ESP8266 Arduino Core Documentation

Release 3.1.2-7-g65579d29

Ivan Grokhotkov

Apr 05, 2023

CONTENTS:

CHAPTER

ONE

INSTALLING

1.1 Boards Manager

This is the suggested installation method for end users.

1.1.1 Prerequisites

- Arduino 1.6.8, get it from Arduino website.
- Internet connection
- Python 3 interpreter (Mac/Linux only, Windows installation supplies its own)

1.1.2 Instructions

- Start Arduino and open Preferences window.
- Enter `https://arduino.esp8266.com/stable/package_esp8266com_index.json` into *Additional Board Manager URLs* field. You can add multiple URLs, separating them with commas.
- Open Boards Manager from Tools > Board menu and find *esp8266* platform.
- Select the version you need from a drop-down box.
- Click *install* button.
- Don't forget to select your ESP8266 board from Tools > Board menu after installation.

For more information on the Arduino Board Manager, see:

- https://www.arduino.cc/en/guide/cores

1.2 Using git version

This is the suggested installation method for contributors and library developers.

1.2.1 Prerequisites

- Arduino 1.6.8 (or newer, current working version is 1.8.5)
- git
- Python 3.7 (https://python.org)
- terminal, console, or command prompt (depending on your OS)
- Internet connection
- Uninstalling any core version installed via Board Manager

1.2.2 Instructions - Windows 10

- First, make sure you don't already have an ESP8266 core version installed using the Board Manager (see above). If you do, uninstall it from the Board Manager before proceeding. It is also advisable to erase the Arduino15 contents.
- Install git for Windows (if not already; see https://git-scm.com/download/win)
- Open a command prompt (cmd) and go to Arduino default directory. This is typically the *sketchbook* directory (usually `C:\users\{username}\Documents\Arduino` where the environment variable `%USERPROFILE%` usually contains `C:\users\{username}`)
- Clone this repository into hardware/esp8266com/esp8266 directory.

```
cd %USERPROFILE%\Documents\Arduino\
if not exist hardware mkdir hardware
cd hardware
if not exist esp8266com mkdir esp8266com
cd esp8266com
git clone https://github.com/esp8266/Arduino.git esp8266
```

You should end up with the following directory structure in

`C:\Users\{your username}\Documents\`

```
Arduino
|
--- libraries
--- hardware
    |
    --- esp8266com
        |
        --- esp8266
            |
            --- bootloaders
            --- cores
            --- doc
            --- libraries
            --- package
            --- tests
            --- tools
            --- variants
            --- platform.txt
```

(continues on next page)

(continued from previous page)

```
            --- programmers.txt
            --- README.md
            --- boards.txt
            --- LICENSE
```

- Initialize the submodules

```
cd %USERPROFILE%\Documents\Arduino\hardware\esp8266com\esp8266
git submodule update --init
```

  If error messages about missing files related to `SoftwareSerial` are encountered during the build process, it should be because this step was missed and is required.

- Download binary tools

```
cd tools
python3 get.py
```

- Restart Arduino
- If using the Arduino IDE for Visual Studio (https://www.visualmicro.com/), be sure to click Tools - Visual Micro - Rescan Toolchains and Libraries
- When later updating your local library, goto the esp8266 directory and do a git pull

```
cd %USERPROFILE%\Documents\Arduino\hardware\esp8266com\esp8266
git status
git pull
```

Note that you could, in theory install in `C:\Program Files (x86)\Arduino\hardware` however this has security implications, not to mention the directory often gets blown away when re-installing Arduino IDE. It does have the benefit (or drawback, depending on your perspective) - of being available to all users on your PC that use Arduino.

1.2.3 Instructions - Other OS

- First, make sure you don't already have an ESP8266 core version installed using the Board Manager (see above). If you do, uninstall it from the Board Manager before proceeding. It is also advisable to erase the .arduino15 (Linux) or Arduino15 (MacOS) contents.
- Open the console and go to Arduino directory. This can be either your *sketchbook* directory (usually `<Documents>/Arduino`), or the directory of Arduino application itself, the choice is up to you.
- Clone this repository into hardware/esp8266com/esp8266 directory. Alternatively, clone it elsewhere and create a symlink, if your OS supports them.

```
cd hardware
mkdir esp8266com
cd esp8266com
git clone https://github.com/esp8266/Arduino.git esp8266
```

  You should end up with the following directory structure:

```
Arduino
|
--- hardware
```

(continues on next page)

(continued from previous page)

```
    |
    --- esp8266com
        |
        --- esp8266
            |
            --- bootloaders
            --- cores
            --- doc
            --- libraries
            --- package
            --- tests
            --- tools
            --- variants
            --- platform.txt
            --- programmers.txt
            --- README.md
            --- boards.txt
            --- LICENSE
```

- Initialize the submodules

```
cd esp8266
git submodule update --init
```

If error messages about missing files related to `SoftwareSerial` are encountered during the build process, it should be because this step was missed and is required.

- Download binary tools

```
cd tools
python3 get.py
```

If you get an error message stating that python3 is not found, you will need to install it (most modern UNIX-like OSes provide Python 3 as part of the default install). To install you will need to use `sudo yum install python3`, `sudo apt install python3`, or `brew install python3` as appropriate. On the Mac you may get an error message like:

```
   python3 get.py
   Platform: x86_64-apple-darwin
   Downloading python3-macosx-placeholder.tar.gz
   Traceback (most recent call last):
     File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/
↪request.py", line 1317, in do_open
       encode_chunked=req.has_header('Transfer-encoding'))
     ...
     File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py",␣
↪line 1117, in do_handshake
       self._sslobj.do_handshake()
   ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate␣
↪verify failed: unable to get local issuer certificate (_ssl.c:1056)

This is because Homebrew on the Mac does not always install the required SSL␣
↪certificates by default.  Install them manually (adjust the Python 3.7 as needed)␣
```

(continues on next page)

(continued from previous page)

```
↪with:

.. code:: bash

    cd "/Applications/Python 3.7/" && sudo "./Install Certificates.command"
```

- Restart Arduino
- When later updating your local library, goto the esp8266 directory and do a git pull

```
cd hardware\esp8266com\esp8266
git status
git pull
```

1.3 Using PlatformIO

PlatformIO is an open source ecosystem for IoT development with a cross-platform build system, a library manager, and full support for Espressif (ESP8266) development. It works on the following popular host operating systems: macOS, Windows, Linux 32/64, and Linux ARM (like Raspberry Pi, BeagleBone, CubieBoard).

- What is PlatformIO?
- PlatformIO IDE
- PlatformIO Core (command line tool)
- Advanced usage - custom settings, uploading to LittleFS, Over-the-Air (OTA), staging version
- Integration with Cloud and Standalone IDEs - Cloud9, Codeanywhere, Eclipse Che (Codenvy), Atom, CLion, Eclipse, Emacs, NetBeans, Qt Creator, Sublime Text, VIM, Visual Studio, and VSCode
- Project Examples

CHAPTER

TWO

ESP8266 CONFIGURATION

2.1 Overview

There are a number of specific options for esp8266 in the Arduino IDE Tools menu. Not all of them are available for every board. If one is needed and not visible, please try using the generic esp8266 or esp8285 board.

In every menu entry, the first option is the default one and is suitable for most users (except for flash size in the generic ESP8266 board).

2.2 Note about PlatformIO

PlatformIO specific documentation is also available. Note that this link is available here for reference and is not maintained by the esp8266 Arduino core platform team.

2.3 Arduino IDE Tools Menu

2.3.1 Board

Most of the time there is only one type of ESP8266 chip and only one type of ESP8285(1M) chip shipped with hardware or DIY boards. Capabilities are the same everywhere. Hardware devices differ only on routed GPIO and external components.

If a specific hardware is not available in this list, “Generic ESP82xx” always work.

2.3.2 Upload Speed

This the UART speed setup for flashing the ESP. It is not related with the UART(Serial) speed programmed from inside the sketch, if enabled. Default values are legacy. The highest speed option (around 1Mbaud) should always work. For specific boards, defaults can be updated using the board.txt generator.

2.3.3 CPU Frequency

Any ESP82xx can run at 80 or 160MHz.

2.3.4 Crystal Frequency

This is the on-board crystal frequency (26 or 40Mhz). Default is 26MHz. But the chip was designed with 40MHz. It explains the default strange 74880 baud rate at boot, which is 115200*26/40 (115200 being quite a lot used by default nowadays).

2.3.5 Flash Size

With the Arduino core, ESP82xx can use at most 1MB to store the main sketch in flash memory.

ESP8285 has 1MB internal flash capacity. ESP8266 is always shipped with an external flash chip that is most often 1MB (esp01, esp01s, lots of commercial appliances), 4MB (DIY boards like wemos/lolin D1 mini or nodemcu) or 16MB (lolin D1 mini pro). But configurations with 2MB and 8MB also exist. This core is also able to use older 512KB chips that are today not much used and officially deprecated by Espressif.

Flash space is divided into 3 main zones. The first is the user program space, 1MB at most. The second is enough space for the OTA ability. The third, the remaining space, can be used to hold a filesystem (LittleFS).

This list proposes many different configurations. In the generic board list, the first one of each size is the default and suitable for many cases.

Example: `4MB (FS:2MB OTA:~1019KB)`:

- 4MB is the flash chip size (= 4 MBytes, sometimes oddly noted 32Mbits)
- `OTA:~1019KB` (around 1MB) is used for Over The Air flashing (note that OTA binary can be gzip-ed)
- `FS:2MB` means that 2MBytes are used for an internal filesystem (LittleFS).

2.3.6 Flash Mode

There are four options. The most compatible and slowest is `DOUT`. The fasted is `QIO`. ESP8266 mcu is able to use any of these modes, but depending on the flash chips capabilities and how it is connected to the esp8266, the fastest mode may not be working. Note that ESP8285 requires the `DOUT` mode.

Here is some more insights about that in esp32 forums.

2.3.7 Reset Method

On some boards (commonly NodeMCU, Lolin/Wemos) an electronic trick allows to use the UART DTR line to reset the esp8266 *and* put it in flash mode. This is the default `dtr (aka nodemcu)` option. It provides an extra-easy way of flashing from serial port.

When not available, the `no dtr` option can be used in conjunction with a flash button on the board (or by driving the ESP dedicated GPIOs to boot in flash mode at power-on).

2.3.8 Debug Port

There are three UART options:

- disabled
- Serial
- Serial1

When on `Serial` or `Serial1` options (see *reference*), messages are sent at 74880 bauds at boot time then baud rate is changed to user configuration in sketch. These messages are generated by the internal bootloader. Subsequent serial data are coming either from the firmware, this Arduino core, and user application.

2.3.9 Debug Level

There are a number of options.

- The first (`None`) is explained by itself.
- The last (`NoAssert - NDEBUG`) is even quieter than the first (some internal guards are skipped to save more flash).
- The other ones may be used when asked by a maintainer or if you are a developer trying to debug some issues.

2.3.10 Debug Optimization

Due to the limited resources on the device, our default compiler optimizations focus on creating the smallest code size (`.bin` file). That is fine for release but not ideal for debugging.

`Debug Optimization` use to improve Exception Decoder results.

- `Lite` impact on code size uses `-fno-optimize-sibling-calls` to alter the `-Os` compiler option to place more caller addresses on the Stack.
- `Optimum` offers better quality stack content for the Exception Decoder at the expense of a larger code size. It uses the `-Og` compiler option, which turns off optimizations that can make debugging difficult while keeping others.
- `None` no changes for debugging continue using `-Os`.

