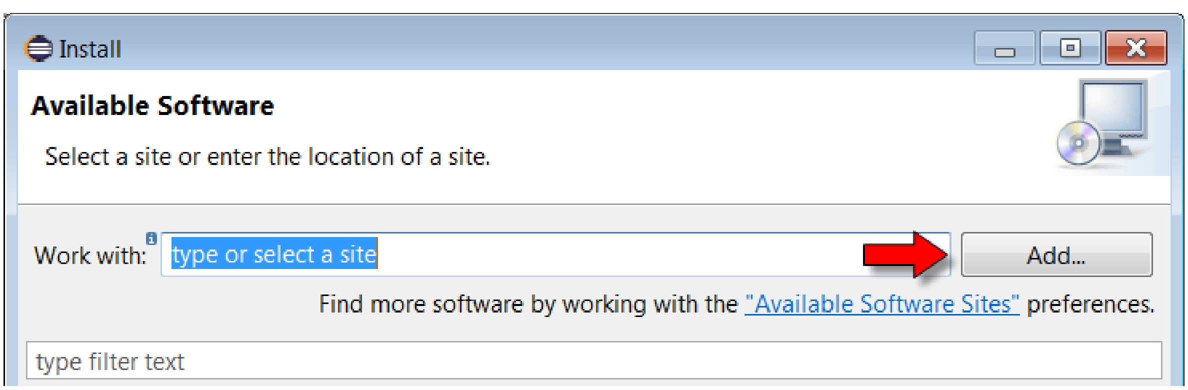
Select `Mobile and Device development > TM Terminal`.



Step through the following sections and when prompted to restart, accept yes.

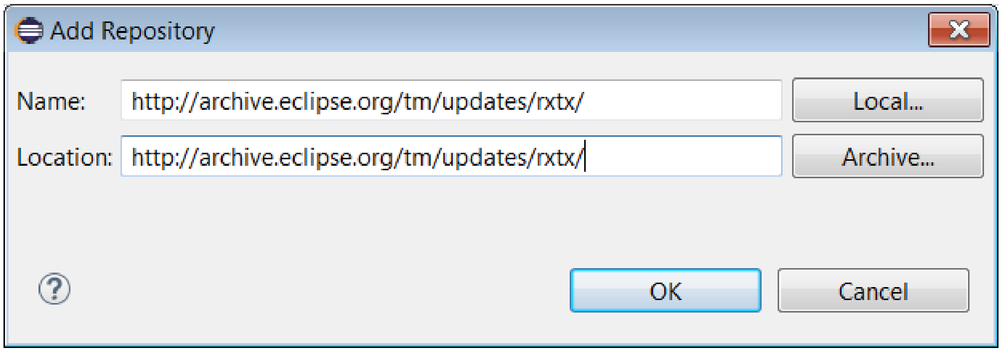
We are not ready to use it yet, we must add serial port support into Eclipse.

Go back to `Help > Install new software` and add a new repository

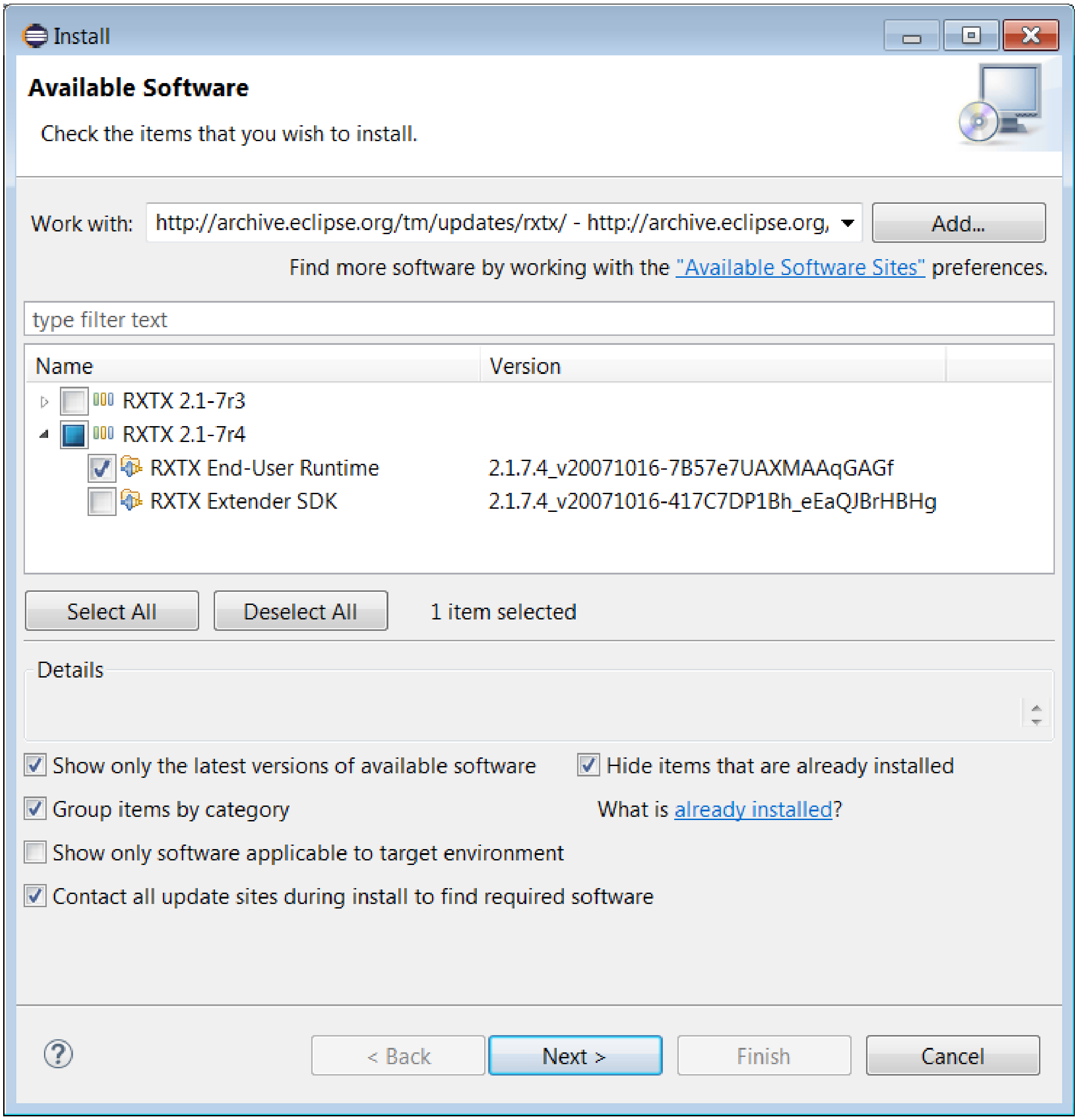


The repository URL is:

http://archive.eclipse.org/tm/updates/rxtx/

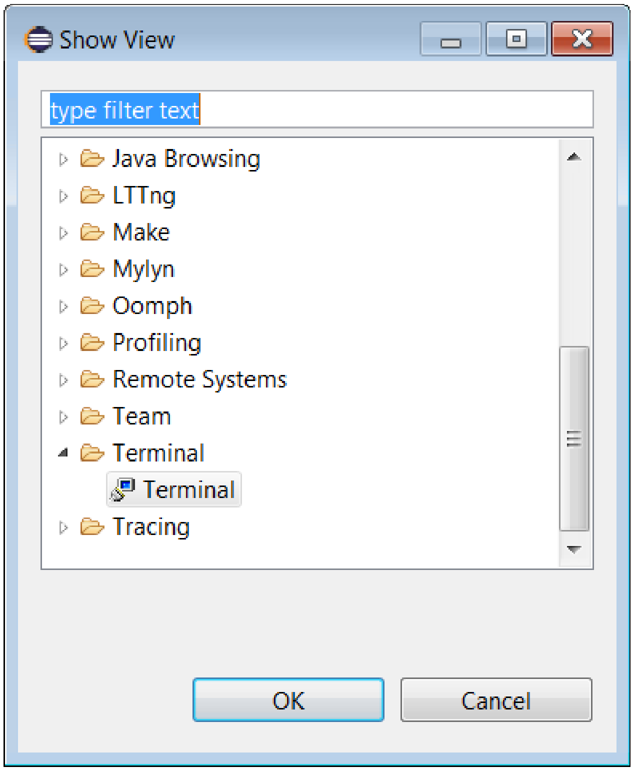


Now we can select the Serial port runtime support library:

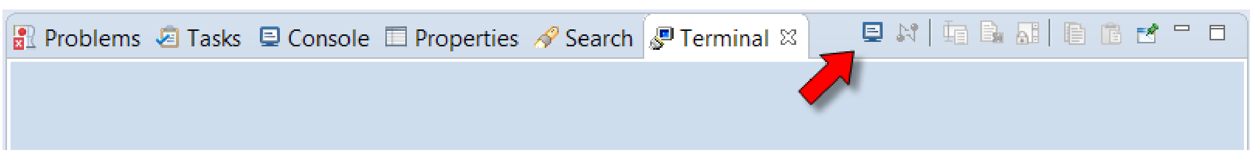


Follow through the further navigation screens and restart Eclipse when prompted.

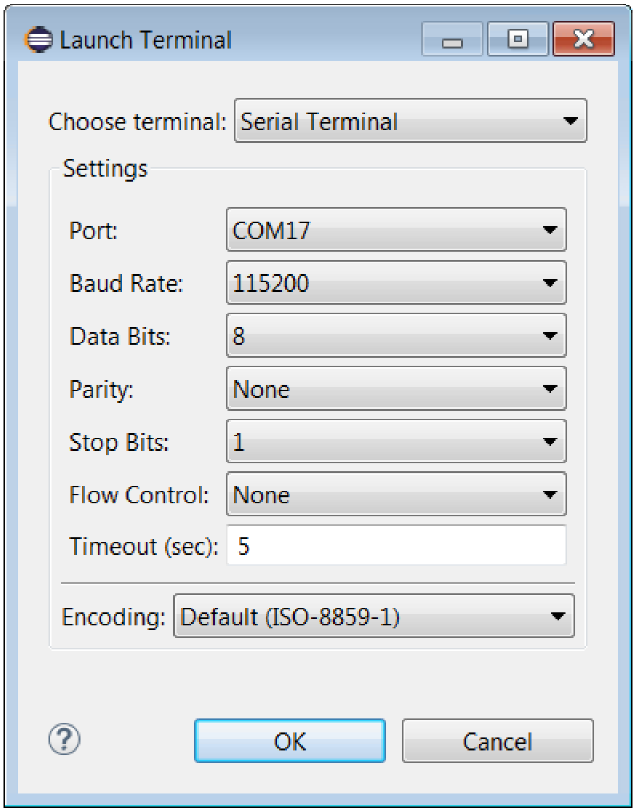
We now have terminal support installed and are ready to use it. From `Windows > Show View > Other` we will find a new category called "`Terminal`".



Opening this adds a Terminal view to our perspective. There is a button that will allow us to open a new terminal instance that is shown in the following image:

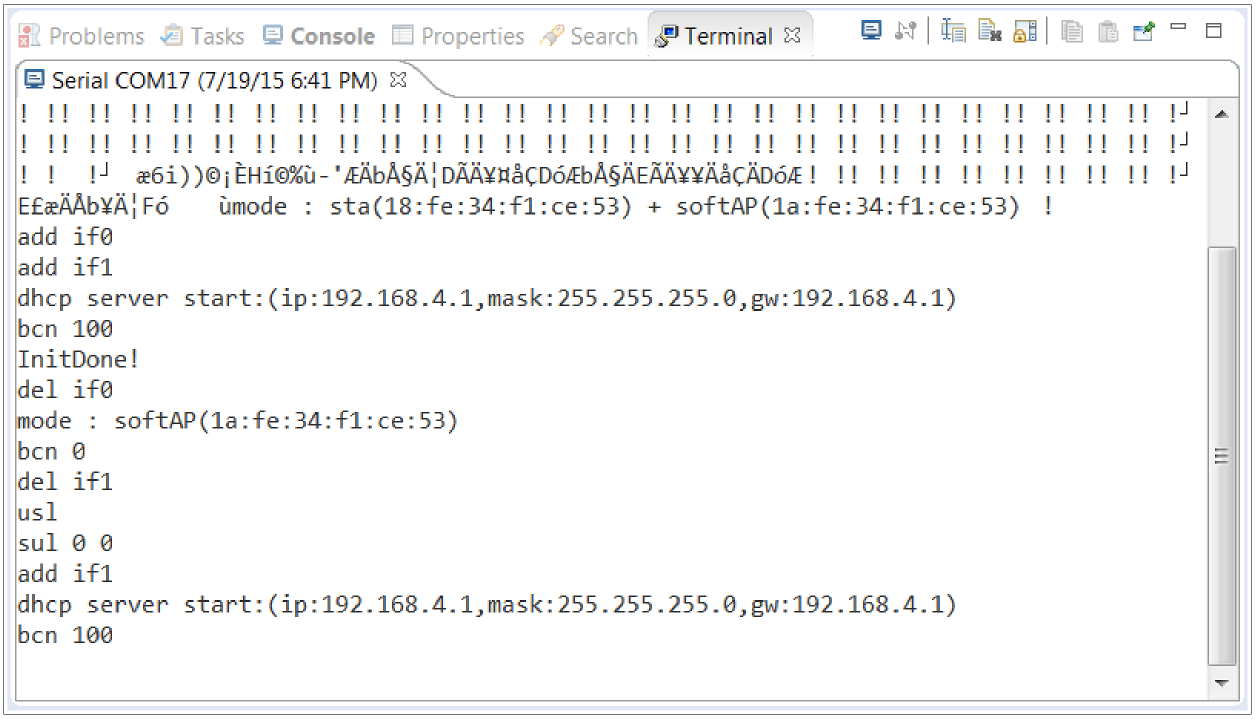


Clicking this brings up the dialog asking us for the type of terminal and the properties. For our purposes, we wish to choose a serial terminal. Don't forget to also set the port and baud rate to match what your ESP8266 uses.



After clicking OK, after a few seconds we will see that we are connected and a new disconnect icon appears:

And now the terminal is active. For my purposes, I connect this terminal to UART1 of the ESP8266 for debugging while leaving UART0 for flashing new copies of my application. Here is an example of what my typical window looks like:



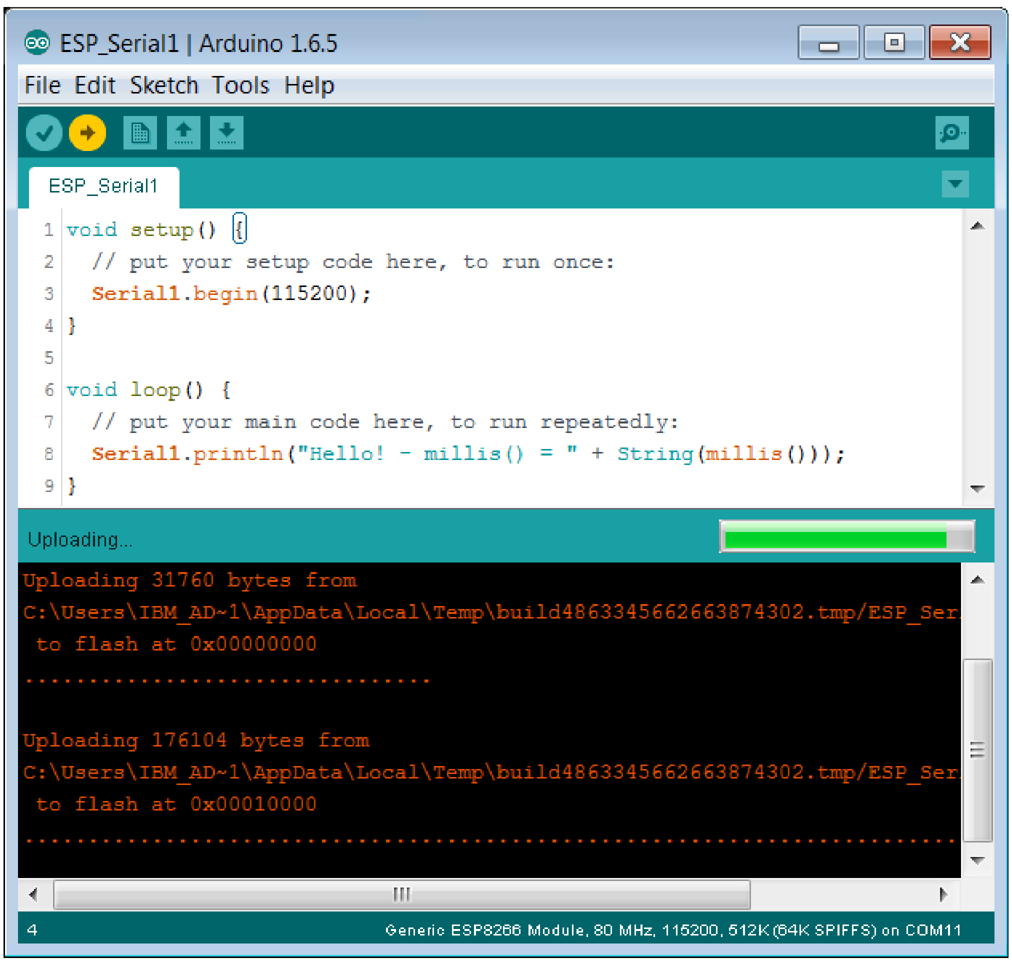
You can invert the colors to produce a white on black visualization which many users prefer.

Programming using the Arduino IDE

Long before there was an ESP8266, there was the Arduino. A vitally important contribution to the open source hardware community and the entry point for the majority of hobbyists into the world of home built circuits and processors.

One of the key attractions about the Arduino is its relative low complexity allowing someone to build something quickly and easily. The Integrated Development Environment (IDE) for the Arduino has always been free of charge for download. If a professional programmer were to sit down with it, they would be shocked at its apparent limited capabilities. However, the subset of function it provides compared to a "full featured" IDE happen to cover 90% of what one wants to achieve. Combine that with the intuitive interface and the Arduino IDE is a force to be reckoned with.

