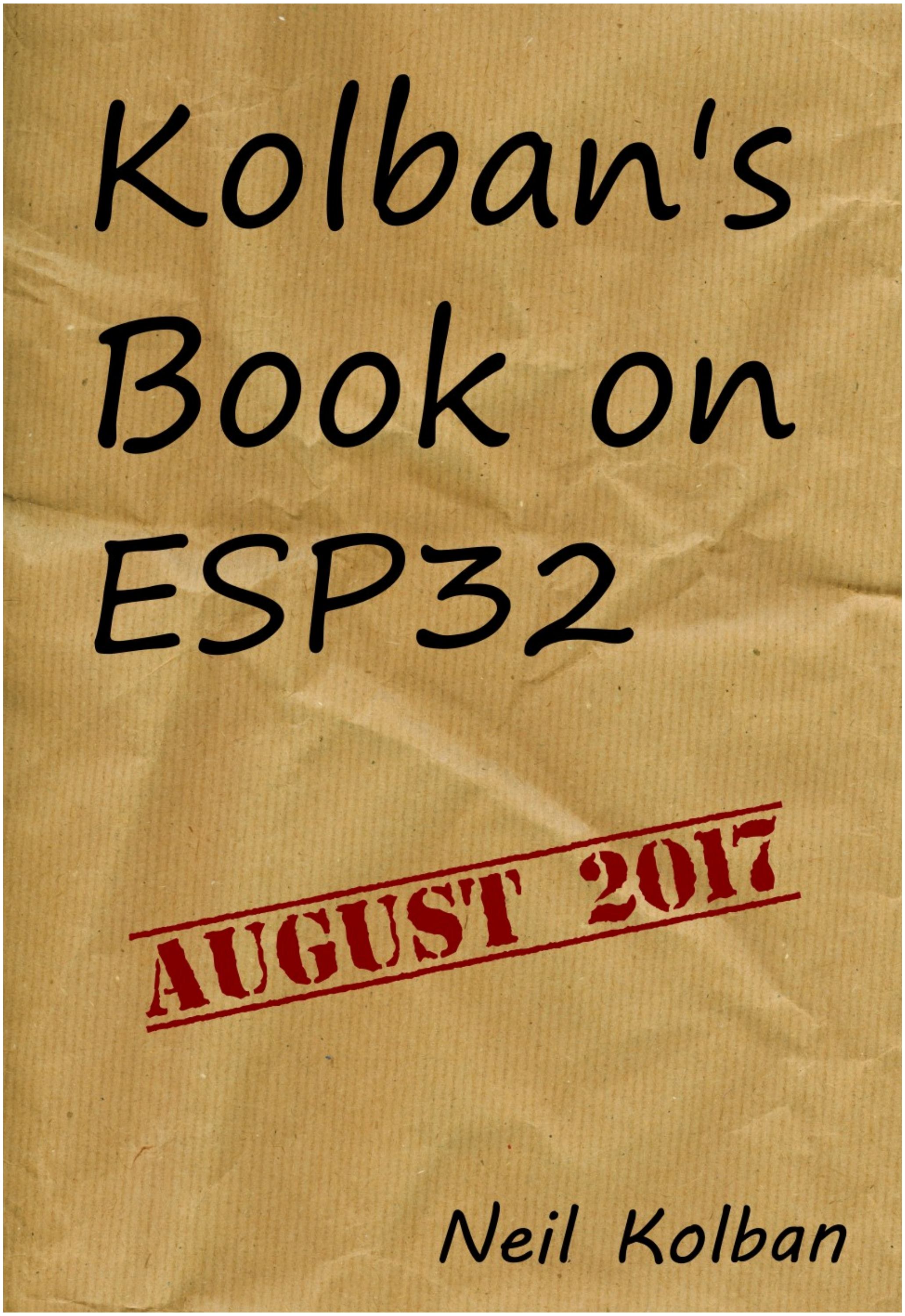


Kolban's Book on ESP32

AUGUST 2017

Neil Kolban

Table of Contents

Introduction

Howdy Folks,

I've 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 in early 2015, it piqued 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. About a year later, the ESP32 found its way into my possession.

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 ESP32 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 so please check back at the book's home page for the latest.

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 ESP32 minded enthusiasts, hobbyists and professionals.

Neil Kolban

Texas, USA

Important Documentation Notes – ESP8266 and ESP32

At the start of 2015 I released a book of notes on ESP8266 and the ESP32 was still a long way off. At the time we knew very little about the ESP32 and my initial thinking was that I would cover **both** the ESP8266 and the ESP32 in one book. Now that we have the ESP32 it has become obvious that the distinctions between that device and the ESP8266 are far too great to be accommodated in one document. If we tried to merge the two we would be forever reading that "this applies to ESP8266" and "that applies to ESP32". As such, this book is explicitly for the ESP32. However … there is currently a catch. Since there is indeed a goodly amount of material that is common to both devices, rather than start from scratch on a new book, I have taken the ESP8266 book as a base and am working through **this** book reworking it for ESP32 exclusivity. This is a work in progress. I have added the majority of what is new in ESP32 but haven't yet gotten to removing much of what is ESP8266 specific. As such, when your read this work, you will have to pay attention because some of the story is ESP8266 only and will either be removed or reworked over the period ahead.

Overview

A micro controller 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 micro controllers 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 micro controllers 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, few 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 micro controller 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 micro controller 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 ESP32 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 ESP32 modules around the $6 price point.

The ESP32

The ESP32 is the name of a micro controller designed by Espressif Systems. Espressif is a Chinese company based out of Shanghai. The ESP32 advertises itself as a self-contained WiFi networking solution offering itself as a bridge from existing micro controllers to WiFi … and … is also capable of running self contained applications.

Volume production of the ESP32 didn't start until the late of 2016 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 ESP32 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, 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 but from a programming perspective, they are all the same.

The ESP32 specification

When we look at a new electronic device, we are always interested in its specifications. This is the set of characteristics as described by the manufacturer. Sometimes it is immediately obvious to us what a specification item means and for others, it takes some time to appreciate its ramifications. Here is a summary list of the core ESP32 items:

Attribute	**Details**
Voltage	3.3V
Current consumption	Unknown
Flash memory attachable	Module based
Processor	Tensilica L108 32 bit
Processor speed	Dual 160MHz
RAM	520K
GPIOs	34
Analog to Digital	7
802.11 support	11b/g/n/e/i
Bluetooth	BLE
Maximum concurrent TCP connections	16
SPI	3
I2S	2
I2C	2
UART	3

The ESP32 is a dual core processor running the Xtensa LX6 instructions. The cores are called "`PRO_CPU`" and "`APP_CPU`".

The question of determining how long an ESP32 can run on batteries is an interesting one. The current consumption is far from constant. When transmitting at full power, it can consume 260mA but when in a deep sleep, it only need 20uA. That is quite a difference. This means that the run-time of an ESP32 on a fixed current reservoir is not just a function of time but also of what it is doing during that time … and that is a function of the program deployed upon it.

The ESP32 is designed to be used with a partner memory module and this is most commonly flash memory. Most of the modules come with some flash associated with them. Realize that flash has a finite number of erases per page before something fails. They are rated at about 10,000 erases. This is not normally an issue for configuration change writes or daily log writes … but if your application is continually writing new data extremely fast, then this may be an issue and your flash memory will fail.

Modules

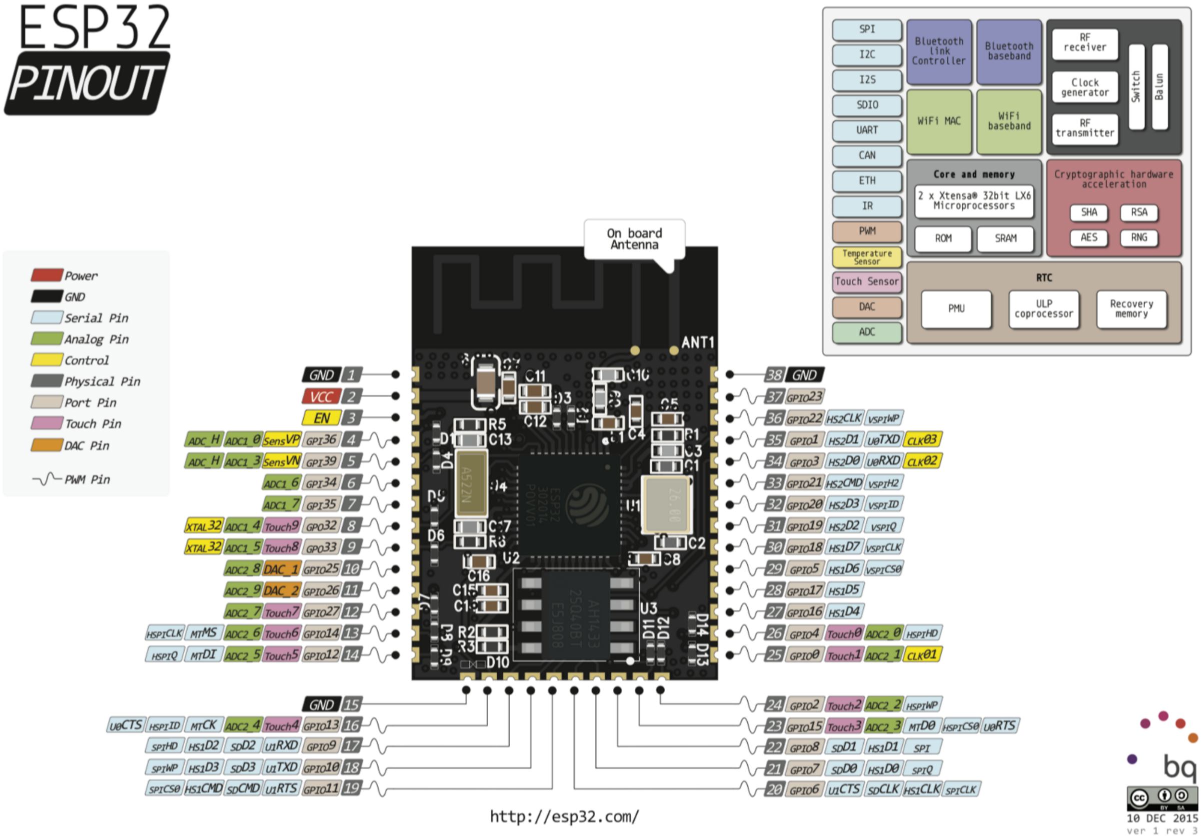
At the time of writing, the ESP32 has only recently become generally available, as such there are still relatively few modules to be bought.

ESP-WROOM-32

A module called the ESP-WROOM-32 is available from Espressif that contains an ESP32 plus accompanying supporting hardware such as flash memory. Don't mistake this for a breadboard friendly device … this is much more oriented to those with good electronics skills that wish to embed an ESP32 in a specific project. The module is only 18mm x 25.5mm and has pin spacings at the 1.27mm pitch. This is an extremely small spacing.

The fantastic web site called PIGHIXXX provides the most top quality pin-out images I have ever seen. Please visit their site. They have pin-outs for almost every conceivable device I have ever wanted to know about.

Here is a schematic of the current device:



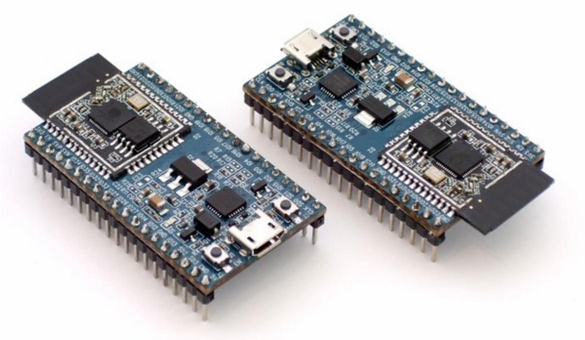
The WROOM-32 uses 6 GPIO pins for driving the external flash and these must **not** be used for other purposes. They are off limits. The pins are GPIO6, GPIO7, GPIO8, GPIO9, GPIO10 and GPIO11.

See also:

- ESP-WROOM-32 Datasheet
- Adafruit – seller of devices in US
- ESP-WROOM-32 Home Page

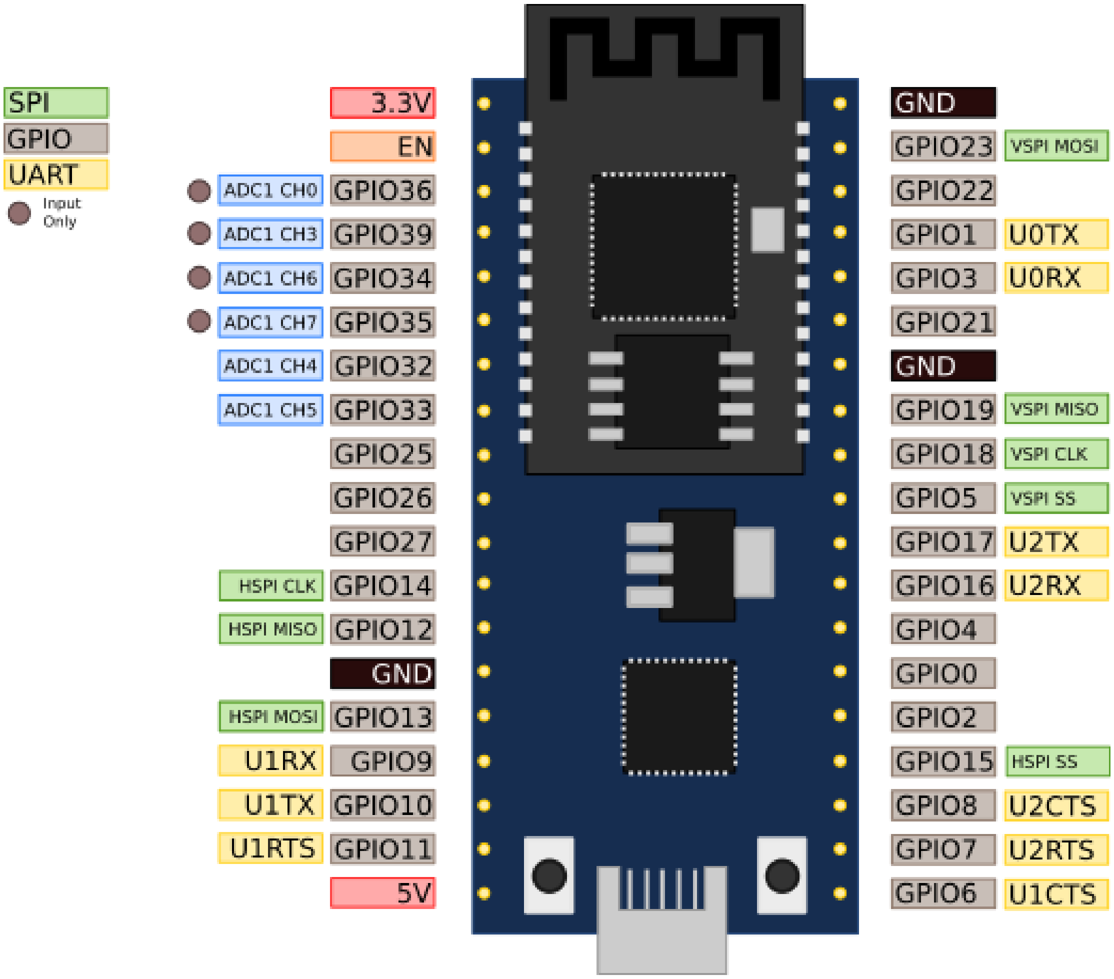
ESP32-DevKitC

With the release of the ESP32, Espressif have released their own module for exposing the ESP32 to more consumers. The board they have produced is called the "ESP32-DevKit" and is considered bread-board friendly.



The board contains headers for the ESP32 as well as a micro USB adapter and two buttons called enable and boot. These buttons can be used to "flash" or "download" new application code into the module. To perform this task, hold the "`EN`" button down while pressing and releasing "`Boot`".

The pin out of the module is shown in the following image:



To reboot the device, pulse `EN` low. If `GPIO0` is high, the device will normal boot while if `GPIO0` is low, it will boot into flash mode allowing us to upload a new application into the device's flash storage.

See also:

- ESP32-DevKitC – Getting Started Guide
- ESP32-DevKitC home page
- Adafruit reseller

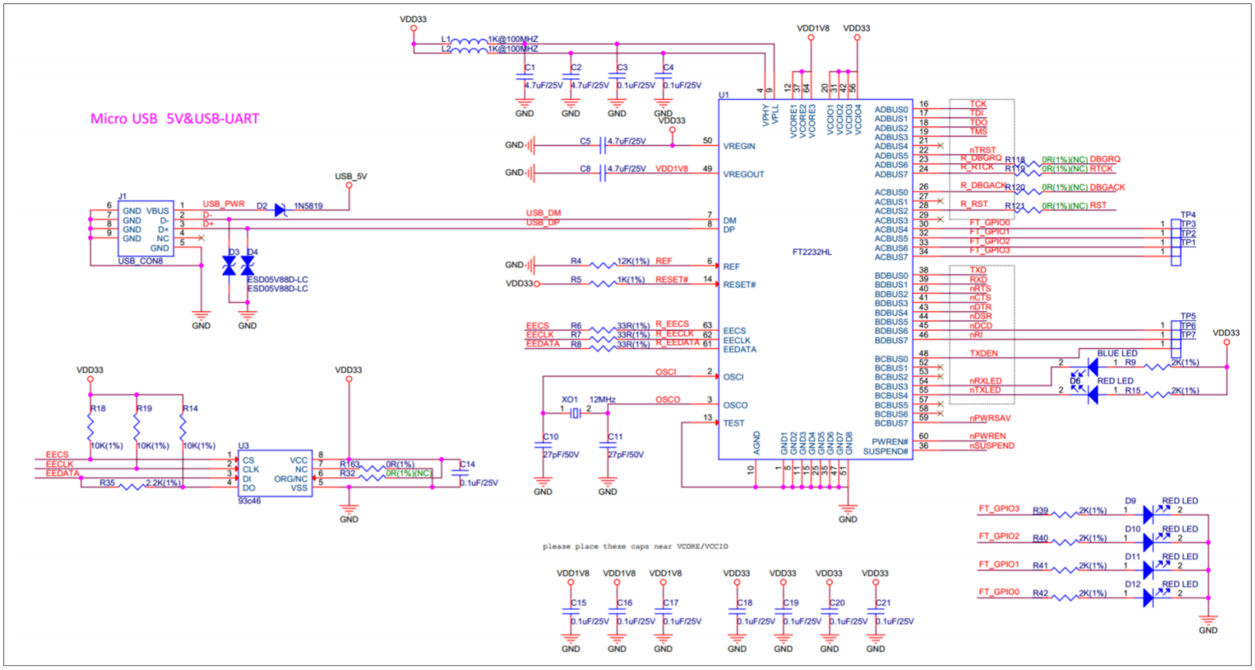
ESP-WROVER-KIT

The ESP-WROVER-KIT is the be-all and end-all of ESP32 development. It is pricier than the other modules coming in at about $50, however it is without question the most "robust" module currently available. It sports an LCD display, a socket for a camera, an on/off switch, a micro-sd connector, an RGB LED and many broken out connector pins.

The pins are broken out both on the front and rear.

This module is not at all bread-board friendly but that should be inherently understood from its purpose.

Here is the schematic of the USB interface:



The LED anode pins are connected to IO2, IO0 and IO4.

The camera is connected:

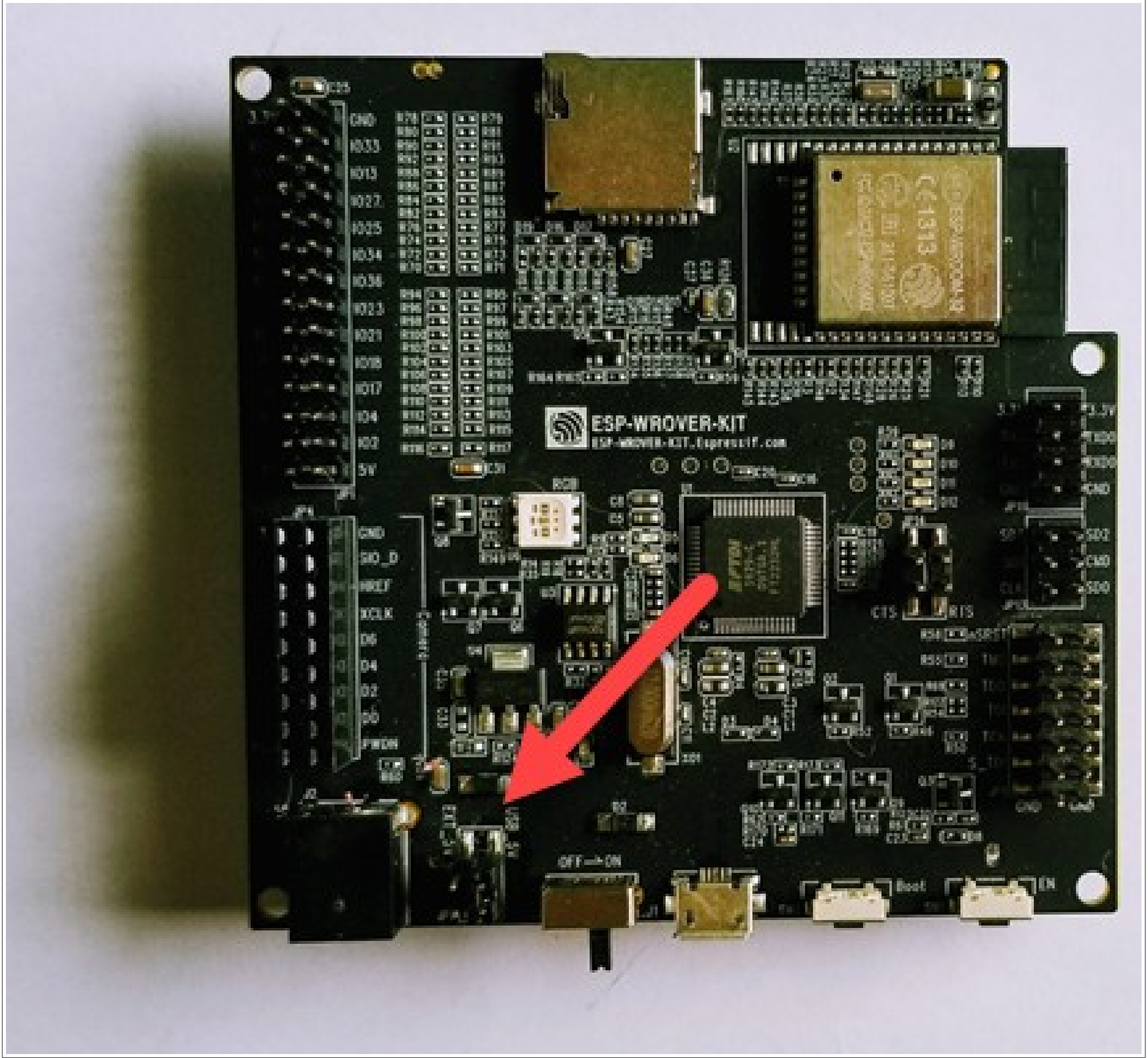
Function	Pin
SIO_C	IO27
SIO_D	IO26
VSYNC	IO25
HREF	IO23
PCLK	IO22
XCLK	IO21
RESET	IO2
D0	IO4
D1	IO5
D2	IO18
D3	IO19
D4	IO36
D5	IO39
D6	IO34
D7	IO35
PWDN	GND

The LCD screen is SPI attached at:

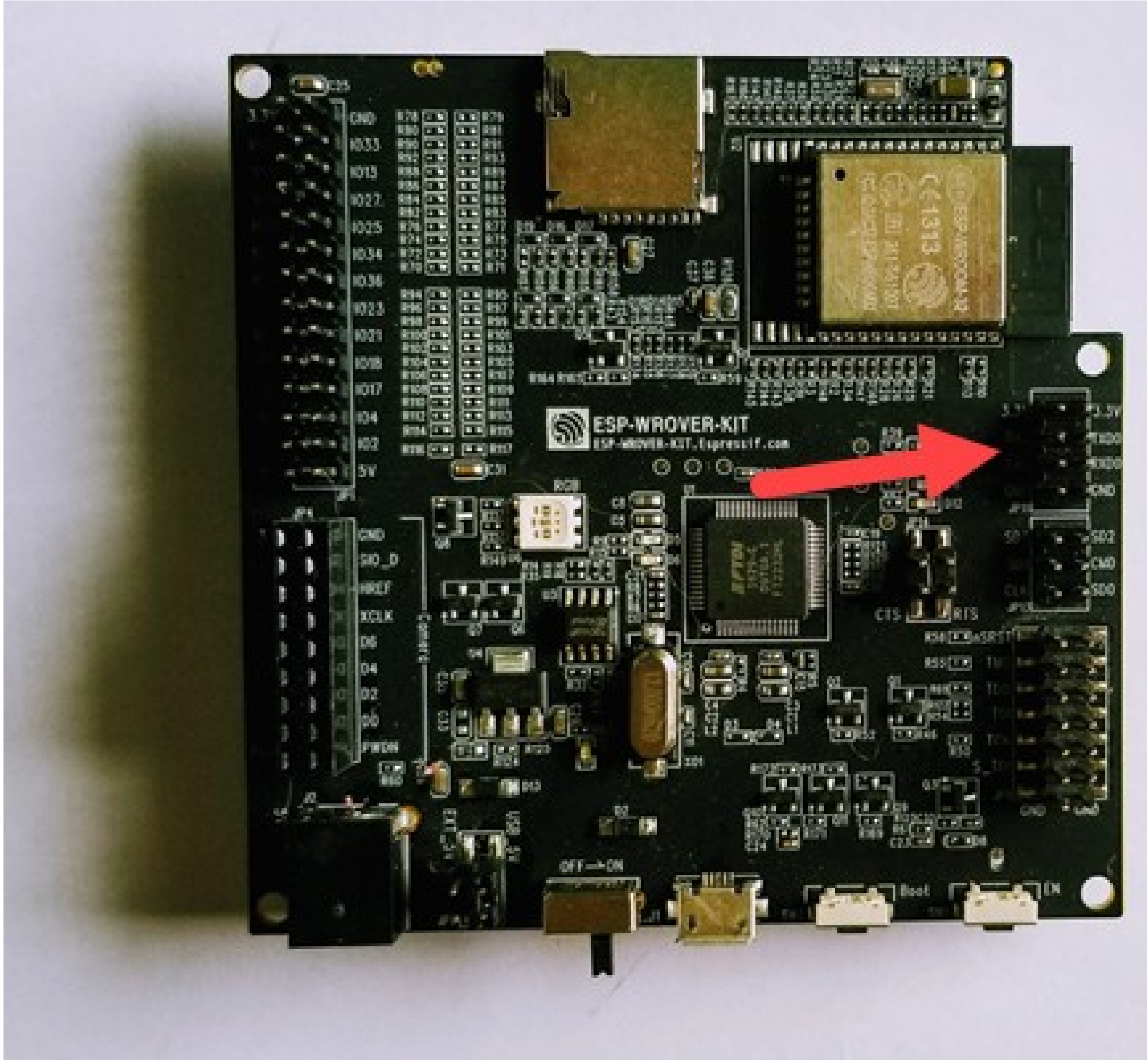
Function	Pin
RESET	IO18
CLK	IO19
D/C	IO21
CS	IO22
MOSI	IO23
MISO	IO25
Backlight?	IO5

There is an RGB led attached to GPIO 0, GPIO 2 and GPIO 4 (Blue)

The module can be powered from an external 2.5m power source (5V) or from USB. There is a jumper which **must** be in place to select which is enabled.



In addition, you should also enable the UARTs with two jumpers here:



There have been a number of releases of the WROVER board. Unfortunately, determining which board you have is not the easiest task because the identity isn't on the boad. The following are good indications:

- red board, female camera header – V1
- black (shiny) board, male camera header – V2
- black (matte) board, female camera header – V3

See also: ESP32 Modules and Boards for good pictures to aid in identification.

The USB device is an FTDI2232HL. This manifests as two serial ports. On Linux these are `/dev/ttyUSB0` and `/dev/ttyUSB1`. It is `/dev/ttyUSB1` that we wish to use for console and flashing.

Given that the WROVER's primary distinction between itself and the WROOM is the inclusion of the 4MBytes of psRAM we want to take advantage of this. At this early date (2017-07), we are required to use a distinct build environment that includes a custom compiler and a custom build of the ESP-IDF (Special instructions).

In summary, the ESP-IDF is:

https://github.com/espressif/esp-idf/tree/feature/psram_malloc

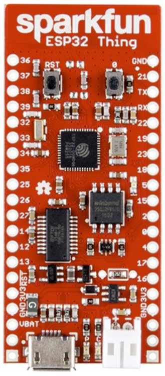
and the tool chain is:

See also:

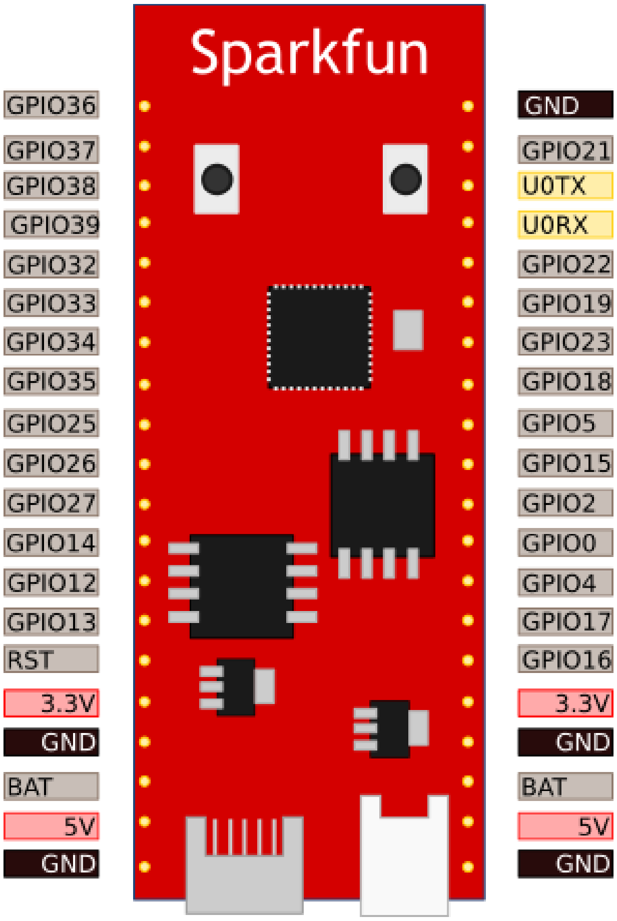
- ESP-WROVER-KIT Home Page
- ESP-WROVER-KIT Getting Started Guide
- Complete Schematic – V1
- Complete Schematic – V2
- Complete Schematic – V3
- Adafruit reseller
- FTDI – FT2232H home page

The SparkFun ESP32 thing

SparkFun make an ESP32 board called the "ESP32 Thing".



One of the distinguishing features of this board is the built in battery charger socket. We can use this board with a LIPO battery and charge that battery at the same time.



The width of the Sparkfun board allows it to sit in a breadboard with one row open either side of it. This makes it more convenient for quick prototyping.

See also:

- SparkFun ESP32 Thing

Connecting to the ESP32

The ESP32 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 ESP32 has a dedicated UART interface with pins labeled TX and RX. The TX pin is the ESP32 transmission (outbound from ESP32) and the RX pin is used to receive data (inbound into the ESP32). These pins can be connected to a

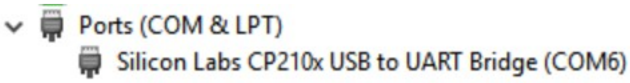
UART partner. By far the easiest and most convenient partner for us is a USB → UART converter. These are discussed in detail later in the book. For now let us assume that we have set those up. Through the UART, we can attach a terminal emulator to send keystrokes and have data received from the ESP32 displayed as characters on the screen. 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 ESP32 and its partner. During boot, the ESP32 attempts to automatically determine the baud rate of the partner and match it. It assumes a default of 115200 and if you have a serial terminal attached, you will see a message like:

```
???
```

if it is configured to receive at 115200.

When connected to a Windows 10 machine via micro USB, it shows up as a serial device:



Ports (COM & LPT)
Silicon Labs CP210x USB to UART Bridge (COM6)

The default serial baud rate is 115200.

Once the ESP32 is serially connected, we typically want to attach a monitor or terminal emulator against the device. The ESP-IDF framework provides a nice technique to achieve that. We can run the command "make monitor". This launches a monitor tool that tails the ESP32 serial output. This tool is based on "miniterm" which is part of the "PySerial" package. We can terminate miniterm with `CTRL+]`.

From a Linux environment, we see the serial port (usually) as `/dev/ttyUSB0`. If we install the "`screen`" application, we can then connect a terminal using:

```
$ screen /dev/ttyUSB0 115200
```

To exit screen, enter "`CTRL+A`" followed by "`:quit`".

An an alternative to `screen`, we also have "`cu`":

```
$ cu -l /dev/ttyUSB0 -s 115200
```

To terminate the program enter "`~.`"

Another program is "`minicom`":

```
$ minicom --baudrate 115200 --device /dev/ttyUSB0
```

And of course, since the serial port is just a stream of characters, there is nothing to prevent you from simply running "`cat`" against it:

```
$ cat /dev/ttyUSB0
```

If we are not sure about the settings of the serial port, we can run:

```
$ stty -F /dev/ttyUSB0 -a
```

which will return the current settings. Following a flash, you may need to change some setting to perform a simple cat:

```
$ stty -F /dev/ttyUSB0 ispeed 115200 ospeed 115200 min 100 ixon
```

See also:

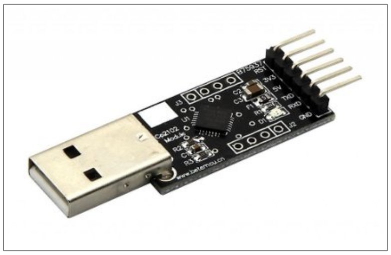
- USB to UART converters
- Loading a program
- Screen user's manual
- PySerial – miniterm
- man(1) – stty

Assembling circuits

Since the ESP32 is an actual electronic component, some physical assembly is required. This book will not attempt to cover non-ESP32 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 ESP32 solutions.

USB to UART converters

You can't program an ESP32 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 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 …

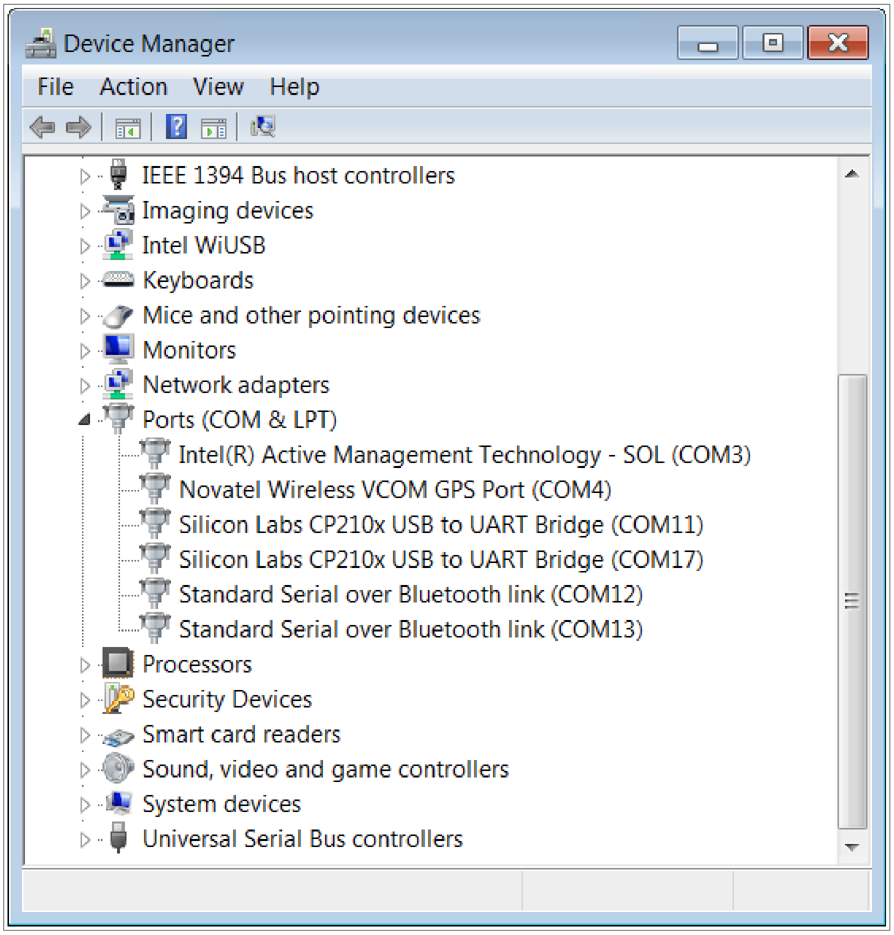


When ordering, don't forget to get some male-female USB extender cables as it is unlikely you will be able to attach your USB devices to both a breadboard and the PC at the same time via direct connection and although connector cables will work, plugging into the breadboard is just so much easier. USB 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).



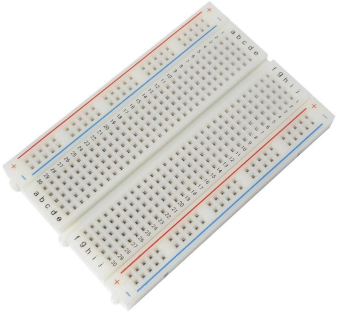
- IEEE 1394 Bus host controllers
- Imaging devices
- Intel WiUSB
- Keyboards
- Mice and other pointing devices
- Monitors
- Network adapters
- Ports (COM & LPT)
  - Intel(R) Active Management Technology - SOL (COM3)
  - Novatel Wireless VCOM GPS Port (COM4)
  - Silicon Labs CP210x USB to UART Bridge (COM11)
  - Silicon Labs CP210x USB to UART Bridge (COM17)
  - Standard Serial over Bluetooth link (COM12)
  - Standard Serial over Bluetooth link (COM13)
- Processors
- Security Devices
- Smart card readers
- Sound, video and game controllers
- System devices
- Universal Serial Bus controllers

See also:

- Connecting to the ESP32
- Working with UART/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. It appears that the plug for wall-wart power is 2.1mm and center positive however I strongly suggest that you read your specific supplier's data sheets very carefully. There is also a potential concern that the barrel socket is wired in parallel with the USB input which could mean that if you attach a high voltage input (eg. 12V) while also having a USB source connected, you may very well damage your USB device. 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 ESP32. By having two power rails, one at 3.3V and the other at 5V, you can power both the ESP32 and devices/circuits that require 5V.

When the ESP32 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 ESP32 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 ESP32. Think of this as the best source of debugging available to you. Not only can these devices display signals over time, the software also allows us to specify the protocol being used and the content of the signal displayed at a higher level with interpretation.

See also:

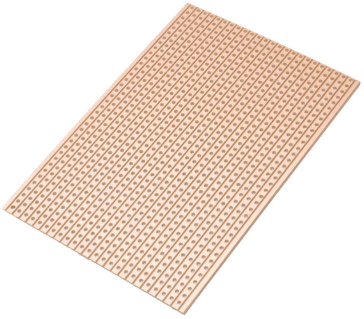
- Saleae logic analyzers
- YouTube: ESP32 and logic analyzers

Sundry components

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

Physical construction

When you have bread-boarded 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 ESP32 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.



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 indication 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 for ESP32

The ESP32 allows you to write applications that can run natively on the device. You can compile applications written in the C programming language and deploy them 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 ESP32 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 ESP32 is the ESP32 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

Espressif IoT Development framework

For the ESP32, a framework has been developed by Espressif called the IoT Development Framework which has become commonly known as "`ESP-IDF`". It can be found on Github here:

https://github.com/espressif/esp-idf

The documentation for this can be found in the esp-idf/docs folder. The documentation is very good and should always be read thoroughly. This book should not be considered the primary source for this information.

Here is a walk through of building an environment on Linux. The full details are found in the doc file called "`linux-setup.rst`". Our environment build will be broken down into a number of distinct parts. The first is the generation of the tool-chain which are the tools necessary to compile the programs:

```
$ cd ~
$ mkdir esp32
$ cd esp32
$ sudo apt-get install git wget make libncurses-dev flex bison gperf python python-
serial
# // >> Here we download and install an x86-64 build chain (compilers)
$ wget https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-61-gab8375a-
5.2.0.tar.gz
$ tar -xvf xtensa-esp32-elf-linux64-1.22.0-61-gab8375a-5.2.0.tar.gz
$ sudo mv xtensa-esp32-elf /opt
$ export PATH=/opt/xtensa-esp32-elf/bin:$PATH
# // << End of build chain install
```

With the tools in place, what remains is the creation of the development environment using the ESP-IDF framework:

```
$ git clone --recursive https://github.com/espressif/esp-idf.git
$ cd esp-idf
$ git submodule update --init
$ export IDF_PATH=~/esp32/esp-idf
```

At this point, we should have all the ingredients necessary for building an application. The recommended way to build an application is to clone the Espressif template app and use that as the basis for your own work.

```
$ cd ~/esp32
$ mkdir apps
$ cd apps
$ git clone https://github.com/espressif/esp-idf-template.git myapp
$ cd myapp
$ make menuconfig
```

At this point an attractive configuration menu is presented that allows us to customize our build environment.

We are now ready to compile our application:

```
$ make
```

The targets that we can make are:

- `make menuconfig` – Run the configuration menu.
- `make deconfig` –
- `make all` – Compile all the code.
- `make flash` – Flash the code to the device.
- `make clean` – Clean the build removing anything that was present previously.
- `make monitor` – Connect to the ESP32 serial port and display messages.
- `make erase_flash` – Erase the flash on the ESP32.
- `make app` –
- `make app-flash` –
- `make app-clean` –
- `make bootloader` –
- `make bootloader-flash` –
- `make bootloader-clean` –
- `make partition-table` –

It is important to note that the ESP-IDF is an evolving platform. It is being actively worked upon by Espressif and the community. What that means is that from time to time you should review how current your ESP-IDF build is and consider replacing it with a newer build. Realize that this *may* result in some rework to your applications, especially if build procedures or APIs change. If you are building production level versions, make a note of the dates of download from Guthub so that if needed, you can checkout those specific versions which may have worked successfully for you in the past.

I recommend using "`doxygen`" to build an HTML document tree of the content of the IDF. If you don't have doxygen installed run:

```
$ sudo apt-get install doxygen
```

Once installed, create a doxygen configuration file by running:

```
$ doxygen -g
```

The result will be a file called "`Doxyfile`".

Some of the doxygen configuration changes you will want to make will include:

- `INPUT=<Root of IDF Install>/components`
- `OUTPUT_DIRECTORY=output`
- `OPTIMIZE_OUTPUT_FOR_C=YES`
- `RECURSIVE=YES`
- `GENERATE_HTML=YES`
- `GENERATE_LATEX=NO`
- `EXTRACT_ALL=YES`
- `EXTRACT_PRIVATE=YES`
- `EXTRACT_PACKAGE=YES`
- `EXTRACT_STATIC= YES`
- `EXTRACT_LOCAL_CLASSES =YES`
- `EXTRACT_LOCAL_METHODS =YES`
- `CLASS_DIAGRAMS=NO`
- `HAVE_DOT=NO`

See also:

- Github: espressif/esp-idf-template
- Github: espressif/esp-idf
- ESP-IDF Getting Started Guide

Application entry point

When your custom ESP32 application boots within an ESP32 device, it has to start somewhere and that somewhere is called the "application entry point". The entry point into your custom ESP32 application is a function called `app_main` with the following signature:

```
int app_main(void)
```

It is your responsibility to implement this function in your C code and provide the logic that is performed.

How ESP-IDF works

When we type "make" in an ESP-IDF template project, the Makefile is executed. It does us no harm in understanding how this works.

The Makefile in our project template sets a variable called "PROJECT_NAME" to the name of our project. It then includes the Makefile found at <IDF_PATH>/make/project.mk and make processing continues.

We can run a make with verbosity switched on to see exactly what is performed:

```
$ make VERBOSE=1
```

The build performs the following major steps:

- build the "conf" tool found in <IDF>/tools/kconfig
- The various components are built including:
  - bt/libbt.a - Bluetooth
  - driver/libdriver.a – Access for GPIO
  - esp32/libesp32.a
  - expat/libexpat.a – A C library for parsing XML
  - freertos/libfreertos.a
  - json/libjson.a – A C library for working with JSON
  - log/liblog.a – Logging capabilities
  - lwip/liblwip.a – Implementation of TCP/IP
  - main/libmain.a
  - mbed/libmbedtls.a – SSL/TLS support
  - nvs_flash/libnvs_flash.a – Non Volatile storage
  - spi_flash/libspi_flash.a – SPI flash driver
  - tcpip_adapter/libtcpip_adapter.a – TCP/IP adapter
- Build the application

A source file compilation includes:

- -std=gnu99
- -Og
- -ggdb
- -ffunction-sections
- -fdata-sections

- -fstrict-volatile-bitfields
- -mlongcalls
- -nostdlib
- -MMD
- -MP
- -Wall
- -Werror
- -Wno-error=unused-function
- -Wno-error=unused-but-set-variable
- -Wno-error=unused-variable
- others

The last step is the one that is likely most useful to us. At a high level it runs:

- `xtensa-esp32-elf-gcc` – The C compiler
- `-nostdlib` – Don't include the standard C library
- `-L<directory>` - Include a bunch of directories looking for libraries
- `-u call_user_start_cpu0` – Specify the entry point into code
- `-Wl,--gc-sections`
- `-Wl,-static`
- `-Wl,--start-group`
- Link with the following libraries:
  - `-lbt`
  - `-lbtdm_app`
  - `-ldriver`
  - `-lesp32`
`-lhal.a`
  - `-lcrypto`
  - `-lcore`
  - `-lnet80211`
  - `-lphy`
  - `-lrtc`

  - `-lpp`
  - `-lwpa`
- Do whatever -T says
  - `-T esp32_out.ld`
  - `-T esp32.common.ld`
  - `-T esp32.rom.ld`
  - `-T esp32.peripherals.ld`
- Link with:
  - `-lexpat`
  - `-lfreertos`
  - `-Wl,--undefined=uxTopUsedPriority`
  - `-ljson`
  - `-llog`
  - `-llwip`
  - `-lmbedtls`
  - `/home/kolban/projects/esp32/esp-idf/components/newlib/lib/libc.a`
  - `/home/kolban/projects/esp32/esp-idf/components/newlib/lib/libm.a`
  - `-lnvs_flash`
  - `-lspi_flash`
  - `-ltcpip_adapter`
  - `-lmain`
  - `-lgcc`
  - `-Wl,--end-group`
  - `-Wl,-EL`
  - `-o /home/kolban/projects/esp32/apps/myapp/build/app-template.elf`
  - `-Wl,-Map=/home/kolban/projects/esp32/apps/myapp/build/app-template.map`

Error handling

Most of the ESP32 functions we call can return an error indication in the event that something goes wrong. The result is an `esp_err_t` which can be treated as an integer. If the function call succeeded, then the return value is `ESP_OK`. Any other value is an error indication. A macro is available called `ESP_ERROR_CHECK()` which takes a statement as a parameter. The statement is expected to return an `esp_err_t`. If the return is other than `ESP_OK`, then an assertion is raised, the ESP32 halts and the statement is written to the console. We must include "`esp_err.h`" to use this capability.

The other codes defined that are values that can be returned from ESP-IDF API calls include:

Symbol	Value
ESP_OK	0
ESP_FAIL	-1
ESP_ERR_NO_MEM	257 (0x101)
ESP_ERR_INVALID_ARG	258 (0x102)
ESP_ERR_INVALID_STATE	259 (0x103)
ESP_ERR_INVALID_SIZE	260 (0x104)
ESP_ERR_NOT_FOUND	261 (0x105)
ESP_ERR_NOT_SUPPORTED	262 (0x106)
ESP_ERR_TIMEOUT	263 (0x107)

If you need to convert an error code into a string representation a sample code fragment can be found here: https://github.com/nkolban/esp32-snippets/blob/master/error%20handling/fragments/espToError.c

See also:

- ESP-IDF logging

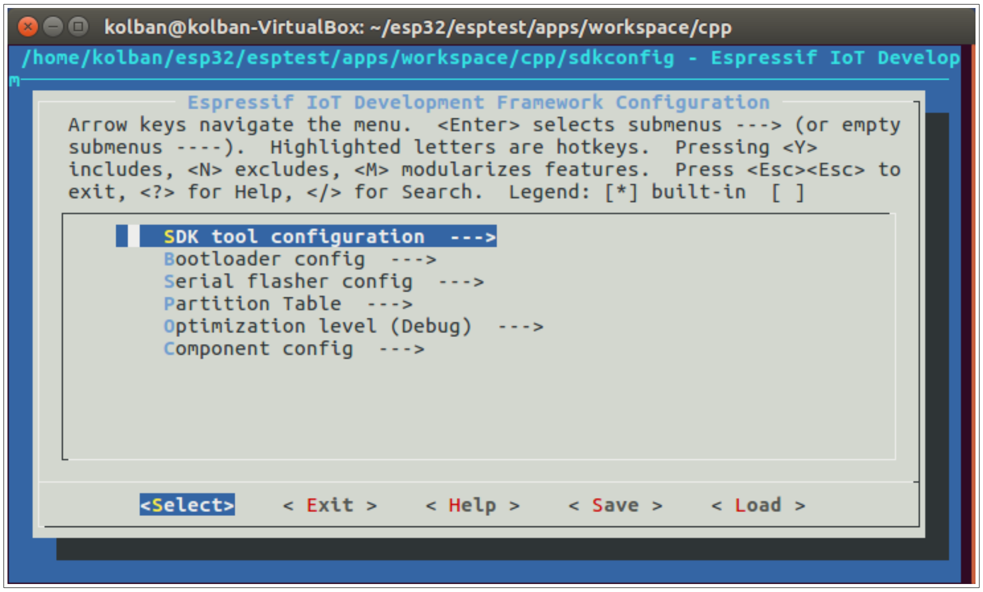
The build environment menu configuration

Within a project, we can create a configuration file that controls how the build of a project progresses. This configuration file (`sdkconfig`) has a very attractive menu configuration tool that can be opened by running:

```
$ make menuconfig
```

Note: I am especially impressed with this component of the ESP-IDF. An attention to detail and quality that makes configuration a lot nicer to use and less error prone than hand editing files.

After running the command we see a text based menu editor:



```
kolban@kolban-VirtualBox: ~/esp32/esptest/apps/workspace/cpp
/home/kolban/esp32/esptest/apps/workspace/cpp/sdkconfig - Espressif IoT Develop

  Espressif IoT Development Framework Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

        SDK tool configuration   --->
        Bootloader config  --->
        Serial flasher config  --->
        Partition Table  --->
        Optimization level (Debug)  --->
        Component config  --->

     <Select>    < Exit >    < Help >    < Save >    < Load >
```

We can tab between major components and make changes. There is even selection context help assistance.

Under SDK tool configuration

- Compiler toolchain path/prefix – xtensa-esp32-elf-
- Python 2 interpreter – python

Under Bootloader config

- Bootloader log verbosity – Warning
- Build bootloader with Link Time Optimisation

Under Security features

- Enable secure boot in bootloader
- Enable flash encryption on boot

Under Serial flasher config

- Default serial port – /dev/ttyUSB0
- Default baud rate – 115200 baud
- Use compressed upload – false
- Flash SPI mode – DIO
- Flash SPI speed – 40MHz
- Flash size – 2MB
- Detect flash size when flashing bootloader

- Before flashing
- After flashing
- 'make monitor' baud rate

Under Partition Table

- Partition Table – Single factory app, no OTA

Under Compiler options

- Optimisation Level – Debug (-Og)
- Assertion level – Enabled

Under Component config

- Application Level Tracing
  - Data Destination
  - FreeRTOS SystemView Tracing
- Amazon Web Services IoT Platform
- Bluetooth
- ESP32-specific config
  - CPU frequency – 240MHz
  - Reserve memory for two cores – true
  - Use TRAX tracing feature – false
  - Core dump destination
  - Number of universally administered MAC address
  - System event queue size – 32
  - Event loop task stack size – 2048
  - Main task stack size – 4096
  - Inter-Processor Call (IPC) task stack size
  - Standard-out outputs adds carriage return before newline – true
  - Enable 'nano' formatting options for printf/scanf family
  - UART for console output
  - UART console baud rate

    - Enable Ultra Low Power (ULP) Coprocessor
    - Panic handler behavior – Print registers and reboot
    - Make exception and panic handlers JTAG/OCD aware
    - Interrupt watchdog
        - Interrupt watchdog timeout (ms)
    - Task watchdog
        - Invoke panic handler when Task Watchdog is triggered
        - Task watchdog timeout (seconds)
        - Task watchdog watches CPU0 idle task
    - Hardware brownout detect & reset
    - Timers used for gettimeofday function
    - RTC clock source
    - Number of cycles for RTC_SLOW_CLK calibration
    - Extra delay in deep sleep wake stub (in us)
    - Main XTAL frequency
- Wifi
    - Software controls WiFi/Bluetooth coexistence
    - Max number of WiFi RX buffers
    - Max number of WiFi dynamic RX buffers
    - Type of WiFi TX buffers
    - Max number of WiFi dynamic TX buffers
    - WiFi AMPDU
        - WiFi AMPDU TX BA window size
        - WiFi AMPDU RX BA window size
    - WiFi NVS flash
- PHY
    - Do phy calibration and store calibration data in NVS
    - Use a partition to store PHY init data
    - Max WiFi TX power

- Enable Ethernet
- FAT Filesystem support
    - OEM Code Page
    - Max long filename length
- FreeRTOS
    - Run FreeRTOS only on first core – true
    - Xtensa timer to use as the FreeRTOS tick source – Timer 0 (int (1000) tick rate (Hz))
    - Tick Rate (HZ)
    - Halt when an SMP-untested function is called
    - Check for stack overflow – Check by stack pointer value
    - Set a debug watchpoint as a stack overflow check
    - Number of thread local storage pointers – 3
    - FreeRTOS assertions – abort() on failed assertions
    - Stop program on scheduler start when JTAG/OCD is detected – true
    - Enable heap memory debug – false
    - Idle Task stack size
    - ISR stack size
    - Use FreeRTOS legacy hooks
    - Maximum task name length
    - Enable FreeRTOS static allocation API
    - FreeRTOS timer task priority
    - FreeRTOS timer task stack size
    - FreeRTOS timer queue length
    - Debug FreeRTOS internals – ?
- Log output
    - Default log verbosity – Warning
    - Use ANSI terminal colors in log output – true
- LWIP

    - Enable copy between Layer2 and Layer3 packets
    - Max number of open sockets – 4
    - Index for thread-local-storage pointer for lwip – 0
    - Enable SO_REUSEADDR option – false
    - Enable SO_RCVBUF option
    - Maximum number of NTP servers – 1
    - Enable fragment outgoing IP packets
    - Enable reassembly incoming fragmented IP packets
    - Enable an ARP check on the offered address
    - TCP/IP Task Stack Size
    - Enable PPP support
    - ICMP
        - Respond to multicast pings
        - Respond to broadcast pins
- mbedTLS
    - TLS maximum message content length – 16384
    - Enable mbedTLS debugging – false
    - Enable hardware AES acceleration
    - Enable hardware MPI (bignum) acceleration
        - Use interrupt for MPI operations
    - Enable hardware SHA acceleration
    - Enable mbedtls time
        - Enable mbedtls time data
- OpenSSL
    - Enable OpenSSL debugging
    - Select OpenSSL assert function
- SPI Flash driver
    - Enable operation counters – false
    - Enable SPI flash ROM driver patched functions

After changing a configuration parameter, one should perform a complete clean rebuild of the ESP-IDF environment.

Adding a custom ESP-IDF component

As you build projects, you may want to build your own custom components. To create a component, create a directory in the root of your application called "`components`" and within there, create a directory named after your new component. In that directory, we will construct our component. It should contain:

- component.mk
- <your source files>.c
- Kconfig (optional)

Should you wish to place source files in sub-directories beneath your component directory, you can do so and name the additional directories in "`component.mk`" using the `COMPONENT_SRCDIRS` variable. For example:

```
COMPONENT_SRCDIRS:=dir1 dir2
```

By default, this will compile all the C source files in the component directory and link them into a library for linking with your application. If you wish to name the source files to be compiled explicitly you can do so by naming the resulting object files using the `COMPONENT_OBJS` variable. For example:

```
COMPONENT_OBJS := file1.o file2.o dir1/file3.o
```

The `Kconfig` file contains details that are shown in the menu system shown by "`make menuconfig`". The format of this file is a configuration language which is called "Kconfig". The idea is that we wish to have a custom set of configuration options and we wish to be able to set/change the value of these options. The Kconfig file defines the existence of a set of options, their data types, their default values, the allowable values for an option and help text associated with an option. The "`make menuconfig`" then parses the set of distinct Kconfig files that it can find and creates a text/full-screen menu showing each of the options and navigation to set and change them.

Rather than show you all the possible options that can be set as one long huge list, we can create menus and sub-menus of options to allow the user of the configuration to drill down and locate the specific options that they might want to change.

It is described in a web file but at a high level, it has the following format:

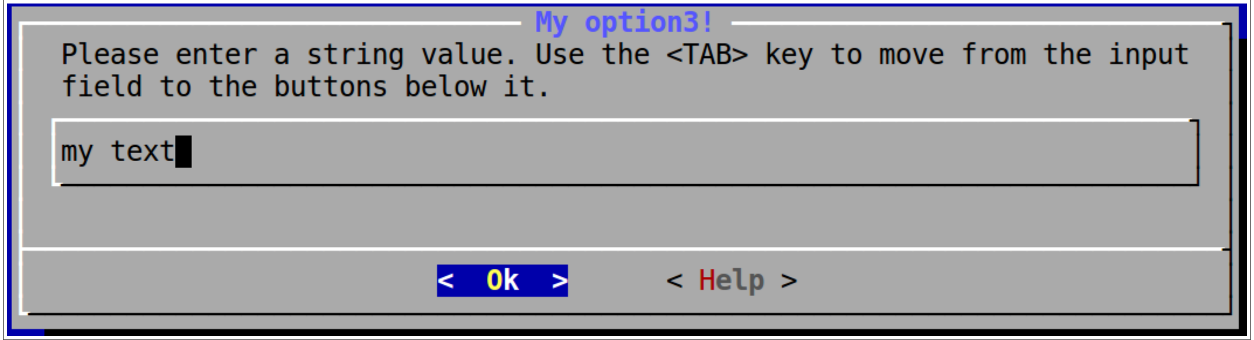
```
menu "<Menu heading text>"
config <ITEM_NAME>
   <type>
   prompt "<text>"
   default <value>
   help
      help text
endmenu
```

A specific configuration item is given a name. For example we can define:

```
config OPTION1
```

This will create a new configuration option called `CONFIG_OPTION1`. Note that the "`CONFIG_`" prefix is added to each option automatically. Each option can have a type associated with it. The choices of types are:

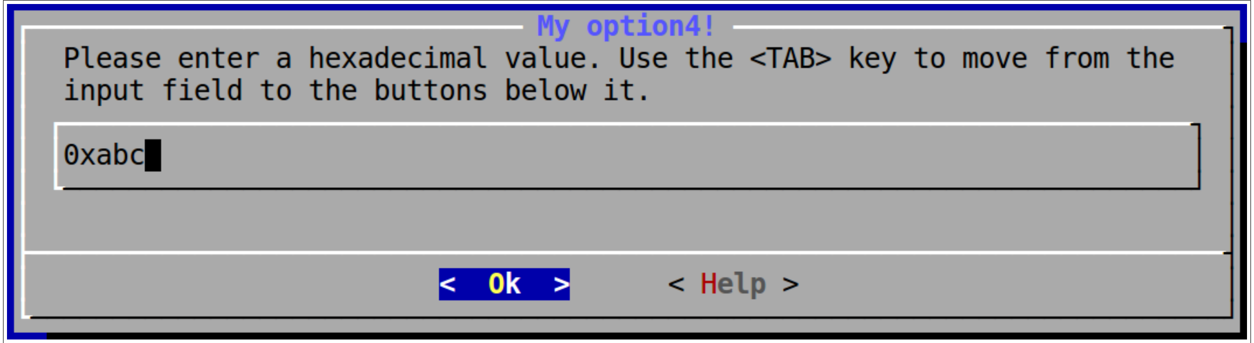
- `bool` – A true/false value specified by "y" or "n".
- `tristate` – A boolean option that is either "y", "n" or not set.
- `string` – A text string.



Please enter a string value. Use the <TAB> key to move from the input field to the buttons below it.

my text

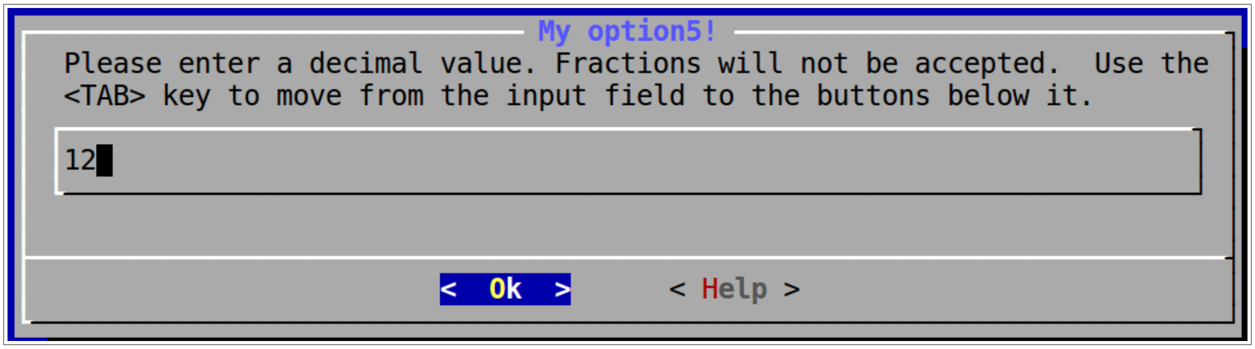
- `hex` – A hex number.



Please enter a hexadecimal value. Use the <TAB> key to move from the input field to the buttons below it.

0xabc

- `int` – An integer.



```
Please enter a decimal value. Fractions will not be accepted.  Use the
<TAB> key to move from the input field to the buttons below it.
```

```
12
```

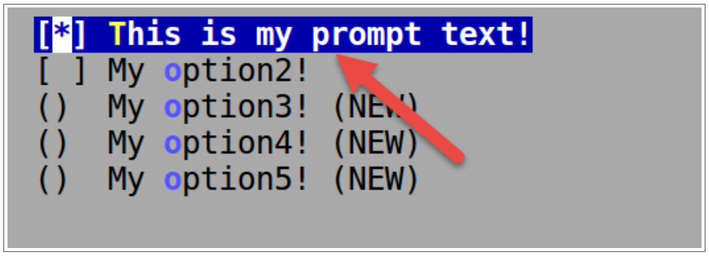
Following the definition of a configuration option, we can have prompt text. For example:

```
config OPTION1
   bool
   prompt "This is my prompt text!"
```

A default value can be supplied using the "default" option:

```
config OPTION1
   bool
   prompt "This is my prompt text!"
   default y
```

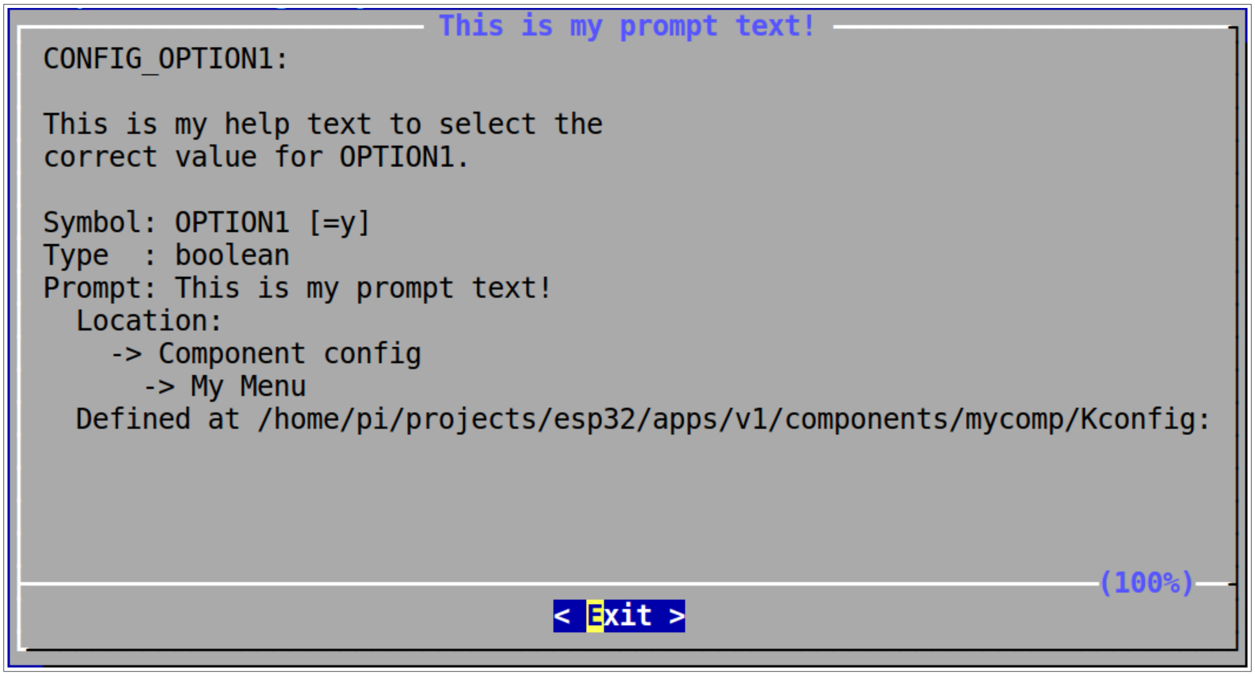
The prompt text is shown beside the current value of the option in the menu:



```
[*] This is my prompt text!
[ ] My option2!
()  My option3! (NEW)
()  My option4! (NEW)
()  My option5! (NEW)
```

Help text can be provided which is shown as assistance to the user.

```
config OPTION1
   bool
   prompt "This is my prompt text!"
   default y
   help
      This is my help text to select the
      correct value for OPTION1.
```



```
CONFIG_OPTION1:

This is my help text to select the
correct value for OPTION1.

Symbol: OPTION1 [=y]
Type  : boolean
Prompt: This is my prompt text!
  Location:
    -> Component config
      -> My Menu
  Defined at /home/pi/projects/esp32/apps/v1/components/mycomp/Kconfig:
```

Detailed documentation on the ESP-IDF build system and how to configure it can be found in the "`build_system.rst`" document in Github.

If you have defined your own constants, your code will need to include "sdkconfig.h" in order to be able to test the value of the menu defined entries. This file is generated by the ESP-IDF at compilation time from the "sdkconfig" file generated by running the menu system.

See also:

- Kconfig language
- The component.mk settings

Working with memory

Heap based storage can allocated with `malloc()` and released with a corresponding call to `free()`. There are quite a number of ESP-IDF function that take pointers to storage as input. Unless these explicitly state that the data must be maintained then we can assume a deep copy for asynchronous function calls. With that in mind, the following is a legal pattern:

```
uint8_t* pMyData = malloc(size);
// populate data
esp_idf_function(pMyData);
free(pMyData);
```

Putting this into words, unless explicitly stated to the contrary, you can pass in a pointer to storage to an ESP-IDF function and assume that you are at liberty to release or otherwise re-use that storage on return from the ESP-IDF call. If the call needs the content that was in the storage at the time the function was called, it will have taken a copy of that data.

Compiling

Application code for an ESP32 program is commonly written in C. Before we can deploy an application, we must compile the code into binary machine code instructions. Before that though, let us spend a few minutes thinking about the code.

We write code using an editor and ideally an editor that understands the programming language in which we are working. These editors provide syntax assistance, keyword coloring and even contextual suggestions. After we save our entered code, we compile it and then deploy it and then test it. This cycle is repeated so often that we often use a product that encompasses editing, compilation, execution and testing as an integrated whole. The generic name for such a product is an "Integrated Development Environment" or "IDE". There are instances of these both fee and free. In the free camp, my weapons of choice are Eclipse and Arduino IDE.

The Eclipse IDE is an extremely rich and powerful environment. Originally written by IBM, it was open sourced many years ago. It is implemented in Java which means that it runs and behaves identically across all the common platforms (Windows, Linux, OSx). The nature of Eclipse is that it is architected as a series of extensible plug-ins. Because of this, many contributors across many disciplines have extended the environment and it is now a cohesive framework for just about everything. Included in this mix is a set of plug-ins which, on aggregate, are called the "C Developers Tools" or "CDT". If one takes a bare bones Eclipse and adds the CDT, one now has a first rate C IDE. However, what the CDT does not supply (and for good reason) are the actual C compilers and associated tools themselves. Instead, one "defines" the tools that one wishes to use within the CDT and the CDT takes it from there.

For our ESP32 story, this means that if we can find (which we can) a set of C compiler tools that take C source and generate Xtensa binary, we can use CDT to build our programs.

To make things more interesting though, we need to realize that C is not the only language we can use for building ESP32 applications. We can also use C++ and assembly. You may be surprised that I mention assembly as that is as low level as we can possibly get however there are odd times when we need just that (thankfully rarely) … especially when we realize that we are pretty much programming directly to the metal. The Arduino libraries (for example) have at least one assembly language file.

For physical file types, the suffixes used for different file we will come across during development include:

- `.h` – C and C++ language header file
- `.c` – C language source file
- `.cpp` – C++ source file
- `.S` – Assembler source file
- `.o` – Object file (compiled source)
- `.a` – Archive library

To perform the compilations, 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.

Compilation

Let us imagine we have a "`main.c`" source file. How then can we properly compile that to get it ready for linking into an ESP32 binary? The Espressif documented mechanism is to use the ESP-IDF and that works great … but there are times when we can't start with the Espressif make system but instead need to work in the opposite direction … namely starting from an existing make system and integrating the correct steps into that already existent environment. What we have done is spent time studying how the ESP-IDF works and reverse engineered enough of it to build this recipe. First, if we run the ESP-IDF make system with the environment variable `VERBOSE` set to `1`, we get a lot of detail. Here is what I currently see from a current build:

```
xtensa-esp32-elf-gcc -DESP_PLATFORM -Og -g3 -Wpointer-arith -Werror -Wno-error=unused-
function -Wno-error=unused-but-set-variable -Wno-error=unused-variable -Wall
-ffunction-sections -fdata-sections -mlongcalls -nostdlib -MMD -MP -std=gnu99 -g3
-fstrict-volatile-bitfields -DMBEDTLS_CONFIG_FILE='"mbedtls/esp_config.h"'
-DHAVE_CONFIG_H
-I /home/pi/projects/esp32/apps/parallel1/main/include
-I /home/pi/projects/esp32/esp-idf/components/bt/include
-I /home/pi/projects/esp32/esp-idf/components/driver/include
-I /home/pi/projects/esp32/esp-idf/components/esp32/include
-I /home/pi/projects/esp32/esp-idf/components/expat/port/include
-I /home/pi/projects/esp32/esp-idf/components/expat/include/expat
-I /home/pi/projects/esp32/esp-idf/components/freertos/include
-I /home/pi/projects/esp32/esp-idf/components/json/include
-I /home/pi/projects/esp32/esp-idf/components/json/port/include
```

```
-I /home/pi/projects/esp32/esp-idf/components/log/include
-I /home/pi/projects/esp32/esp-idf/components/lwip/include/lwip
-I /home/pi/projects/esp32/esp-idf/components/lwip/include/lwip/port
-I /home/pi/projects/esp32/esp-idf/components/lwip/include/lwip/posix
-I /home/pi/projects/esp32/esp-idf/components/mbedtls/port/include
-I /home/pi/projects/esp32/esp-idf/components/mbedtls/include
-I /home/pi/projects/esp32/esp-idf/components/newlib/include
-I /home/pi/projects/esp32/esp-idf/components/nghttp/port/include
-I /home/pi/projects/esp32/esp-idf/components/nghttp/include
-I /home/pi/projects/esp32/esp-idf/components/nvs_flash/include
-I /home/pi/projects/esp32/esp-idf/components/spi_flash/include
-I /home/pi/projects/esp32/esp-idf/components/tcpip_adapter/include
-I /home/pi/projects/esp32/apps/parallel1/build/include/
-I. -c /home/pi/projects/esp32/apps/parallel1/main/./main.c -o main.o
```

Let us now pull it apart piece by piece and see what it going on:

- `xtensa-esp-elf-gcc` – This is the C compiler that generates compiled code for the ESP32 CPUs.
- `-DESP_PLATFORM` – This sets the existence of a macro definition called "`ESP_PLATFORM`". The belief is that this can be used by the C pre-processor to include, exclude or otherwise manipulate the source before compilation. For example, there may be source files that have common code within them where some code is valid for one environment/platform and some other code, also contained in the same source file, is valid for a different environment/platform. The existence of ESP_PLATFORM could be used as a distinguisher within the code.
- `-Og` – Optimize for debugging experience.
- `-g3` – Level 3 debugging information included.
- `-Wpointer-arith` – Produce warnings relating to pointer arithmetic. Specifically, anything that relies on the size of a function type or the size of void.
- `-Werror` – Make all warnings into errors. This stops/fails a compilation on any warnings produced during compile.
- `-Wno-error=unused-function` – Don't flag an un-used function as an error.
- `-Wno-error=unused-but-set-variable` – Don't flag an unused variable as an error that had a value set upon it.
- `-Wno-error=unused-variable` – Don't flag an unused variable as an error.
- `-Wall` – Enable a large set of warnings.
- `-ffunction-sections` – Place each function in its own section.

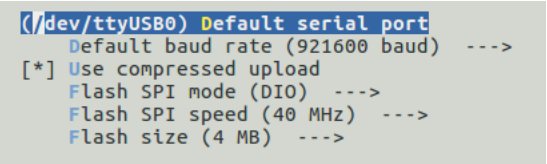
- `-fdata-sections` – Place each piece of data in its own section.
- `-mlongcalls` – An Xtensa specific option
- `-nostdlib` – Do not use the standard startup files or libraries when linking.
- -MMD
- `-MP` – Something about makefiles.
- `-std=gnu99` – Set the compiler standard.
- `-fstrict-volatile-bitfields` – How volatile bit fields should be accessed.
- -DMBEDTLS_CONFIG_FILE='"mbedtls/esp_config.h"'
- `-DHAVE_CONFIG_H` – Define a macro flag.
- `-I <various>` – Specify directories which should be searched for includes. The majority of additional directories are included in the ESP-IDF directory structures.
- `-c` – Compile to object file with no linking.
- `-o <filename>` – write the output into the given file.

If we need to include additional code that is only present on an ESP32, we can use the existence of the macro definition "`ESP_PLATFORM`" as the indicator. For example:

```
#ifdef ESP_PLATFORM
// ESP32 specific code
#endif
```

Flashing

Flashing is the mechanism used to move a binary from the file system into the ESP32. If we are using the ESP-IDF, then the process is very straightforward. We can use the command "make flash" and the tools needed to flash the ESP32 will be run for us with the correct inputs. The primary parameters for a flash are the port used to communicate with the ESP32 and the baud rate (transmission rate) to be used. These and other settings can be configured in the "`make menuconfig`" in the sub menu called "`Serial flasher config`":



```
(/dev/ttyUSB0) Default serial port
    Default baud rate (921600 baud)  --->
[*] Use compressed upload
    Flash SPI mode (DIO)  --->
    Flash SPI speed (40 MHz)  --->
    Flash size (4 MB)  --->
```

We can over-ride these at the command line by using:

```
$ make flash ESPPORT=<serial port> ESPBAUD=<baud rate>
```

The environment variable `ESPPORT` specifies the serial port and `ESPBAUD` specifies the baud rate. If not supplied, the values defined in `make menuconfig` will be used.

By examination of the recipes supplied by Espressif, we find that the underlying command is as follows

```
python /home/pi/projects/esp32/esp-idf/components/esptool_py/esptool/esptool.py --chip
esp32 --port "/dev/ttyUSB0" --baud 230400 write_flash -z --flash_mode "dio"
--flash_freq "40m"
0x1000 /home/pi/projects/esp32/apps/parallel1/build/bootloader/bootloader.bin
0x10000 /home/pi/projects/esp32/apps/parallel1/build/app-template.bin
0x4000 /home/pi/projects/esp32/apps/parallel1/build/partitions_singleapp.bin
```

Breaking this down we arrive at:

- `python` – run a python script.
- `esptool.py` – Run the program called "`esptool.py`".
- `--chip esp32` – Declare that we are flashing an ESP32.
- `--port "/dev/ttyUSB0"` – Define the serial port to which the ESP32 is connected.
- `--baud 115200` – Define the transmission baud rate across the serial port.
- `write_flash` – Perform the write flash command.
- `-z` – Unknown.
- `--flash_mode "dio"` – Use "dio" flash mode.
- `--flash_freq "40m"` – Use a flash frequency of "40m"
- `0x1000 bootloader.bin` – Load the bootloader at 0x1000.
- `0x10000 appname.bin` – Load the application at 0x10000.
- `0x4000 partitions_singleapp.bin` – Load the partition table at 0x4000.

Following the construction of the ELF file, we now run a tool which converts the ELF into an uploadable image. Again, looking at the ESP-IDF tools, we find the following command being run:

```
python /home/pi/projects/esp32/esp-idf/components/esptool_py/esptool/esptool.py --chip
esp32 elf2image --flash_mode "dio" --flash_freq "40m" -o
/home/pi/projects/esp32/apps/v1/build/app-template.bin
/home/pi/projects/esp32/apps/v1/build/app-template.elf
```

- python
- esptool.py
- --chip esp32
- elf2image
- --flash_mode "dio"
- --flash_freq "40m"
- -o <output file>
- <input elf file>

Loading a program

Once the program has been compiled, it needs to be loaded into the ESP32. This task is called "flashing". In order to flash the ESP32, 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 ESP32 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 start-up, 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.

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 ESP32s, we commonly connect them to USB->UART converters. These devices are able to supply UART used to program the ESP32. 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 ESP32 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 stored upon 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
- Error: Reference source not found
- What is a UART?
- esptool.py

Programming environments

We can program the ESP32 using the Espressif supplied SDK on Windows using Eclipse. A separate chapter on setting up that environment is supplied. We also have the ability to program the ESP32 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 ESP32 applications. Many of these tools are supplied as part of the Xtensa tool chain. Let us remind ourselves that an executable that runs on the ESP32 executes machine code in Xtensa instruction set. This means that we need a compiler that generates Xtensa code as opposed to a compiler that generates Intel x86 code.

When a compiler runs on one architecture and generates codes for a different architecture, that is called a cross compiler. For example, if we have a C source application and compile it with the Xtensa compiler on an Intel platform, we generate Xtensa executable code that is not executable on the Intel platform on which it was compiled. My convention is to install the Xtensa tool set at:

```
/opt/xtensa-esp32-elf
```

with the binaries themselves being in

```
/opt/xtensa-esp32-elf/bin
```

If we list the tools contained within we will find that they are all prefixed with "`xtensa-esp32-elf-`*". The reason for the long name is that we want to be very clear that we are using a cross compiler tool as opposed to a tool with the exact same name (without the prefix) that may already be present on the platform where we are performing the compilation. For example, on a Linux platform, it is not uncommon to compile a C program using the compiler called "`gcc`". If one typed "`gcc main.c`", it is likely it will compile and produce an object file. However, the code produced would be the native code for the compilation platform … likely to be Intel x86. The same program to compile for ESP32 would then be "`xtensa-esp32-elf-gcc main.c`".

xntensa-esp32-elf-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
- xtensa-esp32-elf-nm

esptool.py

This tool is an open source implementation used to flash the ESP32 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 run-time installation. The location of the tool is:

```
$IDF_PATH/components/esptool_py/esptool/esptool.py
```

The flags are:

- `-p port` | `--port port` – The serial port to use

- `-b baud` | `--baud baud` – The baud rate to use for serial. Common baud rates include:
  - `115200` – basic speed
  - `921600` – high speed
- `-h` – Help
- `{command} -h` – Help for that command
- `load_ram {filename}` – Upload 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
  - `-z` or `--compress` – compress during transmission
  - `{address} {fileName}` – Address to write, file to write … repeatable
- `read_flash` – Read a blob of flash
  - {address} {size} {fileName}
- `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)
```

Here is an example of image_info against the bootloader.bin

```
esptool.py v2.0-dev
Image version: 1
Entry point: 40098200
4 segments

Segment 1: len 0x00000 load 0x3ffc0000 file_offs 0x00000018
Segment 2: len 0x00a08 load 0x3ffc0000 file_offs 0x00000020
Segment 3: len 0x01068 load 0x40078000 file_offs 0x00000a30
Segment 4: len 0x00378 load 0x40098000 file_offs 0x00001aa0
Checksum: 92 (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. This is an especially useful command if one ends up someone bricking the device as it should reset the device to its defaults.

See also:

- Loading a program
- Github: themadinventor/esptool

xtensa-esp32-elf-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 ESP32 we find the following:

Compiling

- `-c` – Compile the code to a `.o` object file.
- `-Os` – Optimize code generation for size.
- `-O2` – Optimize for performance which code result in larger code size. For example, instead of making a function call, code could be in-lined.
- `-ggdb` – Generate debug code that can be used by the `gdb` debugger..
- `-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

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`.

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.

xtensa-esp32-elf-nm

List symbols from object files.

Useful flags:

- `--defined-only` – Show only defined exports
- `--undefined-only` – Show only undefined exports
- --line-numbers

See also:

- xntensa-esp32-elf-ar
- GNU – nm

xtensa-esp32-elf-objcopy

See also:

- GNU – objcopy

xtensa-esp32-elf-objdump

Some of the more important flags are:

- `--syms` – Dump the symbols in the archive.
- `--headers` – Dump the section headers.

See also:

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

xxd

This is a deceptively simple but useful tool. What it does is dump binary data contained within 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 content of the file. This means that you can take binary data and include it in your applications.

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

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

Linking

We have seen how to compile a C source file into its object file (`.o`) representation. Now we turn our attention on how to link these object files together with the ESP32 libraries in order to produce a binary file that can be flashed into the ESP32 for execution.

If we examine the output of compiling a project using the ESP-IDF with the `VERBOSE=1` flag set, we see the underlying command used to perform the linking. Here is an example and then we'll start to pull it apart:

```
xtensa-esp32-elf-gcc -nostdlib -L/home/pi/projects/esp32/esp-idf/lib
-L/home/pi/projects/esp32/esp-idf/ld
-L/home/pi/projects/esp32/apps/parallel1/build/bootloader
-L/home/pi/projects/esp32/apps/parallel1/build/bt
-L/home/pi/projects/esp32/apps/parallel1/build/driver
-L/home/pi/projects/esp32/apps/parallel1/build/esp32
-L/home/pi/projects/esp32/apps/parallel1/build/esptool_py
-L/home/pi/projects/esp32/apps/parallel1/build/expat
-L/home/pi/projects/esp32/apps/parallel1/build/freertos
-L/home/pi/projects/esp32/apps/parallel1/build/json
-L/home/pi/projects/esp32/apps/parallel1/build/log
-L/home/pi/projects/esp32/apps/parallel1/build/lwip
-L/home/pi/projects/esp32/apps/parallel1/build/mbedtls
-L/home/pi/projects/esp32/apps/parallel1/build/newlib
-L/home/pi/projects/esp32/apps/parallel1/build/nghttp
-L/home/pi/projects/esp32/apps/parallel1/build/nvs_flash
-L/home/pi/projects/esp32/apps/parallel1/build/partition_table
-L/home/pi/projects/esp32/apps/parallel1/build/spi_flash
-L/home/pi/projects/esp32/apps/parallel1/build/tcpip_adapter
-L/home/pi/projects/esp32/apps/parallel1/build/main -u call_user_start_cpu0 -Wl,--gc-
sections -Wl,-static -Wl,--start-group  -lbt -L/home/pi/projects/esp32/esp-
idf/components/bt/lib -lbtdm_app    -ldriver   -lesp32 /home/pi/projects/esp32/esp-
idf/components/esp32/libhal.a -L/home/pi/projects/esp32/esp-idf/components/esp32/lib
-lcrypto -lcore -lnet80211 -lphy -lrtc -lpp -lwpa -L /home/pi/projects/esp32/esp-
idf/components/esp32/ld -T esp32_out.ld -T esp32.common.ld -T esp32.rom.ld -T
esp32.peripherals.ld   -lexpat   -lfreertos -Wl,--undefined=uxTopUsedPriority   -ljson
```

```
-llog   -llwip   -lmbedtls   /home/pi/projects/esp32/esp-
idf/components/newlib/lib/libc.a /home/pi/projects/esp32/esp-
idf/components/newlib/lib/libm.a   -lnghttp   -lnvs_flash   -lspi_flash
-ltcpip_adapter   -lmain  -lgcc -Wl,--end-group -Wl,-EL -o
/home/pi/projects/esp32/apps/parallel1/build/app-template.elf -Wl,-
Map=/home/pi/projects/esp32/apps/parallel1/build/app-template.map
```

- `xtensa-esp32-elf-gcc` – The compiler for the Xtensa architecture which also knows how to link.
- `-nostdlib` – Do not link with the standard startup files and libraries.
- `-L <various>` – Specify the directories which should be searched for library files.
- `-u call_user_start_cpu0` – Pretend that the symbol called. "`call_user_start_cpu0`" is undefined to force the linker to resolve it.
- `-Wl,--gc-sections` – Garbage collect unused input sections.
- `-Wl,-static` – Do not link against shared libraries.
- `-Wl,--start-group` – Start a group of archives. Used to resolve circular references.
- `-Wl,-EL` – Link in little-endian format.
- `<various libraries>` – Link with the named libraries.
- `-Wl,--end-group` – End a group of archives.
- `-T esp32_out.ld` – Use a Linker script. Found in `build/esp32.ld`.
- `-T esp32.common.ld` – Use a Linker script. Found in `ESP_IDF/components /esp32/ld/esp32.common.ld`.
- `-T esp32.rom.ld` – Use a Linker script. Found in `ESP_IDF/components /esp32/ld/esp32.rom.ld`.
- `-T esp32.peripherals.ld` – Use a Linker script. Found in `ESP_IDF/components/esp32/ld/esp32.peripherals.ld`.
- `-o <filename>` – Write the result to the named file.
- `-Wl,-Map=<filename>` – Write a link map to the named file.

Linked libraries:

- `-lbt` – build/bt/libbt.a
- `-lbtdm_app` – ESP_IDF/components/bt/lib/libbtdm_app.a
- `-ldriver` – build/driver/libdriver.a

- `-lesp32` – build/esp32/libesp32.a
- `libhal.a` – ESP_IDF/components/esp32/libhal.a
- `-lcrypto` – ESP_IDF/components/esp32/lib/libcrypto.a
- `-lcore` – ESP_IDF/components/esp32/lib/libcore.a
- `-lnet80211` – ESP_IDF/components/esp32/lib/libnet80211.a
- `-lphy` – ESP_IDF/components/esp32/lib/libphy.a
- `-lrtc` – ESP_IDF/components/esp32/lib/librtc.a
- `-lpp` – ESP_IDF/components/esp32/lib/libpp.a
- `-lwpa` – ESP_IDF/components/esp32/lib/libwpa.a
- `-lexpat` – build/expat/libexpat.a
- `-lfreertos` – build/freertos/libfreertos.a
- `-ljson` – build/json/libjson.a
- `-llog` – build/log/liblog.a
- `-llwip` – build/lwip/liblwip.a
- `-lmbedtls` – build/mbedtls/libmbedtls.a
- `libc.a` – ESP_IDF/components/newlib/lib/libc.a
- `libm.a` – ESP_IDF/components/newlib/lib/libm.a
- `-lnghttp` – build/nghttp/libnghttp.a
- `-lnvs_flash` – build/nvs_flash/libnvs_flash.a
- `-lspi_flash` – build/spi_flash/libspi_flash.a
- `-ltcpip_adapter` – build//tcpip_adapter/libtcpip_adapter.a
- `-lmain` – Project files
- -lgcc

Within a linked executable there are a variety of "sections". We can dump those with

```
$ xtensa-esp32-elf --headers <file.elf>
```

Here is an example (note some of the more uninteresting entries were removed

```
app-template.elf:     file format elf32-xtensa-le

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
```

```
 0 .iram0.vectors 00000400  40080000  40080000  00006b40  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .iram0.text   000167d2  40080400  40080400  00006f40  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .dram0.bss    0000f310  3ffb0000  3ffb0000  00005878  2**3
                  ALLOC
  3 .dram0.data   000012c0  3ffbf310  3ffbf310  00005880  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  4 .flash.rodata 00005794  3f400010  3f400010  000000e0  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  5 .flash.text   00028ded  400d0018  400d0018  0001d714  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .comment      00000062  00000000  00000000  0020f49b  2**0
                  CONTENTS, READONLY
 17 .xtensa.info  00000038  00000000  00000000  0020f4fd  2**0
                  CONTENTS, READONLY
```

From the above, here are some of the interesting parts:

Section	Load address	size
`.iram0.vectors`	`0x4008 0000`	`1024   (0x400)`
`.iram0.text`	`0x4008 0400`	`91114  (0x167d2)`
`.dram0.bss`	`0x3ffb 0000`	`62224  (0xf310)`
`.dram0.data`	`0x3ffb f310`	`4800   (0x12c0)`
`.flash.rodata`	`0x3f40 0010`	`22420  (0x5794)`
`.flash.text`	`0x400d 0018`	`167405 (0x28ded)`

After we have linked our libraries together to form an application that is ready for flashing to the ESP32, we can ask the question: "What does the binary actually contain?". There are a set of make tools available that will produce diagnostic logs of the executable that will tell us this information. These commands are:

- `make size` – Show application wide information.
- `make size-components` – Show component information of components used by the application.
- `make size-files` – Show files (object files) information from each component used by the application.

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.

ESP-IDF logging

The ESP-IDF framework provides a logging set of features. Use these as opposed to simple "`printf()`" statements. These logging statements can then be inserted in your own application for diagnosing problems or capturing traces.

To use the logging functions, we must include "`esp_log.h`" and also include the generated "`sdkconfig.h`".

The high level logging function is called "`esp_log_write()`" which has the following signature:

```
void esp_log_write(esp_log_level_t level, const char *tag, const char * format, …)
```

Think of it like a specialized `printf` logger. The format and following parameters follow the `printf` style convention.

By default, when logging is requested, the output is sent to the primary serial stream. However, we can over-ride that destination by using the function called `esp_log_set_vprintf()`. This takes as a parameter a reference to a C function that has the same syntax as `vprintf`. Specifically:

```
int myPrintFunction(const char *format, va_list arg)
```

When we wish to log a message, we choose a log level to write to. The log levels available are:

- `ESP_LOG_NONE`
- `ESP_LOG_ERROR`
- `ESP_LOG_WARN`
- `ESP_LOG_INFO`
- `ESP_LOG_DEBUG`
- `ESP_LOG_VERBOSE`

The logged output is of the format:

```
<log level> (<time stamp>) <tag>: <message>
```

Where log level is one of "`E`", "`W`", "`I`", "`D`" or "`V`". The time stamp is the number of milliseconds since boot.

We also have a global setting which is the maximum log level we should log. For example if we set `ESP_LOG_WARN` then messages at level `ESP_LOG_WARN`, `ESP_LOG_ERROR` will be logged but `ESP_LOG_INFO`, `ESP_LOG_DEBUG` and `ESP_LOG_VERBOSE` will be excluded.

The tag parameter to the logging function provides an indication of which logical component/module issues the message. This provides context to what otherwise might be ambiguous messages.

C language macros are provided to make using the logging simpler. The macros are:

- `ESP_LOGE(tag, format, …)` - Log an error.
- `ESP_LOGW(tag, format, …)` - Log a warning.
- `ESP_LOGI(tag, format, …)` - Log information.
- `ESP_LOGD(tag, format, …)` - Log debug.
- `ESP_LOGV(tag, format, …)` - Log verbose information.

Since logging is included or excluded at compile time, we can specify the logging level to include in our builds. At compile time, this may exclude certain log statements from the source. The compilation flag `-DLOG_LOCAL_LEVEL` controls the logging levels included.

For the log statements that remain in the code after compilation that were not excluded at build time, we can control the log level at run-time by calling `esp_log_level_set()`. The signature of this function is:

```
void esp_log_level_set(const char *tag, esp_log_level_t level)
```

The tag names the logging groups that we will show. If the special tag of name "`*`" is supplied, this matches all tags.

If we are writing interrupt handling routines, do not use these logging functions within those.

There are a couple of global configuration settings relating to log output that can be set within the "`menuconfig`". These are:

- Default log verbosity – choice – `CONFIG_LOG_DEFAULT_LEVEL`
- Use ANSI terminal colors in log output – boolean – `CONFIG_LOG_COLORS`

See also:

- Error handling
- esp_log_level_set
- esp_log_set_vprintf
- esp_log_write

Exception handling

At run-time, 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 an exception is detected on an ESP32, a register dump is performed on the primary serial output. For example the text may look like:

```
Guru Meditation Error of type LoadProhibited occured on core   0. Exception was un-
handled.
Register dump:
PC      :  400f835d  PS      :  00060a30  A0      :  800f83e9  A1      :  3ffc45a0
A2      :  3f4084bc  A3      :  3ffc4738  A4      :  00000001  A5      :  00000000
A6      :  3ffb013c  A7      :  00000001  A8      :  df405982  A9      :  3ffc4550
A10     :  ffffffff  A11     :  3ffc4738  A12     :  000000ba  A13     :  0000002b
A14     :  0000001b  A15     :  00000001  SAR     :  00000020  EXCCAUSE:  0000001c
EXCVADDR:  df405986  LBEG    :  4000c28c  LEND    :  4000c296  LCOUNT  :  00000000
Rebooting...
```

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.

```
xtensa-esp32-elf-objdump -x app-template.elf -d
```

Another option is to load the binary with GDB as in:

```
xtensa-esp32-elf-gdb ./build/app-template.elf
```

From there we can run:

```
info symbol 0x<address>
```

This will return an indication of where within the code an error was detected. Here is an example:

```
(gdb) info symbol 0x400f8806
initSockets + 114 in section .flash.text
```

This showed that an exception occurred within my "`initSockets`" function (which was indeed the case).

If we then run:

```
list *0x<address>
```

it will show us the source line number and lines. Here is another example:

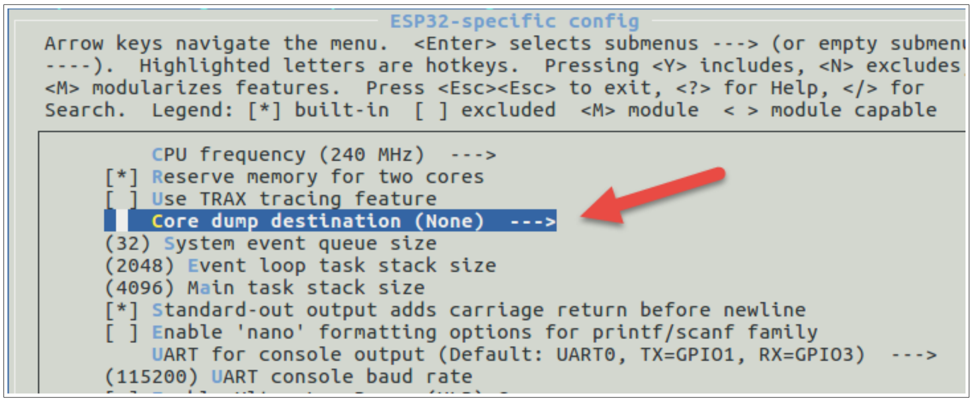
```
(gdb) list *0x400f8806
0x400f8806 is in initSockets
(/home/pi/projects/esp32/apps/myapp/main/./app_main.c:54).
49
50              struct sockaddr_in clientAddress;
51              socklen_t clientAddressLength = sizeof(clientAddress);
52              int clientSock = accept(sock, (struct sockaddr *)&clientAddress,
&clientAddressLength);
        checkSocketRC(clientSock, "accept");
        printf("Accepted a client connection\n");
    }

    static void dumpState() {
        esp_err_t err;
```

Core dump processing

When a computer program runs, there is always "state" associated with that program. At a minimum, there is the current location at which the program is executing (this is called the "program counter"). In addition, there are the values of variables currently in effect as well as the stack which contains information about the nested functions that may have been called to arrive at the current program location. Taken in aggregate, all this information can be extremely useful to you should your program fail or crash unexpectedly. This set of information is sometimes called a memory dump or a system dump and in Unix terms it is called a "core dump". ESP32 architecture has decided to use the term "core dump".

When an ESP32 application fails, we can configure that application to generate a core dump that can subsequently used for analysis. By default, the generation of a core dump is disabled. We can change the setting within "`make menuconfig`" under the settings:

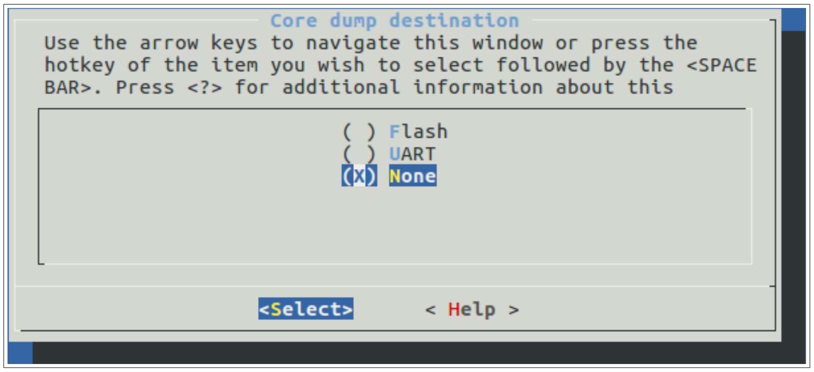
`Component config → ESP32-specific → Core dump destination`



```
                      ESP32-specific config
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenu
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for
  Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

          CPU frequency (240 MHz)  --->
      [*] Reserve memory for two cores
      [ ] Use TRAX tracing feature
          Core dump destination (None)  --->
      (32) System event queue size
      (2048) Event loop task stack size
      (4096) Main task stack size
      [*] Standard-out output adds carriage return before newline
      [ ] Enable 'nano' formatting options for printf/scanf family
          UART for console output (Default: UART0, TX=GPIO1, RX=GPIO3)  --->
      (115200) UART console baud rate
```

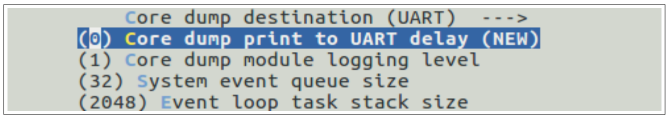
If we select that option, we will find it has three possible settings:

- `Flash` – The core dump data will be written to flash memory.
- `UART` – The core dump data will be encoded and written to the serial (UART) output stream.
- `None` – No core dump information will be saved or written.



```
Core dump destination
Use the arrow keys to navigate this window or press the
hotkey of the item you wish to select followed by the <SPACE
BAR>. Press <?> for additional information about this

( ) Flash
( ) UART
(X) None

<Select>      < Help >
```

If we change the core dump destination to UART, an additional pair of options appear:



```
    Core dump destination (UART)  --->
(0) Core dump print to UART delay (NEW)
(1) Core dump module logging level
(32) System event queue size
(2048) Event loop task stack size
```

These are:

- `Core dump print to UART delay` – A time specified in milliseconds to pause before the core dump data is written to the console. This can be interrupted by a keystroke (assuming a terminal is attached). The default is 0 which means dump immediately.

- `Core dump module logging level` – This is a value between 0 (no output) and 5 (most verbose). It seems that this effects the debug output of the core dump generator as opposed to the content of the core dump itself.

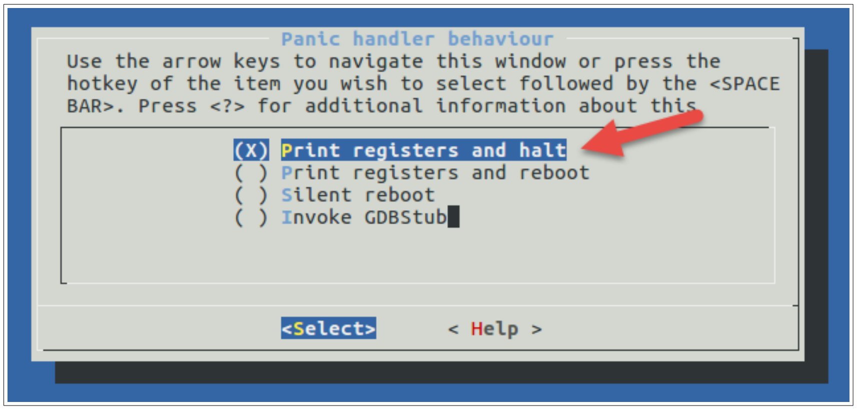
There is a relationship between the core dump setting and what the ESP32 should do as a whole when there is a problem. The options for the ESP32 are found in

`Component config → ESP32-specific config → Panic handler behavior`

The options for that setting are:

- Print registers and halt
- Print registers and reboot
- Silent reboot
- Invoke GDBStub

For use with the core dump, I recommend "`Print registers and halt`".



```
Panic handler behaviour
Use the arrow keys to navigate this window or press the
hotkey of the item you wish to select followed by the <SPACE
BAR>. Press <?> for additional information about this

(X) Print registers and halt
( ) Print registers and reboot
( ) Silent reboot
( ) Invoke GDBStub

<Select>      < Help >
```

Once setup, when a fatal exception occurs, the following is an example of output:

```
Guru Meditation Error of type StoreProhibited occurred on core  0. Exception was
unhandled.
Register dump:
PC      : 0x400e6c23  PS      : 0x00060330  A0      : 0x800d0b41  A1      : 0x3ffb83b0
A2      : 0x00000000  A3      : 0x00000000  A4      : 0x00060023  A5      : 0x3ffb6c54
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x0000007b  A9      : 0x3ffb8390
A10     : 0x000001f6  A11     : 0x3ffb1bf0  A12     : 0x3f4052a8  A13     : 0x0000001f
A14     : 0x00000001  A15     : 0x00000005  SAR     : 0x0000001b  EXCCAUSE: 0x0000001d

EXCVADDR: 0x00000000  LBEG    : 0x00000000  LEND    : 0x00000000  LCOUNT  : 0x00000000

Backtrace: 0x400e6c23:0x3ffb83b0 0x400d0b41:0x3ffb83d0

================= CORE DUMP START =================
kA8AAAYAAABkAQAA
VGz7P/CC+z9QhPs/
cIP7P/CD+z/2AQAASD37P0g9+z9UbPs/QD37PxgAAAAAAAAAYogwEVRs+z8AAAAA
AQAAAFR0+z9tYWluAIgwIiJISgAAAAAAAAAAAFCE+z8AAAAAIAMGAAEAAAAAAAAA
AAAAAAAAAAAAAAAAFT7P2hU+z/QVPs/AAAAAAAAAABAAAAAAAAAAIhCQD8AAAAA
…
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==
================= CORE DUMP END =================
CPU halted.
```

Once we have captured core dump data, the next step is to examine it to determine what it means. By and large it is a blob of binary data that by itself is not meaningful to us. Fortunately the ESP-IDF provides a tool to help us analyze it. That tool is called "`espcoredump.py`".

The primary input is a text file that contains the core dump output. This is the text **between** the "`CORE DUMP START`" and "`CORE DUMP END`" tags. This data is encoded in base 64. Once the file has been written (in the following examine into a file called `core.dat`), we can run the formatting command:

```
python $IDF_PATH/components/espcoredump/espcoredump.py info_corefile -t b64 -c
core.dat build/app-template.elf
```

Here is an example of the output of that tool:

```
===============================================================
==================== ESP32 CORE DUMP START ====================

================== CURRENT THREAD REGISTERS ===================
pc             0x400e6c17  0x400e6c17 <app_main+47>
lbeg           0x0  0
lend           0x0  0
lcount         0x0  0
sar            0x0  0
ps             0x60330     394032
threadptr      <unavailable>
br             <unavailable>
scompare1      <unavailable>
acclo          <unavailable>
acchi          <unavailable>
m0             <unavailable>
m1             <unavailable>
m2             <unavailable>
m3             <unavailable>
expstate       <unavailable>
f64r_lo        <unavailable>
f64r_hi        <unavailable>
f64s           <unavailable>
fcr            <unavailable>
fsr            <unavailable>
a0             0x3ffb83b0  1073447856
a1             0x0  0
a2             0x0  0
a3             0x60023     393251
a4             0x0  0
a5             0x0  0
a6             0x0  0
a7             0x0  0
a8             0x0  0
a9             0x0  0
a10            0x0  0
a11            0x0  0
a12            0x0  0
a13            0x0  0
a14            0x0  0
a15            0x0  0

==================== CURRENT THREAD STACK =====================
#0    0x400e6c17 in app_main () at
/home/kolban/esp32/esptest/apps/workspace/test_core_dump/main/./main.c:29

======================== THREADS INFO =========================
  Id   Target Id         Frame
  6    process 5         0x40082ae7 in xQueueGenericReceive (xQueue=0x0, pvBuffer=0x0,
xTicksToWait=<unavailable>, xJustPeeking=0) at /home/kolban/esp32/esptest/esp-
idf/components/freertos/./queue.c:1594
```

```
  5    process 4         0x40082ae7 in xQueueGenericReceive (xQueue=0x1, pvBuffer=0x1,
xTicksToWait=<unavailable>, xJustPeeking=0) at /home/kolban/esp32/esptest/esp-
idf/components/freertos/./queue.c:1594
  4    process 3         0x40084716 in prvProcessTimerOrBlockTask
(xNextExpireTime=<optimized out>, xListWasEmpty=<optimized out>) at
/home/kolban/esp32/esptest/esp-idf/components/freertos/./timers.c:487
  3    process 2         0x400d13ac in esp_vApplicationIdleHook () at
/home/kolban/esp32/esptest/esp-idf/components/esp32/./freertos_hooks.c:52
  2    process 1         0x400d13ac in esp_vApplicationIdleHook () at
/home/kolban/esp32/esptest/esp-idf/components/esp32/./freertos_hooks.c:52
* 1    <main task>       0x400e6c17 in app_main () at
/home/kolban/esp32/esptest/apps/workspace/test_core_dump/main/./main.c:29

======================= ALL MEMORY REGIONS ========================
Name   Address   Size   Attrs
.rtc.text 0x400c0000 0x0 RW
.iram0.vectors 0x40080000 0x400 R XA
.iram0.text 0x40080400 0x13280 R XA
.dram0.data 0x3ffb0000 0x2434 RW A
.flash.rodata 0x3f400010 0x64b0 RW A
.flash.text 0x400d0018 0x18380 R XA
.coredump.tasks 0x3ffb6c54 0x164 RW
.coredump.tasks 0x3ffb82f0 0x160 RW
.coredump.tasks 0x3ffb8b10 0x164 RW
.coredump.tasks 0x3ffb89a0 0x164 RW
.coredump.tasks 0x3ffb845c 0x164 RW
.coredump.tasks 0x3ffb7190 0x16c RW
.coredump.tasks 0x3ffb92b8 0x164 RW
.coredump.tasks 0x3ffb91c0 0xec RW
.coredump.tasks 0x3ffb631c 0x164 RW
.coredump.tasks 0x3ffb6230 0xe0 RW
.coredump.tasks 0x3ffb5c14 0x164 RW
.coredump.tasks 0x3ffb5b20 0xe8 RW

===================== ESP32 CORE DUMP END =====================
===============================================================
```

At first glance, it looks like it contains a bewildering amount of data … too much to make sense of, however, I hope you will take the time to learn how to interpret it as there is much to see. Lets start at the top section called:

```
CURRENT THREAD REGISTERS
```

These are the CPU registers in effect when the exception happened. Personally, there is little there that I can interpret with one important exception which is "pc". This is the "Program Counter" which is the address of the instruction that was being executed when the exception was encountered. An address by itself though isn't that useful.

Next comes the section called:

```
CURRENT THREAD STACK
```

This is where things get interesting. This contains the call stack of the current thread … the one that contained the exception. Here we will see the path of nested calls that took us to our failure. The final element in the list is exactly where the failure was encountered. For example:

```
#0  0x400e6c17 in app_main () at
/home/kolban/esp32/esptest/apps/workspace/test_core_dump/main/./main.c:29
```

But this time … notice that the source file and exact line number within the source is displayed. We can't get much more precise than this. I believe that this is the most important information that we can gleam from the examination of the core.

Next we find a section called:

```
THREADS INFO
```

This contains a record for each "thread" or FreeRTOS task running in the ESP32. This can provide a context on what else may have been running.

Finally we have a section called:

```
ALL MEMORY REGIONS
```

This lists all the memory sections within the environment. Good to see where things are mapped and useful to see if we might be treading on storage we don't own.

See also:

- ESP-IDF documentation on core dump processing
- Wikipedia – core dump

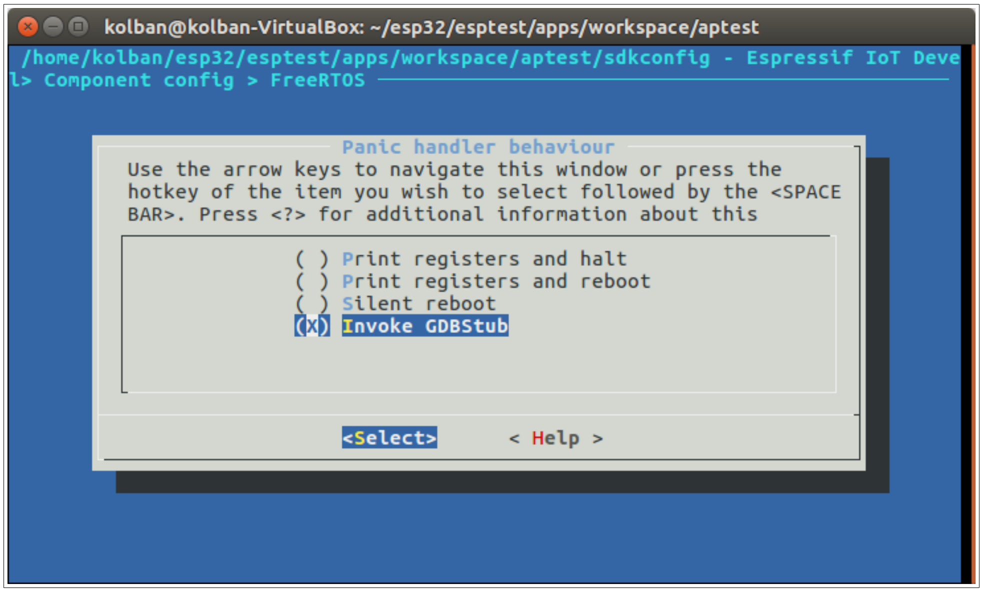
Using a debugger (GDB)

GDB is the GNU Debugger and is an excellent tool for debugging compiled C source code. Although it is primarily designed to debug OS hosted applications such as those compiled for Windows or Linux it can be used to debug code on the ESP32.

First we must change a configuration setting in our compilation process for our ESP32 application using "`make menuconfig`". Visit the menu setting:

Component config → FreeRTOS → Panic handler behavior

From there, select "`Invoke GDBStub`":



```
kolban@kolban-VirtualBox: ~/esp32/esptest/apps/workspace/aptest
 /home/kolban/esp32/esptest/apps/workspace/aptest/sdkconfig - Espressif IoT Deve
l> Component config > FreeRTOS ──────────────────────────────────────────────────

              ┌──────────── Panic handler behaviour ────────────┐
              │  Use the arrow keys to navigate this window or press the
              │  hotkey of the item you wish to select followed by the <SPACE
              │  BAR>. Press <?> for additional information about this
              │ ┌──────────────────────────────────────────────┐
              │ │            ( ) Print registers and halt
              │ │            ( ) Print registers and reboot
              │ │            ( ) Silent reboot
              │ │            (X) Invoke GDBStub
              │ └──────────────────────────────────────────────┘
              ├──────────────────────────────────────────────────┤
              │            <Select>        < Help >
              └──────────────────────────────────────────────────┘
```

Now rebuild and re-deploy your solution. Now the next time a crash/exception happens in your ESP32 application, you will enter a GDB state where you can run `xtensa-esp32-elf-gdb` on your PC and use a serial connection to interact with the ESP32. For example:

```
xtensa-esp32-elf-gdb ./build/app-template.elf -b 115200 -ex 'target remote
/dev/ttyUSB0'
```

You will now be in a gdb session where you can execute the normal gdb commands you would use for debugging.

OpenOCD and JTAG

We must install the Espressif distributed version of OpenOCD. Perform the following:

1. sudo apt get install libusb-1.0
2. git clone --recursive https://github.com/espressif/openocd-esp32.git
3. cd openocd-esp32
4. git submodule init
5. git submodule update
6. ./bootstrap
7. ./configure
8. make
9. cp $IDF_PATH/docs/api-guides/esp32.cfg .

For my testing I bought an FTDI based C232HM-DDHSL-0. These run about $40 on eBay. The pin codings are:

Function	Id	Wire color
VCC	1	Red
TCK – Test Interface Clock	2	Orange
TDI – Test Data Input	3	Yellow
TDO – Test Data Output	4	Green
TMS – Test Mode Select	5	Brown
GPIOL0	6	Grey
GPIOL1	7	Purple
GPIOL2	8	White
GPIOL3	9	Blue
GND	10	Black

Given the four signals needed for JTAG (TCK, TDI, TDO and TMS), these can be found mapped to the following ESP32 pins:

JTAG function	GPIO	Wire color
TCK	GPIO 13	Orange
TDI	GPIO 12	Yellow
TDO	GPIO 15	Green
TMS	GPIO 14	Brown

A clean start of openocd looks like:

```
$ sudo ./src/openocd -s ./tcl -f ./esp32.cfg
Open On-Chip Debugger 0.10.0-dev-ged7b1a9f (2017-07-15-12:33)
Licensed under GNU GPL v2
For bug reports, read
       http://openocd.org/doc/doxygen/bugs.html
none separate
adapter speed: 200 kHz
force hard breakpoints
Info : clock speed 200 kHz
Info : JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
part: 0x2003, ver: 0x1)
Info : JTAG tap: esp32.cpu1 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
part: 0x2003, ver: 0x1)
Info : esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
Info : esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
```

We can restart the ESP32 with "mon reset".

The launch command for gdb is

xtensa-esp32-elf-gdb -ex 'target remote localhost:3333' ./build/app-template.elf

This command says: "Run the xtensa ESP32 `gdb` program and execute the single command that is `target remote localhost:3333` against the executable called `./build/app-template.elf`".

See also:

- JTAG Debugging for ESP32
- Open On-Chip Debugger – OpenOCD
- OpenOCD docs – PDF
- Github: espressif/openocd-esp32
- FTDI C232HM Data Sheet

Using the ESP-WROVER-KIT for JTAG

To use the ESP-WRIVER-KIT for JTAG you have to make some jumper bridges on JP8. Specifically, I find that I needed TMS, TDO, TDI and TCK bridged.

Dumping IP Addresses

Being a WiFi and TCP/IP device, you would imagine that the ESP32 works a lot with IP addresses and you would be right. We can generate a string representation of an IP address using:

```
printf(IPSTR, IP2STR(pIpAddrVar))
```

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

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

which may be more useful in certain situations. The IP2STR macro takes a 32bit integer (an IP address) and produces 4 bytes worth of parameters corresponding to the 4 bytes of an IP address.

See also:

- IP2STR

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 ESP32 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 also:

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

SocketTest

This is a Java app that also works well on Windows. Again, an old app but works exactly as advertized. Nothing fancy.

See also:

- http://sockettest.sourceforge.net/

Linux – netcat (nc)

Netcat can send and receive arbitrary data over TCP/UDP.

For example, to listen on UDP and dump the incoming data:

```
$ nc -u -l -p 9999 | od -x
```

To listen on a TCP port and dump the incoming data:

```
$ nc -l -p 9999 -k | od -x
```

See also:

- man(1) – nc
- man(1) – od

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.

Here are some simple recipes for some of the more common curl commands:

Issue a get against a target:

```
$ curl http://hostname
```

Send data in an HTTP post:

```
$ curl http://hostname --data "data to send"
```

Include an additional header:

```
$ curl http://hostname --header "name: value"
```

Redirect the output to a file

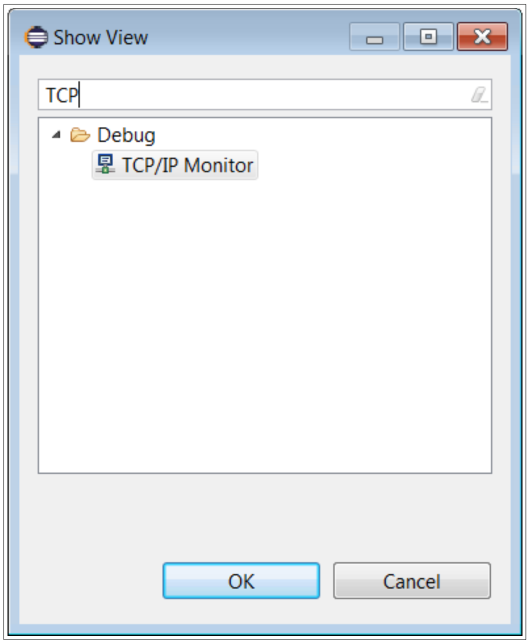
```
$ curl http://hostname --output <filename>
```

See also:

- Making a REST request using Curl
- Curl
- Curl C API
- curl_east_setopt
- libcurl examples

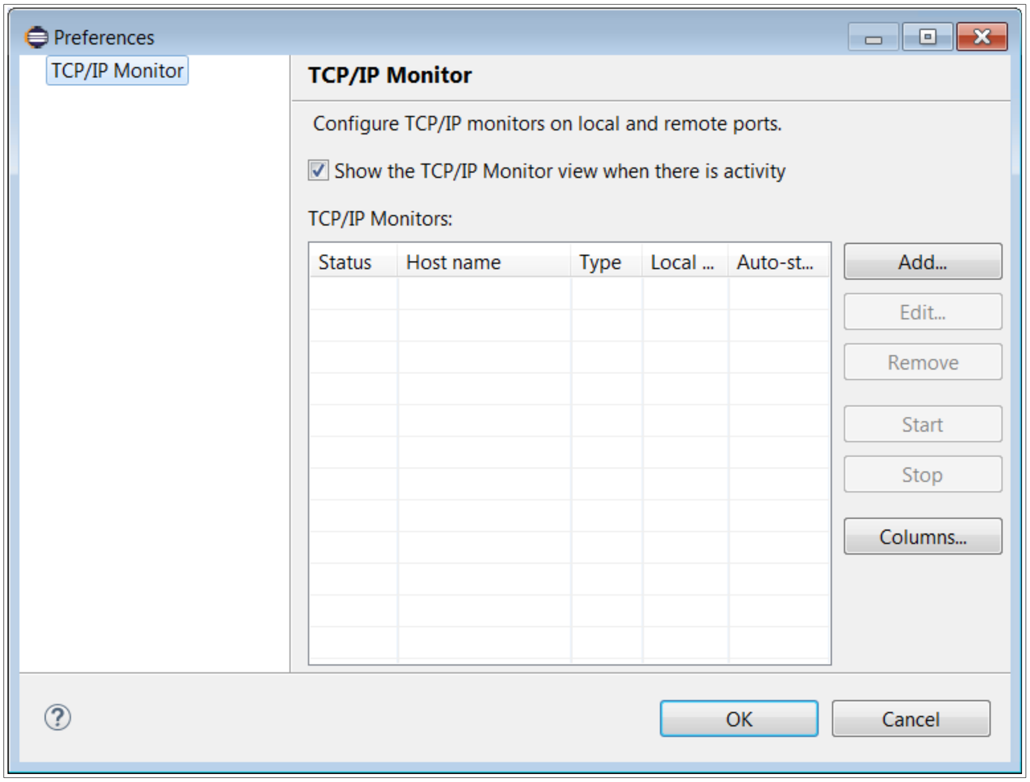
Eclipse – TCP/MON

One of the most powerful and useful tools available is called TCP/IP Monitor that is supplied as part of Eclipse and distributed with the "Eclipse Web Developer Tools". The TCP/IP monitor is opened through the Eclipse view called "TCP/IP Monitor".



TCP

If you can't find it in the view finder, the chances are high that you haven't installed "Eclipse Web Developer Tools". Once launched, open its properties pane:



TCP/IP Monitor

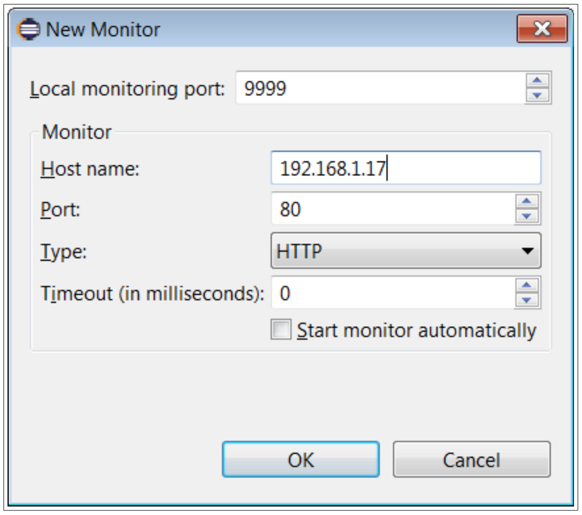
Configure TCP/IP monitors on local and remote ports.

☑ Show the TCP/IP Monitor view when there is activity

TCP/IP Monitors:

Status	Host name	Type	Local ...	Auto-st...

From there you can add local listeners. These will be TCP/IP listeners that listen on a local port where Eclipse is running. During configuration, you specify another IP address and port number. When TCP traffic now arrives at the listener on which TCP/IP Monitor is watching, it will forward that traffic to the partner while at the same time logging it to the TCP/IP Monitor screen.



Local monitoring port: 9999

Monitor

Host name: 192.168.1.17

Port: 80

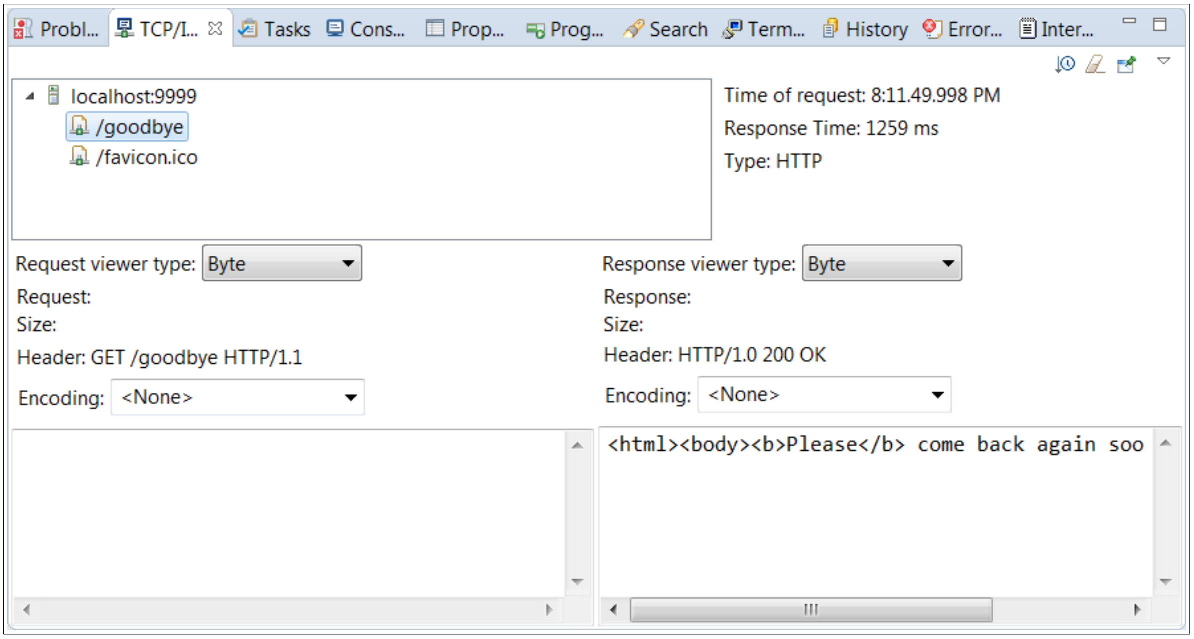
Type: HTTP

Timeout (in milliseconds): 0

□ Start monitor automatically

For example, here TCP/IP Monitor is listening on 192.168.1.2 (localhost) which is where Eclipse is running. It is listening on port 9999. When TCP/IP traffic arrives at that address, it will be sent onwards to 192.168.1.17 (which happens to be my ESP32 device) to port 80.

Here is an example of log I saw when sending a browser request:



localhost:9999
- /goodbye
- /favicon.ico

Time of request: 8:11.49.998 PM

Response Time: 1259 ms

Type: HTTP

Request viewer type: Byte

Request:

Size:

Header: GET /goodbye HTTP/1.1

Encoding: <None>

Response viewer type: Byte

Response:

Size:

Header: HTTP/1.0 200 OK

Encoding: <None>

```
<html><body><b>Please</b> come back again soo
```

As you can see, the information captured here is powerful stuff. We can see each traffic request, its content and HTTP headers.

httpbin.org

When testing HTTP protocols, connecting to the web site at http://httpbin.org can be invaluable. It provides a host of services for testing HTTP requests.

RequestBin

Another excellent HTTP testing resource. We can send a request to a URL and then see exactly what was received by the target. The URL for this service is http://requestb.in/.

tcpdump

This Linux based tool can capture and report on IP traffic into and out of your PC. Where this might be useful for our ESP32 work is that if we direct traffic from the ESP32 to a Linux app, we can look at the low level protocol.

ESP-IDF component debugging

Some of the components of the ESP-IDF have their own debugging techniques and have special instructions for enablement.

LWIP

In `lwipopts.h` change the #define for `LWIP_DEBUG` to `LWIP_DBG_ON`. This is the master debug flag. However there are sub components that then declare what to debug. These include:

- DNS_DEBUG

To enable these, edit the `component.mk` for LWIP and add `CFLAGS` with these variables set to `LWIP_DBG_ON`.

Run a Blinky

Physically looking at an ESP32 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 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.

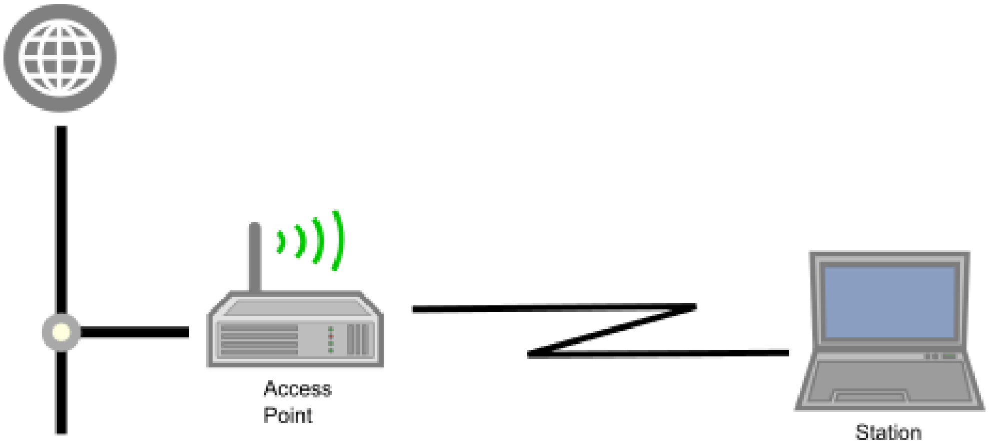
WiFi subsystem

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 ESP32 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

Initializing the WiFi environment

WiFi is only a part of the capabilities of an ESP32. As such, there may be some times when you actually don't want to use the WiFi subsystem. To accommodate those patterns, the initialization of the WiFi subsystem is expected to be performed by you when you write your applications. This is done by calling the `esp_wifi_init()` method. The recommended way of doing this is as follows:

```
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
esp_wifi_init(&config);
```

See also:

- esp_wifi_init

Setting the operation mode

The ESP32 can either be a station in the network, an access point for other devices or both. Remember, when an ESP32 is being a station, it can connect to a remote access point (your WiFi hub) while when being an access point, other WiFi stations can connect to the ESP32 (think of the ESP32 as becoming a WiFi hub). This is a fundamental consideration and we will want to choose how the device behaves early on in our application 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).

This choice is set with a call to `esp_wifi_set_mode()`. The parameter is an instance of `wifi_mode_t` which can have a value of `WIFI_MODE_NULL`, `WIFI_MODE_STA`, `WIFI_MODE_AP` or `WIFI_MODE_APSTA`. We can call `esp_wifi_get_mode()` to retrieve our current mode state.

Scanning for access points

If the ESP32 is going to be 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 `esp_wifi_scan_start()` function.

The results of a WiFi scan are stored internally in ESP32 dynamically allocated storage. The data is returned to us when we call `esp_wifi_scan_get_ap_records()` which also releases the internally allocated storage. As such, this should be considered a destructive read.

A scan record is contained in an instance of a `wifi_ap_record_t` structure that contains:

```
            uint8_t bssid[6]
            uint8_t ssid[32]
            uint8_t primary
  wifi_second_chan_t second
             int8_t rssi
    wifi_auth_mode_t authmode
```

The `wifi_auth_mode_t` is one of:

- `WIFI_AUTH_OPEN` – No security.
- `WIFI_AUTH_WEP` – WEP security.
- `WIFI_AUTH_WPA_PSK` – WPA security.
- `WIFI_AUTH_WPA2_PSK` – WPA2 security.
- `WIFI_AUTH_WPA_WPA2_PSK` – WPA or WPA2 security.

After issuing the request to start performing a scan, we will be informed that the scan completed when a `SYSTEM_EVENT_SCAN_DONE` event is published. The event data contains the number of access points found but that can also be retrieved with a call to `esp_wifi_scan_get_ap_num()`.

Should we wish to cancel the scanning before it completes on its own, we can call `esp_wifi_scan_stop()`.

Here is a complete sample application illustrating performing a WiFi scan. Much of the work is performed in the event handler. When we detect a scan completion event, we retrieve the located access points and log their details.

```
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"

esp_err_t event_handler(void *ctx, system_event_t *event)
{
   if (event->event_id == SYSTEM_EVENT_SCAN_DONE) {
      printf("Number of access points found: %d\n",
         event->event_info.scan_done.number);
      uint16_t apCount = event->event_info.scan_done.number;
      if (apCount == 0) {
         return ESP_OK;
      }
      wifi_ap_record_t *list =
        (wifi_ap_record_t *)malloc(sizeof(wifi_ap_record_t) * apCount);
      ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&apCount, list));
      int i;
      for (i=0; i<apCount; i++) {
         char *authmode;
         switch(list[i].authmode) {
            case WIFI_AUTH_OPEN:
               authmode = "WIFI_AUTH_OPEN";
               break;
            case WIFI_AUTH_WEP:
               authmode = "WIFI_AUTH_WEP";
               break;
            case WIFI_AUTH_WPA_PSK:
               authmode = "WIFI_AUTH_WPA_PSK";
               break;
            case WIFI_AUTH_WPA2_PSK:
               authmode = "WIFI_AUTH_WPA2_PSK";
               break;
            case WIFI_AUTH_WPA_WPA2_PSK:
               authmode = "WIFI_AUTH_WPA_WPA2_PSK";
               break;
            default:
               authmode = "Unknown";
               break;
         }
         printf("ssid=%s, rssi=%d, authmode=%s\n",
            list[i].ssid, list[i].rssi, authmode);
      }
      free(list);
   }
   return ESP_OK;
}

int app_main(void)
```

```
{
   nvs_flash_init();
   tcpip_adapter_init();
   ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
   wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
   ESP_ERROR_CHECK(esp_wifi_init(&cfg));
   ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
   ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
   ESP_ERROR_CHECK(esp_wifi_start());

   // Let us test a WiFi scan ...
   wifi_scan_config_t scanConf = {
      .ssid = NULL,
      .bssid = NULL,
      .channel = 0,
      .show_hidden = 1
   };
   ESP_ERROR_CHECK(esp_wifi_scan_start(&scanConf, 0));

   return 0;
}
```

Using the Arduino libraries we can also make network scans. Here is an example:

```
int8_t count = WiFi.scanNetworks();
printf("Found %d networks\n", count);
for (uint8_t i=0; i<count; i++) {
   String ssid;
   uint8_t encryptionType;
   int32_t RSSI;
   uint8_t *BSSID;
   int32_t channel;
   WiFi.getNetworkInfo(i, ssid, encryptionType, RSSI, BSSID, channel);
   printf("ssid=%s\n", ssid.c_str());
}
```

See also:

- Handling WiFi events
- esp_wifi_scan_start
- esp_wifi_scan_stop
- esp_wifi_scan_get_ap_records
- esp_wifi_scan_get_ap_num

Handling WiFi events

During the course of operating as a WiFi device, certain events may occur that ESP32 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 `esp_event_loop_init()` does just that. It registers a function that will be called when the ESP32 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
- A station connected to us when we are in Access Point mode
- A station disconnected from us when we are in Access Point mode
- A SSID scan completes

When the ESP32 WiFi environment operates, it publishes "events" when something at the WiFi level occurs such as a new station connecting. We can register a callback function that is invoked when an event is published. The signature of the callback function is:

```
esp_err_t eventHandler(void *ctx, system_event_t *event) {
   // Handle event here ...
   return ESP_OK;
}
```

Typically, we need to also include the following:

- #include <esp_event.h>
- #include <esp_event_loop.h>
- #include <esp_wifi.h>
- #include <esp_err.h>

To register the callback function, we invoke:

```
esp_event_loop_init(eventHandler, NULL);
```

If we wish to subsequently change the event handler associated with our WiFi handling we can call:

```
esp_event_loop_set_cb(eventHandler, NULL);
```

When the event handler is invoked, the event parameter is populated with details of the event. The data type of this parameter is a "`system_event_t`" which contains:

```
  system_event_id_t event_id
system_event_info_t event_info
```

We should include "`esp_event.h`" to gain access to these functions and definitions. Let us now look at the two properties passed to the event handler in the system_event_t data structure. These properties are "`event_id`" and "`event_info`". At a high level, the `event_id` describes what kind of event was detected while event_info contains the specific details of the event based on the type identified in `event_id`.

- `event_id` – An enumeration type with the following potential values:
    - `SYSTEM_EVENT_WIFI_READY` – ESP32 WiFi is ready. Although this is currently listed as an event type, it is not actually used and may be removed from the ESP-IDF at some future date. Don't use it in your applications … it will never arrive.
    - `SYSTEM_EVENT_SCAN_DONE` – Finished scanning for access points. The `scan_done` data field is valid to be accessed.
    - `SYSTEM_EVENT_STA_START` – Started being a station.
    - `SYSTEM_EVENT_STA_STOP` – Stopped being a station.
    - `SYSTEM_EVENT_STA_CONNECTED` – Connected to an access point as a station. The `connected` data field is valid to be accessed.
    - `SYSTEM_EVENT_STA_DISCONNECTED` – Disconnected from access point while being a station. The `disconnected` data field is valid to be accessed.
    - `SYSTEM_EVENT_STA_AUTHMODE_CHANGE` – Authentication mode has changed. The `auth_change` data field is valid to be accessed.
    - `SYSTEM_EVENT_STA_GOT_IP` – Got an assigned IP address from the access point that we connected to while being a station. The `got_ip` data field is valid to be accessed.
    - `SYSTEM_EVENT_AP_START` – Started being an access point.
    - `SYSTEM_EVENT_AP_STOP` – Stopped being an access point.
    - `SYSTEM_EVENT_AP_STACONNECTED` – A station connected to us while we are being an access point. The `sta_connected` data field is valid to be accessed.
    - `SYSTEM_EVENT_AP_STADISCONNECTED` – A station disconnected from us while we are being an access point. The `sta_disconnected` data field is valid to be accessed.

  - `SYSTEM_EVENT_AP_PROBEREQRECVED` – Received a probe request while we are being an access point. The `ap_probereqrecved` data field is valid to be accessed.
- `event_info` – This is a C language union of distinct data types that are keyed off the `event_id`. The different structures contained within are:

Structure	**Field**	**Event**
system_event_sta_connected_t	connected	SYSTEM_EVENT_STA_CONNECTED
system_event_sta_disconnected_t	disconnected	SYSTEM_EVENT_STA_DISCONNECTED
system_event_sta_scan_done_t	scan_done	SYSTEM_EVENT_SCAN_DONE
system_event_sta_authmode_change_t	auth_change	SYSTEM_EVENT_STA_AUTHMODE_CHANGE
system_event_sta_got_ip_t	got_ip	SYSTEM_EVENT_STA_GOT_IP
system_event_ap_staconnected_t	sta_connected	SYSTEM_EVENT_AP_STACONNECTED
system_event_ao_stadisconnected_t	sta_disconnected	SYSTEM_EVENT_AP_STADISCONNECTED
system_event_ap_probe_req_rx_t	ap_probereqrecved	SYSTEM_EVENT_AP_PROBEREQRECVED

These data structures contain information pertinent to the event type received.

system_event_sta_connected_t

This data type is associated with the `SYSTEM_EVENT_STA_CONNECT` event.

```
         uint8_t ssid[32]
         uint8_t ssid_len
         uint8_t bssid[6]
         uint8_t channel
wifi_auth_mode_t authmode
```

The `ssid` is the WiFi network name to which we connected. The `ssid_len` is the number of bytes in the `ssid` field that contain the name. The `bssid` is the MAC address of the access point. The `channel` is the wireless channel used for the connection. The `authmode` is the security authentication mode used during the connection.

system_event_sta_disconnected_t

This data type is associated with the `SYSTEM_EVENT_STA_DISCONNECTED` event.

```
uint8_t ssid[32]
uint8_t ssid_len
uint8_t bssid[6]
uint8_t reason
```

The reason code is an indication of why we disconnected. Symbolics are defined for each of the numeric reason codes to allow us to write more elegant and comprehensible applications should we need to consider a reason code.:

- `WIFI_REASON_UNSPECIFIED` – 1
- `WIFI_REASON_AUTH_EXPIRE` – 2
- `WIFI_REASON_AUTH_LEAVE` – 3
- `WIFI_REASON_ASSOC_EXPIRE` – 4
- `WIFI_REASON_ASSOC_TOOMANY` – 5
- `WIFI_REASON_NOT_AUTHED` – 6
- `WIFI_REASON_NOT_ASSOCED` – 7
- `WIFI_REASON_ASSOC_LEAVE` – 8
- `WIFI_REASON_ASSOC_NOT_AUTHED` – 9
- `WIFI_REASON_DISASSOC_PWRCAP_BAD` – 10
- `WIFI_REASON_DISASSOC_SUPCHAN_BAD` – 11
- `WIFI_REASON_IE_INVALID` – 13
- `WIFI_REASON_MIC_FAILURE` – 14
- `WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT` – 15
- `WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT` – 16
- `WIFI_REASON_IE_IN_4WAY_DIFFERS` – 17
- `WIFI_REASON_GROUP_CIPHER_INVALID` – 18
- `WIFI_REASON_PAIRWISE_CIPHER_INVALID` – 19
- `WIFI_REASON_AKMP_INVALID` – 20
- `WIFI_REASON_UNSUPP_RSN_IE_VERSION` – 21
- `WIFI_REASON_INVALID_RSN_IE_CAP` – 22

- `WIFI_REASON_802_1X_AUTH_FAILED` – 23
- `WIFI_REASON_CIPHER_SUITE_REJECTED` – 24
- `WIFI_REASON_BEACON_TIMEOUT` – 200
- `WIFI_REASON_NO_AP_FOUND` – 201
- `WIFI_REASON_AUTH_FAIL` – 202
- `WIFI_REASON_ASSOC_FAIL` – 203
- `WIFI_REASON_HANDSHAKE_TIMEOUT` – 204

system_event_sta_scan_done_t

This data type is associated with the `SYSTEM_EVENT_SCAN_DONE` event.

```
uint32_t status
 uint8_t number
 uint8_t scan_id
```

See also:

- Scanning for access points
- esp_wifi_scan_get_ap_records

system_event_authmode_change_t

This data type is associated with the `SYSTEM_EVENT_STA_AUTHMODE_CHANGE` event.

```
wifi_auth_mode_t old_mode
wifi_auth_mode_t new_mode
```

system_event_sta_got_ip_t

This data type is associated with the `SYSTEM_EVENT_STA_GOT_IP` event.

```
tcpip_adapter_ip_info_t ip_info
```

The `ip_info` element is an instance of a `tcpip_adapter_ip_info_t` which contains three fields:

- `ip` -The IP address.
- `netmask` – The network mask.

- gw – The gateway for communications.

All three of these fields are of ip4_addr_t which is a 32bit representation of an IP address. During development, you might want to consider logging the IP address of the device. You can do this using:

```
ESP_LOGD(tag, "Got an IP: " IPSTR, IP2STR(&event->event_info.got_ip.ip_info.ip));
```

system_event_ap_staconnected_t

This data type is associated with the SYSTEM_EVENT_AP_STACONNECTED event.

```
  uint8_t mac[6]
  uint8_t aid
```

system_event_ap_stadisconnected_t

This data type is associated with the SYSTEM_EVENT_AP_STADISCCONNECTED event.

```
  uint8_t mac[6]
  uint8_t aid
```

system_event_ap_probe_req_rx_t

This data type is associated with the SYSTEM_EVENT_AP_PROBREQRECVED event.

```
      int rssi
 uint8_t mac[6]
```

If we enable the correct logging levels, we can see the events arrive and their content. For example:

```
D (2168) event: SYSTEM_EVENT_STA_CONNECTED, ssid:RASPI3, ssid_len:6, 
bssid:00:00:13:80:3d:bd, channel:6, authmode:3
V (2168) event: enter default callback
V (2174) event: exit default callback
```

and

```
D (9036) event: SYSTEM_EVENT_STA_GOTIP, ip:192.168.5.62, mask:255.255.255.0, 
gw:192.168.5.1
V (9036) event: enter default callback
I (9037) event: ip: 192.168.5.62, mask: 255.255.255.0, gw: 192.168.5.1
V (9043) event: exit default callback
```

Station configuration

When we think of an ESP32 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 set within a data structure called `wifi_sta_config_t`.

The `wifi_sta_config_t` contains:

```
    char ssid[32]
    char password[64]
    bool bssid_set
 uint8_t bssid[6]
```

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.

An example initialization for this structure might be:

```
wifi_config_t staConfig = {
   .sta = {
      .ssid="<access point name>",
      .password="<password>",
      .bssid_set=false
   }
};
```

Once we have populated an instance of this structure, we can instruct ESP32 about its content using:

```
esp_wifi_set_config(WIFI_IF_STA, (wifi_config_t *)&staConfig);
```

We should previously have called `esp_wifi_set_mode()`. with either:

```
esp_wifi_set_mode(WIFI_MODE_STA)
```

or

```
esp_wifi_set_mode(WIFI_MODE_APSTA)
```

See also:

- esp_wifi_set_mode
- esp_wifi_set_config

Starting up the WiFi environment

Since WiFi has states that it must go through, a question that may be asked is "When is WiFi ready to be used?". If we imagine that an ESP32 boots from cold, the chances are that we want to tell it to be either a station or an access point and then configure it with parameters such as which access point to connect to (if it is a station) or what its own access point identity should be (if it is going to be an access point). Given that these are a sequence of steps, we actually don't want the ESP32 to execute on these tasks until after we have performed all our setup. For example, if we boot an ESP32 and ask it to be an access point, if it started being an access point immediately then it may not yet know the details of the access point it should be or, worse, may transiently appear as the wrong access point. As such, there is final command that we must learn which is the instruction to the WiFi subsystem to start working. That command is `esp_wifi_start()`. Prior to calling that, all we are doing is setting up the environment. Only by calling `esp_wifi_start()` does the WiFi subsystem start doing any real work on our behalf. If our mode is that of an access point, calling this function will start us being an access point. If our mode is that of a station, now we are allowed to subsequently connect as a station. There is a corresponding command called `esp_wifi_stop()` which stops the WiFi subsystem.

See also:

- esp_wifi_start
- esp_wifi_stop

Connecting to an access point

Once the ESP32 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 `esp_wifi_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 ESP32 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 `SYSTEM_EVENT_STA_CONNECTED` indicating that we have connected to the access point. The second event is `SYSTEM_EVENT_STA_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.

Should we disconnect from an access point, we will see a `SYSTEM_EVENT_STA_DISCONNECTED` event. To disconnect from a previously connected access point we issue the `esp_wifi_disconnect()` call.

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 ESP32 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 `esp_wifi_set_auto_connect()`.

Here is a complete sample illustrating all the steps needed to connect to an access point and be informed when we are ready to being work:

```
#include "freertos/FreeRTOS.h"
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "tcpip_adapter.h"

esp_err_t event_handler(void *ctx, system_event_t *event)
{
   if (event->event_id == SYSTEM_EVENT_STA_GOT_IP) {
      printf("Our IP address is " IPSTR "\n",
         IP2STR(&event->event_info.got_ip.ip_info.ip));
      printf("We have now connected to a station and can do things...\n")
   }
   return ESP_OK;
}

int app_main(void)
{
   nvs_flash_init();
   tcpip_adapter_init();
   ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
   wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
   ESP_ERROR_CHECK(esp_wifi_init(&cfg) );
   ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
   ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA));
   wifi_config_t sta_config = {
      .sta = {
         .ssid = "RASPI3",
         .password = "password",
         .bssid_set = 0
      }
   };
   ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &sta_config));
   ESP_ERROR_CHECK(esp_wifi_start());
   ESP_ERROR_CHECK(esp_wifi_connect());

   return 0;
}
```

When we connect to an access point, our device is being a station. The connection to the access point doesn't automatically mean that we now have an IP address. We still have to request an allocated IP address from the DHCP server. This can take a few seconds. In some cases, we can get away with the device requesting a specific IP address. This results in a much faster connection time. If we do specify data, we also need to supply DNS information should we need to connect to DNS servers for name resolution.

Here is a fragment of logic that allocates us a specific IP address:

```
#include <lwip/sockets.h>

// The IP address that we want our device to have.
#define DEVICE_IP          "192.168.1.99"

// The Gateway address where we wish to send packets.
// This will commonly be our access point.
#define DEVICE_GW          "192.168.1.1"

// The netmask specification.
#define DEVICE_NETMASK     "255.255.255.0"

// The identity of the access point to which we wish to connect.
#define AP_TARGET_SSID     "RASPI3"

// The password we need to supply to the access point for authorization.
#define AP_TARGET_PASSWORD "password"

esp_err_t wifiEventHandler(void *ctx, system_event_t *event)
{
    return ESP_OK;
}


// Code fragment here ...
  nvs_flash_init();
  tcpip_adapter_init();

  tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA); // Don't run a DHCP client
  tcpip_adapter_ip_info_t ipInfo;

  inet_pton(AF_INET, DEVICE_IP, &ipInfo.ip);
  inet_pton(AF_INET, DEVICE_GW, &ipInfo.gw);
  inet_pton(AF_INET, DEVICE_NETMASK, &ipInfo.netmask);
  tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);

  ESP_ERROR_CHECK(esp_event_loop_init(wifiEventHandler, NULL));
  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK(esp_wifi_init(&cfg) );
  ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
  ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
  wifi_config_t sta_config = {
    .sta = {
      .ssid      = AP_TARGET_SSID,
      .password  = AP_TARGET_PASSWORD,
      .bssid_set = 0
    }
  };
  ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &sta_config));
  ESP_ERROR_CHECK(esp_wifi_start());
  ESP_ERROR_CHECK(esp_wifi_connect());
```

See also:

- Handling WiFi events
- esp_wifi_connect
- esp_wifi_disconnect

Being an access point

So far we have only considered the ESP32 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 ESP32s.

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 found in a scan. 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 ESP32 as such using the `esp_wifi_set_mode()` function and pass in the flag that requests we be either a dedicated access point or an access point **and** a station. This will be either:

```
esp_wifi_set_mode(WIFI_MODE_AP);
```

or

```
esp_wifi_set_mode(WIFI_MODE_APSTA);
```

Next we need to supply the configuration information. We do this by populating an instance of `wifi_ap_config_t`. The `wifi_ap_config_t` contains:

- `ssid` – The WiFi ssid name upon which we will listen for connecting stations.
- `ssid_len` – The length in bytes of the ssid if not NULL terminated.

- `password` – The password used for station authentication.
- `channel` – The channel we will use for networking.
- `authmode` – How we wish stations to authenticate (if at all). The choices are
  - open
  - wep
  - wpa
  - wpa2
  - wpa_wpa2
- `ssid_hidden` – Should we broadcast our ssid.
- `max_connection` – The number of concurrent stations. The default and maximum is 4.
- `beacon_interval` – Unknown. 100.

An example of initialization of this structure might be:

```
wifi_config_t apConfig = {
   .ap = {
      .ssid="<access point name>",
      .ssid_len=0,
      .password="<password>",
      .channel=0,
      .authmode=WIFI_AUTH_OPEN,
      .ssid_hidden=0,
      .max_connection=4,
      .beacon_interval=100
   }
};
```

With the structure populated, we call `esp_wifi_set_config()` … for example:

```
esp_wifi_set_config(WIFI_IF_AP, &apConfig);
```

Finally, we call `esp_wifi_start()`.

Here is a snippet of code that can be used to setup and ESP32 as an access point:

When we become an access point, an ESP32 WiFi event is produced of type `SYSTEM_EVENT_AP_START`. Note that there is no payload data associated with this event.

Once the ESP32 starts listening for station connects by being an access point, we are going to want to validate that this works. You can use any device or system to scan and connect. Personally, I use a Raspberry PI 3 for testing as it provides a nice Linux environment and has a WiFi adapter build in. You can also choose to plug in a separate

WiFi dongle into one of the extra USB ports. One of the first tools we want to run is called "`iwlist`" which will perform a scan for us:

```
$ sudo iwlist wlan1 scan
```

In the results, we can look for our ESP32 … for example:

```
Cell 02 - Address: 18:FE:34:6A:94:EF
          ESSID:"ESP32"
          Protocol:IEEE 802.11bgn
          Mode:Master
          Frequency:2.412 GHz (Channel 1)
          Encryption key:off
          Bit Rates:150 Mb/s
          Quality=100/100  Signal level=100/100
```

One of the other tools available on that environment is called "`wpa_cli`" which provides a wealth of options for testing WiFi. The recipe I use is to connect to an access point from the command line is:

```
$ sudo wpa_cli
add_network
set_network <num> ssid "<SSID>"
set_network <num> key_mgmt NONE
enable_network <num>
status
```

You may have to run

```
select_network <num>
reconnect
```

or

```
reasociate
```

to connect to the target and you can run

```
disconnect
```

to disconnect from the access point.

`ifname` – show current interface

`interface <name>` - select current interface

To perform a scan run the command "`scan`". When complete, run "`scan_results`" to see the list.

When a station connects, the ESP32 will raise the `SYSTEM_EVENT_AP_STACONNECTED` event. When a station disconnects, we will see the `SYSTEM_EVENT_AP_DISCONNECTED` event.

See also:

- man(8) – wpa_cli

When a remote station connects to the ESP32 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 ESP32, 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 `wifi_sta_list_t`. 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);
wifi_sta_list_t *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();
}
```

When an ESP32 acts as an access point, this allows other devices to connect to it and form a WiFi connection. However, it appears that two devices connected to the same ESP32 acting as an access point can not directly communicate between each other. For example, imagine two devices connecting to an ESP32 as an access point. They may be allocated the IP addresses 192.168.4.2 and 192.168.4.3. We might imagine that 192.168.4.2 could ping 192.168.4.3 and visa versa but that is not allowed. It appears that they only direct network connection permitted is between the newly connected stations and the access point (the ESP32) itself.

This seems to limit the applicability of the ESP32 as an access point. The primary intent of the ESP32 as an access point is to allow mobile devices (eg. your phone) to connect to the ESP32 and have a conversation with an application that runs upon it.

See also:

- esp_wifi_set_config
- esp_wifi_set_mode

Working with connected stations

When our ESP32 is being an access point, we are saying that we wish to allow stations to connect to it. This brings in the story of managing those stations. Common things we might want to do are:

- Determine when a new station connects
- Determine when a previously connected station leaves
- List the currently connected stations
- Disconnect one or more currently connected stations

We can register an event handler for detecting new station connects and existing station disconnects. The event handler will receive `SYSTEM_EVENT_AP_STACONNECTED` when a station connects and `SYSTEM_EVENT_AP_STADISCONNECTED` what a station leaves.

We can get the list of currently connected stations using the `esp_wifi_get_station_list()` function. This returns a linked list of stations. The storage for this list is allocated for us and we should indicate that we are no longer in need of it by calling `esp_wifi_free_station_list()` when done.

See also:

- Handling WiFi events
- esp_wifi_free_station_list
- esp_wifi_get_station_list

WiFi at boot time

The ESP32 can store WiFi start-up information in flash memory. This allows it to perform its functions at start-up without having to ask the user for any special or additional information. This capability is controlled by a function called `esp_wifi_set_auto connect()` and its partner called `esp_wifi_get_auto_connect()`. The values of the settings used for an auto connect are those that are saved in flash memory. These are the values set when we call `esp_wifi_set_config()` but **only** if we have instructed the ESP32 to record those settings to flash. This is itself controlled by a call to `esp_wifi_set_storage()`.

See also:

- esp_wifi_set_auto_connect
- esp_wifi_get_auto_connect
- esp_wifi_set_storage

The DHCP client

When the ESP32 connects to an access point as a station, it also runs a DHCP client to connect to the DHCP server that it assumes is also available at the access point. From there, the station is supplied its IP address, gateway address and netmask. There are times however when we want to supply our own values for this data. We can do this by calling `tcpip_adapter_set_ip_info()` during setup. The recipe is as follows:

```
tcpip_adapter_init();
tcpip_adapter_dhcpc_stop();
tcpip_adapter_set_ip_info();
esp_wifi_init();
esp_wifi_set_mode();
esp_wifi_set_config();
esp_wifi_start();
esp_wifi_config();
```

(Note that the parameters are omitted in the above).

The setup for calling `tcpip_adapter_set_ip_info()` can be as follows:

```
tcpip_adapter_ip_info_t ipInfo;
IP4_ADDR(&ipInfo.ip, 192,168,1,99);
IP4_ADDR(&ipInfo.gw, 192,168,1,1);
IP4_ADDR(&ipInfo.netmask, 255,255,255,0);
tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);
```

Alternative, using strings we have:

```
tcpip_adapter_ip_info_t ipInfo;
inet_pton(AF_INET, "192.168.1.99", &ipInfo.ip);
inet_pton(AF_INET, "192.168.1.1", &ipInfo.gw);
inet_pton(AF_INET, "255.255.255.0", &ipInfo.netmask);
tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);
```

See also:

- tcpip_adapter_set_ip_info
- tcpip_adapter_dhcpc_start
- tcpip_adapter_dhcpc_stop
- tcpip_adapter_dhcpc_get_status
- tcpip_adapter_dhcpc_option
- inet_pton

The DHCP server

When the ESP32 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:

- Error: Reference source not found

Current IP Address, netmask and gateway

Should we need it, we can query the 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 `tcpip_adapter_get_ip_info()` returns our current value. Since the ESP32 can have two IP interfaces (one for an access point and one for a station), we supply which interface we wish to retrieve.

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.

See also:

- Handling WiFi events
- Error: Reference source not found
- tcpip_adapter_get_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

Designs for bootstrapping WiFi

Imagine that we have built a project using an ESP32 that wishes to be network connected. In order for that to happen, we want the ESP32 to connect to an existing access point. That's works, because the ESP32 can be a WiFi station. In order for the ESP32 to connect to an access point, it needs to know two important items. It needs to know which network to join (the SSID) and it will need to know the password to use to connect to that network as most networks require authentication. And there is the puzzle. If the ESP32 is brought to a physically new environment, how will it "know" which network to connect with and what password to use? We should assume that the ESP32 doesn't have a screen attached to it. If it did, we could prompt the user for the information.

One solution is to have the ESP32 initially "be" an access point. If it were an access point then we could use our phone to connect with it, ask it what WiFi networks it can see, provide a password for the network and allow it to connect.

```
while (not done) {
   if (we know our ssid and password) {
      attempt to connect to the access point;
      if (we succeeded in the connection) {
         return;
      }
   }
   become an access point ourselves;
   listen for incoming browser requests;
   wait for an SSID/password pair to be entered;
}
```

We also need to handle the case where we think we have an SSID and password used to connect to an access point but either those have changed or else we are in a foreign location. In that case we must also fall back to being an access point and await new instructions.

We can use non-volatile storage to save our SSID and password. We may wish to save not just one SSID/password pair but perhaps save an ordered list. That way when we teach our device how to connect to an access point and then later teach it how to connect to another one, we might end up back at the first. For example, imagine using an ESP32 at home with one network and the same ESP32 at work with a different network.

We may also want to save static interface information if we either don't have or don't want to use the services of a DHCP server when we start as a station.

See also:

- Non Volatile Storage

Working with TCP/IP

TCP/IP is the network protocol that is used on the Internet. It is the protocol that the ESP32 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 (4 x 8=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 Lightweight IP Stack – lwip

If we think of TCP/IP as a protocol then we can break up our understanding of networking into two distinct layers. One is the hardware layer that is responsible for getting a stream of 1's and 0's from one place to another. Common implementations for that include Ethernet, Token Ring and (yes … I'm dating myself now … dial-up modems). These are characterized by physical wires from your devices. WiFi is itself a transport layer. It deals with using radio waves as the communication medium of 1's and 0's between two points. The specification for WiFI is IEEE 802.11.

Once we can transmit and receive data, the next level is organizing the data over that physical network and this is where TCP/IP comes into play. It provides the rules and governance of data transmission, addressing, routing, protocol negotiations and more. Typically, TCP/IP is implemented in software over the underlying physical transport mechanism. Think about this a moment. Imagine I said to you that I have a "magic box" and if you put something in that box, it will magically be transported to a different box. That is the analogy of physical transport. The software that is TCP/IP adds mechanisms above that. For example, imagine the box is only 6 inches wide. If you want to send me something through our boxes, you have to chop it up and send it in pieces. Your end of the box story handles that. My box will receive the parts and re-assemble them for me. Parts may arrive in order and some parts may even get lost on route and have to be re-sent from the originals. The hardware (the boxes) have no idea how to achieve that. All they know is a piece of data in one end will hopefully arrive at the other … but not guaranteed.

TCP/IP is a big protocol. It contains lots of parts. Fortunately it is well specified and has been implemented by many vendors over the last 45 years. Some of the implementations of the whole stack of TCP/IP parts have been written as open source and are distributed and maintained by the community. What this means is that if one has a new hardware layer, one can (in principle) lift an already written implementation of TCP/IP, map it to your hardware, compile it for your environment and you are good to go. This is actually much easier said than done … and fortunately for us, our friends at Espressif have done the work for us.

One such open source implementation of a TCP/IP stack is called "The LightweightIPStack" which is commonly referred to as "lwIP". This can be read about in detail at its home page (see the references). As part of the distribution of the ESP-IDF, we have libraries that provide an implementation lwIP. It is lwIP that provides the ESP32 the following services:

- IP
- ICMP
- IGMP
- MLD
- ND
- UDP

- TCP
- sockets API
- DNS

Again, the good news is that the vast majority of lwIP is of no importance to us, ESP32 application designers and developers. It is vitally important … but important to the internal operation of ESP32 and not exposed to us as consumers.

See also:

- lwIP 2.0.0

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 running 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 your phone 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 ESP32 can set itself up as a listener for an incoming TCP/IP connection and this requires that we begin to understand the important "sockets" API.

TCP/IP Sockets

The sockets API is a programming interface for working with TCP/IP networking. It is probably the most familiar API for network programming. Sockets programming is familiar to programmers on Linux, Windows, Java and more.

TCP/IP network flows come in two flavors … connection oriented over TCP and datagram oriented over UDP. The sockets API provides distinct patterns of calls for both styles.

For TCP, a server is built by:

1. Creating a TCP socket
2. Associating a local port with the socket
3. Setting the socket to listen mode
4. Accepting a new connection from a client
5. Receive and send data
6. Close the client/server connection
7. Going back to step 4

For a TCP client, we build by:

1. Creating a TCP socket
2. Connecting to the TCP server
3. Sending data/receiving data
4. Close the connection

Now let us break these up into code fragments that we can analyze in more depth. The header definitions for the sockets API can be found in `<lwip/sockets.h>`.

For both the client and the server applications, the task of creating a socket is the same. It is an API call to the `socket()` function.

```
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
```

The return from `socket()` is an integer handle that is used to refer to the socket. Sockets have lots of state associated with them, however that state is internal to the TCP/IP and sockets implementation and need not be exposed to the network programmer. As such, there is no need to expose that data to the programmer. We can think of calling `socket()` as asking the run-time to create and initialize all the data necessary for a network communication. That data is owned by the run-time and we are passed a "reference number" or handle that acts as a proxy to the data. When ever we wish to subsequently perform work on that network connection, we pass back in that handle that was previously issued to us and we can correlate back to the connection.

This isolates and insulates the programmer from the guts of the implementation of TCP/IP and leaves us with a useful abstraction.

When we are creating a server side socket, we want it to listen for incoming connection requests. To do this, we need to tell the socket which TCP/IP port number it should be listening upon. Note that we don't supply the port number a directly from an int/short value. Instead we supply the value as returned by the `htons()` function. What this function does is convert the number into what is called "network byte order". This is the byte order that has been chosen by convention to be that used for transmitting unsigned multi byte binary data over the internet. It's actual format is "big endian" which means that if we take a number such as 9876 (decimal) then it is represented in binary as 00100110 10010100 or 0x26D4 in hex. For network byte order, we first transmit 00100110 (0x26) followed by 10010100 (0xD4). It is important to realize that the ESP32 is a **little endian** native architecture which means that we absolutely **must** transform 2 byte and 4 byte numbers into network byte order (big endian).

On a given device, only one application at a time can be using any given local port number. If we want to associate a port number with an application, such as our server application in this case, we perform a task called "binding" which binds (or assigns) the port number to the socket which in turn is owned by the application.