Take note some sketches may start working after changing the optimization. Or fail less often. And it is also possible (not likely) that source code that was working with `-Os` may break with `-Og`.

For more topic depth, read Improving Exception Decoder Results

2.3.11 lwIP variant

lwIP is the internal network software stack. It is highly configurable and comes with features that can be enabled, at the price of RAM or FLASH space usage.

There are 6 variants. As always, the first and default option is a good compromise. Note that cores v2.x were or could be using the lwIP-v1 stack. Only lwIP-v2 is available on cores v3+.

- v2 Lower Memory

  This is lwIP-v2 with MSS=536 bytes. MSS is TCP's *Maximum Segment Size*, and different from MTU (IP's Maximum Transfer Unit) which is always 1480 in our case. Using such value for MSS is 99.9% compatible with any TCP peers, allows to store less data in RAM, and is consequently slower when transmitting large segments of data (using TCP) because of a larger overhead and latency due to smaller payload and larger number of packets.

UDP and other IP protocols are not affected by MSS value.

- v2 Higher Bandwidth

  When streaming large amount of data, prefer this option. It uses more memory (MSS=1460) so it allows faster transfers thanks to a smaller number of packets providing lower overhead and higher bandwidth.

- … (no features)

  Disabled features to get more flash space and RAM for users are:

  - No IP Forwarding (=> no NAT),
  - No IP Fragmentation and reassembly,
  - No AutoIP (not getting 169.254.x.x on DHCP request when there is no DHCP answer),
  - No SACK-OUT (= no Selective ACKnowledgements for OUTput):
    no better stability with long distance TCP transfers,
  - No listen backlog (no protection against DOS attacks for TCP server).

- IPv6 …

  With these options, IPv6 is enabled, with features. It uses about 20-30KB of supplementary flash space.

2.3.12 VTable location

This is the mechanism used in C++ to support dynamic dispatch of virtual methods. By default these tables are stored in flash to save precious RAM bytes, but in very specific cases they can be stored in Heap space, or IRAM space (both in RAM).

2.3.13 C++ Exceptions

- C++ exceptions are disabled by default. Consequently the `new` operator will cause a general failure and a reboot when memory is full.

  Note that the C-`malloc` function always returns `nullptr` when memory is full.

- Enabled: on this Arduino core, exceptions are possible. Note that they are quite ram and flash consuming.

2.3.14 Stack protection

- This is disabled by default
- When Enabled, the compiler generated extra code to check for stack overflows. When this happens, an exception is raised with a message and the ESP reboots.

2.3.15 Erase Flash

- `Only sketch`: When WiFi is enabled at boot and persistent WiFi credentials are enabled, these data are preserved across flashings. Filesystem is preserved.
- `Sketch + WiFi settings`: persistent WiFi settings are not preserved accross flashings. Filesystem is preserved.
- `All Flash`: WiFi settings and Filesystems are erased.

2.3.16 NONOS SDK Version

Our Core is based on [Espressif NONOS SDK](https://github.com/espressif/ESP8266_NONOS_SDK).

- **2.2.1+100 (190703)** (default)
- 2.2.1+119 (191122)
- 2.2.1+113 (191105)
- 2.2.1+111 (191024)
- 2.2.1+61 (190313)
- 2.2.1 (legacy)
- 3.0.5 (experimental)

See our issue tracker in regards to default version selection.

- #6724 (comment)
- #6826

Notice that 3.x.x is provided **as-is** and remains **experimental**.

2.3.17 SSL Support

The first and default choice (`All SSL ciphers`) is good. The second option enables only the main ciphers and can be used to lower flash occupation.

2.3.18 MMU (Memory Management Unit)

Head to its *specific documentation*. Note that there is an option providing an additional 16KB of IRAM to your application which can be used with `new` and `malloc`.

2.3.19 Non-32-Bit Access

On esp82xx architecture, DRAM can be accessed byte by byte, but read-only flash space (`PROGMEM` variables) and IRAM cannot. By default they can only be safely accessed in a compatible way using special macros `pgm_read_some()`.

With the non-default option `Byte/Word access`, an exception manager allows to transparently use them as if they were byte-accessible. As a result, any type of access works but in a very slow way for the usually illegal ones. This mode can also be enabled from the MMU options.

CHAPTER

THREE

REFERENCE

3.1 Interrupts

Interrupts can be used on the ESP8266, but they must be used with care and have several limitations:

- Interrupt callback functions must be in IRAM, because the flash may be in the middle of other operations when they occur. Do this by adding the `IRAM_ATTR` attribute on the function definition. If this attribute is not present, the sketch will crash when it attempts to `attachInterrupt` with an error message.

```
IRAM_ATTR void gpio_change_handler(void *data) {...
```

- Interrupts must not call `delay()` or `yield()`, or call any routines which internally use `delay()` or `yield()` either.
- Long-running (>1ms) tasks in interrupts will cause instabilty or crashes. WiFi and other portions of the core can become unstable if interrupts are blocked by a long-running interrupt. If you have much to do, you can set a volatile global flag that your main `loop()` can check each pass or use a scheduled function (which will be called outside of the interrupt context when it is safe) to do long-running work.
- Heap API operations can be dangerous and should be avoided in interrupts. Calls to `malloc` should be minimized because they may require a long running time if memory is fragmented. Calls to `realloc` and `free` must NEVER be called. Using any routines or objects which call `free` or `realloc` themselves is also forbidden for the same reason. This means that `String`, `std::string`, `std::vector` and other classes which use contiguous memory that may be resized must be used with extreme care (ensuring strings aren't changed, vector elements aren't added, etc.). The underlying problem, an allocation address could be actively in use at the instant of an interrupt. Upon return, the address actively in use may be invalid after an ISR uses `realloc` or `free` against the same allocation.
- The C++ `new` and `delete` operators must NEVER be used in an ISR. Their call path is not in IRAM. Using any routines or objects that use the `new` or `delete` operator is also forbidden.

3.2 Digital IO

Pin numbers in Arduino correspond directly to the ESP8266 GPIO pin numbers. `pinMode`, `digitalRead`, and `digitalWrite` functions work as usual, so to read GPIO2, call `digitalRead(2)`.

Digital pins 0—15 can be `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. Pin 16 can be `INPUT`, `OUTPUT` or `INPUT_PULLDOWN_16`. At startup, pins are configured as `INPUT`.

Pins may also serve other functions, like Serial, I2C, SPI. These functions are normally activated by the corresponding library. The diagram below shows pin mapping for the popular ESP-12 module.

Digital pins 6—11 are not shown on this diagram because they are used to connect flash memory chip on most modules. Trying to use these pins as IOs will likely cause the program to crash.

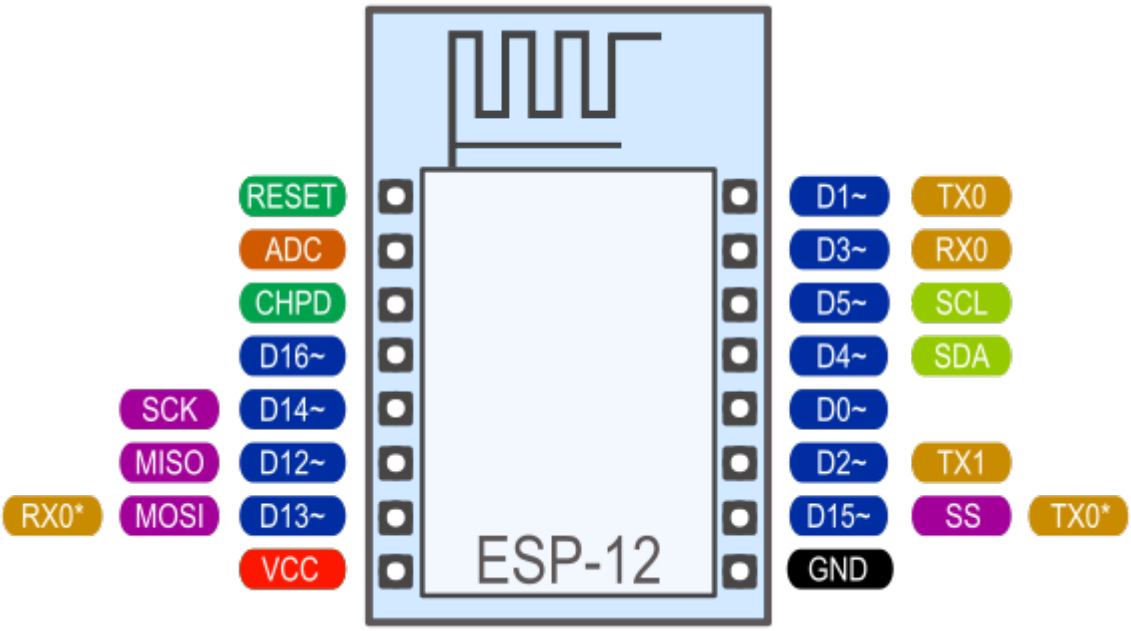


Fig. 1: Pin Functions

Note that some boards and modules (ESP-12ED, NodeMCU 1.0) also break out pins 9 and 11. These may be used as IO if flash chip works in DIO mode (as opposed to QIO, which is the default one).

Pin interrupts are supported through `attachInterrupt`, `detachInterrupt` functions. Interrupts may be attached to any GPIO pin, except GPIO16. Standard Arduino interrupt types are supported: `CHANGE`, `RISING`, `FALLING`. ISRs need to have `IRAM_ATTR` before the function definition.

3.3 Analog input

NOTE: Calling `analogRead()` too frequently causes WiFi to stop working. When WiFi is under operation, `analogRead()` result may be cached for at least 5ms between effective calls.

ESP8266 has a single ADC channel available to users. It may be used either to read voltage at ADC pin, or to read module supply voltage (VCC).

To read external voltage applied to ADC pin, use `analogRead(A0)`. Input voltage range of bare ESP8266 is 0 — 1.0V, however some boards may implement voltage dividers. To be on the safe side, <1.0V can be tested. If e.g. 0.5V delivers values around ~512, then maximum voltage is very likely to be 1.0V and 3.3V may harm the ESP8266. However values around ~150 indicates that the maximum voltage is likely to be 3.3V.

To read VCC voltage, use `ESP.getVcc()` and ADC pin must be kept unconnected. Additionally, the following line has to be added to the sketch:

```
ADC_MODE(ADC_VCC);
```

This line has to appear outside of any functions, for instance right after the `#include` lines of your sketch.

3.4 Analog output

`analogWrite(pin, value)` enables software PWM on the given pin. PWM may be used on pins 0 to 16. Call `analogWrite(pin, 0)` to disable PWM on the pin.

`value` may be in range from 0 to 255 (which is the Arduino default). PWM range may be changed by calling `analogWriteRange(new_range)` or `analogWriteResolution(bits)`. `new_range` may be from 15…65535 or `bits` may be from 4…16.

The function `analogWriteMode(pin, value, openDrain)` allows to sets the pin mode to `OUTPUT_OPEN_DRAIN` instead of `OUTPUT`.

NOTE: The default `analogWrite` range was 1023 in releases before 3.0, but this lead to incompatibility with external libraries which depended on the Arduino core default of 256. Existing applications which rely on the prior 1023 value may add a call to `analogWriteRange(1023)` to their `setup()` routine to return to their old behavior. Applications which already were calling `analogWriteRange` need no change.

PWM frequency is 1kHz by default. Call `analogWriteFreq(new_frequency)` to change the frequency. Valid values are from 100Hz up to 40000Hz.