Here is what a simple program looks like in the Arduino IDE:



In Arduino parlance, an application is termed a "sketch". Personally, I'm not a fan of that phrase but I'm sure research was done to learn that this is the least intimidating name for a C language program that would scare the least number of people.

The IDE has a button called "`Verify`" which, when clicked, compiles the program. Of course, this will also have the side-effect that it will verify that the program compiles cleanly … but compilation is what it does. A second button is called "`Upload`" and, when clicked, what it does is deploy the application to the Arduino.

In addition to providing a C language editor plus tools to compile and deploy, the Arduino IDE provides pre-supplied libraries of C routines that "hide" complex implementation details that might be needed when programming to the Arduino boards. For example, UART programming would undoubtedly have to set registers, handle interrupts and more. Instead of making the poor users have to learn these technical APIs. the Arduino folks provided high level libraries that could be called from the sketches with cleaner interfaces which hide the mechanical "gorp" that happens under the covers. This notion is key … as these libraries, as much as anything else, provide the environment for Arduino programmers.

Interesting as this story may be, you may be asking how this relates to our ESP8266 story? Well, a bunch of talented individuals have built out an Open Source project on Github that provides a "plugin" or "extension" to the Arduino IDE tool (remember, that the Arduino IDE is itself free). What this extension does is allow one to write sketches in the Arduino IDE that leverage the Arduino library interfaces which, at compile and deployment time, generate code that will run on the ESP8266. What this effectively means is that we can use the Arduino IDE and build ESP8266 applications with the minimum of fuss.

Implications of Arduino IDE support

The ESP8266 is still new (as of July 2015) and no-one knows where this little chip will be in a year or five years time. Will it become the heart and soul of a new range of hobbyist boards and professional appliances? Will there be something newer and better just around the next corner? We simply don't know.

The ability to treat it as though it were "like" an Arduino is a notion that I haven't been able to fully absorb yet. ESP8266 is a Tensilica CPU unlike the Arduino which is an ATmega CPU. Espressif have created dedicated and architected API in the form of their SDK for directly exposed ESP8266 APIs. The Arduino libraries for ESP8266 seem to map their intent to these exposed APIs. For these reasons and similar, one might argue that the Arduino support is an unnecessary facade on top of a perfectly good environment and by imposing an "alien" technology model on top of the ESP8266 native functions, we are masking access to lower levels of knowledge and function. Further, thinking of the ESP8266 as though it were an Arduino can lead to design problems. For example, the ESP8266 needs regular control in order to handle WiFi and other internal actions. This conflicts with the Arduino model where the programmer can do what he wants within the loop function for as long as he wants.

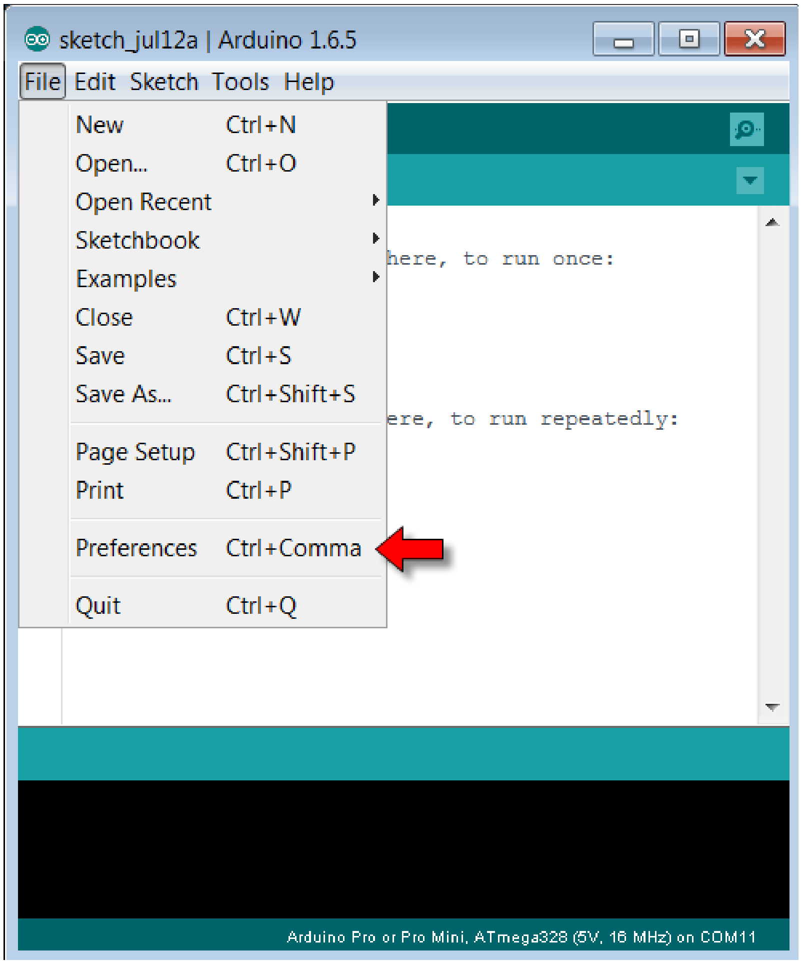
The flip side is that the learning curve to get something running on an Arduino has been shown to be extremely low. It doesn't take long at all to get a blinky light going on a breadboard. With that train of thought, why should users of the ESP8266 be penalized for having to install and learn more complex tool chains and syntax to achieve the same result with more ESP8266 oriented tools and techniques? The name of the game should be to allow folks to tinker with CPUs and sensors without having to have university degrees in computing science or electrical engineering and if the price one pays to get there is to insert a "simple to use" illusion then why not? If I build a paper airplane and throw it out my window … I may get pleasure from that. A NASA rocket scientist shouldn't scoff at my activities or lack of knowledge of aerodynamics … the folded paper did its job and I achieved my goal. However, if my job was to put a man on the moon, the ability to visualize the realities of the technology at the "realistic" level becomes extremely important.

Installing the Arduino IDE with ESP8266 support

To assemble this environment, one must download a current version of the Arduino IDE. This will be about 140 Mbytes.

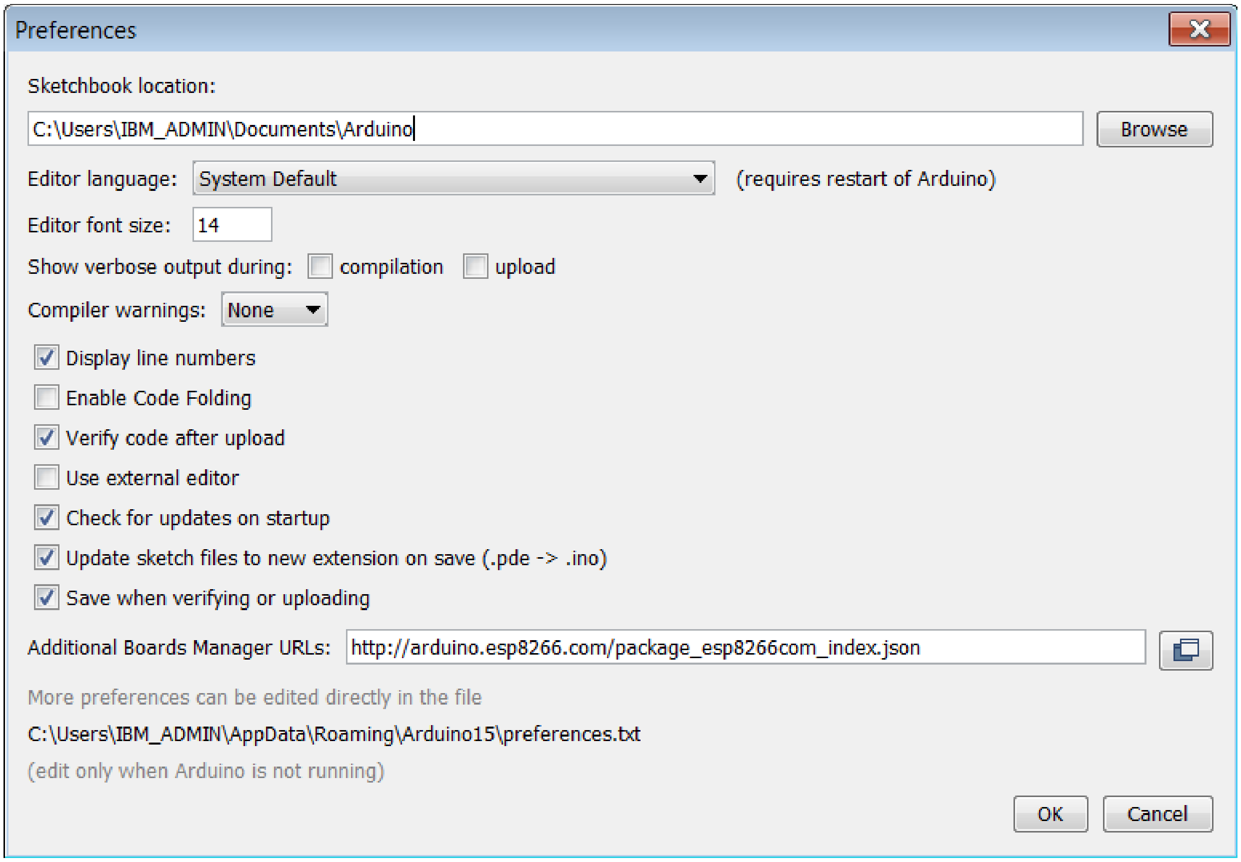
I download the ZIP file version and then extract its content.

Next, we launch the Arduino IDE and open the `Preferences` dialog:

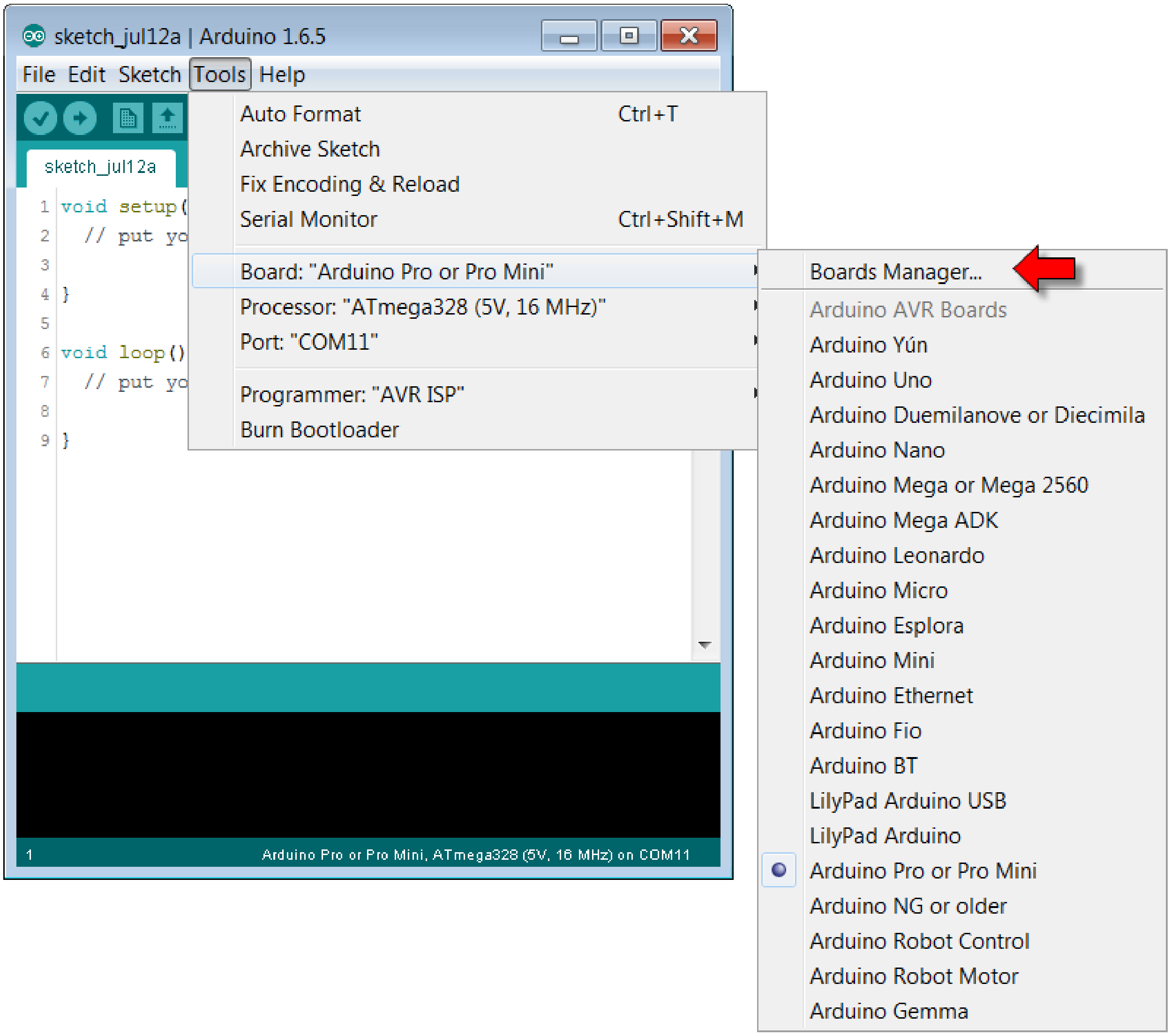


In the `Additional Boards Manager URLs` field enter the URL for the ESP8266 package which is:

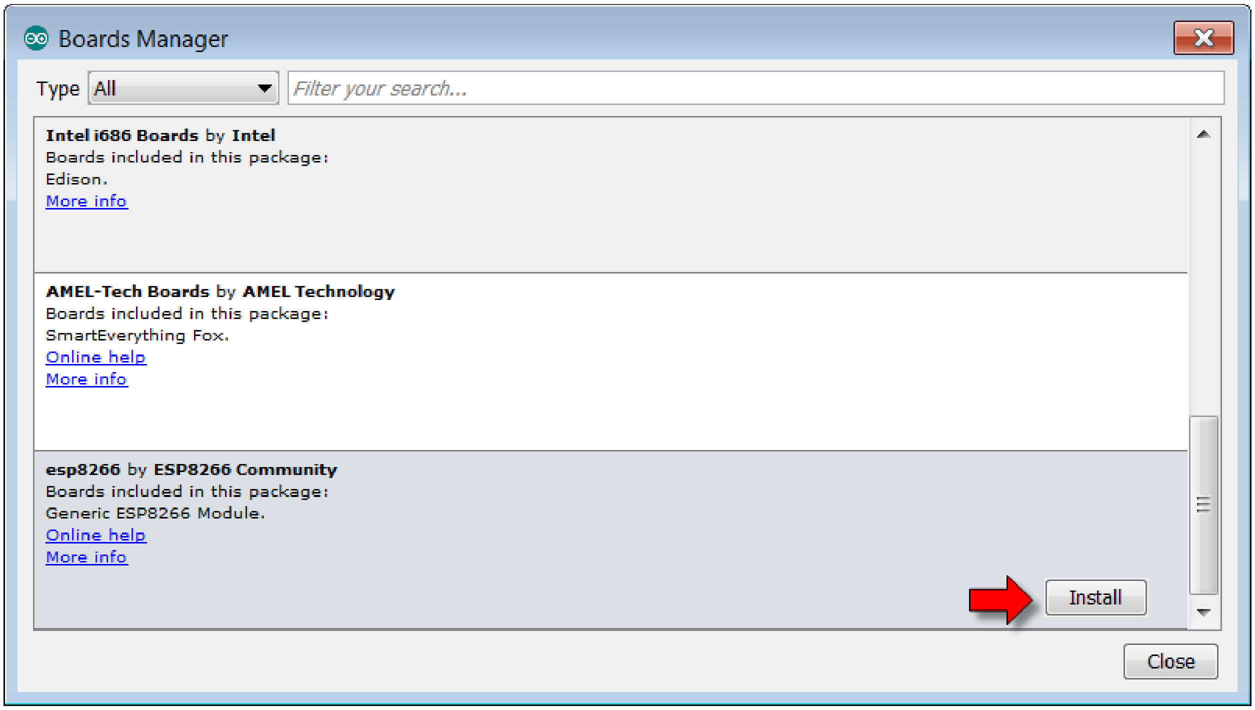
`http://arduino.esp8266.com/package_esp8266com_index.json`



Select the Boards Manager from the `Tools > Board` menu:

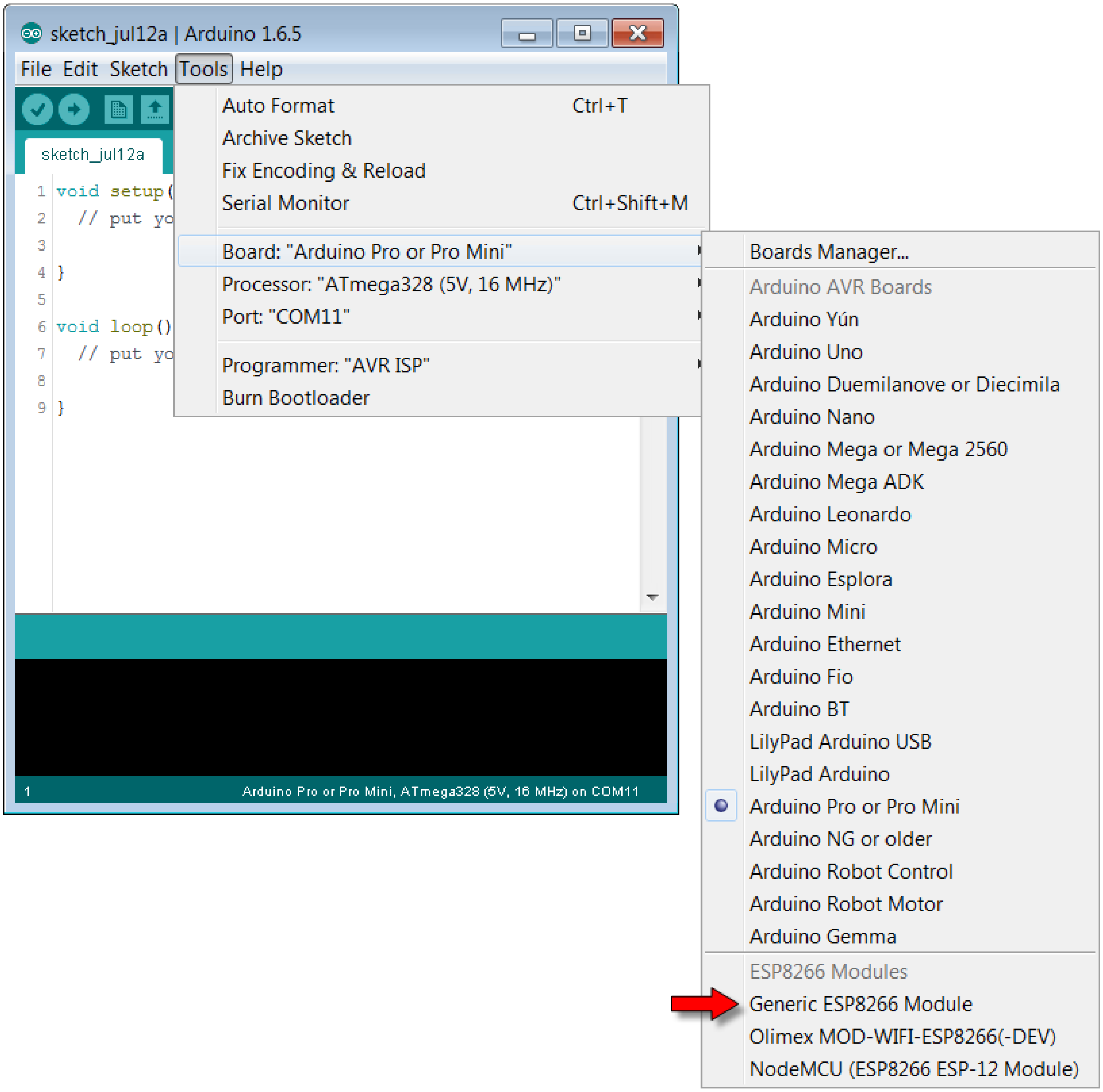


Install the ESP8266 support:



This will contact the Internet and download the artifacts necessary for ESP8266 support.