```
struct sockaddr_in serverAddress;
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(portNumber);
bind(sock, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
```

With the socket now associated with a local port number, we can request that the run-time start listening for incoming connections. We do this by calling the `listen()` API. Before calling `listen()`, connections from clients would have been rejected with an indication to the client that there was nothing at the corresponding target address. Once we call `listen()`, the server will start accepting incoming client connections. The API looks like:

```
listen(sock, backlog)
```

The backlog is the number of connection requests that the run-time will listen for and accept before they are handed off to the application for processing. The way to think about this is imagine that you are the application and you can only do one thing at a time. For example, you can only be talking to one person at a time on the phone. Now imagine you have a secretary who is handling your incoming calls. When a call arrives and you are not busy, the secretary hands off the call to you. Now imagine that you are busy. At that time, the secretary answers the phone and asks the caller to wait. When you free up, she hands you the waiting call. Now let us assume that you are still busy when yet another client calls. She also tells this caller to wait. We are starting to build a queue of callers. And this is where the backlog concept comes into play. The backlog instructs the run-time how many calls can be received and asked to wait. If more calls arrive than our backlog will allow, the run-time rejects the call immediately. Not only does this prevent run-away resource consumption at the server, it also can be used as an indication to the caller that it may be better served trying elsewhere.

Now from a server perspective, we are about ready to do some work. A server application can now block waiting for incoming client connections. The thinking is that a server application's purpose in life is to handle client requests and when it doesn't have an active client request, there isn't anything for it to do but wait for a request to arrive. While that is certainly one model, it isn't necessarily the only model or even the best model (in all cases). Normally we like our processors to be "utilized". Utilized means that while it has productive work it can do, then it should do it. If the only thing our program can do is service client calls, then the original model makes sense. However, there are certain programs that if they don't have a client request to immediately service, might spend time doing something else that is useful. We will come back to that notion later on. For now, we will look at the `accept()` function call. When `accept()` is called, one of two things will happen. If there is no client connection immediately waiting for us, then we will block until such time in the future when a client connection does arrive. At that time we will wake up and be handed the connection to the newly arrived client. If on the other hand we called `accept()` and there was already a client connection waiting for us, we will immediately be handed that connection and we carry on. In both cases, we call `accept()` and are returned a connection to a client. The distinction between the cases is whether or not we have to wait for a connection to arrive.

The API call looks like:

```
struct sockaddr_in clientAddress;
socklen_t clientAddressLength = sizeof(clientAddress);
int clientSock = accept(sock, (struct sockaddr *)&clientAddress,
&clientAddressLength);
```

The return from `accept()` is a new socket (an integer handle) that represents the connection between the requesting client and the server. It is vital to realize that this is distinct from the server socket we created earlier which we bound to our server listening port. That socket is still alive and well and exists to continue to service further client connections. The newly returned socket is the connection for the conversation that was initiated by this single client. Like all TCP connections, the conversation is symmetric and bi-directional. This means that there is now no longer the notion of a client and server … both parties can send and receive as they would like at any time.

If we wish to create a socket client, the story is similar. Again we create a `socket()` but this time there is no need for a `bind()`/`listen()`/`accept()` story. Instead we use the `connect()` API to connect to the target TCP/IP endpoint.

For example:

```
struct sockaddr_in serverAddress;
serverAddress.sin_family = AF_INET;
inet_pton(AF_INET, "192.168.1.200", &serverAddress.sin_addr.s_addr);
serverAddress.sin_port = htons(9999);

int rc = connect(sock, (struct sockaddr *)&serverAddress, sizeof(struct sockaddr_in));
```

See also:

- Native byte order, endian and network byte order
- socket
- bind
- listen
- accept
- send
- recv
- connect
- Wikipedia – Berkeley Sockets
- Beej's Guide to Network Programming

Handling errors

Most of the sockets APIs return an int return code. If this code is < 0 then an error has occurred.

The nature of the error can be found using the global int called "`errno`". However, in a multitasking environment, working with global variables is not recommended. In the sockets area, we can ask a socket for the last error it encountered using the following code fragment:

```
int espx_last_socket_errno(int socket) {
   int ret = 0;
   u32_t optlen = sizeof(ret);
   getsockopt(socket, SOL_SOCKET, SO_ERROR, &ret, &optlen);
   return ret;
}
```

The meanings of the errors can be compared against constants. Here is a table of constants used in the current FreeRTOS implementation:

Symbol	Value	Description
EPERM	1	Operation not permitted

ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Arg list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Device or resource busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func
ERANGE	34	Math result not representable
EDEADLK	35	Resource deadlock would occur
ENAMETOOLONG	36	File name too long
ENOLCK	37	No record locks available

ENOSYS	38	Function not implemented
ENOTEMPTY	39	Directory not empty
ELOOP	40	Too many symbolic links encountered
EWOULDBLOCK EAGAIN	41	Operation would block
ENOMSG	42	No message of desired type
EIDRM	43	Identifier removed
ECHRNG	44	Channel number out of range
EL2NSYNC	45	Level 2 not synchronized
EL3HLT	46	Level 3 halted
EL3RST	47	Level 3 reset
ELNRNG	48	Link number out of range
EUNATCH	49	Protocol driver not attached
ENOCSI	50	No CSI structure available
EL2HLT	51	Level 2 halted
EBADE	52	Invalid exchange
EBADR	53	Invalid request descriptor
EXFULL	54	Exchange full
ENOANO	55	No anode
EBADRQC	56	Invalid request code
EBADSLT	57	Invalid slot
EBFONT	59	Bad font file format
ENOSTR	60	Device not a stream
ENODATA	61	No data available
ETIME	62	Timer expired
ENOSR	63	Out of streams resources
ENONET	64	Machine is not on the network
ENOPKG	65	Package not installed
EREMOTE	66	Object is remote
ENOLINK	67	Link has been severed
EADV	68	Advertise error
ESRMNT	69	Srmount error
ECOMM	70	Communication error on send
EPROTO	71	Protocol error
EMULTIHOP	72	Multihop attempted
EDOTDOT	73	RFS specific error
EBADMSG	74	Not a data message
EOVERFLOW	75	Value too large for defined data type

ENOTUNIQ	76	Name not unique on network
EBADFD	77	File descriptor in bad state
EREMCHG	78	Remote address changed
ELIBACC	79	Can not access a needed shared library
ELIBBAD	80	Accessing a corrupted shared library
ELIBSCN	81	.lib section in a.out corrupted
ELIBMAX	82	Attempting to link in too many shared libraries
ELIBEXEC	83	Cannot exec a shared library directly
EILSEQ	84	Illegal byte sequence
ERESTART	85	Interrupted system call should be restarted
ESTRPIPE	86	Streams pipe error
EUSERS	87	Too many users
ENOTSOCK	88	Socket operation on non-socket
EDESTADDRREQ	89	Destination address required
EMSGSIZE	90	Message too long
EPROTOTYPE	91	Protocol wrong type for socket
ENOPROTOOPT	92	Protocol not available
EPROTONOSUPPORT	93	Protocol not supported
ESOCKTNOSUPPORT	94	Socket type not supported
EOPNOTSUPP	95	Operation not supported on transport endpoint
EPFNOSUPPORT	96	Protocol family not supported
EAFNOSUPPORT	97	Address family not supported by protocol
EADDRINUSE	98	Address already in use
EADDRNOTAVAIL	99	Cannot assign requested address
ENETDOWN	100	Network is down
ENETUNREACH	101	Network is unreachable
ENETRESET	102	Network dropped connection because of reset
ECONNABORTED	103	Software caused connection abort
ECONNRESET	104	Connection reset by peer
ENOBUFS	105	No buffer space available
EISCONN	106	Transport endpoint is already connected
ENOTCONN	107	Transport endpoint is not connected
ESHUTDOWN	108	Cannot send after transport endpoint shutdown
ETOOMANYREFS	109	Too many references: cannot splice
ETIMEDOUT	110	Connection timed out
ECONNREFUSED	111	Connection refused

EHOSTDOWN	112	Host is down
EHOSTUNREACH	113	No route to host
EALREADY	114	Operation already in progress
EINPROGRESS	115	Operation now in progress
ESTALE	116	Stale NFS file handle
EUCLEAN	117	Structure needs cleaning
ENOTNAM	118	Not a XENIX named type file
ENAVAIL	119	No XENIX semaphores available
EISNAM	120	Is a named type file
EREMOTEIO	121	Remote I/O error
EDQUOT	122	Quota exceeded
ENOMEDIUM	123	No medium found
EMEDIUMTYPE	124	Wrong medium type

Configuration settings

Within the "`menuconfig`" there are some settings that relate to TCP/IP and can be found within the lwIP settings. The settings are:

- Max number of open sockets – integer – `CONFIG_LWIP_MAX_SOCKETS` – This is the number of concurrently open sockets. The default is 4 and the maximum appears to be 16.
- Enable SO_REUSEADDR – boolean – `LWIP_SO_REUSE` –

Using select()

Imagine that we have multiple sockets each of which may be the source of incoming data. If we try and `read()` data from a socket, we normally block until data is ready. If we did this, then if data becomes available on another socket, we wouldn't know. An alternative is to try and read data in a non-blocking fashion. This too would be useful but would require that we test each socket in turn in a busy or polling fashion. This too is not optimal. Ideally what we would like to do is block while watching multiple sockets simultaneously and wake up when the *first* one has something useful for us to do.

See also:

- select
- The world of select()

Differences from "standard" sockets

Two header files that are commonly found in other sockets implementations are not part of the ESP-IDF definition. They are:

- netinet/in.h
- arpa/inet.h

Despite not being present, no obvious issues have been found and it is assumed that the content normally contained within has been distributed across other headers.

UDP/IP Sockets

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 re-transmit 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.

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 12345, 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 12345 over a TCP connection, it is routed to the TCP application watching port 12345. If a request arrives at the same machine for port 12345 over UDP, it is routed to the UDP application watching port 12345. 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 ESP32 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.

To program with UDP, once again we use sockets. To set up a socket server using UDP again we call `socket()` to create a socket and again we call `bind()` to specify the port number we wish to listen upon. There is no need for a call to `listen()`. When the server is ready to receive an incoming request, we call `recvfrom()` which blocks until a datagram is received. Once one arrives, we wake up and can process the request. The request contains a return address and we can send a response using `sendto()` should we wish.

On the client side, we create a `socket()` and then can invoke `sendto()`. The call to `sendto()` takes the IP address and port of the target as parameters as well as the payload data.

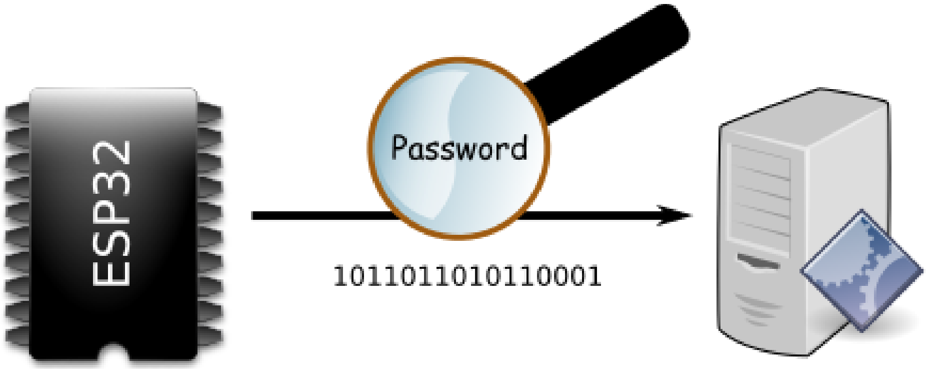
For example:

```
int socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
sendto(socket_fd, data, size, 0, destAddr, destAddrLen);
```

- socket
- sendto
- recvfrom

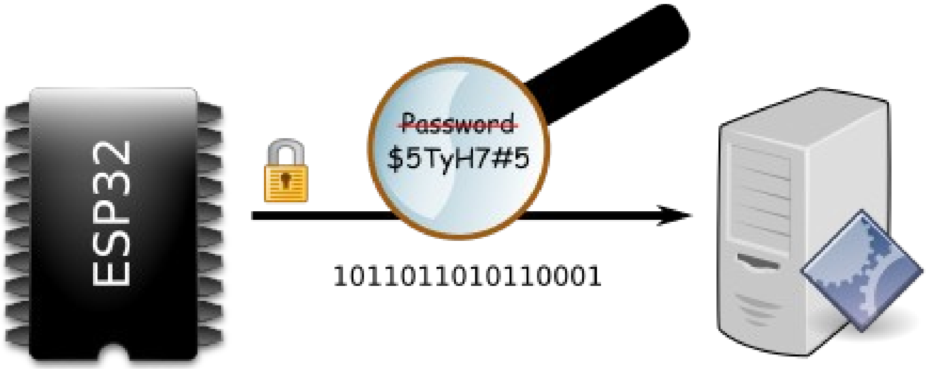
TLS, SSL and security

So far we have been thinking about making sockets calls that form a network connection and then sending and receiving data over that connection. However we have a security problem. The data that flows over the wire is not encrypted. This means that if one were to "sniff" or otherwise examine the network data, we would see the content of the data being transmitted. For example, if I send a password used for authentication, if we were to examine the content of the data, we would be able to determine the password I am using.



It actually isn't that difficult to sense the data being sent and received. Excellent tools such as wireshark are used for debugging and can easily be used to examine the content of the network packets or stream. Obviously we are already exchanging credit card data, email and other sensitive information over the Internet so how is that done?

The answer is a concept called the "Secure Socket Layer" or SSL. SSL provides the capability to encrypt the data before transmission such that only the intended recipient can decrypt it. Conversely, any responses sent by the recipient are also encrypted such that only we can decrypt the data. If someone were to capture or otherwise examine the data being sent over the wire, there is no way for them to get back to the original data content.



The way this works is through the concept of private keys. Imagine I think of a very large random number (and by large I mean VERY large). We call this private number my private key. Now imagine that associated with the private key is a corresponding number (the public key) that can be used to decrypt a message that was encoded using the private key. Now imagine I want to correspond with a partner securely. I send a request (unencrypted) to the partner and ask for his public key. He sends that back and I send to him a copy of MY public key encrypted with his public key. Since only a matching pair of public/private keys can be used to decrypt data, only the desired recipient can decrypt the message at which point he will have a copy of my public key. Now in the future I can send him messages encrypted with my private key and further encrypted with his public key and he will be able to decode them with his copy of his private key and my public key while he can send me encrypted messages encoded with his private key which I can decode with my copy of his public key. By having exchanged public keys, we are now good to continue exchanging data without fear that it will be seen by anyone else.

All of this encryption of data happens outside and above the knowledge of TCP/IP networking. TCP/IP provides the delivery of data but cares nothing about its content. As such, and at a high level, if we wish to exchange secure data, we must perform the encryption and decryption using algorithms and libraries that live outside of the sockets API and use sockets as the transport for transmitting and receiving the encrypted data that is fed into and received from the encryption algorithms.

When using mbed TLS, we need a large stack size. I don't yet know how small we can get away with but I have been using 8000 bytes.

See also:

- mbed TLS
- mbed TLS home page
- mbed TLS tutorial
- mbed TLS API reference

mbedTLS app structure

Let us start to break down the structure of a TLS application that uses the mbedTLS APIs.

First there is the notion of a network context that is initialized by a call to `mbedtls_net_init()`.

```
mbedtls_net_context server_fd;
mbedtls_net_init(&server_fd);
```

There is nothing more to explain here. The data that is initialized is "opaque" to us and the invocation of this function is part of the rules.

next comes the initialization of the SSL context with a call to `mbedtls_ssl_init()`.

```
mbedtls_ssl_context ssl;
mbedtls_ssl_init(&ssl);
```

Again, there is nothing more to explain here. The data is again opaque and this function merely initializes it for us. Calling this function is also part of the rules.

Now we call `mebtls_ssl_config_init()`.

```
mbedtls_ssl_config config;
mbedtls_ssl_config_init(&config);
```

These initializations repeat for other data types including:

```
mbedtls_ctr_drbg_context ctr_drbg;
mbedtls_ctr_drbg_init(&ctr_drbg);

mbedtls_entropy_context entropy;
mbedtls_entropy_init(&entropy);

mbedtls_x509_crt cacert;
mbedtls_x509_crt_init(&cacert);
```

SSL utilizes good random number generators. What is a "good" random number? Since computers are deterministic devices, the generation of a random number is performed through the execution of an algorithm and since algorithms are deterministic, then a sequence of numbers generated by these functions might, in principle, be predictable. A good random number generator is one where the sequence of numbers produced is not at all easily predictable and generates values with no biases towards their values with an equal probability of any number within a range being chosen.

We initialize the random number generator with a call to `mbedtls_ctr_drbg_seed()`.

> Note: We see the phrase "`ctr_drbg`" … that is an acronym for "Counter mode Deterministic Random Byte Generator". It is an industry standard specification/algorithm for generating random numbers.
> http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf

With the setup under our belts, it is now time to start considering SSL based communication. Since we are considering SSL over sockets, if we were not using TLS sockets, we would perform a call to `socket()` to create a socket and then `connect()` to connect to our partner. In the world of mbedtls, we call `mbedtls_net_connect()`. This has the form:

```
mbedtls_net_connect(&server_fd, <hostname>, <port>, MBEDTLS_NET_PROTO_TCP);
```

The hostname and port define where we are connecting to. Notice the first parameter. This is the `mbedtls_net_context` structure that we initialized with a call to `mbedtls_net_init()` previously. We should always check the return code to ensure that the connection was successful.

Now we get to configure our SSL defaults with a call to `mbedtls_ssl_config_defaults()`. For example:

```
mbedtls_ssl_config_defaults(
   &conf,
   MBEDTLS_SSL_IS_CLIENT,
   MBEDTLS_SSL_TRANSPORT_STREAM,
   MBED_SSL_PRESET_DEFAULT)
```

When we are communicating via SSL, we commonly wish to validate that the credentials provided by our partner indicate that they are who they claim to be. This process is called authentication. We can define what kind of authentication we wish to perform by calling `mbedtls_ssl_conf_authmode()`.

```
mbedtls_ssl_conf_authmode(&ssl, MBEDTLS_SSL_VERIFY_NONE)
```

Earlier we said that SSL is heavily dependent on a good random number generator. Now we tell the environment which random number generator we wish to use:

```
mbedtls_ssl_conf_rng(&ssl, mbedtls_ctr_drbg_random, &ctr_drbg)
```

Next we do some more SSL context setup by calling `mbedtls_ssl_set_hostname()`.

```
mbedtls_ssl_set_hostname(&ssl, "name")
```

Now we instruct the SSL environment which functions to use to send and receive data by calling `mbedtls_ssl_set_bio()`.

```
mbedtls_ssl_set_bio(&ssl, &server_fd, mbedtls_net_send, mbedtls_net_recv, NULL)
```

At this point, we have formed a connection to our partner and configured the SSL environment. What remains is to actually read and write data. To write data we call `mbed_ssl_write()`.

```
mbedtls_ssl_write(&ssl, buf, len)
```

and to read data we call `mbedtls_ssl_read()`.

```
mbedtls_ssl_read(&ssl, buf, len)
```

See also:

- mbedtls_net_init
- mbedtls_ssl_init
- mbedtls_ssl_config_init
- mbedtls_net_connect
- mbedtls_ssl_config_defaults
- mbedtls_ssl_conf_authmode
- mbedtls_ssl_conf_rng
- mbedtls_ssl_set_hostname
- mbedtls_ssl_set_bio
- mbedtls_ssl_write
- mbedtls_ssl_read

mbedTLS Example

Here is a sample function that has been tested on an ESP32 to make an HTTPS call to an HTTPS server to retrieve some results.

```
#include "mbedtls/platform.h"

#include "mbedtls/ctr_drbg.h"
#include "mbedtls/debug.h"
#include "mbedtls/entropy.h"
#include "mbedtls/error.h"
#include "mbedtls/net.h"
#include "mbedtls/ssl.h"
#include "esp_log.h"
#include "string.h"
#include "stdio.h"

#define SERVER_NAME "httpbin.org"
#define SERVER_PORT "443"
static char tag[] = "callhttps";

static char errortext[256];

static void my_debug(void *ctx, int level, const char *file, int line, const char *str) {
   ((void) level);
   ((void) ctx);
   printf("%s:%04d: %s", file, line, str);
}

void callhttps() {
   ESP_LOGD(tag, "--> callhttps\n");
   mbedtls_net_context server_fd;
   mbedtls_entropy_context entropy;
   mbedtls_ctr_drbg_context ctr_drbg;
   mbedtls_ssl_context ssl;
   mbedtls_ssl_config conf;
   mbedtls_x509_crt cacert;

   int ret;
   int len;
   char *pers = "ssl_client1";
   unsigned char buf[1024];


   mbedtls_net_init(&server_fd);
   mbedtls_ssl_init(&ssl);
   mbedtls_ssl_config_init(&conf);
   mbedtls_x509_crt_init(&cacert);
   mbedtls_ctr_drbg_init(&ctr_drbg);
   mbedtls_entropy_init(&entropy);
   mbedtls_ssl_conf_dbg(&conf, my_debug, stdout);

   mbedtls_debug_set_threshold(2); // Log at error only
   ret = mbedtls_ctr_drbg_seed(
      &ctr_drbg,
      mbedtls_entropy_func, &entropy, (const unsigned char *) pers, strlen(pers));
   if (ret != 0) {
      ESP_LOGE(tag, " failed\n  ! mbedtls_ctr_drbg_seed returned %d\n", ret);
      return;
   }

   ret = mbedtls_net_connect(&server_fd, SERVER_NAME, SERVER_PORT, MBEDTLS_NET_PROTO_TCP);
   if (ret != 0) {
      ESP_LOGE(tag, " failed\n  ! mbedtls_net_connect returned %d\n\n", ret);
      return;
   }

   ret = mbedtls_ssl_config_defaults(
      &conf,
      MBEDTLS_SSL_IS_CLIENT,
      MBEDTLS_SSL_TRANSPORT_STREAM,
      MBEDTLS_SSL_PRESET_DEFAULT);
   if (ret != 0) {
      ESP_LOGE(tag, " failed\n  ! mbedtls_ssl_config_defaults returned %d\n\n", ret);
      return;
   }

   mbedtls_ssl_conf_authmode(&conf, MBEDTLS_SSL_VERIFY_NONE);

   mbedtls_ssl_conf_rng(&conf, mbedtls_ctr_drbg_random, &ctr_drbg);

   ret = mbedtls_ssl_setup(&ssl, &conf);
   if (ret != 0) {
      mbedtls_strerror(ret, errortext, sizeof(errortext));
      ESP_LOGE(tag, "error from mbedtls_ssl_setup: %d - %x - %s\n", ret, ret, errortext);
      return;
   }

   ret = mbedtls_ssl_set_hostname(&ssl, "httpbin.org");
   if (ret != 0) {
      mbedtls_strerror(ret, errortext, sizeof(errortext));
      ESP_LOGE(tag, "error from mbedtls_ssl_set_hostname: %d - %x - %s\n", ret, ret, errortext);
      return;
   }

   mbedtls_ssl_set_bio(&ssl, &server_fd, mbedtls_net_send, mbedtls_net_recv, NULL);

   char *requestMessage = \
         "GET /ip HTTP/1.1\r\n" \
         "User-Agent: kolban\r\n" \
         "Host: httpbin.org\r\n" \
         "Accept-Language: en-us\r\n" \
         "Accept-Encoding: gzip, deflate\r\n" \
         "\r\n";
   sprintf((char *)buf, requestMessage);
   len = strlen((char *)buf);

   ret = mbedtls_ssl_write(&ssl, buf, len);
   if (ret < 0) {
      mbedtls_strerror(ret, errortext, sizeof(errortext));
      ESP_LOGE(tag, "error from write: %d -%x - %s\n", ret, ret, errortext);
      return;
   }

   len = sizeof(buf);
   ret = mbedtls_ssl_read(&ssl, buf, len);
   if (ret < 0) {
      ESP_LOGE(tag, "error from read: %d\n", len);
      return;
   }

   printf("Result:\n%.*s\n", len, buf);

   mbedtls_net_free(&server_fd);
   mbedtls_ssl_free(&ssl);
   mbedtls_ssl_config_free(&conf);
   mbedtls_ctr_drbg_free(&ctr_drbg);
   mbedtls_entropy_free(&entropy);
   ESP_LOGV(tag, "All done");
}
```

Notes:

When debugging MBED in Curl set MBEDTLS_DEBUG to 1 in curl_config.h

OpenSSL

OpenSSL is a popular implementation of an SSL stack. In the ESP32 environment, the selected stack for SSL/TLS is mbedTLS which is not the same as OpenSSL. As part of the ESP-IDF, a mapping layer has been provided that exposes the OpenSSL API on top of an mbedTLS implementation.

Name Service

On the Internet, server machines can be found by their Domain Name Service (DNS) names. This is the service that resolves a human readable representation of a machine such as "`google.com`" into the necessary IP address value (eg. `216.58.217.206`). In order for this transformation to happen, the ESP32 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 (for example we are using static IP addresses), then we need to instruct the ESP32 of the locations of the DNS servers manually. We can do this using `dns_setserver()` function. This takes an IP address as input along with which of the two possible DNS servers to use. The ESP32 is configured to know the identity of up to two external name servers. The reason for two is that if an attempt to reach the first one fails, we will utilize the second one. We can retrieve our current DNS server identities using `dns_getserver()`.

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

Once we have define the name servers, we can look up the address of a host name using the `gethostbyname()` function.

During development, we may wish to test a specific DNS server to validate that it can resolve a host name. An excellent Linux tool to perform this task is "`nslookup`". It has many options but for our purposes, we can supply it the host name to lookup and the DNS server to use:

```
$ nslookup example.com 8.8.8.8
Server:   8.8.8.8
Address:  8.8.8.8#53

Non-authoritative answer:
Name: example.com
Address: 93.184.216.34
```

See also:

- gethostbyname
- dns_getserver
- dns_setserver
- Wikipedia: Domain Name System
- Google: Public DNS

Multicast Domain Name Systems

On a local area network with dynamic devices coming and going, we may want one device to find the IP address of another device so that they may interact with each other. The problem though is that IP addresses can be dynamically allocated by a DHCP server running on a WiFi access point. This means that the IP address of a device is likely not going to be static. In addition, it is not a great usability story to refer to devices by their IP addresses. What we need is some form of dynamic name service for finding devices by name where their IP addresses aren't administrator configured. This is where the Multicast Domain Name System (mDNS) comes into play.

At a high level, when a device wishes to find another device with a given name, it broadcasts a request to all members of the network asking for a response from the device that has that name. If a machine believes it has that identity, it responds with its own broadcast which includes its name and IP address. Not only does this satisfy the original request, but other machines on the network can see this interaction and cache the response for themselves. This means that should they need to resolve the same host in the future, they already have the answer.

Using the Multicast Domain Name System (mDNS) an ESP32 can attempt to resolve a host name 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.

The name service demons are implemented by Bonjour and nss-mdns (Linux).

Normally, hosts located using this technique belong to a domain ending in ".local".

To determine if your PC is participating in mDNS you can examine whether or not it is listening on UDP port 5353. This is the port used for mDNS communications.

See also:

- Wikipedia – Multicast DNS
- IETF RFC 6762: Multicast DNS
- Multicast DNS
- New DNS Technologies in the Lan
- Avahi – Implementation of mDNS … source project for Unix machines
- Adafruit – Bonjour (Zeroconf) Networking for Windows and Linux
- chrome.mdns – API description for Chrome API for mDNS
- Android – ZeroConf Browser

mDNS API programming

The ESP-IDF provides a set of rich APIs for programming mDNS on the ESP32. Specifically, we can either advertise ourselves in mDNS or else query existing mDNS information.

The attributes of an mDNS server entry appear to be:

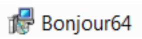
- hostname – mdns_set_hostname()
- default instance – mdns_set_instance()
- service – mdns_service_add()
  - type – _http, _ftp etc etc
  - protocol – _tcp, _udp etc etc
  - port – port number
- instance name for service – mdns_service_instance_set()
- TXT data for service – mdns_service_txt_set()

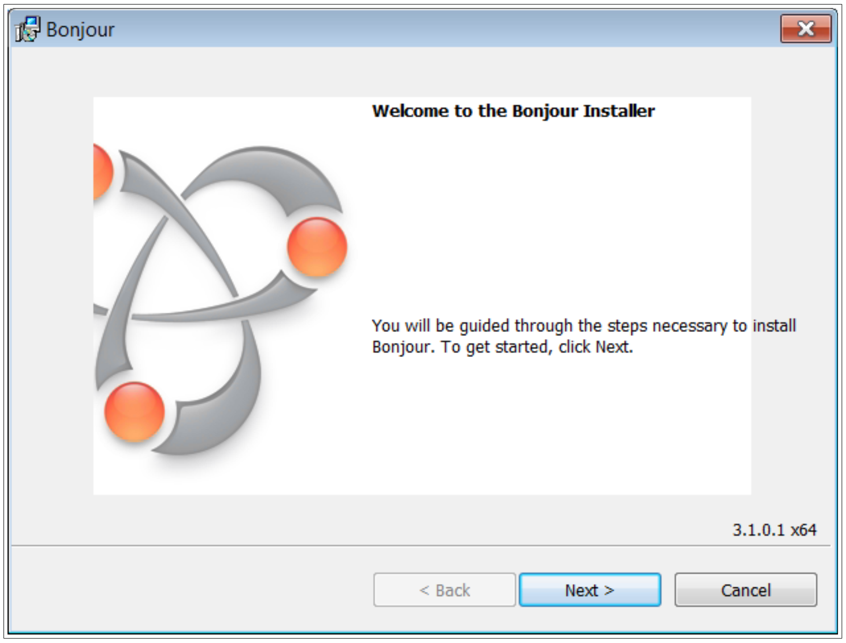
See also:

- mdns_set_hostname
- mdns_set_instance
- mdns_service_add
- mdns_service_instance_set
- mdns_service_port_set

Installing Bonjour

Launch the Bonjour installer:

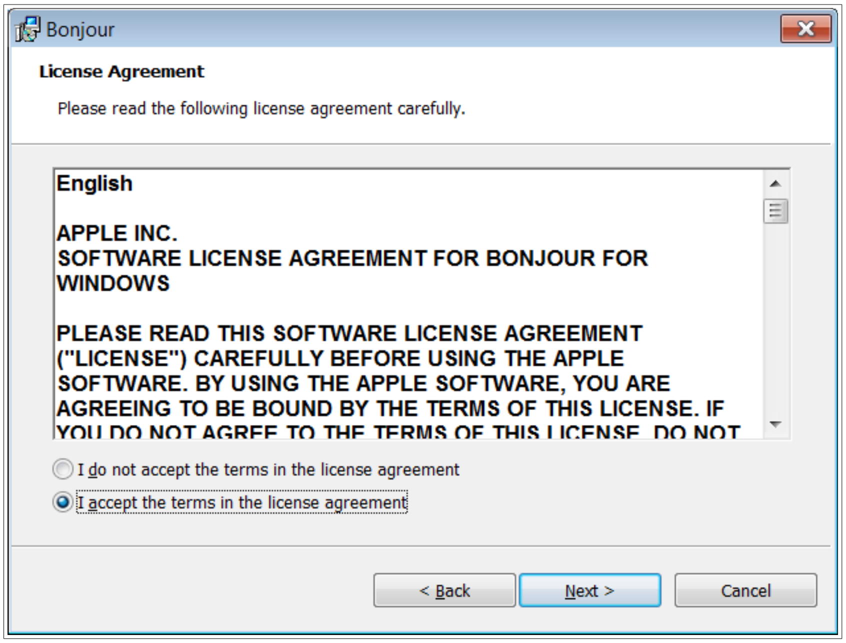




Welcome to the Bonjour Installer

You will be guided through the steps necessary to install Bonjour. To get started, click Next.

3.1.0.1 x64



License Agreement

Please read the following license agreement carefully.

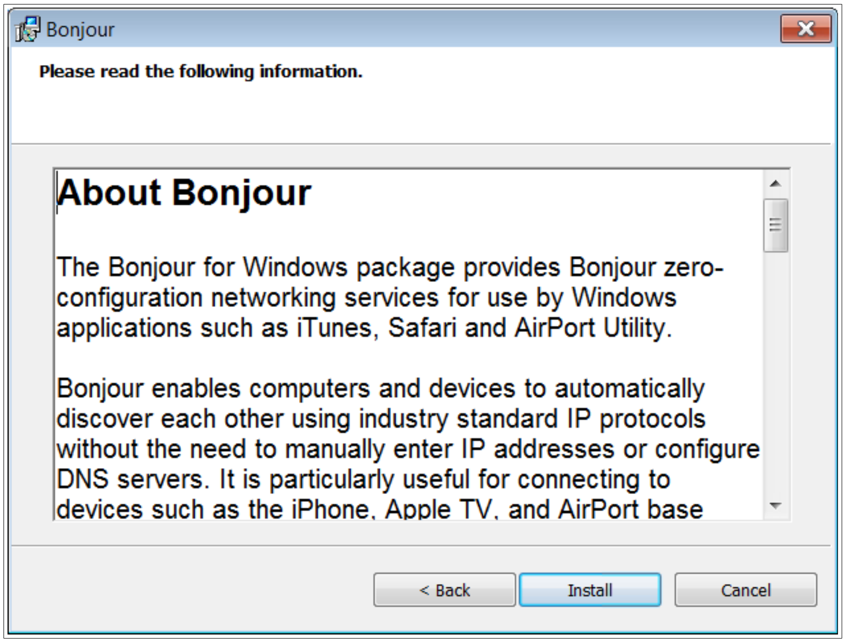
English

APPLE INC.
SOFTWARE LICENSE AGREEMENT FOR BONJOUR FOR WINDOWS

PLEASE READ THIS SOFTWARE LICENSE AGREEMENT ("LICENSE") CAREFULLY BEFORE USING THE APPLE SOFTWARE. BY USING THE APPLE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, DO NOT

○ I do not accept the terms in the license agreement

● I accept the terms in the license agreement

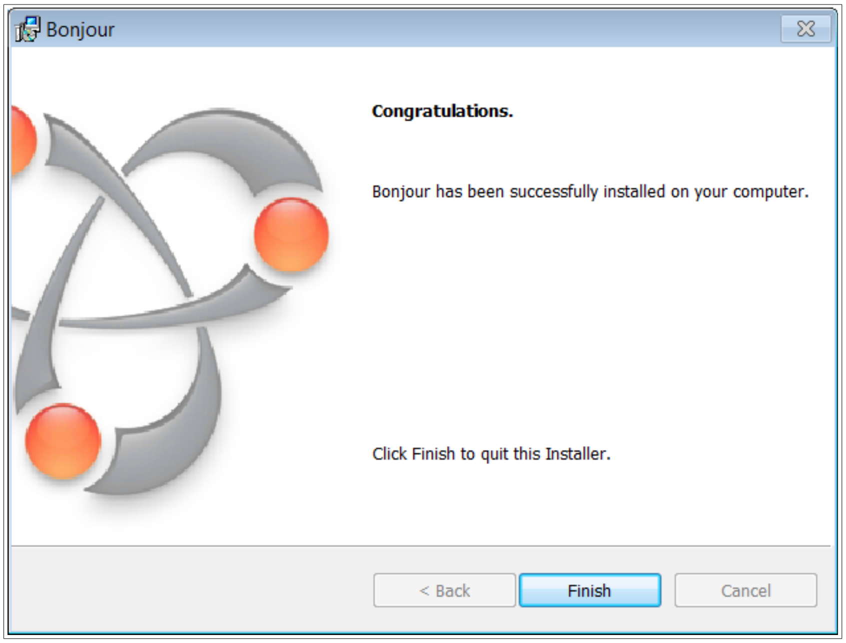


Please read the following information.

About Bonjour

The Bonjour for Windows package provides Bonjour zero-configuration networking services for use by Windows applications such as iTunes, Safari and AirPort Utility.

Bonjour enables computers and devices to automatically discover each other using industry standard IP protocols without the need to manually enter IP addresses or configure DNS servers. It is particularly useful for connecting to devices such as the iPhone, Apple TV, and AirPort base

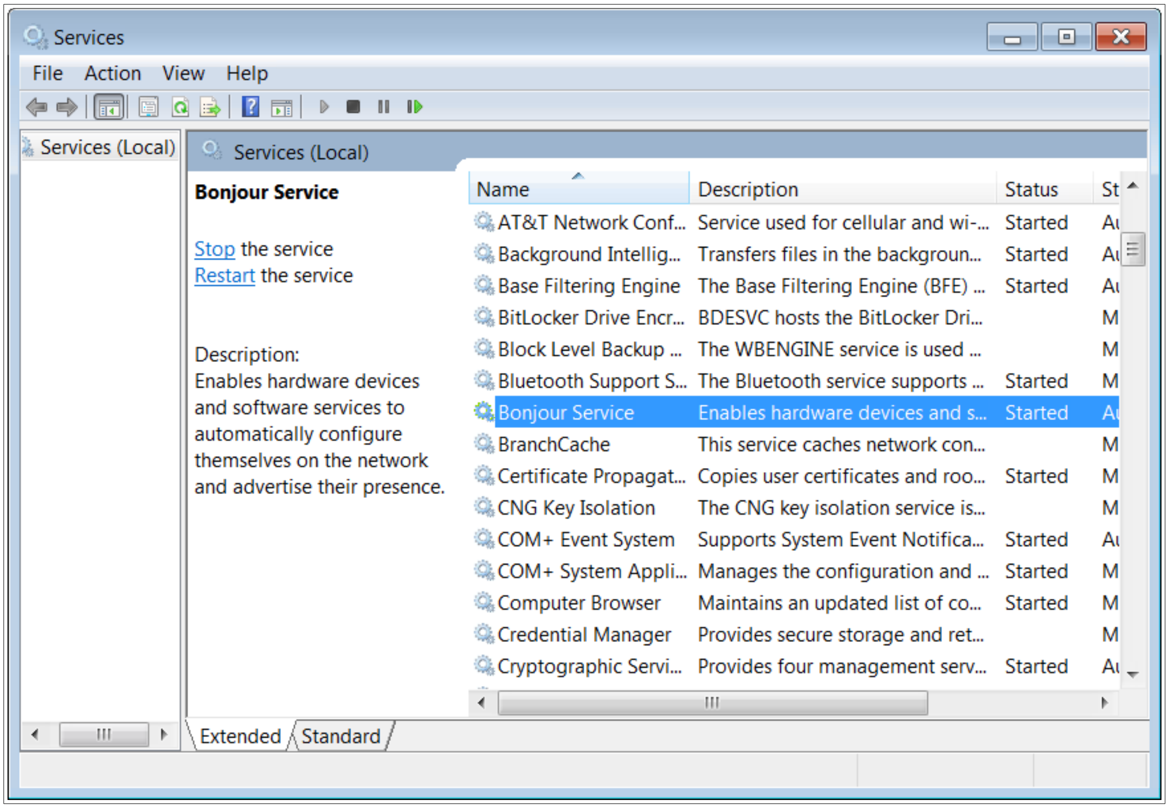


Congratulations.

Bonjour has been successfully installed on your computer.

Click Finish to quit this Installer.

If all has gone well, we will find a new Windows service running called "Bonjour Service":



Bonjour Service

Stop the service
Restart the service

Description:
Enables hardware devices and software services to automatically configure themselves on the network and advertise their presence.

Name	Description	Status	St
AT&T Network Conf...	Service used for cellular and wi-...	Started	Au
Background Intellig...	Transfers files in the backgroun...	Started	Au
Base Filtering Engine	The Base Filtering Engine (BFE) ...	Started	Au
BitLocker Drive Encr...	BDESVC hosts the BitLocker Dri...		M
Block Level Backup ...	The WBENGINE service is used ...		M
Bluetooth Support S...	The Bluetooth service supports ...	Started	M
Bonjour Service	Enables hardware devices and s...	Started	Au
BranchCache	This service caches network con...		M
Certificate Propagat...	Copies user certificates and roo...	Started	M
CNG Key Isolation	The CNG key isolation service is...		M
COM+ Event System	Supports System Event Notifica...	Started	Au
COM+ System Appli...	Manages the configuration and ...	Started	M
Computer Browser	Maintains an updated list of co...	Started	M
Credential Manager	Provides secure storage and ret...		M
Cryptographic Servi...	Provides four management serv...	Started	Au

There is also a Bonjour browser available here …

http://hobbyistsoftware.com/bonjourbrowser

Avahi

An implementation of Multicast DNS on Linux is called Avahi. Avahi runs as the `systemd` daemon called "`avahi-daemon`". We can determine whether or not it is running with:

```
$ systemctl status avahi-daemon
● avahi-daemon.service - Avahi mDNS/DNS-SD Stack
   Loaded: loaded (/lib/systemd/system/avahi-daemon.service; enabled)
   Active: active (running) since Wed 2016-01-20 22:13:35 CST; 1 day 13h ago
 Main PID: 384 (avahi-daemon)
   Status: "avahi-daemon 0.6.31 starting up."
   CGroup: /system.slice/avahi-daemon.service
           ├─384 avahi-daemon: running [raspberrypi.local]
           └─426 avahi-daemon: chroot helper
```

The avahi-daemon utilizes a configuration file found at `/etc/avahi/avahi-daemon.conf`. The default name that avahi advertizes itself as is the local hostname.

When host-name resolution is performed, the system file called `/etc/nsswitch.conf` is used to determine the order of resolution. Specifically the hosts entry contains the name resolution. An example would be:

```
hosts:          files mdns4_minimal [NOTFOUND=return] dns
```

Which says "first look in /etc/hosts, then consult mDNS and then use full DNS". What this means is that a device which advertizes itself with mDNS can be found via a lookup of "<hostname>.local". For example, if I boot up a Linux machine which gets a dynamic IP address through DHCP and the hostname of that machine is "chip1", then I can reach it with a domain name address of "chip1.local". If the IP address of the device changes, subsequent resolutions of the domain name will continue to correctly resolve.

Avahi tools are not installed by default but can be installed using the "avahi-utils" package:

```
$ sudo apt-get install avahi-utils
```

To see the list of mDNS devices in your network, we can use the avahi-browse command. For example:

```
$ avahi-browse -at
+  wlan1 IPv6 chip1 [ce:79:cf:21:db:95]                     Workstation          local
+  wlan1 IPv4 chip1 [ce:79:cf:21:db:95]                     Workstation          local
+  wlan0 IPv6 pizero [00:36:76:21:97:a3]                    Workstation          local
+  wlan0 IPv6 raspi3 [b8:27:eb:9d:fc:60]                    Workstation          local
+  wlan0 IPv6 chip1 [cc:79:cf:21:db:95]                     Workstation          local
+  wlan0 IPv4 pizero [00:36:76:21:97:a3]                    Workstation          local
+  wlan0 IPv4 raspi3 [b8:27:eb:9d:fc:60]                    Workstation          local
+  wlan0 IPv4 chip1 [cc:79:cf:21:db:95]                     Workstation          local
+  wlan0 IPv6 pizero                                        Remote Disk Management 
local
+  wlan0 IPv6 raspi3                                        Remote Disk Management 
local
+  wlan0 IPv4 raspi3                                        Remote Disk Management 
local
+  wlan0 IPv4 pizero                                        Remote Disk Management 
local
+  wlan0 IPv4 WDMyCloud                                     Apple File Sharing   local
+  wlan0 IPv4 WDMyCloud                                     _wd-2go._tcp         local
+  wlan0 IPv4 WDMyCloud                                     Web Site             local
+  wlan0 IPv4 Living Room                                   _googlecast._tcp     local
+  wlan0 IPv4 123456789                                     _teamviewer._tcp     local
```

To access an mDNS advertized server from Microsoft Windows, you will need a service similar to Apple's Bonjour installed. Bonjour is distributed as part of Apple's iTunes product. Once installed, we should be able to access the published servers at their <name>.local address. A partner windows tool called "Bonjour Browser" is available which displays an mDNS listing of servers on windows.

See also:

- avahi home page
- man(1) – avahi-browse

- man(5) – avahi-daemon.conf

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 host name 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 ESP32 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 ESP32 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`". Although this function is present, I would suggest using the POSIX `tzset()` function instead.

With these configured, we can start the SNTP service on the ESP32 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 ESP32 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. While these functions obviously exist, I would not recommend using them. Instead look at the POSIX alternatives which are `time()` and `asctime()`.

Here is an example of using SNTP to set the time:

```
ip_addr_t addr;
sntp_setoperatingmode(SNTP_OPMODE_POLL);
```

```
inet_pton(AF_INET, "129.6.15.28", &addr);
sntp_setserver(0, &addr);
sntp_init();
```

The time can be accessed from a variety of POSIX compliant functions including:

- `asctime` – Build a string representation of time.
- `clock` – Return processor time.
- `ctime` – Build a string representation of time.
- `difftime` – Calculate a time difference.
- `gettimeofday` – Retrieve the current time of day.
- `gmtime` – Produce a struct tm from a time_t.
- `localtime` – Produce a `struct tm` from a `time_t`.
- `settimeofday` – Set the current time.
- `strftime` – Format a `time_t` to a string.
- `time` – Get the current time as a `time_t` (seconds since epoch).

See also:

- SNTP API
- Timers and time
- asctime
- ctime
- gmtime
- localtime
- strftime
- IETF RFC5905: Network Time Protocol Version 4: Protocol and Algorithms Specification

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 communicate with the ESP32. 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 ESP32 can be either a client or a server, both of these Java classes will come into play.

To connect to an ESP32 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 ESP32. 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

Bluetooth

The ESP32 has native Bluetooth support (version 4.2). This means that it can interact with Bluetooth devices such as keyboards, mice and cell phones. Let us review what Bluetooth means for us. Bluetooth is a wireless communication protocol/technology that provides data transfer over a radio signal. Let us assume that ESP32 is one end of the connection and any other Bluetooth device can be at the other. For security purposes, arbitrary Bluetooth devices can't simply be "used" without some explicit authorization. For example, it would be very wrong if I could bring my Bluetooth headset near your phone and start listening to your calls. To achieve security, a process called "pairing" needs to be performed. This achieves a level of trust between the two Bluetooth devices such that they subsequently allow connection without having to be re-paired.

Bluetooth specification

Bluetooth is a specification for wireless communication between multiple electronic devices. At present, there are two primary standards … these are Bluetooth (Classic) and Bluetooth LE. The "LE" stands for Low Energy and is the specification for devices that wish to be powered by batteries and yet have sufficient life span.

In the Bluetooth story, we have devices which are "masters" and devices which are "slaves". A slave can only form a connection and communicate with a master while a master can form concurrent connections with multiple slaves. One slave can not directly communicate with another slave. Instead it must communicate with the master and the master relay the request.

The simplest communication is one master and one slave but if we have multiple slaves connected to the same master, the resulting "network" is termed a "piconet".

Each device that participates in the conversation has a unique address that is a 48 bit value commonly written as 12 hex values (6 bytes). This address is known as the "Bluetooth Device Address" and may be seen in other documentation abbreviated to "`BD_ADDR`".

The encoding of a Bluetooth address is that the first 24 bits encode the organization responsible for allocating the remaining address and the remaining 24 bits are the address itself. However the full 48 bits are the complete identity of the device.

As well as having a unique address, each device can have a symbolic name to help us meaningfully identify it. This is termed the *display name*. The display name is only a mapping to a Bluetooth address and it is really the `BD_ADDR` address that is used to distinguish one device from another. It is also important to note that there are no uniqueness constraints on device names. Multiple devices may select the same device name.

Let us assume that initially, we have two devices and neither of them know about the other. What must now happen is a discovery process. One of them will broadcast an "inquiry" request. Devices receiving the inquiry can respond with their own existence by transmitting their own address and possibly additional information. It is common that the response to an inquiry does not contain the display name of the responding device. If the display name is needed, the inquirer can transmit a directed request now that it knows the address of the responding device which will solicit the name as a further response.

A device does not have to respond to an inquiry request. It has a property setting called an "inquiry scan" that controls whether or not it responds to such. If the attribute is on, then it will respond to an inquiry request and if off, then it will not respond. Think of the phrase "inquiry scan" as the device's choice as to whether it performs the action of "scanning for inquires".

Once the two devices know each others addresses, a connection can be formed between them through a process known as "paging". Again, a device does not have to service a received connection request. It has a property setting called a "paging scan" which controls this. If on, then a paging request causes a new connection to be formed. If off, it will not accept a new connection request.

Once a connection is formed between the devices, that connection can be maintained in a variety of states, the most common being active. However other states are available and are used to save power when there is no active communication of data anticipated.

In order to permit devices to communicate with each other, there has to be an element of security involved. We usually don't want arbitrary devices to be able to connect with each other and share arbitrary information. To achieve this, a process called "bonding" is enacted. Bonding is achieved through the notion of "pairing". In pairing, the devices exchange their addresses, names and other data and generate keys that they share with each other. Pairing typically requires an explicit interaction from the user to permit the pairing to succeed. The user interaction can be as simple as "I approve this device" with a button click or it can be richer with the entry of a pass-code to authenticate and prove that one is who one claims to be.

The Bluetooth protocol provides support for different classes of power. This translates directly into the signal strength of the radio. Remember that the more power used by the radio, the heavier the drain on the power source. If the power source is batteries, the more power used to transmit data, the shorter the life of the batteries.

At the lower levels, Bluetooth takes care of exchanging data between partners. However, there is much more to Bluetooth than just simple data exchange. In order to provide interoperability between devices built by multiple manufacturers, higher level protocols called "profiles" have been defined. These profiles define "what" is transmitted over Bluetooth for a given device function.

Some of the profiles we will come across include:

- `HSP` – Head Set Profile (eg. a Bluetooth ear piece).
- `HFP` – Hands Free Profile (eg. Bluetooth communication in a car).
- `HID` – Human Interface Device (eg. a keyboard or mouse).
- `SPP` – Serial Port Profile.
- `A2DP` – Advanced Audio Distribution profile (eg. connection to Bluetooth speakers).
- `AVRCP` – Audio Visual Remote Control Profile.

Knowing that these protocols exist, it is not sufficient that two Bluetooth devices are in range of each other, they must also both support the same profile that is desired to be used.

At a higher level than the connection are the transport protocols available to us. These include:

- `RFCOMM` – Radio Frequency Communications Protocol. This protocol provides a reliable stream oriented transmission. Loosely, you can compare it to TCP.
- `L2CAP` – Logical Link Control and Adaption Protocol. This is a packet oriented protocol. RFCOMM builds on L2CAP.
- `ACL` – Asynchronous Connection oriented Logical protocol. L2CAP builds on ACL.
- `SCO` – Voice quality audio protocol.

To allow multiple conversations to be processed in parallel, the concept of the "port" is introduced. This is similar to a TCP/IP port number. L2CAP ports can be odd numbered values between 1 and 32767. For RFCOMM, the port numbers are called channels and are between 1 and 30. In the Bluetooth documentation, ports are referred to as "Protocol Service Multiplexers" or "PSMs".

Certain port numbers used by L2CAP are designated as reserved for well defined purposes.

See also:

- Introduction to Bluetooth Low Energy

Bluetooth UUIDs

A UUID is a 16 byte number (128 bits). They are commonly written in hexadecimal with 1 byte corresponding to 2 hex digits. The most common written format is 4-2-2-2-6 (bytes).

XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

When one receives a UUID, it is the value of the UUID that is used to describe the nature of the service or data. Each different type of service will have its own unique UUID.

To reduce transmission overhead, the specification defines some well described services. If one implements one of those services, then we don't need to transmit the whole 128 bit UUID but can get away with less. A special UUID of the form:

XXXXXXXX-0000-1000-8000-00805F9B34FB

is available where only the first 4 bytes (32 bits) are needed to identify the type of service and

0000XXXX-0000-1000-8000-00805F9B34FB

is available where only the first 2 bytes (16 bits) are needed to identity the type of service.

We can generate a UUID using the `uuidgen` command.

Bluetooth GAP

The Generic Access Protocol (GAP). It is GAP that determine which devices can interact with each other. In the GAP story there are primarily two classes of "things". There is the central device and the peripheral device.

We can send advertising using the Advertising Data payload or the Scan Response payload.

A peripheral device will constantly transmit its advertising payload which can contain up to 31 bytes of data. The peripheral transmits its advertising data every advertising interval period. The Advertising Data payload is present in all BT devices. If the central receiver of the advertising data payload wishes, it can request a scan response and the peripheral will send back a scan response payload.

When a peripheral is transmitting advertising data, we can think of it as being in the mode of broadcasting. It is transmitting its data which may or may not be seen by a corresponding central.

The concept of BLE advertising is powerful beyond just finding devices to form subsequent connections. The advertising packets can contain data in their payload. If the data being transmitted does not need to be secure (for example, the outside temperature), then we have the core of an interesting solution in its own right. The peripheral can simply broadcast its packets of data and the central can receive them without the need to form a connection. The receiver can then examine the payload and receive the data, as long as the data is small enough. In this story, the peripheral is performing the role of a "broadcaster" while the central would be performing the role of an "observer". When working in this mode, do realize that the data is flowing in only one direction … from the publisher to the observer. If you need to send response back, you will need to form a connection.

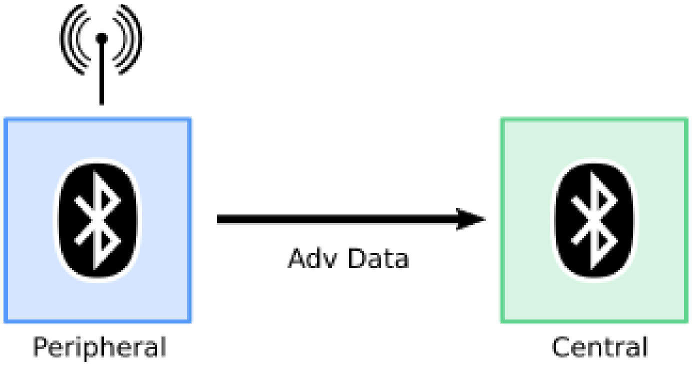
The rate of advertising can be set to be a period between 20ms and 10.24seconds in steps of 0.625ms. Thus we can transmit our advertising packets frequently or slowly.

See also:

- A BLE Advertising Primer

GAP Advertizing data

There is theory and there is practice and studying BLE gives us the opportunity for both. Let us focus on the notion of a BLE peripheral broadcasting advertising messages. At a high level it will look like:



The advertised data payload has a maximum size of 31 bytes. Each payload is composed of one or more data structures where the format of each structure is:

Length	Advertising Data Type	Data ...

The Length is 1 byte in size and indicates how many bytes, following the length byte, this record will be. The number of records in a payload is variable but, of course, the total amount of data has to be 31 bytes or less. Either a length of 0 or an ignorable structure type can be used.

Here is an example of a real payload that I received:

```
020105020A000319C1030302E0FF11094D4C452D3135202020202020202020
```

Now we can parse this as follows:

`02 01 05` – advertising type 0x01

`02 0a 00` – advertising type 0x0A

`03 19 c1 03` – advertising type 0x19

`03 02 e0 ff` – advertising type 0x02

`11 09 4d 4c 45 2d 31 35 20 20 20 20 20 20 20 20 20` – advertising type 09

Here is a simple routine that will step through the structures …

```
while(!finished) {
   length = *payload;
   payload++;
   if (length != 0) {
      ad_type = *payload;
      payload += length;
      ESP_LOGD(tag, "Type: 0x%.2x, length: %d", ad_type, length);
   }
   sizeConsumed += 1 + length;
   if (sizeConsumed >=31 || length == 0) {
      finished = 1;
   }
} // !finished
```

Now that we know how to see the structures and access them, the obvious question is what do each of the structures "mean". For most, they are architected in the BLE specification. Each structure (after the length) has a single byte that is the "advertising data type". These single byte number codes are described here:

https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile

In our ESP32 environment, we have constant definitions for each:

- `ESP_BLE_AD_TYPE_FLAG (0x01)` – The advert contains a byte of flags that are defined as following:
  - Bit 0 – LE Limited Discoverable Mode
  - Bit 1 – LE General Discoverable Mode
  - Bit 2 – BR/EDR is NOT supported.
  - Bit 3 – Indicates whether LE and BR/EDR Controller operates simultaneously
  - Bit 4 – Indicates whether LE and BR/EDR Host operates simultaneously
  - Bits 5-7 – Reserved.
- `ESP_BLE_AD_TYPE_16SRV_PART (0x02)` – Incomplete list of 16bit service UUIDs.
- `ESP_BLE_AD_TYPE_16SRV_CMPL (0x03)` – Complete list of 16 bit service UUIDs.
- `ESP_BLE_AD_TYPE_32SRV_PART (0x04)` – Incomplete list of 32bit service UUIDs.
- `ESP_BLE_AD_TYPE_32SRV_CMPL (0x05)` – Complete list of 32bit service UUIDs.
- `ESP_BLE_AD_TYPE_128SRV_PART (0x06)` – Incomplete list of 128bit service class UUIDs.
- `ESP_BLE_AD_TYPE_128SRV_CMPL (x07)` – Complete list of 128bit service class UUIDs.
- `ESP_BLE_AD_TYPE_NAME_SHORT (0x08)` – Shortened local name.

- ESP_BLE_AD_TYPE_NAME_CMPL (0x09) – Complete local name.
- ESP_BLE_AD_TYPE_TX_PWR (0x0A) – Transmit power level.
- ESP_BLE_AD_TYPE_DEV_CLASS (0x0D)
- ESP_BLE_AD_TYPE_SM_TK (0x10)
- ESP_BLE_AD_TYPE_SM_OOB_FLAG (0x11)
- ESP_BLE_AD_TYPE_INT_RANGE (0x12)
- ESP_BLE_AD_TYPE_SOL_SRV_UUID (0x14)
- ESP_BLE_AD_TYPE_128SOL_SRV_UUID (0x15)
- ESP_BLE_AD_TYPE_SERVICE_DATA (0x16)
- ESP_BLE_AD_TYPE_PUBLIC_TARGET (0x17)
- ESP_BLE_AD_TYPE_RANDOM_TARGET (0x18)
- ESP_BLE_AD_TYPE_APPEARANCE (0x19) – It is likely this conforms to the assigned numbers found here https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.appearance.xml and
- ESP_BLE_AD_TYPE_ADV_INT (0x1A)
- ESP_BLE_AD_TYPE_32SOL_SRV_UUID (0x1B)
- ESP_BLE_AD_TYPE_32SERVICE_DATA (0x1C)
- ESP_BLE_AD_TYPE_128SERVICE_DATA (0x1D)
- ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE (0xFF) – Custom payload.

With this in mind, if we look at our sample data … we can see that it means:

02 01 05 – advertising type 0x01 – ESP_BLE_AD_TYPE_FLAG

02 0a 00 – advertising type 0x0A – ESP_BLE_AD_TYPE_TX_PWR

03 19 c1 03 – advertising type 0x19 – ESP_BLE_AD_TYPE_APPEARANCE

03 02 e0 ff – advertising type 0x02 – ESP_BLE_AD_TYPE_16SRV_PART

11 09 4d 4c 45 2d 31 35 20 20 20 20 20 20 20 20 20 – advertising type 0x09 – ESP_BLE_AD_TYPE_NAME_CMPL

Now we can consult the specification and determine what the data part of each advertising type means and we have our information.

Because examining the advertising data of a GAP message is so important and so common, the ESP-IDF contains a very useful utility function that takes as an input the advertisement data and looks for an advertising type structure within it of a defined type. The function is called `esp_ble_resolv_adv_data()`.

See also:

- esp_ble_resolve_adv_data

Advertisability – limited and general

Is "advertisability" even a real word? I think not … but it describes what we want to cover. When we have a bluetooth peripheral, we somehow need to make it available to be discovered by a bluetooth central in order for the central to make requests of the peripheral. The way we achieve this is by making the peripheral visible by it broadcasting its advertising packets. When a peripheral starts advertising, it can tag the advertisement as either limited-discoverable or general-discoverable. The notion here is that limited-discoverable devices have "recently" been enabled to start broadcasting while "general-discoverable" devices usually broadcast continuously. "So what" you may ask? Well, think of a user looking for a device. It is not uncommon for a new device to be added by pressing some button on it to start it broadcasting so that it can be found and paired. We want this new device to show up higher in the list of found devices than others … and this can be achieved by recognizing that a device which only broadcasts for a limited period of time will enable its "limited-discoverable" flag and provide a hint to the software running on the central that the device is likely going to be the one the user is looking for.

Filtering devices

When we are looking for a device, we may find more devices than we want. Rather than present all devices to the user, we can choose to filter the set of available devices and hide the ones that we believe are not of interest. For example, if I am running a heart-rate monitor application on my phone, I'm really only interested in devices that can provide such information. My latest bluetooth headset or the outside temperature is not what I'm looking for. What we want to do is filter the set of ALL found devices down to only the ones that match some criteria of interest to us. In order for that to happen, each device must not only advertise its existence, but also supply information that can be used to include or eliminate it from selection.

Performing a scan

There isn't much value in generating advertising data if no-one is listening. In BLE the act of listening for advertising data is called "scanning". To perform a scan in the ESP32 we perform the following tasks:

1. We register a callback function to handle the received data.
2. We define the parameters for how we would like the scan to be performed.
3. We ask the ESP32 to start scanning.

Translating these into ESP32 APIs we have `esp_ble_gap_register_callback()` to register our GAP event callback function to be invoked when events arrive.

We have the `esp_ble_gap_set_scan_params()` to setup how we wish the scanning to be performed.

We have `esp_ble_gap_start_scanning()` to initiate a scan request.

If we need to interrupt our scanning before the requested duration of scanning has completed, we can call `esp_ble_gap_stop_scanning()`.

The function registered when we call `esp_ble_gap_register_callback()` is where the majority of our logic will happen. The parameter to this registration function is itself a C function which the following signature:

```
void gap_event_handler(
   esp_gap_ble_cb_event_t event,
   esp_ble_gap_cb_param_t *param)
```

The `param` is a union of structures.

Event Type	Data Property
ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT	adv_data_cmpl
ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT	adv_data_raw_cmpl
ESP_GAP_BLE_ADV_START_COMPLETE_EVT	adv_start_cmpl
ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT	scan_param_cmpl
ESP_GAP_BLE_SCAN_RESULT_EVT	scan_rst
ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT	scan_rsp_data_raw_cmpl
ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT	scan_rsp_data_cmpl
ESP_GAP_BLE_SCAN_START_COMPLETE_EVT	scan_start_cmpl

The `event` parameter defines the type of event we have received. Event types include:

- `ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT` – Raw advertising data operation complete.
- `ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT` – Called when advertising data set is complete. Structure parameter is called `scan_rsp_data_cmpl`.

    - `esp_bt_status_t status` – The status of the event.
- `ESP_GAP_BLE_ADV_START_COMPLETE_EVT` – Called when advertising scan startup is complete. The parameter is a property called `scan_start_cmpl` which contains:
    - `esp_bt_status_t status` – The status of the event.
- `ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT` – Called when scan parameters set complete. Structure parameter is called `scan_param_cmpl`.
    - esp_bt_status_t status
- `ESP_GAP_BLE_SCAN_RESULT_EVT` – The `param` is an instance of `esp_ble_gap_cb_param_t`. Called when one scan result is ready. The structure parameter is called `scan_rst`.
    - `esp_gap_search_evt_t search_evt` – Choices are:
        - `ESP_GAP_SEARCH_INQ_RES_EVT` – We have received a search result.
        - `ESP_GAP_SEARCH_INQ_CMPL_EVT` – The search is complete.
        - ESP_GAP_SEARCH_DISC_RES_EVT
        - ESP_GAP_SEARCH_DISC_BLE_RES_EVT
        - ESP_GAP_SEARCH_DISC_CMPL_EVT
        - ESP_GAP_SEARCH_DI_DISC_CMPL_EVT
        - ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT
    - `esp_bd_addr_t bda` – The address of the device. 6 bytes of data.
    - `esp_bt_dev_type_t dev_type` – One of:
        - ESP_BT_DEVICE_TYPE_BREDR
        - ESP_BT_DEVICE_TYPE_BLE
        - ESP_BT_DEVICE_TYPE_DUMO
    - `esp_ble_addr_type_t ble_addr_type` – One of
        - BLE_ADDR_TYPE_PUBLIC
        - BLE_ADDR_TYPE_RANDOM
        - BLE_ADDR_TYPE_RPA_PUBLIC
        - BLE_ADDR_TYPE_RPA_RANDOM
    - `esp_ble_evt_type_t ble_evt_type` – One of
        - ESP_BLE_EVT_CONN_ADV

        - ESP_BLE_EVT_CONN_DIR_ADV
        - ESP_BLE_EVT_DISC_ADV
        - ESP_BLE_EVT_NON_CONN_ADV
        - ESP_BLE_EVT_SCAN_RSP
    - `int rssi` – The signal strength.
    - `uint8_t ble_adv[ESP_BLE_ADV_DATA_LEN_MAX]` – The advertised data.
    - `int flag` – Flags
        - bit 0 – Limited Discoverable
        - bit 1 – General Discoverable
        - bit 2 – BR/EDR not supported
        - bit 3 – Simultaneous LE and BR/EDR (Controller)
        - bit 4 – Simultaneous LE and BR/EDR (Host)
    - `int num_resps` – The number of responses received. This is valid when the `search_evt` type is `ESP_GAP_SEARCH_INQ_CMPL_EVT`.
    - uint8_t adv_data_len
    - uint8_t scan_rsp_len
- ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT – ???.
- `ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT` – Called when the scan response data set is complete. Structure parameter is called: `scan_rsp_data_cmpl`.
    - esp_bt_status_t status
- ESP_GAP_BLE_SCAN_START_COMPLETE_EVT – ???.

A typical series of events received might be:

- `ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT`
- `ESP_GAP_BLE_SCAN_START_COMPLETE_EVT`
- `ESP_GAP_BLE_SCAN_RESULT_EVT – ESP_GAP_SEARCH_INQ_RES_EVT`
- …
- `ESP_GAP_BLE_SCAN_RESULT_EVT – ESP_GAP_SEARCH_INQ_CMPL_EVT`

See also:

- esp_ble_gap_register_callback
- esp_ble_gap_set_scan_params
- esp_ble_gap_start_scanning
- esp_ble_gap_stop_scanning

Performing advertising

If our ESP32 is going to be a peripheral, then it will be advertising its existence. The ESP-IDF provides some APIs to achieve that task. At a high level:

1. Call `esp_ble_gap_config_adv_data()` to specify the content of our periodic advertisement.
2. Call `esp_ble_gap_start_advertising()` to initiate the periodic advertisement.

While superficially simple, we need to consider all the distinct parameters that are available to us … and there are many.

We will start with the `esp_ble_gap_config_adv_data()`. This is where we specify the content of the advertisement payload. An example structure would be:

```
static esp_ble_adv_data_t test_adv_data;
test_adv_data.set_scan_rsp        = false,
test_adv_data.include_name        = true,
test_adv_data.include_txpower     = true,
test_adv_data.min_interval        = 0x20,
test_adv_data.max_interval        = 0x40,
test_adv_data.appearance          = 0x00,
test_adv_data.manufacturer_len    = 0,
test_adv_data.p_manufacturer_data = NULL,
test_adv_data.service_data_len    = 0,
test_adv_data.p_service_data      = NULL,
test_adv_data.service_uuid_len    = 32,
test_adv_data.p_service_uuid      = test_service_uuid128,
test_adv_data.flag                = 0x2,
};
```

Once we have started advertising, we can check the published information using:

```
$ sudo hcitool lescan
```

See also:

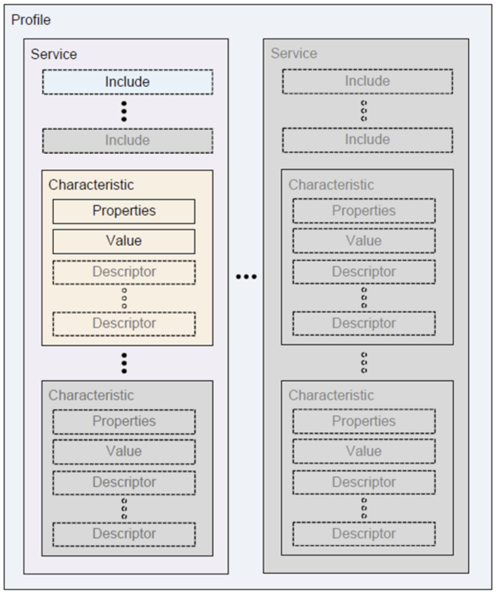
- esp_ble_gap_config_adv_data
- esp_ble_gap_start_advertising
- esp_ble_gap_stop_advertising

Bluetooth GATT

The Generic Attribute Protocol (strangely called GATT) provides a mechanism for passing data in a standard form. GATT is always present in BLE. Think of GATT as a way to send and receive data that is "remembered" at the GATT server (while it is running). A client can explicitly request the values of data items as well as receive pushed asynchronous notifications as events when something interesting happens at the GATT server.

In this model of an attribute, each attribute consists of a row in the table. The "handle" property of the attribute forms its key. It is important to note that a handle is not the same as a row number. Handles need not be consecutive. However, no two attributes can have the same handle value. How the server internally manages the attributes is not part of the specification and is left to the BLE run-time designers.

At the high level of the protocol there is the concept of a service. The service is the grouping of functionally related attributes. Each service has its own UUID. Within a service are characteristics where a characteristic is a property of that service. Each characteristic type is identified by its own UUID value. In addition a characteristic contains a value, properties, security and may also include descriptors.



Within the specification of GATT we have the notion of two roles … namely that of a client and that of a server. A GATT server is commonly a passive entity that waits for requests from GATT clients and services them as they arrive.

When a GATT client starts, it assumes little about the GATT server and first performs an inquiry upon the server to determine its characteristics.

Every BLE device **must** include the capability to be a GATT server.

Before we talk more about GATT, let us first talk about ATT. ATT is the Attribute Protocol and serves as the underpinnings for GATT. Think of a logical server that maintains a set of attributes. We define an attribute as consisting of:

- `handle` – A 16 bit number representing the key/identity of the attribute.
- `type` – 128 bit UUID describing the type of the attribute.
- `permissions` – Permissions on the attribute. The permissions can be:
    - None
    - Read
    - Write
    - Read/Write
- `value` – The value of the attribute. Can be up to 512 bytes.

Loosely, we can think of the attributes as being in a table form:

handle1	type1	permissions1	value1
handle2	type2	permissions2	value2
handle3	type3	permissions3	value3
handle...	type...	permissions...	value...

The GATT server manages attributes that can be read by or written from a GATT client. How these attributes are stored by the GATT server internally is not defined in the specification and left to the implementers to choose.

This story of raw attributes is further specified in the GATT protocol by the notion of services and characteristics.

Think of a service a logical description that a GATT server is prepared to do something that is well defined. If it claims to provide a service, then it must adhere to the contract described by that service. This contract is defined by a set of characteristics and the set of those make up the service.

See also:

- Generic Attribute Profile (GATT)
- generic attributes (gatt) and the generic attribute profile
- GATT-Based specifications

- GATT services – Assigned numbers
- GATT delarations – Assigned number
- GATT characteristics – Assigned numbers
- gatttool

GATT Characteristic

Working on the notion that in the underlying story everything is an attribute, let us explain the concept of characteristics in terms of attributes.

Each characteristic is declared as:

<handle>	UUID (0x2803)	Read only	• properties – 1 byte • value handle – 2 bytes • characteristic UUID – 2, 4 or 16 bytes

The properties are a bit field including:

- `broadcast` – Characteristic value can be in advertising packets.
- `read` – Client may read the characteristic value.
- `write` – Client may write the characteristic value and send a response.
- `write without response` – Client may write the characteristic value with no response.
- `notify` – Value notification available.
- `indicate` – Value indication available.
- `signed write command` – ???

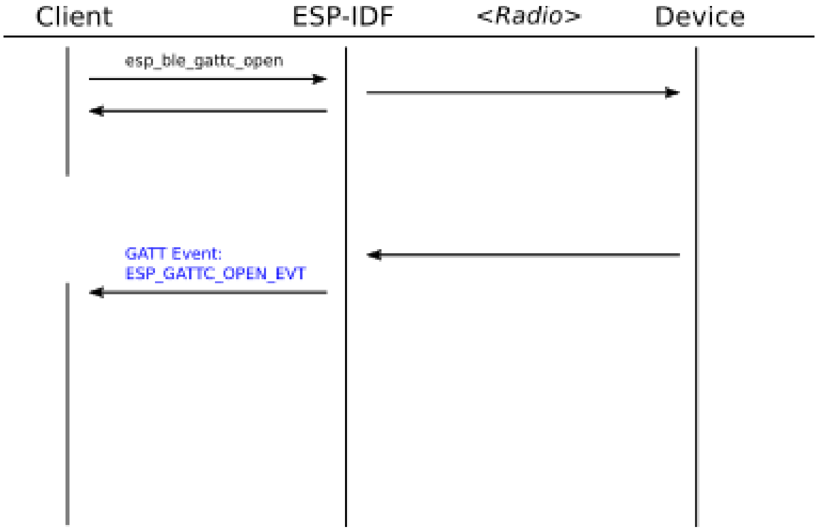
Being a GATT client

From the ESP32 perspective, to be a GATT client we perform:

1. Register a callback to receive GATT events using `esp_ble_gattc_register_callback()`.
2. Call `esp_ble_gattc_app_register()` to register this application.
3. Open a connection to the GATT server using `esp_ble_gattc_open()`.
4. When an open event arrives, execute a search using `esp_ble_gattc_search_service()`.

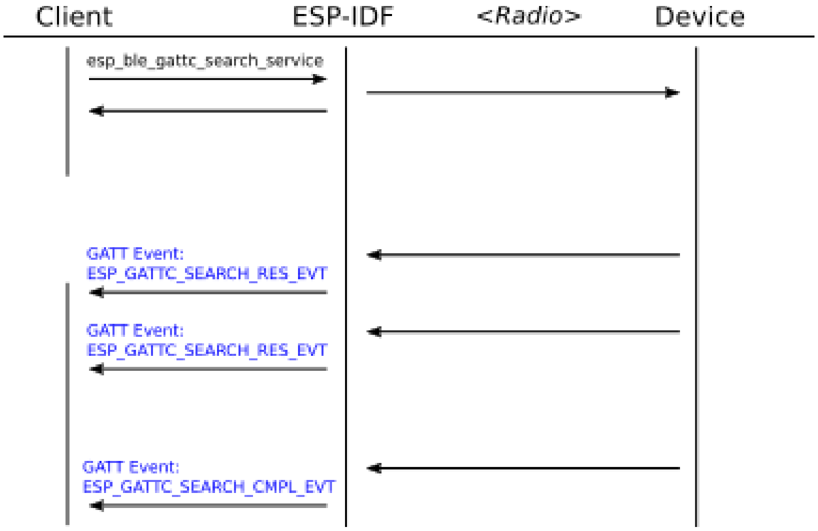
When we call `esp_ble_gattc_open()` we are requesting to open a GATT connection to a specific device. This request is non-blocking and will execute in the background.

Eventually we will receive a GATT event indicating the outcome. The event type will be `ESP_GATTC_OPEN_EVT`.



As part of the response data from the `ESP_GATTC_OPEN_EVT` we will receive a connection identifier (`conn_id`). This `conn_id` can be loosely thought of as a socket to the partner device.

Once we have formed a connection to the partner device, we can ask it about the services that it offers. We do this by calling `esp_ble_gattc_search_service()` function. Like other BLE mechanisms, this is an asynchronous operation that results in a series of GATT events being generated. For each service possessed by the device, an `ESP_GATTC_SEARCH_RES_EVT` is fired and finally, when we have seen all the services, we get an `ESP_GATTC_SEARCH_CMPL_EVT`. This is illustrated in the following diagram:



For each of the services we get back we can start to invoke those services on the device.

See also:

- esp_ble_gattc_register_callback
- esp_ble_gattc_app_register
- esp_ble_gattc_open
- esp_ble_gattc_search_service
- GATT Services

Being a GATT Server

The high level of being a GATT server is:

```
esp_bt_controller_init()
esp_bt_controller_enable()
esp_bluedroid_init()
esp_bluedroid_enable()
esp_ble_gatts_register_callback()
esp_ble_gap_register_callback()
esp_ble_gatts_app_register()
```

The call to `esp_ble_gatts_app_register()` registers our application. This passes control back to the BLE subsystem and, when ready, will call back the GATT server event handler with an event type of `ESP_GATTS_REG_EVT`. When we receive that, we then do the next part of the setup:

```
esp_ble_gap_set_device_name()
esp_ble_gap_config_adv_data()
esp_ble_gatts_create_service()
```

To test that all is working, we can run the hcitool from a Linux system:

```
$ sudo hcitool lescan
24:0A:C4:00:00:96 MYDEVICE
```

See also:

- esp_ble_gatts_register_callback
- esp_ble_gap_register_callback
- esp_ble_gap_set_device_name
- esp_ble_gap_config_adv_data
- esp_ble_gatts_app_register
- esp_ble_gatts_create_service

Notifications and indications

We might not want a BLE client to continuously poll a BLE server to read a characteristic value to determine if and when it changes. There are two primary downsides to this. The first is that if we assume that in the majority of cases there will have been no change, we are wasting energy asking for a value, waking up the server and being told it hasn't changed. Instead what we want is an interrupt mechanism. This is where notifications and indications come into play. When a client connects to the server, the server can *push* notifications that a value has changed to the client. This way the client doesn't have to do any work except when it is informed that the value changed. No radio transmissions are occurring until the value changes.

This takes us to the second down-side of polling. If we are polling, we are presumably waiting between polls. In that case we have a latency between when a value does change and when the peer is notified. This latency is reduced in the notification story.

A notification is a "fire and forget" story. The server that is informed that its value changes notifies its peer without ever knowing that the peer received (or did not receive) the notification. Similar to notifications is the concept of an indication. Like a notification, nothing is transmitted to the peer until the value is flagged as having changed … however, unlike notification, an indication will result in an acknowledgment that the partner received the update.

Associated with polling and notification is the Client Characteristic Descriptor with UUID 0x2902. This contains a couple of flags. One for notification and one for indication. A server which is able to generate notifications or indications for a given characteristic should have this descriptor associated with it. Before actually performing a notification or indication, the server should check the bits in the flags before performing the operation. The client will update the bits if it desires to have notifications or indications performed. It is presumed that this protocol is in place to prevent a server expending radio transmission energy when a notification would be produced but a client isn't interested in receiving at this time.

GATT XML descriptions

The GATT service, characteristic, descriptors and declaration descriptions are available as XML documents.

See also:

- GATT XML

Service Discovery Protocol

When a client application wishes to request that a connection be established, the client needs to know the port number on which the server is listening. In TCP/IP land, this is achieved by having the client and the server share the implicit knowledge of the port number to use. In Bluetooth, extra functions have been made available. Specifically, there is a service available called the Service Discovery Protocol or "SDP". At the server, when a service is offered, the port number of that service is registered to the local SDP. When a client now wishes to use the target service, it first requests endpoint information from the SDP running on the server. The SDP returns the endpoint information and the client now has all the information it needs to create a direct connection to the desired target service. The unit of information managed by the SDP server is called a "service record" or "SDP record".

A command line interface tool called `sdptool` is available to examine a Bluetooth device's SDP data. A simple command is:

```
$ sdptool browse <Bluetooth Address>
```

This returns a series of records of the form:

- Service Name
- Service Description
- Service Provider
- Service RecHandler
- Service Class ID List
- Protocol Descriptor List
- Profile Descriptor List
- Language Base Attr List

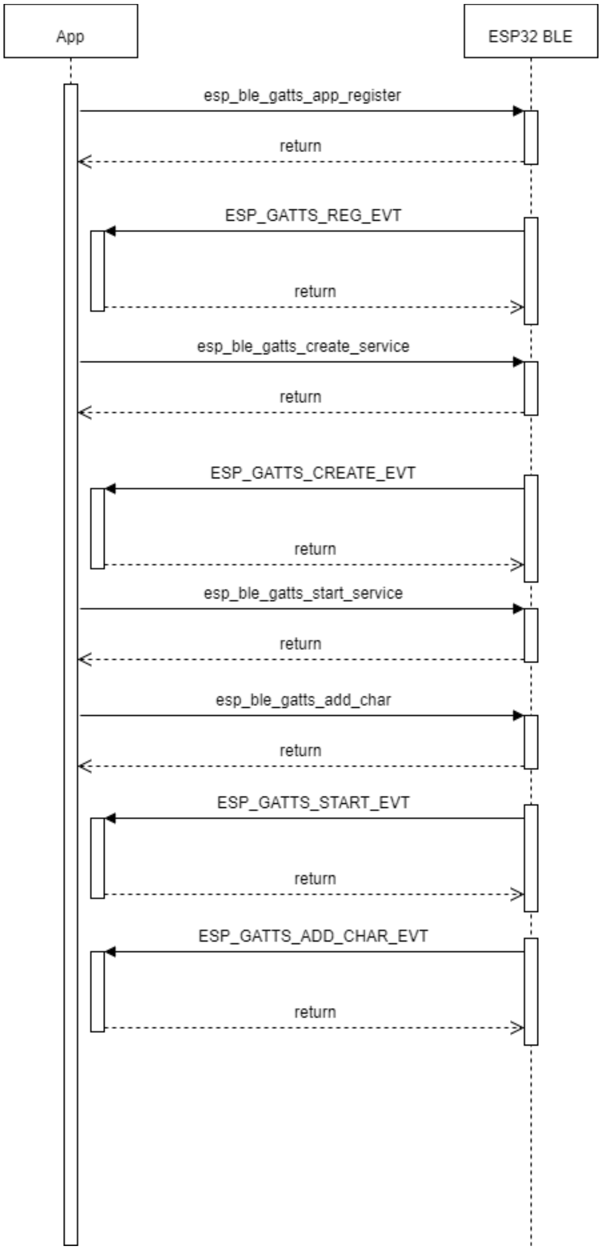
See also:

- man(1) – sdptool

ESP32 and Bluetooth

Logic appears to be:

```
esp_bt_controller_init()
esp_bt_controller_enable(ESP_BT_MODE_BTDM)
esp_bluedroid_init()
esp_bluedroid_enable()
esp_ble_gap_register_callback()
esp_ble_gattc_register_callback()
esp_ble_gattc_app_register()
esp_ble_gap_set_scan_params()
esp_ble_gap_start_scanning(20)
```



GATT Server - Read request

When a partner requests that a characteristic be read, an `ESP_GATTS_READ_EVT` is received. This should send a response using the `esp_ble_gatts_send_response()` function.

We can test this with gatttool

```
$ sudo gatttool --device=<deviceAddr> --interactive
??> connect
??> char-read-uuid <charUUID>
handle: 0x????   value: ??
```

See also:

- ESP_GATTS_READ_EVT
- esp_ble_gatts_send_response

Debugging ESP32 Bluetooth

The ESP32 bluetooth implementation is built upon an environment called Bluedroid. The reason this becomes important is that for **full** understanding of the ESP32 Bluetooth environment, we need to understand the Bluedroid environment as well. For example, the art of getting lower level diagnostics drops us down to the Bluedroid APIs. For example:

```
#include <gatt_api.h> // bluedroid include
…
GATT_SetTraceLevel(6);
```

will switch on GATT level tracing.

Bluetooth C Programming in Linux

Within the C / Linux environment we have an implementation of the API stack called "BlueZ". The BlueZ implementation supports RFCOMM, L2CAP, SCO and HCI.

In order to perform Bluetooth programming we must install the package called "libbluetooth-dev" using:

```
$ sudo apt-get install libbluetooth-dev
```

When compiling, we need to link with libbluetooth.

See also:

- An Introduction to Bluetooth Programming

hci_get_route

Retrieve a handle to the specified Bluetooth device or the first available if NULL is supplied.

```
int hci_get_route(bdaddr_t *bdaddr)
```

hci_open_dev

Open the specified device and get a handle to that device. The returned value is a socket.

```
int hci_open_dev(int dev_id)
```

If the return is -1 then an error was encountered and the details of the error can be found in `errno`.

hci_inquiry

```
int hci_inquiry(
       int dev_id,
       int len,
       int max_rsp,
       const uint8_t *lap,
       inquiry_info **ii,
       long flags)
```

where:

- `dev_id` – the device id of the adapter as retruned by hci_get_route.
- `len` – The duration of the scan * 1.28 seconds.
- `max_rsp` – The maximum number of responses we are willing to accept.
- `lap` – may be NULL
- `ii` – Pointer to an array of inquiry_info structures to be populated. The storage must exist and be at least of size `max_rsp * sizeof(inquiry_info)`.
- `flags`
  - `0` – Cached results allowed
  - `IREQ_CACHE_FLUSH` – Any cached values are discarded and only new responses will be used.

The inquiry_info is a struct containing:

- bdaddr_t bdaddr
- uint8_t pscan_rep_mode
- uint8_t pscan_period_mode
- uint8_t pscan_mode
- uint8_t dev_class[3] – The device class is encoded in the assigned numbers – baseband.
- uint16_t clock_offset

See also:

- Assigned numbers – baseband

hci_read_remote_name

Retrieve the display name of a specified device.

```
int hci_read_remote_name(
       int hci_sock,
       const baddr_t *addr,
       int len,
       char *name,
       int timeout
)
```

- `len` – the size of the name buffer to hold the display name.
- `name` – a buffer to hold the display name.
- `timeout` – Maximum number of milliseconds to wait before giving up.

On return, a value of `0` indicates success.

str2ba

Convert a string representation of a Bluetooth address into an address.

```
str2ba(const char *str, bdaddr_t *ba)
```

Where `str` is the string representation of the Bluetooth address and `ba` is a pointer to a `bdaddr_t` structure to hold the resulting address.

ba2str

Convert a Bluetooth address to a string. The string buffer must be at least 18 bytes long.

```
ba2str(const bdaddr_t *ba, char *str)
```

The ba is a pointer to a `bdaddr_t` structure while `str` is the buffer to be populated with the string representation.

sdp_connect()

sdp_service_search_attr_req()

sdp_record_register()

Bluetooth Audio

Bluetooth speakers and headphones are common so a natural question would be how to get sound out of them from the Pi. The answer is to install an application called PulseAudio.

```
$ sudo apt-get install pulseaudio pulseaudio-module-bluetooth
```

The Bluetooth profile we want to work with is called A2DP (Advanced Audio Distribution Profile).

Bluetooth RFCOMM

A serial protocol is available via Bluetooth and is called "RFCOMM" for "Radio Frequency Communication".

When programming with C, we want to create a socket using:

```
int s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

A socket address structure for Bluetooth RFCOMM is a struct called `sockaddr_rc` which contains:

- `sa_family rc_family` – This will always be `AF_BLUETOOTH`.
- `bdaddr_t rc_bdaddr` – The address of the device to which we wish to connect or listen upon. If any local Bluetooth adapter will suffice when we are a server, we can supply `BDADDR_ANY`.
- `uint8_t rc_channel` – The channel to which we wish to connect.

To be a client of an RFCOMM server, we would use:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Usage: %s bdaddr\n", argv[0]);
    return 0;
  }

  struct sockaddr_rc addr = {0};
  int s, status;
  char *dest = argv[1];

  s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
  addr.rc_family = AF_BLUETOOTH;
  addr.rc_channel = 1;
  str2ba(dest, &addr.rc_bdaddr);

  status = connect(s,(struct sockaddr *)&addr, sizeof(addr ));
  if(status == 0) {
     status = send(s, "hello!", 6, 0);
  }

  if(status < 0) {
     perror("connect");
  }

  close(s);
  return 0;
}
```

while to be a server we would use:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 1;
    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);

    ba2str(&rem_addr.rc_bdaddr, buf);
    fprintf(stderr, "accepted connection from %s\n", buf);
    memset(buf, 0, sizeof(buf));
    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if (bytes_read > 0) {
        printf("received [%s]\n", buf);
    }

    // close connection
    close(client);
    close(s);
    return 0;
}
```

See also:

- RFCOMM with TS 07.10
- man(1) – rfcomm
- An Introduction to Bluetooth Programming

Bluetooth tools

l2ping

In TCP/IP networking, we have a tool called ping which sends an ICMP packet over the network to which the partner responds. We can use this command to determine the "liveness" of the partner as well as get the round trip response times. For Bluetooth, we have a similar tool called "`l2ping`". This command sends a L2CAP echo request to the partner and waits for the response.

At the highest level, we use:

```
$ sudo l2ping <bd_addr>
```

where bd_addr is the address of the target device. Here is an example output:

```
$ sudo l2ping 00:1A:7D:DA:71:13
Ping: 00:1A:7D:DA:71:13 from B8:27:EB:62:03:9F (data size 44) ...
44 bytes from 00:1A:7D:DA:71:13 id 0 time 11.32ms
44 bytes from 00:1A:7D:DA:71:13 id 1 time 66.06ms
44 bytes from 00:1A:7D:DA:71:13 id 2 time 19.84ms
44 bytes from 00:1A:7D:DA:71:13 id 3 time 52.38ms
```

As of 05/2016, running the `l2ping` command on current Raspbian ends after about 5 seconds of pinging with a message:

```
Send failed: Connection reset by peer
```

Current thinking is that this is caused by a deliberate kernel timeout of L2CAP requests that don't result in a connection. The theory believed to be that to save battery life in real Bluetooth devices, if a connection isn't established in the timeout period, don't bother to keep trying and, presumably, waste power resources.

The l2ping command is delivered as part of the "`bluez`" package.

See also:

- man(1) – l2ping

rfcomm

See also:

- man(1) – rfcomm

bluetoothctl

One of the primary tools for working with Bluetooth is called `bluetoothctl`. Oddly, this command seems to have only the bare bones of a man page (shame).

One should run `bluetoothctl` as root using:

```
$ sudo bluetoothctl
```

If you fail to run it as root, it will just silently sit there until you interrupt it. This command line tool has the following commands:

- `list` – List available controllers
- `show [ctrl]` – Controller information
- `select <ctrl>` – Select default controller
- `devices` – List available devices
- `paired-devices` – List paired devices
- `power <on/off>` – Set controller power
- `pairable <on/off>` – Set controller pairable mode
- `discoverable <on/off>` – Set controller discoverable mode
- `agent <on/off/capability>` – Enable/disable agent with given capability
- `default-agent` – Set agent as the default one
- advertize <on/off/capability>
- `scan <on/off>` – Scan for devices
- `info <dev>` – Device information
- `pair <dev>` – Pair with device
- `trust <dev>` – Trust device
- `untrust <dev>` – Untrust device

- `block <dev>` – Block device
- `unblock <dev>` – Unblock device
- `remove <dev>` – Remove device
- `connect <dev>` – Connect device
- `disconnect <dev>` – Disconnect device
- `list-attributes <dev>` – List the attributes on the device.
- set-alias <alias>
- select-attribute <attribute>
- attribute-info [attribute]
- `read` – Read the value of the currently selected attribute.
- write
- `notify <on/off>` - Enable or disable notification for the currently selected attribute.
- register-profile
- unregister-profile
- `version` – Display version
- `quit` – Quit program

See also:

hciconfig

As mentioned, HCI is the "Host-Controller Interface" which is the layer of communication between the higher level protocols of bluetooth and the bluetooth lower level controller. The "`hciconfig`" command allows us to execute commands through this logical interface.

Running `hciconfig` by itself will list all the bluetooth devices found on the computer:

```
$ hciconfig
hci0:  Type: BR/EDR  Bus: UART
       BD Address: B8:27:EB:62:03:9F  ACL MTU: 1021:8  SCO MTU: 64:1
       UP RUNNING PSCAN
       RX bytes:19100 acl:150 sco:0 events:457 errors:0
       TX bytes:7952 acl:150 sco:0 commands:184 errors:0
```

Each bluetooth device has a logical identifier of the form "`hciX`" where the devices are numbered 0, 1, 2 ... etc.

To refer to a specific device, most of the commands that we issue through `hciconfig` will take the `hciX` device identifier to target the correct instance.

Some of the more interesting commands we can perform with `hciconfig` include:

- Getting and setting the devices display name property
- Enabling/disabling page support
- Enabling/disabling scan inquiry support

The `hciconfig` command is supplied as part of the "bluez" package.

Some useful commands include:

Start LE broadcasting connectable undirected advertising

```
$ sudo hciconfig hci0 leadv 0
```

Start LE broadcasting non-connectable undirected advertising

```
$ sudo hciconfig hci0 leadv 3
```

Stop LE broadcasting

```
$ sudo hciconfig hci0 noleadv
```

See also:

- hcitool
- man(8) – hciconfig

hcidump

This tool is installed through:

```
$ sudo apt-get install bluez-hcidump
```

The tool appears to dump the commands sent through the host/controller interface.

Try running with:

```
$ sudo hcidump -x -R
```

to see low-level data.

hcitool

The `hcitool` is delivered as part of the "`bluez`" package on Linux.

We can issue raw commands through the HCI using:

```
hcitool cmd <OGF> <OCF>
```

Where the combination of the two bytes <OGF> and <OCF> define the command to run. For LE controller commands, the <OGF> is 0x08.

To scan for bluetooth LE devices, use:

```
$ sudo hcitool lescan
```

We can start advertising packets. See BT Spec 4.2 Vol 2, Part E 7.8.7

Using:

```
$ sudo hcitool cmd 0x08 0x0008 <Length> <Content>
$ hciconfig hci0 leadv 0
```

See also:

- man(1) – hcitool
- hciconfig

gatttool

Interact with a BLE device at the GATT level. In order to interact with a BLE device at the gatt level, we need its device address. Using "`hictool lescan`" is a good way to get the address. Typically the program is run with:

```
$ sudo gatttool --device=<Address> --interactive
```

This will return a command prompt that starts with the partner device address:

```
[FF:FF:45:19:14:80][LE]>
```

The sub-commands available to us include:

```
connect         [address [address type]]       Connect to a remote device
disconnect                                     Disconnect from a remote device
primary         [UUID]                         Primary Service Discovery
included        [start hnd [end hnd]]          Find Included Services
characteristics [start hnd [end hnd [UUID]]]   Characteristics Discovery
char-desc       [start hnd] [end hnd]          Characteristics Descriptor Discovery
char-read-hnd   <handle>                       Characteristics Value/Descriptor Read
by
                                               handle
char-read-uuid  <UUID> [start hnd] [end hnd]   Characteristics Value/Descriptor Read
by UUID
char-write-req  <handle> <new value>           Characteristic Value Write (Write
Request)
char-write-cmd  <handle> <new value>           Characteristic Value Write (No
response)
sec-level       [low | medium | high]          Set security level. Default: low
mtu             <value>                        Exchange MTU for GATT/ATT
```

Once connected, we can interrogate the device about its primary function by running the "`primary`" command:

```
[FF:FF:45:19:14:80][LE]> primary
attr handle: 0x0001, end grp handle: 0x0005 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0006, end grp handle: 0x0008 uuid: 0000180f-0000-1000-8000-00805f9b34fb
attr handle: 0x0009, end grp handle: 0x000b uuid: 00001802-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000e uuid: 0000ffe0-0000-1000-8000-00805f9b34fb
```

Notice specifically the UUIDs. These correspond to the assigned numbers of the GATT specifications. Contrast this with the output of "bluetoothctl info" which shows the following:

```
[bluetooth]# info FF:FF:45:19:14:80
Device FF:FF:45:19:14:80
       Alias: FF-FF-45-19-14-80
       Appearance: 0x03c1
       Icon: input-keyboard
       Paired: no
       Trusted: no
       Blocked: no
       Connected: yes
       LegacyPairing: no
       UUID: Generic Access Profile    (00001800-0000-1000-8000-00805f9b34fb)
       UUID: Immediate Alert           (00001802-0000-1000-8000-00805f9b34fb)
       UUID: Battery Service           (0000180f-0000-1000-8000-00805f9b34fb)
       UUID: Unknown                   (0000ffe0-0000-1000-8000-00805f9b34fb)
```

For example, 0x...1802… is the Immediate Alert service.

Looking back at the `gatttool` output, for each of the services, we see a "handle range". This describes the handles to the characteristics offered by that service. From the handles, we can ask the device what characteristics these represent:

```
[FF:FF:45:19:14:80][LE]> char-desc 0x0009 0x000b
handle: 0x0009, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x000a, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x000b, uuid: 00002a06-0000-1000-8000-00805f9b34fb
```

The UUIDs of the declarations and characteristics can then be examine in the assigned-numbers lists.

For example

- 0x...2800… – GATT Primary Service Declaration
- 0x...2803 … – GATT Characteristic Declaration
- 0x...2A06... – Alert Level

See also:

- Bluetooth GATT
- man(1) – gatttool

Bluetooth examples

The iTag peripheral

The iTag is a cheap little thingy ($2-$4 on eBay) that is a bluetooth device. Its purpose is to form a connection with a bluetooth master. If the connection is subsequently lost, the device starts beeping. In addition, the device can receive a "ping" from the master to instruct it to immediately start beeping. In all these cases, it acts as a useful device to help you locate it should it get lost. Now imagine connecting it to your keys, pet, kid or other loose-able thing and now we have a potential of either being notified that it is out of range or else it will beep to say "help me".

This makes a great device for testing the ESP32's ability to work as a BLE master.

If we run a BLE scan and listen for advertising packets, we will find that it shows up. From there we can get its bluetooth address. For example, when I ran an ESP32 to look for devices, the ESP32 found my tag with the address "FF:FF:45:14:80". Once the ESP32 found the address, I was the able to issue an open request to it which succeeded. Once I had an open connection, I issued a search request upon it and four services were returned. These were:

- UUID: 0x1800
- UUID: 0x1802
- UUID: 0x180f
- UUID: 0xffe0

Since these were 16bit UUIDs, that is the indication that they are specified by the bluetooth special interest group (SIG). A search at the GATT services web page found the first three:

- UUID: 0x1800 – Generic Access
- UUID: 0x1802 – Immediate Alert
- UUID: 0x180f – Battery Service

Great … this is starting to make sense. Now we can drill down into the characteristics for each service. From the bluetooth specs, the characteristics possible are:

- UUID: 0x1800 – Generic Access
  - UUID: 0x2a00 – Device Name
  - UUID: 0x2a01 – Appearance
  - UUID: 0x2a02 – Peripheral Privacy Flag
  - UUID: 0x2a03 – Reconnection Address
  - UUID: 0x2a04 – Peripheral Preferred Connection Parameters
- UUID: 0x1802 – Immediate Alert
  - UUID: 0x2a06 – Alert Level
- UUID: 0x180f – Battery Service
  - UUID: 0x2a19 – Battery Level

When we actually performed a characteristics query, what was returned were:

- UUID: 0x1800 – Generic Access
  - UUID: 0x2a00 – Device Name
  - UUID: 0x2a01 – Appearance
- UUID: 0x1802 – Immediate Alert
  - UUID: 0x2a06 – Alert Level
- UUID: 0x180f – Battery Service
  - UUID: 0x2a19 – Battery Level

Close enough to what we expected.

Smart Watch / The TW64 Band

The TW64 watch/band can be found on eBay for about $9. Performing an eBay search using "TW64" will turn up many.

We start by running a BLE scan:

```
$ sudo hcitool lescan
```

Which came back with:

```
LE Scan ...
A4:C1:38:77:1A:19 KeepBand
A4:C1:38:77:1A:19 (unknown)
```

And now we know the device address. Of course, each instance will vary.

next we can connect to the device and ask it about itself:

```
$ sudo gatttool --device=A4:C1:38:77:1A:19 --interactive
> connect
> primary
attr handle: 0x0001, end grp handle: 0x0007 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0008, end grp handle: 0x000b uuid: 00010203-0405-0607-0809-0a0b0c0d1911
attr handle: 0x000c, end grp handle: 0x0011 uuid: 66880000-0000-1000-8000-008012563489
```

and similarly, we can also run:

```
$ sudo bluetoothctl
# info A4:C1:38:77:1A:19
Device A4:C1:38:77:1A:19
       Name: KeepBand
       Alias: KeepBand
       Appearance: 0x0180
       Paired: no
       Trusted: no
       Blocked: no
       Connected: no
       LegacyPairing: no
       UUID: Human Interface Device    (00001812-0000-1000-8000-00805f9b34fb)
       UUID: Battery Service           (0000180f-0000-1000-8000-00805f9b34fb)
```

Interestingly, notice the distinction in services offered.

A search at the GATT services web page found:

- UUID: 0x1800 – Generic Access

- UUID: 0x180f – Battery Service
- UUID: 0x1812 – Human Interface Device

Again using gattool, we can ask for the description of characteristics:

```
char-desc
handle: 0x0001, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x0002, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0003, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0005, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0006, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0007, uuid: 00002a04-0000-1000-8000-00805f9b34fb
handle: 0x0008, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x0009, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x000a, uuid: 00010203-0405-0607-0809-0a0b0c0d2b12
handle: 0x000b, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x000c, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x000d, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x000e, uuid: 66880001-0000-1000-8000-008012563489
handle: 0x000f, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0010, uuid: 66880002-0000-1000-8000-008012563489
handle: 0x0011, uuid: 00002902-0000-1000-8000-00805f9b34fb
```

Web Bluetooth

Consider a browser. This is a software application that runs on your desktop or phone that retrieves web page descriptions from remote web servers and displays them. In addition, you can usually interact with those pages. This much we all know.

Now also consider that a browser can also download JavaScript which runs within the context of the browser. Typically, this JavaScript is client side code that works with the visualization of the browser to provide enhanced user interaction. Again, this we know.

But here is what may be new to you. There is a specification in the works that would allow a browser to interact with external Bluetooth devices that are in peripheral mode. The browser would perform the role of the Bluetooth central. The JavaScript code running within the browser would then be able to connect to, read from and write to Bluetooth devices … and since this code is dynamically downloaded as JavaScript from the Internet, we have effectively connected our BLE devices through to the Internet.

While interesting in principle, what practical use might we have for this capability?

First of all it allows the browser to perform a richer role as the UI presentation of your applications. We are continually finding that browsers are becoming the common denominator for many user interfaces. Many of these are internet enabled but just as many are what are known as "single page applications" and may be loaded from HTML and JavaScript that is local to where it is being executed. If the browser permits connection to a BLE device then now we can visualize data upon it or send new data to it.

At the highest level, calling `navigator.requestDevice()` returns a promise that resolves to an instance of a `BluetoothDevice`. A parameter to the `requestDevice()` call is an object of the form:

```
{
   filters:
   optionalServices:
   acceptAllDevices: <Boolean>
}
```

The structure seems to allow the following formats:

```
{
   filters: [
      {
         service: ["<UUID>"]
      }
   ]
}
```

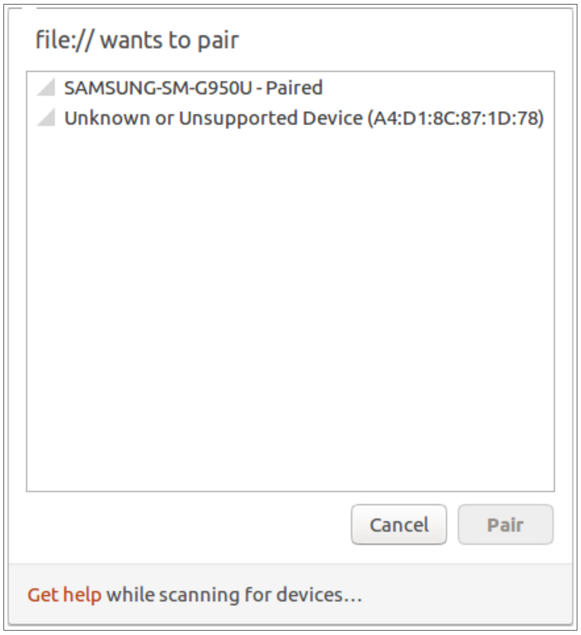
or:

```
{
   filters: [
      {
         name: ["<DEVICE NAME>"]
      }
   ],
   optionalServices: ["<UUID>"]
}
```

or:

```
{
   acceptAllDevice: true,
   optionalServices: ["<UUID>"]
}
```

The Web Bluetooth specification allows for a browser to become a BLE Central component and hence form connections to BLE Peripherals. When accessing `navigtaor.bluetooth.requestDevice()`, this **must** be part of a user interaction (for example a button press). This means that a user must explicitly and interactively choose to connect to a BLE device. An arbitrary piece of JavaScript downloaded in a web page does not have the capability to scan your BLE environment. This restriction is deliberate for security purposes.



file:// wants to pair

SAMSUNG-SM-G950U - Paired

Unknown or Unsupported Device (A4:D1:8C:87:1D:78)

Get help while scanning for devices…

A dialog is then shown asking the user for permission to connect to a device. The return from the `requestDevice()` call is an instance of `BluetoothDevice` which contains:

```
{
  id:   <String> // An id for the device.  Can be used for equality comparison.
  name: <String> // Human readable name for the device.
  gatt: <BluetoothRemoteGATTServer>
  ongattserverdisconnect: <function>
  <others>
}
```

Notice in particular the field called `gatt` that is an instance of `BluetoothRemoteGATTServer`.

The `BluetoothRemoteGATTServer` contains:

```
{
   device:     <BleutoothDevice>,
   connected:  <boolean>,
   connect:    <function>, // Return a promise returning a BluetoothRemoteGATTServer
   disconnect: <function>,
   getPrimaryService: <function>, // Return a BluetoothRemoteGATTService
   getPrimaryServices: <function> // Return a promise for the remote GATT services
(BluetoothRemoteGATTService) on this device.  Must be connected first.
}
```

- `device` (`BluetoothDevice`) – Reference to the Bluetooth device.
- `connected` (`boolean`) – Are we connected to the GATT server?

- `connect` (`function`) – Connect to the GATT server.
- `disconnect` (`function`) – Disconnect from the GATT Server.
- `getPrimaryService` (`function`) – Get the primary service. The input to the function is a service UUID and the return is a `BluetoothRemoteGATTService`. Typically, the UUID has to be explicitly permitted in the `optionalServices` field of the `requestDevice()` call.
- `getPrimaryServices` (`function`) – Get the array of primary services.

Notice the `connect()` function. This allows us to connect from our browser to the BLE peripheral. Once done, we can start to work with the services.

A service is described by the `BluetoothRemoteGATTService` object.

```
{
   isPrimary: <Boolean>
   uuid: <String>
   getCharacteristic: <function>
   getCharacteristics: <function>
}
```

- `isPrimary` (`Boolean`) – True if this is a primary service.
- `uuid` (`String`) – UUID of the service.
- `getCharacteristic` (`function`) – A promise to return a `BluetoothGATTCharacteristic`. A UUID for a characteristic is required.
- `getCharacteristics` (`function`) – A promise to return an array of `BluetoothGATTCharacteristic`.

A characteristic is described by the `BluetoothRemoteGATTCharacteristic` object.

```
{
   oncharacteristicvaluechanged: <function>
   properties: <BluetoothCharacteristicProperties>
   service: <BluetoothRemoteGATTService>
   uuid: <String>
   value: <DataView>,
   getDescriptor: <function>
   getDescriptors: <function>
   readValue: <function>
   startNotifications: <function>
   stopNotifiications: <function>
   writeValue: <function>
}
```

- `oncharacteristicvaluechanged` (`function`) – A function that you can specify that will be invoked when the value of the characteristic is changed.

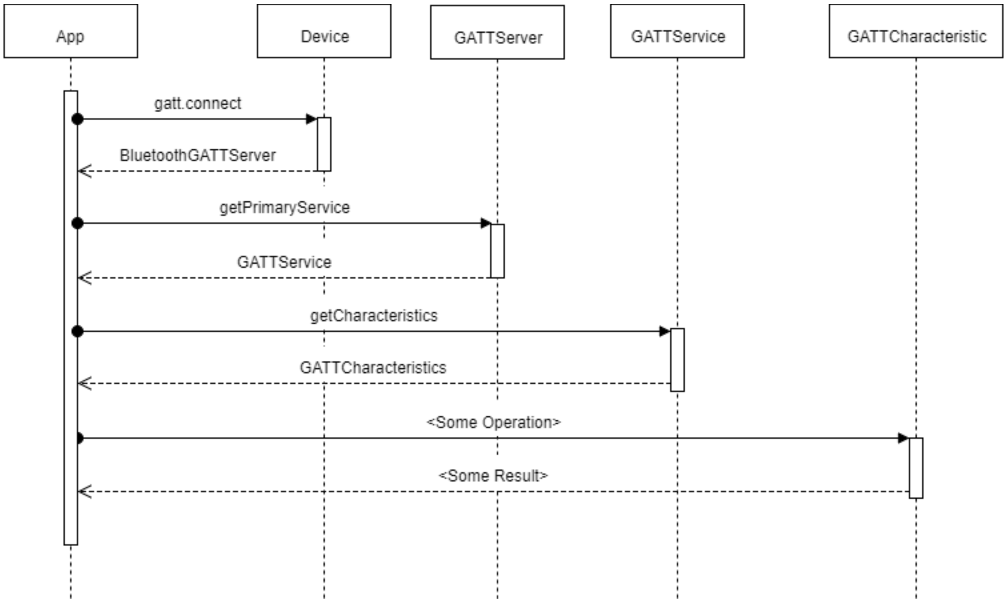
- `properties` (`BluetoothCharacteristicProperties`) – An object that defines the properties of this characteristic.
- `service` (`BluetoothRemoteGATTService`) – A reference to the service that owns this characteristic.
- `uuid` (`UUID`) – The UUID of this characteristic.
- `value` – The value … may be null.
- getDescriptor
- getDescriptors
- `readValue` (`function`) – A function which, if available and called, will return a promise for a `DataView`. This `DataView` will contain the value of the characteristic.
- `startNotifications` (`function`) – Call to start notification events. Returns a promise.
- `stopNotifications` (`function`) – Call to stop notification events. Returns a promise.
- writeValue

The `BluetoothCharacteristicProperties` is an object that defines the core properties of the characteristic. It contains:

```
{
   authenticatedSignedWrite: <boolean>,
   broadcast:                <boolean>,
   indicate:                 <boolean>,
   notify:                   <boolean>,
   read:                     <boolean>,
   reliableWrites:           <boolean>,
   writeableAuxiliaries:     <boolean>,
   write:                    <boolean>,
   writeWithoutResponse:     <boolean>
}
```

- authenticatedSignedWrites (boolean)
- broadcast (boolean)
- `indicate (boolean)` – Can we be indicated of a value change?
- `notify` (`boolean`) – Can we be notified of a value change?
- `read` (`boolean`) – Can we read the value?
- reliableWrites (boolean)

- writeableAuxiliaries (boolean)
- `write` (`boolean`) – Can we write the value?
- writeWithoutResponse (boolean)



Once we have the `BleutoothRemoteGATTCharacteristic`, the fun can start.

First, we need to be cognizant of the values contained in the properties of this object. The properties declare what we are and are not permitted to do. For example, if the `read` property is true, then we are allowed to invoke `readValue()`. If the value of the property if false, then we may not invoke `readValue()`. Putting it simpler, just because there is a function that appears to be available, doesn't mean that we can sensibly invoke it.

We can register to be notified when a value changes:

```
var handlerFunc = function(event) {
   ...
}
characteristic.addEventListener("characteristicvaluechanged", handlerFunc);
```

The event object passed into the callback handler function has a `target` property which will be the instance of the `BluetoothRemoteGATTCharacteristic`. We can read the `value` property from this to get the current value. The `value` is an instance of a `DataView`. A `DataView` is a buffer of bytes from which we can read either bytes or numeric values. Should we want a string representation, the following is a useful little algorithm:

```
function dataViewToString(dataview) {
   var result = "";
   for (var i=dataview.byteOffset; i<dataview.byteLength; i++) {
      result += String.fromCharCode(dataview.getInt8(i));
   }
   return result;
}
```

To run a Web browser with Web Bluetooth on a Linux environment, some OS prerequisites are required including:

- Kernel 3.19+
- Bluez 5.41+

Both of these are satisfied with Ubuntu 17.04 or better.

In chrome, we can examine the Bluetooth information that is found using:

```
chrome://bluetooth-internals
```

We can start trace using:

```
google-chrome --enable-logging=stderr --vmodule=*bluetooth*=9 --enable-experimental-web-platform-features
```

Note: When working with Web Bluetooth, the use of "arrow functions" and "promises" can come in very handy.

Imagine a place where a function reference might be supplied. For example:

```
function myFunction(functionRef) {
   functionRef(value);
}
```

In the above, the parameter to myFunction must itself be a function. A typical call might be:

```
myFunction(function(myValue) {
   // do something with myValue
});
```

The concept of the Arrow Function is an alternate declaration style. The equivalent to the above would be:

```
myFunction((myValue)=>{
  // do something with myValue
});
```

If there is only one parameter, we can shorten further by eliminating the parenthesis:

```
myFunction(myValue =>{
  // do something with myValue
});
```

See also:

- Web Bluetooth Community
- Github: WebBluetoothCG/web-bluetooth
- Web Bluetooth Specification
- Interact with Bluetooth devices on the Web
- Google – Web-Bluetooth
- Logging Web Bluetooth – Chrome

- MDN – BluetoothDevice
- MDN – BluetoothRemoteGATTCharacteristic
- MDN – BluetoothRemoteGATTServer
- MDN – BluetoothRemoteGATTService
- MDN – DataView

- Arrow functions

The Physical Web

Have a mobile device browser "see" BLE devices that become available to launch web pages.

Hardware interfacing

When working with an ESP32 we will quickly recognize that it has a lot of capabilities for interacting with a variety of hardware devices. These may be through GPIO, SPI, I2C, Serial or other techniques. In this section we start to examine these distinct techniques and look at each one in turn.

To interface your ESP32 to hardware, you will be delving down into the hardware levels and attaching devices to the pins of your ESP32. Since there are many modules available, you will want to become familiar with the specifics of the modules you are working with as physical pin locations may differ from one manufacturer to another.

See also:

- Modules

GPIOs

GPIO or General Purpose Input / Output is the ability to drive external pins of the ESP32 to a signal level of "1" or "0" under application control. We can also choose to read the signal level that may be present on the pin.

When we think of a GPIO we must realize that at any one time, each pin instance has two operational 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.

Another vital consideration when working with GPIOs is the voltage level. The ESP32 is a 3.3V device. You need to be extremely cautious if 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 should you think you can power or connect to the ESP32 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 that. 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.

It is also important to realize that the maximum amount of current you should anticipate drawing from an output GPIO is 12mA.

Because accidents happen when building GPIO based circuits, I recommend buying more ESP32 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.

To use the GPIO functions supplied by the ESP-IDF, we must include "`driver/gpio.h`".

Next we must call `gpio_pad_select_gpio()` to specify that the function of a given pin should be that of GPIO as opposed to some other function.

There are 34 distinct GPIOs available on the ESP32. They are identified as:

- `GPIO_NUM_0` – `GPIO_NUM_19`
- `GPIO_NUM_21` – `GPIO_NUM_23`
- `GPIO_NUM_25` – `GPIO_NUM_27`
- `GPIO_NUM_32` – `GPIO_NUM_39`

The ones that are omitted are 20, 24, 28, 29, 30 and 31.

Note that `GPIO_NUM_34` – `GPIO_NUM_39` are input mode only. You can not use these pins for signal output. Also, pins 6, 7, 8, 9, 10 and 11 are used to interact with the SPI flash chip … you can **not** use those for other purposes.

The data type called `gpio_num_t` is a C language enumeration with values corresponding to these names. It is recommended to use these values rather than attempt to use numeric values.

When we want to use a GPIO we need to be cognizant of whether we are using it as a signal input or a signal output. Think of this as the direction setting. We can set the direction with a call to `gpio_set_direction()`. For example, to set a pin as output we might call:

```
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_OUTPUT);
```

while to set it as input we might call:

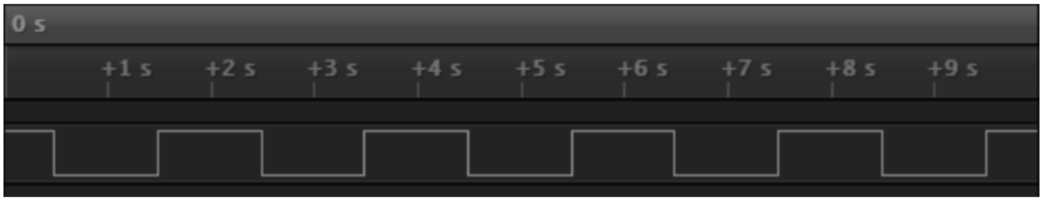
```
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_INPUT);
```

If we have set our GPIO as output, we can now set its signal value to be either 1 or 0. We do that by calling `gpio_set_level()`.

Here is an example that toggles a GPIO on and off once a second:

```
gpio_pad_select_gpio(GPIO_NUM_17);
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_OUTPUT);
while(1) {
   printf("Off\n");
   gpio_set_level(GPIO_NUM_17, 0);
   vTaskDelay(1000 / portTICK_RATE_MS);

   printf("On\n");
   gpio_set_level(GPIO_NUM_17, 1);
   vTaskDelay(1000 / portTICK_RATE_MS);
}
```

When we run this and examine the output on a logic analyzer, all is as desired:



As an alternative to setting all the attributes of individual pins, we can set the attributes of one or more pins via a single call using the gpio_config() function. This takes a structure called gpio_config_t as input and sets the direction, pull up, pull down and interrupt settings of all the pins supplied in a bit mask. For example:

```
gpio_config_t gpioConfig;
gpioConfig.pin_bit_mask = (1 << 16) | (1 << 17);
gpioConfig.mode         = GPIO_MODE_OUTPUT;
gpioConfig.pull_up_en   = GPIO_PULLUP_DISABLE;
gpioConfig.pull_down_en = GPIO_PULLDOWN_ENABLE;
gpioConfig.intr_type    = GPIO_INTR_DISABLE;
gpio_config(&gpioConfig);
```

See also:

- gpio_pad_select_gpio
- gpio_set_direction
- gpio_set_level
- gpio_config

Pull up and pull down settings

We commonly think of an input GPIO pin as having either a high or low signal supplied to it. This means that it is connected to +ve or ground. But what if it is connected to neither? In this case, the pin is considered to be in a floating state. There are times where we wish to define an unconnected pin as logically being high or low. An unconnected pin that is to be considered high is termed "pulled up" while an unconnected pin that is to be considered low is termed "pulled down". This comes from the physical hardware practice of attaching resistors to pull up or pull down the signal when it otherwise would be floating.

In the ESP32 SDK, we can define a GPIO as being pulled-up or pulled-down by using the `gpio_set_pull_mode()` function. This function takes as input the pin number we wish to set and the pull mode associated with that pin.

For example:

```
gpio_set_pull_mode(21, GPIO_PULLUP_ONLY);
```

See also:

- gpio_set_pull_mode

GPIO Interrupt handling

If we consider that the signal on a pin can move from high to low or from low to high, such a change might be something our application would be interested in knowing. To determine when such a change happens, we can continually poll the value to detect a transition change. However this in not the best solution for a number of reasons. First, we have to busily perform checking to see whether a value has changed. Secondly, there will be a latency from the time the event happens to the time when we check. Thirdly, it is possible to completely miss a signal change if the duration of the change is short. For example, if we check the value of a pin and find it high and then immediately after it goes low and then high again, the next time we poll we will still see the pin high and never have known that it was ever low for a short period.

The solution to all these problems is the notion of an interrupt. An interrupt is similar to your doorbell at your house. Without a door bell (or listening for someone knocking) you would have to periodically check to see if there is anyone at the door. This wastes your time for the majority of instances where there is no-one there and also makes sure that when there is someone there, you attend to them in a timely fashion.

In the land of ESP32s, we can define an interrupt callback function that will be called when a pin changes its signal value. We can also define what constitutes a reason for invoking the callback. We can configure the callback handler (technically called an interrupt handler) on a pin by pin basis.

First, let us consider the interrupt callback function. This is registered with a call to `gpio_isr_register()` which takes a callback handler that is invoked when an interrupt occurs on any GPIO pin occurs. Within the callback handler, we can then ask for the interrupt flag status with:

```
uint32_t gpio_intr_status = READ_PERI_REG(GPIO_STATUS_REG);    // 0-31
uint32_t gpio_intr_status_h = READ_PERI_REG(GPIO_STATUS1_REG); // 31-39
```

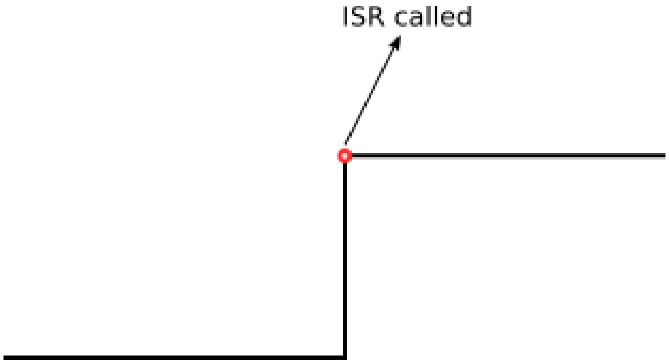
Alternative, we can use the GPIO global variable which is a pre-mapped structure and access:

- `GPIO.status_w1tc` – Flags for GPIO0 to GPIO31
- `GPIO.status1_w1tc.val` – Flags for GPIO32 to GPIO39

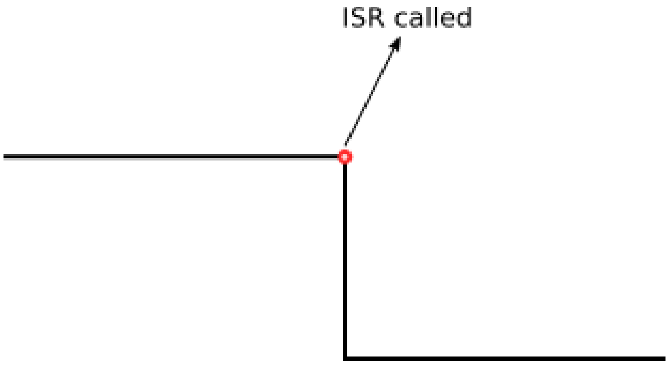
We can enable or disable interrupt handling on a GPIO by GPIO basis by calling `gpio_intr_enable()`.

To enable an interrupt for a specific pin, we use the function called `gpio_set_intr_type()`. This allows us to set the reason that an interrupt might occur. The reasons include:

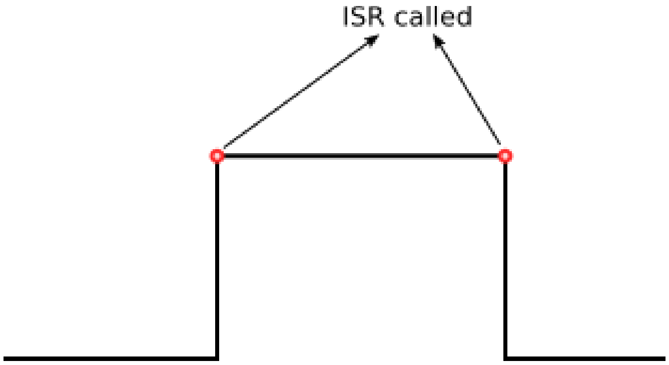
- `Disable` – don't call an interrupt on a signal change.
- `PosEdge` – Call the interrupt handler on a change from low to high.



- `NegEdge` – Call the interrupt handler on a change from high to low.



- `AnyEdge` – Call the interrupt handler on either a change from low to high or a change from high to low.



- `Hi` – Call the interrupt handler while the signal is high.
- `Lo` – Call the interrupt handler while the signal is low.

The interrupt handler can be flagged to be loaded into instruction RAM at compilation time. The default is that the generated code can live in flash. By flagging it up front as living in instruction RAM, it will always be there and ready to immediately be executed.

For example:

```
void IRAM_ATTR my_gpio_isr_handle(void *arg) {
   ...
}
```

A second implementation of interrupt handling is also provided that allows us to register a callback function associated with any given pin. This alleviates us from having to write switch code in a common interrupt handler. Think of it as a high level set of convenience functions.

Here is an example interrupt processor:

```
static char tag[] = "test_intr";
static QueueHandle_t q1;

#define TEST_GPIO (25)
static void handler(void *args) {
   gpio_num_t gpio;
   gpio = TEST_GPIO;
   xQueueSendToBackFromISR(q1, &gpio, NULL);
}

void test1_task(void *ignore) {
   ESP_LOGD(tag, ">> test1_task");
   gpio_num_t gpio;
   q1 = xQueueCreate(10, sizeof(gpio_num_t));

   gpio_config_t gpioConfig;
   gpioConfig.pin_bit_mask = GPIO_SEL_25;
   gpioConfig.mode         = GPIO_MODE_INPUT;
   gpioConfig.pull_up_en   = GPIO_PULLUP_DISABLE;
   gpioConfig.pull_down_en = GPIO_PULLDOWN_ENABLE;
   gpioConfig.intr_type    = GPIO_INTR_POSEDGE;
   gpio_config(&gpioConfig);

   gpio_install_isr_service(0);
   gpio_isr_handler_add(TEST_GPIO, handler, NULL);
   while(1) {
      ESP_LOGD(tag, "Waiting on queue");
      BaseType_t rc = xQueueReceive(q1, &gpio, portMAX_DELAY);
      ESP_LOGD(tag, "Woke from queue wait: %d", rc);
   }
   vTaskDelete(NULL);
}
```

See also:

- Interrupt Service Routines – ISRs
- gpio_install_isr_service
- gpio_isr_handler_add
- gpio_isr_handler_remove
- gpio_isr_register
- gpio_intr_enable
- gpio_intr_disable
- gpio_set_intr_type
- gpio_intr_enable
- gpio_intr_disable

Expanding the number of available GPIOs

Although the ESP devices only have a limited number of GPIO pins, that needn't be a restriction for us. We have the capability to expand the number of GPIOs available to us through some relatively inexpensive integrated circuits.

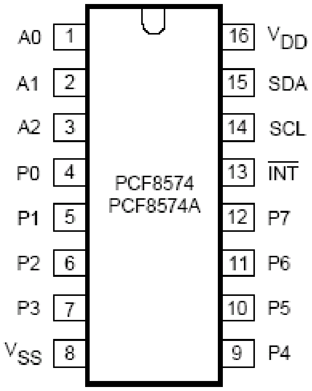
PCF8574

One of the available GPIO expanders is called the PCF8574. (The PFC8574A is the same but has a different set of addresses).

This is an I^{2}C device and hence works over only two wires. Using this IC we supply a 3 bit address (000-111) that is used to select the slave address of the device. Since each address has 8 IOs and we can have up to 8 devices, this means a total of 64 additional pins.

It appears that the device will use a pull-up resistor for a high and bring the pin to ground for low. This means that we can't use the pins for a high current source but can for low.

Here is the pin diagram for the device:



Here is a description of the pins:

Symbol	Pin	Description
A0-A2	1, 2, 3	Addressing
P0-P7	4, 5, 6, 7, 9, 10, 11, 12	Bi directional I/O
INT	13	Interrupt output
SCL	14	Serial Clock Line
SDA	15	Serial Data Line
V_{DD}	16	Supply Voltage (2.5V – 6V)
Vss (Ground)	8	Ground

The address that the slave device can be found upon is configurable via the `A0-A2` pins. It appears at the following address:

PCF8574

0	1	0	0	A2	A1	A0

PCF8574A

0	1	1	1	A2	A1	A0

The pins `A0-A2` must not float.

This results in the following table:

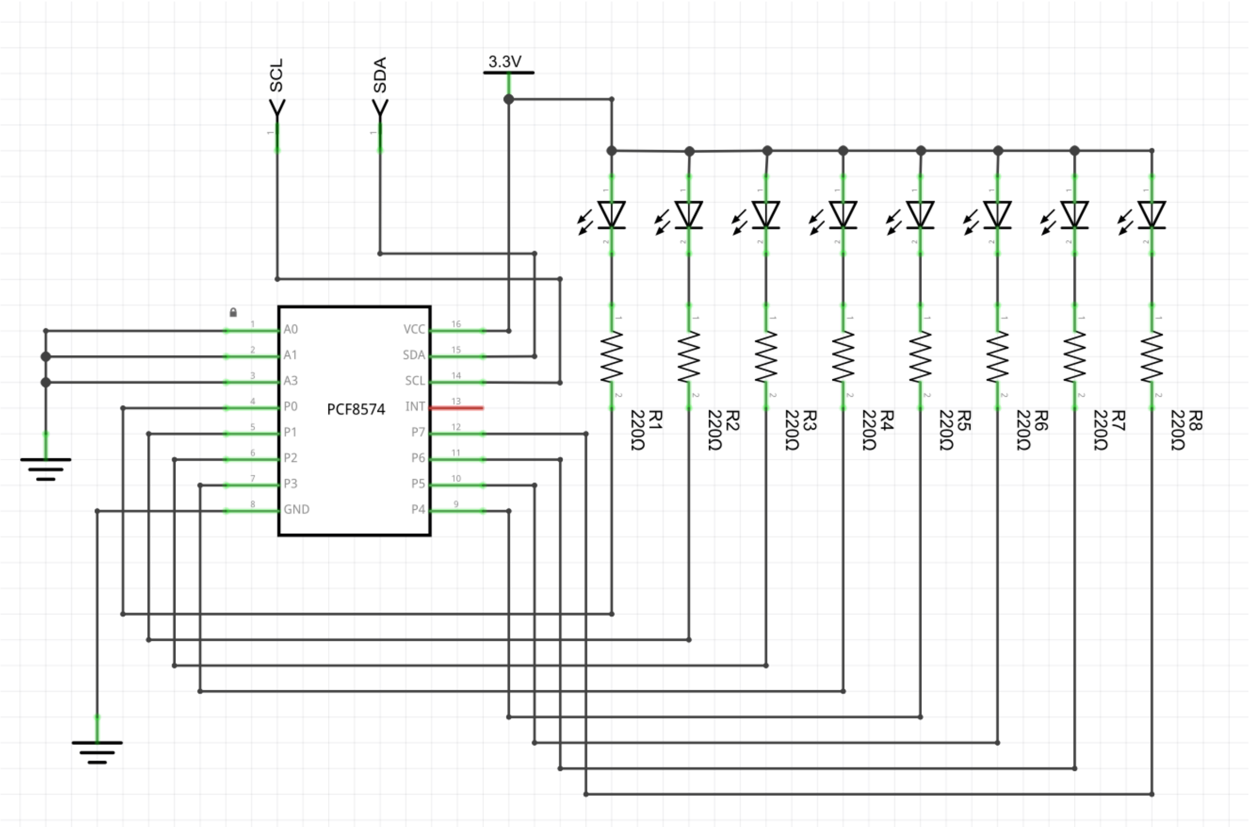
A2	A1	A0	Address PCF8574	Address PCF8574A
0	0	0	0x20	0x38
0	0	1	0x21	0x39
0	1	0	0x22	0x3a
0	1	1	0x23	0x3b
1	0	0	0x24	0x3c
1	0	1	0x25	0x3d
1	1	0	0x26	0x3e
1	1	1	0x27	0x3f

Here is an Arduino example program that drives LEDs to create a Cylon effect.

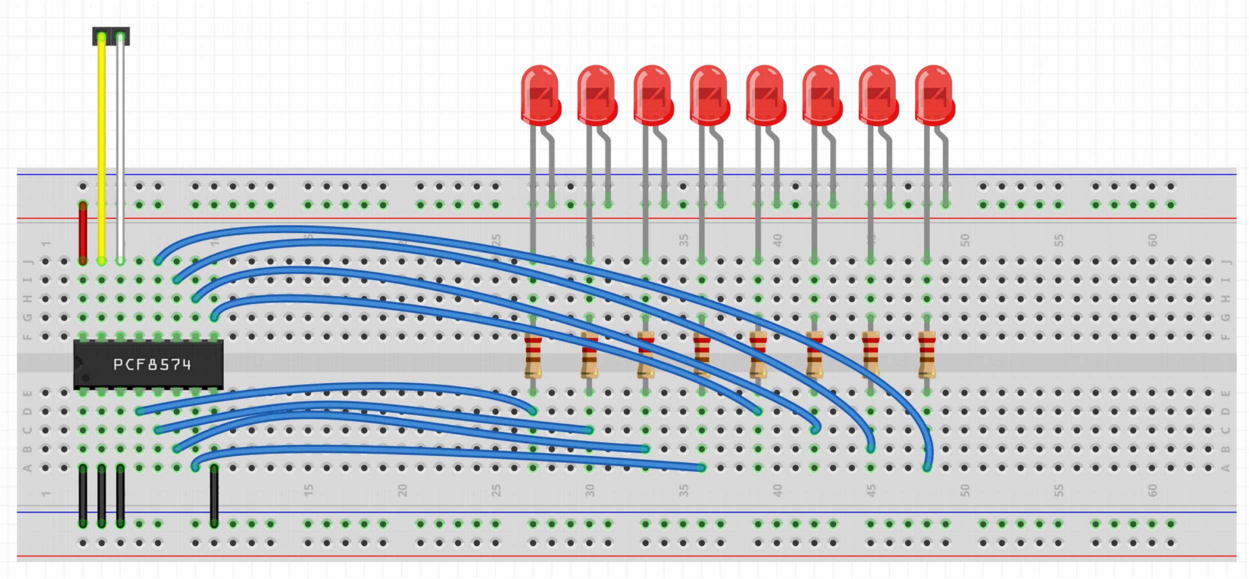
```
#include <Wire.h>
#include <Ticker.h>
// SDA - Yellow – 4
// CLK - White – 5

#define SDA_PIN 4
#define CLK_PIN 5

Ticker ticker;
int counter = 0;
int dir = 1;

void timerCB() {
   Wire.beginTransmission(0x20);
   Wire.write(~((uint8_t)1<<counter));
   Wire.endTransmission();
   counter += dir;
   if (counter == 8) {
      counter = 6;
      dir = -1;
   } else if (counter == -1) {
      counter = 1;
      dir = 1;
   }
}
```

```
void setup()
{
   Wire.begin(SDA_PIN,CLK_PIN);
   ticker.attach(0.1, timerCB);
}

void loop()
{
}
```

The corresponding circuit is:



And on a breadboard:



See also:

- 8BIT IO EXPANDER (PCF8574)
- Datasheet – NXP
- YouTube – ESP32 Technical Tutorial: PCF8574 GPIO Extender
- Product page – TI
- Working with I2C

MCP23017

The MCP23017 from Microchip is a 16 bit input/output port expander that uses the I2C interface. The device can operate from 1.8V to 5.5V. The going price for an instance of one of these on eBay is about $1. The device has 16 GPIO pins that can be set as input or output controlled in two banks (ports). We can read or write the values of one bank at a time meaning that if we want to write all 16 bits, this would be two I2C operations and the same for reading. The device is also capable of generating interrupts for input signal detection. If we imagine a 100KHz clock rate, then to switch a bit on or off (and we can do these in groups of 8) then that would be 3 bytes of data plus acknowledgments … ~30 bits … which would imply a maximum switching rate of about 0.3ms. The MCP23017 can operate at a variety of speeds including 100KHz and 400KHz.

The pin interface is:

Pin	Label	Description
1	GPB0	Bi-directional I/O
2	GBP1	Bi-directional I/O
3	GBP2	Bi-directional I/O
4	GBP3	Bi-directional I/O
5	GBP4	Bi-directional I/O
6	GBP5	Bi-directional I/O
7	GBP6	Bi-directional I/O
8	GBP7	Bi-directional I/O
9	VDD	Power (3.3V – 5V)
10	VSS	Ground
11	NC	Not connected
12	SCL	Serial clock input
13	SDA	Serial data input/output
14	NC	Not connected
15	A0	Address pin
16	A1	Address pin
17	A2	Address pin
18	RESET	Hardware reset
19	INTB	Interrupt output for port B
20	INTA	Interrupt output for port A
21	GPA0	Bi-directional I/O
22	GPA1	Bi-directional I/O
23	GPA2	Bi-directional I/O
24	GPA3	Bi-directional I/O
25	GPA4	Bi-directional I/O
26	GPA5	Bi-directional I/O
27	GPA6	Bi-directional I/O
28	GPA7	Bi-directional I/O

The I2C address of the device is 7 bits given by

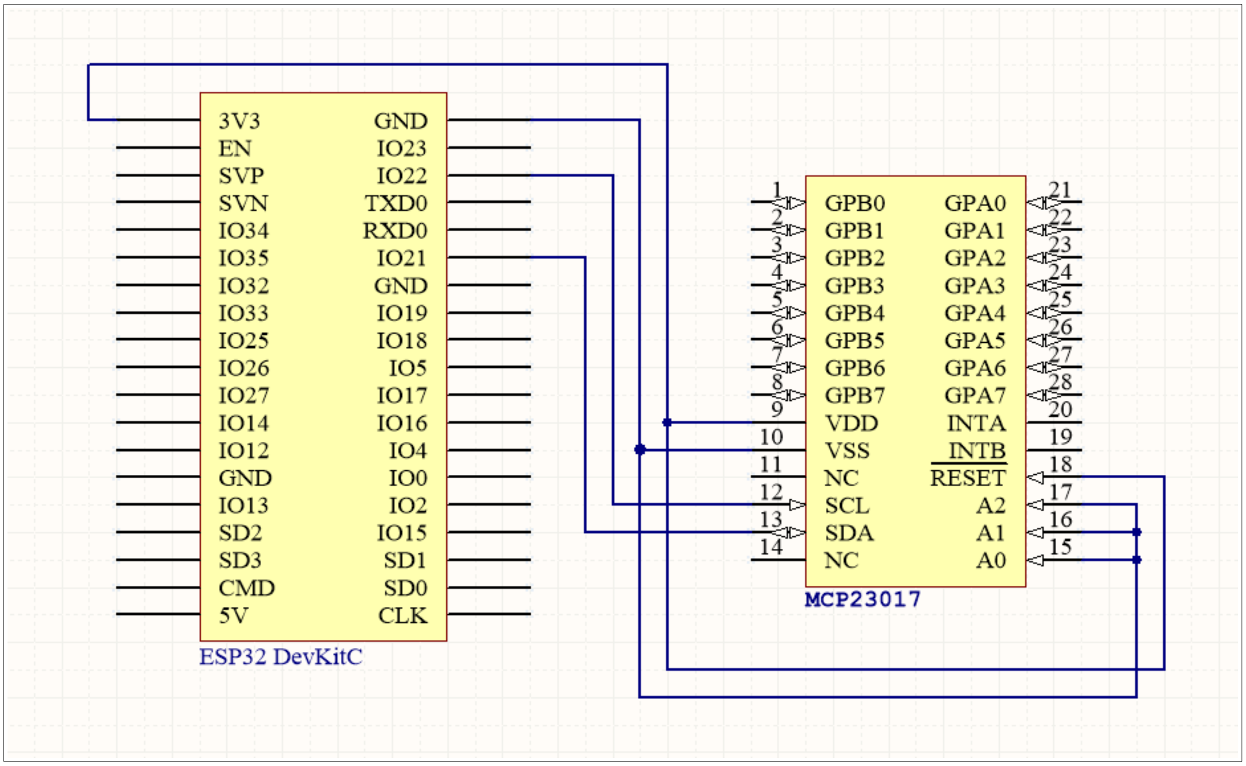
6	5	4	3	2	1	0
0	1	0	0	A2	A1	A0

And hence has addresses in the range `0x20` – `0x27`.

The register addresses are:

Register address	Name	Description
0x00	IODIRA	Direction control for port A: • 1 – input • 0 – output
0x01	IODIRB	Direction control for port B: • 1 – input • 0 – output
0x02	IPOLA	Polarity inversion for input for port A: • 1 – inverted • 0 – as-is
0x03	IPOLB	Polarity inversion for input for port B: • 1 – inverted • 0 – as-is
0x04	GPINTENA	Interrupt enable for a change for port A: • 1 – Enable input pin for interrupt on change event • 0 – Disable input pin for interrupt on change event
0x05	GPINTENB	Interrupt enable for a change for port B: • 1 – Enable input pin for interrupt on change event • 0 – Disable input pin for interrupt on change event
0x06	DEFVALA	Default values for interrupt change comparison for port A.
0x07	DEFVALB	Default values for interrupt change comparison for port B.
0x08	INTCONA	Interrupt control for port A: • 1 – Compare the value on the pin to the value in DEFVALA • 0 – Compare the value on the pin to its previous value
0x09	INTCONB	Interrupt control for port B: • 1 – Compare the value on the pin to the value in DEFVALB • 0 – Compare the value on the pin to its previous value
0x0a	IOCON	Same register as 0x0b

<table>
<tr><td></td><td></td><td><table>
<tr><th>Bit</th><th>Label</th><th>Description</th></tr>
<tr><td>7</td><td>BANK</td><td>Register mapping:
• Registers in banks
• Registers sequential</td></tr>
<tr><td>6</td><td>MIRROR</td><td>Interrupt pin mapping:
• 1 – INT pins connected
• 0 – INT pins separate. INTA is associated with port A and INTB is associated with port B</td></tr>
<tr><td>5</td><td>SEQOP</td><td>Sequential operation mode:
• 1 – Sequential operation disabled
• 0 – Sequential operation enabled</td></tr>
<tr><td>4</td><td>DISSLW</td><td>Slew rate control</td></tr>
<tr><td>3</td><td>HAEN</td><td>Hardware address enable</td></tr>
<tr><td>2</td><td>ODR</td><td>Open drain output for INT</td></tr>
<tr><td>1</td><td>INTPOL</td><td>Polarity of INT output:
• 1 – Active high
• 0 – Active low</td></tr>
<tr><td>0</td><td>N/A</td><td>Not used. Set as 0.</td></tr>
</table></td></tr>
<tr><td>0x0b</td><td>IOCON</td><td>Same register as 0x0a</td></tr>
<tr><td>0x0c</td><td>GPPUA</td><td>Pull up control for Port A:
• 1 – Pull-up via a 100K resistor
• 0 – Pull-up disabled</td></tr>
<tr><td>0x0d</td><td>GPPUB</td><td>Pull up control for Port B:
• 1 – Pull-up via a 100K resistor
• 0 – Pull-up disabled</td></tr>
<tr><td>0x0e</td><td>INTFA</td><td>Interrupt flags for Port A:
• 1 – Pin caused an interrupt
• 0 – No interrupt detected</td></tr>
<tr><td>0x0f</td><td>INTFB</td><td>Interrupt flags for Port B:
• 1 – Pin caused an interrupt
• 0 – No interrupt detected</td></tr>
<tr><td>0x10</td><td>INTCAPA</td><td>Values of GPIO when interrupt occurred on port A:
• 1 – Pin was high
• 0 – Pin was low</td></tr>
<tr><td>0x11</td><td>INTCAPB</td><td>Values of GPIO when interrupt occurred on port B:
• 1 – Pin was high
• 0 – Pin was low</td></tr>
<tr><td>0x12</td><td>GPIOA</td><td>Read – Reads the values on port A</td></tr>
<tr><td>0x13</td><td>GPIOB</td><td>Read – Reads the values on port B</td></tr>
<tr><td>0x14</td><td>OLATA</td><td></td></tr>
<tr><td>0x15</td><td>OLATB</td><td></td></tr>
</table>



See also:

- Data sheet
- Github: code fragment
- Github: telanoc/esp32_generic_i2c_rw

Interrupt Service Routines - ISRs

Imagine that you have written an ESP32 application in C. As you write that application, you make assumptions about what will happen. Imagine you write:

```
statement A;
statement B;
statement C;
```

Your assumption is that the program will perform statement A and then move on to statement B and then move on to statement C. In addition, if you have variables that are defined, unless the statements change those variables, you expect them to have the same values at the start as at the end. These are assumption that you should be able accept as true.

With an ESP32, we are programming a lot closer to the hardware than in some other environments. When working with hardware, events may happen externally to the ESP32 that require our attention. A GPIO might transition its signal level or new network packet arrive or serial data appear. In each of these cases (and more) it might be necessary to interrupt what ever we may be doing at the time and handle that event. To achieve this, we have access to what are called "Interrupt Service Routines" or ISRs. These are commonly C functions that we provide references to that may be invoked at almost anytime. They may be invoked between C statements or even during C statements. The state of the currently running program is saved and a "context switch" occurs causing the ISR to run. In order to maintain sanity, when you write an ISR there are rules that you absolutely must follow.

First an ISR must be as short as possible. Remember, when you jump into an ISR the ISR has no knowledge of what you were doing when the interrupt occurred. You may have been doing something that was time critical and you have to get back to as quickly as you can. You most certainly must never perform any activity which might be blocking.

Next, you must assume little about the state of the environment. You can't assume values of variables that you don't own and you absolutely must not change variable values that you don't own. Never mind being risky, trying to debug problems that show up once in ever 1000 runs because the timing was "just right" is a nightmare to be avoided.

You should never invoke something that might result in another interrupt being generate. This can result in a never ending loop where an ISR causes an immediate interrupt which is processed when you return from the ISR which then goes on and on.

With these limitations in mind, what then can you do? The answer is to design a solution that correctly anticipates and handles interrupt requests. There is no one formula that fits all but generally, if you can respond to the interrupt immediately with little or no side effects, then it is safe to do so. For example if an interrupt signals that a GPIO pin has changed value, you might update a memory location that is defined to reflect the "current" state of the pin and then immediately return. If you have to perform extended processing, then consider writing a message to a queue or unlocking a semaphore and then again, immediately returning. Since the ESP32 employs FreeRTOS which provides a preemptive task scheduler, you can have tasks that run in the background that "wake up" when events occur. These wake-ups need to be decoupled from the ISR and a queue or semaphore is a great way to achieve that. The following APIs are safe in ISR routines:

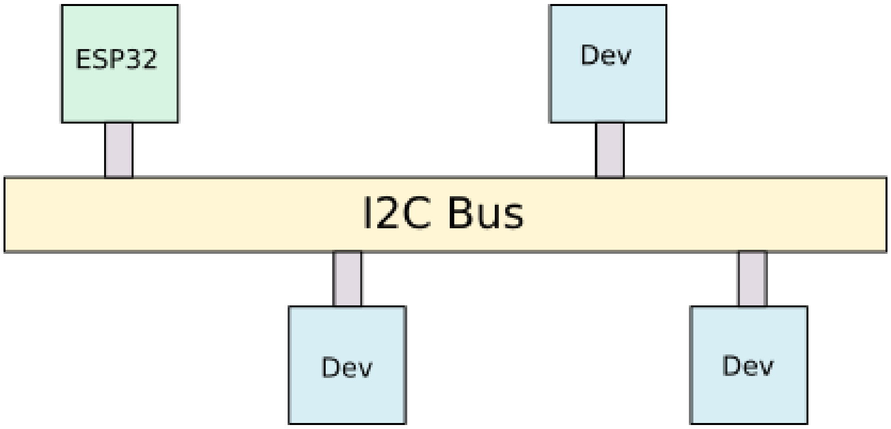
- xTaskResumeFromISR()
- xTaskNotifyGiveFromISR()

- xTaskNotifyAndQueryFromISR()
- xTaskNotifyFromISR()
- xQueueSendFromISR()
- xQueueSendToBackFromISR()
- xQueueSendToFrontFromISR()
- xQueueReceiveFromISR()
- uxQueueMessagesWaitingFromISR()
- xQueueOverwriteFromISR()
- xQueuePeekFromISR()
- xQueueIsQueueFullFromISR()
- xQueueIsQueueEmptyFromISR()
- xQueueSelectFromSetFromISR()
- xSemaphoreTakeFromISR()
- xSemaphoreGiveFromISR()
- xTimerStartFromISR()
- xTimerStopFromISR()
- xTimerChangePeriodFromISR()
- xTimerResetFromISR()
- xTimerPendFunctionCallFromISR()
- xEventGroupSetBitsFromISR()
- xEventGroupClearBitsFromISR()
- xEventGroupGetBitsFromISR()

Working with I2C

The I2C interface is a serial interface technology for accessing devices. It has two signal lines called SDA (Serial Data) and SCL (Serial Clock). The ESP32 can act as a master and the devices connected downstream act as slaves. Up to 127 distinct slaves are theoretically attachable. Each slave device has a unique address and the master decides which slave is to receive data or be allowed to speak next. In addition, the ESP32 can also be a slave device communicating with an external master.

All the slaves connected use an "open drain" connection to the bus. This means that when they connect, their attachment is either open circuit or ground as an output. Because of this it is impossible for there to be an electrical conflict as it would be impossible for one device to assert a high signal while another tried to assert a low signal. The presence of a logical high signal occurs when the current slave device goes open circuit. This means that we need pull-up resistors on the lines such that when no-one is actively asserting a low signal, they are pulled-up to a logical high signal. A resistor value of 4.7KΩ is recommended.



The start of a transmission is indicated when the SCL is left high and SDA is pulled low. This informs all the slave devices that an address is about to be issued. When the address is seen by all the slaves, only one of them should match and the other devices ignore the request.

The address of a slave follows the initial start indication and is comprised of 7 bits with most significant bit first. Following the 7 bit address is a final 8th bit that indicates whether this is a read or a write request. A value of 1 indicates a read from the slave while a value of 0 indicates a write from the master.

On the SDA line, immediately after the 8 bits of address, comes the acknowledgment bit. This bit is **not** transmitted from the master to the slave but is instead transmitted from the slave to the master. Be sure you understand that as when looking at diagrams showing data on the SDA wire, those diagrams typically do not show the origination of the data, only their sequence. The turn around time from the last bit of the 8 bit address/direction data sent from the master to the acknowledgment bit sent from the slave happens without missing any clock cycles so has to be fast. A value of 0 in the acknowledgment states that the slave has received the data. A value of 1 in the acknowledgment states that no-one is responding or the slave is not present.

Following this addressing frame comes the data frame or frames. For a master write request, the master will send 8 bits of data and expect a single bit of acknowledgment.

For a read from the slave, the slave will send 9 bits of data (8 data bits and an acknowledgment).

The master will finally send an end of communication (or stop) indication which is a transition to high on the clock with NO corresponding transition to low and **then** a transition from low to high on the SDA line.

See also:

- I2C Bus
- Sparkfun – Tutorial: I2C

Using the ESP-IDF I2C driver

The ESP-IDF provides an I2C driver that allows us to control I2C functions from a C program at a high level without having to resort to low level register manipulation. Within the ESP32, the I2C functions are baked into the hardware of the IC and the drivers make working with I2C significantly simpler. The ESP32 can provide the services of both an I2C master controller as well as the ability to be an I2C slave. There are two independent I2C ports so we can participate in two I2C buses simultaneously … either as two masters, two slaves or one master and one slave. Or, obviously, if we only need one bus, we can just ignore the other. The two ports are named `I2C_NUM_0` and `I2C_NUM_1`. Before we can use an I2C port, we need to configure it.

In many ICs, the pin numbers exposed for I2C are fixed. This is not the case in the ESP32. Arbitrary exposed pins can be used for I2C. Since I2C is a "two wire" bus with one wire for clock and the other for data, for each I2C port we want to use we have to select two pins for these purposes. One pin will be the "`scl`" (clock) pin and the other will be the "`sda`" (data) pin.

To use I2C, we must configure the I2C environment. This involves a call to `i2c_param_config()`. This function takes two parameters. The first is the I2C port we are going to configure and the second is a structure containing the details of the configuration. Within this structure we have the following fields:

- `mode` – The mode that we are configuring this I2C port. This is where we define the I2C participation of the ESP32 as either a master or a slave. The choices we can supply here are either `I2C_MODE_MASTER` or `I2C_MODE_SLAVE`.
- `sda_io_num`, `scl_io_num` – These two fields define the pin number we will use for data and clock. Again, there are no fixed/pre-defined pin numbers.
- `sda_pullup_en`, `scl_pullup_en` – These two fields define whether the pins selected for data and clock will be pulled-up to a default of high.

- `clk_speed` – This field defines the clock speed of the bus when it is a master. A value of 100000 is normal for an I2C standard mode 100KHz bus but the ESP32 also supports the I2C fast mode 400KHz bus.

Here is an example:

```
i2c_config_t conf;
conf.mode = I2C_MODE_MASTER;
conf.sda_io_num = 25;
conf.scl_io_num = 26;
conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
conf.master.clk_speed = 100000;
i2c_param_config(I2C_NUM_0, &conf);
```

After we have configured how we want our I2C bus to behave, we can enable it by actually installing the driver. We do this with a call to `i2c_driver_install()`. When calling this function we name the port, whether we are a master or a slave and, if we are a slave what buffer sizes to use.

For example, when being a master:

```
i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0);
```

Once we have initialized the I2C environment, we can now start thinking of the permutations of actions that we wish to perform. If we are an I2C master, then we can perform a write operation to an identified slave or perform a read operation to an identified slave.

First let us refresh ourselves on the notion of slave addresses. A slave address is typically 7 bits of address plus 1 bit flag of whether this is a read or write operation.

Address (7 bits)							Read or Write
Add 6	Add 5	Add 4	Add 3	Add 2	Add 1	Add 0	R/$\overline{W}$

A read operation is a "1" and a write is a "0". These are defined as the constants `I2C_MASTER_READ` and `I2C_MASTER_WRITE` so we don't have to remember these values. If we want to read from a device at address `0x12` … then our transmitted I2C address would be `(0x12 << 1) | I2C_MASTER_READ` while if we wished to write to a device at address `0x12`, our transmitted I2C address would be `(0x12 << 1) | I2C_MASTER_READ`.

To send a command, we build the structure of that command then ask for it to be transmitted. We start by creating a command handle using `i2c_cmd_link_create()`. Next we declare that the command starts with an I2C start request by calling `i2c_master_start(cmd)`. Now we can associate the data we wish to send with that command by calling `i2c_master_write_byte()` and/or `i2c_master_write()`. Next, we indicate that the command population has been completed. Now we can actually ask the ESP32 to perform the command with a call to `i2c_master_cmd_begin()`. This API causes all the commands buffered to be transmitted. After calling `i2c_master_cmd_begin()`, we should release the command handle and create a new one for the next transmission.