The ESP doesn't have hardware PWM, so the implementation is by software. With one PWM output at 40KHz, the CPU is already rather loaded. The more PWM outputs used, and the higher their frequency, the closer you get to the CPU limits, and the fewer CPU cycles are available for sketch execution.

3.5 Timing and delays

`millis()` and `micros()` return the number of milliseconds and microseconds elapsed after reset, respectively.

`delay(ms)` pauses the sketch for a given number of milliseconds and allows WiFi and TCP/IP tasks to run. `delayMicroseconds(us)` pauses for a given number of microseconds.

Remember that there is a lot of code that needs to run on the chip besides the sketch when WiFi is connected. WiFi and TCP/IP libraries get a chance to handle any pending events each time the `loop()` function completes, OR when `delay` is called. If you have a loop somewhere in your sketch that takes a lot of time (>50ms) without calling `delay`, you might consider adding a call to `delay` function to keep the WiFi stack running smoothly.

There is also a `yield()` function which is equivalent to `delay(0)`. The `delayMicroseconds` function, on the other hand, does not yield to other tasks, so using it for delays more than 20 milliseconds is not recommended.

3.6 Serial

The `Serial` object works much the same way as on a regular Arduino. Apart from the hardware FIFO (128 bytes for TX and RX), `Serial` has an additional customizable 256-byte RX buffer. The size of this software buffer can be changed by the user. It is suggested to use a bigger size at higher receive speeds.

The `::setRxBufferSize(size_t size)` method changes the RX buffer size as needed. This should be called before `::begin()`. The size argument should be at least large enough to hold all data received before reading.

For transmit-only operation, the 256-byte RX buffer can be switched off to save RAM by passing mode SERIAL_TX_ONLY to Serial.begin(). Other modes are SERIAL_RX_ONLY and SERIAL_FULL (the default).

Receive is interrupt-driven, but transmit polls and busy-waits. Blocking behavior is as follows: The `::write()` call does not block if the number of bytes fits in the current space available in the TX FIFO. The call blocks if the TX FIFO is full and waits until there is room before writing more bytes into it, until all bytes are written. In other words, when the call returns, all bytes have been written to the TX FIFO, but that doesn't mean that all bytes have been sent out

through the serial line yet. The `::read()` call doesn't block, not even if there are no bytes available for reading. The `::readBytes()` call blocks until the number of bytes read complies with the number of bytes required by the argument passed in. The `::flush()` call blocks waiting for the TX FIFO to be empty before returning. It is recommended to call this to make sure all bytes have been sent before doing configuration changes on the serial port (e.g. changing baudrate) or doing a board reset.

`Serial` uses UART0, which is mapped to pins GPIO1 (TX) and GPIO3 (RX). Serial may be remapped to GPIO15 (TX) and GPIO13 (RX) by calling `Serial.swap()` after `Serial.begin`. Calling `swap` again maps UART0 back to GPIO1 and GPIO3.

`Serial1` uses UART1, TX pin is GPIO2. UART1 can not be used to receive data because normally it's RX pin is occupied for flash chip connection. To use `Serial1`, call `Serial1.begin(baudrate)`.

If `Serial1` is not used and `Serial` is not swapped - TX for UART0 can be mapped to GPIO2 instead by calling `Serial.set_tx(2)` after `Serial.begin` or directly with `Serial.begin(baud, config, mode, 2)`.

By default the diagnostic output from WiFi libraries is disabled when you call `Serial.begin`. To enable debug output again, call `Serial.setDebugOutput(true)`. To redirect debug output to `Serial1` instead, call `Serial1.setDebugOutput(true)`.

You also need to use `Serial.setDebugOutput(true)` to enable output from `printf()` function.

Both `Serial` and `Serial1` objects support 5, 6, 7, 8 data bits, odd (O), even (E), and no (N) parity, and 1 or 2 stop bits. To set the desired mode, call `Serial.begin(baudrate, SERIAL_8N1)`, `Serial.begin(baudrate, SERIAL_6E2)`, etc. Default configuration mode is SERIAL_8N1. Possibilities are SERIAL_[5678][NEO][12]. Example: `SERIAL_8N1` means 8bits No parity 1 stop bit.

A new method has been implemented on both `Serial` and `Serial1` to get current baud rate setting. To get the current baud rate, call `Serial.baudRate()`, `Serial1.baudRate()`. Return a `int` of current speed. For example

```
// Set Baud rate to 57600
Serial.begin(57600);

// Get current baud rate
int br = Serial.baudRate();

// Will print "Serial is 57600 bps"
Serial.printf("Serial is %d bps", br);
```

`Serial` and `Serial1` objects are both instances of the `HardwareSerial` class.

This is also done for official ESP8266 *Software Serial* library, see this pull request.

Note that this implementation is **only for ESP8266 based boards**, and will not works with other Arduino boards.

To detect an unknown baudrate of data coming into Serial use `Serial.detectBaudrate(time_t timeoutMillis)`. This method tries to detect the baudrate for a maximum of timeoutMillis ms. It returns zero if no baudrate was detected, or the detected baudrate otherwise. The `detectBaudrate()` function may be called before `Serial.begin()` is called, because it does not need the receive buffer nor the SerialConfig parameters.

The uart can not detect other parameters like number of start- or stopbits, number of data bits or parity.

The detection itself does not change the baudrate, after detection it should be set as usual using `Serial.begin(detectedBaudrate)`.

Detection is very fast, it takes only a few incoming bytes.

SerialDetectBaudrate.ino is a full example of usage.

3.7 Progmem

The Program memory features work much the same way as on a regular Arduino; placing read only data and strings in read only memory and freeing heap for your application.

In core versions prior to 2.7, the important difference is that on the ESP8266 the literal strings are not pooled. This means that the same literal string defined inside a `F("")` and/or `PSTR("")` will take up space for each instance in the code. So you will need to manage the duplicate strings yourself.

Starting from v2.7, this is no longer true: duplicate literal strings within r/o memory are now handled.

There is one additional helper macro to make it easier to pass `const PROGMEM` strings to methods that take a `__FlashStringHelper` called `FPSTR()`. The use of this will help make it easier to pool strings. Not pooling strings...

```
String response1;
response1 += F("http:");
...
String response2;
response2 += F("http:");
```

using FPSTR would become...

```
const char HTTP[] PROGMEM = "http:";
...
{
    String response1;
    response1 += FPSTR(HTTP);
    ...
    String response2;
    response2 += FPSTR(HTTP);
}
```

3.8 C++

- About C++ exceptions, `operator new`, and Exceptions menu option

  The C++ standard says the following about the `new` operator behavior when encountering heap shortage (memory full):

  - has to throw a `std::bad_alloc` C++ exception when they are enabled
  - will `abort()` otherwise

  There are several reasons for the first point above, among which are:

  - guarantee that the return of new is never a `nullptr`
  - guarantee full construction of the top level object plus all member subobjects
  - guarantee that any subobjects partially constructed get destroyed, and in the correct order, if oom is encountered midway through construction

  When C++ exceptions are disabled, or when using `new(nothrow)`, the above guarantees can't be upheld, so the second point (`abort()`) above is the only `std::c++` viable solution.

  Historically in Arduino environments, `new` is overloaded to simply return the equivalent `malloc()` which in turn can return `nullptr`.

This behavior is not C++ standard, and there is good reason for that: there are hidden and very bad side effects. The *class and member constructors are always called, even when memory is full* (`this == nullptr`). In addition, the memory allocation for the top object could succeed, but allocation required for some member object could fail, leaving construction in an undefined state. So the historical behavior of Ardudino's `new`, when faced with insufficient memory, will lead to bad crashes sooner or later, sometimes unexplainable, generally due to memory corruption even when the returned value is checked and managed. Luckily on esp8266, trying to update RAM near address 0 will immediately raise an hardware exception, unlike on other uC like avr on which that memory can be accessible.

As of core 2.6.0, there are 3 options: legacy (default) and two clear cases when `new` encounters oom:

- `new` returns `nullptr`, with possible bad effects or immediate crash when constructors (called anyway) initialize members (exceptions are disabled in this case)
- C++ exceptions are disabled: `new` calls `abort()` and will "cleanly" crash, because there is no way to honor memory allocation or to recover gracefully.
- C++ exceptions are enabled: `new` throws a `std::bad_alloc` C++ exception, which can be caught and handled gracefully. This assures correct behavior, including handling of all subobjects, which guarantees stability.

History: #6269 #6309 #6312

3.9 Streams

Arduino API

Stream is one of the core classes in the Arduino API. Wire, serial, network and filesystems are streams, from which data are read or written.

Making a transfer with streams is quite common, like for example the historical WiFiSerial sketch:

```
//check clients for data
//get data from the telnet client and push it to the UART
while (serverClient.available()) {
  Serial.write(serverClient.read());
}

//check UART for data
if (Serial.available()) {
  size_t len = Serial.available();
  uint8_t sbuf[len];
  Serial.readBytes(sbuf, len);
  //push UART data to all connected telnet clients
  if (serverClient && serverClient.connected()) {
    serverClient.write(sbuf, len);
  }
}
```

One will notice that in the network to serial direction, data are transferred byte by byte while data are available. In the other direction, a temporary buffer is created on stack, filled with available serial data, then transferred to network.

The `readBytes(buffer, length)` method includes a timeout to ensure that all required bytes are received. The `write(buffer, length)` (inherited from `Print::`) function is also usually blocking until the full buffer is transmitted. Both functions return the number of transmitted bytes.

That's the way the Stream class works and is commonly used.

Classes derived from `Stream::` also usually introduce the `read(buffer, len)` method, which is similar to `readBytes(buffer, len)` without timeout: the returned value can be less than the requested size, so special care must be taken with this function, introduced in the Arduino `Client::` class (cf. AVR reference implementation). This function has also been introduced in other classes that don't derive from `Client::`, e.g. `HardwareSerial::`.

Stream extensions

Stream extensions are designed to be compatible with Arduino API, and offer additional methods to make transfers more efficient and easier to use.

The serial to network transfer above can be written like this:

```
serverClient.sendAvailable(Serial); // chunk by chunk
Serial.sendAvailable(serverClient); // chunk by chunk
```

An echo service can be written like this:

```
serverClient.sendAvailable(serverClient); // tcp echo service

Serial.sendAvailable(Serial);             // serial software loopback
```

Beside reducing coding time, these methods optimize transfers by avoiding buffer copies when possible.

- User facing API: `Stream::send()`

  The goal of streams is to transfer data between producers and consumers, like the telnet/serial example above. Four methods are provided, all of them return the number of transmitted bytes:

  - `Stream::sendSize(dest, size [, timeout])`

    This method waits up to the given or default timeout to transfer `size` bytes to the the `dest` Stream.

  - `Stream::sendUntil(dest, delim [, timeout])`

    This method waits up to the given or default timeout to transfer data until the character `delim` is met. Note: The delimiter is read but not transferred (like `readBytesUntil`)

  - `Stream::sendAvailable(dest)`

    This method transfers all already available data to the destination. There is no timeout and the returned value is 0 when there is nothing to transfer or no room in the destination.

  - `Stream::sendAll(dest [, timeout])`

    This method waits up to the given or default timeout to transfer all available data. It is useful when source is able to tell that no more data will be available for this call, or when destination can tell that it will no be able to receive anymore.

    For example, a source String will not grow during the transfer, or a particular network connection supposed to send a fixed amount of data before closing. `::sendAll()` will receive all bytes. Timeout is useful when destination needs processing time (e.g. network or serial input buffer full = please wait a bit).