Once completed, in the Arduino IDE Board selections, you will find the "`Generic ESP8266 Module`":



Now we are ready to start building, compiling and running sketches.

A simple and sample sketch I recommend for testing is:

```
void setup() {
  Serial1.begin(115200);
}

void loop() {
  Serial1.println("Hello! - millis() = " + String(millis()));
}
```

When run, a loop of messages will appear on the UART1 output saying hello and the number of milliseconds since last boot. As much as anything, this will validate that the environment has been setup correctly, you can compile a program and that deployment to the ESP8266 is successful.

See also:

- Github: esp8266/Arduino
- Arduino IDE

The Arduino IDE ESP8266 Libraries

There is no question our language is going to get odd here. We are using the Arduino IDE with Arduino API libraries to compile and deploy to an ESP8266. More than likely, there isn't an actual Arduino device in sight here but yet the word Arduino keeps being used. Take care to understand in your mind that we are piggybacking on a technology which has become as much a philosophy as physical implementation. Maybe I'm showing my age here, but there are times when I still think to myself … "Hmm … my carpet needs cleaning, I think I'll hoover it". I don't think I've owned a Hoover brand vacuum cleaner in decades … but the nomenclature has become ingrained.

The WiFi library

The Arduino has a WiFi library for use with its WiFi shield. A library with a similar interface has been supplied for the Arduino environment for the ESP8266.

To use the ESP8266 WiFi library you must include its header:

```
#include <ESP8266WiFi.h>
```

To be a station and connect to an access point, execute a call to `WiFi.begin(ssid, password)`. Now we need to to poll `WiFi.status()`. When this returns WL_CONNECTED, then we are connected to the network.

See also:

- Arduino WiFi library

WiFi.begin

Start a WiFi connection as a station.

```
int begin(
       const char *ssid,
       const char *passPhrase=NULL,
       int32_t channel=0,
       uint8_t bssid[6]=NULL)

int begin(
       char *ssid,
       char *passPhrase=NULL,
       int32_t channel=0,
       uint8_t bssid[6]=NULL)
```

Begin a WiFi connection as a station. The `ssid` parameter is mandatory but the others can be left as default. The return value is our current connection status.

WiFi.beingSmartConfig

bool beginSmartConfig()

WiFi.beginWPSConfig

bool beginWPSConfig()

WiFi.BSSID

Retrieve the current BSSID.

```
uint8_t BSSID()
uint8_t *BSSID(uint8_t networkItem)
```

Retrieve the current BSSID.

WiFi.BSSIDstr

Retrieve the current BSSID as a string representation.

```
String BSSIDstr()
String BSSIDstr(uint8_t networkItem)
```

Retrieve the current BSSID as a string representation.

WiFi channel

Retrieve the current channel.

```
int32_t channel()
int32_t channel(uint8_t networkItem)
```

Retrieve the current channel.

WiFi.config

Set the WiFi connection configuration.

```
void config(IPAddress local_ip, IPAddress gateway, IPAddress subnet)
void config(IPAddress local_ip, IPAddress gateway, IPAddress subnet, IPAddress dns)
```

Set the configuration of the WiFi using static parameters. This disables DHCP.

WiFi.disconnect

Disconnect from an access point.

```
int disconnect(bool wifiOff = false)
```

Disconnect from the current access point.

WiFi.encryptionType

Return the encryption type of the scanned WiFi access point.

```
uint8_t encryptionType(uint8_t networkItem)
```

Return the encryption type of the scanned WiFi access point.

The values are one of:

- ENC_TYPE_NONE
- ENC_TYPE_WEP
- ENC_TYPE_TKIP
- ENC_TYPE_CCMP
- ENC_TYPE_AUTO

WiFi.gatewayIP

Get the IP address of the station gateway.

```
IPAddress gatewayIP()
```

Retrieve the IP address of the station gateway.

WiFi.getNetworkInfo

Retrieve all the details of the specified scanned networkItem.

```
bool getNetworkInfo(uint8_t networkItem,
       String &ssid,
       uint8_t &encryptionType,
       int32_t &RSSI,
       uint8_t *&BSSID,
       uint32_t &channel,
       bool &isHidden)
```

Retrieve all the details of the specified scanned networkItem.

WiFi.hostByName

Lookup a host by a name.

```
int hostByName(const char *hostName, IPAddress &result)
```

Look up a host by name and get its IP address. This function returns 1 on success and 0 on failure.

WiFi.hostname

Retrieve and set the hostname used by this station.

```
String hostname()
bool hostname(char *hostName)
bool hostname(const char *hostName)
bool hostname(String hostName)
```

WiFi.isHidden

Determine if the scanned network item is flagged as hidden.

```
bool isHiddem(uint8_t networkItem)
```

Determine if the scanned network item is flagged as hidden.

WiFi.localIP

Get the station IP address.

```
IPAddress localIP()
```

Get the IP address for the station.

WiFi.macAddress

Get the station interface MAC address.

```
uint_t *macAddress(uint8_t *mac)
String macAddress()
```

Get the station interface MAC address.

WiFi.mode

Set the operating mode.

```
void mode(WiFiMode mode)
```

Set the operating mode of the WiFi. This is one of:

- `WIFI_OFF` – Switch off WiFi
- `WIFI_STA` – Be a WiFi station
- `WIFI_AP` – Be a WiFi access point
- `WIFI_AP_STA` – Be both a WiFi station and a WiFi access point

See also:

- Defining the operating mode

WiFi.printDiag

Log the state of the WiFi connection.

```
void printDiag(Print &dest)
```

Log the state of the WiFi connection.

WiFi.RSSI

Retrieve the RSSI value of the scanned network item.

```
int32_t RSSI(uint8_t networkItem)
```

Retrieve the RSSI value of the scanned network item.

WiFi.scanComplete

Determine the status of a previous scan request.

```
int8_t scanComplete()
```

If the result is >= 0 then this is the number of WiFi access points found. Otherwise, the value is less than 0 and the codes are:

- `SCAN_RUNNING` – A scan is currently in progress.
- `SCAN_FAILD` – A scan failed.

WiFi.scanDelete

Delete the results from a previous scan.

```
void scanDelete()
```

Delete the results from a previous scan. A request to scan the network results in the allocation of memory. This call releases that memory.

WiFi.scanNetworks

Scan the access points in the environment.

```
int8_t scanNetworks(bool async = false)
```

Scan the access points in the environment. We can either perform this synchronous or asynchronous. On a synchronous call, the result is the number of access points found.

WiFi.smartConfigDone

bool smartConfigDone()

WiFi.softAP

Setup an access point.

void softAP(const char *ssid)

void softAP(const char *ssid, const char *passPhrase, int channel=1, int ssid_hidden=0)

WiFi.softAPConfig

void softAPConfig(IPAddress local_ip, IPAddress gateway, IPAddress subnet)

WiFi.softAPdisconnect

int softAPdisconnect(bool wifiOff=false)

WiFi.softAPmacAddress

uint8_t *softAPmacAddress(uint8_t *mac)

WiFi.softAPIP

Get the IP address of the access point interface.

```
IPAddress softAPIP()
```

Return the IP address of the access point interface.

WiFi.SSID

Retrieve the SSID.

```
char *SSID()
const char *SSID(uint8_t networkItem)
```

Here we retrieve the SSID of the current station or the SSID of the scanned network id.

WiFi.status

Retrieve the current WiFi status.

```
wl_status_t status()
```

The status returned will be one of:

- WL_CONNECTED
- WL_NO_SSID_AVAIL
- WL_CONNECT_FAILED
- WL_IDLE_STATUS
- WL_DISCONNECTED

WiFi.stopSmartConfig

void stopSmartConfig()

WiFi.subnetMask

IPAddress subnetMask()

WiFi.waitForConnectResult

Wait until the WiFi connection has been formed or failed.

```
uint8_t waitForConnectResult()
```

If we are a station, then block waiting for us to become disconnected or failed. The return code is the status.

WiFi Client

WiFiClient

WiFiClient.available

int available()

WiFiClient.connect

Connect to the given host at the given port using TCP.

```
int connect(const char* host, uint16_t port)
int connect(IPAddress ip, uint16_t port)
```

Connect to the given host at the given port using TCP. This function returns 0 on a failure.

WiFiClient.connected

uint8_t connected()

WiFiClient.flush
void flush()

WiFiClient.getNoDelay
bool getNoDelay()

WiFiClient.peek
int peek()

WiFiClient.read
Read data from the partner.

```
int read()
int read(uint8_t *buf, size_t size)
```

Read data from the partner. These functions read either a single byte or a sequence of bytes from the partner.

WiFiClient.remoteIP
Retrieve the remote IP address of the connection.

```
IPAddress remoteIP()
```

Retrieve the remote IP address of the connection.

WiFiClient.remotePort
Return the remote port being used in an existing connection.

```
uint16_t remotePort()
```

Return the remote port being used in an existing connection.

WiFiClient.setLocalPortStart
Set the initial port for allocating local ports for connections.

```
void setLocalPortStart(uint16_t port)
```

Set the initial port for allocating local ports for connections.

WiFiClient.setNoDelay
void setNoDelay(bool nodelay)

WiFiClient.status
uint8_t status()

WiFiClient.stop
void stop()

WiFiClient.stopAll
Stop all the connections formed by this WiFi client.

```
void stopAll()
```

WiFiClient.write
Write data to the partner.

```
size_t write(uint8_t b)
size_t write(const uint8_t *buf, size_t size)
```

Write data to the partner. The first function writes one byte, while the second function writes an array of characters.

WiFiServer

WiFiServer
Create an instance of a Server listening on the supplied port.

```
WiFiServer(uint16_t port)
```

Create an instance of a Server listening on the supplied port.

WiFiServer.available
Retrieve a WiFiClient object that can be used for communications.

WiFiClient available(byte* status)

WiFiServer.begin
Start listening for incoming connections.

void begin()

Start listening for incoming connections.

WiFiServer.getNoDelay

WiFiServer.hasClient
Return true if we have a client connected.

bool hasClient()

WiFiServer.setNoDelay

WiFiServer.status

WiFiServer.write

Sample applications

Reading and reviewing sample applications is good practice. It allows you to study what others have written and see if you can understand each of the statements and the program flow as a whole.

Sample – Light an LED based on the arrival of a UDP datagram

In this sample we will have the ESP8266 become a WiFi station and connect. It will start to listen for incoming datagrams and if the first byte of received data is the character "1", it will light an LED. If the character is "0", it will extinguish the LED.

Here is the full code of the application with commentary following:

```
#include <ets_sys.h>
#include <osapi.h>
#include <os_type.h>
#include <gpio.h>
#include <user_interface.h>
#include <espconn.h>
#include <mem.h>
#include "driver/uart.h"


#define LED_GPIO 15

LOCAL struct espconn conn1;
LOCAL esp_udp udp1;

LOCAL void recvCB(void *arg, char *pData, unsigned short len);
LOCAL void eventCB(System_Event_t *event);
LOCAL void setupUDP();
LOCAL void initDone();

LOCAL void recvCB(void *arg, char *pData, unsigned short len) {
	struct espconn *pEspConn = (struct espconn *)arg;
	os_printf("Received data!! - length = %d\n", len);
	if (len == 0 || (pData[0] != '0' && pData[0] != '1')) {
		return;
	}
	int v = (pData[0] == '1');
	GPIO_OUTPUT_SET(LED_GPIO, v);
} // End of recvCB

LOCAL void initDone() {
	wifi_set_opmode_current(STATION_MODE);
	struct station_config stationConfig;
	strncpy(stationConfig.ssid, "myssid", 32);
	strncpy(stationConfig.password, "password", 64);
	wifi_station_set_config(&stationConfig);
	wifi_station_connect();
} // End of initDone

LOCAL void setupUDP() {
```

```
	conn1.type = ESPCONN_UDP;
	conn1.state = ESPCONN_NONE;
	udp1.local_port = 25867;
	conn1.proto.udp = &udp1;
	espconn_create(&conn1);
	espconn_regist_recvcb(&conn1, recvCB);
	os_printf("Listening for data\n");
} // End of setupUDP


LOCAL void eventCB(System_Event_t *event) {
	switch (event->event) {
	case EVENT_STAMODE_GOT_IP:
		os_printf("IP: %d.%d.%d.%d\n", IP2STR(&event->event_info.got_ip.ip));
		setupUDP();
		break;
	}
} // End of eventCB

void user_rf_pre_init(void) {
}

void user_init(void) {
	uart_init(BIT_RATE_115200, BIT_RATE_115200);

	// Set GPIO15 as a GPIO pin
	PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDO_U, FUNC_GPIO15);

	// Call "initDone" when the ESP8266 has initialized
	system_init_done_cb(initDone);
	wifi_set_event_handler_cb(eventCB);
} // End of user_init
```

Control starts in the `user_init()` function where we setup the UART baud. In this example, we have chosen GPIO15 as our output pin so we map the function of the physical pin called "MTDO_U" to the logical function of "GPIO15". We register a function called `initDone()` to be called when initialization of the device is complete and we also register a function called `eventCB()` to be called when WiFi events arrive indicating a change of state.

With these items having been setup, we return control back to the OS. We expect to be called back through `initDone()` when the device is fully read for work. In `initDone()` we define ourselves as a Wifi Station and name the access point with its password that we wish to use. Finally we ask for a connection to the access point.

If all goes well, we will be connected to the access point and then be allocated an IP address. Both of these will result in events being generated which will cause us to wake up in `eventCB()`. The only event we are interested in seeing is the allocation of the IP address. When we are notified of that, we call the function called `setupUDP()` to initialize our UDP listening environment.

In `setupUDP()`, we create a `struct espconn` control block defined for UDP and configured to listen on our chosen port of 25867. We also register a receive callback to the function `recvCB()`. This will be called when new data arrives. At this point, all our setup is completed and we have a device connected to the WiFi network listening on UDP port 25867 for datagrams.

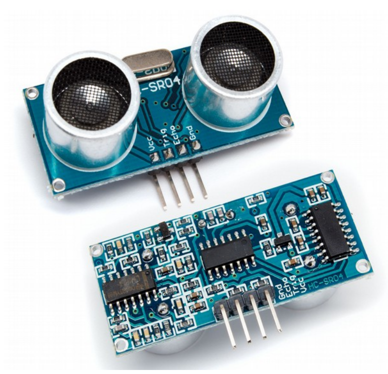
When a datagram arrives, we wake up in `recvCB()` having been passed in the datagram data. We check that we actually have data and that it is good … if not, we end the callback straight away.

Finally, we look at the first character of the data and, based on its value, change the output value of the GPIO. The physical GPIO is wired to an LED and a resistor.

If a character of '1' is transmitted, the output of GPIO15 goes high and the LED lights. If the character value is '0', the output of GPIO15 goes low, and the LED is extinguished.

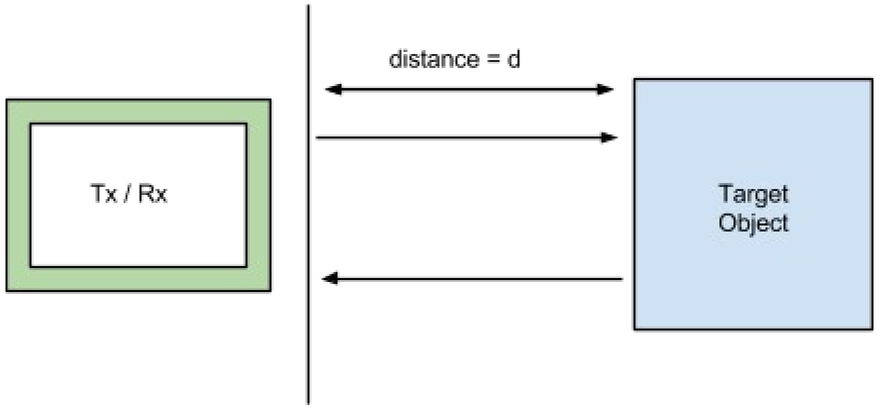
Sample – Ultrasonic distance measurement

The HC SR-04 is an ultrasonic distance measurement sensor.