Here are some examples. Imagine we want to send the byte `0x34` to the slave at address `0x12`:

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (0x12 << 1) | I2C_MASTER_WRITE, 1 /* expect ack */);
i2c_master_write_byte(cmd, 0x34, 1);
i2c_master_stop(cmd);
i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000/portTICK_PERIOD_MS);
i2c_cmd_link_delete(cmd);
```

When working with I2C, I recommend that once you have attached the devices to your I2C bus then run an I2C scanner to ensure that they are responding properly. If we think about an I2C slave devices, when the master starts a transaction and transmits the address of the slave, the slave will respond with a positive acknowledgment … assuming it is present to respond. If there is no such slave with that address, there obviously won't be such a response. As such, if we walk through each of the possible I2C addresses and start a transactions merely to see if we get a response, we can build a map of which slaves are present and which are not. An example ESP-IDF program that just does that can be found here:

https://github.com/nkolban/esp32-snippets/blob/master/i2c/scanner/i2cscanner.c

An example output might be:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

See also:

- i2c_cmd_link_create
- i2c_cmd_link_delete
- i2c_master_start
- i2c_master_stop
- i2c_master_cmd_begin

- i2c_master_write
- i2c_master_write_byte
- i2c_driver_install
- i2c_param_config

Using Arduino I2C libraries

The Arduino libraries provide an interface to I2C. At a high level we request a handle that we use to communicate as an I2C master to a connected I2C slave. We use the i2cInit() method to obtain such a handle. The handle is an instance of a pointer to an `i2c_t` piece of data.

See also:

- i2cInit
- Arduino Wire Library

Common I2C devices

The following is a partial list of some of the more popular I2C based devices.

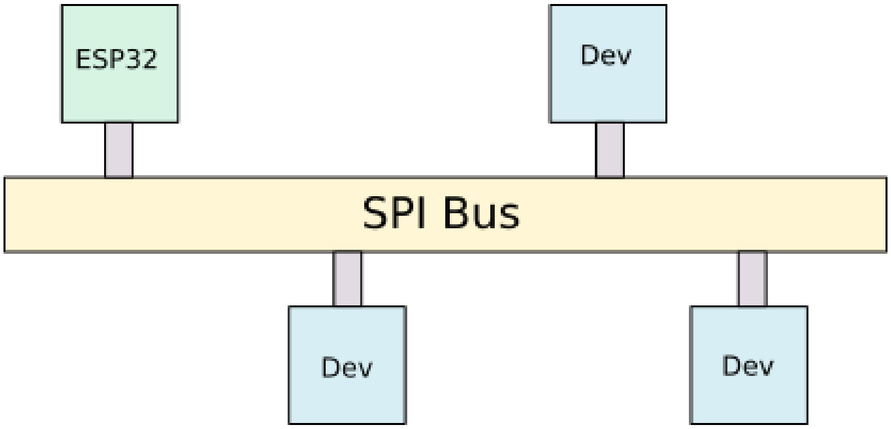
Device	Description	Address
ADS1015	Analog to Digital	
DS1307	Real Time Clock	0x68
BH1750FVI	Light level sensor	0x23
HMC5883L	Compass	0x1E
MCP4725	Digital to Analog	
MCP23017	GPIO extender	0x20-0x27
MPU6050	Accelerometer / Gyroscope	0x68
MPU6500	Accelerometer / Gyroscope	
MPU9255	Accelerometer / Gyroscope / Compass	
PCA9685	PWM controller	
PCF8574	GPIO extender	0x20-0x27
TEA5767	FM Radio	

See also:

- Compass – HMC5883L (aka GY-271) (aka CJ-M49)
- Ambient light level sensor – BH1750FVI
- MCP23017
- Accelerometer and Gyroscope – MPU-6050 (aka GY-521)
- Real time clocks

Working with SPI - Serial Peripheral Interface

SPI is a bus protocol that will allow an ESP32 to communicate with peripheral devices or ICs that comply with the SPI protocol. Logically, it looks as follows:



Within the protocol, one device (typically the ESP32) acts as the bus "master" and all the other devices act as bus "slaves". What this means is that the communication is between the master and one individual slave at a time. The slave devices don't communicate with each other. The value of a bus is that we can attach multiple devices to the ESP32 (the master) using only a few wires as opposed to having to have multiple wires out to each individual device.

Within the SPI protocol there are primarily three connections (wires) to each device.

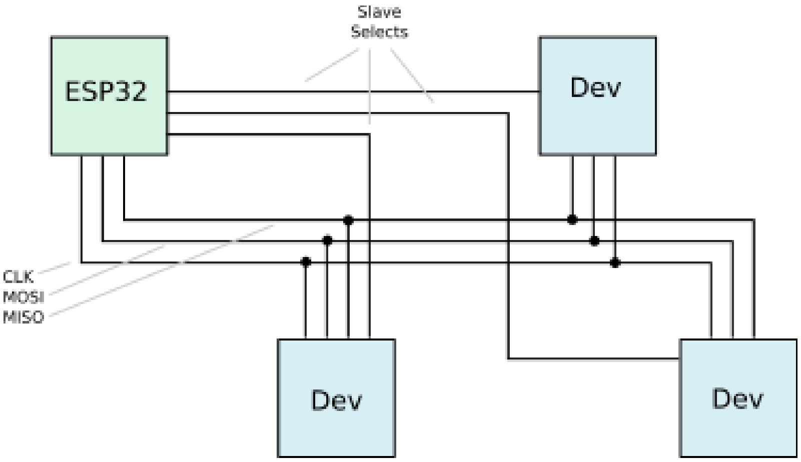
The first is called the "clock" line. This is an outbound signal from the master which is used to synchronize activities between all the participants. It is usually a regular high/low equally spaced train of signals. The frequency of the clock defines the speed of transmission across the bus. When the clock transitions from high to low or low to high, this is an indication in the SPI protocol that the slave device should either send or receive data.

The next line we will look at is a data line that is exclusively written by the master and read by the slaves. The line is called the "Master Out / Slave In" which is an excellent description of its purpose. Commonly this is abbreviated to MOSI. The master serially streams data outbound down this line and the slave receives the data. The data bits on the line are synchronized with the clock line so that the slave knows when the next bit is ready to be read.

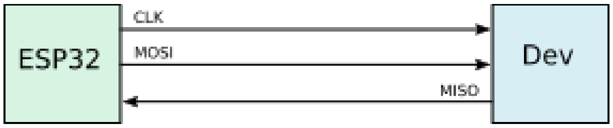
The third line is also a data line that is exclusively written by a slave and read by the master. The line is intelligently called "Master In / Slave Out" and abbreviated to

"MISO". Again, the clock line is used to indicate when the slave should transitions its output bits.

If there were only one device to which the master were communicating, these three lines would be sufficient … however that isn't much of a bus. If we have multiple devices, the obvious problem is how does one slave know that it is being talked to as opposed to some potentially distinct slave? The answer is that each slave has a unique input line called the "Slave Select". Normally the slave select has one value (eg. high) and only when the master drives the slave select to the other value (eg. low) does that slave know that it is being addressed. The rules state that a slave should only respond (receive or transmit) on the bus when its slave select line indicates that it is to do so.



While a device is active because it is selected by the master, it can simultaneously read data from the MOSI line while concurrently writing data to the master through the MISO line. Focusing on a single master/slave interaction, the story looks like:



Since this is a serial protocol and we will receive data in bytes, we need to be cognizant of whether or not data will arrive LSB first or MSB first. There will be an option to control this.

For the clock, we will be latching data and we will need to know what edges and settings are important. There will be a clock mode option to control this. In SPI there are two attributes called phase and polarity. Phase (`CPHA`) is whether we are latching data on high or low and Polarity (`CPOL`) is whether high or low means that the clock is idle.

`CPOL=0` means clock is default low, `CPOL=1` means clock is default high.

When `CPOL=0`, then the following are the values for CPHA

`CPHA=0` means data is captured on clock rising edge, `CPHA=1` means data is captured on clock falling edge.

When `CPOL=1`, then the following are the values for CPHA

`CPHA=0` means data is captured on clock falling edge, `CPHA=1` means data is captured on clock rising edge.

SPI wraps these two flags into four defined and named modes:

Mode	Clock Polarity – CPOL	Clock Phase – CPHA
`SPI_MODE0`	0 (Clock default low)	0
`SPI_MODE1`	0 (Clock default low)	1
`SPI_MODE2`	1 (Clock default high)	0
`SPI_MODE3`	1 (Clock default high)	1

The default for most purposes will be `MODE0`.

Also for the clock, what speed are we will need to know what speed the data is to be moved. There will be a clock control speed option to control this.

Using the ESP-IDF SPI driver

Unlike other devices and processors on the market, the ESP32 doesn't dictate that a fixed set of pins **must** be used for the CLK, MOSI and MISO functions. Instead the pins that you select will perform these functions. Should you be designing PCBs this opens up many more levels of flexibility as it simplifies connectivity and link routing.

Because the ESP32 doesn't limit you to just one SPI bus, the utilization of SPI on ESP32 is architected into a set of discrete steps.

1. We initialize a bus defining its nature.
2. We describe a device to the ESP32 that is attached to a configured bus. From this we get a handle that refers to that device.
3. We communicate with a device using its handle reference.

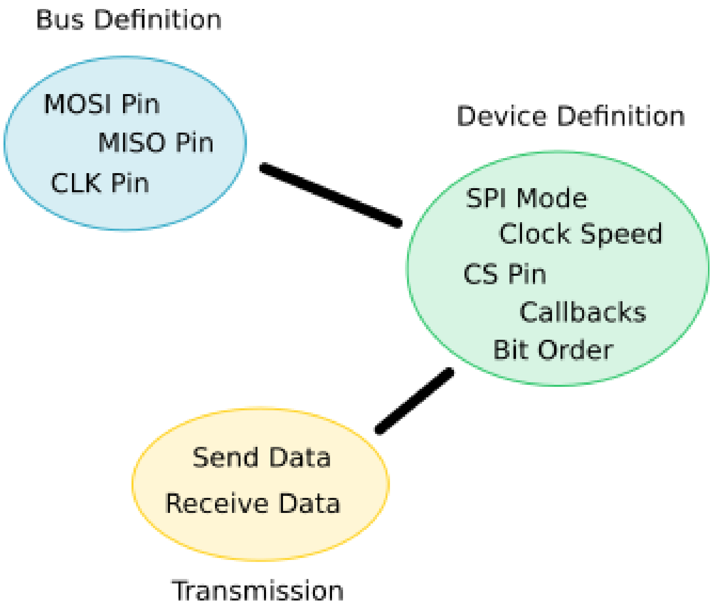
We use the notion of a transaction to describe the ESP32 interacting as a master with a slave peripheral.

We can imagine there being the following actions performed:

- A command is sent
- An address is sent
- Data is read from the slave and simultaneously data is written to the slave

The command and address actions are optional and can be omitted resulting in only the parallel send and receive of data.

The following illustrates the relationships between some of the logical components we will be working with from an SPI driver perspective. The Bus definition identifies the pins used for a bus for MOSI, MISO and CLK. Next we define a logical device and the characteristics of that device on the bus. Notice that the clock speed of the bus is an attribute of the device definition. When I first tried to understand this it confused me. Surely a bus has a clock speed that is the same for all devices? In the ESP32 we have more flexibility that ever before. The ESP32 can communicate with distinct devices using different bus speeds as a function of the currently selected device. For example, if we are talking with device "A", we may use a clock speed of 100KHz however if we are talking with device "B", we can use a different clock speed. Since only one device at a time has is active because of its slave selection, the other devices can ignore the clock rate (and other data lines) and hence speed of clock is not a consideration for a device that is not selected.



To use the SPI driver, we start by calling `spi_bus_initialize()`. Here we identify the pins on the ESP32 that are to be used for SPI functions such as CLK, MISO and MOSI.

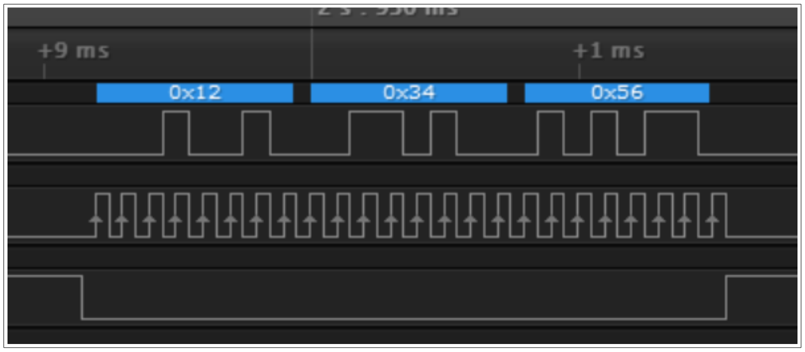
Next we call `spi_bus_add_device()` to tell the ESP32 about an external SPI device. This includes optionally nominating a pin for the slave selection (CS) as well as a clock speed for communications with that device. Calling `spi_bus_add_device()` populates a handle for us that we can subsequently use to refer to that device.

Once done, the ESP32 and external SPI device have a relationship and it is now possible for them to interact. We can either queue transactions using the call to `spi_device_queue_trans()` and subsequently querying their results with a call to `spi_device_get_trans_result()` or we can synchronously interact with the SPI device using a call to `spi_device_transmit()`.

Putting this all together, here is a sample that transmits a few bytes over a bus:

```
#include <driver/spi_master.h>
void test_spi_task(void *ignore) {
   ESP_LOGD(tag, ">> test_spi_task");

   spi_bus_config_t bus_config;
   bus_config.sclk_io_num   = clkPin; // CLK
   bus_config.mosi_io_num   = mosiPin; // MOSI
   bus_config.miso_io_num   = misoPin; // MISO
   bus_config.quadwp_io_num = -1; // Not used
   bus_config.quadhd_io_num = -1; // Not used
   ESP_LOGI(tag, "... Initializing bus.");
   ESP_ERROR_CHECK(spi_bus_initialize(HSPI_HOST, &bus_config, 1));


   spi_device_handle_t handle;
   spi_device_interface_config_t dev_config;
   dev_config.address_bits     = 0;
   dev_config.command_bits     = 0;
   dev_config.dummy_bits       = 0;
   dev_config.mode             = 0;
   dev_config.duty_cycle_pos   = 0;
   dev_config.cs_ena_posttrans = 0;
   dev_config.cs_ena_pretrans  = 0;
   dev_config.clock_speed_hz   = 10000;
   dev_config.spics_io_num     = csPin;
   dev_config.flags            = 0;
   dev_config.queue_size       = 1;
   dev_config.pre_cb           = NULL;
   dev_config.post_cb          = NULL;
   ESP_LOGI(tag, "... Adding device bus.");
   ESP_ERROR_CHECK(spi_bus_add_device(HSPI_HOST, &dev_config, &handle));


   char data[3];
   spi_transaction_t trans_desc;
   trans_desc.address   = 0;
   trans_desc.command   = 0;
   trans_desc.flags     = 0;
   trans_desc.length    = 3 * 8;
   trans_desc.rxlength  = 0;
   trans_desc.tx_buffer = data;
   trans_desc.rx_buffer = data;

   data[0] = 0x12;
   data[1] = 0x34;
   data[2] = 0x56;

   ESP_LOGI(tag, "... Transmitting.");
   ESP_ERROR_CHECK(spi_device_transmit(handle, &trans_desc));

   ESP_LOGI(tag, "... Removing device.");
   ESP_ERROR_CHECK(spi_bus_remove_device(handle));

   ESP_LOGI(tag, "... Freeing bus.");
   ESP_ERROR_CHECK(spi_bus_free(HSPI_HOST));

   ESP_LOGD(tag, "<< test_spi_task");
   vTaskDelete(NULL);
}
```

If we then examine the result in the output of a logic analyzer we find:



See also:

- spi_bus_initialize
- spi_bus_add_device
- spi_device_transmit
- spi_device_queue_trans
- spi_device_get_trans_result

The Arduino Hardware Abstraction Layer SPI

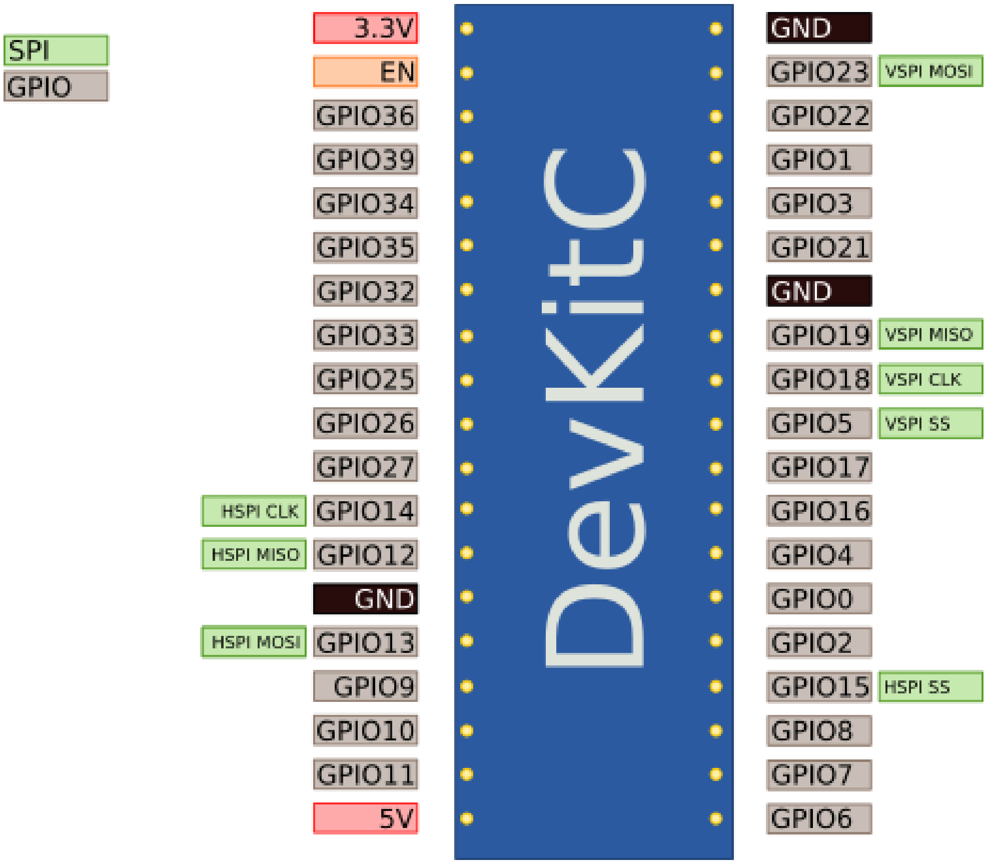
The Arduino hardware abstraction layer for SPI provides a high level API for working with SPI. On an ESP32 there are multiple SPI peripherals so we need to always identify which one we are working with. The identity of an SPI instance is contained within an "`spi_t`" instance. We get a pointer to one of these by calling "`spiStartBus`".

The pins on an ESP32 that can be used for SPI can be dynamically mapped. Remember that a single SPI instance can have up to 4 pins … these are `SCK`, `MOSI`, `MISO` and `SS`. For the three SPI interfaces supplied by ESP32, the following are suggestions with native support:

SPI	MOSI	MISO	CLK	SS
HSPI	13	12	14	15
VSPI	23	19	18	5
FSPI	8	7	6	11

Of these three, the `FSPI` bus should **not** be used for applications as it is used internally by ESP32.

The following diagram illustrates the default pin numbers for SPI for `VSPI` and `HSPI` … but please realize that these are only defaults and can be re-mapped as needed.



Once we have the SPI instance, we call `spiAttachMOSI()`, `spiAttachMISO()`, `spiAttachCLK()` and `spiAttachSS()` to set the corresponding pin mappings. Passing a pin value of -1 uses the default otherwise the explicit pin number supplied is used.

Once we have an SPI instance and attached the pins, we can start sending and receiving data.

```
spi_t *spi = spiStartBus(VSPI, 1000000, SPI_MODE0, SPI_MSBFIRST);
spiAttachSCK(spi, sckPin);
spiAttachMISO(spi, misoPin);
spiAttachMOSI(spi, mosiPin);
spiAttachSS(spi, 0, ssPin);//if you want hardware SS
spiEnableSSPins(spi, 1 << 0);//activate SS for CS0
spiSSEnable(spi);

//transfer some data
const char * data = "hello spi";
uint8_t out[strlen(data)];
spiTransferBytes(spi, (uint8_t *)data, out, strlen(data));
//out now contains the response from the slave
```

See also:

- spiStartBus
- spiAttachMISO
- spiAttachMOSI
- spiAttachSCK
- spiAttachSS

Common SPI devices

Now that we know about the existence of SPI, we might ask ourselves what kinds of peripheral devices are available for use? Here is a short list of some of the more interesting ones I have come across:

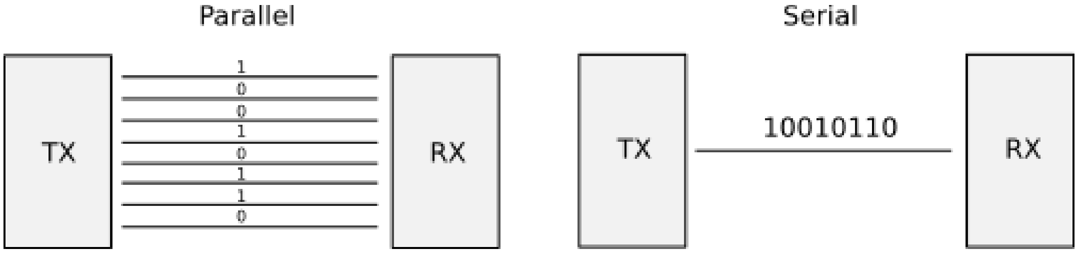
Device	**Description**
MAX7219/MAX7221	7 segment or 8x8 matrix LED controller.
MCP3208	12 bit Analog to Digital
Nokia 5110 / PCD8544	84x48 pixel LCD
SSD1306	128x64 OLED display
ST7735	128x128 display
TJCTM24024-SPI	320x240 TFT display
MFRC522	RFID
nRF24L01	Communications device

Working with UART/serial

The goal of a data bus is to move information from one place to another. In our digital electronics world, the unit of information is a bit … a value of 1 or 0. Typically we group these bits into a larger unit called a byte which a sequence 8 bits. These 8 bits can represent a numeric value from 0 to 255.

Now let us suppose we have a CPU (eg. an ESP32) and a remote device which wants a piece of data that the ESP32 has. For example, we'll keep it simple, assume it is some kind of smart light bulb where a value of 0 means dark and a value of 255 means full luminance. How do we transmit our data from the ESP32 to the device? One way would be to have 8 parallel wires between the ESP32 and the device and we would simultaneously set each of the digital values on the wires to the 8 bits of our value. That works great … but it has two draw backs. The first is that it means that we have to use thick cables carrying 8 separate discrete strands. It also limits us to only 8 bits of data. What if we want to send 24 bits of data such as the luminescence for red, green and blue?

The data bus we are considering here is known as a parallel bus because we are sending data in parallel. However in our story, we are going to talk about a serial bus.



Imagine if you will that that you can sense a remote value of either "1" or "0". Imagine you are miles away from your friend and you have a pair of binoculars. You can look through them and see that he has a flag pole whether the flag can either be all the way up or all the way down (he is so far away that you can't properly distinguish a flag in the middle of the pole and you aren't sure if it is all the way up or all the way down … so you can only see the presence of absence of the flag). Can you use that to communicate?

The answer is yes … but it takes a little work.

Let's imagine you and your friend agree on a timing rate … lets say "10 seconds". What that means is that you will look through your binoculars every 10 seconds and write down whether the flag is high or low. You will ignore any other measurements. Your friend will also know that rate and won't change the flag any quicker or any slower than once every 10 seconds (we'll assume he can magically change the flag from low to high or high to low instantly).

You both start and you read the following flag values:

- 0 – up
- 10 – up
- 20 – down
- 30 – up
- 40 – down
- 50 – up
- 60 – up
- 70 – down

This would correspond to the value 01101011. Ta da!! You have now transmitted 8 bits of data from one place to another … simply by agreeing the rate of change. This rate of change is called the "baud rate" and is how many bits per second can be transmitted. In our story we transmitted 1 bit every 10 seconds … and hence had a baud rate of 0.1. In the electronics world where we can sense at extremely high speeds, typical baud rates are 9600, 4800, 19200, 38400, 57600 and115200. As we see, we can send data very quickly. The primary requirement is that the baud rate used between a sender and a receiver be agreed upon. Our story won't work if they think that they are transmitting at different rates.

The next part of our story is knowing when we are done. We could keep looking through our binoculars and writing down the flag position every 10 seconds … but how much data is our friend sending? This is where the bit count comes into the story. Again, we agree how many bits constitute a unit of transmission. Typically this is 8 bits corresponding to a byte but the protocol allows us to agree on a transmission of 5, 6, 7, 8 or 9 bits per transmission.

If you are following the story, there is another puzzle we have to consider … and that is when do we start recording data? If I am going to sample the data once every period, when does that period start? The answer to that one is that every transmission of a unit of data starts with the signal high and when it goes low, that indicates that we have agreed that will be the start of the transmission period. Since we have the start indicated from a transition from high to low, then at the end of the last transmission, the signal must be left high … and that's where a stop bit comes into play. An additional but of data is added at the end which drives the line high.

There are three hardware supported serial interfaces on the ESP32 known as UART0, UART1 and UART2. Like all peripherals, the pins for the UARTs can be logically mapped to any of the available pins on the ESP32. However, the UARTs can also have direct access which marginally improves performance. The pin mapping table for this hardware assistance is as follows

UART	RX IO	TX IO	CTS	RTS
UART0	GPIO3	GPIO1	N/A	N/A
UART1	GPIO9	GPIO10	GPIO6	GPIO11
UART2	GPIO16	GPIO17	GPIO8	GPIO7

Having said that, the UART drivers that I recommend to use don't have this level of optimization built into them and as a result, you are pretty much free to use any pins you choose.

There is low level hardware serial support and higher level driver support. I recommend the driver level be used wherever possible.

The driver APIs are included through the inclusion of "`driver/uart.h`".

We start by populating a `uart_config_t` structure instance. This provides the core settings for a UART we want to use. An example might be:

```
uart_config_t myUartConfig;
myUartConfig.baud_rate = UART_BITRATE_115200;
myUartConfig.data_bits = UART_DATA_8_BITS;
myUartConfig.parity = UART_PARITY_DISABLE;
myUartConfig.stop_bits = UART_STOP_BITS_1;
myUartConfig.flow_ctrl = UART_HW_FLOWCTRL_DISABLE;
myUartConfig.rx_flow_ctrl_thresh = 120;
```

Once populated, we can set the parameter of the UART with a call to `uart_param_config()`.

Next we define the pins we wish to associate with our UART using `uart_set_pin()`. For example:

```
uart_set_pin(uartNum, 21, 22, UART_PIN_NO_CHANGE,  UART_PIN_NO_CHANGE);
```

We can now initialize a driver using:

```
uart_driver_install(uartNum, 2048, 0, 10, NULL, 0);
```

One of the options we can specify when initializing a driver is to supply a FreeRTOS queue handle. If we supply this, then events that are detected by the UART are then posted onto the queue. We can have tasks that are blocked watching the queue ready to process incoming events when they arrive. This allows us to perform UART data processing asynchronously. If we don't want to use a queue, we specify NULL for the queue parameter.

To write data to the UART we use the `uart_write_bytes()` function.

See also:

- UART driver API
- UART low level APIs
- uart_param_config
- uart_driver_install
- uart_write_bytes
- Wikipedia – UART
- Serial Communication

Using the VFS component with serial

The ESP32 virtual file system (VFS) component provides an interface to UART. To use this, we initialize with a call to `esp_vfs_dev_uart_register()`. Following this we can perform file I/O operations against the VFS file system at the following paths:

- `/dev/uart/0` – UART0
- `/dev/uart/1` – UART1
- `/dev/uart/2` – UART2

Currently we can open, close, read and write to the device.

See also:

- esp_vfs_dev_uart_register

I2S Bus

The Inter-IC Sound (I2S) bus is a serial link protocol specifically for sound data. The protocol utilizes three lines. These are:

- `SCK` – Serial Clock line.
- `WS` – Word Select line. Indicates the channel being transmitted. 0 – channel 1 (left), 1 – channel 2 (right).
- `SD` – Serial Data line.

See also:

- I2S APIs

- I2S bus specification
- Wikipedia: I2S

I2S – Camera

The I2S component is also capable of performing additional functions beyond the basic I2S. I'm not sure if this type of function is related to an I2S standards … it may be that the Espressif designers needed a place to "host" this function and the I2S component seemed the closest.

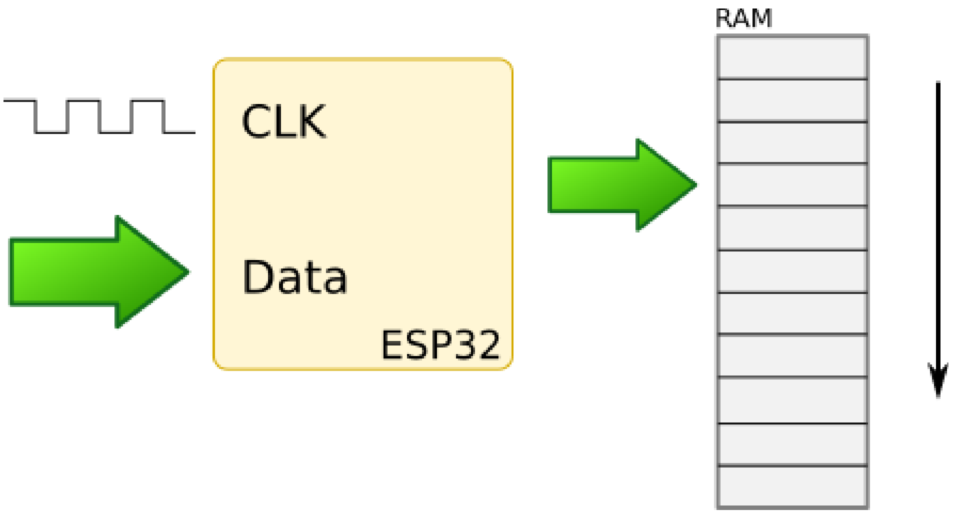
When we think of a video camera, we should consider that it has a number of concepts that are common to most camera. First there are the concepts of horizontal and vertical sync signals. These are signals that indicate when a complete scan line has been received (or a new one is about to be received) and also an indication when a complete "image" of information has been received (or is about to be received). Then there is the data itself. If we think of an image as being composed of pixels and each pixel being represented by 16 bits of information, then we quickly see that there is a lot of information to be received. For example, if a row in an image is 320 pixels and each pixel is 16 bits then one row is 5120 bits. If an image is 240 rows then a complete image would be 1228800 bits (153,600 bytes). That's a lot of data. If we used a serial bus that could read at 400Kbps, that would be about 3 images (frames) per second. That's not great. One solution would be to use a faster serial bus … but that may not be possible. The solution in the ESP32 is to use a wider data bus. A camera such as the OV7670 exposes an 8 bit data bus allowing us to transfer 8 bits at a time. This would give us a factor of 8 speed improvement or about 24 images per second which is usually fast enough.

The I2S support for cameras allows us to grab these 8 bits of bus data and using direct memory access, stuff them directly in RAM with minimal expense.

I2S – LCD

I2S – DMA

Direct Memory Access is the ability to receive parallel incoming data over a parallel data bus and place it directly in RAM without intervention of application CPU instructions.



A clock signal is presented to the ESP32 along with up to 16 bits of incoming data from a bus. The pins for both the clock and the data are configurable. When the clock transitions, this is an indication that a new piece of data is available and it is automatically read by the ESP32. Upon reading the parallel data bits, the ESP32 places the read data in RAM memory at the next location it is configured to write to and then increments that pointer ready to begin again with the next byte cycle.

The I2S peripheral is the component that provides DMA support. There is quite a lot of setup involved in this so buckle up.

The I2S DMA support is technically provided for inbound (to ESP32) camera data but can be used for non video purposes. Video data requires a high bandwidth and a serial protocol would not be sufficient. Camera modules typically have an 8 bit (or more) data bus over which the data is received. The data is considered "valid" on the bus under the following conditions:

- The VSYNC signal from the camera is high.
- The HSYNC signal from the camera is high.
- The HREF signal from the camera is high.
- The PCLK signal from the camera is high.

Since we just mentioned that the I2S DMA is primarily there for camera support we have to take this into account when considering DMA by itself outside the context of cameras. Specifically, we want to set the VSYNC, HSYNC and HREF signals to be permanently high and use the PCLK signal as the clocking in of new data on the bus.

In the ESP32, pins are multiplexed meaning that almost any pin can serve almost any purpose. This means we can map physical pins to their logical mappings. For our discussion on DMA, we have the following logical pins:

I2S0I_DATA_IN0_IDX to I2S0I_DATA_IN15_IDX	16 bits of input data
I2S0I_V_SYNC_IDX	VSYNC logical pin
I2S0I_H_SYNC_IDX	HSYNC logical pin
I2S0I_H_ENABLE_IDX	HREF logical pin
I2S0I_WS_IN_IDX	PCLK logical pin

We can use the ESP-IDF function called `gpio_matrix_in()` to map from physical pins to their logical counterparts. There are two "special" but "logical" pins known as `0x30` and `0x38`. Specifying a pin of `0x30` means a constant low value while a pin of `0x38` means a constant high value.

By experimentation:

When we ask to read samples and we have enough buffer space to hold all the samples in one buffer, we get an interrupt with:

IN_DONE=1, IN_DSCR_EMPTY=0, IN_DSCR_ERR=0, IN_ERR_EOF=0, IN_SUC_EOF=1

and the descriptor holds

size=4092, length=2000, offset=0, sosf=1, eof=1

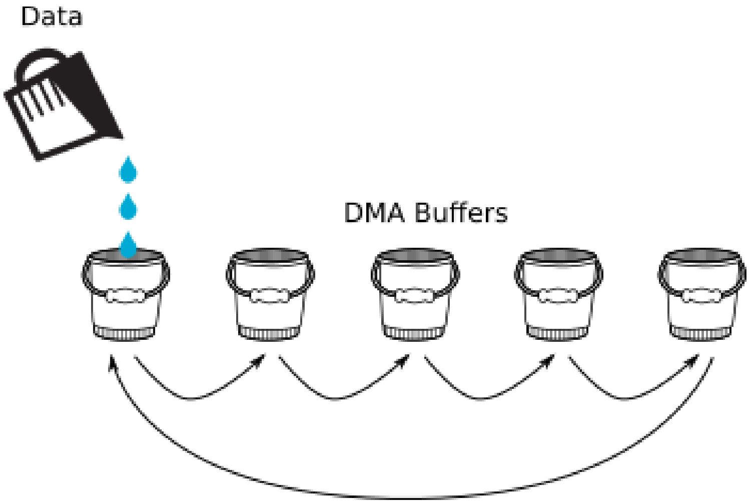
When we ask to read more samples than will fit in one buffer, we get an interrupt with

IN_DONE=1, IN_DSCR_EMPTY=0, IN_DSCR_ERR=0, IN_ERR_EOF=0, IN_SUC_EOF=0

and the descriptor holds

size=4092, length=4092, offset=0, sosf=1, eof=0

Now we get to introduce the concept of a DMA buffer. The notion here is that DMA wants to fill a memory buffer with data asynchronously retrieved from a data bus. However, if all we are doing is stuffing the received data into RAM without processing it, that is going to be of limited use. Ideally we want to not only receive data but process it as it arrives. If we think about the alternative which would be to receive all the data we need and then process it, we might find that we run out of RAM space. For example, if we are processing an incoming video stream and are wanting to stream it out over the network, we will usually want to stream what we have received during the last short interval while at the same time accumulating the next interval. To that end we have a series of DMA buffers that are chained together in list. Pictorially, what we then have is the following:



Data is arriving at a steady rate and a DMA buffer starts to fill. When the buffer fills, two things happen. First we receive an interrupt telling us that a DMA buffer is now full and ready for consumption. In addition (and instantaneously) newly arriving data starts to fill the next buffer with no lost or missed data (in the analogy, we didn't spill any). We can now start to consume the previously filled buffer without having to worry about what is happening with the other buffers. When the next buffer fills, a new interrupt will be generated.

It takes some thought to see the benefits of this story but now we can start to reason about capacity and throughput. Let us imagine the simplest story which consists of two buffers that we call BufferA and BufferB. Initially, BufferA starts to fill with data. When BufferA fills, DMA switches to start filling BufferB and informs us via interrupt that BufferA is full. We can now process the data in BufferA while concurrently, BufferB is being filled with new data. When BufferB is full, DMA will have cycled around and will now start filling BufferA again. For this story to work, we must be able to process a buffer's worth of data in less time than it takes a buffer to be filled by DMA. Specifically, if we are told that we can start processing BufferA, we had better have consumed all the data we need from it quicker than DMA can fill BufferB because when BufferB has filled, DMA will expect BufferA to be empty again.

If we can't process a buffer's worth of data faster than a buffer is filled, we probably want to re-think our design or implementation … at a minimum, we will then find that we need to maintain as many buffers as necessary to hold all the data we are expecting to receive. Another strength of buffers is the notion that we may be able to get away with processing buffer data if the *average* time to process a buffer is less than it takes to fill a buffer. Here we will likely want to have more than two buffers. On average each time we fill a buffer, by the time the next one is also filled, we will have consumed the original data. However, if our processing of data is *laggy* or has other non-constant time processing characteristics, then as long as we have sufficient buffers, we don't have to have consumed a buffers worth of data by the time the next one fills as long as on average we have a sufficient processing rate.

Enough with the theory. Let us now consider the DMA implementation as found in the ESP32. First we start with the data type called "`lldesc_t`". This cryptically named type is a "linked list descriptor". Loosely, you can think of this as the buffer. The structure contains a set of fields that (among others) are:

- `buf` – A pointer to a chunk of storage that will contain DMA received data.
- `size` – The size (in bytes) of the storage pointed to by buf.
- `length` – The number of bytes actually written by DMA into the buffer.
- *`next`* – The pointer to the next instance of an "`lldesc_t`" structure. This will be the next buffer to be filled after this one has been filled. If this points to a previous "`lldesc_t`" then we will have formed a circular chain. (Note: The field is not actually called *next* but I wanted to keep this clean for our discussion … there are more implementation details that are not pertinent to the story).

Hopefully, you are following the story so far … unfortunately though, things about to get much more complex.

First, let us think about the implementation of `lldesc_t`. For reasons unknown to me, the implementation has sized both `size` and `length` fields to be a maximum of 12 bits in length. This means that the size of a buffer can be between 0 and 4095 bytes (2^12-1 = 4095 = 0b1111 1111 1111). Take this into account and ensure you configure your `lldesc_t` instances correctly.

The DMA implementation does **not** write a byte at a time into the buffer. It wants to be much more efficient than that and writing individual bytes as they are received wouldn't give us the maximum rate we desire. The ESP32, being a 32 bit device, is able to move 32bits of data at a time. As such, the ESP32 writes a full 32 bits each time DMA writes into the buffer. If we think about this a moment, we will find that the length of data used in the buffer will thus be a multiple of 4 (32 bits = 4 bytes). This then further says that the maximum buffer size can only be 4092 bytes. Why? Well 4092 is evenly divisible by 4 and the next multiple up would be 4096 which is more than 12 bits in length and hence is too large to hold in a 12 bit number.

Next let us think about the unit of DMA read data. DMA can read up to 16 parallel bits of information at a time. But since DMA writes 32 bits of data to the buffer as a unit, it must thus read two 16 bit values for each buffer write.

For many applications of the ESP32 we may not need to read a data bus that is 16 bits wide. We may only need to work with a bus that is 8 bits wide. Unfortunately, the

ESP32 DMA has no accommodation for this notion. We can tell the ESP32 to not bother with the upper 8 pins of the 16 input pins and simply assume them to be 0, however the ESP32 wishes to process 16 bit values only.

See also:

RMT – The Remote Peripheral

The primary purpose of this component of the ESP32 is to generate pulses sent via an infrared LED. When you point your TV remote control at your TV and press a button, an infrared LED at the front of your controller sends a sequence of pulses that are received by the TV and decoded. Since an infrared LED can be either "on" or "off" at any given time, the signal is encoded in the duration that the LED is either on or off.

A data item in the RMT represents one bit of information and is composed of a 16 bit value.

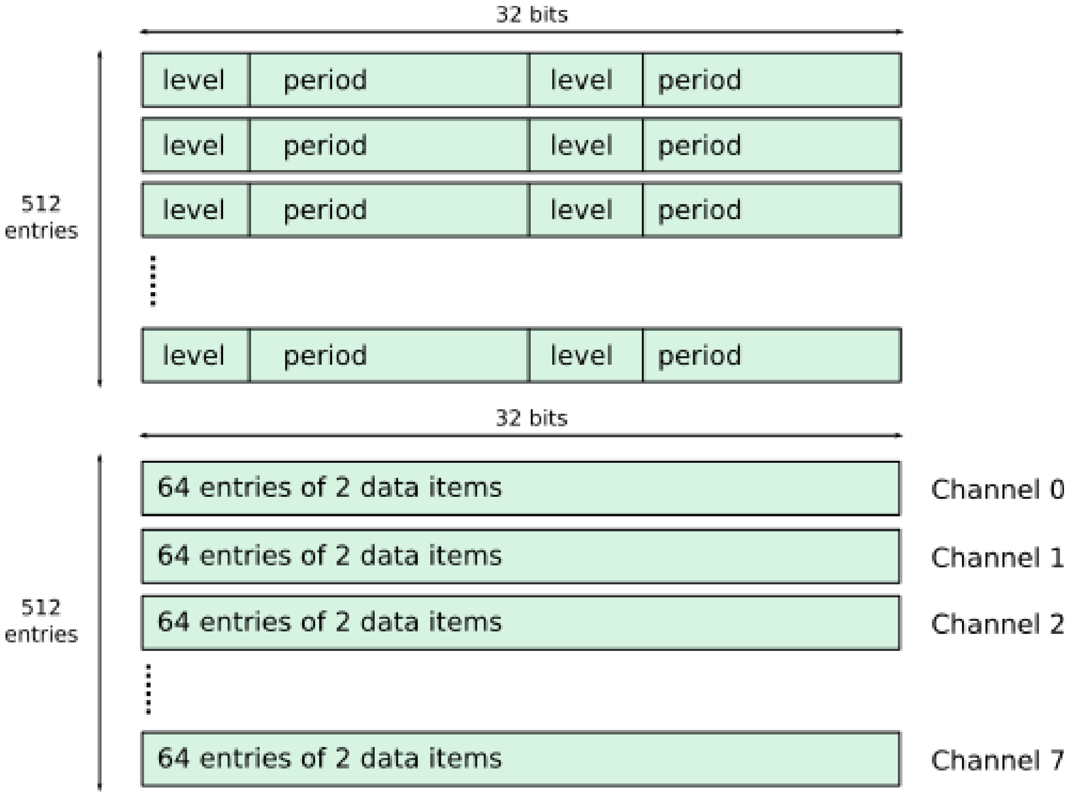
level [15]	period [14:0]

The level bit is either 1 or 0 describing the output signal while the period (15 bits in length) is the duration in clock ticks for which that level will be sent. 15 bits give us a range from 1-32767. A value of 0 for the period is used as an end marker. The normal maximum number of bit records is 128 (or 256 bytes of record data).

The RMT has 8 distinct channels with each channel having 128 16 bit records. Should we wish to send more than 128 records, we can extend a channel to use the channel data of subsequent adjacent channels. For example, if we are using only channel 0 then the data available for it can be that of channel 0, channel 1 up to channel 7 giving us a total of 1024 value records (1024 data items which means 1024 bits of data or 2048 bytes of data items).

There is also an interesting mechanism that involves the buffer for a channel wrapping around. Imagine that we have a default buffer size of 128 * 16bit records. If we need to send more than 128 records, we can pre-load a buffer with our first 128 values and set the RMT transmitting. When it has transmitted the first 128 values, it will generate an interrupt that can be used to re-fill the buffer with the new values and progress will continue.

The hardware allocates 512 instances of 32 Bit records (512 * 32/8 = 2048 bytes) which can be read and written by the RMT hardware in the CPU. Each 32 Bit record contains two data items. Each channel has one block of 64 instances of 32 bit records associated with it which gives us 128 data items of wave form (1 transition is encoded in 16 bits).



Should we need a transmission that is greater than 128 data items, we can consume the adjacent block which will give us a further 128 data items transitions. We can only do this by using channel block for the next higher adjacent channel. We can actually specify that we want to use ALL the channels for a single transmission giving us a maximum transmit bit transition of 8 x 128 = 1024. Note that there is no wrap around. So channel 0 can use the blocks of channel 0 through channel 7, channel 1 can use the blocks of channel 1 through channel 7 … and channel 7 can only use its own memory block.

To use the driver, one typically follows this recipe:

- Complete an `rmt_config_t` structure.
- Call `rmt_config()` passing in the `rmt_config_t` structure.
- Call `rmt_driver_install()` to install the driver.
- Build an array of records to send.

- Call `rmt_write_items()` to send the stream.

Since RMT is all about setting signals for interval durations, there is a lot to be discussed about time. First we must discuss the base clock. By default this runs at 80MHz. That means it ticks 80,000,000 times a second or 80,000 times a millisecond or 80 times a microsecond or 0.08 times a nano second. Flipping this around, our granularity of interval is 1/80,000,000 is 0.0000000125 seconds or 0.0000125 milliseconds or 0.0125 microseconds or 12.5 nanoseconds. I hope we agree, that's pretty fast.

Since these are pretty small durations, we can "logically" slow our clock down for timing purposes. The base clock speed is 80MHz but we can provide a divider which is an integer value against which the base clock speed is divided to provide a new tick rate that is used as the interval value for timings. The integer value is 8 bits so we can divide between 1 and 255. At 255 we have are down to about 314KHz. Since the base speed is 80, it seems to make sense to divide by 8, 80 or 160, giving rates of 10MHz, 1MHz and 500KHz.

The period range of a cycle is 15 bits giving us a value between 1 and 2^15 (32768). For example, if we divide the base clock by 80 then the granularity unit becomes 1 microsecond so if we delay for 1000 units, we end up at 1msec.

When there is no current transmission, we can set the idle level to be enabled and either high or low. This sets the output pin to this idle value when not in use.

Here is a sample RMT application:

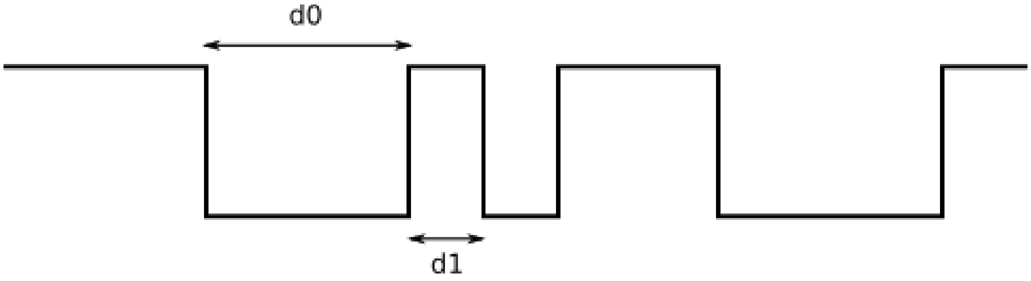
```
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_log.h>
#include <driver/rmt.h>
#include "sdkconfig.h"

static char tag[] = "rmt_tests";
static void dumpStatus(rmt_channel_t channel) {
   bool loop_en;
   uint8_t div_cnt;
   uint8_t memNum;
   bool lowPowerMode;
   rmt_mem_owner_t owner;
   uint16_t idleThreshold;
   uint32_t status;
   rmt_source_clk_t srcClk;

   rmt_get_tx_loop_mode(channel, &loop_en);
   rmt_get_clk_div(channel, &div_cnt);
   rmt_get_mem_block_num(channel, &memNum);
   rmt_get_mem_pd(channel, &lowPowerMode);
   rmt_get_memory_owner(channel, &owner);
   rmt_get_rx_idle_thresh(channel, &idleThreshold);
   rmt_get_status(channel, &status);
   rmt_get_source_clk(channel, &srcClk);

   ESP_LOGD(tag, "Status for RMT channel %d", channel);
   ESP_LOGD(tag, "- Loop enabled: %d", loop_en);
   ESP_LOGD(tag, "- Clock divisor: %d", div_cnt);
   ESP_LOGD(tag, "- Number of memory blocks: %d", memNum);
   ESP_LOGD(tag, "- Low power mode: %d", lowPowerMode);
   ESP_LOGD(tag, "- Memory owner: %s", owner==RMT_MEM_OWNER_TX?"TX":"RX");
   ESP_LOGD(tag, "- Idle threshold: %d", idleThreshold);
   ESP_LOGD(tag, "- Status: %d", status);
   ESP_LOGD(tag, "- Source clock: %s", srcClk==RMT_BASECLK_APB?"APB (80MHz)":"1MHz");
}

void runRmtTest() {
   ESP_LOGD(tag, ">> runRmtTest");

   rmt_config_t config;
   config.rmt_mode = RMT_MODE_TX;
   config.channel = RMT_CHANNEL_0;
   config.gpio_num = 21;
   config.mem_block_num = 1;
   config.tx_config.loop_en = 0;
   config.tx_config.carrier_en = 0;
   config.tx_config.idle_output_en = 1;
   config.tx_config.idle_level = 0;
   config.tx_config.carrier_duty_percent = 50;
   config.tx_config.carrier_freq_hz = 10000;
   config.tx_config.carrier_level = 1;
   config.clk_div = 80;

   ESP_ERROR_CHECK(rmt_config(&config));
   ESP_ERROR_CHECK(rmt_driver_install(config.channel, 0, 0));
   dumpStatus(config.channel);

   rmt_item32_t items[3];
   items[0].duration0 = 10000;
   items[0].level0 = 1;
   items[0].duration1 = 10000;
   items[0].level1 = 0;

   items[1].duration0 = 10000;
   items[1].level0 = 1;
   items[1].duration1 = 5000;
   items[1].level1 = 0;

   items[2].duration0 = 0;
   items[2].level0 = 1;
   items[2].duration1 = 0;
   items[2].level1 = 0;

   while(1) {
   ESP_ERROR_CHECK(rmt_write_items(config.channel, items,
         3, /* Number of items */
         1 /* wait till done */));
      vTaskDelay(1000/portTICK_PERIOD_MS);
   }
   ESP_LOGD(tag, "<< runRmtTest");
}
```

So far we have discussed emitting a waveform of signals but there is a powerful second capability of the RMT and this to be able to capture received waveforms that might be arriving.

To use the RMT in receiver mode, we call `rmt_config()` specifying that the `rmt_mode` of the given channel should be `RMT_MODE_RX`. Note that a channel can be only one of a receiver or transmitter at any given time. Once a channel is configured to be a receiver, it will start watching the input and be considered to be in its idle state. When it starts to see transitions, it will begin recording their values and durations into RAM. It will continue to do this until it no longer sees transitions for a period of time known as the idle threshold at which time it will return to its idle state and write a termination record into the output data. A termination record is simply a data item that has a duration of 0. As an indicator that a signal train has been received, when it returns to idle state, an `RX_END` interrupt is generated. There is also the ability to perform simple noise filtering. Records of a duration less than a configurable threshold can be automatically removed.

To obtain the received data, we need to become familiar with the ring buffer functions. These are supplied as part of the RTOS set of functions. Through the RMT APIs, we can call `rmt_get_ringbuf_handler()` which returns an instance of `RingbufHandle_t` which is a reference to a ring buffer who's buffer size is defined when we made the call to `rmt_driver_install()`. Through this `RingbufHandle_t` we can then call `xRingbufferReceive()` to get (with optional blocking) an array of `rmt_item32_t` records that contains the received data. Once we are done with this, we call `vRingbufferReturnItem()` to release the storage.

When working with RMT, measurement of times is crucial, after all that's the core of the story. When we consider the data structure representing a value is 16 bits long with 1 bit used for the signal value, that leaves us 15 bits per measurement to encode the value. The maximum value a 15 bit integer can have is 32767 (2^15 – 1). Let us put this in perspective when looking at the following signal trace:



If we focus on just the start, we see that the signal was low for a duration of `d0` and then high for a duration of `d1`. We measure duration in seconds and, of course we can perform simple arithmetic to scale into milliseconds or microseconds with just multiplication factors.

Now let us think of the duration returned by RMT as a 15 bit value. That duration is **not** a direct measurement of time. It is actually a count of clock ticks. If you think of a wall clock with a second hand, each tick represents one second … and hence one tick indicates one second has passed. The clock rate of our ESP32 is 80MHz (80 million ticks per second). That means that 1 tick is 1/80,000,000 of a second or 12.5 nano seconds. Note that is **nano** seconds and not milli or micro seconds. So if 1 tick is 12.5 nano seconds, at 80MHz, the maximum duration that we can measure would be 15 bits worth of data or 65535 * 12.5 ns = 819us. And in there is the problem. 819us is still less than a millisecond and that is not much time at all. Many of the measurements we care about might be far more than this … measurements in the 10s of milliseconds are not uncommon. If the maximum measurement we can take is 819us what is our solution?

One way would be to increase the number of bits available in the recorded data but there is a better solution. We can "logically" use a slower clock. Imagine if the clock didn't tick at 80MHz but instead ticked at 800KHz … a factor of 100 slower. This would mean that 1 tick would be 1/800,000 of a second or 1.25 micro seconds. A measurement of 65535 tick would now have a duration of 81.9 milliseconds … and that is more usable. The downside of this is that we have reduced the granularity of a tick. If we are measuring changes in signal which change faster than once every 1.25 microseconds, then we will miss some of the changes. However, for the vast majority of purposes, we won't need that level of granularity. When we configure the RMT peripheral, we can supply a "clock divider" which is an 8 bit value that allows us to specify the divisor for our clock.

Imagine our clock divisor is `CLK_DIV`. The number of ticks in 10 micro seconds then becomes:

```
TICK_10_US = 80,000,000 / CLK_DIV / 100,000
```

If we then are told by RMT that COUNT ticks have occurred, that would be:

```
COUNT * 10 / TICK_10_US microseconds
```

See also:

- Remote Control Peripheral – RMT
- rmt_config
- rmt_driver_install
- rmt_write_items

- rmt_wait_tx_done
- xRingbufferReceive
- Ring buffer withing FreeRTOS
- Chapter 7 of ESP32 Technical Reference Manual – Remote Control Peripheral

Timers and time

The ESP32 leverages the FreeRTOS environment which provides the concept of a clock tick. This isn't a real-clock (as in a CPU clock) but is instead a logical timer period defined by the FreeRTOS. Currently, the default tick is one millisecond but this can be changed in the make menuconfig options (but I wouldn't recommend it). There is a constant called `portTICK_PERIOD_MS` that defines the duration (in milliseconds) of a FreeRTOS clock tick.

Within our code, we may wish to delay for a period of time. We can use the `vTaskDelay()` function. This function takes as input the number of FreeRTOS ticks to sleep. We can convert from time to ticks using:

```
timeValInMillisecs/portTICK_PERIOD_MS
```

for example:

```
vTaskDelay(timeValInMillisecs/portTICK_PERIOD_MS)
```

Another aspect of working with time is time calculations and measurement. The function `system_get_time()` returns a 32 bit unsigned integer (`unit32_t)` value which is the microseconds since the device booted. This value will roll over after 71 minutes.

Note: `system_get_time()` is deprecated, use `gettimeofday()` instead.

When we use `gettimeoday()` we get back a `struct timeval` which contains two fields:

- `tv_sec` – The time in seconds
- `tv_usec` – The time in micro seconds

The reason that a structure is returned is that it might not be possible to accommodate this value in a single number, especially on a 32bit system such as the ESP32. We may need to perform arithmetic on these values … for example to measure the duration of some activity, we might get the time before and the time after and subtract one from another. A library of simple routines is provided here:

https://github.com/nkolban/esp32-snippets/tree/master/c-utils

called "`c_timeutils.c`" and "`c_timeutils.h`". The functions supplied include:

- `struct timeval timeval_add(struct timeval *a, struct timeval *b)` – Add two struct timeval structures together resulting in the sum.
- `void timeval_addMsecs(struct timeval *a, uint32_t msecs)` – Add a specified number of milliseconds to the timeval.
- `uint32_t timeval_durationBeforeNow(struct timeval *a)` – Return the number of milliseconds since the past time.
- `uint32_t timeval_durationFromNow(struct timeval *a)` – Return the number of milliseconds until the future time.
- `struct timeval timeval_sub(struct timeval *a, struct timeval *b)` – Subtract one time vale from another.
- `uint32_t timeval_toMsecs(struct timeval *a)` – return the number of milliseconds represented by the timeval.

Another useful function is one supplied by FreeRTOS called xTaskGetTickCount(). This returns the number of clock ticks that have elapsed since FreeRTOS was started. We can use this to measure elapsed time between distinct points.

What if we need a granularity of time smaller than a microsecond? Hopefully you won't need this often … however some solutions such as working with the WS2812 LEDs do in fact require ultra fine precision.

One possible solution is to drop down to assembly language programming. There is a special register managed by the ESP32 which is called "`ccount`" which measures cycles of operation. The value of this is incremented each time an operational cycle completes.

The value of this register can be retrieved with the following C code:

```
static inline uint32_t getCycleCount() {
  uint32_t ccount;
  __asm__ __volatile__("rsr %0,ccount":"=a" (ccount));
  return ccount;
}
```

This fragment uses the in-line assembler to transform an assembly language statement into its corresponding operational instruction.

A specialized function called `xthal_get_ccount()` provides a similar function.

Note: The following is **disabled** in EPS32 at this time.

Another mechanism to suspend execution is a call to `nanosleep()`. This takes as input a `struct timespec` that includes seconds and nanoseconds. Remember 1 second = 1000 milliseconds = 1000000 microseconds = 1000000000 nanoseconds.

If we need additional timer functions, the ESP32 provides four high resolution timers that are divided into two groups of two.

For the most part, when we want our applications to perform tasks at certain times or block for configurable amounts of time, the use of the FreeRTOS timers will be the best solution.

After having looked at internal timing, let us now look at real-world or "wall clock" timing. The ESP-IDF provides a rich assortment of POSIX based APIs for working with wall clock times. There are two primary representations of a wall clock time those are "`time_t`" which is basically a large integer. The value is the number of seconds that have elapsed since the point of time known as the "epoch" which is midnight on the 1st of January 1970 in GMT (near London England). Given a time_t value, we can calculate the date and time. Thankfully, we don't have to worry about that complex arithmetic as library functions are supplied for us.

The second data type we will consider is a structure called "`struct tm`". This structure contains fields for all the useful date and time values such as the hour, minute and second as well as year, month and day (and a few more). When we start to think about wall clock time, a new consideration comes into play. If I look at my clock and it says 3:51pm I know what time it is. However, that is my "local" time sitting in Texas. If I called a friend in London, he would say its 9:51pm. The reason for this is that the world is divided into distinct time zones and the declaration of any given time is relative to the time zone. If I tell my wife that the movie starts at 12:30pm, there is usually no ambiguity as we are "local" to each other. However, if a friend in London sends me a directory in a ZIP file and I look inside that directory, what time should I see the files as having been modified? If he modified a file and immediately sent it to me, he would say "I changed the file at 9:51pm" … is that what I should "see" when I look at the file? The answer is no, I should see that he modified the file at "3:51pm" which would be my equivalent local time.

Within an ESP32 environment, we can set an environment variable using the `setenv()` API. The variable is called `TZ` and is used to define the local time zone in which the ESP32 is operating. From a time zone string, the ESP32 can then perform the additions or subtractions to determine the local time. Note that this is only possible if we record the time stamp of some data relative to an agreed convention … and this is where Universal Coordinated Time (UTC) comes into the picture. UTC is the GMT time zone and is the value we get back from a call to time().

Let us break down the available functions supplied by ESP-IDF for time manipulation:

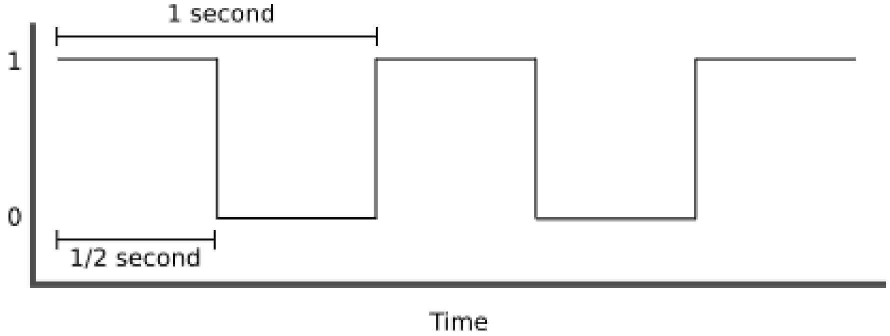
- Get the current time as a `time_t` – `time()`
- Convert a `time_t` to a `struct tm` ignoring timezone – `gmtime()`
- Convert a `time_t` to a `struct tm` using the timezone – `localtime()`
- Convert a `struct tm` to a `time_t` – `mktime()`
- Convert a `struct tm` into a string – `asctime()`
- Convert a `time_t` into a string – `ctime()`. Basically a convenience function for `asctime(gmtime())`.

See also:

- Working with SNTP
- system_get_time
- Hardware Timers
- gettimeofday
- Timers in FreeRTOS
- asctime
- ctime
- gmtime
- localtime
- mktime
- settimeofday
- time
- times
- tzset
- vTaskDelay

LEDC - Pulse Width Modulation - PWM

The idea behind pulse width modulation is that we can think of regular pulses of output signals as encoding information in the duration 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.



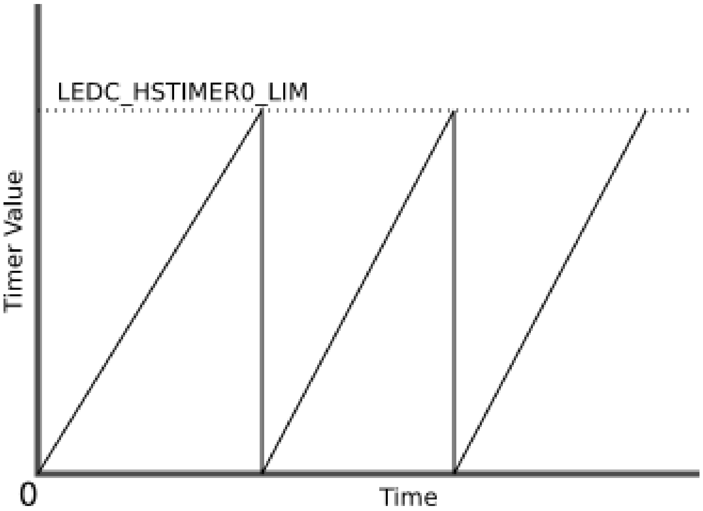
This repeats on into the future. The duration that the pulse is high relative to the period as a whole allows us to encode an analog value onto digital signals. If the pulse is 100% high for the period then the encoded value would be 1.0. If the pulse is 100% low for the period, then the encoded value 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. Internet articles suggest a frequency of between 300 and 1000 Hz are good values for LED dimming.

Now let us look at how the ESP32 provides PWM support. Within the ESP32 there is hardware support for PWM supplied in a component called the "LEDC/PWM". It supplies 8 channels of output each of which can be controlled independently from the others. The ESP32 PWM functions are powerful but at first they can sound complex. The reality is that they actually aren't that hard and the power and flexibility within them is worth the learning. One just has to slow down and contemplate them for a bit.

First, understand that there are four distinct timers. Think of an individual timer as "ticking along" counting up until it reaches a maximum number and then resets itself to zero and the story repeats. The time between resets (i.e. how long it takes to count to the maximum value) is the frequency value measured in Hz (things per second). For example, if we specified a frequency of 1Hz, it would take 1 second to count from 0 to the maximum value and then repeat. If we specified a frequency of 1000Hz (1000 things per second), it would only take 1 millisecond to count from 0 to the maximum value.

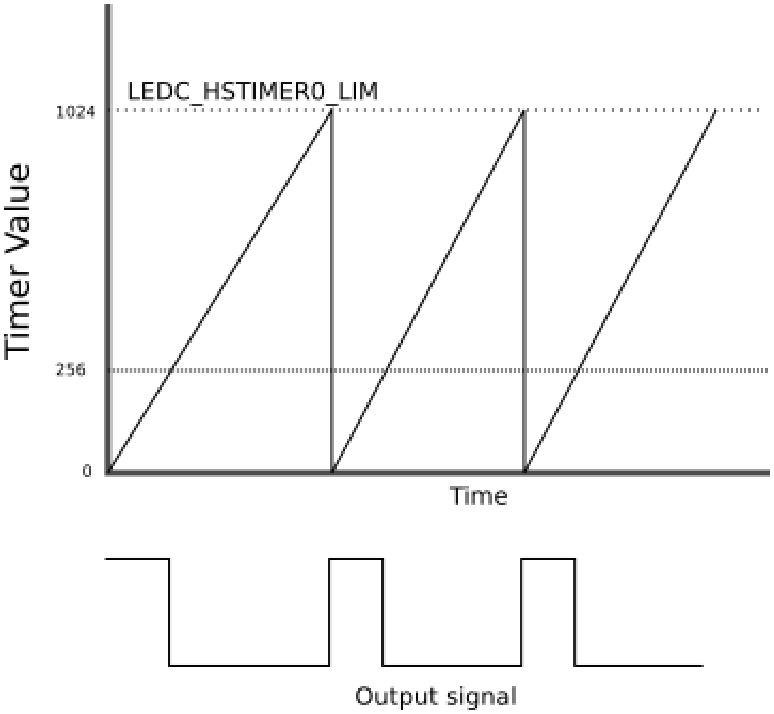


So what is this "maximum" value that I have been talking about? Each timer counts upwards and we can define how many "bits" in a number before it resets. Our choices when configuring a timer are 10, 11, 12, 13, 14 or 15 bits. These correspond to maximum values of 1023, 2047, 4095, 8191, 16383 and 32767. Pause here and think about the relationship between frequency and bit size of the counter. If we set our timer to have a frequency of 1Hz and set our bit size to be 12 bits, then it will take 1 second for our timer to count from 0 to 4095. If our bit size were 15 bits, it would take 1 second for our timer to count from 0 to 32767. It is important to realize that the frequency is how long it takes to count from 0 to the maximum number. If we have higher bit sized maximums, then we just have more "timer increments" … but in the same frequency specified period of time. What we thus have is more "granularity" in the timings.

Now that we have looked at the timer, lets turn our attention to the notion of a channel. A channel be thought of as the PWM output signal. When we define a channel, we specify which of the GPIO pins the signal will appear upon. We also specify which of the four timers we plan on associating with the channel. Since a timer defines a frequency, this defines the frequency (or period) of our PWM signal. Finally, and arguably most importantly, we define the duty cycle of the PWM output. This is the time duration within a period that the PWM output signal will be high before it goes low. The value is supplied as a number of timer ticks and **not** a period of time and this is where folks can get lost.

Think about the timer. It starts at 0 and counts upwards to its maximum value (as defined by its bit size) and then resets. The rate of counting upwards is specified by the frequency defined on the timer. If the frequency is 1Hz, then the timer takes 1 second to count from 0 to the maximum. The duty cycle value specifies the timer value after which the PWM output signal will flip from high to low.

For example, imagine we wanted a PWM signal with a period of 1 second that is high for ¼ of a second with a 10 bit granularity. We would set the frequency of the timer to be 1 Hz and the granularity of the timer to be 10 bits. This then says it takes 1 second for the timer to count from 0 to 1024. If we want the duty cycle of the PWM to be ¼ of a second, then we want to switch the output signal ¼ of the way through the timer count which would be 1024 * ¼ = 256.



To use the LEDC/PWM functions we would follow this recipe:

1. Set a LEDC timer. We do this by calling `ledc_timer_config()` passing in a `ledc_timer_config_t` structure pointer.

```
ledc_timer_config_t timer_conf;
timer_conf.bit_num    = LEDC_TIMER_12_BIT;
timer_conf.freq_hz    = 1000;
timer_conf.speed_mode = LEDC_HIGH_SPEED_MODE;
timer_conf.timer_num  = LEDC_TIMER_0;
ledc_timer_config(&timer_conf);
```

2. Set a LEDC channel.

```
ledc_channel_config_t ledc_conf;
ledc_conf.channel    = LEDC_CHANNEL_0;
ledc_conf.duty       = 1024;
ledc_conf.gpio_num   = 16;
ledc_conf.intr_type  = LEDC_INTR_DISABLE;
ledc_conf.speed_mode = LEDC_HIGH_SPEED_MODE;
ledc_conf.timer_sel  = LEDC_TIMER_0;
ledc_channel_config(&ledc_conf);
```

At a low level, the frequency is given by

$$f=\frac{Clock\,Speed}{Divisor\times precision}$$

So Divisor is

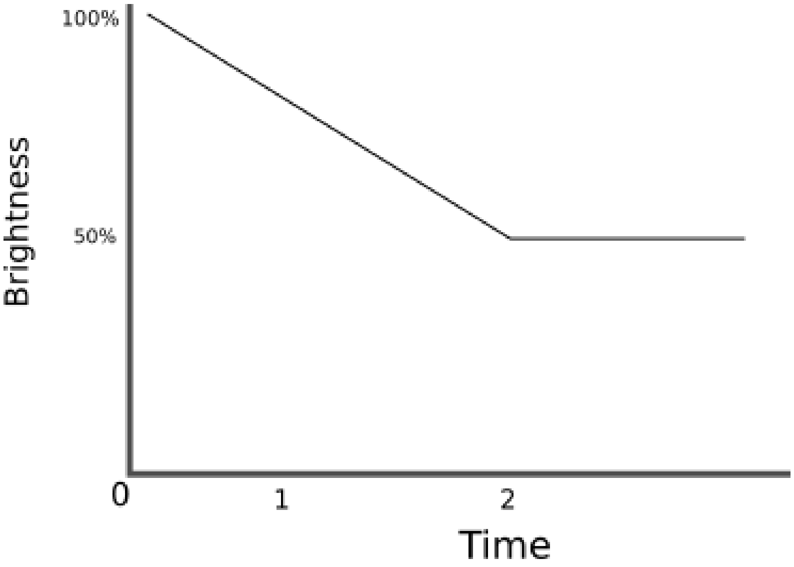
$$Divisor=\frac{Clock\,Speed}{f\times precision}$$

See also:

- Wikipedia: Pulse-width modulation
- ledc_channel_config
- ledc_timer_config
- Servos

Automated PWM fading

The hardware support for PWM in the ESP32 also includes an intriguing feature that relates to "fading". If we consider the duty cycle of PWM, we can think of it as the percentage of frequency that the signal is high vs low. So a 100% duty cycle will be constantly high and a 0% duty cycle constantly low with values in between giving us an "average" output over time. This can be used to "dim" LEDs. Now imagine that we wanted to change the brightness of the LED over time. Effectively we would be "fading" the brightness. For example, if we wanted to have the LED go from 100% brightness to 50% brightness over 2 seconds, the brightness output might look as follows:



Since the value of the duty cycle corresponds to the brightness of the LED, then what we are talking about here is reducing the duty cycle value over time. Within the hardware of the ESP32 PWM, we can specify an initial duty cycle value and the amount that the duty cycle should be reduced by each configurable timer interval.

Analog to digital conversion

Analog to digital conversion is the ability to read a voltage level found on a pin between 0 and some maximum value and convert that analog value into a digital representation. Varying the voltage applied to the pin will change the value read. The ESP32 has an analog to digital converter built into it with a resolution of up to 12 bits which is 4096 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 4095 and voltage ranges between these will produce a correspondingly scaled digital value.

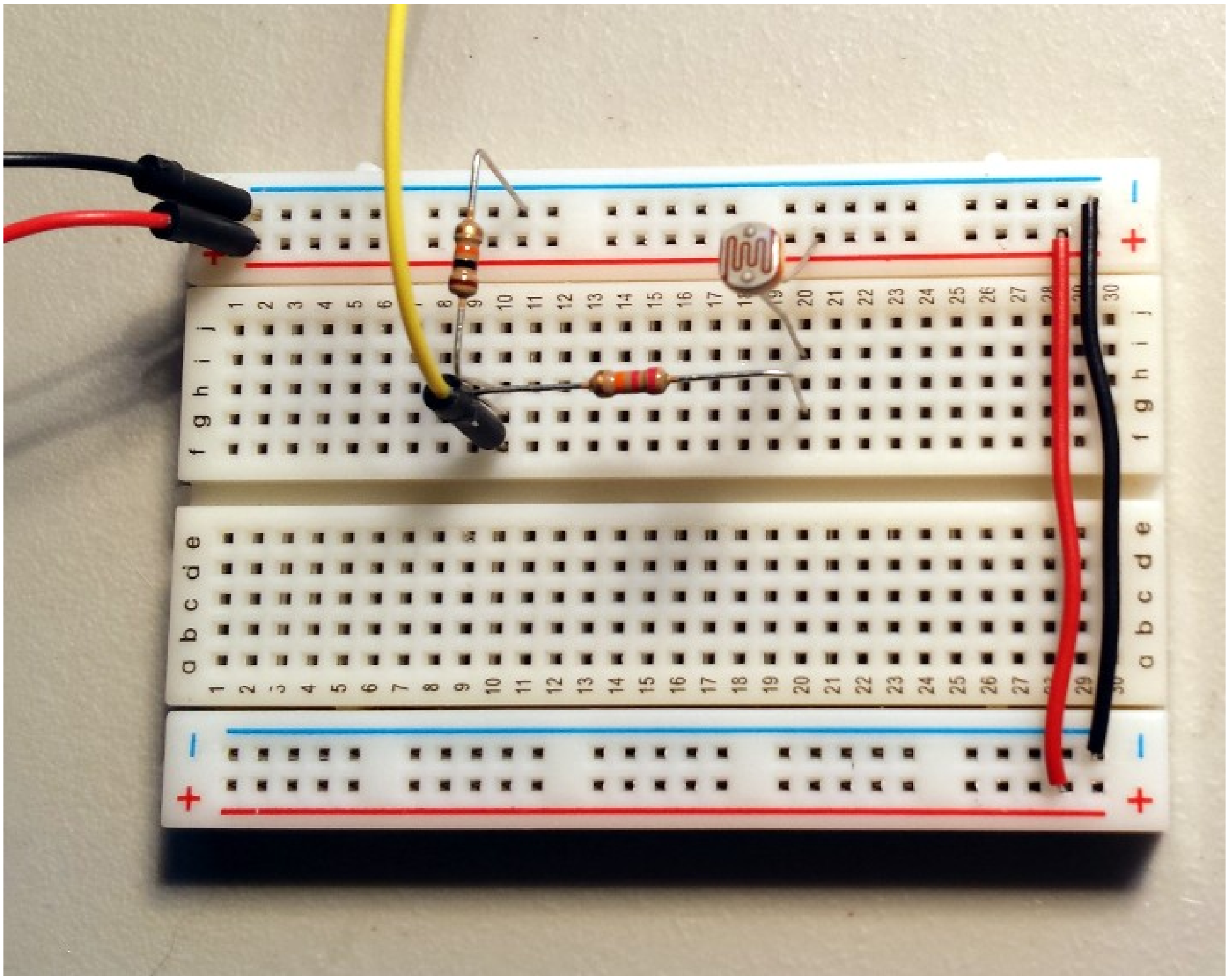
One of the properties on the analog to digital converter channels is attenuation. This is a voltage scaling factor. Normally the input range is 0-1V but with different attenuations we can scale the input voltage into this range. The available scales beyond the 0-1V include 0-1.34V, 0-2V and 0-3.6V.

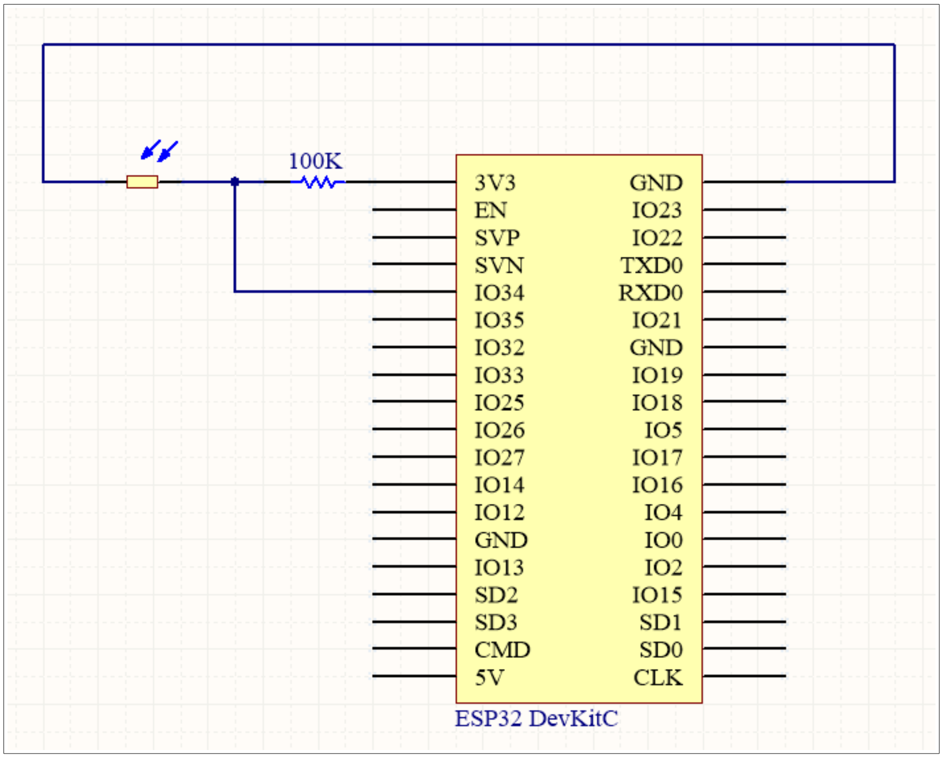
Here is an example application using the APIs. What this example does is print the value read from the ADC every second.

```
#include <driver/adc.h>
#include <esp_log.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include "sdkconfig.h"

static char tag[] = "adc1";
```

```
void task_adc1(void *ignore) {
   ESP_LOGD(tag, ">> adc1");
   adc1_config_width(ADC_WIDTH_12Bit);
   adc1_config_channel_atten(ADC1_CHANNEL_6, ADC_ATTEN_6db);
   while(1) {
      int value = adc1_get_voltage(ADC1_CHANNEL_0);
      ESP_LOGD(tag, "value: %d", value);
      vTaskDelay(2000/portTICK_PERIOD_MS);
   }
   vTaskDelete(NULL);
}
```

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?

GPIO	ADC Channel
GPIO 0	ADC2_CH1
GPIO 2	ADC2_CH2
GPIO 4	ADC2_CH0
GPIO 12	ADC2_CH5
GPIO 13	ADC2_CH4
GPIO 14	ADC2_CH6
GPIO 15	ADC2_CH3
GPIO 25	ADC2_CH8
GPIO 26	ADC2_CH9
GPIO 27	ADC2_CH7
GPIO 32	ADC1_CH4
GPIO 33	ADC1_CH5
GPIO 34	ADC1_CH6
GPIO 35	ADC1_CH7
GPIO 36	ADC1_CH0
GPIO 37	ADC1_CH1
GPIO 38	ADC1_CH2
GPIO 39	ADC1_CH3

Note that only a subset of ADC pins and functions are exposed. First, the supplied drivers expose only ADC1. The board layout of the ESP32-DevKitC only exposes some of the pins. Specifically, the following are exposed: `ADC1_CH0`, `ADC1_CH3`, `ADC1_CH4`, `ADC1_CH5`, `ADC1_CH6` and `ADC1_CH7`.

See also:

- adc1_config_channel_atten
- adc1_config_width
- adc1_get_voltage
- Wikipedia: Voltage divider
- YouTube: Analog to Digital with ESP32
- YouTube: Electronics Basics #27: ADC

Sleep modes

If the ESP32 device is constantly on, then it is constantly consuming current. If the power source is unlimited, then this need not necessarily be an issue however when running on batteries or other finite supply, we may need to minimize consumption. One way to achieve this is to suspend the operation of the device when not in use. When the device is suspended, the notion is that consumption will be reduced. There are three defined sleep modes. These are called modem-sleep, light-sleep, deep-sleep.

By looking at the following table we can get a sense of the abilities in each of these three modes:

Function	**Modem**	**Light**	**Deep**
WiFi	off	off	off
System Clock	on	off	off
Real Time Clock	on	on	on
CPU	on	pending	off
Current consumption	15mA	0.5mA	20µA

The modem-sleep can only be used when the ESP32 is in station mode connected to an access point. The application of this mode is when the ESP32 needs to still perform work but minimizes the amount of wireless transmissions.

The light-sleep mode is the same as modem-sleep but in this case the clocks will be suspended.

In deep-sleep mode, the device is really asleep. Neither CPU nor WiFi activities take place. The device is to all intents and purposes off … with one exception … it can wake up at a specified regular interval.

To enter deep sleep mode, we can call `system_deep_sleep()`. This can be supplied with a suspension time. The device will go to sleep and after the interval has elapsed, the device will wake up again. In addition to having a timer, we can also awake from a deep sleep by toggling the value of a signal on a pin.

Write about ...

esp_deep_sleep_pd_config

esp_deep_sleep_enable_ext0_wakeup

esp_deep_sleep_enable_ext1_wakeup

esp_deep_sleep_start

esp_deep_sleep_enable_timer_wakeup

esp_deep_sleep

RTC_DATA_ATTR – Specify memory is RTC memory

How do we determine WHY we woke up?

See also:

- YouTube: ESP32 #18 – Deep sleep API and how to wake up

Security

The ESP32 has the ability to store the password used to connect to the access point in memory. This means that if one were to physically compromise the device (i.e. steal it) then they could, in principle, dump the flash memory and retrieve your password. You could choose not to cache the password in the clear in flash but instead have your applications "decode" an encoded version that is saved in the flash memory … this would prevent an obvious retrieval through a simple memory grab. The encoding scheme could be a simple XOR against a magic number (either hard-coded or your own MAC address).

Working with flash memory

Flash memory provides a non-volatile repository of information that survives a power cycle of the device.

Data contained within flash is stored in units of sectors which are 4096 bytes in size. To write data we can call `spi_flash_write`. To read data we call `spi_flash_read`.

Since writing to flash is performed in units of 4096 bytes, we can not change a single byte by just over-writing it, instead we must retrieve the whole sector, erase the sector and then write back the sector with the changed content. This can take some time to complete and because of this, we may find that a failure is more likely to occur (eg. a loss of power). If a failure occurs after we have erased a sector or during the re-write of the sector, it should immediately become apparent that we will result in an overall corruption of data.

Data reads and writes have to be 4 bytes aligned within flash.

The ESP32 has to be instructed about the size of the flash memory available to it. Attempting to use flash memory addresses that differ from the expected size of flash memory available can result in unexpected results.

When using `esptool.py`, the `--flash_size` flag can be supplied. For esptool-ck, the corresponding flag is `-bz`.

The memory map for flash is flexible. A 1 MByte flash has an address range of

0x00 0000 - 0x0F FFFF

The bootloader loads at `0x00 1000`.

The partition table loads at `0x00 8000`.

NVS loads at `0x00 9000` to `0x00 EFFF` which is 24K.

ESP32 data lives at `0x00 F000` to `0x00 FFFF` which is 4K.

The single app loads starting at `0x01 0000`. On a typical 4MByte flash space, this would then end at `0x3F FFFF`.

Address	Description
0x00 1000	Bootloader
0x00 ????	Unknown
0x00 8000	Partition table
0x00 ????	Unknown
0x00 9000 0x00 EFFF	NVS
0x00 F000 0x00 FFFF	ESP32 data
0x01 0000 …	App

See also:

- esptool.py
- spi_flash_erase_sector
- spi_flash_read
- Cesanta: ESP8266, flash and alignment

Working with RAM memory

We can allocate memory using `malloc()` or `zalloc()`. The first function allocates and returns memory and the second does exactly the same but zeros the memory before returning. When your application logic no longer needs the memory, it can return it back to the heap with `free()`. To determine how much heap size is available, we can call `esp_get_free_heap_size()`. Once we have the memory pointer to allocated storage, we can start to manipulate it through a series of memory commands. The `memset()` command will set a block of memory to a specific value. The `memcpy()` will copy a block of memory to a different block. The `bzero()` function will set the values of a block of memory to zero.

Memory on the ESP32 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 ESP32 needs to read an instruction from memory in order to execute it, that instruction can come from one of two places. The instruction can be 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 ESP32 there is less RAM than there is flash. What this means is that you are far more likely to run out of RAM 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 ESP32s 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 short 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 "`.irom0.text`" section of code.

Note: From a raw technical perspective, `ICACHE_FLASH_ATTR` is a `#define` that maps to:

```
__attribute__((section(".irom0.text")))
```

The mapping of RAM is

0x3FFB 0000	Data Ram 256K
0x3FFF 0000	
0x4008 0000	Instruction Ram 128K
0x400A 0000	

See also:

- esptool.py
- gen_appbin.py
- Wikipedia – Data segment

RAM Utilization

As we write applications using the ESP-IDF, we will likely perform repeated tasks such as creating sockets, creating FreeRTOS tasks and other items. Since anytime we create something that has state, we will be consuming RAM, the question arises of "how much RAM"? The amount of RAM used for common tasks governs just how many instances of that task we can perform without relinquishing previously used memory.

An easy way to determine the current amount of free memory is to use the `esp_get_free_heap_size()` function. This returns the number of free bytes of RAM on the heap. As we use more RAM, the value returned by this function decreases. If we wish to measure how much RAM a particular function might use, one way would be to measure the free RAM before the call and then measure it again after the call and the difference is some indication of how much might be used.

As a base line, after we start an ESP32 application and connect to an access point, I find that the amount of RAM reported back is 191452. In these notes, I'll round that to the nearest 1000 bytes and refer to this as "K" … for example 191K (not to be confused with common "K" meaning 1024).

Let us start by looking at creating a FreeRTOS task. We commonly use `xTaskCreatePinnedToCore()` for that purpose. The 3rd parameter to that function is the stack that is allocated for the new task. This is carved up front from the heap when the task is created. As you can immediately see, creating a stack far larger than needed will immediately waste RAM.

Next, lets look at static RAM usage. This is storage data that is read/write but is declared as static in C application. For example,

```
static char buffer[10*1024];
void myFunc() {
   …
}
```

In the above, buffer is readable and writeable … but does NOT reduce the heap size.

However not declaring as static, DOES reduce the heap size:

```
char buffer[10*1024];
void myFunc() {
   …
}
```

What about the following:

```
void myFunc() {
   char buffer[10*1024];
   …
}
```

In this case, the storage is taken from that of the FreeRTOS task. So the global heap does not decrease but the amount of free space of the FreeRTOS task will diminish.

In C, when we define storage with the "const" modifier, the storage is allocated in flash (.text) as opposed to RAM.

For example:

```
const uint8_t data[100]
```

will be allocated in flash

while:

```
uint8_t data[100]
```

will be allocated in RAM.

Turning to sockets based APIs. The act of creating a listening socket seems to only cost ~700 bytes. Accepting a new connection which itself creates a new socket again only appears to cost about 700 bytes.

- esp_get_free_heap_size

Using PSRAM

Espressif not only manufactures the ESP32 but they also make a Pseudo SRAM device called the ESP-PSRAM32. This is a 32MBit PSRAM device that supports serial and quad parallel interfaces. 32MBits provides 4 MBytes.

To net this out, by attaching one of these devices to the ESP32, we can expand its apparent RAM availability from 512K to 4 MBytes and above.

The page size of the devices is 1K.

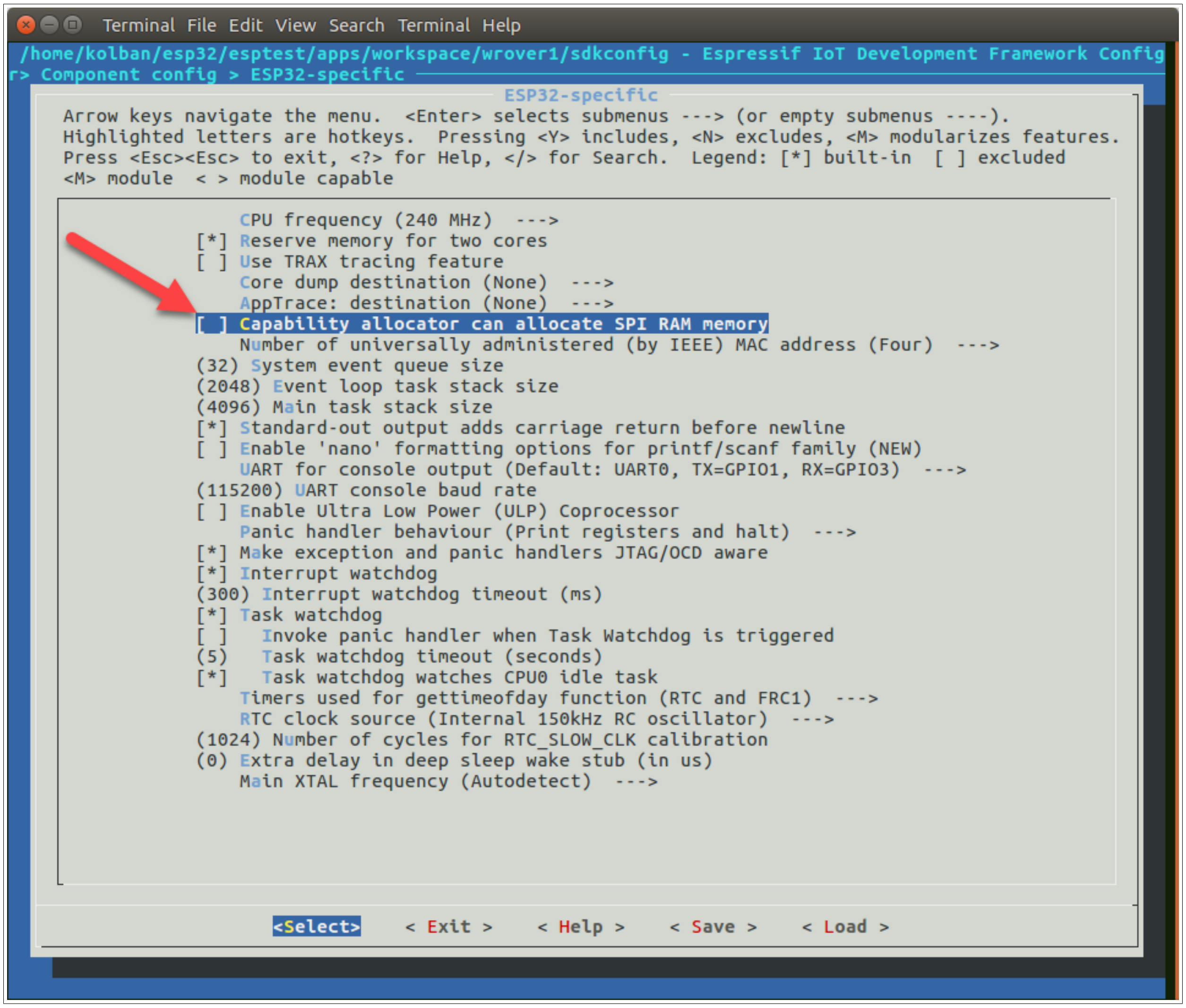
Since this devices is a serial/quad bus addition to the ESP32 and not architected directly into the ESP32 address bus standard, software running on the ESP32 must provide the drivers and controllers necessary for it to work.

At the time of writing (2016/06) these have **not** been built into the *standard* ESP-IDF nor *standard* tool chain. Special variants of these must be used. These can be separately download and installed. Specifically:

tool-chain:

ESP-IDF:

Applications that will run on the ESP32 and access the PSRAM need to be build against these versions. In addition, SPI RAM support must be explicitly enable in `make menuconfig` through the option "`Capability allocator can allocate SPI RAM memory`".



```
/home/kolban/esp32/esptest/apps/workspace/wrover1/sdkconfig - Espressif IoT Development Framework Config
r> Component config > ESP32-specific ──────────────────────────────────────────────────────────────────
                                        ESP32-specific
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.
  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
  <M> module  < > module capable

                CPU frequency (240 MHz)  --->
            [*] Reserve memory for two cores
            [ ] Use TRAX tracing feature
                Core dump destination (None)  --->
                AppTrace: destination (None)  --->
            [ ] Capability allocator can allocate SPI RAM memory
                Number of universally administered (by IEEE) MAC address (Four)  --->
            (32) System event queue size
            (2048) Event loop task stack size
            (4096) Main task stack size
            [*] Standard-out output adds carriage return before newline
            [ ] Enable 'nano' formatting options for printf/scanf family (NEW)
                UART for console output (Default: UART0, TX=GPIO1, RX=GPIO3)  --->
            (115200) UART console baud rate
            [ ] Enable Ultra Low Power (ULP) Coprocessor
                Panic handler behaviour (Print registers and halt)  --->
            [*] Make exception and panic handlers JTAG/OCD aware
            [*] Interrupt watchdog
            (300) Interrupt watchdog timeout (ms)
            [*] Task watchdog
            [ ]   Invoke panic handler when Task Watchdog is triggered
            (5)   Task watchdog timeout (seconds)
            [*]   Task watchdog watches CPU0 idle task
                Timers used for gettimeofday function (RTC and FRC1)  --->
                RTC clock source (Internal 150kHz RC oscillator)  --->
            (1024) Number of cycles for RTC_SLOW_CLK calibration
            (0) Extra delay in deep sleep wake stub (in us)
                Main XTAL frequency (Autodetect)  --->

                 <Select>    < Exit >    < Help >    < Save >    < Load >
```

These settings can be found under the ESP32-specific section.

There are additional options that become active once this is checked:

- Enable workaround for bug in SPI RAM cache for Rev1 ESP32s.
- Heavy-handed workaround for bug: Always do memory barrier
- Debug: Test workaound by generating a lot of interrupts
- Type of dual-core PSRAM caching strategy
    - Even/Odd

    - Low/High
- Type of SPI RAM chip in use
- Initialize PSRAM memory but do not add to heap allocator
- malloc() can also allocate in SPI SRAM
- ON SPI RAM init, do a quick memory test.
- Always put malloc()s smaller than this this size, in bytes, in internal RAM
- Reserve a region of memory for allocations that need to be in internal memory or DMA'able memory.

See also:

- ESP-PSRAM32 Datasheet

EFUSE

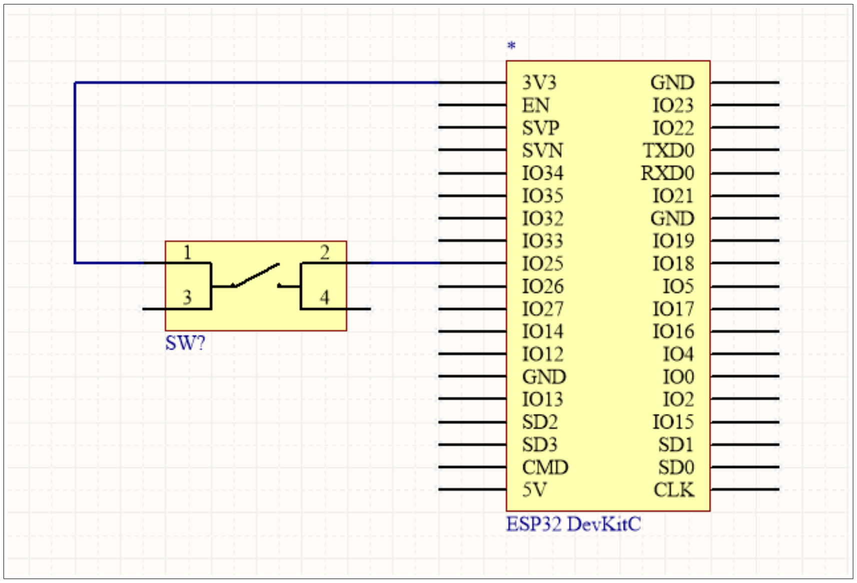
Think of the classic (old?) household fuse. When too much current passes through it, the filament wire inside gets too hot and melts through thus breaking the circuit and electricity stops flowing. If we assign the flow of electricity the value "1" and the inability to flow electricity the value "0", we have a simple boolean value that we can detect. The concept of an EFUSE in the ESP32 is a set of bits that can be electrically set **one time**. Imagine they have a default value of "1", we could select 8 bits and for some of them, pass enough current through them to "blow the fuse". This doesn't harm the ESP32 … its what these EFUSEs were designed to do. As a result, we now have 8 bits of data some of which are "1"s and some of which are "0"s … in effect … a byte of data that we set to a fixed value in the hardware. Realize that setting an EFUSE to 0 is a one time operation, you can't subsequently change it back to 1 … however it does afford us a useful capability. We can imagine ESP32s that have electrically "hard coded" values pre-assigned at manufacturing or distribution time. These could be device identifiers or keys for encryption. Because the values are burned into the device, they can't subsequently be altered or tampered with.

The ESP32 has 1024 bits of EFUSE of which 256 are reserved for the of Espressif for purposes such as the network MAC addresses and chip configuration. However, this still leaves 768 distinct bits at your disposal.

For the tinkerer, there is unlikely to be a compelling reason to use EFUSEs, but for the production customer, this is yet another feature you can choose to leverage.

Button press detection

Button press detection is one of the simplest circuits there is. A schematic may look as follows:



What we see here is that when the button is pressed, the GPIO 25 goes high. When the button is not pressed, there is an open input to GPIO 25. As such, we want to flag GPIO 25 as having a pull-down associated with it so that it has a default signal of low.

Now we can define an interrupt handler to GPIO 25 that will be called when GPIO 25 transitions from low to high. Unfortunately, this is not enough. When a button is pressed, it usually "bounces" for a very small period. This will be interpreted as a series of transitions and hence a series of interrupt handler events. What we want to do is introduce a debounce handler. One way to achieve this is not to trigger our processing when the signal transitions from low to high but to trigger our processing when the signal transitions from low to high **and** the last transition from low to high was at least 50-100 msecs ago. When bounces happen, they are usually much shorter than those time periods and can be eliminated.

Here is an example application which illustrates a GPIO interrupt handler:

```
#include <driver/gpio.h>
#include <esp_log.h>
#include <freertos/FreeRTOS.h>
#include <freertos/queue.h>
#include <freertos/task.h>

#include "c_timeutils.h"
#include "sdkconfig.h"

static char tag[] = "test_intr";
static QueueHandle_t q1;

#define TEST_GPIO (25)

static void handler(void *args) {
   gpio_num_t gpio;
   gpio = TEST_GPIO;
   xQueueSendToBackFromISR(q1, &gpio, NULL);
}

void test1_task(void *ignore) {
   struct timeval lastPress;
   ESP_LOGD(tag, ">> test1_task");
   gettimeofday(&lastPress, NULL);
   gpio_num_t gpio;
   q1 = xQueueCreate(10, sizeof(gpio_num_t));
   gpio_config_t gpioConfig;
   gpioConfig.pin_bit_mask = GPIO_SEL_25;
   gpioConfig.mode = GPIO_MODE_INPUT;
   gpioConfig.pull_up_en = GPIO_PULLUP_DISABLE;
   gpioConfig.pull_down_en = GPIO_PULLDOWN_ENABLE;
   gpioConfig.intr_type = GPIO_INTR_POSEDGE;
   gpio_config(&gpioConfig);

   gpio_install_isr_service(0);

   gpio_isr_handler_add(TEST_GPIO, handler, NULL);

   while(1) {
      ESP_LOGD(tag, "Waiting on queue");
      BaseType_t rc = xQueueReceive(q1, &gpio, portMAX_DELAY);
      ESP_LOGD(tag, "Woke from queue wait: %d", rc);
      struct timeval now;
      gettimeofday(&now, NULL);
      if (timeval_durationBeforeNow(&lastPress) > 100) {
         ESP_LOGD(tag, "Registered a click");
      }
      lastPress = now;
   }
   vTaskDelete(NULL);
}
```

GPS

The world is a sphere (hopefully that isn't news to you). We can specify a point on the Earth through a coordinate system known as latitude (lat) and longitude (lng). Given a lat/lng coordinate, we know where we are. The Global Positioning System (GPS) is a technology that allows a GPS receiver to determine its own lat/lng coordinates.

Given a pair of coordinates, for example one coordinate that says where a device is and a second coordinate saying (for example) where we want to eventually be, we can calculate some interesting data such as how far away are we from the target and what compass bearing we need to head to get there.

There are many applications of GPS and it has entered the popular domain. Most cell phones now have in-built GPS receivers such that they can display a map showing your current location. Combine that with a database of driving directions and we have real-time route planners where we mount either a dedicated GPS receiver or a cell phone on our car dashboards and provide destination details and the devices tell us the turns to make.

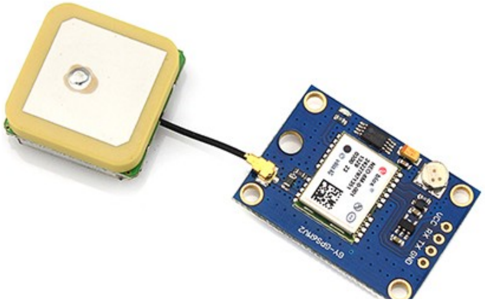
The Global Positioning System (GPS) is a set of satellites in orbit. They are continually transmitting a very precise time signal. If we have a suitable receiver, we can receive the time signals from some number of those satellites that are over head at any given time. Since the speed of radio transmission is constant and not instantaneous and each satellite is producing an extremely accurate and synchronized time signal, then a signal clock pulse emitted by all satellites at exactly the same time will be received at very slightly staggered times by the receiver as a function of the distance that the signal had to travel through space. This is the same as the distance that the satellite was from the receiver. By receiving enough signals from a sufficient number of distinct satellites and by using complex mathematics, the receiver can then triangulate its own position and thus know where it is on the surface of the Earth.

Putting that in perspective, an electronic module can determine physically where it is. Such modules are now common within your cell phone and within many car dashboards. They are often used in conjunction with mapping software to provide a real-time map showing your location. The accuracy of GPS is commonly about 4 meters.

Devices can be picked up for about the $12 mark. The unit I worked with is called the GY-GPS6MV2 which is based on the u-blox NEO-6m. The pins on this breakout board are vcc, gnd, RX and TX. This is 5V device. As such, you **must** use a level shifter or voltage divider between the TX pin of the GPS and the RX pin of the ESP32 as the ESP32 can't accept a 5V signal input.

Since it is a UART device, we can test this on a regular PC. If we connect a serial port terminal, we can watch the data be received. The data that comes across is in NMEA format (National Marine Electronics Association). There are various NMEA reader applications freely available which can format the data. The baud rate for the device defaults to 9600 bps.

To test the data, you should really be out-doors or otherwise have an un-obstructed view of the sky. Testing in the interior of a building is basically fruitless as the GPS signals will not penetrate and you will have learned nothing.



The device has an LED on the circuit that illuminates (flashes green) when a lock has been made.

Here is a sample that reads input from GPIO34 which is connected to the GPS TX:

```
#include "esp_log.h"
#include "driver/uart.h"

static char tag[] = "gps";
void doGPS() {
   ESP_LOGD(tag, ">> doGPS");
   uart_config_t myUartConfig;
   myUartConfig.baud_rate           = 9600;
   myUartConfig.data_bits           = UART_DATA_8_BITS;
   myUartConfig.parity              = UART_PARITY_DISABLE;
   myUartConfig.stop_bits           = UART_STOP_BITS_1;
   myUartConfig.flow_ctrl           = UART_HW_FLOWCTRL_DISABLE;
   myUartConfig.rx_flow_ctrl_thresh = 120;

   uart_param_config(UART_NUM_1, &myUartConfig);

   uart_set_pin(UART_NUM_1,
         UART_PIN_NO_CHANGE, // TX
         34,                 // RX
         UART_PIN_NO_CHANGE, // RTS
         UART_PIN_NO_CHANGE  // CTS
  );

   uart_driver_install(UART_NUM_1, 2048, 2048, 10, 17, NULL);

   unsigned char buf[100];
   int size;
   while(1) {
      size = uart_read_bytes(UART_NUM_1, buf, sizeof(buf), 1000/portTICK_PERIOD_MS);
      ESP_LOGD(tag, "Bytes read = %d", size);
      if (size >0) {
         ESP_LOGD(tag, "%.*s", size, buf);
      }
```

```
    }
}
```

GPS decoding

An open source project on Github called "minmea" provides an excellent parsing library for the protocol generated by these GPS devices. If we read a line at a time from the GPS device and pass that into the parsing routines, we can get data including position, speed and visible satellites. The library compiles 100% cleanly with ESP-IDF as long as one adds:

```
CFLAGS+=-Dtimegm=mktime
```

to the `component.mk` used to build from the source. The sample snippets supplied with the project proved to be more than sufficient to get a solution going in no time.

At a high level, the protocol exposed by the GPS device is called NMEA 0183. It is composed of one line at a time where each line is considered to be a "sentence". The sentence starts with a designator describing the types of sentence it represents. The minmea parser has support for the following types:

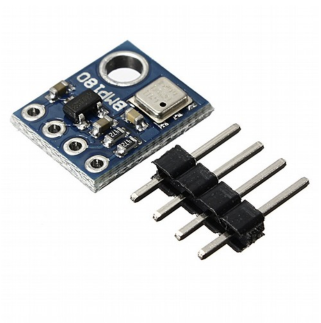
- `RMC` – Time, latitude, longitude, speed, track, date
- `GGA` – Time, latitude, longitude, GPS quality, satellite count, altitude
- `GSA` – Active satellites
- `GLL` – Latitude, longitude, time
- `GST` – Error measurements
- `GSV` – Satellites in view
- `VTG` – Track and speed

See also:

- Github: cloudyourcar/minmea
- NMEA Revealed

Temperature and pressure - BMP180

The BMP180 module can be found on eBay for less than $2.00. This device provides both temperature and barometric pressure readings via an I2C bus. It is less expensive than the DHT22 but replaces humidity for air pressure measurement. Arguably, the BMP180 is also easier to use as it does not require as finicky data stream timings. Although common, this device has been superseded by the manufacturer with a later model called the BMP280.



This is a 3V device. The I2C address of the device is `0x77`.

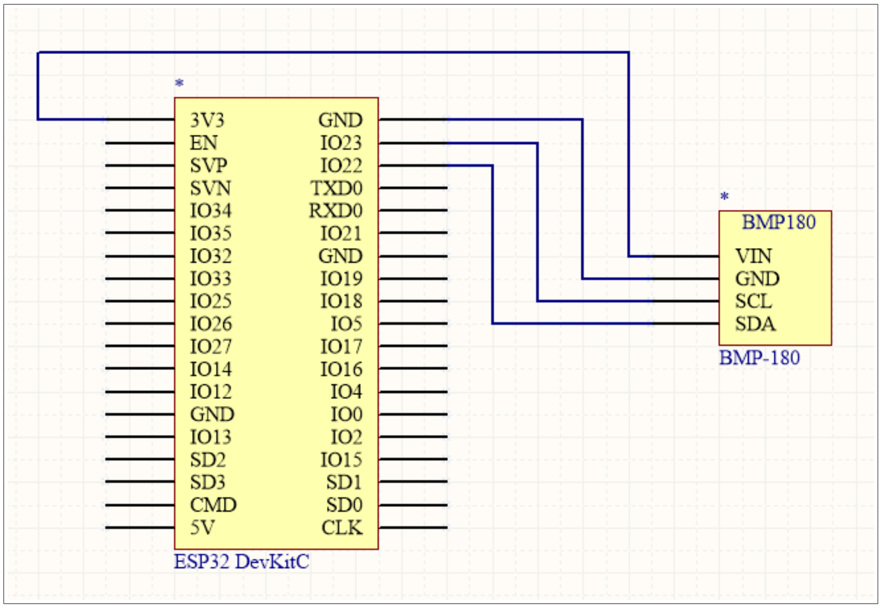
The I2C commands to read the device are described in detail in the data sheet. At a high level, we read out a series of EEPROM registers and then ask for both the temperature and pressure. The values returned for these later values need to them be arithmetically combined with the EEPROM registers using some equations and the result will be two outputs. A temperature in °C and an air pressure in Pa.

The pin out of the some device types are:

Pin	Label	Description
1	3.3	Not used
2	SDA	I2C SDA
3	SCL	I2C SCLK
4	GND	GND
5	VCC	3.3V

While others have the pin out of:

Pin	Label	Description
1	VIN	3.3V
2	GND	GND
3	SCL	I2C SCLK
4	SCA	I2C SDA



At the driver level, here are the low level details. First we read compensation parameters through I2C. The parameters are identified in the data sheet as 16 bit integers (MSB) read from the following I2C registers:

Parameter	C Type	Registers
AC1	short	0xAA, 0xAB
AC2	short	0xAC, 0xAD
AC3	short	0xAE, 0xAF
AC4	unsigned short	0xB0, 0xB1
AC5	unsigned short	0xB2, 0xB3
AC6	unsigned short	0xB4, 0xB5
B1	short	0xB6, 0xB7
B2	short	0xB8, 0xB9
MB	short	0xBA, 0xBB
MC	short	0xBC, 0xBD
MD	short	0xBE, 0xBF

My understanding (loosely) is that when an instance of a BPM180 is manufactured, there can be slight variances in the construction from one batch of devices from another. These variances are anticipated and have to be accommodated. As such a set of mathematical compensation values are written into an EEPROM contained within the device which provide the details of the specific variants for this exact device. When we ask the device for temperature and pressure values, we then have to also read the compensation values and apply some mathematics using those plus the obtained readings to achieve the final numbers. Fortunately, we don't have to worry about the actual underlying math, these have been worked out for us by the device designers. However, in our software, we do have to actually perform the math using the compensation values and the sensor read data. The details of the math can be found in the device data sheets and also from the many examples in Github that have already implemented the correct algorithms.

To read a register we perform:

<Start> <Address + WRITE> <Register> <Stop>
<Start> <Address + READ> <Read MSB> <Read LSB> <Stop>

See also:

- Data sheet
- Sparkfun – BMP180 Barometric Pressure Sensor Hookup
- BMP180 I2C Digital Barometric Pressure Sensor
- Github: Sparkfun/BMP180

Using the Arduino APIs

Adafruit provide an Arduino library for interacting with the device. This is part of their unified sensor series. It has been tested with the Arduino environment running on the Pi and found to work without issue. Stay away from the similarly named project called `Adafruit-BMP085-Library` as it appears to be buggy.

See also:

- Adafruit Unified Sensor Driver
- Adafruit Unified BMP085/BMP180 Driver

NeoPixels

NeoPixel theory

NeoPixels are LEDs that are driven by a single data line of high speed signaling. Most NeoPixels have a +ve and ground voltage source as well as a data line for input and a data line for output. The output of one NeoPixel can be fed into the input of the next one to produce a string of such LEDs. The input data to the LED is a stream of 24 bits of encoded data which should be interpreted as 8 bits for the red channel, 8 bits for the green channel and 8 bits for the blue channel. Each channel can thus have a luminance value of between 0 and 255. By mixing the values for each of the channels together, you can color an LED to any color you may choose. After sending in a stream of 24 bits, if we send in a second stream of 24 bits quickly after the first stream, the second stream is "pushed" through to the next LED in the chain. This can be repeated as far as desired. If we pause sending in data, the current values are "latched" into place and each LED them remembers its own value.

The timings of the data signals for these LEDs can be quite tricky but fortunately great minds have already built fantastic libraries for driving them correctly so we need not concern ourselves with these low level timings and can instead concentrate on devising interesting projects and purposes to which the LEDs can be placed. There are a number of different types of these LEDs with the most common ones being known as WS2811, WS2812 or PL9823.

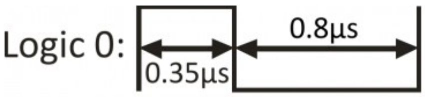
NeoPixels are usually 5V devices but check the specs carefully. Since NeoPixels draw a lot of current (60mA each), do **not** try and power them from low current sources. You also need to be careful when sending signals into them and getting their polarity correct. If you mess up, they are unforgiving. I recommend buying more instances that you will need in case you make a mistake.

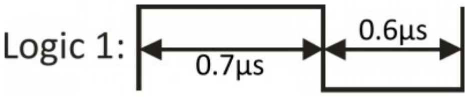
The eight piece strip is labeled CJMCU-2812-8. It has 4 pins at each end of the strip labeled:

Label	Description	Color
GND	Ground.	Yellow
DIN/DOUT	One side is DIN the other DOUT … this means that the strip is polarized. These are the control data input and output lines.	Green
4-7VDC	Source voltage.	Red
GND	Ground.	Yellow

The data is sent as 24 bits in Green/Red/Blue order (GRB). Note that this is NOT RGB. The format of the data uses NZR (Non-return-to-zero) encoding.

Specifically, the logic 0 and logic 1s are encoded as following.

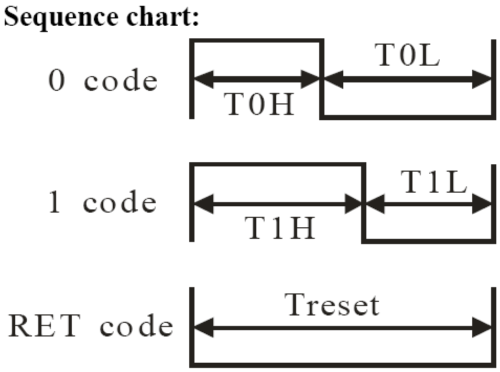




Both bits correspond to a transition from high to low to high again but the duration at which they are held indicates the encoding of a 1 or 0.

The data send to an individual device is 24 bits corresponding to 8 bits of color data for each of the RGB components. The 8 bits identify the brightness of each of the channels with 1 being lowest and 255 being brightest.

Various diagrams show the timing as follows:



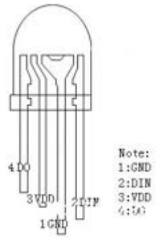
Various data sheets show different timing values in micro seconds:

Device	T0H	T0L	T0	T1H	T1L	T1
WS2811	0.5	2.0	2.5	1.2	1.3	2.5
WS2812	0.35	0.8	1.15	0.7	0.6	1.3
WS2812B	0.4	0.85	1.25	0.8	0.45	1.25
PL9823	0.35	1.36	1.71	1.36	0.35	1.71

As we see, for a WS2812B, the "width" of a bit is about 1.25 microseconds. For a full pixel, this would be 24 bits or 30 microseconds. Thinking of it another way, that would be over 33,000 pixels that could be updated per second.

The data is sent high bit first.

For testing circuits, I like to use PL9823. These can be had on eBay for between 25 cents and 35 cents each. It is physically easy to work with and in the event of a design or assembly error, they are cheap enough to shrug off damage and throw away ones you damage or break.



F8 9823 LED Chip

See also:

- [WS2812 Data Sheet]
- [PL9823 Data Sheet]
- [Use a $1 ATTiny to drive addressable RGB LEDs]
- [Adafruit NeoPixel Uberguide]
- [NeoPixels Revealed: How to (not need to) generate precisely timed signals]

NeoPixels and the ESP32

If we take the WS2812 as an average device, we see that we need to create timings on the order of 400ns. That is much too short an interval to perform in code logic so we will need hardware support to achieve the task. Fortunately, the RMT driver gives us exactly what we need. If we look at the base clock, we find that it has a maximum resolution of 80MHz which is a resolution of 12.5ns. In fact if we use a divisor of 8 we then have a granularity of 100ns which puts us right on target. Thus to send a 0, we would transmit high for 400ns and low for 800ns and to send a 1 we would transmit high for 800ns and low for 500ns. A latch is low for 50us (50,000ns).

A sample C++ class for driving NeoPixels from the ESP32 is available here:

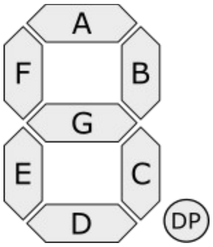
https://github.com/nkolban/esp32-snippets/tree/master/hardware/neopixels

See also:

- RMT – The Remote Peripheral

LED 7-Segment displays

The 7-Segment display is an LED device that is composed of 7 line segments that can be arranged to display numbers. There is also an eighth LED that is used to display a decimal point. The makeup of the segments are shown in the following diagram:



Although a seven segment display looks "quite dated" they should not be discounted. If what one wants to do is show a numeric value that can be read from a distance and which is very cheap to use, then this device may be just perfect.

It is common to see 7-segment displays used in conjunction with the MAX7219 or MAX7221 ICs. These ICs know how to drive up to eight multiplexed 7-segment displays with data received over an SPI bus. As such, the ESP32 can send in an SPI signal and these devices can easily display the results.

See also:

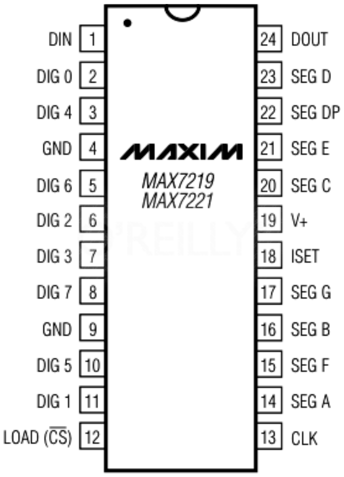
- Wikipedia – Seven-segment display

MAX7219/MAX7221 – Serial interface, 8-digit, led display drivers

The MAX7219 and MAX7221 are ICs that can drive up to eight digits of 7 segment displays or drive an 8x8 matrix of LEDs. The LEDs should be common cathode. The device works using the SPI protocol.

This is a 5V device. A 3.3V device known as the MAX6951 is also available. Note that the device is input-signal only. As such, there should be no issues connecting it to any of your ESP32 pins.

The physical layout of the IC looks as follows:



Place a 10uF electrolytic and 0.1uF ceramic as close to the device as possible.

For 7-Segment displays, these should be common cathode.

The pin out of the IC is shown in the next table.

Name	Pin	Description
DIN	1	Serial data input (MOSI). Data loaded on clock rising edge.
DIG0-DIG7	2, 3, 5-8, 10, 11	Connections to each of the eight digits.
GND	4, 9	**Both** ground pins must be connected.
$\overline{CS}$	12	Chip select. Data is loaded into serial register while $\overline{CS}$ is low and latched on $\overline{CS}$ rising edge. Unlike other SPI devices where you can leave the CS enabled, this device needs to actively control the CS.
CLK	13	The clock for the serial data,
SEGA-SEGG, DP	14-17, 20-23	Connection to each of the eight segments in a digit.
ISET	18	Connect to V_{DD} through a resistor to set peak segment current. The resistor values are shown in a following table. The resistor is referred to as R_{SET}. The current sent to a segment is 100 times the current entering ISET.
V_{DD}	19	5V
DOUT	24	Serial data output for daisy chaining.

Data format:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
X	X	X	X	Register Address				Data							

Registers

Name	Bits	Description
Shutdown	0b1100 (0xc)	Should the device shutdown the display • 0 – Shutdown. In shutdown mode the display is blanked. • 1 – Normal
Decoding	0b1001 (0x9)	How is decoding performed? • 0x00 – No decoding for digits 7-0 • Corresponding bit on – Code B decode for digit (0-9, E, H, L, P and '-')
Intensity	0b1010 (0xa)	The brightness of the display 0000 – 1111 with 0000 being minimum and 1111 being maximum.
Scan Limit	0b1011 (0xb)	Controls how many digits are to be included in the scan multiplexing. • 0b0000 – Digit 0 • 0b0001 – Digits 0, 1 • 0b0010 – Digits 0, 1, 2 • 0b0011 – Digits 0, 1, 2, 3 • 0b0100 – Digits 0, 1, 2, 3, 4 • 0b0101 – Digits 0, 1, 2, 3, 4, 5 • 0b0110 – Digits 0, 1, 2, 3, 4, 5, 6 • 0b0111 – Digits 0, 1, 2, 3, 4, 5, 6, 7
Display Test	0b1111 (0xf)	Test the display. • 0 – Normal operation • 1 – Test mode
No-Op	0b0000 (0x0)	Don't perform any task. Useful for passing data through the device to a daisy chained instance.
Digit 0	0b0001 (0x1)	Set the value for digit 0.
Digit 1	0b0010 (0x2)	Set the value for digit 1.
Digit 2	0b0011 (0x3)	Set the value for digit 2.
Digit 3	0b0100 (0x4)	Set the value for digit 3.
Digit 4	0b0101 (0x5)	Set the value for digit 4.
Digit 5	0b0110 (0x6)	Set the value for digit 5.
Digit 6	0b0111 (0x7)	Set the value for digit 6.
Digit 7	0b1000 (0x8)	Set the value for digit 7.

The value of the `RSET` resistor can be seen in the following table:

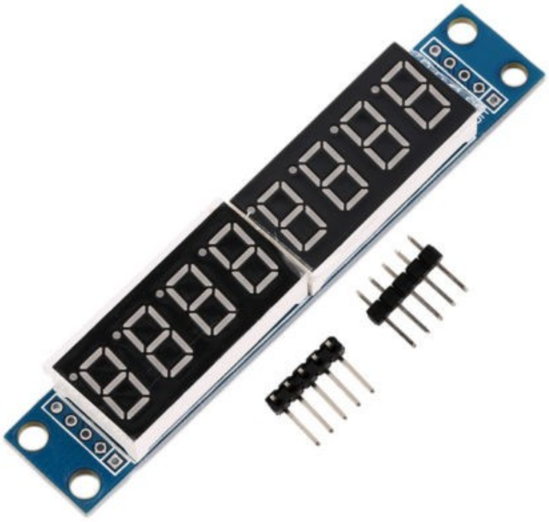
LED Forward Current	1.5V	2.0V	2.5V	3.0V	3.5V
10ma	66.7k	63.7k	59.3k	55.4k	51.2k
20ma	29.8k	28.0k	25.9k	24.5k	22.6k
30ma	17.8k	17.1k	15.8k	15.0k	14.0k
40ma	12.2k	11.8k	11.0k	10.6k	9.7k

Anodes of the LEDs must be connected to the `SEGx` lines while cathodes must be connected to the `DIGx` lines.

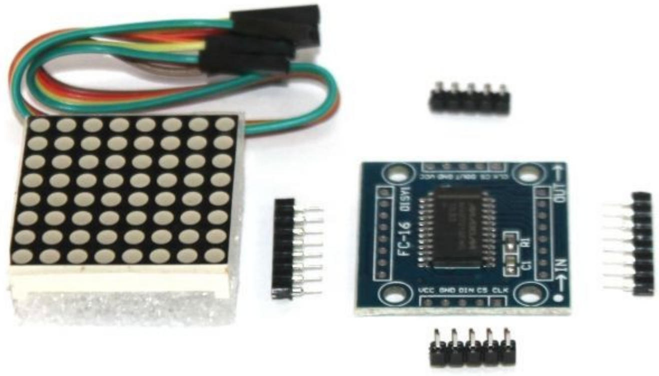
There are boards available which have MAX7219's already mounted and ready for work including 8x8 LED matrices. These are much easier to work with than wiring together a rats-nest of links. The boards cost less than $2 an instance. The pin out from the board are:

Pin	Label	Description
1	Vcc	+ve
2	GND	Ground.
3	DIN	Data in.
4	CS	Chip Select.
5	CLK	Clock.

Here is an image of such a board:



In addition to these 7-segment LEDs, there are also 8x8 matrix boards that use the same MAX7219 and have the same pin outs.



See also:

- MAX7221 Home Page
- Drive MAX7219/MAX7221 with common anode displays
- Arduino – The MAX7219 and MAX7221 LED drivers
- Github: Arduino – wayoda/LedControl library
- Instructables – 16x8 LED dot matrix with MAX7219 module
- YouTube – ESP32 Technical Tutorials: Driving the MAX7219
- YouTube – BrainyBits - How to use MAX7219 Dot LED matrix with Arduino
- YouTube – Scrolling text using the MAX7219 and an Arduino
- YouTube – Arduino tutorial: LED Matrix red 8x8 64 Led driven by MAX7219 (or MAX7221) and Arduino Uno

The U8g2 library

Available on Github is a rather good library for driving monochrome displays. The library is called U8g2. See the links for the web page. There are actually two libraries. One is called U8g2 which requires a memory buffer in the ESP32 to hold the raster of the image and a second library called U8x8 which is text only and needs no raster buffer. In order to work with the library, you need to know the controller type of the graphics device plus the size of the display in pixels. There are separate constructors for each of the distinct supported combinations. The supported combinations can be found here.

The primitive drawing commands are bound to C functions and include filled rectangles, frames, circles, ellipses, text and much more. Tests I have performed show that it worked fine with an SSD1306.

When we think of a display, we should think of its as a rectangular array of pixels of a fixed width and height. It is common to keep a data representation of what should be shown on the display in RAM and this is known as a frame buffer. A change in the RAM which is then pushed to the display causes an update of the display. If we set a bit in the frame buffer to be 1 and then cause the frame buffer to be displayed, the corresponding pixel will illuminate.

The manufacturers of distinct displays have chosen distinct technologies for representing pixel data and how that pixel data is pushed to the displays. For example, some vendors have 8 bits of data as the vertical set of pixels top to bottom while others have them going from left to right. The U8g2 library attempts to normalize the story by providing logical primitives that are then mapped to the correct data streams and frame buffer updates appropriate for the specific device.

The U8g2 library is MCU agnostic. That means that it knows nothing about ESP32 and everything about displays. To have the library work with the ESP32 we need a hardware abstraction level or HAL. The HAL maps logical requests issued by the library such as set a GPIO pin from high to low or send data via SPI to the actual APIs that are needed by the target platform, in our case an ESP32. A set of HAL routines for the ESP32 has been written. These routines can be compiled and linked with the U8g2 library and the ESP32 application to provide ESP32 support.

See also:

- Github: olikraus/u8g2
- U8g2 API reference
- YouTube – ESP32 Technical Tutorials: Displays and the U8g2 library

LCD display – Nokia 5110 – PCD8544

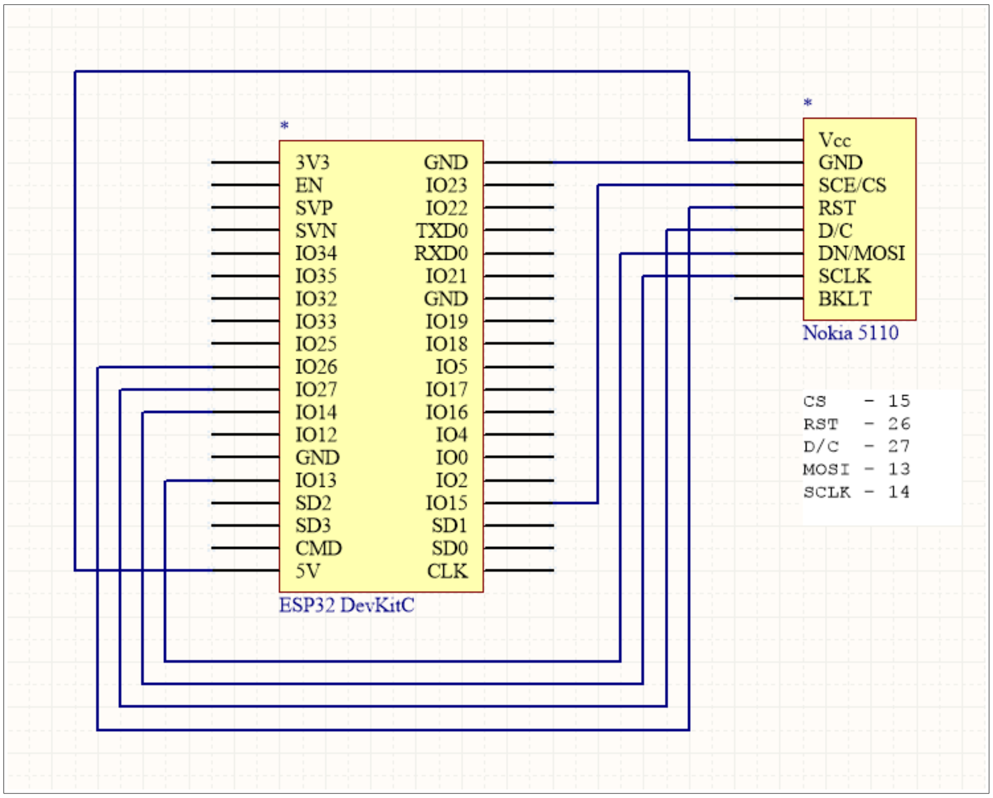
This little LCD screen has a resolution of 84x48 pixels and can be picked up on eBay for about $3. The underlying IC is the PCD8544 but it is also known as the Nokia 5110. The device is driven by an SPI interface. Since the device is input (to the device) only, we need not be concerned with overloading ESP32 input pins.

The pins on the board are:

Pin	Description
1	Vcc – 6V – 8.5V
2	GND
3	$\overline{\text{SCE}}$ / $\overline{\text{CS}}$ – Chip enable. A low edge indicates start of data transmission.
4	$\overline{\text{RST}}$ – Reset the device. Active low.
5	D/$\overline{\text{C}}$ – Input is Data (1) or command (0).
6	DN <MOSI> – The SPI MOSI
7	SCLK – The SPI clock. The maximum permissible clock speed is 100KHz.
8	Backlight LED

The data sheet explains well the sequence of commands that need to be sent to drive the display however, as always should be the case, you should look to be leveraging what already exists rather than re-inventing for the sake of it. The fantastic folks at Adafruit with their Adafruit GFX graphics library have provided an Arduino library for working with the device. This library has been ported to work with the ESP32.   As such, all you need are copies of those libraries and you are ready to go.

An example set of pin mappings might be:



```
CS   - 15
RST  - 26
D/C  - 27
MOSI - 13
SCLK - 14
```

See also:

- Data Sheet

OLED 128x32, 128x64 – SSD1306

Another screen device that is readily available are the small OLED displays. These are based on an IC called the SSD1306 and can be found on eBay using that phrase as a search. The price for an instance seems to range between $5 and $12.

The device can operate at either 3.3V or 5V.

The typical resolution is either 128x32 or 128x64 pixels. At a 6x8 character resolution, this would give us 4 or 8 rows of 21 (and a bit) columns (128 pixels/6 pixels per column = 21.333) .

The pin out on the SPI device is:

Pin	Description
1	GND – Ground
2	VCC – 3.3v or 5.0v
3	D0 – Clock
4	D1 – MOSI
5	RES – Reset
6	DC – Data / Command
7	CS – Chip Select

The pin out on the I2C device is:

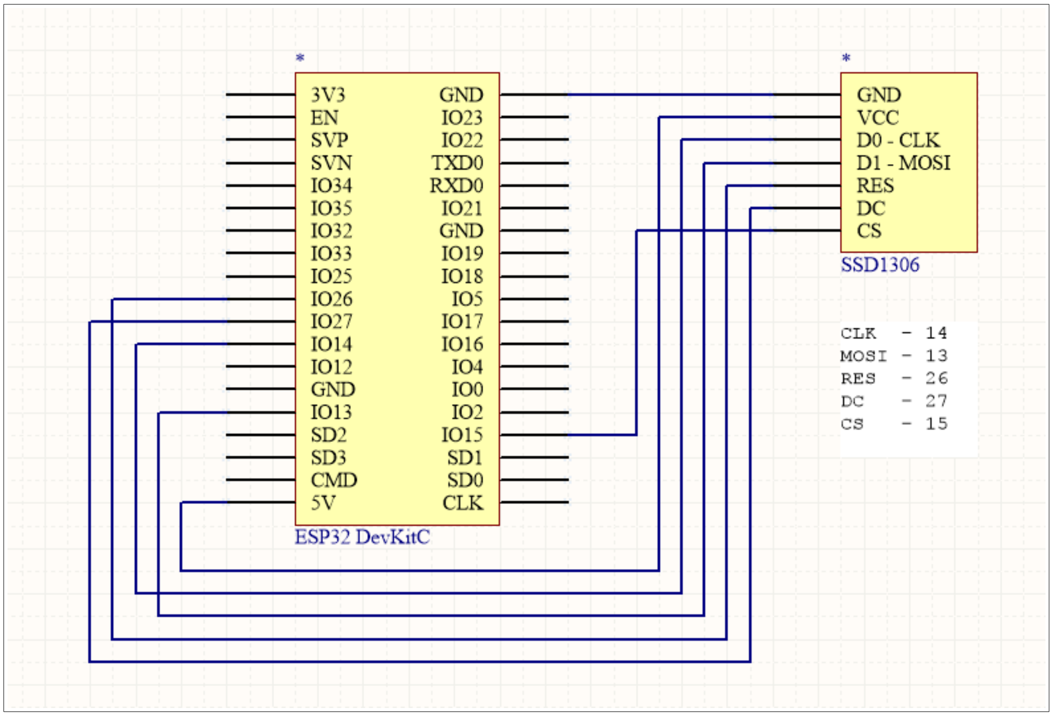
Pin	Description
SDA	Data
SCL	Clock
VCC	3.3V
GND	GND

The I2C address is 0x3C or 0x3D.

The device can support a variety of protocols including 3 or 4 wire SPI and I2C. Depending on the model / variety of the board it you obtain, it is likely that it will be physically configured in one particular mode. Examine the board and the associated documentation.

Again the excellent Adafruit folks have produced libraries for the Arduino which have been ported to the ESP32.

A potential wiring diagram is as follows:



This display has also been tested with the U8g2 lib and worked as advertised.

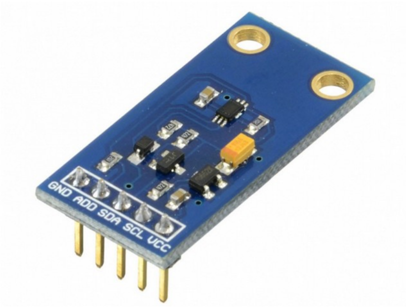
See also:

- The U8g2 library
- SSD1306 Data Sheet
- Github – Adafruit SSD1306
- Adafruit – Monochrome OLED breakouts
- ESP32 - Adafruit_SSD1306-Library

Ambient light level sensor – BH1750FVI

An interesting little module incorporating the BH1750FVI (aka GY-302) is available from a number of suppliers on eBay. Simply put, this device measures the amount of light falling upon it and returns a measurement of that value. While that can be done through a simple light dependent resistor and an analog to digital converter, this device is arguably easier to use as the device attaches to an I2C bus. In addition, the device characteristics are to sense light in the same spectrum as the human eye, something I'm not sure is true for a simple light dependent resistor solution. Further, the device has 16bits of sensitivity as compared to a maximum of 12 bits of sensitivity found the ESP32 ADC.

This is a 3.3V device, do **not** attempt to power fro 5V.



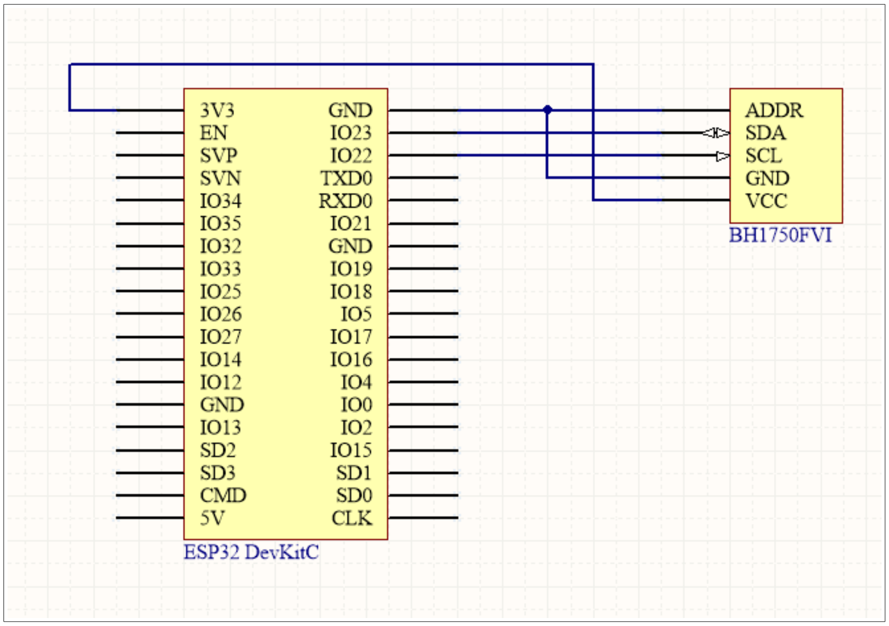
Pin	Function
VCC	3.3V
GND	Ground
SCL	I2C Clock
SDA	I2C Data
ADDR	Address select

The following commands can be sent by an I2C write:

Value	Function
0x00	Power down
0x01	Power on
0x07	Reset
0x10	Mode H
0x11	Mode 2H
0x13	Mode L
0x20	One time Mode H
0x21	One time Mode 2H
0x23	One time Mode L

The I2C bus address of the device is `0x23`.

Here is a simple schematic illustrating the a wiring of the device.

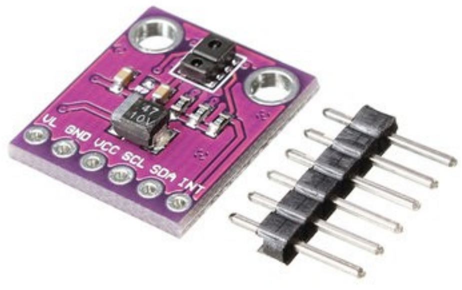


See also:

- Rohm BH1750FVI – Data sheet
- Wikipedia – Lux
- YouTube: ESP32 Technical Tutorial – Ambient light levels

Ambient light and proximity sensor

The APDS-9930 is an ambient light and proximity sensor. This is a 3.3V device using I2C.



Module pins:

VL	IR LED power
GND	Ground
VCC	3.3V
SCL	I2C Clock
SDA	I2C Data
INT	Interrupt

See also:

- Data sheet

Infrared receivers

Infrared is a frequency range of light that is not visible to our eyes. If we look at a source of infrared light, we simply won't see any emissions even though they are actually being emitted. There are LEDs available that can generate infra-red frequency light and there are corresponding detectors that can measure the incoming amount of infrared light. Since these light sources don't distract us, we can use them for signaling by having an infra-red transmitter send pulses of light to an infra-red receiver. This is precisely how many home remote control hand-sets work for devices such as TVs and music systems.

What we might want to do is to control the ESP32 remotely using an infra-red hand-set.

There are many infra red hand-sets that we can use that can be purchase on eBay for a few dollars. Alternative we can use any hand-set from an old TV.

We will also need to wire in an infra-red receiver circuit to our ESP32.

With the transmitter (hand-set) and the receiver (electronics connected to ESP32) we are about ready to go. The next thing we have to consider is how to receive the data transmitted and how to decode it.

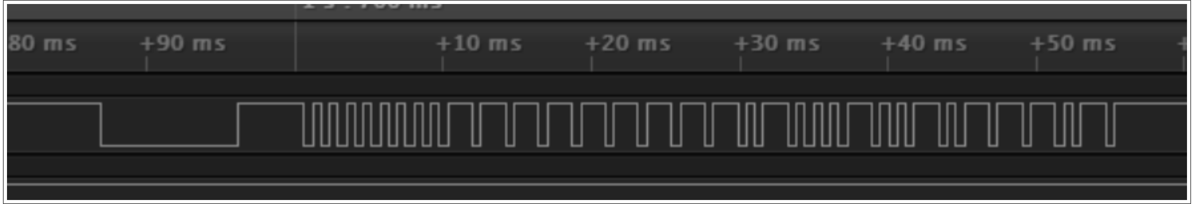
One of the more popular infrared receivers is the VS1838B.

The pin out of this device (from left to right facing the front):

Pin	Function
1	Data
2	GND
3	Vcc 2.7-5.5V

This device is able to receive at a frequency resolution of 38KHz.

If we connect the data out to a logic analyzer and then apply an IR signal from a controller, we can see the incoming pulse-train. For example:



Obtaining the data from an IR receiver LED is supported natively by the hardware of the ESP32 using the RMT peripheral. It can be configured to watch for pulse trains and return us an array of items where each item represents the duration a signal is low and the duration a signal is high. From this data we obtain very accurate representations of the IR data. However, there are still puzzles to be solved. Most remote controls have multiple buttons … power on, power off, volume up, volume down etc. Each button performs a distinct function and hence must be sending a distinct pattern of data to represent the function of that button. As such, not only do we want to detect that a pulse train has arrived, we also need to decode its content. It would have been nice if there had been one format for all remote controls, but sadly, that isn't the case. There are in fact quite a few protocols made by different vendors. The protocols include NEC, Sony SIRC, Phillips RC5, Phillips RC6 and more.

Let us use the NEC protocol as an example. A train starts with a 9ms mark followed by a 4.5ms space. It is safe to assume that the durations measured won't be exactly 9ms and 4.5ms … so we need to accommodate tolerance to the values actually measured.

See also:

- SB-Projects IT Remote Control Theory
- NEC protocol

RFID MFRC522

Perhaps you have stayed at a modern hotel recently. In some cases when they give you the key-card at the reception desk and you get to your door, you may find that you don't actually insert the card in a slot but instead bring it close to the door lock in order to gain entry. Maybe you are an employee at a big corporation where you wear an ID badge and in order to gain entry into the building, you bring your card near a turnstile or door badge reader.

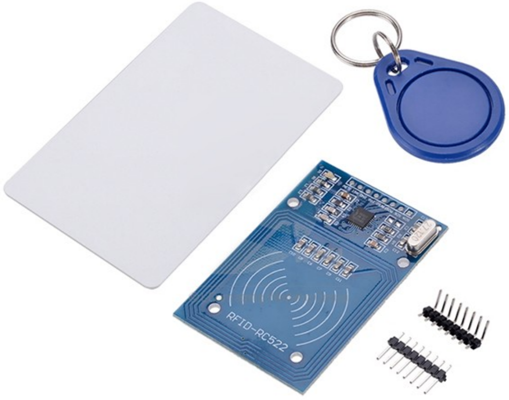
In other cases, we find that products contain identification tags that can be used at check-out or during stocking to determine information about them. These packages can then be scanned to learn more:

What these have in common is the inclusion of an RFID.

Radio frequency identification (RFID) is the notion that we can have a receiver that is listening for low power radio frequency emissions. When a device that emits a signal on the correct frequency is brought in range, the receiver is triggered. Typically, the range of such a device is only a few inches and the transmitter itself is a passive device … meaning it has no power source within it but is instead activated by the receiver itself transmitting enough energy for the transmitter to modulate a response. We see these kinds of devices in hotels when they can be used to open doors. Instead of inserting a physical key or even a key card into a slot, we simply bring the card "near" or "tap it" on the lock and the door opens.

The transmitters can also transmit small amounts of information to the receiver and that information can be used allow or disallow access.

For our purposes, we can use the MFRC522 receiver and associated cards and transmitters. If you see the phrase "`PICC`" … this stands for "proximity integrated circuit card" which is the formal name for the RFID cards and tags.

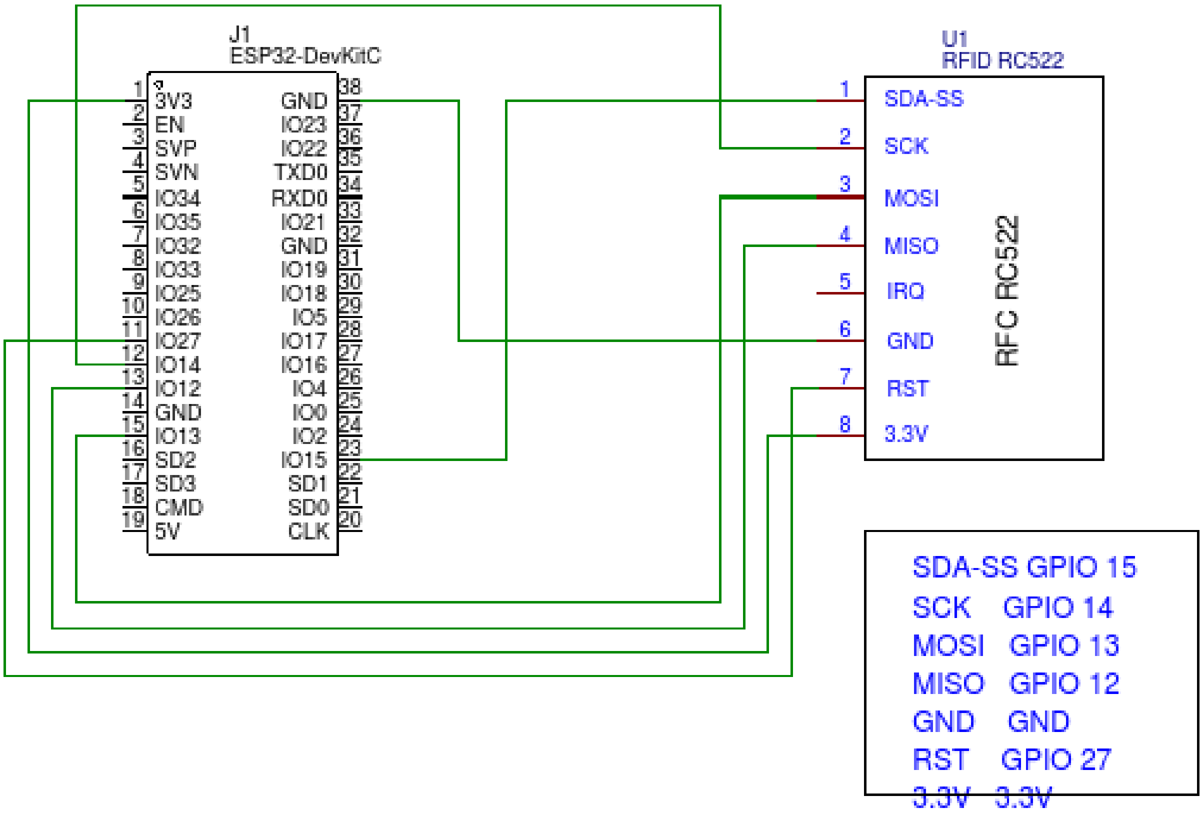


A receiver and some transmitters can be picked up on eBay for under $3.

The MFRC522 is a 3.3V device. Do not try and power it from the 5V source. The cards can hold up to 1K bytes of data and have a range of 1cm or 2cm. The communication between the receiver and the ESP32 is via the SPI protocol.

The pin out on the device is:

Pin	Label	Description
1	3.3V	Source
2	RST	Reset
3	GND	Ground
4	IRQ	Can be left unconnected
5	MISO	Master In / Slave Out
6	MOSI	Master Out / Slave In
7	SCK	Clock
8	SDA	Slave select or SDA. Quite why this is labeled SDA is a mystery as that is an I2C term and this board only supports SPI. It does seem to serve as a slave select.



SDA-SS	GPIO 15
SCK	GPIO 14
MOSI	GPIO 13
MISO	GPIO 12
GND	GND
RST	GPIO 27
3.3V	3.3V

The data on the card is broken out into sectors, blocks and bytes. There are 16 sectors identified as 0 through 15. Each sector contains 4 blocks. These are identified as 0 through 3. Each block contains 16 bytes. If we look at the total … 16 sectors * 4 blocks * 16 bytes we end up with 1024 bytes (or 1K).

See also:

- Wikipedia – Radio-frequency identification
- MFRC522 Data sheet
- YouTube: Tutorial:Using a RFID Card Reader with the Arduino
- YouTube: ESP32 #33: RFID Read and Write with MFRC522 Module
- Github: miguelbalboa/rfid – Arduino library for MFRC522

MFRC522 - Low levels

This is a 3.3V device, you will ruin it if you attempt to connect it to a 5V source.

The chances are that this section will be of little value to you. Most of the components I work with have relatively simple interfaces but this one has a lot of options and unless one wants to become a deep student of the device, it is likely that one of the pre-existing libraries will be what is most beneficial. However, it is my intent that, over time, I will read the data sheet carefully and combine that with the current know open source implementations and try and provide at least a readers guide to understanding the protocols and algorithms.

The device has a lot of registers:

<table>
<tr><th>Address</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x01</td><td>CommandReg</td><td>Starts and stops command execution.</td></tr>
<tr><td>0x02</td><td>ComlEnReg</td><td>Control of interrupt requests.</td></tr>
<tr><td>0x03</td><td>DivlEnReg</td><td>Interrupt request bits.</td></tr>
<tr><td>0x04</td><td>ComIrqReg</td><td></td></tr>
<tr><td>0x05</td><td>DivIrqReg</td><td></td></tr>
<tr><td>0x06</td><td>ErrorReg</td><td>Error status of last command executed.</td></tr>
<tr><td>0x07</td><td>Status1Reg</td><td>Status register.</td></tr>
<tr><td>0x08</td><td>Status2Reg</td><td>Status register.</td></tr>
<tr><td>0x09</td><td>FIFODataReg</td><td></td></tr>
<tr><td>0x0a</td><td>FIFOLevelReg</td><td>Number of bytes in FIFO queue.</td></tr>
<tr><td>0x0b</td><td>WaterLevelReg</td><td>Threshold for FIFO queue under and over flow warning.</td></tr>
<tr><td>0x0c</td><td>ControlReg</td><td>Misc control bits.</td></tr>
<tr><td>0x0d</td><td>BitFramingReg</td><td>Bit oriented frames.</td></tr>
<tr><td>0x0e</td><td>CollReg</td><td>Collision detection handling.</td></tr>
<tr><td>0x0f</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x10</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x11</td><td>ModeReg</td><td>General transmit and receiver controls.
<table>
<tr><th>Pin</th><th>Name</th></tr>
<tr><td>7</td><td>MSBFirst</td></tr>
<tr><td>6</td><td>N/A</td></tr>
<tr><td>5</td><td>TxWaitRF</td></tr>
<tr><td>4</td><td>N/A</td></tr>
<tr><td>3</td><td>Polarity of MFIN</td></tr>
<tr><td>2</td><td>N/A</td></tr>
<tr><td>1:0</td><td>CRC Preset</td></tr>
</table></td></tr>
<tr><td>0x12</td><td>TxModeReg</td><td>Transmit rate.
<table>
<tr><th>Pin</th><th>Name</th></tr>
<tr><td>7</td><td>TAuto</td></tr>
<tr><td>6:5</td><td>TGated</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td><table><tr><td>4</td><td>TAutoRestart</td></tr><tr><td>3:0</td><td>TPrescaler_Hi</td></tr></table></td></tr>
<tr><td>0x13</td><td>RxModeReg</td><td>Receive rate.</td></tr>
<tr><td>0x14</td><td>TxControlReg</td><td>Controls antenna driver pins.<table><tr><th>Pin</th><th>Name</th></tr><tr><td>7</td><td>InvTx2RFOn</td></tr><tr><td>6</td><td>InvTx1RFOn</td></tr><tr><td>5</td><td>InvTx2RFOff</td></tr><tr><td>4</td><td>InvTx1RFOff</td></tr><tr><td>3</td><td>TX2CW</td></tr><tr><td>2</td><td>N/A</td></tr><tr><td>1</td><td>TX2RFEn</td></tr><tr><td>0</td><td>TX1RFEn</td></tr></table></td></tr>
<tr><td>0x15</td><td>TxASKReg</td><td>Transmit modulation setting.
Bit 6 1 Forces a 100% ASK modulation</td></tr>
<tr><td>0x16</td><td>TxSelReg</td><td>Analog module control.</td></tr>
<tr><td>0x17</td><td>RxSelReg</td><td>Receiver settings.</td></tr>
<tr><td>0x18</td><td>RxThresholdReg</td><td>Thresholds for receiving.</td></tr>
<tr><td>0x19</td><td>DemodReg</td><td>Demodulator settings.</td></tr>
<tr><td>0x1a</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x1b</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x1c</td><td>MfTxReg</td><td>MIFARE transmit parameters.</td></tr>
<tr><td>0x1d</td><td>MfRxReg</td><td>MIFARE reception parameters.</td></tr>
<tr><td>0x1e</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x1f</td><td>SerialSpeedReg</td><td>UART serial speed.</td></tr>
<tr><td>0x20</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x21</td><td>CRCResultReg</td><td>CRC calculation.</td></tr>
<tr><td>0x22</td><td>CRCResultReg</td><td>CRC calculation.</td></tr>
<tr><td>0x23</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x24</td><td>ModWidthReg</td><td>Modulation width.</td></tr>
<tr><td>0x25</td><td>Reserved</td><td>Reserved for future use.</td></tr>
<tr><td>0x26</td><td>RFCfgReg</td><td>Receiver gain.</td></tr>
<tr><td>0x27</td><td>GsNReg</td><td>Conductance of antenna pins.</td></tr>
<tr><td>0x28</td><td>CWGsPReg</td><td>Detailed</td></tr>
<tr><td>0x29</td><td>ModGsPReg</td><td>Detailed.</td></tr>
<tr><td>0x2a</td><td>TModeReg</td><td>Timer settings.</td></tr>
</table>

0x2b	TPrescalerReg	
0x2c	TReloadRegH	
0x2d	TReloadRegL	
0x2e	TCounterValReg	
0x2f	TCounterValReg	
0x30	Reserved	
0x31	TestSel1Reg	Test signal configuration.
0x32	TestSel2Reg	Test signal configuration.
0x33	TestPinEnReg	
0x34	TestPinValueReg	
0x35	TestBusReg	
0x36	AutotestReg	
0x37	VersionReg	MFRC522 software version.
0x38	AnalogTestReg	
0x39	TestDAC1Reg	
0x3a	TestDAC2Reg	
0x3b	TestADCReg	

Be careful when using registers, they are encoded as follows:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1 = read 0 = write	address						0

As you can see, an address contains whether we are reading or writing to it … it is also shifted one bit to the left.

To say that there is a lot of work in building access to RFID by hand is an understatement. However, by looking closely at the work already performed by others in various open source projects, we can get a feel for the steps of an implementation:

Initialization

```
Reset
TModeReg      ← 0x8d - TAuto=1, non-gated, TPrescalerHi=b1101
TPrescalerReg ← 0x3e - Low 8 bits of TPrescaler
TReloadRegL   ← 30
TReloadRegH   ← 0
TxASKReg      ← 0x40 - Force 100% ASK
ModeReg       ← 0x3d - Not MSBFirst, TxWaitRF, MFin active High, CRCPreset=0x6363
AntennaOn
```

AntennaOn

```
If TxControlReg→ Tx1RFEn is off or TxControlReg→Tx2RFEn is off
  TxControlReg→Tx1RFEn = 1
  TxControlReg→Tx2RFEn = 1
```

Cameras

It is tantalizing to think about processing real-time images in an ESP32 and to begin down that journey, we need to be able to source video input.

See also:

- Github: igrr/esp32-cam-demo

Ivan's sample

Ivan over at Espressif has produced a great sample application that utilizes the camera. The camera supported is the OV7725. It exposes the following functions:

- `uint8_t *camera_get_fb()` - Get the data of the frame buffer.

  `int camera_get_fb_height()` - Get the height of the frame buffer.

- `int camera_get_fb_width()` - Get the width of the frame buffer.
- `esp_err_t camera_init(camera_config_t config)` – Configure the camera interface. This is a structure which contains:
  - pin_reset
  - pin_xclk
  - pin_sscb_sda
  - pin_sscb_scl
  - pin_d7
  - pin_d6
  - pin_d5
  - pin_d4
  - pin_d3
  - pin_d2
  - pin_d1
  - pin_d0

    - pin_vsync
    - pin_href
    - pin_pclk
    - int xclk_freq_hz
    - ledc_timer_t ledc_timer
    - ledc_channel_t ledc_channel
    - camera_pixelformat_t pixel_format
        - CAMERA_PF_RGB565
        - CAMERA_PF_YUV422
        - CAMERA_PF_GRAYSCALE
        - CAMERA_PF_JPEG
    - camera_framesize_t frame_size
        - CAMERA_FS_QQVGA
        - CAMERA_FS_QVGA
        - CAMERA_FS_VGA
        - CAMERA_FS_SVGA
    - int jpeg_quality
- camera_print_fb()
- camera_probe(const camera_config_t *config, camera_model_t *out_camera_model)
- `esp_err_t camera_run()` - Ask the camera to grab an image and then wait until it is ready.

In this sample, we get a buffer which is width x height bytes in size and is composed of 8 bit grayscale values. We can save the image data but how can we look at it?

I have had good success using the Linux ImageMagick tools. Specifically, if I save an image to the file "img.gray" and we know that the image is 320 x 240 pixels, we can convert this to a PNG using:

```
$ convert -size 320x240 -depth 8 img.gray img.png
```

See also:

- Github: igrr/esp32-cam-demo
- ImageMagick
- OV7725

OV7670

The OV7670 camera is a very cheap camera device. It is powered at 3V and hence is perfect for use with the ESP32. The resolution is 640x480 pixels.

There are outputs of three synchronization signals. These are called VSYNC, HSYNC, and HREF.

See also:

- OV7670 – Datasheet
- OV7670 – Implementation Guide – 2005-09-02
- OV7670 Software Application Note

Accelerometer and Gyroscope - MPU-6050 (aka GY-521)

The MPU-6050 is a 3.3V device.

An accelerometer measures the force of linear acceleration. Simply put, that when an object is at rest (i.e. sitting peacefully on your desk), it is not being accelerated (well … it actually is but we'll come back to that later). When you start to move the object, it undergoes acceleration. Note that this is not the same as speed. Acceleration is the rate of change in speed over time. When you are sitting as a passenger in your car and the car is moving at a constant speed and if the ride were very smooth and you closed your eyes, you wouldn't know you were moving. However, if the gas or brake are pressed, your car would *change* speed and you would feel acceleration while the speed changed. You would feel yourself pushed back in your seat when the gas is pressed and you would feel yourself wanting to move forward if the brake is pressed. Similarly, if you stand in an elevator and press a button, you feel yourself being pushed into the floor when it starts to rise and you would find yourself wanting to rise when it comes to a stop. Acceleration is a vector quantity meaning that it has a directional quality associated with it. In your car (assuming you are driving forward and we call the forward direction `x`) then stepping on the gas produces an acceleration in the `x` direction while stepping on the brake produces an acceleration in the `-x` direction. Similarly with the elevator, when you rise there will be an acceleration in the `z` direction and when you stop, there will be an acceleration in the `-z` direction. There is one more twist to the story … gravity. Although you may not have thought about it, as you sit in your chair there is a force acting on you that wishes to accelerate you. That force is called gravity and the direction of the acceleration is `-z`. If it weren't for the chair, your body would be accelerated towards the floor. Even though your body isn't moving, we can consider this an acceleration force being applied. What this means for us is that when a device such as the MPU-6050 is sitting on your desk, it will be reporting that it is accelerating downwards.

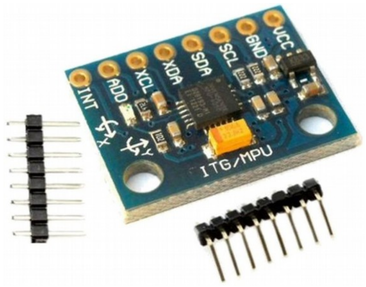
The practical implication of this is very interesting. If the MPU-6050 is flat on the desk, it will report an acceleration in its `-z` direction. If we now were to tilt the device, the acceleration due to gravity is still present but now the direction of that force will have changed. The magnitude of the acceleration will remain the same but the measurements will now show it spread over the `x`, `y` and `z` axises. Working backwards, by examining the values of the acceleration on the `x`, `y` and `z` axises, we can calculate the orientation of the device relative to the direction of gravity (i.e. relative to up and down).

Having just discussed the concept of measurement of acceleration through the notion of an accelerometer, we can now look at how a gyroscope comes into play. A gyroscope measures the change in velocity (if at all) of a device rotating around its own axis, A gyroscope measures changes in angular velocity.

The MPU-6050 combines these to measure both linear acceleration and angular velocity. The device measures both qualities in the `x`, `y` and `z` axis and hence is considered a measurement in 6 degrees of freedom (linear acceleration in `x`, `y` and `z` and angular velocity in `x`, `y` and `z`). The measurement values have a 16 bit resolution.

If our goal is to measure the orientation of the device, we may be able to see that the orientation can be found by examining the accelerometer values however these values only result in accurate answers if the device is at rest. If it is moved, then the acceleration due to movement can produce jittery results. If we look at the gyroscope, if we start from a known orientation (eg. flat on the desk), then in principle we should also be able to determine the device's current orientation by adding together all the gyroscopic changes that have happened to it since its original known (calibrated) position. Unfortunately, both of these techniques introduce errors. Using the accelerometer, we can determine our orientation by averaging values over time to remove jitter. As such, it gives good values over time but poor for short term measurement. The gyroscope gives us good angular momentum values in the short term but the errors accumulate over time when we try and calculate the orientation of the device from its base state by successive addition of delta values. Fortunately, we can combine these two techniques through a variety of algorithms to give us a good value that is built from the combination of the two measurements.

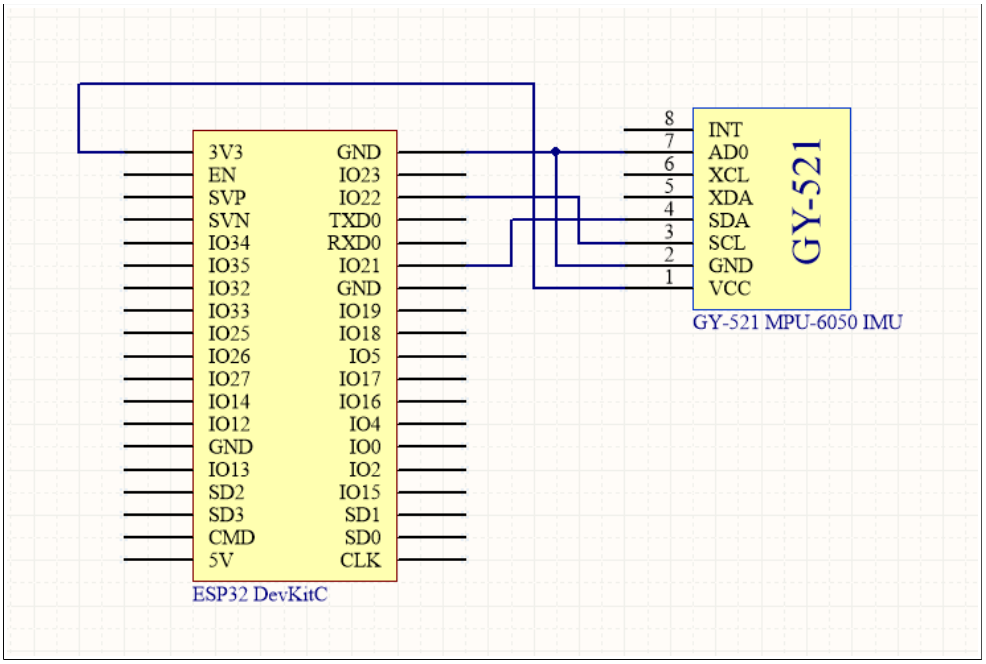
The MPU-6050 device connects via an I2C bus (default address is `0x68` but can be changed to `0x69`). This is 3.3V device and hence is safe to connect to ESP32 pins directly.



The device has the following pins exposed from the breakout boards:

Pin	Description
`VCC`	3.3V
`GND`	Ground
`SCL`	I2C Clock
`SDA`	I2C Data
`XDA`	Auxiliary I2C Data
`XCL`	Auxiliary I2C Clock
`ADD`	I2C address selection, either `0x68` or `0x69`. Low = `0x68`, high = `0x69`. It is not recommend to let this line float. Tie it to the desired signal.
`INT`	

A sample wiring of the device to ESP32 looks as follows:



After wiring, if we run an I2C scanner we will see the device:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Register	Offset	Name	Description
0x3B	0	ACCEL_XOUT_H	AccelX High
0x3C	1	ACCEL_XOUT_L	AccelX Low
0x3D	2	ACCEL_YOUT_H	AccelY High
0x3E	3	ACCEL_YOUT_L	AccelY Low
0x3F	4	ACCEL_ZOUT_H	AccelZ High
0x40	5	ACCEL_ZOUT_L	AccelZ Low
0x41	6	TEMP_OUT_H	Temp High
0x42	7	TEMP_OUT_L	Temp Low
0x43	8	GYRO_XOUT_H	GyroX High
0x44	9	GYRO_XOUT_L	GyroX Low
0x45	10	GYRO_YOUT_H	GyroY High
0x46	11	GYRO_YOUT_L	GyroY Low
0x47	12	GYRO_ZOUT_H	GyroZ High
0x48	13	GYRO_ZOUT_L	GyroZ Low

Note that the data values are signed 16 bits in big endian format.

To use the I2C protocol, we send the value 0 to register PWR_MGMT_1 (0x6b).

Flag	Description
Device Reset [7]	Reset the device
Sleep [6]	Put the device to sleep
Cycle [5]	Cycle between awake and asleep
Temp disable [3]	Disable the temperature sensor
Clksel [2:0]	Clock source of the devuce

You will sometimes see these devices referred to as Inertial Measurement Units (IMUs).

The value returned from the accelerometer is a raw value. The actual result we want is measured in gravities (g) which is about 9.81 ms^{-2}. The device has various sensitivities built in. To get to the real value, we need to divide by a scaling factor as show below:

Acceleration Limit	Sensitivity Factor
2g	16384
4g	8192
8g	4096
16g	2048

Thus to get the required value from the raw value we would use the equation:

$$required\,value = \frac{raw\,value}{sensitivity\,factor}$$

By default, the device is configured for a 2g acceleration limit

Similarly, there is an angular velocity scaling:

Angular Velocity Limit	Sensitivity Factor
250°/s	131
500°/s	65.5
1000°/s	32.8
2000°/s	16.4

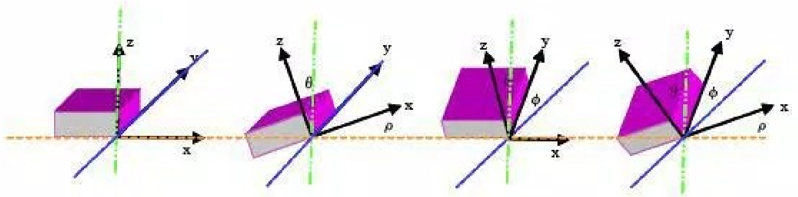


Figure 8. Three Axis for Measuring Tilt

$$\rho = \arctan\left(\frac{A_X}{\sqrt{A_Y^2 + A_Z^2}}\right)$$

$$\phi = \arctan\left(\frac{A_Y}{\sqrt{A_X^2 + A_Z^2}}\right)$$

$$\theta = \arctan\left(\frac{\sqrt{A_X^2 + A_Y^2}}{A_Z}\right)$$

Technical task … send in accel values X, Y and Z.

See also:

- Accelerometers
- YouTube – ESP32 Technical Tutorials: MPU6050 Accelerometer
- Tilt Sensing Using a Three-Axis Accelerometer
- MPU-6050 Data Sheet
- MPU6050 Register Map and Descriptions
- MPU-6050 Accelerometer + Gyro
- I2Cdevlib – MPU-6050
- Filters
- InvenSense – MPU-6050 Home Page
- Gyroscopes and Accelerometers on a Chip
- Using an accelerometer for inclination sensing
- YouTube: MPU-6050 Data with a Complementary Filter
- Digi-Key: How the latest MEMS inertial modules help overcome application development challenges

The math of accelerometers

Our real world is composed of three spatial dimensions. In English we might think of these as left/right, up/down and in/out (or forward/back). In our math, we will label these by the common x, y and z. If we want to think about the orientation of something, we can think of it as the rotation about these axis. We define the rotation around the x axis as "roll" designated by the symbol "Φ". We define the rotation around the y axis as "pitch" designated by the symbol "Θ" and finally, we define the rotation around the z axis as "yaw" designated by the symbol "Ψ". So, to think about the orientation of an object, we can think about its roll, pitch and yaw.

Axis	Name	Symbol
x	roll	Φ
y	pitch	Θ
z	yaw	Ψ

Now, let us merge in the notion of how we are going to measure our roll, pitch and yaw. Assuming no acceleration of our device (I.e it is stationary) then there will be a constant force on it of 1g in the z axis. So the force measured in the z axis will be 1 and x and y will both be 0. As we tilt the device from its horizontal orientation, that force will start to be distributed across multiple axis and not just the z axis. So a tilt will produce a new measured force value in both x and y. And here comes the magic. The measured force change is proportional to the roll and pitch angles. But wait, what about the yaw? Well … if we think about it, a rotation around the z axis does not change the tilt of what we want to measure. Think of it like this, if we have a perfectly flat table top and place some marbles on its surface, if we tilt the table left or right, the marbles will roll. If we tilt it forward or back, the marbles will roll. However, if we are gentle and rotate the table around its central vertical axis, the marbles will not roll. They will remain in their same position relative to the table.

Another way to think about this is to hold your cell phone in your hand flat while sitting on a chair. If you now spin around on your chair, have you changed the "tilt" of your cell phone? I would say no.

What this means to us is that we can calculate the orientation of our device only by figuring out the roll and pitch. The yaw becomes immaterial to our story (for just now).

By some math, we can find that:

$$\tan(\Phi)=\frac{G_{py}}{G_{pz}}$$

$$\tan(\Theta)=\frac{-G_{px}}{\sqrt{G_{py}^2+G_{pz}^2}}$$

See also:

- Using an Accelerometer for Inclination Sensing
- Understanding Euler Angles

Visualizing orientation

It is all very good getting raw numbers representing the amount and direction of force on our device against different axis, but triplets of numbers isn't at all intuitive to our ways of thinking. Instead, what we want is a mechanism to "see" the orientation of the device. Since orientation is inherently a three dimensional physical consideration, what we want is to visualize the data as a 3D scene. There are many 3D programming packages on the Internet with varying degrees of sophistication and capabilities. Since my belief is that all user interfaces should be browser based as opposed to thick client, I wanted one that could run in a browser. I also wanted one that is open source. For this I chose "`three.js`".

See also:

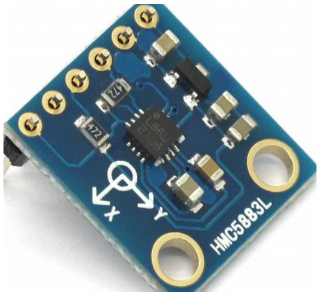
- three.js

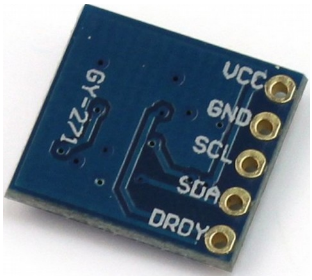
Compass – HMC5883L (aka GY-271) (aka CJ-M49)

In the real world, a compass is a device which can be used to determine the direction of magnetic north. There are also electronic components that can perform the same task. One such component is the HMC5883L (which has been superseded by the HMC5983 but both are compatible with each other). Strictly speaking, the device measures the intensity of any magnetic field around it. Since field strength is a vector quantity (has both magnitude and direction), we will be measuring the field strength in the X, Y and Z axis. The going price for one of these on eBay is about $3.

The calculation of a compass bearing is affected by the relative tilt of the device. What this means is that if the device is tilted the compass bearing result will be wrong. We can compensate for this if we know the amount of tilt. This can be calculated from an MPU-6050.

This is a 3.3V device.





The pin configuration for GY271 is (Left-Right as seen from chip side with connector at top):

Pin	Function
DRDY	Data ready/Interrupt
SDA	I2C Data line
SCL	I2C Clock line
GND	Ground
VCC	+ve voltage – 3.3V

The pin configuration for the CJ-M49 is (Left-Right as seen from chip side with connector at top):

Pin	Function
`3V3`	+ve voltage – 3.3V
`DRDY`	Data ready/Interrupt
`SDA`	I2C Data line
`SCL`	I2C Clock line
`GND`	Ground

`VCC`
`+5V`
	+ve voltage – 5V

The module makes its data available via the I2C bus. The I2C address of the device is `0x1E`. If we run an I2CScanner we will see the device as being present:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 1e --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

From I2C there are a set of registers that can be accessed:

Address	Description	Access
00	Configuration register A	Read / Write
01	Configuration register B	Read / Write
02	Mode register	Read / Write
03	Data output X MSB register	Read
04	Data output X LSB register	Read
05	Data Output Z MSB register	Read
06	Data Output Z LSB register	Read
07	Data output Y MSB register	Read
08	Data output Y LSB register	Read
09	Status register	Read
10	Identification register A	Read
11	Identification register B	Read
12	Identification register C	Read

Think of the device as having a register cursor. We can move the cursor simply by sending the id of the register to move to. For example, sending the device `0x03` will move the cursor to register `3`. Subsequent reads will read from the cursor location **and** auto-increment the cursor. So moving to register `3` and reading `6` bytes will read the next `6` register values [`0x03-0x08`].

Configuration Register A (register `00`) is as follows:

Name	**Bits**	**Description**
	7	Reserved. Must be 0.
MA	6:5	Averaged samples per measurement: ● 00 = 1 (default) ● 01 = 2 ● 10 = 4 ● 11 = 8
DO	4:2	Data output rate in measurements/second: ● 000 = 0.75 ● 001 = 1.5 ● 010 = 3 ● 011 = 7.5 ● 100 = 15 (default) ● 101 = 30 ● 110 = 75 ● 111 = Reserved
MS	1:0	Measurement configuration ● 00 – Normal (default) ● 01 – Positive bias ● 10 – Negative bias ● 11 – Reserved

Configuration Register B (register 01) is as follows:

Name	Bits	Description
GN	7:5	Gain values: ● 000 = 1370 ● 001 = 1090 (default) ● 010 = 820 ● 011 = 660 ● 100 = 440 ● 101 = 390 ● 110 = 330 ● 111 = 230
	4:0	Reserved. Must be 0.

Mode Register (02)

Name	Bits	Description
HS	7	High speed I2C control
	6:2	Reserved. Must be 0.
MD	1:0	Operating mode: ● 00 = Continuous ● 01 = Single measurement (Default) ● 10 = Idle mode ● 11 = Idle mode

The three identification registers return the ASCII values 'H' (0x48), '4' (0x34) and 'C' or 'H' (0x48), '4' (0x34) and '3' (0x33)

To create an angle in degrees, the following formula can be used:

```
angle = atan2((double)y,(double)x) * (180 / 3.14159265) + 180; // angle in degrees
```

The driving logic at a high level is:

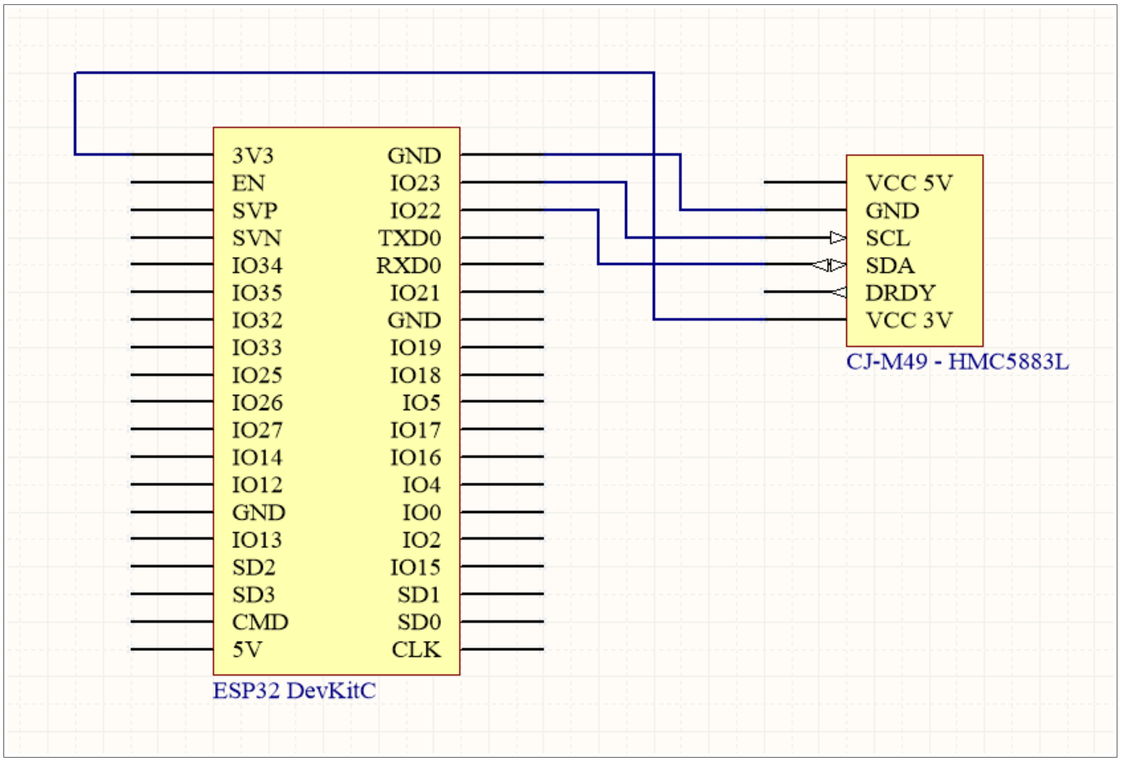
Function	Description
Write 0x02	Select mode register
Write 0x00	Continuous measurement mode
Write 0x03	Select register 3
Read 6 bytes	X, Z, Y value pairs MSB/LSB

we combine MSB and LSB of data with:

```
data[0] << 8 | data[1];
```

Attachment of the device to ESP32 is particularly easy. As you can see it only needs 4 wires to connect.

Here is a schematic of how an instance of the board may be wired to ESP32.



When we use a compass, the result we usually want is "which way is North". On a real physical compass, we have a needle that points in that direction and we know our answer. Through a digital device, what we should expect to get back is an angle that the device would have to be oriented for it to point North. This assumes that there is a reference mark on the device from which the angle has meaning.

When experimenting with the device, don't bring strong magnets too close to the device as it may become magnetized or otherwise damaged.

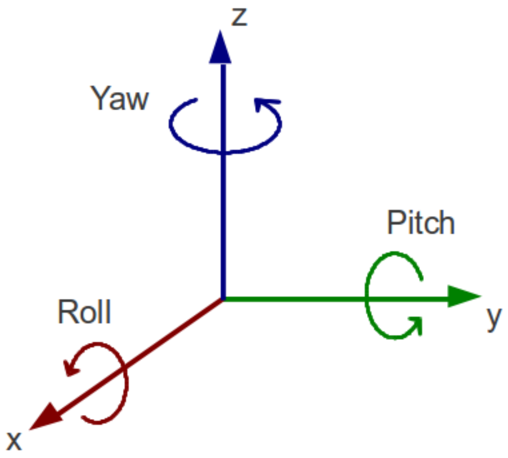
See also:

- Data sheet
- Sparkfun Tutorial
- Arduino Nano + GY-271 (Digital Compass module) + OLED
- Electrodragon - HMC5883L - Three-Axis Compass
- YouTube – ESP32 Technical Tutorials: HMC5883L – Compass
- YouTube – Arduino How To: HMC5883L Compass Magnetometer Tutorial
- YouTube – Use the HMC5883L 3-axis sensor with an Arduino – Tutorial

- YouTube – Let's build an Arduino electronic Compass using the HMC5883L and a Ring of LEDs - Tutorial
- Wikipedia – atan2
- Github – Arduino Library

Tilt compensation of the compass

When the compass module is perfectly planar, it returns a good value but if it is tilted, then the value is no longer true. Look at the following diagram and imagine your compass center in a perfect horizontal plane. At that time, there will be no roll and no pitch. For our purposes, the yaw of the board will be pointing vertically straight up and we don't care. The value of the compass will be correct. However if the board is tilted with roll and pitch, then the compass value is no longer accurate. However, given then value measured from the compass and if we have the values of roll and pitch, we can compensate and calculate a correct compass value.



Again by looking at the above image, consider an accelerometer. This will measure the force of gravity and gives us a reference to the orientation of the compass sensor. When the sensor is horizontal, the acceleration measured will be exclusively down the Z axis however when the sensor is titled, the acceleration will be distributed across X, Y and Z and by using math, we can determine the orientation of the board. As a side note, when the board is horizontal, the compass bearing will be the yaw value.

We will define the following definitions:

Φ – phi	Roll (X)
Θ – theta	Pitch (Y)
Ψ – psi	Yaw (Z)

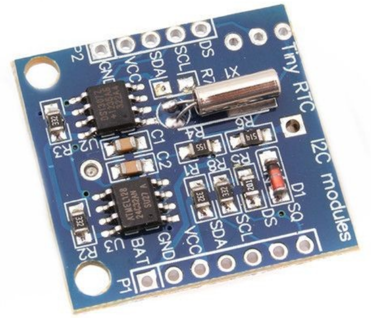
See also:

- Implementing a Tilt-Compensated eCompass using Accelerometer and Magnetometer Sensors
- Wikipedia: Axes conventions

Real time clocks

When we start an ESP32, it doesn't know what the real time is. We can tell it once booted but even then, once we power it off, it will forget again. If the ESP32 is connected to the Internet, it can learn the real time by using a network time protocol. However, if our ESP32 is not connected to the Internet and is network isolated, by default, it has no mechanism to know the current time. This is what we can add a real-time-clock (RTC) module. A real-time clock module is a small piece of electronics that contains a battery and is taught the current date and time. Because it has an on-board battery, it can remember the time even when the ESP32 is powered off. When the ESP32 boots, it can be configured to ask the RTC module for the current date/time and use that from then on.

There are many RTC modules available but one of the more prevalent is the DS1307.



These can be picked up on eBay for about $1. The RTC uses battery backup when not powered. This is a standard CR2032 type. When inserting the battery, it is +ve down. When soldering on header pins, think through how you will mount the board when complete. It may be that you want to solder the header pins such that you can easily access and replace the battery once attached to your PCB or strip board.

This device provides a real time clock capability that includes hours, minutes, seconds, month, day of month, day of week and year. Month calculations including leap year support are also accommodated. Interestingly, it also provides 56 bytes of non-volatile RAM. The device uses I2C for communication.

Note: Before plugging your board to the ESP32, realize that this is a 5V module. If you connect 5V to the ESP32 GPIO pins bad things can happen because they are 3.3V pins. The I2C protocol uses pull-up resistors to bring the SDA and SCL lines high by default. If we were to look at the schematic diagram of this module we would find that resistors R2 and R3 are pull-up resistors. Unfortunately, we have two problems. First is that the ESP32 already pulls up the pins and secondly the module pulls them high to a 5V line which is too much for the ESP32. The solution is to physically remove these two resistors from your module before use. The resistors are clearly marked on the board as R2 and R3 and carry the mark 332 (3.3k).

Since the device has battery backup, an interesting question is how long can this module keep time for? If we examine the data sheet, we find that it consumes 500nA while running on battery. If we understand that a typical CR2032 battery has a capacity of 200mAh then, if my math is correct, this works out to be about 45 years … which is well past the life of the battery itself.

The boards have 5 pins on one side 7 on the other.

Pin	Label	Description
1	DS	DS18B20 Temp.
2	SCL	I2C clock.
3	SDA	I2C data line.
4	Vcc	Vcc (5V).
5	GND	Ground.

Pin	Label	Description
1	SQ	Square wave output.
2	DS	DS18B20 Temp.
3	SCL	I2C clock.
4	SDA	I2C data line.
5	VCC	Vcc (5V).
6	GND	Ground.
7	BAT	Battery voltage

Once plugged in, we will see the RTC at I2C address `0x68`:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

The device has registers contained within it that are laid out as follows:

Address	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Function	Range
0x00	CH	10 Seconds			Seconds				Seconds	00-59
0x01	0	10 Minutes			Minutes				Minutes	00-59
0x02	0	0 – 24	10 hour	10 Hour	Hours				Hours	1-12 AM/PM 00-23
		1 – 12	AM/PM							
0x03	0	0	0	0	0	Day			Day	01-07
0x04	0	0	10 Date		Date				Date	01-31
0x05	0	0	0	10 Month	Month				Month	01-12
0x06	10 Year				Year				Year	00-99
0x07	OUT	0	0	SQWE	0	0	RS1	RS0	Control	
0x08 - 0x3f									RAM	0x00 - 0xFF

The encoding of the data is known as "Binary Coded Decimal" (BCD). Each decimal digit is encoded as a 4 bit representation. For example, the decimal number "34" would be encoded as 0011 1000. The first four bits representing "3" and the last four bits representing "4".

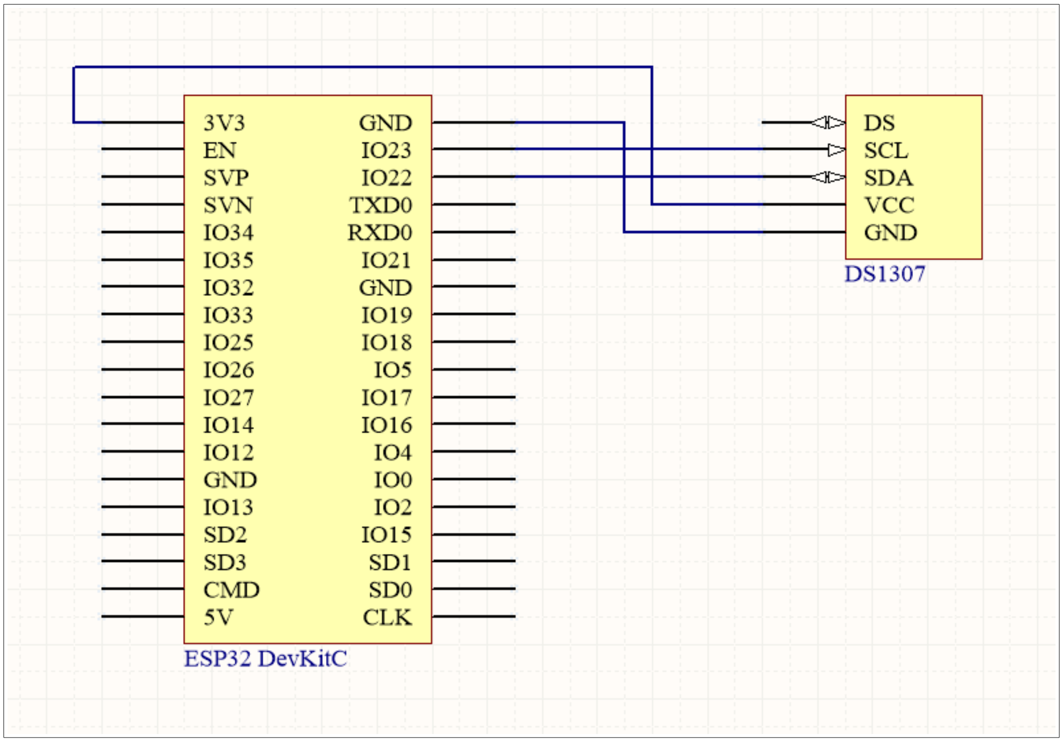
To convert a two digit decimal valued integer to a BCD representation we would use the expression:

```
bcdValue = ((num / 10) << 4) | (num%10)
```

to convert from a BCD to an integer, we would use the expression:

```
num = (bcdValue >> 4) * 10 + (bcdValue & 0x0f)
```

In addition to performing the service of a real time clock, this device also provides 56 bytes of battery backed up RAM. While 56 bytes doesn't sound like much, it could be used to hold fourteen 32 bit integer values or perhaps an SSID/password pair.



See also:

- YouTube – ESP32 Technical Tutorials: DS1307 Real Time Clock
- Tutorial – Using DS1307 and DS3231 Real-time Clock Modules with Arduino
- Maxim integrated datasheet

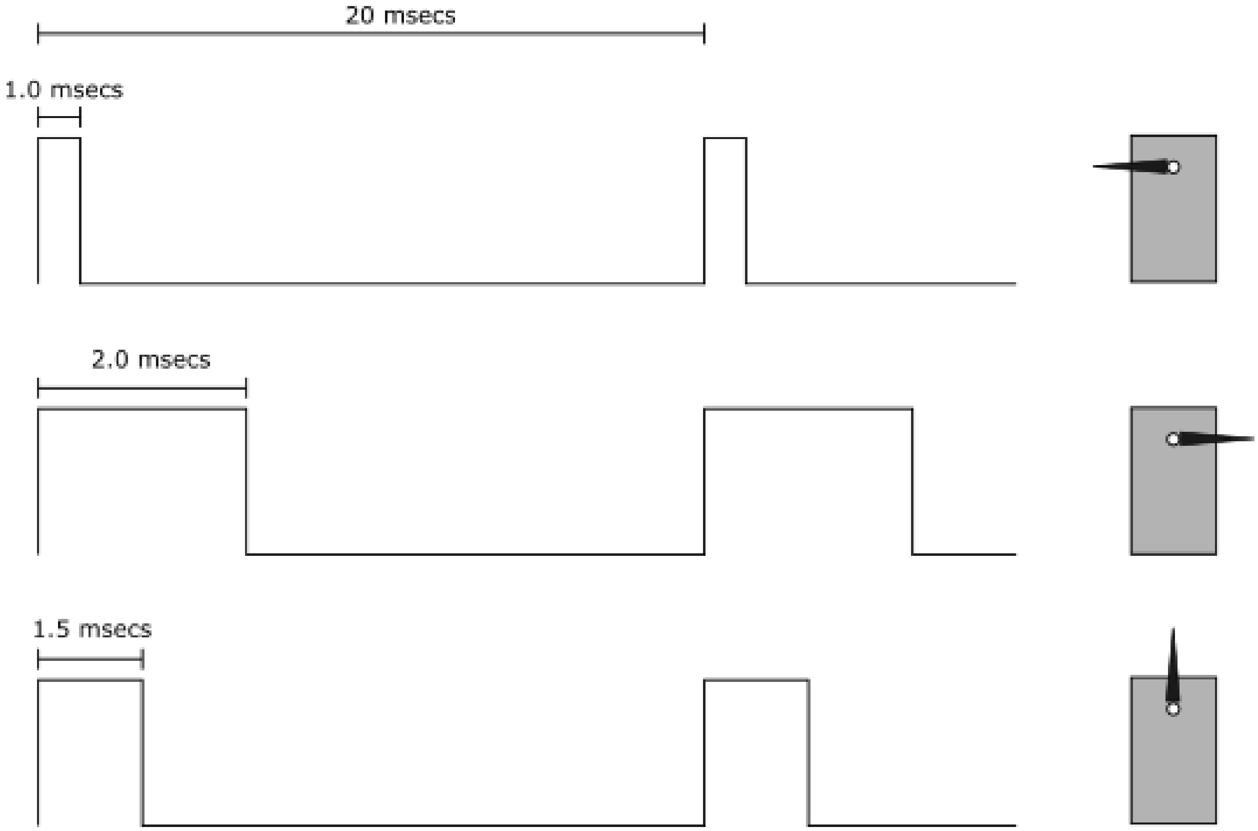
Servos

A servo is a physical movement device similar to an electric motor. Typically, a servo has a rotation shaft that can rotate from 0 degrees to 180 degrees as a function of the incoming signal applied to it. If one sets the signal to a specific value, the shaft will rotate to a specific position as a function of that signal. The actual minimum and maximum angles are specified by the manufacturer of any particular servo model.

The signal expected by a servo is encoded as Pulse Width Modulation (PWM). One should always consult the manufacturers data sheet to find the specific values. However, in general, the story works as follows.

Typically, the period of the PWM is 20 milliseconds. If we think this through, we will realize that 20ms is 50Hz as there are 50 instances of 20ms periods in 1 seconds (1000ms). At the start of the period, the duration that the signal is high determines the rotation of the shaft. For example, it may be defined that there is a minimum value of 1 millisecond and a maximum value of 2 milliseconds. This means that if the signal is high for 1 millisecond, the shaft rotates to its minimum value. If the signal is high for 2 milliseconds, the shaft rotates to its maximum value. For high durations between 1 and 2 milliseconds, the shaft rotates proportionally to that duration.

Here is a diagram (not drawn to scale) to illustrate the principle:



Again, it is important to read the data sheet to find the correct values for minimum and maximum duty cycle widths.

We also need to remember that the voltage output from a ESP32 GPIO pin is 3.3v. Most of the servos require a signal at the 5v level. This means that we can't directly drive a servo's data signal input from a ESP32 pin. We will need to employ a logic level shifter to shift the output PWM signal from 3.3V up to 5V.

If we use the ESP-IDF drivers and we imagine that the minimum duty cycle is 1ms and the max is 2ms with a period of 20ms, then we see that the range of duty cycle is 1ms which is 1/20th of the period. This then gives us the granularity of adjustment for the servos by looking at the different bit sizes available for the PWM timers:

Bit Size	Granularity (1ms spread)	Low (1ms)	High (2ms)
10	51	51.2	102.4
11	102	102.4	204.8
12	204	204.8	409.6
13	408	409.6	819.2
14	819	819.2	1638.4
15	1638	1638.4	3276.8

See also:

- LEDC – Pulse Width Modulation – PWM

The Mini/Micro SG90

The mini/micro SG90 servos can be found on eBay for under $2 each. These servos weight only 9g. They have a stall torque of 1.8kg·cm, operate at 5v, have a speed of about 0.3 seconds for 180° and have discrete step instances of 10µs. The stalled current draw on one of these servos is reported to be 600mA.

Color	Function
Red	Vcc
Brown	Ground
Orange	PWM

See also:

- Mini/Micro SG90 – Data sheet

Audio

PCM5102 - I2S DAC

This device takes as input an I2S signal and produces an analog audio output.

Pin	Function
VCC	
3.3V	
GND	
FLT	Filter select
DMP	De-emphasis
SCK	System clock input
BCK	Audio data bit clock input
DIN	Audio data in
LCK	Audio data word clock input
FMT	Audio format selection
XMT	Mute control

See also:

- I2S Bus
- Data sheet

Graphic Equalizer

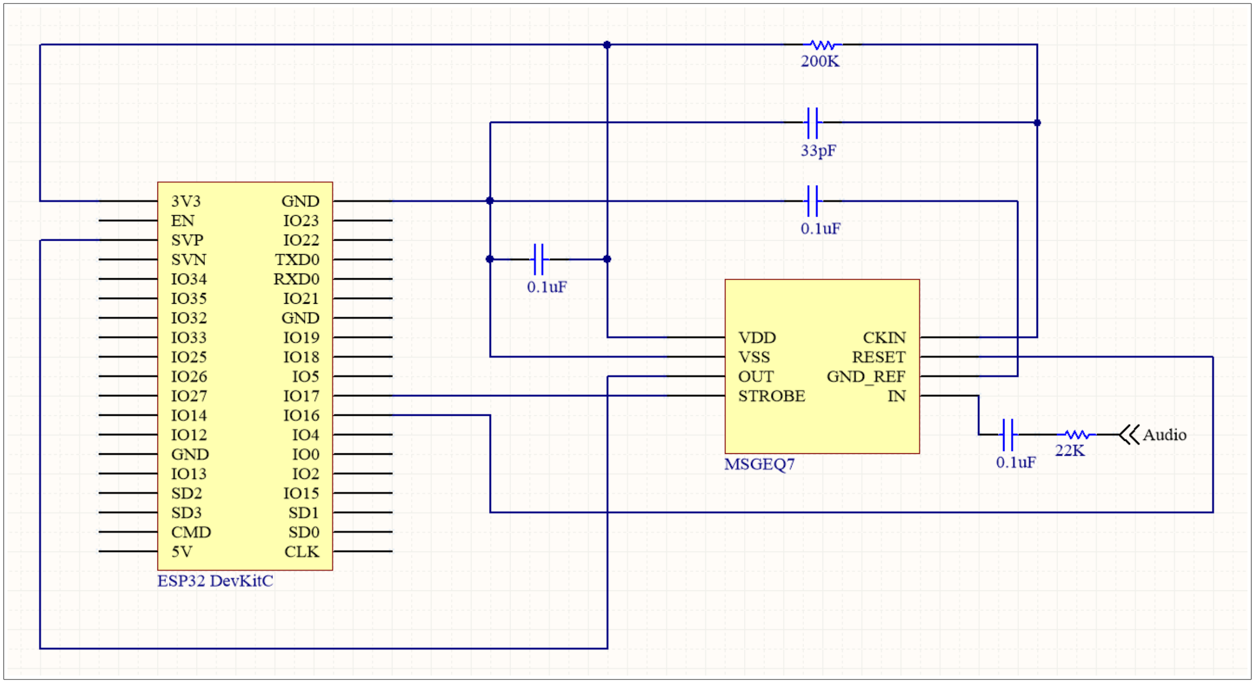
An integrated circuit called the MSGEQ7 is a graphic equalizer. It can operate at 3.3V or 5V.

If we provide it an audio input signal, it will analyze the signal into 7 discrete frequency bands of 63Hz, 160Hz, 400Hz, 1KHz, 2.5KHz, 6.25KHz and 16KHz.

The integrated circuit is supplied in an 8 pin configuration with:

Pin	Function	Description
1	VDDA	3.3-5V power supply
2	VSSA	Negative
3	OUT	Multiplexed DC Output. This is an analog output value.
4	STROBE	Channel selection
5	IN	Audio input
6	GND_REF	
7	RESET	Reset. A high resets the multiplexor, low enables the strobe.
8	CKIN	Clock. This is passive component clock circuit.

VDDA	CKIN
VSSA	RESET
OUT	GND_REF
STROBE	IN



Yellow reset - 16

Blue Strobe - 17

Green Out

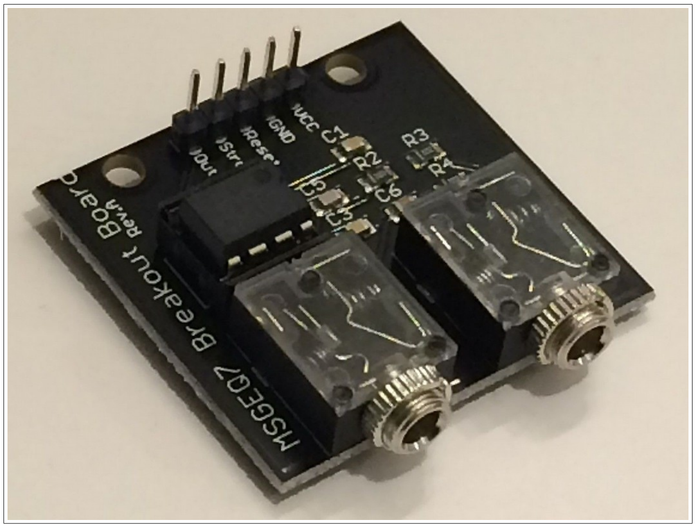
[3.3V] [GND] [OUT - GREEN – GPIO 36] [STROBE BLUE – GPIO 17] [RESET – YELLOW - GPIO 16]

The high level of operation is that we apply our audio signal in "`IN`" pin. Normally, the `RESET` pin is low. When it goes high and then low again, the decoding of the signal is latched. Now we can read data from the "`OUT`" pin which is an analog value. The `STROBE` pin is initially high. When it goes low, we can read the value of `OUT`. When it goes high and then low again, we will have stepped onto the next frequency.

A high level algorithm would be:

```
pinMode(RESET, OUTPUT);
pinMode(STROBE, OUTPUT);
pinMode(OUT, ANALOG);

digitalWrite(RESET, LOW);
digitalWrite(STROBE, HIGH);

digitalWrite(RESET, HIGH);
delay(1);
digitalWrite(RESET, LOW);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v63hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v160hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v400hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v1000hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v2500hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v6250hz = analogRead(OUT);
digitalWrite(STROBE, HIGH);
delay(1);

digitalWrite(STROBE, LOW);
delay(1);
v16000hz = analogRead(OUT);
```

For an easy wiring solution, breakout modules exist. These provide an MSGEQ7, pin outs, passive components and signal input sockets on a single board ready to be wired in.



The pins exposed are:

- `VCC` – Power
- `GND` – Ground
- `RESET` – Reset pin (input)
- `STROBE` – Strobe Pin (input)
- `OUT` – Analog output

See also:

- Datasheet
- MSGEQ7-Based DIY Audio Spectrum Analyzer: Construction

External networking

It may seem strange to include additional networking hardware interfacing given that the ESP32 is a first class WiFi and Bluetooth device however there are times when we want to performing networking which doesn't fit either of those categories. While WiFi and Bluetooth are great, you may have to interact remotely with devices which don't support those protocols. A classic example are the cheap nRF24 modules from Nordic. These devices are reported to easily be able to transmit and receive over 300 feet and hence become ideal components for longer distance communications.

The nRF24

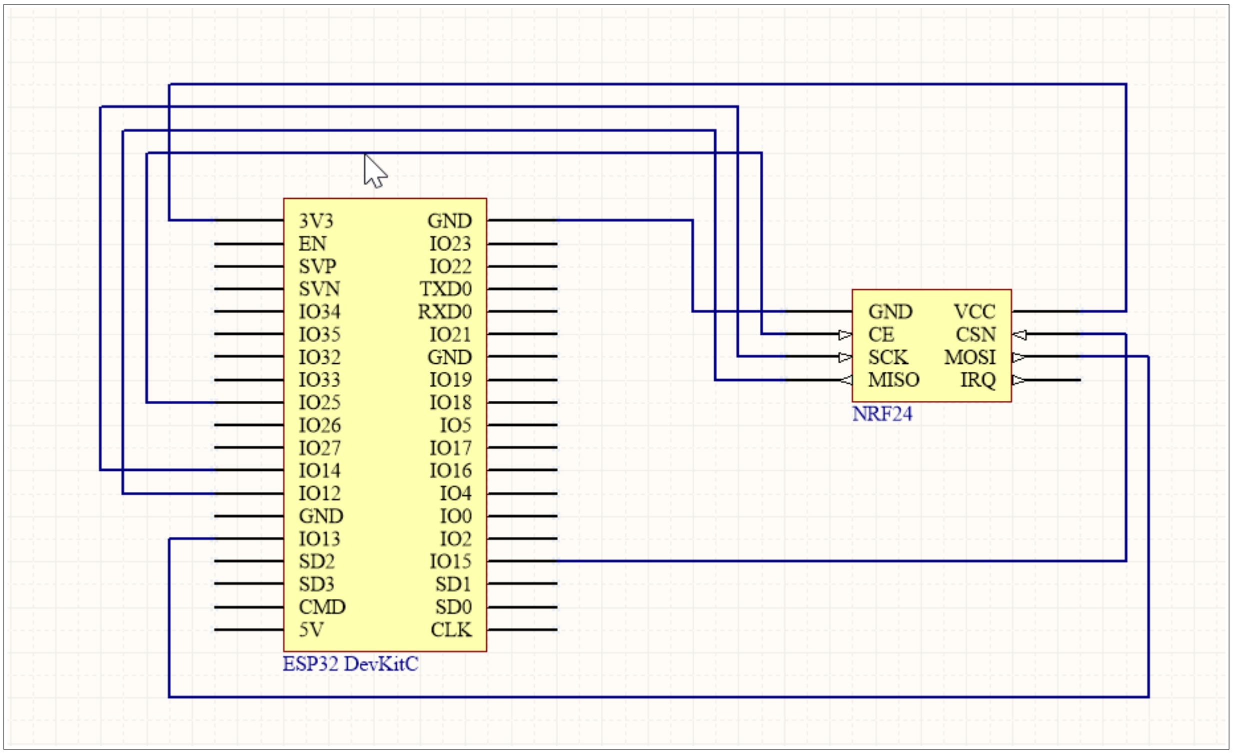
The nRF24L01 is an extremely cheap communication device. It allows one to transmit and receive data over large distances (hundreds of feet). It transmits and receives packets between similar devices. Do not confuse it with WiFi or Bluetooth. You can transmit from one nRF24 to another nRF24 but not to anything else. There are a number of reasons to consider this device for projects over others. First is the very low cost. An instance of the device can be picked up for less than a dollar. Second is the peer-to-peer nature of the devices. There does not need to be a master/slave relationship in the story. In fact, two or more devices can exchange packets with each other without any of them acting as controllers. There are occasions where this is exactly what is needed.

The device operates on 3.3V and uses SPI as a protocol. The pin-out on the nRF24L01 is:

Pin	Description
Vcc	3.3V
GND	Ground
MISO	SPI Master In/Slave Out.
MOSI	SPI Master Out/Slave In.
SCK	SPI System clock.
CSN	SPI Chip Select – Enable the device (active low). Normally this will be wired to ground.
CE	RX or TX mode. The Data sheet calls this Chip Enable which seems an awfully strange name for a communication mode selector. CE=1 for transmit while CE=0 for receive.
IRQ	Maskable interrupt.

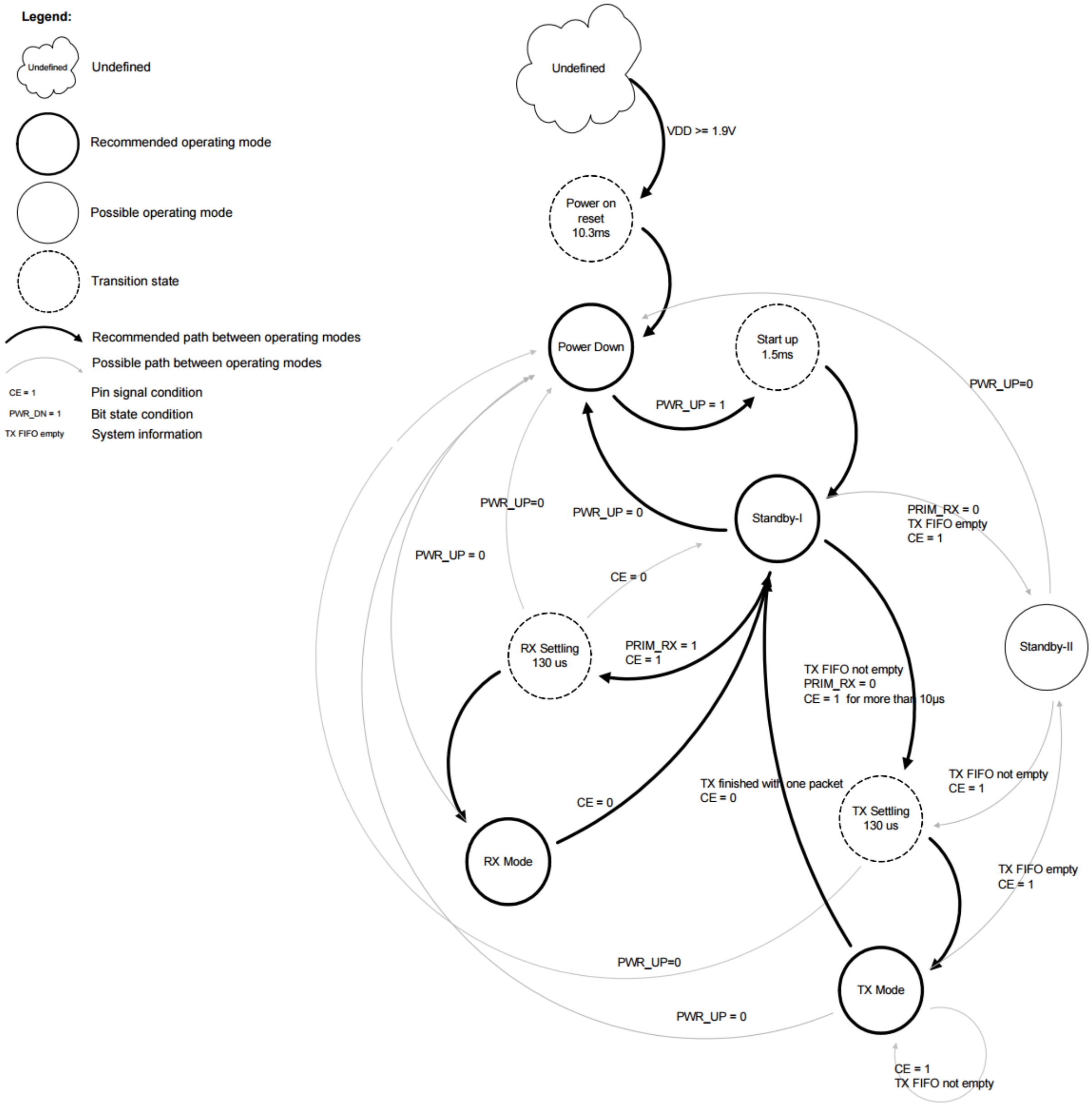
Seen from above, the pins are:

GND	VCC
CE	CSN
SCK	MOSI
MISO	IRQ



Some of the key specifications of this component include up to 2Mbps transmission rate and low power consumption. It operates in the 2.4GHz wireless band.

The device operates in a variety of different modes … here is the state transition diagram for them which taken from the data sheet:



The first mode is "Power Down". This is the lowest power mode while still maintaining state/configuration.

The second mode is called "Standby". Again in this mode it is consuming low amounts of power but not as much as "Power Down". The difference between this mode and "Power Down" is how quickly it can begin transmitting and receiving.

The third mode is called "RX". In this state the device is a receiver.

The final mode is called "TX" and is entered when the device is transmitting data.

The modes are entered through:

	Power Up	CE	PRIM_RX	TX FIFO
Power Down	0	-	-	
Standby	1	0	-	
RX	1	1	1	
TX	1	1	0	>0

There are two data transmission rates … 1MBps and 2MBps. It is vital that both the transmitter and receiver devices use the same data rate.

The communications protocol used by the nRF24 devices is called "Enhanced ShockBurst". It has a dynamic payload length of between 1 and 32 bytes. What this means is that a single transmission can can contain a maximum of 32 bytes of payload data. If we need to transmit additional data, we would send a stream of packets one after the other. Automatic transaction handling is also provided. We should think of a transaction as the request from the transmitter to send a packet and have the receiver pick it up. It is possible that there may be errors in the transmission. The transaction solution is to hold the transmitted data until there is a positive acknowledgment that the receiver has actually received it. If that doesn't happen, the transmitter can re-transmit the packet.

To set the size of the payload of a packet, we have two choices available to us … these are static and dynamic payload sizes. For the static payload size, the receiver specifies the size of the payload and the sender must send that size before a transmission is complete. For dynamic payload, the transmitter can send a variable number of bytes before that packet is acknowledged as received by the receiver.

Each device has a configurable address. When data is transmitted from one device to another, the transmitter specifies the address of the receiver. An address can be between 3 and 5 bytes in length.

Devices communicate on a "channel". Think of this as a specific radio frequency bandwidth chunk. In order for a transmitter and receiver to talk to each other, they must both agree on the channel to be used. A channel is 7 bits in length (0 to 127). The default is channel 2.

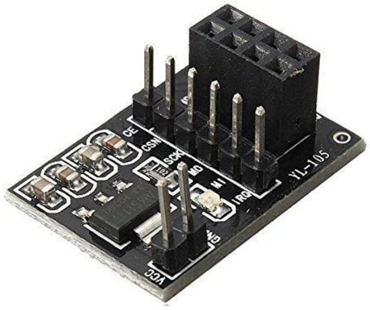
The nRF24 has a variety of communication speeds including 1Mbps and 2Mbps.

A Cyclic Redundancy Check (CRC) can be applied to the payload to assist in detection of transmission errors. The CRC size can be set to be 8 bits or 16 bits.

The transmission power can be adjusted. The more transmission power you request the more overall power will be consumed and if you are running from batteries, this may be a consideration. There are four settings:

Output power	Current consumption
0dBm	11.3mA
-6dbM	9.0mA
-12dBm	7.5mA
-18dBm	7.0mA

A handy little addition to a nRF24 module is a YL-105. This is a connector board that hosts a nRF24 but also exposes the pins on the top in a line. One of the issues (at least to me) when working with nRF24s is that they have the pins pointing downwards and I have to do mental mirror imaging to get things right (and don't always).



The pins are:

- CE
- CSN
- SCK
- MOSI
- MISO
- IRQ

and two further pins for:

- VCC
- GND

When pairing a couple of nRF24s, I find it useful to create a spreadsheet with two columns; one for each of the devices. We can then validate that the settings necessary to be paired on both are correct.

Attribute	Device A	Device B
Addr1	TX:	RX:
Addr2	RX:	TX:
Address Width		
RF_CH		
Data Rate		
CRC Length		

Notes:

- The air data rate must be the same on all participants.
- The RF channel frequency must be the same for all participants.

See also:

- Nordic Semi – Home Page
- Data sheet
- SPI – Serial Peripheral Interface
- Tutorial: Ultra Low Cost 2.4 GHz Wireless Transceiver with the FRDM Board
- Julian Ilett #1
- NRF24L01 - How To
- Adding a nRF24L01 to a breadboard or stripboard
- Tutorial 0: Everything You Need to Know about the nRF24L01 and MiRF-v2
- YouTube: Getting started with the nRF24L01

Using the Arduino APIs

An Arduino library for the nRF24 is available called RF24. This library is highly active and well documented. It appears to be a fork of the RF24 library originally created by Manicbug. It has a detailed set of documentation including descriptions of all the class APIs.

A typical flow would be:

```
RF24 radio(CE_PIN, CSN_PIN);
radio.begin();
radio.setPALevel(RF24_PA_LOW);
radio.openWritingPipe(addressTX);
radio.openReadingPipe(1,addressRX);
// To be a receiver ...
radio.startListening();
```

To stop being a receiver, call:

```
radio.stopListening();
```

Since there are only two modes, a receiver and a transmitter, when one stops being a receiver, one immediately becomes a transmitter.

To determine if there is data available to receive, call:

```
radio.available()
```

To read data, we can call:

```
radio.read(&buffer, length);
```

To write, we can call:

```
radio.write(&buffer, length);
```

The library comes with some samples.

- `GettingStarted` – Two devices. One can act as the transmitter and one as the receiver. They can be flipped around through user input from the serial port.

Before the library can be used, an instance of it must be constructed. The constructor looks as follows:

```
RF24(cePin, csPin)
```

The Arduino pins for `ce` and `cs` need not be fixed and hence we must instruct the library as to which pins we actually used. Take care that when an instance is created you specify the pins in the correct order as there is no guard against that.

Before any other functions can be called, we must call `begin()`. This perform the initialization of the device plus sets defaults for many of the operational parameters.

Every device has an address associated with it. This allows a transmission from one source device to be addressed to a specific destination device. When a transmission occurs, devices other than the one with the matching address will ignore the transmission. The device requires the size of the address to be supplied and fixed. The allowable address sizes are 3, 4 or 5 bytes. The default is 5 bytes. The size of the address can be changed with the `setAddressWidth()` method however you are unlikely going to want to change the default value. You must remember to supply the full data for the address. For example, don't set an address of "`123`" when the address size is 5 bytes as your device address will become "`123<?><?>`" and you will likely waste time trying to find out why no data is coming to you.

At this point, we must consider what we are going to do with the device from here on. The device has two modes of operation. It can transmit data or it can receive data but it can't do both simultaneously.

Let us look first at being a transmitter. First we create a "pipe" over which the data will flow. We supply the address of the destination.

```
openWritingPipe(const unit8_t *address)
```

For example:

```
unit8_t address = "1Node";
openWritingPipe(address);
```

We are now ready to actually transmit some data.

```
bool write(const void *data, uint8_t length);
```

This method will transmit the data pointed to by the data pointer for the number of bytes supplied in length. There is a maximum size as specified by the `getPayloadSize()` method and you should not exceed that. The method will block until the data has been transmitted or the transmission request has timed out. The time out is short, about 70ms so you won't block for long. Upon return from the method call, we can determine whether the transmission was successful or if it failed. A return value of `true` means we succeeded while `false` means as failure.

The device has a maximum transmission size that is 32 bytes. You can not transmit packets larger than this. The default maximum transmission set in the library is 32 bytes which is the maximum that the device will permit. You can change this with the `setPayloadSize()` method but it is unlikely you will need that function. Simply remember that we can not ever send a payload size greater than 32 bytes.

The other mode of the device is that of a receiver where we actually listen for incoming data. As mentioned, the device can either be in transmit mode or receive mode.

To receive incoming data, we need to supply the address of the device from which we are expecting incoming data to arrive. We do this using the `openReadingPipe()` method.

```
openReadingPipe(uint8_t number, const uint8_t *address)
```

Once we have specified the address of the transmitting device we will receive data from, we can start actively listening using the `startListening()` method. Should we wish to end listening mode, we can call the `stopListening()` method which will return us to transmission mode.

Once we have entered listening mode, we can ask the library if there is data available for us to read. The method for this is `available()`. If there is data available, the return value is true and false otherwise.

To actually read the data that was received, we can use the method called `read()`.

```
read(void *buf, uint8_t len)
```

This method supplies a buffer into which the received data will be stored. The length of the buffer is also supplied.

A debug file called `printDetails()` is provided which logs the state of the nRF24 to the `stdout`. In order to use this one must:

- Include `<printf.h>`
- Call `printf_begin()`

When the `printDetails()` function is then called, a status of the nRF24 is written to the Serial port. This can be a powerful tool for debugging problems. Here is an example of the output:

```
STATUS       = 0x0e RX_DR=0 TX_DS=0 MAX_RT=0 RX_P_NO=7 TX_FULL=0
RX_ADDR_P0-1 = 0x0045444f4e 0x0045444f4e
RX_ADDR_P2-5 = 0xc3 0xc4 0xc5 0xc6
TX_ADDR      = 0x0045444f4e
RX_PW_P0-6   = 0x20 0x20 0x00 0x00 0x00 0x00
EN_AA        = 0x3f
EN_RXADDR    = 0x02
RF_CH        = 0x4c
RF_SETUP     = 0x03
CONFIG       = 0x0f
DYNPD/FEATURE= 0x00 0x00
Data Rate    = 1MBPS
Model        = nRF24L01+
CRC Length   = 16 bits
PA Power     = PA_LOW
```

The complete API can be found documented at the project's web site but here is a quick summary:

Method	Notes
bool begin()	
void startListening()	Enter receiving mode.
void stopListening()	Enter transmitting mode.
bool available()	Returns true if there is data available to be read.

void read(
  void *buf,
  uint8_t len)
	Read data.

bool write(
  const void *buf,
  uint8_t len)
	One must call stopListening() before writing.

void openWritingPipe(const uint8_t *address)	Write data.

void openReadingPipe(
  uint8_t number,
  const uint8_t *address)	

void setAddressWidth(uint8_t addressWidth)	

void setRetries(
  uint8_t delay,
  uint8_t count)	

void setChannel(uint8_t channel)	
uint8_t getChannel()	
void setPayloadSize(uint8_t size)	
uint8_t getPayloadSize()	
uint8_t getDynamicPayloadSize()	
void enableAckPayload()	
void enableDynamicPayloads()	
void enableDynamicAck()	
bool isPVariant()	
void setAutoAck(bool enable)	
void setAutoAck(uint8_t pipe, bool enable)	
void setPALevel(uint8_t level)	
uint8_t getPALevel(void)	
bool setDataRate(rf24_datarate_e speed)	Values of the enumeration are: • RF24_1MBPS • RF24_2MBPS • RF24_250KBPS
rf24_datarate_e getDataRate()	Values of the enumeration are: • RF24_1MBPS

	• RF24_2MBPS • RF24_250KBPS

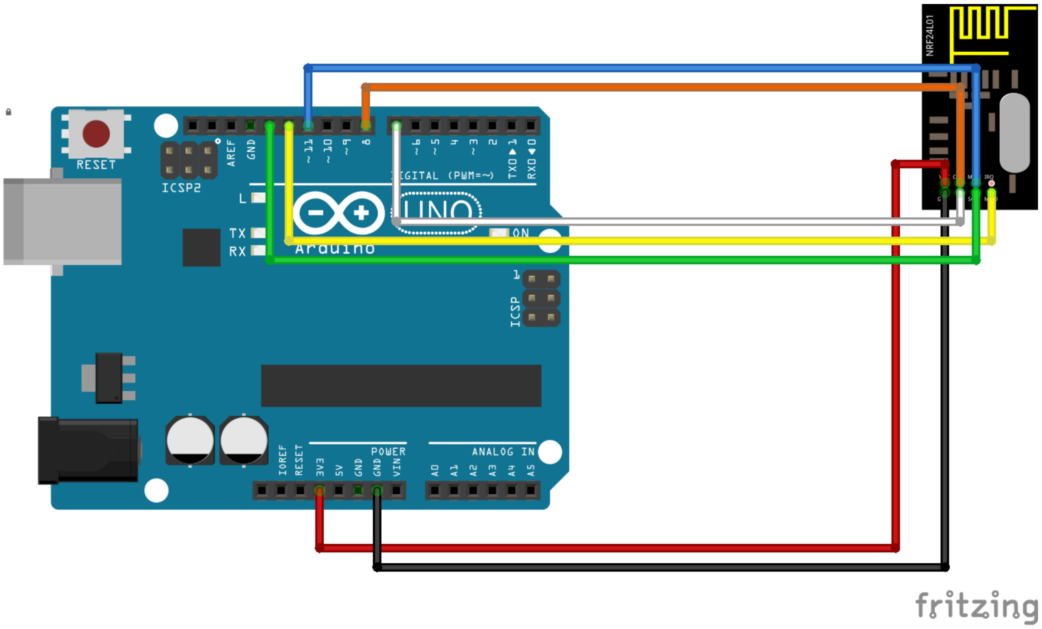
`void setCRCLength(`
`  rf24_crclength_e length)`
	Values of the enumeration are:
	• RF24_CRC_DISABLED
	• RF24_CRC_8
	• RF24_CRC_16

`rf24_crclength_e getCRCLength()`	Values of the enumeration are: • RF24_CRC_DISABLED • RF24_CRC_8 • RF24_CRC_16
`void disableCRC()`	

`void maskIRQ(`
`  bool txOk,`
`  bool txFail,`
`  bool rxReady)`	

`void printDetails()`	

Here is an example breadboard layout:



Arduino pin	nRF24 Function	Color code
Pin 13	SPI CLK	🟩
Pin 12	MISO	🟨
Pin 11	MOSI	🟦
Pin 8	CSN	🟧
Pin 7	CE	⬜
3v3	3v3	🟥
GND	GND	⬛

Here is a sample to have an Arduino read data from the device:

```
#include <SPI.h>
#include <printf.h>
#include "RF24.h"

RF24 radio(7,8);

byte addressrx[5] = {0x11, 0x22, 0x33, 0x44, 0x55};

void setup() {
   Serial.begin(115200);

   radio.begin();
   radio.setPALevel(RF24_PA_HIGH);
   radio.setChannel(0x4c);
   radio.setCRCLength(RF24_CRC_8);
   radio.setDataRate(RF24_2MBPS);
   radio.setAddressWidth(5);
   radio.setRetries(5,15);
   radio.enableDynamicPayloads();
   radio.openReadingPipe(1,addressrx);
   radio.startListening();
   printf_begin();
   radio.printDetails();
} // End of setup

void loop()
{
   if (radio.available()){
      Serial.print("Size of payload is ");
      Serial.println(radio.getDynamicPayloadSize());
      while (radio.available()) {
         char buffer[33];
         uint8_t length = radio.getDynamicPayloadSize();
         radio.read(buffer, length);
         buffer[length] = 0;
         Serial.print("Received: ");
         Serial.println(buffer);
      }
   }
} // End of Loop
```

See also:

- Github project – tmrh20.github.io
- Github project – source code
- Project Blog
- Forum thread

Integrating the nRF24 with the ESP32

Our goal will be to have the ESP32 interact with "something else" using nRF24 technology. This means that we will need **two** environments. One for our ESP32 and one for "something else". We could use two ESP32s but I tend to go for the simplest environments where possible. In addition, while we are developing and testing solutions, it is a good idea to have at least one known/reference environment. If we were working exclusively on the ESP32 and introduced an error it is likely that error would be on both environments and would either cancel each other out or result in something extremely difficult to diagnose.

As such, I recommend a simple Arduino environment with an attached nRF24. We can use a simple sketch on the Arduino that will listen for incoming network traffic and log it as it arrives. This then allows us to write the ESP32 side of the story and validate that it is working. If all is working, we will see ESP32 transmitted data arrive on the Arduino.

Here is an example ESP32 application.

```
#include <esp_log.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <RF24.h>

#include "sdkconfig.h"

static char tag[] = "test_nrf24";

extern "C" {
   void task_test_nrf24(void *ignore);
}

void task_test_nrf24(void *ignore) {
   RF24 rf24(25,15);
   rf24.begin();
   rf24.setPALevel(RF24_PA_HIGH);
   rf24.setChannel(0x4c);
   rf24.setCRCLength(RF24_CRC_8);
   rf24.setDataRate(RF24_2MBPS);
   rf24.setAddressWidth(5);
   rf24.setRetries(5,15);
   rf24.enableDynamicPayloads();

   rf24.printDetails();

   uint8_t txAddress[5] = {0x11, 0x22, 0x33, 0x44, 0x55};
   rf24.openWritingPipe(txAddress);
   rf24.write("123", 3);
   ESP_LOGD(tag, "Transmitted data");

   ESP_LOGD(tag, "test done");
   vTaskDelete(NULL);
}
```

and here is a corresponding example Arduino sketch:

```
#include <SPI.h>
#include <printf.h>
#include "RF24.h"

RF24 radio(7,8);

byte addressrx[5] = {0x11, 0x22, 0x33, 0x44, 0x55};

void setup() {
   Serial.begin(115200);

   radio.begin();
   radio.setPALevel(RF24_PA_HIGH);
   radio.setChannel(0x4c);
   radio.setCRCLength(RF24_CRC_8);
   radio.setDataRate(RF24_2MBPS);
   radio.setAddressWidth(5);
   radio.setRetries(5,15);
   radio.enableDynamicPayloads();
   radio.openReadingPipe(1,addressrx);
   radio.startListening(); // Become a receiver.
   printf_begin();
   radio.printDetails();
} // End of setup

void loop()
{
   if (radio.available()){
      Serial.print("Size of payload is ");
      Serial.println(radio.getDynamicPayloadSize());
      while (radio.available()) {
         char buffer[33];
         uint8_t length = radio.getDynamicPayloadSize();
         radio.read(buffer, length);
         buffer[length] = 0;
         Serial.print("Received: ");
         Serial.println(buffer);
      }
   }
} // End of Loop
```

When we run these, a message is sent from the ESP32 to the Arduino.

Now let us look at how to build an environment for running nRF24 on the ESP32. First, please realize that the library selected is based on C++. As such, you should be comfortable working in both C and C++ environments. In addition, since the nRF24 library from GitHub doesn't yet understand the ESP32 environment, we will have to augment the functions currently present with ESP32 logic. Rather than re-invent the wheel, we use classes from the example C++ ESP32 classes found here:

https://github.com/nkolban/esp32-snippets

Lets begin.

- Create a new ESP-IDF template application.
- Create a components directory.
- In the components directory, symbolic link to the cpp_utils
- In the components directory, download the nRf24 library

```
$ git clone https://github.com/nRF24/RF24.git
```

- In the RF24 directory, create a file called component.mk that includes:

```
COMPONENT_ADD_INCLUDEDIRS=.
COMPONENT_SRCDIRS=. utility/ESP_IDF
```

- Create the file called components/RF24/utility/includes.h which contains:

```
#ifndef __RF24_INCLUDES_H__
#define __RF24_INCLUDES_H__

#define RF24_ESP_IDF
#include "ESP_IDF/RF24_arch_config.h"
#include "ESP_IDF/RF24_ESP_IDF.h"
#include "ESP_IDF/NRF24_spi.h"
#endif
```

- Copy in the esp32-snippets/hardware/utility/ESP_IDF directory

And that's it.

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. At the time of writing, the currently available release of Eclipse is called "Neon".

Let us now look at how we can setup an Eclipse environment for an `ESP-IDF` template application.

1. Start with ESP-IDF installed and the environment variable `IDF_PATH` defined.

2. Make a directory called workspace and change into it.

```
$ mkdir workspace
$ cd workspace
```

3. Clone the template app into the workspace folder

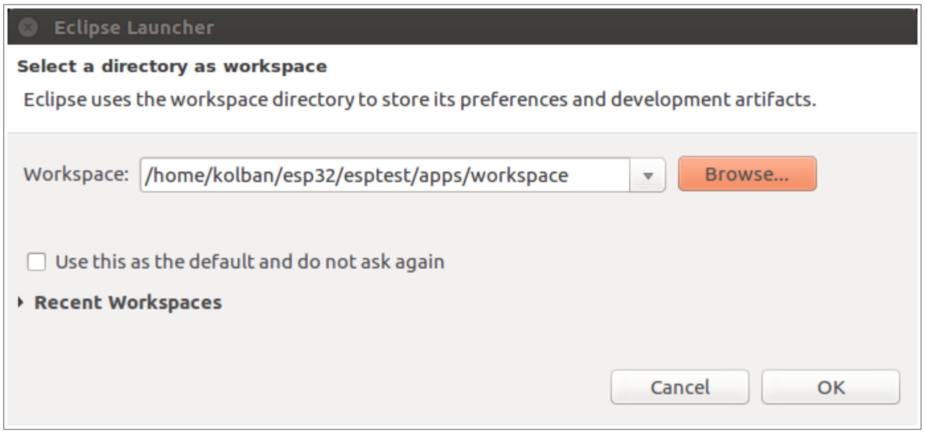
```
git clone https://github.com/espressif/esp-idf-template.git myapp
```

4. Run the `make menuconfig` command to create the `sdkconfig` file.

```
$ cd myapp
$ make menuconfig
$ cd ..
```

5. Launch Eclipse.

6. When prompted for the workspace, provide the directory you just created



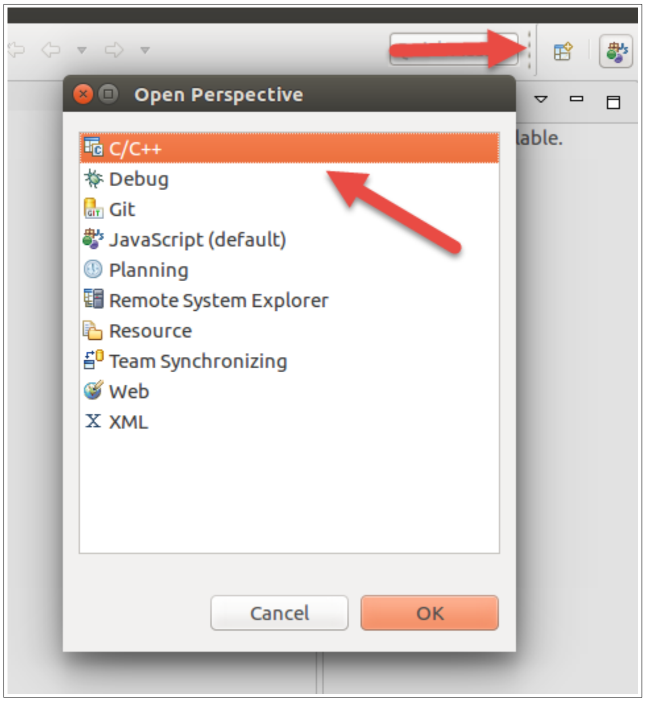
Eclipse Launcher

Select a directory as workspace

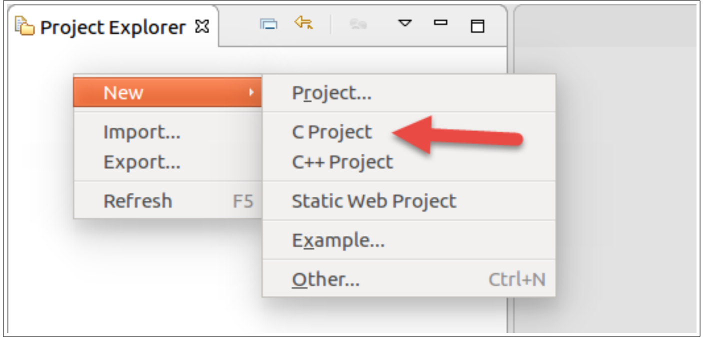
Eclipse uses the workspace directory to store its preferences and development artifacts.

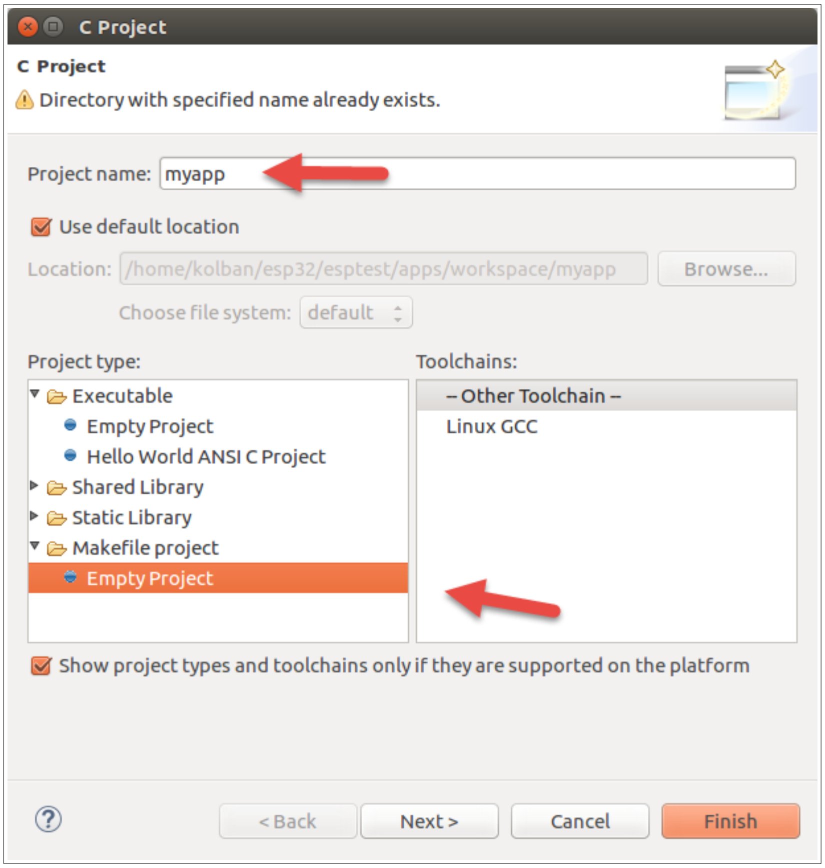
Workspace: /home/kolban/esp32/esptest/apps/workspace

7. Switch to the C/C++ Perspective



8. Create a new empty C project with the same name as your project (eg. `myapp`).





C Project

⚠ Directory with specified name already exists.

Project name: myapp

☑ Use default location

Location: /home/kolban/esp32/esptest/apps/workspace/myapp Browse...

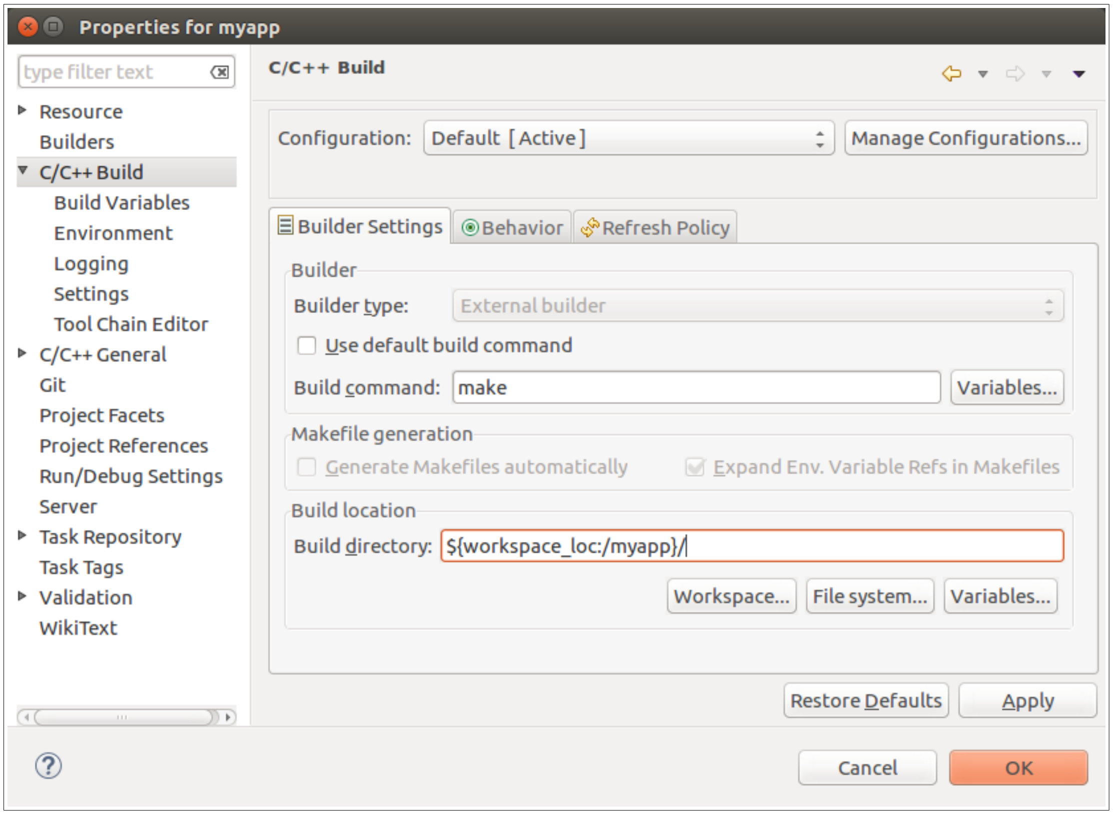
Choose file system: default

Project type: Makefile project / Empty Project

Toolchains: – Other Toolchain –

☑ Show project types and toolchains only if they are supported on the platform

9. Open up the project properties and visit C/C++ Build. Change the settings to look as follows:



Configuration: Default [Active]

Builder type: External builder

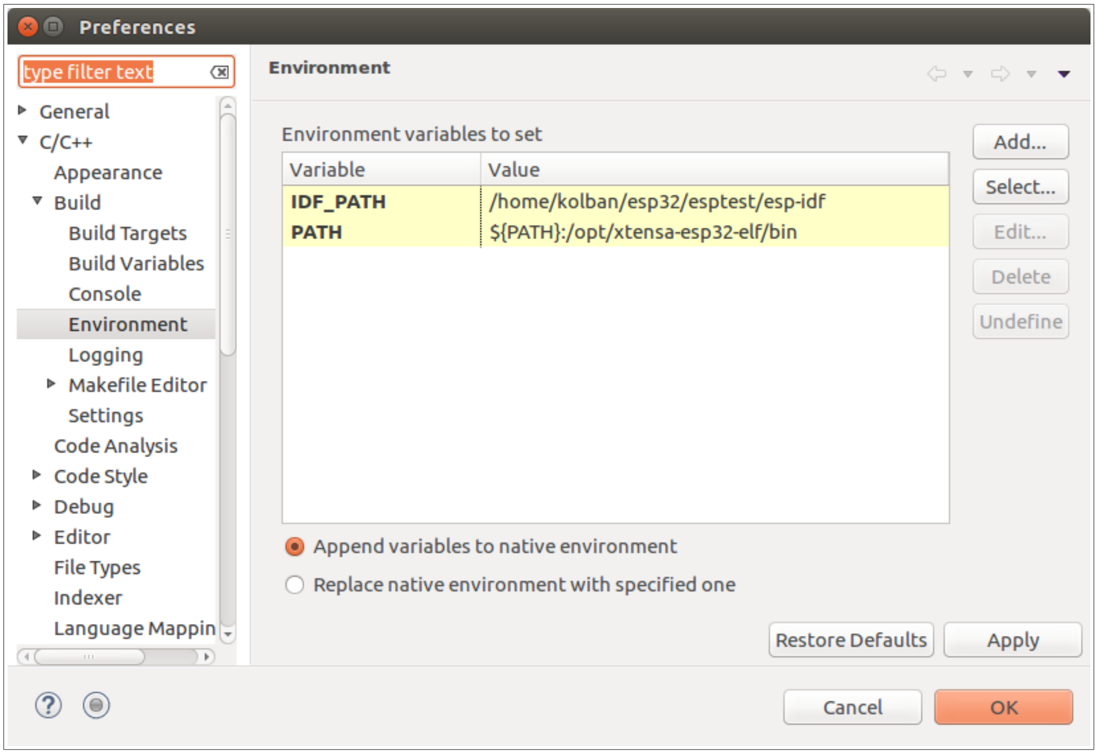
□ Use default build command

Build command: make

□ Generate Makefiles automatically ☑ Expand Env. Variable Refs in Makefiles

Build directory: ${workspace_loc:/myapp}/

10. In the **global** properties set `C/C++ > Build > Environment` add two variables called `IDF_PATH` and `PATH` as shown:



Environment

Environment variables to set

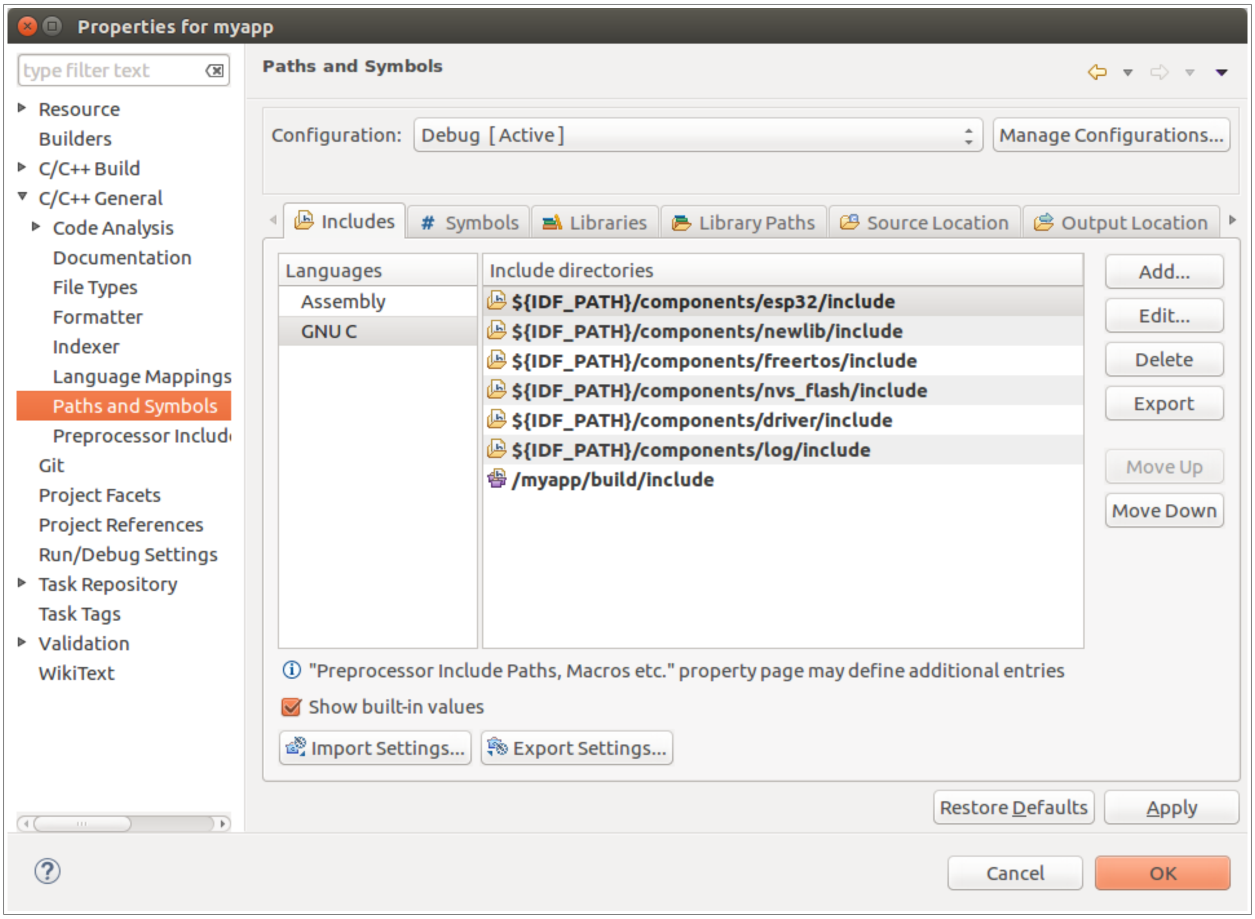
Variable	Value
IDF_PATH	/home/kolban/esp32/esptest/esp-idf
PATH	${PATH}:/opt/xtensa-esp32-elf/bin

● Append variables to native environment

○ Replace native environment with specified one

The `IDF_PATH` variable points to the root of your ESP-IDF installation and the path `PATH` is augmented to point to the Xtensa ESP32 tool chain.

11. Open the properties dialog called `C/C++ General > Paths and Symbols` and add the following:



Configuration: Debug [Active]

Languages	Include directories
Assembly	${IDF_PATH}/components/esp32/include
GNU C	${IDF_PATH}/components/newlib/include
	${IDF_PATH}/components/freertos/include
	${IDF_PATH}/components/nvs_flash/include
	${IDF_PATH}/components/driver/include
	${IDF_PATH}/components/log/include
	/myapp/build/include

ⓘ "Preprocessor Include Paths, Macros etc." property page may define additional entries

☑ Show built-in values

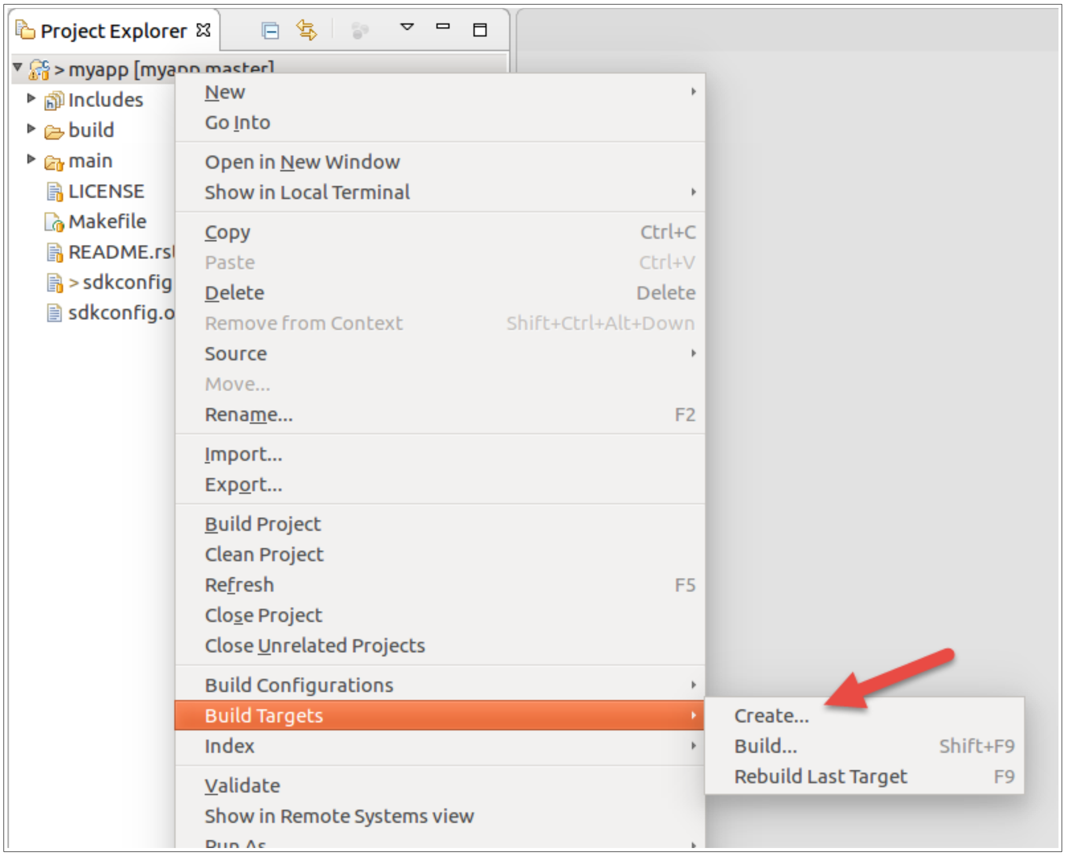
Also … components/tcpip_adapter/include

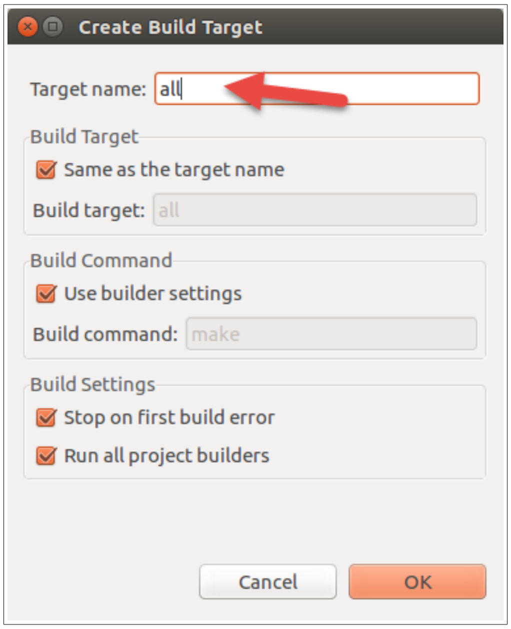
Also … components/lwip/include/lwip

Also … components/spi_flash/include

What this does is tell the Eclipse tooling where to go look for include directories to satisfy the inclusions from the C source files.

12. Add a build target for the Makefile called `all`:





Create Build Target

Target name: all

Build Target

☑ Same as the target name

Build target: all

Build Command

☑ Use builder settings

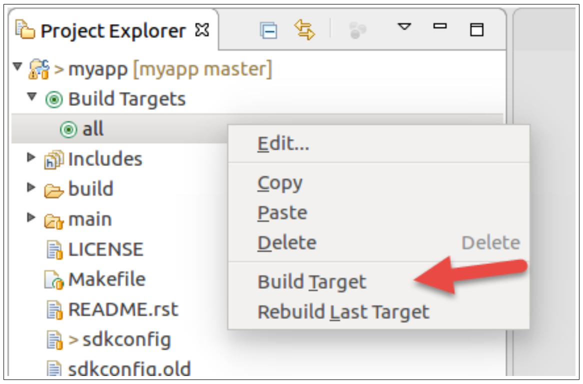
Build command: make

Build Settings

☑ Stop on first build error

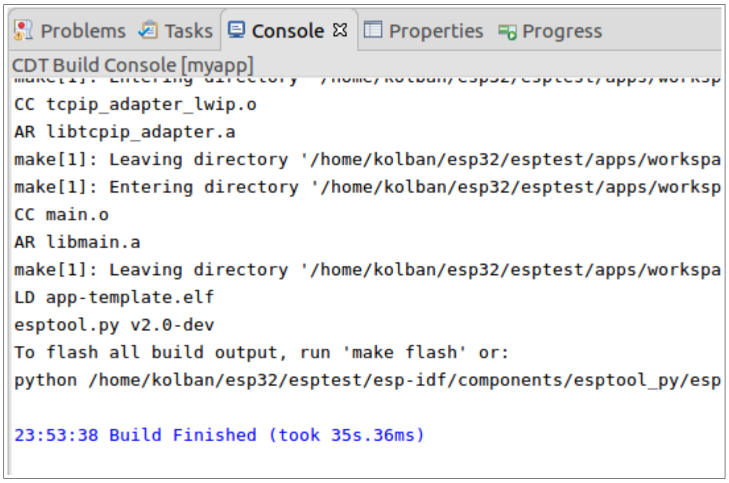
☑ Run all project builders

13. Run `Build Targets > all`:



This causes the Makefile for the project to be run and the compilation proceeds.

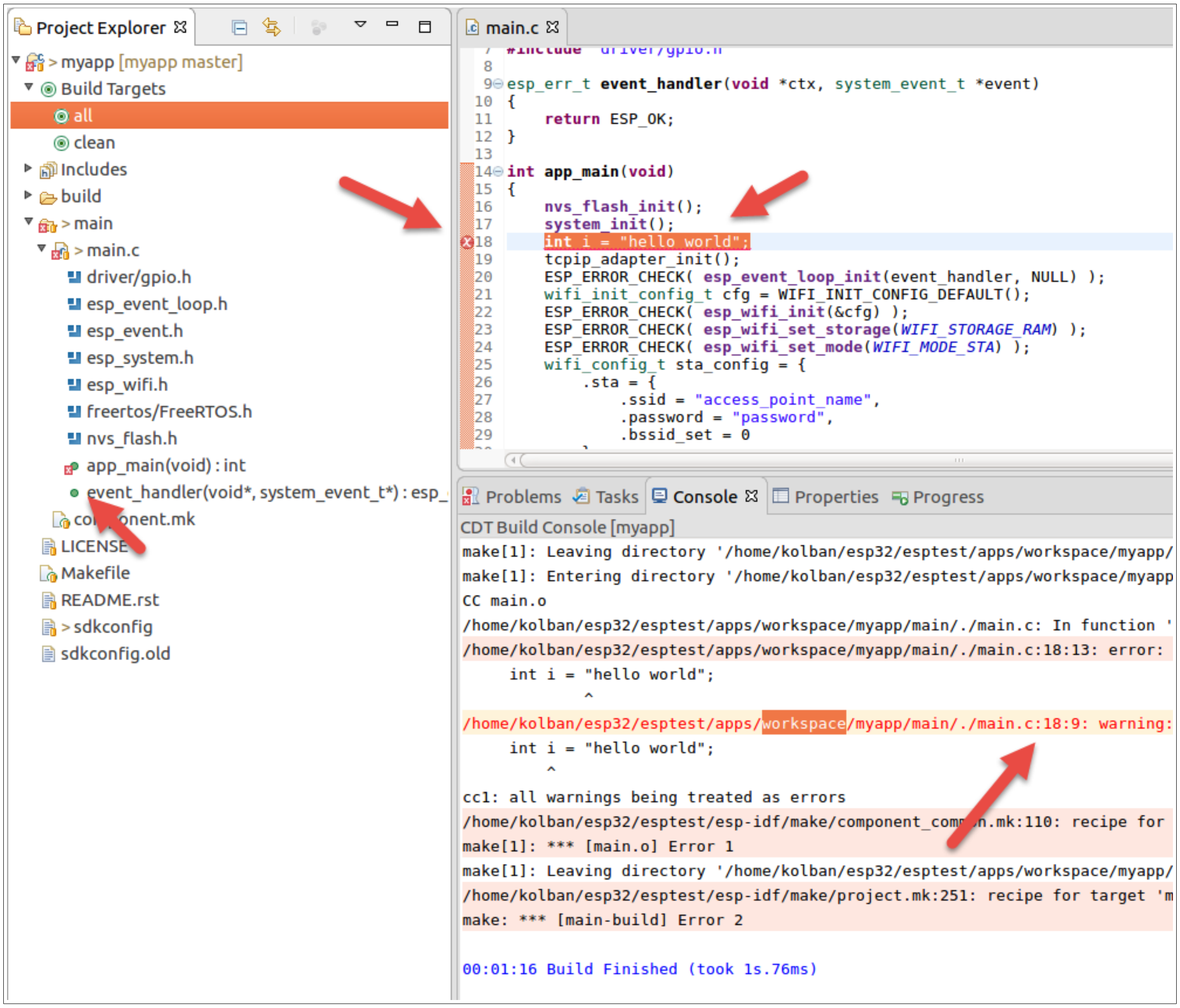
14. Examine the console.



```
make[1]: Entering directory '/home/kolban/esp32/esptest/apps/worksp
CC tcpip_adapter_lwip.o
AR libtcpip_adapter.a
make[1]: Leaving directory '/home/kolban/esp32/esptest/apps/workspa
make[1]: Entering directory '/home/kolban/esp32/esptest/apps/worksp
CC main.o
AR libmain.a
make[1]: Leaving directory '/home/kolban/esp32/esptest/apps/workspa
LD app-template.elf
esptool.py v2.0-dev
To flash all build output, run 'make flash' or:
python /home/kolban/esp32/esptest/esp-idf/components/esptool_py/esp

23:53:38 Build Finished (took 35s.36ms)
```

We should see a clean build.

And with that you have build a binary. Now if you edit the source files and re-build the Makefile target called "`all`", only the changed files will be shown. Should there be a compilation error, we will see them all highlighted in the console, in the code and in the project explorer. Double clicking in the console will take us to the associated source file and line within that source file.



```
#include "driver/gpio.h"

esp_err_t event_handler(void *ctx, system_event_t *event)
{
    return ESP_OK;
}

int app_main(void)
{
    nvs_flash_init();
    system_init();
    int i = "hello world";
    tcpip_adapter_init();
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) );
    ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) );
    wifi_config_t sta_config = {
        .sta = {
            .ssid = "access_point_name",
            .password = "password",
            .bssid_set = 0
```

```
make[1]: Leaving directory '/home/kolban/esp32/esptest/apps/workspace/myapp/
make[1]: Entering directory '/home/kolban/esp32/esptest/apps/workspace/myapp
CC main.o
/home/kolban/esp32/esptest/apps/workspace/myapp/main/./main.c: In function '
/home/kolban/esp32/esptest/apps/workspace/myapp/main/./main.c:18:13: error:
     int i = "hello world";
             ^
/home/kolban/esp32/esptest/apps/workspace/myapp/main/./main.c:18:9: warning:
     int i = "hello world";
         ^
cc1: all warnings being treated as errors
/home/kolban/esp32/esptest/esp-idf/make/component_common.mk:110: recipe for
make[1]: *** [main.o] Error 1
make[1]: Leaving directory '/home/kolban/esp32/esptest/apps/workspace/myapp/
/home/kolban/esp32/esptest/esp-idf/make/project.mk:251: recipe for target 'm
make: *** [main-build] Error 2

00:01:16 Build Finished (took 1s.76ms)
```

You can also add other build targets such as:

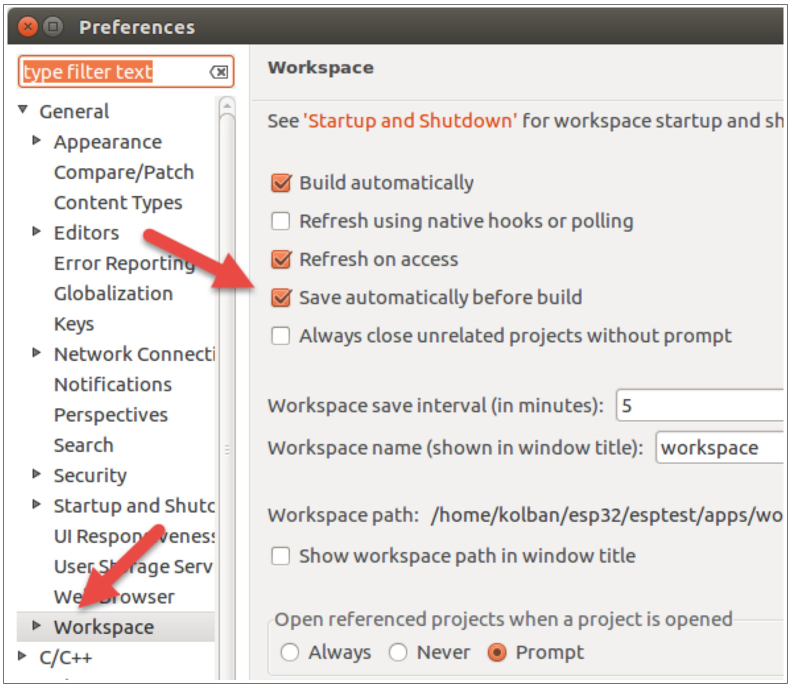
- `flash` – To perform a flash of your application against your ESP32.
- `clean` – To clean the current build.

Short cuts are also available for some of the more common tasks. For example:

- `F9` – Will re-execute the last build you performed.
- The menu bar entry 010 will perform a "`make all`".

Here are some additional Eclipse tips I recommend:

- By default, any changed files are **not** saved before building. I often find I make a change and rebuild and test only to find no difference in execution only to realize that what I built did not include my changes because I hadn't saved the source file. We can automate the saving of source files before a build by setting the preferences entry in `General > Workspace` and checking "`Save automatically before build`":



Workspace

See 'Startup and Shutdown' for workspace startup and sh

☑ Build automatically

□ Refresh using native hooks or polling

☑ Refresh on access

☑ Save automatically before build

□ Always close unrelated projects without prompt

Workspace save interval (in minutes): 5

Workspace name (shown in window title): workspace

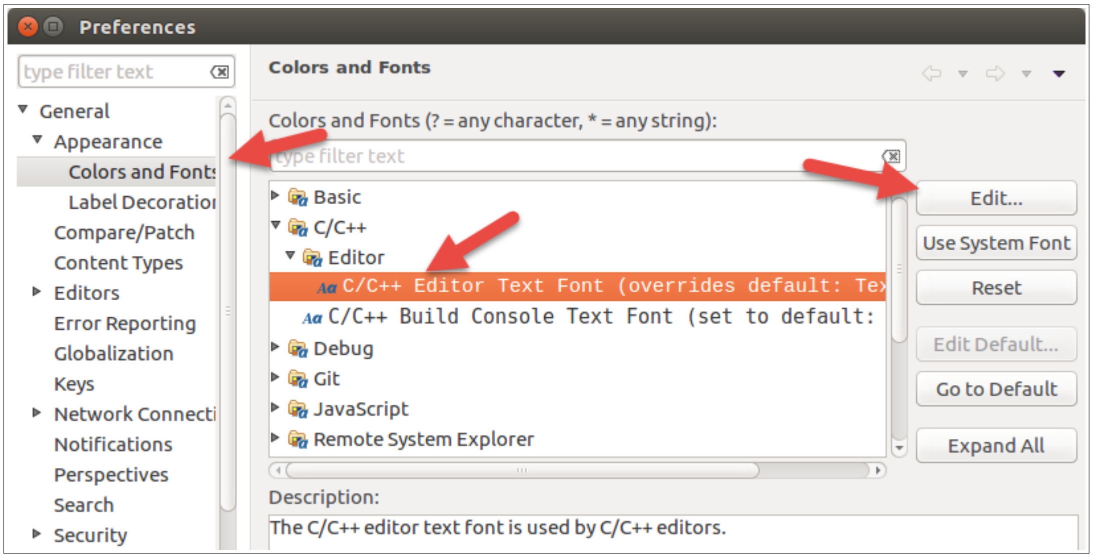
Workspace path: /home/kolban/esp32/esptest/apps/wo

□ Show workspace path in window title

Open referenced projects when a project is opened

○ Always ○ Never ● Prompt

- Install the Eclipse Path Tools package.
- The primary editor font and font size can be changed at the preferences setting located at `General > Appearance > Colors and Fonts | C/C++ > Editor > C/C++ Editor Text Font` and then clicking "`Edit…`":



The C/C++ editor text font is used by C/C++ editors.

My personal preference is "Liberation Mono" at 11pt.

Installing the Eclipse Serial terminal

Note: My retests of this crash Eclipse. I have since switched to opening a shell within Eclipse and running minicom within the shell.

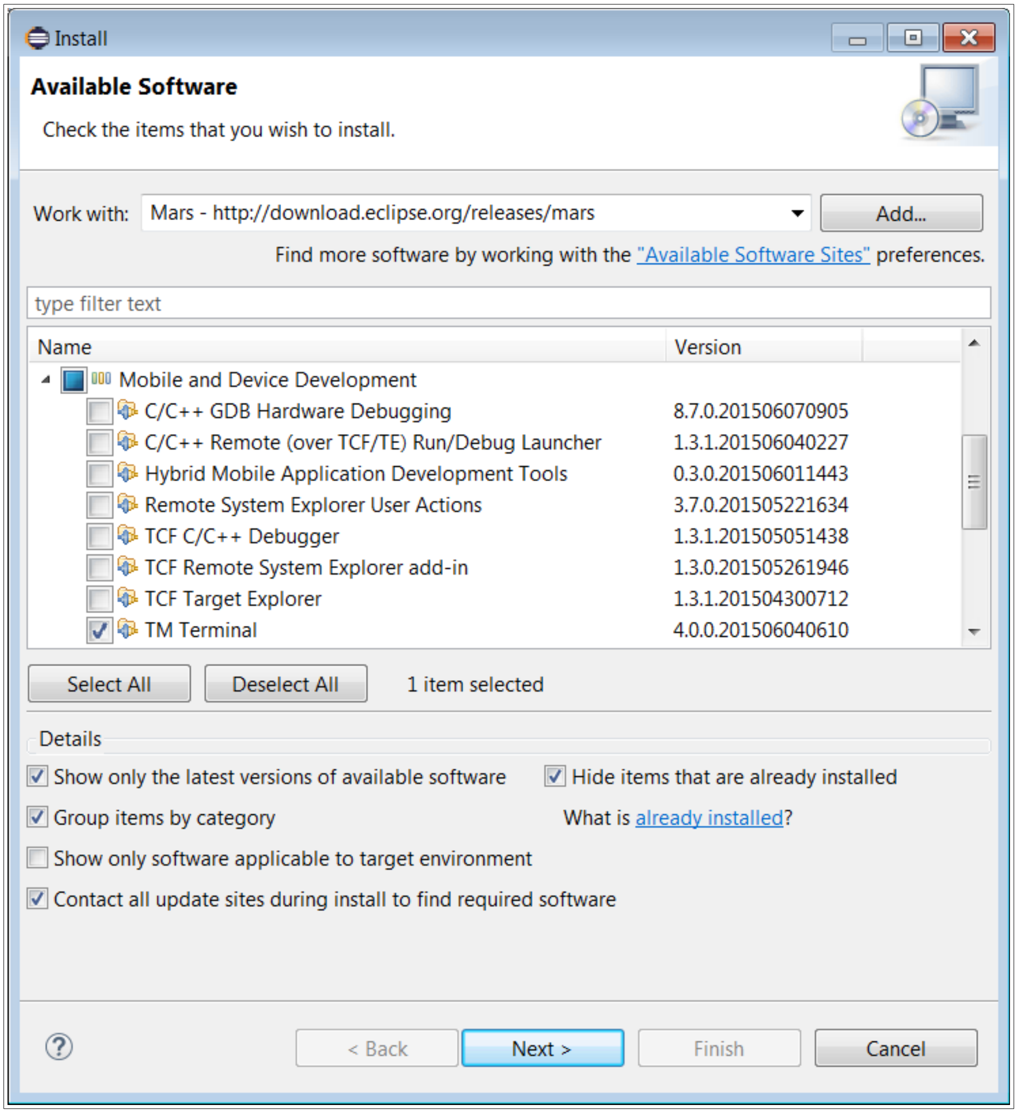
Although there are many excellent serial terminals available as stand-alone Windows applications, an alternative is the Eclipse Terminal which also has serial support. This allows a serial terminal to appear as a view within the Eclipse IDE. It does not come installed by default but the steps to add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

First start Eclipse (I use the Luna release).

Go to `Help > Install new software...`

Select the eclipse download repository.

Select `General Purpose Tools > TM Terminal via Remote API Connector Extensions`



Available Software

Check the items that you wish to install.

Work with: Mars - http://download.eclipse.org/releases/mars

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
■ Mobile and Device Development	
□ C/C++ GDB Hardware Debugging	8.7.0.201506070905
□ C/C++ Remote (over TCF/TE) Run/Debug Launcher	1.3.1.201506040227
□ Hybrid Mobile Application Development Tools	0.3.0.201506011443
□ Remote System Explorer User Actions	3.7.0.201505221634
□ TCF C/C++ Debugger	1.3.1.201505051438
□ TCF Remote System Explorer add-in	1.3.0.201505261946
□ TCF Target Explorer	1.3.1.201504300712
☑ TM Terminal	4.0.0.201506040610

1 item selected

Details

☑ Show only the latest versions of available software ☑ Hide items that are already installed

☑ Group items by category What is already installed?

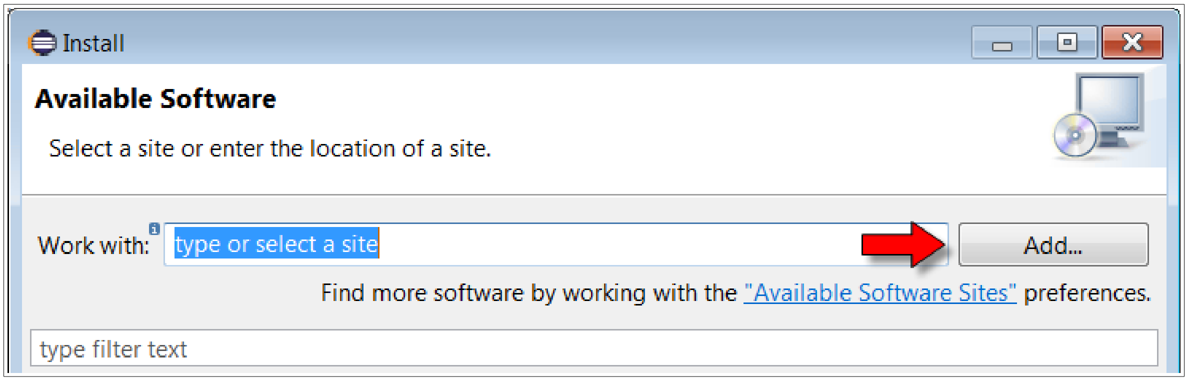
□ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

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

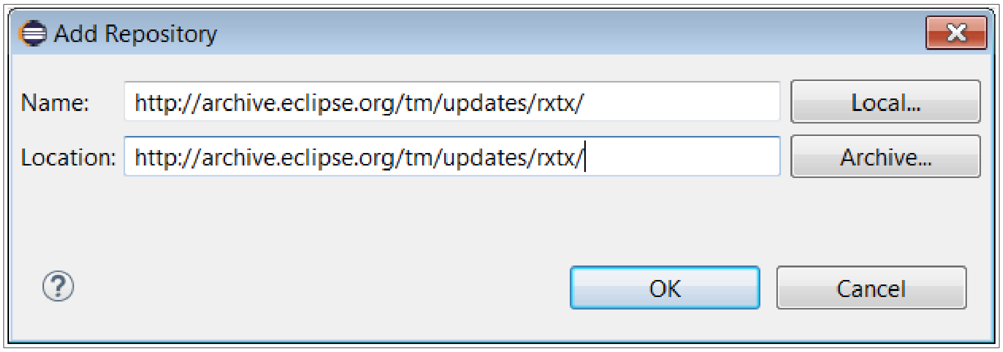
We are not ready to use it yet, we must add serial port support into Eclipse.

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


Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text

The repository URL is:

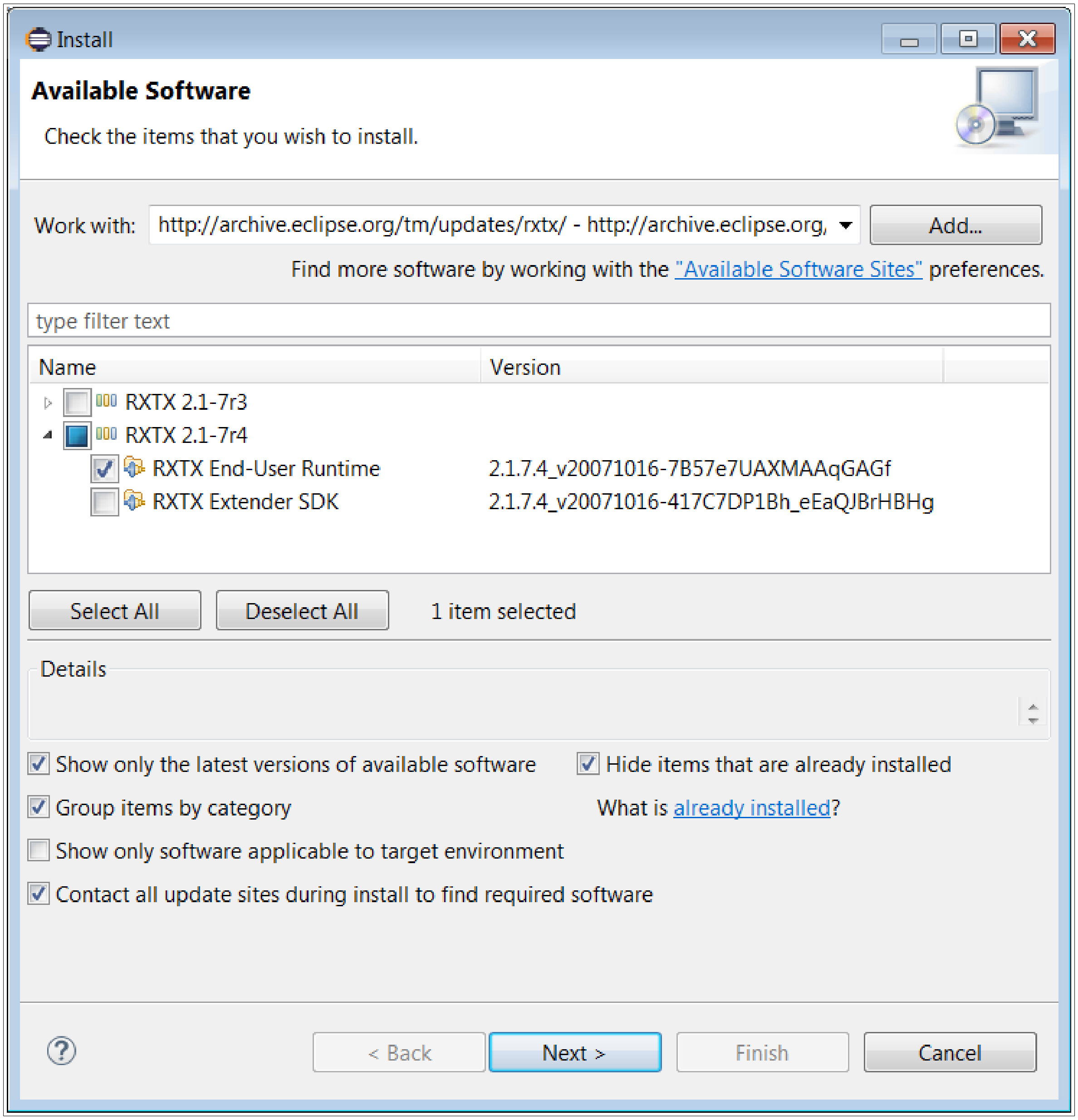
- http://archive.eclipse.org/tm/updates/rxtx/



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

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

Now we can select the Serial port run-time support library:



Available Software

Check the items that you wish to install.

Work with: http://archive.eclipse.org/tm/updates/rxtx/ - http://archive.eclipse.org,

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
□ RXTX 2.1-7r3	
■ RXTX 2.1-7r4	
☑ RXTX End-User Runtime	2.1.7.4_v20071016-7B57e7UAXMAAqGAGf
□ RXTX Extender SDK	2.1.7.4_v20071016-417C7DP1Bh_eEaQJBrHBHg

1 item selected

Details

☑ Show only the latest versions of available software

☑ Hide items that are already installed

☑ Group items by category

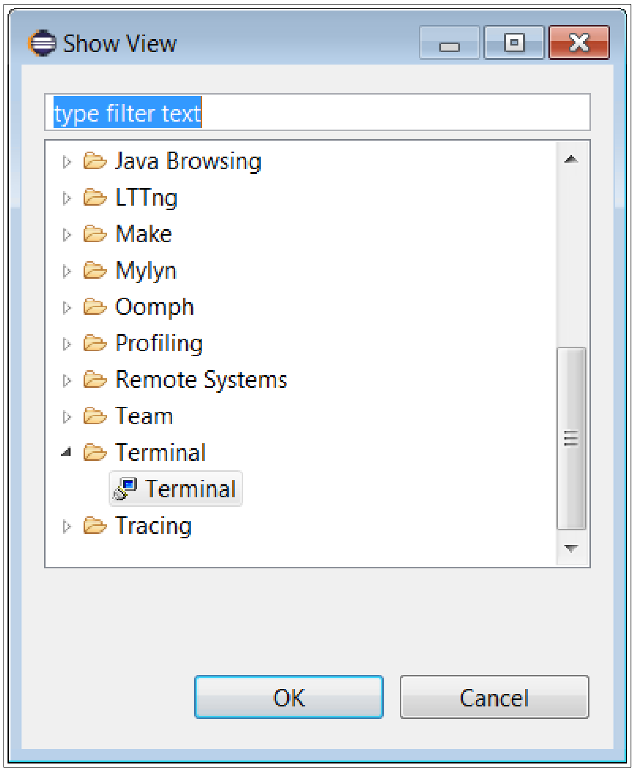
What is already installed?

□ Show only software applicable to target environment

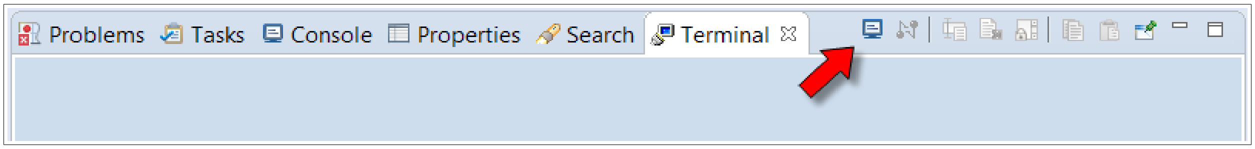
☑ Contact all update sites during install to find required software

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

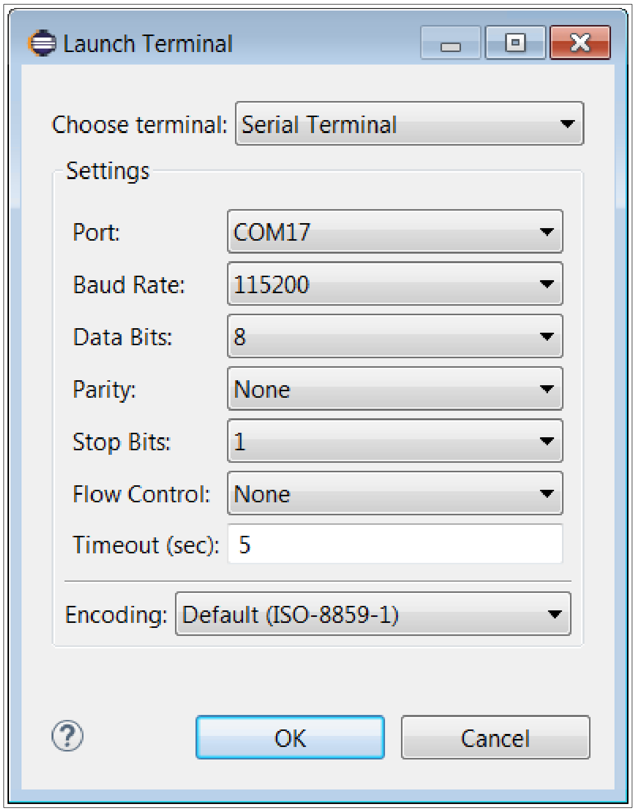
We now have terminal support installed and are ready to use it. From `Windows > Show View > Other` we will find a new category called "`Terminal`".



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.



Choose terminal: Serial Terminal

Settings

Port: COM17

Baud Rate: 115200

Data Bits: 8

Parity: None

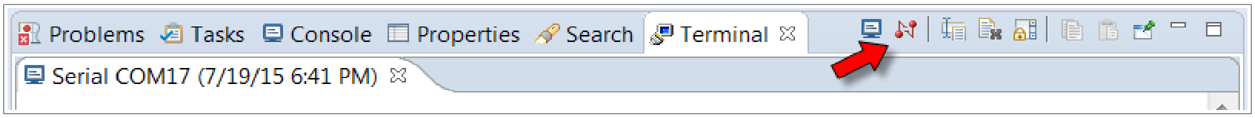
Stop Bits: 1

Flow Control: None

Timeout (sec): 5

Encoding: Default (ISO-8859-1)

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.

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

Web development using Eclipse

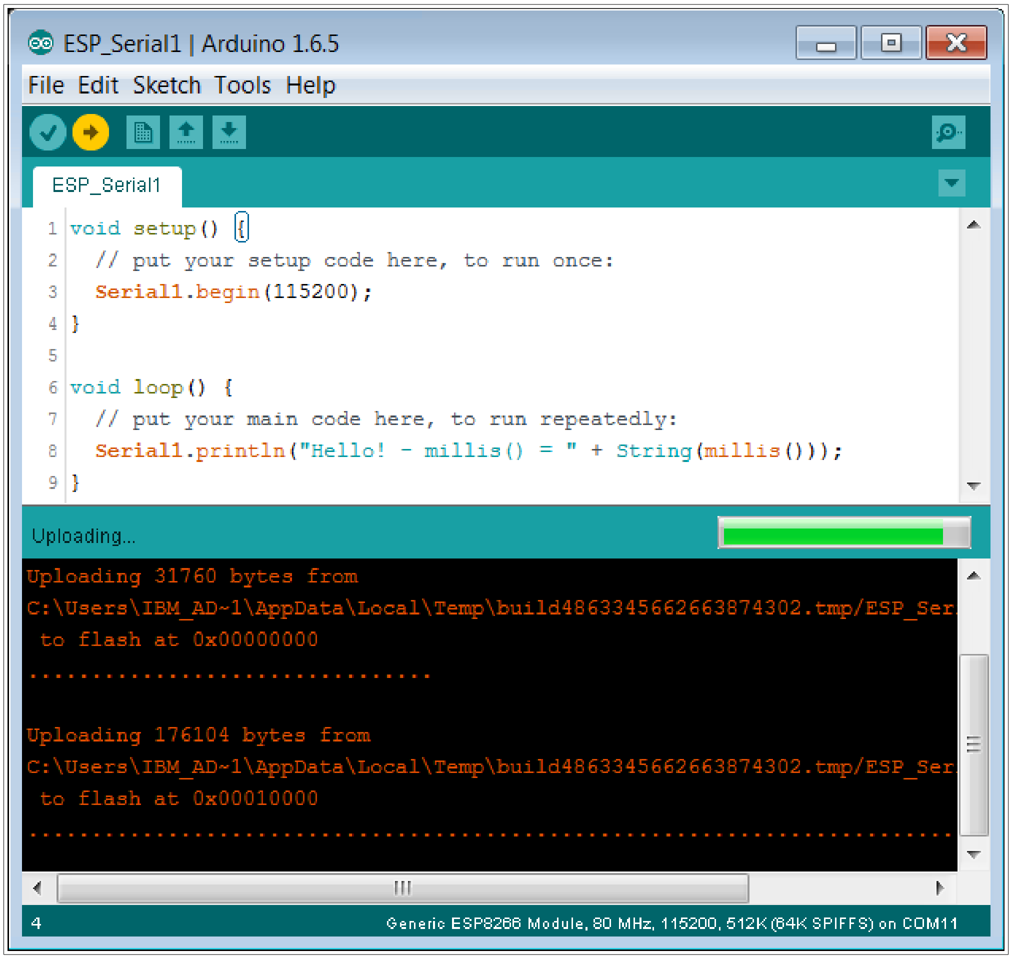
Eclipse also provides a first class web development environment for writing and testing web apps including HTML pages. It is suggested that the Eclipse Web Developer Tools be installed.

Programming using the Arduino IDE

Long before there was an ESP8266 or ESP32, 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 everyone the ability to build something quickly and easily. The Integrated Development Environment (IDE) for the Arduino has always been free of charge for download from the Internet. 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:



```
void setup() {
  // put your setup code here, to run once:
  Serial1.begin(115200);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial1.println("Hello! - millis() = " + String(millis()));
}
```

Uploading...

```
Uploading 31760 bytes from
C:\Users\IBM_AD~1\AppData\Local\Temp\build4863345662663874302.tmp/ESP_Ser
 to flash at 0x00000000
................................

Uploading 176104 bytes from
C:\Users\IBM_AD~1\AppData\Local\Temp\build4863345662663874302.tmp/ESP_Ser
 to flash at 0x00010000
........................................................................
```

4 Generic ESP8266 Module, 80 MHz, 115200, 512K (64K SPIFFS) on COM11

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 what would otherwise be called a C language program and that would intimidate 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`" that, when clicked, what it does is deploy the application to the Arduino.

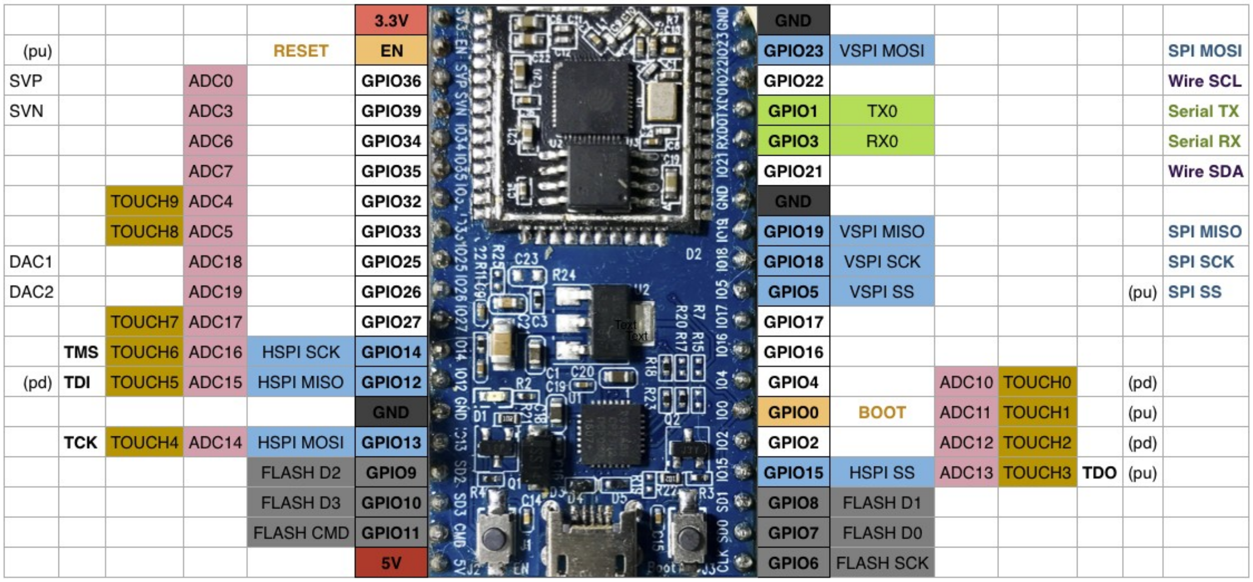
In addition to providing a C language editor plus tools to compile and deploy, the Arduino IDE provides pre-supplied libraries of C routines that "hide" complex implementation details that might otherwise be needed when programming to the Arduino boards. For example, UART programming would undoubtedly 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 ESP32 story? Well, a bunch of talented individuals have built out an Open Source project on Github that provides a "plug-in" 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 ESP32. What this effectively means is that we can use the Arduino IDE and build ESP32 applications with the minimum of fuss.

Mapping from the Arduino to the ESP32

Obviously, an Arduino is not the same thing as an ESP32 and as such, assumptions about the Arduino hardware don't map to the same things on the ESP32.

ESP32 Dev Board PINMAP



					3.3V	GND						
(pu)				RESET	EN	GPIO23	VSPI MOSI					SPI MOSI
SVP			ADC0		GPIO36	GPIO22						Wire SCL
SVN			ADC3		GPIO39	GPIO1	TX0					Serial TX
			ADC6		GPIO34	GPIO3	RX0					Serial RX
			ADC7		GPIO35	GPIO21						Wire SDA
		TOUCH9	ADC4		GPIO32	GND						
		TOUCH8	ADC5		GPIO33	GPIO19	VSPI MISO					SPI MISO
DAC1			ADC18		GPIO25	GPIO18	VSPI SCK					SPI SCK
DAC2			ADC19		GPIO26	GPIO5	VSPI SS				(pu)	SPI SS
		TOUCH7	ADC17		GPIO27	GPIO17						
	TMS	TOUCH6	ADC16	HSPI SCK	GPIO14	GPIO16						
(pd)	TDI	TOUCH5	ADC15	HSPI MISO	GPIO12	GPIO4		ADC10	TOUCH0		(pd)	
					GND	GPIO0	BOOT	ADC11	TOUCH1		(pu)	
	TCK	TOUCH4	ADC14	HSPI MOSI	GPIO13	GPIO2		ADC12	TOUCH2		(pd)	
				FLASH D2	GPIO9	GPIO15	HSPI SS	ADC13	TOUCH3	TDO	(pu)	
				FLASH D3	GPIO10	GPIO8	FLASH D1					
				FLASH CMD	GPIO11	GPIO7	FLASH D0					
					5V	GPIO6	FLASH SCK					

Here we discuss some of the distinctions:

- I2C and the Wire library – The default pins on the ESP32 for the I2C and Wire library are GPIO21 for SDA and GPIO22 for SCL.

Implications of Arduino IDE support

The ability to treat an ESP32 as though it were "like" an Arduino is a notion that I haven't been able to fully absorb yet. ESP32 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 ESP32 APIs. The Arduino libraries for ESP32 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 ESP32 native functions, we are masking access to lower levels of knowledge and function. Further, thinking of the ESP32 as though it were an Arduino can lead to design problems. For example, the E 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/ESP32 be penalized for having to install and learn more complex tool chains and syntax to achieve the same result with more ESP8266/ESP32 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 ESP32 support

The Arduino IDE support for ESP32 can be found on Github here.

https://github.com/espressif/arduino-esp32

As of October 2016, this is an extremely new component and is still in the process of baking so be gentle with it.

First we download and install an Arduino IDE at 1.6.9 or later. We then follow the installation/configuration instructions which are:

1. Enter the Arduino installation directory
2. Change into the "`hardware`" directory:

```
$ cd hardware
```

3. Make a directory called "`espressif`":

```
$ mkdir espressif
```

4. Change into the "`espressif`" directory:

```
$ cd espressif
```

5. Download the ESP32 package for Espruino:

```
$ git clone https://github.com/espressif/arduino-esp32.git esp32
```

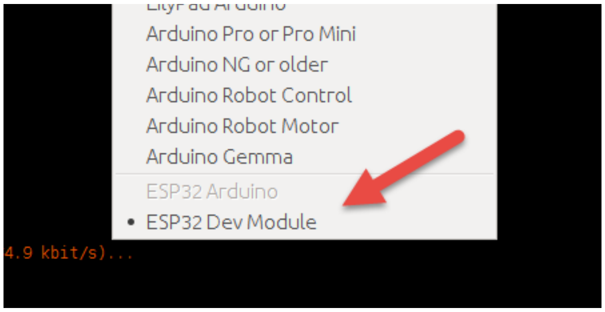
6. Change into the "`esp32/tools`" directory:

```
$ cd esp32/tools
```

7. Run the supplied "get.py" script

```
$ python get.py
```

8. Start Arduino
9. Pick "`ESP32 Dev Module`" from the Boards:



Using the Arduino libraries as an ESP-IDF component

Now comes the capability that excites me the most. By now we should be fully appreciative of the capabilities provided by the ESP-IDF and realize that it is component driven. This means that as we need functions, we can "drop them in" as components as long as they conform to the governance rules of the ESP-IDF.

The Arduino libraries can be used as an ESP-IDF component.

Let that sink in for a few moments. What this effectively means is that all the functions provided by the Arduino ESP32 libraries are available to us in our "normal" ESP-IDF applications. This is huge!! It opens up a world of possibilities for us.

Now let us look at how we go about using the Arduino libraries in our project. Let us imagine that we have created a new ESP-IDF template application by using:

```
$  git clone https://github.com/espressif/esp-idf-template.git myapp
```

Now we want to perform this recipe:

1. Create a new directory called "`components`" in the app directory.

```
$ mkdir components
```

2. Change into the components directory.

```
$ cd components
```

3. Install the Arduino-ESP32 library relative to here by performing a Github Clone:

```
$ git clone https://github.com/espressif/arduino-esp32.git
```

And that's it!! We can now build our sample application as normal.

Tips for working in the Arduino environment

Remember that the Arduino environment is two things. First, an actual application that you install on your machine providing the Arduino IDE. Second, a set of libraries that model those available to an actual Arduino device which are mapped to ESP* capabilities. With that in mind, here are some hints and tips that I find useful when writing Arduino sketches for an ESP32 environment.

Initialize global classes in setup()

Within an Arduino sketch, we have a pre-supplied function called `setup()` that is called only once during ESP32 boot-up. Within this function, you perform one time initialization functions. In C++, we have the ability to create class instances globally. For example:

```
MyClass myClass(123);

void setup() {
       // Some code here
}
```

instead of this, use the following:

```
MyClass *myClass;

void setup() {
       myClass = new MyClass(123);
       // Some code here
}
```

This of course changes your variable's data type. It went from being an instance of `MyClass` to being a pointer to an instance of `MyClass` which means that you might have to change other aspects of your program … but the reason for this is that in the first case, the constructor for your `MyClass` instance ran outside of the `setup()` and we can't say what state the environment might have been in at that point. Within the `setup()` code, we have a reasonable expectation of the environment context.

Invoking Espressif SDK API from a sketch

- There is nothing to prevent you from invoking Espressif SDK API from within your sketch. You must include any include files that are necessary. Here is an example of including "`XXXX.h`".
-
  ```
  extern "C" {
     #include "XXXX.h"
  }
  ```

Notice the bracketing with the C++ construct that causes the content to appear as though it were being defined in a C program.

Reasons to consider using Eclipse over Arduino IDE

As previously mentioned, there is no question that the Arduino IDE is much more friendly and consumable that the professional Eclipse environment for folks new to the area. There doesn't appear to be anything that one can't build using the Arduino IDE that would mean one would have to switch to Eclipse. So why then would one ever consider using Eclipse?

There is a trade-off between ease of use and richness of function. For example, Eclipse has built in syntax assistance, error checking, code cross references, refactoring and much more. None of these things are "essential" but any one of them can be considered to make a programming job easier if and when needed. If I need to rename a variable, in Arduino IDE I have to manually find and replace each occurrence. In Eclipse, I can re-factor the variable using a built-in wizard and the IDE does the work for me. As another example, if I can't remember the syntax for a method, in Arduino IDE I would go to the web and look it up while in Eclipse I could type the name of the method and hover my mouse over it and the tooling will show me the possible options for the parameters.

Programming with JavaScript

JavaScript is a high level interpreted programming language. Some of its core constructs are loose typing, object oriented, support of lambda functions, support of closures and, most importantly, has become **the** language of the web. If one is writing a browser hosted application, then it is a certainty that it will be written in JavaScript. But what of non-browser environments? For a while now JavaScript has been moving into server side code through projects such as Node.js. As a language for running server code, it has a significant set of features to realize this capability. Specifically, it supports an event driven architecture paradigm. In JavaScript, we can register functions to be called back upon events being detected. These callbacks can be defined as simple in-line functions on what to do. In these made up examples, we illustrate this:

```
httpServer.on("/path1", function() {
   // Do something for /path1
   httpServer.send(response);
});
```

or

```
socket.accept(port, function(newSocket) {
   newSocket.on("receive", function(data) {
      print("We received new data: " + data);
      newSocket.send("We got the data", function() {
         newSocket.close();
      });
   });
});
```

And if we can implement a good JavaScript model, it maps excellently to the ESP32 model of the world which is itself event driven via callbacks. This mapping won't be easy … but plans are afoot.

The Duktape and Jerry Script engines provide excellent access to standard JS run-times. In addition Espruino is an open source project to provide a JavaScript run-time for embedded devices. It has been implemented for the ARM Cortex M3/M4 processors and others.

The question now is … can these be used for the ESP32? Active projects are attempting to do just that.

See also:

- ESP32-Duktape
- Espruino

Duktape

Duktape is an open source implementation of JavaScript written exclusively in highly portable C with minimal footprint or requirements. It lends itself extremely well to running in embedded systems such as an ESP32.

Being open source, one wondered if it would be able to build as an ESP-IDF component and … yes it does. A project is now under way to provide a distribution.

The core pattern for usage is:

1. Create a context
2. Evaluate JavaScript
3. Delete the context

We create a context by calling `duk_create_heap_default()`.

To evaluate a JavaScript string, we call `duk_eval_string()`.

To delete the context we call `duk_destroy_heap()`.

See also:

- duktape.org
- duktape API reference

Compiling code

The JavaScript that is to be run can be executed in a variety of ways:

- `duk_eval` – Evaluate code on the stack.
- `duk_compile` – Compile code on the stack and leave it there as a function to be called.
- `duk_compile_lstring` – Compile a string representation of JavaScript with a specific length.
- `duk_compile_file` – Compile a file containing JavaScript.
- duk_compile_lstring_filename –
- `duk_compile_string` – Compile a NULL terminated string.
- `duk_compile_string_filename` – Compile a source file who's identity is on the stack.

To invoke the code, we can either use one of the `duk_eval()` functions which compiles and executes or else we can compile and then invoke with `duk_call()`.

Building for ESP32

To our absolute delight, Duktape compiled first time using the ESP-IDF environment without any issues at all.

Integrating Duktape in an ESP32 application

To use Duktape in a C program we must include "`duktape.h`".

We create a context for our JavaScript execute environment using `duk_create_heap_default()`.

The Duktape stack

To work with Duktape C API is unusual relative to some other programming languages and concepts. It is based on a solid understanding of stack processing concepts such as push and pop and the stack frame in which we are working.

Working with object properties

Imagine we wish to set a property on an object. First we push that object onto the stack. Next we push the name and then the value onto the stack. Finally we call `duk_put_prop()` and here is where the magic happens. The top two items are popped from the stack and considered to be the name and value. Next, the object given by its location on the stack from the top has a property created on it given by `name` and set to the value given by `value`. There are variants of this function including:

- `duk_put_prop` – Pop the top 2 items and use as name and value.
- `duk_put_prop_index` – Pop the value from the stack and set to the property with a supplied index.
- `duk_put_prop_string` – Pop the value from the stack and set to the property with the supplied name.

Let us imagine that we want to create a new object with properties. We would do this by:

```
duk_idx_t objIdx = duk_push_object(ctx);
duk_push_string(ctx, "Hello world");
duk_put_prop_string(ctx, objIdx, "greeting");
```

The result will be a new object on the stack with the new property.

We can retrieve values from the stack with `duk_get_???()` functions. These are non-destructive retrievals.

Calling C from a JavaScript program

We can call native C function from a Duktape script. The functions must conform to the syntax:

```
duk_ret_t myFunction(duk_context *ctx) {
   return 0;
}
```

A return of 1 means the top of the stack is to be interpreted as a return value.

A return of 0 means that there is no return value and `undefined` is returned to the caller.

A negative value means an error and should map to `DUK_RET_???`.

To add a global function we would call:

```
duk_push_global_object(ctx); // Push the object into which we are going to set the
function.
duk_push_c_function(ctx, myFunction, argCount); // Push the function onto the stack.
duk_put_prop_string(ctx, -2, "<functionName>"); // Set the object at top-2 to have a
new function property.
```

JerryScript

JerryScript is an open source implementation of JavaScript written exclusively in C with minimal footprint or requirements. It lends itself extremely well to running in embedded systems such as an ESP32.

Being open source, one wondered if it would be able to build as an ESP-IDF component and … yes it does. A project is now under way to provide a distribution.

See also:

- jerryscript.net
- JerryScript & IoT.js: JavaScript for IoT from Samsung
- JerryScript mailing list
- Zephyr OS
- IoT-JS

Platform specific files

When porting JerryScript to a new platform such as the ESP32, there are certain files that are environment specific. These can be found in the `<root>/targets` directory. The `default` directory has proven to work just fine. The functions that have to be implemented are:

- `jerry_port_fatal` – Handle an error from which JerryScript can't recover.
- `jerry_port_console` – Print to the console.
- `jerry_port_log` – Display or log diagnostics. The options levels are:
    - `JERRY_LOG_LEVEL_ERROR` –
    - `JERRY_LOG_LEVEL_WARNING` –
    - `JERRY_LOG_LEVEL_DEBUG` –
    - `JERRY_LOG_LEVEL_TRACE` –
- `jerry_port_get_time_zone` –
- `jerry_port_get_current_time` –

JerryScript life-cycle

To run a JerryScript JavaScript program we go through a life cycle that starts with a call to `jerry_init()`. If we have a buffer that holds the JavaScript source, we then parse that source to create the environment suitable for execution. We do this by calling `jerry_parse()`. When we wish to run the code, we call `jerry_run()`. To cleanup we call `jerry_release_value()` against the parsed code and finally `jerry_cleanup()`.

To interpret code, we have an API call `jerry_eval()` which takes a string of JavaScript and interprets it in the current context. The return is the value of the evaluation which may indicate an error has occurred.

Accessing the global environment

There is a special object that represents the full environment. We can obtain this through a call to `jerry_get_global_object()`.

The jerry_value_t

The data type called `jerry_value_t` is the data type that references a JavaScript value. It can refer to:

- boolean
- number
- null
- object
- string
- undefined

We can create a `jerry_value_t` representing a string by creating one using a C NULL terminated string. For example:

```
jerry_value_t name = jerry_create_string((const jerry_char_t *)"John");
```

We can create a new object instance using `jerry_create_object()`. Given an object, we can set a property within that object using `jerry_set_property()`.

A `jerry_value_t` variable has a reference counter on it. Be sure and call `jerry_release_value()` when you no longer need a reference to it.

A value also has an error flag which can be checked with `jerry_value_has_error_flag()`.

Handling errors

Errors encountered while parsing or running result in a `jerry_value_t` which has an error flag associated with it. We can check that error flag with a call to `jerry_value_has_error_flag()`. It is common that the resulting `jerry_value_t` corresponds to an object that has two properties:

- `name` – The identity (name) of the error … for example "ReferenceError"
- `message` – An English description of the error.

The generation of error messages is optionally compiled in and is off by default. To compile JerryScript with error messages, add

```
-DJERRY_ENABLE_ERROR_MESSAGES
```

to the compilation.

Interfacing JerryScript with C

From within a JavaScript program running in JerryScript, we can invoke a C function. This is a very important concept. It allows us to invoke arbitrary code such as ESP32 specific functions such as WiFi enablement.

The function that we wish to expose from C has to conform to a specific signature as defined by the `jerry_external_handler_t`.

```
jerry_value_t myHandler(
   const jerry_value_t functionObj,
   const jerry_value_t thisVal,
   const jerry_value_t args[],
   const jerry_length_t argsCount)
```

The `args` is an array of arguments passed in.

The `argsCount` is the number of arguments passed in.

We register the function with a call to `jerry_create_external_function().`

```
jerry_value_t jerry_create_external_function(jerry_external_handler_t funcHandle)
```

Here is a full example:

```
static jerry_value_t testFunction(
   const jerry_value_t functionObj,
   const jerry_value_t thisVal,
   const jerry_value_t args[],
   const jerry_length_t argsCount) {

   ESP_LOGI(tag, "testFunction called!");
   return 0;
}

{
   ...
   jerry_value_t testF = jerry_create_external_function(testFunction);
   jerry_value_t name = jerry_create_string((const jerry_char_t *)"f1");
   jerry_value_t global_object = jerry_get_global_object();
   jerry_set_property(global_object, name, testF);
   jerry_release_value(testF);
   jerry_release_value(name);
   jerry_release_value(global_object);
   ...
   // Now within the JavaScript we have a global function called f1() that
   // when it is called will invoke the C function called "testFunction".
}
```

IoT.js

JavaScript is a powerful language but, like both C and Java, provides very little in the way of native libraries. The IoT.js project provides an environment with a rich set of pre-built objects and classes that you can use in your JavaScript applications.

See also:

- IoT.js home page

Programming with Python

Pycom Micropython

See also:

- Github: pycom/pycom-micropython
- YouTube: MicroPython ESP32 Building and loading firmware with Tony D!

Programming with Lua

Lua-RTOS for ESP32

See also:

- Github: whitecatboard/Lua-RTOS-ESP32

Integration with Web Apps

HTTP Protocol

The HTTP protocol is the underpinnings of all web based interactions including browser/web-server, REST requests and WebSockets.

HTTP Headers

Following a HTTP request or response line, there will be zero or more HTTP headers. Think of these as name/value pairs that are used to pass additional information. Most

HTTP headers are well document and architected into the standards while other headers can be used for application logic.

Some headers are common to both requests and responses while others are found only on requests or only on responses.

Accept header

Used to indicate in a request the media types allowed in a response. The basic format is:

```
Accept: <type>/<subtype>
```

If we really don't care, we can specify the wildcards:

```
Accept: */*
```

However if we specify a type, we can wildcard the subtype:

```
Accept: text/*
```

or we can be explicit and define both type and subtype:

```
Accept: text/plain
```

Multiple entries can be supplied separated by commas. We can further qualify a type with a semicolon (";") followed by parameters.

Authorization header

Provides authorization credentials. Remember that unless we are using transport level encryption, these authorization credentials will be visible in a network examination.

Connection header

Used to indicate that the network connection should be closed by a request receiver after it has sent its response.

```
Connection: close
```

Content-Length header

Used to indicate the size of the payload/content.

Content-Type header

Used to indicate the type of data that is being sent in a request or response.

Host header

The Host header is mandatory on an HTTP/1.1 request and identifies who the transmitted claims to be. For example:

```
Host: www.example.org
```

User-Agent header

A header that identifies the originator of the request.

Web Servers

A Web Server is a software component that listens for incoming HTTP requests from Web browsers. On receipt of a request, the web server sends a response. This can be the return of an HTML document for display in a browser or can be a payload of data that forms a service call response. An HTTP request can also include an incoming payload to send data into the ESP32 for processing. There are many implementations of Web Servers that can run within an ESP32 environment.

See also:

- RFC7230 – HTTP/1.1 – Message Syntax and Routing
- HTTP: The Protocol Every Web Developer Must Know – Part 1
- Github: lighttpd - nope
- GNU Libmicrohttpd - nope
- LibHTTPD - nope
- Libonion - nope
- EmbedThis – GoAhead
- Monkey

Mongoose networking library

The Mongoose networking library provides a library implementation and associated API that one can use to build a rich and powerful HTTP server. If we think about the core nature of a web server, we will find that it passively listens on a TCP port waiting for incoming requests. When a request arrives it services that request and then returns to its listening state waiting for the next request to arrive. The request is a text stream encoded in the HTTP protocol. The way mongoose works is to provide a framework following this model. We can tell mongoose to start listening and then when a request arrives, an event handler (a subroutine of code that we are responsible for writing) is invoked. The parameters to the event handler include the nature of the event (there are different types) and a parsed representation of the incoming HTTP request. For example, when a request arrives to retrieve a web page, the mongoose framework will call our event handler and tell us the path to the page being requested. Our code can then send back an HTTP response containing the file. Whether we get that data from a "file" or generate it via a dynamic call in code is immaterial.

To use the framework, we call `mg_mgr_init()` to initialize the environment. Next we bind it to a handler using `mg_bind()`. Finally, we poll the server for work using `mg_mgr_poll()`.

```
void setupMongoose() {
  struct mg_mgr mgr;
  mg_mgr_init(&mgr, NULL);
  mg_bind(&mgr, ":80", evHandler);
  while(1) {
    mg_mgr_poll(&mgr, 1000);
  }
}
```

When a request arrives from a browser, we consider that an event which is handed off to an event handler. The signature of an event handler is:

```
void eventHandler(
  struct mg_connection *nc,
  int event,
  void *eventData)
```

Where:

- `nc` – The network connection that received the event.
- `event` – The type of event that triggered the callback.
- `eventData` – Data associated with the event.

Thus far we will receive callbacks for socket connections. If we wish, we can now register that we wish to parse the incoming data as an HTTP request. We do that by making a call to `mg_set_protocol_http_websocket()`. For example:

```
struct mg_connection *c = mg_bind(&mgr, ":80", mongoose_event_handler);
mg_set_protocol_http_websocket(c);
```

When setup as an HTTP server, an incoming browser request will appear with the event type of `MG_EV_HTTP_REQUEST`. The `eventData` passed in will be an instance of `struct http_message` from which the nature of the request can be determined.

Should we wish to setup debugging, we can compile with:

- `-DCS_ENABLE_DEBUG=1` – Specify if we want low level debugging.

`MG_EV` event types we might see include:

- `MG_EV_ACCEPT` – We have accepted a new connection.
- `MG_EV_RECV` – Data has been received. The `ev_data` is a point to an int describing how many bytes of data were received.

- MG_EV_SEND
- MG_EV_CLOSE – The connection has been close.
- MG_EV_HTTP_REQUEST – Data is an instance of a pointer to a "struct http_message".
- MG_EV_HTTP_REPLY – A response has been detected that is a response/reply to a previous HTTP request. Data is an instance of a pointer to a "struct http_message".

The struct http_message describes the nature of the message and contains:

- message – The whole incoming message. The type is a struct mg_str.
- method – The method sent with the request … eg. "GET". The type is a struct mg_str.
- uri – The URI part of the request … eg. "/hello". The type is a struct mg_str.
- proto
- resp_code
- resp_status_msg
- query_string
- header_names – An array of header names. The type is a struct mg_str.
- header_values – An array of header values. The type is a struct mg_str.
- body – The body of the incoming request. The type is a struct mg_str.

To send a response when we have received a request, we will start by calling mg_send_head(). This takes the status code, the length of the payload and any extra headers. To send the payload we will call mg_send(), mg_printf() or mg_vprintf(). Once all the data that we wish to send has been built, we should set the nc.flags value to include MG_F_SEND_AND_CLOSE to indicate that we have finished.

Note that some of the string data types are coded as the special type "mg_str" which is not the same as a null terminated string. Specifically, an mg_str is a C structure that contains two fields:

- p – A pointer to data.
- len – The length of the data.

We have some utility functions to work with instances of mg_str including:

- `struct mg_str mg_mk_str(const char *s)` – Make a new `mg_str` from a null terminated string.
- `struct mg_str mg_mk_str_n(const char *s, size_t len)` – Make a new `mg_str` from a pointer to data and length.
- `int mg_vcmp(const struct mg_str *str1, const char *str2)` – Compare an `mg_str` against a null terminated string.
- `struct mg_str mg_strdup(const struct mg_str s)` – Duplicate an `mg_str`.
- `struct mg_str mg_strcmp(const struct mg_str str1, const struct mg_str str2)` – Compare two `mg_str` instances.

To get a null terminated String from an `mg_str`, the following high level algorithm can be used:

```
char *mgStrToStr(struct mg_str mgStr) {
   char *str = (char *)malloc(mgStr.len + 1);
   memcpy(str, mgStr.p, mgStr.len);
   str[mgStr.len] = 0;
   return str;
}
```

Here is an example mongoose event handler that is used to return the time since ESP32 start-up when `/time` is the path request.

```
void mongoose_event_handler(struct mg_connection *nc, int event, void *eventData) {
   switch (event) {
   case MG_EV_HTTP_REQUEST:
      printf("HTTP REQUEST\n");
      struct http_message *message = (struct http_message *)eventData;

      char *uri = mgStrToStr(message->uri);

      if (strcmp(uri, "/time") == 0) {
         char payload[256];
         sprintf(payload, "Time since start: %d ms", system_get_time()/1000);
         int length = strlen(payload);
         mg_send_head(nc, 200, length, "Content-Type: text/plain");
         mg_printf(nc, "%s", payload);
      }
      else {
         mg_send_head(nc, 404, 0, "Content-Type: text/plain");
      }
      nc->flags |= MG_F_SEND_AND_CLOSE;
      free(uri);
      break;
   }
}
```

The determination that the event is an HTTP request and the examination of the path is such a common occurrence that a helper function is provided called mg_register_http_endpoint(). This function registers a relative URI path and an event handler to be called when a request arrives that matches the path. This makes it easy for us to break out different paths.

Since the ESP32 uses co-routines to provide an illusion of parallelism, this then asks the question about whether we can make mongoose API calls from multiple tasks. The answer is no. Instead we must invoke mg_broadcast() to broadcast an event. The signature of mg_broadcast is:

```
mg_broadcast(struct mg_mgr *, mg_event_handler_t func, void *, size_t)
```

By default, this feature is disabled, to enable we have to add the following:

```
CFLAGS+=-DMG_ENABLE_BROADCAST
```

See also:

- Github: cesanta/mongoose
- Mongoose Developer Centre
- Mongoose WebSocket

Setting up Mongoose on an ESP32

The recipe I follow to setup Mongoose is as follows:

1. In your project directory, create a components directory.
2. In the components directory, clone Mongoose:

```
git clone https://github.com/cesanta/mongoose.git
```

3. In the new Mongoose directory, create a file called component.mk containing:

```
COMPONENT_ADD_INCLUDEDIRS=.
```

4. Compile your solution.

As you see there isn't much to it; it just works.

Sending a request from Mongoose

To send an HTTP request outbound from Mongoose we can use the mg_connect_http() API. This has the signature:

```
struct mg_connection *mg_connect_http(struct mg_mgr *mgr,
   mg_event_handler_t eventHandler,
   const char *url,
   const char *extraHeaders,
   const char *postData)
```

The eventHandler is a function that will be called to process events for this request.

The Mongoose struct mg_connection

There are a number of fields/properties in the `struct mg_connection` that might be of use to use. Here is a list of some of the more important/interesting ones:

- `union socket_address sa` – The address and port number of the other end of the network connection. We can use the `mg_conn_addr_to_str()` function to convert the `mg_connection` into a printable string.

Handling file uploads

To understand file uploads, we should start by thinking about an HTTP POST request with `multipart/form-data` encoding. In this scheme, the body of the HTTP POST contains multiple sections. Each section can contain a "`name`" attribute and the body of the section is the value of that attribute. For files, the section also contains an attribute called "`filename`" which is the simple name of the file that was supplied by the user.

If mongoose is compiled with the definition

```
MG_ENABLE_HTTP_STREAMING_MULTIPART
```

then additional events are presented during an incoming HTTP request. These events are:

- `MG_EV_HTTP_MULTIPART_REQUEST` – Indication of the start of a set of multipart form data sections.
- `MG_EV_HTTP_PART_BEGIN` – Indication that the start of a part is beginning.
- `MG_EV_HTTP_PART_DATA` – Data is being made available.
- `MG_EV_HTTP_PART_END` – The end of the part.
- `MG_EV_HTTP_MULTIPART_END` – Indication of the end of a set of multipart form data sections.

Associated with multi-part handling is the data structure called:

```
struct mg_http_multipart_part
```

This contains:

- `const char *file_name` – The name of the file.
- `const char *var_name` – The name of the variable.
- `struct mg_str data` – The data of the variable/file.
- `int status` – Status code.
- `void *user_data` – User taggable data.

A utility function is provided called `mg_file_upload_handler()`. This can be called when an `MG_EV_HTTP_PART_BEGIN`, `MG_EV_HTTP_PART_DATA` or `MG_EV_HTTP_PART_END` events are received. The signature of this function is:

```
void mg_file_upload_handle(
   struct mg_connection *nc,
   int ev,
   void *ev_data,
   mg_fu_fname_fn local_name_fn)
```

The local_name parameter is a function reference to a function with the signature:

```
struct mg_str upload_fname(struct mg_connection *nc, struct mg_str fname)
```

This function is responsible for determining the name of the file to be created.

GoAhead Web Server

The high level structure of GoAhead is:

See also:

- The EmbedThis GoAhead Web Server

JavaScript Webserver

Running JavaScript on the ESP32 opens up some elegant techniques to serve web pages. Using JavaScript for web servers is discussed in the JavaScript section and not repeated here.

See also:

- ESP32 Technical Tutorials: ESP32 Duktape WebServer

REST Services

The notion of distributed computing dates back many decades. The idea that one computer could perform a service on behalf of another is a classic concept. The thinking is that work could be distributed across systems, data could be centralized or dedicated systems could perform specialized roles. Over the years, many forms of distributed computing have been tried. These include socket servers, remote procedure calls (RPC), Systems Network Architecture (SNA), Distributed Computing Environment (DCE), Web Services and others.

Today (2017), the current incumbent of distributed computing protocols and technology is REST. REST is a simple protocol that leverages the existing Hyper Text Transport

Protocol (HTTP) used as the transport between browsers and web servers. This protocol was build to allow a browser to request data from a remote file system hosted by a web server. It provides HTTP "commands" which include GET, POST, PUT and others. The notion behind REST is more of an accident than a design. REST re-purposes HTTP as a communication conduit from a client to a server where a client makes a REST request and the server offers up a REST service. From the network perspective, it "looks" just like a browser/Web Server interaction but both ends choose to agree on the formation and interpretation of the communication.

When we add an ESP32 into the mix, our desire is two-fold. We want the ESP32 to be able to be a client to external REST service providers and we want the ESP32 to be the target of clients making REST requests. From the partner perspective, it should be unaware that it is interacting with the ESP32 as compared to any other computing device.

REST protocol

The REST protocol is built on top of HTTP.

See also:

- RFC7230 – HTTP/1.1 – Message Syntax and Routing
- HTTP: The Protocol Every Web Developer Must Know – Part 1

ESP32 as a REST client

For the ESP32 to be a REST client, it must build and transmit HTTP requests to the service provider. This will include building HTTP headers, transmitting the data in a form expected by the provider (eg. JSON, XML or other textual representation) and handling the response from the provider which may include interpreting the received payload.

To transmit a REST request is composed of two parts. First it opens a TCP connection to the partner and then transmits the HTTP compliant data down that connection. The first part is easy, the second part is more of a challenge. We could read and understand the HTTP spec and build the request part by part but this would have to be done for each project that wishes to use REST client technology. What would be better is if we had a library that "knows" how to make well formed REST requests and we simply leveraged its existing functions.

Making a REST request using Curl

Not only is curl a command line tool, it is also a very rich library of function that can be used in your own projects. From an architectural perspective, any app that wants to use curl as a library should perform an initialization call, request an opaque "handle", setup parameters on the handle and then perform the request against the handle. The handle is a black-box that encapsulates the nature of the request. It hides from us all the low level details and gives us a higher level abstraction against which we can think of working with network endpoints.

At a high level we call `curl_global_init()` for coarse initialization. Next we get a handle using `curl_easy_init()`. When we are finished with the handle we should call `curl_easy_cleanup()` to dispose of it and release any resources that may have been allocated on its behalf. The handle is the container that is used to associate options. We set options against a handle using `curl_easy_setopt()`. We will use this function to set the target URL at a minimum.

We can define a callback function that is invoked when data is received upon the handle. That callback function is registered against the handle. The signature of the callback function looks like:

```
size_t writeData(void *buffer, size_t size, size_t nmemb, void *userp)
```

Think of this function as receiving a buffer of records where each record is of size bytes and the number of records is `nmemb`. Typically, size will be `1` and `nmemb` will then be the total size of data in the buffer. The function should return the size of data consumed. If it is not `size * nmemb` then that will stop processing.

Once written, we can register the callback with:

```
curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, writeData);
```

Notice that the data callback has a parameter at the end called `userp`. This is a pointer to context data that relates the request to the response. The value passed in to the callback function is the value registered with the handle using the `CURLOPT_WRITEDATA` parameter:

```
curl_easy_setopt(handle, CURLOPT_WRITEDATA, dataPointer);
```

When ready, we can ask the handle to perform its task using:

```
success = curl_easy_perform(handle);
```

It is common that we want to pass HTTP headers with our request. To do this we build a list where each entry in the list is a name/value header entry. We use the data type `struct curl_slist` to hold the list:

```
struct curl_slist *headers = NULL;
```

To add a header we can then use:

```
headers = curl_slist_append(headers, "Content-Type: application/json");
```

To associate the headers with a handle, we call:

```
curl_easy_setopt(handle, CURLOPT_HTTPHEADER, headers);
```

When the list has served its purpose, we can release the resources used to manage the list with a call:

```
curl_slist_free_all(headers);
```

When making a REST request, we commonly wish to set the command in the HTTP payload. We can use the following to set different styles:

To set the command to GET, use:

```
curl_easy_setopt(handle, CURLOPT_HTTPGET, 1);
```

To set the command to POST, use:

```
curl_easy_setopt(handle, CURLOPT_POST, 1);
```

To set the command to PUT, use:

```
curl_easy_setopt(handle, CURLOPT_PUT, 1);
```

These commands will set the Content-Type to the form format. If we wish to send data as-is, don't call either of the above but just specify CURLOPT_POSTFIELDS data by itself.

To send data with the request, we can use:

```
curl_easy_setopt(handle, CURLOPT_POSTFIELDS, data);
```

Where the data is a null terminated string.

The Curl package has a number of great samples. Here is a good instance that makes an HTTP GET request.

```
#include <stdio.h>
#include <curl/curl.h>

int main(void)
{
  CURL *curl;
  CURLcode res;

  curl = curl_easy_init();
  if(curl) {
    curl_easy_setopt(curl, CURLOPT_URL, "http://example.com");
    /* example.com is redirected, so we tell libcurl to follow redirection */
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);

    /* Perform the request, res will get the return code */
    res = curl_easy_perform(curl);
    /* Check for errors */
    if(res != CURLE_OK)
      fprintf(stderr, "curl_easy_perform() failed: %s\n",
              curl_easy_strerror(res));

    /* always cleanup */
    curl_easy_cleanup(curl);
  }
  return 0;
}
```

An excellent resource for testing Curl requests is the "httpbin.org" website. This provides HTTP rest responses for a variety of tests.

To build libcurl we create a components directory and perform some commands within.

The curl source can be downloaded from github using:

```
$ git clone https://github.com/curl/curl.git
```

Next we run:

```
$  ./buildconf
$ ./configure
```

Remove the following from lib/curl_config.h

HAVE_LIBSSL

USE_OPENSSL

HAVE_ZLIB

HAVE_LIBZ

HAVE_SIGSETJMP

HAVE_SYS_POLL

HAVE_POLL

HAVE_POLL_H

HAVE_CLOCK_GETTIME_MONOTONIC

HAVE_NET_IF_H

HAVE_SYS_IOCTL_H

HAVE_SYS_SELECT_H

HAVE_SYS_UN_H

HAVE_NETINET_TCP_H

HAVE_MSG_NOSIGNAL

HAVE_POLL_FINE

USE_UNIX_SOCKETS

HAVE_IFADDRS_H

HAVE_STROPTS_H

HAVE_GETIFADDRS

HAVE_IOCTL_SIOCGIFADDR

HAVE_GETHOSTNAME

HAVE_GETEUID

HAVE_BASENAME

HAVE_SIGNAL

When building curl, we may need to set some definitions:

The following can be found `include/curl/curlbuild.h`:

- `CURL_SIZEOF_LONG` – `4` – The size in bytes of a long.
- `CURL_TYPEOF_CURL_SOCKLEN_T` – `unsigned int` – The data type that is a `socklen_t`.
- `CURL_TYPEOF_CURL_OFF_T` – `long`
- `CURL_FORMAT_CURL_OFF_T` – `"ld"`
- `CURL_FORMAT_CURL_OFF_TU` – `"lu"`
- `CURL_FORMAT_OFF_T` – `"%ld"`
- `CURL_SIZEOF_CURL_OFF_T` – `4`
- `CURL_SUFFIX_CURL_OFF_T` – `L`
- `CURL_SUFFIX_CURL_OFF_TU` – `UL`

We also need a lib/curl_config.h

We will also need a `component.mk` with the following entries:

```
COMPONENT_SRCDIRS:=lib
COMPONENT_PRIV_INCLUDEDIR:=lib include
include $(IDF_PATH)/make/component_common.mk
```

We need to say that we don't have includes:

- `netinet/in.h` – HAVE_NETINET_IN_H
- `arpa/inet.h` – HAVE_ARPA_INET_H
- `net/if.h` – HAVE_NET_IF_H
- `sys/ioctl.h` – HAVE_SYS_IOCTL_H
- `process.h` – HAVE_PROCESS_H

- `fcntl.h` – HAVE_FCNTL_H
- `netinet/tcp.h` – HAVE_NETINET_TCP_H

To enable SSL, we add:

#define USE_MBEDTLS 1

And in the CFLAGS+=-DMBEDTLS_FS_IO to both the CURL and the MBEDTLS component.

See also:

- Curl
- Curl C API
- curl_east_setopt
- libcurl examples

Making a REST request using Mongoose

Using the Mongoose APIs, we can quite easily send a REST request and work with the response. The high level story is to initialize Mongoose with `mg_mgr_init()`, request a connection to the REST service provider with `mg_connect()`, associate the connection as being HTTP oriented and then start processing events. The first event to return will be an `MG_EV_CONNECT` event indicating that we are now network connected. From there we can use `mg_printf()` to send the REST request. When the REST partner responds, we will get an `MG_EV_HTTP_REPLY` event and we have completed our request/response pairing.

ESP32 as a REST service provider

For an ESP32 to be a REST service provider, basically means that it has to play the role of a Web Server and respond to Web Server requests. However, unlike a simple Web Server which simply retrieves and sends file content as a function of the path on the URL, it is likely that the REST service provider will perform some computation when an HTTP client request arrives. For example, if we attached a temperature sensor to the GPIOs of the ESP32, when a REST request arrives, the ESP32 could read the current temperature value and send the encoded result back as a the response to the request.

Being a Web Server basically means listening on a TCP port and when connections arrive, interpreting the data received as HTTP protocol. This would be a lot of work on a project by project basis but thankfully there are a number of pre-written libraries that perform this task for us and all we need concern ourselves with is examination of any parameters passed with the request and performing the logic we wish performed when ever a new request is received.

Once again, mongoose becomes an excellent consideration for being a Web Server and handling incoming requests.

See also:

- Mongoose networking library

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.

Here we see a raw request to upgrade an HTTP connection to a WebSocket connection:

```
GET / HTTP/1.1
Host: 192.168.1.10
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: file://
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Sec-WebSocket-Key: saim6TzFH+zVb4qY2nrh0Q==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

See also:

- html5rocks – Introducing WebSockets: Bringing Sockets to the Web
- The WebSocket protocol – RFC6455
- The WebSocket API

A WebSocket browser app

The highest likelihood is that you will be running your ESP32 as a WebSocket server. This would imply that you are going to have browser hosted applications that will be connecting to a WebSocket server as clients and from there, you will likely be writing some WebSocket client code … if nothing else then for unit testing your server.

Because of that, we will now spend some time talking about what is involved in writing a WebSocket client application.

Let us assume that we will be writing JavaScript hosted in the browser. We start by creating an instance of a WebSocket object passing in the URL to the WebSocket server:

```
var ws = new WebSocket("ws://<somehost>[:<someport>]");
```

The WebSocket API is mostly event driven and there are a number event types of interest to us:

- `open` – Invoked when the connection to the WebSocket server has been established and we are now ready to send or receive data. We can define this with the "`onopen`" property of the WebSocket as a function reference.
- `message` – Receive a message from the server. We can define this with the "`onmessage`" property of the WebSocket as a function reference. The incoming message data passed to the handler function contains a `MessageEvent` which contains:
  - `data` – The payload of the message.
- `error` – Receive an indication that an error was detected. We can define this with the "`onerror`" property of the WebSocket as a function reference.
- `close` – Receive an indication that a request to close the connection was detected. We can define this with the "`onclose`" property of the WebSocket as a function reference.

Event handlers can be registered either with an "`on<Event>`" mechanism or with an `addEventListener()` call.

There are two methods defined on a WebSocket object. Those are:

- `send` – Send data to a WebSocket server. The signatures are:
  - send(String)
  - send(Blob)
  - send(ArrayBuffer)
  - send(ArrayBufferView)
- `close` – Close the connection to a WebSocket server. The close() method takes two parameters:
  - close code – An integer close code describing the reason for the close.

        - 1000 – CLOSE_NORMAL
        - 1001 – CLOSE_GOING_AWAY
    - status message – A string describing the close reason.

Finally, there are a few attributes:

- readyState – The state of the WebSocket connection. Values include:
    - WebSocket.CONNECTING
    - WebSocket.OPEN
    - WebSocket.CLOSING
    - WebSocket.CLOSED
- bufferedAmount – Amount of data that is buffered pending transmission to the WebSocket server
- protocol – The WebSocket server selected protocol being used.

Mongoose WebSocket

Using Cesanta's Mongoose libraries, we can setup a WebSocket server. After setting up a binding for incoming network requests we can call mg_set_protocol_http_websocket(). This will attach an event handler to the network protocol level to handle events associated with WebSockets. Specifically, these are the following web socket events we are interested in:

- MG_EV_WEBSOCKET_HANDSHAKE_REQUEST –
- MG_EV_WEBSOCKET_HANDSHAKE_DONE –
- MG_EV_WEBSOCKET_FRAME –

When an MG_EV_WEBSOCKET_HANDSHAKE_REQUEST is received, the data contains a parsed HTTP request as a struct http_message. When we receive the MG_EVENT_WEBSOCKET_HANDSHAKE_DONE event, we are ready to send and receive web socket messages.

When an MG_EV_WEBSOCKET_FRAME is received, the data contains a reference to a struct websocket_message. The struct websocket_message contains:

- data – The data passed from the partner.
- size – The size of the passed data.
- flags – Flags (unknown).

To send outbound (out from Mongoose) web socket messages we use the `mg_send_websocket_frame()` API call. The signature of this function is:

```
void mg_send_websocket_frame(
   struct mg_connection *nc,
   int op_and_flags,
   const void * data,
   size_t data_len)
```

The `op_and_flags` should be one of:

- WEBSOCKET_OP_CONTINUE
- WEBSOCKET_OP_TEXT
- WEBSOCKET_OP_BINARY
- WEBSOCKET_OP_CLOSE
- WEBSOCKET_OP_PING
- WEBSOCKET_OP_PONG
- WEBSOCKET_DONT_FIN – This can be boolean or'd with one of the above to indicate that we will be continuing.

For example, if we wish to send a "`Hello World`" message we might code:

```
char *message = "Hello World";
mg_send_websocket_frame(nc, WEBSOCKET_OP_TEXT, message, strlen(message));
```

See also:

- Mongoose networking library

Other Websocket implementations

See also:

- WebSockets on the ESP32

Tasker

Tasker is an Android application that automates and scripts tasks to be executed on an Android device. Using Tasker we can create a task which is defined as a sequence of commands and actions to be executed. Next we can create a Profile which maps an event, that when detected, executes a task. Although this is useful, how does that relate to an ESP32? Imagine that the event that occurs is an ESP32 sending a message to your phone. With that notion, an ESP32 can, effectively, trigger anything that one might be able to do with such a phone. For example, it might make a phone call, send an SMS message or capture a photograph.

See also:

- Tasker home page
- YouTube: Tasker 101 Tutorials

AutoRemote

Following on from our discussion of Tasker above, we now have an admission. It appears that Tasker does **not** have the ability to listen for incoming TCP/IP based events and messages. However, because Tasker is extensible and developers can write plug-ins for it, Tasker can be augmented. One such augmentation is the AutoRemote plugin. Using that plugin, a TCP/IP message can then be sent and received by AutoRemote which can then act as a source of events for Tasker.

With AutoRemote configured as a Tasker plugin, we can configure it to listen for HTTP requests. This causes AutoRemote to listen on TCP port number `1817`. The data it is listening for is an HTTP request. For example:

```
http://<phone ip>:1817/sendmessage?message=1
```

With both Tasker and AutoRemote installed, it will still not be listening for incoming WiFi messages over a local WiFi environment unless we are Internet connected. We must run a Tasker Task called "AutoRemote WiFi".

For example, in Tasker:

1. Create a new profile triggered by Event → System → Device Boot
2. Create a New Task associated with the profile
3. Add an action from Plugin → AutoRemote → Wifi
4. In the configuration for the action, check "Wifi Service"

What this will do is start the Wifi Service whenever the device (Android) boots.

Unfortunately, AutoRemote has a serious drawback. It doesn't allow Tasker to send a response back in the original REST request that might contain data that could be used. For example, if we wish to use AutoRemote to send a request that returned the current GPS location, that is simply not possible.

When an AutoRemote request arrives, it sets a number of variables within the Tasker environment that can be used as parameters to Tasker tasks. These include:

- %armessage
- %arpar()

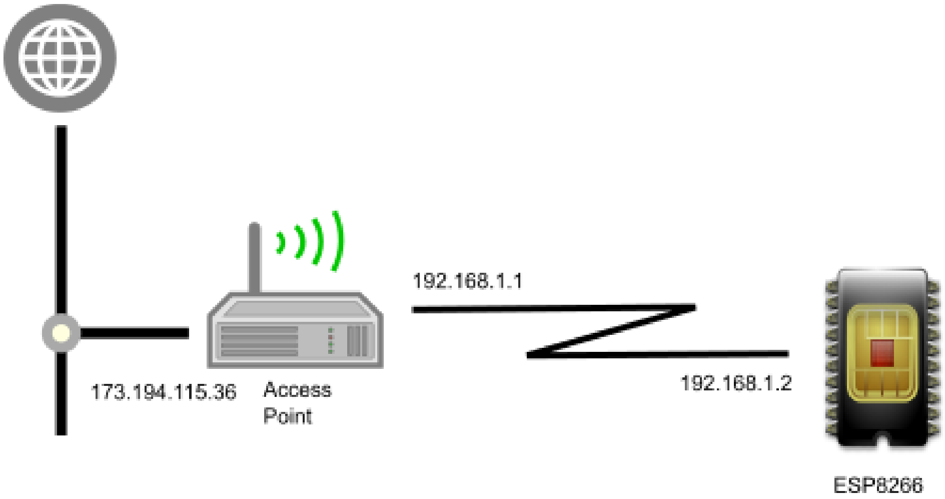
- %arcomm
- %artime
- %arfiles
- %arsenderbtmac
- %arsenderid
- %arsenderlocalip
- %arsendername
- %arsenderpublicip
- %arsendertype
- %arvia
    - wifi

See also:

- AutoRemote home page

DuckDNS

I anticipate that in most folks houses there is a WiFi access point that either directly or through a modem, connects to the Internet. Since the WiFi access point offers a local network to which the ESP32 can join, we now see that the ESP32 can reach the outside world through the access point. However, what about the reverse? What if we want a client on the Internet to reach our ESP32. How could we achieve that?



If we look at the above diagram (all IP address made up), we see that the ESP32 knows its own IP address as 192.168.1.2. However, this can't be "shared" with the Internet as that is a local address and not a global IP address. What would need to be shared is the IP address of the access point as seen on the Internet.