- String, flash strings helpers

  Two additional classes are provided.

  - `StreamConstPtr::` is designed to hold a constant buffer (in ram or flash).

With this class, a `Stream::` can be made from `const char*`, `F("some words in flash")` or `PROGMEM` strings. This class makes no copy, even with data in flash. For flash content, byte-by-byte transfers is a consequence when "memcpy_P" cannot be used. Other contents can be transferred at once when possible.

```
StreamConstPtr css(F("my long css data")); // CSS data not copied to RAM
server.sendAll(css);
```

- `S2Stream::` is designed to make a `Stream::` out of a `String::` without copy.

```
String helloString("hello");
S2Stream hello(helloString);
hello.reset(0);        // prevents ::read() to consume the string

hello.sendAll(Serial); // shows "hello"
hello.sendAll(Serial); // shows nothing, content has already been read
hello.reset();         // reset content pointer
hello.sendAll(Serial); // shows "hello"
hello.reset(3);        // reset content pointer to a specific position
hello.sendAll(Serial); // shows "lo"

hello.setConsume();    // ::read() will consume, this is the default
Serial.println(helloString.length()); // shows 5
hello.sendAll(Serial);                // shows "hello"
Serial.println(helloString.length()); // shows 0, string is consumed
```

`StreamString::` derives from `S2Stream`

```
StreamString contentStream;
client.sendSize(contentStream, SOME_SIZE); // receives at most SOME_
↪SIZE bytes

// equivalent to:

String content;
S2Stream contentStream(content);
client.sendSize(contentStream, SOME_SIZE); // receives at most SOME_
↪SIZE bytes
// content has the data
```

- Internal Stream API: `peekBuffer`

  Here is the method list and their significations. They are currently implemented in `HardwareSerial`, `WiFiClient` and `WiFiClientSecure`.

  - `virtual bool hasPeekBufferAPI ()` returns `true` when the API is present in the class
  - `virtual size_t peekAvailable ()` returns the number of reachable bytes
  - `virtual const char* peekBuffer ()` returns the pointer to these bytes

    This API requires that any kind of `"read"` function must not be called after `peekBuffer()` and until `peekConsume()` is called.
  - `virtual void peekConsume (size_t consume)` tells to discard that number of bytes
  - `virtual bool inputCanTimeout ()`

A `StringStream` will return false. A closed network connection returns false. This function allows `Stream::sendAll()` to return earlier.

- `virtual bool outputCanTimeout ()`

  A closed network connection returns false. This function allows `Stream::sendAll()` to return earlier.

- `virtual ssize_t streamRemaining()`

  It returns -1 when stream remaining size is unknown, depending on implementation (string size, file size..).

CHAPTER

FOUR

LIBRARIES

4.1 WiFi (ESP8266WiFi library)

ESP8266WiFi library has been developed basing on ESP8266 SDK, using naming convention and overall functionality philosophy of the Arduino WiFi Shield library. Over time the wealth Wi-Fi features ported from ESP8266 SDK to this library outgrew the APIs of WiFi Shield library and it became apparent that we need to provide separate documentation on what is new and extra.

ESP8266WiFi library documentation

4.2 Ticker

Library for calling functions repeatedly with a certain period. Three examples included.

It is currently not recommended to do blocking IO operations (network, serial, file) from Ticker callback functions. Instead, set a flag inside the ticker callback and check for that flag inside the loop function.

Here is library to simplificate `Ticker` usage and avoid WDT reset: TickerScheduler

4.3 EEPROM

This is a bit different from standard EEPROM class. You need to call `EEPROM.begin(size)` before you start reading or writing, size being the number of bytes you want to use. Size can be anywhere between 4 and 4096 bytes.

`EEPROM.write` does not write to flash immediately, instead you must call `EEPROM.commit()` whenever you wish to save changes to flash. `EEPROM.end()` will also commit, and will release the RAM copy of EEPROM contents.

EEPROM library uses one sector of flash located just after the embedded filesystem.

Three examples included.

Note that the sector needs to be re-flashed every time the changed EEPROM data needs to be saved, thus will wear out the flash memory very quickly even if small amounts of data are written. Consider using one of the EEPROM libraries mentioned down below.

4.4 I2C (Wire library)

Wire library currently supports master mode up to approximately 450KHz. Before using I2C, pins for SDA and SCL need to be set by calling `Wire.begin(int sda, int scl)`, i.e. `Wire.begin(0, 2)` on ESP-01, else they default to pins 4(SDA) and 5(SCL).

4.5 SPI

SPI library supports the entire Arduino SPI API including transactions, including setting phase (CPHA). Setting the Clock polarity (CPOL) is not supported, yet (SPI_MODE2 and SPI_MODE3 not working).

The usual SPI pins are:

- `MOSI` = GPIO13
- `MISO` = GPIO12
- `SCLK` = GPIO14

There's an extended mode where you can swap the normal pins to the SPI0 hardware pins. This is enabled by calling `SPI.pins(6, 7, 8, 0)` before the call to `SPI.begin()`. The pins would change to:

- `MOSI` = SD1
- `MISO` = SD0
- `SCLK` = CLK
- `HWCS` = GPIO0

This mode shares the SPI pins with the controller that reads the program code from flash and is controlled by a hardware arbiter (the flash has always higher priority). For this mode the CS will be controlled by hardware as you can't handle the CS line with a GPIO, you never actually know when the arbiter is going to grant you access to the bus so you must let it handle CS automatically.

4.6 SoftwareSerial

An ESP8266 port of SoftwareSerial library done by Peter Lerup (@plerup) supports baud rate up to 115200 and multiples SoftwareSerial instances. See https://github.com/plerup/espsoftwareserial if you want to suggest an improvement or open an issue related to SoftwareSerial.

4.7 ESP-specific APIs

Some ESP-specific APIs related to deep sleep, RTC and flash memories are available in the `ESP` object.

`ESP.deepSleep(microseconds, mode)` will put the chip into deep sleep. `mode` is one of `WAKE_RF_DEFAULT`, `WAKE_RFCAL`, `WAKE_NO_RFCAL`, `WAKE_RF_DISABLED`. (GPIO16 needs to be tied to RST to wake from deepSleep.) The chip can sleep for at most `ESP.deepSleepMax()` microseconds. If you implement deep sleep with `WAKE_RF_DISABLED` and require WiFi functionality on wake up, you will need to implement an additional `WAKE_RF_DEFAULT` before WiFi functionality is available.

`ESP.deepSleepInstant(microseconds, mode)` works similarly to `ESP.deepSleep` but sleeps instantly without waiting for WiFi to shutdown.

`ESP.rtcUserMemoryWrite(offset, &data, sizeof(data))` and `ESP.rtcUserMemoryRead(offset, &data, sizeof(data))` allow data to be stored in and retrieved from the RTC user memory of the chip respectively. `offset` is measured in blocks of 4 bytes and can range from 0 to 127 blocks (total size of RTC memory is 512 bytes). `data` should be 4-byte aligned. The stored data can be retained between deep sleep cycles, but might be lost after power cycling the chip. Data stored in the first 32 blocks will be lost after performing an OTA update, because they are used by the Core internals.

`ESP.restart()` restarts the CPU.

`ESP.getResetReason()` returns a String containing the last reset reason in human readable format.

`ESP.getFreeHeap()` returns the free heap size.

`ESP.getHeapFragmentation()` returns the fragmentation metric (0% is clean, more than ~50% is not harmless)

`ESP.getMaxFreeBlockSize()` returns the largest contiguous free RAM block in the heap, useful for checking heap fragmentation. **NOTE:** Maximum `malloc()` -able block will be smaller due to memory manager overheads.

`ESP.getChipId()` returns the ESP8266 chip ID as a 32-bit integer.

`ESP.getCoreVersion()` returns a String containing the core version.

`ESP.getSdkVersion()` returns the SDK version as a char.

`ESP.getCpuFreqMHz()` returns the CPU frequency in MHz as an unsigned 8-bit integer.

`ESP.getSketchSize()` returns the size of the current sketch as an unsigned 32-bit integer.

`ESP.getFreeSketchSpace()` returns the free sketch space as an unsigned 32-bit integer.

`ESP.getSketchMD5()` returns a lowercase String containing the MD5 of the current sketch.

`ESP.getFlashChipId()` returns the flash chip ID as a 32-bit integer.

`ESP.getFlashChipSize()` returns the flash chip size, in bytes, as seen by the SDK (may be less than actual size).

`ESP.getFlashChipRealSize()` returns the real chip size, in bytes, based on the flash chip ID.

`ESP.getFlashChipSpeed(void)` returns the flash chip frequency, in Hz.

`ESP.getCycleCount()` returns the cpu instruction cycle count since start as an unsigned 32-bit. This is useful for accurate timing of very short actions like bit banging.

`ESP.random()` should be used to generate true random numbers on the ESP. Returns an unsigned 32-bit integer with the random number. An alternate version is also available that fills an array of arbitrary length. Note that it seems as though the WiFi needs to be enabled to generate entropy for the random numbers, otherwise pseudo-random numbers are used.

`ESP.checkFlashCRC()` calculates the CRC of the program memory (not including any filesystems) and compares it to the one embedded in the image. If this call returns `false` then the flash has been corrupted. At that point, you may want to consider trying to send a MQTT message, to start a re-download of the application, blink a LED in an *SOS* pattern, etc. However, since the flash is known corrupted at this point there is no guarantee the app will be able to perform any of these operations, so in safety critical deployments an immediate shutdown to a fail-safe mode may be indicated.

`ESP.getVcc()` may be used to measure supply voltage. ESP needs to reconfigure the ADC at startup in order for this feature to be available. Add the following line to the top of your sketch to use `getVcc`:

```
ADC_MODE(ADC_VCC);
```

TOUT pin has to be disconnected in this mode.

Note that by default ADC is configured to read from TOUT pin using `analogRead(A0)`, and `ESP.getVCC()` is not available.

4.8 mDNS and DNS-SD responder (ESP8266mDNS library)

Allows the sketch to respond to multicast DNS queries for domain names like "foo.local", and DNS-SD (service discovery) queries. See attached example for details.

4.9 SSDP responder (ESP8266SSDP)

SSDP is another service discovery protocol, supported on Windows out of the box. See attached example for reference.

4.10 DNS server (DNSServer library)

Implements a simple DNS server that can be used in both STA and AP modes. The DNS server currently supports only one domain (for all other domains it will reply with NXDOMAIN or custom status code). With it, clients can open a web server running on ESP8266 using a domain name, not an IP address.

4.11 Servo

This library exposes the ability to control RC (hobby) servo motors. It will support up to 24 servos on any available output pin. By default the first 12 servos will use Timer0 and currently this will not interfere with any other support. Servo counts above 12 will use Timer1 and features that use it will be affected. While many RC servo motors will accept the 3.3V IO data pin from a ESP8266, most will not be able to run off 3.3v and will require another power source that matches their specifications. Make sure to connect the grounds between the ESP8266 and the servo motor power supply.

