The ESP8266 as a Microcontroller

Pieter P

The ESP8266 as a microcontroller - Hardware

While the ESP8266 is often used as a 'dumb' Serial-to-WiFi bridge, it's a very powerful microcontroller on its own. In this chapter, we'll look at the non-Wi-Fi specific functions of the ESP8266.

Digital I/O

Just like a normal Arduino, the ESP8266 has digital input/output pins (I/O or GPIO, General Purpose Input/Output pins). As the name implies, they can be used as digital inputs to read a digital voltage, or as digital outputs to output either 0V (sink current) or 3.3V (source current).

Voltage and current restrictions

The ESP8266 is a 3.3V microcontroller, so its I/O operates at 3.3V as well. The pins are **not 5V tolerant, applying more than 3.6V on any pin will kill the chip.**

The maximum current that can be drawn from a single GPIO pin is **12mA**.

Usable pins

The ESP8266 has 17 GPIO pins (0-16), however, you can only use 11 of them, because 6 pins (GPIO 6 - 11) are used to connect the flash memory chip. This is the small 8-legged chip right next to the ESP8266. If you try to use one of these pins, you might crash your program.

GPIO 1 and 3 are used as TX and RX of the hardware Serial port (UART), so in most cases, you can't use them as normal I/O while sending/receiving serial data.

Boot modes

As mentioned in the previous chapter, some I/O pins have a special function during boot: They select 1 of 3 boot modes:

GPIO15	GPIO0	GPIO2	Mode
0V	0V	3.3V	Uart Bootloader
0V	3.3V	3.3V	Boot sketch (SPI flash)
3.3V	x	x	SDIO mode (not used for Arduino)

Note: you don't have to add an external pull-up resistor to GPIO2, the internal one is enabled at boot.

We made sure that these conditions are met by adding external resistors in the previous chapter, or the board manufacturer of your board added them for you. This has some implications, however:

- GPIO15 is always pulled low, so you can't use the internal pull-up resistor. You have to keep this in mind when using GPIO15 as an input to read a switch or connect it to a device with an open-collector (or open-drain) output, like I²C.
- GPIO0 is pulled high during normal operation, so you can't use it as a Hi-Z input.
- GPIO2 can't be low at boot, so you can't connect a switch to it.

Internal pull-up/-down resistors

GPIO 0-15 all have a built-in pull-up resistor, just like in an Arduino. GPIO16 has a built-in pull-down resistor.

PWM

Unlike most Atmel chips (Arduino), the ESP8266 doesn't support hardware PWM, however, software PWM is supported on all digital pins. The default PWM range is 10-bits @ 1kHz, but this can be changed (up to >14-bit@1kHz).

Analog input

The ESP8266 has a single analog input, with an input range of 0 - 1.0V. If you supply 3.3V, for example, you will damage the chip. Some boards like the NodeMCU have an on-board resistive voltage divider, to get an easier 0 - 3.3V range. You could also just use a trimpot as a voltage divider.

The ADC (analog to digital converter) has a resolution of 10 bits.

Communication

Serial

The ESP8266 has two hardware UARTS (Serial ports):
UART0 on pins 1 and 3 (TX0 and RX0 resp.), and UART1 on pins 2 and 8 (TX1 and RX1 resp.), however, GPIO8 is used to connect the flash chip. This means that UART1 can only transmit data.

UART0 also has hardware flow control on pins 15 and 13 (RTS0 and CTS0 resp.). These two pins can also be used as alternative TX0 and RX0 pins.

I²C

The ESP doesn't have a hardware TWI (Two Wire Interface), but it is implemented in software. This means that you can use pretty much any two digital pins. By default, the I²C library uses pin 4 as SDA and pin 5 as SCL. (The data sheet specifies GPIO2 as SDA and GPIO14 as SCL.) The maximum speed is approximately 450kHz.

SPI

The ESP8266 has one SPI connection available to the user, referred to as HSPI. It uses GPIO14 as CLK, 12 as MISO, 13 as MOSI and 15 as Slave Select (SS). It can be used in both Slave and Master mode (in software).