One way to achieve that is through the use of a service provider such as DuckDNS. This free service allows you to register a name. Your device (usually a PC) periodically sends a request to the DuckDNS web site saying "Hello … I am here!". The return address implicitly sent with the request is always the IP address of your access point connected to the Internet and hence DuckDNS learns your external address. Later, someone (perhaps a third party) can ask "What is the IP address" of the name you registered and that address is made available. Essentially, DuckDNS acts as a real-time broker of logical names to IP addresses.

If you are concerned that "some scary person" can learn the IP address of your access point … then don't use DuckDNS. However, for the majority of us, our modem/router/access point prevents incoming traffic from reaching us and essentially blocks anything we don't want. But wait … won't this also block requests to the ESP32? The answer is "yes it will" which is why you have to define port-forwarding. Port forwarding a function of your modem/router/access point that says that when a request arrives for a given port location, automatically forward it to an IP address on your local network … for example, the network address of your ESP32.

https://www.duckdns.org/update?domains=XXX&token=XXX&ip=

Networking protocols

MQTT

The MQ Telemetry Transport (MQTT) is a protocol for publish and subscribe style messaging. It was originally invented by IBM as part of the MQSeries family of products but since has become an industry standard governed by the Oasis standards group. The latest specification version is 3.1.1.

Being Pub/Sub, this means that there is a broker (an MQTT Broker) to which subscribers can register their subscriptions and publishers can submit their publications. Publications and subscriptions agree on the topics to be used to link the messages together. A client can be a publisher, a subscriber or both.

The value of MQTT is that it can be used to deliver data from an application running on one machine to an application running on another. Immediately we seem to see an overlap between MQTT and REST calls but there are some major differences. In a REST environment, when you form a connection from a client to a server, the server must be available in order for the client to deliver the data. With MQTT that is not necessarily the case. The client can publish a message which can then be held by the broker until such time as the receiving application comes on-line to retrieve it. This is a store and forward mechanism.

Every published message must have a topic associated with it that is used to determine which subscribers would be interested in receiving a copy.

The structure of a topic is broken into topic levels separated by a "/". Subscribers can include wild cards in their topic selections of copies of messages that they would like to receive:

- `+` – Single topic level wild-card

eg. `a/+/c`

would subscribe to `a/b/c` and `a/x/c`.

- `#` – Multi topic level wild-card

eg. `a/#`

would subscribe to `a/<anything>`.

MQTT is commonly implemented on top of TCP/IP. Clients connect to the broker (not to each other) over a TCP connection.

There is a quality of service requested by a client. This is encoded in the QoS field:

- `QoS=0` – Send at most once. This can lose messages. At most once means perhaps never.
- `QoS=1` – Send at least once. This means that the message will be delivered. Saying this another way, a message will not be discarded or lost. However, duplicates can arrive … i.e. the message can be delivered twice or more.
- `QoS=2` – Send exactly once. This means that the message will not be lost and will be delivered once and once only.

MQTT also has the capability to buffer messages for subsequent delivery. For example, if a client subscriber is not currently connected, a message can be queued or stored for delivery to the client when it eventually re-connects. We call a client that is not connected an off-line client. For a subscription, we have the choice to deliver all the queued messages for a client or just the last message. To understand the difference, we can imagine a published message that says "I sold your stock for <$$$> price" … we want all such messages sent to the client because they are all of interest. However if we think of a published message of "Today's forecast is sunny and warm" then there may be no need for old messages and only the current weather forecast is of interest to us. During publishing we can declare that a message is eligible for retention and this is called the "retained message". When a client subscribes, it can ask to receive the last retained message immediately … so even if a subscription takes place after a previous publication, it can still receive data immediately.

Clients make their status known to the broker so the broker can tell if a client is connected. This is achieved via a keep-alive/heartbeat. When forming a connection to a broker, the client provides a keep-alive interval (in seconds). If the broker hasn't received a message from the client in this interval then the broker can disconnect the client assuming it to have been lost/disconnected. If the keep-alive interval is set to 0, then there will be no validation from the broker.

If a client connection is lost because of a network disconnection, the broker can detect that occurrence. This is where we get morbid. We define this as the client having "died". In the real world, when someone dies, there may be a last "will and testament" which are the desired instructions of what the person wanted to happen when they die. MQTT has a similar concept. A client can register a message to be published in the event of the clients death. This is remembered by the broker and in the event of the client dieing, the broker will perform the role of the attorney and publish the last registered "will and testament" message on behalf of the deceased client.

The default port number for an MQTT broker is 1883.

See also:

MQTT Hive

Mosquitto

- MQTT.org
- Oasis MQTT spec – 3.1.1
- Mosquitto,org
- YouTube: Internet of Things – Why You Need MQTT
- OASIS Message Queuing Telemetry Transport (MQTT)
- OASIS MQTT V3.1.1 Specification

Mosquitto MQTT

One of the most prevalent implementations of MQTT is called `Mosquitto` and is available as an open source implementation. On a Linux system we would install with:

```
$ sudo apt-get install mosquitto
```

Where `systemd` is installed, `mosquitto` is controlled by it. To see if it is running execute

```
$ systemctl status mosquitto
● mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
   Loaded: loaded (/etc/init.d/mosquitto)
   Active: active (running) since Thu 2016-01-21 21:32:26 CST; 6min ago
```

```
     Docs: man:systemd-sysv-generator(8)
   CGroup: /system.slice/mosquitto.service
           └─12871 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Jan 21 21:32:26 kolban-VirtualBox systemd[1]: Starting LSB: mosquitto MQTT v3.1
message broker...
Jan 21 21:32:26 kolban-VirtualBox mosquitto[12862]: * Starting network daemon:
mosquitto
Jan 21 21:32:26 kolban-VirtualBox mosquitto[12862]: ...done.
Jan 21 21:32:26 kolban-VirtualBox systemd[1]: Started LSB: mosquitto MQTT v3.1 message
broker.
```

The default configuration file for `mosquitto` is `/etc/mosquitto/mosquitto.conf`. Messages from `mosquitto` are logged to `/var/log/mosquitto/mosquitto.log`.

Two sample applications are provided that perform subscriptions and publications. These are `mosquitto_sub` and `mosquitto_pub`. These are distributed as part of the `mosquitto-clients` package.

As a simple test, open two terminal sessions. In one run:

```
$ mosquitto_sub -t greeting
```

When the subscriber is started it will sit quietly waiting for an arriving publication.

In the other run:

```
$ mosquitto_pub -t greeting -m "Hello World"
```

Some other common flags we use with this command include:

- `-r` – Mark this publication as "retained".
- `-q 0|1|2` – Set the Quality Of Service of the message.
- `-d` – Include debugging in the output.
- `-h` – Host (server) to connect to. Defaults to localhost.
- `-p` – Port number to connect to. Defaults to 1883.

You will see the published message appear in the subscriber window.

Let us now dig a little further into a `mosquitto` configuration file. Within there, we can specify multiple "`listener`" entries. These can bind to a port or a port and interface. Within a listener section, we can configure the protocol that the listener should use. By default, it will be plain TCP for normal client connections but we can specify that the protocol should be "`websockets`". For example, adding the following into the `mosquitto.conf` file:

```
listener 8081
protocol websockets
```

will cause `mosquitto` to listen for websocket clients on port `8081`. If you want to listen on **both** standard mqtt protocol and websocket, then you will need multiple listener definitions.

```
listener 1883
protocol mqtt

listener 8081
protocol websockets
```

See also:

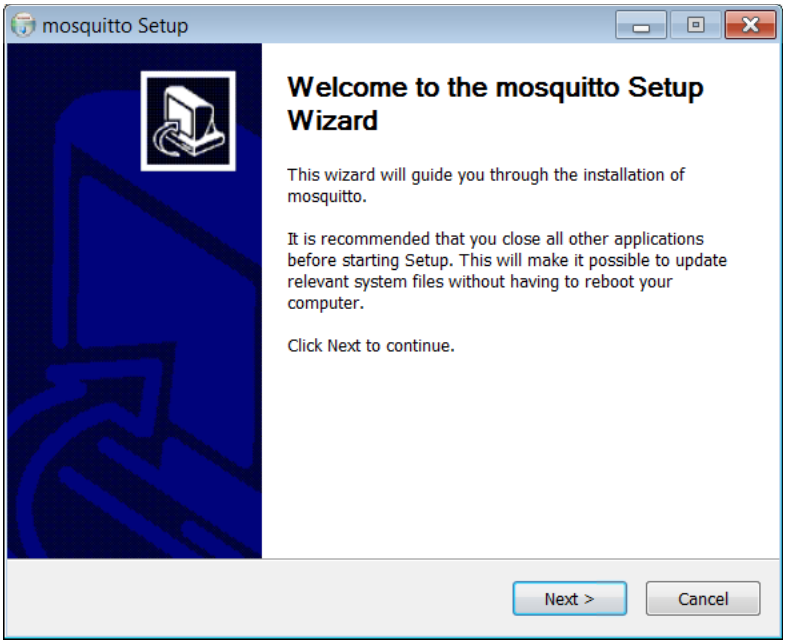
- Moqsuitto.org
- man(1) – mosquitto_sub
- man(1) – mosquitto_pub
- man(5) – mosquitto.conf
- MQTT Community Wiki

Installing on Windows

We can download and install a version of Mosquitto for Windows. Binary downloads are available here:

http://mosquitto.org/download/

The version illustrated here is 1.4.8.



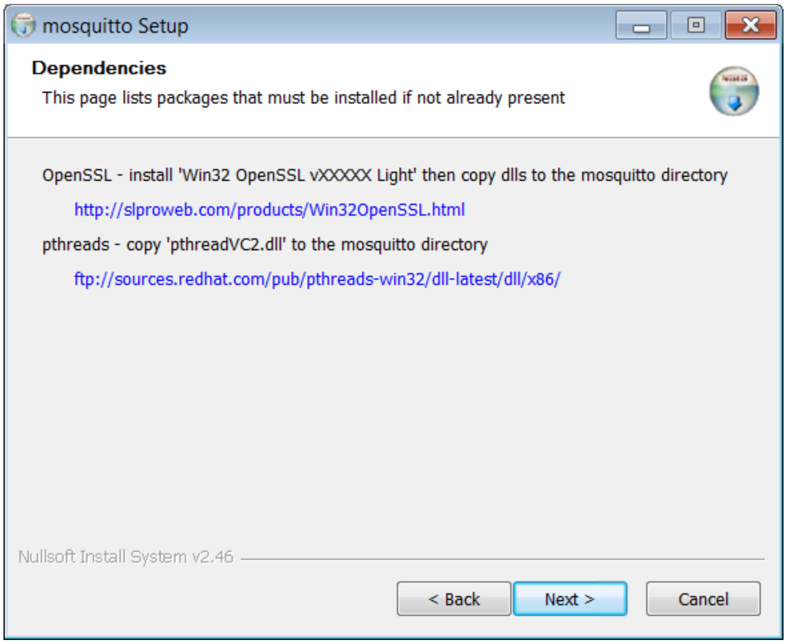
mosquitto Setup

Welcome to the mosquitto Setup Wizard

This wizard will guide you through the installation of mosquitto.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.



Dependencies

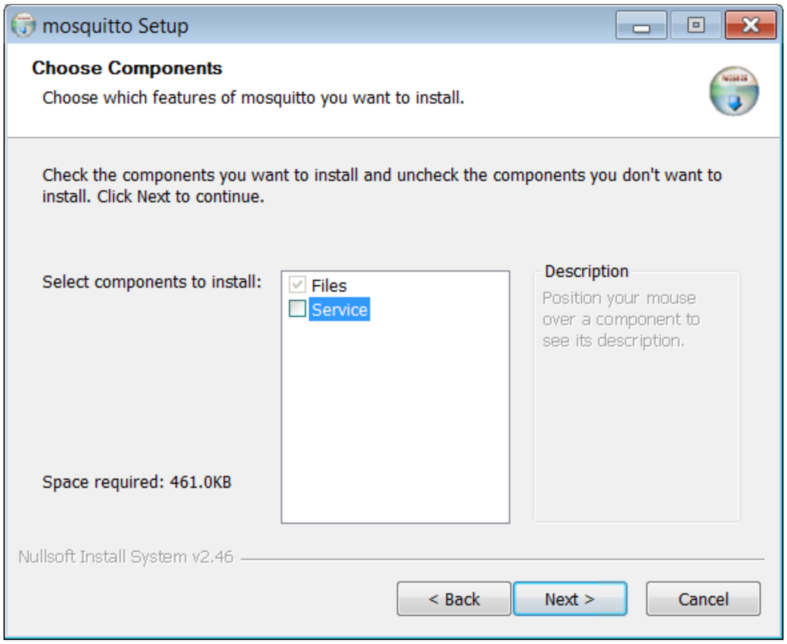
This page lists packages that must be installed if not already present

OpenSSL - install 'Win32 OpenSSL vXXXXX Light' then copy dlls to the mosquitto directory

http://slproweb.com/products/Win32OpenSSL.html

pthreads - copy 'pthreadVC2.dll' to the mosquitto directory

ftp://sources.redhat.com/pub/pthreads-win32/dll-latest/dll/x86/



Choose Components

Choose which features of mosquitto you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

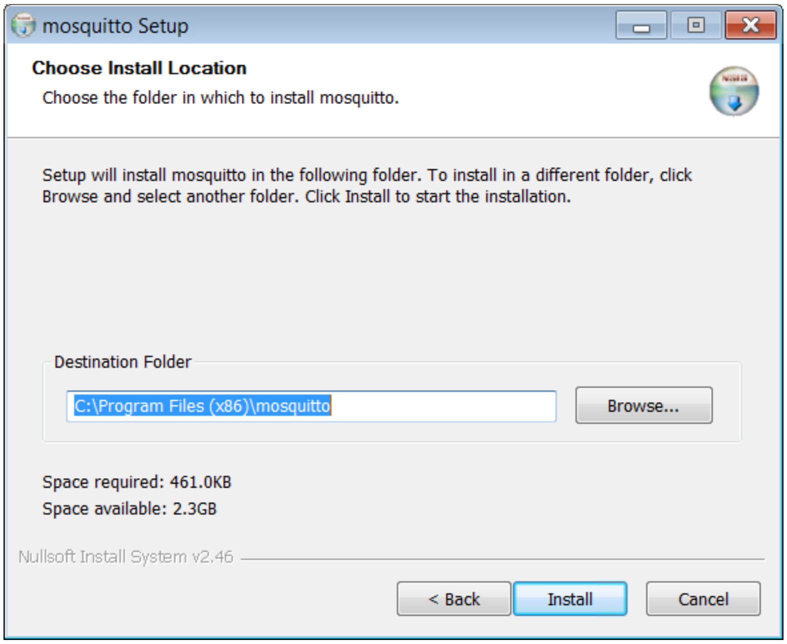
Select components to install:

☑ Files
□ Service

Description

Position your mouse over a component to see its description.

Space required: 461.0KB



Choose Install Location

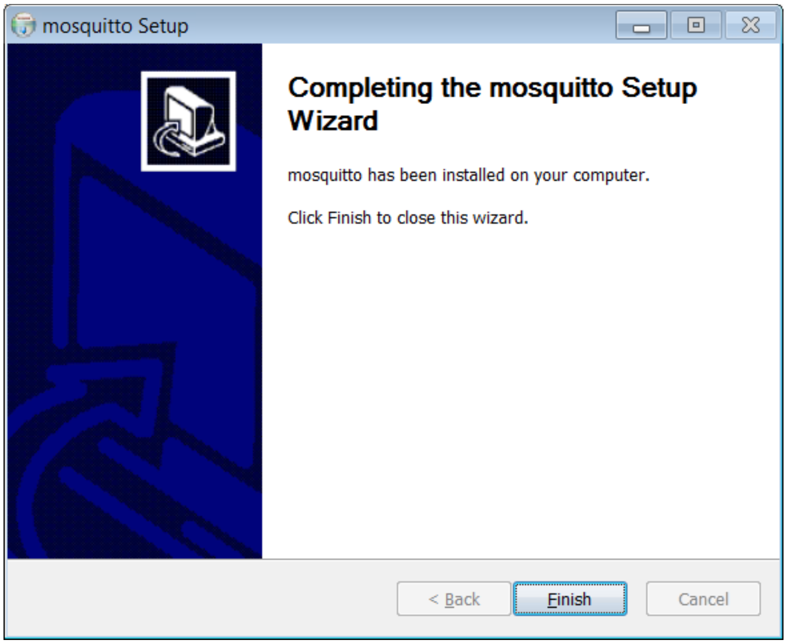
Choose the folder in which to install mosquitto.

Setup will install mosquitto in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\Program Files (x86)\mosquitto

Space required: 461.0KB

Space available: 2.3GB



Completing the mosquitto Setup Wizard

mosquitto has been installed on your computer.

Click Finish to close this wizard.

Writing ESP32 MQTT clients

Using Mongoose as an MQTT client

Once again we can turn to Mongoose to provide higher level networking services. Not only can Mongoose be a Web Server and provide HTTP calling services, it also provide MQTT services.

To be a client, at a high level we perform the following operations:

```
mg_mgr_init(…);
mg_connect(…);
while(1) {
   mg_mgr_poll(…);
}
```

In the event handler we registered in `mg_connect()`, we process events as follows:

MG_EV_CONNECT

```
mg_set_protocol_mqtt(…);
mg_send_mqtt_handshake_opt(…);
```

MG_EV_MQTT_CONNACK

```
// Do something now we are MQTT broker connected ...
```

To publish a message we can call `mg_mqtt_publish()`.

To subscribe, we can call `mg_mqtt_subscribe()`. The signature for a subscription is:

```
void mg_mqtt_subscribe(
   struct mg_connection *nc,
   const struct mg_mqtt_topic_expression *topics,
   size_t topicsLen,
   uint16_t messageId)
```

The `mg_mqtt_topic_expression` is a structure that contains two fields:

- `topic` – `char *` – The identity of the topic on which to subscribe.
- `qos` – `uint8_t` – The quality of service.

Following a subscription, when a publication arrives, we will receive the `MG_EV_MQTT_PUBLISH` event. The payload of the event is an instance of an `mg_mqtt_message` which contains:

Type	Name	Notes
int	cmd	
struct mg_str	payload	The payload of a received message.
uint8_t	connack_ret_code	Used on MG_EV_MQTT_CONNACK.
uint16_t	message_id	Used on MG_EV_MQTT_PUBLISH.
char *	topic	The topic on which the message was received.

Here is a concrete example:

```
void event_handler_mqtt(struct mg_connection *nc, int ev, void *evData) {
   switch (ev) {
   case MG_EV_CONNECT:
      mg_set_protocol_mqtt(nc);
      mg_send_mqtt_handshake(nc, "dummy");
      break;
   case MG_EV_MQTT_CONNACK: {
         struct mg_mqtt_message *msg = (struct mg_mqtt_message *) evData;
         if (msg->connack_ret_code == 0) {
            mg_mqtt_publish(nc, "/greeting", 0, MG_MQTT_QOS(0), "hello", 5);
         }
         break;
      }
   } // End of switch
} // End of event_handler_mqtt


...

struct mg_connection *nc = mg_connect(&mgr, "192.168.5.1:1883",   event_handler_mqtt);
if (nc == NULL) {
   printf("Error with mg_connect()\n");
}
```

See also:

- Mongoose networking library

Using Espruino as an MQTT client

The Espruino JavaScript environment has native MQTT support.

```
var ssid="RASPI3";
var password="password";

var wifi=require("Wifi");
wifi.connect(ssid, {password: password}, function() {
  console.log("Connected to access point");
  var mqtt = require("MQTT").create("192.168.5.1");
  mqtt.on("connected", function() {
    console.log("MQTT connected");
    mqtt.subscribe("test");
    mqtt.on('publish', function (pub) {
      console.log("topic: " + pub.topic);
      console.log("message: " + pub.message);
    });
  });
  console.log("Doing a connect");
  mqtt.connect();
});
```

See also:

- Espruino MQTT

Writing non ESP32 MQTT clients

Having an MQTT environment available is nice but without clients doesn't add much value. Your clients will be the applications that will register subscriptions and publish messages. The subscriber will name the topic on which messages published will be received.

While the MQTT protocol may be standardized, the client APIs appear to not be. As such there are multiple APIs for any given language.

Eclipse paho

We start by creating a network representation object using NewNetwork(&n) followed by a connect to the host and port via `NetworkConnect()`. From there we have a number of MQTT primitives:

- `MQTTClientInit` – Initialize a client.
- `MQTTConnect` – Connect to an MQTT engine.
- `MQTTSubscribe` – Subscribe to a topic and invoke a message handler when a subscription message arrives.
- `MQTTUnsubscribe` – Unsubscribe from a topic.
- `MQTTPublish` – Publish a message to a topic.
- `MQTTDisconnect` – Disconnect from an MQTT server.
- `MQTTYield` – Unknown.
- `MQTTRun` – Run an MQTT client.

The message handler passed to `MQTTConnect()` is invoked when a message is received by the MQTT client after it has been published to the broker. The handler function has the following signature:

```
void function(MessageData *messageData)
```

The `MessageData` data type is a C structure containing:

- `MQTTMessage *message` – The message that was published.
- `MQTTString *topicName` – The topic on which the original message was published.

The MQTTMessage data type is a C structure containing:

- enum QoS qos – The quality of service of the published message.
- unsigned char retained – A flag indicating whether this was a retained message.
- unsigned char dup – A flag indicating whether this message was attempted to be delivered previously and hence may be a duplicate.
- unsigned short id – The identifier of this message.
- void *payload – A pointer to the payload of the message.
- size_t payloadlen – The size in bytes of the message.

Here is an illustrative client that takes a subscription:

See also:

- Paho home page
- Practical MQTT with Paho
- MQTTClientInit
- MQTTConnect
- MQTTDisconnect
- MQTTPublish
- MQTTUnsubscribe
- MQTTSubscribe

C - Mosquitto client library

The library called libmosquitto is available for linking with C applications. You will need to install the package called libmosquitto-dev.

- mosquitto_lib_version – Determine the version of the library in use
- mosquitto_lib_init – Initialize the library
- mosquitto_lib_cleanup – Conclude the use of the library
- mosquitto_new – Create a client
- mosquitto_destroy – Destroy a client
- mosquitto_reinitialize – Destroy a client and then create a new one
- mosquitto_username_pw_set – Set the userid and password for authentication
- mosquitto_will_set – Set the last will of the client for estate planning purposes
- mosquitto_will_clear – Revoke the last will
- mosquitto_connect – Connect a client to a broker

- mosquitto_connect_bind – Same as mosquitto_connect but constrains interface to bind with
- mosquitto_connect_async – Asynchronous connection to the broker
- mosquitto_reconnect – Reconnect to a broker after a lost connection
- mosquitto_reconnect_async – Same as mosquitto_reconnect but asynchronous
- mosquitto_disconnect – Disconnect a client from a broker
- mosquitto_publish – Publish a message on a given topic
- mosquitto_subscribe – Subscribe to messages on a topic
- mosquitto_unsubscribe – Unsubscribe to messages on a topic
- mosquitto_loop – Perform processing for the mosquitto client. It is here that incoming message from the broker are received or previously unsent publications transmitted.
- mosquitto_loop_read –
- mosquitto_loop_write –
- mosquitto_loop_misc –
- mosquitto_loop_forever – Same as mosquitto_loop but does not return and keeps processing until the client is disconnected
- mosquitto_socket – Retrieve the low level TCP/IP socket that the mosquitto client is using
- mosquitto_want_write – Return true if there is pending data to be sent to the broker
- mosquitto_loop_start – Start a thread to process a mosquitto_loop in the background
- mosquitto_loop_end – Stop a previously started loop that was created using mosquitto_loop_start

Here is an example publishing client:

```
#include <stdio.h>
#include <string.h>
#include <mosquitto.h>

int main(int argc, char *argv[]) {
  char *host="pc9100";
  int port = 1883;
  char *message = "hello world!";
  char *topic = "greeting";

  mosquitto_lib_init();
  struct mosquitto *mosq = mosquitto_new(
    NULL,  // Generate an id
    true,  // Create a clean session
    NULL); // No callback param
  int rc = mosquitto_connect(
    mosq,   // Client handle
    host,   // Host of the broker
    port,   // Port of the broker
    false); // No keepalive
  if (rc != MOSQ_ERR_SUCCESS) {
    printf("Error with connect: %d\n", rc);
    return(rc);
  }
  mosquitto_publish(
    mosq,                  // Client handle
    NULL,                  // Message id
    topic,                 // Topic
    strlen(message)+1,     // Length of message
    (const void *)message, // Message to be sent
    0, // QoS = 0
    false); // Not retained
  mosquitto_destroy(mosq);
  mosquitto_lib_cleanup();
}
```

See also:

- man(3) – libmosquitto
- mosquitto.h

Node.js JavaScript – MQTT

There is Node.js package called `mqtt` that provides MQTT functions for JavaScript applications. To install this package we should run:

```
$ npm install mqtt
```

Here is a sample JavaScript application that acts as a publisher:

```
var mqtt = require("mqtt");
var client = mqtt.connect("mqtt://pc9100");
client.on('connect', function() {
  console.log("Connected ... now publishing");
  client.publish("greeting", "Hello from JavaScript");
  client.end();
});
```

And here is an application fragment that acts as a subscriber:

```
var mqtt = require("mqtt");
var client = mqtt.connect("mqtt://pc9100");
```

```
client.on('connect', function() {
  client.subscribe("greeting");
});
client.on('message', function(topic, message) {
  console.log("Received message for topic=" + topic + ", message=" + message);
});
```

Error handling can be accommodated by registering a client handler for an error event:

```
client.on("error", function(error) {
  console.log("We detected an error: " + error);
});
```

The full details of the API can be found on the NPM home page for the MQTT package. At a high level, the functions are:

- `connect` – Connect to a broker.
- `Client.publish` – Publish a message. The message content may be a `Buffer` or a `string`. The identity of the topic on which the message is being published must also be supplied.
- `Client.subscribe` – Subscribe to a topic.
- `Client.unsubscribe` – Unsubscribe from a topic. The signature is:

```
client.unsubscribe(topic, [options], [callback])
```

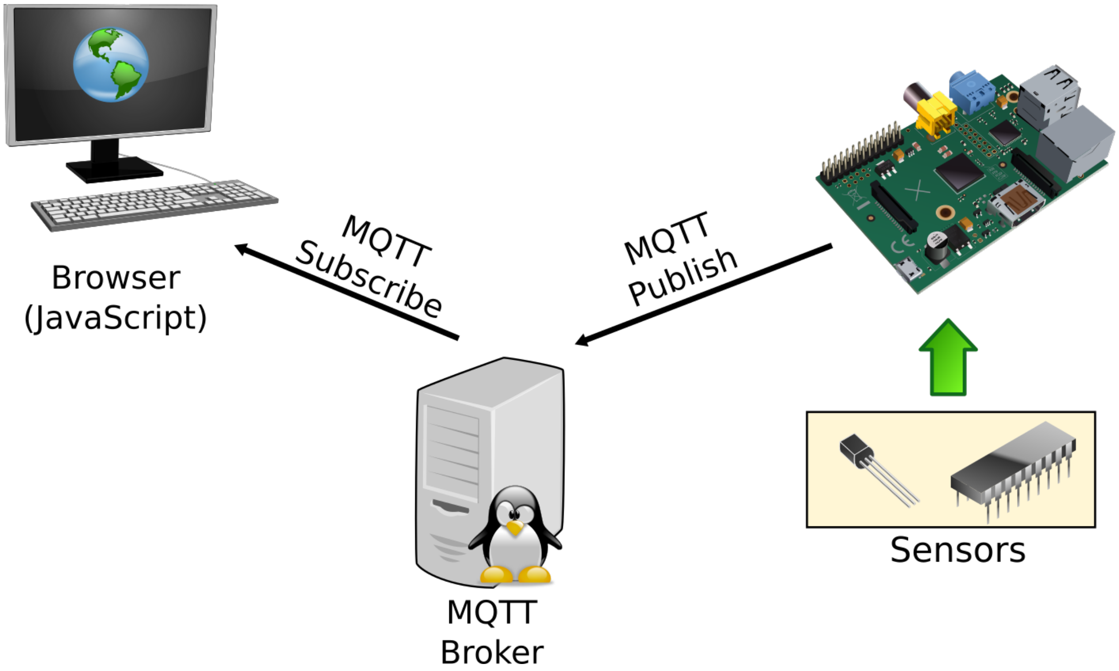
- `Client.end` – Close the client connection to the broker.
- `Client.handleMessage` – A mechanism for handling messages.

See also:

- npm – mqtt

Browser JavaScript – MQTT

The Eclipse Paho project includes a JavaScript client API that is suitable for running within the context of a browser. This client API uses the HTML5 WebSocket API to communicate with an MQTT broker.



The MQTT broker chosen must support MQTT protocol through WebSocket. Popular brokers such as Mosquitto are able to perform that role. From a security standpoint, browsers are only allowed to make WebSocket requests back to the HTTP server from which the page running in the browser was originally loaded. As such, the browser can only make WebSocket requests to the MQTT broker if the MQTT broker served up the page. We can alleviate that issue by introducing full function Web Servers and proxies into the story but for simple purposes, brokers such as Mosquitto can serve up static web pages directly. This means that we can point our browser to Mosquitto which will then look for an HTML file on the local file system to Mosuqitto and send that back to the browser. The browser will now see the MQTT broker as the server of the web page and when the JavaScript in that web page asks for an MQTT subscription, it will succeed. Within the Mosquitto configuration file, each listener definition that wishes to also provide HTTP files must supply an "`http_dir`" option which names the directory that will be searched for HTML file requests from the browser.

We are not limited running an MQTT broker on a separate server machine, we can run the MQTT broker directly on the Pi should we desire.

Now let us look and see what is needed to run an MQTT client in the browser.

First we need to download the Paho JavaScript client. If we visit the Paho downloads page:

https://projects.eclipse.org/projects/technology.paho/downloads

we will find an entry for "JavaScript client 1.0.1". Follow through that link and download the ZIP file. At the time of writing the result will be:

```
paho.javascript-1.0.1.zip
```

we can then unzip the file using `unzip`. Within we will find a file called `mqttws31.js`. This is the JavaScript source of the client and needs to be included in the HTML file that will be using MQTT. Here is an example HTML with embedded JavaScript illustrating us connecting to a broker and registering a subscription. When a message is published to the matching topic, it is logged to the browser console.

```
<html>
  <head>
    <script src="mqttws31.js"></script>
    <script>
var client = new Paho.MQTT.Client("192.168.1.101", 8081, "/", "CLIENT1");
client.onMessageArrived = function(message) {
  console.log("Message arrived ...: " + message.payloadString);
};
client.onMessageDelivered = function(message) {
  console.log("Message delivered ...");
};
client.connect({
  onSuccess: function() {
    console.log("On success ... we are now client connected to the broker!");
    client.subscribe("greeting", {
      onSuccess: function() {
        console.log("Subscription success ...");
      },
      onFailure: function(context, errorCode) {
        console.log("Failed to make a subscription ... code=" + errorCode);
      }
    });
  },
  onFailure: function(context, errorCode, errorMesage) {
    console.log("On failure ... code=" + errorCode + ", message=" + errorMesage);
  }
});
    </script>
  </head>
  <body>
  </body>
</html>
```

Within the `mosquitto.conf` file, we will have an entry that looks like:

```
listener 1883
protocol mqtt

listener 8081
protocol websockets
http_dir /mnt/pc/projects/robot
```

This defines that we will listen on port 1883 for standard MQTT protocol while we will also listen on port 8081 for websocket HTTP requests and if a request to load a page arrives, we will serve it from a given directory.

The API to publish a message is called "send" and takes as a parameter an MQTT message object.

See also:

- Eclipse Paho JavaScript client
- Paho API JavaScript documentation
- The Mosquitto MQTT broker gets Websockets support

CoAP - Constrained Application Protocol

When we typically think of networked computers, we imagine our desktop PCs. These have powerful CPUs, lots of memory, networking cards and unlimited electrical power sources (they are plugged into power outlets). Even our cell phones and tablets aren't too dissimilar to these desktop PCs. Their CPUs are still relatively powerful, RAM is still measured in megabytes and when the batteries in these devices gets low (after many hours of usage) we think nothing about recharging them.

With IoT devices, characteristics change. We have limited CPU capabilities, very low memory availability and electrical power is at a premium. For these devices to communicate, every moment of radio transmission is a drain on power consumption. If our transmission unit of information is 1000 bytes we could consider that 1000 units of consumption. If we could reduce that to a 100 bytes or 10 bytes would the potentiality to improve our power usage by a factor of 10 or 100 … and **that** is important.

If we think about common communication protocols like TCP/IP we will find that they are quite "chatty" and have relatively high overheads. If we go even further up the stack and start considering HTTP as a protocol, the overheads become extremely large. Layer on top of these other data such as JSON or XML … and all of a sudden we find that the number of bytes we are passing for "control" and "structure" as compared to the actual content payload is grossly disproportionate. So … why then do we use these protocols? The answer is standardization to achieve interoperability. We could invent any number of binary protocols that are solution specific but these, by definition, would have difficulty inter operating with each other. In addition, each time we wanted to leverage one of these protocols, we would have to go through some degree of learning curve. Not a great situation.

Thinking this through, we find that what we ideally want is a standardized protocol that is designed to accommodate as many IoT use cases as possible but yet is designed from the ground up to minimize byte sizes for transmission and hence reduce radio "on time". Such a standard has been written and is called "The Constrained Application

Protocol" or "CoAP". This standard has been issued as an IETF request for comments (RFC 7252) which adds credibility to the selection. Let us now look and see how this standard can be utilized in an ESP32 environment.

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Ver		T		TKL				Code								Message ID															
Token (if any, TKL bytes) …																															
Options (if any) …																															
1	1	1	1	1	1	1	1	Payload (if any) ...																							

Ver: Version. Must be 0x01.

T: Type of message:

- Confirmable (0x00)
- Non-Confirmable (0x01)
- Acknowledgement (0x02)
- Reset (0x03)

TKL: Token length – A value between 0 and 8 (inclusive). 9-15 are reserved.

Code: An 8 bit unsigned value that is coded in 3 bits (called the class) and 5 bits (called the detail). The class can be:

- request – 0
- success response – 2
- client error response – 4
- server error response – 5

The options that follow are encoded for minimal space. While the encoding scheme is undoubtedly sensible to achieve minimum data size, it is worthless to describe its minutia here. Suffice it to say that there can be an arbitrary number of options of arbitrary length. Each option is represented by a numeric code and a value of zero or more bytes. There is a special "terminator" option type that can be used to terminate the sequence of options.

See also:

- IETF RFC 7252 – The Constrained Application Protocol (CoAP)

FTP

FTP is the File Transfer Protocol. On Linux we can install an FTP server using:

```
$ sudo apt-get install ftpd
```

and we can install an FTP client using:

```
$ sudp apt-get install ftp
```

TFTP

The Trivial File Transfer Protocol is a simple protocol for moving named files around. The underlying transport is a sequence of data-grams as opposed to a connection.

It has no directory listing capabilities and no security features.

A TFTP server can be installed on Linux using:

```
$ sudo apt-get install tftpd
```

An implementation of a TFTP server is available as part of Kolban's C++ library.

A TFTP server listens on a well known port (69 by default). A client can either request to upload a new file or request to download a file.

For upload, this is called a "write request". The client sends a message of the format:

```
 2 bytes     string    1 byte     string   1 byte
 ------------------------------------------------
| Opcode |  Filename  |   0  |    Mode    |   0  |
 ------------------------------------------------
```

The Opcode in this case will be WRQ (2). On receipt of the request, the TFTP server will create a new socket and bind it to a local port. It will then send an acknowledgment back to the client. An acknowledgment consists of:

```
 2 bytes     2 bytes
 ---------------------
| Opcode |   Block #  |
 ---------------------
```

A WRQ is acknowledged with an ACK packet having a block number of 0. Following the acknowledgment, data packets will be received of the format:

```
 2 bytes     2 bytes      n bytes
 ----------------------------------
| Opcode |   Block #  |   Data     |
 ----------------------------------
```

For a download a "read request" is received. The client sends a message of the format:

```
  2 bytes     string    1 byte     string   1 byte
  ------------------------------------------------
 | Opcode |  Filename  |   0  |    Mode    |   0  |
  ------------------------------------------------
```

This will then cause the TFTP server to send data packets and wait for a corresponding ACK. The TFTP server may send data in chunks of 512 bytes. To signal the end of the transmission, it will send a data block of less than 512 bytes in length. If the last block should happen to be exactly 512 blocks it will send a 0 length block after the ACK.

See also:

- Wikipedia – Trivial File Transfer Protocol
- RFC1350 – The TFTP Protocol Rev 2
- atftp – man(1)

Telnet

An open source implementation of the telnet protocol is available on Github.

When we think of Telnet, we should imagine an initial conversation between the client and the server where they negotiate the options that each supports. Each possible option is described by a `struct telnet_telopt_t` which defines whether "we" support it or not and whether "he" supports it or not. If we support an option, we supply `TELNET_WILL` and if not, we supply `TELNET_WONT`. If we require that the partner must do something we specify `TELNET_DO` otherwise `TELNET_DONT`.

We initialize a telnet environment with a call to `telnet_init()`. This has the signature:

```
telnet_t *telnet_init(
   const telnet_telopts_t *telopts,
   telnet_event_handler_t handler,
   unsigned char flags,
   void *userData)
```

The returned instance of `telnet_t` can be released with a call to `telnet_free()`.

On receipt of data from the partner, we pass that into the telnet environment with a call to `telnet_recv()`. This has the signature:

```
telnet_recv(
   telnet_t *telnet,
   const char *buffer,
   unsigned int size)
```

To send a telnet command, we execute `telnet_iac()`.

To negotiate an option, we use `telnet_negotiate()`.

To send data, we use `telnet_send()` or `telnet_printf()`.

For sub-option negotiation we have the pair `telnet_begin_sb()` and `telnet_finish_subnegotiation()`.

Th event handler is an instance of a `telnet_event_handler_t` which is a function with the following signature:

```
void handler(telnet_t *telnet, telnet_event_t *event, void *userData)
```

A `telnet_event` always contains a field called type which identifies what type of event it is. The choices are:

- data
  - type
  - buffer
  - size
- error
  - type
  - file
  - func
  - msg
  - line
  - errcode
- iac
  - type
  - cmd
- neg
  - type
  - telopt
- sub
  - type
  - buffer
  - size
  - telopt

The telnet types are:

- `TELNET_EV_DATA` – Data received from the partner.
- `TELNET_EV_SEND` – Request to send data to the partner.

See also:

- Github: seanmiddleditch/libtelnet
- RFC854
- RFC855

DNS Protocol

The Domain Name Service protocol is responsible for resolving a domain name (eg. www.google.com) into the IP address that one should use to form a connection to that service provider. A TCP/IP network doesn't use string names but instead uses IP addresses which are typically 4 byte values. Since we humans don't want to try and remember these numbers there is the ability to provide a mapping from text names to those IP addresses. Servers located on the internet collaborate together to provide the mapping from names to addresses. Such servers are called "name servers". The protocol an application uses to resolve a name to a corresponding IP address in the DNS protocol.

DNS servers typically listen on UDP port 53. When a local application has a domain name that it wishes to resolve to an IP address, it passes it to a local component logically called a "resolver". The resolver then works with distributed name servers to obtain the result.

When a UDP message is received by a DNS server, it has the following header:

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td colspan="16">ID</td></tr>
<tr><td>QR</td><td colspan="4">OPCODE</td><td>AA</td><td>TC</td><td>RD</td><td>RA</td><td colspan="3">Z</td><td colspan="4">RCODE</td></tr>
<tr><td colspan="16">QDCOUNT</td></tr>
<tr><td colspan="16">ANCOUNT</td></tr>
<tr><td colspan="16">NSCOUNT</td></tr>
<tr><td colspan="16">ARCOUNT</td></tr>
</table>

- `QR` – Query/Response flag
  - 0 – query
  - 1 – response

- `OPCODE`
    - 0 – query
    - 1 – inverse query
    - 2 – server status request
- `AA` – Authoritative Answer
    - 0 – no
    - 1 – yes
- `TC` – Truncated Message
    - 0 – no
    - 1 - yes
- `RD` – Recursion Desired
- `RA` – Recursion Available
- `Z` – Reserved (must be 0)
- `RCODE` – Response code
    - 0 – no error
    - 1 – format error
    - 2 – server failure
    - 3 – name error
    - 4 – not implemented
    - 5 – refused

A Question section record has the format:

- `QNAME` – a repeating section
    - Length byte = 0 for last entry
    - Label for length bytes
- `QTYPE` – 16 bits
    - 1 – Host record (A)
    - 2 – NS record
- `QCLASS` – 16 bits

- 1 – Internet

A response should also contain the original request.

An answer to a query is in the resource record format. The format of the record looks like:

- `NAME` – The domain name to which this record corresponds. This is the same format as the `QNAME` field.
- `TYPE` – Resource record type.
  - 1 – IPv4 Address
  - others ...
- `CLASS` – The class of data.
  - 1 – Internet
  - others ...
- `TTL` – A 32 bit unsigned supplying the seconds for which this record is valid for. A value of 0 means valid for this response but not to be cached.
- `RDLENGTH` – The size of `RDATA` as an unsigned 16 bit value.
- `RDATA` – The data of the response.

See also:

- DNS Protocol
- RFC1035
- Network sorcery – DNS protocol

Mobile apps

Blynk

Blynk is reported as being able to work with the ESP32 using the Arduino libraries.

See also:

- Blynk home page
- Blynk – ESP32
- YouTube: thingSoC ESP32 with Blynk

Cloud environments

It is becoming increasingly common to leverage the services of a cloud hosted environment to process Internet Of Things (IoT) based data. A cloud provider provides an end-point on the internet which is managed by the provider. It commonly accepts incoming data in a variety of protocols over a variety of transports and handles the data when it arrives. That could be as simple as storing it in a data store or could be more complex such as running a server side application on the received data. In addition to receiving the incoming data, some cloud environment provide the mechanics to manage an IoT based device such as controlling its operation or pushing new firmware to it. All of this is done at scale. The number of devices and the amounts of data needing to be transmitted can be as high as needed. Vendors offer these cloud based services for a fee. After all, running a server and ensuring it is operational requires resources including time, staff and systems. In addition, to accommodate large volumes of huge number of devices, these vendors may be managing high end environments including network redundancy, transparent fail-over and "elasticity" in the provisioned services … growing as needed and shrinking when idle. The fees charged by the vendors are commonly "metric" based fees charged as a function of how much resources you consumed. A customer with 100 devices with 1 message a minute per device doesn't consume the same resources as a customer with 100000 devices with 10 messages a second per device. Consult with the vendors for pricing. Typically, many offer evaluation access (you can play for free for 30 days) or constrained access (you can submit 100 messages a day).

IBM Bluemix

Bluemix is a quality offering from IBM that provides a wealth of cloud based capabilities including application hosting and much more. However, for our story, all we concern ourselves with is the IoT portion. With a Bluemix account, one can create "IoT Services" which run in the Bluemix cloud. These become the "end-points" of an IoT client. For our discussions, such a client is an ESP32. When we configure a Bluemix IoT service we start by defining a logical device type. This allows us to model the characteristics of our IoT device and its properties. With a template of an IoT device type created, we can then tell Bluemix that we have device instances. Each instance is created from the IoT device type model and hence inherits its attributes. When we have modeled a device instance, Bluemix will supply us a security token which is a string of characters. When that actual device connects to Bluemix, the connection request will include the security token plus the identity name of the device. Bluemix will use this pair to validate that the device is who it claims to be.

We can model each instance of physical ESP32 devices in this manner or, we can have each instance use the same device identifier. Flexibility is the name of the game. By giving each device its own device identifier, we can instruct Bluemix to "disallow" specific connections from specific devices should we choose.

The messages sent by the ESP32 to Bluemix are based on the MQTT protocol so if we wish to write an ESP32 based Bluemix application, we would start by ensuring that we can publish MQTT messages. There are a variety of technologies to help us achieve that.

If This Then That - IFTTT

The notion behind the "If This Then That" technology is the simple concept of pairing sources of events with actions that can be performed. The If This Then That (IFTTT) web site allows anyone to pair together a trigger (a source of an event) and action (an action to be performed). The sources of events are rich … extremely rich and there are hundreds to choose from. However, from the perspective of an ESP32 programmer, there is one source that stands out … it is called "Maker".

Once configured, we can send in a REST request with a JSON payload to a URL endpoint. The REST request includes:

- The type of event we are submitting
- A token that identifies the service we are targeting
- Three optional parameter values

Using this as a source of an event, we can then wire that into an action to be performed when such an event arrives. This takes us to the actions that can be performed by IFTTT. Once again there are hundreds of choices to choose from including sending SMS messages, emails, tweets and more … including making a REST request which itself could be targeted at an ESP32.

See also:

- IFTTT

Storage programming

The notion of storage programming encompasses techniques for storing data for later retrieval or examination. The ESP32 has RAM but when the ESP32 is switched off, the RAM content is lost. As such we need a mechanism to make that storage more permanent. The ESP32 usually has access to flash memory that is electrically connected via a dedicated SPI bus. Typically the flash memory size is 4MBytes. We can access the flash memory through the SPI Flash APIs.

See also:

- SPI Flash

Partition table

The ESP32 architects a concept called the partition table which is basically a "map" or "layout" of what is contained within the flash storage. The partition table is found at `0x8000` in flash for a length of `0xC00` bytes providing space for about 95 distinct table entries. Each entry in the table has a record structure which logically contains:

- `type` – The type of the partition. One of:
  - data
  - app
- `subtype` – The sub-type of the partition. One of:
  - nvs – Used for non volatile storage.
  - phy
  - factory
  - coredump – Used to hold core dumps.
  - ota
  - fat – Used for the FAT file system.
- `address` – The offset address (flash) of the start of the partition.
- `size` – The size of the partition in bytes.
- `label` – An optional null terminated string (max 16 characters + NULL)
- `encrypted` – Is the partition encrypted.

The partition table is read-only to our applications and can be accessed with a rich API provided by the ESP-IDF. The table is written into flash storage by the flash tool.

The offset size is optional. Blank offsets will be placed contiguously after the previous data. Offsets are 64K aligned.

A tool called "`gen_esp32part.py`" is available as part of the tooling to build binary representations of the table. We can build a binary table from a comma separated value file using:

```
$ gen_esp32part.py -verify input_partitions.csv binary_partitions.bin
```

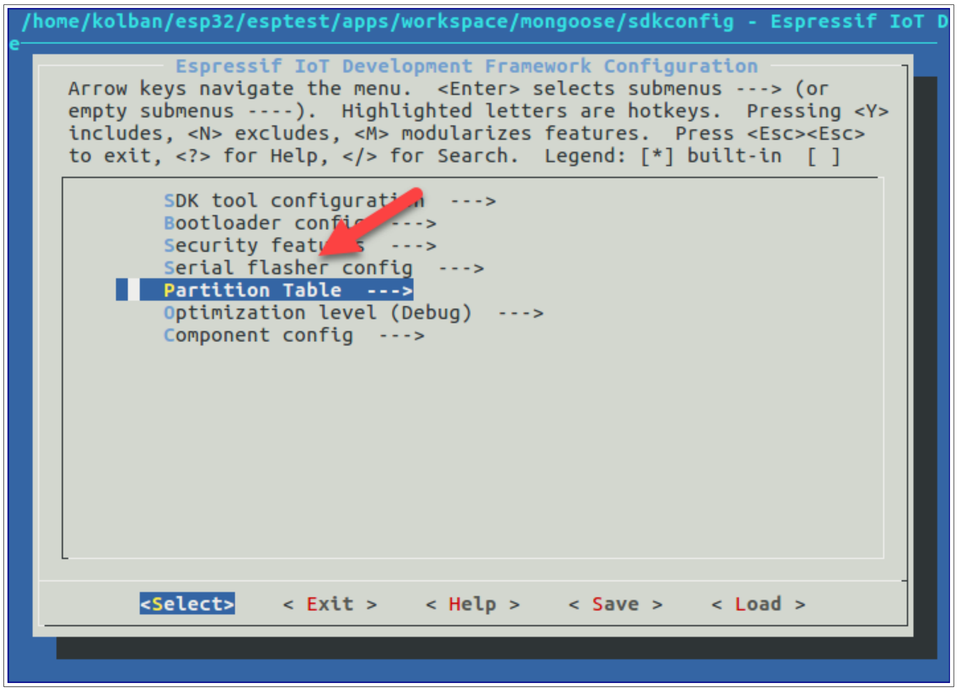
We can convert a binary file back to a CSV using:

```
$ gen_esp32part.py --verify binary_partitions.bin input_partitions.csv
```

and we can list the content of a binary file using:

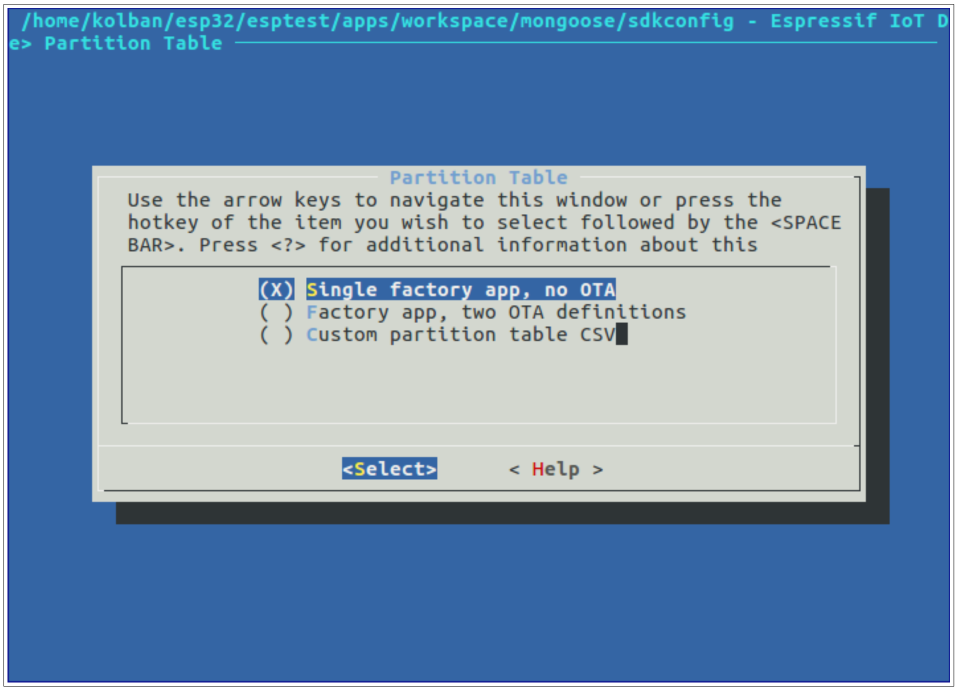
```
$ gen_esp32part.py binary_partitions.bin
```

The partition table used by your application is defined by `make menuconfig` in the Partition Table entry:



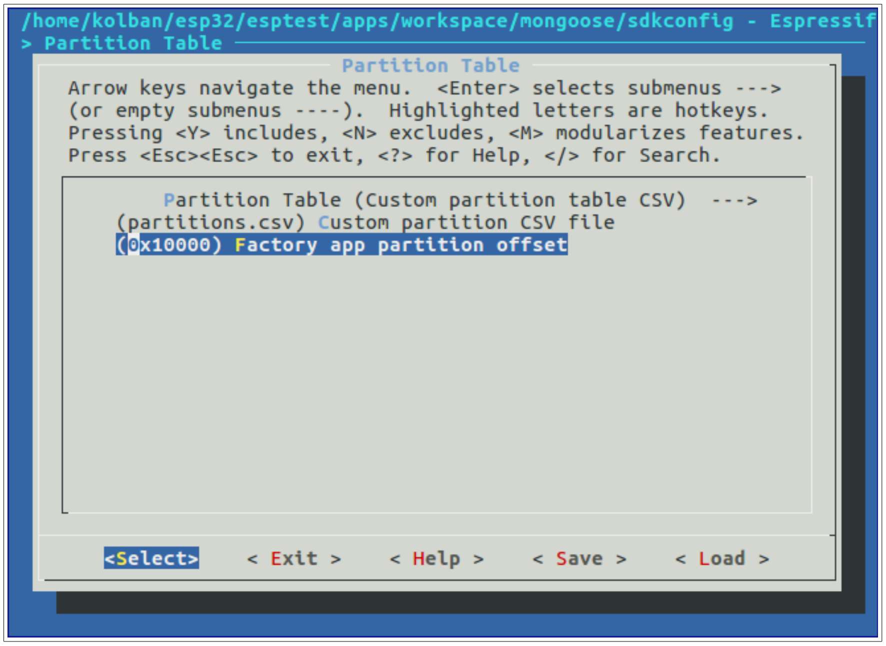
```
 /home/kolban/esp32/esptest/apps/workspace/mongoose/sdkconfig - Espressif IoT D
e
              Espressif IoT Development Framework Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or
  empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc>
  to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          SDK tool configuration  --->
          Bootloader config  --->
          Security features  --->
          Serial flasher config  --->
          Partition Table  --->
          Optimization level (Debug)  --->
          Component config  --->

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Within that section we have three possibilities:



```
 /home/kolban/esp32/esptest/apps/workspace/mongoose/sdkconfig - Espressif IoT D
e> Partition Table ──────────────────────────────────────────────────────────
                            Partition Table
      Use the arrow keys to navigate this window or press the
      hotkey of the item you wish to select followed by the <SPACE
      BAR>. Press <?> for additional information about this

                  (X) Single factory app, no OTA
                  ( ) Factory app, two OTA definitions
                  ( ) Custom partition table CSV

                     <Select>      < Help >
```

If we select to use a custom partition table, there are further options:



```
 /home/kolban/esp32/esptest/apps/workspace/mongoose/sdkconfig - Espressif
 > Partition Table ──────────────────────────────────────────────────
                        Partition Table
   Arrow keys navigate the menu.  <Enter> selects submenus --->
   (or empty submenus ----).  Highlighted letters are hotkeys.
   Pressing <Y> includes, <N> excludes, <M> modularizes features.
   Press <Esc><Esc> to exit, <?> for Help, </> for Search.

           Partition Table (Custom partition table CSV)  --->
       (partitions.csv) Custom partition CSV file
       (0x10000) Factory app partition offset

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

- `Custom partition CSV file` – The Comma Separated Values file that contains the text description of our partition definitions.
- `Factory app partition offset` – The location of our factory application.

I recommend starting with the ESP-IDF supplied partition table as a skeleton. Take a copy of this and place in your project. The supplied file can be found at:

`<ESP-IDF>/components/partition_table/partitions_singleapp.csv`

and copy this to the file:

`paritions.csv`

See also:

- Partition API
- esp_vfs_fat_spiflash_mount
- esp_vfs_fat_register

Non Volatile Storage

Non volatile storage is memory that can be written to such that after a power off or restart, the same data can be read from it again without loss. It is preserved over a restart. Within this data we can store configuration and operational values for our applications. For example, we might store the network SSID and password such that when the device is restarted, it know which network to connect to and the password to present.

The storage is partitioned into named areas. For a given named area, we can then write name/value pairs to the storage and also read name/value pairs. There are getter/setter functions for most data types including signed and unsigned integers, strings and blobs of data.

A named area is opened for access with a call to `nvs_open()`. The name of the area is passed in as a parameter. We are returned a logical "`handle`" that we can subsequently use to refer to this storage area. Once we have a handle, we can get/set items of named data. The data items are referenced by a key name … effectively turning the storage area into a hash map. If we change data by performing a `set` function, this does not automatically cause the data to be written to the nonvolatile storage. Instead, the storage is updated when we call `nvs_commit()`. It is up to the internal implementation as to when the actual update is performed and it *could* happen prior to `nvs_commit()`. The contract is that when we return from `nvs_commit()` then we are assured that all updates have been processed. When we have done all our sets and gets, we should call `nvs_close()` to declare that we are not going to work with storage any more at this time so that the run-time can clean up any resources it may have opened.

The details of the algorithms used to manage NVS are exposed in the documentation. The high level intent of NVS is to store simple strings and integers and other flags as opposed to be a rich "file system" like structure. There is currently no de-fragmentation performed on the storage.

See also:

- nvs_open
- nvs_commit
- nvs_close

Virtual File System

The Virtual File System (VFS) is the architecture provided by the ESP-IDF that gives us the capability of saving and loading data from our applications using a file system I/O. The VFS isn't tied to any one particular technology but is instead an architectural abstraction used to provide the I/O interface to a variety of different implementations.

The key to the VFS is a data type called `esp_vfs_t`. This structure contains the following:

- `fd_offset` –
- `flags` – Operational flags. Use `ESP_VFS_FLAG_DEFAULT`.
- `close`/`close_p` – Close a previously opened file.
- `closedir/closedir_p` – Close a previously opened directory.
- `fstat`/`fstat_p` – Get stats/details of a file.
- `link`/`link_p` – Create a new link to a file.
- `lseek`/`lseek_p` – Change the data pointer within a file.
- `mkdir/mkdir_p` – Create a new directory entry.
- `open`/`open_p` – Open a named file.
- `opendir/opendir_p` – Open a directory for reading.
- `read`/`read_p` – Read the contents of a file.
- `readdir/readdir_p` – Read a record from a directory.
- `rename`/`rename_p` – Rename a file.
- `rmdir/rmdir_p` – Delete a directory entry.
- `seekdir/seekdir_p` – Set the position of the next `readdir()`.
- `stat`/`stat_p` – Get stats/details of a file.
- `telldir/telldir_p` – Return the current direction stream.
- `unlink`/`unlink_p` – Remove a file.
- `write`/`write_p` – Write into a file.

After populating the structure, we need to register our new virtual file system with a call to `esp_vfs_register()`.

We need to be cognizant that the intended caller of file I/O expects a POSIX like environment.

See also:

- esp_vfs_register
- Virtual filesystem component

VFS Implementations

Since the VFS provides an architectural model, we need to consider actual implementations of it. As of 2016-11, none are yet available. The first anticipated implementations will be file systems stored in flash. These will provide persistent storage of data through a file API. Potential implementations will include FAT or SPIFFS.

We can also produce our own specialized implementations. One interesting idea is to allow the ESP32 to be a network client of external file systems. Possibilities include:

- NFS
- SSH
- FTP
- TFTP
- HTTP servers
- Google Drive
- Other cloud based systems

It may seem strange to have a network device access data through a file mechanism only to have it then farm out the requests as another network call … however there may be benefits. The ESP32 could cache the received data either in RAM or local flash and only perform external network requests if the requested data is not available elsewhere.

When working with file I/O, we can use the streams file mechanisms imported via "`stdio.h`" or use the lower level file I/O imported through "`fcntl.h`".

See also:

- VFS mapping to SPIFFS

FATFS File System

The FatFs file system is an implementation of the FAT/exFAT file system as found in earlier PC operating systems such as MS-DOS and early Windows (before FAT32 and NTFS). The implementation is open source and is supplied "pre-ported" to the ESP32 as part of the ESP-IDF distribution.

The ESP-IDF mapping for the FATFS maps the file system to the posix IO functions. This means that we don't need to learn any *special* APIs in order to read and write files. We can use `open()`, `close()`, `read()`, `write()` and the other methods exposed through Virtual File System.

Before we can use these APIs, we need to perform some preliminary setup.

1. Call esp_vfs_fat_register
2. Call ff_diskio_register
3. Call f_mount

To unregister

1. Close all open files
2. Call f_mount with NULL
3. Call ff_diskio_register with NULL
4. Call esp_vfs_fat_unregister

By default, the filenames are constrained to the old 8.3 format (short names), however, should we choose, we can enable long file name control in the make menuconfig settings.

See also:

- FatFS – Generic FAT File System Module
- Virtual File System
- FatFs file system

Spiffs File System

The SPI Flash File System (SPIFFS) is a file system mechanism intended for embedded devices. To configure SPIFFs we need to determine some numbers.

First is the physical page size. Next comes the physical block size. Next we decide on the logical block size. This will be some integer multiplier of the physical block size.

The whole SPIFFS file system must be a multiple of the logical block size. Next comes the logical page size which is some multiplier of the logical block size.

A common ESP32 sizing is 64K for the logical block size and 256 for the logical page size.

To be clear a 1 block is n *x* pages.

When a SPIFFS API call is made, a zero or positive response indicates success while a value < 0 indicates an error. The nature of the error can be retrieved through the `SPIFFS_errno()` call.

The SPIFFS implementation does not directly access the flash memory. Instead, a functional area called a hardware abstraction layer ("hal") provides this service. A SPIFFS integration requires that three functions be created that have the following signatures:

```
s32_t (*spiffs_read)(u32_t addr, u32_t size, u8_t *dst)
s32_t (*spiffs_write)(u32_t addr, u32_t size, u8_t *src)
s32_t (*spiffs_erase)(u32_t addr, u32_t size)
```

If they succeed, the return code should be `SPIFFS_OK` (0). On an ESP32, these will map to the SPI flash APIs.

To use a SPIFFS file system, we must perform a call to `SPIFFS_mount()`. This takes as input a configuration structure that tells SPIFFS how much flash is available and a variety of other properties. In addition, some working storage must be allocated for various internal operations. These sizes can be tuned.

Here is an example of configuration for mounting a file systems:

```
#define LOG_PAGE_SIZE       256

static uint8_t spiffs_work_buf[LOG_PAGE_SIZE*2];
static uint8_t spiffs_fds[32*sizeof(uint32_t)];
static uint8_t spiffs_cache_buf[(LOG_PAGE_SIZE+32)*4];

spiffs fs;
spiffs_config cfg;
cfg.phys_size = 512*1024;                   // use 512K
cfg.phys_addr = 2*1024*1024 - cfg.phys_size; // start spiffs at 2MB - 512K
cfg.phys_erase_block = 65536;               // according to datasheet
cfg.log_block_size = 65536;                 // let us not complicate things
cfg.log_page_size = LOG_PAGE_SIZE;          // as we said

cfg.hal_read_f = esp32_spi_flash_read;
cfg.hal_write_f = esp32_spi_flash_write;
cfg.hal_erase_f = esp32_spi_flash_erase;

int res = SPIFFS_mount(&fs,
   &cfg,
   spiffs_work_buf,
   spiffs_fds,
   sizeof(spiffs_fds),
   spiffs_cache_buf,
   sizeof(spiffs_cache_buf),
   0);
```

Once we have mounted the file system, we can then open a file, write content into it and close it. For example:

```
char *fileName = "/f1/my_file";
spiffs_file fd = SPIFFS_open(&fs, fileName,
   SPIFFS_CREAT | SPIFFS_TRUNC | SPIFFS_RDWR, 0);
SPIFFS_write(&fs, fd, (u8_t *)"Hello world", 12);
SPIFFS_close(&fs, fd);
```

Similarly, if we wish to read data from the file we can perform the following:

```
char buf[12];
spiffs_file fd = SPIFFS_open(&fs, fileName, SPIFFS_RDWR, 0);
SPIFFS_read(&fs, fd, (u8_t *)buf, 12);
SPIFFS_close(&fs, fd);
```

Using the

The SPIFFS file system could be hierarchical in nature such that it contains both directories and files but it seems that in reality it is not. There is only one directory called the root. The root directory is "/". To determine the members of a directory, we can open a directory for reading with the SPIFFS_opendir() API and, when we are finished, close the reading operation with a SPIFFS_closedir() API call. We can walk through the directory entries with calls to SPIFFS_readdir().

For example:

```
spiffs_DIR spiffsDir;
SPIFFS_opendir(&fs, "/", &spiffsDir);
struct spiffs_dirent spiffsDirEnt;
while(SPIFFS_readdir(&spiffsDir, &spiffsDirEnt) != 0) {
  printf("Got a directory entry: %s\n", spiffsDirEnt.name);
}
SPIFFS_closedir(&spiffsDir);
```

To make this clear, in Linux, if we created "/a/b/c.txt" this would normally create a file called "c.txt" in a directory called "b" in a directory called "c". In SPIFFS, this actually creates a single file called "/a/b/c.txt" where the "/" characters are merely part of the file name. When we perform SPIFFS_opendir(), there isn't actually a directory structure but just one single flat list of ALL files which may or may not have "slashes" in their names.

To create a file, we can use the SPIFFS_open() API by supplying a SPIFFS_CREAT flag.

See also:

- SPIFFs API
- Github: pellepl/spiffs
- Github: igrr/mkspiffs – The mkspiffs tool.
- SPI Flash
- Virtual File System mapping to SPIFFS

Building SPIFFs for the ESP32

Under the heading of "let's build on each other", an excellent job has been done of porting SPIFFs to the ESP32 by the LUA team (Jaume Olivé Petrus). The source code can be found:

https://github.com/whitecatboard/Lua-RTOS-ESP32/tree/master/components/spiffs.

They have packaged it as an ESP-IDF component.

mkspiffs tool

In addition to the fantastic SPIFFs library, there is also a tool called "mkspiffs" that can take a directory structure on your file system and build a SPIFFs image from it that can then be loaded into flash memory to provide pre-loaded data.

One can download the Git repository for mkspiffs and compile it. I found no issues and it compiled at first go.

The syntax is:

```
mkspiffs { -c <packdir> | -u <destdir>|-l|-i} \
   -b <number> -p <number> -s <number>
```

One of:

- -c <directory to pack>
- -u <dest to unpack into>
- -l – list content
- -i – visualize content

and

- -b <number> – Block size in bytes (for example 65536)
- -p <number> – Page size in bytes (for example 256)
- -s <number> – fs image size in bytes.

Visualizing an image file shows results such as:

```
   0 idid___________       era_cnt: 0
   1 _______________       era_cnt: 0
   2 _______________       era_cnt: 0
   3 _______________       era_cnt: 0
   4 _______________       era_cnt: 0
   5 _______________       era_cnt: 0
   6 _______________       era_cnt: 0
   7 _______________       era_cnt: 0
   8 _______________       era_cnt: 0
   9 _______________       era_cnt: 0
  10 _______________       era_cnt: 0
  11 _______________       era_cnt: 0
  12 _______________       era_cnt: 0
  13 _______________       era_cnt: 0
  14 _______________       era_cnt: 0
  15 _______________       era_cnt: 0
era_cnt_max: 1
last_errno:  0
blocks:      16
free_blocks: 15
page_alloc:  4
page_delet:  0
used:        1004 of 52961
total: 52961
used: 1004
```

Once we have an image file, we can the load it to flash with:

```
esptool.py --chip esp32 --port "/dev/ttyUSB0" --baud 115200 write_flash -z
--flash_mode "dio" --flash_freq "40m" <address> <file>
```

See also:

- Github: igrr/mkspiffs – The mkspiffs tool.

The ESP File System - EspFs

Part of the Github project known as "`Spritetm/libesphttpd`" is a module called "`espfs`" which is the "ESP File System". What this module does is allow one to make an image from a set of files on your PC and store that image in flash memory. From there, a set of APIs are provided to read and access those files and their content. It is vital to note that the data in these files is read-only. There is no API to update the content of the files. Only the data that is initially written to flash is available to be read.

As part of the project there is a utility called "`mkespfsimage`" that takes as input a set of file names and streams as output the image data that should be flashed. For example:

```
find | ./mkespfsimage [-c compressor] [-l compression_level] > out.espfs
```

(Note that the project has compression capabilities that I am ignoring at this point).

Once the data is in flash, we can then use the APIs supplied by the component to perform the underlying data access.

They are:

- EspFsInitResult espFsInit(void *flashAddress)
- int espFsFlags(EspFsFile *fh)
- EspFsFile *espFsOpen(char *fileName)
- int espFsRead(EspFsFile *fh, char *buff, int len)
- void espFsClose(EspFsFile *fh)

An attempt to port the code to utilize ESP32 technologies was undertaken and can be found here:

https://github.com/nkolban/esp32-snippets/tree/master/filesystems/espfs

This adds a new function called:

- int espFsAccess(EspFsFile *fh, void **buf, size_t *len)

This function returns a pointer to the whole content of the file which is stored in `buf`. The length of the file is stored in `len` and also returned from the function as a whole. The data is accessed directly from flash without any RAM copies.

In addition, the function called:

- EspFsInitResult espFsInit(void *flashAddress, size_t size)

was augmented to include the size of the flash storage to map.

Here is an example application:

```
ESP_LOGD(tag, "Flash address is 0x%x", (int)flashAddress);
if (espFsInit(flashAddress, 64*1024) != ESPFS_INIT_RESULT_OK) {
  ESP_LOGD(tag, "Failed to initialize espfs");
  return;
}

EspFsFile *fh = espFsOpen("files/test3.txt");

if (fh != NULL) {
  int sizeRead = 0;
  char buff[5*1024];
  sizeRead = espFsRead(fh, buff, sizeof(buff));
  ESP_LOGD(tag, "Result: %.*s", sizeRead, buff);

  size_t fileSize;
  char *data;
  sizeRead = espFsAccess(fh, (void **)&data, &fileSize);
  ESP_LOGD(tag, "Result from access: %.*s", fileSize, data);

  espFsClose(fh);
}
```

SD, MMC and SDIO interfacing

Secure Digital (SD) is a standard for removable media. These devices are also known as "flash cards" or "SD cards". The idea is that an SD card contains data that can be both read and written. The SD cards store the data as raw memory and it is common to create a file system that lives on top of the data. The FAT16 and FAT32 file system formats are commonly used. SD cards come in a variety of physical dimensions and with a variety of capacities and speeds. For the physical dimensions there are three distinct types known as "SD", "miniSD" and "microSD" ranging from largest to smallest. For capacity, there are again three distinct types known as "SD", "SDHC" and "SDXC".

	SD	**SDHC**	**SDXC**
Capacity	x <= 2GB	2GB <= x <=32GB	32GB <= x <= 2TB
File system	FAT12, FAT16	FAT32	exFAT

For our story, we will ignore SDXC.

An additional characteristic of SD cards is their rated speed. The common speeds are:

Class 2	2MB/s
Class 4	4MB/s
Class 6	6MB/s
Class 10	10MB/s

The SD specification is large and comprehensive. If we were to try and implement the SD specification ourselves we would be delving down into a whole host of puzzles. As such, it is common to leverage pre-existing implementations of the specification and, thankfully, the ESP-IDF provides us with exactly that.

There is also an excellent example application provide in the examples/storage/sd_card directory of the ESP-IDF.

The SD card can be used to hold data but can not be used to hold instruction code for execution. As such, the SD card shouldn't be considered as an alternative to the flash memory accessible via SPI for code storage. The SD card should be used to store application data that can be read by or written by running applications.

See also:

- Wikipedia: Multi Media Card
- Wikipedia: Secure Digital
- SD Association

ZIP files

The ZIP file format is a common technique for archiving and compressing a set of files into a single file for subsequent extraction.

On Linux, for example:

```
$ zip -r mydata.zip mydir
```

See also:

- libzip
- Github: kuba--/zip
- miniz
- Github: miniz
- man(1) – zip

miniz

```
mz_zip_zero_struct(mz_zip_archive *pZip)
```

Initialize a reader

```
mz_bool mz_zip_reader_init_mem(
   mz_zip_archive *pZip,
   const void *pMem,
   mz_uint64 size,
   mz_uint flags)
```

```
mz_zip_reader_get_num_files(mz_zip_archive *pZip)
```

```
mz_bool mz_zip_reader_end(mz_zip_archive *pZip)
```

```
mz_zip_error mz_zip_get_last_error(mz_zip_archive *pZip);
```

kuba--/zip

zip_open(const char *fileName, int level, char mode)

zip_close

zip_entry_open

zip_entry_close

zip_entry_read

Charting data

Consider some of the sensors we have been discussing. By definition, a sensor reports on data that it measures. We can have our ESP32 perform actions based on the sensed data, however there are other things that we might want to do with that data.

For example, using a temperature and pressure sensor, we almost immediately have a weather station and we might like to chart our data over time. This then leads us to think about different charting options available to us.

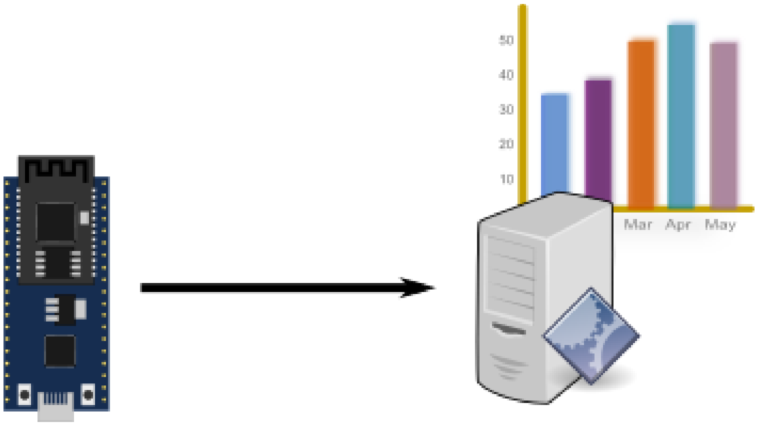
The puzzle of charting data can usually be broken up into a set of distinct areas:

1. Reading data from sensors.
2. Recording / transmitting the sensor data.
3. Drawing / updating the charts from the data.

The first of these we won't discuss here. The reason for that is that the techniques for interacting with a sensor varies by sensor type and, as such, are wide and varied and covered elsewhere in this book.

Once we have started reading data from the sensor, what are we to do with it? There are some obvious possibilities. The first is to save it locally on the ESP32 for subsequent retrieval or analysis. We can write our data into flash memory or into a micro SD card. This data can then be subsequently processed at a later time on the ESP32 or in the case of a micro SD card, the card can be extracted and placed in another reader. While this is of course of some use, the real beauty of the ESP32 is its ability to form network connections. If the ESP32 were to form a network connection to a some "consumer" of the data, then the data could either be extracted at a later time or, as we are about to discuss, could be used to stream the data live from the device.

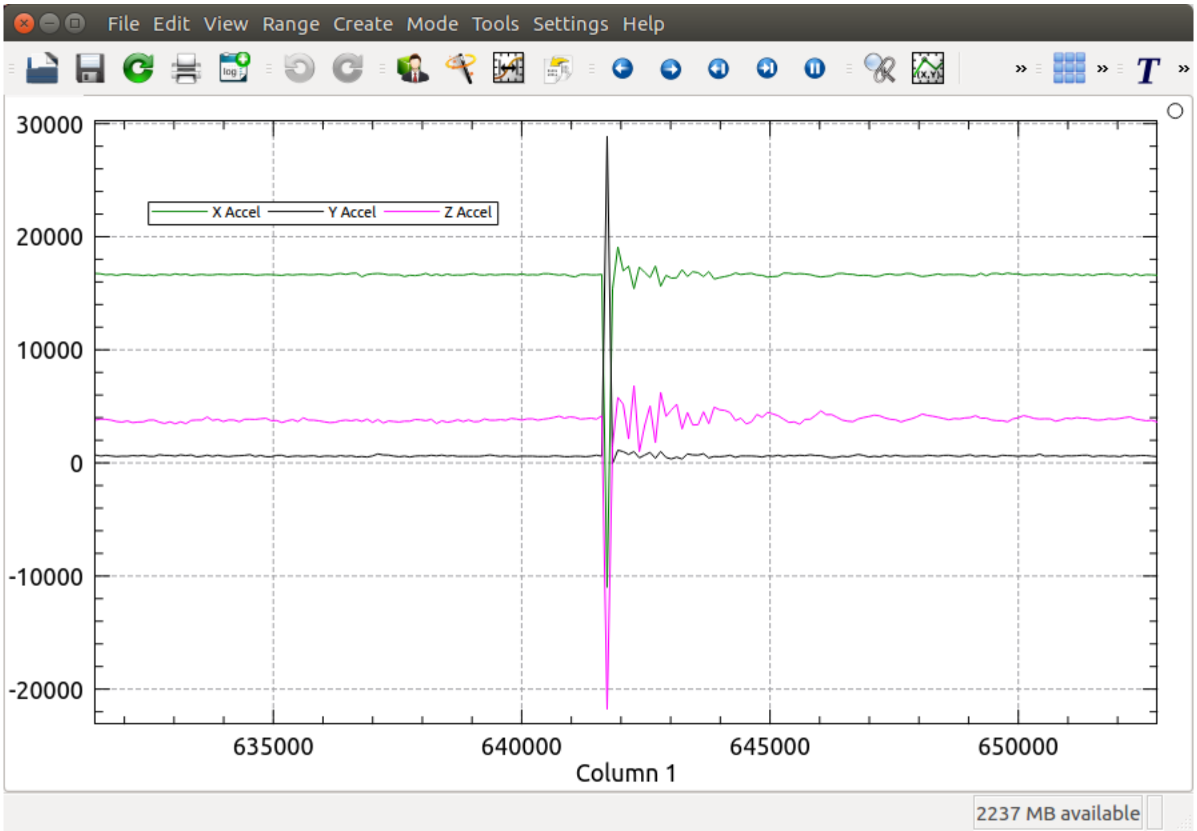
If we read data from a sensor and have a TCP connection to the ESP32 from some external computer, the ESP32 could send the data down the TCP connection for consumption at the receiver. This way the ESP32 would not need to locally save the data as it is immediately off-loaded to the back-end computer. With cloud based technologies, we can choose a cloud provider and send the data there. Alternatively, we can use simple tools to get the data from Windows or Linux as the target of the ESP32 transmitted readings.



An example of such a tool is the Linux "netcat" launched through "nc". This can connected to an arbitrary IP/port pair and echos the data received to stdout. We can redirect the output to a file and we end up with a growing file of data. The ESP32 app reading from the sensor can then send the sensor data (perhaps formatted for output) down the socket and it will end up in the file.

Kst

The latest version of Kst is launched from Linux using "`kst2`".



See also:

- Kst home page

Sample Snippets

There are times when all we need is a snippet of code that we can copy to achieve a task. Here we present a set of such snippets that may of use simply by copying and pasting them. A repository on Github has been created for hosting the snippets. The repository may be found at:

https://github.com/nkolban/esp32-snippets

Included in the repository are:

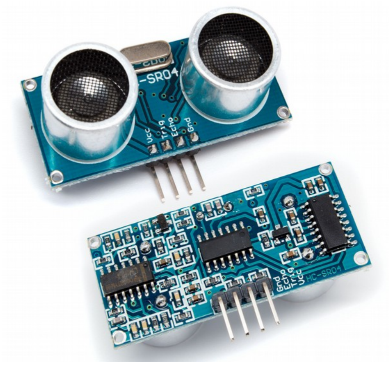
- vfs/vfs-skeleton – A sample skeleton app for VFS

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.

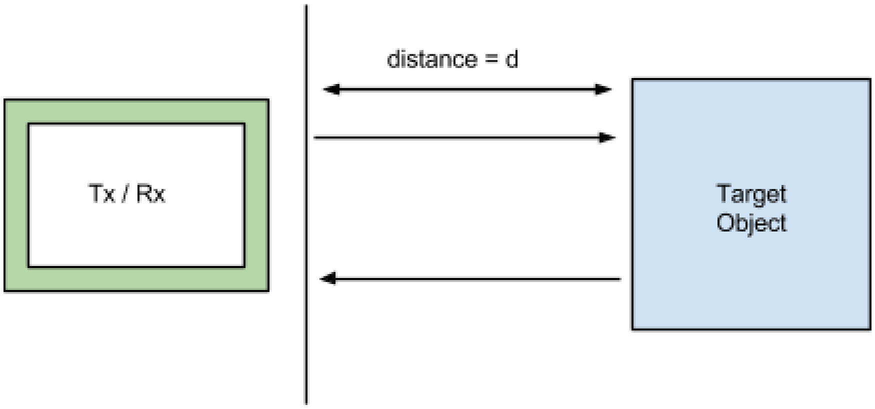
Sample - Ultrasonic distance measurement

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}

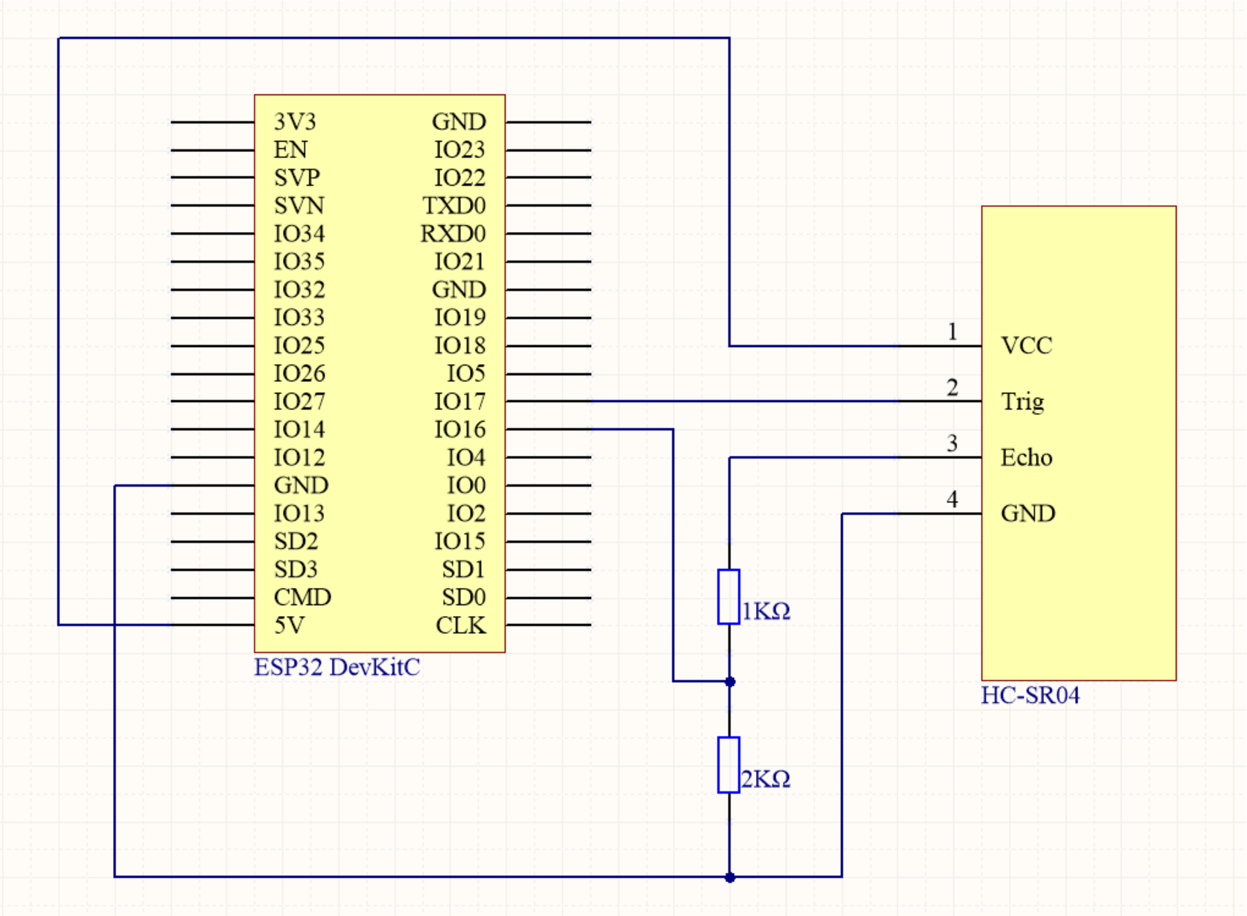
For example:

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 ESP32. 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 ESP32 that we will logically call `Trig` and `Echo`. In my design, I set `Trig` to be GPIO17 and `Echo` to be GPIO16.



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:

```
void mainThread(void *data) {
   gpio_pad_select_gpio(TRIG);
   gpio_pad_select_gpio(ECHO);
   gpio_set_direction(TRIG, GPIO_MODE_OUTPUT);
   gpio_set_direction(ECHO, GPIO_MODE_INPUT);
   while (1) {
      gpio_set_level(TRIG, 1);
      ets_delay_us(100);
      gpio_set_level(TRIG, 0);
      uint32_t startTime = system_get_time();
      // Wait for echo to go high and THEN start the time
      while (gpio_get_level(ECHO) == 0 &&
        (system_get_time() - startTime) < 500 * 1000) {

      }
      startTime = system_get_time();
      while (gpio_get_level(ECHO) == 1 &&
        (system_get_time() - startTime) < 500 * 1000) {
         // Do nothing;
      }
      if (gpio_get_level(ECHO) == 0) {
         uint32_t diff = system_get_time() - startTime; // Diff time in uSecs
         // Distance is TimeEchoInSeconds * SpeedOfSound / 2
         double distance = 340.29 * diff / (1000 * 1000 * 2); // Distance in meters
         printf("Distance is %f cm\n", distance * 100);

      } else {
         // No value
         printf("Did not receive a response!\n");
      }
      // Delay and re run.
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}
```

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 – A changeable mood light

NeoPixels are LEDs that are driven by a single data line of high speed signaling. Most NeoPixels have a +ve and ground voltage source as well as a data line for input and a data line for output. The output of one NeoPixel can be fed into the input of the next one to produce a string of such LEDs. The input data to the LED is a stream of 24 bits of encoded data which should be interpreted as 8 bits for the red channel, 8 bits for the green channel and 8 bits for the blue channel. Each channel can thus have a luminance value of between 0 and 255. By mixing the values for each of the channels together, you can color an LED to any color you may choose. After sending in a stream of 24 bits, if we send in a second stream of 24 bits quickly after the first stream, the second stream is "pushed" through to the next LED in the chain. This can be repeated as far as desired. If we pause sending in data, the current values are "latched" into place and each LED them remembers its own value.

The timings of the data signals for these LEDs can be quite tricky but fortunately great minds have already built fantastic libraries for driving them correctly so we need not concern ourselves with these low level timings and can instead concentrate on devising interesting projects and purposes to which the LEDs can be placed. There are a number of different types of these LEDs with the most common ones being known as WS2811, WS2812 or PL9823.

Within the Espruino JavaScript environment, a method called `neopixelWrite()` can be found. This takes two parameters. The first is the ESP32 GPIO pin that will be used as the source of the signals to the LEDs. It is to this pin that the LEDs should be wired. The pin used for data output from the ESP32 to the NeoPixels should be set in GPIO output mode. For example:

```
pinMode(pin, "output");
```

The second parameter is an array of integers. The values of the array should be supplied in groups of 3 corresponding to the 3 channels of red, green and blue. For example, if we had one NeoPixel connected to GPIO4 on an ESP32 and we wanted to set it to all red, we might code:

```
neopixelWrite(new Pin(4), [255, 0, 0]);
```

If we wanted the next pixel to be green while the first is red, we might write:

```
neopixelWrite(new Pin(4), [255, 0, 0,   0, 255, 0]);
```

Again, there is no obvious limit to the number of LEDs we can string together.

Now that we see that we can set the brightness and color of an LED, let us look at how we might design some code to do something.   Let us imagine that we had a string of 16 LEDs and wanted to make them the same color … we might define a function as follows:

```
function colorLeds(red, green, blue) {
  var data = [];
  for (var i=0; i<16; i++) {
    data.push(green);
    data.push(red);
    data.push(blue);
  }
  esp8266.neopixelWrite(NodeMCU.D2, data);
}
```

If we call this function with the correct red, green and blue values, it will set the LEDs string correctly.

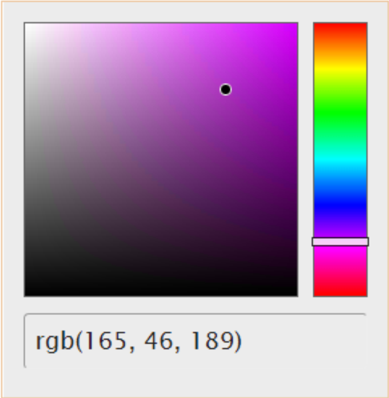
Now let us go one step further. Imagine that we received a network REST request that described the color that we want the LEDs to show. A complete application may be:

```
var esp = require("ESP8266");
var NodeMCU = {
  // D0: new Pin(16),
  D1 : new Pin(5),
  D2 : new Pin(4),
  D3 : new Pin(0),
  D4 : new Pin(2),
  D5 : new Pin(14),
  D6 : new Pin(12),
  D7 : new Pin(13),
  D8 : new Pin(15),
  D9 : new Pin(3),
  D10 : new Pin(1)
};

pinMode(NodeMCU.D2, "output");

function colorLeds(red, green, blue) {
  var data = [];
  for (var i=0; i<16; i++) {
    data.push(green);
    data.push(red);
    data.push(blue);
  }
  esp.neopixelWrite(NodeMCU.D2, data);
}

function beServer() {
  var http = require("http");
  var httpServer = http.createServer(function(request, response) {
    print(request);
    var partsOfUrl = request.url.split("?");
    if (partsOfUrl.length > 1) {
      var options = partsOfUrl[1].split('&');
      var optionsObj = {};
      for (var i=0; i<options.length; i++) {
        var splitEquals = options[i].split('=');
        optionsObj[splitEquals[0]] = splitEquals[1];
      }
      print("Final obj: " + JSON.stringify(optionsObj));
      if (optionsObj.color !== null) {
        var red = parseInt(optionsObj.color.substr(0,2), 16);
        var green = parseInt(optionsObj.color.substr(2,2), 16);
        var blue = parseInt(optionsObj.color.substr(4,2), 16);
        print("red: " + red + ", green: " + green + ", blue: " + blue);
        colorLeds(red, green, blue);
      }
    }
    print("Result url = " + url);
    response.writeHead(200, {
      "Access-Control-Allow-Origin": "*"
    });
    response.end("");
  }); // End of on new browser request

  httpServer.listen(80);
  print("Now being an HTTP server!");
} // End of beServer

var ssid     = "ssid";
var password = "password";

// Connect to the access point
var wifi = require("wifi");
print("Connecting to access point.");
wifi.connect(ssid, password, null, function(err, ipInfo) {
  if (err) {
    print("Error connecting to access point.");
    return;
  }
  var ESP8266 = require("ESP8266");
  print("Connect says that we are now connected!!!");
  print("Starting web server at http://" + ESP8266.getAddressAsString(ipInfo.ip)
+":80");
  beServer();
});
```

When this application runs, it connects to the local WiFI access point and then starts listening for incoming REST requests. A rest request is expected to have a query parameter at the end with the format `color=value` where value is encoded as 6 hex characters corresponding to the color. Finally, we can write a web page that will present a color picker and, when we pick a color, send a REST request to the ESP32 to illuminate the LEDs appropriately. Here is a sample web page to achieve this task:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Set LED colors</title>

<link
   href="http://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.11.2/jquery-ui.min.css"
   rel="stylesheet" type="text/css" />
<script
src="http://cdnjs.cloudflare.com/ajax/libs/require.js/2.1.15/require.min.js"></script>
<link rel='stylesheet' href='spectrum.css' />
<script>
  require
      .config({
        baseUrl : "src",
        paths : {
          "jquery" : "http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min",
          "jquery-ui" : "http://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.11.2/jquery-
i.min",
        },
        shim : {
          "jquery-ui" : {
            deps : [ "jquery" ],
            exports : 'jQueryUI'
          }
        }
      // End of shims
      });
  require([ "jquery", "spectrum", "jquery-ui" ], function($) {
    $(function() {
      var allowHttp = true;
      $("#flat").spectrum({
        flat : true,
        preferredFormat : "rgb",
        move : function(color) {
          if (allowHttp) {
            allowHttp = false;
            $.ajax({
              url : "http://192.168.1.10",
              data : {
                color : color.toHex()
              },
              success: function() {
                allowHttp = true;
              },
              error: function() {
                allowHttp = true;
              }
            });
          }
        },
        showInput : true,
        showButtons : false
      });
    }); // End of on load
  }); // End of require
</script>
</head>
<body>
   <div id="flat" style="width:500px; height: 500px;"></div>
</body>
</html>
```

The end result as seen on the web page looks as follows:



rgb(165, 46, 189)

Selecting a new color causes the data to sent to the ESP32 which colors the LEDs appropriately with the over-all end result being the ability to change the mood light of the LED string.

Using FreeRTOS

When we think of a modern computer, we quickly realize that it has an operating system of some sort. Common examples of these are Microsoft Windows or Linux. The purpose of an operating system is to provide an interface between software applications and the underlying hardware infrastructure. If it wasn't for an operating system, each application would likely have to perform its own similar implementation of such functions which would be a waste. Why not write it once and provide an abstraction layer upon which higher level functions (such as applications) can be built. The capabilities of operating systems on PCs are very similar. They handle memory management, hardware I/O (reading from keyboards and mice and driving graphics cards), task management (multiple programs running concurrently), disk and file system interactions and much more. Early operating systems provided basic functions while today's operating systems have become richer and richer to the point where they may no longer be considered as just operating systems. Since when did an operating system need to provide Freecell or Minesweeper?

If we rewind the clock and start again and look to the core aspects of an operating system, we come to today's FreeRTOS. FreeRTOS is an open source operating system that provides very basic functions to higher level applications … again … the core notion of the purpose of an operating system in the first place. However, FreeRTOS is designed for embedded systems such as the ESP32. It is orders of magnitude simpler than other operating systems such as Linux but this is by design.

FreeRTOS has been ported to a wide variety of hardware platforms including the Xtensa CPUs used in the ESP32. When compiled, it results in a library that is under 5K Bytes in size.

The core functions it provides are:

- memory management
- task management
- API synchronization

See also:

- Free RTOS home page
- Study of an operating system: FreeRTOS
- Github: espressif/ESP32_RTOS_SDK
- Mastering the FreeRTOS real time kernel

The architecture of a task in FreeRTOS

Let us start with the notion of a task. A task is a piece of work that we wish to perform. If you wish, you can think of this as a C language function. For example:

```
int add(int a, int b) {
  return a + b;
}
```

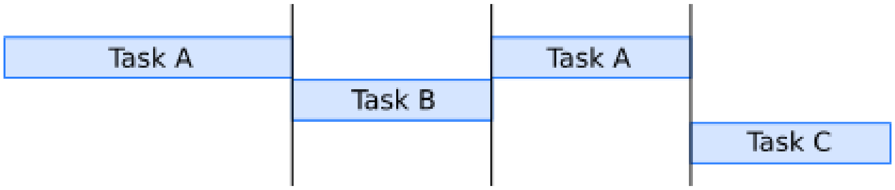
could be considered a task … although this would be ridiculously simple. Generically, think of a task as the execution of a piece of C code that you have authored. We normally think of code running from its start all the way through to its end … however, this is not necessarily the most efficient way to proceed. Consider the idea of an application which wishes to send some data over the network. It may wish to send a megabyte of data … however it may also find that it can only send 100K at a time before it has to wait for the transmitted data to be delivered. In that story, it would send

100K and wait for the transmission to complete, send the next 100K and wait for that transmission to complete and so on. But what of those periods of time where the code is waiting for a previous transmission to complete? What is the CPU doing at those times?

The chances are that it is doing nothing but monitoring the flag that states that the transmission has completed. This is a waste. In theory the CPU could be performing other work (assuming that there is in fact other work that could be performed). If there is indeed other work available, we could "context switch" between these work items such that when one blocks waiting for something to happen, control could be passed to another to do something useful.

If we call each piece of work "a task", that is the value of a task in FreeRTOS. The task represents a piece of work to be performed but instead of assuming that the work will quickly go from start to end, we are declaring that there may be times within the work where it can relinquish control to other work (tasks).

This can be illustrated pictorially in the following. First Task A is running and then it either blocks or else is preempted and Task B runs. It runs for a bit and then there is a context switch back to Task A and finally, Task C gets control. At any one time, an individual core is only ever running one task but because of the context switching, we achieve the effect that over some measured time period, ALL the tasks ran.



With this in mind, we should think about how a task is created. There is an API provided by FreeRTOS called "`xTaskCreate()`" which creates an instance of a task.

Here it is important to realize that a task is a logical abstraction. There isn't anything specific provided in the CPU that knows what a task is. Instead, it is the operating system (FreeRTOS in our case) that is providing the model of the task for us.

If we think deeply about a task, we can conceive of the task having a state. At any given time, either a task is running or it is not running. A task that is running is one that is actively using the CPU (i.e. not waiting for anything else to happen). A task that is not running is one that doesn't have the CPU. For example, if we created two tasks, one of them would be running and the other not running. If the one that is running reaches a point where it can no longer perform meaningful work, it will relinquish CPU control and become not running. The other task then has the opportunity to become running.

Going even deeper, when a task is not running, it may be "not running" for a particular reason … such as:

- Blocked waiting for something to complete
- Suspended by the user
- Ready to run such that when the task that is running is no-longer running, this task is eligible to become running

In FreeRTOS we define a task as a C function that takes a `void *` parameter. For example,

```
void myTask(void *myParameters)
```

might be a signature for a task function.

A task function is expected to run forever. Should it need to end, it should clean itself up before returning by invoking `vTaskDelete()`.

When a task relinquishes control back to the OS, the OS then may have a choice between multiple tasks as to which one should become running. This selection process is called "scheduling". FreeRTOS uses the concept of a "priority" to determine which task to run next. Each task that is ready to run is considered a potential candidate and the one that has the highest priority will become the one that is running.

When coding directly to FreeRTOS in non-ESP32 environment, one would normally have to make a call to `vTaskStartScheduler()` to ensure that the task scheduler is operational. This should not be attempted in the ESP32 environment as the internals of the ESP32 environment have already registered other tasks and already started the scheduler.

- vTaskDelete
- xTaskCreatePinnedToCore
- YouTube: Tasks and concurrent Sockets

Stacks and FreeRTOS tasks

Consider a simple fragment of C code that has a couple of functions:

```
int foo(int a, char b[200]) {
   int c;
   c = bar(a);
   printf("%d %s", c, b);
   return c;
}

int bar(int d) {
   int e;
   e = d * d;
   return e;
}
```

Specifically, I want you to think about the variables in the functions. Hopefully we know that when we create a variable in C that the data that represents the value of the variable is stored in memory. For example, if I define an "`int`" variable, then 4 bytes of storage are required to hold that variable. If I define a variable as "`char[200]`", then 200 bytes of storage are required to hold the maximum value of the variable. The question I pose to you is "where is the storage for those variables allocated?".

The answer to that question is the idea of a "stack". A stack is a contiguous area of storage that contains the "local" or "contextual" variables of your currently executing function. For example, in the function foo shown above, the storage for variables "a", "`b`" and "`c`" is created on the stack. Now let us consider what happens when `foo()` calls `bar()`. At the point when `foo()` calls `bar()`, the address of the currently executing state is pushed onto the stack. This is subsequently used to determine where the called function will eventually return to. Next the parameter passed into `bar()` (i.e. "`int d`") is pushed onto the stack. Finally the storage for variable "`e`" is pushed onto the stack. Now we pass control to the compiled code that represents the entry point for `bar()`. Note that the stack has "grown" … this means that the state of the data that was in place for `foo()` has not been lost … only new storage has been added to provide the context for `bar()`.

When `bar()` eventually returns, the stack allocated storage for the local context is deleted and we return to the stack structure that was in place at the point in `foo()` where we were actually going to call `bar()`. We have "unwound" the stack.

Here are some stack descriptions at different points in the life of our running program.

First here is what the stack looks like within `foo()` just before calling `bar()`.

int a
char b[200]
int c

Next here is what it looks like just after calling `bar()` and while inside the `bar()` function:

int a
char b[200]
int c
return address into foo()
int d
int e

Notice that the stack held values for the function state of `foo()` have not been lost … they are just further back in the stack.

When eventually the code of `bar()` executes a return, we remove the state of `bar()` from the stack and return control to the address of the code within `foo()` that was the location where we called `bar()`.

int a
char b[200]
int c

This story repeats for arbitrary nested function calls. Each time we call a function, we are creating the local context of that function on the stack and passing control to the target entry point of the function. When the function returns we unwind the stack back to its state at the point of call.

While all this is interesting, it feels a little academic for our work in ESP32s. Do we actually have to know this stuff? The answer is "sort of". In the ESP-IDF framework, we have the concept of "tasks" and each task can be thought of as its own thread of control. This means that, logically, there can be many functions executing in parallel to each other. Now let us map that to what we were just talking about relating to stacks and functions. Each task needs to have its **own** stack storage space. The variables in effect for one task are independent and isolated from the variables in effect for a separate task. Since a task can call arbitrary numbers of functions to an arbitrary depth and the amount and types of data can vary from one function call to another, the compiler doesn't know up-front how big the stack might possibly need to be. Instead, the decision of the amount of space to allocate for a stack for a given task is left to the programmer. If we look at the FreeRTOS API for creating a new task, we see that a parameter defines the amount of space to allocate for a stack for this specific task:

```
BaseType_t xTaskCreate(
  pdTASK_CODE pvTaskCode,
  const signed portCHAR *pcName,
  unsigned portSHORT usStackDepth, // <----- Stack size defined here
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pxCreatedTask)
```

How big should you make the stack? The answer is a glib "big enough". Look at your application, estimate how much RAM your application needs to execute and then allocate some function of the remainder to the stack.

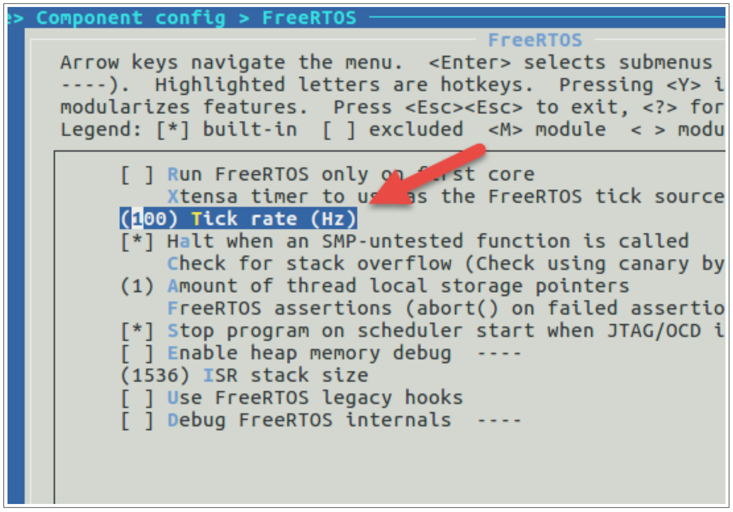
If you size your stack too low, then the internal operation of the ESP-IDF will fail as it will wish to push more data onto the local task stack than there is space within that stack.

The good news is that is "trapped" and your program will stop and report the problem (as opposed to carrying on but broken).

Timers in FreeRTOS

There are a number of timer related functions within FreeRTOS that work on the notion of "ticks" where a tick is a unit of time. The default value of the FreeRTOS tick rate is 100 times per second. This can be set in the "`make menuconfig`" settings found at:

`Component config → FreeRTOS → Tick rate (Hz)`



```
=> Component config > FreeRTOS ──────────────────────────
                                  FreeRTOS
   Arrow keys navigate the menu.  <Enter> selects submenus
   ----).  Highlighted letters are hotkeys.  Pressing <Y> i
   modularizes features.  Press <Esc><Esc> to exit, <?> for
   Legend: [*] built-in  [ ] excluded  <M> module  < > modu

        [ ] Run FreeRTOS only on first core
            Xtensa timer to use as the FreeRTOS tick source
        (100) Tick rate (Hz)
        [*] Halt when an SMP-untested function is called
            Check for stack overflow (Check using canary by
        (1) Amount of thread local storage pointers
            FreeRTOS assertions (abort() on failed assertio
        [*] Stop program on scheduler start when JTAG/OCD i
        [ ] Enable heap memory debug  ----
        (1536) ISR stack size
        [ ] Use FreeRTOS legacy hooks
        [ ] Debug FreeRTOS internals  ----
```

While it seems that the ability to change the value is exposed, I would be wary in doing so. A rate of 100Hz means a tick every 1/100th of a second or every 10msecs.

Rather than think of ticks as absolute numbers, we can thin of them as time durations relative to time. The duration of a tick can be specified as `portTICK_PERIOD_MS` which is the duration of a tick in milliseconds. For example, if I need to wait for 1 second (1000 msecs) then I would wait for 1000 / `portTICK_PERIOD_MS` ticks.

Within FreeRTOS, we can block for a period of time using `vTaskDelay()`. This takes the number of ticks to wait until we unblock. Since the block affects only the current FreeRTOS task, other tasks will continue to be eligible for execution.

See also:

- xTaskGetTickCount
- xTaskGetTickCountFromISR
- vTaskDelay
- vTaskDelayUntil

Blocking and synchronization within FreeRTOS

With the notion of parallel processing tasks within FreeRTOS, we must have a mechanism to synchronize actions between tasks. For example, imagine a task that produces data and a second task that consumes data produced by the first. The producing task must have a mechanism that describes that data has been produced and the consuming task must have a mechanism to block waiting for data to be produced.

One way to achieve this is through the notion of an "event group". Think of an event group as a set of flags that can have the value "0" or "1". A task can set the value of a flag and a second task can be configured to wait (block) until a flag transitions from "0" to "1". What this means is that there is an asynchronous and loosely coupled communication through the use of these flags. From an implementation perspective, FreeRTOS provides a data type called an "event group handle" that is implemented by the opaque data type called "`EventGroupHandle_t`". An instance of this is created through a call to `xEventGroupCreate()`. We should assume that that an event group handle can contain a maximum of 8 distinct flags that are identified as 0 through 7 (however, a note states that there may be up to 24 bits available if `configUSE_16_BIT_TICKS` is set to 0). We can set the flags within an event group using `xEventGroupSetBits()` and clear flags using `xEventGroupClearBits()`. Should we need to obtain the values of an event group, we can call `xEventGroupGetBits()`. Simply toggling bits isn't that useful, but we get into the core of the story with the `xEventGroupWaitBits()` function call. When invoked, it causes the caller to be blocked until a named bit or bits becomes set.

For example:

```
static EventGroupHandle_t eventGroup;
#define EVENT_GROUP_SCAN_COMPLETE (1<<0)

void init() {
   eventGroup = xEventGroupCreate();
   xEventGroupClearBits(eventGroup, 0xff);
}

void doSomething() {
   // Start something in the background and wait
   xEventGroupWaitBits(eventGroup,
      EVENT_GROUP_SCAN_COMPLETE,
      1, // Clear on exit
      0, // Wait for all bits
      portMAX_DELAY);
}

void backgroundEventHandler() {
   xEventGroupSetBits(eventGroup, EVENT_GROUP_SCAN_COMPLETE);
}
```

See also:

- xEventGroupCreate
- xEventGroupSetBits
- xEventGroupWaitBits
- YouTube: ESP32 #25: FreeRTOS – Inter-task Communications – EventGroups

Semaphores and Mutices within FreeRTOS

When we have multiple tasks executing concurrently, we run the risk of those tasks stepping on each other when it comes to working with shared resources.

Consider the following simple illustrative code:

```
int globalCount;

void add(int value) {
   int temp = globalCount;
   temp = temp + value;
   globalCount = temp;
}
```

It adds a value to a global counter. Imagine the counter starts at 0 and two tasks call add(2) at the same time. Logically, after the two tasks complete we want the `globalCount` to equal 4 … we added 2 and 2 to 0 resulting in 4.

However, consider the interleaved execution as follows:

Task	Statment	Local value of temp	Value of globalCount
Task1	int temp = globalCount	0	0
Task 2	int temp = globalCount	0	0
Task 1	temp = temp + value	2	0
Task 2	temp = temp + value	2	0
Task 1	globalCount = temp	2	2
Task2	globalCount = temp	2	2

and as we can see … we have ended up with the wrong resulting value. There are excellent books and articles written on "multi-threading" theory and practices and we will not attempt to regurgitate that material here. Instead, we will cut to the chase and introduce the concept of the "Mutex". The name mutex is a shorthand word for "mutual exclusion". It basically states that only one task can be executing in a section of code at a time. An analogy would be the bathroom at a gas station. In order to use the bathroom, I have to get the key from the clerk. If I visit the clerk and ask for the key he will either give it to me or tell me that he doesn't have it because someone else is already using the bathroom. In the later case, I wait until the other customer finishes and returns the key. At that time the clerk will give me the key and I will go and use the bathroom. While I am using the bathroom, I can be confident that no-one will interrupt me because I am in possession of the key. In this story, the key is an example of the mutex and the bathroom is the shared resource that we are guarding for sole use.

In FreeRTOS, we use the method:

```
SemaphoreHandle_t xSemaphore = xSemaphoreCreateMutex()
```

to create an instance of a mutex.

When we wish to obtain the mutex, we call:

```
xSemphoreTake(xSemaphore, portMAX_DELAY)
```

This will cause the caller to block indefinitely until the mutex is available.

When we are finished with the task that needs guarded, it is **vital** to call:

```
xSemaphoreGive(xSemaphore)
```

to release the mutex and allow anyone else waiting to continue.

See also:

- vSemaphoreGive
- vSemaphoreTake

Queues within FreeRTOS

A queue is a data container abstraction. Think of a queue as holding items. When an item is added to a queue, it is added to the end. When an item is removed from a queue, it is removed from the front. This provides a first-in/first-out paradigm. Think of a queue as the line of folks waiting at the department of motor vehicles. Folks join the end of the queue and those that have arrived earlier are serviced earlier.

In FreeRTOS queues, all items on the queue are placed there by copy and not by reference.

When a queue is created, the maximum number of items it can hold is set and fixed at that time. The storage for the queue is also allocated. This means that we can't run out of storage for the queues when adding new items.

In RTOS we call `xQueueSendToBack()` to add an entry at the end of the queue. The consumer calls `xQueueReceive()` to retrieve the person at the head of the queue. Items in the queue are always stored by copy rather than by reference.

Attempt to receive from an empty queue can cause the caller to block. Conversely, attempting to add an item into a full queue can also cause it to block until space is freed.

When we attempt to read an item from the queue and there is no item to read, the call to read will block. We can specify a timeout value measured in ticks. If no new message arrives in that interval, we will return with an indication that no message was retrieved. We can also specify 0 to return immediately if there are no messages. While a task is blocked reading from a queue that task is placed in the blocked state meaning that other tasks can run. This means that we won't be blocking the system while waiting on a queue. When a message arrives, the task that was previously blocked is returned to the ready state and becomes eligible to be executed by the task scheduler.

See also:

- xQueueCreate
- xQueueCreateStatic
- vQueueDelete
- xQueueOverwrite
- xQueuePeek
- xQueueReceive
- xQueueReset
- xQueueSendToBack
- xQueueSendToFront
- FreeRTOS Queues
- YouTube: ESP32 #20: FreeRTOS – Inter-task Communication – Queues

Ring buffer withing FreeRTOS

A ring buffer is similar to the notion of a queue with a few major distinctions. Firstly, it is strictly first-in/first-out. There are no options to read from the head of the queue, only from its tail. In addition, new items can only be added to the head. While the FreeRTOS queues have slots for a fixed number of items of equal size, the ring buffer can hold items with distinct sizes but all the items must exist within a configured size buffer.

To create a ring buffer we call `xRingbufferCreate()` passing in the size of the buffer we wish to allocate (in bytes) as well as how we wish to handle items that may need to wrap-around.

Calls can then be made to insert records of a given length into the buffer using the `xRingbufferSend()` function. Depending on our request and the way the buffer was created, if there is insufficient space in the buffer to insert the record, this call can block for a given period of time or indefinitely until space becomes available.

A consumer that wishes to process records added to the buffer can call `xRingbufferReceive()` to obtain a record if one is available or else block until one does become available or a time interval elapses. It is important to note that a pointer to the storage of the record within the buffer is returned. This means that the record has NOT been removed from the buffer and its space has not yet been reclaimed. When the consumer has received a record and has finished processing it, the consumer should call `xRingbufferReturnItem()` to indicate that it is no longer needed. At that time the storage for it will be released.

- xRingbufferCreate
- xRingbufferReceive
- vRingbufferReturnItem
- xRingbufferSend

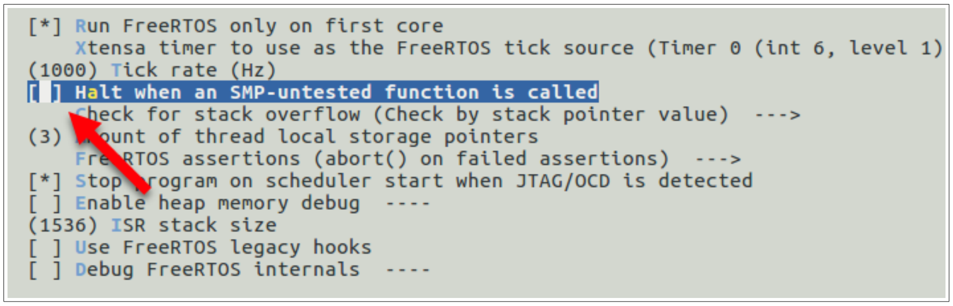
Working with queue sets

What if we have multiple source of events such as multiple queues and semaphores? We could poll each one to see if it has data but this isn't efficient. FreeRTOS adds the concept of a "queue set". This is a set of items against which we wish to know when one of them becomes "ready". We use this by creating a queue set by calling `xQueueCreateSet().` Next we add one or more watchable items using the `xQueueAddToSet()` API as many times as we wish. Now, when ready, we can call `xQueueSelectFromSet()` to ask the set if any of the previously added items are ready to be read from without blocking. Effectively, we can block on all of these items simultaneously and the first one to have data ready for us will be acknowledged.

- xQueueCreateSet
- xQueueAddToSet
- xQueueRemoveFromSet

Running untested functions

Some of the FreeRTOS functions have not been tested as fully as necessary to be claimed "supported" by Espressif. What that means is that you should be cautious about using them in your applications. If using them causes failures, Espressif can't assure that a fix will be readily available. By the same token however, there are no known exposed functions that are known *not* to work. Again, this does **not** mean that all functions work, only that we have the absence of information about them. If you do use a function that is not flagged as supported, a run-time assertion will be deliberately thrown to cause you to consider not using that function. If you decide that you do wish to step off the path and use these un-tested functions, you can flag your desire to do so by changing a configuration flag set by "`make menuconfig`" within the "Component config → FreeRTOS" settings:



```
[*] Run FreeRTOS only on first core
    Xtensa timer to use as the FreeRTOS tick source (Timer 0 (int 6, level 1)
(1000) Tick rate (Hz)
[ ] Halt when an SMP-untested function is called
    Check for stack overflow (Check by stack pointer value)  --->
(3) Amount of thread local storage pointers
    FreeRTOS assertions (abort() on failed assertions)  --->
[*] Stop program on scheduler start when JTAG/OCD is detected
[ ] Enable heap memory debug  ----
(1536) ISR stack size
[ ] Use FreeRTOS legacy hooks
[ ] Debug FreeRTOS internals  ----
```

The Serial AT command processor

Serial (UART) connections are one of the simplest ways to communicate directly between two physically close devices. With reasonably high speeds, two devices can connect easily. There are a variety of serial/USB connectors also on the market which means that your PC doesn't need to have a dedicated serial port.

The ESP32 natively supports UART connections meaning that you can connect an ESP32 to your PC and have some application on the PC interact with the ESP32.

Espressif, the markers of the ESP32, have produced a very high level application that can be downloaded and flashed to the device. This application is called the AT Command Processor. Its purpose is to send and receive ASCII serial commands over a serial link and process them. What kinds of commands? The answer is networking commands. This means that the ESP32 can "off-load" networking from other devices. While it would be unlikely you would off-load networking from a PC, other devices such as, for example, an Arduino, could benefit from the simple off-loading of networking with the translation of such command to serial mappings.

See also:

- ESP32 AT Instruction Set and Examples

Mongoose OS

Mongoose OS is …

To install on Linux, run

This will result in the installation of the command called "mos".

The mos command has the following options:

- ui
- init
- build

- flash
- flash-read
- console
- ls
- get
- put
- rm
- config-get
- config-set
- call
- aws-iot-setup
- update
- wifi

When we run mos, it wishes to connect to the ESP32 and needs to know which serial port to use. We can supply this with the "`--port <port>`" flags. We can also use the `MOS_PORT` environment variable to supply a default.

Next we must load the Mongoose OS firmware into the ESP32 using:

```
$ mos flash esp32
```

On my system this took only a few minutes and resulted in the following output:

```
$ ./mos --port /dev/ttyUSB1 flash mos-esp32
Fetching https://mongoose-os.com/downloads/mos-esp32.zip...
Loaded mjs_base/esp32 version 1.0 (20170529-125521/???)
Opening /dev/ttyUSB1...
Connecting to ESP32 ROM, attempt 1 of 10...
  Connected
Running flasher @ 460800...
  Flasher is running
Flash size: 4194304, params: 0x0220 (dio,32m,40m)
Deduping...
   16384 @ 0x9000 -> 12288
  816624 @ 0x10000 -> 746992
Writing...
   16384 @ 0x1000
    4096 @ 0x8000
    8192 @ 0x9000
    4096 @ 0xc000
    8192 @ 0xd000
    4096 @ 0x10000
  188416 @ 0x20000
  557056 @ 0x50000
  131072 @ 0x190000
Wrote 921600 bytes in 20.32 seconds (354.38 KBit/sec)
Verifying...
   16320 @ 0x1000
    3072 @ 0x8000
   16384 @ 0x9000
    8192 @ 0xd000
  816624 @ 0x10000
  131072 @ 0x190000
Booting firmware...
All done!
```

See also:

- Mongoose OS home page
- Mongoose OS Documentation
- Github: cesanta/mongoose-os

The Mongoose OS file system

A file system is present within the Mongoose OS. We can access this from the command line using:

- `mos ls` – List files.
- `mos get <file>` – Retrieve the named file.
- `mos put <file>` – Store the named file.

Setting up Mongoose OS WiFi

Mongoose OS can use the ESP32 WiFi but needs to know the access point SSID and password. We can set this from the command line using:

```
$ ./mos wifi <SSID> <PASSWORD>
```

Building a Mongoose OS App

Create a directory and run:

```
$ mos init --arch esp32
```

This will populate the current directory with:

- ./mos.yml
- ./fs
- ./src/main.c
- ./src/conf_schema.yaml

AWS IoT

- Device gateway
- Message broker
- Rule engine
- Security and Identity service
- Thing registry
- Thing shadow
- Thing Shadows service

AWS IoT can connect an ESP32 to the AWS IoT servers using MQTT, HTTP or WebSockets protocols. An SDK is available for devices in C, JavaScript, and Arduino.

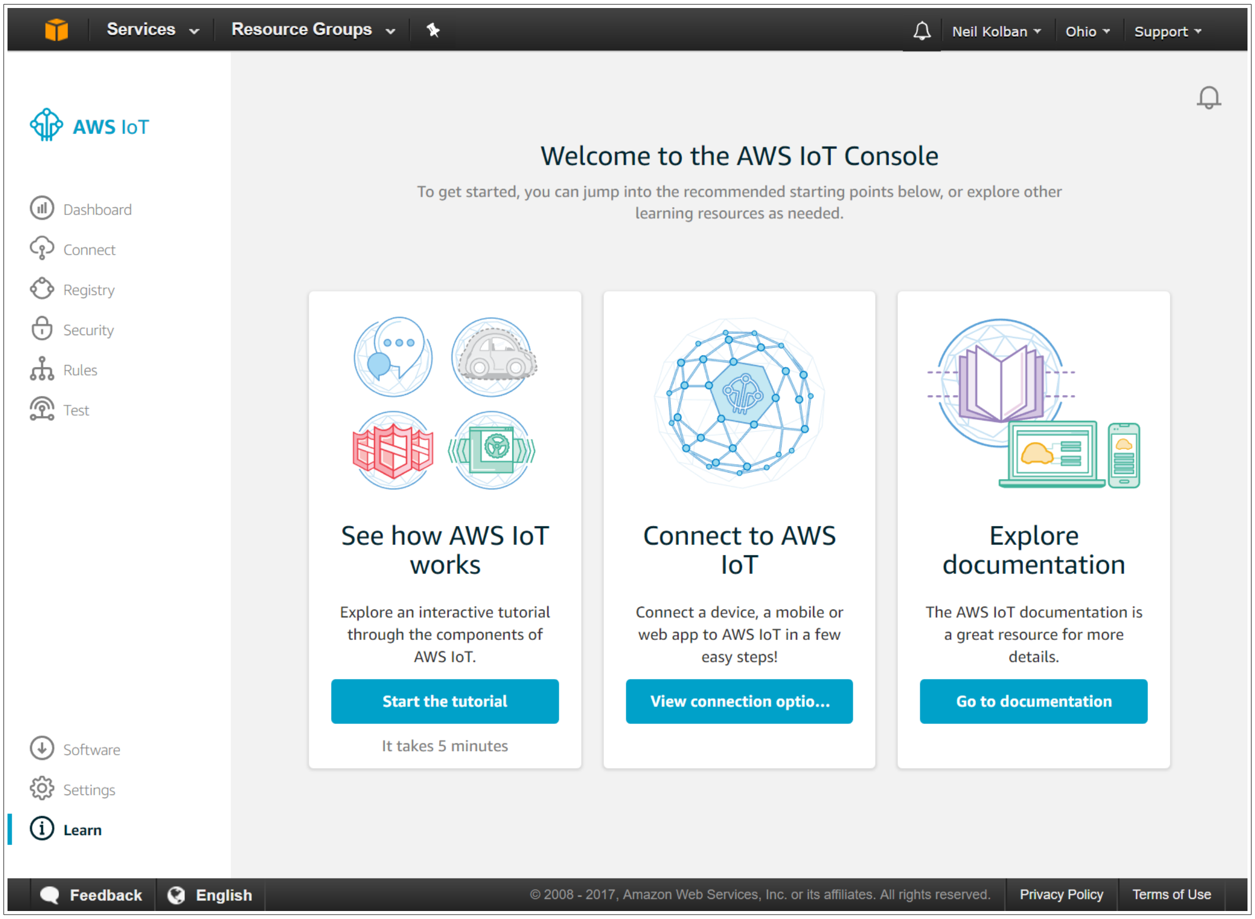
The target of an ESP32 request the AWS Device Gateway which is located on the cloud and managed by Amazon.

Each device that can connect to the AWS Device Gateway has a record of it stored in the AWS IoT Registry. This registry can store meta data about the device.

A device may not be powered up or it may not be networked connected when a message is to be sent to it. Rather than have to worry about detecting that the message was not delivered, AWS IoT provides the concept of a "device shadow". Think of this as the model of the device as known by the AWS Device Gateway. Your application can request a message to be sent to the device and it is the shadow that remembers this request. When the device does power on and connects to the network, the shadow's state is pushed to the device such that we "eventually" synchronize the desired states.

A component called the Rule Engine can quickly examine data contained in incoming messages sent from the device and route/process the request based on the content.

To work with AWS IoT, we work through the AWS IoT Console:



Welcome to the AWS IoT Console

To get started, you can jump into the recommended starting points below, or explore other learning resources as needed.

See how AWS IoT works

Explore an interactive tutorial through the components of AWS IoT.

It takes 5 minutes

Connect to AWS IoT

Connect a device, a mobile or web app to AWS IoT in a few easy steps!

Explore documentation

The AWS IoT documentation is a great resource for more details.

© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.

In order for a device to interact with the AWS IoT servers in the cloud, the devices must communicate over a network. To make such communication easier, Amazon has provided a set of SDKs that can be used. These include:

- C-SDK – C programming
- JS-SDK – JavaScript programming
- Java – Java programming
- Python – Python programming
- Mobile SDK – Android and iOS programming

In order for a device to communicate with AWS IoT in the cloud, there must be mutual authentication. This is performed using TLS 1.2. This means that the device will have:

- A private key
- A public certificate

The important concept here is that there is mutual proof of identity. AWS IoT knows the device and trusts that it is who it claims to be and the device knows that it is actually talking to AWS IoT.

Each device has an identity called an "Amazon Resource Name" which is abbreviated to an "ARN". For example:

```
arn:aws:iot:us-east-2:521670688398:thing/ESP32-1
```

See also:

- AWS IoT – Developers Guide
- Github: aws/aws-iot-device-sdk-embedded-C
- Video: Getting started with AWS IoT

The ESP-IDF aws_iot component

The ESP-IDF provides a component that is a pre-cooked mechanism to interact with AWS IoT.

See also:

- AWS-IoT
- ESP-IDF AWS IoT Example

Developing solutions on Linux

When working in a Linux environments, there are certain tips and techniques which might be useful/valuable.

- When connecting to an ESP32 board using a USB→UART connector, the device may show up under `/dev` as `ttyUSB0`. If we examine the permissions upon this file, we may find that it is configured as:

  ```
  crw-rw---- root dialout
  ```

  This means that it is accessible by root and users in the `dialout` group. If you wish to flash the ESP32 through this device, your userid should thus be a member of this group. To add your user to the group, the following Linux command may be used:

  ```
  sudo usermod -a -G dialout <yourUserid>
  ```

  after making the change, you must log out and log back in again.
- A useful terminal client is `GtkTerm`. This tool provides a terminal viewer that can be used to monitor the USB→UART connector to view log and debug messages. It creates a configuration file in `$HOME/.gtktermrc` that can be edited to change the default serial port (eg. `/dev/ttyUSB0`) as well as changing the baud rate to your desired value.
- Another good terminal client is `screen`. Screen is a full screen terminal emulator.

- Yet another terminal client is the classic `cu` command. Again, very easy to use. An example of use would be:

```
$ cu --line /dev/ttyUSB0 --speed 115200
```

To quit a cu session, enter "~.".

Building a Linux environment

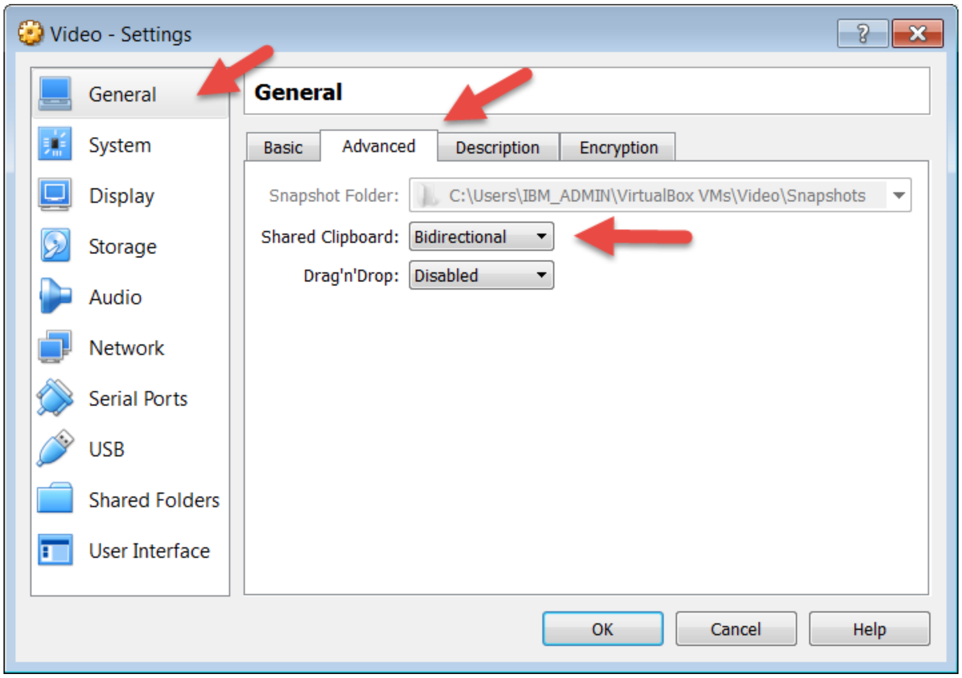
If you don't run Linux natively you may wish to consider running Oracle VirtualBox to host a Linux environment on your Windows or Mac machine. Oracle VirtualBox is an Open Source implementation of an operating system virtualization product. One can download VirtualBox from here:

https://www.virtualbox.org/

In my tests, I ran Ubuntu 15.10.

I define a disk size of at least 20GBytes and 2GBytes of RAM. If you have multiple cores, you may want to define those as being available.

After building an image, make sure that you enable the ability to copy and paste between the host OS and the guest OS.



Snapshot Folder: C:\Users\IBM_ADMIN\VirtualBox VMs\Video\Snapshots

Shared Clipboard: Bidirectional

Drag'n'Drop: Disabled

Also make sure that the VirtualBox guest tools are installed.

There are some packages that you really can't do without including:

- git

You can install new packages with:

```
sudo apt-get install <package>
```

Once you have a Linux OS installed, next we want to build a compilation environment.

I would also install:

- Chrome

See also:

- Oracle Virtual Box
- Ubuntu downloads

Hardware architecture

The majority of this book describes programming the device at the software level through driver libraries. However, let us take a step back. The ESP32 is a low-level hardware component. It is basically a low level System On a Chip (SOC) device.

The CPU and cores

There are two cores within the ESP32 CPU named "PRO" and "APP".

Intrinsic data types

When we program in C, we are programming pretty close to the metal of the environment. Data type sizes and other considerations can and do come into consideration. Let us now examine the data sizes of some of the core types:

Type	Size (bytes)	Size (bits)
short	2	16
int	4	32
long	4	32
long long	8	64
int64_t	8	64
float	4	32
double	8	64

Native byte order, endian and network byte order

When we wish to represent a number, for example 9876 (in decimal) we find that it will not fit in a single byte since a byte can hold only 8 bits of data and hence has a value from 0-255. As such, we need to split it across multiple bytes. If we think of two bytes, that is 16 bits of information and hence can represent a number from 0-65535. When we think of a 16 bit number we can think of that as two 8 bit numbers such that the result is `a * 256 + b`. For example 9876 is `38 * 256 + 148` … and in our equation `a=38` and `b=148`. No great mystery yet. Now we can see that a 16 bit number can be represented by two distinct 8 bit numbers. Now imagine a piece of memory where we wish to store our 16 bit number. We can immediately see that we will store our 16 bit number as two consecutive bytes. One byte representing "`a`" and one representing "`b`". The question becomes should we store "`a`" then "`b`" or "`b`" then "`a`"? From an underlying technical perspective both are equally valid … however we need to be consistent across our architecture. Storing "`a`" then "`b`" is called a "big endian" architecture while storing "`b`" then "`a`" is called a "little endian" architecture"

…	a	b	…

or

…	b	a	…

Here is a simple program fragment to test for ourselves what is in play in the ESP32 environment:

```
char *p = malloc(2);
*(unsigned short *)p = (unsigned short)9876;
printf("low: %d high %d\n", *(p), *(p+1));
```

And when we run, the result is:

```
low: 148 high 38
```

Showing that the byte ordering for storing integers is "little endian". This concept becomes exceptionally important when reading binary data that was written on a different system. If we do not handle this correctly, we can interpret data in an incorrect fashion. For example `38*256+148` is a very different number from `38+148*256`.

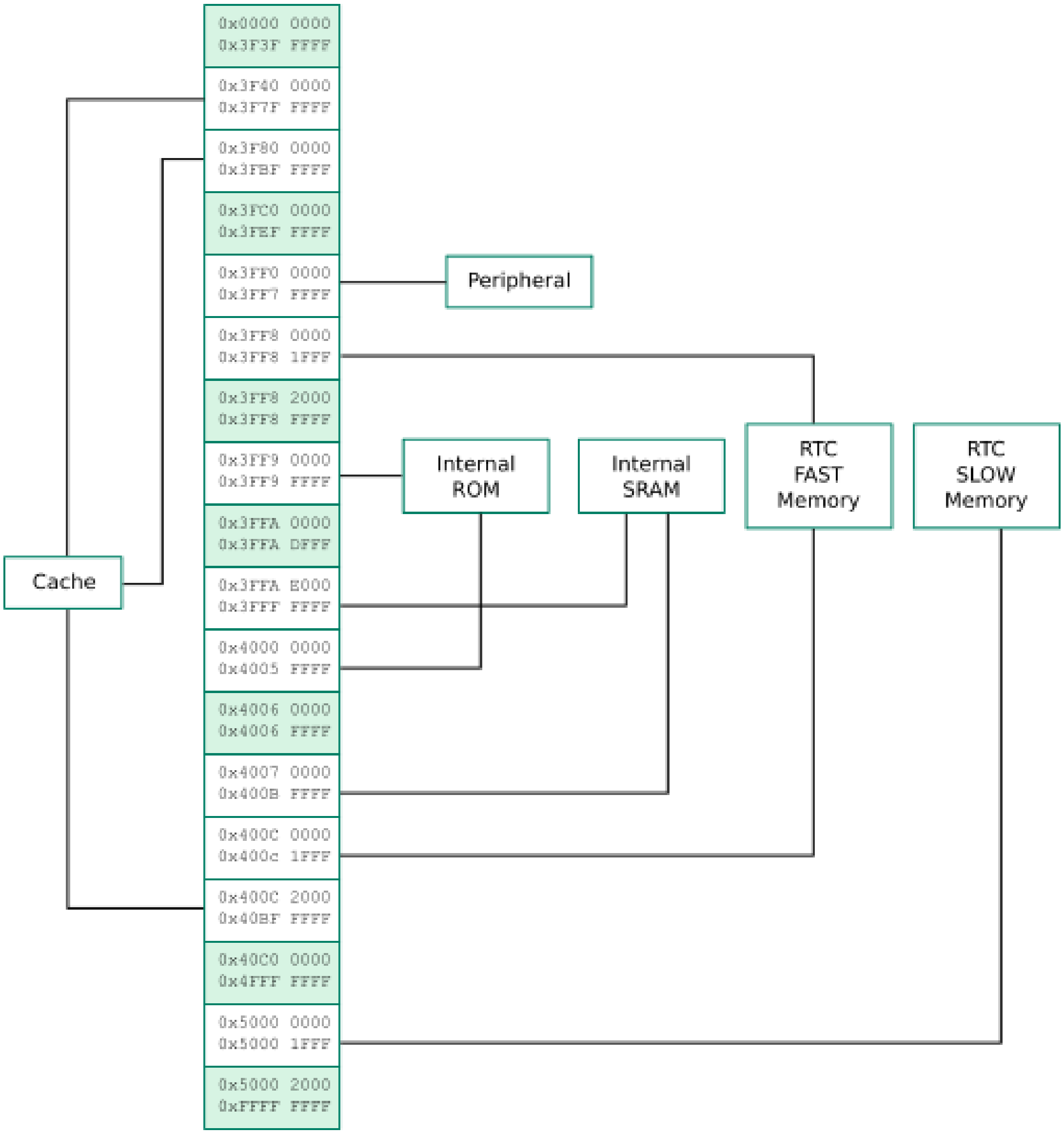
Note that network data transport uses big endian format which means that the data format for native data on an ESP32 is **not** the same format as network byte order which means that we **must** transform data using `htons()`, `htonl()`, `ntohs()` and `ntohl()`.

A more precise terminology for byte ordering uses the phrases "Most Significant Byte" and "Least Significant Byte". With big endian encoding, the most significant byte is first while with little endian, the least significant byte is first.

Memory mapping and address spaces

When we program to the APIs exposed by Espressif provided libraries, we are taking advantage of the higher level APIs and internals that they expose. But let us now contemplate how those libraries work and what we can do with that knowledge.

As a SOC, there is a processor (well, actually two of them), RAM and an address space. Being a 32 bit processor, the address space of the ESP32 is 32 bits in length. This means that, in theory, an application can read or write a byte at memory location `0x0000 0000` or `0xffff ffff` or any address in between. At the hardware mapping level, some of these addresses are mapped to RAM, some to an internal ROM, some to peripherals and some to flash memory. By consulting the ESP32 Technical Reference manual, we can start to learn more.



Within the ESP32 there are four segments of internal memory:

- ROM – 448KB – Read only memory with the content provided by Espressif.
- SRAM – 520KB – Internal RAM that is supplied inside the ESP32.
- RTC FAST – 8KB
- RTC SLOW – 8KB

There is also external memory. This is memory that can be addressed but that is provided by components outside of the ESP32 IC itself such as SPI flash or SPI ram.

The architecture of an ESP32 is called the "Harvard Architecture" and we need to explain that a bit more. When we think of a classic architecture, we usually think of the existence of a memory bus that reads and writes from RAM based storage. Both instructions and data can be found in the memory. The CPU retrieves the next instruction by requesting the data at a specific RAM address. When the instruction is retrieved, the CPU executes the instruction which may read or write data at other addresses in RAM. When the instruction completes, the story repeats for the next instruction.

The Harvard Architecture introduces a second memory bus. The thinking is that RAM will either be used to hold instructions to be executed or will hold data to be manipulated. By separating RAM into these two categories, a number of interesting things start to happen. Instructions are read from the instruction bus and data is read and written from the data bus. The actual RAM at the ends of these buses is now categorized by function. We have IRAM for RAM that holds instructions to be executed and we have DRAM for RAM that holds data.

We can catch a badly behaving application should it try and write into the instruction area of RAM. Since this bus is considered (for the most part) read only, the hardware can detect and trap invalid access.

A second benefit of having two buses is that during the execution of an instruction, the next instruction can be started to be read without impacting access to any data needing to be accessed by the current instruction as that access will be happening over the database. Effectively this allows us to perform certain operations in parallel and allows us to perceive an increase in performance.

Reading and writing registers

We can the value of a peripheral register using the macro `READ_PERI_REG(`*`name`*`)`. This is defined in `<soc/soc.h>`.

- REG_WRITE

- REG_READ
- REG_GET_BIT
- REG_SET_BIT
- REG_CLR_BIT
- REG_SET_BITS
- REG_GET_FIELD
- REG_SET_FIELD
- VALUE_GET_FIELD
- VALUE_GET_FIELD2
- VALUE_SET_FIELD
- VALUE_SET_FIELD2
- FIELD_TO_VALUE
- FIELD_TO_VALUE2
- READ_PERI_REG
- WRITE_PERI_REG
- CLEAR_PERI_REG_MASK
- SET_PERI_REG_MASK
- GET_PERI_REG_MASK
- GET_PERI_REG_BITS
- SET_PERI_REG_BITS
- GET_PERI_REG_BITS2

Pads and multiplexing

A pad is an externalized input or output on the ESP32 device. The ESP32 has a total of 40 pads but only a subset of them are exposed. Within the ESP32 there is a logical component called the "`IO_MUX`". This is an "Input/Output Multiplexer". In English imagine this as a logical "switch board" with some fixed number of actual telephones attached to it. When caller wishes to communicate with a telephone, it has to pass through the `IO_MUX` which decides which physical telephone to pass the call to. This means that there is a separation between logical "calls" and physical "phones".

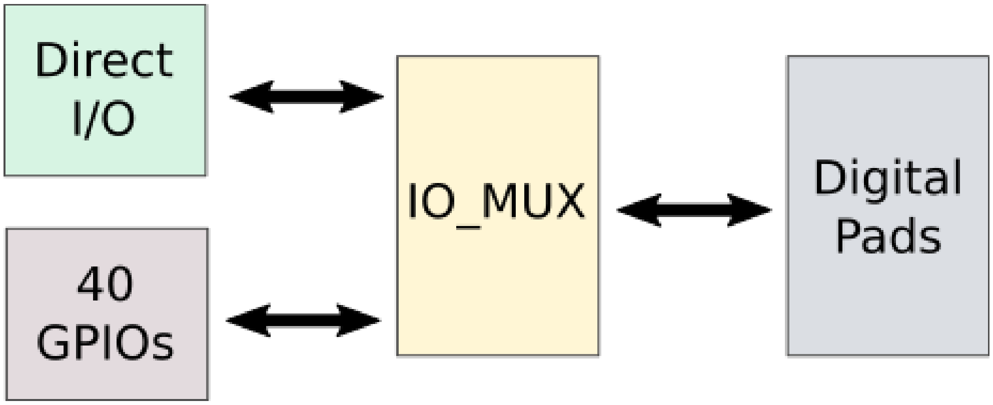
- Pads 0-39 can be input
- Pads 0-33 can be output

GPIO	Pad Name	Function 1	Function 2	Function 3	Function 4	Function 5	Function 6
0	GPIO0	GPIO0	CLK_OUT1	GPIO0			
1	U0TXD	U0TXD	CLK_OUT3	GPIO1			
2	GPIO2	GPIO2	HSPIWP	GPIO2	HS2_DATA0	SD_DDATA0	
3	U0RXD	U0RXD	CLK_OUT2	GPIO3			
4	GPIO4	GPIO4	HSPIHD	GPIO4	HS2_DATA1	SD_DATA1	
5	GPIO5	GPIO5	VSPICS0	GPIO5	HS1_DATA6		
6	SD_CLK	SD_CLK	SPICLK	GPIO6	HS1_CLK	U1CTS	
7	SD_DATA_0	SD_DATA_0	SPIQ	GPIO7	HS1_DATA0	U2RTS	
8	SD_DATA_1	SD_DATA_1	SPID	GPIO8	HS1_DATA1	U2CTS	
9	SD_DATA_2	SD_DATA_2	SPIHD	GPIO9	HS1_DATA2	U1RXD	
10	SD_DATA_3	SD_DATA_3	SPIWP	GPIO10	HS1_DATA3	U1TXD	
11	SD_CMD	SD_CMD	SPICS0	GPIO11	HS1_CMD	U1RTS	
12	MTDI	MTDI	HSPIQ	GPIO12	HS2_DATA2	SD_DATA2	
13	MTCK	MTCK	HSPID	GPIO13	HS2_DATA3	SD_DATA3	
14	MTMS	MTMS	HSPICLK	GPIO14	HS2_CLK	SD_CLK	
15	MTDO	MTDO	HSPICS0	GPIO15	HS2_CMD	SD_CMD	
16	GPIO16	GPIO16		GPIO16	HS1_DATA4	U2RXD	
17	GPIO17	GPIO17		GPIO17	HS1_DATA5	U2TXD	
18	GPIO18	GPIO18	VSPICLK	GPIO18	HS1_DATA7		
19	GPIO19	GPIO19	VSPIQ	GPIO19	U0CTS		
20	GPIO20	GPIO20		GPIO20			
21	GPIO21	GPIO21	VSPIHD	GPIO21			
22	GPIO22	GPIO22	VSPIHD	GPIO22	U0RTS		
23	GPIO23	GPIO23	VSPID	GPIO23	HS1_STROBE		
24							
25	GPIO25	GPIO25		GPIO25			
26	GPIO26	GPIO26		GPIO26			
27	GPIO27	GPIO27		GPIO27			
28							
29							
30							
31							
32	32K_XP	GPIO32		GPIO32			
33	32K_XN	GPIO33		GPIO33			
34	VDET_1	GPIO34		GPIO34			
35	VDET_2	GPIO35		GPIO35			

36	SENSOR_VP	GPIO36		GPIO36			
37	SENSOR_CAPP	GPIO37		GPIO37			
38	SENSOR_CAPN	GPIO38		GPIO38			
39	SENSOR_VN	GPIO39		GPIO39			

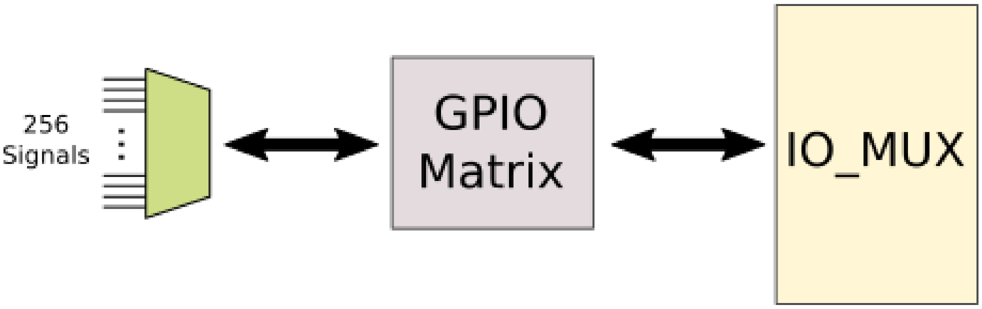
Now let us think of the "things" that wish to read and write from these pad pins. Some of them need to be really fast while others can afford to be slower. For the fast ones, we say that these need direct I/O. Examples of fast components include Ethernet, SDIO, SPI, JTAG and UART. The other ones we say are GPIO based and can be used by slower functions. Examples of these include I2C, I2S, PWM, LEDC and others.

Schematically, this might look as follows:



What this means is that input/output from the digital pads is governed directly by the `IO_MUX` controller. Each of the 40 digital pads can be thought of as having a control register that says "For a given pad, is this pad connected to GPIO or is it connected to a Direct I/O device and, if a direct I/O device … which one?".

Now let us add another twist. There is the concept of a "signal". Think of a signal as being either output from an internal component or input from an internal component. We can map a signal to a GPIO for input or output. This effectively allows us to matrix map anything to anything. In the following diagram, we show this component.



To say that a signal will appear on a particular GPIO, we execute the following logic:

```
PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[gpioNum], PIN_FUNC_GPIO);
gpio_set_direction(gpioNum, GPIO_MODE_OUTPUT);
gpio_matrix_out(gpioNum, signal, 0, 0);
```

What we do is say that a particular pad (PIN) is going to be used for GPIO which means that its data will come from a signal as opposed to direct I/O. Then we say that a particular signal is mapped to a GPIO.

See also:

- gpio_matrix_out
- gpio_set_direction

Register based GPIO

There are 34 usable GPIO pins on the ESP32. We should also remember that there are 40 pads on an ESP32. Some of the GPIOs are in the 0-31 pad range and some in the 32-39 pad range. Since the ESP32 is a 32bit processor, this means that a single register can only hold 32 bits of information. As such, we have pairs of registers for working with GPIO functions. The first in the pair corresponds to GPIOs on pads 0-31 and the second on pads 32-39.

The registers called GPIO_ENABLE_REG (0-31) and GPIO_ENABLE1_REG (32-39) control whether or not a GPIO is input or output. If the corresponding bit is high then the GPIO is output while if it is low then the GPIO is input.

When working with GPIO, there are registers that we can use as follows:

- Setting a bit in a register causes the output value of the corresponding GPIO to be high
- Setting a bit in a register causes the output value of the corresponding GPIO to be low

An individual GPIO pin can be defined as either input or output. Input means we can read the signal present on that pin. Output means that we can set the signal that is present on the pin. A GPIO can be *either* input or output at a given time but, obviously, can not be both.

A very useful data structure is defined in `soc/gpio_struct.h`. The type of the structure is called `gpio_dev_t` and it is exposed via the global GPIO. It contains:

	bt_select	
GPIO_OUT_REG	out	GPIO 0-31 output
GPIO_OUT_W1TS_REG	out_w1ts	GPIO 0-31 set
GPIO_OUT_W1TC_REG	out_w1tc	GPIO 0-31 clear
GPIO_OUT1_REG	out1	GPIO 32-39 output
GPIO_OUT1_W1TS_REG	out1_w1ts	GPIO 32-39 set
GPIO_OUT_W1TC_REG	out1_w1tc	GPIO 32-39 clear
	sdio_select	
GPIO_ENABLE_REG	enable	GPIO 0-31 output enable
GPIO_ENABLE_W1TS_REG	enable_w1ts	GPIO 0-31 output set
GPIO_ENABLE_W1TC_REG	enable_w1tc	GPIO 0-31 output clear
GPIO_ENABLE1_REG	enable1	GPIO 32-39 output enable
GPIO_ENABLE1_W1TS_REG	enable1_w1ts	GPIO 32-39 output set
GPIO_ENABLE1_W1TC_REG	enable1_w1tc	GPIO 32-39 output clear
GPIO_STRAP_REG	strap	Boot strapping input data
GPIO_IN_REG	in	GPIO 0-31 input
GPIO_IN1_REG	in1	GPIO 32-39 input
GPIO_STATUS_REG	status	GPIO 0-31 interrupt status
GPIO_STATUS_W1TS_REG	status_w1ts	GPIO 0-31 interrupt status set
GPIO_STATUS_W1TC_REG	status_w1tc	GPIO 0-31 interrupt status cleat
GPIO_STATUS1_REG	status1	GPIO 32-39 interrupt status
GPIO_STATUS1_W1TS_REG	status1_w1ts	GPIO 32-39 interrupt status set
GPIO_STATUS1_W1TC_REG	status1_w1tc	GPIO 32-39 interrupt status clear
	acpu_int	GPIO 0-31 APP CPU interrupt status
	acpu_nmi_int	GPIO 0-31 APP CPU non-maskable interrupt status
	pcpu_int	GPIO 0-31 PRO CPU interrupt status
	pcpu_nmi_int	GPIO 0-31 PRO CPU non-maskable interrupt status
	cpusdio_int	
	acpu_int1	
	acpu_nmi_int1	
	pcpu_int1	
	pcpu_nmi_int1	
	cpusdio_int1	
	pin[]	
	cali_conf	
	cali_data	

GPIO_FUNCm_IN_SEL_CFG_REG	func_in_sel_cfg[]	
GPIO_FUNCn_OUT_SEL_CFG_REG	func_out_sel_cfg	

IO_MUX_x_REG

x is 0 to 39.

IO_x_MCU_SEL	Select the IO_MUX function.
IO_x_FUNC_DRV	Drive strength of the pad
IO_x_FUNC_IE	Input Enable of the pad.
IO_x_FUNC_WPU	Internal pull-up.
IO_x_FUNC_WPD	Internal pull-down.
IO_x_MCU_DRV	Drive strength in sleep mode.
IO_x_MCU_IE	Input enable during sleep mode.
IO_x_MCU_WPU	Internal pull-up during sleep mode.
IO_x_MCU_WPD	Internal pull-down during sleep mode.
IO_x_SLP_SEL	Sleep mode selection.
IO_x_MCU_OE	Output enable in sleep mode.

IO_x_MCU_SEL

GPIO_OUT_REG

C Struct: GPIO.out

The values of the output signal on GPIO pins 0-31.

GPIO_OUT_W1TS_REG

C Struct: GPIO.out_w1ts

W1TS is Write One To Set. Setting the corresponding bit to 1, cause the GPIO output to be 1.

GPIO_OUT_W1TC_REG

C Struct: GPIO.out_w1tc

W1TC is Write One To Clear. Setting the corresponding bit to 1, cause the GPIO output to be 0.

GPIO_OUT1_REG

C Struct: GPIO.out1.data, GPIO.out1.val

The values of the output signal on GPIO pins 32-39.

GPIO_OUT1_W1TS_REG

C Struct: GPIO.out1_w1ts.data, GPIO.out1_w1ts.val

W1TS is Write One To Set. Setting the corresponding bit to 1, cause the GPIO output to be 1.

GPIO_OUT1_W1TC_REG

C Struct: GPIO.out1_w1tc.data, GPIO.out1_w1tc.val

W1TC is Write One To Clear. Setting the corresponding bit to 1, cause the GPIO output to be 0.

GPIO_ENABLE_REG

C Struct: GPIO.enable

The enable register is oddly named. It defines whether or not a pin will be output or input. If the corresponding bit is 1, the pin will be output while if the corresponding bit is 0, the pin will be input. This register covers pins 0-31.

GPIO_ENABLE_W1TS_REG

C Struct: GPIO.enable_w1ts

GPIO_ENABLE_W1TC_REG

C Struct: GPIO.enable_w1tc

GPIO_ENABLE1_REG

C Struct: GPIO.enable1.data, GPIO.enable1.val

The enable register is oddly named. It defines whether or not a pin will be output or input. If the corresponding bit is 1, the pin will be output while if the corresponding bit is 0, the pin will be input. This register covers pins 32-39.

GPIO_ENABLE1_W1TS_REG

C Struct: GPIO.enable_w1ts.data, GPIO.enable_w1ts.val

GPIO_ENABLE1_W1TC_REG

C Struct: GPIO.enable_w1tc.data, GPIO.enable_w1tc.val

GPIO_STRAP_REG

C Struct: GPIO.strap.strapping, GPIO.strap.val,

The values of strapping pins at boot:

MTDI, GPIO0, GPIO2, GPIO4, MTDO, GPIO5

GPIO_IN_REG

C Struct: GPIO.in

The values of the input signal on GPIO pins 0-31. This is a read only register. It makes no sense to try and change the value of the register that reflects the current value found on the pins.

GPIO_IN1_REG

C Struct: GPIO.in1.data, GPIO.in1.val

The values of the input signal on GPIO pins 32-39. This is a read only register. It makes no sense to try and change the value of the register that reflects the current value found on the pins.

GPIO_STATUS_REG

C Struct: GPIO.status

GPIO_STATUS_W1TS_REG

C Struct: GPIO.status_w1ts

GPIO_STATUS_W1TC_REG

C Struct: GPIO.status_w1tc

GPIO_STATUS1_REG

C Struct: GPIO.status1.intr_st, GPIO.status1.val

GPIO_STATUS1_W1TS_REG

C Struct: GPIO.status1_w1ts.intr_st, GPIO.status1_w1ts.val

GPIO_STATUS1_W1TC_REG

C Struct: GPIO.status1_w1tc.intr_st, GPIO.status1_w1tc.val

GPIO_PCPU_NMI_INT1_REG

GPIO_PCPU_NMI_INT1_REG

GPIO_PINn_REG

GPIO_PINn_INT_ENA	
GPIO_PINn_WAKEUP_ENABLE	
GPIO_PINn_INT_TYPE	
GPIO_PINn_PAD_DRIVER	

GPIO_FUNC*m*_IN_SEL_CFG_REG

C Struct: GPIO.func_in_sel_cfg[m]

m is 0 to 255.

There are 256 different possible functions. There is a register for each of the functions. The register defines which of the 40 GPIOs is mapped as input to that function signal.

The field `GPIO_FUNCm_IN_SEL` defines the GPIO pad for the corresponding function. This will be 0-39 for the pads with special values of `0x38` for always high and `0x30` for always low.

If we want to invert the incoming signal, we can set the `GPIO_FUNCm_IN_INV_SEL` field to 1.

If we wish to bypass the GPIO matrix, set the `GPIO_SIGm_IN_SEL` to 1.

Field	Bits	Struct field
GPIO_SIGm_IN_SEL	[7]	GPIO.func_in_sel_cfg[n].sig_in_sel
GPIO_FUNCm_IN_INV_SEL	[6]	GPIO.func_in_sel_cfg[n].sig_in_inv
GPIO_FUNCm_IN_SEL	[5:0]	GPIO.func_in_sel_cfg[n].func_sel

GPIO_FUNCn_OUT_SEL_CFG_REG

C Struct: GPIO.func_out_sel_cfg[n]

n is 0 to 39.

If `GPIO_FUNCn_OEN_INV_SEL` is set to 1, then the output signal is inverted.

Field	Bits	Struct field
GPIO_FUNCn_OEN_INV_SEL	[11]	GPIO.func_out_sel_cfg[n].oen_inv_sel
GPIO_FUNCn_OEN_SEL	[10]	GPIO.func_out_sel_cfg[n].oen_sel
GPIO_FUNCn_OUT_INV_SEL	[9]	GPIO.func_out_sel_cfg[n].inv_sel
GPIO_FUNCn_OUT_SEL	[8:0]	GPIO.func_out_sel_cfg[n].func_sel

See also:

- GPIOs

Strapping pins

When an ESP32 is powered on, we may wish to control/configure how it boots. To achieve this, a set of pins are defined that are read for the very short time that is considered the bootstrap time. For the sake of discussion, we should image this as the instantaneous time at which the device is powered up.

The values of these pins are "remembered" and available in the ESP32 register called `GPIO_STRAPPING`. The boot-loader will examine these values and act accordingly. We can also read this register later to see what values were in effect at boot time.

The pins have prescribed meanings in the ESP32 architecture:

Boot mode source

We can boot from flash memory for normal operation or we can boot to read a new program into flash memory for loading an application.

- `GPIO0` = 1 – Boot from flash.
- `GPIO0` = 0 – Boot for loading a new program.

The floating state is 1 and hence will boot from flash.

Debugging on U0TX0 at boot

When the ESP32 boot, it can write debugging information out through UART0 and its TX pin. This is useful during development but if we want to use this UART in our solutions, this diagnostic information will be treated as data over the UART and may interfere or confuse any devices attached. We can disable diagnostics from the UART by setting a strapping pin value:

- `MTDO` = 1 – Use UART0 for debugging (default).
- `MTDO` = 0 – Do not use UART0 for debugging.

The floating state is 1 and hence will use UART0 for debugging.

Timing of SDIO slave

A couple of pins are used to configure the SDIO slave interfacing.

Boot-loader

The booting mechanism of the ESP32 examines the settings of the "strapping" pins and based on their configuration we got into SPI boot mode or UART download mode or some other mode that isn't specified. In normal boot mode we examine address `0x1000` for the image of a software boot-loader. The standard software boot-loader reads the partition table to find the "app" image to give control to.

If we look at an example run of the `esptool.py` tool, we find that it loads the following:

```
bootloader.bin 0x1000
app.bin        0x10000
partitions.bin 0x4000
```

The in-built ROM in the ESP32 looks in flash address space at `0x1000` and expects to find a mapping table. This table lists a set of "segments". Each segment consists of:

- Offset from start of flash area (`0x1000`) of where the segment starts.
- Length in bytes of the segment.
- Location in ESP32 processor address space where the segment should be copied.

For example, if we run `esptool.py --chip image_info bootloader.bin` we will find:

```
esptool.py v2.0-dev
Image version: 1
Entry point: 40098200
```

```
4 segments

Segment 1: len 0x00000 load 0x3ffc0000 file_offs 0x00000018
Segment 2: len 0x00a08 load 0x3ffc0000 file_offs 0x00000020
Segment 3: len 0x01068 load 0x40078000 file_offs 0x00000a30
Segment 4: len 0x00378 load 0x40098000 file_offs 0x00001aa0
Checksum: 92 (valid)
```

We can also run `xtensa-esp32-elf-objdump -h bootloader.elf`:

```
Idx Name               Size      VMA       LMA       File off  Algn
  0 .iram1.text        00000378  40098000  40098000  00001b24  2**2
  1 .dram0.bss         00000000  3ffc0000  3ffc0000  00000abc  2**0
  2 .dram0.data        00000000  3ffc0000  3ffc0000  000000b4  2**0
  3 .dram0.rodata      00000a08  3ffc0000  3ffc0000  000000b4  2**2
  4 .iram_pool_1.text 00001065  40078000  40078000  00000abc  2**2
  5 .debug_frame       00000230  00000000  00000000  00001e9c  2**2
  6 .debug_info        00002a37  00000000  00000000  000020cc  2**0
  7 .debug_abbrev      000008fc  00000000  00000000  00004b03  2**0
  8 .debug_loc         00000cac  00000000  00000000  000053ff  2**0
  9 .debug_aranges     00000118  00000000  00000000  000060ab  2**0
 10 .debug_ranges      00000110  00000000  00000000  000061c3  2**0
 11 .debug_line        000016a7  00000000  00000000  000062d3  2**0
 12 .debug_str         00000d36  00000000  00000000  0000797a  2**0
 13 .comment           00000022  00000000  00000000  000086b0  2**0
 14 .xtensa.info       00000038  00000000  00000000  000086d2  2**0
```

As a further example, here is an `image_info` listing of an application

```
esptool.py v2.0-dev
Image version: 1
Entry point: 40080868
8 segments

Segment 1: len 0x0ffe8 load 0x00000000 file_offs 0x00000018
Segment 2: len 0x049b0 load 0x3f400010 file_offs 0x00010008 → .flash.rodata
Segment 3: len 0x01aa8 load 0x3ffbf2c0 file_offs 0x000149c0 → .dram0.data
Segment 4: len 0x00400 load 0x40080000 file_offs 0x00016470 → .iram0.vectors
Segment 5: len 0x163b0 load 0x40080400 file_offs 0x00016878 → .iram0.text
Segment 6: len 0x00000 load 0x400c0000 file_offs 0x0002cc30 → .rtc.test
Segment 7: len 0x033d0 load 0x00000000 file_offs 0x0002cc38
Segment 8: len 0x2965c load 0x400d0018 file_offs 0x00030010 → .flash.text
Checksum: c5 (valid)
```

and an edited version of the objdump:

```
Idx Name            Size      VMA       LMA       File off  Algn
  0 .iram0.vectors 00000400  40080000  40080000  00006538  2**2
  1 .iram0.text    000163ae  40080400  40080400  00006938  2**2
  2 .dram0.bss     0000f2c0  3ffb0000  3ffb0000  00004a90  2**3
  3 .dram0.data    00001aa8  3ffbf2c0  3ffbf2c0  00004a90  2**4
  4 .flash.rodata  000049b0  3f400010  3f400010  000000e0  2**4
  5 .flash.text    00029659  400d0018  400d0018  0001cce8  2**2
  6 .rtc.text      00000000  400c0000  400c0000  00046341  2**0
  7 .debug_frame   00005590  00000000  00000000  00046344  2**2
  8 .debug_info    00070fbb  00000000  00000000  0004b8d4  2**0
  9 .debug_abbrev  000101a2  00000000  00000000  000bc88f  2**0
 10 .debug_loc     000231ce  00000000  00000000  000cca31  2**0
 11 .debug_aranges 000024c0  00000000  00000000  000efc00  2**3
 12 .debug_ranges  00002a20  00000000  00000000  000f20c0  2**3
 13 .debug_line    0002d918  00000000  00000000  000f4ae0  2**0
 14 .debug_str     00012101  00000000  00000000  001223f8  2**0
 15 .comment       00000062  00000000  00000000  001344f9  2**0
 16 .xtensa.info   00000038  00000000  00000000  0013455b  2**0
```

Power modes

The ESP32 can operate in different power modes. The distinguishing feature is how much current is consumed. Ideally we want the ESP32 to consume as little power as possible and the trade off on this is the amount of function that can be performed against current consumption. The items that affect current consumption includes

- Radio on or off
- CPUs on or off
- RAM being maintained

The distinct modes of operation have names associated with them:

- Active mode – This is the normal operational mode. The radio is on and it can both transmit and receive.
- Modem-sleep mode – In this mode the radio is powered off but other functions remain enabled. Radio is the biggest consumer of power.
- Light-sleep mode – The CPUs are paused however the RTC and ULP-co-processor remain operational. A variety of events can wake up the processor including timers or external interrupts.
- Deep-sleep mode – Only the RTC has power.
- Hibernation mode – Only one timer or an external RTC interrupt can wake us up.

Bootloader

When a device is powered up, control is given in a raw state to an area of code called the "bootloader". The responsibility of a bootloader is to bring the system up to a usable state and then load and give control to a user written application. The bootloader for

ESP32 is supplied as the bootloader component within the ESP-IDF environment. Studying the bootloader provides us great incite into how the ESP32 operates.

Control starts at `call_start_cpu0`

```
call_start_cpu0 {
   reset CPU caches;
   call bootloader_main()
}
```

The `bootloader_main()` is responsible for much of the work:

```
bootloader_main() {
   // Load the bootloader header data from SPI address 0x1000
   call esp_image_load_header();