Send a minimum of a 10us pulse to `Trig` (low to high to low). Later, `Echo` will go low/high/low. The time that `Echo` is high is the time it takes the sonic pulse to reach a back-end and bounce back.

Speed of sound is 340.29 m/s (340.29 * 39.3701 inches/sec). Call this V_{sound}.



If T_{echo} is the time for echo response then d = (T_{echo} * V_{sound}) / 2.

Also the equation for expected T_{echo} lengths is given by:

$T_{echo} = 2d/V_{sound}$

For example:

Distance	Time
1cm	2 * 0.01 / 340 = 0.058 msecs = 59 usecs
10cm	2 * 0.1 / 340 = 0.59 msecs = 590 usecs
1m	2 * 1 /340 = 5.9 msecs = 5900 usecs (5.9 msecs)

Because the `Echo` response is a 5V signal, it is vital to reduce this to 3.3V for input into into the ESP8266. A voltage divider will work. The pins on the device are:

- `Vcc` – The input voltage is 5V.
- `Trig` – Pulse (low to high) to trigger a transmission … minimum of 10usecs.
- `Echo` – Pulses low to high to low when an echo is received. Warning, this is a 5V output.
- `Gnd` – Ground.

To drive this device, we need to utilize two pins on the ESP8266 that we will logically call `Trig` and `Echo`. In my design, I set `Trig` to be GPIO4 and `Echo` to be GPIO5.

Our design for the application will not include any networking but it should be straightforward to ass it as needed. We will setup a timer that fires once a second which is how often we wish to take a measurement. When the timer wakes up, we will pulse Trig from low to high and back to low holding high for 10 microseconds. We will now record the time and start polling the Echo pin waiting for it to go high. When it does, we will record the time again and subtracting one from the one will tell us how long it took the sound to bounce back. From that we can calculate the distance to an object. If no response is received in 20 msecs, we will assume that there was no object to detect. We will then log the result to the Serial console.

An example program that performs this design is shown next:

```
#define TRIG_PIN 4
#define ECHO_PIN 5

os_timer_t myTimer;

void user_rf_pre_init(void) {
}

void timerCallback(void *pArg) {
	os_printf("Tick!\n");
	GPIO_OUTPUT_SET(TRIG_PIN, 1);
	os_delay_us(10);
	GPIO_OUTPUT_SET(TRIG_PIN, 0);
	uint32 val = GPIO_INPUT_GET(ECHO_PIN);
	while(val == 0) {
		val = GPIO_INPUT_GET(ECHO_PIN);
	}
	uint32 startTime = system_get_time();
	val = GPIO_INPUT_GET(ECHO_PIN);
	while(val == 1 && (system_get_time() - startTime) < (20 * 1000)) {
		val = GPIO_INPUT_GET(ECHO_PIN);
	}
	if (val == 0) {
		uint32 delta = system_get_time() - startTime;
		// Calculate the distance in cm.
		uint32 distance = 340.29 * 100 * delta / (1000 * 1000 * 2);
		os_printf("Distance: %d\n", distance);
	} else {
		os_printf("No echo!\n");
	}
} // End of timerCallback

void user_init(void) {
	uart_init(BIT_RATE_115200, BIT_RATE_115200);
	// Setup ultrasonics pins as GPIO
	setAsGpio(TRIG_PIN);
	setAsGpio(ECHO_PIN);
	setupBlink(15);
	// Set the trigger pin to be default low
	GPIO_OUTPUT_SET(TRIG_PIN, 0);
	os_timer_setfn(&myTimer, timerCallback, NULL);
	os_timer_arm(&myTimer, 1000, 5);
} // End of user_init
```

Once this has been written and tested, we will make a second pass at the puzzle but this time using an interrupt to trigger the response to the echo.

See also:

- GPIOs

Sample – WiFi Scanner

A WiFi scanner is an application which periodically scans for available WiFi networks and shows them to the user. In our design, we will scan periodically and remember the set of networks we find. When we perform re-scans, we will check to see if each of the networks located is a network we have previously seen and, if not, list it to the user. We will also keep a "last seen" time for each network and if a network has not been seen for a minute, then we will forget about it such that if it appears again, we will once more list it to the user.

To illustrate our design, we will break the solution into a number of parts. The first part will be to register a callback function that is called every 30 seconds. This callback will be responsible for requesting a WiFi scan using `wifi_station_scan()`. This takes a callback function which itself will be invoked when the scan is complete.

When the scan completes, we will have a new list of detected networks. We will walk this list and for each network detected, determine if we have seen it before. If we have, we will update the last seen time. If not, we will add it to the list of previously seen networks and log it to the user.

A second timer callback will run once a minute and will walk the list of previously seen networks. If any of them are older than a minute, we will remove them.

See also:

- Scanning for access points

Sample Libraries

There are times when commonly used functions can be captured and reused over and over. This section describes just such a set of functions which have been collected. The source for these functions has been placed in Github at <location to be provided>.

The functions, when compiled, are placed in a library called `libcommon.a`. This can then be linked within your Makefile so that unresolved references to these functions can be satisfied.

A header file called "`common.h`" is all that one needs to add into your own applications.

Function list

authModeToString

Given an AUTH_MODE, return a string representation of the mode.

```
char *authModeToString(AUTH_MODE mode)
```

checkError

Check a return code for an error.

```
void checkError(sint8 err)
```

Check the `err` code for an error and if it is one, log it.

delayMilliseconds

Delay for a period of milliseconds.

```
void delayMilliseconds(uint32 milliseconds)
```

The `milliseconds` parameters is the number of milliseconds to delay before returning.

dumpBSSINFO

Dump an instance of struct bss_info to the log.

```
void dumpBSSINFO(struct bss_info *bssInfo)
```

dumpEspConn

Dump to the log a decoded representation of the `struct espconn`.

```
void dumpEspConn(struct espconn *pEspConn)
```

dumpRestart

Dump the restart information to the log.

```
void dumpRestart()
```

See also:

- Exception handling

dumpState

Dump the WiFi station state to the log.

```
void dumpState()
```

See also:

- system_print_meminfo
- system_get_free_heap_size
- system_get_boot_version
- system_get_userbin_addr
- system_get_boot_mode
- system_get_flash_size_map
- system_get_sdk_version()

errorToString

Given an error code, return a string representation of it.

```
char *errorToString(sint8 err)
```

eventLogger

Write a WiFi event to the log.

```
void eventLogger(System_Event_t *event)
```

We can register this function as a callback for a WiFi event. Write the event data to the log.

See also:

- Handling WiFi events
- wifi_set_event_handle_cb

eventReasonToString

Convert an event reason to a string representation.

```
char *eventReasonToString(int reason)
```

Some of the WiFi event callbacks can return a reason value that is an encoding of the reason that something failed. This function returns a string representation of the int value code.

flashSizeAndMapToString

Return a string representation of the flash size and map.

```
char *flashSizeAndMapToString()
```

setAsGpio

Set a pin to be used as a GPIO.

```
void setAsGpio(uint8 pin)
```

Set the GPIO supplied as `pin` to be GPIO function.

See also:

- GPIOs

setupBlink

Setup a blinking LED on the given pin.

```
void setupBlink(uint8 blinkPin)
```

The `blinkPin` parameter is the pin to use for blinking.

toHex

Convert an array of bytes to a hex string.

```
uint8 *toHex(uint8 *ptr, int size, uint8 *buffer)
```

Convert the bytes pointed to by `ptr` for `size` bytes into a hex string. The `buffer` parameter will be where the result will be stored. It must be 2 * size + 1 bytes in length (or more). Each byte is 2 hex characters plus a single byte NULL terminator at the end. The function returns the start of the buffer.

API Reference

Now we have a mini reference to the syntax of many of the ESP8266 exposed APIs. Do not use this reference exclusively. Please also refer to the published Espressif SDK Programming Guide.

Some acronyms and other names are used in the naming of APIs and may need some explanation to fully appreciate them:

- dhcpc – DHCP client
- dhcps – DHCP server
- softap – Access point implemented in software
- wps – Unknown
- sntp – Simple Network Time Protocol
- mdns – Multicast Domain Name System
- uart – Universal asynchronous receiver/transmitter
- pwm – Pulse width modulation

Timer functions

Timer functions allow us to register functions that will be executed at a time in the future or periodically after time passes. We also group functions that manipulate or retrieve time values in this set.

os_timer_arm

Enable a millisecond granularity timer.

```
void os_timer_arm(
       os_timer_t *pTimer,
       uint32_t milliseconds,
       bool repeat)
```

Arm a timer such that is starts ticking and fires when the clock reaches zero.

The `pTimer` parameter is a pointed to a timer control structure.

The `milliseconds` parameter is the duration of the timer measured in milliseconds.

The `repeat` parameter is whether or not the timer will restart once it has reached zero.

Includes:

- osapi.h

See also:

- Timers and time
- os_timer_disarm
- os_timer_setfn

os_timer_disarm

Disarm/Cancel a previously armed timer.

```
void os_timer_disarm(os_timer_t *pTimer)
```

Stop a previously started timer which was started by a call to `os_timer_arm()`.

The `pTimer` parameter is a pointer to a timer control structure.

Includes:

- osapi.h

See also:

- Timers and time
- os_timer_arm
- os_timer_setfn

os_timer_setfn

Define a function to be called when the timer fires

```
void os_timer_setfn(
       os_timer_t *pTimer,
       os_timer_func_t *pFunction,
       void *pArg)
```

Define the callback function that will be called when the timer reaches zero.

The `pTimer` parameters is a pointer to the timer control structure.

The `pFunction` parameters is a pointer to the callback function.

The `pArg` parameter is a value that will be passed into the called back function.

The callback function should have the signature:

```
void (*functionName)(void *pArg)
```

The `pArg` parameter is the value registered with the callback function.

Includes:

- osapi.h

See also:

- Timers and time
- os_timer_arm
- os_timer_disarm

system_timer_reinit

Used to set a uSecond timer

os_timer_arm_us
Enable a microsecond timer

hw_timer_init
Initialize a hardware timer

hw_timer_arm
Set the trigger delay

hw_timer_set_func
Set the timer callback

System Functions

system_restore
Reset some system settings to defaults

system_restart
Restart the system

system_init_done_cb
Register a function to be called when system initialization is complete

```
void system_init_done_cb(init_done_cb_t callbackFunction)
```

This function is designed only be called in `user_init()`. It will register a function to be called one time after the ESP8266 has been initialized. The `init_done_cb_t` defines a function:

```
void (*functionName)(void)
```

See also:

- Custom programs

system_get_chip_id
Get the id of the chip

```
long system_get_chip_id()
```

For example: 0xf94322

system_get_vdd33

Measure voltage

Unknown … but related to analog to digital conversion.

See also:

- Analog to digital conversion
- system_adc_read

system_adc_read

Read the A/D converter value.

```
uint16 system_adc_read()
```

Read the value of the analog to digital converter. The granularity is 1024 discrete steps.

See also:

- Analog to digital conversion

system_deep_sleep

Puts the device to sleep for a period of time.

```
void system_deep_sleep(uint32 microseconds)
```

system_deep_sleep_set_option

Define what the chip will do when it next wakes up.

```
bool system_deep_sleep_set_option(uint8 option)
```

system_phys_set_rfoption

Enable the RF after waking up from a sleep (or not)

system_phys_set_max_tpw

Set the maximum transmission power

system_phys_set_tpw_via_vdd33

Set the transmission power as a function of voltage

system_set_os_print

Turn on or off logging.

```
void system_set_os_print(unint8 onOff)
```

A value of 0 switches it off while a value of 1 switches it on. It was initially thought that this controlled OS level logging however it seems to control **all** logging via `os_printf()`.

Includes:

- user_interface.h

See also:

- Logging to UART1

system_print_meminfo

Print memory information

```
void system_print_meminfo()
```

Memory information for diagnostics is written to the output stream which is commonly UART1. The format of the data looks as follows:

```
data  : 0x3ffe8000 ~ 0x3ffe853c, len: 1340
rodata: 0x3ffe8540 ~ 0x3ffe8af0, len: 1456
bss   : 0x3ffe8af0 ~ 0x3fff1c18, len: 37160
heap  : 0x3fff1c18 ~ 0x3fffc000, len: 41960
```

The `.data` section is where global and static local initialized variables are kept.

The `.rodata` section is where read-only global and static data is kept.

The `.bss` is where un-initialized global and local static data is kept.

The `.heap` is where the heap of the program can be found.

See also:

- Wikipedia – .bss
- Wikipedia – Data segment

system_get_free_heap_size

Get the size of the available memory heap

```
int system_get_free_heap_size()
```

For example "40544".

See also:

- os_malloc
- os_free

system_os_task

Setup a task for execution

```
bool system_os_task(os_task_t task,
      uint8 priority,
      os_event_t *queue,
      uint queueLength)
```

The "`os_task_t`" is a pointer to a task function which has the signature:

```
void (*functionName)(os_event_t *event)
```

The `os_event_t` is a structure which contains:

- `os_signal_t signal`
- `os_param_t param`

Both of these are unsigned 32bit integers.

The return is true on success and false on failure.

See also:

- Task handling

system_os_post

Post a message to a task

```
bool system_os_post(uint8 priority,
      os_signal_t signal,
      os_param_t parameter)
```

The return is true on success and false on failure.

See also:

- Task handling
- system_os_task

system_get_time

Get the system time. This is measured in microseconds since last device startup.

```
uint32 system_get_time()
```

This timer will roll over after 71 minutes.

Includes:

- <Include missing for this function>

See also:

- Timers and time

system_get_rtc_time

Get the real time clock time

system_rtc_clock_cali_proc

Clock calibration

system_rtc_mem_write

Storage space for saving data during a deep sleep in RTC storage

system_rtc_mem_read

Read data from RTC available storage.

system_uart_swap

Swap serial UARTs

system_uart_de_swap

Go back to original UART

system_get_boot_version

The version of the boot loader.

```
uint8 system_get_boot_version()
```

The current value returned through testing of my devices is "5".

system_get_userbin_addr

Get the address of user bin

```
uint32 system_get_userbin_addr()
```

The current value returned on my devices is 0x0.

system_get_boot_mode

Get the current boot mode

```
uint8 system_get_boot_mode()
```

The return value indicates the current boot mode and will be one of:

* `SYS_BOOT_ENHANCE_MODE` – 0
* `SYS_BOOT_NORMAL_MODE` – 1

On my devices, the value being returned is "0".

system_restart_enhance

Restarts the system in enhanced boot mode

system_update_cpu_freq

Set the CPU frequency

system_get_cpu_freq

Get the current CPU frequency

```
int system_get_cpu_freq()
```

Returns the CPU frequency in MHz. For example "80".

system_get_flash_size_map

Get current flash size and map

```
enum flash_size_map system_get_flash_size_map()
```

The value returned is an enum which has the following definitions:

* FLASH_SIZE_4M_MAP_256_256
* FLASH_SIZE_2M
* FLASH_SIZE_8M_MAP_512_512
* FLASH_SIZE_16M_MAP_512_512
* FLASH_SIZE_32M_MAP_512_512
* FLASH_SIZE_16M_MAP_1024_1024
* FLASH_SIZE_32M_MAP_1024_1024

See also:

* Flashing the ESP8266

system_get_rst_info

Information about the current startup.

```
struct rst_info* system_get_rst_info()
```

Retrieve information about the current device startup.

See also:

- Exception handling
- struct rst_info

system_get_sdk_version()

Return the version of the SDK

```
char *system_get_sdkVersion()
```

For example "1.1.1".

system_soft_wdt_stop

Disable the software watchdog.

```
void system_soft_wdt_stop()
```

Stop the software watchdog. It is recommended not to stop this timer for too long (8 seconds or less) otherwise the hardware watchdog will force a reset.

See also:

- Watchdog timer

system_soft_wdt_restart

Restart the software watchdog.

```
void system_soft_wdt_restart()
```

Restart the software watchdog following a previous call to stop it.

See also:

- Watchdog timer

os_memset

Set the values of memory

```
void os_memset(void *pBuffer, int value, size_t size)
```

Set the memory pointed to by `pBuffer` to the `value` for `size` bytes.

Includes:

- osapi.h

See also:

- Working with memory

- os_memcpy

os_memcmp

Compare two regions of memory.

```
int os_memcmp(uint8 *ptr1, uint8 *ptr2, int size)
```

Compare two regions of memory. The return is 0 if they are equal.

Includes:

- osapi.h

os_memcpy

Copy the values of memory.

```
void os_memcpy(void *destination, void *source, size_t size)
```

Copy the memory from the buffer pointed to by `source` to the buffer pointed to by `destination` for the number of bytes specified by `size`.

Includes:

- osapi.h

See also:

- Working with memory
- os_memset

os_malloc

Allocate storage from the heap.

```
void *malloc(size_t size)
```

Allocate `size` bytes from the heap and return a pointer to the allocated storage.

Includes:

- mem.h

See also:

- Working with memory
- os_zalloc
- os_free

os_zalloc

Allocate storage from the heap and zero its values.

```
void *zalloc(size_t size)
```

Allocate `size` bytes from the heap and return a pointer to the allocated storage. Before returning, the storage area is zeroed.

Includes:

- mem.h

See also:

- Working with memory
- os_malloc
- os_free

os_free

Release previously allocated storage back to the heap.

```
void os_free(void *pBuffer)
```

Release the storage previously allocated by `os_malloc()` or `os_zalloc()` back to the heap.

Includes:

- mem.h

See also:

- Working with memory
- os_malloc
- os_zalloc

os_bzero

Set the values of memory to zero.

```
void os_bzero(void *pBuffer, size_t size)
```

Sets the data pointer to by `pBuffer` to zero for `size` bytes.

Includes:

- osapi.h

See also:

- Working with memory

os_delay_us

Delay for microseconds.

```
void os_delay_us(uint16 us)
```

Delay for a maximum interval of 65535 microseconds.