4.12 Other libraries (not included with the IDE)

Libraries that don't rely on low-level access to AVR registers should work well. Here are a few libraries that were verified to work:

- Adafruit_ILI9341 - Port of the Adafruit ILI9341 for the ESP8266
- arduinoVNC - VNC Client for Arduino
- arduinoWebSockets - WebSocket Server and Client compatible with ESP8266 (RFC6455)
- aREST - REST API handler library.
- Blynk - easy IoT framework for Makers (check out the Kickstarter page).
- DallasTemperature
- DHT-sensor-library - Arduino library for the DHT11/DHT22 temperature and humidity sensors. Download latest v1.1.1 library and no changes are necessary. Older versions should initialize DHT as follows: `DHT dht(DHTPIN, DHTTYPE, 15)`
- DimSwitch - Control electronic dimmable ballasts for fluorescent light tubes remotely as if using a wall switch.
- Encoder - Arduino library for rotary encoders. Version 1.4 supports ESP8266.
- esp8266_mdns - mDNS queries and responses on esp8266. Or to describe it another way: An mDNS Client or Bonjour Client library for the esp8266.

- ESP-NOW - Wrapper lib for ESP-NOW (See #2227)
- ESPAsyncTCP - Asynchronous TCP Library for ESP8266 and ESP32/31B
- ESPAsyncWebServer - Asynchronous Web Server Library for ESP8266 and ESP32/31B
- Homie for ESP8266 - Arduino framework for ESP8266 implementing Homie, an MQTT convention for the IoT.
- NeoPixel - Adafruit's NeoPixel library, now with support for the ESP8266 (use version 1.0.2 or higher from Arduino's library manager).
- NeoPixelBus - Arduino NeoPixel library compatible with ESP8266. Use the "DmaDriven" or "UartDriven" branches for ESP8266. Includes HSL color support and more.
- PubSubClient - MQTT library by @Imroy.
- RTC - Arduino Library for Ds1307 & Ds3231 compatible with ESP8266.
- Souliss, Smart Home - Framework for Smart Home based on Arduino, Android and openHAB.
- ST7735 - Adafruit's ST7735 library modified to be compatible with ESP8266. Just make sure to modify the pins in the examples as they are still AVR specific.
- Task - Arduino Nonpreemptive multitasking library. While similar to the included Ticker library in the functionality provided, this library was meant for cross Arduino compatibility.
- TickerScheduler - Library provides simple scheduler for `Ticker` to avoid WDT reset
- Teleinfo - Generic French Power Meter library to read Teleinfo energy monitoring data such as consuption, contract, power, period, … This library is cross platform, ESP8266, Arduino, Particle, and simple C++. French dedicated post on author's blog and all related information about Teleinfo also available.
- UTFT-ESP8266 - UTFT display library with support for ESP8266. Only serial interface (SPI) displays are supported for now (no 8-bit parallel mode, etc). Also includes support for the hardware SPI controller of the ESP8266.
- WiFiManager - WiFi Connection manager with web captive portal. If it can't connect, it starts AP mode and a configuration portal so you can choose and enter WiFi credentials.
- OneWire - Library for Dallas/Maxim 1-Wire Chips.
- Adafruit-PCD8544-Nokia-5110-LCD-Library - Port of the Adafruit PCD8544 - library for the ESP8266.
- PCF8574_ESP - A very simplistic library for using the PCF857//PCF8574A I2C 8-pin GPIO-expander.
- Dot Matrix Display Library 2 - Freetronics DMD & Generic 16 x 32 P10 style Dot Matrix Display Library
- SdFat-beta - SD-card library with support for long filenames, software- and hardware-based SPI and lots more.
- FastLED - a library for easily & efficiently controlling a wide variety of LED chipsets, like the Neopixel (WS2812B), DotStar, LPD8806 and many more. Includes fading, gradient, color conversion functions.
- OLED - a library for controlling I2C connected OLED displays. Tested with 0.96 inch OLED graphics display.
- MFRC522 - A library for using the Mifare RC522 RFID-tag reader/writer.
- Ping - lets the ESP8266 ping a remote machine.
- AsyncPing - fully asynchronous Ping library (have full ping statistic and hardware MAC address).
- ESP_EEPROM - This library writes a new copy of your data when you save (commit) it and keeps track of where in the sector the most recent copy is kept using a bitmap. The flash sector only needs to be erased when there is no more space for copies in the flash sector.
- EEPROM Rotate - Instead of using a single sector to persist the data from the emulated EEPROM, this library uses a number of sectors to do so: a sector pool.

CHAPTER

FIVE

FILESYSTEM

5.1 Flash layout

Even though file system is stored on the same flash chip as the program, programming new sketch will not modify file system contents. This allows to use file system to store sketch data, configuration files, or content for Web server.

The following diagram illustrates flash layout used in Arduino environment:

```
|--------------|-------|---------------|--|--|--|--|
^              ^       ^               ^     ^
Sketch    OTA update   File system   EEPROM  WiFi config (SDK)
```

File system size depends on the flash chip size. Depending on the board which is selected in IDE, the following table shows options for flash size.

Another option called `Mapping defined by Hardware and Sketch` is available. It allows a sketch, not the user, to select FS configuration at boot according to flash chip size.

This option is also enabled with this compilation define: `-DFLASH_MAP_SUPPORT=1`.

There are three possible configurations:

- `FLASH_MAP_OTA_FS`: largest available space for onboard FS, allowing OTA (noted 'OTA' in the table)
- `FLASH_MAP_MAX_FS`: largest available space for onboard FS (noted 'MAX' in the table)
- `FLASH_MAP_NO_FS`: no onboard FS

Sketch can invoke a particular configuration by adding this line:

```
FLASH_MAP_SETUP_CONFIG(FLASH_MAP_OTA_FS)
void setup () { ... }
void loop () { ... }
```

Board	Flash chip size (bytes)	File system size (bytes)
Any	512KBytes	32KB(OTA), 64KB, 128KB(MAX)
Any	1MBytes	64KB(OTA), 128KB, 144KB, 160KB, 192KB, 256KB, 512KB(MAX)
Any	2MBytes	64KB, 128KB, 256KB(OTA), 512KB, 1MB(MAX)
Any	4MBytes	1MB, 2MB(OTA), 3MB(MAX)
Any	8MBytes	6MB(OTA), 7MB(MAX)
Any	16MBytes	14MB(OTA), 15MB(MAX)

Note: to use any of file system functions in the sketch, add the following include to the sketch:

```
//#include "FS.h" // SPIFFS is declared
#include "LittleFS.h" // LittleFS is declared
//#include "SDFS.h" // SDFS is declared
```

5.2 SPIFFS Deprecation Warning

SPIFFS is currently deprecated and may be removed in future releases of the core. Please consider moving your code to LittleFS. SPIFFS is not actively supported anymore by the upstream developer, while LittleFS is under active development, supports real directories, and is many times faster for most operations.

5.3 SPIFFS and LittleFS

There are two filesystems for utilizing the onboard flash on the ESP8266: SPIFFS and LittleFS.

SPIFFS is the original filesystem and is ideal for space and RAM constrained applications that utilize many small files and care about static and dynamic wear levelling and don't need true directory support. Filesystem overhead on the flash is minimal as well.

LittleFS is recently added and focuses on higher performance and directory support, but has higher filesystem and per-file overhead (4K minimum vs. SPIFFS' 256 byte minimum file allocation unit).

They share a compatible API but have incompatible on-flash implementations, so it is important to choose one or the other per project as attempting to mount a SPIFFS volume under LittleFS may result in a format operation and definitely will not preserve any files, and vice-versa.

The actual `File` and `Dir` objects returned from either filesystem behave in the same manner and documentation is applicable to both. To convert most applications from SPIFFS to LittleFS simply requires changing the `SPIFFS.begin()` to `LittleFS.begin()` and `SPIFFS.open()` to `LittleFS.open()` with the rest of the code remaining untouched.

5.4 SDFS and SD

FAT filesystems are supported on the ESP8266 using the old Arduino wrapper "SD.h" which wraps the "SDFS.h" filesystem transparently.

Any commands discussed below pertaining to SPIFFS or LittleFS are applicable to SD/SDFS.

For legacy applications, the classic SD filesystem may continue to be used, but for new applications, directly accessing the SDFS filesystem is recommended as it may expose additional functionality that the old Arduino SD filesystem didn't have.

Note that in earlier releases of the core, using SD and SPIFFS in the same sketch was complicated and required the use of `NO_FS_GLOBALS`. The current design makes SD, SDFS, SPIFFS, and LittleFS fully source compatible and so please remove any `NO_FS_GLOBALS` definitions in your projects when updgrading core versions.

5.5 SPIFFS file system limitations

The SPIFFS implementation for ESP8266 had to accommodate the constraints of the chip, among which its limited RAM. SPIFFS was selected because it is designed for small systems, but that comes at the cost of some simplifications and limitations.

First, behind the scenes, SPIFFS does not support directories, it just stores a "flat" list of files. But contrary to traditional filesystems, the slash character `'/'` is allowed in filenames, so the functions that deal with directory listing (e.g. `openDir("/website")`) basically just filter the filenames and keep the ones that start with the requested prefix (`/website/`). Practically speaking, that makes little difference though.

Second, there is a limit of 32 chars in total for filenames. One `'\0'` char is reserved for C string termination, so that leaves us with 31 usable characters.

Combined, that means it is advised to keep filenames short and not use deeply nested directories, as the full path of each file (including directories, `'/'` characters, base name, dot and extension) has to be 31 chars at a maximum. For example, the filename `/website/images/bird_thumbnail.jpg` is 34 chars and will cause some problems if used, for example in `exists()` or in case another file starts with the same first 31 characters.

Warning: That limit is easily reached and if ignored, problems might go unnoticed because no error message will appear at compilation nor runtime.

For more details on the internals of SPIFFS implementation, see the SPIFFS readme file.

5.6 LittleFS file system limitations

The LittleFS implementation for the ESP8266 supports filenames of up to 31 characters + terminating zero (i.e. `char filename[32]`), and as many subdirectories as space permits.

Filenames are assumed to be in the root directory if no initial "/" is present.

Opening files in subdirectories requires specifying the complete path to the file (i.e. `open("/sub/dir/file.txt");`). Subdirectories are automatically created when you attempt to create a file in a subdirectory, and when the last file in a subdirectory is removed the subdirectory itself is automatically deleted. This is because there was no `mkdir()` method in the existing SPIFFS filesystem.

Unlike SPIFFS, the actual file descriptors are allocated as requested by the application, so in low memory conditions you may not be able to open new files. Conversely, this also means that only file descriptors used will actually take space on the heap.

Because there are directories, the `openDir` method behaves differently than SPIFFS. Whereas SPIFFS will return files in "subdirectories" when you traverse a `Dir::next()` (because they really aren't subdirs but simply files with "/"s in their names), LittleFS will only return files in the specific subdirectory. This mimics the POSIX behavior for directory traversal most C programmers are used to.

5.7 Uploading files to file system

ESP8266FS is a tool which integrates into the Arduino IDE. It adds a menu item to *Tools* menu for uploading the contents of sketch data directory into ESP8266 flash file system.

Warning: Due to the move from the obsolete esptool-ck.exe to the supported esptool.py upload tool, upgraders from pre 2.5.1 will need to update the ESP8266FS tool referenced below to 0.5.0 or later. Prior versions will fail with a "esptool not found" error because they don't know how to use esptool.py.

- Download the tool: https://github.com/esp8266/arduino-esp8266fs-plugin/releases/download/0.5.0/ESP8266FS-0.5.0.zip
- In your Arduino sketchbook directory, create `tools` directory if it doesn't exist yet.
- Unpack the tool into `tools` directory (the path will look like `<home_dir>/Arduino/tools/ESP8266FS/tool/esp8266fs.jar`) If upgrading, overwrite the existing JAR file with the newer version.
- Restart Arduino IDE.
- Open a sketch (or create a new one and save it).
- Go to sketch directory (choose Sketch > Show Sketch Folder).
- Create a directory named `data` and any files you want in the file system there.
- Make sure you have selected a board, port, and closed Serial Monitor.
- If your board requires you to press a button (or other action) to enter bootload mode for flashing a sketch, do that now.
- Select Tools > ESP8266 Sketch Data Upload. This should start uploading the files into ESP8266 flash file system. When done, IDE status bar will display `SPIFFS Image Uploaded` message.