GPIO overview

GPIO	Function	State	Restrictions
0	Boot mode select	3.3V	No Hi-Z
1	TX0	-	Not usable during Serial transmission
2	Boot mode select TX1	3.3V (boot only)	Don't connect to ground at boot time Sends debug data at boot time
3	RX0	-	Not usable during Serial transmission
4	SDA (I²C)	-	-
5	SCL (I²C)	-	-
6 - 11	Flash connection	x	Not usable, and not broken out
12	MISO (SPI)	-	-
13	MOSI (SPI)	-	-
14	SCK (SPI)	-	-
15	SS (SPI)	0V	Pull-up resistor not usable
16	Wake up from sleep	-	No pull-up resistor, but pull-down instead Should be connected to RST to wake up

The ESP8266 as a microcontroller - Software

Most of the microcontroller functionality of the ESP uses exactly the same syntax as a normal Arduino, making it really easy to get started.

Digital I/O

Just like with a regular Arduino, you can set the function of a pin using `pinMode(pin, mode);` where `pin` is the GPIO number*, and `mode` can be either `INPUT`, which is the default, `OUTPUT`, or `INPUT_PULLUP` to enable the built-in pull-up resistors for GPIO 0-15. To enable the pull-down resistor for GPIO16, you have to use `INPUT_PULLDOWN_16`.

(*) NodeMCU uses a different pin mapping, read more here. To address a NodeMCU pin, e.g. pin 5, use D5: for instance: `pinMode(D5, OUTPUT);`

To set an output pin high (3.3V) or low (0V), use `digitalWrite(pin, value);` where `pin` is the digital pin, and `value` either 1 or 0 (or `HIGH` and `LOW`).

To read an input, use `digitalRead(pin);`

To enable PWM on a certain pin, use `analogWrite(pin, value);` where `pin` is the digital pin, and `value` a number between 0 and 1023.

You can change the range (bit depth) of the PWM output by using `analogWriteRange(new_range);`

The frequency can be changed by using `analogWriteFreq(new_frequency);`. `new_frequency` should be between 100 and 1000Hz.

Analog input

Just like on an Arduino, you can use `analogRead(A0)` to get the analog voltage on the analog input. (0 = 0V, 1023 = 1.0V).

The ESP can also use the ADC to measure the supply voltage (V_{CC}). To do this, include `ADC_MODE(ADC_VCC);` at the top of your sketch, and use `ESP.getVcc();` to actually get the voltage.
If you use it to read the supply voltage, you can't connect anything else to the analog pin.

Communication

Serial communication

To use UART0 (TX = GPIO1, RX = GPIO3), you can use the `Serial` object, just like on an Arduino: `Serial.begin(baud)`.

To use the alternative pins (TX = GPIO15, RX = GPIO13), use `Serial.swap()` after `Serial.begin`.

To use UART1 (TX = GPIO2), use the `Serial1` object.

All Arduino Stream functions, like `read, write, print, println, ...` are supported as well.

I²C and SPI

You can just use the default Arduino library syntax, like you normally would.

Sharing CPU time with the RF part

One thing to keep in mind while writing programs for the ESP8266 is that your sketch has to share resources (CPU time and memory) with the Wi-Fi- and TCP-stacks (the software that runs in the background and handles all Wi-Fi and IP connections).
If your code takes too long to execute, and don't let the TCP stacks do their thing, it might crash, or you could lose data. It's best to keep the execution time of you loop under a couple of hundreds of milliseconds.

Every time the main loop is repeated, your sketch yields to the Wi-Fi and TCP to handle all Wi-Fi and TCP requests.

If your loop takes longer than this, you will have to explicitly give CPU time to the Wi-Fi/TCP stacks, by using including `delay(0);` or `yield();`. If you don't, network communication won't work as expected, and if it's longer than 3 seconds, the soft WDT (Watch Dog Timer) will reset the ESP. If the soft WDT is disabled, after a little over 8 seconds, the hardware WDT will reset the chip.

From a microcontroller's perspective however, 3 seconds is a very long time (240 million clockcycles), so unless you do some extremely heavy number crunching, or sending extremely long strings over Serial, you won't be affected by this. Just keep in mind that you add the `yield();` inside your `for` or `while` loops that could take longer than, say 100ms.

Sources

This is where I got most of my information to writ this article, there's some more details on the GitHub pages, if you're into some more advanced stuff, like EEPROM or deep sleep etc.

- https://github.com/esp8266/Arduino/issues/2942
- https://github.com/esp8266/Arduino/pull/2533/files
- https://github.com/esp8266/Arduino/blob/master/doc/libraries.md
- https://github.com/esp8266/Arduino/blob/master/doc/reference.md
- https://github.com/esp8266/Arduino/blob/master/doc/boards.md