Includes:

- osapi.h

See also:

- Timers and time

os_printf

Print a string to UART.

`void os_printf(char *format, …)`

The format flags that are known to work include:

- `%d` – display a decimal
- `%ld` – display a long decimal
- `%x` – display as a hex number
- `%s` – display as a string
- `"\n"` – display a newline (includes a prefixed carriage return)

The output text is sent to the function registered with `os_install_putc1()`. By default, this is UART0 but can be changed to UART1 by setting the `uart1_write_char()` function.

Includes:

- osapi.h

See also:

- Debugging

os_install_putc1

Register a function print a character

`void os_install_putc1(void (*pFunc)(char c));`

Register a function that will be called by output functions such as `os_printf()` that will log output. For example, this can be used to write to the serial ports. When a call is made to the supplied `uart_init()` method, the writing function is set to write to UART1.

Includes:

- osapi.h

See also:

- os_printf

os_random

`unsigned long os_random()`

Includes:

- osapi.h

os_get_random

```
int os_get_random(unsigned char *buf, size_t len)
```

Includes:

- osapi.h

os_strlen

Get the length of a string.

```
int os_strlen(char *string)
```

Return the length of the null terminated string.

Includes:

- osapi.h

os_strcat

Concatenate two strings together.

```
char *os_strcat(char *str1, char *str2)
```

Concatenate the null terminated sting pointed to by `str1` with the string pointed to by `str2` and store the result at `str1`.

Includes:

- osapi.h

os_strchr

Includes:

- osapi.h

os_strcmp

Compare two strings.

```
int os_strcmp(char *str1, char *str2)
```

Compare the null terminated string pointed to by `str1` with the null terminated string pointed to by `str2`. If `str1` < `str2` then the return is < 0. If `str1` > `str2` then the return is > 0 otherwise they are equal and the return is 0.

Includes:

- osapi.h

os_strcpy

Copy one string to another.

```
char *os_strcpy(char *dest, char *src)
```

Copy the null terminated string pointed to by `src` to the memory located at `dest`.

Includes:

- osapi.h

os_strncmp

Includes:

- osapi.h

os_strncpy

Copy one string to another but be sensitive to the amount of memory available in the target buffer.

```
char *os_strncpy(char *dest, char *source, size_t sizeOfDest)
```

Understand that the resulting string in `dest` may **not** be null terminated.

Includes:

- osapi.h

os_sprintf

```
sprintf(char * buffer, char *format, ...)
```

The format is not as rich as normal `sprintf()` in a C library. For example, no float or double support.

Includes:

- osapi.h

os_strstr

Includes:

- osapi.h

SPI Flash

spi_flash_get_id

Ge the ID info of SPI flash

```
uint32 spi_flash_get_id(void)
```

Includes:

- spi_flash.h

See also:

spi_flash_erase_sector

Erase a flash sector

SpiFlashOpResult spi_flasg_erase_sector(uint16 sec)

Includes:

- spi_flash.h

See also:

spi_flash_write

Write data to flash

SpiFlashOpResult spi_flash_write(uint32 des_addr, uint32 *src_addr, unit32 size)

Includes:

- spi_flash.h

See also:

spi_flash_read

Read data from flash

SpiFlashOpResult spi_flash_read(uint32 src_addr, uint32 des_addr, uint32 size)

Includes:

- spi_flash.h

See also:

spi_flash_set_read_func

void spi_flash_set_read_func(user_spi_flash_read read)

Includes:

- spi_flash.h

See also:

system_param_save_with_protect

Memory saving

bool system_param_save_with_protect(unit16 start_sec, void *param, uint16 len)

Includes:

- spi_flash.h

See also:

system_param_load

Read data saved with flash protection

bool system_param_load(uint16 start_sec, uint16 offset, void *param, unit16 len)

Includes:

- spi_flash.h

See also:

Wifi

wifi_get_opmode

Get the operating mode of the WiFi

`uint8 wifi_get_opmode()`

Return the current operating mode of the device.

There are three values defined:

- `STATION_MODE` – Station mode

- `SOFTAP_MODE` – Soft Access Point (AP) mode
- `STATIONAP_MODE` – Station + Soft Access Point (AP) mode

Includes:

- user_interface.h

See also:

- Defining the operating mode
- wifi_get_opmode_default
- wifi_set_opmode
- wifi_set_opmode_current

wifi_get_opmode_default

Get the default operating mode

```
uint8 wifi_get_opmode_default()
```

Return the default operating mode of the device following startup.

There are three values defined:

- `STATION_MODE` – Station mode
- `SOFTAP_MODE` – Soft Access Point (AP) mode
- `STATIONAP_MODE` – Station + Soft Access Point (AP) mode

Includes:

- user_interface.h

See also:

- Defining the operating mode
- wifi_get_opmode
- wifi_set_opmode
- wifi_set_opmode_current

wifi_set_opmode

Set the operating mode of the WiFi including saving to flash.

```
bool wifi_set_opmode(uint8 opmode)
```

There are three values defined:

- `STATION_MODE` – Station mode
- `SOFTAP_MODE` – Soft Access Point (AP) mode
- `STATIONAP_MODE` – Station + Soft Access Point (AP) mode

Includes:

- user_interface.h

See also:

- Defining the operating mode
- wifi_get_opmode
- wifi_get_opmode_default
- wifi_set_opmode_current

wifi_set_opmode_current

Set the operating mode of the WiFi but don't save to flash.

```
bool wifi_set_opmode_current(uint8 opmode)
```

There are three values defined:

- `STATION_MODE` – Station mode
- `SOFTAP_MODE` – Soft Access Point (AP) mode
- `STATIONAP_MODE` – Station + Soft Access Point (AP) mode

Includes:

- user_interface.h

See also:

- Defining the operating mode
- wifi_get_opmode
- wifi_get_opmode_default
- wifi_set_opmode

wifi_set_broadcast_if

bool wifi_set_broadcast_if(uint8 interface)

Includes:

- user_interface.h

See also:

- Broadcast with UDP

wifi_get_broadcast_if

uint8 wifi_get_broadcast_if()

Includes:

- user_interface.h

See also:

* Broadcast with UDP

wifi_set_event_handle_cb

Define a callback function to sense WiFi events.

```
void wifi_set_event_handler_cb(wifi_event_handler_cb_t callbackFunction)
```

Registers a function to be called when an event is detected by the WiFi subsystem. The signature of the registered callback function is:

```
void (*functionName)(System_Event_t *event)
```

Includes:

* user_interface.h

See also:

* Handling WiFi events
* System_Event_t

wifi_get_ip_info

Retrieve the current IP info about the station.

```
bool wifi_get_ip_info(
      uint8 if_index,
      struct ip_info *info)
```

The `if_index` parameter defines the interface to retrieve. Two values are defined:

* `STATION_IF` – 0 – The station interface
* `SOFTAP_IF` – 1 – The Soft Access Point interface

The `info` parameter is populated with details of the current ip address, netmask and gateway.

Includes:

* user_interface.h

See also:

* Current IP Address, netmask and gateway
* struct ip_info

wifi_set_ip_info

Set the interface data for the device.

```
bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)
```

The `if_index` parameter defines the interface to retrieve. Two values are defined:

* `STATION_IF` – 0 – The station interface

- `SOFTAP_IF` – 1 – The Soft Access Point interface

The `info` parameter is a pointer to a `struct ip_info` that contains the values we wish to set.

Includes:

- user_interface.h

See also:

- Current IP Address, netmask and gateway
- struct ip_info

wifi_set_macaddr

Set the MAC address.

```
bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)
```

A MAC address is 6 bytes.

Includes:

- user_interface.h

wifi_get_macaddr

Get the MAC address.

```
bool wifi_get_macaddr(uint8 if_index, uint8 *macaddr)
```

A MAC address is 6 bytes.

Includes:

- user_interface.h

wifi_set_sleep_type

Includes:

- user_interface.h

wifi_get_sleep_type

Includes:

- user_interface.h

wifi_status_led_install

Associate a GPIO pin with the WiFi status LED.

```
void wifi_status_led_install(
      uint8 gpio_id,
      uint32 mux_name,
      uint8 gpio_func)
```

When WiFi traffic flows, we may wish a status LED to flicker or blink indicating flowing traffic. This function allows us to specify a GPIO that should be pulsed to indicate WiFi traffic.

The `gpio_id` parameter is the numeric pin number.

The `mux_name` is the name of the multiplexer logical name.

The `gpio_func` is the function to be enabled for that multiplexer.

Includes:

- user_interface.h

See also:

- wifi_status_led_uninstall

wifi_status_led_uninstall

Disassociate a status LED from a GPIO pin.

```
void wifi_status_led_uninstall()
```

Disassociates a previous association setup with a call to `wifi_status_led_install()`.

Includes:

- user_interface.h

See also:

- wifi_status_led_install

wifi_station_get_config

Get the current station configuration

```
bool wifi_station_get_config(struct station_config *config)
```

Retrieve the current station configuration settings.

Includes:

- user_interface.h

See also:

- Station configuration
- station_config

wifi_station_get_config_default

Get the default station configuration

Includes:

- user_interface.h

See also:

- Station configuration

wifi_station_set_config

Set the configuration of the station.

```
bool wifi_station_set_config(struct station_config *config)
```

This function can only be called when the device mode includes Station support. Specifically, the details of which access point to interact with are supplied here. The details are persisted across a restart of the device.

A return value of true indicates success and a value of false indicates failure.

Includes:

- user_interface.h

See also:

- Station configuration
- station_config

wifi_station_set_config_current

Set the configuration of the station but don't save to flash.

```
bool wifi_station_set_config_current(struct station_config *config)
```

This function can only be called when the device mode includes Station support. Specifically, the details of which access point to interact with are supplied here. The details are not persisted across a restart of the device.

A return value of true indicates success and a value of false indicates failure.

Includes:

- user_interface.h

See also:

- Station configuration
- station_config

wifi_station_connect

Connect the station to an access point.

```
bool wifi_station_connect()
```

If we are already connected to a different access point then we first need to disconnect from it using `wifi_station_disconnect()`. There is also an auto connect attribute which can be used to allow the device to attempt to connect to the last access point seen when it is powered on. This can be set with the `wifi_station_set_auto_connect()` function.

Includes:

- user_interface.h

See also:

- Connecting to an access point
- wifi_station_disconnect
- wifi_station_set_auto_connect
- wifi_station_get_auto_connect

wifi_station_disconnect

Disconnect the station from an access point.

```
bool wifi_station_disconnect()
```

We should presume that we have previously connected via a `wifi_station_connect()`. We can determine our current connection status through `wifi_station_get_connect_status()`.

Includes:

- user_interface.h

See also:

- wifi_station_connect
- wifi_station_get_connect_status

wifi_station_get_connect_status

Get the connection status of the station.

```
uint8 wifi_station_get_connect_status()
```

The result is an enum with the following possible values:

Enum name	Value
STATION_IDLE	0
STATION_CONNECTING	1
STATION_WRONG_PASSWORD	2
STATION_NO_AP_FOUND	3
STATION_CONNECT_FAIL	4
STATION_GOT_IP	5

Includes:

- user_interface.h

wifi_station_scan

Scan for available access points

```
bool wifi_station_scan(
      struct scan_config *config,
      scan_done_cb_t callbackFunction)
```

We can scan the WiFi frequencies looking for access points. We must be in station mode in order to execute the command. When the function is executed, we provide a callback function that will be asynchronously invoked at some time in the future with the results.

The `scan_config` structure contains:

- uint8 *ssid
- uint8 *bssid
- uint8 channel
- uint8 show_hidden

If we supply this structure, then only access points that match are returned.

The `scan_config` parameter can be `NULL` in which case no filtering will be performed and all access points will be returned.

The `scan_done_cb_t` is a function with the following structure:

```
void (*functionName)(void *arg, STATUS status)
```

The `arg` parameter is a pointer to a `struct bss_info`.

It is important to note that the **first** entry in the chain must be skipped over as it is the head of the list.

To get the next entry, we can use `STAILQ_NEXT(pBssInfoVar, next)`.

The AUTH_MODE is an enum

Enum name	Value
AUTH_OPEN	0
AUTH_WEP	1
AUTH_WPA_PSK	2
AUTH_WPA2_PSK	3
AUTH_WPA_WPA2_PSK	4

`STATUS` is an enum containing:

Enum name	Value
OK	0
FAIL	1
PENDING	2
BUSY	3
CANCEL	4

On success, the function returns true and false on a failure.

The name of this function is peculiar. Given that it appears to locate access points and not stations, I believe a more appropriate name would have been `wifi_access_point_scan()`.

Includes:

- user_interface.h

See also:

- Scanning for access points
- struct bss_info
- STATUS

wifi_station_ap_number_set

Number of stations that will be cached

```
bool wifi_station_ap_number_set(uint8 ap_number)
```

Includes:

- user_interface.h

wifi_station_get_ap_info

Get the information of access points cached

```
uint8 wifi_station_get_ap_info(struct station_config configs[])
```

Includes:

- user_interface.h

wifi_station_ap_change

Change the connection to another access point

```
bool wifi_station_ap_change(uint newApId)
```

Includes:

- user_interface.h

wifi_station_current_ap_id

Get the current access point id

```
uint8 wifi_station_get_current_ap_id()
```

Includes:

- user_interface.h

wifi_station_get_auto_connect

Determine whether or not the ESP will auto connect to the last access point on boot.

```
unit8 wifi_station_get_auto_connect()
```

Determine whether or not the device will attempt to auto-connect to the last access point on restart. A value if 0 means it will not while non 0 means it will.

Includes:

- user_interface.h

See also:

- wifi_station_connect
- wifi_station_disconnect
- wifi_station_set_auto_connect

wifi_station_set_auto_connect

Set whether or not the ESP will auto connect to the last access point on boot.

```
bool wifi_station_set_auto_connect(uint8 setValue)
```

Set whether or not the device will attempt to auto-connect to the last access point on restart. A value of 0 means it will not while a non 0 value means it will. If called in `user_init()`, the setting will be effective immediately. If called elsewhere, the setting will take effect on next restart.

Includes:

- user_interface.h

See also:

- wifi_station_connect
- wifi_station_disconnect
- wifi_station_get_auto_connect

wifi_station_dhcpc_start

Start the DHCP client.

```
bool wifi_station_dhcpc_start()
```

If DHCP is enabled, then the IP, netmask and gateway will be retrieved from the DHCP server while if disabled, we will be using static values.

Includes:

- user_interface.h

See also:

- Current IP Address, netmask and gateway
- wifi_set_ip_info
- wifi_station_dhcpc_stop

wifi_station_dhcpc_stop

Stop the DHCP client

```
bool wifi_station_dhcpc_stop()
```

If DHCP is enabled, then the IP, netmask and gateway will be retrieved from the DHCP server while if disabled, we will be using static values.

Includes:

- user_interface.h

See also:

- Current IP Address, netmask and gateway
- wifi_set_ip_info
- wifi_station_dhcpc_start

wifi_station_dhcpc_status

Get the DHCP client status

```
enum dhcp_status wifi_station_dhcpc_status()
```

One of:

- DHCP_STOPPED

- DHCP_STARTED

Includes:

- user_interface.h

wifi_station_set_reconnect_policy

What should happen when the ESP gets disconnected from the AP

```
bool wifi_station_set_reconnect_policy(bool set)
```

Includes:

- user_interface.h

wifi_station_get_rssi

Get the received signal strength indication (rssi)

```
sint8 wifi_station_get_rssi()
```

Includes:

- user_interface.h

wifi_station_set_hostname

```
bool wifi_station_set_hostname(char *name)
```

Includes:

- user_interface.h

wifi_station_get_hostname

```
char* wifi_station_get_hostname()
```

Includes:

- user_interface.h

wifi_softap_get_config

Retrieve the current softAP configuration details.

```
bool wifi_softap_get_config(struct softap_config *pConfig)
```

When called, the `struct softap_config` pointed to be `pConfig` will be filled in with the details of the current softAP configuration. The details returned are those actually in use and may differ from the ones saved for default.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- user_interface.h

See also:

- struct softap_config
- wifi_softap_get_config_default
- wifi_softap_set_config
- wifi_softap_set_config_current

wifi_softap_get_config_default

Retrieve the default softAP configuration details.

```
bool wifi_softap_get_config_default(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be filled in with the details of the default softAP configuration. The details returned are those used at boot and may be different from the ones currently in use.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- user_interface.h

See also:

- struct softap_config
- wifi_station_get_config
- wifi_softap_set_config
- wifi_softap_set_config_current

wifi_softap_set_config

Set the current and default softAP configuration.

```
bool wifi_softap_set_config(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be used as the details of the default and current softAP configuration.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- user_interface.h

See also:

- struct softap_config
- wifi_station_get_config
- wifi_softap_get_config_default
- wifi_softap_set_config_current

wifi_softap_set_config_current

Set the default softAP configuration.

```
bool wifi_softap_set_config_current(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be used as the details of the current softAP configuration but will not be saved as default.

Includes:

- user_interface.h

See also:

- struct softap_config
- wifi_station_get_config
- wifi_softap_get_config_default
- wifi_softap_set_config

wifi_softap_get_station_num

Return the count of stations currently connected.

```
uint8 wifi_softap_get_station_num()
```

Returns the number of stations currently connected. The maximum number of connections on an ESP8266 is 4 but we can reduce this in the softAP configuration if needed.

Includes:

- user_interface.h

See also:

- Being an access point
- wifi_softap_get_station_info

wifi_softap_get_station_info

Return the details of all connected stations.

```
struct station_info *wifi_softap_get_station_info()
```

The return data is a linked list of `struct station_info` data structures.

Includes:

- user_interface.h

See also:

- Being an access point
- wifi_softap_get_station_num
- wifi_softap_free_station_info

wifi_softap_free_station_info

Release the data associated with a `struct station_info`.

```
void wifi_softap_free_station_info()
```

Following a call to `wifi_softap_get_station_info()` we may have data returned to us. The data was allocated by the OS and we must return it with this function call. Note that this function does **not** take in the data that was returned.

Includes:

- user_interface.h

See also:

- Being an access point
- wifi_softap_get_station_info

wifi_softap_dhcps_start

Start the DHCP server service.