   // Load the partition table.
   call load_patition_table();
}
```

Peripherals

Remote Control Peripheral – RMT

The primary purpose of this component of the ESP32 is to generate pulses sent via an infrared LED. When you point your TV remote control at your TV and press a button, an infrared LED at the front sends a sequence of pulses that are received by the TV and decoded. Since an infrared LED can be either "on" or "off" at any given time, the signal is encoded in the duration that the LED is either on or off. Whether the LED is on or off and for how long we call a data item. To use the RMT component, we build an array of data items and pass those to the RMT with the intent that it will then own the generation of the correct signals.

A data item is composed of a 16 bit value.

level [15]	period [14:0]

The level bit is either 1 or 0 describing the output signal while the period (15 bits in length) is the duration in clock ticks. 15 bits give us a range from 1-32767. A value of 0 for the period is used as an end marker. The normal maximum number of data item records is 128.

The RMT has 8 distinct channels with each channel having 128 16 bit records. Should we wish to send more than 128 records, we can extend a channel to use the channel data of subsequent adjacent channels. For example, if we are using only channel 0 then the data available for it can be that of channel 0, channel 1 up to channel 7 giving us a total of 1024 value records (1024 data items or 2048 bytes). The number of channels of data used by a specific channel is set in the register `RMT_MEM_OWNER_CH`*n*.

There is also an interesting mechanism that involves the buffer for a channel wrapping around. Imagine that we have a default buffer size of 128 * 16bit records. If we need to send more than 128 records, we can pre-load a buffer with our first 128 values and set the RMT transmitting. When it has transmitted the first 128 values, it will generate an interrupt that can be used to re-fill the buffer with the new values and progress will continue.

The RAM for these blocks starts at RMT base address + 0x800.

Assuming each channel uses just its own block, the channels start at:

Channel	Address
0	RMT base address + 0x800
1	RMT base address + 0x800 + 0x100
2	RMT base address + 0x800 + 0x200
3	RMT base address + 0x800 + 0x300
4	RMT base address + 0x800 + 0x400
5	RMT base address + 0x800 + 0x500
6	RMT base address + 0x800 + 0x600
7	RMT base address + 0x800 + 0x700

Each channel has two control registers associated with it called `RMT_CHnCONF0_REG` and `RMT_CHnCONF1_REG`.

Register – RMT_CHnCONF0_REG

Name	Bits	Field	Description
`reserved`	`[31]`	`clk_en`	Reserved. Set to 0.
`RMT_MEM_PD`	`[30]`	`mem_pd`	Power down memory. • **0** – memory is powered up • 1 – memory is powered down
`RMT_CARRIER_OUT_LV_CHn`	`[29]`	`carrier_out_lv`	Carrier present on: • 0 – transmit on high • **1** – transmit on low
`RMT_CARRIER_EN_CHn`	`[28]`	`carrier_en`	Is carrier signal enabled? • 0 – disabled • **1** – enabled
`RMT_MEM_SIZE_CHn`	`[27:24]`	`mem_size`	Memory blocks allocated to channel. 4bits. (**0x01**)
`RMT_IDLE_THRESH_CHn`	`[23:8]`	`idle_thres`	Clock cycles of no change indicates an end of reception in receive mode. (**0x1000**)
`RMT_DIV_CNT_CHn`	`[7:0]`	`div_cnt`	Divider for channel clock. (**0x02**)

Register – RMT_CHnCONF1_REG

Name	**Bits**	**Field**	**Description**
reserved	[31:20]	reserved20	Reserved. Set to 0.
RMT_IDLE_OUT_EN_CHn	[19]	idle_out_en	Output enable. (**0**)
RMT_IDLE_OUT_LV_CHn	[18]	idle_out_lv	Output signal in idle state. (**0**)
RMT_REF_ALWAYS_ON_CHn	[17]	ref_always_on	Selection of channel base clock: • **0** – clk_ref. • 1 – clk_apb.
RMT_REF_CNF_RST_CHn	[16]	ref_cnf_rst	Reset the clock divider. (**0**)
RMT_RX_FILTER_THRES_CHn	[15:8]	rx_filter_thres	Threshold for of pulse width in receive mode. (**0x0f**)
RMT_RX_FILTER_EN_CHn	[7]	rx_filter_en	Enable receive filter. (**0**)
RMT_TX_CONTI_MODE_CHn	[6]	tx_conti_mode	Enable repeating transmission. • **0** – Go idle at end of transmission. • 1 – Repeat at end of transmission.
RMT_MEM_OWNER_CHn	[5]	mem_owner	Block used for receive or transmit? • 0 – RAM used for transmission. • **1** – RAM used for reception.
reserved	[4]	apb_mem_rst	Reserved. Set to 0.
RMT_MEM_RD_RST_CHn	[3]	mem_rd_rst	Reset read ram for transmitter access. (**0**)
RMT_MEM_WR_RST_CHn	[2]	mem_wr_rst	Reset write ram for receiver access. (**0**)
RMT_RX_EN_CHn	[1]	rx_en	Enable receiving on channel. 1 – Start receiving. (**0**)
RMT_TX_START_CHn	[0]	tx_start	Start sending on channel. 1 – Start transmitting. (**0**)

Let us imagine we want to send 8 bits of square wave through channel 0. This means we will want 17 records (one for each value transition and a terminator). We will then want to set `RMT_CH0CONF0_REG` to

- RMT_MEM_PD = 0 (Power up memory)
- RMT_CARRIER_EN_CH0 = 0 (carrier disabled)
- RMT_MEM_SIZE_CH0 = 1 (1 memory block)
- RMT_DIV_CNT_CH0 = 1 (divider for clock)

We will also want to set RMT_CH0CONF1_REG to:

- RMT_IDLE_OUT_EN_CH0 = 1 (Output enable)
- RMT_IDLE_OUT_LV_CH0 = 0 (0 in idle state)
- RMT_REF_ALWAYS_ON_CH0 = 0 (channel base clock)
- RMT_MEM_OWNER_CH0 = 0 (used for transmission)
- RMT_MEM_RD_RST_CH0 = 1 (Reset read ram)
- RMT_RX_EN_CH0 = 0 (Enable transmit on channel)
- RMT_TX_START_CH0 = 1 (Start transmitting)

Within the header file called "`soc/rmt_struct.h`" we have a global variable defined called `RMTMEM` that contains definitions of the data buffers.

There is also a variable called RMT that contains the

The signal mapping for GPIOs is:

Function	Signal	Constant
rmt_sig_out0	87	RMT_SIG_OUT0_IDX
rmt_sig_out1	88	RMT_SIG_OUT1_IDX
rmt_sig_out2	89	RMT_SIG_OUT2_IDX
rmt_sig_out3	90	RMT_SIG_OUT3_IDX
rmt_sig_out4	91	RMT_SIG_OUT4_IDX
rmt_sig_out5	92	RMT_SIG_OUT5_IDX
rmt_sig_out6	93	RMT_SIG_OUT6_IDX
rmt_sig_out7	94	RMT_SIG_OUT7_IDX

To say that a signal will appear on a particular GPIO, we execute the following logic:

```
include "soc/gpio_sig_map.h"

int gpioNum = 17;
PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[gpioNum], PIN_FUNC_GPIO);
gpio_set_direction(gpioNum, GPIO_MODE_OUTPUT);
gpio_matrix_out(gpioNum, RMT_SIG_OUT0_IDX, 0, 0);
```

SPI

The header file called "`soc/spi_struct.h`" provides a data type called "`spi_dev_t`" that is a memory map into the registers that control SPI. There are four instances of this structure, one for each of the hardware SPIs available on an ESP32. These are called `SPI0`, `SPI1`, `SPI2` and `SPI3`. `SPI0` is reserved for internal use and should be ignored.

If one looks there, you will find a bewilderingly large number of settings. Poking those values correctly to achieve SPI is not a trivial matter and documentation on those features is still a while away.

See also:

- Error: Reference source not found
- The Arduino Hardware Abstraction Layer SPI

PID Controller

I have no idea what this is.

UART

Read the UART status register?

READ_PERI_REG(UART_INT_ST_REG(??))

UART_FRM_ERR_INT_ENA ---?

UART_RXFIFO_TOUT_INT_ENA – Timeout interrupt enable

UART_RXFIFO_FULL_INT_ENA – RX FIFO length is > than threshold

UART_GET_RXFIFO_CNT → #define to

```
#define UART_GET_RXFIFO_CNT(i)
GET_PERI_REG_BITS2(UART_STATUS_REG(i) , UART_RXFIFO_CNT_V,
UART_RXFIFO_CNT_S)
```

Number of bytes in RX queue

UART_GET_RXFIFO_RD_BYTE → #define to

```
#define UART_GET_RXFIFO_RD_BYTE(i)
GET_PERI_REG_BITS2(UART_FIFO_REG(i) , UART_RXFIFO_RD_BYTE_V,
UART_RXFIFO_RD_BYTE_S)
```

Read a byte from the RX queue

I2S

The I2S peripheral is more than just working with the I2S protocol. It is also a powerful data mover of parallel data into and out of RAM without application involvement.

First, let us realize that there are **two** I2S buses called I2S0 and I2S1.

The I2S bus can output a high precision clock.

A very special *mode* of the I2S bus is called the "LCD mode". This deviates dramatically from any I2S audio protocols but gives us access to driving an external parallel display (LCD) or receiving parallel data from a camera.

See also:

- AMBA 3 AHB-Lite Protocol

I2S Clock

The `I2S_CLK` signal is an output signal used to communicate with the partner. The clock is derived from either the `PLL_D2_CLK` or the `APLL_CLK` as defined by the `I2S_CLKA_ENA` bit of `I2S_CLCKM_CONFIG_REG`. The default is `PLL_D2_CLK`.

The frequency of the clock is given by the following formula:

$$Fi2s = \frac{Fpll}{N + \frac{b}{a}}$$

For the `PLL_D2_CLK`, Fpll will be 160MHz.

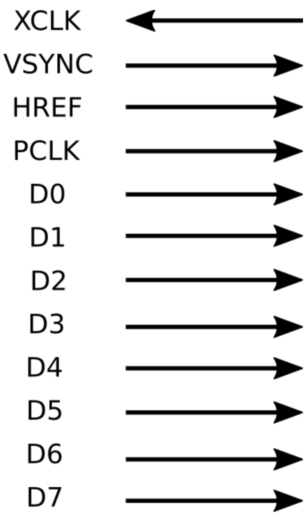
- `N` is the value `REG_CLKM_DIV_NUM` of `I2S_CLKM_CONF_REG`
- `b` is `I2S_CLKM_DIV_A` of `I2S_CLCKM_CONF_REG`
- `a` is `I2S_CLKM_DIV_B` of `I2S_CLCKM_CONF_REG`

Camera mode

The I2S peripheral has a mode called "camera" mode. In that mode, it is able to retrieve data from a camera. To do this, we need some logical pins:

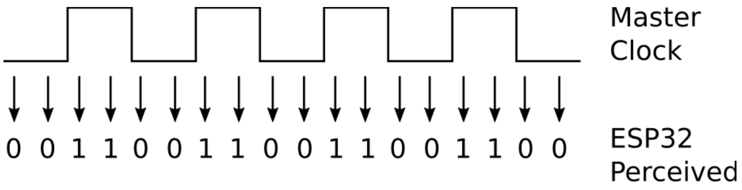
Logical Name	ESP32 Identity	WROOM Dev Kit
Data 0	I2S0I_DATA_IN0_IDX	GPIO4
Data 1	I2S0I_DATA_IN1_IDX	GPIO5
Data 2	I2S0I_DATA_IN2_IDX	GPIO18
Data 3	I2S0I_DATA_IN3_IDX	GPIO19
Data 4	I2S0I_DATA_IN4_IDX	GPIO36
Data 5	I2S0I_DATA_IN5_IDX	GPIO39
Data 6	I2S0I_DATA_IN6_IDX	GPIO34
Data 7	I2S0I_DATA_IN7_IDX	GPIO35
VSYNC	I2S0I_V_SYNC_IDX	GPIO25
HREF	I2S0_H_ENABLE_IDX	GPIO23
PCLK	I2S0_WS_IN_IDX	GPIO22

Consider a logical camera input. It would consist of:



The way to read this is that the camera has input from an external clock (`XCLK`) and generates 8 parallel lines of data output (the camera data bus) plus three additional interrupt signals for vertical sync, horizontal sync and pixel clock.

So far, so good. Now let us consider the ESP32 I2S peripheral as consuming this data … and it is here that things get interesting. While one might imagine that the I2S peripheral should behave as a master (providing the clock) that isn't how it is designed to be used. Instead the I2S peripheral should be placed into slave mode which means that it is the consumer of a clock signal. That seems OK as we have a `PCLK` signal to use. However, that's not how the story works. The ESP32 (acting as a slave) does **not** synchronize clocks with the master (the camera). Instead, the ESP32 runs its own internal clock which must be at least twice as fast as the master (camera) clock. (Is this true?) For technical reasons the ESP32 I2S peripheral has a maximum value of 40MHz which means that the master (camera) clock can never be greater than 20MHz. With this in mind, we now learn that the ESP32 isn't actually "interrupted" by the master (camera's) clock, instead the ESP32 is sampling the incoming clock signal. As long as the sampling is fast enough, the ESP32 can detect a clock transition by seeing that the signal has gone from a previous high to a current reading of low or a previous low to a current reading of high.



This design is known in digital electronics as a "synchronizer stage". Thankfully, we do not need to understand all the mechanical details but what we **do** have to get right is the setting of the ESP32 I2S clock. The I2S clock is known as the `I2S_CLK` signal. We do

not need to expose this on an IO pin since it need not be connected to anything. However, we **do** need to set its speed. We do this through three settings in the I2S_CLKM_CONF_REG as follows:

Set REG_CLKM_DIV_NUM to be 2

Set REG_CLKM_DIV_A to 1

Set REG_CLKM_DIV_B to 0

From the clock speed calculation formula for I2S which is:

$$Fi2s = \frac{Fpll}{N + \frac{b}{a}}$$

This gives us a clock speed value of:

$$Fi2s = \frac{Fpll}{2 + \frac{0}{1}}$$

which is:

$$Fi2s = \frac{Fpll}{2}$$

Since Fpll is the PLL_D2_CLK which runs at 160MHz, this then gives us the final result of 80MHz as the sampling rate of the external clock.

In actual code, we would write:

```
// Configure clock divider
I2S0.clkm_conf.clkm_div_a   = 1;
I2S0.clkm_conf.clkm_div_b   = 0;
I2S0.clkm_conf.clkm_div_num = 2;
```

WAIT … SOMETHING IS NOW WRONG WITH THE STORY

Reading the Tech Ref, there are some "rules" to put the I2S into camera mode. Specifically:

Field	Register	Value
I2S_CAMERA_EN	I2SCONF2_REG	1
I2S_LCD_EN	I2SCONF2_REG	1
I2S_RX_SLAVE_MOD	I2SCONF_REG	1
I2S_RX_MSB_RIGHT	IS2CONF_REG	0
I2S_RX_RIGHT_FIRST	IS2CONF_REG	0
I2S_RX_CHAN_MOD	I2SCONF_CHAN_REG	1
I2S_RX_FIFO_MOD	I2S_FIFO_CONF_REG	1

Studying the sample made available from Ivan at Espressif, we find that it performs the following logical functions:

Toggles the reset flags in I2S_LC_CONF_REG register:

I2S_IN_RST – Reset the RX DMA finite state machine.

I2S_AHBM_RST – Reset the AHB interface of DMA.

I2S_AHBM_FIFO_RST – Reset the AHB interface cmdFIFO of DMA.

Next it toggles further reset flags:

I2S_RX_RESET

I2S_RX_FIFO_RESET

I2S_TX_RESET

I2S_TX_FIFO_RESET

We then block until some state changes.

Next set:

I2S_RX_SLAVE_MOD=1 rx_slave_mode = 1

I2S_LCD_EN = 1

I2S_CAMERA_EN = 1

The Clock

I2S_DSCR_EN = 1 (Enable I2S DMA mode)

I2S_RX_FIFO_MOD_FORCE_EN = 1

I2S_RX_CHAN_MOD = 1

I2S_RX_BITS_MOD = 16

Registers

The registers are:

Name	Description	Access
I2S_CONF_REG	Configuration and start/stop bits	R/W
I2S_CONF1_REG	PCM configuration	R/W
I2S_CONF2_REG	ADC/LCD/Camera configuration	R/W
I2S_TIMING_REG	Signal delay and timing	R/W
I2S_FIFO_CONF_REG	FIFO configuration	R/W
I2S_CONF_SINGLE_DATA_REG	Static channel output value	R/W
I2S_CONF_CHAN_REG	Channel configuration	R/W
I2S_LC_HUNG_CONF_REG	Timeout detection configuration	R/W
I2S_CLKM_CONF_REG	Bitclock configuration	R/W
I2S_SAMPLE_RATE_CONF_REG	Sample rate configuration	R/W
I2S_PD_CONF_REG	Power down register	R/W
I2S_LC_CONF_REG	DMA configuration	R/W
I2S_RXEOF_NUM_REG	Receive data count	R/W
I2S_OUT_LINK_REG	DMA transmit linked list configuration address	R/W
I2S_IN_LINK_REG	DMA receive linked list configuration and address	R/W
I2S_OUT_EOF_DES_ADDR_REG	Address of transmit link descriptor producing EOF	RO
I2S_IN_EOF_DES_ADDR_REG	Address of receive link descriptor producing EOF	RO
I2S_OUT_EOF_BFR_DES_ADDR_REG	Address of transmit buffer producing EOF	RO
I2S_INLINK_DSCR_REG	Address of current inlink descriptor	RO
I2S_INLINK_DSCR_BF0_REG	Address of next inlink descriptor	RO
I2S_INLINK_DSCR_BF1_REG	Address of next inlink data buffer	RO
I2S_OUTLINK_DSCR_REG	Address of current outlink descriptor	RO
I2S_OUTLINK_DSCR_BF0_REG	Address of next outlink descriptor	RO
I2S_OUTLINK_DSCR_BF1_REG	Address of next outlink data buffer	RO
I2S_LC_STATE0_REG	DMA receive status	RO
I2S_LC_STATE1_REG	DMA transmit status	RO
I2S_PDM_CONF_REG	PDM configuration	R/W
I2S_PDM_FREQ_CONF_REG	PDM frequencies	R/W
I2S_INT_RAW_REG	Raw interrupt status	RO
I2S_INT_ST_REG	Masked interrupt status	RO
I2S_INT_ENA_REG	Interrupt enable bits	R/W
I2S_INT_CLR_REG	Interrupt clear bits	RO

See also:

- soc/esp32/include/soc/i2s_reg.h
- soc/esp32/include/soc/i2s_struct.h

I2S_CONF_REG

This field is `conf` in the struct.

I2S_SIG_LOOPBACK	
I2S_RX_MSB_RIGHT	
I2S_TX_MSB_RIGHT	
I2S_RX_MONO	
I2S_TX_MONO	
I2S_RX_SHORT_SYNC	
I2S_TX_SHORT_SYNC	
I2S_RX_MSB_SHIFT	
I2S_TX_MSB_SHIFT	
I2S_RX_RIGHT_FIRST	Set this bit receiver right channel data first.
I2S_TX_RIGHT_FIRST	Set this bit to transmit right channel data first.
I2S_RX_SLAVE_MOD	Set this bit to enable slave receiver mode.
I2S_TX_SLAVE_MOD	Set this bit to enable slave transmitter mode.
I2S_RX_START	Set this bit to start receiving.
I2S_TX_START	Set this bit to start transmitting.
I2S_RX_FIFO_RESET	Set this bit to reset the receive FIFO.
I2S_TX_FIFO_RESET	Set this bit to reset the transmit FIFO/
I2S_RX_RESET	Set this bit to reset the receiver.
I2S_TX_RESET	Set this bit to reset the transmitter.

I2S_CONF2_REG

This field is `conf2` in the struct.

I2S_INTER_VALID_EN	Enable internal validation.
I2S_EXT_ADC_START_EN	Enable external ADC.
I2S_LCD_EN	Enable LCD mode.
I2S_LCD_TX_SDX2_EN	Duplicate data pairs.
I2S_LCD_TX_WRX2_EN	Write data in duplicate in LCD mode.
I2S_CAMERA_EN	Enable camera mode.

Looking at the struct there also appears to be "`data_enable`" and "`data_enable_test_en`".

I2S_CLKM_CONF_REG

This is field `clkm_conf`.

The high level function which sets these values is `i2s_set_clk()`.

I2S_CLKA_ENA [21]	• 0 – Use PLL D2 • 1 – Use clk_apll
I2S_CLKM_DIV_A [19:14]	Divider denominator
I2S_CLKM_DIV_B [13:8]	Divider numerator
I2S_CLKM_DIV_NUM[7:0]	Divider integral value

I2S_CONF_CHAN_REG

This field is `conf_chan` in the struct.

I2S_RX_CHAN_MOD [4:3]	
I2S_TX_CHAN_MOD [2:0]	

I2S_LC_CONF_REG

This field is `lc_conf` in the struct.

I2S_MEM_TX_EN	
I2S_CHECK_OWNER	
I2S_OUT_DATA_BURST_EN	
I2S_INDSCR_BURST_EN	
I2S_OUTDSCR_BURST_EN	DMA outlink descriptor transfer mode configuration.
I2S_OUT_EOF_MODE	Interrupt config.
I2S_OUT_AUTO_WRBACK	Set to 1 to enable automatic outlink-writeback when all the data in tx buffer has been transmitted.
I2S_OUT_LOOP_TEST	Set to loop test outlink.
I2S_IN_LOOP_TEST	Set to loop test inlink.
I2S_AHBM_RST	Toggle to 1 to reset the AHB interface of DMA.
I2S_AHBM_FIFO_RST	Toggle to 1 to reset the AHB interface cmdFIFO of DMA.
I2S_OUT_RST	Toggle to 1 to reset the outgoing DMA state machine.
I2S_IN_RST	Toggle to 1 to reset the incoming DMA state machine.

I2S_FIFO_CONF_REG

This field is `fifo_conf` in the struct.

I2S_RX_FIFO_MOD_FORCE_EN	Should always be 1.
I2S_TX_FIFO_MOD_FORCE_EN	Should always be 1.
I2S_RX_FIFO_MOD [19:16]	Receive FIFO mode config. • 0 – 16 bit dual channel data • 1 – 16 bit single channel data • 2 – 32 bit dual channel data • 3 – 32 bit single channel data
I2S_TX_FIFO_MOD [15:13]	Transmit FIFO mode config. • 0 – 16 bit dual channel data • 1 – 16 bit single channel data • 2 – 32 bit dual channel data • 3 – 32 bit single channel data
I2S_DSCR_EN	Set this to enable I2S DMA mode.
I2S_TX_DATA_NUM [11:6]	Threshold of data length in the transmit FIFO.
I2S_RX_DATA_NUM [5:0]	Threshold of data length in the receive FIFO.

I2S_IN_LINK_REG

This field is `in_link` in the struct.

I2S_INLINK_RESTART	
I2S_INLINK_START	
I2S_INLINK_STOP	
I2S_INLINK_ADDR	

I2S_RXEOF_NUM_REG

This field is `rx_eof_num` in the struct.

This defined the length of the data to be received. After receiving this data, an `I2S_IN_SUC_EOF_INT` interrupt will be triggered. This will indicate that all the data expected to have been received has been received.

I2S_CONF_CHAN_REG

This field is `conf_chan` in the struct.

I2S_RX_CHAN_MOD [4:3]	I2S receiver channel mode.
I2S_TX_CHAN_MOD [2:0]	I2S transmitter channel mode.

I2S_SAMPLE_RATE_CONF_REG

This field is `sample_rate_conf` in the struct.

I2S_RX_BITS_MOD [23:18]	Bit length of I2S receiver channel.
I2S_TX_BITS_MOD [17:12]	Bit length of I2S transmitter channel.
I2S_RX_BCK_DIV_NUM [11:6]	Bit clock configuration in receiver.
I2S_TX_BCK_DIV_NUM [5:0]	Bit clock configuration in transmitter.

I2S_INT_RAW_REG

This field is `int_raw` in the struct.

I2S_OUT_TOTAL_EOF_INT_RAW	
I2S_IN_DSCR_EMPTY_INT_RAW	
I2S_OUT_DSCR_ERR_INT_RAW	
I2S_IN_DSCR_ERR_INT_RAW	
I2S_OUT_EOF_INT_RAW	
I2S_OUT_DONE_INT_RAW	
I2S_IN_SUC_EOF_INT_RAW	
I2S_IN_DONE_INT_RAW	
I2S_TX_HUNG_INT_RAW	
I2S_RX_HUNG_INT_RAW	
I2S_TX_REMPTY_INT_RAW	
I2S_TX_WFULL_INT_RAW	
I2S_RX_REMPTY_INT_RAW	
I2S_RX_WFULL_INT_RAW	
I2S_TX_PUT_DATA_INT_RAW	
I2S_RX_TAKE_DATA_INT_RAW	

I2S_INT_ENA_REG

This field is int_ena in the struct.

I2S_OUT_TOTAL_EOF_INT_RAW	
I2S_IN_DSCR_EMPTY_INT_RAW	
I2S_OUT_DSCR_ERR_INT_RAW	
I2S_IN_DSCR_ERR_INT_RAW	
I2S_OUT_EOF_INT_RAW	
I2S_OUT_DONE_INT_RAW	
I2S_IN_SUC_EOF_INT_RAW	
I2S_IN_DONE_INT_RAW	
I2S_TX_HUNG_INT_RAW	
I2S_RX_HUNG_INT_RAW	
I2S_TX_REMPTY_INT_RAW	
I2S_TX_WFULL_INT_RAW	
I2S_RX_REMPTY_INT_RAW	
I2S_RX_WFULL_INT_RAW	
I2S_TX_PUT_DATA_INT_RAW	
I2S_RX_TAKE_DATA_INT_RAW	

I2S_INT_CLR_REG

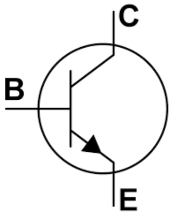
This field is `int_clr` in the struct.

I2S_OUT_TOTAL_EOF_INT_ENA	
I2S_IN_DSCR_EMPTY_INT_ENA	
I2S_OUT_DSCR_ERR_INT_ENA	
I2S_IN_DSCR_ERR_INT_ENA	
I2S_OUT_EOF_INT_ENA	
I2S_OUT_DONE_INT_ENA	
I2S_IN_SUC_EOF_INT_ENA	
I2S_IN_DONE_INT_ENA	
I2S_TX_HUNG_INT_ENA	
I2S_RX_HUNG_INT_ENA	
I2S_TX_REMPTY_INT_ENA	
I2S_TX_WFULL_INT_ENA	
I2S_RX_REMPTY_INT_ENA	
I2S_RX_WFULL_INT_ENA	
I2S_TX_PUT_DATA_INT_ENA	
I2S_RX_TAKE_DATA_INT_ENA	

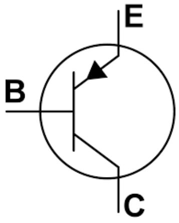
Electronics

Transistors as switches

If we consider that a GPIO pin can produce a signal of high or low we see that the signal can be used to perform work. However there is a maximum current that can be drawn from any given I/O pin. If, for example, we wished to attach a device to a pin which would draw more current than the pin is rated to supply, we may very easily damage our ESP32. This is where a transistor can come into play. A transistor can be thought of as an electronic switch. A transistor has three pins labeled emitter, base and collector. If a small current flows between base and emitter a correspondingly higher current will be allowed to flow between collector and emitter. There are two types of transistor we will come across called NPN and PNP. The difference between them is whether or not the "switch" is triggered by a high signal or a low signal. The following is the schematic symbol for an NPN transistor. Normally we connect the emitter to ground and a load between the collector and +ve. If we then connect the base to the output of an I/O pin (through a resistor(, then when the pin goes high, a small current will flow between B and E and a much higher current will flow between C and E.



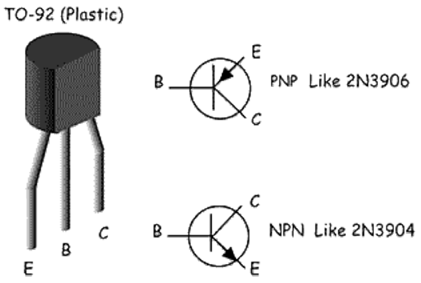
For the other type of transistor (PNP) whose symbol is shown in the following diagram, the emitter is usually connected to +ve and the collector to the load and then ground. With the base pin connected to a GPIO (through a resistor), when the base goes low, the small current flowing from the emitter through the base will be greatly amplified as the current flowing through the emitter to the collector.

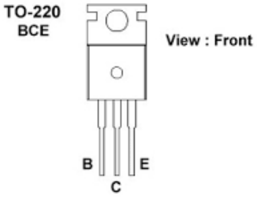


By using a transistor, we are no longer directly drawing load through the GPIO pin but instead simply using the signal present on the pin to energize the transistor.

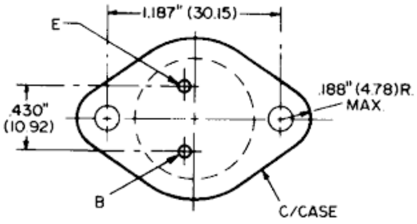
Common transistors that can easily be obtained include:

Name	Type	Style
2N3904 (Signal transistor) – 200mA max current	NPN	TO-92
2N3906 (Signal transistor)	PNP	TO-92
PN2222	NPN	TO-92
TIP120 – 5A max current	NPN	TO-220
TIP125 – 5A max current	PNP	TO-220
BC547 – 100mA max current	NPN	TO-92
BC557 – 100mA max current	PNP	TO-92
2N3055 – 15A max current	NPN	TO-3





TO-3



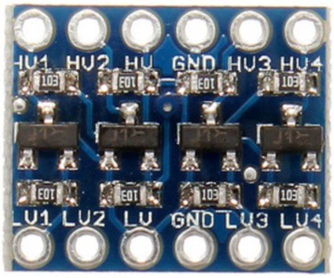
See also:

- 2N3904 Data Sheet
- 2N3906 Data Sheet
- PN2222 Data Sheet
- TIP120 Data Sheet
- TIP125 Data Sheet
- BC547 Data Sheet
- BC557 Data Sheet
- You Tube: Electronic Basics #22: Transistor (BJT) as a switch
- Using bipolar transistors as switches

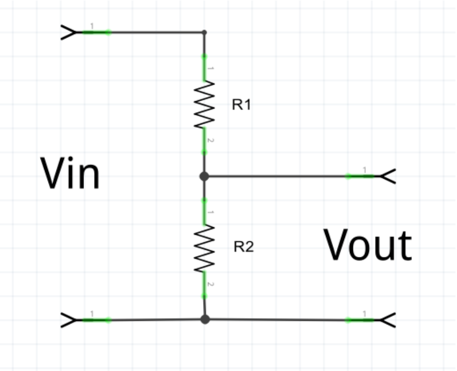
Logic Level Shifting

We have already read that the ESP32 GPIO pins are 3.3V tolerant. This means that the maximum input voltage must be 3.3V. If you go higher than that, you are at risk for frying your device. But what if we have a peripheral device that outputs a 5V signal? Conversely, the maximum output voltage on a Pi GPIO is 3.3V. If we feed that as input to a peripheral that is expecting a 5V input, there is no assurance that it will detect the signal correctly. Fortunately there is a solution in a circuit called a logic level shifter. This circuit can convert a 3.3V signal to a 5V signal and can also convert a 5V signal to a 3.3V signal.

The module takes as input +ve and ground of the high level and +ve and ground of the low level and provides bi-directional switching.



If you don't have access to such logic level circuits, an alternative solution is to create a voltage divider. This will protect 3V Pi inputs from over voltage from a 5V source. This is a simple circuit made with two resistors. A higher voltage is provided to input than output and using a correct combination of resistors, the input voltage is proportionally scaled to the output voltage.



There is an equation available to us:

$$V_{out} = \frac{V_{in} \times R2}{R1 + R2}$$

As an example of a solution to this equation, if V_{in} is 5V and we want V_{out} to be 3.3V and choose 1KΩ for R2 then …

$$3.3 = \frac{1000}{R1 + 1000} \times 5$$

Which evaluates to R1 having a value of 515Ω. We can use a higher value for R1 but not lower.

Expressing the equation in a different order we have:

$$R1=\frac{(V_{in}-V_{out})\times R2}{V_{out}}$$

See also:

- Sparkfun – Bi-Directional Logic Level Converter Hookup Guide
- Wikipedia – Voltage divider

Projects

JerryScript library for ESP32

JerryScript is JavaScript for embedded devices including the ESP32. However, just like JavaScript on other platforms, it provides only the language. What this means is that there are no libraries of higher functions supplied by default. Projects such as IoT.js would be perfect, but it appears that their requirements and dependencies are currently too function rich for a trivial port to ESP32.

As a result of that, we now look to see if we can't do something more pragmatic. We will borrow wherever possible from the semantics and descriptions of Node.js which is the industry de-facto standard.

The "require" capability

The concept is that we can load "modules" by name. A module is the dynamic sourcing of function which returns a JavaScript object that services that function.

Generically, we will code:

```
let myVar = require("<Module Name>");
```

Our implementation will be general enough to return built-in modules as well as external modules.

See also:

- JerryScript

API Reference

Now we have a mini reference to the syntax of many of the ESP32 exposed APIs as well as related libraries. Do not use this reference exclusively. Please also refer to the published Espressif SDK Programming Guide and other sources of literature. Part of the purpose of recreating this quick listing is so that it can be cross referenced in the examples as well as serving as a place where notes about using an API in the context of an ESP32 can be captured.

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 – WiFi Protected Setup
- sntp – Simple Network Time Protocol
- mdns – Multi-cast Domain Name System
- uart – Universal asynchronous receiver/transmitter
- pwm – Pulse width modulation

Configuration, status and operational retrieval

There are certain APIs available to us that can be used to interrogate the operation of an ESP32 and some of its distinct parts and components. The detailed documentation for these APIs are found in the corresponding links … but here we list their existence and brief description of what they provide:

- system_get_time - Get the system time measured in microseconds since last device start-up.
- esp_chip_info - Get information about the chip.
- esp_get_free_heap_size - Get the amount of free heap size.
- esp_get_idf_version - Get the version of the ESP-IDF in use.
- esp_wifi_scan_get_ap_records - Retrieve the access points found in a previous scan.
- esp_wifi_scan_get_ap_num - Retrieve the count of found access points from a previous scan.
- esp_wifi_get_auto_connect - Determine whether or not auto connect at boot is enabled.
- esp_wifi_get_bandwidth - Get the current bandwidth setting.
- esp_wifi_get_channel - Get the current channel.
- esp_wifi_get_config - Retrieve the current connection information associated with the specified WiFi interface.
- esp_wifi_get_country - Retrieve the currently configured WiFi country.
- esp_wifi_get_mac - Retrieve the current MAC address for the interface.
- esp_wifi_get_mode - Get the WiFi operating mode.
- esp_wifi_get_promiscuous
- esp_wifi_get_protocol - Get the 802.11 protocol (b/g/n).
- esp_wifi_get_ps - Get the power save type.
- esp_wifi_get_station_list - Get the list of stations connected to ESP32 when it is behaving as an access point.
- tcpip_adapter_get_ip_info
- tcpip_adapter_get_sta_list
- tcpip_adapter_get_wifi_if

Arduino Mapping

The Arduino APIs are a library of functions available within the Arduino ecosystem. There may be times when we want to port an Arduino library to the ESP-IDF environment. This section lists some of the more common Arduino APIs and how they can be mapped to the ESP-IDF.

bitRead

```
bitRead(x, n)
```

The equivalent is:

```
(x & 1<<n) != 0
```

bitWrite

```
bitWrite(x, n, b)
```

```
x = (x & (~(1<<n)) | (b << n))
```

delay

```
delay(ms)
```

The equivalent in ESP-IDF is:

```
vTaskDelay(ms/portTICK_PERIOD_MS)
```

See also:

- vTaskDelay
- Arduino – delay()

digitalWrite

```
digitalWrite(pin, value)
```

The equivalent in ESP-IDF is:

```
gpio_set_level()
```

Parameter mappings:

HIGH	1
LOW	0

See also:

- gpio_set_level
- Arduino – digitalWrite()

pinMode

```
pinMode(pin, mode)
```

The equivalent in ESP-IDF is:

```
gpio_set_direction()
```

Parameter mappings:

mode=INPUT	GPIO_MODE_INPUT
mode=OUTPUT	GPIO_MODE_OUTPUT
mode=INPUT_PULLUP	GPIO_MODE_INPUT with an additional call to `gpio_set_pull_mode()`.

See also:

- gpio_set_direction
- Arduino – pinMode()

SPI.begin

```
SPI.begin()
```

The equivalent in ESP-IDF is:

```
spi_bus_initialize();
spi_bus_add_device();
```

See also:

- spi_bus_add_device
- spi_bus_initialize
- Arduino – SPI.begin()

SPI.setBitOrder

```
SPI.setBitOrder
```

The equivalent in ESP-IDF is:

```
spi_bus_add_device()
```

and setting the flags properties. The default is MSB but within the flags we can specify LSB for either or both of the RX or TX.

See also:

- spi_bus_add_device
- Arduino – SPI.setBitOrder()

SPI.setClockDivider

```
SPI.setClockDivider()
```

The equivalent in ESP-IDF is:

```
spi_bus_add_device()
```

and setting the clock speed in Hz.

See also:

- spi_bus_add_device
- Arduino – SPI.setClockDivider()

SPI.setDataMode

```
SPI.setDataMode()
```

The equivalent in ESP-IDF is:

```
spi_bus_add_device()
```

and setting the `mode` property of the structure.

See also:

- spi_bus_add_device
- Arduino – SPI.setDataMode()

SPI.transfer

```
SPI.transfer(val)
SPI.transfer(buffer, size)
```

The equivalent in ESP-IDF is:

```
spi_device_transmit()
```

See also:

- spi_device_transmit
- Arduino – SPI.transfer()

Wire.begin

Initialize the I2C interface.

```
Wire.begin()
Wire.begin(address)
```

The equivalent ESP-IDF is

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (address << 1) | I2C_MASTER_WRITE, 1 /* expect ack */);
```

Wire.beginTransmission

```
Wire.beginTransmission(address)
```

The equivalent ESP-IDF is:

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (address << 1) | I2C_MASTER_WRITE, 1 /* expect ack */);
```

Wire.endTransmission

```
Wire.endTransmission()
```

The equivalent ESP-IDF is:

```
i2c_master_stop(cmd);
i2c_master_cmd_begin(0, cmd, 0);
i2c_cmd_link_delete(cmd);
```

Wire.read

```
Wire.read()
```

The equivalent ESP-IDF is:

```
i2c_master_read(cmd, data, length, ack)
i2c_master_read_byte(cmd, data, ack)
```

Wire.requestFrom

```
Wire.requestFrom(address, quantity)
```

Wire.write

```
Wire.write(string)
Wire.write(data, length)
```

The equivalent ESP-IDF is:

```
i2c_master_write(cmd, data, length, 1);
```

FreeRTOS API reference

See also:

- Using FreeRTOS

portENABLE_INTERRUPTS

Enable interrupts and context switching.

portDISABLE_INTERRUPTS

Disable interrupts and context switching.

xPortGetCoreID

This is a macro that returns the current core ID on which we are running.

```
uint32 xPortGetCoreID()
```

Since the ESP32 has two cores, the result will either only ever be 0 or 1.

pvPortMalloc

Allocate storage.

```
void *pvPortMalloc(size_t size)
```

See also:

- malloc

pvPortFree

Release storage.

```
void vPortFree(void *data)
```

xEventGroupClearBits

Clear one or more bits in an event group.

```
EventBits_t xEventGroupClearBits(
   EventGroupHandle_t eventGroup,
   const EventBits_t bitsToClear)
```

- `eventGroup` – The event group that contains the bits to be cleared.
- `bitsToClear` – The set of bits to be cleared within the event group.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/event_groups.h>

See also:

- Blocking and synchronization within FreeRTOS

xEventGroupCreate

Create a new FreeRTOS event group.

```
EventGroupHandle_t xEventGroupCreate()
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/event_groups.h>

See also:

- Blocking and synchronization within FreeRTOS

xEventGroupSetBits

Set one or more bits in an event group.

```
EventBits_t xEventGroupSetBits(
   EventGroupHandle_t eventGroup,
   const EventBits_t bitsToSet)
```

- `eventGroup` – The event group that contains the bits to be set.
- `bitsToSet` – The set of bits to be set within the event group. This is a bit mask.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/event_groups.h>

See also:

- Blocking and synchronization within FreeRTOS

xEventGroupWaitBits

Block waiting for one or more "bits" to become set.

```
EventBits_t xEventGroupWaitBits(
   const EventGroupHandle_t eventGroup,
   const EventBits_t bitsToWaitFor,
   const BaseType_t clearOnExit,
   const BaseType_t waitForAllBits,
   TickType_t ticksToWait)
```

Calling this function can cause the caller to block until bits are set or a timeout is met. Bits can be set through `xEventGroupSetBits()`.

- `eventGroup` – The event group that references the bits to be watched.
- `bitsToWaitFor` – The set of bits within the event group that we are waiting upon. This is a bit mask.
- `clearOnExit` – Should the bits that we are waiting on be cleared automatically when set and this method returns?
- `waitForAllBits` – Should we unblock on the first bit that is set that we are watching for or alternatively should we wait on all the bits being set?
- `ticksToWait` – How many ticks should we wait for before returning? This provides a timeout (if needed). The RTOS variable "`portMAX_DELAY`" can be used to specify that we wish to wait indefinitely.

Includes:

- #include <freertos/FreeRTOS.h>

- #include <freertos/event_groups.h>

See also:

- xEventGroupSetBits
- Blocking and synchronization within FreeRTOS

xQueueAddToSet

Add a queue to the set of queues monitored by this queue set.

```
BaseType_t xQueueAddToSet(
   QueueSetMemberHandle_t xQueueOrSemaphore,
   QueueSetHandle_t xQueueSet)
```

See also:

- xQueueCreateSet
- xQueueRemoveFromSet
- xQueueSelectFromSet

xQueueCreate

Create a queue for holding items.

```
QueueHandle_t xQueueCreate(
  UBaseType_t uxQueueLength,
  UBaseType_t uxItemSize)
```

This call creates a queue for holding queue items. The maximum number of items that the queue can hold is supplied as well as the size of each item in a queue. On return, a handle to the new queue is supplied or has a value of NULL on error.

If the queue is no longer needed, it can be deleted with a call to `vQueueDelete()`.

- `uxQueueLength` – The maximum number of items that the queue can contain.
- `uxItemSize` – The size in bytes reserved for each element in the queue.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

See also:

- vQueueDelete
- xQueueCreate

xQueueCreateSet

Create a queue set.

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength)
```

The uxEventQueueLength must be at least the size of all the queues that will be concurrently watched by this queue set.

See also:

- xQueueAddToSet
- xQueueRemoveFromSet
- xQueueSelectFromSet

xQueueCreateStatic

Create a queue for holding items but allow the application to own the management of queue records.

```
QueueHandle_t xQueueCreateStatic(
  UBaseType_t uxQueueLength,
  UBaseType_t uxItemSize,
  uint8_t *queueStorageBuffer,
  StaticQueue_t pxQueueBuffer)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

vQueueDelete

Delete a previously created queue.

```
void vQueueDelete(xQueueHandle xQueue)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

See also:

- xQueueCreate
- xQueueCreateStatic

xQueueGenericReceive

```
BaseType_t xQueueGenericReceive( QueueHandle_t xQueue, void * const pvBuffer,
TickType_t xTicksToWait, const BaseType_t xJustPeek)
```

uxQueueMessagesWaiting

Return the count of messages on the queue.

```
UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue)
```

The return is the number of messages currently on the queue.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueueOverwrite

```
BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const coid *pvItemToQueue)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueuePeek

```
BasType_t_TYPE xQueuePeek(
  QueueHandle_t xQueue,
  void *pvBuffer,
  TickType_t xTicksToWait)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueuePeekFromISR

```
BaseType_t xQueuePeekFromISR( QueueHandle_t xQueue, void * const pvBuffer)
```

xQueueReceive

Retrieve an item from the head of the queue.

```
BaseType_t xQueueReceive(
  QueueHandle_t xQueue,
  void *pvBuffer,
  TickType_t xTicksToWait)
```

The `xQueue` is the handle to the queue we wish to receive from. The `pvBuffer` is a pointer to storage that will receive the item from the queue. The `xTicksToWait` is how long we wish to wait for an item to become available assuming the queue is empty.

Specifying 0 means return immediately while specifying `portMAX_DELAY` means block indefinitely.

The return is `pdPASS` if we receive an item or `errQUEUE_EMPTY` is no item was retrieved.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueueReceiveFromISR

```
BaseType_t xQueueReceiveFromISR(
   QueueHandle_t xQueue,
   void *pvBuffer,
   BaseType_t *pxHigherPriorityTaskWoken
)
```

xQueueRemoveFromSet

```
BaseType_t xQueueRemoveFromSet(
   QueueSetMemberHandle_t xQueueOrSemaphore,
   QueueSetHandle_t xQueueSet)
```

xQueueReset

```
BaseType_t xQueueReset(QueueHandle_t xQueue)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueueSelectFromSet

Watch a queue set for a member becoming ready.

```
QueueSetMemberHandle_t xQueueSelectFromSet(
   QueueSetHandle_t xQueueSet,
   const TickType_t xTicksToWait)
```

The return is the handle of the queue which is ready for reading without blocking.

See also:

- xQueueAddToSet
- xQueueCreateSet
- xQueueRemoveFromSet

xQueueSelectFromSetFromISR

```
QueueSetMemberHandle_t xQueueSelectFromSetFromISR(QueueSetHandle_t xQueueSet)
```

xQueueSend

This is a macro for backwards compatibility only. Do not use for new projects.

```
BaseType_t xQueueSend(
   QueueHandle_t xQueue,
   const void*   pvItemToQueue,
   TickType_t    xTicksToWait
);
```

xQueueSendFromISR

This is a macro for backwards compatibility only. Do not use for new projects.

```
BaseType_t xQueueSendFromISR(
   QueueHandle_t xQueue,
   const void*   pvItemToQueue,
   BaseType_t*   pxHigherPriorityTaskWoken
);
```

xQueueSendToBack

Place an item at the end of the queue.

```
BaseType_t xQueueSendToBack(
  QueueHandle_t xQueue,
  const void*   pvItemToQueue,
  TickType_t    xTicksToWait)
```

The `xQueue` is the reference to the queue into which we are going to place the item. The `pvItemToQueue` is a pointer to the item we are going to copy into the queue. Note that this is indeed a copy and not saved as a reference. The `xTicksToWait` is how long we are prepared to wait if the queue is currently full. Specifying 0 returns immediately while `portMAX_DELAY` blocks indefinitely.

On return the value `pdPASS` indicates that the message was queued while `errQUEUE_FULL` indicates that the message was not queued and we timed out waiting for space to become available.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xQueueSendToBackFromISR

Place an item at the end of the queue.

```
BaseType_t xQueueSendToBackFromISR(
   QueueHandle_t xQueue,
   const void*   pvItemToQueue,
   BaseType_t*   pxHigherPriorityTaskWoken
)
```

This function is an ISR safe version of the xQueueSendToBack() function. The `xQueue` is the reference to the queue into which we are going to place the item. The `pvItemToQueue` is a pointer to the item we are going to copy into the queue. Note that this is indeed a copy and not saved as a reference.

The `pxHigherPriorityTaskWoken` is a pointer to a flag that is set on return. If its value is `pdTRUE` then a higher priority task is ready and a context switch is possible. It can be `NULL` to be ignored.

The return is `pdPASS` if the item was queued otherwise `errQUEUE_FULL` to indicate that the queue was full.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

See also:

- xQueueSendToBackFromISR

xQueueSendToFront

```
BaseType_t xQueueSendToFront(
 QueueHandle_t xQueue,
 const void *  pvItemToQueue,
 TickType_t    xTicksToWait)
```

The `xQueue` is the reference to the queue into which we are going to place the item. The `pvItemToQueue` is a pointer to the item we are going to copy into the queue. Note that this is indeed a copy and not saved as a reference. The `xTicksToWait` is how long we are prepared to wait if the queue is currently full. Specifying 0 returns immediately while `portMAX_DELAY` blocks indefinitely.

On return the value `pdPASS` indicates that the message was queued while `errQUEUE_FULL` indicates that the message was not queued and we timed out waiting for space to become available.

Includes:

- #include <freertos/FreeRTOS.h>

- #include <freertos/queue.h>

xQueueSendToFrontFromISR

```
BaseType_t xQueueSendToFrontFromISR(
   QueueHandle_t xQueue,
   const void*   pvItemToQueue,
   BaseType_t*   pxHigherPriorityTaskWoken
)
```

uxQueueSpacesAvailable

```
UBaseType_t uxQueueSpacesAvailable(QueueHandle_t xQueue)
```

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/queue.h>

xRingbufferAddToQueueSetRead

```
BaseType_t xRingbufferAddToQueueSetRead(
   RingbufHandle_t  ringbuf,
   QueueSetHandle_t xQueueSet)
```

xRingbufferAddToQueueSetWrite

Add the ring buffer as an indicator item to the queue set.

```
BaseType_t xRingbufferAddToQueueSetWrite(
   RingbufHandle_t  ringbuf,
   QueueSetHandle_t xQueueSet)
```

Making this call adds the ring buffer to the queue set such that a write to the ring buffer will cause a trigger of any queue set being watched.

See also:

- xQueueCreateSet
- xQueueSelectFromSet

xRingbufferCreate

Create a new ring buffer.

```
RingbufHandle_t xRingbufferCreate(
   size_t         buf_length,
   ringbuf_type_t type)
```

- `length` – The size in bytes that we wish to allocate for the ring buffer.
- `type` – The type of ring buffer with respect to how wrapping is to occur. The choices are:
    - RINGBUF_TYPE_NOSPLIT
    - RINGBUF_TYPE_ALLOWSPLIT
    - RINGBUF_TYPE_BYTEBUF

vRingbufferDelete

Delete a ring buffer.

```
void vRingbufferDelete(RingbufHandle_t ringbuf)
```

Delete a ring buffer previously created with a call to `xRingbufferCreate()`.

xRingbufferGetMaxItemSize

Get the maximum size of an item that will fit within the remaining space in the ring buffer.

```
size_t xRingbufferGetMaxItemSize(RingbufHandle_t ringbuf)
```

xRingBufferPrintInfo

```
void xRingbufferPrintInfo(RingbufHandle_t ringbuf)
```

xRingbufferReceive

Retrieve an item from the ring buffer.

```
void* xRingbufferReceive(
   RingbufHandle_t ringbuf,
   size_t*         item_size,
   TickType_t      ticks_to_wait)
```

- `ringbuf` – The ringbuf to receive an item from.
- `item_size` – The size of the item received.
- `ticks_to_wait` – How long to wait. Specify `portMAX_DELAY` for an indefinite block.

The return from this call is a pointer to storage that is the next item within the buffer. Since items in a ring buffer can be of distinct sizes, the `item_size` variable is populated with the size of the data in this returned record instance. We must subsequently call `vRingbufferReturnItem()` to release the storage when we are finished with it.

xRingbufferReceiveFromISR

```
void *xRingbufferReceiveFromISR(
   RingbufHandle_t ringbuf,
   size_t*         item_size)
```

xRingbufferReceiveUpTo

Return data from the ring buffer up to a maximum size.

```
void *xRingbufferReceiveUpTo(
   RingbufHandle_t ringbuf,
   size_t*         item_size,
   TickType_t      ticks_to_wait,
   size_t          wanted_size)
```

xRingbufferReceiveUpToFromISR

```
void *xRingbufferReceiveUpToFromISR(
   RingbufHandle_t ringbuf,
   size_t*         item_size,
   size_t          wanted_size)
```

xRingbufferRemoveFromQueueSetRead

```
BaseType_t xRingbufferRemoveFromQueueSetRead(
   RingbufHandle_t  ringbuf,
   QueueSetHandle_t xQueueSet)
```

xRingbufferRemoveFromQueueSetWrite

```
BaseType_t xRingbufferRemoveFromQueueSetWrite(
   RingbufHandle_t  ringbuf,
   QueueSetHandle_t xQueueSet)
```

vRingbufferReturnItem

Release the storage for a previously retrieved item.

```
void vRingbufferReturnItem(
   RingbufHandle_t ringbuf,
   void*           item)
```

When we make a call to `xRingbufferReceive()` we are supplied a pointer to storage that is contained within the ring buffer. That storage is not released until we invoke `vRingbufferReurnItem()`.

vRingbufferReturnItemFromISR

```
void vRingbufferReturnItemFromISR(
   RingbufHandle_t ringbuf,
   void*           item,
   BaseType_t*     higher_prio_task_awoken)
```

xRingbufferSend

Insert an item into the ring buffer.

```
BaseType_t xRingbufferSend(
   RingbufHandle_t ringbuf,
   void*           data,
   size_t          data_size,
   TickType_t      ticks_to_wait)
```

- `ringbuf` – A ring buffer handle.
- `data` – Pointer to the data to be inserted into the buffer.
- `data_size` – Size of the data to insert into the buffer.
- `ticks_to_wait` – Ticks to wait before giving up for enough space to be available in the buffer to insert the item.

xRingbufferSendFromISR

Insert an item into the ring buffer from an ISR.

```
BaseType_t xRingbufferSendFromISR(
   RingbufHandle_t ringbuf,
   void*           data,
   size_t          data_size,
   BaseType_t*     higher_prio_task_awoken)
```

- `ringbuf` – A ring buffer handle.
- `data` – Pointer to the data to be inserted into the buffer.
- `data_size` – Size of the data to insert into the buffer.
- higher_prio_task_awoken

vSemaphoreCreateBinary

Includes:

- #include <freertos/semphr.h>

xSemaphoreCreateCounting

Includes:

- #include <freertos/semphr.h>

xSemaphoreCreateMutex

Create a mutex.

```
SemaphoreHandle_t xSeamphoreCreateMutex()
```

Create an instance of a semaphore.

Includes:

- #include <freertos/semphr.h>

vSemaphoreDelete

Delete a semaphore.

```
void vSemaphoreDelete(SemaphoreHandle_t semaphore);
```

Release resources for the given semaphore.

Includes:

- #include <freertos/semphr.h>

vSemaphoreGive

Release a mutex.

```
xSemaphoreGive(SemaphoreHandle_t semaphore)
```

Release a mutex.

Includes:

- #include <freertos/semphr.h>

xSemaphoreGiveFromISR

```
xSemaphoreGive(SemaphoreHandler_t semaphore, BaseType_t *pxHigherPriorityTaskWoken)
```

Includes:

- #include <freertos/semphr.h>

vSemaphoreTake

Take a semaphore/mutex.

```
bool xSemaphoreTake(SemaphoreHandle_t semaphore, TickType_t ticksToWait)
```

- `semaphore` – A semaphore handle.
- `ticksToWait` – How long to wait before returning due to a timeout because we didn't obtain the semaphore. The value `portMAX_DELAY` will block indefinitely. A value of 0 will return immediately, effectively polling the semaphore.

Returns true if the semaphore was taken and false otherwise.

Includes:

- #include <freertos/semphr.h>

xTaskCreate

Create a new instance of a task.

```
BaseType_t xTaskCreate(
  pdTASK_CODE            pvTaskCode,
  const signed portCHAR* pcName,
  unsigned portSHORT     usStackDepth,
  void*                  pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle*           pxCreatedTask)
```

- `pvTaskCode` – Pointer to the task function. In C programming, we can simply supply the name of a function or, as has been seen in some samples, the address of the name of the function. Apparently these equate to items that are close enough to be used interchangeably.
- `pcName` – Debugging name of the task.
- `usStackDepth` – Size of the stack for the task.
- `pvParameters` – Parameters for the task instance. This may be NULL.
- `uxPriority` – Priority of the task instance.
- `pxCreatedTask` – Reference to the newly created task instance. This may be passed in as NULL if no task handle is required to be returned.

In an ESP32 environment, an example, we might have:

```
void myTask(void *parms) {
   // Do something
   vTaskDelete(NULL);
}
```

```
main() {
   xTaskCreate(&myTask, "myTask", 2048, NULL, 5, NULL);
}
```

When a created task is invoked and then decides to end via a return, it is essential that the task call `vTaskDelete(NULL)` before it completes. Calling this is an indication to FreeRTOS that the task is finished and need no longer be considered for context switching. Experience seems to show that if we don't do this then the ESP32 will crash. In summary, end your return logic in your task functions with:

```
{
   // Previous code here.
   vTaskDelete(NULL);
   return;
}
```

Include:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- vTaskDelete
- xTaskCreate

xTaskCreatePinnedToCore

```
BaseType_t xTaskCreatePinnedToCore(
   TaskFunction_t  pxTaskCode,
   const char*     pcName,
   const uint16_t  usStackDepth,
   void*           pvParameters,
   UBaseType_t     uxPriority,
   TaskHandle_t*   pxCreatedTask,
   const BasType_t xCoreID)
```

This is a mysterious function that appears to be ESP32 specific.

When a created task is invoked and then decides to end via a return, it is essential that the task call `vTaskDelete(NULL)` before it completes. Calling this is an indication to FreeRTOS that the task is finished and need no longer be considered for context switching. Experience seems to show that if we don't do this then the ESP32 will crash. In summary, end your return logic in your task functions with:

```
{
   // Previous code here.
   vTaskDelete(NULL);
   return;
}
```

In an ESP32 environment, an example, we might have:

```
void myTask(void *parms) {
   // Do something
   vTaskDelete(NULL);
}

main() {
   xTaskCreatePinnedToCore(&myTask, "myTask", 2048, NULL, 5, NULL, 0);
}
```

The `xCoreID` specifies which core to run on. Choices are 0 (PRO CPU) or 1 (APP CPU) to "`tskNO_AFFINITY`".

Include:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- vTaskDelete

vTaskDelay

Delay a task for a specified number of ticks.

```
void vTaskDelay(const TickType_t xTicksToDelay)
```

The constant called `portTICK_PERIOD_MS` provides the number of ticks in a millisecond. If we wished to delay for 1 second, we could then supply 1000 / `portTICK_PERIOD_MS`.

Include:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- vTaskDelay
- Timers in FreeRTOS

vTaskDelayUntil

Delay a task until a specified absolute time.

```
void vTaskDelayUntil(
   const TickType_t* pxPreviousWakeTime,
   const TickType_t  xTimeIncrement)
```

Note: This is flagged as an un-tested FreeRTOS function.

This function blocks a task until some absolute time in the future.

- `pxPreviousWakeTime` – The base time which the increment will be relative from.
- `xTimeIncrement` – The time in ticks which, when added to the `pxPreviousWakeTime`, will be the time that the task is ready to run again.

Include:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- vTaskDelayUntil
- Timers in FreeRTOS

vTaskDelete

Delete an instance of a task.

```
void vTaskDelete(TaskHandle_t pxTask)
```

This function will delete an instance of a task. If the `pxTask` handle is `NULL` then the current task will be deleted.

Include:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- vTaskDelete

vTaskGetInfo

```
void vTaskGetInfo(
   TaskHandle_t  xTask,
   TaskStatus_t* pxTaskStatus,
   BaseType_t    xGetFreeStackSpace,
   eTaskState    eState)
```

The calculation of the current state that is returned can be time consuming, to calculate its value, pass in `eState` with a value of `eInvalid`.

The calculation of the minimum amount of free stack space we have seen is also expensive so pass in a value of `pdTRUE` to retrieve it and `pdFALSE` to ignore it.

The TaskStatus_t contains:

- `TaskHandle_t xHandle` –
- `const signed char *pcTaskname` –
- `UBaseType_t xTaskNumber` –
- `eTaskState eCurrentState` –
- `UBaseType_t uxCurrentPriority` –
- `UBaseType_t uxBasePriority` –
- `unsigned long ulRunTimeCounter` –
- `StackType_t *pxStackBase` –
- `unsigned short usStackHighWaterMark` –

xTaskGetCurrentTaskHandle

Get the current task handle.

```
TaskHandle_t xTaskGetCurrentTaskHandle()
```

pcTaskGetTaskName

Get the name of the task.

```
char *pcTaskGetTaskName(TaskHandle_t xTaskToQuery)
```

If we supply NULL for the task handle, we get our own task name.

uxTaskGetNumberOfTasks

Get the number of tasks.

```
UBaseType_t uxTaskGetNumberOfTasks()
```

Once we know the number of tasks, we can retrieve the details of them through a call to `uxTaskGetSystemState()`.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- uxTaskGetSystemState

eTaskGetState

Retrieve the state of a task.

```
eTaskState eTaskGetState(TaskHandle_t xTask)
```

The possible values are:

- eReady –
- eRunning –
- eBlocked –
- eSuspended –
- eDeleted –

uxTaskGetSystemState

Note: This is disabled by default. To enable the configUSE_TRACE_FACILITY definition in FreeRTOSConfig.h must be set to 1. When enabled, we still get an assert as this is flagged as an "untested" function.

Retrieve the task state for all tasks in the environment.

```
UBaseType_t uxTaskGetSystemState(
   TaskStatus_t*     pxTaskStatusArray,
   const UBaseType_t uxArraySize,
   uint32_t*         pulTotalRunTime)
```

The TaskStatus_t contains:

- TaskHandle_t xHandle –
- const signed char *pcTaskname –
- UBaseType_t xTaskNumber –
- eTaskState eCurrentState –
- UBaseType_t uxCurrentPriority –
- UBaseType_t uxBasePriority –
- unsigned long ulRunTimeCounter –
- StackType_t *pxStackBase –
- unsigned short usStackHighWaterMark –

The return type is the number of tasks populated. We can call uxTaskGetNumberOfTasks() to determine how many tasks are present.

The `pulTotalRunTime` is set to the total run time since booted. We can pass NULL if we don't need the value.

Includes:

- #include <freertos/FreeRTOS.h>
- #include <freertos/task.h>

See also:

- uxTaskGetNumberOfTasks
- vTaskGetInfo

xTaskGetTickCount

Get the current tick count.

```
TickType_t xTaskGetTickCount()
```

Return the number of ticks that have occurred since the task scheduler was started.

The macro called `portTICK_PERIOD_MS` defines the duration of a single tick in milliseconds.

Include:

- #include <freertos/task.h>

See also:

- Timers in FreeRTOS

xTaskGetTickCountFromISR

Get the current tick count from an Interrupt Service Routine.

```
TickType_t xTaskGetTickCountFromISR()
```

Include:

- #include <freertos/task.h>

vEventGroupDelete

Delete an event group.

```
void vEventGroupDelete(EventGroupHandle_t eventGroup)
```

Includes:

- #include <freertos/event_groups.h>

vTaskList

```
void vTaskList(char *pcWriteBuffer)
```

NOT AVAILABLE

uxTaskPriorityGet

Get the priority of the task.

```
UbaseType_t uxTaskPriorityGet(TaskHandle_t xTask)
```

vTaskPrioritySet

```
void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority)
```

Include:

- #include <freertos/task.h>

See also:

- Error: Reference source not found

vTaskResume

Resume a task that was previously suspended.

```
void vTaskResume(TaskHandle_t pxTaskToResume)
```

Include:

- #include <freertos/task.h>

See also:

- vTaskSuspend

xTaskResumeAll

Include:

- #include <freertos/task.h>

See also:

- vTaskResumeAll

vTaskResumeFromISR

```
void xTaskResumeFromISR(TaskHandle_t pxTaskToResume)
```

Include:

- #include <freertos/task.h>

vTaskSuspend

Suspend the given task.

```
void vTaskSuspend(TaskHandle_t pxTaskToSuspend)
```

The identified task is suspended until it is allowed to continue.

Include:

- #include <freertos/task.h>

See also:

- vTaskResume
- xTaskResumeAll

vTaskSuspendAll

Include:

- #include <freertos/task.h>

See also:

- vTaskSuspendAll

xTimerChangePeriod

Change the period of an existing timer.

```
BaseType_t xTimerChangePeriod(
   TimerHandle_t xTimer,
   TickType_t    xNewPeriod,
   TickType_t    xBlockTime)
```

When a timer is created through a call to xTimerCreate(), we can subsequently change the period of that timer.

- `xTimer` – The handle to the existing timer.
- `xNewPeriod` – The new period value for the timer.
- `xBlockTime` – The duration that this function can maximally wait for processing.

Includes:

- #include <freertos/timers.h>

xTimerChangePeriodFromISR

Change the period of an existing timer from within an Interrupt Service Routine.

```
BaseType_t xTimerChangePeriodFromISR(
   TimerHandle_t xTimer,
   TickType_t  xNewPeriod,
   BaseType_t* pxHigherPriorityTaskWoken)
```

- `xTimer` – The handle to the existing timer.
- `xNewPeriod` – The new period value for the timer.
- `pxHigherPriorityTaskWoken` – Flag indicating that a context switch should occur.

Includes:

- #include <freertos/timers.h>

xTimerCreate

Create a new timer.

```
TimerHandle_t xTimerCreate(
   const char*             pcTimerName,
   const TickType_t        xTimerPeriod,
   const UBaseType_t       uxAutoReload,
   void*                   pvTimerID,
   TimerCallbackFunction_t pxCallbackFunction)
```

- `pcTimerName` – The name of the timer used for debugging and reporting.
- `xTimerPeriod` – The period, measured in ticks, after which the timer should fire.
- `uxAutoReload` – A flag which indicates whether or not the timer should restart once fired. If restarted, it will start again using the xTimerPeriod value for the next interval.
- `pvTimerID` – Data available to the timer callback function when fired.
- `pxCallbackFunction` – Function called when the timer expires. The signature of this function is:

```
void vCallbackFunction(TimerHandle_t xTimer)
```

The return from the `xTimerCreate` function is a handle that can be used to refer to this timer in the future.

Includes:

- #include <freertos/timers.h>

xTimerCreateStatic

```
TimerHandle_t xTimerCreateStatic(
   const char*      pcTimerName,
   const TickType_t xTimerPeriod,
   const UBaseType_t       uxAutoReload,
   void*                   pvTimerID,
   TimerCallbackFunction_t pxCallbackFunction
   StaticTimer_t*          pxTimerBuffer)
```

Includes:

- #include <freertos/timers.h>

xTimerDelete

Delete an existing timer.

```
BaseType_t xTimerDelete(TimerHandle_t xTimer, TickType_t xBlockTime)
```

This call deletes a timer that was previous created by an `xTimerCreate()` call. Following a successful call, we should not attempt to refer to this timer again.

- `xTimer` – The handle to the timer to delete.
- `xBlockTime` – The length of timer we should block waiting for the delete to be processed.

Includes:

- #include <freertos/timers.h>

pcTimerGetName

Get the name of the timer.

```
const char * pcTimerGetName(TimerHandle_t xTimer)
```

When we created a timer using `xTimerCreate()` we supplied a name identity for the timer. This call returns that value.

- `xTimer` – The handle to a previously created timer.

Includes:

- #include <freertos/timers.h>

xTimerGetExpiryTime

Note: Not implemented in ESP-IDF.

Return the time at which the timer will fire.

```
TickType_t xTimerGetExpiryTime(TimerHandle_t xTimer)
```

For an active timer, this call will return the tick value at which the timer will fire. This is an absolute value. If we want to know how many ticks in the future this will be, we can obtain the current tick value and subtract one from the other.

- `xTimer` – The handle to a previously created timer.

Includes:

- #include <freertos/timers.h>

See also:

- xTaskGetTickCount

xTimerGetPeriod

Note: Not implemented in ESP-IDF.

Get the period value for a timer.

```
TickType_t xTimerGetPeriod(TimerHandle_t xTimer)
```

Return the period value for the timer. This is the value passed in when `xCreateTimer()` was initially created. It is a relative value rather than an absolute expiry time. If the timer is configured to restart, this is the value that will be used to fire the next instance of the timer.

- `xTimer` – The handle to a previously created timer.

Includes:

- #include <freertos/timers.h>

pvTimerGetTimerDaemonTaskHandle

Retrieve the task handle to the timer daemon.

```
TaskHandle_t xTimerGetTimerDaemonTaskHandle(void)
```

Behind the scenes, FreeRTOS is running a task that is managing the timers. There may be times when we want to get access to the handle for this task. This API returns the handle.

Includes:

- #include <freertos/timers.h>

pvTimerGetTimerID

Retrieve the data associated with an instance of the timer.

```
void *pvTimerGetTimerID(TimerHandle_t xTimer)
```

When a timer is created, we can associate data with it. We can also update that data at a later timer. This API retrieves the current value of the data.

- `xTimer` – The handle to a previously created timer.

Includes:

- #include <freertos/timers.h>

xTimerIsTimerActive

Determine if the timer is active.

```
BaseType_t xTimerIsTimerActive(TimerHandle_t xTimer)
```

An active timer is one that is ticking down to a time when it will fire.

- `xTimer` – The handle to a previously created timer.

Includes:

- #include <freertos/timers.h>

xTimerPendFunctionCall

Execute a function on the timer daemon task.

```
BaseType_t xTimerPendFunctionCall(
   PendedFunction_t xFunctionToPend,
   void*            pvParameter1,
   uint32_t         ulParameter2,
   TickType_t       xTicksToWait)
```

Calling this API causes the function supplied to execute on the timer daemon task.

- `xFunctionToPend` – The function to be executed in the context of the timer daemon task. The function has the following signature:

```
void vPendableFunction(void *pvParameter1, uint32_t ulParameter2)
```

- `pvParameter1` – A parameter to be passed to the function.
- `ulParameter2` – A parameter to be passed to the function.
- `xTicksToWait` – A duration to wait for the function to be started.

Includes:

- #include <freertos/timers.h>

xTimerPendFunctionCallFromISR

```
BaseType_t xTimerPendFunctionCallFromISR(
   PendedFunction_t xFunctionToPend,
   void*            pvParameter1,
   uint32_t         ulParameter2,
   BaseType_t*      pxHigherPriorityTaskWoken)
```

Includes:

- #include <freertos/timers.h>

xTimerReset

Reset the value of the timer.

```
BaseType_t xTimerReset(TimerHandle_t xTimer, TickType_t xBlockTime)
```

Reset the value of the timer to start ticking from the time when this API was called with the period associated with the timer.

- `xTimer` – The handle to a previously created timer.
- `xBlockTime` – The time to wait for the reset to complete.

Includes:

- #include <freertos/timers.h>

xTimerResetFromISR

```
BaseType_t xTimerResetFromISR(
   TimerHandle_t xTimer,
   BaseType_t*   pxHigherPriorityTaskWoken)
```

vTimerSetTimerID

Note: Not implemented in ESP-IDF.

Associate data with the timer.

```
void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID)
```

- `xTimer` – The handle to a previously created timer.
- `pvNewID` – Reference to data to be associated with the timer.

Includes:

- #include <freertos/timers.h>

xTimerStart

Start a timer ticking.

```
BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xBlockTime)
```

The timer can be stopped before expiration with a call to `xTimerStop()`.

- `xTimer` – The handle to a previously created timer.
- `xBlockTime` – The time to wait for the start request to complete.

Includes:

- #include <freertos/timers.h>

xTimerStartFromISR

```
BaseType_t xTimerStartFromISR(
   TimerHandle_t xTimer,
   BaseType_t *pxHigherPriorityTaskWoken)
```

Includes:

- #include <freertos/timers.h>

xTimerStop

Stop a timer ticking.

```
BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xBlockTime)
```

- `xTimer` – The handle to a previously created timer.
- `xBlockTime` – The time to wait for the stop request to complete.

Includes:

- #include <freertos/timers.h>

xTimerStopFromISR

```
BaseType_t xTimerStopFromISR(
   TimerHandle_t xTimer,
   BaseType_t *pxHigherPriorityTaskWoken)
```

Includes:

- #include <freertos/timers.h>

List Processing

vListInitialise

Initialize a list.

```
void vListInitialise(xList * const pxList)
```

The `pxList` is a list that should be initialized.

vListInitialiseItem

Initialize an item for insertion into a list.

```
void vListInitialiseItem(xListItem * const pxItem)
```

Initialize an item that can be added to a list.

vListInsert

Insert an item into a list.

```
void vListInsert(xList * const pxList, xListItem * const pxNewListItem)
```

vListInsertEnd

Insert an item at the end of a list

```
void vListInsertEnd( xList * const pxList, xListItem * const pxNewListItem)
```

Sockets APIs

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.

See also:

- TCP/IP Sockets

accept

Accept an incoming request.

```
int accept(int socket_fd, struct sockaddr* addr, socklen_t* addrlen)
```

Here we can block waiting for an incoming connection request from the server socket. We will return immediately if there is a client connection awaiting acceptance. The address of the client is returned to us along with its length.

If we try and accept too many concurrent sockets, the ESP32 may return `ENFILE` to indicate that we have an overflow.

The return is the accepted socket connection to the client or -1 if an error.

See also:

- TCP/IP Sockets
- man(2) – accept

bind

Associate a socket with an address.

```
int bind(int s, const struct sockaddr* name, socklen_t namelen)
```

The `name` parameter is the socket address to be bound to the socket. The `namelen` provides the length of the address. If the `sin_add.s_addr` is `htonl(INADDR_ANY)` then we are being a server listening on any incoming IP address.

A return of < 0 on error.

Here is an example of us defining ourselves as a server:

```
struct sockaddr_in serverAddr;
serverAddr.sin_family      = AF_INET;
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddr.sin_port        = htons(80);

int rc = bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
```

If we have bound a socket to a local port and then close that bound socket, the port number remains reserved for a period of time to "cleanup" any in-flight traffic. This prevents the socket from being re-used and a bind() attempt can result in an "address already in use" error. We can request that the bind() succeed even if the port is in this timed wait state. We do this by running:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int));
```

which sets the sockets level "`SO_REUSEADDR`" flag.

See also:

- TCP/IP Sockets
- man(2) – bind

close

Close the corresponding socket.

```
int close(int s)
```

Close the socket. After closing a socket, no further sockets APIs should be called against it. This is the last call in the chain. Take care to realize that when a socket is created with the `socket()` API call the return is a socket handle that is also known as a file descriptor. A socket handle is unique **only** for the lifetime of the socket. For example, if you call `socket()`, you might get back a handle that has a value of 3. While that socket is alive, no other caller of `socket()` will be issued the same handle. However, once you close the `socket()`, it is entirely possible that the next caller of `socket()` will itself be issued the value of 3 because, at that time, the socket handle is unused. The reason for spending time on this is that if you are saving socket handles it is not difficult to close a handle and then forget that you have closed it and later make a sockets API call against the value of the handle when in fact it has been assigned to a different socket in the intervening time period. No technical errors will be thrown as at the time you are making a socket API call, it is indeed a valid socket handle you are using … it just happens to not be the one your were expecting to use.

See also:

- man(2) – close
- socket

closesocket

Close the corresponding socket.

```
closesocket(int s)
```

Close the socket.

connect

Connect to a server.

```
int connect(int sockFd, const struct sockaddr* partnerAddr, socklen_t addrlen)
```

Connect the socket to a partner. The address of the partner is supplied in the `partnerAddr` field. This is a client initiated call and is expected to connect with a listening server.

For example:

```
struct sockaddr_in serverAddress;
serverAddress.sin_family = AF_INET;
inet_pton(AF_INET, "192.168.1.200", &serverAddress.sin_addr.s_addr);
serverAddress.sin_port = htons(9999);

int rc = connect(sock, (struct sockaddr *)&serverAddress, sizeof(struct sockaddr_in));
```

Includes:

- #include <lwip/sockets.h>

See also:

- man(2) – connect

fcntl

Perform control functions.

```
fcntl(int s, int cmd, int val)
```

We can set control functions on sockets here.

Command	Value	Description
F_SETFL	O_NONBLOCK	Set the socket non blocking.

See also:

- man(2) – fcntl

freeaddrinfo

Release storage allocated by `getaddrinfo()`.

```
void freeaddrinfo(struct addrinfo* res)
```

Includes:

- #include <lwip/netdb.h>

See also:

- getaddrinfo

getaddrinfo

Build an address structure for a desired target.

```
int getaddrinfo(
   const char*            node,
   const char*            service,
   const struct addrinfo* hints,
   struct addrinfo**      res)
```

The `node` parameter is the host-name if the target server. The `service` is the name of a service used to look-up the target port number. It isn't clear how that would work on an ESP32. Where would the service to port mappings be kept?

The `hints` provides a filter for what to return. I can be `NULL`. If it is supplied, it is a pointer to a `struct addrinfo` with the following fields filled in:

The `ai_family` can be one of:

- AF_INET
- AF_INET6
- AF_UNSPEC

The ai_socktype can be one of:

- SOCK_STREAM
- SOCK_DGRAM

The `addrinfo` structure is defined as follows:

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr* ai_addr;
    char*            ai_canonname;
    struct addrinfo* ai_next;
};
```

A return code of 0 indicates success otherwise it returns an error code. Note that errno is not used as that is not thread safe.

The res pointer points to an allocate `struct addrinfo` storage area that must be subsequently released with a call to `freeaddrinfo()`.

The `struct sockaddr` is common cast to a `struct sockaddr_in` which is defined as:

```
struct sockaddr_in {
    short            sin_family;
    unsigned short   sin_port;
    struct in_addr   sin_addr;
    char             sin_zero[8];
};

struct in_addr {
    unsigned long s_addr;
};
```

What this means is that the IP address can be found at `var.sin_addr` or `var.sin_addr.s_addr`.

Includes:

- #include <lwip/netdb.h>

See also:

- freeaddrinfo
- gethostbyname
- gethostbyname_r
- man(3) – getaddrinfo

gethostbyname

Look-up a host name through name resolution.

```
struct hostent *gethostbyname(const char *name)
```

Most likely a wrapper around the lwip `dns_gethostbyname()` implementation provided by the underlying lwip implementation.

A pointer to a `struct hostent` is returned. This is a buffer maintained by the network layer and will be changed in the next call. The members of a struct hostent are:

Member	Description
`char* h_name`	The name of the host.
`char** h_aliases`	An array of alternative names. Last entry is a NULL pointer.
`int h_addrtype`	The type of addresses. One of AF_INET or AF_INET6.
`int h_length`	The length of an address in bytes.
`char** h_addr_list`	An array of network addresses for the host. Last entry is a NULL pointer.

Includes:

- #include <lwip/netdb.h>

See also:

- dns_getserver
- dns_setserver
- inet_ntop
- inet_pton
- LWIP Wiki – DNS
- man(3) – gethostbyname

gethostbyname_r

A re-entrant version of name resolution.

```
int gethostbyname_r(
   const char*      name,
   struct hostent*  ret,
   char*            buf,
   size_t           buflen,
   struct hostent** result,
   int*             h_errnop)
```

See here for an example of use.

See also:

- man(3) – gethostbyname_r

getpeername

Retrieve the address associated with the partner/peer to which the socket is connected.

```
int getpeername(int s,
   struct sockaddr* peerAddr,
   socklen_t*       namelen)
```

Note that `namelen` must be primed with the size of the available address buffer.

See also:

- man(2) – getpeername

getsockname

Retrieve the current local address to which the socket is bound.

```
getsockname(int s,
   struct sockaddr* name,
   socklen_t*       namelen)
```

Note that `namelen` must be primed with the size of the available address buffer.

See also:

- man(2) – getsockname

getsockopt

```
int getsockopt(
   int        s,
   int        level,
   int        optname,
   void*      optval,
   socklen_t* optlen)
```

An important example of using this function is to retrieve the last error associated with the socket. The following code fragment illustrates this:

```
int espx_last_socket_errno(int socket) {
   int ret = 0;
   u32_t optlen = sizeof(ret);
   getsockopt(socket, SOL_SOCKET, SO_ERROR, &ret, &optlen);
   return ret;
}
```

See also:

- man(2) - getsockopt

htonl

Convert a host formatted long integer to network byte order.

```
uint32_t htonl(uint32_t netLong)
```

See also:

- man(3) – htonl

htons

Convert a host formatted short integer to network byte order.

```
uint16_t htons(uint16_t hostShort)
```

See also:

- man(3) – htons

inet_ntop

Convert an IP address structure into a character string.

```
char *inet_ntop(int af, const char* src, char* dst, socklen_t size)
```

The `af` parameter specified the address family. It may have a value of:

- `AF_INET` – IPv4 network addresses.
- `AF_INET6` – IPv6 network addresses.

The `src` is the pointer to the address structure. The `dst` is the buffer that will be filled with the text and `size` is the length of that buffer available to be filled. For `AF_INET`, the buffer should be at least `INET_ADDRSTRLEN` bytes long while for `AF_INET6`, it should be at least `INET6_ADDRSTRLEN` bytes long.

Includes:

- #include <lwip/sockets.h>

See also:

- man(3) – inet_ntop

inet_pton

Convert an IP address from string to binary form.

```
int inet_pton(int af, const char* src, void* dst)
```

The `af` parameter is the address family. It must be one of:

- `AF_INET` – IPv4 network address.
- `AF_INET6` – IPv6 network address.

The `src` is the pointer to a null terminate string that represents the text format of the address.

Returns 1 on success.

Includes:

- #include <lwip/sockets.h>

See also:

- man(3) – inet_pton

ioctlsocket

```
ioctl(int s, long cmd, void *argp)
```

listen

Start listening for incoming connections.

```
int listen(int socket_fd, int backlog)
```

If we are bound as a server, we will start to listen for incoming connections requests on the socket. The `backlog` parameter defines how many sockets we can keep a handle to before we accept them.

A return value < 0 means an error.

See also:

- TCP/IP Sockets
- man(2) – listen

read

Receive data from a partner.

```
ssize_t read(int s, void* mem, size_t len)
```

Similar to the `recv()` function.

See also:

- recv
- recvfrom
- man(2) – read

recv

Receive data from a partner.

```
ssize_t recv(int s,
   void*  mem,
   size_t len,
   int    flags)
```

This function returns the number of bytes actually received. A value of -1 indicates an error. A value of zero indicates the partner having closed the connection.

The flags is the boolean combination of:

- `MSG_CMSG_CLOEXEC` –
- `MSG_DONTWAIT` – Indicate that we don't want to block waiting for data. If there is no data immediately available for us to receive, we return -1 to indicate an error and the error code is "`EAGAIN`".
- `MSG_ERRQUEUE` –
- `MSG_OOB` –
- `MSG_PEEK` –
- `MSG_TRUNC` –
- `MSG_WAITALL` –

Includes:

- #include <lwip/sockets.h>

See also:

- read
- recvfrom
- man(2) – recv

recvfrom

Receive a datagram from another machine/device.

```
ssize_t recvfrom(int sock,
   void*  mem,
   size_t len,
   int              flags,
   struct sockaddr* from,
   socklen_t*       fromlen)
```

The `sock` is a socket handle that was previously opened by calling `socket()`. The `mem` is a pointer to a buffer that will hold the incoming data. The `len` parameter supplies the length of the buffer. The `from` is a pointer to a storage area that will hold the address of the partner and `fromLen` will be the length of that address. On initial call, it should contain the length of the `from` address. If we don't need the return address of the caller, we can supply NULL for both `from` and `fromlen`.

The flags can be specified as:

- `MSG_DONTWAIT` – Don't block waiting for a message not yet arrived.
- `MSG_OOB` – Check for an out of band message.
- `MSG_PEEK` – Receive the first message without consuming it.

The return value is the number of bytes actually received or -1 on an error.

Includes

- #include <sys/types.h>

See also:

- read
- recv
- socket
- sendto
- man(2) – recvfrom

select

Check for data available for reading or writing.

```
int select(int maxfdp1,
   fd_set* readset,
   fd_set* writeset,
   fd_set* exceptset,
   struct timeval* timeout)
```

The notion of "ready" is that an operation could be performed without blocking.

The `maxfdp1` is the maximum value of the file descriptor plus 1.

On some Linux systems, to use selected, one would include `<sys/select.h>`. In ESP-IDF that is not present but does not appear to be needed.

Each of the `fd_set` variables defines a set of file descriptors. To include a file descriptor in a set we can run `FD_SET(fd, &set)`. To remove an entry we can call `FD_CLR(fd,`

&set). To test if a file descriptor is in a set we can run FD_ISSET(fd, *set) and to clear a complete set we can run FD_ZERO(&set). If we aren't interested in some of the sets, we can supply NULL to have them ignored.

If timeout is NULL, then there will be no timeout and select will wait indefinitely. If both fields in the timeval are 0, then we return immediately.

On return, select() returns the number of bits set in the various sets (which may be 0 if there was a timeout). An error of -1 is returned on an error.

A socket that is a server socket is considered "ready" if a call to accept() would not block. This socket should be supplied in the readset.

Take care with the first parameter. This is the number of file descriptor bits to check in the file descriptor mask set. If the highest file descriptor value we have is "n" then the number of file descriptor bits we are examining is "n+1".

See also:

- Using select()
- man(2) – select

send

Send a set of bytes down the socket to the partner.

```
ssize_t send(int s, const void* dataptr, size_t size, int flags)
```

The data pointed to by dataptr for size bytes is transmitted.

See also:

- man(2) – send

sendmsg

```
ssize_t sendmsg(int s, const struct msghdr* msg, int flags)
```

See also:

- man(2) – sendmsg

sendto

Send data to a UDP partner.

```
ssize_t sendto(int sock,
  const void* dataptr,
  size_t      size,
  int         flags,
  const struct sockaddr* to,
  socklen_t   tolen)
```

The `sock` is the socket handle that was previously opened by a call to `socket()`. The `dataptr` is a pointer to a buffer of data that we wish to send. The `size` is the size of the datagram that we wish to send. It can be a maximum of 64K. The `to` is the address of the destination of the message and `tolen` is the length of the address structure.

The flags can be one or more of:

- `MSG_DONTWAIT` – Do not wait if waiting would be needed.
- `MSG_OOB` – Send a message out of band.

The return is the number of bytes actually sent or -1 on an error.

See also:

- recvfrom
- socket
- man(2) – sendto

setsockopt

```
int setsockopt(
   int         s,
   int         level,
   int         optname,
   const void* optval,
   socklen_t   optlen)
```

The options that are anticipated to be available are:

- `TCP_NODELAY` – Disable the Nagle algorithm.
- `SO_KEEPALIVE` – Enable liveness pinging.

See also:

- man(2) – setsockopt

shutdown

Shutdown parts of a socket.

```
int shutdown(int s, int how)
```

Shutdown all or part of the socket.

The socket is shutdown based on the how parameter which may be one of:

- `SHUT_RD` – No further receives are allowed.
- `SHUT_WR` – No further writes are allowed.
- `SHUT_RDWR` – No further reads or writes are allowed.

See also:

- man(2) – shutdown

socket

Create a new socket for the specific domain, type and protocol.

```
int socket(int domain, int type, int protocol)
```

Domain can be one of:

- `AF_INET` – TCP/IP
- Others …

Type can be one of:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

Protocol can be one of:

- IPPROTO_IP
- IPPROTO_TCP
- IPPROTO_UDP

A common usage pattern is:

```
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
```

Returns a new socket descriptor or a value < 0 on error.

Includes:

- #include <lwip/sockets.h>

See also:

- TCP/IP Sockets
- man(2) – socket

write

```
ssize_t write(int s, const void* dataptr, size_t size)
```

See also:

- man(2) – write

writev

See also:

- man(2) – writev

Socket data structures

Sockets - struct sockaddr

Sockets - struct sockaddr_in

- sin_family – AF_INET
- sin_port
- `struct in_addr sin_addr` – This structure has a member called `s_addr` which is an IP address. Special values have special meanings. For example `INADDR_ANY` is any address.

Working with WiFi

In the ESP32 environment, we include the header `<esp_wifi.h>` which provides the signatures for the WiFi functions we will be using. Prior to working with any WiFi components, we want to call `esp_wifi_init()`. This function initializes the WiFi subsystem. Internally it will allocate resources. We have a partner function called `esp_wifi_deinit()` which will destroy our WiFi environment and release the resources. We should not call any other WiFi functions prior to `esp_wifi_init()` and call no further WiFi functions following an `esp_wifi_deinit()`. There is a mandatory parameter to the `esp_wifi_init()` which is a pointer to an instance of `wifi_init_config_t`. We can supply a useful default to this call using the `WIFI_INIT_CONFIG_DEFAULT` macro.

```
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
err = esp_wifi_init(&config);
```

Since an ESP32 can be simultaneously both a station and an access point, that means it has two interfaces. Two interfaces means two distinct mac addresses. We can set or get the values for these 6 byte mac addresses using `esp_wifi_set_mac()` and `esp_wifi_get_mac()`. The first parameter of these calls defines which interface we wish to interact. The value can either be `WIFI_IF_STA` for the station interface or `WIFI_IF_AP` for the access point interface.

If we are acting as an access point, we can ask what stations are connected to us with a call to `esp_wifi_get_station_list()`. This returns us a list of stations that is based on the ESP32 queuing structure.

Once we have performed an access point scan, the ESP32 has a notion of the access points out there. We can then ask for a copy of this information. We need to understand that each access point is identified by a record of type `wifi_ap_record_t` which is composed of:

```
            uint8_t bssid[6]
            uint8_t ssid[32]
            uint8_t primary
  wifi_second_chan_t second
             int8_t rssi
    wifi_auth_mode_t authmode
```

When we call `esp_wifi_get_station_list()` we pass in a pointer to storage which will be populated by these records. We also pass in the maximum number of records we will accept. On return, we are told how many records were populated. We are managing the storage for these records and ESP32 neither allocates nor releases this data.

For example:

```
wifi_ap_record_t apList[10];
uint16_t count = 10;
esp_wifi_get_station_list(&count, apList);
printf("Number of stations populated = %d\n", count);
```

DNS

The DNS component of ESP32 is provided by the underlying LWIP implementation.

See also:

- lwip – DNS – Doxygen

dns_getserver

Retrieve one of the defined DNS servers.

```
const ip_addr_t dns_getserver(u8_t numdns)
```

- `numdns` – The index of the defined DNS server. ESP32 defined 0 and 1.

Includes:

- #include <lwip/dns.h>

See also:

- gethostbyname
- lwip – DNS – Doxygen

dns_setserver

Set one of the defined DNS servers.

```
void dns_setserver(u8_t numdns, const ip_addr_t *dnsserver)
```

- `numdns` – The index of the defined DNS server. ESP32 defines 0 and 1.

Here is an example of usage:

```
ip_addr_t dnsserver;
inet_pton(AF_INET, "8.8.8.8", &dnsserver);
dns_setserver(0, &dnsserver);
inet_pton(AF_INET, "8.8.4.4", &dnsserver);
dns_setserver(1, &dnsserver);
```

Includes:

- #include <lwip/dns.h>

See also:

- gethostbyname
- lwip – DNS – Doxygen

System Functions

esp_chip_info

Retrieve information about this instance of the ESP32.

```
void esp_chip_info(esp_chip_info_t *outInfo)
```

The `outInfo` is a structure which contains:

- `esp_chip_model_t model` – Which model of chip are we?
  - `CHIP_ESP32` – We are an ESP32
- `uint32_t features` – A bit mask describing a set of features:
  - `CHIP_FEATURE_EMB_FLASH` – Do we have embedded flash?
  - `CHIP_FEATURE_WIFI_BGN` – Do we have Wifi?
  - `CHIP_FEATURE_BLE` – Do we have BLE?

  - `CHIP_FEATURE_BT` – Do we have bluetooth?
- `uint8_t cores` – The number of cores present on the device.
- `uint8_t revision` – The revision of the device.

Includes:

- #include <esp_system.h>

esp_cpu_in_ocd_debug_mode

Determine if a JTAG debugger is attached to CPU.

```
bool esp_cpu_in_ocd_debug_mode()
```

esp_efuse_read_mac

```
esp_err_t system_efuse_read_mac(uint8_t mac[6])
```

esp_get_free_heap_size

Get the size of the available memory heap.

```
uint32_t esp_get_free_heap_size()
```

For example 40544.

As a test I wrote an app that logged the free heap size after boot. This is as of 2016-10-05. The result was 195568.

Includes:

- #include <esp_system.h>

See also:

- RAM Utilization

esp_get_idf_version

Return a representation of the ESP-IDF version against which the application was compiled.

```
const char *esp_get_idf_version()
```

An example of the return might be:

```
v2.0-rc1-930-g058eb26
```

Includes:

- #include <esp_system.h>

esp_random

Get a hardware based random number.

```
uint32_t esp_random()
```

Includes

- #include <esp_system.h>

esp_restart

Restart the system.

```
void esp_restart()
```

Includes

- #include <esp_system.h>

system_rtc_mem_write

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

Includes:

- #include <esp_system.h>

rtc_get_reset_reason

Get the reset reason for a CPU.

```
RESET_REASON rtc_get_reset_reason(int cpuNo)
```

The `RESET_REASON` that is returned may be one of:

- `NO_MEAN` –
- `POWERON_RESET` –
- `SW_RESET` –
- `OWDT_RESET` –
- `DEEPSLEEP_RESET` –
- `SDIO_RESET` –
- `TG0WDT_SYS_RESET` –

- TG1WDT_SYS_RESET –
- RTCWDT_SYS_RESET –
- INTRUSION_RESET –
- TGWDT_CPU_RESET –
- SW_CPU_RESET –
- RTCWDT_CPU_RESET –
- EXT_CPU_RESET –
- RTCWDT_BROWN_OUT_RESET –
- RTCWDT_RTC_RESET –

software_reset

```
void software_reset()
```

Includes:

- #include <rom/rtc.h>

software_reset_cpu

```
void software_reset_cpu(int cpuNo)
```

The CPU numbers supplied by `cpuNo` are 0 for the PRO CPU and 1 for the APP CPU.

Includes:

- #include <rom/rtc.h>

system_deep_sleep

Puts the device to sleep for a period of time.

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

The device goes to sleep and when it awakes, it will start at the user_init location.

Includes

- #include <esp_system.h>

system_get_time

Note: `system_get_time()` has been deprecated. Use `gettimeofday()` as a replacement.

Get the system time measured in microseconds since last device start-up.

```
uint32_t system_get_time()
```

This timer will roll over after 71 minutes.

Note: Experimentation seems to show that as of 2016-10, the timer does **not** tick until after a call to `esp_wifi_init()`.

Includes:

- #include <esp_system.h>

See also:

- Timers and time
- gettimeofday

system_restore

Reset some system settings to defaults.

```
void system_restore()
```

The settings returned to defaults include:

- wifi_station_set_auto_connect
- wifi_set_phy_mode
- wifi_softap_set_config
- wifi_station_set_config
- wifi_set_opmode

Includes

- #include <esp_system.h>

system_rtc_mem_read

Read data from RTC available storage.

Includes:

- #include <esp_system.h>

system_rtc_mem_write

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

Includes:

- #include <esp_system.h>

system_rtc_mem_read

Read data from RTC available storage.

Includes:

- #include <esp_system.h>

WiFi

The WiFi function provide access to the WiFi capabilities of the device.

See also:

esp_event_loop_init

Initialize the WiFi event loop processing and specify an event handler.

```
esp_err_t esp_event_loop_init(system_event_cb_t cb, void* ctx)
```

Initialize the WiFi event loop processing.

The callback function for the event handler is defined as:

```
esp_err_t (*system_event_cb_t)(void* ctx, system_event_t* event)
```

See also:

- Handling WiFi events
- esp_event_loop_set_cb

esp_event_loop_set_cb

Change the event handler used for WiFi events.

```
system_event_cb_t esp_event_loop_set_cb(system_event_cb_t cb, void* ctx)
```

Specify an event handler to be invoked when a WiFi event occurs. The previous event handler is returned.

The callback function for the event handler is defined as:

```
esp_err_t (*system_event_cb_t)(void* ctx, system_event_t* event)
```

Includes:

- #include <esp_event_loop.h>

See also:

- Handling WiFi events
- esp_event_loop_init

esp_wifi_ap_get_sta_list

```
esp_err_t esp_wifi_ap_get_sta_list(wifi_sta_list_t* sta)
```

esp_wifi_clear_fast_connect

Not implemented.

```
esp_err_t esp_wifi_clear_fast_connect()
```

esp_wifi_connect

Connect to an access point.

```
esp_err_t esp_wifi_connect()
```

Prior to calling this function we should have:

- Initialized WiFi – `esp_wifi_init()`
- Set our mode to be either a station or station+ap – `esp_wifi_set_mode()`
- Set our desired access point connection information – `esp_wifi_set_config()`
- Started the WiFi subsystem – `esp_wifi_start()`

We can disconnect from an access point by calling `esp_wifi_disconnect()`. Experience seems to show that before we can connect to a new access point, we must first explicitly disconnect from a previous access point.

Includes:

- #include <esp_wifi.h>

See also:

- Connecting to an access point
- esp_wifi_init
- esp_wifi_set_mode
- esp_wifi_set_config
- esp_wifi_start
- esp_wifi_disconnect

esp_wifi_deauth_sta

```
esp_err_t esp_wifi_deauth_sta(uint16_t aid)
```

esp_wifi_deinit

Release the ESP32 WiFi environment.

```
esp_err_t esp_wifi_deinit()
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_init

esp_wifi_disconnect

Disconnect from an access point.

```
esp_err_t esp_wifi_disconnect()
```

We assume that we have previously connected to an access point using `esp_wifi_connect()`.

Includes:

- #include <esp_wifi.h>

See also:

- Connecting to an access point
- esp_wifi_connect

esp_wifi_free_station_list

Release the storage for the previously returned list of stations.

```
esp_err_t esp_wifi_free_station_list()
```

Includes:

- #include <esp_wifi.h>

See also:

- Working with connected stations
- esp_wifi_get_station_list

esp_wifi_get_auto_connect

Determine whether or not auto connect at boot is enabled.

```
esp_err_t esp_wifi_set_auto_connect(bool *enabled)
```

Includes:

- esp_wifi.h

See also:

- Connecting to an access point
- esp_wifi_set_auto_connect

esp_wifi_get_bandwidth

Get the current bandwidth.

```
esp_err_t esp_wifi_get_bandwidth(wifi_interface_t ifx, wifi_bandwidth_t *bandWidth)
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_bandwidth

esp_wifi_get_channel

Get the current channel.

```
esp_err_t esp_wifi_get_channel(uint8_t *primary, wifi_second_chan_t *second)
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_channel

esp_wifi_get_config

Retrieve the current connection information associated with the specified WiFi interface.

```
esp_err_t esp_wifi_get_config(wifi_interface_t interface, wifi_config_t *conf)
```

The `interface` is one of:

- `WIFI_IF_STA` – The station interface.
- `WIFI_IF_AP` – The access point interface.

The `conf` parameter is populated with the current configuration. See the details of `esp_wifi_set_config` for the nature and content of this data type.

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_config

esp_wifi_get_country

Retrieve the currently configured WiFi country.

```
esp_err_t esp_wifi_get_country(wifi_country_t *country)
```

The default WiFi country is China. Allowable values are:

- `WIFI_COUNTRY_CN`
- `WIFI_COUNTRY_JP`
- `WIFI_COUNTRY_US`
- `WIFI_COUNTRY_EU`

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_country

esp_wifi_get_mac

Retrieve the current MAC address for the interface.

```
esp_err_t esp_wifi_get_mac(wifi_interface_t interface, uint8_t mac[6])
```

The `interface` is one of:

- `WIFI_IF_STA` – The station interface.
- `WIFI_IF_AP` – The access point interface.

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_mac

esp_wifi_get_mode

Get the WiFi operating mode.

```
esp_err_t esp_wifi_get_mode(wifi_mode_t *mode)
```

Get the operating WiFi mode. The choices available are:

- `WIFI_MODE_NULL` – No WiFi.
- `WIFI_MODE_STA` – A station.
- `WIFI_MODE_AP` – An access point.
- `WIFI_MODE_APSTA` – Both a station and an access point.

Includes:

- #include <esp_wifi.h>

See also:

- Setting the operation mode
- Station configuration
- esp_wifi_set_mode
- esp_wifi_set_config

esp_wifi_get_promiscuous

```
esp_err_t esp_wifi_get_promiscuous(uint8_t *enable)
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_promiscuous

esp_wifi_get_protocol

Get the 802.11 protocol (b/g/n).

```
esp_err_t esp_wifi_get_protocol(wifi_interface_t ifx, uint8_t *protocolBitmap)
```

The possible protocols are:

- WIFI_PROTOCOL_11B
- WIFI_PROTOCOL_11G
- WIFI_PROTOCOL_11N
- WIFI_PROTOCOL_LR

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_protocol

esp_wifi_get_ps

Get the power save type.

```
esp_err_t esp_wifi_get_ps(wifi_ps_type_ t *type)
```

The type of power save will be one of:

- `WIFI_PS_NONE` –
- `WIFI_PS_MODEM` –
- `WIFI_PS_LIGHT` –
- `WIFI_PS_MAC` –

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_set_ps

esp_wifi_get_station_list

Get the list of stations connected to ESP32 when it is behaving as an access point.

```
esp_err_t esp_wifi_get_station_list(wifi_sta_list_t **station)
```

The structure of a "`struct station_info`" is:

```
  STAILQ_ENTRY(station_info) next
                     uint8_t bssid[6]
```

To walk through this list we can use the following algorithm:

```
wifi_sta_list_t *stations;
//
while (stations != NULL) {
   printf("bssid: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
      stations->bssid[0],stations->bssid[1],stations->bssid[2],
      stations->bssid[3],stations->bssid[4],stations->bssid[5]);
   stations = STAILQ_NEXT(stations, next);
}
```

Includes:

- #include <esp_wifi.h>

See also:

- Working with connected stations
- esp_wifi_free_station_list

- tcpip_adapter_get_sta_list

esp_wifi_init

Initialize the ESP32 WiFi environment.

```
esp_err_t esp_wifi_init(wifi_init_config_t *config)
```

This API call should be invoked before all other WiFi related calls. The `wifi_init_config_t` contains:

```
  void * event_q
 uint8_t rx_ba_win
 uint8_t rx_ba_win
 uint8_t rx_ba_win
```

A macro called `WIFI_INIT_CONFIG_DEFAULT` can be used to initialize the configuration structure. For example:

```
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
esp_wifi_init(&config);
```

Includes:

- #include <esp_wifi.h>

See also:

- Initializing the WiFi environment
- esp_wifi_start
- esp_wifi_deinit

esp_wifi_restore

```
esp_err_t esp_wifi_restore()
```

esp_wifi_reg_rxcb

```
esp_err_t esp_wifi_reg_rxcb(wifi_interface_t ifx, wifi_rxcb_t fn)
```

Includes:

- #include <esp_wifi.h>

esp_wifi_scan_get_ap_records

Retrieve the access points found in a previous scan.

```
esp_err_t esp_wifi_get_ap_records(uint16_t *number, wifi_ap_record_t *apRecords);
```

The `apList` is a contiguous chunk of storage capable of holding objects of type `wifi_ap_record_t`. The number of such records is the initial value of the number parameter. On return, the actual number of items will be updated. A `wifi_ap_record_t` record contains:

```
           uint8_t bssid[6]
           uint8_t ssid[32]
           uint8_t primary
wifi_second_chan_t second
            int8_t rssi
  wifi_auth_mode_t authmode
```

The `authmode` can be one of:

- `WIFI_AUTH_OPEN` –
- `WIFI_AUTH_WEP` –
- `WIFI_AUTH_WPA_PSK` –
- `WIFI_AUTH_WPA2_PSK` –
- `WIFI_AUTH_WPA_WPA2_PSK` –

Includes:

- #include <esp_wifi.h>

See also:

- Scanning for access points
- esp_wifi_scan_stop

esp_wifi_scan_get_ap_num

Retrieve the count of found access points from a previous scan.

```
esp_err_t esp_wifi_scan_get_ap_num(uint16_t *number)
```

Retrieve the number of discovered access points from the previous scan. We need to be careful that the scan has completed before getting the count.

Includes:

- #include <esp_wifi.h>

See also:

- Scanning for access points

esp_wifi_scan_start

Scan for access points.

```
esp_err_t esp_wifi_scan_start(wifi_scan_config_t *conf, bool block)
```

Govern how the scan should be performed. The `wifi_scan_config_t` contains the following fields:

```
     char * ssid
  uint8_t * bssid
    uint8_t channel
       bool show_hidden
```

The `block` parameter defines whether or not this call blocks until the data is available.

The results of a WiFi scan are stored internally in ESP32 dynamically allocated storage. The data is returned to us when we call `esp_wifi_get_ap_list()` which also releases the internally allocated storage. As such, this should be considered a destructive read.

Includes:

- #include <esp_wifi.h>

See also:

- Scanning for access points
- esp_wifi_scan_stop

esp_wifi_scan_stop

Stop an access point scan that is in progress.

```
esp_err_ t esp_wifi_scan_stop()
```

By calling `esp_wifi_scan_start()`, we can request that a WiFi scan be performed in the background. Should we wish to interrupt or stop that activity, this function can be used.

Includes:

- #include <esp_wifi.h>

See also:

- Scanning for access points
- esp_wifi_scan_start
- Error: Reference source not found
- esp_wifi_set_config

esp_wifi_set_auto_connect

```
esp_err_t esp_wifi_set_auto_connect(bool enabled)
```

Includes:

- #include <esp_wifi.h>

See also:

- Connecting to an access point
- esp_wifi_get_auto_connect

esp_wifi_set_bandwidth

Set the current bandwidth.

```
esp_err_t esp_wifi_set_bandwidth(
   wifi_interface_t interface,
   wifi_bandwidth_t bandWidth)
```

The interface is one of:

- `WIFI_IF_STA` – The station interface.
- `WIFI_IF_AP` – The access point interface.

The `bandWidth` parameter can be one of:

- `WIFI_BW_HT20` –
- `WIFI_BW_HT40` –

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_bandwidth

esp_wifi_set_channel

```
esp_err_t esp_wifi_set_channel(uint8_t primary, wifi_second_chan_t second)
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_channel

esp_wifi_set_config

Set the WiFi interface configuration.

```
esp_err_t esp_wifi_set_config(
   wifi_interface_t interface,
   wifi_config_t *conf)
```

The interface is one of:

- `WIFI_IF_STA` – The station interface.
- `WIFI_IF_AP` – The access point interface.

The choice of which interface will be used will be dependent on whether or not we are being an access point, a station or both. If a station, we will configure the station interface, if an access point, we will configure the access point interface and if both, then we will configure both interfaces. We should previously have called `esp_wifi_set_mode()`.

The `wifi_config_t` defines properties of the interface. It is a C language union of `wifi_ap_config_t` and `wifi_sta_config_t`.

The `wifi_sta_config_t` contains:

```
   char ssid[32]
   char password[64]
   bool bssid_set
uint8_t bssid[6]
```

An example initialization for this structure might be:

```
wifi_config_t staConfig = {
   .sta = {
      .ssid="<access point name>",
      .password="<password>",
      .bssid_set=false
   }
};
```

The `wifi_ap_config_t` contains:

```
            char ssid[32]
            char password[64]
         uint8_t ssid_len
         uint8_t channel
wifi_auth_mode_t authmode
         uint8_t ssid_hidden
         uint8_t max_connection
        uint16_t beacon_interval
```

If `ssid_len` is `0`, then look for a string termination character in the `ssid` field. Otherwise if `ssid_len` is greater than `0`, the value defines the number of bytes in `ssid` to read for the ssid value.

The `channel` is the channel we are using for WiFi.

The `authmode` indicates how stations can connect. Options include:

- `WIFI_AUTH_OPEN`
- `WIFI_AUTH_WEP`
- `WIFI_AUTH_WPA_PSK`
- `WIFI_AUTH_WPA2_PSK`
- `WIFI_AUTH_WPA_WPA2_PSK`

The `ssid_hidden` is 0 meaning that the SSID is broadcast and can be found.

The `max_connection` is the maximum number of stations that can connect. The default is 4.

The `beacon_interval` is some magic related to WiFi and should have a default value of 100.

An example of initialization of this structure might be:

```
wifi_config_t apConfig = {
   .ap = {
      .ssid="<access point name>",
      .password="<password>",
      .ssid_len=0,
      .channel=0,
      .authmode=WIFI_AUTH_OPEN,
      .ssid_hidden=0,
      .max_connection=4,
      .beacon_interval=100
   }
};
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_config
- Being an access point
- Station configuration

esp_wifi_set_country

Set the WiFi country.

```
esp_err_t esp_wifi_set_country(wifi_country_t country)
```

The default WiFi country is China. Allowable values are:

- `WIFI_COUNTRY_CN` – China.
- `WIFI_COUNTRY_JP` – Japan.
- `WIFI_COUNTRY_US` – United States of America.
- `WIFI_COUNTRY_EU` – European Union.

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_country

esp_wifi_set_mac

```
esp_err_t esp_wifi_set_mac(wifi_interface_t ifx, uint8_t mac[6])
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_mac

esp_wifi_set_mode

Set the operating mode.

```
esp_err_t esp_wifi_set_mode(wifi_mode_t mode)
```

Set the operating WiFi mode. The choices are:

- `WIFI_MODE_NULL` – No WiFi.

- `WIFI_MODE_STA` – A station.
- `WIFI_MODE_AP` – An access point.
- `WIFI_MODE_APSTA` – Both a station and an access point.

You will also need to call `esp_wifi_set_config()` to specify the configuration parameters. If we are being an access point, we will not actually start listening for stations until after `esp_wifi_start()` and if we are being a station, we will not connect to an access point until after a call to `esp_wifi_start()` and then `esp_wifi_connect()`.

Includes:

- #include <esp_wifi.h>

See also:

- Setting the operation mode
- Station configuration
- esp_wifi_get_mode
- esp_wifi_set_config

esp_wifi_set_promiscuous_rx_cb

```
esp_err_t esp_wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)
```

Includes:

- #include <esp_wifi.h>

See also:

esp_wifi_set_promiscuous

```
esp_err_t esp_wifi_set_promiscuous(uint8_t enable)
```

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_promiscuous

esp_wifi_set_protocol

Set the 802.11 protocol (b/g/n)

```
esp_err_t esp_wifi_set_protocol(wifi_interface_t ifx, uint8_t protocolBitmap)
```

The allowable protocols are:

- WIFI_PROTOCOL_11B

- WIFI_PROTOCOL_11G
- WIFI_PROTOCOL_11N
- WIFI_PROTOCOL_LR

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_protocol

esp_wifi_set_ps

Set the power save type.

```
esp_err_t esp_wifi_set_ps(wifi_ps_type_t type)
```

The type of power save will be one of:

- `WIFI_PS_NONE` –
- `WIFI_PS_MODEM` –
- `WIFI_PS_LIGHT` –
- `WIFI_PS_MAC` –

Includes:

- #include <esp_wifi.h>

See also:

- esp_wifi_get_ps

esp_wifi_set_storage

Define where WiFi configuration will be stored.

```
esp_err_t esp_wifi_set_storage(wifi_storage_t storage)
```

The options available are:

- `WIFI_STORAGE_FLASH` – Configuration stored in both flash and RAM. This is the default.
- `WIFI_STORAGE_RAM` – Configuration stored in RAM only (not in flash as well).

The value of this method is that we can choose to save our configuration set by `esp_wifi_set_config()` to RAM or both flash and RAM. If we save our settings in flash then it is these settings that will be used during an auto connect when we call `esp_wifi_set_auto_connect()` to connect at start up.

Includes:

- #include <esp_wifi.h>

See also:

- Connecting to an access point
- esp_wifi_set_config
- esp_wifi_set_auto_connect

esp_wifi_set_vendor_ie

```
esp_err_t esp_wifi_set_vendor_ie(
   bool enable,
   wifi_vendor_ie_type_t type,
   wifi_vendor_ie_id_t idx,
   uint8_t *vnd_ie)
```

Includes:

- #include <esp_wifi.h>

esp_wifi_set_vendor_ie_cb

```
esp_err_t esp_wifi_set_vendor_ie_cb(
   esp_vendor_ie_cb_t cb,
   void *ctx)
```

Includes:

- #include <esp_wifi.h>

esp_wifi_sta_get_ap_info

Get information about the access point to which we are connected,

```
esp_err_t esp_wifi_sta_get_ap_info(wifi_ap_record_t *apInfo)
```

esp_wifi_start

Start the WiFi subsystem.

```
esp_err_t esp_wifi_start()
```

Prior to calling this function, we should have called `esp_wifi_init()` and configured our mode (`esp_wifi_set_mode`) and interfaces (`esp_wifi_set_config`). If we are an access point, then after calling this function, we will start to accept incoming client connections.

Includes:

- #include <esp_wifi.h>

See also:

- Starting up the WiFi environment
- esp_wifi_stop
- esp_wifi_init
- esp_wifi_set_mode

esp_wifi_stop

Stop the WiFi subsystem.

```
esp_err_t esp_wifi_stop()
```

This will stop the ESP32 WiFi subsystem. This could be called after a previous `esp_wifi_start()` call.

Includes:

- #include <esp_wifi.h>

See also:

- Starting up the WiFi environment
- esp_wifi_start

WiFi WPS

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

mbed TLS

The mbed TLS is an implementation of SSL/TLS security and is supplied with ESP-IDF.

See also:

- TLS, SSL and security
- mbed TLS home page
- mbed TLS tutorial

mbedtls_ctr_drbg_free

```
mbedtls_ctr_drbg_free(mbedtls_ctr_drbg_context *ctx)
```

mbedtls_ctr_drbg_init

```
void mbedtls_ctr_drbg_init(mbedtls_ctr_drbg_context *ctx)
```

mbedtls_ctr_drbg_seed

Initial seeding.

```
int mbedtls_ctr_drbg_seed(
   mbedtls_ctr_drbg_context *ctx,
   int(*)(void *, unsigned char *, size_t) f_entropy,
   void *p_entropy,
   const unsigned char *custom,
   size_t len)
```

Includes:

- mbed/ctr_drbg.h

mbedtls_debug_set_threshold

Set the debug level.

```
void mbedtls_debug_set_threshold(int threshold)
```

Set the level of debugging to be generated from the library. The threshold value can be one of:

- 0 – no debug
- 1 – error
- 2 – state change
- 3 – informational
- 4 – verbose

A higher level captures that level and the entries of lower level.

Note: To use this function, mbedTLS debugging must be enabled by editing the Curl mbedtls.c.

Includes:

- mbedtls/debug.h

See also:

- mbedtls_ssl_conf_dbg
- mbedtls_strerror

mbedtls_entropy_free

```
void mbedtls_entropy_free(mbedtls_entropy_context *ctx)
```

Includes:

- mbed/entropy.h

mbedtls_entropy_init

```
void mbedtls_entropy_init(mbedtls_entropy_context *ctx)
```

Includes:

- mbed/entropy.h

mbedtls_net_accept

```
int mbedtls_net_accept(
   mbedtls_net_context *bind_ctx,
   mbedtls_net_context *client_ctx,
   void *client_ip,
   size_t buf_size,
   size_t *ip_len)
```

mbedtls_net_bind

```
int mbedtls_net_bind(
   mbedtls_net_context *ctx,
   const char *bind_ip,
   const char *port,
   int proto)
```

The `proto` is one of `MBEDTLS_NET_PROTO_TCP` or `MBEDTLS_NET_PROTO_UDP`.

mbedtls_net_connect

Initiate a network connection.

```
int mbedtls_net_connect(
   mbedtls_net_context *context,
   const char *host,
   const char *port,
   int proto)
```

The `context` is the network context to use, `host` is the host to connect to, `port` is the port number (as a string) and `proto` is one of `MBEDTLS_NET_PROTO_TCP` or `MBEDTLS_NET_PROTO_UDP`.

Includes:

- mbed/net.h

See also:

- mbedtls_net_init

mbedtls_net_free

Gracefully shutdown the connection and release data.

```
void mbedtls_net_free(mbedtls_net_context *context)
```

Release any resources associated with the context. A call to `mbedtls_net_init()` will have been called prior to this release call.

See also:

- mbedtls_net_init

mbedtls_net_init

Initialize a context for use.

```
void mbedtls_net_init(mbedtls_net_context *context)
```

Invoke this function early to initialize the context. Don't try and use a context before calling this function.

Includes:

- mbed/net.h

See also:

- mbedtls_net_free

mbedtls_net_recv

Read data from a socket.

```
int mbedtls_net_recv(
   void *ctx,
   unsigned char *buf,
   size_t len
)
```

mbedtls_net_recv_timeout

```
int mbedtls_net_recv_timeout(
   void *ctx,
   unsigned char *buf,
   size_t len,
   uint32_t timeout
)
```

mbedtls_net_send

Send data down a socket.

```
int mbedtls_net_send(
   void *ctx,
   const unsigned char *buf,
   size_t len)
```

mbedtls_net_set_block

```
int mbedtls_net_set_block(mbedtls_net_context *ctx)
```

mbedtls_net_set_nonblock

```
int mbedtls_net_set_nonblock(mbedtls_net_context *ctx)
```

mbedtls_printf

Printf macro.

Includes:

- mbed/platform.h

mbedtls_sha1

Produce a 160bit (20 byte) hash value.

```
void mbedtls_sha1(const unsigned char *input, size_t ilen, unsigned char output[20])
```

The `input` is the pointer to the data to be hashed. The `ilen` is the length of the input. The `output` is the resulting hash.

Includes:

- mbedtls/sha1.h

mbedtls_ssl_close_notify

mbedtls_ssl_conf_authmode

Set the certificate verification mode.

```
void mbedtls_ssl_conf_authmode(
   mbedtls_ssl_config *sslConf,
   int authmode)
```

The `sslConf` is the SSL configuration. The `authmode` is one of:

- `MBEDTLS_SSL_VERIFY_NONE` –
- `MBEDTLS_SSL_VERIFY_OPTIONAL` –
- `MBEDTLS_SSL_VERIFY_REQUIRED` –

Includes:

- mbed/ssl.h

mbedtls_ssl_conf_ca_chain

mbedtls_ssl_conf_dbg

Set the debug callback.

```
void mbedtls_ssl_conf_dbg(
   mbedtls_ssl_config *conf,
   void(*func)(void *debugContext, int level, char *file, int line, const char *str),
   void *debugContext)
```

An example of a function that can be used to log data might be:

```
static void my_debug(
   void *ctx,
   int level,
   const char *file,
   int line,
   const char *str) {

   ((void) level);
   ((void) ctx);
   printf("%s:%04d: %s", file, line, str);
}
```

See also:

- mbedtls_debug_set_threshold
- mbedtls_strerror

mbedtls_ssl_conf_rng

Set the random number generator callback.

```
void mbedtls_ssl_conf_rng(
   mbedtls_ssl_config conf,
   int(*)(void *, unsigned char *, size_t) f_rng,
   void *p_rng)
```

Includes:

- mbed/ssl.h

mbedtls_ssl_config_defaults

```
int mbedtls_ssl_config_defaults(
   mbedtls_ssl_config *sslConfig,
   int endpoint,
   int transport,
   int preset)
```

The `endpoint` is one of `MBEDTLS_SSL_IS_CLIENT` or `MBEDTLS_SSL_IS_SERVER`.

The `transport` is `MBEDTLS_SSL_TRANSPORT_STREAM` or `MBEDTLS_SSL_TRANSPORT_DATAGRAM`.

The `preset` is currently not used. Supply `MBEDTLS_SSL_PRESET_DEFAULT`.

Includes:

- mbed/ssl.h

mbedtls_ssl_config_free

Release the resources for an SSL configuration context.

```
void mbedtls_ssl_config_free(mbedtls_ssl_config *sslConf)
```

Includes:

- mbed/ssl.h

See also:

- mbedtls_ssl_config_init

mbedtls_ssl_config_init

Initialize an SSL Config record.

```
void mbedtls_ssl_config_init(mbedtls_ssl_config *sslConf)
```

Includes:

- mbed/ssl.h

See also:

- mbedtls_ssl_config_free

mbedtls_ssl_free

Release resources for an SSL context.

```
void mbedtls_ssl_free(mbedtls_ssl_context *sslContext)
```

Release any resources that may have been previously allocated by a call to `mbedtls_ssl_init()`.

Includes:

- mbed/ssl.h

See also:

- mbedtls_ssl_init

mbedtls_ssl_get_verify_result

mbedtls_ssl_handshake

mbedtls_ssl_init

Initialze an SSL context.

```
void mbedtls_ssl_init(mbedtls_ssl_context *sslContext)
```

The data structure is a black box and this function initializes it for us.

Includes:

- mbed/ssl.h

See also:

- mbedtls_ssl_free

mbedtls_ssl_read

```
int mbedtls_ssl_read(
   mbedtls_ssl_context *sslContext,
   unsigned char *buf,
   size_t len)
```

mbedtls_ssl_session_reset

mbedtls_ssl_set_bio

Identify the functions to be used for sending and receiving data.

```
void mbedtls_ssl_set_bio(
   mbedtls_ssl_context *context,
   void *p_bio,
   mbedtls_ssl_send_t *f_send,
   mbedtls_ssl_recv_t *f_recv,
   mbedtls_ssl_recv_timeout_t *f_recv_timeout)
```

The context is the SSL context we are currently using for this communication.

The p_bio is a pointer to storage that is passed into the send and receive functions to give them context.

The `f_send` is a function to be called to transmit SSL encrypted data. It can be `mbedtls_net_send`. The signature for this function is:

```
int functionName(void *ctx, const unsigned char *buf, size_t len)
```

The `f_recv` is a function to be called to receive SSL encrypted data. It can be `mbedtls_net_recv`. The signature for this function is:

```
int functionName(void *ctx, unsigned char *buf, size_t len)
```

The `f_recv_timeout` is a function to be called to receive SSL encrypted data with a timeout. It can be `NULL`. The signature of this function is:

```
int functionName(void *ctx, unsigned char *buf, size_t len, uint32_t timeout)
```

mbedtls_ssl_set_hostname

Set the hostname to check against.

```
int mbedtls_ssl_set_hostname(
   mbedtls_ssl_context *context,
   const char *hostname)
```

Returns 0 on success.

Includes:

- mbed/ssl.h

mbedtls_ssl_setup

Setup an SSL context for use.

```
int mbedtls_ssl_setup(
   mbedtls_ssl_context *sslContext,
   mbedtls_ssl_config *sslConfig)
```

Includes:

- mbed/ssl.h

mbedtls_ssl_write

Write a buffer of data down the socket.

```
int mbedtls_ssl_write(
   mbedtls_ssl_context *ssl,
   const unsigned char *buf,
   size_t len)
```

The number of bytes written or < 0 on error.

Includes:

- mbed/ssl.h

mbedtls_strerror

```
void mbedtls_strerror(
   int errnum,
   char *buffer,
   size_t bufflen)
```

Convert an error code into a C string representing the error.

Includes:

- mbed/error.h

See also:

- mbedtls_debug_set_threshold
- mbedtls_ssl_conf_dbg

mbedtls_x509_crt_init

Initialize a certificate chain.

```
void mbedtls_x509_crt_init(mbedtls_x509_crt *crt)
```

mbedtls_x509_crt_parse

mbedtls_x509_crt_veryify_info

Bluetooth LE

See also:

- Bluetooth

esp_bt_uuid_t

A representation of a UUID.

- `uint16_t len` – The length of the UUID
  - `ESP_UUID_LEN_16` – 16 bits in length.
  - `ESP_UUID_LEN_32` – 32 bits in length.
  - `ESP_UUID_LEN_128` – 128 bits in length.
- union uuid
  - `uint16_t uuid16` – Data for a 16 bit uuid.
  - `uuid32_t uuid32` – Data for a 32 bit uuid.
  - `uint8_t uuid128[ESP_UUID_LEN_128]` – Data for a 128 bit uuid.

For the 128 bit UUID, you need to take care that you realize that the data is in little endian format. Putting it another way, the data is Least Significant Byte First.

For example, if your UUID is `12345678-90ab-cdef-1234-567890abcdef`, the you would store this in memory as

```
0xef 0xcd 0xab 0x90 0x78 0x56 0x 34 0x12 0xef 0xcd 0xab 0x90 0x78 0x56 0x34
0x12
```

Includes:

- #include <esp_gatt_defs.h>

esp_attr_value_t

Characteristic value.

- uint16_t attr_max_len
- uint16_t attr_len
- uint8_t *attr_value

esp_gatt_id_t

The description of a characteristic.

This data type describes a characteristic.

- esp_bt_uuid_t uuid
- uint8_t inst_id

esp_gatt_srvc_id_t

The description of a service.

- `esp_gatt_id_t id` – The identity of the service.
  - `esp_bt_uuid uuid` – The UUID of the service.
  - `uint8_t inst_id` – The instance index of this service on the server.
- `bool is_primary` – Is this a primary service?

See also:

- esp_gatt_id_t

esp_gatt_status_t

The status code for GATT.

- ESP_GATT_OK
- ESP_GATT_INVALID_HANDLE

- ESP_GATT_READ_NOT_PERMIT
- ESP_GATT_WRITE_NOT_PERMIT
- ESP_GATT_INVALID_PDU
- ESP_GATT_INSUF_AUTHENTICATION
- ESP_GATT_REQ_NOT_SUPPORTED
- ESP_GATT_INVALID_OFFSET
- ESP_GATT_INSUF_AUTHORIZATION
- ESP_GATT_PREPARE_Q_FULL
- ESP_GATT_NOT_FOUND
- ESP_GATT_NOT_LONG
- ESP_GATT_INSUF_KEY_SIZE
- ESP_GATT_INVALID_ATTR_LEN
- ESP_GATT_ERR_UNLIKELY
- ESP_GATT_INSUF_ENCRYPTION
- ESP_GATT_UNSUPPORT_GRP_TYPE
- ESP_GATT_INSUF_RESOURCE
- ESP_GATT_NO_RESOURCES
- ESP_GATT_INTERNAL_ERROR
- ESP_GATT_WRONG_STATE
- ESP_GATT_DB_FULL
- ESP_GATT_BUSY
- ESP_GATT_ERROR
- ESP_GATT_CMD_STARTED
- ESP_GATT_ILLEGAL_PARAMETER
- ESP_GATT_PENDING
- ESP_GATT_AUTH_FAIL
- ESP_GATT_MORE
- ESP_GATT_INVALID_CFG

- ESP_GATT_SERVICE_STARTED
- ESP_GATT_ENCRYPED_MITM
- ESP_GATT_ENCRYPED_NO_MITM
- ESP_GATT_NOT_ENCRYPTED
- ESP_GATT_CONGESTED
- ESP_GATT_DUP_REG
- ESP_GATT_ALREADY_OPEN
- ESP_GATT_CANCEL
- ESP_GATT_STACK_RSP
- ESP_GATT_APP_RSP
- ESP_GATT_UNKNOWN_ERROR
- ESP_GATT_CCC_CFG_ERR
- ESP_GATT_PRC_IN_PROGRESS
- ESP_GATT_OUT_OF_RANGE

esp_ble_resolve_adv_data

```
uint8_t* esp_ble_resolve_adv_data(uint8_t* advData, uint8_t type, uint8_t *length)
```

esp_ble_gap_config_adv_data

Specify the data that we wish to advertise.

```
esp_err_t esp_ble_gap_config_adv_data(esp_ble_adv_data_t *adv_data)
```

The adv_data is a structure including:

- bool set_scan_rsp
- bool include_name
- bool include_txpower
- int min_interval
- int max_interval
- `int appearance` – Indicate how this advertises device should be presented to the user in terms of appearance. Typically this is used to define the icon that represents the device.
- uint16_t manufacturer_len
- uint8_t *p_manufacturer_data

- uint16_t service_data_len
- uint8_t *p_service_data
- `uint16_t service_uuid_len` – The length of the advertised service UUID either 2,4 or 16. Specify 0 if no service UUID is being advertised.
- `uint8_t *p_service_uuid` – Pointer to the storage of the UUID being advertised. Can be NULL if no service UUID is being advertised.
- `uint8_t flag` – The flag is a boolean "or" of the following:
    - `ESP_BLE_ADV_FLAG_LIMIT_DISC` – Limited discovery flag.
    - `ESP_BLE_ADV_FLAG_GEN_DISC` – General Discovery flag.
    - `ESP_BLE_ADV_FLAG_BREDR_NOT_SPT` – Bluetooth standard not supported.
    - ESP_BLE_ADV_FLAG_DMT_CONTROLLER_ST
    - ESP_BLE_ADV_FLAG_DMT_HOST_SPT
    - ESP_BLE_ADV_FLAG_NON_LIMIT_DISC

An example structure might be:

```
static esp_ble_adv_data_t adv_data;
adv_data.set_scan_rsp        = false;
adv_data.include_name        = true;
adv_data.include_txpower     = true;
adv_data.min_interval        = 0x20;
adv_data.max_interval        = 0x40;
adv_data.appearance          = 0x00;
adv_data.manufacturer_len    = 0;
adv_data.p_manufacturer_data = NULL;
adv_data.service_data_len    = 0;
adv_data.p_service_data      = NULL;
adv_data.service_uuid_len    = 0;
adv_data.p_service_uuid      = NULL;
adv_data.flag                = (ESP_BLE_ADV_FLAG_GEN_DISC |
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT);
```

Includes:

- #include <esp_gap_ble_api.h>

See also:

- Performing advertising
- esp_ble_gap_start_advertising
- esp_ble_gap_stop_advertising

esp_ble_gap_config_adv_data_raw

Set the data that is to be sent in advert at the raw level.

```
esp_err_t esp_ble_gap_config_adv_data_raw(
   uint8_t* rawData,
   uint32_t len)
```

esp_ble_gap_config_scan_rsp_data_raw

Set the data that is to be sent in a scan response at the raw level.

```
esp_err_t esp_ble_gap_config_scan_rsp_data_raw(
   uint8_t* rawData,
   uint32_t len)
```

esp_ble_gap_config_local_privacy

```
esp_err_t esp_ble_gap_config_local_privacy(bool privacy_enable)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_gap_register_callback

Register a callback for gap events.

```
esp_err_t esp_ble_gap_register_callback(esp_profile_cb_t callback)
```

While not explicitly stated it is assumed that the ESP-IDF maintains knowledge of just a single callback. Thus if you want multiple functions to be invoked on an event, you would be responsible for implementing a mediator using your own logic.

The `esp_profile_cb_t` is a C function with the following signature:

```
void func(esp_gap_ble_cb_event_t event, esp_ble_gap_cb_param_t *param)
```

The `param` is an an instance of a pointer to an `esp_ble_gap_cb_param_t`. This is a union of structures. These are:

- struct ble_adv_data_cmpl_evt_param adv_data_cmpl
- struct ble_sca_rsp_data_cmpl_evt_param scan_rsp_data_cmpl
- struct ble_scan_param_cmpl_evt_param scan_param_cmpl
- struct ble_scan_result_evt_param scan_rst
- struct ble_adv_data_raw_cmpl_evt_param adv_data_raw_cmpl
- struct ble_adv_start_cmpl_evt_param adv_start_cmpl
- struct ble_scan_start_cmpl_evt_param scan_start_cmpl

- esp_ble_sec_t ble_security
  - esp_ble_sec_key_notif_t key_notif
  - esp_ble_sec_req_t ble_req
  - esp_ble_key_t ble_key
  - esp_ble_local_id_keys_t ble_id_keys
  - esp_ble_auth_cmpl_t auth_cmpl
- struct ble_scan_stop_cmpl_evt_param scan_stop_cmpl
- struct ble_adv_stop_cmpl_evt_param adv_stop_cmpl

Event types include:

ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT

Called when advertising data set is complete. Structure parameter is called `scan_rsp_data_cmpl`.

- esp_bt_status_t status

ESP_GAP_BLE_ADV_START_COMPLETE_EVT

Structure parameter is called `scan_start_cmpl`.

- esp_bt_status_t status

See also:

- esp_ble_gap_start_scanning

ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT

On receipt, the param field called `scan_stop_cmpl` is populated, This is a structure which contains:

- esp_bt_status_t status

ESP_GAP_BLE_AUTH_CMPL_EVT

On receipt, the `param` field called `ble_security` is populated which is itself a union. The field called `auth_cmpl` within it is populated:

- esp_bd_addr_t bd_addr
- bool key_present

- esp_link_key key
- uint8_t key_type
- bool success
- uint8_t fail_reason
- esp_bd_addr_type_t addr_type
- esp_bt_dev_type_t dev_type

ESP_GAP_BLE_KEY_EVT

ESP_GAP_BLE_LOCAL_ER_EVT

ESP_GAP_BLE_LOCAL_IR_EVT

ESP_GAP_BLE_NC_REQ_EVT
Numeric comparison request.

The parameter passed in the event is an instance of `esp_ble_sec_t` called `ble_security`. This is itself a union and the populated field is `key_notif`.

- esp_ble_sec_key_notif_t key_notif
  - esp_bd_addr_t bd_addr
  - uint32_t passkey

ESP_GAP_BLE_OOB_REQ_EVT

ESP_GAP_BLE_PASSKEY_NOTIF_EVT

ESP_GAP_BLE_PASSKEY_REQ_EVT

ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT
Called when scan parameters set complete. Structure parameter is called `scan_param_cmpl`.

- esp_bt_status_t status

See also:

- esp_ble_gap_set_scan_params

ESP_GAP_BLE_SCAN_RESULT_EVT
The param is an instance of `esp_ble_gap_cb_param_t`. Called when one scan result is ready. Structure parameter is called `scan_rst`.

- esp_gap_search_evt_t search_evt – Choices are
  - ESP_GAP_SEARCH_INQ_RES_EVT – We have received a search result.
  - ESP_GAP_SEARCH_INQ_CMPL_EVT – The search is complete.
  - ESP_GAP_SEARCH_DISC_RES_EVT
  - ESP_GAP_SEARCH_DISC_BLE_RES_EVT
  - ESP_GAP_SEARCH_DISC_CMPL_EVT
  - ESP_GAP_SEARCH_DI_DISC_CMPL_EVT
  - ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT
- esp_bd_addr_t bda – The address of the device. 6 bytes of data.
- esp_bt_dev_type_t dev_type – One of:
  - ESP_BT_DEVICE_TYPE_BREDR
  - ESP_BT_DEVICE_TYPE_BLE
  - ESP_BT_DEVICE_TYPE_DUMO
- esp_ble_addr_type_t ble_addr_type – One of
  - BLE_ADDR_TYPE_PUBLIC
  - BLE_ADDR_TYPE_RANDOM
  - BLE_ADDR_TYPE_RPA_PUBLIC
  - BLE_ADDR_TYPE_RPA_RANDOM
- esp_ble_evt_type_t ble_evt_type – One of
  - ESP_BLE_EVT_CONN_ADV
  - ESP_BLE_EVT_CONN_DIR_ADV
  - ESP_BLE_EVT_DISC_ADV
  - ESP_BLE_EVT_NON_CONN_ADV
  - ESP_BLE_EVT_SCAN_RSP
- int rssi – The signal strength.
- uint8_t ble_adv[ESP_BLE_ADV_DATA_LEN_MAX]
- int flag
- int num_resps

    - uint8_t adv_data_len
    - uint8_t scan_rsp_len
- `ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT` – Called when the scan response data set is complete. Structure parameter is called `scan_rsp_data_cmpl`.
    - esp_bt_status_t status

ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT

ESP_GAP_BLE_SCAN_START_COMPLETE_EVT

ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT

The structure associated with this event is `struct ble_scan_stop_cmpl_evt_param` found in the field called `scan_stop_cmpl`. It contains:

- `esp_bt_status_t status` – The status of the event.

ESP_GAP_BLE_SEC_REQ_EVT

BLE Security request.

The parameter passed in the event is an instance of `esp_ble_sec_t` called `ble_security`. This is itself a union and the populated field is `ble_req`. It is expected that a call to `esp_ble_gap_security_rsp()` will be made on receipt.

A union of:

- esp_ble_sec_key_notif_t key_notif
    - esp_bd_addr_t bd_addr
    - uint32_t passkey
- esp_ble_sec_req_t ble_req
    - esp_bd_addr_t bd_addr
- esp_ble_key_t ble_key
    - esp_bd_addr_t bd_addr
    - esp_ble_key_type_t key_type
    - esp_ble_key_value_t p_key_value
- esp_ble_local_id_keys_t ble_id_keys
    - esp_bt_octet16_t ir

    - esp_bt_octet16_t irk
    - esp_bt_octet16_t dhk
- esp_ble_auth_cmpl_t auth_cmpl
    - esp_bd_addr_t bd_addr
    - bool key_present
    - esp_link_key key
    - uint8_t key_type
    - bool success
    - uint8_t fail_reason
    - esp_ble_addr_type_t addr_type
    - esp_bt_dev_type_t dev_type

Includes:

- #include <esp_gap_ble_api.h>

See also:

esp_ble_gap_security_rsp

Respond to a secure request.

```
esp_err_t esp_ble_gap_security_rsp(esp_bd_addr_t bd_addr, bool accept)
```

- `bd_addr` – The address of the peer.
- `accept` – True to accept the request and false to reject.

See also:

- ESP_GAP_BLE_SEC_REQ_EVT

esp_ble_gap_set_device_name

Set the name of the device as it will appear in advertising.

```
esp_err_t esp_ble_gap_set_device_name(const char *name)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_set_encryption

```
esp_err_t esp_ble_set_encryption(
   esp_bd_addr_t bd_addr,
   esp_ble_sec_act_t sec_act)
```

- bd_addr
- sec_act
  - ESP_BLE_SEC_NONE
  - ESP_BLE_SEC_ENCRYPT
  - ESP_BLE_SEC_ENCRYPT_NO_MITM
  - ESP_BLE_SEC_ENCRYPT_MITM

esp_ble_gap_set_scan_params

Set the parameters for a subsequent BLE scan.

```
esp_err_t esp_ble_gap_set_scan_params(esp_ble_scan_params_t *scan_params)
```

- `esp_ble_scan_type_t scan_type` – Scan type. One of:
  - `BLE_SCAN_TYPE_PASSIVE` – Perform a passive scan where no scan response is requested from the advertiser.
  - `BLE_SCAN_TYPE_ACTIVE` – Perform an active scan where a scan response is requested from the advertiser.
- `esp_ble_addr_type_t own_addr_type` – Own address type. One of:
  - BLE_ADDR_TYPE_PUBLIC
  - BLE_ADDR_TYPE_RANDOM
  - BLE_ADDR_TYPE_RPA_PUBLIC
  - BLE_ADDR_TYPE_RPA_RANDOM
- `esp_ble_scan_filter_t scan_filter_policy` – Filter policy. One of:
  - BLE_SCAN_FILTER_ALLOW_ALL
  - BLE_SCAN_FILTER_ALLOW_ONLY_WLST
  - BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR

    - BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR
- `uint16_t scan_interval` – Interval between scans. Value is multiplied by 0.625 msecs.
- `uint16_t scan_window` – Duration of the scan. Must be less than or equal to the scan_interval.

Includes:

- #include <esp_gap_ble_api.h>

See also:

- esp_ble_resolve_adv_data
- esp_ble_gap_start_scanning
- esp_ble_gap_stop_scanning
- ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT

esp_ble_gap_set_pkt_data_len

```
esp_err_t esp_ble_gap_set_pkt_data_len(
   esp_bd_addr_t remote_device,
   uint16_t      tx_data_length)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_gap_set_rand_addr

```
esp_err_t esp_ble_gap_set_rand_addr(esp_bd_addr_t rand_addr)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_gap_set_security_param

```
esp_err_t esp_ble_gap_set_security_param(
   esp_ble_sm_param_t param_type,
   void*              value,
   uint8_t            len)
```

- `param_type` – The type of parameter to set.
    - ESP_BLE_SM_PASSKEY
    - ESP_BLE_SM_AUTHEN_REQ_MODE

    - `ESP_BLE_SM_IOCAP_MODE` – Does the device have IO capability? The value parameter would be an instance of `esp_ble_io_cap_t`. Values of this are:
        - `ESP_IO_CAP_OUT` – Display only.
        - `ESP_IO_CAP_IO` – Keyboard and display.
        - `ESP_IO_CAP_IN` – Keyboard input only.
        - `ESP_IO_CAP_NONE` – No input and no output.
        - `ESP_IO_CAP_KBDISP` – Keyboard display.
    - ESP_BLE_SM_SET_INIT_KEY
    - ESP_BLE_SM_SET_RSP_KEY
    - ESP_BLE_SM_MAX_KEY_SIZE
- `value` – The value of the parameter.
- `len` – The length of the parameter.

For example:

```
esp_ble_io_cap_t iocap = ESP_IO_CAP_NONE;
esp_ble_gap_set_security_param(ESP_BLE_SM_IOCAP_MODE, &iocap, sizeof(uint8_t));
```

esp_ble_gap_start_advertising

Start GAP protocol advertising.

```
esp_err_t esp_ble_gap_start_advertising(esp_ble_adv_params_t *adv_params)
```

We can stop GAP protocol advertising by calling `esp_ble_gap_stop_advertising()`.

The `adv_params` structure contains the following:

- uint16_t adv_int_min
- uint16_t adv_int_max
- `esp_ble_adv_type_t adv_type` – One of:
    - `ADV_TYPE_IND` – Un-directed connectable mode (ADV_IND)
    - ADV_TYPE_DIRECT_IND_HIGH
    - ADV_TYPE_DIRECT_IND_LOW
    - `ADV_TYPE_NONCONN_IND` – Non-connectable (ADV_NONCONN_IND)
    - `ADV_TYPE_SCAN_IND` – Non-connectable but scan response available (ADV_SCAN_IND)
- `esp_ble_addr_type_t own_addr_type` – One of
        - BLE_ADDR_TYPE_PUBLIC

        - BLE_ADDR_TYPE_RANDOM
        - BLE_ADDR_TYPE_RPA_PUBLIC
        - BLE_ADDR_TYPE_RPA_RANDOM
- esp_bd_addr_t peer_addr
- `esp_ble_addr_type_t peer_addr_type` – One of
    - BLE_ADDR_TYPE_PUBLIC
    - BLE_ADDR_TYPE_RANDOM
    - BLE_ADDR_TYPE_RPA_PUBLIC
    - BLE_ADDR_TYPE_RPA_RANDOM
- `esp_ble_adv_channel_t channel_map` – One of:
    - ADV_CHNL_37
    - ADV_CHNL_38
    - ADV_CHNL_39
    - ADV_CHNL_ALL
- `esp_ble_adv_filter_t adv_filter_policy` – One of:
    - ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY
    - ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY
    - ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST
    - ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST

An example structure might be:

```
esp_ble_adv_params_t adv_params;
adv_params.adv_int_min       = 0x20;
adv_params.adv_int_max       = 0x40;
adv_params.adv_type          = ADV_TYPE_IND;
adv_params.own_addr_type     = BLE_ADDR_TYPE_PUBLIC;
adv_params.channel_map       = ADV_CHNL_ALL;
adv_params.adv_filter_policy = ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY;
```

Includes:

- #include <esp_gap_ble_api.h>

See also:

- Performing advertising
- esp_ble_gap_stop_advertising

- esp_ble_gap_config_adv_data
- esp_ble_gap_register_callback

esp_ble_gap_start_scanning

Start scanning for arriving advertising packets.

```
esp_err_t esp_ble_gap_start_scanning(uint32_t duration)
```

The duration is how long we are to perform scanning measured in seconds.

Includes:

- #include <esp_gap_ble_api.h>

See also:

- esp_ble_gap_set_scan_params
- esp_ble_gap_stop_scanning

esp_ble_gap_stop_advertising

Stop advertising.

```
esp_err_t esp_ble_gap_stop_advertising(void)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_gap_stop_scanning

Stop the current in progress scanning looking for advertising packets.

```
esp_err_t esp_ble_gap_stop_scanning(void)
```

Includes:

- #include <esp_gap_ble_api.h>

See also:

- esp_ble_gap_start_scanning

esp_ble_gap_update_conn_params

```
esp_err_t esp_ble_gap_update_conn_params(esp_ble_conn_update_params_t *params)
```

Includes:

- #include <esp_gap_ble_api.h>

esp_ble_gattc_app_register

Register application.

```
esp_err_t esp_ble_gattc_app_register(uint16_t app_id)
```

The `app_id` is the identity of an application. This can be any numeric smaller than `0x7fff`.

Includes:

- #include <esp_gattc_api.h>

See also:

- ESP_GATTC_REG_EVT

esp_ble_gattc_app_unregister

```
esp_err_t esp_ble_gattc_app_unregister(esp_gatt_if_t gatt_if)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_close

Close a previously opened connection.

```
esp_err_t esp_ble_gattc_close(
   esp_gatt_if_t gattc_if,
   uint16_t      conn_id)
```

Includes:

- #include <esp_gattc_api.h>

See also:

- esp_ble_gattc_open

esp_ble_gattc_config_mtu

```
esp_err_t esp_ble_gattc_config_mtu(
   uint16_t conn_id,
   uint16_t mtu)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_execute_write

```
esp_err_t esp_ble_gattc_execute_write(
   esp_gattc_if_t gattc_if,
   uint16_t       conn_id,
   bool           is_execute)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_get_characteristic

Ask the BLE partner to return a characteristic.

```
esp_err_t esp_ble_gattc_get_characteristic(
   esp_gatt_if_t       gattc_if,
   uint16_t            conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      last_characteristic_id)
```

When this call is made a request is sent to the GATT server to return the "next" characteristic. We should imagine that a GATT service has a set of 0 or more characteristics that it maintains in some form of internal order. The numeric order is not important, only that there is the concept of a "next" and "no more". When we call this function with `last_characteristic_id` set to NULL then we are saying "get me the first characteristic". If we subsequently call the function again passing in the id of the last characteristic returned, we will get the next characteristic. When we reach the end of the characteristics, the status code in the returned event will no longer be ok.

This call is a request only and the response will come asynchronously as an `ESP_GATTC_GET_CHAR_EVT` event.

The `conn_id` is the identity of a connection to the server.

The `srvc_id` is the identity of a service on the GATT server. This can be obtained by performing a search request against the server and the `esp_gatt_srvc_id_t` is returned in an `ESP_GATTC_SEARCH_RES_EVT` callback.

The `last_characteristic_id` can be NULL to get the first characteristic. Since we are iterating through characteristics, there must be a way to learn that there are no more. This is determined because we get a status value of `ESP_GATT_ERROR` for the `ESP_GATTC_GET_CHAR_EVT` event.

Includes:

- #include <esp_gattc_api.h>

See also:

- ESP_GATTC_SEARCH_RES_EVT
- ESP_GATTC_GET_CHAR_EVT
- esp_ble_gattc_search_service

esp_ble_gattc_get_descriptor

```
esp_err_t esp_ble_gattc_get_descriptor(
   esp_gatt_if_t       gattc_if,
   uint16_t            conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      characteristic_id,
   esp_gatt_id_t*      start_descriptor_id)
```

The `start_descriptor_id` can be NULL to get the first descriptor.

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_get_included_service

```
esp_err_t esp_ble_gattc_get_included_service(
   esp_gatt_if_t       gattc_if,
   uint16_t            conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_srvc_id_t* start_incl_srvc_id)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_open

Open a connection to the GATT server.

```
esp_err_t esp_ble_gattc_open(
   esp_gatt_if_t gatt_if,
   esp_bd_addr_t remote_bda,
   bool          is_direct)
```

The `gatt_if` is the application client interface. The initial is `ESP_GATT_IF_NONE`. We must store the gattc_if when we receive the first `ESP_GATTC_REG_EVT`.

The `remote_bda` is the bluetooth device address to which we wish to connect.

If `is_direct` then we are asking for a direct connection otherwise a background auto connection.

Includes:

- #include <esp_gattc_api.h>

See also:

- esp_ble_gattc_close
- ESP_GATTC_REG_EVT
- ESP_GATTC_OPEN_EVT
- ESP_GATTC_CLOSE_EVT

esp_ble_gattc_prepare_write

```
esp_err_t esp_ble_gattc_prepare_write(
   esp_gattc_if_t gattc_if,
   uint16_t           conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      char_id,
   uint16_t           offset,
   uint16_t           value_len,
   uint8_t*           value,
   esp_gatt_auth_req_t auth_req)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_read_char

Read the value of a characteristic.

```
esp_err_t esp_ble_gattc_read_char(
   esp_gatt_if_t       gattc_if,
   uint16_t           conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      characteristic_id,
   esp_gatt_auth_req_t auth_req)
```

The `auth_req` is one of:

- ESP_GATT_AUTH_REQ_NONE
- ESP_GATT_AUTH_REQ_NO_MITM

- ESP_GATT_AUTH_REQ_MITM
- ESP_GATT_AUTH_REQ_SIGNED_NO_MITM
- ESP_GATT_AUTH_REQ_SIGNED_MITM

Includes:

- #include <esp_gattc_api.h>

See also:

- ESP_GATTC_READ_CHAR_EVT

esp_ble_gattc_read_char_descr

```
esp_err_t esp_ble_gattc_read_char_descr(
   esp_gatt_if_t       gattc_if,
   uint16_t            conn_id,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      char_id,
   esp_gatt_id_t*      descr_id,
   esp_gatt_auth_req_t auth_req)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_register_callback

Register a callback to be invoked when a GATT event is received.

```
esp_err_t esp_ble_gattc_register_callback(esp_gattc_cb_t callback)
```

The `esp_gattc_cb_t` is a C function definition for a function with the signature:

```
void func(
   esp_gattc_cb_event_t      event,
   esp_gatt_if_t             gattc_if,
   esp_ble_gattc_cb_param_t* param)
```

Event types include:

- ESP_GATTC_ACL_EVT
- ESP_GATTC_ADV_DATA_EVT
- ESP_GATTC_ADV_VSC_EVT
- ESP_GATTC_BTH_SCAN_CFG_EVT
- ESP_GATTC_BTH_SCAN_DIS_EVT
- ESP_GATTC_BTH_SCAN_ENB_EVT

- ESP_GATTC_BTH_SCAN_PARAM_EVT
- ESP_GATTC_BTH_SCAN_RD_EVT
- ESP_GATTC_BTH_SCAN_THR_EVT
- ESP_GATTC_CANCEL_OPEN_EVT
- ESP_GATTC_CFG_MTU_EVT
- `ESP_GATTC_CLOSE_EVT` – Invoked when a connection is closed.
- ESP_GATTC_CONGEST_EVT
- ESP_GATTC_ENC_CMPL_CB_EVT
- ESP_GATTC_EXEC_EVT
- `ESP_GATTC_GET_CHAR_EVT` – Response to getting a characteristic.
- `ESP_GATTC_GET_DESCR_EVT` – Response to getting a characteristic description.
- ESP_GATTC_GET_INCL_SRVC_EVT
- ESP_GATTC_MULT_ADV_DATA_EVT
- ESP_GATTC_MULT_ADV_DIS_EVT
- ESP_GATTC_MULT_ADV_ENB_EVT
- ESP_GATTC_MULT_ADV_UPD_EVT
- ESP_GATTC_NOTIFY_EVT
- `ESP_GATTC_OPEN_EVT` – Invoked when a connection is opened.
- `ESP_GATTC_PREP_WRITE_EVT` – Response to write a characteristic.
- `ESP_GATTC_READ_CHAR_EVT` – Response to read a characteristic.
- `ESP_GATTC_REG_EVT` – Invoked when a GATT client has been registered.
- ESP_GATTC_REG_FOR_NOTIFY_EVT
- ESP_GATTC_SCAN_FLT_CFG_EVT
- ESP_GATTC_SCAN_FLT_PARAM_EVT
- ESP_GATTC_SCAN_FLT_STATUS_EVT
- `ESP_GATTC_SEARCH_CMPL_EVT` – Invoked when we have seen all search results.
- `ESP_GATTC_SEARCH_RES_EVT` – Invoked when we have a search result.
- ESP_GATTC_SRVC_CHG_EVT
- ESP_GATTC_READ_DESCR_EVT

- ESP_GATTC_UNREG_EVT
- ESP_GATTC_UNREG_FOR_NOTIFY_EVT
- ESP_GATTC_WRITE_CHAR_EVT
- ESP_GATTC_WRITE_DESCR_EVT

The `param` is a data structure that provides further details on the event. It appears to be an instance of `esp_ble_gattc_cb_param_t` which is a union of:

- `cfg_mtu` – set for `ESP_GATTC_CFG_MTU_EVT`.
- `close` – set for `ESP_GATTC_CLOSE_EVT`.
- `congest` – `ESP_GATTC_CONGEST_EVT`.
- `exec_cmpl` – set for `ESP_GATTC_EXEC_EVT`.
- `get_char` – set for `ESP_GATTC_GET_CHAR_EVT`.
- `get_descr` – set for `ESP_GATTC_GET_DESCR_EVT`.
- `get_incl_srvc` – set for `ESP_GATTC_GET_INCL_SRVC_EVT`.
- `notify` – set for `ESP_GATTC_NOTIFY_EVT`.
- `open` – set for `ESP_GATTC_OPEN_EVT`.
- `read` – set for `ESP_GATTC_READ_CHAR_EVT` and `ESP_GATTC_READ_DESCR_EVT`.
- `reg` – set for `ESP_GATTC_REG_EVT`.
- `reg_for_notify` – set for `ESP_GATTC_REG_FOR_NOTIFY_EVT`.
- `search_cmpl` – set for `ESP_GATTC_SEARCH_CMPL_EVT`.
- `search_res` – set for `ESP_GATTC_SEARCH_RES_EVT`.
- `srvc_chg` – set for `ESP_GATTC_SRVC_CHG_EVT`.
- `unreg_for_notify` – set for `ESP_GATTC_UNREG_FOR_NOTIFY_EVT`.
- `write` – set for `ESP_GATTC_WRITE_CHAR_EVT`, `ESP_GATTC_PREP_WRITE_EVT` and `ESP_GATTC_WRITE_DESCR_EVT`.

Includes:

- #include <esp_gattc_api.h>

Now let us look more deeply at each event type.

ESP_GATTC_ACL_EVT

ESP_GATTC_ADV_DATA_EVT

ESP_GATTC_ADV_VSC_EVT

ESP_GATTC_BTH_SCAN_CFG_EVT

ESP_GATTC_BTH_SCAN_DIS_EVT

ESP_GATTC_BTH_SCAN_ENB_EVT

ESP_GATTC_BTH_SCAN_PARAM_EVT

ESP_GATTC_BTH_SCAN_RD_EVT

ESP_GATTC_BTH_SCAN_THR_EVT

ESP_GATTC_CANCEL_OPEN_EVT

ESP_GATTC_CFG_MTU_EVT
struct gattc_cfg_mtu_evt_param

cfg_mtu

- esp_gatt_status_t status
- uint16_t conn_id
- uint16_t mtu

ESP_GATTC_CLOSE_EVT
Published when a GATT client connection is closed. This will commonly be the result of a call to `esp_ble_gattc_close()`. The `close` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status.
- `uint16_t conn_id` – Connection id.
- `esp_gatt_if_t gatt_if` – Gatt interface id, different application on gatt client different gatt_if.
- `esp_bd_addr_t remote_bda` – Remote bluetooth device address.
- `esp_gatt_conn_reason_t reason` – The reason of gatt connection close. One of:
    - `ESP_GATT_CONN_UNKNOWN` – Gatt connection unknown.

- ○ `ESP_GATT_CONN_L2C_FAILURE` – General L2cap failure.
- ○ `ESP_GATT_CONN_TIMEOUT` – Connection timeout.
- ○ `ESP_GATT_CONN_TERMINATE_PEER_USER` – Connection terminate by peer user.
- ○ `ESP_GATT_CONN_TERMINATE_LOCAL_HOST` – Connection terminated by local host.
- ○ `ESP_GATT_CONN_FAIL_ESTABLISH` – Connection fail to establish.
- ○ `ESP_GATT_CONN_LMP_TIMEOUT` – Connection fail for LMP response tout.
- ○ `ESP_GATT_CONN_CONN_CANCEL` – L2CAP connection canceled.
- ○ `ESP_GATT_CONN_NONE` – No connection to cancel.

See also:

- esp_ble_gattc_close

ESP_GATTC_CONGEST_EVT

struct gattc_congest_evt_param

congest

- uint16_t conn_id
- bool congested

ESP_GATTC_CONNECT_EVT

struct gattc_connect_evt_param

connect

- esp_gatt_status_t status
- uint16_t conn_id
- esp_bd_addr_t remote_bda

ESP_GATTC_DISCONNECT_EVT

struct gattc_disconnect_evt_param

disconnect

- esp_gatt_status_t status
- uint16_t conn_id

- esp_bd_addr_t remote_bda

ESP_GATTC_ENC_CMPL_CB_EVT

ESP_GATTC_EXEC_EVT

struct gattc_exec_cmpl_evt_param

exec_cmpl

- esp_gatt_status_t status
- uint16_t conn_id

ESP_GATTC_GET_CHAR_EVT

Published when a GATT call to `esp_ble_gattc_get_characteristic()` has been made. The `get_char` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status. On success, `ESP_GATT_OK` is returned.
- `uint16_t conn_id` – Connection id.
- `esp_gatt_srvc_id_t srvc_id` – Service id, includes service uuid and other information.
- `esp_gatt_id_t char_id` – Characteristic id, include characteristic uuid and other information. If the status field indicates an error, this field should not be examined.
    - esp_bt_uuid_t uuid
    - uint8_t inst_id
- `esp_gatt_char_prop_t char_prop` – Characteristic properties. This is the set of flags that identify what can be done against the characteristics. If the status field indicates an error, this field should not be examined. The bit flags are:
    - ESP_GATT_CHAR_PROP_BIT_AUTH
    - ESP_GATT_CHAR_PROP_BIT_BROADCAST
    - ESP_GATT_CHAR_PROP_BIT_EXT_PROP
    - ESP_GATT_CHAR_PROP_BIT_INDICATE
    - ESP_GATT_CHAR_PROP_BIT_NOTIFY
    - ESP_GATT_CHAR_PROP_BIT_READ
    - ESP_GATT_CHAR_PROP_BIT_WRITE

    - ESP_GATT_CHAR_PROP_BIT_WRITE_NR

See also:

- esp_ble_gattc_get_characteristic

ESP_GATTC_GET_DESCR_EVT
struct gattc_get_descr_evt_param

get_descr

- esp_gatt_status_t status
- uint16_t conn_id
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_id_t char_id
- esp_gatt_id_t descr_id

ESP_GATTC_GET_INCL_SRVC_EVT
struct gattc_get_incl_srvc_evt_param

get_incl_srvc

- esp_gatt_status_t status
- uint16_t conn_id
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_srvc_id_t incl_srvc_id

ESP_GATTC_MULT_ADV_DATA_EVT

ESP_GATTC_MULT_ADV_DIS_EVT

ESP_GATTC_MULT_ADV_ENB_EVT

ESP_GATTC_MULT_ADV_UPD_EVT

ESP_GATTC_NOTIFY_EVT
struct gattc_notify_evt_param

notify

- uint16_t conn_id
- esp_bd_addr_t remote_bda
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_id_t char_id
- esp_gatt_id_t descr_id
- uint16_t value_len
- uint8_t *value
- bool is_notify

ESP_GATTC_OPEN_EVT

Published when a GATT client open call has been made. This is normally the result of calling `esp_ble_gattc_open()`. The `open` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status. The status should be checked to see what the outcome of the request may be. A good return is `ESP_GATT_OK`.
- `uint16_t conn_id` – Connection id.
- `esp_gatt_if_t gatt_if` – GATT interface id, different application on GATT client different gatt_if
- `esp_bd_addr_t remote_bda` – Remote bluetooth device address.
- `uint16_t mtu` – MTU size.

See also:

- esp_ble_gattc_open

ESP_GATTC_PREP_WRITE_EVT

ESP_GATTC_READ_CHAR_EVT

Published when a GATT characteristic value has been received. The `read` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status.
- `uint16_t conn_id` – Connection id.
- `esp_gatt_srvc_id_t srvc_id` – Service id, include service uuid and other information.

- `esp_gatt_id_t char_id` – Characteristic id, include characteristic uuid and other information.
- `esp_gatt_id_t descr_id` – Descriptor id, include descriptor uuid and other information.
- `uint8_t* value` – Characteristic value.
- `uint16_t value_type` – Characteristic value type. By experimentation we seem to find:
  - 0 – string
- `uint16_t value_len` – Characteristic value length.

See also:

- esp_ble_gattc_read_char

ESP_GATTC_READ_DESC_EVT

ESP_GATTC_REG_EVT

Published when a GATT client has been registered. This will result from a previous call to `esp_ble_gattc_app_register()`. The `reg` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status. A GATT status code.
- `uint16_t app_id` – Application id which input in register API.

See also:

- esp_ble_gattc_app_register

ESP_GATTC_REG_FOR_NOTIFY_EVT

struct gattc_reg_for_notify_evt_param

reg_for_notify

- esp_gatt_status_t status
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_id_t char_id;

ESP_GATTC_SEARCH_CMPL_EVT

Published when all search results has been received following a call to `esp_ble_gattc_search_service()`. The `search_cmpl` field of `esp_ble_gattc_cb_param_t` is populated. It contains:

- `esp_gatt_status_t status` – Operation status.
- `uint16_t conn_id` – Connection id.

See also:

- esp_ble_gattc_search_service

ESP_GATTC_SEARCH_RES_EVT

Published a search result has been received following a call to `esp_ble_gattc_search_service()`. The `search_res` field of `esp_ble_gattc_cb_param_t` is populated. This is an instance of `struct gattc_search_res_evt_param`. It contains:

- `uint16_t conn_id` – Connection id.
- `esp_gatt_srvc_id_t srvc_id` – Service id, includes service uuid and other information. Specifically:
  - `esp_gatt_id_t id` – The details of the GATT id. This contains:
    - `esp_bt_uuid_t uuid` – The UUID of the service.
    - `uint8_t inst_id` – The instance of the service.
  - `bool is_primary` – Is this a primary service.

See also:

- esp_ble_gattc_search_service
- esp_ble_gattc_get_characteristic
- ESP_GATTC_SEARCH_CMPL_EVT

ESP_GATTC_SCAN_FLT_CFG_EVT

ESP_GATTC_SCAN_FLT_PARAM_EVT

ESP_GATTC_SCAN_FLT_STATUS_EVT

ESP_GATTC_SRVC_CHG_EVT

struct gattc_srvc_chg_evt_param

- srvc_chg
- esp_bd_addr_t remote_bda

ESP_GATTC_UNREG_EVT

ESP_GATTC_UNREG_FOR_NOTIFY_EVT

struct gattc_unreg_for_notify_evt_param

unreg_for_notify

- esp_gatt_status_t status
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_id_t char_id;

ESP_GATTC_WRITE_CHAR_EVT

struct gattc_write_evt_param

write

- esp_gatt_status_t status
- uint16_t conn_id
- esp_gatt_srvc_id_t srvc_id
- esp_gatt_id_t char_id
- esp_gatt_id_t descr_id

esp_ble_gattc_register_for_notify

Called to register for notification.

```
esp_err_t esp_ble_gattc_register_for_notify(
   esp_gatt_if_t       gatt_if,
   esp_bd_addr_t       server_bda,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      char_id)
```

- esp_gatt_if_t gatt_if
- esp_bd_addr_t server_bda
- esp_gatt_srvc_id_t *srvc_id
- esp_gatt_id_t *char_id

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_unregister_for_notify

```
esp_err_t esp_ble_gattc_unregister_for_notify (
   esp_gatt_if_t       gatt_if,
   esp_bd_addr_t       server_bda,
   esp_gatt_srvc_id_t* srvc_id,
   esp_gatt_id_t*      char_id);
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gattc_search_service

Ask the BLE device for the set of services it provides. The services will be returned asynchronously via GATT events.