ESP8266LittleFS is the equivalent tool for LittleFS.

- Download the 2.6.0 or later version of the tool: https://github.com/earlephilhower/arduino-esp8266littlefs-plugin/releases
- Install as above
- To upload a LittleFS filesystem use Tools > ESP8266 LittleFS Data Upload

5.8 File system object (SPIFFS/LittleFS/SD/SDFS)

5.8.1 setConfig

```
SPIFFSConfig cfg;
cfg.setAutoFormat(false);
SPIFFS.setConfig(cfg);
```

This method allows you to configure the parameters of a filesystem before mounting. All filesystems have their own `*Config` (i.e. `SDFSConfig` or `SPIFFSConfig` with their custom set of options. All filesystems allow explicitly enabling/disabling formatting when mounts fail. If you do not call this `setConfig` method before perforing `begin()`, you will get the filesystem's default behavior and configuration. By default, SPIFFS will autoformat the filesystem if it cannot mount it, while SDFS will not.

5.8.2 begin

```
SPIFFS.begin()
or LittleFS.begin()
```

This method mounts file system. It must be called before any other FS APIs are used. Returns *true* if file system was mounted successfully, false otherwise. With no options it will format SPIFFS if it is unable to mount it on the first try.

Note that both methods will automatically format the filesystem if one is not detected. This means that if you attempt a `SPIFFS.begin()` on a LittleFS filesystem you will lose all data on that filesystem, and vice-versa.

5.8.3 end

```
SPIFFS.end()
or LittleFS.end()
```

This method unmounts the file system. Use this method before updating the file system using OTA.

5.8.4 format

```
SPIFFS.format()
or LittleFS.format()
```

Formats the file system. May be called either before or after calling `begin`. Returns *true* if formatting was successful.

5.8.5 open

```
SPIFFS.open(path, mode)
or LittleFS.open(path, mode)
```

Opens a file. `path` should be an absolute path starting with a slash (e.g. `/dir/filename.txt`). `mode` is a string specifying access mode. It can be one of "r", "w", "a", "r+", "w+", "a+". Meaning of these modes is the same as for `fopen` C function.

```
r       Open text file for reading.  The stream is positioned at the
        beginning of the file.

r+      Open for reading and writing.  The stream is positioned at the
        beginning of the file.

w       Truncate file to zero length or create text file for writing.
        The stream is positioned at the beginning of the file.

w+      Open for reading and writing.  The file is created if it does
        not exist, otherwise it is truncated.  The stream is
        positioned at the beginning of the file.

a       Open for appending (writing at end of file).  The file is
        created if it does not exist.  The stream is positioned at the
        end of the file.

a+      Open for reading and appending (writing at end of file).  The
        file is created if it does not exist.  The initial file
        position for reading is at the beginning of the file, but
        output is always appended to the end of the file.
```

Returns *File* object. To check whether the file was opened successfully, use the boolean operator.

```
File f = SPIFFS.open("/f.txt", "w");
if (!f) {
    Serial.println("file open failed");
}
```

5.8.6 exists

```
SPIFFS.exists(path)
or LittleFS.exists(path)
```

Returns *true* if a file with given path exists, *false* otherwise.

5.8.7 mkdir

```
LittleFS.mkdir(path)
```

Returns *true* if the directory creation succeeded, *false* otherwise.

5.8.8 rmdir

```
LittleFS.rmdir(path)
```

Returns *true* if the directory was successfully removed, *false* otherwise.

5.8.9 openDir

```
SPIFFS.openDir(path)
or LittleFS.openDir(path)
```

Opens a directory given its absolute path. Returns a *Dir* object. Please note the previous discussion on the difference in behavior between LittleFS and SPIFFS for this call.

5.8.10 remove

```
SPIFFS.remove(path)
or LittleFS.remove(path)
```

Deletes the file given its absolute path. Returns *true* if file was deleted successfully.

5.8.11 rename

```
SPIFFS.rename(pathFrom, pathTo)
or LittleFS.rename(pathFrom, pathTo)
```

Renames file from `pathFrom` to `pathTo`. Paths must be absolute. Returns *true* if file was renamed successfully.

5.8.12 gc

```
SPIFFS.gc()
```

Only implemented in SPIFFS. Performs a quick garbage collection operation on SPIFFS, possibly making writes perform faster/better in the future. On very full or very fragmented filesystems, using this call can avoid or reduce issues where SPIFFS reports free space but is unable to write additional data to a file. See *this discussion <https://github.com/esp8266/Arduino/pull/6340#discussion_r307042268>* for more info.

5.8.13 check

```
SPIFFS.begin();
SPIFFS.check();
```

Only implemented in SPIFFS. Performs an in-depth check of the filesystem metadata and correct what is repairable. Not normally needed, and not guaranteed to actually fix anything should there be corruption.

5.8.14 info

```
FSInfo fs_info;
SPIFFS.info(fs_info);
or LittleFS.info(fs_info);
```

Fills *FSInfo structure* with information about the file system. Returns `true` if successful, `false` otherwise.

5.9 Filesystem information structure

```
struct FSInfo {
    size_t totalBytes;
    size_t usedBytes;
    size_t blockSize;
    size_t pageSize;
    size_t maxOpenFiles;
    size_t maxPathLength;
};
```

This is the structure which may be filled using FS::info method. - `totalBytes` — total size of useful data on the file system - `usedBytes` — number of bytes used by files - `blockSize` — filesystem block size - `pageSize` — filesystem logical page size - `maxOpenFiles` — max number of files which may be open simultaneously - `maxPathLength` — max file name length (including one byte for zero termination)

5.9.1 info64

```
FSInfo64 fsinfo;
SD.info(fsinfo);
or LittleFS(fsinfo);
```

Performs the same operation as `info` but allows for reporting greater than 4GB for filesystem size/used/etc. Should be used with the SD and SDFS filesystems since most SD cards today are greater than 4GB in size.

5.9.2 setTimeCallback(time_t (*cb)(void))

```
time_t myTimeCallback() {
    return 1455451200; // UNIX timestamp
}
void setup () {
    LittleFS.setTimeCallback(myTimeCallback);
    ...
    // Any files will now be made with Pris' incept date
}
```

The SD, SDFS, and LittleFS filesystems support a file timestamp, updated when the file is opened for writing. By default, the ESP8266 will use the internal time returned from `time(NULL)` (i.e. local time, not UTC, to conform to the existing FAT filesystem), but this can be overridden to GMT or any other standard you'd like by using `setTimeCallback()`. If your app sets the system time using NTP before file operations, then you should not need to use this function. However, if you need to set a specific time for a file, or the system clock isn't correct and you need to read the time from an external RTC or use a fixed time, this call allows you do to so.

In general use, with a functioning `time()` call, user applications should not need to use this function.

5.10 Directory object (Dir)

The purpose of *Dir* object is to iterate over files inside a directory. It provides multiple access methods.

The following example shows how it should be used:

```
Dir dir = SPIFFS.openDir("/data");
// or Dir dir = LittleFS.openDir("/data");
while (dir.next()) {
    Serial.print(dir.fileName());
    if(dir.fileSize()) {
        File f = dir.openFile("r");
        Serial.println(f.size());
    }
}
```

5.10.1 next

Returns true while there are files in the directory to iterate over. It must be called before calling `fileName()`, `fileSize()`, and `openFile()` functions.

5.10.2 fileName

Returns the name of the current file pointed to by the internal iterator.

5.10.3 fileSize

Returns the size of the current file pointed to by the internal iterator.

5.10.4 fileTime

Returns the time_t write time of the current file pointed to by the internal iterator.

5.10.5 fileCreationTime

Returns the time_t creation time of the current file pointed to by the internal iterator.

5.10.6 isFile

Returns *true* if the current file pointed to by the internal iterator is a File.

5.10.7 isDirectory

Returns *true* if the current file pointed to by the internal iterator is a Directory.

5.10.8 openFile

This method takes *mode* argument which has the same meaning as for `SPIFFS/LittleFS.open()` function.

5.10.9 rewind

Resets the internal pointer to the start of the directory.

5.10.10 setTimeCallback(time_t (*cb)(void))

Sets the time callback for any files accessed from this Dir object via openNextFile. Note that the SD and SDFS filesystems only support a filesystem-wide callback and calls to `Dir::setTimeCallback` may produce unexpected behavior.

5.11 File object

`SPIFFS/LittleFS.open()` and `dir.openFile()` functions return a *File* object. This object supports all the functions of *Stream*, so you can use `readBytes`, `findUntil`, `parseInt`, `println`, and all other *Stream* methods.

There are also some functions which are specific to *File* object.

5.11.1 seek

```
file.seek(offset, mode)
```

This function behaves like `fseek` C function. Depending on the value of `mode`, it moves current position in a file as follows:

- if `mode` is `SeekSet`, position is set to `offset` bytes from the beginning.
- if `mode` is `SeekCur`, current position is moved by `offset` bytes.
- if `mode` is `SeekEnd`, position is set to `offset` bytes from the end of the file.

Returns *true* if position was set successfully.

5.11.2 position

```
file.position()
```

Returns the current position inside the file, in bytes.

5.11.3 size

```
file.size()
```

Returns file size, in bytes.

5.11.4 name

```
String name = file.name();
```

Returns short (no-path) file name, as `const char*`. Convert it to *String* for storage.

5.11.5 fullName

```
// Filesystem:
//   testdir/
//           file1
Dir d = LittleFS.openDir("testdir/");
File f = d.openFile("r");
// f.name() == "file1", f.fullName() == "testdir/file1"
```

Returns the full path file name as a `const char*`.

5.11.6 getLastWrite

Returns the file last write time, and only valid for files opened in read-only mode. If a file is opened for writing, the returned time may be indeterminate.

5.11.7 getCreationTime

Returns the file creation time, if available.

5.11.8 isFile

```
bool amIAFile = file.isFile();
```

Returns *true* if this File points to a real file.

5.11.9 isDirectory

```
bool amIADir = file.isDir();
```

Returns *true* if this File points to a directory (used for emulation of the SD.* interfaces with the `openNextFile` method).

5.11.10 close

```
file.close()
```

Close the file. No other operations should be performed on *File* object after `close` function was called.

5.11.11 openNextFile (compatibiity method, not recommended for new code)

```
File root = LittleFS.open("/");
File file1 = root.openNextFile();
File files = root.openNextFile();
```

Opens the next file in the directory pointed to by the File. Only valid when `File.isDirectory() == true`.

5.11.12 rewindDirectory (compatibiity method, not recommended for new code)

```
File root = LittleFS.open("/");
File file1 = root.openNextFile();
file1.close();
root.rewindDirectory();
file1 = root.openNextFile(); // Opens first file in dir again
```

Resets the `openNextFile` pointer to the top of the directory. Only valid when `File.isDirectory() == true`.

5.11.13 setTimeCallback(time_t (*cb)(void))

Sets the time callback for this specific file. Note that the SD and SDFS filesystems only support a filesystem-wide callback and calls to `Dir::setTimeCallback` may produce unexpected behavior.