```
bool wifi_softap_dhcps_start()
```

Start the DHCP server service inside the device.

Includes:

- user_interface.h

See also:

- The DHCP server
- wifi_softap_dhcps_stop
- wifi_softap_set_dhcps_lease
- wifi_softap_dhcps_status
- wifi_softap_dhcps_offer_option

wifi_softap_dhcps_stop

Stop the DHCP server service.

```
bool wifi_softap_dhcps_stop()
```

Stop the DHCP server service inside the device.

Includes:

- user_interface.h

See also:

- The DHCP server
- wifi_softap_dhcps_start
- wifi_softap_set_dhcps_lease
- wifi_softap_dhcps_status
- wifi_softap_dhcps_offer_option

wifi_softap_set_dhcps_lease

Define the IP address range that will be leased by this DHCP server.

```
bool wifi_softap_set_dhcps_lease(struct dhcps_lease *pLease)
```

The `pLease` parameter is a pointer to a `struct dhcps_lease` which contains an IP address range of IP addresses that will be leased by this DHCP server. The difference between the upper and lower bound of the IP addresses must be 100 or less. This function will not take effect until the DHCP server is stopped and restarted (assuming it is already running).

Includes:

- user_interface.h

See also:

- The DHCP server
- wifi_softap_dhcps_start
- wifi_softap_dhcps_stop
- wifi_softap_dhcps_status
- wifi_softap_dhcps_offer_option
- struct dhcps_lease

wifi_softap_dhcps_status

Return the status of the DHCP server service.

```
enum dhcp_status wifi_softap_dhcps_status()
```

Retrieve the status of the DHCP server service. The returned value will be one of:

- DHCP_STOPPED
- DHCP_STARTED

Includes:

- user_interface.h

See also:

- The DHCP server
- wifi_softap_dhcps_start
- wifi_softap_dhcps_stop
- wifi_softap_set_dhcps_lease
- wifi_softap_dhcps_offer_option

wifi_softap_dhcps_offer_option

Set DHCP server options.

```
bool wifi_softap_set_dhcps_offer_option(uint8 level, void *optarg)
```

Currently, the `level` parameter can only be `OFFER_ROUTER` with `optarg` being a bit mask with values:

- `0b0` – Disable router information.
- `0b1` – Enable router information.

Includes:

- user_interface.h

See also:

- wifi_softap_dhcps_start
- wifi_softap_dhcps_stop
- wifi_softap_set_dhcps_lease
- wifi_softap_dhcps_status

wifi_set_phy_mode

Set the physical level WiFi mode.

```
bool wifi_set_phy_mode(enum phy_mode mode)
```

This is used to set the IEEE 802.11 network type such a b/g/n.

Includes:

- user_interface.h

See also:

- enum phy_mode

wifi_get_phy_mode

Get the physical level WiFi mode.

```
enum phy_mode wifi_get_phys_mode();
```

This is used to retrieve the IEEE 802.11 network type such a b/g/n.

Includes:

- user_interface.h

See also:

- enum phy_mode

wifi_wps_enable

bool wifi_wps_enable(WPS_TYPE_t wps_type)

The type parameter can be one of the following:

- `WPS_TYPE_DISABLE` – Unsupported

* `WPS_TYPE_PBC` – Push Button Configuration – Supported
* `WPS_TYPE_PIN` – Unsupported
* `WPS_TYPE_DISPLAY` – Unsupported
* `WPS_TYPE_MAX` – Unsupported

See also:

* WiFi Protected Setup – WPS

wifi_wps_disable

bool wifi_wps_disable()

See also:

* WiFi Protected Setup – WPS

wifi_wps_start

bool wifi_wps_start()

See also:

* WiFi Protected Setup – WPS

wifi_set_wps_cb

bool wifi_set_wps_cb(wps_st_cb_t callback)

The signature of the callback function is:

void (*functionName)(int status)

The status parameter will be one of:

* WPS_CB_ST_SUCCESS
* WPS_CB_ST_FAILED
* WPS_CB_ST_TIMEOUT

See also:

* WiFi Protected Setup – WPS

Upgrade APIs

system_upgrade_userbin_check

uint8 system_upgrade_userbin_check()

system_upgrade_flag_set

void system_upgrade_flag_set(uint8 flag)

system_upgrade_flag_check

uint8 system_upgrade_flag_check()

system_upgrade_start

bool system_upgrade_start(struct upgrade_server_info *server)

system_upgrade_reboot

void system_upgrade_reboot()

Sniffer APIs

wifi_promiscuous_enable

void wifi_promiscuous_enable(uint8 promiscuous)

wifi_promiscuous_set_mac

void wifi_promiscuous_set_mac(const uint8_t *address)

wifi_promiscuous_rx_cb

void wifi_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)

wifi_get_channel

wifi_set_channel

Smart config APIs

smartconfig_start

bool smartconfig_start(sc_callback_t cb, uint8 log)

smartconfig_stop

bool smartconfig_stop(void)

SNTP API

Handle Simple Network Time Protocol request.

sntp_setserver

Set the address of an SNTP server.

```
void sntp_serverserver(unsigned char index, ip_addr_t *addr)
```

Set the address of one of the three possible SNTP servers to be used.

The `index` parameter must be either 0, 1 or 2 and specifies which of the SNTP server slots is to be set.

The `addr` parameter is the IP address of the SNTP server to be recorded.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_getserver

Retrieve the IP address of the SNTP server.

```
ip_addr_t sntp_getserver(unsigned char index)
```

Retrieve the IP address of a previously registered SNTP server.

The `index` parameter is the index of the SNTP server to be retrieved. It may be either 0, 1 or 2.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_setservername

Set the hostname of a target SNTP server.

```
void sntp_setservername(unsigned char index, char *server)
```

Specify an SNTP server by its hostname.

The `index` parameter is the index of an SNTP server to be set. It may be either 0, 1 or 2.

The `server` parameter is a NULL terminated string that names the host that is an SNTP server.

See also:

- Working with SNTP

sntp_getservername

Get the hostname of a target SNTP server.

```
char *sntp_setservername(unsigned char index)
```

Retrieve the hostname of a specific SNTP server that was previously registered.

The `index` parameter is the index of an SNTP server that was previously set. It may be either 0, 1 or 2.

The return from this function is a NULL terminated string.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_init

```
void sntp_init()
```

Initialize the SNTP functions.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_stop

void sntp_stop()

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_current_timestamp

Get the current timestamp as an unsigned 32 bit value representing the number of seconds since January 1st 1970 UTC.

```
uint32 sntp_get_current_timestamp()
```

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_real_time

char *sntp_get_real_time(long t)

????

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_set_timezone

Set the current local timezone.

```
bool sntp_set_timezone(sint8 timezone)
```

Invoking this function declares our local timezone as a signed offset in hours from UTC. It should only be called when the SNTP functions are not running as for example after a call to `sntp_stop()`.

The `timezone` parameter is a time zone in the range -11 to 13.

The return value is true on success and false otherwise.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_timezone

Get the current timezone.

```
sint8 sntp_get_timezone()
```

Retrieve the current value for the timezone as previously set with a call to `sntp_set_timezone()`.

Includes:

- sntp.h

See also:

- Working with SNTP

Generic TCP/UDP APIs

espconn_delete

Delete a transmission

```
sint8 espconn_delete(struct espconn *espconn)
```

The device maintains data and storage for each conversation (TCP and UDP). When these conversations are finished and we no longer are going to communicate with the partners, we can indicate that by calling this function which will release the internal storage. It is anticipated that failure to do this will result in memory leaks.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- UDP
- espconn_create
- espconn_accept

espconn_dns_setserver

Set the default DNS server.

```
void espconn_dns_setserver(char numdns, ip_addr_t *dnsservers)
```

The `numdns` is the number of DNS servers supplied which must be 1 or 2. No more than 2 DNS servers may be supplied. This function should not be called if DHCP is being used.

The `dnsservers` parameter is an array of 1 or 2 IP addresses.

See also:

- Name Service

espconn_gethostbyname

```
err_t espconn_gethostbyname(struct espconn *espconn,
       const char *hostname,
       ip_addr_t *addr,
       dns_found_callback found)
```

The parameters are:

- espconn
- hostname
- addr
- found

The `dns_found_callback` is a function with the following signature:

```
void  (*functionName)(const char *name, ip_addr_t *ipAddr, void *arg)
```

where the `arg` parameter is a pointer to a `struct espconn`.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_OK` – Succeeded
- `ESPCONN_INPROGRESS` – already connected
- `ESPCONN_ARG` – Illegal argument

espconn_port

uint32 espconn_port()

espconn_regist_sentcb

Register a callback function that will be called when data has been sent.

```
sint8 espconn_regist_sentcb(
      struct espconn *espconn,
      espconn_sent_callback sent_cb)
```

The format of the callback function is:

```
void (*functionName)(void *arg)
```

The `arg` parameter is a pointer to a `struct espconn` that describes the connection.

See also:

- Sending and receiving TCP data
- struct espconn

espconn_regist_recvcb

Register a function to be called when data becomes available on the TCP connection or UDP datagram.

```
sint8 espconn_regist_recvcb(
      struct espconn *espconn,
      espconn_recv_callback recv_cb)
```

The format of the callback function is:

```
void (*functionName)(void *arg, char *pData, unsigned short len)
```

Where `args` is a pointer to a `struct espconn`, `pData` is a pointer to the data received and `len` is the length of the data received.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- Sending and receiving TCP data
- UDP
- espconn_create

espconn_sent

Send data through the connection to the partner.

```
sint8 espconn_sent(
      struct espconn *pEspconn,
      uint8 *pBuffer,
      uint16 length)
```

The `pEspconn` parameter identifies the connection through which to transmit the data.

The `pBuffer` parameter points to a data buffer to be transmitted.

The `length` parameter supplies the length of the data in bytes that is to be transmitted.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_MEM` – Out of memory
- `ESPCONN_ARG` – Illegal argument

See also:

- Sending and receiving TCP data
- UDP

ipaddr_addr

Build a TCP/IP address from a dotted decimal string representation.

```
unit32 ipaddr_addr(char *addressString)
```

Return an IP address (4 byte) value from a dotted decimal string representation supplied in the `addressString` parameter.

IP4_ADDR

Set the value of a variable to an IP address from its decimal representation.

```
IP4_ADDR(struct ip_addr * addr, a, b, c, d)
```

The `addr` parameter is a pointer to storage to hold an IP address. This may be an instance of `struct ip_addr`, a `uint32`, `uint8[4]`. It must be cast to a pointer to a `struct ip_addr` if not already of that type.

The parameters `a`, `b`, `c` and `d` are the parts of an IP address if it were written in dotted decimal notation.

Includes:

- ip_addr.h

See also:

- struct ip_addr

IP2STR

Generate four int values used in a `os_printf` statement

```
IP2STR(ip_addr_t *address)
```

This is a macro which takes a pointer to an IP address and returns four comma separated decimal values representing the 4 bytes of an IP address. This is commonly used in code such as:

```
os_printf("%d.%d.%d.%d\n", IP2STR(&addr));
```

TCP APIs

espconn_accept

Listen for an incoming TCP connection.

```
sint8 espconn_accept(struct espconn *espconn)
```

After calling this function, the ESP8266 starts listening for incoming connections. Any callback functions registered with `espconn_regist_connectcb()` will be invoked when new connections arrive.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_MEM` – Out of memory
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_ARG` – Illegal argument

See also:

- TCP
- espconn_regist_connectcb
- espconn_delete

espconn_get_connection_info

```
sint8 espconn_get_connection_info(
      struct espconn *espconn,
      remot_info **pcon_info,
      uint8 typeFlags)
```

The `espconn` is a pointer to the TCP control block.

The `pcon_info` parameter is the partner info.

The `typeFlags` defines what kind of partner we are getting information about:

- 0 – regular partner
- 1 – SSL partner

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

espconn_connect

Connect to a remote application using TCP.

```
sint8 espconn_connect(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_RTE` – Routing problem
- `ESPCONN_MEM` – Out of memory
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_ARG` – Illegal argument

See also:

- TCP
- espconn_disconnect

espconn_disconnect

Disconnect a TCP connection.

```
sint8 espconn_disconnect(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- TCP
- espconn_accept
- espconn_connect

espconn_regist_connectcb

Register a function that will be called when a TCP connection is formed.

```
sint8 espconn_regist_connectcb(
      struct espconn *espconn,
      espconn_connect_callback connect_cb)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

The callback function should have the following signature:

```
void (*functionName)(void *arg)
```

Where the `arg` parameter is a pointer to an `struct espconn` instance.

Question: Is this a NEW `struct espconn` or the original one?

See also:

- The ESPConn architecture
- espconn_accept

espconn_regist_disconcb

Register a function that will be called back after a disconnection.

```
sint8 espconn_regist_disconcb(
      struct espconn *espconn,
      espconn_connect_callback discon_cb)
```

The signature of the disconnect callback function is the same as the connect callback:

```
void (*functionName)(void *arg)
```

where `arg` is a `struct espconn` pointer.

See also:

- TCP
- The ESPConn architecture

espconn_regist_reconcb

Register a function that will be called when an error is detected.

```
sint8 espconn_regist_reconcb(
      struct espconn *espconn,
      espconn_reconnect_callback recon_cb)
```

The signature of the callback function is:

```
void (*functionName)(void *arg, sint8 err)
```

The `arg` parameter is a pointer to a `struct espconn`.

The `err` parameter is one of the following:

- ESPCONN_TIMEOUT
- ESPCONN_ABRT
- ESPCONN_RST
- ESPCONN_CLSD
- ESPCONN_CONN
- ESPCONN_HANDSHAKE
- ESPCONN_PROTO_MSG

See also:

- The ESPConn architecture
- TCP
- espconn_accept
- struct espconn

espconn_regist_write_finish

See also:

- The ESPConn architecture

espconn_set_opt

Define which options to turn on for a connection.

```
sint8 espconn_set_opt(
       struct espconn *espconn,
       uint8 opt)
```

This function should be called in an `espconn_connect_callback`. The espconn parameter is the control block for the connection that is to be modified.

The opt parameter is a bit encoding of flags that are to be set on. The opt parameter is an enum of type `espconn_option`:

Enum Name	Value
ESPCONN_REUSEADDR	0x01
ESPCONN_NODELAY	0x02
ESPCONN_COPY	0x04
ESPCONN_KEEPALIVE	0x08

Bits that are not set on are left unchanged from their current existing values.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- espconn_clear_opt
- espconn_set_keepalive
- espconn_get_keepalive

espconn_clear_opt

Define which options to turn off for a connection.

```
sint8 espconn_clear_opt(
      struct espconn *espconn,
      uint8 opt)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

The opt value is an enum of type `espconn_option`:

Enum Name	Value
ESPCONN_REUSEADDR	0x01
ESPCONN_NODELAY	0x02
ESPCONN_COPY	0x04
ESPCONN_KEEPALIVE	0x08

See also:

- TCP Error handling
- espconn_set_opt
- espconn_set_keepalive
- espconn_get_keepalive

espconn_regist_time

Define an idle connection timeout value.

```
sint8 espconn_regist_time(
      struct espconn *espconn,
      unit32 interval,
      uint8 typeFlag)
```

If a connection is idle for a period of time, the ESP8266 is configured to automatically close the connection. It appears that the default is 10 seconds.

The `espconn` parameter describes the connection that is to have its timeout changed.

The `interval` parameter defines the timeout interval in seconds. The maximum value is 7200 seconds (2 hours).

The `typeFlag` parameter can be 0 to indicate that all connections are to be changed or 1 to set just this connection.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

espconn_set_keepalive

sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void *optArg)

espconn_get_keepalive

sint8 espconn_get_keepalive(struct espconn *espconn, unit8 level, void *optArg)

???

espconn_secure_accept

Listen for an incoming SSL TCP connection

```
sint8 espconn_secure_accept(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_MEM` – Out of memory
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_ARG` – Illegal argument

espconn_secure_set_size

espconn_secure_get_size

espconn_secure_connect

espconn_secure_sent

espconn_secure_disconnect

Secure TCP disconnection.

espconn_tcp_get_max_con

Return the maximum number of concurrent TCP connections.

uint8 espconn_tcp_get_max_con()

espconn_tcp_set_max_con

Set the maximum number of concurrent TCP connections

sint8 espconn_tcp_set_max_con(uint8 num)

espconn_tcp_get_max_con_allow

Get the maximum number of TCP clients allowed to connect inbound.

espconn_tcp_set_max_con_allow

Set the maximum number of TCP clients allowed to connect inbound.

espconn_recv_hold

Suspend receiving TCP data.

espconn_recv_unhold

Unblock receiving TCP data.