```
esp_err_t esp_ble_gattc_search_service(
   esp_gatt_if_t  gattc_if,
   uint16_t       conn_id,
   esp_bt_uuid_t* filter_uuid)
```

The `gattc_if` is returned to us in the open event callback.

The `conn_id` is the identity of a connection that was opened by a call to `esp_ble_gattc_open()` and obtained from a GATT event callback.

The `filter_uuid` allows us to filter UUIDs or else we can specify `NULL`.

After calling this function we might expect to see a sequence of `ESP_GATTC_SEARCH_RES_EVT` events published finalized with an `ESP_GATTC_SEARCH_CMPL_EVT`.

Includes:

- #include <esp_gattc_api.h>

See also:

- ESP_GATTC_SEARCH_CMPL_EVT
- ESP_GATTC_SEARCH_RES_EVT

esp_ble_gattc_write_char

Set the value of a remote device's characteristic.

```
esp_err_t esp_ble_gattc_write_char(
   esp_gatt_if_t         gattc_if,
   uint16_t              conn_id,
   esp_gatt_srvc_id_t*   srvc_id,
   esp_gatt_id_t*        characteristic_id,
   uint16_t              value_len,
   uint8_t*               value,
   esp_gatt_write_type_t  write_type,
   esp_gatt_auth_req_t    auth_req)
```

- esp_gatt_if_t gattc_if
- uint16_t conn_id
- esp_gatt_srvc_id_id* srvc_id
- esp_gatt_id_t* characteristic_id
- `uint16_t value_len` – The length, in bytes, of the value.
- `uint8_t* value` – A pointer to the start of the data for the value.
- esp_gatt_write_type_t write_type
  - ESP_GATT_WRITE_TYPE_NO_RSP
  - ESP_GATT_WRITE_TYPE_RSP
- esp_gatt_auth_req_t auth_req
  - ESP_GATT_AUTH_REQ_NONE
  - ESP_GATT_AUTH_REQ_NO_MITM
  - ESP_GATT_AUTH_REQ_MITM
  - ESP_GATT_AUTH_REQ_SIGNED_NO_MITM
  - ESP_GATT_AUTH_REQ_SIGNED_MITM

Includes:

- #include <esp_gattc_api.h>

See also:

- ESP_GATTC_WRITE_CHAR_EVT

esp_ble_gattc_write_char_descr

```
esp_err_t esp_ble_gattc_write_char_descr(
   esp_gatt_if_t          gattc_if,
   uint16_t               conn_id,
   esp_gatt_srvc_id_t*    srvc_id,
   esp_gatt_id_t*         char_id,
   esp_gatt_id_t*         descr_id,
   uint16_t               value_len,
   uint8_t*               value,
   esp_gatt_write_type_t  write_type,
   esp_gatt_auth_req_t    auth_req)
```

Includes:

- #include <esp_gattc_api.h>

esp_ble_gatts_add_char

Add a characteristic to the service.

```
esp_err_t esp_ble_gatts_add_char(
   uint16_t             serviceHandle,
   esp_bt_uuid_t*       characteristicUuid,
   esp_gatt_perm_t      permissions,
   esp_gatt_char_prop_t properties,
   esp_attr_value_t*    characteristicValue,
   esp_attr_control_t*  control)
```

Within BLE we have the notion of a service and a service can have zero or more characteristics associated with it. Invoking this function declares a new characteristics associated with the service. When we think about a characteristics, we should also understand that it has a value associated with it. This value can either be managed by our application logic explicitly or it can be managed by the ESP-IDF environment. The choice is governed by the control property. If we set that to be `ESP_GATT_AUTO_RSP` then the ESP environment is managing our value on our behalf and we must supply an initial value through the `characteristicValue` parameter. However, if we set the control parameter to be `ESP_GATT_RSP_BY_APP` then we are declaring that there will be events arriving to set/get our characteristic value.

- `serviceHandle` – Attribute handle for the service to which this characteristic is to be added.
- `characteristicUuid` – UUID for the characteristic.
- `permissions` – Attribute permissions. These can be "or'd" together
  - ESP_GATT_PERM_READ
  - ESP_GATT_PERM_READ_ENCRYPTED
  - ESP_GATT_PERM_READ_ENC_MITM
  - ESP_GATT_PERM_WRITE
  - ESP_GATT_PERM_WRITE_ENCRYPTED
  - ESP_GATT_PERM_WRITE_ENC_MITM
  - ESP_GATT_PERM_WRITE_SIGNED
  - ESP_GATT_PERM_WRITE_SIGNED_MITM
- `properties` – Characteristic properties.

  - ESP_GATT_CHAR_PROP_BIT_BROADCAST
  - ESP_GATT_CHAR_PROP_BIT_READ
  - ESP_GATT_CHAR_PROP_BIT_WRITE_NR
  - ESP_GATT_CHAR_PROP_BIT_WRITE
  - ESP_GATT_CHAR_PROP_BIT_NOTIFY
  - ESP_GATT_CHAR_PROP_BIT_INDICATE
  - ESP_GATT_CHAR_PROP_BIT_AUTH
  - ESP_GATT_CHAR_PROP_BIT_EXT_PROP
- `characteristicValue` – Characteristic value.
  - uint16_t attr_max_len
  - uint16_t attr_len
  - uint8_t *attr_value
- `control` – Control flags.
  - `auto_rsp` – How to automatically respond.
    - ESP_GATT_RSP_BY_APP
    - ESP_GATT_AUTO_RSP

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_ADD_CHAR_EVT

esp_ble_gatts_add_char_descr

```
esp_err_t esp_ble_gatts_add_char_descr(
   uint16_t            serviceHandle,
   esp_bt_uuid_t*      descriptorUuid,
   esp_gatt_perm_t     permissions,
   esp_attr_value_t*   characteristicDescriptorValue,
   esp_attr_control_t* control)
```

- `serviceHandle` – The handle of the service to which to add this characteristic descriptor.
- `descriptorUuid` – The UUID for the new descriptor.
- `permissions` – Attribute permissions. These can be "or'd" together

  - ESP_GATT_PERM_READ
  - ESP_GATT_PERM_READ_ENCRYPTED
  - ESP_GATT_PERM_READ_ENC_MITM
  - ESP_GATT_PERM_WRITE
  - ESP_GATT_PERM_WRITE_ENCRYPTED
  - ESP_GATT_PERM_WRITE_ENC_MITM
  - ESP_GATT_PERM_WRITE_SIGNED
  - ESP_GATT_PERM_WRITE_SIGNED_MITM
- `characteristicDescriptorValue` – Characteristic descriptor value:
  - uint16_t attr_max_len
  - uint16_t attr_len
  - uint8_t *attr_value
- `control` – Control flags.
  - `auto_rsp` – How to automatically respond.
    - ESP_GATT_RSP_BY_APP
    - ESP_GATT_AUTO_RSP

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_ADD_CHAR_DESCR_EVT

esp_ble_gatts_add_included_service

```
esp_err_t esp_ble_gatts_add_included_service(
   uint16_t service_handle,
   uint16_t included_service_handle)
```

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_ADD_INCL_SRVC_EVT

esp_ble_gatts_app_register

```
esp_err_t esp_ble_gatts_app_register(uint16_t app_id)
```

The `app_id` is the identity of an application. This can be any numeric smaller than `0x7fff`.

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_REG_EVT

esp_ble_gatts_app_unregister

```
esp_err_t esp_ble_gatts_app_unregister(esp_gatt_if_t gatts_if)
```

Includes:

- #include <esp_gatts_api.h>

esp_ble_gatts_close

Close a connection that was opened by a client.

```
esp_err_t esp_ble_gatts_close(
   esp_gatt_if_t gatts_if,
   uint16_t conn_id)
```

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_CLOSE_EVT

esp_ble_gatts_create_attribute_tab

Have the attributes automatically maintained for us.

```
esp_err_t esp_ble_gatts_create_attr_tab(
   const esp_gatts_attr_db_t *gatts_attr_db,
   esp_gatt_if_t gatts_if,
   uint8_t max_nb_attr,
   uint8_t srvc_inst_id)
```

- gatts_attr_db
- gatts_if
- max_nb_attr

- srvc_inst_id

Don't mix this technique with esp_gatts_create_service/esp_ble_gatts_add_char.

Includes:

- #include <esp_gatts_api.h>

See also:

- esp_ble_gatts_set_attr_value
- esp_ble_gatts_get_attr_value

esp_ble_gatts_create_service

Create a service definition.

```
esp_err_t esp_ble_gatts_create_service(
   esp_gatt_if_t       gatts_if,
   esp_gatt_srvc_id_t* service_id,
   uint16_t            num_handle)
```

The `service_id` is a structure containing the identity of the service.

- esp_gatt_id_t id
- bool is_primary

The `num_handle` is the number of the handle requested for this service.

A call to this function will result in an eventual `ESP_GATTS_CREATE_EVT` event being raised. It is that event which will contain the service handle used in subsequent calls.

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_CREATE_EVT
- esp_gatt_srvc_id_t

esp_ble_gatts_delete_service

```
esp_err_t esp_ble_gatts_delete_service(uint16_t service_handle);
```

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_DELETE_EVT

esp_ble_gatts_get_attr_value

```
esp_err_t esp_ble_gatts_get_attr_value(
   uint16_t attr_handle,
   uint16_t *length,
   const uint8_t **value)
```

Includes:

- #include <esp_gatts_api.h>

esp_ble_gatts_open

```
esp_err_t esp_ble_gatts_open(
   esp_gatt_if_t gatts_if,
   esp_bd_addr_t remote_bda,
   bool is_direct)
```

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_OPEN_EVT

esp_ble_gatts_register_callback

Register GATT server callbacks.

```
esp_err_t esp_ble_gatts_register_callback(esp_gatts_cb_t callback)
```

The callback is an instance of a callback function that has the following signature:

```
void callback(
   esp_gatts_cb_event_t event,
   esp_gatt_if_t gatts_if,
   esp_ble_gatts_cb_param_t *param
)
```

The `esp_ble_gatts_cb_param_t` is a union of:

- struct gatts_add_attr_tab_evt_param add_attr_tab
- struct gatts_add_char_evt_param add_char
- struct gatts_add_char_descr_evt_param add_char_descr
- struct gatts_conf_evt_param conf
- struct gatts_connect_evt_param connect
- struct gatts_create_evt_param create
- struct gatts_delete_evt_param del

- struct gatts_disconnect_evt_param disconnect
- struct gatts_exec_write_evt_param exec_write
- struct gatts_mtu_evt_param mtu
- struct gatts_read_evt_param read
- struct gatts_reg_evt_param reg
- struct gatts_rsp_evt_param rsp
- struct gatts_set_attr_val_evt_param set_attr_val
- struct gatts_start_evt_param start
- struct gatts_stop_evt_param stop
- struct gatts_write_evt_param write

The event is one of:

ESP_GATTS_ADD_CHAR_DESCR_EVT
The passed in parameter is an instance of `struct gatts_add_char_descr_evt_param` which is identified by the field called `add_char_descr`. The structure contains:

- `esp_gatt_status_t status` – Status of the descriptor creation.
- `uint16_t attr_handle` – Attribute handle of the descriptor.
- `uint16_t service_handle` – Handle of the service.
- `esp_bt_uuid_t char_uuid` – UUID of descriptor.

See also:

- esp_ble_gatts_add_char_descr

ESP_GATTS_ADD_CHAR_EVT
When a GATT client connects, we receive this event. The passed in parameter is an instance of `struct gatts_add_char_evt_param` which is identified by the field called `add_char`. The structure contains:

- `esp_gatt_status_t status` – Status of the creation of the characteristic.
- `uint16_t attr_handle` – Attribute handle for the characteristic.
- `uint16_t service_handle` – Service handle.
- `esp_bt_uuid_t char_uuid` – UUID of the characteristic.

See also:

- esp_ble_gatts_add_char
- esp_bt_uuid_t

ESP_GATTS_ADD_INCL_SRVC_EVT

The passed in parameter is an instance of `struct gatts_add_incl_srvc_evt_param` identified by the field called `add_incl_srvc`. The structure contains:

- esp_gatt_status_t status
- uint16_t attr_handle
- uint16_t service_handle

ESP_GATTS_CANCEL_OPEN_EVT

The passed in parameter is an instance of `struct gatts_cancel_open_evt_param` identified by the field called `cancel_open`. The structure contains:

- esp_gatt_status_t status

ESP_GATTS_CLOSE_EVT

The passed in parameter is an instance of `struct gatts_close_evt_param` which is identified by the field called `close`. The structure contains:

- esp_gatt_status_t status
- uint16_t conn_id

ESP_GATTS_CONF_EVT

A confirmation event of a request that we issued.

The passed in parameter is an instance of `struct gatts_conf_evt_param` which is identified by the field called `conf`. The structure contains:

- `esp_gatt_status_t status` – The status code.
- `uint16_t conn_id` – The connection used.

See also:

- esp_ble_gatts_send_indicate

ESP_GATTS_CONGEST_EVT

The passed in parameter is an instance of `struct gatts_congest_evt_param` which is identified by the field called `congest`. The structure contains:

- uint16_t conn_id
- bool congested

ESP_GATTS_CONNECT_EVT

When a GATT client connects, we receive this event. The passed in parameter is an instance of `struct gatts_connect_evt_param` which is identified by the field called `connect`. The structure contains:

- `uint16_t conn_id` – The connection id.
- `esp_bd_addr_t remote_bda` – The address of the peer device.
- `bool is_connected` – Are we connected?

ESP_GATTS_CREAT_ATTR_TAB_EVT

The passed in parameter is an instance of `struct gatts_add_attr_tab_evt_param` which is identified by the field called `add_attr_tab`. The structure contains:

- esp_gatt_status_t status
- esp_bt_uuid_t svc_uuid
- uint16_t num_handle
- uint16_t *handles

ESP_GATTS_CREATE_EVT

This event is found when a new service has been created. As such, it is commonly found after a call to esp_ble_gatts_create_service(). The passed in parameter is an instance of `struct gatts_create_evt_param` which is identified by the field called `create`. The structure contains:

- `esp_gatt_status_t status` – The status of the creation request.
- `uint16_t service_handle` – The handle allocated by the environment to this service.
- `esp_gatt_srvc_id_t service_id` – The identity of the service.

See also:

- esp_ble_gatts_create_service
- esp_gatt_srvc_id_t

ESP_GATTS_DELETE_EVT

The passed in parameter is an instance of `struct gatts_delete_evt_param` which is identified by the field called `del`. The structure contains:

- esp_gatt_status_t status
- uint16_t service_handle

ESP_GATTS_DISCONNECT_EVT

When a GATT client connects, we receive this event. The passed in parameter is an instance of `struct gatts_disconnect_evt_param` which is identified by the field called `disconnect`. The structure contains:

- `uint16_t conn_id` – The connection id that just disconnected.
- `esp_bd_addr_t remote_bda` – The device address of the device which just disconnected.
- `bool is_connected` – Are we connected? Should be false.

If we are a GATT server, following a disconnect, we must start advertising again in order to be able to receive further incoming connections.

See also:

- esp_ble_gap_config_adv_data

ESP_GATTS_EXEC_WRITE_EVT

The passed in parameter is an instance of `struct gatts_exec_write_evt_param` which is identified by the field called `exec_write`. The structure contains:

- uint16_t conn_id
- uint32_t trans_id
- esp_bd_addr_t bda
- uint8_t exec_write_flag
  - ESP_GATT_PREP_WRITE_CANCEL
  - ESP_GATT_PREP_WRITE_EXEC

Upon receipt of one of these events, we should acknowledge it with a call to `esp_ble_gatts_send_response()`.

See also:

- ESP_GATTS_WRITE_EVT
- esp_ble_gatts_send_response

ESP_GATTS_LISTEN_EVT

ESP_GATTS_MTU_EVT

Called when set MTU complete. The passed in parameter is an instance of `struct gatts_mtu_evt_param` which is identified by the field called `mtu`. The structure contains:

- `uint16_t conn_id` – Connection id.
- `uint16_t mtu` – MTU size.

ESP_GATTS_OPEN_EVT

The passed in parameter is an instance of `struct gatts_open_evt_param` which is identified by the field called `open`. The structure contains:

- esp_gatt_status_t status

ESP_GATTS_READ_EVT

When a GATT client requests to read an attribute, we receive this event. The passed in parameter is an instance of `struct gatts_read_evt_param` which is identified by the field called `read`. The structure contains:

- `uint16_t conn_id` – Connection id.
- `uint32_t trans_id` – Transfer id.
- `esp_bd_addr_t bda` – Address of partner requesting read.
- `uint16_t handle` – Attribute handle.
- `uint16_t offset` – offset within value.
- `bool is_long` – value is long or not.
- `bool need_rsp` – Read operations needs a response.

See also:

- esp_ble_gatts_send_response

ESP_GATTS_REG_EVT

Invoked when ??.

The passed in parameter is an instance of `struct gatts_reg_evt_param` which is identified by the field called `reg`. The structure contains:

- esp_gatt_status_t status
- uint16_t app_id

ESP_GATTS_RESPONSE_EVT

The passed in parameter is an instance of `struct gatts_rsp_evt_param` which is identified by the field called `rsp`. The structure contains:

- esp_gatt_status_t status
- uint16_t handle

ESP_GATTS_SET_ATTR_VAL_EVT

The passed in parameter is an instance of `struct gatts_set_attr_val_evt_param` which is identified by the field called `set_attr_val`. The structure contains:

- uint16_t srvc_handle
- uint16_t attr_handle
- esp_gatt_status_t status

ESP_GATTS_START_EVT

When a GATT service started, we receive this event. As such, this is commonly the response from a call to esp_ble_gatts_start_service(). The passed in parameter is an instance of `struct gatts_start_evt_param` which is identified by the field called `start`. The structure contains:

- esp_gatt_status_t status
- uint16_t service_handle

See also:

- esp_ble_gatts_start_service

ESP_GATTS_STOP_EVT

The passed in parameter is an instance of `struct gatts_stop_evt_param` which is identified by the field called `stop`. The structure contains:

- esp_gatt_status_t status
- uint16_t service_handle

ESP_GATTS_UNREG_EVT

ESP_GATTS_WRITE_EVT

The passed in parameter is an instance of `struct gatts_write_evt_param` which is identified by the field called `write`. The structure contains:

- `uint16_t conn_id` – The connection id.
- `uint16_t trans_id` – The transfer id.
- `esp_bd_addr_t bda` – The address of the partner.
- `uint16_t handle` – The attribute handle.
- `uint16_t offset` – The offset of the currently received within the whole value.
- `bool need_rsp` – Do we need a response?
- `bool is_prep` – Is this a write prepare? If set, then this is to be considered part of the received value and not the whole value. A subsequent ESP_GATTS_EXEC_WRITE will mark the total.
- `uint16_t len` – The length of the incoming value part.
- `uint8_t* value` – The data for this value part.

If a write event arrives and the `need_rsp` is true, we must call `esp_ble_gatts_send_response()` to respond. The response `rsp` structure should contain the handle that we are responding to.

Because the packet size of a BLE message is not very large, and is usually much smaller than the maximum size of a characteristic value, a request to write a new value of a characteristic must arrive as a sequence of packets. This is indicated by the `is_prep` flag being set. The full name for this flag might be considered to be "is part of a prepare to write". The piece parts should arrive and be accumulated without actually setting the value of the characteristic. Once all the parts have arrived and are assembled, an event of type `ESP_GATTS_EXEC_WRITE` will arrive indicating that the accumulated new value is complete and can be written as the value of a characteristic as a complete whole.

The high level algorithm for handling a write may thus be:

```
if (is_prep == TRUE) {
   append this current data to the accumulating data starting at offset in the
      accumulating data.  The new total data will be original accumulating data plus
      an additional "len" bytes of new data.
}
```

We should also be aware of trans_id. This is an incrementing transfer id. Each part will have a value one higher than the previous value. If we receive a part that is not expected, this can indicate an error to which we can respond appropriately.

The need_rsp flag indicates whether or not we should send a response back to the client. If yes, then we call `esp_ble_gatts_send_response()`. The response payload will include the passed in data offset, length and a copy of the received data.

Includes:

- #include <esp_gatts_api.h>

See also:

- esp_ble_gatts_send_response
- ESP_GATTS_EXEC_WRITE_EVT

esp_ble_gatts_send_indicate

Send an indication or notification to the GATT client.

```
esp_err_t esp_ble_gatts_send_indicate(
   esp_gatt_if_t gatts_if,
   uint16_t      conn_id,
   uint16_t      attr_handle,
   uint16_t      value_len,
   uint8_t*      value,
   bool          need_confirm);
```

Note that the BLE specification constrains the maximum data size to be 20 bytes or less. If we try and send more, the data is truncated to 20 bytes. If more data needs to be sent, consider using the indication as an indication of new/more data being available and have the partner perform a read request to get the full span of data.

- `esp_gatt_if_t gatts_if` – The interface on which to send the indication.
- `uint16_t conn_id` – The connection on which to send the indication.
- `uint16_t attr_handle` – The handle of the characteristic we are indicating.
- `uint16_t value_len` – The length of the data value. The maximum data size is 20 bytes.
- `uint8_t* value` – The data value.

- `bool need_confirm` – Whether or not we need a confirmation from the peer. If we ask for a confirmation the request is known as indicate while if we don't require a confirmation, the request is known as a notify.

The need_confirm requires a little explanation. If set to false, then we **will** receive a `ESP_GATTS_CONF_EVT` event … but this will be from the *local* BLE stack indicating whether or not the request was transmitted.

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_CONF_EVT

esp_ble_gatts_send_response

Send a response back for a request.

```
esp_err_t esp_ble_gatts_send_response(
   esp_gatt_if_t     gatts_if,
   uint16_t          conn_id,
   uint32_t          trans_id,
   esp_gatt_status_t status,
   esp_gatt_rsp_t*   rsp)
```

- `gatts_if` – The server access interface.
- `conn_id` – The connection id.
- `trans_id` – The transfer id.
- `status` – The status. ESP_GATT_OK for normal.
- `rsp` – The response to send back to the partner. A union of:
  - esp_gatt_value_t attr_value
    - uint16_t value[ESP_GATT_MAX_ATTR_LEN] // 600 bytes
    - uint16_t handle
    - uint16_t offset
    - uint16_t len
    - `uint16_t auth_req` – Authorization
      - ESP_GATT_AUTH_REQ_NONE
      - ESP_GATT_AUTH_REQ_NO_MITM
      - ESP_GATT_AUTH_REQ_MITM

        - ESP_GATT_AUTH_REQ_SIGNED_NO_MITM
        - ESP_GATT_AUTH_REQ_SIGNED_MITM
    - uint16_t handle

The responses sent back from the server are indications of outcome for the following events:

- ESP_GATTS_EXEC_WRITE_EVT
- ESP_GATTS_READ_EVT
- ESP_GATTS_WRITE_EVT

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_READ_EVT

esp_ble_gatts_set_attr_value

```
esp_err_t esp_ble_gatts_set_attr_value(
   uint16_t attr_handle,
   uint16_t length,
   const uint8_t *value)
```

Includes:

- #include <esp_gatts_api.h>

esp_ble_gatts_start_service

Start the service identified by the service handle.

```
esp_err_t esp_ble_gatts_start_service(uint16_t serviceHandle)
```

The `serviceHandle` is the local handle for the service. This is the value returned when we created the service and received the corresponding `ESP_GATTS_CREATE_EVT`.

A call to this function will result in an eventual `ESP_GATTS_START_EVT` event being raised.

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_START_EVT

esp_ble_gatts_stop_service

```
esp_err_t esp_ble_gatts_stop_service(uint16_t service_handle)
```

Includes:

- #include <esp_gatts_api.h>

See also:

- ESP_GATTS_STOP_EVT

esp_ble_resolve_adv_data

Resolve Advertized data.

```
uint8_t *esp_ble_resolve_adv_data(
   uint8_t *adv_data,
   uint8_t type,
   uint8_t *length)
```

When a GAP protocol callback occurs, one of the event types that can cause this is the `ESP_GAP_BLE_SCAN_RESULT_EVT` which indicates that we have received a scan result. If found, we can further ask what kind of a result and we can get an `ESP_GAP_SEARCH_INQ_RES_EVT`. If that is the case, then our ADV data is good and we can "decode it".

The `adv_data` is a pointer to the advertized data.

The type specifies the decode value. In the spec this is also known as the "Advertising Data Type"

- `ESP_BLE_AD_TYPE_FLAG (0x01)` – The advert contains flags that are defined as following:
    - Bit 0 – LE Limited Discoverable Mode
    - Bit 1 – LE General Discoverable Mode
    - Bit 2 – BR/EDR is NOT supported.
    - Bit 3 – Indicates whether LE and BR/EDR Controller operates simultaneously
    - Bit 4 – Indicates whether LE and BR/EDR Host operates simultaneously
    - Bits 5-7 – Reserved.
- `ESP_BLE_AD_TYPE_16SRV_PART (0x02)` – Incomplete list of 16bit service class UUIDs.
- ESP_BLE_AD_TYPE_16SRV_CMPL (0x03)
- ESP_BLE_AD_TYPE_32SRV_PART (0x04)
- ESP_BLE_AD_TYPE_32SRV_CMPL (0x05

- ESP_BLE_AD_TYPE_128SRV_PART (0x06) – Incomplete list of 16bit service class UUIDs.
- ESP_BLE_AD_TYPE_128SRV_CMPL (x07)
- ESP_BLE_AD_TYPE_NAME_SHORT (0x08) – Shortened local name.
- ESP_BLE_AD_TYPE_NAME_CMPL (0x09) – Complete local name.
- ESP_BLE_AD_TYPE_TX_PWR (0x0A)
- ESP_BLE_AD_TYPE_DEV_CLASS (0x0D)
- ESP_BLE_AD_TYPE_SM_TK (0x10)
- ESP_BLE_AD_TYPE_SM_OOB_FLAG (0x11)
- ESP_BLE_AD_TYPE_INT_RANGE (0x12)
- ESP_BLE_AD_TYPE_SOL_SRV_UUID (0x14)
- ESP_BLE_AD_TYPE_128SOL_SRV_UUID (0x15)
- ESP_BLE_AD_TYPE_SERVICE_DATA (0x16)
- ESP_BLE_AD_TYPE_PUBLIC_TARGET (0x17)
- ESP_BLE_AD_TYPE_RANDOM_TARGET (0x18)
- ESP_BLE_AD_TYPE_APPEARANCE (0x19) – It is likely this conforms to the assigned numbers found here https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.appearance.xml and
- ESP_BLE_AD_TYPE_ADV_INT (0x1A)
- ESP_BLE_AD_TYPE_32SOL_SRV_UUID (0x1B)
- ESP_BLE_AD_TYPE_32SERVICE_DATA (0x1C)
- ESP_BLE_AD_TYPE_128SERVICE_DATA (0x1D)
- ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE (0xFF) – Custom payload.

The length is the length of the advertized data.

Includes:

- #include <esp_gap_ble_api.h>

See also:

- GAP Advertizing data
- esp_ble_gap_register_callback

esp_bluedroid_deinit

Disable and release all resources for bluetooth.

```
esp_err_t esp_bluedroid_deinit(void);
```

Includes:

- #include <esp_bt_main.h>

esp_bluedroid_disable

Disable bluetooth.

```
esp_err_t esp_bluedroid_disable(void)
```

Includes:

- #include <esp_bt_main.h>

esp_bluedroid_enable

Enable bluetooth.

```
esp_err_t esp_bluedroid_enable(void)
```

Includes:

- #include <esp_bt_main.h>

esp_bluedroid_init

Initialize and enable bluetooth resources.

```
esp_err_t esp_bluedroid_init(void)
```

Includes:

- #include <esp_bt_main.h>

esp_bt_controller_init

```
esp_err_t esp_bt_controller_init(esp_bt_controller_config_t *cfg)
```

Typically, we will code this as:

```
esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
ret = esp_bt_controller_init(&bt_cfg);
```

Includes:

- #include <bt.h>

esp_bt_controller_enable

Enable the bluetooth controller.

```
esp_err_t esp_bt_controller_enable(esp_bt_mode_t mode)
```

The value of `mode` can be one of:

- `ESP_BT_MODE_ILDE` – Not run.
- `ESP_BT_MODE_BLE – BLE` mode.
- `ESP_BT_MODE_CLASSIC_BT` – Classic mode.
- `ESP_BT_MODE_BTDM` – Dual mode.

Note: Currently only `ESP_BT_MODE_BTDM` is supported.

Includes:

- #include <bt.h>

esp_vhci_host_check_send_available

```
bool esp_vhci_host_check_send_available()
```

Includes:

- #include <bt.h>

esp_vhci_host_register_callback

```
void esp_vhci_host_register_callback(const esp_vhci_host_callback_t *callback)
```

Includes:

- #include <bt.h>

esp_vhci_host_send_packet

```
void esp_vhci_host_send_packet(uint8_t *data, uint16_t len)
```

Includes:

- #include <bt.h>

Upgrade APIs

system_upgrade_flag_check

Retrieve the upgrade status flag.

```
uint8 system_upgrade_flag_check()
```

The returned value will be one of:

- `UPGRADE_FLAG_IDLE` –
- `UPGRADE_FLAG_START` –
- `UPGRADE_FLAG_FINISH` –

system_upgrade_flag_set

Set the upgrade status flag.

```
void system_upgrade_flag_set(uint8 flag)
```

The flag can be one of:

- `UPGRADE_FLAG_IDLE` –
- `UPGRADE_FLAG_START` –
- `UPGRADE_FLAG_FINISH` –

system_upgrade_reboot

Reboot the ESP8266 and run the new firmware.

```
void system_upgrade_reboot()
```

system_upgrade_start

Start downloading the new firmware from the server.

```
bool system_upgrade_start(struct upgrade_server_info *server)
```

The server parameter is a structure ...

system_upgrade_userbin_check

Determine which of the two possible firmware images can be upgraded.

```
uint8 system_upgrade_userbin_check()
```

The result will be either `UPGRADE_FW_BIN1` or `UPGRADE_FW_BIN2`.

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 requests.

See also:

- Working with SNTP

sntp_enabled

Determine whether or not SNTP is enabled.

```
u8_t sntp_enabled()
```

Includes:

- #include <apps/sntp/sntp.h>

sntp_getoperatingmode

Retrieve the current operating mode.

```
u8_t sntp_getoperatingmode()
```

The operating mode returned can be one if:

- `SNTP_OPMODE_POLL` – This is the default.
- `SNTP_OPMODE_LISTENONLY` –

Includes:

- #include <apps/sntp/sntp.h>

sntp_getserver

```
ip_addr_t sntp_getserver(u8_t idx)
```

Includes:

- #include <apps/sntp/sntp.h>

sntp_getservername

Get the hostname of a target SNTP server.

```
char *sntp_setservername(u8_t index)
```

Retrieve the host name 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:

- #include <apps/sntp/sntp.h>

See also:

- Working with SNTP

sntp_init

```
void sntp_init()
```

Initialize the SNTP functions. Prior to calling this function, set the operating mode and server name.

Includes:

- #include <apps/sntp/sntp.h>

See also:

- Working with SNTP
- sntp_stop

sntp_servermode_dhcp

Enable use of DHCP for SNTP server location.

```
sntp_servermode_dhcp(int setServersFromDhcp)
```

The `setServersFromDhcp` is a flag that should be 1 or 0. If set to 1, then we are requesting that the servers to be used for SNTP should be requested from the DHCP server.

Includes:

- #include <apps/sntp/sntp.h>

sntp_setoperatingmode

Set the operating mode of SNTP access.

```
void sntp_setoperatingmode(u8_t operatingMode)
```

Operating mode can be one of:

- `SNTP_OPMODE_POLL` – This is the default.
- `SNTP_OPMODE_LISTENONLY` –

Includes:

- #include <apps/sntp/sntp.h>

sntp_setserver

Set the address of an SNTP server.

```
void sntp_serverserver(u8_t 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:

- #include <apps/sntp/sntp.h>

See also:

- Working with SNTP

sntp_setservername

Set the host name of a target SNTP server.

```
void sntp_setservername(u8_t index, char *server)
```

Specify an SNTP server by its host name.

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 instance.

Includes:

- #include <apps/sntp/sntp.h>

See also:

- Working with SNTP

sntp_stop

Stop SNTP processing.

```
void sntp_stop()
```

Includes:

- #include <apps/sntp/sntp.h>

See also:

- Working with SNTP

Generic TCP/UDP APIs

ipaddr_addr

Build a TCP/IP address from a dotted decimal string representation.

```
uint32 ipaddr_addr(char *addressString)
```

Return an IP address (4 byte) value from a dotted decimal string representation supplied in the `addressString` parameter. Note that the uint32 type is **not** assignable to the addresses in an esp_tcp or esp_udp structure. Instead we have to use a local variable and then copy the content. For example:

```
uint32 addr = ipaddr_addr(server);
memcpy(m_tcp.remote_ip, &addr, 4);
```

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:

- Error: Reference source not found

IP2STR

Generate four int values used in a `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:

```
printf("%d.%d.%d.%d", IP2STR(&addr));
```

The macro `IPSTR` can be used in place of 4 "%d" macros:

```
printf(IPSTR, IP2STR(&addr));
```

Includes:

- tcpip_adapter.h

See also:

- tcpip_adapter_get_ip_info

MAC2STR

Generate 6 values for a `printf` format.

```
MAC2STR(char *[6])
```

Use this in conjunction with the `MACSTR` macro. Here is an example:

```
char mac[6];
…
printf(MACSTR, MAC2STR(mac));
```

Includes:

- rom/ets_sys.h

TCP Adapter APIs

The TCP/IP adapter.

tcpip_adapter_ap_input

Receive low level data.

```
esp_err_t tcpip_adapter_ap_input(
   void *buffer,
   uint16_t len,
   void *eb)
```

This function is exposed but somewhat of a mystery. It appears to be related to getting data from the WiFi layer to be supplied to the TCP/IP layer but we have no knowledge about when or how it might be used.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_create_ip6_linklocal

```
esp_err_t tcpip_adapter_create_ip6_linklocal(tcpip_adapter_if_t tcpip_if)
```

tcpip_adapter_dhcpc_get_status

Get the status of the DHCP client subsystem.

```
esp_err_t tcpip_adapter_dhcpc_get_status(
   tcpip_adapter_if_t tcpip_if,
   tcpip_adapter_dhcp_status_t *status)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The status can be one of:

- `TCPIP_ADAPTER_DHCP_STARTED` – DHCP client has started for this interface.
- `TCPIP_ADAPTER_DHCP_STOPPED` – DHCP client has stopped for this interface.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcpc_option

```
esp_err_t tcpip_adapter_dhcpc_option(
   tcpip_adapter_option_mode_t opt_op,
   tcpip_adapter_option_id_t opt_id,
   void *opt_val, uint32_t opt_len)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `opt_op` can be one of:

- `TCPIP_ADAPTER_OP_SET` – Set the option.
- `TCPIP_ADAPTER_OP_GET` – Get the option.

The `opt_id` can be one of:

- TCPIP_ADAPTER_ROUTER_SOLICITATION_ADDRESS
- TCPIP_ADAPTER_REQUESTED_IP_ADDRESS
- TCPIP_ADAPTER_IP_ADDRESS_LEASE_TIME
- TCPIP_ADAPTER_IP_REQUEST_RETRY_TIME

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcpc_start

Start the DHCP client on the specified interface.

```
esp_err_t tcpip_adapter_dhcpc_start(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

tcpip_adapter_dhcpc_stop

Stop the DHCP client on the specified interface.

```
esp_err_t tcpip_adapter_dhcpc_stop(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcps_get_status

Retrieve the status of the DHCP server subsystem.

```
esp_err_t tcpip_adapter_dhcps_get_status(
   tcpip_adapter_if_t tcpip_if,
   tcpip_adapter_dhcp_status_t *status)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `status` parameter returns the possible states of the DHCP subsystem which are:

- `TCPIP_ADAPTER_DHCP_INIT` – Initial state.
- `TCPIP_ADAPTER_DHCP_STARTED` – The DHCP service is running.
- `TCPIP_ADAPTER_DHCP_STOPPED` – The DHCP service is stopped.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcps_option

Set or get the options for the DHCP server

```
esp_err_t tcpip_adapter_dhcps_option(
   tcpip_adapter_option_mode_t optMode,
   tcpip_adapter_option_id_t optId,
   void *opt_val, uint32_t opt_len)
```

The `optMode` defines whether we are getting or setting an option. The possible values are:

- `TCPIP_ADAPTER_OP_SET` – Used to set an option.
- `TCPIP_ADAPTER_OP_GET` – Used to get an option.

The `optId` defines the option being set or retrieved. The currently defined values are:

- `TCPIP_ADAPTER_ROUTER_SOLICITATION_ADDRESS` –
- `TCPIP_ADAPTER_REQUESTED_IP_ADDRESS` –
- `TCPIP_ADAPTER_IP_ADDRESS_LEASE_TIME` –
- `TCPIP_ADAPTER_IP_REQUEST_RETRY_TIME` –

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcps_start

Start the DHCP server on the specified interface.

```
esp_err_t tcpip_adapter_dhcps_start(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_dhcps_stop

Stop the DHCP server on the specified interface.

```
esp_err_t tcpip_adapter_dhcps_stop(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_down

Stop the TCP/IP interface.

```
esp_err_t tcpip_adapter_down(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

tcpip_adapter_eth_input

```
esp_err_t tcpip_adapter_eth_input(void *buffer, uint16_t len, void *eb)
```

tcpip_adapter_free_sta_list

Release storage for the connected station list.

```
esp_err_t tcpip_adapter_free_sta_list(tcpip_adapter_sta_list_t *sta_list)
```

The storage was allocated with a call to `tcpip_adapter_get_sta_list()`.

Includes:

- #include <tcpip_adapter.h>

See also:

- tcpip_adapter_get_sta_list

tcpip_adapter_get_esp_if

```
esp_interface_t tcpip_adapter_get_esp_if(void *dev)
```

tcpip_adapter_get_hostname

Get the host name of the interface.

```
esp_err_t tcpip_adapter_get_hostname(tcpip_adapter_if, const char **hostname)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The hostname is the address of a pointer to a string. On successful return, the string pointer will now point to the null terminated string representing the host name of that interface.

Includes:

- #include <tcpip_adapter.h>

See also:

- tcpip_adapter_set_hostname

tcpip_adapter_get_ip_info

Get the current IP address information.

```
esp_err_t tcpip_adapter_get_ip_info(
   tcpip_adapter_if_t tcpip_if,
   tcpip_adapter_ip_info_t *ipInfo)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `ipInfo` contains:

- `ip` – The ip address of the interface.
- `netmask` – The network mask of the interface.
- `gw` – The default gateway of the interface.

Includes:

- #include <tcpip_adapter.h>

See also:

- IP2STR

tcpip_adapter_get_ip6_linklocal

```
esp_err_t tcpip_adapter_get_ip6_linklocal(tcpip_adapter_if_t tcpip_if, ip6_addr_t
*if_ip6);
```

tcpip_adapter_get_sta_list

Populate a list of `tcpip_adapter_sta_list_t` from data.

```
esp_err_t tcpip_adapter_get_sta_list(
   wifi_sta_list_t *sta_info,
   tcpip_adapter_sta_list_t **sta_list)
```

Populate a list of "`tcpip_adapter_sta_list_t`" from a list of "`wifi_sta_list_t`". The `tcpip_adapter_sta_list_t` contains:

```
  STAILQ_ENTRY(station_list) next
                 uint8_t [6] mac
                  ip4_addr_t ip
```

Here is an example of use:

```
wifi_sta_list_t *stations;
ESP_ERROR_CHECK(esp_wifi_get_station_list(&stations));
tcpip_adapter_sta_list_t *infoList;

ESP_ERROR_CHECK(tcpip_adapter_get_sta_list(stations, &infoList));
struct station_list *head = infoList;
while(infoList != NULL) {
   printf("mac: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x " IPSTR " %d\n",
      infoList->mac[0],infoList->mac[1],infoList->mac[2],
      infoList->mac[3],infoList->mac[4],infoList->mac[5],
      IP2STR(&(infoList->ip)),
      (uint32_t)(infoList->ip.addr));
   infoList = STAILQ_NEXT(infoList, next);
}
ESP_ERROR_CHECK(esp_adapter_free_sta_list(head));
ESP_ERROR_CHECK(esp_wifi_free_station_list());
```

Includes:

- #include <tcpip_adapter.h>

See also:

- esp_wifi_get_station_list

tcpip_adapter_get_wifi_if

```
wifi_interface_t tcpip_adapter_get_wifi_if(void *dev)
```

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_init

```
void tcpip_adapter_init(void)
```

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_set_hostname

Set the host name associated with the supplied interface.

```
esp_err_t tcpip_adapter_set_hostname(
   tcpip_adapter_if_t tcpip_if,
   char *hostname)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `hostname` is the host name to be associated with the interface. The maximum size of the host name is `TCPIP_HOSTNAME_MAX_SIZE`.

Includes:

- #include <tcpip_adapter.h>

See also:

- tcpip_adapter_get_hostname

tcpip_adapter_set_ip_info

Set our IP info.

```
esp_err_t tcpip_adapter_set_ip_info(
   tcpip_adapter_if_t tcpip_if,
   tcpip_adapter_ip_info_t *ip_info)
```

Set the IP address of the adapter. If we are setting the station address, then the DHCP client must be already stopped. If we are setting the access point interface address, then the DHCP server must be already stopped.

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `ipInfo` contains:

- ip
- netmask
- gw

Each of these fields are an "`ip4_addr_t`" which is itself a struct containing a field called "`addr`" which is a 32bit unsigned integer. One can set the value of such an address using the macro "`IP4_ADDR(ip4_addr_t *, int, int, int, int)`".

For example:

```
tcpip_adapter_ip_info_t ipInfo;
IP4_ADDR(&ipInfo.ip, 192,168,1,99);
```

Includes:

- #include <tcpip_adapter.h>

See also:

- The DHCP client

tcpip_adapter_sta_input

```
esp_err_t tcpip_adapter_sta_input(
   void *buffer,
   uint16_t len, void *eb)
```

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_start

Start the TCP/IP interface.

```
esp_err_t tcpip_adapter_start(
   tcpip_adapter_if_t tcpip_if,
   uint8_t *mac,
   tcpip_adapter_ip_info_t *ip_info)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

The `mac` contains the hardware mac address we are to use.

The `ipInfo` contains:

- ip
- netmask
- gw

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_stop

```
esp_err_t tcpip_adapter_stop(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

Includes:

- #include <tcpip_adapter.h>

tcpip_adapter_up

Bring up the TCP/IP interface.

```
esp_err_t tcpip_adapter_up(tcpip_adapter_if_t tcpip_if)
```

The `tcpip_if` is the interface is one of the following:

- `TCPIP_ADAPTER_IF_STA` – Station interface.
- `TCPIP_ADAPTER_IF_AP` – Access point interface.

Includes:

- #include <tcpip_adapter.h>

mdns

See also:

- Multicast Domain Name Systems

mdns_free

Stop and release resources for a previously initialized mDNS server.

```
void mdns_free(mdns_server_t *server)
```

The server parameter would have been previously created with a call to `mdns_init()`.

Includes:

- #include <mdns.h>

mdns_init

Initialize mDNS on an interface.

```
esp_err_t mdns_init(
   tcpip_adapter_if_t tcpip_if,
   mdns_server_t **server)
```

The `tcpip_if` is the interface on which mDNS is being initialized. The choices are `TCPIP_ADAPTER_IF_STA` or `TCPIP_ADAPTER_IF_AP`. The server is a pointer that is updated to point to an `mdns_server_t` data structure. That structure appears to be opaque.

Includes:

- #include <mdns.h>
- #include <tcpip_adapter.h>

mdns_query

Search for mDNS records of a given service or protocol.

```
size_t mdns_query(
   mdns_server_t *server,
   const char *service,
   const char *proto,
   uint32_t timeout)
```

If the timeout is 0, then `mdns_query_end()` must be called to explicitly end the search.

The return is the number of results found.

Includes:

- #include <mdns.h>

mdns_query_end

Terminate a query where no timeout was specified.

```
size_t mdns_query_end(mdns_server_t *server)
```

Includes:

- #include <mdns.h>

mdns_result_free

```
esp_err_t mdns_result_free(mdns_server_t *server)
```

Includes:

- #include <mdns.h>

mdns_result_get

Get a specific result by index.

```
const mdns_result_t * mdns_result_get(
   mdns_server_t *server,
   size_t num)
```

Includes:

- #include <mdns.h>

mdns_result_get_count

Get the number of results currently in memory.

```
size_t mdns_result_get_count(mdns_server_t * server)
```

Includes:

- #include <mdns.h>

mdns_service_add

Define a service we are advertizing when we are being an mDNS responder/server.

```
esp_err_t mdns_service_add(
   mdns_server_t *server,
   const char *service,
   const char *proto,
   uint16_t port)
```

The `server` is the value we got when we previously called `mdns_init()` to initialize ourselves as an mDNS responder.

The `port` is the port number on which our advertized service will be listening on for incoming connections.

We can call `mdns_service_remove()` to remove a service.

Includes:

- #include <mdns.h>

mdns_service_instance_set

```
esp_err_t mdns_service_instance_set(
   mdns_server_t *server,
   const char *service,
   const char *proto,
   const char *instance)
```

Includes:

- #include <mdns.h>

mdns_service_port_set

```
esp_err_t mdns_service_port_set(
   mdns_server_t *server,
   const char *service,
   const char *proto, uint16_t port)
```

Includes:

- #include <mdns.h>

mdns_service_remove

```
esp_err_t mdns_service_remove(
   mdns_server_t *server,
   const char *service,
   const char *proto)
```

We call `mdns_service_add()` to add a service.

Includes:

- #include <mdns.h>

mdns_service_remove_all

```
esp_err_t mdns_service_remove_all(mdns_server_t *server)
```

Includes:

- #include <mdns.h>

mdns_service_txt_set

```
esp_err_t mdns_service_txt_set(
   mdns_server_t *server,
   const char *service,
   const char *proto,
   uint8_t num_items,
   const char **txt)
```

The `txt` is an array of strings where each entry is of the form "name=value".

Includes:

- #include <mdns.h>

mdns_set_hostname

Set the host name of our mDNS server.

```
esp_err_t mdns_set_hostname(
   mdns_server_t *server,
   const char *hostname)
```

Includes:

- #include <mdns.h>

mdns_set_instance

Set the instance name for our mDNS server.

```
esp_err_t mdns_set_instance(
   mdns_server_t *server,
   const char *instance)
```

Includes:

- #include <mdns.h>

OTA

Over The Air (OTA) is the capability to save a new image in flash for subsequent execution on next boot/restart.

Here is how we believe OTA works:

1. We call esp_ota_begin() to start the beginning of an OTA image write

2. We receive data that is part of the image

3. We call esp_ota_write() to write the data we just read

4. Go back to 2 while there is more of the image to receive over the network

5. We call esp_ota_end() to flag the end of the OTA image write

6. We call esp_ota_set_boot_partition() to specify which partition will be booted on next reboot

esp_ota_begin

```
esp_err_t esp_ota_begin(
   const esp_partition_t* partition,
   size_t image_size,
   esp_ota_handle_t* out_handle)
```

Includes:

- esp_ota_ops.h

esp_ota_end

Finish the update and validate the image.

```
esp_err_t esp_ota_end(esp_ota_handle_t handle)
```

The `handle` comes from a previous call to `esp_ota_begin()`.

Includes:

- esp_ota_ops.h

esp_ota_get_boot_partition

Get the partition information of the current boot partition.

```
const esp_partition_t *esp_ota_get_boot_partition(void)
```

Includes:

- esp_ota_ops.h

esp_ota_set_boot_partition

Set the partition of the next partition to be booted.

```
esp_err_t esp_ota_set_boot_partition(const esp_partition_t *partition)
```

Includes:

- esp_ota_ops.h

esp_ota_write

```
esp_err_t esp_ota_write(
   esp_ota_handle_t handle,
   const void *data, size_t size)
```

The `handle` comes from a previous call to `esp_ota_begin()`.

Includes:

- esp_ota_ops.h

GPIO Driver

The gpio functions in the ESP32 are provided through the ESP-IDF. One must include the "`driver/gpio.h`" header.

See also:

- GPIOs
- Analog to Digital Conversion

gpio_config

Configure one or more GPIO settings in a single operation.

```
esp_err_t gpio_config(gpio_config_t *pGPIOConfig)
```

The `gpio_config_t` data structure contains:

```
       uint64_t pin_bit_mask
     gpio_mode_t mode
   gpio_pullup_t pull_up_en
 gpio_pulldown_t pull_down_en
 gpio_int_type_t intr_type
```

The `pin_bit_mask` defines which pins we are configuring. Constants are defined to assist us here. For example, if we are configuring GPIO16 and GPIO34 we can set the `pin_bit_mask` to `GPIO_SEL_16 | GPIO_SEL_34` which is the boolean "or" of the two constant values.

The `mode` is used to set the mode of all of the pins we are configuring. The allowable values are:

- `GPIO_MODE_INPUT`
- `GPIO_MODE_OUTPUT`
- `GPIO_MODE_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT`

The `pull_up_en` enables an internal pull-up resistor. The allowable values are:

- `GPIO_PULLUP_ENABLE`
- `GPIO_PULLUP_DISABLE`

The `pull_down_en` enables an internal pull-down resistor. The allowable values are:

- `GPIO_PULLDOWN_ENABLE`
- `GPIO_PULLDOWN_DISABLE`

The `intr_type` configures how interrupts are handled for the pin. The allowable values are:

- `GPIO_INTR_DISABLE`
- `GPIO_INTR_POSEDGE`
- `GPIO_INTR_NEGEDGE`
- `GPIO_INTR_ANYEDGE`
- `GPIO_INTR_LOW_LEVEL`
- `GPIO_INTR_HIGH_LEVEL`

Here is an example:

```
gpio_config_t gpioConfig;
gpioConfig.pin_bit_mask = (1 << 16) | (1 << 17);
gpioConfig.mode = GPIO_MODE_OUTPUT;
gpioConfig.pull_up_en = GPIO_PULLUP_DISABLE;
gpioConfig.pull_down_en = GPIO_PULLDOWN_DISABLE;
gpioConfig.intr_type = GPIO_INTR_DISABLE;
gpio_config(&gpioConfig);
```

Includes:

- #include <driver/gpio.h>

gpio_get_level

Retrieve the signal level on the pin.

```
int gpio_get_level(gpio_num_t gpioNum)
```

Get the signal level on the specified pin. Either 0 or 1. The direction of the pin should be set to be INPUT to retrieve a useful result.

Includes

- #include <driver/gpio.h>

See also:

- gpio_set_direction
- GPIOs

gpio_install_isr_service

Install a GPIO ISR service.

```
esp_err_t gpio_install_isr_service(int intr_alloc_flags)
```

Includes:

- #include <driver/gpio.h>

See also:

- gpio_uninstall_isr_service

gpio_intr_enable

Enable interrupts on the specified pin.

```
esp_err_t gpio_intr_enable(gpio_num_t gpioNum)
```

Enabling the interrupt on the pin will cause an interrupt trigger when the type of interrupt as specified by `gpio_set_intr_type()` occurs.

Includes

- #include <driver/gpio.h>

See also:

- GPIO Interrupt handling
- gpio_set_intr_type
- gpio_intr_disable

gpio_intr_disable

Disable interrupts on the specified pin.

```
esp_err_t gpio_intr_disable(gpio_num_t gpioNum)
```

Disabling the interrupt on the pin will prevent an interrupt trigger from being registered.

Includes

- #include <driver/gpio.h>

See also:

- gpio_intr_enable

gpio_isr_handler_add

Associate an ISR with a given pin.

```
esp_err_t gpio_isr_handler_add(
   gpio_num_t gpio_num,
   gpio_isr_t isr_handler,
   void*      args)
```

The `gpio_num` is the pin to which we are associating an interrupt handler. This can be one of the `GPIO_NUM_xx` values.

The `isr_handler` is the function we wish to invoke to process the interrupt. The handler does not need to be declared in instruction RAM unless the ISR is defined with the `ESP_INTR_FLAG_IRAM`.

The signature of an interrupt handler function is:

```
void func(void *args)
```

The `args` is a pointer that will be passed to the ISR.

Includes:

- #include <driver/gpio.h>

See also:

- gpio_isr_handler_remove
- GPIO Interrupt handling

gpio_isr_handler_remove

Remove a previously associated ISR handler.

```
esp_err_t gpio_isr_handler_remove(gpio_num_t gpio_num)
```

The `gpio_num` identifies the pin against which any ISR handler should be removed. This can be one of the `GPIO_NUM_xx` values.

Includes:

- #include <driver/gpio.h>

See also:

- gpio_isr_handler_add

gpio_isr_register

Register an interrupt handler.

```
esp_err_t gpio_isr_register(
   void (*fn)(void *), void *arg,
   int intr_alloc_flags,
   gpio_isr_handle_t handle)
```

The `fn` is a function that will be called to handle the interrupt. The implementation of this function must reside in instruction RAM (IRAM) as it must be handled as quickly as possible and we can't afford loading it from flash. The function must be defined with the "`IRAM_ATTR`" function to force it to be in IRAM.

For example:

```
static void IRAM_ATTR myISR(void *arg) {
   // Do something.
}
```

If we want to perform logging within the handler, we must use:

```
ESP_EARLY_LOGx()
```

The `arg` is a parameter passed into the interrupt handler function (`fn`).

The flags are flags that control the definition of the interrupt.

- ESP_INTR_FLAG_INTRDISABLED
- ESP_INTR_FLAG_IRAM
- ???

The `handle` is a handle used to reference this ISR and is populated on return from this call.

Note that this is only *one* of the techniques for ISR processing. The other is the combination of gpio_install_isr_service() and gpio_isr_handled_add().

Includes

- #include <driver/gpio.h>

See also:

- GPIO Interrupt handling

gpio_set_direction

Set the direction of a pin.

```
esp_err_t gpio_set_direction(gpio_num_t gpioNum, gpio_mode_t mode)
```

The `mode` is used to set the mode of the pin we are configuring. The allowable values are:

- `GPIO_MODE_INPUT`
- `GPIO_MODE_OUTPUT`
- `GPIO_MODE_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT`

Includes:

- #include <driver/gpio.h>

See also:

- GPIOs
- gpio_get_level
- gpio_set_level

gpio_set_intr_type

Set the interrupt type of a pin.

```
esp_err_t gpio_set_intr_type(gpio_num_t gpioNum, gpio_int_type_t intrType)
```

The `gpioNum` defines which pins is having its interrupt type changed. This can be one of the `GPIO_NUM_xx` values.

The `intr_type` configures how interrupts are handled for the pin. The allowable values are:

- `GPIO_INTR_ANYEDGE`
- `GPIO_INTR_DISABLE`
- `GPIO_INTR_NEGEDGE`
- `GPIO_INTR_POSEDGE`
- `GPIO_INTR_LOW_LEVEL`
- `GPIO_INTR_HIGH_LEVEL`

Includes

- #include <driver/gpio.h>

See also:

- gpio_isr_register

gpio_set_level

Set the level of a pin.

```
esp_err_t gpio_set_level(gpio_num_t gpioNum, uint32_t level)
```

Se the level of an output pin. The `level` should be either 0 or 1.

Includes:

- #include <driver/gpio.h>

See also:

- GPIOs
- gpio_set_direction
- gpio_get_level

gpio_set_pull_mode

Set the pullup/pulldown mode of the pin.

```
esp_err_t gpio_set_pull_mode(gpio_num_t gpioNum, gpio_pull_mode_t pull)
```

The allowable values for pull are:

- `GPIO_PULLUP_ONLY` – Set the pin to be pulled-up. If no active signal on the pin, it will register as high.
- `GPIO_PULLDOWN_ONLY` – Set the pin to be pulled-down. If no active signal on the pin, it will register as low.
- `GPIO_PULLUP_PULLDOWN` – Set the pin to be both pulled-up and pulled-down so that if there is no actual signal on the input, it will be ½ the reference voltage.
- `GPIO_FLOATING` – No pull-up or pull-down. It is not defined what signal will be read from this pin if there is no active signal.

Includes:

- #include <driver/gpio.h>

See also:

- Pull up and pull down settings

gpio_uninstall_isr_service

```
void gpio_uninstall_isr_service()
```

Includes:

- #include <driver/gpio.h>

See also:

- gpio_install_isr_service

gpio_wakeup_enable

Enable GPIO wake-up.

```
esp_err_t gpio_wakeup_enable(gpio_num_t gpioNum, gpio_int_type_t intrType)
```

Wake-up a sleeping device when an interrupt occurs. The allowable interrupt types are:

- GPIO_INTR_LOW_LEVEL
- GPIO_INTR_HIGH_LEVEL

Includes

- #include <driver/gpio.h>

See also:

- gpio_wakeup_disable

gpio_wakeup_disable

Disable GPIO wake-up.

```
esp_err_t gpio_wakeup_disable(gpio_num_t gpioNum)
```

Disable waking up on a pin interrupt.

Includes

- #include <driver/gpio.h>

See also:

- gpio_wakeup_enable

GPIO Low Level

gpio_init

Initialize GPIO.

```
void gpio_init()
```

This is a ROM exposed function. It should not be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_input_get

Retrieve a bit-mask of the values of the first 32 GPIOs (0-31).

```
uint32_t gpio_input_get()
```

Includes:

- #include <rom/gpio.h>

gpio_input_get_high

Retrieve a bit-mask of the values of the last 8 GPIOs (32-39).

```
uint32_t gpio_input_get_high()
```

Includes:

- #include <rom/gpio.h>

gpio_intr_ack

```
void gpio_intr_ack(uint32_t ackMask)
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes:

- rom/gpio.h

gpio_intr_ack_high

```
void gpio_intr_ack_high(uint32_t ackMask)
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_intr_handler_register

```
void gpio_intr_handler_register(
   gpio_intr_handler_fn_t fn,
   void *arg)
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_intr_pending

```
uint32_t gpio_intr_pending()
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_intr_pending_high

```
uint32_t gpio_intr_pending_high()
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_matrx_in

Map a physical pin to its logical mapping.

```
void gpio_matrix_in(
   uint32_t gpioNum,
   uint32_t signalIdx,
   bool     inv)
```

Here we can map a physical input pin to its logical mapping. This is the heart of the multiplexing function.

Two special physical gpio number are used. The value `0x30` means a constant low value while `0x38` means a constant high value.

If the `inv` flag is set then the incoming signal is logically inverted before processing.

Includes

- #include <rom/gpio.h>

See also:

- Pads and multiplexing

gpio_matrix_out

Set the signal output to a given gpio.

```
void gpio_matrix_out(
   uint32_t gpio,
   uint32_t signalIdx,
   bool outInv,
   bool oenInv)
```

The `gpio` is the gpio output number between 0 and 39. The `signalIdx` is the index of the signal to be sent to that gpio. To invert the signal set `outInv` to be 1. To invert the signal output enable set `oenInv` to be 1.

Includes

- #include <rom/gpio.h>

See also:

- Pads and multiplexing

gpio_output_set

Set GPIOs that need to be input, output, high or low on bulk

```
void gpio_output_set(
   uint32_t setMask,
   uint32_t clearMask,
   uint32_t enableMask,
   uint32_t disableMask)
```

Work with the first 32 GPIOs (0-31) specifying which ones are input, which ones are output, which ones should be left alone and which ones should be set high/low. This API lets us set a batch of GPIOs in one operation.

Includes

- #include <rom/gpio.h>

gpio_output_set_high

Set GPIOs that need to be input, output, high or low on bulk

```
void gpio_output_set_high(
   uint32_t setMask,
   uint32_t clearMask,
   uint32_t enableMask,
   uint32_t disableMask)
```

Work with the last 8 GPIOs (32-39) specifying which ones are input, which ones are output, which ones should be left alone and which ones should be set high/low. This API lets us set a batch of GPIOs in one operation.

Includes

- #include <rom/gpio.h>

gpio_pad_hold

```
void gpio_pad_hold(uint8_t gpioNum)
```

Includes

- #include <rom/gpio.h>

gpio_pad_pulldown

```
void gpio_pad_pulldown(uint8_t gpioNum)
```

Includes

- #include <rom/gpio.h>

gpio_pad_pullup

```
void gpio_pad_pullup(uint8_t gpioNum)
```

Includes

- #include <rom/gpio.h>

gpio_pad_select_gpio

Specify that a given pin should be used for GPIO.

```
void gpio_pad_select_gpio(uint8_t gpioNum)
```

The `gpioNum` is the identity of the GPIO (0-39).

Includes

- #include <rom/gpio.h>

See also:

- GPIOs

gpio_pad_set_drv

```
void gpio_pad_set_drv(uint8_t gpioNum, uint8_t drv)
```

Includes

- #include <rom/gpio.h>

gpio_pad_unhold

```
void gpio_pad_unhold(uint8_t gpioNum)
```

Includes

- #include <rom/gpio.h>

gpio_pin_wakeup_disable

```
void gpio_pin_wakeup_disable()
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

gpio_pin_wakeup_enable

```
void gpio_pin_wakeup_enable(uint32_t I, GPIO_INT_TYPE intrState)
```

This is a ROM exposed function. It is not expected to be called in normal applications.

Includes

- #include <rom/gpio.h>

Analog to Digital Conversion

See also:

- Analog to digital conversion

adc1_config_channel_atten

Change the attenuation value of the channel.

```
esp_err_t adc1_config_channel_atten(
   adc1_channel_t channel,
   adc_atten_t atten)
```

By default, the input voltage can be between 0V and 1V, however using the attenuation values, we can scale the voltage to be between 0-1.34V, 0-2V and 0-3.6V.

The `channel` can be between `ADC1_CHANNEL_0` and `ADC1_CHANNEL_7`.

The `atten` can be one of:

- ADC_ATTEN_0db – 1/1
- ADC_ATTEN_2_5db – 1/1.34

- ADC_ATTEN_6db – 1/2
- ADC_ATTEN_11db – 1/3.6

Includes:

- #include <driver/adc.h>

adc1_config_width

Set the ADC resolution.

```
esp_err_t adc1_config_width(adc_bits_width_t width_bit)
```

The `width_bit` value may be one of:

- ADC_WIDTH_9Bit
- ADC_WIDTH_10Bit
- ADC_WIDTH_11Bit
- ADC_WIDTH_12Bit

Includes:

- #include <driver/adc.h>

adc1_get_voltage

Read the digital value of the analog input.

```
int adc1_get_voltage(adc1_channel_t channel)
```

If the value <0 then an error was encountered otherwise the value is the digital value of the voltage on the channel. The `channel` can be between `ADC1_CHANNEL_0` and `ADC_CHANNEL_7`.

Includes:

- #include <driver/adc.h>

hall_sensor_read

```
int hall_sensor_read()
```

UART driver API

The ESP-IDF provides a driver for the UART interfaces.

See also:

- Working with UART/serial

uart_clear_intr_status

```
esp_err_t uart_clear_intr_status(
   uart_port_t uart_num,
   uint32_t clr_mask)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `clr_mask` represents the bits to be cleared.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_disable_intr_mask

```
esp_err_t uart_disable_intr_mask(
   uart_port_t uart_num,
   uint32_t disable_mask)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `disable_mask` represents the bits to be disabled.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_driver_delete

Un-install the UART driver.

```
esp_err_t uart_driver_delete(uart_port_t uart_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_driver_install

```
esp_err_t uart_driver_install(
   uart_port_t uart_num,
   int rx_buffer_size,
   int tx_buffer_size,
   int queue_size,
   void *uart_queue,
   int intr_alloc_flags)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `rx_buffer_size` is the size of the receiving ring buffer. This should be greater than the constant UART_FIFO_LEN.

The `tx_buffer_size` is the size of the transmitting ring buffer. If 0, then UART output will block until the data is sent.

The `queue_size` is the size of the event queue.

The `uart_queue`, if set to NULL, means no event queue. If we wish, we can specify the address of an instance of a FreeRTOS `QueueHandle_t` variable. If we specify that, then events related to the UART will be posted to the queue. We can then use FreeRTOS functions such as `xQueueReceive()` to read events. An event is a `uart_event_type_t` value that can be one of:

- `UART_DATA` – Data is available.
- `UART_BREAK` –
- `UART_BUFFER_FULL` –
- `UART_FIFO_OV` – A queue overflow was detected.
- `UART_FRAME_ERR` –
- `UART_PARITY_ERR` – A data parity error was detected.
- `UART_DATA_BREAK` –

The `intr_alloc_flags` are one or more flags or'd together. The value of `0` is a good default.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_disable_intr_mask

```
esp_err_t uart_disable_intr_mask(uart_port_t uart_num, uint32_t disable_mask)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_disable_pattern_det_intr

```
esp_err_t uart_disable_pattern_det_intr(uart_port_t uart_num)
```

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_disable_rx_intr

```
esp_err_t uart_disable_rx_intr(uart_port_t uart_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_disable_tx_intr

```
esp_err_t uart_disable_tx_intr(uart_port_t uart_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_enable_intr_mask

Enable interrupts for the UART.

```
esp_err_t uart_enable_intr_mask(
   uart_port_t uart_num,
   uint32_t enable_mask)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The enable mask are the bit fields set for the ESP32 register called `UART_INT_ENA_REG`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_enable_pattern_det_intr

```
esp_err_t uart_enable_pattern_det_intr(
   uart_port_t uart_num,
   char pattern_chr,
   uint8_t chr_num,
   int chr_tout,
   int post_idle,
   int pre_idle);
```

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_enable_rx_intr

```
esp_err_t uart_enable_rx_intr(uart_port_t uart_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_enable_tx_intr

```
esp_err_t uart_enable_tx_intr(
   uart_port_t uart_num,
   int enable, int thresh)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_flush

```
esp_err_t uart_flush(uart_port_t uart_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_baudrate

Get the current baud rate.

```
esp_err_t uart_get_baudrate(
   uart_port_t uart_num,
   uint32_t *baudrate)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `baud_rate` is the current transmission rate in bits per second.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_buffered_data_len

```
esp_err_t uart_get_buffered_data_len(uart_port_t uart_num, size_t * size)
```

Get the size of data buffered in the RX ring buffer waiting to be read.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_hw_flow_ctrl

```
esp_err_t uart_get_hw_flow_ctrl(
   uart_port_t uart_num,
   uart_hw_flowcontrol_t *flow_ctrl)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_parity

Get the current parity setting.

```
esp_err_t uart_get_parity(
   uart_port_t uart_num,
   uart_parity_t *parity_mode)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `parity_mode` can be one of `UART_PARITY_DISABLE`, `UART_PARITY_EVEN`, `UART_PARITY_ODD`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_stop_bits

```
esp_err_t uart_get_stop_bits(
   uart_port_t uart_num,
   uart_stop_bits_t *stop_bits)
```

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_get_word_length

Get the current word length of a transmission.

```
esp_err_t uart_get_word_length(
   uart_port_t uart_num,
   uart_word_length_t *data_bit)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The data_bit is the word length being used for a transmission. It can be one of `UART_DATA_5_BITS`, `UART_DATA_6_BITS`, `UART_DATA_7_BITS` or `UART_DATA_8_BITS`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_intr_config

```
esp_err_t uart_intr_config(
   uart_port_t uart_num,
   const uart_intr_config_t *intr_conf)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The intr_conf is a structure containing:

- intr_enable_mask
  - UART_RXFIFO_FULL_INT_ENA_M
  - UART_TXFIFO_EMPTY_INT_ENA_M

    - UART_PARITY_ERR_INT_ENA_M
    - UART_FRM_ERR_INT_ENA_M
    - UART_RXFIFO_OVF_INT_ENA_M
    - UART_DSR_CHG_INT_ENA_M
    - UART_CTS_CHG_INT_ENA_M
    - UART_BRK_DET_INT_ENA_M
    - UART_RXFIFO_TOUT_INT_ENA_M
    - UART_SW_XON_INT_ENA_M
    - UART_SW_XOFF_INT_ENA_M
    - UART_GLITCH_DET_INT_ENA_M
    - UART_TX_BRK_DONE_INT_ENA_M
    - UART_TX_BRK_IDLE_DONE_INT_ENA_M
    - UART_TX_DONE_INT_ENA_M
    - UART_RS485_PARITY_ERR_INT_ENA_M
    - UART_RS485_FRM_ERR_INT_ENA_M
    - UART_RS485_CLASH_INT_ENA_M
    - UART_AT_CMD_CHAR_DET_INT_ENA_M
- rxfifo_full_thresh
- rx_timeout_thresh
- txfifo_empty_intr_thresh

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_isr_free

`esp_err_t uart_isr_free(uart_port_t uart_num)`

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_isr_register

```
esp_err_t uart_isr_register(
   uart_port_t uart_num,
   void (*fn)(void*), void * arg,
   int intr_alloc_flags,
   uart_isr_handle_t *handle)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_param_config

Configure all the parameters for a UART.

```
esp_err_t uart_param_config(
   uart_port_t uart_num,
   const uart_config_t *uart_config)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Calling this function is a convenience for setting the properties of a UART without us having to set the properties one by one with a stream of other calls. Here we populate a structure and set all the properties in one shot.

```
uart_config_t myUartConfig;
myUartConfig.baud_rate = UART_BITRATE_115200;
myUartConfig.data_bits = UART_DATA_8_BITS;
myUartConfig.parity = UART_PARITY_DISABLE;
myUartConfig.stop_bits = UART_STOP_BITS_1;
myUartConfig.flow_ctrl = UART_HW_FLOWCTRL_DISABLE;
myUartConfig.rx_flow_ctrl_thresh = 120;
```

Typically we would call this before installing the driver.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

- uart_set_baudrate
- uart_set_word_length
- uart_set_parity
- uart_set_stop_bits
- uart_set_hw_flow_ctrl

uart_read_bytes

Read data from the UART.

```
int uart_read_bytes(
   uart_port_t uart_num,
   uint8_t *buf,
   uint32_t length,
   TickType_t ticks_to_wait)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Here we read data from the UART. The maximum number of bytes we will read is supplied by `length` and `buf` must point to a memory buffer of at least that capacity. The return from the function will be the number of bytes that were actually read. We can also specify the number of FreeRTOS ticks to wait should no data be available.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_baudrate

Set the current baud rate.

```
esp_err_t uart_set_baudrate(
   uart_port_t uart_no,
   uint32_t baud_rate)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `baud_rate` is the desired transmission rate in bits per second.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_dtr

```
esp_err_t uart_set_dtr(
   uart_port_t uart_num,
   int level)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_hw_flow_ctrl

```
esp_err_t uart_set_hw_flow_ctrl(
   uart_port_t uart_no,
   uart_hw_flowcontrol_t flow_ctrl,
   uint8_t rx_thresh)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_line_inverse

```
esp_err_t uart_set_line_inverse(
   uart_port_t uart_no,
   uint32_t inverse_mask)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_parity

Set the parity check values.

```
esp_err_t uart_set_parity(
   uart_port_t uart_no,
   uart_parity_t parity_mode)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `parity_mode` can be one of `UART_PARITY_DISABLE`, `UART_PARITY_EVEN`, `UART_PARITY_ODD`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_pin

Set the pins used by this UART.

```
esp_err_t uart_set_pin(
   uart_port_t uart_num,
   int tx_io_num,
   int rx_io_num,
   int rts_io_num,
   int cts_io_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `tx_io_num` is the pin number used for TX for this uart.

The `rx_io_num` is the pin number used for RX for this uart.

The `rts_io_num` is the pin number used for RTS for this uart.

The `cts_io_num` is the pin number used for CTS for this uart.

We can specify the value `UART_PIN_NO_CHANGE` as a value and what ever the UART thinks it is currently using will continue to be used.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_rts

```
esp_err_t uart_set_rts(uart_port_t uart_num, int level)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_stop_bits

Set the number of data bits in the uart.

```
esp_err_t uart_set_stop_bits(
   uart_port_t uart_no,
   uart_stop_bits_t bit_num)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `bit_num` is the number of stop bits in effect. It can be one of `UART_STOP_BITS_1`, `UART_STOP_BITS_1_5` or `UART_STOP_BITS_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_set_word_length

Set the number of bits in a transmission unit.

```
esp_err_t uart_set_word_length(
   uart_port_t uart_num,
   uart_word_length_t data_bit)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The data_bit is the word length of a transmission. It can be one of `UART_DATA_5_BITS`, `UART_DATA_6_BITS`, `UART_DATA_7_BITS` or `UART_DATA_8_BITS`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_tx_chars

```
int uart_tx_chars(
   uart_port_t uart_no,
   const char* buffer,
   uint32_t len)
```

The `uart_no` is the identity of the UART being transmitted over. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

The `buffer` is a pointer to data to be transmitted.

The `len` is the number of bytes to transmit.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_wait_tx_done

```
esp_err_t uart_wait_tx_done(
   uart_port_t uart_num,
   TickType_t ticks_to_wait)
```

The `uart_no` is the identity of the UART being changed. It can be one of `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_write_bytes

Write bytes through the UART.

```
int uart_write_bytes(
   uart_port_t uart_num,
   const char* src,
   size_t size)
```

The uart_no is the identity of the UART being changed. It can be one of UART_NUM_0, UART_NUM_1 or UART_NUM_2.

Here we write data through the UART. The number of bytes to be written is specified by size. That number of bytes will be read starting at the memory location pointed to by src.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

uart_write_bytes_with_break

```
int uart_write_bytes_with_break(
   uart_port_t uart_num,
   const char* src,
   size_t size,
   int brk_len)
```

The uart_no is the identity of the UART being changed. It can be one of UART_NUM_0, UART_NUM_1 or UART_NUM_2.

Includes:

- #include <drivers/uart.h>

See also:

- Working with UART/serial

UART low level APIs

uartAttach

Uart_Init

uart_div_modify

uart_buff_switch

uart_tx_switch

uart_baudrate_detect

uart_rx_one_char

Retrieve a character from the RX FIFO buffer if one is available.

```
STATUS uart_rx_one_char(uint8_t *rxChar)
```

Retrieve a single character from the serial input buffer. If a character was available, the return value is `OK` else it is `FAIL`. If a character was retrieved, it will be stored in `rxChar`.

Includes:

- #include <rom/uart.h>

See also:

- Working with UART/serial

uart_tx_wait_idle

uart_tx_flush

```
void uart_tx_flush(uint8_t uartNumber)
```

uart_tx_one_char

```
STATUS uart_tx_one_char(uint8_t txChar)
```

Output a char to printf channel.

Includes:

- rom/uart.h

uart_tx_one_char2

I2C APIs

See also:

- Using the ESP-IDF I2C driver

i2c_cmd_link_create

Create a command handle.

```
i2c_cmd_handle_t i2c_cmd_link_create()
```

The `i2c_cmd_handle_t` is an opaque data type and should be treated as such. Think of a command as a "future" request to send a message over the I2C bus. Because I2C is timing dependent, what we want to do is first build the message we wish to send and then actually send it. The thinking here is that if we were to build and send simultaneously, any interrupts that occurred would cause timing problems. In addition, the ESP32 has built-in hardware support for I2C transmissions but only if we can give it a "unit of transmission" as an atomic operation. The high level of operation is:

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (0x12 << 1) | I2C_MASTER_WRITE, 1 /* expect ack */);
i2c_master_write_byte(cmd, 0x34, 1);
i2c_master_stop(cmd);
i2c_master_cmd_begin(0, cmd, 0);
i2c_cmd_link_delete(cmd);
```

Note that we should create a `cmd_handle`, work with it and then delete it. Although not specified in the documentation, tests seem to show that once used, it should not be used again.

Includes:

- #include <driver/i2c.h>

See also:

- i2c_master_cmd_begin
- i2c_cmd_link_delete

i2c_cmd_link_delete

```
void i2c_cmd_link_delete(i2c_cmd_handle_t cmd_handle)
```

Includes:

- #include <driver/i2c.h>

See also:

- i2c_cmd_link_create

i2c_driver_delete

```
esp_err_t i2c_driver_delete(i2c_port_t i2c_num)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_driver_install

```
esp_err_t i2c_driver_install(
   i2c_port_t i2c_num,
   i2c_mode_t mode,
   size_t slv_rx_buf_len,
   size_t slv_tx_buf_len,
   int intr_alloc_flags)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

The `mode` can be one of:

- I2C_MODE_SLAVE
- I2C_MODE_MASTER

The `slv_rx_buf_len` defines the buffer for receiving if we are a slave.

The `slv_tx_buf_len` defines the buffer for transmitting if we are a slave.

The `intr_alloc_flags` can be 0.

Includes:

- #include <driver/i2c.h>

See also:

- i2c_driver_delete
- i2c_param_config

i2c_get_data_mode

```
esp_err_t i2c_get_data_mode(
   i2c_port_t i2c_num,
   i2c_trans_mode_t *tx_trans_mode,
   i2c_trans_mode_t *rx_trans_mode)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_get_data_timing

```
esp_err_t i2c_get_data_timing(
   i2c_port_t i2c_num,
   int *sample_time,
   int *hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2s_get_period

```
esp_err_t i2c_get_period(
   i2c_port_t i2c_num,
   int *high_period,
   int *low_period)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_get_start_timing

```
esp_err_t i2c_get_start_timing(
   i2c_port_t i2c_num,
   int *setup_time,
   int *hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_get_stop_timing

```
esp_err_t i2c_get_stop_timing(
   i2c_port_t i2c_num,
   int *setup_time,
   int* hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- driver/i2c.h

i2c_isr_free

```
esp_err_t i2c_isr_free(intr_handle_t handle)
```

Includes:

- #include <driver/i2c.h>

i2c_isr_register

```
esp_err_t i2c_isr_register(
   i2c_port_t i2c_num,
   void (*fn)(void*), void * arg,
   int intr_alloc_flags,
   intr_handle_t *handle)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_master_cmd_begin

Process an I2C sequence of commands.

```
esp_err_t i2c_master_cmd_begin(
   i2c_port_t i2c_num,
   i2c_cmd_handle_t cmd_handle,
   portBASE_TYPE ticks_to_wait)
```

The i2c_num can be one of:

- I2C_NUM_0
- I2C_NUM_1

The cmd_handle should be a cmd_handle that has been previously created and then populated with the I2C commands/data to execute.

The return codes include:

- ESP_OK (0) – Command stream processed.
- ESP_ERR_INVALID_ARG (0x102) – Invalid parameter.
- ESP_FAIL (-1)– Command error; no ACK received from slave device.
- ESP_ERR_INVALID_STATE (0x103) – Driver not installed or not an I2C master.
- ESP_ERR_TIMEOUT (0x107) – Timed out because the bus was busy.

Includes:

- #include <driver/i2c.h>

See also:

- i2c_cmd_link_create
- i2c_master_write
- i2c_master_write_byte

i2c_master_read

Read a sequence of bytes.

```
esp_err_t i2c_master_read(
   i2c_cmd_handle_t cmd_handle,
   uint8_t *data,
   size_t data_len,
   int ack)
```

The cmd_handle is a handle that was previously created with an i2c_cmd_link_create() call.

The data is a buffer into which the read I2C bytes will be stored.

The data_len is the number of bytes we wish to read from the I2C.

The ack is a flag which indicates whether or not the ESP32 should respond with an I2C ACK upon reading a byte. Take care here. The value 0 means DO send an ACK while the value 1 means DO NOT send an ACK.

Includes:

- #include <driver/i2c.h>

i2c_master_read_byte

Read a byte from the I2C bus.

```
esp_err_t i2c_master_read_byte(
   i2c_cmd_handle_t cmd_handle,
   uint8_t *data,
   int ack)
```

The `cmd_handle` is a handle that was previously created with an `i2c_cmd_link_create()` call.

The `data` is a buffer into which the read I2C byte will be stored.

The `ack` is a flag which indicates whether or not the ESP32 should respond with an I2C ACK upon reading a byte. Take care here. The value 0 means DO send an ACK while the value 1 means DO NOT send an ACK.

Includes:

- #include <driver/i2c.h>

i2c_master_start

Queue command for I2C master to generate a start signal.

```
esp_err_t i2c_master_start(i2c_cmd_handle_t cmd_handle)
```

The start signal is used at the beginning of an I2C transmission and is followed by the address of the slave with which we wish to communicate. For examplem in the following we create a command stream, indicate that we are starting a new transaction and then supply the address. Note that an address is 7 bits of address data plus an indication of whether it is a read or write request we are making.

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (0x12 << 1) | I2C_MASTER_WRITE, 1 /* expect ack */);
```

Includes:

- #include <driver/i2c.h>

See also:

- i2c_master_cmd_begin

i2c_master_stop

Queue the command to indicate an I2C protocol stop.

```
esp_err_t i2c_master_stop(i2c_cmd_handle_t cmd_handle)
```

Includes:

- #include <driver/i2c.h>

i2c_master_write

Write a sequence of bytes as a master.

```
esp_err_t i2c_master_write(
   i2c_cmd_handle_t cmd_handle,
   uint8_t *data,
   size_t data_len,
   bool ack_en)
```

The `data` is a pointer to the buffer of data to send. This data is **not** copied and must be preserved until after a transmission using `i2c_master_cmd_begin()`.

The `data_len` is the length of the data to send.

The `ack_en` is used to flag whether we are looking for an acknowledgment. When the master (the ESP32) transmits 8 bits of data, we should expect to see the slave acknowledge receipt. Setting the ack_en flag to true causes the driver to validate that the transmission worked. Setting it to false bypasses any check for an acknowledgment.

Includes:

- #include <driver/i2c.h>

i2c_master_write_byte

Write a byte as a master.

```
esp_err_t i2c_master_write_byte(
   i2c_cmd_handle_t cmd_handle,
   uint8_t data,
   bool ack_en)
```

The `data` is the byte to write.

The `ack_en` is used to flag whether we are looking for an acknowledgment. When the master (the ESP32) transmits 8 bits of data, we should expect to see the slave acknowledge receipt. Setting the ack_en flag to true causes the driver to validate that the transmission worked. Setting it to false bypasses any check for an acknowledgment.

Includes:

- #include <driver/i2c.h>

i2c_param_config

```
esp_err_t i2c_param_config(
   i2c_port_t i2c_num,
   i2c_config_t *i2c_conf)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

The i2c_conf is a structure containing:

- `i2c_mode_t mode` – One of
  - I2C_MODE_SLAVE
  - I2C_MODE_MASTER
- gpio_num_t sda_io_num
- gpio_pullup_t sda_pullup_en
- gpio_num_t scl_io_num
- gpio_pullup_t scl_pullup_en
- master
  - uint32_t clk_speed
- slave
  - uint8_t addr_10bit_en
  - uint16_t slave_addr

Includes:

- #include <driver/i2c.h>

See also:

- i2c_driver_install

i2c_reset_rx_fifo

```
esp_err_t i2c_reset_rx_fifo(i2c_port_t i2c_num)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_reset_tx_fifo

```
esp_err_t i2c_reset_tx_fifo(i2c_port_t i2c_num)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_set_data_mode

```
esp_err_t i2c_set_data_mode(
   i2c_port_t i2c_num,
   i2c_trans_mode_t tx_trans_mode,
   i2c_trans_mode_t rx_trans_mode)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Both the `tx_trans_mode` and `rx_trans_mode` can be one of:

- I2C_DATA_MODE_MSB_FIRST
- I2C_DATA_MODE_LSB_FIRST

Includes:

- #include <driver/i2c.h>

i2c_set_data_timing

```
esp_err_t i2c_set_data_timing(
   i2c_port_t i2c_num,
   int sample_time,
   int hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_set_period

```
esp_err_t i2c_set_period(
   i2c_port_t i2c_num,
   int high_period,
   int low_period)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_set_pin

```
esp_err_t i2c_set_pin(
   i2c_port_t i2c_num,
   gpio_num_t sda_io_num,
   gpio_num_t scl_io_num,
   gpio_pullup_t sda_pullup_en,
   gpio_pullup_t scl_pullup_en,
   i2c_mode_t mode)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_set_start_timing

```
esp_err_t i2c_set_start_timing(
   i2c_port_t i2c_num,
   int setup_time,
   int hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_set_stop_timing

```
esp_err_t i2c_set_stop_timing(
   i2c_port_t i2c_num,
   int setup_time,
   int hold_time)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_slave_read_buffer

```
int i2c_slave_read_buffer(
   i2c_port_t i2c_num,
   uint8_t *data,
   size_t max_size,
   portBASE_TYPE ticks_to_wait)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

i2c_slave_write_buffer

```
int i2c_slave_write_buffer(
   i2c_port_t i2c_num,
   uint8_t *data,
   int size,
   portBASE_TYPE ticks_to_wait)
```

The `i2c_num` can be one of:

- I2C_NUM_0
- I2C_NUM_1

Includes:

- #include <driver/i2c.h>

SPI APIs

The ESP-IDF provides a driver for the SPI interface.

See also:

- Using the ESP-IDF SPI driver

spi_bus_add_device

Define a new device to SPI.

```
esp_err_t spi_bus_add_device(
   spi_host_device_t              host,
   spi_device_interface_config_t* dev_config,
   spi_device_handle_t*           handle)
```

When we wish to interact with a device, we need to describe it to our environment. This returns a handle which we can subsequently use. When we no longer need the device, we can release resources associated with it via a call to `spi_bus_remove_device()`. Core within the characteristics that are supplied with this call are the clock speed with which we will use when communicating and, if needed, which pin we want to use for slave select (CS). The queue size must also be supplied and must be at least 1.

Host can be one of:

- SPI_HOST (0) – Do not use.
- HSPI_HOST (1)
- VSPI_HOST (2)

The `dev_config` is a structure that contains:

- `uint8_t address_bits` – Number of bits to send in address action (0-64).
- `uint8_t command_bits` – Number of bits to send in command action (0-16).
- `uint8_t dummy_bits` – Number of filler bits to send between address and data.

- uint8_t mode – SPI mode (0-3)
- uint8_t duty_cycle_pos – default 0.
- uint8_t cs_ena_pretrans – Number of bit cycles CS should be activated before transmission.
- uint8_t cs_ena_posttrans – Number of bit cycles CS should stay active after transmission.
- int clock_speed_hz – Clock speed in Hz.
- int spics_io_num – CS gpio pin for device or -1 if not used.
- uint32_t flags – Bitwise OR of SPI_DEVICE_* flags.
    - SPI_DEVICE_TXBIT_LSBFIRST
    - SPI_DEVICE_RXBIT_LSBFIRST
    - SPI_DEVICE_3WIRE
    - SPI_DEVICE_POSITIVE_CS
    - SPI_DEVICE_HALFDUPLEX
    - SPI_DEVICE_CLK_AS_CS
- int queue_size – Transaction queue size. This is the number of concurrent SPI requests that may be in flight at one time. It must be at least 1.
- transaction_cb_t pre_cb – Callback function to be invoked before a transmission is started.
- transaction_cb_t post_cb – Callback function to be invoked after a transmission has completed.

The handle is the handle you can subsequently use for interacting with the device.

Includes:

- #include <driver/spi_master.h>

See also:

- spi_bus_initialize
- spi_bus_remove_device

spi_bus_free

Free up a bus.

```
esp_err_t spi_bus_free(spi_host_device_t host)
```

The `host` can be one of:

- SPI_HOST (0) – Do not use.
- HSPI_HOST (1)
- VSPI_HOST (2)

Includes:

- #include <driver/spi_master.h>

See also:

- spi_bus_initialize

spi_bus_initialize

Initialize an SPI Bus.

```
esp_err_t spi_bus_initialize(
   spi_host_device_t host,
   spi_bus_config_t *bus_config,
   int dma_chan)
```

The ESP32 provides multiple SPI bus instances. We use this API to initialize one of these buses. A bus is typically initialized only once during the life of an ESP32 application. Primarily, the initialization identifies the externalized GPIO pins that will be used for the SPI functions core of which are CLK, MOSI and MISO.

Host can be one of:

- SPI_HOST (0) – Do not use. Used internally.
- HSPI_HOST (1)
- VSPI_HOST (2)

The `bus_config` is a structure containing:

- `int mosi_io_num` – GPIO for MOSI or -1 if not used.
- `int miso_io_num` – GPIO for MISO or -1 if not used.
- `int sclk_io_num` – GPIO pin for CLK or -1 if not used.
- `int quadwp_io_num` – Specify -1 if not used.
- `int quadhd_io_num` – Specify -1 if not used.
- `int max_transfer_size` – Maximum transfer size in bytes. Defaults to 4094 if 0.

The `dma_chan` is which of the two possible DMA channels to use … either 1 or 2.

Includes:

- #include <driver/spi_master.h>

See also:

- spi_bus_free
- spi_bus_add_device

spi_bus_remove_device

Release the resources of a previously registered device.

```
esp_err_t spi_bus_remove_device(spi_device_handle_t handle)
```

This function can be called to release the resources allocated by a previous call to `spi_bus_add_device()`. Following this call, no further transmission requests should be made using the now defunct handle.

Includes:

- #include <driver/spi_master.h>

See also:

- spi_bus_add_device

spi_device_get_trans_result

Retrieve results from previous queued transmissions.

```
esp_err_t spi_device_get_trans_result(
   spi_device_handle_t handle,
   spi_transaction_t **trans_desc,
   TickType_t ticks_to_wait)
```

When we invoke spi_device_queue_trans() we are requesting a transmission to occur at some time in the future. Ideally as soon as possible. This function retrieves (if available) the results from previously queued transmissions.

Includes:

- #include <driver/spi_master.h>

See also:

- spi_device_queue_trans
- spi_device_transmit

spi_device_queue_trans

Queue a request for transmission.

```
esp_err_t spi_device_queue_trans(
   spi_device_handle_t handle,
   spi_transaction_t *trans_desc,
   TickType_t ticks_to_wait)
```

This function indicates that we have a transmission to be executed against an SPI device but that we don't need to wait for a response. Rather the request will be executed when it can and the response will subsequently be available for retrieval using the function `spi_device_get_trans_result()`.

The `trans_desc` is a description of a transaction. It contains:

- `uint32_t flags` – Bit wise OR of the `SPI_TRANS_*` flags:
  - SPI_MODE_DIO
  - SPI_MODE_QIO
  - SPI_MODE_DIOQIO_ADDR
  - SPI_USE_RXDATA
  - SPI_USE_TXDATA
- `uint16_t command` – Command data. Number of bits sent is defined by commands_bits value defined in `spi_device_interface_config_t`.
- `uint64_t address` – Address data. Number of bits sent is defined by address_bits value defined in `spi_device_interface_config_t`.
- `size_t length` – Total data length to send and receive in bits.
- `size_t rxlength` – Length of data to receive (in bits). If 0 is supplied, then we use the what ever value is defined in length.
- `void *user` – User defined context data.
- A union of:
  - `const void *tx_buffer` – Pointer to data used to hold data to be transmitted.
  - `uint8_t tx_data[4]` – Used if SPI_USE_TXDATA is flagged.
- A union of:
  - `void *rx_buffer` – Pointer to data used to hold data received.
  - `uint8_t rx_data[4]` – Used if SPI_USE_RXDATA is flagged.

Includes:

- #include <driver/spi_master.h>

See also:

- spi_device_get_trans_result
- spi_device_transmit

spi_device_transmit

Perform a transmission over the SPI bus and wait for a response.

```
esp_err_t spi_device_transmit(
   spi_device_handle_t handle,
   spi_transaction_t*  trans_desc)
```

Logically, this function is an amalgamation of `esp_device_queue_trans()` followed by a call to `spi_device_get_trans_result()` effectively resulting in a synchronous request and block for response of an SPI request.

The `trans_desc` is a description of a transaction. It contains:

- `uint32_t flags` – Bit wise OR of the SPI_TRANS_* flags:
    - SPI_MODE_DIO
    - SPI_MODE_QIO
    - SPI_MODE_DIOQIO_ADDR
    - SPI_USE_RXDATA
    - SPI_USE_TXDATA
- `uint16_t command` – Command data. Number of bits sent is defined by commands_bits value defined in `spi_device_interface_config_t`.
- `uint64_t address` – Address data. Number of bits sent is defined by address_bits value defined in `spi_device_interface_config_t`.
- `size_t length` – Total data length to send and receive in bits. **Note**: The size is in bits … **not** bytes.
- `size_t rxlength` – Length of data to receive (in bits). If 0 is supplied, then we use the what ever value is defined in length.
- `void *user` – User defined context data.
- A union of:
    - `const void *tx_buffer` – Pointer to data used to hold data to be transmitted.
    - `uint8_t tx_data[4]` – Used if `SPI_USE_TXDATA` is flagged.
- A union of:
    - `void *rx_buffer` – Pointer to data used to hold data received.

    - `uint8_t rx_data[4]` – Used if `SPI_USE_RXDATA` is flagged.

For example:

```
char data[3];
spi_transaction_t trans_desc;
trans_desc.address   = 0;
trans_desc.command   = 0;
trans_desc.flags     = 0;
trans_desc.length    = 3 * 8;
trans_desc.rxlength  = 0;
trans_desc.tx_buffer = data;
trans_desc.rx_buffer = data;

data[0] = 0x12;
data[1] = 0x34;
data[2] = 0x56;

ESP_ERROR_CHECK(spi_device_transmit(handle, &trans_desc));
```

Includes:

- #include <driver/spi_master.h>

See also:

- spi_device_get_trans_result
- spi_device_queue_trans

I2S APIs

i2s_driver_install

```
esp_err_t i2s_driver_install(
   i2s_port_t          i2s_num,
   const i2s_config_t* i2s_config,
   int                 queue_size,
   void*               i2s_queue)
```

i2s_driver_uninstall

```
esp_err_t i2s_driver_uninstall(i2s_port_t i2s_num)
```

i2s_pop_sample

```
int i2s_pop_sample(
   i2s_port_t i2s_num,
   char*      sample,
   TickType_t ticks_to_wait);
```

i2s_push_sample

```
int i2s_push_sample(
   i2s_port_t  i2s_num,
   const char* sample,
   TickType_t  ticks_to_wait)
```

i2s_read_bytes

```
int i2s_read_bytes(
   i2s_port_t i2s_num,
   char*      dest,
   size_t     size,
   TickType_t ticks_to_wait);
```

i2s_set_pin

```
esp_err_t i2s_set_pin(
   i2s_port_t             i2s_num,
   const i2s_pin_config_t* pin)
```

i2s_set_sample_rates

```
esp_err_t i2s_set_sample_rates(
   i2s_port_t i2s_num,
   uint32_t   rate)
```

i2s_start

```
esp_err_t i2s_start(i2s_port_t i2s_num)
```

i2s_stop

```
esp_err_t i2s_stop(i2s_port_t i2s_num)
```

i2s_write_bytes

```
int i2s_write_bytes(
   i2s_port_t  i2s_num,
   const char* src,
   size_t      size,
   TickType_t  ticks_to_wait);
```

i2s_zero_dma_buffer

```
esp_err_t i2s_zero_dma_buffer(i2s_port_t i2s_num);
```

RMT APIs

The RMT component is the "remote" controller. The word "remote" here means a remote control such as you would find for controlling your TV or stereo. The primary function is to create complex digital wave forms that will drive infrared LEDs to generate a PWM signal in the infrared spectrum. However, we can attach other signal consuming devices to the output of this data to drive them. A good example is neopixels.

See also:

- Remote Control Peripheral – RMT

rmt_clr_intr_enable_mask

Clear mask value to RMT interrupt enable register.

```
void rmt_clr_intr_enable_mask(uint32_t mask)
```

- `mask` – Bit mask to clear the register.

Includes:

- #include <driver/rmt.h>

rmt_config

```
esp_err_t rmt_config(rmt_config_t* rmt_param)
```

Configure an RMT channel. The `rmt_param` is a rich structure containing:

- `rmt_mode` (`rmt_mode_t`) – This can be one of
  - `RMT_MODE_TX` – Configure this channel for output (transmission).
  - `RMT_MODE_RX` – Configure this channel for input (reception).
- `channel` (`rmt_channel_t`) – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `clk_div` (`uint8_t`) – The clock divider value. If the base clock is 80MHz then a divisor of 8 gives us 10MHz, a divisor of 80 gives us 1MHz.
- `gpio_num` (`gpio_num_t`) – The gpio number associated with the function.
- `mem_block_num` (`uint8_t`) – Memory block number.
- Union of
  - `tx_config` – Configuration for transmission.
    - `loop_en` (`bool`) – True if loop output mode.
    - `carrier_en` (`bool`) – Is the carrier enabled.
    - `carier_freq_hz` (`uint32_t`) – Carrier frequency.
    - carrier_duty_percent (`uint8_t`)
    - carrier_level (`rmt_carrier_level_t`) – One of:
      - RMT_CARRIER_LEVEL_LOW
      - RMT_CARRIER_LEVEL_HIGH
    - `idle_output_en` (`bool`) – Idle level output enabled.
    - `idle_level` (`rmt_idle_level_t`) – The signal level when idle. One of:
      - RMT_IDLE_LEVEL_LOW
      - RMT_IDLE_LEVEL_HIGH
  - `rx_config` – Configuration information for input reception.

- `filter_en` (`bool`) – Should signal filtering be enabled.
- `filter_ticks_thresh` (`uint8_t`) – If signal filtering is enabled, this is the threshold below which transitions will be ignored.
- `idle_threshold` (`uint16_t`) – The duration of idleness after which the signal train will be considered complete. If we have a clock div of 255 (the maximum) and a threshold of 65535 (the maximum), this gives us a maximum idle period duration of about 209 msecs.

Includes:

- #include <driver/rmt.h>

rmt_driver_install

Initialize the RMT driver.

```
esp_err_t rmt_driver_install(
   rmt_channel_t channel,
   size_t rx_buf_size,
   int intr_alloc_flags)
```

- `channel` – The channel to initialize. A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `size` – The size of the RMT channel ring buffer. Can be 0.
- `intr_alloc_flags` – Normally 0.

Includes:

- #include <driver/rmt.h>

rmt_driver_uninstall

Uninstall the RMT driver.

```
esp_err_t rmt_driver_uninstall(rmt_channel_t channel)
```

- `channel` – The channel to un-install. A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.

Includes:

- #include <driver/rmt.h>

rmt_fill_tx_items

```
esp_err_t rmt_fill_tx_items(
   rmt_channel_t channel,
   rmt_item32_t* item,
   uint16_t item_num,
   uint16_t mem_offset)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `item` – Pointer to items.
- `item_num` – Number of items.
- `mem_offset` – Offset into memory

Includes:

- #include <driver/rmt.h>

rmt_get_clk_div

Get the RMT clock divider.

```
esp_err_t rmt_get_clk_div(
   rmt_channel_t channel,
   uint8_t* div_cnt)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `div_cnt` – The divider of the base clock which is 80MHz.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_set_clk_div

rmt_get_mem_block_num

Get the number of memory blocks used by this channel.

```
esp_err_t rmt_get_mem_block_num(
   rmt_channel_t channel,
   uint8_t* rmt_mem_num)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `rmt_mem_num` – A storage area that will be used to hold the number of memory blocks allocated to the channel.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_set_mem_block_num

rmt_get_mem_pd

Get memory low power mode.

```
esp_err_t rmt_get_mem_pd(rmt_channel_t channel, bool* pd_en)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `pd_en` – Pointer to receive the memory low power mode.

Includes:

- #include <driver/rmt.h>

rmt_get_memory_owner

Ge the memory owner of a channel's memory block.

```
esp_err_t rmt_get_memory_owner(
   rmt_channel_t channel,
   rmt_mem_owner_t* owner)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `owner` – A storage area that will be filled in with the owner of the memory block. One of:
  - `RMT_MEM_OWNER_TX`
  - `RMT_MEM_OWNER_RX`.

Includes:

- #include <driver/rmt.h>

rmt_get_ringbuf_handler

Get the ring-buffer handler.

```
esp_err_t rmt_get_ringbuf_handler(
   rmt_channel_t channel,
   RingbufHandle_t *buf_handle)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `buf_handle` – A handle to the ring buffer.

Includes:

- #include <driver/rmt.h>
- #include <freertos/ringbuf.h>

rmt_get_rx_idle_thresh

Get RMT idle threshold.

```
esp_err_t rmt_get_rx_idle_thresh(rmt_channel_t channel, uint16_t *thresh)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `thresh` – A pointer to storage into which the RX idle threshold will be copied.

Includes:

- #include <driver/rmt.h>

rmt_get_status

Get RMT status.

```
esp_err_t rmt_get_status(rmt_channel_t channel, uint32_t* status)
```

- channel
- status

Includes:

- #include <driver/rmt.h>

rmt_get_source_clk

Get the source clock.

```
esp_err_t rmt_get_source_clk(
   rmt_channel_t channel,
   rmt_source_clk_t* src_clk)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `src_clk` – The source of the clock. One of:
  - `RMT_BASECLK_REF` – A 1MHz clock (not supported).
  - `RMT_BASECLK_APB` – The APB clock at 80MHz.

Includes:

- #include <driver/rmt.h>

rmt_get_tx_loop_mode

Get tx loop mode.

```
esp_err_t rmt_get_tx_loop_mode(
   rmt_channel_t channel,
   bool* loop_en)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `loop_en` – The current value of the loop transmission state.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_set_tx_loop_mode

rmt_isr_deregister

De-register a previously registered interrupt handler.

```
esp_err_t rmt_isr_deregister(rmt_isr_handle_t handle)
```

- `handle` – A handle returned from a previous call to `rmt_isr_register()`.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_isr_register

rmt_isr_register

Handler for ISR.

```
esp_err_t rmt_isr_register(
   void (*fn)(void *), void * arg, int intr_alloc_flags, rmt_isr_handle_t *handle)
```

- `fn` – A function to be called when an interrupt occurs.
- `arg` – Parameters/context for the interrupt handler
- `intr_alloc_flags` – Interrupt allocation flags.
- `handle` – A handle returned used to refer to this registered interrupt handler.

The signature of the handler function is:

```
void func(void *args)
```

Includes:

- #include <driver/rmt.h>

See also:

- rmt_isr_deregister
- rmt_clr_intr_enable_mask
- rmt_set_err_intr_en
- rmt_set_intr_enable_mask
- rmt_set_rx_intr_en
- rmt_set_tx_thr_intr_en

rmt_memory_rw_rst

Reset TX/RX memory index.

```
esp_err_t rmt_memory_rw_rst(rmt_channel_t channel)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.

Includes:

- #include <driver/rmt.h>

rmt_rx_start

Start receiving data.

```
esp_err_t rmt_rx_start(rmt_channel_t channel, bool rx_idx_rst)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `rx_idx_rst` – Set to true to reset the receiver index.

Includes:

- #include <driver/rmt.h>

rmt_rx_stop

Stop receiving data.

```
esp_err_t rmt_rx_stop(rmt_channel_t channel)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.

Includes:

- #include <driver/rmt.h>

rmt_set_clk_div

Set RMT clock divider.

```
esp_err_t rmt_set_clk_div(
   rmt_channel_t channel,
   uint8_t div_cnt)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `div_cnt` – RMT counter clock divider.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_get_clk_div

rmt_set_err_intr_en

```
esp_err_t rmt_set_err_intr_en(rmt_channel_t channel, bool en)
```

Includes:

- #include <driver/rmt.h>

rmt_set_idle_level

```
esp_err_t rmt_set_idle_level(
   rmt_channel_t channel,
   bool idle_out_en,
   rmt_idle_level_t level)
```

- `idle_out_en` – Should an idle level be set.
- `level` – The idle level. One of:
  - `RMT_IDLE_LEVEL_LOW` – The value for a low level.
  - `RMT_IDLE_LEVEL_HIGH` – The value for a high level.

Includes:

- #include <driver/rmt.h>

rmt_set_intr_enable_mask

```
void rmt_set_intr_enable_mask(uint32_t mask)
```

Includes:

- #include <driver/rmt.h>

rmt_set_mem_block_num

Set the number of adjacent memory blocks used by the channel.

```
esp_err_t rmt_set_mem_block_num(
   rmt_channel_t channel,
   uint8_t blockCount)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `blockCount` – The number of memory blocks associated with the channel. Between 1 and 8.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_get_mem_block_num

rmt_set_mem_pd

Set RMT memory in low power mode.

```
esp_err_t rmt_set_mem_pd(rmt_channel_t channel, bool pd_en)
```

Includes:

- #include <driver/rmt.h>

rmt_set_memory_owner

Set the memory owner of a channels memory block.

```
esp_err_t rmt_set_memory_owner(
   rmt_channel_t channel,
   rmt_mem_owner_t owner)
```

- `channel` – The type of owner of the memory block associated with the channel.
- `owner` – One of `RMT_MEM_OWNER_TX` or `RMT_MEM_OWNER_RX`.

Includes:

- #include <driver/rmt.h>

rmt_set_pin

```
esp_err_t rmt_set_pin(
   rmt_channel_t channel,
   rmt_mode_t mode,
   gpio_num_t gpio_num)
```

- channel – A value between RMT_CHANNEL_0 and RMT_CHANNEL_7.
- node
- gpio_num

Includes:

- #include <driver/rmt.h>

rmt_set_rx_filter

```
esp_err_t rmt_set_rx_filter(
   rmt_channel_t channel,
   bool rx_filter_en,
   uint8_t thresh)
```

- channel – A value between RMT_CHANNEL_0 and RMT_CHANNEL_7.
- rx_filter_en – true to enable input filtering and false to disable.
- thresh – The threshold in clock ticks below which incoming signals should be discarded.

Includes:

- #include <driver/rmt.h>

rmt_set_rx_idle_thresh

Se the RMT RX idle threshold.

```
esp_err_t rmt_set_rx_idle_thresh(
   rmt_channel_t channel,
   uint16_t thresh)
```

- channel – A value between RMT_CHANNEL_0 and RMT_CHANNEL_7.
- thresh – The value in clocks ticks where the lack of a transition indicates a return to idle mode.

Includes:

- #include <driver/rmt.h>

See also:

- rmt_get_rx_idle_thresh

rmt_set_rx_intr_en

Enable or disable an interrupt for having received a signal train.

```
esp_err_t rmt_set_rx_intr_en(rmt_channel_t channel, bool en)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `en` – True to enable the interrupt, false to disable.

Includes:

- #include <driver/rmt.h>

rmt_set_tx_carrier

```
esp_err_t rmt_set_tx_carrier(
   rmt_channel_t channel,
   bool carrier_en,
   uint16_t high_level,
   uint16_t low_level,
   rmt_carrier_level_t carrier_level)
```

Includes:

- #include <driver/rmt.h>

rmt_set_tx_intr_en

```
esp_err_t rmt_set_tx_intr_en(
   rmt_channel_t channel,
   bool en)
```

Includes:

- #include <driver/rmt.h>

See also:

- rmt_set_err_intr_en
- rmt_set_rx_intr_en
- rmt_set_tx_thr_intr_en

rmt_set_tx_loop_mode

```
esp_err_t rmt_set_tx_loop_mode(
   rmt_channel_t channel,
   bool loop_en)
```

Includes:

- #include <driver/rmt.h>

See also:

- rmt_get_tx_loop_mode

rmt_set_tx_thr_intr_en

```
esp_err_t rmt_set_tx_thr_intr_en_en(
   rmt_channel_t channel,
   bool en,
   uint16_t evt_thresh)
```

Includes:

- #include <driver/rmt.h>

rmt_set_source_clk

Set the reference source clock.

```
esp_err_t rmt_set_source_clk(
   rmt_channel_t channel,
   rmt_source_clk_t base_clk)
```

- `channel` – A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `base_clk` – The source of the clock. One of:
    - `RMT_BASECLK_REF` – A 1MHz clock (not supported).
    - `RMT_BASECLK_APB` – The APB clock at 80MHz.

Includes:

- #include <driver/rmt.h>

rmt_tx_start

Start the channel transmitting.

```
esp_err_t rmt_tx_start(
   rmt_channel_t channel,
   bool tx_idx_rst)
```

- `channel` – The channel to start transmitting upon.
- `tx_idx_rst` – If true, the transmission will start at 1st memory block otherwise it will continue from where it was last stopped.

Includes:

- #include <driver/rmt.h>

rmt_tx_stop

Stop the channel transmitting.

```
esp_err_t rmt_tx_stop(rmt_channel_t channel)
```

- `channel` – The channel to stop transmitting.

Includes:

- #include <driver/rmt.h>

rmt_wait_tx_done

```
esp_err_t rmt_wait_tx_done(rmt_channel_t channel)
```

Includes:

- #include <driver/rmt.h>

rmt_write_items

Send wave forms.

```
esp_err_t rmt_write_items(
   rmt_channel_t channel,
   rmt_item32_t* rmt_item,
   int item_num,
   bool wait_tx_done)
```

- `channel` – The channel to transmit upon. A value between `RMT_CHANNEL_0` and `RMT_CHANNEL_7`.
- `rmt_item` – An array of items to transmit. This is a data type that is a union between:
  - `uint32_t val` – The value of an item as a whole.
  - structure of:
    - `uint32_t duration0` – The duration of the 1st part of the item.
    - `uint32_t level0` – The signal level of the 1st part of the item.
    - `uint32_t duration1` – The duration of the 2nd part of the item.
    - `uint32_t level1` – The signal level of the 2nd part of the item.
- `item_num` – Number of items to transmit.
- `wait_tx_done` – If true, will wait for transmission to complete.

It is vital to note that the array of items passed in is **not** copied. Instead the data is used for the duration of the writing. As such one can't release or otherwise modify the data until after the write is complete.

Includes:

- #include <driver/rmt.h>

LEDC/PWM APIs

This section discusses the driver APIs for the LEDC / PWM access.

See also:

- LEDC – Pulse Width Modulation – PWM

ledc_bind_channel_timer

```
esp_err_t ledc_bind_channel_timer(
   ledc_mode_t speedMode,
   uint32_t channel,
   uint32_t timerIdx)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

Includes:

- #include <driver/ledc.h>

ledc_channel_config

Configure a PWM channel.

```
esp_err_t ledc_channel_config(
   ledc_channel_config_t* ledcConf)
```

The `ledcConf` structure contains:

- `gpio_num` – The GPIO pin upon which the signal will appear.
- `speed_mode` – One of:
  - LEDC_HIGH_SPEED_MODE
- `channel` – One of:
  - LEDC_CHANNEL0 → LEDC_CHANNEL7
- `intr_type` – One of:

  - LEDC_INTR_DISABLE
  - LEDC_INTR_FADE_END
- `timer_sel` – One of:
  - LEDC_TIMER0 → LEDC_TIMER_3
- `duty` – Duty range. The duration of the high signal within any given period. The value is specified as the value of the timer after which the signal will drop low. Note that the granularity of a timer is specified when the timer is configured and can be between 10 and 15 bits. The "speed' of increments is defined by the frequency property of the timer.

Includes:

- #include <driver/ledc.h>

See also:

- ledc_timer_config

ledc_fade_func_install

```
esp_err_t ledc_fade_func_install(int intr_alloc_flags)
```

Includes:

- #include <driver/ledc.h>

ledc_fade_start

```
esp_err_t ledc_fade_start(
   ledc_channel_t  channel,
   ledc_fade_mode_t wait_done)
```

Includes:

- #include <driver/ledc.h>

ledc_fade_func_uninstall

```
void ledc_fade_func_uninstall()
```

Includes:

- #include <driver/ledc.h>

ledc_get_duty

Get the duty value for the channel.

```
int ledc_get_duty(
   ledc_mode_t    speedMode,
   ledc_channel_t channel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

Includes:

- #include <driver/ledc.h>

See also:

- ledc_set_duty
- ledc_channel_config
- ledc_update_duty

ledc_get_freq

Get the frequency for the timer.

```
uint32_t ledc_get_freq(
   ledc_mode_t  speedMode,
   ledc_timer_t timerNum)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `timerNum` can be one of:

- LEDC_TIMER_0
- LEDC_TIMER_1
- LEDC_TIMER_2
- LEDC_TIMER_3

Includes:

- #include <driver/ledc.h>

ledc_set_duty

Set the duty value for the channel.

```
esp_err_t ledc_set_duty(
   ledc_mode_t    speedMode,
   ledc_channel_t channel,
   uint32_t       duty)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

The `duty` value is set as a function of the number of bits set for the channel.

Includes:

- #include <driver/ledc.h>

See also:

- ledc_update_duty
- ledc_channel_config
- ledc_get_duty

ledc_isr_register

Register an ISR.

```
esp_err_t ledc_isr_register(
   uint32_t ledc_intr_num,
   void (*fn)(void *), void *arg)
```

Includes:

- #include <driver/ledc.h>

ledc_set_fade

Set the fade value.

```
esp_err_t ledc_set_fade(
   ledc_mode_t          speedMode,
   uint32_t             channel,
   uint32_t             duty,
   ledc_duty_duration_t graduleDirection,
   uint32_t             stepNum,
   uint32_t             dutyCycleNum,
   uint32_t             dutyScale)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` is the channel against which we are setting the fading.

The `duty` is the start value of the duty against which we will then fade.

The `graduleDuration` is the direction of the fade. It may be one of:

- `LEDC_DUTY_DIR_INCREASE` – Increase the duty value over time.
- `LEDC_DUTY_DIR_DECREASE` – Decrease the duty value over time.

The `dutyScale` is the amount that the duty value is changed each cycle.

The `dutyCycleNum` is the number of clock cycles that pass before the duty value of the channel is changed by the `dutyScale` amount.

The `stepNum` is the count of the number of times the duty value will change before the fade cycle is considered complete.

Includes:

- #include <driver/ledc.h>

ledc_set_fade_with_step

```
esp_err_t ledc_set_fade_with_step(
   ledc_mode_t    speed_mode,
   ledc_channel_t channel,
   int            target_duty,
   int            scale,
   int            cycle_num)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

Includes:

- #include <driver/ledc.h>

ledc_set_fade_with_time

Fade from current value to target valie.

```
esp_err_t ledc_set_fade_with_time(
   ledc_mode_t    speed_mode,
   ledc_channel_t channel,
   int            target_duty,
   int            max_fade_time_ms)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

The `target_duty` is the target duty value we want to reach relative to the current value.

The `max_fade_time_ms` is the time it should take us to reach the duty value.

Includes:

- #include <driver/ledc.h>

ledc_set_freq

Set the frequency for the timer.

```
esp_err_t ledc_set_freq(
   ledc_mode_t  speedMode, // LEDC_HIGH_SPEED_MODE only
   ledc_timer_t timerNum,
   uint32_t     freqHz)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `timerNum` can be one of:

- LEDC_TIMER_0
- LEDC_TIMER_1
- LEDC_TIMER_2
- LEDC_TIMER_3

Includes:

- #include <driver/ledc.h>

ledc_stop

Halt the PWM signal output.

```
esp_err_t ledc_stop(
   ledc_mode_t    speedMode,
   ledc_channel_t channel,
   uint32_t       idleLevel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

Includes:

- #include <driver/ledc.h>

ledc_timer_config

Configure a timer.

```
esp_err_t ledc_timer_config(
   ledc_timer_config_t* timerConf)
```

The `timerConf` is a pointer to a structure containing:

- `ledc_mode_t speed_mode` – Can be one of:
  - LEDC_HIGH_SPEED_MODE
- `ledc_timer_bit_t bit_num` – Can be one of:
  - `LEDC_TIMER_10_BIT` – 0-1023

  - `LEDC_TIMER_11_BIT` – 0-2047
  - `LEDC_TIMER_12_BIT` – 0-4095
  - `LEDC_TIMER_13_BIT` – 0-8191
  - `LEDC_TIMER_14_BIT` – 0-16383
  - `LEDC_TIMER_15_BIT` – 0-32767
- `ledc_timer_t timer_num` – Can be one of:
  - LEDC_TIMER_0
  - LEDC_TIMER_1
  - LEDC_TIMER_2
  - LEDC_TIMER_3
- `uint32_t freq_hz` – The frequency of the signal in Hz.

Includes:

- #include <driver/ledc.h>

See also:

- LEDC – Pulse Width Modulation – PWM
- ledc_channel_config

ledc_timer_pause

Pause the timer.

```
esp_err_t ledc_timer_pause(
   ledc_mode_t speedMode,
   uint32_t timerSel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

Includes:

- #include <driver/ledc.h>

ledc_timer_resume

Resume the timer.

```
esp_err_t ledc_timer_resume(
   ledc_mode_t speedMode,
   uint32_t timerSel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

Includes:

- #include <driver/ledc.h>

ledc_timer_rst

Reset the timer.

```
esp_err_t ledc_timer_rst(
   ledc_mode_t speedMode,
   uint32_t timerSel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

Includes:

- #include <driver/ledc.h>

ledc_timer_set

Explicitly set the timer details.

```
esp_err_t ledc_timer_set(
   ledc_mode_t speedMode,
   ledc_timer_t timerSel,
   uint32_t divNum,
   uint32_t bitNum,
   ledc_clk_src_t clkSrc)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE` (value is 0).

The `timerSel` can be one of:

- LEDC_TIMER_0
- LEDC_TIMER_1
- LEDC_TIMER_2
- LEDC_TIMER_3

The `divNum` is ??.

The `bitNum` is ??.

The `clkSrc` is one of `LEDC_REF_TICK` (value is 0, 1MHz) or `LEDC_APB_CLK` (value is 1, 80Mhz).

Includes:

- #include <driver/ledc.h>

ledc_update_duty

Update the duty value or the fade parameters for the channel.

```
esp_err_t ledc_update_duty(
   ledc_mode_t speedMode,
   ledc_channel_t channel)
```

Currently, `speedMode` can only be `LEDC_HIGH_SPEED_MODE`.

The `channel` can be one `LED_CHANNEL_0` → `LEDC_CHANNEL_7`.

Includes:

- #include <driver/ledc.h>

See also:

- ledc_get_duty
- ledc_set_duty
- ledc_channel_config

Pulse Counter

pcnt_counter_clear

```
esp_err_t pcnt_counter_clear(pcnt_unit_t pcnt_unit)
```

pcnt_counter_pause

```
esp_err_t pcnt_counter_pause(pcnt_unit_t pcnt_unit)
```

pcnt_counter_resume

```
esp_err_t pcnt_counter_resume(pcnt_unit_t pcnt_unit)
```

pcnt_event_disable

```
esp_err_t pcnt_event_disable(
   pcnt_unit_t unit,
   pcnt_evt_type_t evt_type)
```

pcnt_event_enable

```
esp_err_t pcnt_event_enable(
   pcnt_unit_t unit,
   pcnt_evt_type_t evt_type)
```

pcnt_filter_enable

```
esp_err_t pcnt_filter_enable(pcnt_unit_t unit)
```

pcnt_filter_disable

```
esp_err_t pcnt_filter_disable(pcnt_unit_t unit)
```

pcnt_get_counter_value

```
esp_err_t pcnt_get_counter_value(
   pcnt_unit_t pcnt_unit,
   int16_t* count)
```

pcnt_get_event_value

```
esp_err_t pcnt_get_event_value(
   pcnt_unit_t unit,
   pcnt_evt_type_t evt_type,
   int16_t *value)
```

pcnt_get_filter_value

```
esp_err_t pcnt_get_filter_value(
   pcnt_unit_t unit,
   uint16_t *filter_val)
```

pcnt_intr_enable

```
esp_err_t pcnt_intr_enable(pcnt_unit_t pcnt_unit)
```

pcnt_intr_disable

```
esp_err_t pcnt_intr_disable(pcnt_unit_t pcnt_unit)
```

pcnt_isr_register

```
esp_err_t pcnt_isr_register(
   uint32_t pcnt_intr_num,
   void (*fn)(void*), void * arg)
```

pcnt_set_event_value

```
esp_err_t pcnt_set_event_value(
   pcnt_unit_t unit,
   pcnt_evt_type_t evt_type,
   int16_t value)
```

pcnt_set_filter_value

```
esp_err_t pcnt_set_filter_value(
   pcnt_unit_t unit,
   uint16_t filter_val)
```

pcnt_set_mode

```
esp_err_t pcnt_set_mode(
   pcnt_unit_t unit,
   pcnt_channel_t channel,
   pcnt_count_mode_t pos_mode,
   pcnt_count_mode_t neg_mode,
   pcnt_ctrl_mode_t hctrl_mode,
   pcnt_ctrl_mode_t lctrl_mode)
```

pcnt_set_pin

```
esp_err_t pcnt_set_pin(pcnt_unit_t unit, pcnt_channel_t channel, int pulse_io, int
ctrl_io)
```

pcnt_uint_config

```
esp_err_t pcnt_unit_config(pcnt_config_t *pcnt_config)
```

Logging

ESP-IDF provides a set of logging functions primarily used for debugging. The primary include file is called "`esp_log.h`".

esp_log_level_set

Set the log level for the specified tag.

```
void esp_log_level_set(
   const char *tag,
   esp_log_level_t level)
```

The tag is the identifier for the class of logging. The value "`*`" resets all tags to the specified log level.

The level is the level of logging at or above that will be included in the log output. It may be one of:

- `ESP_LOG_VERBOSE` (5) – Log at the verbose level.
- `ESP_LOG_DEBUG` (4) – Log at the debug level.
- `ESP_LOG_INFO` (3) – Log at the information level.
- `ESP_LOG_WARN` (2) – Log at the warning level.
- `ESP_LOG_ERROR` (1) – Log at the error level.
- `ESP_LOG_NONE` (0) – No logging.

Invoking `esp_log_level_set("*", ESP_LOG_ERROR)`, disables all messages except errors.

Includes:

- #include <esp_log.h>

See also:

- ESP-IDF logging

esp_log_set_vprintf

Specify a function that will be used to output the log data.

```
void esp_log_set_vprintf(vprintf_like_t func)
```

By default, the output of logging is sent to the UART but we may wish to perform other tasks such as a circular log in memory or writing to a storage device or sending over a TCP/IP socket. By providing our own function here, we can control the output destination and event tinker with the format (if needed).

The default function is "`&vprintf`".

Includes:

- #include <esp_log.h>

See also:

- ESP-IDF logging

esp_log_write

Write an entry to the output log.

```
void esp_log_write(
   esp_log_level_t level,
   const char *tag,
   const char *format, …)
```

The `level` is the level of this log record being written. It may be one of:

- `ESP_LOG_VERBOSE` – Log at the verbose level.
- `ESP_LOG_DEBUG` – Log at the debug level.
- `ESP_LOG_INFO` – Log at the information level.
- `ESP_LOG_WARN` – Log at the warning level.
- `ESP_LOG_ERROR` – Log at the error level.
- `ESP_LOG_NONE` – No logging.

The `tag` is the identifier for this tag of log and `format` is the specification of what is to be written. Ideally we shouldn't call this function directly but instead use of the logging macros such as `ESP_LOGV(tag, format, …)`.

Includes:

- #include <esp_log.h>

See also:

- ESP-IDF logging

Non Volatile Storage

The non-volatile storage components save and load data from storage that is preserved between boots.

See also:

- Non Volatile Storage

nvs_close

Close a previously opened handle.

```
void nvs_close(nvs_handle handle)
```

Close a handle that was previously opened by a call to `nvs_open()`. Remember that updates to the store are not completed until after a call to nvs_commit(). Closing a handle without performing a commit will discard any changes.

Includes:

- #include <nvs.h>

See also:

- nvs_open

nvs_commit

Commit changes to non volatile storage.

```
esp_err_t nvs_commit(nvs_handle handle)
```

Any writes made against the storage are committed at this point. Note that writes may have been committed before this point but this forces all the writes to be flushed. Think of this as committing any pending writes as opposed to anything related to a transaction.

Includes:

- #include <nvs.h>

nvs_erase_all

Erase all the name value pairs.

```
esp_err_t nvs_erase_all(nvs_handle handle)
```

Erase all the name value pairs. Note that this may not be truly performed until after a call to `nvs_commit()`.

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_erase_key

Erase a specific name/value key pair.

```
esp_err_t nvs_erase_key(
   nvs_handle handle,
   char *key)
```

Erase the name/value pair with the given key. The deletion may not truly occur until after an associated commit is made.

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_flash_init

Initialize NVS with defaults.

```
esp_err_t nvs_flash_init(void)
```

The defaults as of 2016-10 are a 12KByte area starting at `0x0000 6000` in flash offset. This is `0x0000 6000` to `0x0000 8FFF` (by offsets).

Includes:

- #include <nvs_flash.h>

nvs_flash_init_custom

Custom initialize NVS.

```
esp_err_t nvs_flash_init_custom(
   uint32_t baseSector,
   uint32_t sectorCount)
```

Includes:

- #include <nvs_flash.h>

nvs_get_blob

Retrieve a blob of data from storage.

```
esp_err_t nvs_get_blob(
   nvs_handle handle,
   const char *key,
   void *out,
   size_t *length)
```

The `out` is a pointer to the storage that will be used to hold the retrieved data. On input `length` holds the size of the storage available to hold the result and will be updated with the actual `length` retrieved.

If `out` has a NULL value then the length that is needed to be allocated to hold the data is returned in `length`.

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_str

Retrieve a null terminated string from storage.

```
esp_err_t nvs_get_str(
   nvs_handle handle,
   const char *key,
   char *out,
   size_t *length)
```

The `out` is a pointer to the storage that will be used to hold the retrieved data. On input `length` holds the size of the storage available to hold the result and will be updated with the actual `length` retrieved.

If `out` has a NULL value then the length that is needed to be allocated to hold the data is returned in `length`. This includes the NULL byte terminator.

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_i8

Retrieve an 8 bit integer.

```
esp_err_t  nvs_get_i8(
   nvs_handle handle,
   const char *key,
   int8_t *out_value)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_i16

Retrieve a 16 bit integer.

```
esp_err_t nvs_get_i16(
   nvs_handle handle,
   const char *key,
   int16_t *out_value)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_i32

Retrieve a 32 bit integer.

```
esp_err_t nvs_get_i32(
   nvs_handle handle,
   const char *key,
   int32_t *out_value)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_i64

Retrieve a 64 bit integer.

```
esp_err_t nvs_get_i64(
   nvs_handle handle,
   const char *key,
   int64_t *out_value)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_u8

Retrieve an unsigned 8 bit integer.

```
esp_err_t nvs_get_u8(
   nvs_handle handle,
   const char *key,
   uint8_t *out_value)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_u16

Retrieve an unsigned 16 bit integer.

```
esp_err_t nvs_get_u16(
   nvs_handle handle,
   const char *key,
   uint16_t *outValue)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_u32

Retrieve an unsigned 32 bit integer.

```
esp_err_t nvs_get_u32(
   nvs_handle handle,
   const char *key,
   uint32_t *outValue)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_get_u64

Retrieve an unsigned 64 bit integer.

```
esp_err_t nvs_get_u64(
   nvs_handle handle,
   const char *key,
   uint64_t *outValue)
```

The return value is ESP_OK on success and an indication code on error. Common codes include:

- `ESP_ERR_NVS_NOT_FOUND` – Key not found.

Includes:

- #include <nvs.h>

nvs_open

Open a storage area with a given namespace.

```
esp_err_t nvs_open(
   const char *name,
   nvs_open_mode open_mode,
   nvs_handle *outHandle)
```

Open a given named storage area for access. The name of the area is supplied in the `name` parameter. The `open_mode` may be one of:

- `NVS_READWRITE` – The application can read and write the storage.
- `NVS_READONLY` – The application can only read the storage.

The `outHandle` is a pointer to storage where a handle will be stored. This is the handle passed into other NVS API calls.

If the call succeeded, `ESP_OK` will be returned. Otherwise other codes such as:

- `ESP_ERR_NVS_NOT_FOUND` – Namespace doesn't exist and opened read-only.

For example:

```
nvs_handle handle;
nvs_open("namespace", NVS_READWRITE, &handle);
…
nvs_close(handle);
```

Includes:

- #include <nvs.h>

See also:

- nvs_close

nvs_set_blob

Save a blob of data to storage. The maximum size is 4000 bytes per blob.

```
esp_err_t nvs_set_blob(
   nvs_handle handle,
   const char *key,
   const void *value,
   size_t length)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_str

Save a null terminated string.

```
esp_err_t nvs_set_str(
   nvs_handle handle,
   const char *key,
   const char *value)
```

The `key` is the key against which the string will be saved.

The `value` is the value to save.

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_i8

Save an 8 bit integer.

```
esp_err_t nvs_set_i8(
   nvs_handle handle,
   const char *key,
   int8_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_i16

Save a 16 bit integer.

```
esp_err_t nvs_set_i16(
   nvs_handle handle,
   const char *key,
   int16_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_i32

Save a 32 bit integer.

```
esp_err_t nvs_set_i32(
   nvs_handle handle,
   const char *key,
   int32_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_i64

Save a 64 bit integer.

```
esp_err_t nvs_set_i64(
   nvs_handle handle,
   const char *key,
   int64_t value)
```

See also:

- nvs_commit

nvs_set_u8

Save an unsigned 8 bit integer.

```
esp_err_t nvs_set_u8(
   nvs_handle handle,
   const char *key,
   uint8_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_u16

Save an unsigned 16 bit integer.

```
esp_err_t nvs_set_u16(
   nvs_handle handle,
   const char *key,
   uint16_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_u32

Save an unsigned 32 bit integer.

```
esp_err_t nvs_set_u32(
   nvs_handle handle,
   const char *key,
   uint32_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

nvs_set_u64

Save an unsigned 64 bit integer.

```
esp_err_t nvs_set_u64(
   nvs_handle handle,
   const char *key,
   uint64_t value)
```

Includes:

- #include <nvs.h>

See also:

- nvs_commit

Partition API

See also:

- Partition table

esp_partition_erase_range

Erase a range of data within a partition.

```
esp_err_t esp_partition_erase_range(
   const esp_partition_t* partition,
   uint32_t start_addr,
   uint32_t size);
```

Includes

- #include <esp_partition.h>

esp_partition_find

Find a partition with a given type and optionally a subtype and/or label.

```
esp_partition_iterator_t esp_partition_find(
   esp_partition_type_t type,
   esp_partition_subtype_t subtype,
   const char* label)
```

- `type` – The type of partition being sought. One of:
  - ESP_PARTITION_TYPE_APP
  - ESP_PARTITION_TYPE_DATA
- `subtype` – The sub type of the partition being sought. One of:
  - ESP_PARTITION_SUBTYPE_APP_FACTORY
  - ESP_PARTITION_SUBTYPE_APP_OTA_xxx
  - ESP_PARTITION_SUBTYPE_APP_TEST
  - ESP_PARTITION_SUBTYPE_DATA_OTA
  - ESP_PARTITION_SUBTYPE_DATA_PHY
  - ESP_PARTITION_SUBTYPE_DATA_NVS

  - ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD
  - ESP_PARTITION_SUBTYPE_DATA_FAT
  - ESP_PARTITION_SUBTYPE_DATA_SPIFFS
  - ESP_PARTITION_SUBTYPE_DATA_COREDUMP
  - ESP_PARTITION_SUBTYPE_ANY
- `label` – The label of the partition being sought or NULL for don't care.

The return is an iterator that can be used to iterate over the found partitions. If no partitions were found that matched, then NULL is returned. The iterator must be released with `esp_partition_iterator_release()` when finished. To move to the next iterator call `esp_partition_next()`.

Includes

- #include <esp_partition.h>

See also:

- esp_partition_get
- esp_partition_iterator_release

esp_partition_find_first

Obtain the first partition that matches the details or NULL if none match.

```
const esp_partition_t* esp_partition_find_first(
   esp_partition_type_t type,
   esp_partition_subtype_t subtype,
   const char* label)
```

- `type` – The type of partition being sought. One of:
  - ESP_PARTITION_TYPE_APP
  - ESP_PARTITION_TYPE_DATA
- `subtype` – The sub type of the partition being sought. One of:
  - ESP_PARTITION_SUBTYPE_APP_FACTORY
  - ESP_PARTITION_SUBTYPE_APP_OTA_xxx
  - ESP_PARTITION_SUBTYPE_APP_TEST
  - ESP_PARTITION_SUBTYPE_DATA_OTA
  - ESP_PARTITION_SUBTYPE_DATA_PHY
  - ESP_PARTITION_SUBTYPE_DATA_NVS

  - ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD
  - ESP_PARTITION_SUBTYPE_DATA_FAT
  - ESP_PARTITION_SUBTYPE_DATA_SPIFFS
  - ESP_PARTITION_SUBTYPE_DATA_COREDUMP
  - ESP_PARTITION_SUBTYPE_ANY
- `label` – The label of the partition being sought or NULL for don't care.

The return is an instance of the description of the partition. That is a data structure that contains:

- `type` – One of
  - ESP_PARTITION_TYPE_APP
  - ESP_PARTITION_TYPE_DATA
- `subtype` – One of:
  - ESP_PARTITION_SUBTYPE_APP_FACTORY
  - ESP_PARTITION_SUBTYPE_APP_OTA_xxx
  - ESP_PARTITION_SUBTYPE_APP_TEST
  - ESP_PARTITION_SUBTYPE_DATA_OTA
  - ESP_PARTITION_SUBTYPE_DATA_PHY
  - ESP_PARTITION_SUBTYPE_DATA_NVS
  - ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD
  - ESP_PARTITION_SUBTYPE_DATA_FAT
  - ESP_PARTITION_SUBTYPE_DATA_SPIFFS
  - ESP_PARTITION_SUBTYPE_DATA_COREDUMP
- `address` – Starting address of the partition in flash.
- `size` – Size of the partition in bytes.
- `label` – NULL terminated string describing the partition.
- `encrypted` – True if the partition is encrypted.

Includes

- #include <esp_partition.h>

esp_partition_get

Get the partition details for the current iterator.

```
const esp_partition_t *esp_partition_get(esp_partition_iterator_t iterator)
```

The return is an instance of the description of the partition. That is a data structure that contains:

- `type` – One of
  - ESP_PARTITION_TYPE_APP (0x00)
  - ESP_PARTITION_TYPE_DATA (0x01)
- `subtype` – One of:
  - ESP_PARTITION_SUBTYPE_APP_FACTORY (0x00)
  - ESP_PARTITION_SUBTYPE_APP_OTA_xxx (0x10 - 0x1F)
  - ESP_PARTITION_SUBTYPE_APP_TEST (0x20)
  - ESP_PARTITION_SUBTYPE_DATA_OTA (0x00)
  - ESP_PARTITION_SUBTYPE_DATA_PHY (0x01)
  - ESP_PARTITION_SUBTYPE_DATA_NVS (0x02)
  - ESP_PARTITION_SUBTYPE_DATA_COREDUMP (0x03)
  - ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD (0x80)
  - ESP_PARTITION_SUBTYPE_DATA_FAT (0x81)
  - ESP_PARTITION_SUBTYPE_DATA_SPIFFS (0x82)
- `address` – Starting address of the partition in flash.
- `size` – Size of the partition in bytes.
- `label` – NULL terminated string describing the partition.
- `encrypted` – True if the partition is encrypted.

Includes

- #include <esp_partition.h>

See also:

- esp_partition_find

esp_partition_iterator_release

Release a previously constructed iterator.

```
void esp_partition_iterator_release(esp_partition_iterator_t iterator)
```

Includes

- #include <esp_partition.h>

See also:

- esp_partition_find

esp_partition_mmap

Memory map the partition.

```
esp_err_t esp_partition_mmap(
   const esp_partition_t* partition,
   uint32_t offset,
   uint32_t size,
   spi_flash_mmap_memory_t memory,
   const void** out_ptr,
   spi_flash_mmap_handle_t* out_handle)
```

Includes

- #include <esp_partition.h>

esp_partition_next

Move the iterator to the next partition element.

```
esp_partition_iterator_t esp_partition_next(esp_partition_iterator_t iterator)
```

Includes

- #include <esp_partition.h>

See also:

- esp_partition_find
- esp_partition_iterator_release

esp_partition_read

Read data from the partition.

```
esp_err_t esp_partition_read(
   const esp_partition_t* partition,
   size_t src_offset,
   void* dst,
   size_t size)
```

Includes

- #include <esp_partition.h>

esp_partition_write

Write data to the partition.

```
esp_err_t esp_partition_write(
   const esp_partition_t* partition,
   size_t dst_offset,
   const void* src,
   size_t size)
```

Includes

- #include <esp_partition.h>

Virtual File System

See also:

- Virtual File System

esp_vfs_dev_uart_register

```
void esp_vfs_dev_uart_register()
```

Register the virtual file system known as "`/dev/uart`" as a serial port interface. We can thus access "`/dev/uart/0`", "`/dev/uart/1`" and "`/dev/uart/2`".

Functions support by **this** file system are:

- `open` –
- `close` –
- `write` –
- `fstat` –

Note that `read()` is not yet supported.

See also:

- Using the VFS component with serial

esp_vfs_register

Register a new VFS.

```
esp_err_t esp_vfs_register(
   const char *base_path,
   const esp_vfs_t *vfs,
   void *ctx);
```

Register a new virtual file system. The base_path is a C string that can be no more than `ESP_VFS_PATH_MAX` characters in length (currently defined as 15). It must start with a "`/`" character and must not end with a "`/`". This will serve as the junction point for accessing data through this file system.

The `vfs` is a pointer to an `esp_vfs_t` that must be populated by the application making the call to `esp_vfs_register`. The data structure does not have to be maintained after making the call as a copy is made.

The `ctx` is a pointer to data that is passed to the registered functions if the vfs "`flags`" attribute is set to `ESP_VFS_FLAG_CONTEXT_PTR`. When used, all the callback functions have an additional option which is the context. The names of the functions change to have an "`_p`" after them. For example, with the default setting, the close function signature is:

```
int close(int fd)
```

while when the context flag is set, the function signature changes to:

```
int close_p(void *ctx, int fd)
```

and the assignment property in the `esp_vfs_t` structure changes to the "`_p`" versions.

Within the `esp_vfs_t` data structure we have the following fields:

- `fd_offset` – Offset value for numeric file descriptors.
- `flags` – Flags associated with this VFS. Choices today are:
  - `ESP_VFS_FLAG_DEFAULT` – Default value.
  - `ESP_VFS_FLAG_CONTEXT_PTR` – The context pointer functions are used and the context data is passed in.

- `write` – A function called to write data.

```
size_t (*write)(int fd, const void *data, size_t size)
```

Write into the file specified by the `fd` file descriptor the data pointed to by `data` of length `size` bytes. The return is the number of bytes actually written.

- `lseek` – A function called to seek within a file.

```
off_t (*lseek)(int fd, off_t offset, int whence)
```

The whence parameter is the controller of how offset is applied:

  - SEEK_SET – Set the position to an absolute of offset.
  - SEEK_CUR – Set the position relative to the current position plus offset. Offset can be negative to seek backwards.
  - SEEK_END – Set the position relative to the end of the file plus offset. Offset can be used as 0 or negative.

- read – A function called to read data.

```
ssize_t (*read)(int fd, void *dst, size_t size)
```

- open – A function called to open a file.

```
int (*open)(const char *path, int flags, int accessMode)
```

The path is the path to the local file. The VFS prefix is removed and only the local file part is passed in. The flags are the operational flags as described in the Unix open() system call. The accessMode are the permission of the file when it is created.

- close – A function called to close a file.

```
int (*close)(int fd)
```

- fstat – A function called to stat a file by file descriptor.

```
int (*fstat)(int fd, struct stat *st)
```

- stat – A function called to stat a file by name.

```
int (*stat)(const char *path, struct stat *st)
```

The struct stat contains:

  - st_dev
  - st_ino
  - st_mode
  - st_nlink
  - st_uid
  - st_gid

  - st_rdev
  - st_size
  - st_blksize
  - st_blocks
  - st_atime
  - st_mtime
  - st_ctime

- `link` – A function called to link a file.

```
int (*link)(const char *oldPath, const char *newPath)
```

The mapped POSIX api is: man(2) – link

- `unlink` – A function called to unlink a file.

```
int (*unlink)(const char *path)
```

The mapped POSIX api is: man(2) – unlink

- `rename` – A file called to rename a file.

```
int (*rename)(const char *oldPath, const char *newPath)
```

- `opendir` – Open a directory for reading.

```
DIR *opendir(const char *name)
```

- `readdir` – Read the next entry in the directory stream.

```
struct dirent readdir(DIR *pdir)
```

See also:

- man(3) – readdir

- `telldir` – Return the current location in the directory stream.

```
long telldir(DIR *pdir)
```

See also:

- man(3) – telldir

- `seekdir` – Set the position of the next `readdir()`.

```
void seekdir(DIR *pdir, long offset)
```

See also:

- man(3) – seekdir

- `closedir` – Close a directory.

```
int closedir(DIR *pdir)
```

See also:

- man(3) – closedir

- `mkdir` – Create a directory.

```
int mkdir(const char *name, mode_t mode)
```

See also:

- man(3) – mkdir

- `rmdir` – Delete a directory.

```
int rmdir(const char *name)
```

See also:

- man(2) – rmdir

Includes:

- esp_vfs.h

See also:

- man(2) – open
- man(2) – close
- man(3) – closedir
- man(2) – fstat
- man(2) – link
- man(3) – mkdir
- man(3) – readdir
- man(2) – rename
- man(2) – rmdir
- man(3) – seekdir
- man(2) – stat
- man(3) – telldir
- man(2) – unlink
- man(2) – write

FatFs file system

See also:

- FATFS File System

esp_vfs_fat_register

```
esp_err_t esp_vfs_fat_register(
   const char *base_path,
   const char *fat_drive,
   size_t max_files,
   FATFS **out_fs)
```

The `base_path` is where the FAT file system should be registered.

The `fat_drive` is a FAT file system drive specification. If we only have one drive then this can be the empty string.

The `max_files` is the maximum number of files we can have open at one time.

The `out_fs` is a pointer to the FATFS structure used in the `f_mount()` call.

Includes:

- #include <esp_vfs_fat.h>

esp_vfs_fat_sdmmc_mount

Mount an SD card as a posix file system.

```
esp_err_t esp_vfs_fat_sdmmc_mount(
   const char *base_path,
   const sdmmc_host_t *host_config,
   const sdmmc_slot_config_t *slot_config,
   const esp_vfs_fat_sdmmc_mount_config_t *mount_config,
   sdmmc_card_t **out_card)
```

The `base_path` is the path in the POSIX file system which acts as the mount point for the files hosted in the FAT32 file system contained on the card. A common example is "`/sdcard`".

The `host_config` is an instance of `sdmmc_host_t` and is commonly configured using `SDMMC_HOST_DEFAULT()`. For example:

```
sdmmc_host_t host_config = SDMMC_HOST_DEFAULT();
```

The `sdmmc_host_t` is a C structure that contains:

- `uint32_t flags` – Define operational flags. These include:
    - `SDMMC_HOST_FLAG_1BIT` – Host uses 1-line SD protocol.

  - `SDMMC_HOST_FLAG_4BIT` – Host uses 4-line SD protocol.
  - `SDMMC_HOST_FLAG_8BIT` – Host uses 8-line SD protocol.
  - `SDMMC_HOST_FLAG_SPI` – Host uses SPI protocol.
- `int slot` – Unknown
- `int max_freq_khz` – Unknown
- `float io_voltage` – Unknown
- `init` – Function to be called to initialize the host layer.
- `set_bus_width` – Function to be called to set the bus width.
- `set_card_clk` – Function to be called to set the clock.
- `do_transaction` – Function to be called to perform a transaction.
- `deinit` – Function to be called to de-initialize the host layer.

Using the `SDMMC_HOST_DEFAULT()` to initialize the structure sets the following default values:

Property	**Value**
flags	SDMMC_HOST_FLAG_4BIT
slot	SDMMC_HOST_SLOT_1
max_freq_khz	SDMMC_FREQ_DEFAULT
io_voltage	3.3
init	sdmmc_host_init
set_bus_width	sdmmc_host_set_bus_width
set_card_clk	sdmmc_host_set_card_clk
do_transaction	sdmmc_host_do_transaction
deinit	sdmmc_host_deinit

The `slot_config` parameter is an instance of `sdmmc_slot_config_t` and is commonly configured using `SDMMC_SLOT_CONFIG_DEFAULT()`.

The `mount_config` parameter is an instance of `esp_vfs_far_sdmmc_mount_config_t` which has properties:

- format_if_mount_failed
- max_files

Includes:

- #include <esp_vfs_fat.h>

See also:

- esp_vfs_fat_sdmmc_unmount

esp_vfs_fat_sdmmc_unmount

```
esp_err_t esp_vfs_fat_sdmmc_unmount()
```

Includes:

- #include <esp_vfs_fat.h>

See also:

- esp_vfs_fat_sdmmc_mount

esp_vfs_fat_spiflash_mount

Initialize FAT file system in SPI flash and register with VFS.

```
esp_err_t esp_vfs_fat_spiflash_mount(
   const char *base_path,
   const char *partition_label,
   const esp_vfs_fat_mount_config_t *mount_config,
   wl_handle_t *wl_handle)
```

This is a powerful convenience function that registers the FAT file system and associates it with the Virtual File System (VFS) subsystem this making it available for Posix based I/O APIs.

- `base_path` – Path where FAT file system partition should be mounted (eg. `/spiflash`)
- `partition_label` – Label of the flash partition where the data will be stored.
- `mount_config` – Pointer to structure for parameters for mounting FAT file system.
- `wl_handle` – The returned/populated wear leveling driver handle.

The mount_config parameter points to a structure which provides control information for the mount operation. Specifically, it contains:

- `format_if_mount_failed` (`bool`) – A flag that indicates whether or not to format the SPI flash memory contained in the partition if the mount fails (presumably because the partition was not previously initialized).
- `max_files` (`int`) – The permitted maximum number of concurrently open files.

An example of a partition record for a FAT file system held in SPI flash would be:

```
storage, data, fat, , 1M,
```

As of 2017-05, the smallest allowable size of the partition is 528K.

Note the type of the partition is "`data`" and the subtype is "`fat`".

An example partition file might be (`partitions.csv`):

```
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 1M,
storage,  data, fat,     ,        1M,
```

If the partition area can not be found, we will get an error indication and if logging is enabled, see the following (or similar) message:

```
vfs_fat_spiflash: Failed to find FATFS partition (type='data', subtype='fat',
partition_label='storage'). Check the partition table.
```

Example:

```
wl_handle wl_handle = WL_INVALID_HANDLE;
esp_vfs_fat_mount_config_t mount_config;
mount_config.max_files = 4;
mount_config.format_if_mount_failed = true;

esp_err_t err = esp_vfs_fat_spiflash_mount(
   "/spiflash", "storage", &mount_config, &wl_handle);
```

Includes:

- #include <esp_vfs_fat.h>

See also:

- Partition table

esp_vfs_fat_spiflash_unmount

Unmount the FAT file system and release the resources.

```
esp_err_t esp_vfs_fat_spiflash_unmount(
   const char *base_path,
   wl_handle_t wl_handle)
```

- `base_path` – The path where the FATFS file system was mounted.
- `wl_handle` – The handle returned from `esp_vfs_fat_spiflash_mount()`.

Includes:

- #include <esp_vfs_fat.h>

esp_vfs_fat_unregister

```
esp_err_t esp_vfs_fat_unregister()
```

This function has been deprecated in favor of `esp_vfs_fat_unregister_path()`. Don't use it for new code.

Includes:

- #include <esp_vfs_fat.h>

esp_vfs_fat_unregister_path

Un-register the FATFS from VFS.

```
esp_err_t esp_vfs_fat_unregister_path(const char *base_path)
```

- `base_path` – The path prefix where FATFS was registered. This must be the same path as used in `esp_vfs_fat_register()`.

Includes:

- #include <esp_vfs_fat.h>
-

f_mount

```
FRESULT f_mount(FATFS *fs, const char *path, BYTE opt)
```

The values of `opt` can be:

- 0 – Don't mount now.
- 1 – Mount now and check it is ready for work.

See also:

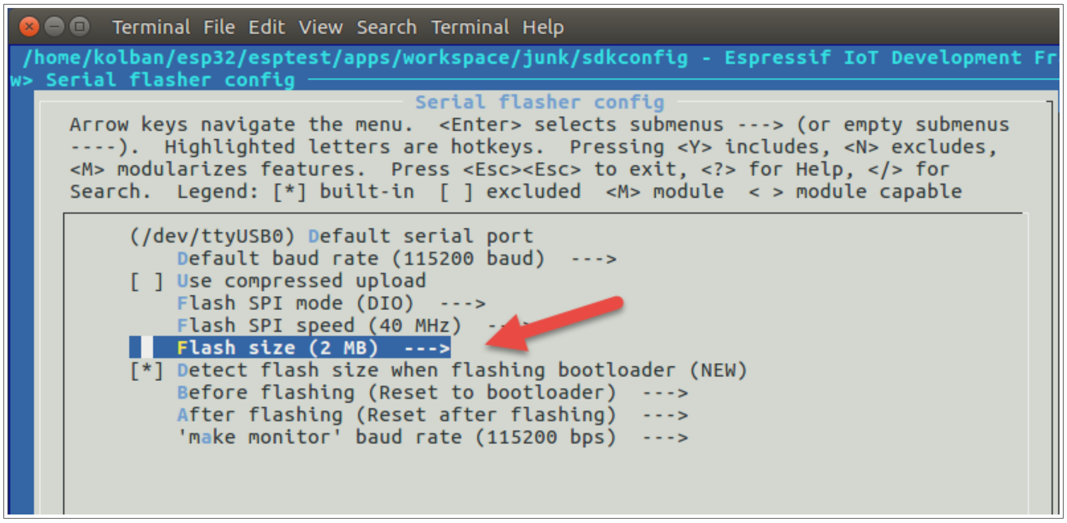
- [FatFs manual – f_mount]

ff_diskio_register

```
void ff_diskio_register(
   BYTE pdrv,
   const ff_diskio_impl_t *diskio_impl)
```

SPI Flash

The SPI Flash apis allow us to read, write and erase sectors contained within flash memory. When working with flash address spaces, we need to ensure that we tell the ESP-IDF configuration how much flash space is available. We do that in the "`make menuconfig`" options. The default is 2MBytes. Attempts to access above 2MBytes will be flagged as an error, even if your real storage exceeds that.



```
 /home/kolban/esp32/esptest/apps/workspace/junk/sdkconfig - Espressif IoT Development Fr
w> Serial flasher config ─────────────────────────────────────────────────────────────
                              Serial flasher config
    Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
    ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
    <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for
    Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

         (/dev/ttyUSB0) Default serial port
             Default baud rate (115200 baud)  --->
         [ ] Use compressed upload
             Flash SPI mode (DIO)  --->
             Flash SPI speed (40 MHz)  --->
             Flash size (2 MB)  --->
         [*] Detect flash size when flashing bootloader (NEW)
             Before flashing (Reset to bootloader)  --->
             After flashing (Reset after flashing)  --->
             'make monitor' baud rate (115200 bps)  --->
```

spi_flash_erase_range

Erase a range of flash storage.

```
esp_err_t spi_flash_erase_range(size_t startAddr, size_t size)
```

Includes:

- #include <esp_spi_flash.h>

spi_flash_erase_sector

Erase a flash sector. Each sector is 4k in size.

```
esp_err_t spi_flash_erase_sector(size_t sector)
```

The sec parameter is the sector number (a sector is 4096 bytes in size).

Includes:

- #include <esp_spi_flash.h>

See also:

spi_flash_get_chip_size

Get the size of the declared storage for the flash device.

```
size_t spi_flash_get_chip_size()
```

The return is the amount of storage that the environment has declared to be available on the flash device.

Includes:

- #include <esp_spi_flash.h>

spi_flash_get_counters

```
const spi_flash_counters_t *spi_flash_get_counters()
```

Includes:

- #include <esp_spi_flash.h>

spi_flash_init

```
void spi_flash_init()
```

Includes:

- #include <esp_spi_flash.h>

spi_flash_mmap

Map flash storage into the ESP32 address bus.

```
esp_err_t spi_flash_mmap(
   uint32_t                 srcAddr,
   size_t                   size,
   spi_flash_mmap_memory_t  memory,
   const void**             outPtr,
   spi_flash_mmap_handle_t* outHandle)
```

The `srcAddr` is the address in flash address range that is the start of the data we are going to map from. It must be located on a 64K boundary (i.e. be evenly divisible by 64*1024 which is 0x1 0000).

The size is the size of data to be mapped. It will be rounded up to the nearest 64K multiple.

The `memory` parameter is where the memory should be mapped. Choices are:

- `SPI_FLASH_MMAP_DATA` – Map to data memory.
- `SPI_FLASH_MMAP_INST` – Map to instruction memory.

The `outPtr` is the address in the CPU address range where the mapped data can be found.

The `outHandle` is the handle used to release the mapping (if needed) in a call to `spi_flash_munmap`.

Includes:

- #include <esp_spi_flash.h>

See also:

- spi_flash_munmap
- spi_flash_mmap_dump
- esp_partition_mmap

spi_flash_mmap_dump

```
void spi_flash_mmap_dump()
```

Includes:

- #include <esp_spi_flash.h>

spi_flash_munmap

```
void spi_flash_munmap(spi_flash_mmap_handle_t handle)
```

Includes:

- #include <esp_spi_flash.h>

spi_flash_read

Read data from flash

```
esp_err_t spi_flash_read(size_t src_addr, void* destAddr, size_t size)
```

The `src_addr` parameter is the address in flash that will be read. The `destAddr` is the address in memory which will be written. The `size` parameter is the size of data to be read.

Includes:

- #include <spi_flash.h>

See also:

spi_flash_reset_counters

```
void spi_flash_reset_counters()
```

spi_flash_write

Write data to flash

```
esp_err_t spi_flash_write(size_t destAddr, const void* srcAddr, size_t_t size)
```

The `destAddr` is the address in flash which is to be written. The `srcAddr` is the source address in memory from where the new data is to be taken. The `size` parameter is the size of the data to be written.

Includes:

- #include <spi_flash.h>

See also:

SDMMC

sdmmc_card_init

```
esp_err_t sdmmc_card_init(
   const sdmmc_host_t* host,
   sdmmc_card_t*       out_card)
```

Includes:

- #include <sdmmc_cmd.h>

sdmmc_card_print_info

```
void sdmmc_card_print_info(
   FILE*               stream,
   const sdmmc_card_t* card)
```

Includes:

- #include <sdmmc_cmd.h>

sdmmc_host_deinit

```
esp_err_t sdmmc_host_deinit()
```

Includes:

- #include <sdmmc_host.h>

sdmmc_host_do_transaction

```
esp_err_t sdmmc_host_do_transaction(
   int              slot,
   sdmmc_command_t* cmdinfo)
```

Includes:

- #include <sdmmc_host.h>

sdmmc_host_init

```
esp_err_t sdmmc_host_init()
```

Includes:

- #include <sdmmc_host.h>

sdmmc_host_init_slot

```
esp_err_t sdmmc_host_init_slot(
   int slot,
   const sdmmc_slot_config_t *slot_config)
```

Includes:

- #include <sdmmc_host.h>

sdmmc_host_set_bus_width

```
esp_err_t sdmmc_host_set_bus_width(
   int slot,
   size_t width)
```

Includes:

- #include <sdmmc_host.h>

sdmmc_host_set_card_clk

```
esp_err_t sdmmc_host_set_card_clk(
   int slot,
   uint32_t freq_khz)
```

Includes:

- #include <sdmmc_host.h>

sdmmc_read_sectors

```
esp_err_t sdmmc_read_sectors(
   sdmmc_card_t *card,
   void *dst,
   size_t start_sector,
   size_t sector_count)
```

Includes:

- #include <sdmmc_cmd.h>

sdmmc_write_sectors

```
esp_err_t sdmmc_write_sectors(
   sdmmc_card_t *card,
   const void* src,
   size_t start_sector,
   size_t sector_count)
```

Includes:

- #include <sdmmc_cmd.h>

Hardware Timers

timer_disable_intr

```
esp_err_t timer_disable_intr(
   timer_group_t group_num,
   timer_idx_t timer_num)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_enable_intr

```
esp_err_t timer_enable_intr(
   timer_group_t group_num,
   timer_idx_t timer_num)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_get_alarm_value

Get the value at which a timer event should fire.

```
esp_err_t timer_get_alarm_value(
   timer_group_t group_num,
   timer_idx_t timer_num,
   uint64_t *alarm_value)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_get_config

```
esp_err_t timer_get_config(
   timer_group_t group_num,
   timer_idx_t timer_num,
   timer_config_t *config)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`
- `config` – Contains
  - alarm_en
  - counter_en
  - counter_dir
  - intr_type
  - auto_reload
  - divider

Includes:

- #include <driver/timer.h>

timer_get_counter_time_sec

Read the counter value as a double in seconds.

```
esp_err_t timer_get_counter_time_sec(
   timer_group_t group_num,
   timer_idx_t timer_num,
   double *time)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_get_counter_value

Get the value of the timer.

```
esp_err_t timer_get_counter_value(
   timer_group_t group_num,
   timer_idx_t timer_num,
   uint64_t *timer_val)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_group_intr_enable

```
esp_err_t timer_group_intr_enable(
   timer_group_t group_num,
   uint32_t en_mask)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`

Includes:

- #include <driver/timer.h>

timer_group_intr_disable

```
esp_err_t timer_group_intr_disable(
   timer_group_t group_num,
   uint32_t disable_mask)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`

Includes:

- #include <driver/timer.h>

timer_isr_register

```
esp_err_t timer_isr_register(
   timer_group_t group_num,
   timer_idx_t timer_num,
   void (*fn)(void*), void * arg
   int intr_alloc_flags,
   timer_isr_handle_t *handle)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_init

```
esp_err_t timer_init(
   timer_group_t group_num,
   timer_idx_t timer_num,
   timer_config_t* config)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`
- `config` – Contains
  - alarm_en
  - counter_en
  - counter_dir
  - intr_type
  - auto_reload
  - `divider`

Includes:

- #include <driver/timer.h>

timer_pause

Pause a specific timer.

```
esp_err_t timer_pause(
   timer_group_t group_num,
   timer_idx_t timer_num)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

timer_set_counter_value

Set the counter value to a specific value.,

```
esp_err_t timer_set_counter_value(
   timer_group_t group_num,
   timer_idx_t timer_num,
   uint64_t load_val)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_start

Start a specific timer.

```
esp_err_t timer_start(
   timer_group_t group_num,
   timer_idx_t timer_num)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_set_alarm

Determine if an alarm should fire.

```
esp_err_t timer_set_alarm(
   timer_group_t group_num,
   timer_idx_t timer_num,
   timer_alarm_t alarm_en)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`
- `alarm_en` – One of:
  - TIMER_ALARM_DIS
  - TIMER_ALARM_EN

Includes:

- #include <driver/timer.h>

timer_set_alarm_value

Define the value at which an alarm event should occur when reached by the timer.

```
esp_err_t timer_set_alarm_value(
   timer_group_t group_num,
   timer_idx_t timer_num,
   uint64_t alarm_value)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

timer_set_auto_reload

Determine whether or not the timer should reload following an alarm event.

```
esp_err_t timer_set_auto_reload(
   timer_group_t group_num,
   timer_idx_t timer_num,
   timer_autoreload_t reload)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`
- `reload` – One of:
  - `TIMER_AUTORELOAD_DIS` – Do not load the counter after an alarm event.
  - `TIMER_AUTORELOAD_EN` – Load the counter after an alarm event.

Includes:

- #include <driver/timer.h>

timer_set_counter_mode

Set the mode of a specific counter.

```
esp_err_t timer_set_counter_mode(
   timer_group_t group_num,
   timer_idx_t timer_num,
   timer_count_dir_t counter_dir)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`
- `counter_dir` – One of:
  - TIMER_COUNT_DOWN
  - TIMER_COUNT_UP

Includes:

- #include <driver/timer.h>

timer_set_divider

Set the underlying clock divider.

```
esp_err_t timer_set_divider(
   timer_group_t group_num,
   timer_idx_t timer_num,
   uint16_t divider)
```

- `group_num` – One of:
  - `TIMER_GROUP_0`
  - `TIMER_GROUP_1`
- `timer_num` – One of:
  - `TIMER_0`
  - `TIMER_1`

Includes:

- #include <driver/timer.h>

Watchdog processing

The watchdog functions are all about monitoring liveness of tasks to ensure that they aren't stuck in loops or otherwise starving. The watchdog functions are controlled by the master settings defined by running "`make menuconfig`". There are entries in there which define whether or not task specific watchdog functions are enabled and, if so, how long between watchdog checks. The interval is measured in seconds with a default of 5. The duration is measured in seconds which feels like an exceptionally long time in processing units.

If a Watchdog timer expires before being fed, then a message is logged (by default). Alternatively, we can invoke a panic handler which can:

- print the registers and halt
- print the registers and reboot
- silently reboot
- invoke the built-in `gdb` stub

esp_int_wdt_init

Initialize the interrupt watchdog.

```
void esp_int_wdt_init()
```

esp_task_wdt_init

Initialize the task watchdog.

```
void esp_task_wdt_init()
```

esp_task_wdt_feed

Register a task or feed a task watchdog.

```
void esp_task_wdt_feed()
```

If called for the first time on a given FreeRTOS task then the task becomes registered as one that is being monitored for liveness. It must be regularly called from then on to ensure that the watchdog is fed.

esp_task_wdt_delete

Delete an association between a task and watchdog.

```
void esp_task_wdt_delete()
```

If in the past, the current task has called `esp_task_wdt_feed()` then it is considered on the watch list. If the task no longer wishes to watched for liveness or else is about to end, then we should call this function to remove the task from the list of monitored tasks by the watchdog.

AWS-IoT

aws_iot_is_autoreconnect_enabled

Determine if auto-reconnect is enabled.

```
bool aws_iot_is_autoreconnect_enabled(AWS_IoT_Client *pClient)
```

See also:

- aws_iot_mqtt_autoreconnect_set_status

aws_iot_mqtt_attempt_reconnect

```
IoT_Error_t aws_iot_mqtt_attempt_reconnect(AWS_IoT_Client *pClient)
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_autoreconnect_set_status

Enabled or disable auto-reconnect.