UDP APIs

espconn_create

Create a UDP control block in preparation for sending datagrams.

```
sint8 espconn_create(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_MEM` – Out of memory

See also:

- UDP
- espconn_regist_sentcb
- espconn_regist_recvcb
- espconn_sent
- espconn_delete

espconn_igmp_join

Join a multicast group.

espconn_igmp_leave

Leave a multicast group.

ping APIs

ping_start

```
bool ping_start(struct ping_option *ping_opt)
```

Includes:

- ping.h

See also:

- Ping request
- struct ping_option

ping_regist_recv

```
bool ping_regist_recv(struct ping_option *ping_opt, ping_recv_function ping_recv)
```

Register a function that will be called when a ping is received. The signature of the function is:

```
void (*functionName)(void* pingOpt, void *pingResp)
```

The parameters passed in are `pingOpt` which is a pointer to the `struct ping_option` and `pingResp` which is a pointer to a `struct ping_resp`.

Includes:

- ping.h

See also:

- Ping request
- struct ping_option
- struct ping_resp

ping_regist_sent

```
bool ping_regist_sent(struct ping_option *ping_opt, ping_sent_function ping_sent)
```

Register a function that will be called when a ping is sent. The signature of the function is:

```
void (*functionName)(void* pingOpt, void *pingResp)
```

The parameters passed in are `pingOpt` which is a pointer to the `struct ping_option` and `pingResp` which is a pointer to a `struct ping_resp`.

Includes:

- ping.h

See also:

- Ping request
- struct ping_option

mDNS APIs

espconn_mdns_init

void espconn_mdns_init(struct mdns_info *info)

espconn_mdns_close

void espconn_mdns_close()

espconn_mdns_server_register

void espconn_mdns_server_register()

espconn_mdns_server_unregister

void espconn_mdns_server_unregister()

espconn_mdns_get_servername

char *espconn_mdns_get_servername()

espconn_mdns_set_servername

char *espconn_mdns_set_servername()

espconn_mdns_set_hostname

void espconn_mdns_set_hostname(char *name)

espconn_mdns_get_hostname

char *espconn_mdns_get_hostname()

espconn_mdns_disable

void espconn_mdns_disable()

espconn_mdns_enable

void espconn_mdns_enable()

GPIO

Pin names are:

- PERIPHS_IO_MUX_GPIO0_U
- PERIPHS_IO_MUX_GPIO2_U
- PERIPHS_IO_MUX_MTDI_U
- PERIPHS_IO_MUX_MTCK_U // GPIO 13
- PERIPHS_IO_MUX_MTMS_U // GPIO 14

Pin Name	Function 1	Function 2	Function 3	Function 4	Physical pin
MTDI_U	MTDI	I2SI_DATA	HSPIQ MISO	GPIO12	10
MTCK_U	MTCK	I2SI_BCK	HSPID MOSI	GPIO13	12
MTMS_U	MTMS	I2SI_WS	HSPICLK	GPIO14	9
MTDO_U	MTDO	I2SO_BCK	HSPICS	GPIO15	13
U0RXD_U	U0RXD	I2SO_DATA		GPIO3	25
U0TXD_U	U0TXD	SPICS1		GPIO1	26
SD_CLK_U	SD_CLK	SPICLK		GPIO6	21
SD_DATA0_U	SD_DATA0	SPIQ		GPIO7	22
SD_DATA1_U	SD_DATA1	SPID		GPIO8	23
SD_DATA2_U	SD_DATA2	SPIHD		GPIO9	18
SD_DATA3_U	SD_DATA3	SPIWP		GPIO10	19
SD_CMD_U	SD_CMD	SPICS0		GPIO11	20
GPIO0_U	GPIO0	SPICS2			15
GPIO2_U	GPIO2	I2SO_WS	U1TXD		14
GPIO4_U	GPIO4	CLK_XTAL			16
GPIO5_U	GPIO5	CLK_RTC			24

Pin functions are:

- FUNC_GPIO0
- FUNC_GPIO12

- FUNC_GPIO13
- FUNC_GPIO14
- FUNC_GPIO15
- FUNC_U0RTS
- FUNC_GPIO3
- FUNC_U0TXD
- FUNC_GPIO1
- FUNC_SDCLK
- FUNC_SPICLK
- FUNC_SDDATA0
- FUNC_SPIQ
- FUNC_U1TXD
- FUNC_SDDATA1
- FUNC_SPID
- FUNC_U1RXD
- FUNC_SDATA1_U1RXD
- FUNC_SDDATA2
- FUNC_SPIHD
- FUNC_GPIO9
- FUNC_SDDATA3
- FUNC_SPIWP
- FUNC_GPIO10
- FUNC_SDCMD
- FUNC_SPICS0
- FUNC_GPIO0
- FUNC_GPIO2
- FUNC_U1TXD_BK
- FUNC_U0TXD_BK
- FUNC_GPIO4

- FUNC_GPIO5
- LED_GPIO_FUNC

PIN_PULLUP_DIS

Disable pin pull-up

PIN_PULLUP_DIS(PIN_NAME)

See also:

- GPIOs

PIN_PULLUP_EN

Enable pin pull-up

PIN_PULLUP_EN(PIN_NAME)

See also:

- GPIOs

PIN_FUNC_SELECT

Set the function of a specific pin.

PIN_FUNC_SELECT(PIN_NAME, FUNC)

See also:

- GPIOs

GPIO_ID_PIN

Get the id of a logical pin.

`GPIO_ID_PIN(pinNum)`

Convert a logical pin number into the identity of a pin. This is an interesting function as `GPIO_ID_PIN(x)` is coded to equal "`x`". The question now becomes whether or not one still needs to code `GPIO_ID_PIN()` when accessing GPIO functions.

GPIO_OUTPUT_SET

Set the output value of a specific pin.

`GPIO_OUTPUT_SET(GPIO_NUMBER, value)`

This is a helper macro that invokes `gpio_output_set()`. Take care when passing in a value that is part of an expression such as `pData=='1'`. The value is evaluated a number of times so should not have side-effects. There is also a current bug related to operator precedence … it is strongly recommended to place the value in extra parenthesis when coding. For example:

```
GPIO_OUTPUT_SET(GPIO_NUMBER, (pData=='1'))
```

Includes:

- gpio.h

See also:

- GPIOs

GPIO_DIS_OUTPUT

Set the pin to be input (disabled output).

```
GPIO_DIS_OUTPUT(GPIO_NUMBER)
```

This is a helper macro that invokes `gpio_output_set()`.

Includes:

- gpio.h

See also:

- GPIOs

GPIO_INPUT_GET

Read the value of the pin.

```
GPIO_INPUT_GET(GPIO_NUMBER)
```

This is a helper macro that invokes `gpio_input_get()`.

Includes:

- gpio.h

See also:

- gpio_input_get

gpio_output_set

Change the values of GPIO pins in one operation.

```
void gpio_output_set(
      uint32 set_mask,
      unit32 clear_mask,
      uint32 enable_output,
      unit32 enable_input)
```

The parameters are:

- `set_mask` – Bits with a "1" are set high, bits with a "0" are left unchanged.
- `clear_mask` – Bits with a "1" are set low, bits with a "0" are left unchanged
- `enable_output` – Bits with a "1" are set to output
- `enable_input` – Bits with a "1" are set to input

Includes:

- gpio.h

See also:

- GPIOs

gpio_input_get

Get the values of the GPIOs.

```
unit32 gpio_input_get()
```

Retrieve the values from the GPIOs and return a bitmask of their values.

Includes:

- gpio.h

See also:

- GPIOs

gpio_intr_handler_register

Register a callback function that will be invoked when a GPIO interrupt occurs.

```
void gpio_intr_handler_register(
      gpio_intr_handler_fn_t callbackFunction,
      void *arg)
```

The signature of the handler function must be:

```
void (*functionName)(uint32 interruptMask, void *arg)
```

Includes:

- gpio.h

gpio_pin_intr_state_set

```
void gpio_pin_intr_state_set(
      uint32 pinId,
      GPIO_INT_TYPE intr_state)
```

The `pinId` is the GPIO pin id value returned from `GPIO_ID_PIN(num)`.

The `intr_state` parameter defines what triggers the interrupt.

Includes:

- gpio.h

See also:

- GPIOs
- GPIO_INT_TYPE

gpio_intr_pending

Obtain the set of pending interrupts

```
uint32 gpio_intr_pending()
```

Includes:

- gpio.h

gpio_intr_ack

Flag a set of interrupts as having been handled. This should be called from an interrupt handler function.

```
void gpio_intr_ack(uint32 ack_mask)
```

Includes:

- gpio.h

gpio_pin_wakeup_enable

Define that the device can wakeup from light-sleep mode when an IO interrupt occurs.

```
void gpio_pin_wakeup_enable(
       uint32 pin,
       GPIO_INT_TYPE intr_state)
```

The `pin` parameter defines the pin number used to wake the device.

The `intr_state` defines which type of transition will wake the device. The choices are:

- GPIO_PIN_INTR_LOLEVEL
- GPIO_PIN_INTR_HILEVEL

Includes:

- gpio.h

See also:

- GPIOs

- GPIO_INT_TYPE

gpio_pin_wakeup_disable

void gpio_pin_wakeup_disable()

Includes:

- gpio.h

UART APIs

These functions have to be compiled in from the uart files in driver_lib.

uart_init

```
void uart_init(UartBautRate uart0BaudRate, UartBautRate uart1BaudRate)
```

There appears to be a typo in the data type … but likely we will be stuck with that now. The `UartBautRate` is an enum that contains:

- BIT_RATE_9600
- BIT_RATE_19200
- BIT_RATE_38400
- BIT_RATE_57600
- BIT_RATE_74880
- BIT_RATE_115200
- BIT_RATE_230400
- BIT_RATE_460800
- BIT_RATE_921600

See also:

- Working with serial

uart0_tx_buffer

Transmit a buffer of data via UART0

```
void uart0_tx_buffer(uint8 *buffer, uint16 length)
```

Transmit the data pointed to by the buffer for the given length.

See also:

- Working with serial

uart0_rx_intr_handler

Handle the receiving of data via UART0.

```
void uart0_rx_intr_handler(void *parameter)
```

The parameter is a pointer to a `RcvMsgBuff` structure. My best guess on how to use this function is to create it in `user_main.c` and its mere existence will cause it to be invoked at the appropriate time.

See also:

- Working with serial

I2C Master APIs

These functions have to be compiled in from the i2c_master files in driver_lib.

i2c_master_gpio_init

void i2c_master_gpio_init()

i2c_master_init

void i2c_master_init()

i2c_master_start

void i2c_master_start()

i2c_master_stop

void i2c_master_stop()

i2c_master_send_ack

void i2c_master_send_ack()

i2c_master_send_nack

void i2c_master_send_nack()

i2c_master_checkAck

bool i2c_master_checkAck()

i2c_master_readByte
uint8 i2c_master_readByte()

i2c_master_writeByte
void i2c_master_writeByte(uint8 wrdata)

i2c_master_setAck
void i2c_master_setAck(uint8 level)

i2c_masetr_getAck
uint8 i2c_master_getAck()

SPI APIs
These functions have to be compiled in from the SPI files in driver_lib.

cache_flush

spi_lcd_9bit_write

spi_mast_byte_write

spi_byte_write_espslave

spi_slave_init

spi_slave_isr_handler

hspi_master_readwrite_repeat

spi_test_init

PWM APIs

pwm_init

Initialize PWM.

```
void pwm_init(
      uint32 period,
      uint32 *duty,
      uint32 num_pwm_channels,
      uint32 (*pin_info_list)[3])
```

The `period` parameter is the PWM period. The value is measured in microseconds with a minimum value of 1000 giving a 1KHz period (there are 1000 periods of 1000 microseconds in a second).

The `duty` parameter is the duty ration of each PWM channel.

The `num_pwm_channels` is the number of PWM channels being defined.

The `pin_info_list` is a pointer to an array of `num_pwm_channels * 3` instances of unit32s that provides the PWM pin mappings.

See also:

- Pulse Width Modulation – PWM
- pwm_set_duty
- pwm_set_period
- pwm_start

pwm_start

```
void pwm_start()
```

After configuring the parameters for PMW, this function can be called.

See also:

- Pulse Width Modulation – PWM

pwm_set_duty

```
void pwm_set_duty(uint32 duty, uint8 channel)
```

The resolution of a duty step is 45 nanoseconds. Here we can set the number of duty steps in a cycle. For example, imagine we have a period of 1KHz. This means that 1 cycle is 1000 microseconds. If we want the duty cycle to be 50%, then the output has to be high for 500 microseconds. 500 microseconds is 11111 units of 45 nanoseconds and that would become the duty value. Formulaically, the duty ratio is (duty * 45) / (period *1000).

The `duty` parameter supplies the number of 45 nanosecond intervals that the output will be high in one period.

The `channel` parameter specifies which of the PWM channels is being changed.

See also:

- Pulse Width Modulation – PWM
- pwm_get_duty
- pwm_init

pwm_get_duty

```
uint32 pwm_get_duty(uint8 channel)
```

Get the duty value of the specified channel.

See also:

- Pulse Width Modulation – PWM
- pwm_get_duty
- pwm_init

pwm_set_period

Set the period for PWM operations.

```
void pwm_set_period(uint32 period)
```

The `period` parameter is the PWM period. The value is measured in microseconds with a minimum value of 1000 giving a 1KHz period (there are 1000 periods of 1000 microseconds in a second).

See also:

- Pulse Width Modulation – PWM
- pwm_get_period
- pwm_init

pwm_get_period

```
uint32 pwm_get_period()
```

Get the current setting of the PWM period.

See also:

- Pulse Width Modulation – PWM
- pwm_set_period
- pwm_init

get_pwm_version

```
uint32 get_pwm_version()
```

See also:

- Pulse Width Modulation – PWM

set_pwm_debug_en(uint8 print_en)

Used to enable or disable debug print.

Bit twiddling

- BIT(b) – The 2^b value

ESP Now

esp_now_init

esp_now_deinit

esp_now_register_recv_cb

esp_now_unregister_recv_cb

esp_now_send

esp_now_add_peer

esp_now_del_peer

esp_now_set_self_role

esp_now_get_self_role

esp_now_set_peer_role

esp_now_get_peer_role

esp_now_set_peer_key

esp_now_get_peer_key

Mystery

- ets_wdt_enable – perhaps enables the watch dog timer
- ets_wdt_disable – perhaps disables the watch dog timer
- ESP_DBG
- system_mktime
- atoi

Data structures

station_config

A description of a station configuration. Contains the following fields:

- `uint8 ssid[32]` – The SSID of the access point.
- `uint8 password[64]` – The password to access the access point.
- `uint8 bssid_set` – Flag to indicate whether or not to use the `bssid` property. A value of 1 means to use and a value of 0 means to not use.
- `uint8 bssid[6]` – If several access points have the same SSID, BSSID can contain a MAC address to indicate which of the access points to connect to.

See also:

- Station configuration
- wifi_station_get_config
- wifi_station_get_config_default
- wifi_station_set_config
- wifi_station_set_config_current

struct softap_config

Configuration control structure for softAP.

- uint8 ssid[32]
- uint8 password[64]
- uint8 ssid_len – The length of the SSID. If 0, then the ssid is null terminated.
- `uint8 channel` – The channel to be used for communication. Values are 1 to 13.
- `uint8 authmode` – The authentication mode required. AUTH_WEP is not supported.
- `uint8 ssid_hidden` – Whether or not this SSID is hidden. A value of 1 makes it hidden.
- `uint8 max_connection` – The maximum number of station connections. The maximum and default is 4.
- `uint16 beacon_interval` – The beacon interval in milliseconds. Values are 100 – 60000.

See also:

- wifi_softap_get_config
- wifi_softap_get_config_default
- wifi_softap_set_config
- wifi_softap_set_config_current

struct station_info

This structure provides information on the stations connected to an ESP8266 while it is an access point. It is a linked list with properties:

- uint8 bssid[6] – The ???
- `struct ipaddr ip` – The IP address of the connected station

To get the next entry, we can use `STAILQ_NEXT(pStationInfo, next).`

See also:

- Being an access point
- wifi_softap_get_station_info
- wifi_softap_free_station_info

struct dhcps_lease

This structure is used by the `wifi_softap_dhcps_lease()` function to define the start and end range of available IP addresses.

The fields contained within are:

- struct ip_addr start_ip
- struct ip_addr end_ip

Includes:

- user_interface.h

See also:

- The DHCP server
- wifi_softap_set_dhcps_lease

struct bss_info

This structure contains:

- STAILQ_ENTRY(bss_info) next
- uint8 bssid[6]
- uint8 ssid[32]
- uint8 channel
- `sint8 rssi` – The received signal strength indication
- AUTH_MODE authmode
- uint8 is_hidden
- sint16 freq_offset

To get the next entry, we can use `STAILQ_NEXT(pBssInfoVar, next).`

The AUTH_MODE is an enum

- `AUTH_OPEN` – No authentication. No challenge on any station connect.
- AUTH_WEP = 1
- AUTH_WPA_PSK = 2
- AUTH_WPA2_PSK = 3
- AUTH_WPA_WPA2_PSK =4

See also:

- Scanning for access points

struct ip_info

This structure defines information about an interface possessed by the ESP8266. It contains the following fields:

- `struct ip_addr ip` – The IP address of the interface.
- `struct ip_addr netmask` – The netmask used by the interface.
- `struct ip_addr gw` – The IP address of the gateway used by the interface.

See also:

- wifi_get_ip_info
- wifi_set_ip_info
- IP4_ADDR

struct rst_info

Information about the current boot/restart

This structure contains:

- uint32 reason
- uint32 exccause
- uint32 epc1
- uint32 epc2
- uint32 epc3
- uint32 excvaddr
- uint32 depc

The `reason` field is an enum with the following values:

* 0 – Default restart – Normal startup on power on
* 1 – Watch dog timer – Hardware watchdog reset
* 2 – Exception – An exception was detected
* 3 – Software watch dog timer – Software watchdog reset
* 4 – Soft restart
* 5 – Deep sleep wake up

See also:

* Exception handling
* system_get_rst_info

struct espconn

This data structure is the representation of a connection between the ESP8266 and a partner. It contains the "control blocks" and identification information … however it is important to note that it is not always an opaque piece of data.

* `enum espconn_type type` – The type can be one of
    * `ESPCONN_TCP` – Identifies this connection as being of type TCP.
    * `ESPCONN_UDP` – Identifies this connection as being of type UDP.
* `enum espconn_state` – The state can be one of
    * ESPCONN_NONE
    * ESPCONN_WAIT
    * ESPCONN_LISTEN
    * ESPCONN_CONNECT
    * ESPCONN_WRITE
    * ESPCONN_READ
    * ESPCONN_CLOSE
* ```
  union {
              esp_tcp *tcp
              esp_udp *udp
  ```
  `} proto` – This field is a union of `tcp` and `udp` meaning that only one of them should ever be used for an instance of this data structure. If the data structure is used for TCP then the `tcp` property should be used while for UDP, the `udp` property should be used.
* `void *reverse` – In the comments, this is flagged as a field *reserved* for user code. It is possible the name chosen (*reverse*) is actually a typo in the header file!!
  ```

- Other fields … there are other fields in the structure but they are not meant to be read or written to by user applications. Ignore them. Using their values is undefined and may have unexpected effects.

See also:

- TCP
- esp_tcp
- esp_udp

esp_tcp

- uint8 local_ip[4]
- int local_port
- uint8 remote_ip[4]
- int remote_port
- Other fields … there are other fields in the structure but they are not meant to be read or written to by user applications. Ignore them. Using their values is undefined and may have unexpected effects.

See also:

- struct espconn

esp_udp

This data structure is used in the `proto` property of the `struct espconn` control block.

- int remote_port
- int local_port
- uint8 local_ip[4]
- uint8 remote_ip[4]

See also:

- UDP

struct ip_addr

A representation of an IP address.

It contains the following field:

- `uint32 addr` – The actual 4 byte IP address.

Includes:

- ip_addr.h

See also:

- ipaddr_addr
- IP4_ADDR
- ipaddr_t

ipaddr_t

A typedef for `struct ipaddr`.

See also:

- struct ip_addr

struct ping_option

The fields contained within the structure are:

- `uint32 count` – The number of times to transmit a ping
- `uint32 ip` – The IP address that is the target of the ping
- uint32 coarse_time
- ping_recv_function recv_function
- ping_sent_function sent_function
- void *reverse;

Includes:

- ping.h

See also:

- Ping request
- ping_start
- ping_regist_recv
- ping_regist_sent

struct ping_resp

The fields contained within the structure are:

- uint32 total_count
- uint32 resp_time
- uint32 seqno
- uint32 timeout_count
- uint32 bytes
- uint32 total_bytes

- uint32 total_time
- `sint8 ping_err` – An indication of whether or not an error occurred. A value of 0 means no error.

Includes:

- ping.h

See also:

- Ping request
- ping_start
- ping_regist_recv
- ping_regist_sent

enum phy_mode

The 802.11 physical mode to be used or being used.

- PHY_MODE_11B
- PHY_MODE_11G
- PHY_MODE_11N

See also:

- wifi_set_phy_mode
- wifi_get_phy_mode

GPIO_INT_TYPE

These are the possible triggers for an interrupt. This is an enum defined as follows:

- `GPIO_PIN_INTR_DISABLE` – Interrupts are disabled.
- `GPIO_PIN_INTR_POSEDGE` – Interrupt on a positive edge transition.
- `GPIO_PIN_INTR_NEGEDGE` – Interrupt on a negative edge transition.
- `GPIO_PIN_INTR_ANYEDGE` – Interrupt on any edge transition.
- `GPIO_PIN_INTR_LOLEVEL` – Interrupt when low.
- `GPIO_PIN_INTR_HILEVEL` – Interrupt when high.

See also:

- gpio_pin_wakeup_enable

System_Event_t

The event type contains:

- `uint32 event` – The type of event that occurred. Can be

- `EVENT_STAMODE_CONNECTED` – We have successfully connected to an access point.
  - `uint8[32] event_info.connected.ssid` – The SSID of the access point.
  - uint8 ssid_len
  - uint8[6] bssid
  - `event_info.connected.channel` – The channel used to connect to the access point.
- `EVENT_STAMODE_DISCONNECTED`
  - `uint8[6] event_info.disconnected.bssid`
  - `uint8[32] event_info.disconnected.ssid`
  - `uint8 ssid_len`
  - `uint8 event_info.disconnected.reason` – The reason is one of the following:
    - REASON_UNSPECIFIED = 1
    - REASON_AUTH_EXPIRE = 2
    - REASON_AUTH_LEAVE = 3
    - REASON_ASSOC_EXPIRE = 4
    - REASON_ASSOC_TOOMANY = 5
    - REASON_NOT_AUTHED = 6
    - REASON_NOT_ASSOCED = 7
    - REASON_ASSOC_LEAVE = 8
    - REASON_ASSOC_NOT_AUTHED = 9
    - REASON_DISASSOC_PWRCAP_BAD = 10
    - REASON_DISASSOC_SUPCHAN_BAD = 11
    - REASON_IE_INVALID = 13
    - REASON_MIC_FAILURE = 14
    - REASON_4WAY_HANDSHAKE_TIMEOUT = 15
    - REASON_GROUP_KEY_UPDATE_TIMEOUT = 16
    - REASON_IE_IN_4WAY_DIFFERS = 17
    - REASON_GROUP_CIPHER_INVALID = 18
    - REASON_PAIRWISE_CIPHER_INVALID = 19
    - REASON_AKMP_INVALID = 20

        - REASON_UNSUPP_RSN_IE_VERSION = 21
        - REASON_INVALID_RSN_IE_CAP = 22
        - REASON_802_1X_AUTH_FAILED = 23
        - REASON_CIPHER_SUITE_REJECTED = 24
        - REASON_BEACON_TIMEOUT = 200
        - REASON_NO_AP_FOUND = 201
    - `EVENT_STAMODE_AUTHMODE_CHANGE`
        - `event_info.auth_change.old_mode`
        - `event_info.auth_change.new_mode`
    - `EVENT_STAMODE_GOT_IP`
        - `event_info.got_ip.ip`
        - `event_info.got_ip.mask`
        - `event_info.got_ip.gw`
    - `EVENT_SOFTAPMODE_STACONNECTED`
        - `event_info.sta_connected.mac`
        - `event_info.sta_connected.aid`
    - `EVENT_SOFTAPMODE_STADISCONNECTED`
        - `event_info.sta_disconnected.mac`
        - `event_info.sta_disconnected.aid`
- `Event_Info_u event_info`

This is a C Union containing data that is available as a function of the event type.

    - `Event_StaMode_Connected_t connected`
    - `Event_StaMode_Disconnected_t disconnected`
    - `Event_StaMode_AuthMode_Change_t auth_change`
    - `Event_StaMode_Got_IP_t got_ip`
    - `Event_SoftAPMode_StaConnected_t sta_connected`
    - `Event_SoftAPMode_StaDisconnected_t sta_disconnected`

See also:

- wifi_set_event_handle_cb

STATUS

This is an enum defined as follows:

Enum Name	Value
OK	0
FAIL	1
PENDING	2
BUSY	3
CANCEL	4

See also:

- wifi_station_scan

Reference materials

There is a wealth of information available on the ESP8266 from a variety of sources.

ESPFS breakdown

The ESPFS is a library which stores "files" within the flash of the ESP8266 and allows an application to read them. It is part of the ESPHTTPD project.

EspFsInit

```
EspFsInitResult espFsInit(char *flashAddress)
```

Initialize the environment pointing to where the file data can be found. The return will be one of:

- ESPFS_INIT_RESULT_OK
- ESPFS_INIT_RESULT_NO_IMAGE
- ESPFS_INIT_RESULT_BAD_ALIGN

espFsOpen

```
EspFsFile *espFsOpen(char *fileName)
```

Open the file specified by the file name and return a structure that is the "handle" to the file or NULL if the file can not be found.

espFsClose

```
void espFsClose(EspFsFile *fileHandle)
```

Close the file that was previously opened by a call to espFsOpen(). No further reads should be performed.

espFsFlags

```
int espFsFlags(EspFsFile *fileHandle)
```

espFsRead

```
int espFsRead(EspFsFile *fileHandle, char *buffer, int length)
```

Read up to length bytes from the file and store them at the memory location pointed to by buffer. The actual number of bytes read is returned by the function call.

mkespfimage

This is not a function but a command which builds the binary data of the files to be placed in flash memory.

```
mkespfimage [-c compressor] [-l compression_level]
```

- -c
  - 0 – None
  - 1 – Heatshrink
- -l

o

ESPHTTPD breakdown

The ESPHTTPD library provides an implementation of an HTTP server running on an ESP8266. In order to use this, we may wish to understand it better.

httpdInit

```
void httpdInit(HttpdBuiltInUrl *fixedUrls, int port)
```

Initialize the HTTP server running in the ESP. The `port` parameter is the port number that the ESP will listen upon for incoming browser requests. The default port number used by browsers is 80.

The `HttpdBuiltInUrl` is a typedef that provides mapping to URLs available on the HTTP server. The fields contained within are:

- `char *url` – The url to match.
- `cgiSendCallback cgiCb` – The callback function to call when matched.
- `const void *cgiArg` – Parameters to pass into the callback function.

It is vital that the last element in the array have NULLs for all attributes. This serves as a termination record. Here is an example definition for a minimal set of built in URLs:

```
HttpdBuiltInUrl builtInUrls[]={
	{NULL, NULL, NULL}
};
```

The `cgiSendCallback` is a function with the following signature:

```
int (* functionName)(HttpdConnData *connData)
```

Includes:

- httpd.h

httpdGetMimetype

```
char *httpdGetMimeType(char *url)
```

Examine the `url` passed in and by looking at its file type, determine the MIME type of the data. If no file type is found, then the default MIME type is "`text/html`".

Includes:

- httpd.h

httpdUrlDecode

`int httpdUrlDecode(char *val, int valLen, char *ret, int retLen)`

Decode a URL according to URL decoding rules. The encoded url is supplied in `val` with a length of `valLen` bytes. The resulting decoded url string will be stored at `ret` with a maximum length of `retLen`. The actual length is returned by the function call itself.

Includes:

- httpd.h

httpdStartResponse

`void httpdStartResponse(HttpdConnData *conn, int code)`

Start sending the response data down the TCP connection to the browser. The `code` value is the primary browser response code.

Includes:

- httpd.h

httpdSend

`int httpdSend(HttpdConnData *conn, const char *data, int len)`

Send data to the browser through the TCP connection. The data is supplied as `data` and the `len` parameters is the number of bytes to write. If `len == -1`, then data is assumed to be a NULL terminated string.

Includes:

- httpd.h

httpdRedirect

`void httpdRedirect(HttpdConnData *conn, char *newUrl)`

Send an HTTP redirect instruction to the browser. The `newUrl` is the URL we wish the browser to use.

Includes:

- httpd.h

httpdHeader

`void httpdHeader(HttpdConnData *conn, const char *field, const char *val)`

Send an HTTP header. The name of the header is supplied in the `field` parameter and its value supplied in the `val` parameter.

Includes:

- httpd.h

httpdGetHeader

```
int httpdGetHeader(HttpdConnData *conn, char *header, char *ret, int retLen)
```

Search the browser supplied data header looking for a header that matches the `header` parameter. If found, return the header value at the buffer pointed to by `ret` which must be at least `retLen` bytes long.

Includes:

- httpd.h

httpdFindArg

```
int httpdFindArg(char *line, char *arg, char *buff, int buffLen)
```

Given a line of text, look for a parameter of the form "`name=value`" within the line. If the name matches our passed in name, then return the value.

Includes:

- httpd.h

httpdEndHeaders

```
void httpdEndHeaders(HttpdConnData *conn)
```

Conclude the output of headers to the output stream.

Includes:

- httpd.h

Makefiles

Books have been written on the language and use of Makefiles and our goal is not to attempt to rewrite those books. Rather, here is a cheaters guide to beginning to understand how to read them.

A general rule in a make file has the form:

```
target: prereqs …
       receipe ...
```

Variables are defined in the form:

name=value

We can use the value of a variable with either $(name) or ${name}.

Another form of definition is:

name:=value

Here, the value is locked to its value at the time of definition and will not be recursively expanded.

Some variables have well defined meanings:

Variable	Meaning
CC	C compiler command
AR	Archiver command
LD	Linker command
OBJCOPY	Object copy command
OBJDUMP	Object dump command

We can use the value of a previously defined variable in other variable definitions. For example:

```
XTENSA_TOOLS_ROOT ?= c:/Espressif/xtensa-lx106-elf/bin
CC             := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
```

defines the C compiler as an absolute path based on the value of a previous variable.

Special expansions are:

- `$@` - The name of the target
- `$<` - The first prereq

Comments are lines that start with an "`#`" character.

Wildcards are:

- `*` - All characters
- `?` - One character
- `[…]` - A set of characters

Make can be invoked recursively using

```
make -C <directoryName>
```

Imagine we wanted to build a list of source files by naming directories and the list of source files then becomes all the "`.c`" files, in those directories? How can we achieve that?

```
SRC_DIR = dir1 dir2
SRC := $(foreach sdir, $(SRC_DIR), $(wildcard $(sdir)/*.c))
OBJ := $(patsubst %.c, $(BUILD_BASE)/%.o, $(SRC))
```

The puzzle

Imagine a directory structure with

```
a
    a1.c
    a2.c
b
    b1.c
    b2.c
```

goal is to compile these to

```
build
    a
        a1.o
        a2.o
    b
        b1.o
        b2.o
```

We know how to compile x.c → x.o

MODULES=a b

BUILD_BASE=build

BUILD_DIRS=$(addprefix $(BUILD_BASE)/,$(MODULES))

SRC=$(foreach dir, $(MODULES), $(wildcard $(dir)/*.c))

Replace all x.c with x.o

OBJS=$(patsubst %.c,%.o,$(SRC))

all:

    echo $(OBJS)

    echo $(wildcard $(OBJS)/*.c)

    echo $(foreach dir, $(OBJS), $(wildcard $(dir)/*.c))

    echo "SRC: " $(SRC)

test: checkdirs $(OBJS)

```
	echo "Compiled " $(SRC)

.c.o:
	echo "Compiling $(basename $<)"
	$(CC) -c $< -o build/$(addsuffix .o, $(basename $<))

checkdirs: $(BUILD_DIRS)

$(BUILD_DIRS):
	mkdir -p $@

clean:
	rm -f $(BUILD_DIRS)
```

See also:

- GNU make

Forums

There are a couple of excellent places to ask questions, answer other folks questions and read about questions and answers of the past.

- Espressif ESP8266 BBS – A moderated forum run by Espressif. The primary source for SDK downloads and the source of much of the core materials.
- ESP8266 Community Forum – A set of fora dedicated to the ESP8266 run for and by the ESP8266 user community.

Reference documents

Espressif distributes PDF and Excel spreadsheets containing core information about the ESP8266. These can be downloaded freely from the web.

- 0A-ESP8266-Datasheet v4.3
- 0B-ESP8266 Hardware User Guide v1.1
- 0C-ESP8266 WROOM WiFi Module Datasheet v0.3
- 0D-ESP8266 Pin List Release 2014-11-15
- 2A-ESP8266 IOT SDK User Manual – Supplied with SDK
- 2B-ESP8266 SDK IOT Demo – Supplied with SDK
- 2C-ESP8266 SDK Programming Guide – Supplied with SDK
- 4A-ESP8266 AT Instruction Set – Supplied with SDK
- 4B-ESP8266 AT Command Examples – Supplied with SDK
- 4C-ESP8266 AT upgrade example
- 8A-ESP8266 Interface GPIO (Not yet published)
- 8B-ESP8266 Interface GPIO Registers Release 2014-11-15
- 8C-ESP8266 Interface I2C (Not yet published)

- 8D-ESP8266 Interface PWM v1.1
- 8E-ESP8266 Interface UART v0.2
- 8F-ESP8266 Interface UART Registers v0.1
- 8G-ESP8266 Interface Infrared Remote Control v0.3
- 8H-ESP8266 Interface SDIO SPI Mode (Not yet published)
- 8I-ESP8266 Interface SPI-WiFi Passthrough 1 – interrupt mode (Not yet published)
- 8J-ESP8266 Interface SPI-WiFi Passthrough 2 – interrupt mode (Not yet published)
- 8K-ESP8266 Sniffer Introduction v0.3
- 8L-ESP8266 Interface SPI Registers Release 2014-11-18
- 8M-ESP8266 Interface Timer Registers Release 2014-11-18
- 8N-ESP8266 SPI Reference v1.0
- 8O-ESP8266 SPI Overlap & Display Application Guide (Not yet published)
- 8Q-ESP8266 HSPI Host Multi-device API v1.0
- 9A-ESP8266 FRC Timer Introduction (not yet published)
- 9B-ESP8266 Sleep Function Description v1.0
- 20A-ESP8266 RTOS SDK Programming Guide V1.0.3
- 99A-ESP8266 Flash RW Operation v0.2
- 99B-ESP8266 Timer (not yet published)
- 99C-ESP8266 OTA Upgrade v1.6

Github

There are a number of open source projects built on top of and around the ESP8266 that can be found on Github. Here is a list of links to some of these projects that are very well worth having a look:

- EspressifApp
- eriksl/esp8266-universal-io-bridge
- CHERTS/esp8266-devkit
- ESPHTTPD project
  - Spritetm/esphttpd
  - Spritetm/libesphttpd

SDK

The Software Development Kit (SDK) is published by Espressif and is required to build C based applications. It contains vital documentation in the form of PDF that don't appear to be available elsewhere.

- ESP8266 SDK v1.2.0

Heroes

Within the ESP8266 user community there are individuals that I consider to have pushed the boundaries of knowledge further or have developed tools that dramatically improve working with the devices. I want to take a few moments and call out these good folks without whom all our ESP8266 travels would be harder:

Max Filippov - jcmvbkbc - GCC compiler for Xtensa

Web site: Github – https://github.com/jcmvbkbc

A compiler for C based on GCC that compiles to Xtensa binary for flashing. It is doubtful that any useful work could be performed without this contribution.

Mikhail Grigorev – CHERTS – Eclipse for ESP8266 development

Web site: Project Unofficial Development Kit for Espressif ESP8266
Web site: Github – CHERTS/esp8266-devkit

An extraordinarily well polished set of artifacts and instructions for building ESP8266 C applications within the Eclipse development environment.

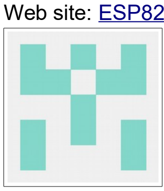
Ivan Grokhotkov – igrr – Arduino IDE for ESP8266 development

Web site: Github – esp8266/Arduino

An implementation of technology that allows one to develop ESP8266 applications using the Arduino IDE as well as libraries that map Arduino functions to ESP8266 equivalents or near equivalents.

Sprite_tm – HTTP server for ESP8266

Web site: ESP8266 Community dedicated forum



An implementation of an HTTP server that runs within an ESP8266 capable of serving up web pages.

Areas to Research

- Hardware timers … when do they get called?
- If I define functions in a library called libcommon.a, what is added to the compiled application when I link with this library? Is it everything in the library or just the object files that are referenced?
- What are the functions called "NOW"?
- What is the memory map/layout of the ESP8266?
- How much RAM is installed and available for use?