```
IoT_Error_t aws_iot_mqtt_autoreconnect_set_status(
   AWS_IoT_Client *pClient,
   bool newStatus)
```

Includes:

- #include <aws_iot_mqtt_client.h>

See also:

- aws_iot_is_autoreconnect_enabled

aws_iot_mqtt_connect

Connecto the AWS IoT service.

```
IoT_Error_t aws_iot_mqtt_connect(
   AWS_IoT_Client *pClient,
   const IoT_Client_Connect_Params *pConnectParams)
```

The `pConnectParams` is a pointer to a structure containing:

- char struct_id
- MQTT_Ver_t MQTTVersion
- const char *pClientID
- uint16_t clientIdLen
- uint16_t keepAliveIntervalInSec
- bool isCleanSession
- bool isWillMsgPresent
- IoT_MQTT_Will_Options will
- char *pUsername
- uint16_t usernamelen
- char *pPassword
- uin16_t passwordLen

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_disconnect

```
IoT_Error_t aws_iot_mqtt_disconnect(AWS_IoT_Client *pClient)
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_get_client_state

Get the current state of the client.

```
ClientState aws_iot_mqtt_get_client_state(AWS_IoT_Client *pClient)
```

Client state is one of:

- CLIENT_STATE_INVALID = 0,
- CLIENT_STATE_INITIALIZED = 1,
- CLIENT_STATE_CONNECTING = 2,
- CLIENT_STATE_CONNECTED_IDLE = 3,
- CLIENT_STATE_CONNECTED_YIELD_IN_PROGRESS = 4,
- CLIENT_STATE_CONNECTED_PUBLISH_IN_PROGRESS = 5,
- CLIENT_STATE_CONNECTED_SUBSCRIBE_IN_PROGRESS = 6,
- CLIENT_STATE_CONNECTED_UNSUBSCRIBE_IN_PROGRESS = 7,
- CLIENT_STATE_CONNECTED_RESUBSCRIBE_IN_PROGRESS = 8,
- CLIENT_STATE_CONNECTED_WAIT_FOR_CB_RETURN = 9,
- CLIENT_STATE_DISCONNECTING = 10,
- CLIENT_STATE_DISCONNECTED_ERROR = 11,
- CLIENT_STATE_DISCONNECTED_MANUALLY = 12,
- CLIENT_STATE_PENDING_RECONNECT = 13

aws_iot_mqtt_get_network_disconnected_count

Get the number of disconnects that have occurred due to errors.

```
uint32_t aws_iot_mqtt_get_network_disconnected_count(AWS_IoT_Client *pClient)
```

aws_iot_mqtt_get_next_packet_id

Get the next packet id.

```
uint16_t aws_iot_mqtt_get_next_packet_id(AWS_IoT_Client *pClient)
```

aws_iot_mqtt_init

Initialize the MQTT client.

```
IoT_Error_t aws_iot_mqtt_init(
   AWS_IoT_Client *pClient,
   const IoT_Client_Init_Params *pInitParams)
```

The pInitParams points to a structure which contains:

- bool enableAutoReconnect
- char *pHostURL
- uint16_t port
- const char *pRootCALocation
- const char *pDeviceCertLocation
- const char *pDevicePrivateKeyLocation
- uint32_t mqttPacketTimeout_ms
- uint32_t mqttCommandTimeout_ms
- uint32_t tlsHandshakeTimeout_ms
- bool isSSLHostnameVerify
- iot_disconnect_handler disconnectHandler
- void *disconnectHandlerData

Includes:

- #include <aws_iot_mqtt_client_interface.h>

See also:

aws_iot_mqtt_is_client_connected

Determine if the device is currently connected.

```
bool aws_iot_mqtt_is_client_connected(AWS_IoT_Client *pClient)
```

aws_iot_mqtt_publish

```
IoT_Error_t aws_iot_mqtt_publish(
   AWS_IoT_Client *pClient,
   const char *pTopicName,
   uint16_t topicNameLen,
   IoT_Publish_Message_Params *pParams)
```

- pClient
- pTopicName
- topicNameLen
- `pParams` – A structure that includes:
  - `QoS qos` – Quality of service.
  - `uint8_t isRetained` – Is this a retained message. Not supported by AWS IoT.
  - `uint8_t isDup` – Is this message a duplicate.
  - `uint16_t id` – Handled automatically.
  - `void *payload` – Pointer to the payload to be published.
  - `size_t payloadLen` – Length of the payload.

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_reset_network_disconnected_count

Reset the network disconnect error counter to zero.

```
void aws_iot_mqtt_reset_network_disconnected_count(AWS_IoT_Client *pClient)
```

aws_iot_mqtt_resubscribe

```
IoT_Error_t aws_iot_mqtt_resubscribe(AWS_IoT_Client *pClient)
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_set_connect_params

Set the connection parameters.

```
IoT_Error_t aws_iot_mqtt_set_connect_params(
   AWS_IoT_Client *pClient,
   const IoT_Client_Connect_Params *pNewConnectParams)
```

The `pNewConnectParams` is a pointer to a structure containing:

- char struct_id
- MQTT_Ver_t MQTTVersion
- const char *pClientID
- uint16_t clientIdLen
- uint16_t keepAliveIntervalInSec
- bool isCleanSession
- bool isWillMsgPresent
- IoT_MQTT_Will_Options will
- char *pUsername
- uint16_t usernamelen
- char *pPassword
- uin16_t passwordLen

aws_iot_mqtt_set_disconnect_handler

Set the IoT client disconnect handler.

```
IoT_Error_t aws_iot_mqtt_set_disconnect_handler(
   AWS_IoT_Client *pClient,
   iot_disconnect_handler pDisconnectHandler,
   void *pDisconnectHandlerData)
```

Invoke a function when the client disconnects due to an error.

aws_iot_mqtt_subscribe

Subscribe to a topic.

```
IoT_Error_t aws_iot_mqtt_subscribe(
   AWS_IoT_Client *pClient,
   const char *pTopicName,
   uint16_t topicNameLen,
   QoS qos,
   pApplicationHandler_t pApplicationHandler,
   void *pApplicationHandlerData)
```

- `pClient` – Reference to the client.
- `pTopicName` – The name of the topic.
- `topicNameLen` – The length of the name of the topic.
- `qos` – Quality of service.
- `pApplicationHandler` – Handler function for this subscription.
- `pApplicationHandlerData` – Data for the handler function.

The callback handler for a subscription has the following signature:

```
typedef void (*pApplicationHandler_t)(
   AWS_IoT_Client *pClient,
   char *pTopicName, uint16_t topicNameLen,
   IoT_Publish_Message_Params *pParams,
   void *pClientData);
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_unsubscribe

```
IoT_Error_t aws_iot_mqtt_unsubscribe(
   AWS_IoT_Client *pClient,
   const char *pTopicFilter,
   uint16_t topicFilterLen)
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

aws_iot_mqtt_yield

```
IoT_Error_t aws_iot_mqtt_yield(
   AWS_IoT_Client *pClient,
   uint32_t timeout_ms)
```

Includes:

- #include <aws_iot_mqtt_client_interface.h>

JSON processing

The ESP-IDF provides a C implementation of JSON processing through the component called "`json`".

Note: There is a second JSON parsing option called "jsmn" which uses an alternate technique that exchanges minimalism for efficiency.

We can parse a JSON string using `cJSON_Parse()`. This will return a `cJSON` data structure. This data structure exposes:

- `valueint` – The number value as an integer.
- `valuedouble` – The number value as a double.
- `valuestring` – The string value.

To serialize a JSON object to a string representation we can call `cJSON_Print()`.

The complete list of functions are:

- cJSON_Parse
- cJSON_Print
- cJSON_PrintUnformatted
- cJSON_PrintBuffered
- cJSON_PrintPreallocated
- cJSON_Delete
- cJSON_GetArraySize
- cJSON_GetArrayItem
- cJSON_GetObjectItem
- cJSON_GetErrorPtr
- cJSON_CreateArray
- cJSON_CreateBool
- cJSON_CreateDoubleArray
- cJSON_CreateFalse
- cJSON_CreateFloatArray
- cJSON_CreateIntArray
- cJSON_CreateObject
- cJSON_CreateNull
- cJSON_CreateNumber
- cJSON_CreateString
- cJSON_CreateStringArray

- cJSON_CreateTrue
- cJSON_AddItemToArray
- cJSON_AddItemToObject
- cJSON_AddItemToObjectCS
- cJSON_AddItemReferenceToArray
- cJSON_AddItemReferenceToObject
- cJSON_DetachItemFromArray
- cJSON_DeleteItemFromArray
- cJSON_DetachItemFromObject
- cJSON_DeleteItemFromObject
- cJSON_InsertItemInArray
- cJSON_ReplaceItemInArray
- cJSON_ReplaceItemInObject
- cJSON_Duplicate
- cJSON_Minify
- cJSON_AddNullToObject
- cJSON_AddTrueToObject
- cJSON_AddFalseToObject
- cJSON_AddBoolToObject
- cJSON_AddNumberToObject
- cJSON_AddStringToObject
- cJSON_ArrayForEach
- cJSON_SetIntValue
- cJSON_SetNumberValue

Includes:

- #include <cJSON.h>

See also:

- Github: DaveGamble/cJSON
- JSMN
- JSON.org

HTTP/2 processing

The ESP-IDF provides an implementation of HTTP/2 parsing through "nghttp2".

See also:

- HTTP/2 C library and tools – nghttp2.org

Parsing XML - expat

The ESP-IDF provides a version of expat which is an XML parsing library.

See also:

- Expat home page
- Using Expat
- Expat reference document

Arduino – ESP32 HAL for UART

uartAvailable

```
uint32_t uartAvailable(uart_t *uart)
```

Includes:

- #include <esp32-hal-uart.h>

uartBegin

Initialize a UART for work.

```
uart_t *uartBegin(
   uint8_t uartNumber,
   uint32_t baudrate,
   uint32_t config,
   int8_t rxPin,
   int8_t txPin,
   uint16_t queueLen,
   bool inverted)
```

The `uartNumber` can be one of the three UARTs on the device. They are numbered 0, 1 and 2. The `rxPin` is the pin used for receiving incoming data, The `txPin` is the pin used to transmit data. If -1 is specified, then the UART is input or output only. It is invalid to specify -1 for both pins (i.e. a uart with neither input nor output). The config parameter is the configuration for the UART. A configuration is broken into three parameters:

- Number of bits – 5,6,7 or 8

- Parity checking – None, Even or Odd
- Number of stop bits – 1 or 2

Constants are defined for all possibilities in the form:

```
SERIAL_<NUMBER OF BITS><PARITY><NUMBER OF STOP BITS>
```

For example:

```
SERIAL_8N1
```

Means 8 bits, no parity check, 1 stop bit.

Includes:

- esp32-hal-uart.h

See also:

- uartEnd

uartEnd

Release resources associated with the uart.

```
void uartEnd(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

See also:

- uartBegin

uartFlush

```
void uartFlush(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

uartGetBaudRate

Get the current baud rate of the uart.

```
uint32_t uartGetBaudRate(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

See also:

- uartSetBaudRate

- uartBegin

uartGetDebug

```
int uartGetDebug()
```

Includes:

- esp32-hal-uart.h

See also:

- uartSetDebug

uartPeek

Retrieve the next byte to be read without actually consuming it.

```
uint8_t uartPeek(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

See also:

- uartRead

uartRead

Read a byte from the uart.

```
uint8_t uartRead(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

See also:

- uartAvailable
- uartPeek

uartSetBaudRate

Set the baud rate of the uart.

```
void uartSetBaudRate(uart_t *uart, uint32_t baud_rate)
```

Includes:

- esp32-hal-uart.h

- uartGetBaudRate
- uartBegin

uartSetDebug

Specify a uart as the target of debug data.

```
void uartSetDebug(uart_t *uart)
```

Includes:

- esp32-hal-uart.h

See also:

- uartGetDebug

uartWrite

Write a single byte to the uart.

```
void uartWrite(uart_t *uart, uint8_t c)
```

This is a blocking call and waits for the transmission to complete.

Includes:

- esp32-hal-uart.h

uartWriteBuf

Write a buffer of bytes to the uart.

```
void uartWriteBuf(
   uart_t *uart,
   const uint8_t *data,
   size_t len)
```

Writes the buffer to the uart blocking until completion.

Includes:

- esp32-hal-uart.h

Arduino – ESP32 HAL for I2C

See also:

- Using Arduino I2C libraries

i2cAttachSCL

Associate the bus's clock signal with a specific GPIO pin.

```
i2c_err_t i2cAttachSCL(i2c_t *i2c, int8_t scl)
```

i2cAttachSDA

```
i2c_err_t i2cAttachSDA(i2c_t *i2c, int8_t sda)
```

Associate the bus's data signal with a specific GPIO pin.

i2cDetachSCL

```
i2c_err_t i2cDetachSCL(i2c_t *i2c, int8_t scl)
```

Disassociate the bus's clock signal with a specific GPIO pin.

i2cDetachSDA

```
i2c_err_t i2cDetachSDA(i2c_t *i2c, int8_t sda)
```

Disassociate the bus's data signal with a specific GPIO pin.

i2cGetFrequency

```
uint32_t i2cGetFrequency(i2c_t *i2c)
```

i2cInit

Initialize access to the I2C bus.

```
i2c_t *i2cInit(
   uint8_t i2c_num,
   uint16_t slave_addr,
   bool addr_10bit_en)
```

The `i2c_num` is the identify of the I2C bus we are planning on using. The `slave_addr` is the address of the slave device we wish to appear as. The `addr_10bit_en` would be set to true if we are using 10 bit slave addressing.

The return from this call is a handle that is passed into other calls to refer to this bus/device.

See also:

- Using Arduino I2C libraries

i2cRead

Read data from the I2C bus.

```
i2c_err_t i2cRead(
   i2c_t *i2c,
   uint16_t address,
   bool addr_10bit,
   uint8_t *data,
   uint8_t len,
   bool sendStop)
```

The `i2c` is a handle to an I2C bus. The `address` is the address of the device with which we are communicating. The `addr_10bit` is set to true if we are using 10 bit addressing. The `data` is a pointer to a buffer to be filled in by the read. The `len` is the number of bytes to read. The `sendStop` is a flag which indicates whether we should send a stop indication.

i2cSetFrequency

```
i2c_err_t i2cSetFrequency(i2c_t *i2c, uint32_t clk_speed)
```

i2cWrite

Write data down the I2C bus.

```
i2c_err_t i2cWrite(
   i2c_t *i2c,
   uint16_t address,
   bool addr_10bit,
   uint8_t *data,
   uint8_t len,
   bool sendStop)
```

The `i2c` is a handle to an I2C bus. The `address` is the address of the device with which we are communicating. The `addr_10bit` is set to true if we are using 10 bit addressing. The `data` is a pointer to a buffer of data to be transmitted. The `len` is the size of the data to transmit. The `sendStop` should be true if we are sending a stop flag at the end of the transmission.

Arduino – ESP32 HAL for SPI

See also:

- The Arduino Hardware Abstraction Layer SPI

spiAttachMISO

Specify the pin to use for MISO.

```
void spiAttachMISO(spi_t *spi, int8_t miso)
```

spiAttachMOSI

Specify the pin to use for MOSI.

```
void spiAttachMOSI(spi_t *spi, int8_t mosi)
```

spiAttachSCK

Specify the pin to use for SCK.

```
void spiAttachSCK(spi_t *spi, int8_t sck)
```

spiAttachSS

Specify the pin to use for SS.

```
void spiAttachSS(spi_t *spi, uint8_t cs_num, int8_t ss)
```

spiClockDivToFrequency

Convert a clock divider to a frequency.

```
uint32_t spiClockDivToFrequency(uint32_t clockDiv)
```

A given clock divider value results in an SPI clock frequency. This function takes a clock divider and returns the frequency that would result from using the divider.

spiDetachMISO

```
void spiDetachMISO(spi_t *spi, int8_t miso)
```

spiDetachMOSI

```
void spiDetachMOSI(spi_t *spi, int8_t mosi)
```

spiDetachSCK

```
void spiDetachSCK(spi_t *spi, int8_t sck)
```

spiDetachSS

Detach the pin from SS.

```
void spiDetachSS(spi_t *spi, int8_t ss)
```

spiDisableSSPins

```
void spiDisableSSPins(spi_t *spi, uint8_t cs_mask)
```

spiEnableSSPins

```
void spiEnableSSPins(spi_t *spi, uint8_t cs_mask)
```

spiFrequencyToClockDiv

Calculate a clock divider needed for a given frequency.

```
uint32_t spiFrequencyToClockDiv(uint32_t freq)
```

Internally, the SPI clock is based on a given clock divider. Commonly, we want to think about SPI clock frequencies. This function calculates the divider necessary to achieve a specific frequency.

spiGetBitOrder

Retrieve the current bit order.

```
uint8_t spiGetBitOrder(spi_t *spi)
```

Retrieve the current bit order being used. It will be one of:

- SPI_LSBFIRST
- SPI_MSBFIRST

spiGetClockDiv

Get the underlying clock divider.

```
uint32_t spiGetClockDiv(spi_t *spi)
```

For a given SPI device, get the clock divider. Note that this is **not** the same as the SPI clock frequency.

spiGetDataMode

Retrieve the current data mode.

```
uint8_t spiGetDataMode(spi_t *spi)
```

Retrieve the current data mode of SPI. It should be one of:

- SPI_MODE0
- SPI_MODE1
- SPI_MODE2

- SPI_MODE3

spiRead

Read a buffer of data through SPI.

```
void spiRead(spi_t *spi, uint32_t *out, uint8_t len)
```

spiReadByte

Read a single byte of data through SPI.

```
uint8_t spiReadByte(spi_t *spi)
```

spiReadLong

Read a 32 bit value (a long) from SPI.

```
uint32_t spiReadLong(spi_t *spi)
```

spiReadWord

Read a 16 bit value (a word) from SPI.

```
uint16_t spiReadWord(spi_t *spi)
```

spiSetBitOrder

Set the bit order.

```
void spiSetBitOrder(spi_t *spi, uint8_t bitOrder)
```

The `bitOrder` should be the bit order to be used. It should be one of:

- SPI_LSBFIRST
- SPI_MSBFIRST

spiSetClockDiv

Set the clock based on a clock divisor.

```
void spiSetClockDiv(spi_t *spi, uint32_t clockDiv)
```

spiSetDataMode

Set the data mode.

```
void spiSetDataMode(spi_t *spi, uint8_t dataMode)
```

The `dataMode` is the data mode of SPI. It should be one of:

- SPI_MODE0
- SPI_MODE1
- SPI_MODE2
- SPI_MODE3

spiSSClear

```
void spiSSClear(spi_t *spi)
```

spiSSDisable

```
void spiSSDisable(spi_t *spi)
```

spiSSEnable

```
void spiSSEnable(spi_t *spi)
```

spiSSSet

```
void spiSSSet(spi_t *spi)
```

spiStartBus

Initialize an instance of the bus.

```
spi_t *spiStartBus(
   uint8_t spi_num,
   uint32_t freq,
   uint8_t dataMode,
   uint8_t bitOrder)
```

Initialize an instance of one of the three buses specifying how it should be driven. The `spi_num` is the identity of one of the three buses. It may be one of:

- FSPI
- HSPI
- VSPI

Note that FSPI is believe to be off-limits for application usage.

The `freq` is the speed of the bus. For example 1000000 for 1MHz. The `dataMode` is the data mode of SPI. It should be one of:

- SPI_MODE0
- SPI_MODE1
- SPI_MODE2
- SPI_MODE3

The `bitOrder` should be the bit order to be used. It should be one of:

- SPI_LSBFIRST
- SPI_MSBFIRST

The return is a reference to an SPI instance.

See also:

- spiStopBus

spiStopBus

Release the resources for an SPI bus.

```
void spiStopBus(spi_t *spi)
```

The `spi` is a handle to a bus previously created with a call to `spiStartBus()`.

See also:

- spiStartBus

spiTransferBits

Transfer bits over SPI and optionally receive response bits.

```
void spiTransferBits(
   spi_t *spi,
   uint32_t sendData,
   uint32_t *receiveData,
   uint8_t bits)
```

The `bits` is the number of bits to send. A maximum of 32 bits at one shot. The `sendData` is the data to send, the `receiveData` is a pointer to storage to hold retrieved data. It may be NULL to indicate that we don't want to receive data. The spi is the handle of the SPI interface.

See also:

- spiStartBus

spiTransferBytes

Write and read data through SPI.

```
void spiTransferBytes(
   spi_t *spi,
   uint8_t *sendData,
   uint8_t *receiveData,
   uint32_t size)
```

This is the key function in SPI. Here we specify a buffer of data to send through SPI and a buffer of data to read from SPI. If we don't need to send data we can specify NULL for the `sendData` and if we don't need to receive, we can specify NULL for `receiveData`. The `size` is the number of bytes that we are to send and receive.

spiWaitReady

```
void spiWaitReady(spi_t *spi)
```

spiWrite

Write a buffer of data out through SPI.

```
void spiWrite(spi_t *spi, uint32_t *data, uint8_t len)
```

spiWriteByte

Write a single byte of data out through SPI.

```
void spiWriteByte(spi_t *spi, uint8_t data)
```

spiWriteLong

Write a 32 bit value (a long) out through SPI.

```
void spiWriteLong(spi_t *spi, uint32_t data)
```

spiWriteWord

Write a 16 bit value (a word) out through SPI.

```
void spiWriteWord(spi_t *spi, uint16_t data)
```

Newlib

When we think of the C programming language we must realize that is exactly what it is … a programming language. Functions like "printf" and "strcpy" and "malloc" are not part of the C language. They are functions commonly provided by the environment in which the compiled program executes. These functions are so prevalent that we assume them just to be there when C programming but the reality is that something has to provide them. On a Unix/Linux environment, it is the kernel and support user level libraries. But what of our ESP32? The answer is an open source library called "newlib". Newlib takes the common specifications for many of the functions that we expect to be there on a C platform and provides an open source implementation of them. Within the ESP32 environment a version of newlib is provided that has been mapped for the ESP32.

From the newlib library, the following functions are exported (and others):

See also:

- newlib

abort

Results in the abnormal termination of the environment (a halt).

```
void abort()
```

See also:

- man(3) – abort

abs

Compute the absolute (non-negative) value of an integer.

```
int abs(int val)
```

Includes:

- stdlib.h

See also:

- man(3) – abs

asctime

Format a struct tm time value into a text string.

```
char *asctime(const struct tm *tm)
char *asctime_r(const struct tm *tm, char *buf)
```

The `struct tm` contains:

- tm_sec – 0-59

- tm_min – 0-59
- tm_hour – 0-23
- tm_mday – 1-31
- tm_mon – 0-11
- tm_year – Years since 1900 (eg. 2017 = 117)
- tm_wday – 0-6
- tm_yday – 0-365
- tm_isdst – 1/0

The output format is the same as `ctime()`.

See also:

- Timers and time
- man(3) – asctime

atoi

Convert a string to an integer.

```
int atoi(const char *s)
```

See also:

- man(3) – atoi

atol

Convert a string to an integer.

```
long atol(const char *s)
```

See also:

- man(3) – atol

bzero

Zero an area of memory.

```
void bzero(void *s, size_t n)
```

See also:

- man(3) – bzero

calloc

Allocate an area of storage for a number of fixed size entries.

```
void *calloc(size_t c, size_t n)
```

See also:

- free
- malloc
- realloc
- man(3) – calloc

check_pos

close

Close a file descriptor.

```
int close(int fd)
```

See also:

- man(3) – close

creat

Create a new file.

```
int create(const char *path, mode_t mode)
```

Includes:

- fcntl.h

See also:

- man(3) – creat

ctime

Convert time_t data to a string.

```
char *ctime(const time_t *timep)
char *ctime_r(const ctime_t *timep, char *buf)
```

The `time_t` can be obtained from a call to `time()`. An example of output would be:

```
Thu Jan  1 00:00:00 1970\n
```

The length of the string is a constant 25 characters (including the newline). This means we can efficiently terminate the line before the newline or just print the first 24 characters if we don't want to print the newline.

Note that a newline is found on the end.

Includes:

- time.h

See also:

- Timers and time
- time
- man(3) – ctime

div

Compute quotient and remainder.

```
div_t div(int numerator, int denominator)
```

Includes:

- stdlib.h

See also:

- man(3) – div

environ

Access environment variables.

```
char **environ
```

Includes:

- unistd.h

See also:

- man(3) – environ

fclose

Close a stream.

```
int fclose(FILE *fp)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – fclose

fflush

Flush a stream.

```
int fflush(FILE *fp)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – fflush

fmemopen

Perform stream I/O on a buffer of memory.

```
FILE *fmemopen(void *buf, size_t size, const char *mode)
```

The return is a FILE pointer that can be used with other stream I/O functions. The mode is the same as found for `fopen()`.

Includes:

- #include <stdio.h>

See also:

- man(3) – fmemopen

fprintf

Print to a stream.

```
int printf(const char *format, …)
```

Includes:

- #include <stdio.h>

See also:

- printf
- man(3) – fprintf

fread

Read from the file into a memory area.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – fread

free

Free allocated memory.

```
void free(void *p)
```

Releases memory previously allocated by a call to `malloc()`.

Includes:

- #include <stdlib.h>

See also:

- calloc
- malloc
- realloc
- man(3) – free

fscanf

Scan from a stream.

```
int fscanf(FULE *stream, const char *format, …)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – fscanf

fseek

fstat

Obtain the stats of an open file.

```
int fstat(int fd, struct stat *buf)
```

See also:

- man(2) – fstat

fwrite

Write data to an output stream.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – fwrite

gettimeofday

Get the current time.

`int gettimeofday(struct timeval *tv, struct timezone *tz)`

Return the current time of day as a struct timeval. This structure contains the following fields:

- `time_t tv_sec` – Seconds since epoch.
- `suseconds_t tv_usec` – Microseconds within current second.

A sample set of C functions is available here <insert URL> that provides some simple routines for working with struct timeval including:

- `struct timeval timeval_add(struct timeval *a, struct timeval *b)` – Add two time values together.
- `void timeval_addMsecs(struct timeval *a, uint32_t msecs)` – Add milliseconds to time value.
- `uint32_t timeval_durationBeforeNow(struct timeval *a)` – Determine how many milliseconds the supplied time value was in the past.
- `uint32_t timeval_durationFromNow(struct timeval *a)` – Determine how many milliseconds the given future time value will be from now or 0 if it has already passed.
- `struct timeval timeval_sub(struct timeval *a, struct timeval *b)` – Subtract one time value from another. The first parameter should be further along the time line (sooner) than the second (later).
- `uint32_t timeval_toMsecs(struct timeval *a)` – Return the number of milliseconds within the time value.

Includes:

- #include <sys/time.h>

See also:

- settimeofday
- Timers and time
- man(2) – gettimeofday

gmtime

Break down a time into its components based on UTC.

```
struct tm *gmtime(const time_t *timep)
struct tm *gmtime_r(const time_t *timep, struct tm *result)
```

The time_t value can be obtained from a call to `time()`.

Includes:

- #include <time.h>

See also:

- Timers and time
- time
- man(3) – gmtime

isalnum

Determine if a character is alpha/numeric.

```
int isalnum(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isalnum

isalpha

Determine if a character is an alpha.

```
int isalpha(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isalpha

isascii

```
int isascii(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isascii

isblank

```
int isblank(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isblank

isdigit

```
int isdigit(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isdigit

islower

```
int islower(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – islower

isprint

```
int isprint(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isprint

ispunct

```
int ispunct(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – ispunct

isspace

```
int isspace(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isspace

isupper

```
int isupper(int c)
```

Includes:

- #include <ctype.h>

See also:

- man(3) – isupper

itoa

See also:

man(3) –

labs

```
long labs(long val)
```

Includes:

- #include <stdlib.h>

See also:

- man(3) – labs

ldiv

```
ldiv_t ldiv(long numerator, long demominator)
```

Includes:

- #include <stdlib.h>

See also:

- man(3) – ldiv

localtime

```
struct tm *localtime(const time_t *timep)
struct tm *localtime_r(const time_t *timep, struct tm *result)
```

Includes:

- #include <time.h>

See also:

- Timers and time
- man(3) – localtime

malloc

Allocate storage from the heap.

```
void *malloc(size_t n)
```

When no longer needed, the storage should be returned by a call to `free()`.

Includes:

- #include <stdlib.h>

See also:

- calloc
- free
- realloc
- man(3) – malloc

memchr

Find a character/byte within a given piece of memory.

```
void *memchr(const void *s, int c, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memchr

memcmp

Compare the content of one piece of memory against another.

```
int memcmp(const void *m1, const void *m2, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memcmp

memcpy

Copy one piece of memory to another.

```
int memcpy(void *dest, const void *src, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memcpy

memmove

Copy memory from one place to another handling overlaps.

```
void *memmove(void *dst, const void *src, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memmove

memrchr

Find a character within a piece of memory searching backwards from the end.

```
void *memrchr(const void *s, int c, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memrchr

memset

Set a piece of memory to a specific value.

```
void *memset(void *dst, int c, size_t n)
```

Includes:

- #include <string.h>

See also:

- man(3) – memset

mkdir

Create a directory.

```
int mkdir(const char *pathname, mode_t mode)
```

Returns 0 on success.

Includes:

- #include <sys/stat.h>

See also:

- man(2) – mkdir

mktime

```
time_t mktime(struct tm *tm)
```

Includes:

- #include <time.h>

See also:

- Timers and time
- man(3) – mktime

open

Open a file.

```
int open(const char *path, int oflag, ...)
```

The `path` is the path to the file to open.

The `oflag` is a set if bitwise flags used to control how the file is opened. One of the following must be supplied:

- O_RDONLY
- O_WRONLY
- O_RDWR

In addition, the following combination may also be supplied:

- O_APPEND
- O_CREAT
- O_EXCL
- O_TRUNC

For example:

```
struct stat statBuf;
int fd = open(path, O_RDONLY);
fstat(fd, &statBuf);
data = malloc(statBuf.st_size);
read(fd, data, statBuf.st_size);
close(fd);
```

On return, the value is the handle to the open file descriptor or -1 if there was an error in which case "`errno`" will be set.

Includes:

- #include <fcntl.h>

See also:

- close
- stat
- fstat
- read
- write
- man(3) – open

open_memstream

Create a dynamic buffer for writing.

```
FILE *open_memstream(char **ptr, size_t *sizeloc)
```

The return is a FILE pointer that can be used to write into. After an `fclose()` or an `fflush()` the `ptr` and `sizeloc` are updated to point to the buffer and size. The storage should be released with a call to `free()` when done.

Includes:

- #include <stdio.h>

See also:

- man(3) – open_memstream

printf

Print a formatted string to stdout.

```
int printf(const char *format, ...)
```

Includes:

- #include <stdio.h>

See also:

- fprintf
- man(3) – printf

qsort

```
void qsort(
   void base,
   size_t nmemb,
   size_t size,
   int (*compar)(const void *, const void *))
```

Includes:

- #include <stdlib.h>

See also:

- man(3) – qsort

rand

Generate a random number.

```
int rand()
```

Return a random number. Note that the result is an integer which is signed.

Includes:

- #include <stdlib.h>

See also:

- man(3) – rand

read

Read data from a file or socket.

```
ssize_t read(int fd, void *buf, size_t nbytes)
```

Includes:

- #include <unistd.h>

See also:

- man(3) – read

readdir

realloc

Reallocate a portion of malloced data to a new size.

```
void *realloc(void *originalData, size_t newSize)
```

Includes:

- #include <stdlib.h>

See also:

- free
- calloc
- malloc
- man(3) – realloc

scanf

```
int scanf(const char *format, …)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – scanf

setenv

Set the value of an environment variable.

```
int setenv(const char *name, const char *value, int overwrite)
```

setlocale

```
char *setlocale(int category, const char *locale)
```

See also:

- man(3) – setlocale

settimeofday

Set the current time of day.

```
int settimeofday(const struct timeval *tv, const struct timezone *tz)
```

Includes:

- #include <sys/time.h>

See also:

- gettimeofday
- Timers and time
- man(2) – settimeofday

sprintf

Perform a printf to a region of memory.

```
int sprintf(char *out, const char *format, ...)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – sprintf

srand

Set the random number seed.

```
void srand(unsigned int seed)
```

See also:

- man(3) – srand

sscanf

```
int sscanf(const char *str, const char *format, ...)
```

See also:

- man(3) – scanf

stat

Get the status of a file by path.

```
int stat(const char *path, struct stat *buf)
```

Includes:

#include <

See also:

- man(2) – stat

strcasecmp

```
int strcasecmp(const char *s1, const char *s2)
```

See also:

- man(3) – strcasecmp

strcasestr

```
char *strcasestr(const char *haystack, const char *needle)
```

See also:

- man(3) – strcasestr

strcat

Concatenate two strings together.

```
char *strcat(char *dst, const char *src)
```

See also:

- man(3) – strcat

strchr

Search for a character in a string.

```
char *strchr(const char *s, int c)
```

See also:

- man(3) – strchr

strcmp

Compare two strings.

```
int strcmp(const char *s1, const char *s2)
```

See also:

- man(3) – strcmp

strcoll

```
int strcoll(const char *s1, const char *s2)
```

See also:

- man(3) – strcoll

strcpy

Copy a null terminated string.

```
char *strcpy(char *dst, const char *src)
```

Copy the null terminated string pointed to by "`src`" to the storage area pointed to by "`dst`". The destination must have enough storage to hold the source string including its null terminator.

Includes:

- #include <string.h>

See also:

- man(3) – strcpy

strcspn

```
size_t strcspn(const char *s, const char *reject)
```

See also:

- man(3) – strcspn

strdup

Duplicate the null terminate string via malloc().

```
char *strdup(const char *s)
```

A simple but powerful function. The null terminated string supplied as an input parameter is duplicated bu mallocing enough storage to hold a copy and then the string is actually copied. The returned value is a pointer to the new string copy. The storage for the new string should eventually be released with a call to `free()`.

See also:

- man(3) – strdup

strerror

Convert an error code into a string representation.

```
char *strerror(int errnum)
```

Includes:

- #include <string.h>

- #include <errno.h>

See also:

- man(3) – strerror

strftime

```
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm)
```

See also:

- man(3) – strftime

strlcat

```
size_t strlcat(char *dst, const char *src, size_t size)
```

See also:

- man(3) – strlcat

strlcpy

```
size_t strlcpy(char *dst, const char *src, size_t size)
```

See also:

- man(3) – strlcpy

strlen

Return the length of a null terminated string.

```
size_t strlen(const char *s)
```

Return the length of a null terminated string.

See also:

- man(3) – strlen

strncasecmp

```
int strncasecmp(const char *s1, const char *s2, size_t n)
```

See also:

- man(3) – strncasecmp

strncat

```
char *strncat(char *dst, const char *src, size_t count)
```

See also:

- man(3) – strncat

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n)
```

See also:

- man(3) – strncmp

strncpy

```
char *strncpy(char *dst, const char *src, size_t n)
```

See also:

- man(3) – strncpy

strndup

```
char *strdup(const char *s)
```

See also:

- man(3) – strndup

strnlen

```
size_t strnlen(const char *s, size_t maxlen)
```

See also:

- man(3) – strnlen

strrchr

```
char *strrchr(const char *s, int c)
```

See also:

- man(3) – strrchr

strsep

```
char *strsep(char **stringp, const char *delim)
```

See also:

- man(3) – strsep

strspn

```
size_t strspn(const char *s, const char *accept)
```

See also:

- man(3) – strspn

strstr

`char *strstr(const char *haystack, const char *needle)`

See also:

- man(3) – strstr

strtod

`double strtod(const char *nptr, char **endptr)`

See also:

- man(3) – strtod

strtof

`float strtof(const char *nptr, char **endptr)`

See also:

- man(3) – strtof

strtol

`long strtol(const char *str, char **endptr, int base)`

See also:

- man(3) – strtol

strtoul

`unsigned long strtoul(const char *nptr, char **endptr, int base)`

See also:

- man(3) – strtoul

strupr

See also:

man(3) – strupr

time

Retrieve the current time as the number of seconds since the epoch.

```
time_t time(time_t *t)
```

If t is supplied, then the time will be stored at that location. If NULL, the time will only be returned as a return value.

Includes:

- #include <time.h>

See also:

- Timers and time
- man(2) – time

times

```
clock_t times(struct tms *buffer)
```

The struct tms contains:

- clock_t tms_utime – user time
- clock_t tms_stime – system time
- clock_t tms_cutime – user time children
- clock_t tms_cstime – system time children

See also:

- Timers and time
- man(3) – times

toascii

```
int toascii(int c)
```

See also:

- man(3) – toascii

tolower

```
int tolower(int c)
```

See also:

- man(3) – tolower

toupper

```
int toupper(int c)
```

See also:

- man(3) – toupper

tzset

Use the time zone information found in the TZ environment variable.

```
void tzset()
```

Examples …

```
CST+6
```

for example, in code:

```
setenv("TZ", "CST+6", 1);
tzset();
```

See also:

- man(3) – tzset
- Specifying the Time Zone with TZ

ungetc

```
int ungetc(int c, FILE *fp)
```

See also:

- man(3) – ungetc

unlink

```
int unlink(const char *pathName)
```

Delete a file.

Includes:

- #include <unistd.h>

See also:

- man(2) – unlink

utoa

See also:

- man(3) – utoa

vprintf

```
int vprintf(const char *format, va_list ap)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – vrprintf

vscanf

```
int vscanf(const char *format, va_list ap)
```

Includes:

- #include <stdio.h>

See also:

- man(3) – vscanf

write

```
ssize_t write(int fd, const void *buf, size_t nbyte)
```

See also:

- man(3) – write

SPIFFs API

See also:

- Spiffs File System
- Github: pellepl/spiffs

SPIFFS_check

Perform a consistency check on the file system.

```
s32_t SPIFFS_check(spiffs *fs)
```

- `fs` – The file system to be checked for consistency.

SPIFFS_clearerr

Clear the last error.

```
void SPIFFS_clearerr(spiffs *fs)
```

- `fs` – The file system reference.

SPIFFS_close

Close a previously opened file.

```
s32_t SPIFFS_close(spiffs *fs, spiffs_file fh)
```

- `fs` – The file system reference.
- `fh` – A handle to a previously opened file.

SPIFFS_closedir

Close a previously opened directory.

```
s32_t SPIFFS_closedir(spiffs_DIR *dir)
```

- `dir` – A directory previously opened by `SPIFFS_opendir()`.

SPIFFS_creat

Create a new file.

```
s32_t SPIFFS_creat(spiffs *fs, const char *path, spiffs_mode mode)
```

- `fs` – The file system reference.
- `path` – The path to the new file.
- `mode` – Ignored.

SPIFFS_eof

Determine if we are at the end of the file.

```
s32_t SPIFFS_eof(spiffs *fs, spiffs_file fh)
```

- `fs` – The file system reference.
- `fh` – A handle to an open file.

SPIFFS_errno

Returns the last error detected by performing a SPIFFs API call.

```
s32_t SPIFFS_errno(spiffs *fs)
```

- `fs` – The file system reference.

The defined codes are:

- `SPIFFS_OK` – 0
- `SPIFFS_ERR_NOT_MOUNTED` – -10000

- `SPIFFS_ERR_FULL` – -10001 – ENOSPC
- `SPIFFS_ERR_NOT_FOUND` – -10002 – ENOENT
- `SPIFFS_ERR_END_OF_OBJECT` – -10003
- `SPIFFS_ERR_DELETED` – -10004
- `SPIFFS_ERR_NOT_FINALIZED` – -10005
- `SPIFFS_ERR_NOT_INDEX` – -10006
- `SPIFFS_ERR_OUT_OF_FILE_DESCS` – -10007 – ENFILE
- `SPIFFS_ERR_FILE_CLOSED` – -10008
- `SPIFFS_ERR_FILE_DELETED` – -10009
- `SPIFFS_ERR_BAD_DESCRIPTOR` – -10010
- `SPIFFS_ERR_IS_INDEX` – -10011
- `SPIFFS_ERR_IS_FREE` – -10012
- `SPIFFS_ERR_INDEX_SPAN_MISMATCH` – -10013
- `SPIFFS_ERR_DATA_SPAN_MISMATCH` – -10014
- `SPIFFS_ERR_INDEX_REF_FREE` – -10015
- `SPIFFS_ERR_INDEX_REF_LU` – -10016
- `SPIFFS_ERR_INDEX_REF_INVALID` – -10017
- `SPIFFS_ERR_INDEX_FREE` – -10018
- `SPIFFS_ERR_INDEX_LU` – -10019
- `SPIFFS_ERR_INDEX_INVALID` – -10020
- `SPIFFS_ERR_NOT_WRITABLE` – -10021
- `SPIFFS_ERR_NOT_READABLE` – -10022
- `SPIFFS_ERR_CONFLICTING_NAME` – -10023
- `SPIFFS_ERR_NOT_CONFIGURED` – -10024
- `SPIFFS_ERR_NOT_A_FS` – -10025 – Commonly seen when we mount a file system at an SPI address range and the data found on flash isn't formatted as a SPIFFs file system. We can then use `SPIFFS_format()` to format the data and attempt the mount again.
- `SPIFFS_ERR_MOUNTED` – -10026

- `SPIFFS_ERR_ERASE_FAIL` – -10027
- `SPIFFS_ERR_MAGIC_NOT_POSSIBLE` – -10028
- `SPIFFS_ERR_NO_DELETED_BLOCKS` – -10029
- `SPIFFS_ERR_FILE_EXISTS` – -10030 – EEXIST
- `SPIFFS_ERR_NOT_A_FILE` – -10031 – EBADF
- `SPIFFS_ERR_RO_NOT_IMPL` – -10032
- `SPIFFS_ERR_RO_ABORTED_OPERATION` – -10033
- `SPIFFS_ERR_PROBE_TOO_FEW_BLOCKS` – -10034
- `SPIFFS_ERR_PROBE_NOT_A_FS` – -10035
- `SPIFFS_ERR_NAME_TOO_LONG` – -10036
- `SPIFFS_ERR_IX_MAP_UNMAPPED` – -10037
- `SPIFFS_ERR_IX_MAP_MAPPED` – -10038
- `SPIFFS_ERR_IX_MAP_BAD_RANGE` – -10039
- `SPIFFS_ERR_INTERNAL` – -10050
- `SPIFFS_ERR_TEST` – -10100

SPIFFS_fflush

Flush the content of a file to flash.

```
s32_t SPIFFS_fflush(spiffs *fs, spiffs_file fh)
```

- `fs` – A reference to the file system structure.
- `fh` – A file handle to a previously opened file.

SPIFFS_format

Format the file system.

```
s32_t SPIFFS_format(spiffs *fs)
```

- `fs` – A reference to the file system structure.

SPIFFS_fremove

Remove a file by file handle.

```
s32_t SPIFFS_fremove(spiffs *fs, spiffs_file fh)
```

- `fs` – A reference to the file system structure.
- `fh` – A file handle to a previously opened file.

SPIFFS_fstat

```
s32_t SPIFFS_fstat(spiffs *fs, spiffs_file fh, spiffs_stat *s)
```

- `fs` – Reference to the file system structure.
- `fh` – A file handle that has been previously opened.
- `s` - A pointer to a `spiffs_stat` data structure that will be populated.
  - obj_id
  - `size` – The size of the file.
  - `type` – The type of entry. Possibilities are:
    - `SPIFFS_TYPE_FILE` – Entry is a file.
    - `SPIFFS_TYPE_DIR` – Entry is a directory.
    - `SPIFFS_TYPE_HARD_LINK` – Entry is a hard link.
    - `SPIFFS_TYPE_SOFT_LINK` – Entry is a soft link.
  - `pix` – Page index.
  - `name` – The name of the file.

SPIFFS_gc

Perform a garbage collection.

```
s32_t SPIFFS_gc(spiffs *fs, u32_t size)
```

- `fs` – Reference to the file system structure.
- `size` – The amount of space being sought.

SPIFFS_gc_quick

```
s32_t SPIFFS_gc_quick(spiffs *fs, u16_t max_free_pages)
```

- `fs` – Reference to the file system structure.
- max_free_pages

SPIFFS_info

Determine information about the file system.

```
s32_t SPIFFS_info(spiffs *fs, u32_t *total, u32_t *used)
```

- `fs` – A reference to the file system structure.
- `total` – The total size of the file system.
- `used` – The number of bytes used from the file system.

SPIFFS_lseek

Change the file pointer.

```
s32_t SPIFFS_lseek(
   spiffs *fs,
   spiffs_file fh,
   s32_t offs,
   int whence)
```

- `fs` – A reference to the file system structure.
- `fh` – A file handle to an open file.
- `offs` – The numeric by which to change the file pointer.
- `whence` – An indication of how to change the file pointer:
  - `SPIFFS_SEEK_SET` – Change the file pointer to an absolute value.
  - `SPIFFS_SEEK_CUR` – Change the file pointer with reference to its current value.
  - `SPIFFS_SEEK_END` – Change the file pointer relative to the end of the file.

SPIFFS_mount

Mount a file system specifying the flash memory address range.

```
s32_t SPIFFS_mount(
   spiffs *fs,
   spiffs_config *config,
   u8_t *work,
   u8_t *fd_space,
   u32_t fd_space_size,
   void *cache,
   u32_t cache_size,
   spiffs_check_callback check_cb_f)
```

- `fs` – The File System control structure.
- `config` – The physical and logical configuration of the file system.
- `work` – A memory buffer that should be 2 * log page size used for local work.

- `fd_space` – A pointer to memory to hold file descriptor working space. Typically sized at the max number of open file descriptors * sizeof(uint32).
- `fd_space_size` – The size of the file descriptor space.
- `cache` – Cache memory.
- `cache_size` – The size of cache memory. An example would be (log page size + 32) x 4.
- `check_cb_f` – Callback function invoked to report file system inconsistencies. Can be NULL.

SPIFFS_mounted

Determine whether or not the file system is mounted.

```
u8_t SPIFFS_mounted(spiffs *fs)
```

- `fs` – The file system to be checked.

SPIFFS_open

Open a new or existing file.

```
spiffs_file SPIFFS_open(
   spiffs *fs,
   const char *path,
   spiffs_flags flags,
   spiffs_mode mode)
```

- `fs` – The reference to the file system control data.
- `path` – The path of the file to be opened or created.
- `flags` – Flags controlling access to the file.
    - SPIFFS_O_APPEND
    - SPIFFS_O_CREAT
    - SPIFFS_O_DIRECT
    - SPIFFS_O_EXCL
    - SPIFFS_O_RDONLY
    - SPIFFS_O_RDWR
    - SPIFFS_O_TRUNC
    - SPIFFS_O_WRONLY

- `mode` – Ignored.

SPIFFS_open_by_dirent

Open a file by its directory entry.

```
spiffs_file SPIFFS_open_by_dirent(
   spiffs *fs,
   struct spiffs_dirent *entry,
   spiffs_flags flags,
   spiffs_mode mode)
```

- `fs` – The reference to the file system control data.
- `entry` – The directory entry to be opened.
- `flags` – Flags controlling access to the file.
  - SPIFFS_O_APPEND
  - SPIFFS_O_DIRECT
  - SPIFFS_O_EXCL
  - SPIFFS_O_RDONLY
  - SPIFFS_O_RDWR
  - SPIFFS_O_TRUNC
  - SPIFFS_O_WRONLY
- `mode` – Ignored.

SPIFFS_open_by_page

Open a file given the page it exists within the file system.

```
spiffs_file SPIFFS_open_by_page(
   spiffs *fs,
   spiffs_page_ix page_ix,
   spiffs_flags flags,
   spiffs_mode mode)
```

- `fs` – The reference to the file system control data.
- `page_ix` – The page to open.
- `flags` – Flags controlling access to the file.
  - SPIFFS_O_APPEND
  - SPIFFS_O_DIRECT
  - SPIFFS_O_EXCL

  - SPIFFS_O_RDONLY
  - SPIFFS_O_RDWR
  - SPIFFS_O_TRUNC
  - SPIFFS_O_WRONLY
- `mode` – Ignored.

SPIFFS_opendir

Open a directory to work with it.

```
spiffs_DIR *SPIFFS_opendir(spiffs *fs, const char *name, spiffs_DIR *dir)
```

- `fs` – The reference to the file system control data.
- `name` – The name of a directory.
- `dir` – Pointer to the directory structure to be populated.

The return is an opaque data type referring to the directory.

SPIFFS_read

Read data from an open file.

```
s32_t SPIFFS_read(
   spiffs *fs,
   spiffs_file fh,
   void *buf, s32_t len)
```

- `fs` – Reference to the file system structure.
- `fh` – The file handle to a previously opened file.
- `buf` – The buffer into which to read the data.
- `len` – The size of the buffer in bytes.

The return is the number of bytes read or -1 if there was an error.

SPIFFS_readdir

Read the content of a directory.

```
struct spiffs_dirent *SPIFFS_readdir(spiffs_DIR *dir, struct spiffs_dirent *dirent)
```

- `dir` – The directory to be read. An opaque data type returned by `SPIFFS_opendir()`.

- `dirent` – A directory entry that was read.
    - `obj_id` – Internal id of the file.
    - `name` – Name of the file.
    - `type (spiffs_obj_type)` – Type of the file. Possibilities are:
        - `SPIFFS_TYPE_FILE` – Entry is a file.
        - `SPIFFS_TYPE_DIR` – Entry is a directory.
        - `SPIFFS_TYPE_HARD_LINK` – Entry is a hard link.
        - `SPIFFS_TYPE_SOFT_LINK` – Entry is a soft link.
    - `size` – Size of the file.
    - `pix` – Page index of the file.

Return NULL on error of end of stream of entries.

SPIFFS_remove

Remove a file by path.

```
s32_t SPIFFS_remove(spiffs *fs, const char *path)
```

- `fs` – Reference to the file system structure.
- `path` – The path to the file to be removed.

SPIFFS_rename

Rename a file.

```
s32_t SPIFFS_rename(spiffs *fs, const char *oldPath, const char *newPath)
```

- `fs` – Reference to the file system structure.
- `oldPath` – The old path of the file.
- `newPath` – The new path of the file.

SPIFFS_stat

Get information about a file.

```
s32_t SPIFFS_stat(spiffs *fs, const char *path, spiffs_stat *s)
```

- `fs` – Reference to the file system structure.
- `path` – The path to the file to be examined.

- s - A pointer to a spiffs_stat data structure that will be populated.
  - obj_id
  - size – The size of the file.
  - type – The type of entry. Possibilities are:
    - SPIFFS_TYPE_FILE – Entry is a file.
    - SPIFFS_TYPE_DIR – Entry is a directory.
    - SPIFFS_TYPE_HARD_LINK – Entry is a hard link.
    - SPIFFS_TYPE_SOFT_LINK – Entry is a soft link.
  - pix – Page index.
  - name – The name of the file.

SPIFFS_tell

Find position within file.

```
s32_t SPIFFS_tell(spiffs *fs, spiffs_file fh)
```

- fs – Reference to the file system structure.
- fh – An open file handle.

SPIFFS_unmount

Un-mount a previously mounted file system.

```
void SPIFFS_unmount(spiffs *fs)
```

The fs is the file system structure previously populated from a SPIFFS_mount() call.

See also:

- SPIFFS_mount

SPIFFS_write

Write data into a file.

```
s32_t SPIFFS_write(
   spiffs *fs,
   spiffs_file fh,
   void *buf, s32_t len)
```

- fs – A reference to the file system structure.

- `fh` – A file handle previously opened.
- `buf` – A pointer to a buffer containing the data to be written.
- `len` – The length of the data to write.

The return is the number of bytes actually written or -1 if there was an error.

Eclipse Paho - MQTT Embedded C

The Eclipse Paho project is a collection of implementations of MQTT based functions governed by the Eclipse foundation. Included in this suite is an embedded C implementation which is an MQTT client written in as vanilla a C language as is possible.

See also:

- MQTT
- Eclipse paho
- Eclipse Paho – embedded C
- Eclipse Paho

MQTTClientInit

Initialize an MQTTClient.

```
void MQTTClientInit(
   MQTTClient* c,
   Network *network,
   unsigned int command_timeout_ms,
   unsigned char *sendbuf,
   size_t sendbuf_size,
   unsigned char *readbuf,
   size_t readbuf_size)
```

The network is a populated connection to the target broker.

MQTTConnect

Connect to a remote MQTT broker.

```
int MQTTConnect(
   MQTTClient *c,
   MQTTPacket_connectData *options)
```

The `options` structure should always be initialized through assignment from `MQTTPacket_connectData_initializer`. For example:

```
MQTTPacket_connectionData data =  MQTTPacket_connectData_initializer
```

Once populated, we can then over-ride some of the defaults to reflect our desired changes. Within the object we have:

- `MQTTString clientID` – An indenitication used by this client.
- `unsigned short keepAliveInterval` – The keep-alive interval for the connection in seconds. If the client doesn't communicate from its last message within this interval then the broker can consider the client lost. The special value of 0 means that there will be no keep-alive checking.
- `unsigned char cleansession` – Set to 1 to begin a clean session otherwise supply 0.
- unsigned char willFlag
- MQTTPacket_willOptions will
- `MQTTString username` – The user name passed for authentication.
- `MQTTString password` – The password passed for the user for authentication.

The return codes from this call are:

- SUCCESS (0) – connected successfully.
- FAILURE (-1) – General failure.
- BUFFER_OVERFLOW (-2) – Buffer overflow.

Other than these, the returns are MQTT CONNACK responses which are:

- 1 – Unacceptable protocol version.
- 2 – Client identifier rejected.
- 3 – Server unavailable.
- 4 – Bad username or password.
- 5 – Not authorized.

MQTTDisconnect

Disconnect from the broker.

```
int MQTTDisconnect(MQTTClient* c)
```

Followiing a successful previous call to `MQTTConnect()`, this function disconnects us from the broker.

MQTTPublish

Publish a message to the broker.

```
int MQTTPublish(
   MQTTClient *c,
   const char *topicName,
   MQTTMessage *message)
```

The `MQTTMessage` is a structure that reflects the message to be published. It includes:

- `enum QoS qos` – The desired quality of service of the published message.
- `unsigned char retained` – Not used in a publish.
- `unsigned char dup` – Not used in a publish.
- `unsigned short id` – The identity of this message.
- `void *payload` – The payload content of this message.
- `size_t payloadlen` – The length of the message payload.

MQTTRun

```
void MQTTRun(void* parm)
```

MQTTSubscribe

Subscribe to a topic.

```
int MQTTSubscribe(
   MQTTClient *c,
   const char *topicFilter,
   enum QoS qos,
   messageHandler messageHandler)
```

The `messageHandler` is a function that is called when a new message is received as published by the broker. The signature of the function is:

```
void func(MessageData *data)
```

The `MessageData` contains:

- `MQTTString *topicName` – The topic on which the message was published. To examine the `topicName`, we should use the form `topicName->lenstring.len` and `topicName->lenstring.data`.

`MQTTMessage *message` – The content of the message received.

The `MQTTMesage` contains:

- `enum QoS qos` – What is the quality of service.
- `unsigned char retained` – Was this a retained message.
- `unsigned char dup` – Was this message sent more than once.
- `unsigned short id` – The id of the message.

- `void *payload` – The payload data of the message.
- `size_t payloadlen` – The length of the payload data.

MQTTUnsubscribe

Unsubscribe from receiving further publications on the topic.

```
int MQTTUnsubscribe(
   MQTTClient *c,
   const char* topicFilter)
```

MQTTYield

```
int MQTTYield(
   MQTTClient *c,
   int timeout_ms)
```

NetworkConnect

Connect to an MQTT broker

```
int NetworkConnect(Network *network, char *address, int port)
```

If the return is 0, then the connection succeeded.

Arduino ESP32 Libraries

Arduino 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 ESP32.

To use the ESP32 WiFi library you must include its header:

```
#include <WiFi.h>
```

We must also call

```
initWiFi()
```

and

```
startWiFi()
```

before using any of the other WiFi functions in the Arduino library. It is recommended NOT to mix WiFi setup in ESP-IDF with WiFi setup in the Arduino environment.

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.

To set up an access point, we would call `WiFi.softAP()` supplying the SSID and password information.

Here is an example of us connecting as a station:

```
WiFi.mode(WIFI_STA);
WiFi.begin(SSID, PASSWORD);
if (WiFi.waitForConnectResult() != WL_CONNECTED) {
       Serial1.println("Failed");
       return;
}
WiFi.printDiag(Serial1);
// We are now connected as a station
```

See also:

- Arduino WiFiClient
- Arduino WiFiServer
- [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.

Includes:

- WiFiSTA.h

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.

Includes:

- WiFiScan.h
- WiFiSTA.h

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.

Includes:

- WiFiScan.h
- WiFiSTA.h

WiFi.channel

Retrieve the current channel.

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

Retrieve the current channel.

Includes:

- WiFiScan.h
- WiFiGeneric.h

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.

Includes:

- WiFiSTA.h

WiFi.disconnect

Disconnect from an access point.

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

Disconnect from the current access point.

Includes:

- WiFiSTA.h

WiFi.dnsIP

Includes:

- WiFiSTA.h

WiFi.enableAP

Includes:

- WiFiGeneric.h

WiFi.enableSTA

Includes:

- WiFiGeneric.h

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

Includes:

- WiFiScan.h

WiFi.gatewayIP

Get the IP address of the station gateway.

```
IPAddress gatewayIP()
```

Retrieve the IP address of the station gateway.

Includes:

- WiFiSTA.h

WiFi.getAutoConnect

Includes:

- WiFiSTA.h

WiFi.getMode

Includes:

- WiFiGeneric.h

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,
   int32_t &channel)
```

Retrieve all the details of the specified scanned `networkItem`.

Includes:

- WiFiScan.h

See also:

- WiFi.scanComplete
- WiFi.scanDelete
- WiFi.scanNetworks

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.

Includes:

- WiFiGeneric.h

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.isConnected

Includes:

- WiFiSTA.h

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. There is a separate IP address if the ESP is an access point.

Includes:

- WiFiSTA.h

See also:

- WiFi.softAPIP

WiFi.macAddress

Get the station interface MAC address.

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

Get the station interface MAC address.

Includes:

- WiFiSTA.h

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:

Includes:

- WiFiGeneric.h

Wifi.persistent

Includes:

- WiFiGeneric.h

WiFi.printDiag

Log the state of the WiFi connection.

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

Log the state of the WiFi connection. We can pass in either Serial or Serial1 as an argument to log the data to the Serial port. An example of output is as shown next:

```
Mode: STA
PHY mode: N
Channel: 7
AP id: 0
Status: 5
Auto connect: 0
SSID (7): yourSSID
Passphrase (8): yourPassword
BSSID set: 0
```

Note that the status value is the result of a `wifi_station_get_connect_status()` call.

Includes:

- WiFi.h

WiFi.psk

Includes:

- WiFiSTA.h
-

WiFi.RSSI

Retrieve the RSSI (Received Signal Strength Indicator) value of the scanned network item.

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

Retrieve the RSSI value of the scanned network item.

Includes:

- WiFiScan.h
- WiFiSTA.h

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.

Includes:

- WiFiScan.h

See also:

- WiFi.scanNetworks
- WiFi.scanDelete

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.

Includes:

- WiFiScan.h

See also:

- WiFi.scanComplete
- WiFi.scanNetworks
- WiFi.getNetworkInfo

WiFi.scanNetworks

Scan the access points in the environment.

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

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

Includes:

- WiFiScan.h

See also:

- WiFi.scanComplete
- WiFi.scanDelete
- WiFi.getNetworkInfo

WiFi.setAutoConnect

Includes:

- WiFiSTA.h

WiFi.setAutoReconnect

Includes:

- WiFiSTA.h

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)
```

The `ssid` is used to advertize our network. The `passPhrase` is the password a station must supply in order to be authorized to access.

Includes:

- WiFiAP.h

WiFi.softAPConfig

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

Includes:

- WiFiAP.h

WiFi.softAPdisconnect

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

Includes:

- WiFiAP.h

WiFi.softAPmacAddress

Get the MAC address of the access point interface.

```
uint8_t *softAPmacAddress(uint8_t *mac)
String softAPmacAddress()
```

Get the MAC address of the access point interface.

Includes:

- WiFiAP.h

WiFi.softAPIP

Get the IP address of the access point interface.

```
IPAddress softAPIP()
```

Return the IP address of the access point interface. There is a separate IP for the station.

See also:

- WiFi.localIP

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.

Includes:

- WiFiScan.h
- WiFiSTA.h

WiFi.status

Retrieve the current WiFi status.

```
wl_status_t status()
```

The status returned will be one of:

- WL_IDLE_STATUS (0)
- WL_NO_SSID_AVAIL (1)
- WL_SCAN_COMPLETED (2)
- WL_CONNECTED (3)
- WL_CONNECT_FAILED (4)
- WL_CONNECTION_LOST (5)
- WL_DISCONNECTED (6)

Includes:

- WiFiSTA.h

WiFi.stopSmartConfig

void stopSmartConfig()

WiFi.subnetMask

IPAddress subnetMask()

Includes:

- WiFiSTA.h

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 connected or failed. The return code is the status. Specifically, this function watches the status to see when it becomes something other than `WL_DISCONNECTED`. Perhaps a more positive form of this function would be:

```
while (WiFi.status() != WL_CONNECTED) {
       delay(500);
       Serial.print(".");
}
```

Includes:

- WiFiSTA.h

Arduino WiFiClient

This library provides TCP connections to a partner. A separate class provides UDP communications.

To use this library, you must include "`WiFi.h`".

We create an instance of this class and then connect to a partner using the `connect()` method.

WiFiClient

WiFiClient.available

Return the amount of data available to be read.

```
int available()
```

Return the amount of data available to be read.

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

Determine if we are connected to a partner.

```
uint8_t connected()
```

Return true if connected and false otherwise.

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.setOption

```
int setOption(int option, int *value)
```

WiFiClient.status

```
uint8_t status()
```

WiFiClient.stop

Disconnect a client.

```
void stop()
```

Disconnect a client.

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)
size_t write(T& source, size_t unitSize);
```

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

Arduino 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. Interesting, it appears that once we crate a server instance within an ESP8266, there is no way to stop it running.

WiFiServer.available

Retrieve a WiFiClient object that can be used for communications.

```
WiFiClient available(byte* status)
```

Retrieve the corresponding WiFiClient.

See also:

- Arduino WiFiClient

WiFiServer.begin

Start listening for incoming connections.

```
void begin()
```

Start listening for incoming connections. Until this method is called, the ESP8266 doesn't accept incoming connections. Interestingly, once called, there is no obvious way to stop listening. The port used for the incoming connections is the one supplied when the WiFiServer object was constructed.

WiFiServer.getNoDelay

WiFiServer.hasClient

Return true if we have a client connected.

bool hasClient()

WiFiServer.setNoDelay

WiFiServer.status

WiFiServer.write

WARNING!! This method is not implemented.

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

Although present on the interface, this method is not yet implemented.

Arduino IPAddress

A representation of an IPAddress. This class has some operator overrides:

[i] – Get the ith byte of the address. I should be 0-3.

Arduino SPI

The Arduino SPI library is loaded from the "`SPI.h`" header file and provides the SPI interface.

SPI.begin

Begin to use the SPI bus.

```
void begin(int8_t sck=-1, int8_t miso=-1, int8_t mosi=-1, int8_t ss=-1)
```

The `sck` is the pin to use for clock. The `miso` is the pin to use for MISO, the `mosi` is the pin to use for MOSI, the `ss` is the pin to use for SS.

SPI.beginTransaction

Begin a transaction.

```
void beginTransaction(SPISettings settings)
```

The settings is an instance of an SPISettings object. This is constructed with three parameters:

```
SPISettings(uint32_t clock, uint8_t bitOrder, uint8_t dataMode)
```

This sets the clock speed, bit order and data mode.

SPI.end

Stop using the bus.

```
void end()
```

SPI.endTransaction

End a transaction.

```
void endTransaction(void)
```

SPI.setBitOrder

Set the bit order value.

```
void setBitOrder(uint8_t bitOrder)
```

SPI.setClockDivider

Set the clock divider value.

```
void setClockDivider(uint32_t clockDiv)
```

SPI.setDataMode

Set the data mode value.

```
void setDataMode(uint8_t dataMode)
```

SPI.setFrequency

Set the frequency value.

```
void setFrequency(uint32_t freq)
```

SPI.setHwC

```
void setHwCs(bool use)
```

SPI.transfer

```
uint8_t transfer(uint8_t data)
```

SPI.transfer16

```
uint16_t transfer16(uint16_t data)
```

SPI.transfer32

```
uint32_t transfer32(uint32_t data)
```

SPI.transferBytes

```
void transferBytes(uint8_t * data, uint8_t * out, uint32_t size)
```

SPI.transferBits

```
void transferBits(uint32_t data, uint32_t * out, uint8_t bits)
```

SPI.write

```
void write(uint8_t data)
```

SPI.wirite16

```
void write16(uint16_t data)
void write16(uint16_t data, bool msb)
```

SPI.write32

```
void write32(uint32_t data)
void write32(uint32_t data, bool msb)
```

SPI.writeBytes

```
void writeBytes(uint8_t * data, uint32_t size)
```

SPI.writePattern

```
void writePattern(uint8_t * data, uint8_t size, uint32_t repeat)
```

Arduino I2C – Wire

The `Wire` class provides I2C support. In order to use this class, import "`Wire.h`" into your sketch. When we use this class, a global instance called "`Wire`" is made available to us. One wire is called SCL which provides the clock while the other wire is called SDA and is the data bus. On the Arduino, the library supports being either a master or a slave however in the current implementation, only being a master is supported.

To use this class, first we define which pins should be used and then start the service.

```
Wire.begin(SDApin, SCLpin);
```

To send data, we begin a transmission using `beginTransmission()`:

```
Wire.beginTransmission(deviceAddress);
```

now we can write some data …

```
Wire.write(value);
```

and finally complete the transmission:

```
Wire.endTransmission();
```

if we wish to receive data from the slave, we can call `requestFrom()`:

```
Wire.requestFrom(deviceAdress, size, true);
```

and data can be read using the `available()` and `read()` functions.

See also:

- Working with I2C
- Arduino – Wire Library

Wire.available

Determine the number of bytes available to read.

```
int available(void)
```

Determine the number of bytes available to read.

See also:

- Wire.read
- Wire.requestFrom

Wire.begin

Initialize the wire library.

```
void begin(int SDApin, int SCLpin)
void begin()
void begin(uint8_t address)
void begin(int address)
```

Initialize the wire library. When an address is supplied, we are a slave otherwise we are a master. We can also specify the pins to be used for SDA and SCL. If we are a master and no pins are supplied, we will use the default pins.

WARNING!! – It appears that there is **NO** support for actually being a slave and only a master is supported at this time.

WARNING!! – In the current code, the `address` parameter is ignored!!

See also:

- Wire.pins

Wire.beginTransmission

Beging a transmission block to a slave.

```
void beginTransmission(uint8_t address)
void beginTransmission(int address)
```

Begin the notion of sending a transmission to a slave device with the supplied address. Further calls to `write()` will queue data to be transmitted which is finally executed with a call to `endTransmission()`.

Wire.endTransmission

End the bracketing of a transmission.

```
uint8_t endTransmission(void)  // Defaults to sendStop = true
uint8_t endTransmission(uint8_t sendStop)
```

End the bracketing of a transmission and perform the actual transmit. The return codes are:

- `0` – Transmitted correctly
- `2` – Received NACK on transmit of address
- `3` – Received NACK on transmit of data
- `4` – line busy

Wire.flush

Discard any un-read or un-written data.

```
void TwoWire::flush(void)
```

Discard any un-read or un-written data. A call to `available()` will return 0 and a call to `endTransmission()` will transmit no data.

Wire.onReceive

A callback when we are in the role of a slave and receive a transmission from a master.

```
void onReceive( void (*function)(int numBytes))
```

A callback when slave receives a transmission from a master.

WARNING!! – This function is not implemented.

Wire.onReceiveService

Not implemented.

```
void onReceiveService(uint8_t* inBytes, int numBytes)
```

Not implemented.

WARNING!! – This function is not implemented.

Wire.onRequest

A callback invoked when we are in the role of a slave and a master requests data from us.

```
void onRequest(void (*function)(void))
```

A callback invoked when we are in the role of a slave and a master requests data from us.

WARNING!! – This function is not implemented.

Wire.onRequestService

Not implemented.

```
void onRequestService(void)
```

Not implemented.

WARNING!! – This function is not implemented.

Wire.peek

Peek at the next byte.

```
int peek(void)
```

Peek at the next byte if one is available. A return of -1 if there is no byte available.

Wire.pins

WARNING!! - This function has been deprecated in favor of `begin(sda, scl)`.

Define the default pins for SDA and SCL.

```
void pins(int sda, int scl)
```

Define the default pins for SDA and SCL.

See also:

- Wire.begin

Wire.read

Read a single byte.

```
int read(void)
```

Read a single byte from the bus. A value of `-1` is returned if there is no byte available.

See also:

- Wire.available
- Wire.requestFrom

Wire.requestFrom

Request data from a slave.

```
size_t requestFrom(uint8_t address, size_t size, bool sendStop)
uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
uint8_t requestFrom(uint8_t address, uint8_t quantity)
uint8_t requestFrom(int address, int quantity)
uint8_t requestFrom(int address, int quantity, int sendStop)
```

Request data from a slave. This method should be called when we are playing the role of a master. The `address` parameter defines the slave address for the device that should respond. If the `sendStop` is true, a stop message is transmitted releasing the I2C bus. If `sendStop` is false, a restart message is transmitted preventing another bus master from taking control.

The `quantity` parameter states how many bytes we wish to receive.

The return value is the number of bytes that were received.

See also:

- Wire.read
- Wire.available

Wire.setClock

Set the clock frequency.

```
void setClock(uint32_t frequency)
```

Set the clock frequency. Always call `setClock()` AFTER a call to `begin()`.

Wire.write

Write one or more bytes to the slave.

```
size_t write(uint8_t data)
size_t write(const uint8_t *data, size_t quantity)
```

Write one or more bytes to the slave.

Arduino Ticker library

This library sets up callback functions that are called after a period of time. To use this library you must include "`Ticker.h`". For example:

```
#include <Ticker.h>

void timerCB() {
      Serial1.println("Tick ...");
}

void setup()
```

```
{
       Serial1.begin(115200);
       ticker.attach(5, timerCB);
       Serial1.println("Ticker attached");
}
```

Ticker

An instance of a Ticker object. Commonly this is created as a global such as:

```
Ticker myTicker;
```

attach

Attach a callback function to the ticker.

```
void attach(float seconds,
       callback_t callback)
void attach(float seconds,
       void (*callback)(TArg),
       TArg arg)
```

Attach a callback function to the ticker such that the callback is invoked each period of seconds. Note that seconds is a float so we can specify values such as 0.1 to indicate a callback every 1/10th of a second (100 milliseconds).

The callback_t is a defined as:

```
void (*callback_t)(void)
```

attach_ms

Attach a callback function to the ticker.

```
void attach_ms(uint32_t milliseconds,
       callback_t callback)
void attach_ms(uint32_t milliseconds,
       void (*callback)(TArg), TArg arg)
```

Attach a callback function to the ticker such that the callback is invoked each period of milliseconds. Only one attachment can be made to a timer.

detach

Detach a ticker from the timer.

```
void detach()
```

Detach a callback function from the timer. No further callbacks will occur.

once

Attach a callback function to the timer for a one-shot firing.

```
void once(float seconds,
       callback_t callback)
void once(float seconds,
       void (*callback)(TArg),
       TArg arg)
```

Attach a callback function to the timer for a one-shot firing. Note that seconds is a float so we can specify values such as 0.1 to indicate a callback every 1/10th of a second (100 milliseconds).

once_ms

Attach a callback function to the timer for a one-shot firing.

```
void once_ms(uint32_t milliseconds,
       callback_t callback)
void once_ms(uint32_t milliseconds,
       void (*callback)(TArg),
       TArg arg)
```

Attach a callback function to the timer for a one-shot firing.

Arduino EEPROM library

This library allows us to store and retrieve data from storage that persists across a device restart. A singleton object called EEPROM is pre-supplied for use.

EEPROM.begin

Begin the process of writing or reading from EEPROM. The size is the amount of storage we wish to work with.

```
void begin(size_t size)
```

EEPROM.commit

The changes to the data are committed to EEPROM. A return of `true` indicates success while a return of false indicates a failure.

```
bool commit()
```

EEPROM.end

Commits the changes to the data and then releases any local storage. No further reads or writes should be attempted until after the next `begin()` call.

```
void end()
```

EEPROM.get

Read a data structure from storage.

```
T &get(int address, T &t)
```

EEPROM.getDataPtr

Retrieve a pointer to the storage we are going to read or write.

```
uint8_t *getDataPtr()
```

EEPROM.put

Put a data structure to storage.

```
const T &put(int address, const T &t)
```

EEPROM.read

Read a byte from storage.

```
uint8_t read(int address)
```

EEPROM.write

Write a byte to storage.

```
void write(int address, uint8_t value)
```

Arduino SPIFFS

FS is the File System library which provides the ability to read and write files from within the Arduino ESP environment. But wait … read and write files to where? There are no "drives" on an ESP8266. The data for the files is read and written to an area of flash memory and since flash is relatively small in size (4MBytes or so max) then that is an upper bound of maximum size of the cumulative files … however, this is still more than enough for many usage patterns such as saving state, logs or configuration information.

SPIFFS.begin

Begin working with the SPIFFS file system.

```
bool begin()
```

Returns true of success and false otherwise.

SPIFFS.open

Open the named file.

```
File open(const char *path, const char *mode)
File open(const String &path, const char *mode)
```

The mode defines how we wish to access the file. The options are:

- `r` – Read the file. The file must exist.
- `w` – Write to the file. Truncate the file if it exists.
- `a` – Append to the file.
- `r+` – Read and write the file.
- `w+` – Read and write the file.
- `a+` – Read and write the file.

See also:

- File.close

SPIFFS.openDir

Open a directory.

```
Dir openDir(const char *path)
Dir openDir(const String &path)
```

SPIFFS.remove

Remove/delete a file from the file system.

```
bool remove(const char *path)
bool remove(const String &path)
```

SPIFFS.rename

Rename a file.

```
bool rename(const char *pathFrom, const char *pathTo)
bool rename(const String &pathFrom, const String &pathTo)
```

File.available

Return the number of bytes that are available within the file from the current file position to its maximum size.

```
int available()
```

File.close

Close a previously opened file.

```
void close()
```

No further reading nor writing should be attempted to be performed.

File.flush

Flush the file.

```
void flush()
```

File.name

Retrieve the name of the file.

```
const char *name()
```

File.peek

Peek at the next byte of data in the file without consuming it.

```
int peek()
```

File.position

Retrieve the current file pointer position.

```
size_t position()
```

File.read

Read data from the file.

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

Read either a single byte of data or a buffer of data from the file.

File.seek

Change the current file pointer position.

```
bool seek(uint32_t pos, SeekMode mode)
```

The mode can be one of:

- `SeekSet` – Change the file pointer position to the absolute value.

- `SeekCur` – Change the file pointer position to be relative to the current position.
- `SeekEnd` – Change the file pointer position to be relative to the end of the file.

File.size

Retrieve the maximum size of the file.

```
size_t size()
```

File.write

Write data to the file.

```
size_t write(uint8_t c)
size_t write(uint8_t *buf, size_t size)
```

Write either a single byte or a buffer of bytes into the file at the current file pointer position.

Dir.fileName

Retrieve the name of the file.

String fileName()

Dir.next

bool next()

Dir.open

```
File open(const char *mode)
File open(String &path, const char *mode)
```

Dir.openDir

```
Dir openDir(const char *path)
Dir openDir(String &path)
```

Dir.remove

Dir.rename

Arduino ESP library

A class has been provided called ESP that provides ESP8266 environment specific functions. You must realize that using these functions will result in your applications not being portable to the Arduino (if that is a desire).

ESP.eraseConfig

```
bool eraseConfig()
```

ESP.getChipId

```
uint32_t getChipId()
```

ESP.getCpuFreqMHz

```
uint8_t getCpuFreqMHz()
```

ESP.getCycleCount

```
uint32_t getCycleCount()
```

ESP.getFlashChipMode

```
FlashMode_t getFlashChipMode()
```

ESP.getFlashChipSize

```
uint32_t getFlashChipSize()
```

ESP.getFlashChipSpeed

```
uint32_t getFlashChipSpeed()
```

ESP.getFreeHeap

```
uint32_t getFreeHeap()
```

ESP.getOption

```
int getOption(int option, int *value)
```

ESP.getSdkVersion

Retrieve the string representation of the SDK being used.

```
const char *getSdkVersion()
```

ESP.flashEraseSector

```
bool flashEraseSector(uint32_t sector)
```

ESP.flashRead

```
bool flashRead(uint32_t offset, uint32_t *data, size_t size)
```

ESP.flashWrite

```
bool flashWrite(uint32_t offset, uint32_t *data, size_t size)
```

ESP.magicFlashChipSize

```
uint32_t magicFlashChipSize(uint8_t byte)
```

ESP.magicFlashChipSpeed

```
uint32_t magicFlashChipSpeed(uint8_t byte)
```

ESP.restart

```
void restart()
```

Arduino String library

Although it is believed that this library may be identical to the Arduino String library, I believe it is so essential to understand that I am going to list the methods again.

String

Constructor

```
String(const char *cstr = "");
String(const String &str)
String(char c)
String(unsigned char, unsigned char base = 10)
String(int, unsigned char base = 10)
String(long, unsigned char base = 10)
String(unsigned long, unsigned char base = 10)
String(float, unsigned char decimalPlaces = 2)
String(double, unsigned char decimalPlaces = 2)
```

Create an instance of the String class seeded with various data type initializers.

String.c_str

Retrieve a C string representation.

```
const char *c_str()
```

String.reserve

unsigned char reserve(unsigned int size)

String.length

Return the length of the string.

```
unsigned int length()
```

Return the length of the string.

String.concat

```
unsigned char concat(const String &str)
unsigned char concat(const char *cstr)
unsigned char concat(char c)
unsigned char concat(unsigned char c)
unsigned char concat(int num)
unsigned char concat(unsigned int num)
unsigned char concat(long num)
unsigned char concat(unsigned long num)
unsigned char concat(float num)
unsigned char concat(double num)
```

String.equalsIgnoreCase

```
unsigned char equalsIgnoreCase(const String &s) const;
```

String.startsWith

Determine whether or not this string starts with another string.

```
unsigned char startsWith(const String &prefix)
unsigned char startsWith(const String &prefix, unsigned int offset)
```

String.endsWith

```
unsigned char endsWith(const String &suffix)
```

String.charAt

```
char charAt(unsigned int index)
```

String.setCharAt

```
void setCharAt(unsigned int index, char c)
```

String.getBytes

```
void getBytes(unsigned char *buf, unsigned int bufsize, unsigned int index = 0)
```

String toCharArray

```
void toCharArray(char *buf, unsigned int bufsize, unsigned int index = 0)
```

String.indexOf

Find the position of a string or character within the current string.

```
int indexOf(char ch)
int indexOf(char ch, unsigned int fromIndex)
int indexOf(const String &str)
int indexOf(const String &str, unsigned int fromIndex)
```

Find the position of a string or character within the current string. If the match is not found, `-1` is returned otherwise the position of the start of the match is returned.

String.lastIndexOf

```
int lastIndexOf(char ch)
int lastIndexOf(char ch, unsigned int fromIndex)
int lastIndexOf(const String &str)
int lastIndexOf(const String &str, unsigned int fromIndex)
```

String.substring

Retrieve a substring from within the current string.

```
String substring(unsigned int beginIndex)
String substring(unsigned int beginIndex, unsigned int endIndex)
```

Retrieve a substring from within the current string.

String.replace

```
void replace(char find, char replace)
void replace(const String& find, const String& replace)
```

String.remove

```
void remove(unsigned int index)
void remove(unsigned int index, unsigned int count)
```

String.toLowerCase

```
void toLowerCase(void)
```

String.toUpperCase

```
void toUpperCase(void)
```

String.trim

```
void trim(void)
```

String.toInt

```
long toInt(void)
```

String.toFloat

```
float toFloat(void)
```

Reference materials

There is a wealth of information available on the ESP32 from a variety of sources.

Read-The-Docs

Some of the best knowledge on the ESP32 comes from the Github source of the ESP-IDF. The project contains markup documentation that is hand written and also the source files are mechanically scanned. The result is some superb documentation that can be found here:

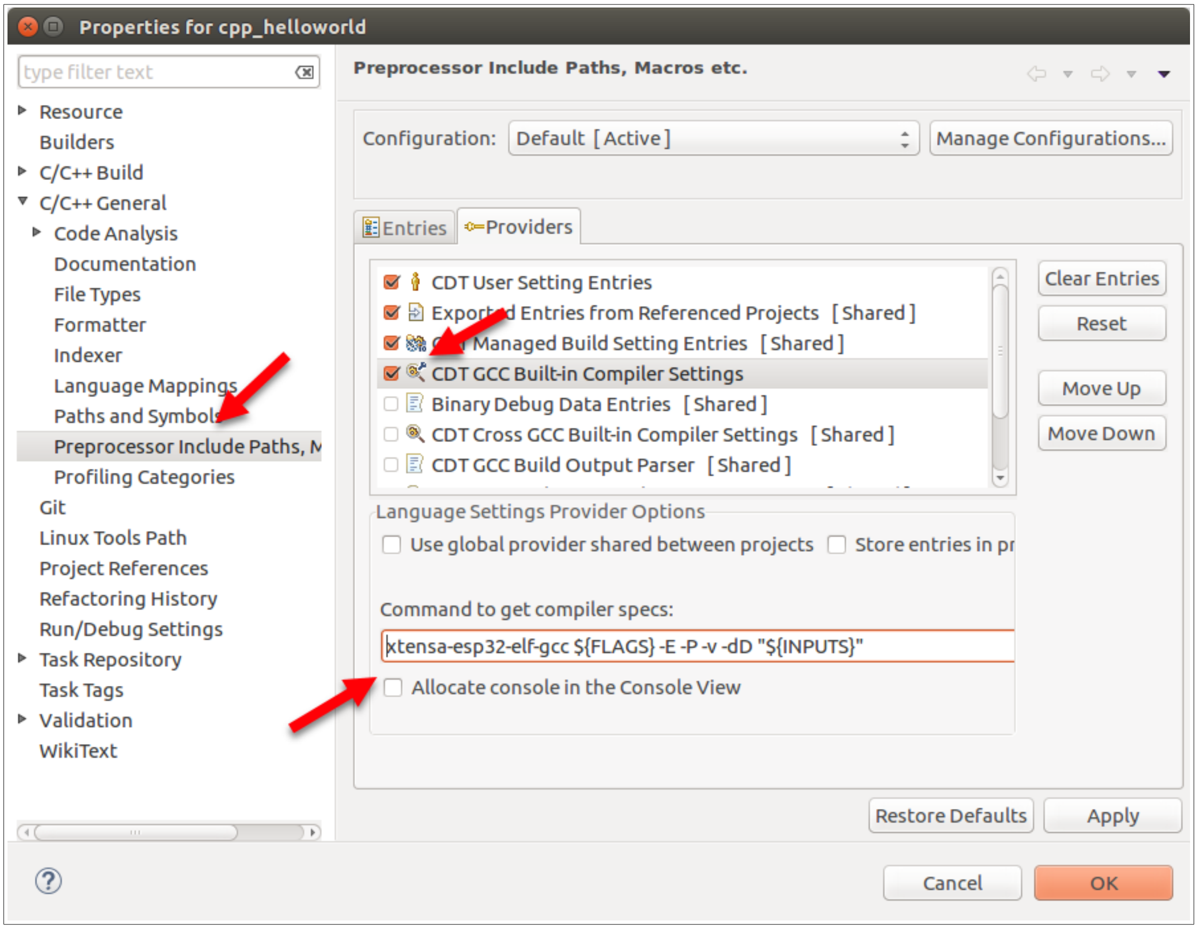
http://esp-idf.readthedocs.io/en/latest/

C++ Programming

Eclipse configuration

To support the correct indexing of the standard templates, the following incantation is required.

1. Open up the project properties
2. Visit C/C++ General > Preprocessor Include Paths, Macros, etc
3. Select the Providers tab
4. Check the box for "`CDT GCC Built-in Compiler Settings`"
5. Set the compiler spec command to
   `xtensa-esp32-elf-gcc ${FLAGS} -std=gnu++11 -E -P -v -dD "${INPUTS}"`
6. Rebuild the index



Configuration: Default [Active]

☑ CDT User Setting Entries
☑ Exported Entries from Referenced Projects [Shared]
☑ CDT Managed Build Setting Entries [Shared]
☑ CDT GCC Built-in Compiler Settings
☐ Binary Debug Data Entries [Shared]
☐ CDT Cross GCC Built-in Compiler Settings [Shared]
☐ CDT GCC Build Output Parser [Shared]

Language Settings Provider Options

☐ Use global provider shared between projects ☐ Store entries in pr

Command to get compiler specs:

`xtensa-esp32-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"`

☐ Allocate console in the Console View

Simple class definition

Classes are defined using the "class" keyword. Within a class we can define public fields and member functions as well as private fields and member functions. Public items may be accessed directly from outside the class if we have an instance of it. Private items may be accessed only from within the class itself.

A constructor method is a public method having no return type and the same name as the class itself. Should we need to perform some specialized cleanup, we can declare a destructor method that is invoked when the class instance is deleted. The name of the method is the same as the class but prefixed with the "~" character and has no return type.

Within a class instance methods, we can use a variable called this which will be an implicit pointer to the instance of the current class.

If we declare a field within a class definition as being static, then just one copy of the variable is shared across all instances of the class. A method within a class can be defined as static and is thus allowed to be executed without reference to a class instance.

One class can inherit from another using:

```
class newClass: public existingClass {
   ...
}
```

Sample class header

```
#ifndef MyClass_h
#define MyClass_h

class MyClass {
public:
       MyClass();
       static void myStaticFunc();
       void myFunc();
};
#endif
```

Sample class source

```
#include <MyClass.h>
MyClass::MyClass() {
   // Constructor code here ...
}
String MyClass::myStaticFunc() {
   // Code here ...
}
void MyClass::myFunc() {
   // Code here ...
}
```

Mixing C and C++

The vast majority of the ESP-IDF platform is written in C. Should you wish to write C++ applications and leverage the ESP-IDF C functions or, conversely, have ESP-IDF application logic call your C++ source files there is an important consideration we have to include.

In C, if I define a function, `foo()` for example, then the function is called `foo()` and there is no ambiguity. Now consider a class called `A` which has a function called `foo()` and a second class called `B` which also has a function called `foo()`. It seems pretty obvious that I can't simply call `foo()` as the environment would not know which one was meant. That is why we prefix a function call with the class name in which it was defined. Now let us consider functions declared outside of a class. For example:

```
void bar() {
   // some code
}
```

If I coded this in a C++ program, I could then call `bar()` and all would be good. However if then wrote a C source file and tried to call bar() we might be surprised to find that the linker would complain that we couldn't find `bar()`. Why is this?

The answer is that when we declare a global function in a C++ source file, the C++ environment implicitly flags the function as being part of a C++ class that represents some global class and, as such, its name is not actually `bar()` but is instead `<someGlobalName>::bar()`.

Fortunately, should we wish, we can explicitly declare that a function definition should be defined without a prefix class name by using the syntax:

```
extern "C" {
   void bar();
}
```

This results in the exported name of `bar()` without any further decoration.

Including stdc++ in your app

The Espressif supplied C++ compiler includes the stdc++ library. If we wish to use this, we must include it in our linking. The easiest way to do this is to add

```
COMPONENT_ADD_LDFLAGS=-lstdc++ -l$(COMPONENT_NAME)
```

To `component.mk` in the main directory.

C++ Specialized Data types

String

The string data type is included through:

```
#include <string>
```

A string can be declared using:

```
std::string myString = "Hello World";
```

Since string is a class, we have objects available on it:

- `length` / `size` – The length/size of the string in characters.
- `find` – Find a sub-string with the string.
- `empty` – True if the string is empty.
- `clear` – Empty the string.
- `+` – Concatenate to the string.
- `c_str` – Return a null terminated string representation (a C language string).

The string type also has some powerful stream processors associated with it.

There is a class called `std::stringstream` which allows us to stream data into it and then retrieve the string representation. For example:

```
#include <sstream>
std::stringstream stream;
// do stream things
std::string myString = stream.str();
```

We can append into the stream with:

```
stream << "hello " << "world: " << 123;
```

We can also do numeric formatting such as setting the width, fill character and hex mode:

```
#include <iomanip>
stream << std::hex << std::setfill('0') << std::setw(2) << 99;
```

List

The list data type is included through:

```
#include <list>
```

```
std::list<type> myList
```

Map

This data type stores name/value pairs.

The map data type is included through:

```
#include <map>
```

We define a map variable using:

```
std::map<keyType, valueType> mapName;
```

To insert a value into the map we use:

```
mapName.insert(std::pair<keyType, valueType>(key, value));
```

To see if a value is within the map, we use:

```
valueType value = mapName.at(key);
```

This will throw a `std::out_of_range` exception if the key is not found in the map.

To iterate through a map we use:

```
for (auto &myPair : mapName) {
   // myPair.second is the value
}
```

We can also use a more conventional iterator:

```
for (std::map<keyType, valueType>::iterator it = mapName.begin();
   it != mapName.end();
   it++) {
   // it->first is the key
   // it->second is the value
}
```

See also:

- std::map Tutorial Part 1: Usage Detail with examples

Queue

The queue data type is included through:

```
#include <queue>
```

Stack

The stack data type is include through:

```
#include <stack>
```

Vector

The vector data type is included through:

```
#include <vector>
```

and is part of the "std::" namespace.

```
std::vector<type> myVector;
```

We can access the member's of a vector through the "[x]" array index construct.

We can append to the end of a vector using "push_back()" and pull from the end using "pop_back()". The size() method returns the number of entries in the vector. The clear() method removes all entities in the vector. The empty() method returns true if the vector is empty.

Lambda functions

Modern C++ has introduced lambda functions. These are C++ language functions that don't have to be pre-declared but can instead be declared "inline". The functions have no names associated with them but otherwise behave just like other functions.

See also:

- Lambda functions

Designated initializers not available in C++

In the latest versions of the C programming language, we can perform interesting structure initializations such as:

```
struct myStruct myVar = {
   .a = 123;
   .b = "xyz";
};
```

This will create an instance of a variable called "myVar" of type "struct myStruct" and initialize its members.

This capability is not present in C++.

Ignoring warnings

From time to time, your code may issue compilation warnings that you wish to suppress. One way to achieve this is through the use of the C compile #pragma directive.

For example:

```
#pragma GCC diagnostic ignored "-Wformat"
```

See also:

- GCC Diagnostic Pragmas

File I/O in C++

Leveraging STL, we have some nice file I/O capabilities.

```
ofstream myfile;
myfile.open("<name>");
myfile << <data>;
myfile.close();
```

The open method has the signature open(filename, mode). The mode can control how the file is opened and is the bitwise or of flags such as:

- ios::in – Open for input.
- ios::out – Open for output.
- ios::binary – Open for binary.
- ios::ate – Set the file pointer to the end of the file.
- ios::app – Append to the end of the file.
- ios::trunc – Truncate the file.

Headers:

- ofstream – Output to files.
- ifstream – Input from files.
- fstream – Input and output from files.
- iostream

See also:

- Input/output with files

The Factory pattern

In an ESP32 environment there are lots of times where "callbacks" of one form or another are desired. If we were in a C environment, I might do the following:

```
void myCallbackFunc() {
   … my code;
}

registerCallback(myCallbackFunc);
```

While this will still work in a C++ environment, I really wanted to take advantage of the nature of C++ and write a C++ class that implements my logic. For example:

```
MyClass : public CallbackInterface {
   void callbackFunc() {
   }
}
```

and thus, when the event fires, we would invoke myCallbackClass() method. And this of course would work if we did something like:

```
MyClass *pMyClassInstance = new MyClass();
registerCallback(pMyClassInstance);
```

However, if we look closely, what we see is that we are providing ONE instance of the class. Ideally I want an instance of the class to be created for each event.

We can do this using a factory pattern.

Now imagine we have:

```
class CallbackInterface {
   virtual void callbackFunct() = 0;
}
class CallbackFactory {
   virtual CallbackInterface* newInstance() = 0;
}
MyCallbackClass : public CallbackInterface {
   void callbackFun() {
   }
}
MyCallbackClass Factory : public CallbackFactory {
   MyClassFactory *newInstance() {
      return new MyCallbackClass();
   }
}
```

Logging pre-defined symbols

The C and C++ compilers have pre-defined symbols within them. We can log those with:

```
$ xtensa-esp32-elf-gcc -dM -E - < /dev/null
```

The ESP-IDF C++ class libraries

Given the richness of function available in ESP-IDF as well as the power of encapsulation in C++, a set of sample classes have been created that encapsulate much of the function of the framework. These classes are open source and can be found here:

https://github.com/nkolban/esp32-snippets/tree/master/cpp_utils

Documentation for the classes is written in-line in the source using Doxygen.

Here we will provide additional thoughts and notes.

GPIO interactions

GPIO functions have been encapsulated in the class called `ESP32CPP::GPIO`. Note the additional `ESP32CPP` name space. There are name space collisions without this as ESP32 itself provides a struct called GPIO.

The pin numbers are specified as the ESP-IDF native `gpio_num_t` so that we may use constants.

Each GPIO pin has a direction; either input for reading or output for writing. We must declare which direction we are using through a call to either `setInput()` or `setOutput()`. If we have set the pin as output, we can then write a value to it. We have a few choices:

- `low(pin)` – Set the output value of the pin to be low (0).
- `high(pin)` – Set the output value of the pin to be high (1).
- `write(pin, value)` – Set the output value of the pin to be the value supplied.

For reading, we call `read(pin)`.

Example – Writing a value

```
ESP32CPP::GPIO::setOutput(GPIO_NUM_27);
ESP32CPP::GPIO:high(GPIO_NUM_27);
```

Example – Reading a value

```
ESP32CPP::GPIO::setInput(GPIO_NUM_27);
bool value = ESP32CPP::GPIO:read(GPIO_NUM_27);
```

Task management

Since the ESP32 is multi-tasking through FreeRTOS, a class has been created that provides a model of such. The class is called `Task`. It is designed to be sub-classed with your own implementation. You are responsible for implementing a function called `run` which has the following signature:

```
void run(void* data)
```

For example, to create your own task, you might code:

```
#include <Task.h>
class MyTask: public Task {
   void run(void *data) {
      // do something
   }
}
```

With your class created, you can now create an instance of the task and cause it to start executing:

```
MyTask* pMyTask = new MyTask();
pMyTask->start();
```

The default stack size for the new task is 2K but you can set this to a different value before starting it. The method `setStackSize()` can be used for that purpose.

SPI Interaction

The C++ wrapper for SPI is found in the class called SPI. We construct an instance of this as follows:

```
SPI mySPI();
```

Next we call a function to initialize it:

```
void init(mosiPin, misoPin, clkPin, csPin);
```

This defines the pins to be used for the SPI functions. These are optional and, if not supplied, the default values will be used which are:

Function	Pin
mosiPin	13
misoPin	12
clkPin	14
csPin	15

Finally, there is a function to transfer data. Remember that for SPI, each time write a byte we simultaneously receive a byte. This means that the input buffer of data can be used to hold the resulting output buffer of data. The signature is:

```
void transfer(uint8_t* data, size_t dataLen);
```

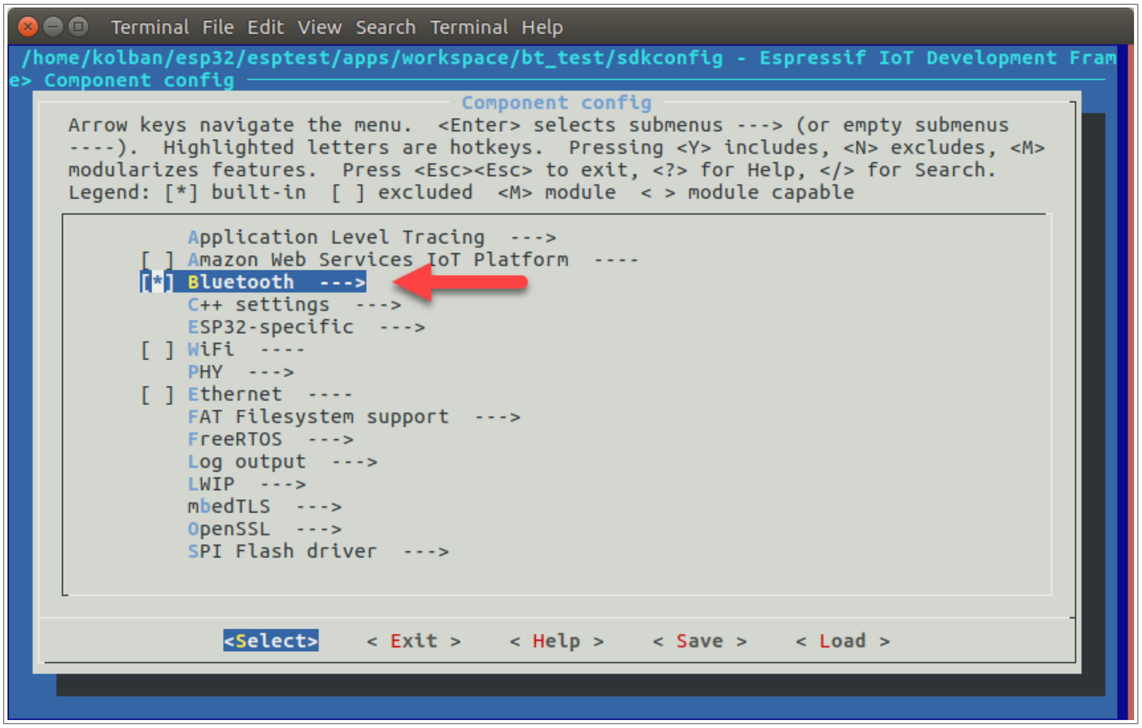
A convenience function is made available for sending and receiving a single byte.

```
uint8_t transferByte(uint8_t value);
```

Bluetooth BLE

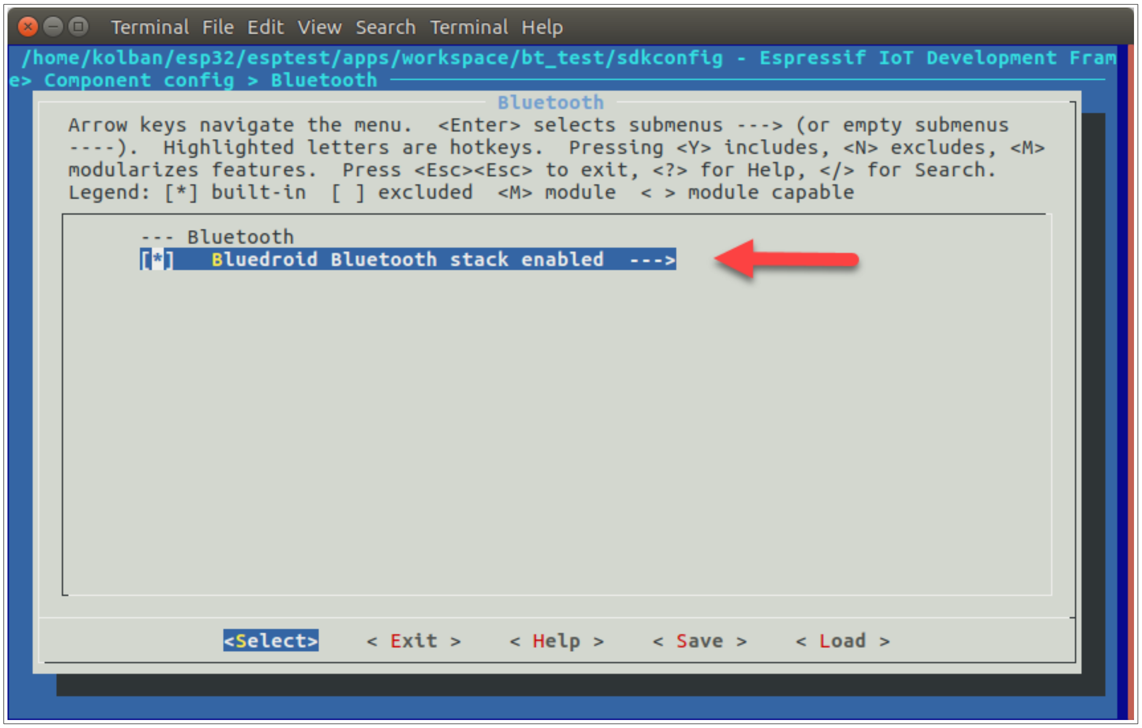
Using the low level ESP-IDF functions to build a BLE solution is considered difficult. There are a lot of concepts and there is a lot of state to be considered plus events needing to be handled. This feels like an ideal candidate for a higher level class encapsulation to simplify our tasks. To that end, we have created a set of C++ classes that provide all the needed BLE capabilities with, what we hope to be, an easier model of usage.

When using these classes, we must use make menuconfig to configure our ESP-IDF environment. First, we need to ensure that Bluetooth is enabled. Start make menuconfig and navigate to Component config and select Bluetooth:



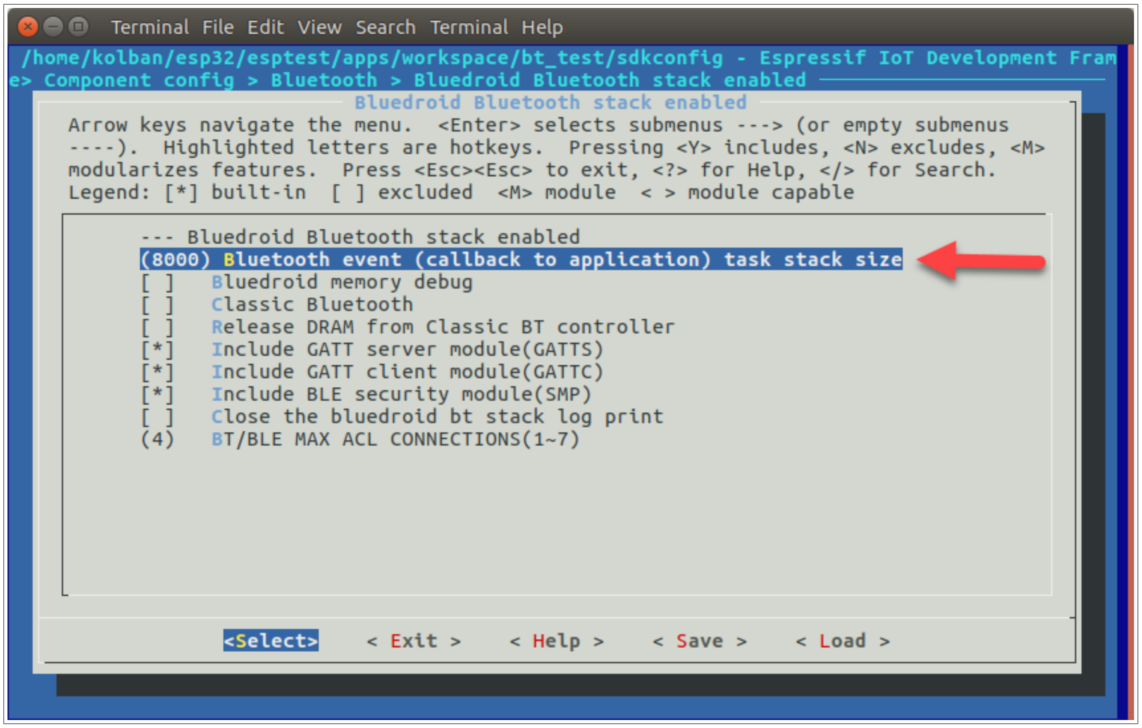
```
/home/kolban/esp32/esptest/apps/workspace/bt_test/sdkconfig - Espressif IoT Development Fram
e> Component config ─────────────────────────────────────────────────────────────
                              Component config
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
  modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

            Application Level Tracing  --->
        [ ] Amazon Web Services IoT Platform  ----
        [*] Bluetooth  --->
            C++ settings  --->
            ESP32-specific  --->
        [ ] WiFi  ----
            PHY  --->
        [ ] Ethernet  ----
            FAT Filesystem support  --->
            FreeRTOS  --->
            Log output  --->
            LWIP  --->
            mbedTLS  --->
            OpenSSL  --->
            SPI Flash driver  --->

              <Select>    < Exit >    < Help >    < Save >    < Load >
```

Next drill into the Bluetooth entry and ensure "Bluedroid Bluetooth stack enable" is selected:



```
/home/kolban/esp32/esptest/apps/workspace/bt_test/sdkconfig - Espressif IoT Development Fram
e> Component config > Bluetooth ─────────────────────────────────────────────────
                                 Bluetooth
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
  modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

        --- Bluetooth
        [*]   Bluedroid Bluetooth stack enabled  --->

              <Select>    < Exit >    < Help >    < Save >    < Load >
```

Finally, drill into the "Bluedroid Bluetooth stack enabled" and set the "Bluetooth event (callback to application) task stack size" to be 8000 (or more).



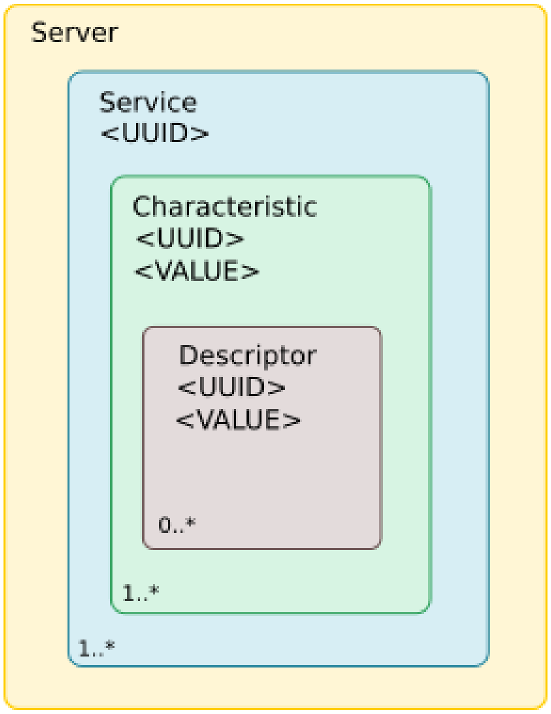
```
/home/kolban/esp32/esptest/apps/workspace/bt_test/sdkconfig - Espressif IoT Development Fram
e> Component config > Bluetooth > Bluedroid Bluetooth stack enabled ──────────
                          Bluedroid Bluetooth stack enabled
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
  modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

        --- Bluedroid Bluetooth stack enabled
        (8000) Bluetooth event (callback to application) task stack size
        [ ]    Bluedroid memory debug
        [ ]    Classic Bluetooth
        [ ]    Release DRAM from Classic BT controller
        [*]    Include GATT server module(GATTS)
        [*]    Include GATT client module(GATTC)
        [*]    Include BLE security module(SMP)
        [ ]    Close the bluedroid bt stack log print
        (4)    BT/BLE MAX ACL CONNECTIONS(1~7)

               <Select>    < Exit >    < Help >    < Save >    < Load >
```

Save all your changes are rebuilt the source.

We'll split our story into two parts … one being a BLE Server and the other being a BLE Client.

A BLE Server

A BLE Server is also known as a BLE Peripheral. It is likely that this is how you will most commonly use the ESP32. With a BLE Server, you will expose one or more services where each service has one or more characteristics and each characteristic may have zero or more descriptors. I think we'll agree that is quite a lot going on.



In our C++ model, we create the concept of classes that represent these items.

- `BLEServer` – Models a server.
- `BLEService` – Models a service. Owned by a `BLEServer`.
- `BLECharacteristic` – Models a characteristic. Owned by a `BLEService`.
- `BLEDescriptor` – Models a descriptor. Owned by a `BLECharacteristic`.

And also:

- `BLEAdvertising` – Models advertising. Owned by a `BLEServer` to let others know of our existence.

At a high level, the pseudo code of a minimal BLE server becomes:

```
// Initialize the BLE environment
BLE::initServer("ServerName");

// Create the server
BLEServer*  pServer  = new BLEServer();

// Create the service
BLEService* pService = pServer->createService(ServiceUUID);
```

```
// Create the characteristic
BLECharacteristic* pCharacteristic =
   pService->createCharacteristic(CharacteristicUUID, properties);
// Set the characteristic value
pCharacteristic->setValue("Hello world");

// Start the service
pService->start();
```

Hopefully you see how this fits together … we create the server, we create the service, create a characteristic upon the service, set a value for the characteristic and then ask the service to start responding to requests.

If you are familiar with the BLE APIs, you may notice what is missing … all the complexity and glue code necessary for event handling and processing of BLE requests. All of this is *handled for you* by the implementation of the classes. This means that you can focus on your intent/usage of BLE while keeping your *plumbing* to a minimum.

The goal of these classes is to efficiently process BLE workflow while encapsulating the plumbing so you don't need to worry about it … but within is the danger that the implementation will restrict you from some tasks by hiding functions that you might otherwise have needed for your own project. Thankfully, this has not been seen to be the case. The classes expose simple high level APIs for the 95% of common practices while at the same time providing methods that can be called to tweak and tailor the operations for the rarer cases. Hopefully you won't need those often but, if and when you do, they are there for you.

When a BLE Server is running, what must happen next is that peer devices (clients) must be able to locate it. This is made possible through the notion of advertising. The BLE Server can broadcast its existence along with sufficient information to allow a client to know what services it can provide.

Once we have started the BLE Server, we can ask it for an object (`BLEAdvertising`) that owns the advertisements that the server produces:

```
BLEAdvertising* pAdvertising = pServer->getAdvertising();
pAdvertising->start();
```

Once performed, the server can start to be dynamically found by the clients. Of course a server doesn't *need* to advertise. If a client should otherwise be informed (or remember) the address of the BLE server, it can request a connection at any time.

A core concept of BLE is the notion of the characteristic. Think of this as a stateful *record* that has an identity and a value. A peer device (if permitted) can read the value of the characteristic or set a new value. Remember, it the BLE characteristic that owns the existence of the value so that it may be served up or changed upon request. This is the core notion of BLE. The value read from the characteristic provides information. For example, if the characteristic represents your heart-rate measured from a sensor, then a remote client can retrieve your current heart rate by reading the current characteristic value. In this case, a remote client will not be setting the value, instead the value will be changed internally by the server either each time a read request by a client is made or when a new sensor reading is taken. Alternatively, the characteristic maintained in the BLE Server may represent the state of something that can be activated or changed. As another example, imagine that the characteristic represents the state of your car door's lock. You can read the characteristic to determine that your door is locked when you leave or, conversely, you can set the characteristic's value to an unlocked state which would result in the server mechanically unlocking the door.

To model these notions in our C++ classes, we leverage the notion that a C++ class can be sub-classed to be more specialized. A class called `BLECharacteristicCallbacks` provides two methods that can be over-riden:

- `onRead(BLECharacteristic* pCharacteristic)` – Called when a read request arrives from a client. A new value for the characteristic can be set before return from the function and will be used as the value received by the client.
- `onWrite(BLECharacteristic* pCharacteristic)` – Called when a client initiated write request arrives. The new value has been set in the characteristic already.

To utilize, we can then override these functions in our own C++ class that subclasses the one provided:

```
class MyCallback: public BLECharateristicCallback {
   void onRead(BLECharacteristic*) {
      // Do something before the read completes.
   }

   void onWrite(BLECharacteristic*) {
      // Do something because a new value was written.
   }
}
```

To use this technique, we inform our `BLECharacteristic` about a callback handler. For example:

```
pCharacteristic->setCallbacks(new MyCallback());
```

Here is an example that sends the time since startup each time a client requests the value:

```
class MyCallbackHandler: public BLECharacteristicCallbacks {
   void onRead(BLECharacteristic* pCharacteristic) {
      struct timeval tv;
      gettimeofday(&tv, nullptr);
      std::ostringstream os;
      os << "Time: " << tv.tv_sec;
      pCharacteristic->setValue(os.str());
   }
};
```

Similar to the characteristic callbacks, we also have server callbacks. These inform about client connection and disconnection events. These are subclassed from the `BLEServerCallbacks` class which has virtual methods for:

- `onConnect(BLEServer* pServer)` – Called when a connection occurs.
- `onDisconnect(BLEServer* pServer)` – Called when a disconnection occurs.

These could be used to enable or disable sensor readings. For example, if we have no clients connected then there is no need to spend energy sampling a value if there is no-one there to read it. However, when a client connects, we can detect that and start reading from the sensor at that point until a subsequent disconnection indication is detected.

Another consideration for our BLE Server is the idea that we might want to "push" data to the peer when something interesting happens. Up until now we have considered the idea that the server can receive read requests to get the current value or can receive write requests to set the current value. There is one more operation that we are interested in that is invoked on the server side by the server and asynchronously to the client. That operation is called "indicate". It is used to signal (or indicate) to the client that the characteristic's value has changed. The client will receive an indication event to let it know that the change has occurred.

On the server, we can cause an indication to occur by invoking:

```
pCharacteristic->indicate();
```

The current value of the characteristic will be the value transmitted to the peer. We might pair this call with a previous request to set a new value:

```
pCharacteristic->setValue("HighTemp");
pCharacteristic->indicate();
```

A similar function to `indicate()` is called `notify()`. The distinction between them is that an `indicate()` receives a confirmation while a `notify()` does not receive a confirmation.

Associated with the idea of indications/notifications, is the architected BLE Descriptor called "Client Characteristic Configuration" which has UUID 0x2902. This contains two distinct bit fields that can be on or off. One bit field governs Notifications while the other governs Indications. If the corresponding bit is on, then the server can/may send the corresponding push. For example if the Notifications bit is on, then the server can/may send notifications. The primary purpose of the descriptor is to allow a partner to request that the server *actually* send notifications or indications. Here is an example. Imagine that we have a BLE Server that can publish notifications when data changes. Now imagine that a BLE Client connects but is actually not interested in receiving these. If the BLE Server executes a notification push and sends the data, that will be wasted effort/energy as the client doesn't want or can't use the information. What we really want is that the client should inform the server that *if* it wants to push new data then it either may or shouldn't. And that is where this descriptor comes into play. If the descriptor is present on a characteristic then then client can remotely change the bit flag to enable or disable notifications and indications. Since the descriptor flag is stored local to the server, the server is at liberty to examine the flag before performing a radio transmission. What this means is that the client/peer can toggle on or off its desire to receive notifications and the server should honor those requests.

The BLE specification constrains the maximum amount of data that can be sent through a notification or indication to be 20 bytes or less. Take this into account in your designs. If the value of a characteristic is greater than this amount, then only the first 20 bytes of the data will be transmitted.

It is likely that your own BLE server application is going to expose its own set of characteristics. Through a characteristic you can set and get the value as a binary piece of storage, however this may be too low level for you. An alternative is to utilize the features of the C++ language and model your own specialized characteristic as a sub class of `BLECharacteristic`. This could then encapsulate the lower level value's getter and setter with your own customized version.

For example, if your characteristic represented a temperature, you could create:

```
class MyTemperatureCharacteristic: public BLECharacteristic {
   MyTemperature(): BleCharacteristic(BLEUUID(MYUUID)) {
      setTemperature(0.0);
   }

   void setTemperature(double temp) {
      setValue(&temp, sizeof temp);
   }

   double getTemperature() {
      return *(double*)getValue();
   }
}
```

A BLE Client

Now we will turn our attention to the second half of our story, namely that of being a client. In this story, our ESP32 doesn't host services but instead wishes to be a consumer of services hosted elsewhere. The story can further be broken down into two sub-stories … namely scanning and interaction.

If we assume that our ESP32 starts up and wishes to be a client of a remote BLE server then it has to connect to that server. In order to connect to the server, we need to know the address of the target server. An address is a 6 byte value commonly written in the form

```
nn:nn:nn:nn:nn:nn
```

While in principle this can be hard-coded into your application or manually entered through some interaction story, this is not the common practice. Instead, we perform a procedure known as a *scan*. Scanning is the idea that we actively listen on the BLE radio frequencies for servers which are actively advertising their existence. As an analogy, imagine we enter a dark room and we have no idea who is in there with us. We take off our ear-muffs (yes, for some reason we entered a dark room with no light at all wearing ear-muffs … but then, this is just an analogy) and we start hearing voices. We hear "Hi this is Bob, I can make toast" and "Hi this is Susan, I can tell you what temperature it is" and we also hear "Hi this is Brian, ask me what I can do". In this analogy, Bob, Susan and Brian are the addresses of the BLE devices. Each of them are continually saying out loud that they exist and, in some cases, what they can do for us.

When we perform a BLE scan, we receive short size records of data that always contain the address of the advertiser and *sometimes* additional information such as the services they provide or other descriptive items. This information arrives at us passively. All we need do is listen and we learn. Should we need to learn more about the devices, we can connect to them (even the ones that told us little) and explicitly ask them for more details of what they can do.

And …. that is the principle of scanning. In our C++ BLE story, we have modeled this through the C++ class called `BLEScan`. We get an instance of this class by asking the BLE device for it using:

```
BLEScan* pMyScan = BLE::getScan();
```

The object returned to us is a singleton. This was chosen because we will only want to be scanning once per ESP32 at any given time.

The `BLEScan` object, when presented to us, isn't actually yet scanning, to start it scanning we invoke its `start()` method passing in the duration of how long we would like it to scan for. This is an interval measured in seconds. For example:

```
pMyScan.start(30); // Scan for 30 seconds
```

This is a blocking call. It will return after the scan period has elapsed. It returns a C++ vector data type containing each of the unique devices found. Since we are scanning, we will want to know as soon as possible about the devices that we find and this is where a callback function comes into play. Before calling `start()` we can register a callback function that will be invoked for each peer device that was found. An abstract C++ class called `BLEAdvertisedDeviceCallbacks` works for us here. This has a method on it called `onResult()` which will be invoked for each unique result found during scanning. Note that we don't pass the same detected device twice; if it wasn't the one you werepreviously looking for, it still won't be the one you want a short while later. This method is passed in a reference to an object of type `BLEAdvertisedDevice` that describes the nature of the device that we found.

For example:

```
class MyCallbacks: public BLEAdvertisedDeviceCallbacks {
   void onResult(BLEAdvertisedDevice* pAdvertisedDevice) {
      // Do something with the found device ...
   }
}

BLE::initClient();
BLESan* pMyScan = BLE::getScan();
pMyScan->setAdvertisedDeviceCallbacks(new MyCallbacks());
pMyScan->start();
```

What this means is that for every unique device that the scan finds, a call to `onResult()` will be made which is passed a reference to a `BLEAdvertisedDevice` which describes that device. Now let us look at what a device *may* tell us. The only thing that a device will tell us for sure is its BLE address. Beyond that, everything else is optional. The *possible* attributes that an advertised device may inform us about is large and the following subset have been modeled (so far):

- Appearance

- Manufacturer data
- Name
- RSSI
- Primary service UUID
- Transmit power

Since none of these are mandatory in an advertisement, for any given advertised device we learn about, we can't immediately ask about the value of the property. Instead, we have methods that tell us which properties are present and, if present, **then** we can ask for its value.

For example, in our processing logic we may code:

```
if (pAdvertisedDevice->hasServiceUUID()) {
   BLEUUID service = pAdvertisedDevice->getServiceUUID();
   if (service.equals(BLEUUID((uint16_t)0x1802) {
      // we found a useful device …
   }
}
```

(The above is merely an example of the logic that can be employed).

If we wish to end the scan early; presumably because we found the device we wanted, we can call the `stop()` method of the `BLEScan` object. A reference to the `BLEScan` is conveniently available within the `BLEAdvertisedDevice` object.

```
pAdvertisedDevice->getScan()->stop();
```

From all of this … we end up with the identification of a device that was of interest to us and since we want to be a BLE Client, this will presumably mean the device to which we wish to connect. The key item in the advertisement now becomes the device's address which we can obtain through a call to `getAddress()`.

Now we can turn our attention to actually connecting to the server.

A BLE Client is modeled as the `BLEClient` class. We obtain an unconnected instance of this by asking the ESP32 for one:

```
BLEClient* pMyClient = BLE::createClient();
```

Next we request a connection to the target device.

```
pMyClient->connect(address);
```

… and that's it. The `connect()` is a blocking call and, on return, we are connected. However, that isn't the end of the story. Just connecting to a peer is usually not enough. Now we want to read and write the values that the remote BLE server is hosting. If we think back to our understanding of the story, the remote server has one or more services and each service has one or more characteristics and it is these characteristics that hold the values. This means that it doesn't make sense to say "I want to read the value of the remote BLE server" … instead we must say "I want to read the value of a named remote characteristic". However, even that is not enough. There may be multiple services on the BLE server each of which have their own characteristic which may be of the same characteristic type and hence not be uniquely identified. Thus we end up with the final concept of "I want to read the value of a named remote characteristic that is owned by a named service". From this notion, we now get to introduce two more models. Those are the `BLERemoteService` and `BLERemoteCharacteristic`.

Once we have connected to a BLE Server, we can request a reference to the `BLERemoteService` that models a service on that server:

```
BLERemoteService* pMyRemoteService = pClient->getService(serviceUUID);
```

and once we have a reference to the service, we can ask *that* service for a reference to the desired characteristic.

```
BLERemoteCharacteristic* pMyRemoteCharacteristic =
   pMyRemoteService->getCharacteristic(characteristicUUID);
```

and ... finally … we can work with the values.

```
std::string myValue = pMyRemoteCharacteristic->readValue();
```

to read the value and …

```
pMyRemoteCharacteristic->writeValue("abc");
```

to write a new value. If all that seems like a lot of work … lets put it together in context and see:

```
// Create the client
BLEClient* pMyClient = BLE::createClient();

// Connect the client to the server
pMyClient->connect(address);

// Get a reference to a specific remote service on the server
BLERemoteService* pMyRemoteService = pClient->getService(serviceUUID);

// Get a reference to a specific remote characteristic owned by the service
BLERemoteCharacteristic* pMyRemoteCharacteristic =
   pMyRemoteService->getCharacteristic(characteristicUUID);

// Retrieve the current value of the remote characteristic.
std::string myValue = pMyRemoteCharacteristic->readValue();
```

… and that's it. Hopefully you see that is pretty elegant (and if not, *please* contact me and we'll see if we can't improve on it … but come bearing suggestions for what can be improved).

POSIX file system APIs

The POSIX specification provides some APIs for working with a file system that are mapped to the ESP-IDF Virtual File System. These include:

- close
- fstat
- link
- lseek
- open
- read
- rename
- stat
- unlink
- write

And a set of directory manipulation functions. These functions are defined in:

#include <dirent.h>

The high level notion is that we open a directory by its path and are returned an object that represents that directory. This is a DIR object. Within the DIR object is maintained the state of what entries are contained within the directory as well as a "current pointer" to the next entry to be retrieved.

A call to readdir() returns the next directory entry. This is an instance struct dirent which will contain:

- `d_ino` – The identity of the directory entry.
- `d_name` – The name of the directory entry.
- d_type – The type of the file:
  - `DT_UNKNOWN` – Unknown type of file.
  - `DT_REG` – Regular file.
  - `DR_DIR` – Directory.

The directory functions are:

- `closedir` – Close a directory previous opened by opendir().

- `mkdir` – Create a new directory.
- `opendir` – Open a directory for access, returns a pointer to a DIR.
- `readdir` – Read a directory entry from a DIR.
- `rmdir` – Remove a directory.
- `seekdir` – Change the position of the next directory entry to be retrieved.
- `telldir` – Retrieve the current location of the next directory entry to be retrieved.

Documenting your code - Doxygen

For C and C++ source applications, the defacto standard for documentation is the Doxygen tool. Doxygen is big and rich and here we will capture a cheat sheet to get you going with working with your source.

Doxygen is an application that has dependencies on another tool called "Graphviz".

A GUI tool called "doxywizard" is available after installing "doxygen-gui"

See also:

- Doxygen
- Graphviz

Creating a build environment on the Raspberry Pi 3

Many programmers believe that Linux is a superior development environment than other operating systems such as Windows or Mac OSX. Words like "superior" and "better" are usually subjective and I will avoid any discussion along those lines. Should one choose to use Linux as a development environment, there are many choices for where one hosts it. In this section, we are going to examine using the 4 core Raspberry Pi model 3 as a platform for hosting Linux and building an ESP32 development environment.

The CPU on the Raspberry Pi is ARM based which means that we need a cross compiler to build executables for an ESP32. At the time of writing, there is not a known location for a binary download of the tool suite for ESP32 so our choice is to build it from scratch. Fortunately there are instructions for this in the "`linux-setup.rst`" document found as part of the ESP-IDF project. In summary the steps are:

```
$ sudo apt-get install gawk gperf grep gettext libcurses5-dev python python-dev
automake bison flex texinfo help2man libtool libtool-bin
$ git clone -b xtensa-1.22.x https://github.com/espressif/crosstool-NG.git
$ cd crosstool-NG
$ ./bootstrap && ./configure --prefix=$PWD && make install
$ ./ct-ng xtensa-esp32-elf
$ ./ct-ng build
$ chmod -R u+w builds/xtensa-esp32-elf
```

Be aware that the build of the tool chain requires about 4GBytes of storage in order to complete. As such, you will likely need a much larger micro SD card than the minimum of 8GBytes that is normally used with a Pi. The resulting binaries for the tool chain end up at about 100MBytes. You might want to download the files pre-compiled. A version of the files can be downloaded from my web site from:

http://www.neilkolban.com/esp32/downloads/xtensa-esp32-elf.tar.gz

Once downloaded, I suggest extracting the content into /opt using:

```
$ sudo tar --extract --directory /opt --ungzip --file xtensa-esp32-elf.tar.gz
```

To physically construct a PI/ESP32 environment, I recommend the following. First, get two full sized breadboards and remove the power rail from one of them. Next, bind the two together at the edge where you removed a power rail such that there is only one power rail between the two boards. This will then allow an ESP32 DevKitC to bridge between them. Now on one of the breadboards, we can plug in a Raspberry PI extender. We now have both the PI and the ESP32 col-located on the boards.

Taking a short USB cable, plug the DevKitC into the PI. This will provide both power to the DevKitC as well as give us a serial port. On the PI, if we now run "`lsusb`", we will see the DevKitC:

```
$ lsusb
Bus 001 Device 018: ID 10c4:ea60 Cygnal Integrated Products, Inc. CP210x UART Bridge /
myAVR mySmartUSB light
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast
Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

The DevKitC is the first entry. To further validate, we can look for `/dev/ttyUSB0`:

```
$ ls -l /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 Sep 25 10:27 /dev/ttyUSB0
```

This is the serial port that shows up with a USB→Serial connector such as DevKitC.

We can now connect a serial terminal to this serial port and see the information produced from it. One can use a tool such as "screen" to achieve this task. This tool is not installed by default on the PI so we need to install it with:

```
$ sudo apt-get install screen
```

Once installed, we can run:

```
$ screen /dev/ttyUSB0 115200
```

At this point we should be seeing the serial output from the ESP32. To end our "screen" session, we use the cryptic:

```
CTRL+A
```

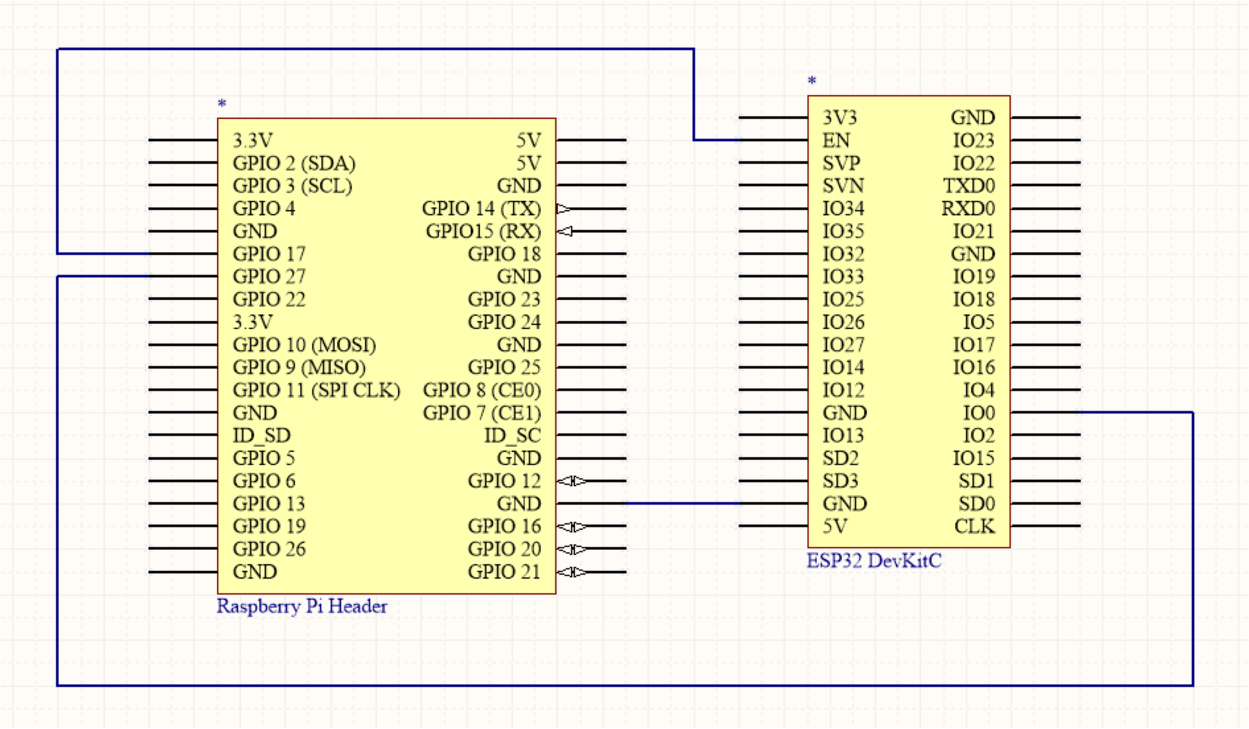
followed by

```
:quit
```

Now let us turn our attention to using the PI to control booting and other useful functions. We want to map some PI GPIO pins to some of the ESP32 functions. The choice of PI pins is arbitrary but in my solution, I used the following:

PI Pin	ESP32 function
17	EN
27	IO0 (Used for boot mode selection)
GND	GND (common ground)

This is illustrated in the following schematic:



Now we can look to some software to achieve our goals.

See also:

- Github: esp-idf/…/linux-setup.rst
- YouTube: ESP32 – Development environment on PI

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
CXX	C++ compiler
CFLAGS	Flags for the C compiler
CPPFLAGS	Flags for the C++ compiler
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.

Wild cards 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)
```

Makefiles also have interesting commands:

- `$(shell <shell command>)` – Run a shell command.
- `$(info "text")`, `$(error "text")`, `$(warning "text")` – Generate output from make.

See also:

- GNU make
- Makefile cheat sheet

The component.mk settings

Within the ESP-IDF environment, we have the `component.mk` build system. This uses specific variables for its configuration.

For a component specific `component.mk` we have:

- `COMPONENT_ADD_INCLUDEDIRS` – Directory to be added to the include path of the entire project.
- `COMPONENT_PRIV_INCLUDEDIRS` – Relative path to include directories that will only be used for module compilation.
- `COMPONENT_DEPENDS` –
- `COMPONENT_ADD_LDFLAGS` –
- `COMPONENT_EXTRA_INCLUDES` –
- `COMPONENT_SRCDIRS` – A set of of one or more directories relative to the location of the component which are expected to contain source files. The files contained in these directories will be compiled and included in the resulting library.
- `COMPONENT_OBJS` – The set of object files that we need as a target.
- `COMPONENT_EXTRA_CLEAN` –
- `COMPONENT_BUILDRECIPE` –
- `COMPONENT_CLEANRECIPE` –
- `COMPONENT_BUILD_DIR` –
- `COMPONENT_EMBED_FILES` – This takes a list of binary files that are then added into the application. The start of the data in flash that corresponds to the file data is then exposed as a linker symbol. The same is also true for the last byte of the file giving two symbols in total. For example, if we were to embed a file called "`mydata.dat`", we might use:

```
COMPONENT_EMBED_FILES:= mydata.dat
```

This would produce two new symbols called `_binary_mydata_dat_start` and `_binary_mydata_dat_end`. These could then be used in an application by using:

```
extern uint8_t mydata_dat_start[] asm("_binary_mydata_dat_start");
extern uint8_t mydata_dat_end[]   asm("_binary_mydata_dat_end");
```

Note that a change to the file on the file systems does not appear to cause a re-build and the old data is used.

- `COMPONENT_EMBED_TXTFILES` – This is identical to `COMPONENT_EMBED_FILES` except that a `NULL` character is added at the end of the data to produce a `NULL` terminated string. This might be used if the file contained character data.
- `CFLAGS` – Use this setting to change C compiler flags passed to the module. Take care to realize that the environment already passes in flags so don't use absolute settings. Instead use "`CFLAGS+=<value>`" to maintain the existing values while adding new ones.

See also:

- Adding a custom ESP-IDF component

Forums

There are a couple of excellent places to ask questions, answer other folks questions and read about questions and answers of the past.

- ESP32 Community Forum – The ESP32 community forum where **all** discussions of ESP32 are happening.

Reference documents

Espressif distributes PDF and Excel spreadsheets containing core information about the ESP32. These can be downloaded freely from the web.

-
- ESP-WROOM-32 Datasheet – 2017-05-04
- ESP32 Technical Reference Manual
- ESP32 Datasheet
- ESP32_RTOS_SDK
- ESP32 Pin List – V2.0
- ESP32 RTOS SDK API Reference v1.1.0
- ESP-IDF Getting Started Guide
- ESP32 AT Instruction Set and Examples
- Mastering the FreeRTOS real time kernel

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:

- espressif/esp-idf – The ESP-IDF … without this we would have little to talk about.
- espressif/esp-idf-template – The template application from which we build new applications.
- espressif/arduino-esp32 – The Arduino environment for the ESP32.

Github quick cheats

When working with open source projects, there are times when we would like to perform some tasks that involve multiple commands. Here we try and capture some of the more interesting ones that are used in ESP32 projects from time to time.

```
git remote -v
git remote add upstream <URL>
git fetch upstream
git merge upstream/master
```

To create a new repository in Github from a directory of goodies, do the following.

1. Create a new github repository
2. echo "# <repository name>" >> README.md
3. git init
4. git add README.md
5. git commit -m "first commit"
6. git remote add origin https://github.com/<repository>.git
7. git push -u origin master

See also:

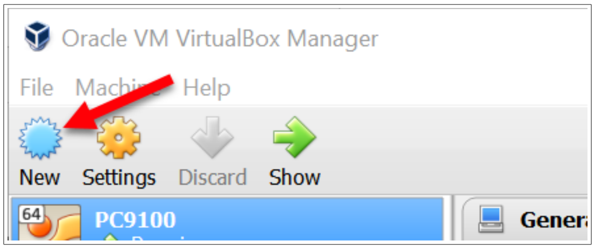
- Simple guide to forks in GitHub and Git

Installing Ubuntu on Virtual Box

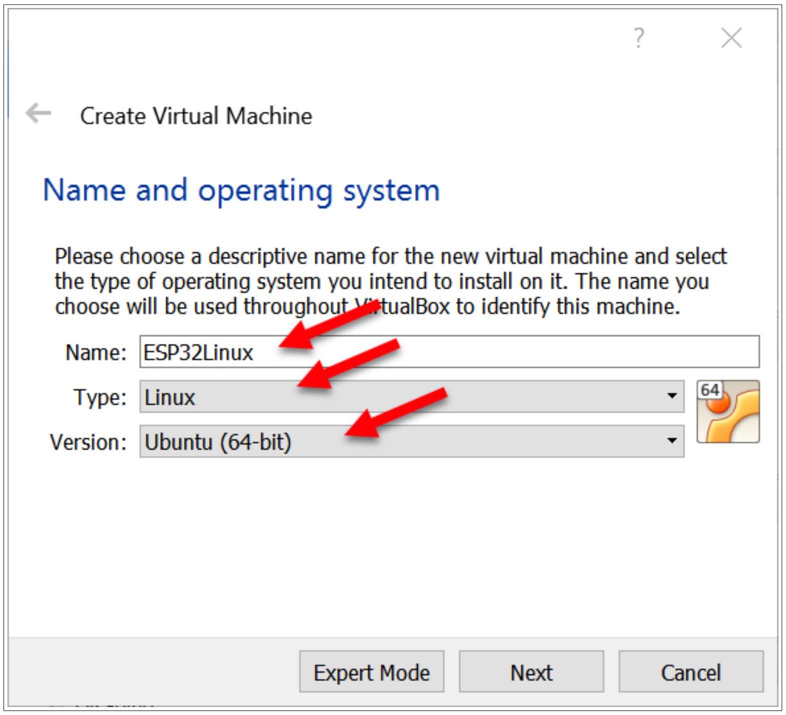
I have a personal preference on the environment on which I perform development. I like to program on a Linux environment. I use Microsoft Windows for all my desktop work including document editing, browsing, email and much, much more … however when it comes to programming, I like Linux. I do **not** want you to believe that in order to work with the ESP32 that you **must** use Linux … that is simply not true. However, if you do decide that you want to use Linux then you will need a Linux environment. Historically, one would load an operating system on a machine and that would be the end of the story. One machine equals one operating system. Thankfully, those days are past and we can now adopt virtual machines. A virtual machine is a logical machine that runs as a "guest" under a host operating system. In our story, we want a virtual Linux machine running as a guest on a Windows or MacOS host. There are many fine vendors of virtual machine technology. The one I have chosen to illustrate in this story is VirtualBox from Oracle. This is an open source implementation that is free for us to use.

Let us now look at the steps necessary to get a Linux environment up and running for ESP32 development. I will assume that you have downloaded and installed the latest VirtualBox. Next we need an image of Linux to install. Personally, I use Ubuntu Desktop which can be downloaded here. The end result of the download will be a file about 1.5GBytes in size. At the time of writing, the download was called "ubuntu-16.04.2-desktop-amd64.iso". The file is an image of a DVD that could be burned and installed from physical media however we don't need to do that, we can install directly from this image.

1. Launch VirtualBox
2. Click "New" to create a new Virtual Machine



3. Give it a name (eg. "ESP32Linux") and specify its type (Linux) and version (Ubuntu 64Bit)



Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

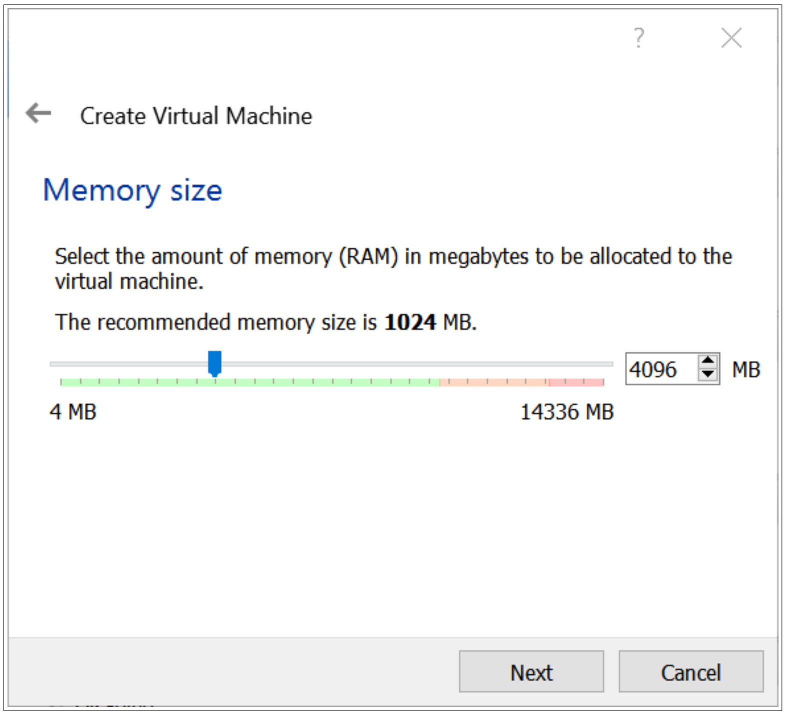
Name: ESP32Linux

Type: Linux

Version: Ubuntu (64-bit)

Click "Next" to continue.

4. Specify how much RAM you want to give the machine.



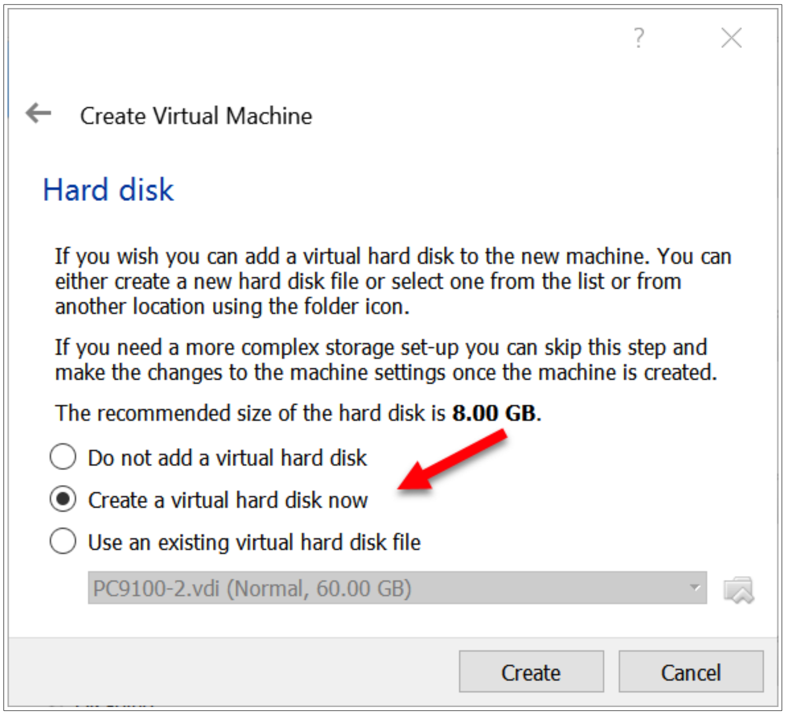
Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1024** MB.

4096 MB

5. Create a virtual disk



Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

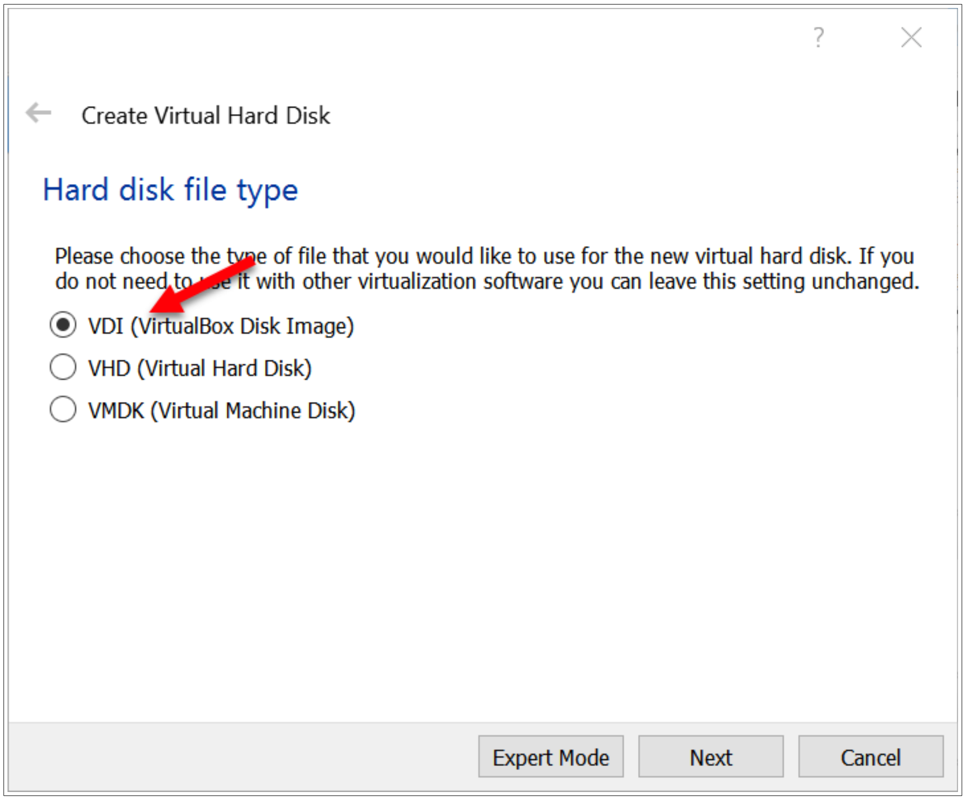
The recommended size of the hard disk is **8.00 GB**.

○ Do not add a virtual hard disk

● Create a virtual hard disk now

○ Use an existing virtual hard disk file

PC9100-2.vdi (Normal, 60.00 GB)

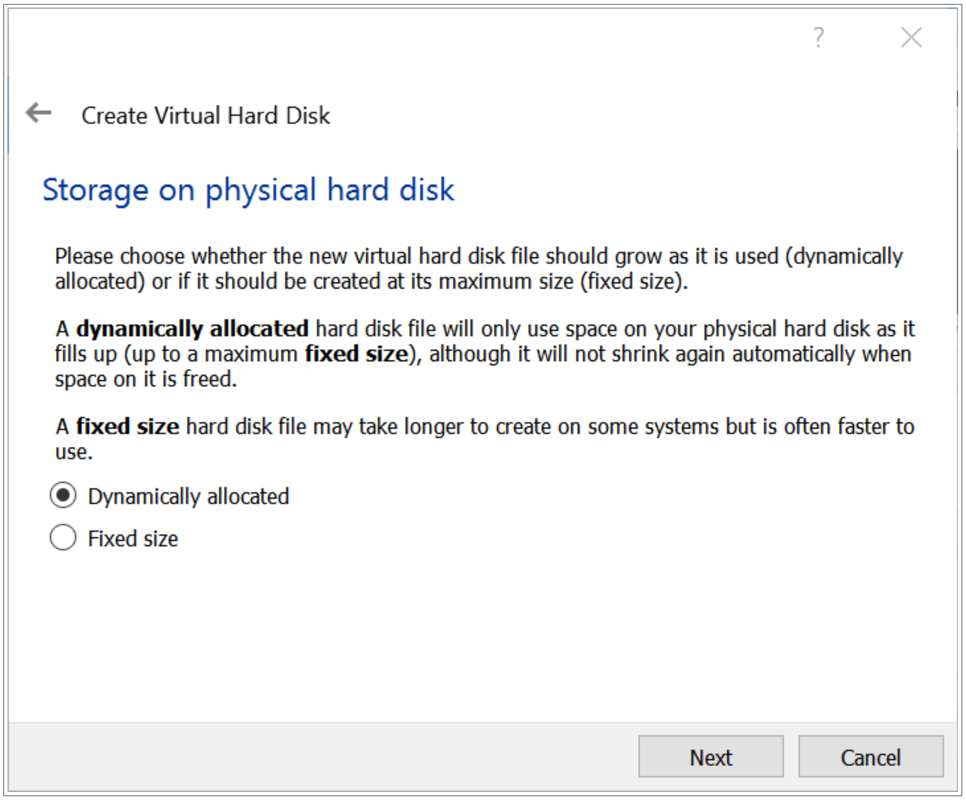


Create Virtual Hard Disk

Hard disk file type

Please choose the type of file that you would like to use for the new virtual hard disk. If you do not need to use it with other virtualization software you can leave this setting unchanged.

● VDI (VirtualBox Disk Image)
○ VHD (Virtual Hard Disk)
○ VMDK (Virtual Machine Disk)



Create Virtual Hard Disk

Storage on physical hard disk

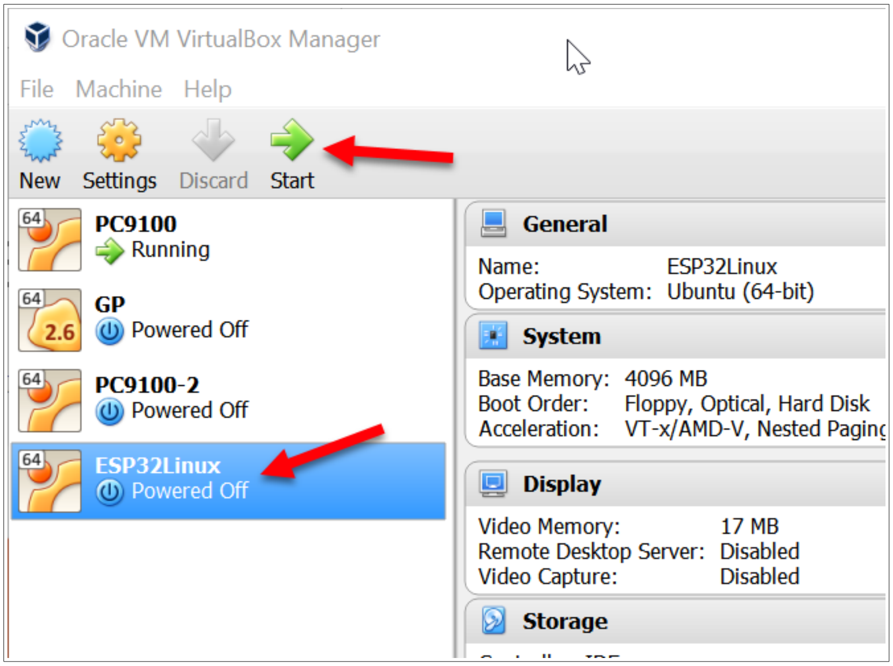
Please choose whether the new virtual hard disk file should grow as it is used (dynamically allocated) or if it should be created at its maximum size (fixed size).

A **dynamically allocated** hard disk file will only use space on your physical hard disk as it fills up (up to a maximum **fixed size**), although it will not shrink again automatically when space on it is freed.

A **fixed size** hard disk file may take longer to create on some systems but is often faster to use.

● Dynamically allocated
○ Fixed size

6. Start the Virtual Machine



PC9100
Running

GP
Powered Off

PC9100-2
Powered Off

ESP32Linux
Powered Off

General

Name: ESP32Linux
Operating System: Ubuntu (64-bit)

System

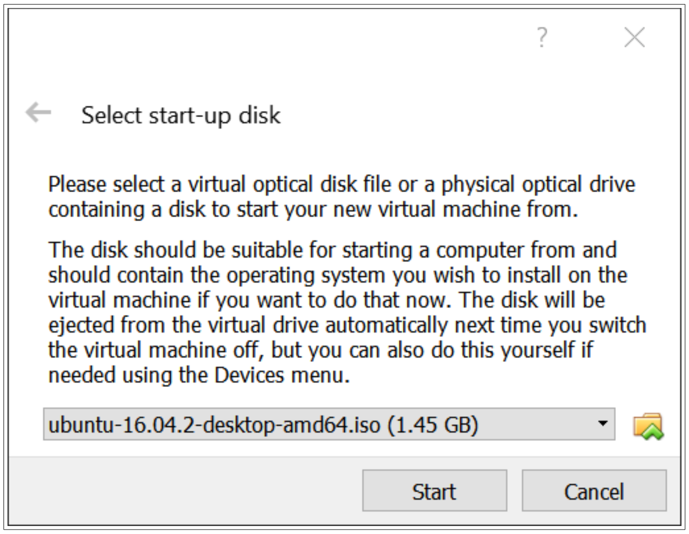
Base Memory: 4096 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Pagin

Display

Video Memory: 17 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

7. Select the Linux installation image



Select start-up disk

Please select a virtual optical disk file or a physical optical drive containing a disk to start your new virtual machine from.

The disk should be suitable for starting a computer from and should contain the operating system you wish to install on the virtual machine if you want to do that now. The disk will be ejected from the virtual drive automatically next time you switch the virtual machine off, but you can also do this yourself if needed using the Devices menu.

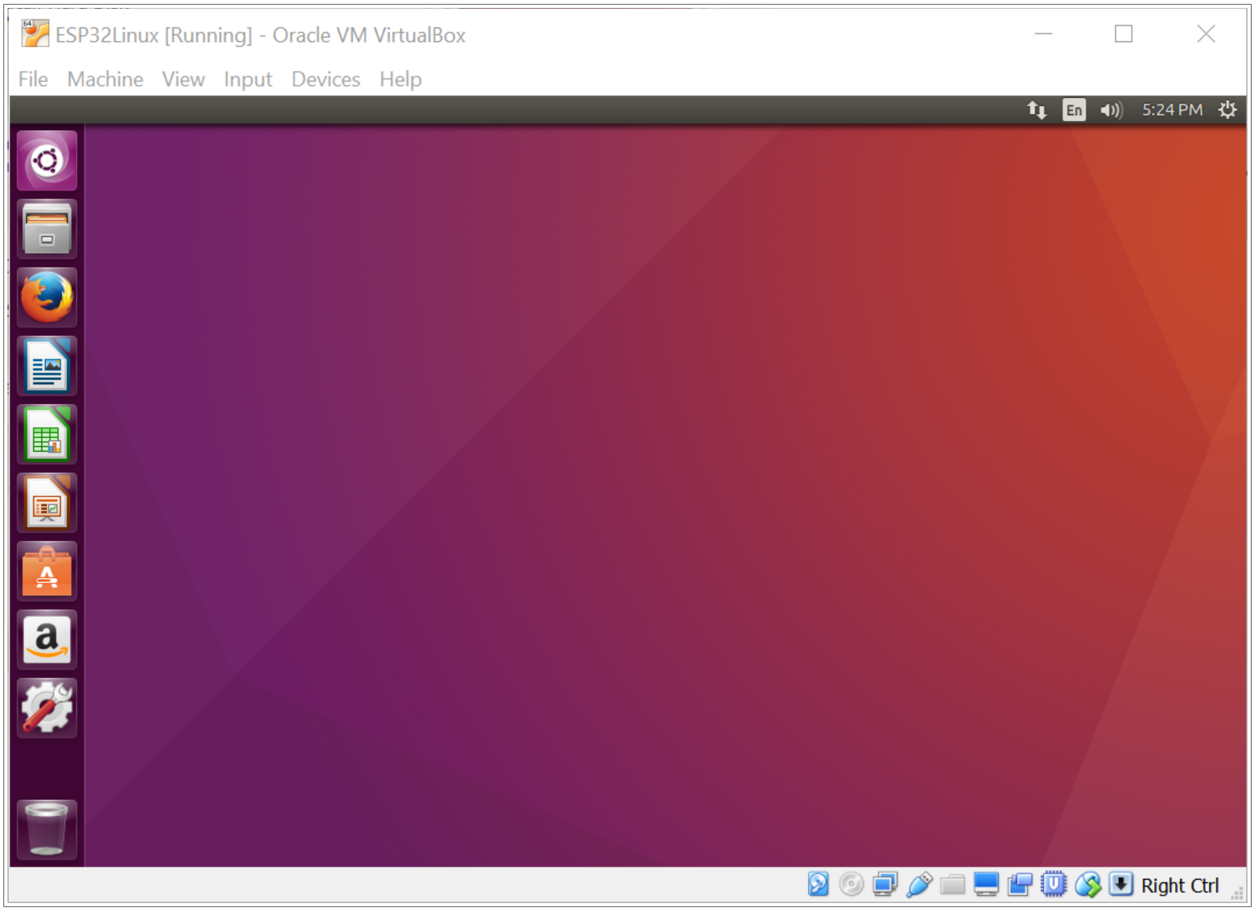
ubuntu-16.04.2-desktop-amd64.iso (1.45 GB)

8. Install Linux

The installation of Ubuntu Linux will now start and take some time to complete. Ten minutes seems to be about average.

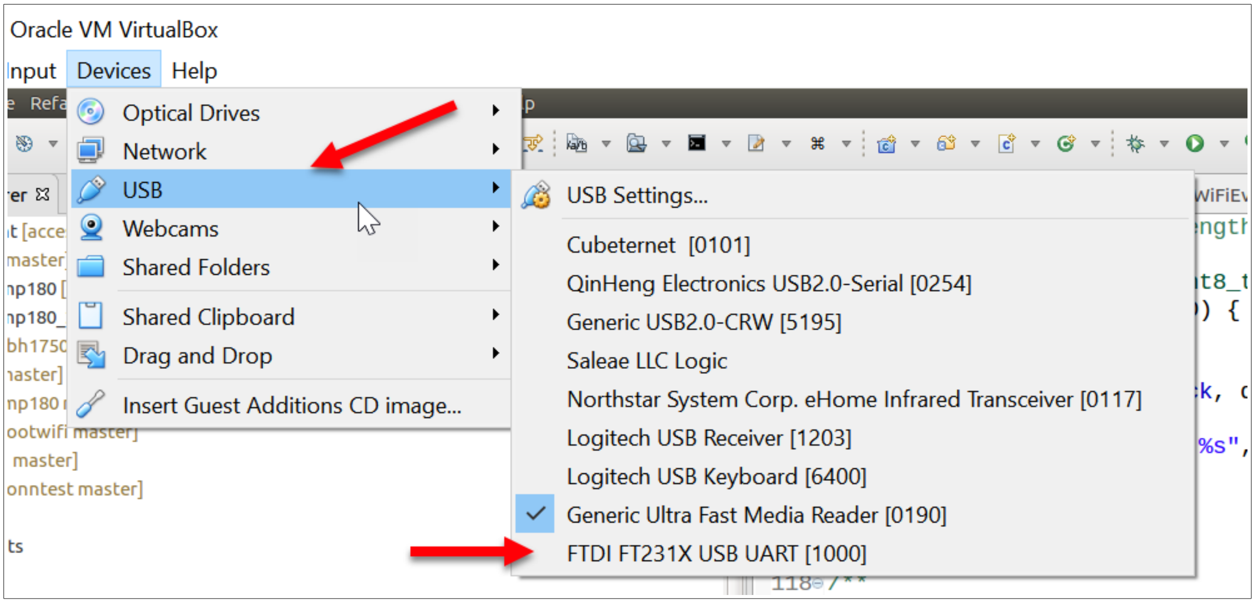
9. Reboot and be in Linux.

Reboot the virtual machine and you will find that you have a Linux environment in which to work.


ESP32Linux [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
En
5:24 PM
Right Ctrl

You can now customize the environment as you wish. Typically I remove the items I don't want from the task bar including FireFox, Libre Office, Ubuntu Software and Amazon.

When you plug in an ESP32 to the USB port of a machine running Virtual Box, you may also have to tell Virtual Box to "own" the USB→UART … in the Devices→USB settings you can see a list of USB devices to add:


Oracle VM VirtualBox
Input
Devices
Help
Optical Drives
Network
USB
Webcams
Shared Folders
Shared Clipboard
Drag and Drop
Insert Guest Additions CD image...
USB Settings...
Cubeternet [0101]
QinHeng Electronics USB2.0-Serial [0254]
Generic USB2.0-CRW [5195]
Saleae LLC Logic
Northstar System Corp. eHome Infrared Transceiver [0117]
Logitech USB Receiver [1203]
Logitech USB Keyboard [6400]
✓ Generic Ultra Fast Media Reader [0190]
FTDI FT231X USB UART [1000]

Its been my experience I have to re-add it here each time after powering on the Virtual Box.

Single board computer comparisons

There are a number of single board computers on the market. Although the ESP8266 is usually not considered one of these, a lot of folks are using it as such. Let us put up a table and contrast the ESP32 against these computers:

Device	CPU	RAM	Flash	Wifi	GPIO	OS	Cost
ESP8266	80MHz	80K	512K	Y	9	FreeRTOS	$4
ESP32	160MHz	512K	Var	Y	?	FreeRTOS	??
Arduino	20MHz	2K	32K	N	?	N/A	$2
Pi Zero	1GHz	512MB	SD	N	?	Linux	$5
Omega	400MHz	64MB	16MB	Y	18	Linux	$19
Omega 2							
C.H.I.P.	1GHz	512MB	4GB	Y	8+	Linux	$9

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?
- How much RAM is installed and available for use?
- Document the information contained here … http://bbs.espressif.com/viewtopic.php?p=3066#p3066
- What is SSDP and how does it related to the SSDP libraries?
- Study Device Hive - http://devicehive.com/
- Document using Visual Micro debugger with Visual Studio. http://www.visualmicro.com/
- Power management
- Research the semantics of a wifi_station_connect() when we are already connected.
- Look at using libssh2 as a library in ESP32
- In power management – what is RTC?
- In power management – what is ULP?
- Study IoT.js
- Look at the GDB post here … interesting subcommand . http://esp32.com/viewtopic.php?f=2&t=767
- How do we set a static IP address for the ESP32 when it is an access point?
- research micro sd APIs
- Study the following http://esp32.com/viewtopic.php?f=13&t=924#p3997
- Write about UDP broadcasting
- Add a section on MCP23017 port expander.
- Study iot.espressif.cn
- Study http://platformio.org/
- Video: Push button interrupts
- Video: Build from scratch – ESP-IDF, compilers etc etc
- Touchpad support
- Document the following http://esp-idf.readthedocs.io/en/latest/build-system.html#embedding-binary-data

- Do a video on watchdog processing
- What is WiFi promiscuous mode?
- Document how to compile JerryScript
- Document how to compile Duktape
- Write an APDS-9330 C++ class
- Add task management methods to the CPP class
- Examine the mDNS responder from apple as a possible port … https://opensource.apple.com/source/mDNSResponder/
- Write and test instructions for building nrf24 on ESP32
- Implement partition logging in C++
- Build samples using ADX345 IC.
- What is Apple iBeacon?
- Write up the AWS support.
- Retest the mDNS support now that we have extra patches.
- study grpc
- Test the libwebsockets https://libwebsockets.org/
- Implement an FTP server
- Implement a ZIP file system
- What is "esp_set_watchpoint" … see https://esp32.com/viewtopic.php?f=2&t=2421
- Build an IR Transmitter app
- Create a video on being a BLE Client
- Build a DMA sample

Private Notes:

My Sparkfun board has devices on the I2C bus at:

- 0x20 – PCF8574

- 0x3C – SSD1305 display
- 0x68 – MPU6050

Special pins are:

PIN 21 – Buzzer – high/on, low/off

PIN 25 – I2C SDA (Green)

PIN 26 – I2C CLK (Yellow)

NRF24

VCC

GND

CE – White - 25

SCK – Blue - 14

MISO – Yellow - 12

IRQ – NA

MOSI – Green - 13

CSN – White - 15