CHAPTER

SIX

ESP8266WIFI LIBRARY

ESP8266 is all about Wi-Fi. If you are eager to connect your new ESP8266 module to a Wi-Fi network to start sending and receiving data, this is a good place to start. If you are looking for more in depth details of how to program specific Wi-Fi networking functionality, you are also in the right place.

6.1 Introduction

The Wi-Fi library for ESP8266 has been developed based on ESP8266 SDK, using the naming conventions and overall functionality philosophy of the Arduino WiFi library. Over time, the wealth of Wi-Fi features ported from ESP8266 SDK to esp8266 / Arduino outgrew Arduino WiFi library and it became apparent that we would need to provide separate documentation on what is new and extra.

This documentation will walk you through several classes, methods and properties of the ESP8266WiFi library. If you are new to C++ and Arduino, don't worry. We will start from general concepts and then move to detailed description of members of each particular class including usage examples.

The scope of functionality offered by the ESP8266WiFi library is quite extensive and therefore this description has been broken up into separate documents marked with :arrow_right:.

6.1.1 Quick Start

Hopefully, you are already familiar how to load the Blink.ino sketch to an ESP8266 module and get the LED blinking. If not, please use this tutorial by Adafruit or another great tutorial developed by Sparkfun.

To hook up the ESP module to Wi-Fi (like hooking up a mobile phone to a hot spot), you need only a couple of lines of code:

```
#include <ESP8266WiFi.h>

void setup()
{
  Serial.begin(115200);
  Serial.println();

  WiFi.begin("network-name", "pass-to-network");

  Serial.print("Connecting");
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
```

(continues on next page)

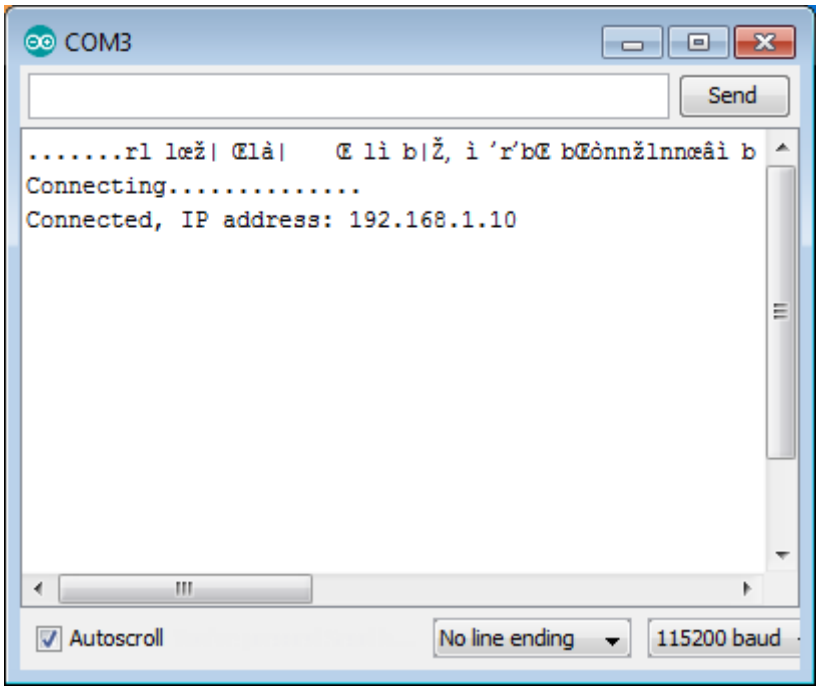
(continued from previous page)

```
    Serial.print(".");
  }
  Serial.println();

  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

In the line `WiFi.begin("network-name", "pass-to-network")` replace `network-name` and `pass-to-network` with the name and password of the Wi-Fi network you would like to connect to. Then, upload this sketch to ESP module and open the serial monitor. You should see something like:



How does it work? In the first line of the sketch, `#include <ESP8266WiFi.h>` we are including the ESP8266WiFi library. This library provides ESP8266 specific Wi-Fi routines that we are calling to connect to the network.

The actual connection to Wi-Fi is initialized by calling:

```
WiFi.begin("network-name", "pass-to-network");
```

The connection process can take couple of seconds and we are checking for whether this has completed in the following loop:

```
while (WiFi.status() != WL_CONNECTED)
{
  delay(500);
  Serial.print(".");
}
```

The `while()` loop will keep looping as long as `WiFi.status()` is other than `WL_CONNECTED`. The loop will exit only if the status changes to `WL_CONNECTED`.

The last line will then print out the IP address assigned to the ESP module by DHCP:

```
Serial.println(WiFi.localIP());
```

If you don't see the last line but just more and more dots `.........`, then likely name or password to the Wi-Fi network is entered incorrectly in the sketch. Verify the name and password by connecting from scratch to this Wi-Fi network with a PC or a mobile phone.

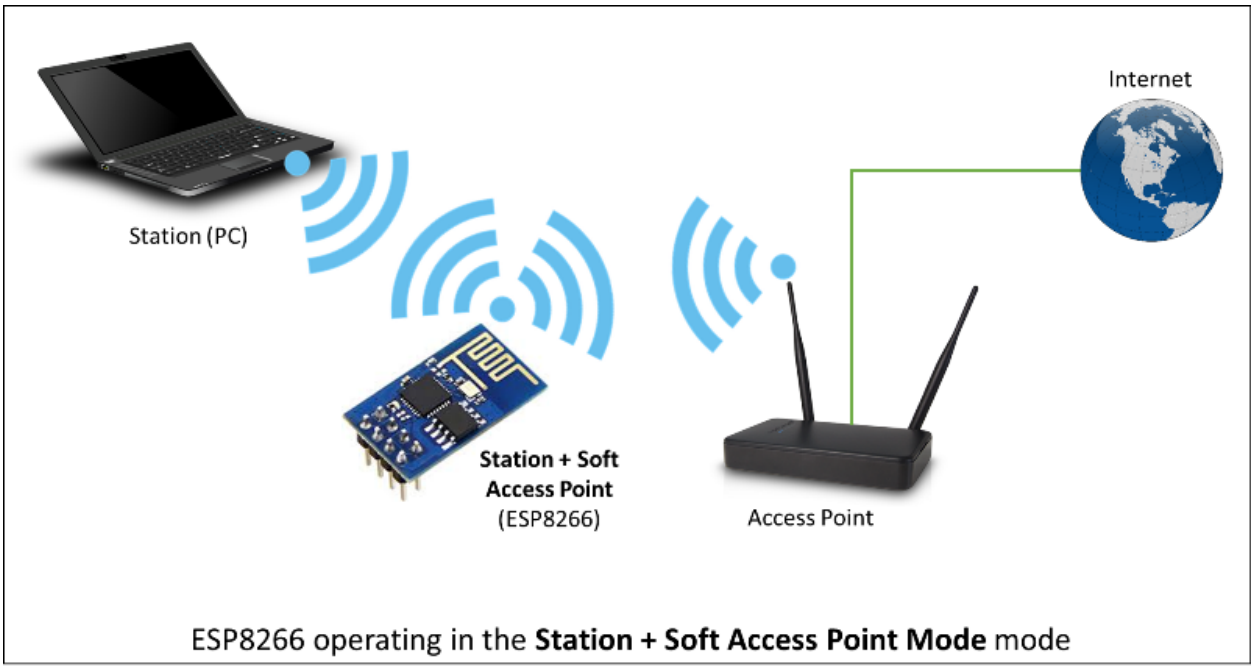
Note: if connection is established, and then lost for some reason, ESP will automatically reconnect to the last used access point once it is again back on-line. This will be done automatically by Wi-Fi library, without any user intervention.

That's all you need to connect ESP8266 to Wi-Fi. In the following chapters we will explain what cool things can be done by the ESP once it's connected.

6.1.2 Who is Who

Devices that connect to Wi-Fi networks are called stations (STA). Connection to Wi-Fi is provided by an access point (AP), that acts as a hub for one or more stations. The access point on the other end is connected to a wired network. An access point is usually integrated with a router to provide access from a Wi-Fi network to the internet. Each access point is recognized by a SSID (**S**ervice **S**et **ID**entifier), that essentially is the name of network you select when connecting a device (station) to the Wi-Fi.

ESP8266 modules can operate as a station, so we can connect it to the Wi-Fi network. It can also operate as a soft access point (soft-AP), to establish its own Wi-Fi network. When the ESP8266 module is operating as a soft access point, we can connect other stations to the ESP module. ESP8266 is also able to operate as both a station and a soft access point mode. This provides the possibility of building e.g. mesh networks.



ESP8266 operating in the **Station + Soft Access Point Mode** mode

The ESP8266WiFi library provides a wide collection of C++ methods (functions) and properties to configure and operate an ESP8266 module in station and / or soft access point mode. They are described in the following chapters.

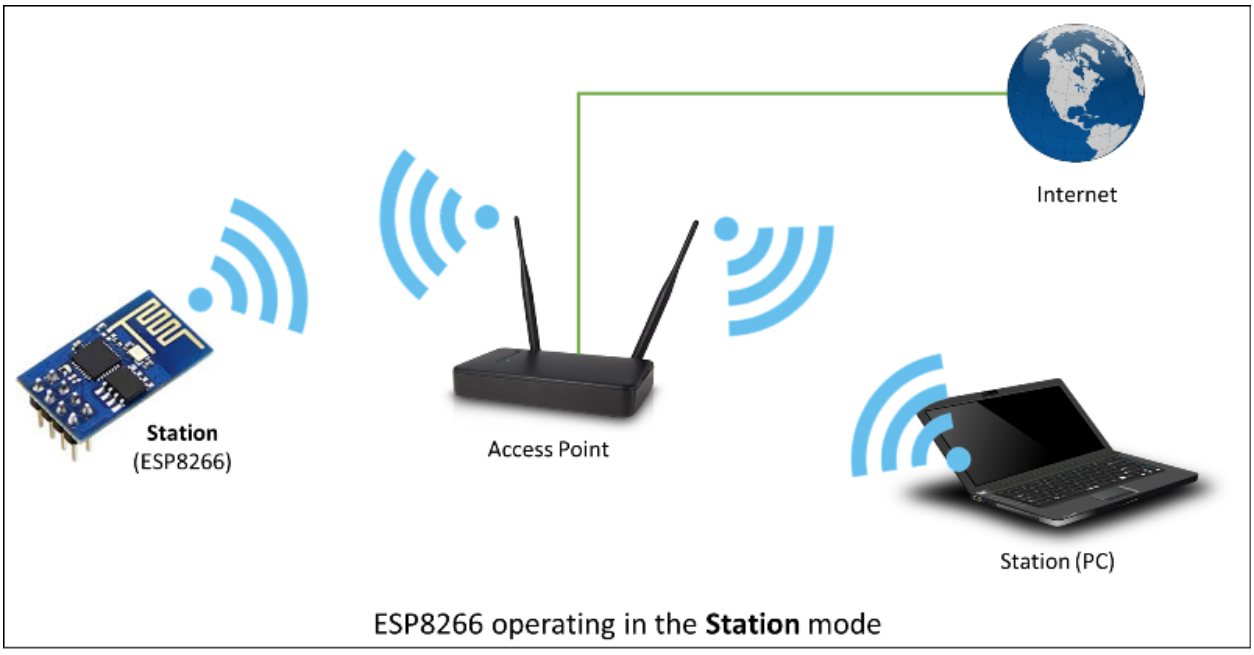
6.2 Class Description

The ESP8266WiFi library is broken up into several classes. In most of cases, when writing the code, the user is not concerned with this classification. We are using it to break up description of this library into more manageable pieces.

Chapters below describe all function calls (methods and properties in C++ terms) listed in particular classes of ESP8266WiFi. The description is illustrated with application examples and code snippets to show how to use functions in practice. This information is broken up into the following documents.

6.2.1 Station

Station (STA) mode is used to get the ESP module connected to a Wi-Fi network established by an access point.



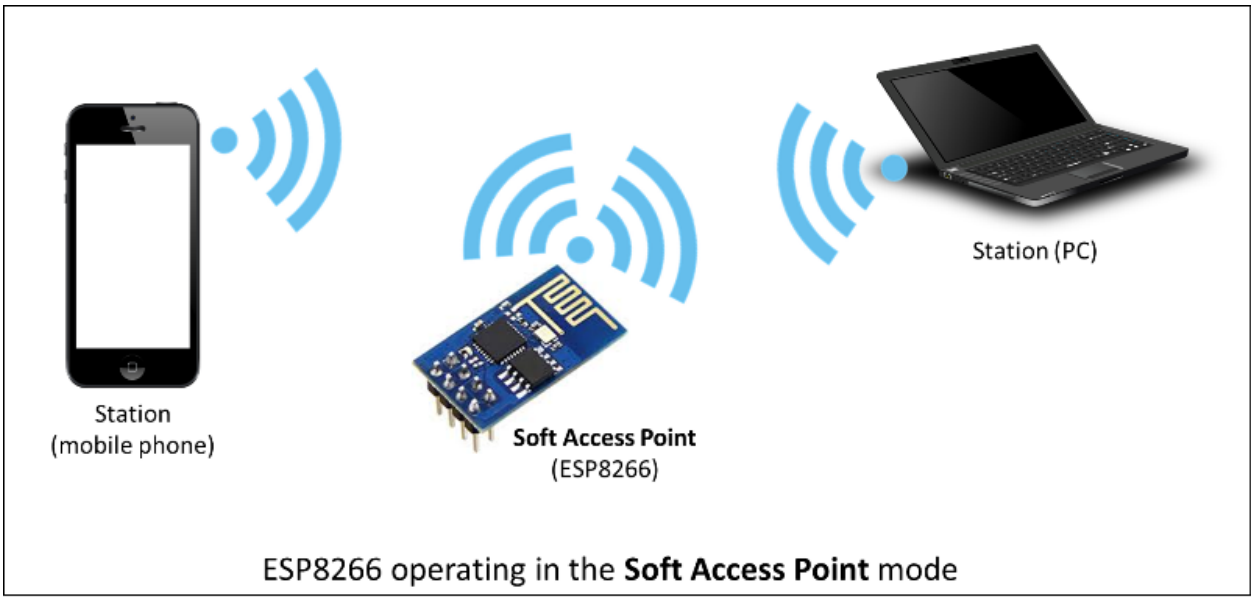
Station class has several features to facilitate the management of a Wi-Fi connection. In case the connection is lost, the ESP8266 will automatically reconnect to the last used access point, once it is available again. The same happens on module reboot. This is possible since ESP saves the credentials to the last used access point in flash (non-volatile) memory. Using the saved data ESP will also reconnect if sketch has been changed but code does not alter the Wi-Fi mode or credentials.

Station Class documentation

Check out separate section with examples.

6.2.2 Soft Access Point

An access point (AP) is a device that provides access to a Wi-Fi network to other devices (stations) and connects them to a wired network. The ESP8266 can provide similar functionality, except it does not have interface to a wired network. Such mode of operation is called soft access point (soft-AP). The maximum number of stations that can simultaneously be connected to the soft-AP can be set from 0 to 8, but defaults to 4.



The soft-AP mode is often used and an intermediate step before connecting ESP to a Wi-Fi in a station mode. This is when SSID and password to such network is not known upfront. ESP first boots in soft-AP mode, so we can connect to it using a laptop or a mobile phone. Then we are able to provide credentials to the target network. Then, the ESP is switched to the station mode and can connect to the target Wi-Fi.

Another handy application of soft-AP mode is to set up mesh networks. The ESP can operate in both soft-AP and Station mode so it can act as a node of a mesh network.

Soft Access Point Class documentation

Check out the separate section with examples.

6.2.3 Scan

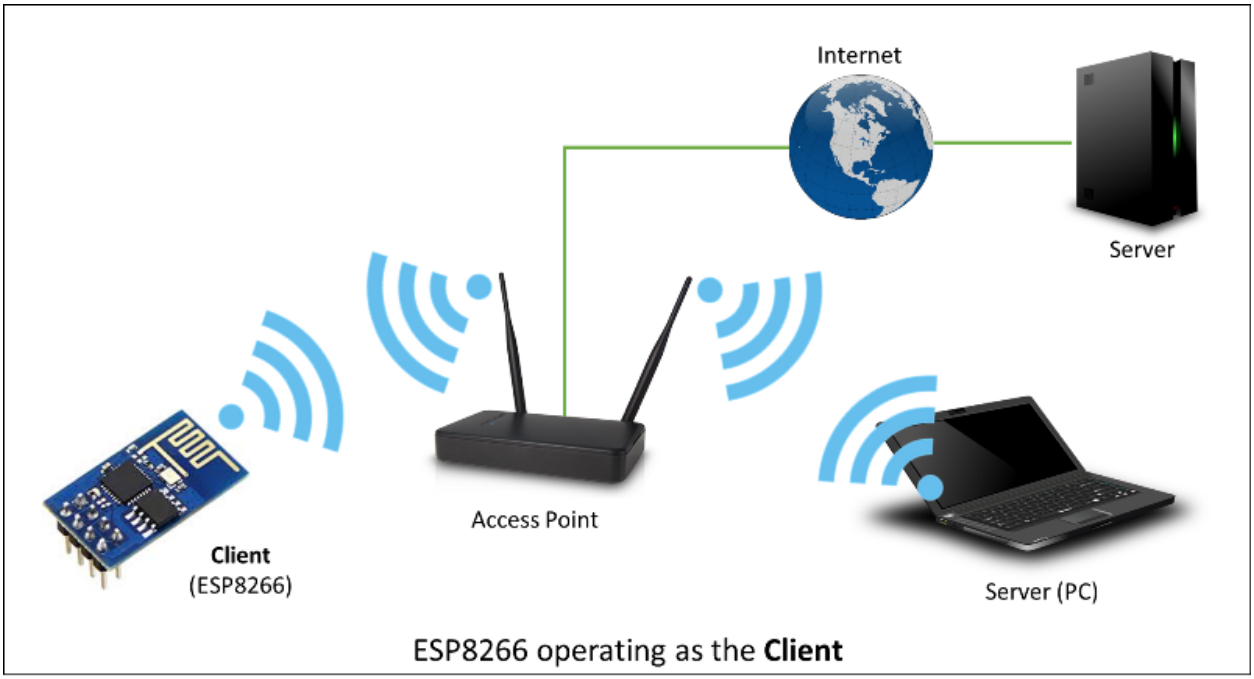
To connect a mobile phone to a hot spot, you typically open Wi-Fi settings app, list available networks and pick the hot spot you need. Then enter a password (or not) and you are in. You can do the same with the ESP. Functionality of scanning for, and listing of available networks in range is implemented by the Scan Class.

Scan Class documentation

Check out the separate section with examples.

6.2.4 Client

The Client class creates clients that can access services provided by servers in order to send, receive and process data.



Check out the separate section with list of functions

6.2.5 WiFi Multi

ESP8266WiFiMulti.h can be used to connect to a WiFi network with strongest WiFi signal (RSSI). This requires registering one or more access points with SSID and password. It automatically switches to another WiFi network when the WiFi connection is lost.

Example:

```
#include <ESP8266WiFiMulti.h>

ESP8266WiFiMulti wifiMulti;

// WiFi connect timeout per AP. Increase when connecting takes longer.
const uint32_t connectTimeoutMs = 5000;

void setup()
{
  // Set in station mode
  WiFi.mode(WIFI_STA);

  // Register multi WiFi networks
  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");
}
```

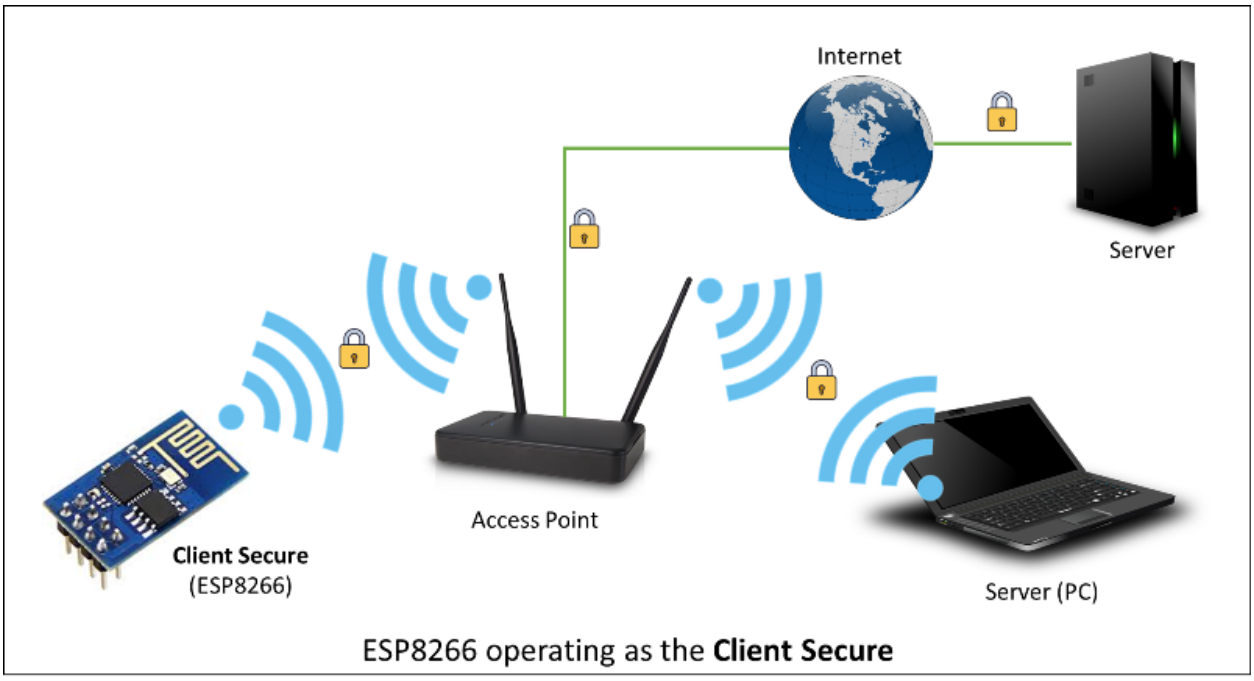
(continues on next page)

(continued from previous page)

```
void loop()
{
  // Maintain WiFi connection
  if (wifiMulti.run(connectTimeoutMs) == WL_CONNECTED) {
      ...
  }
}
```

6.2.6 BearSSL Client Secure and Server Secure

BearSSL::WiFiClientSecure and *BearSSL::WiFiServerSecure* are extensions of the standard *Client* and *Server* classes where connection and data exchange with servers and clients using secure protocol. It supports TLS 1.2 using a wide variety of modern ciphers, hashes, and key types.



Secure clients and servers require significant amounts of additional memory and processing to enable their cryptographic algorithms. In general, only a single secure client or server connection at a time can be processed given the little RAM present on the ESP8266, but there are methods of reducing this RAM requirement detailed in the relevant sections.

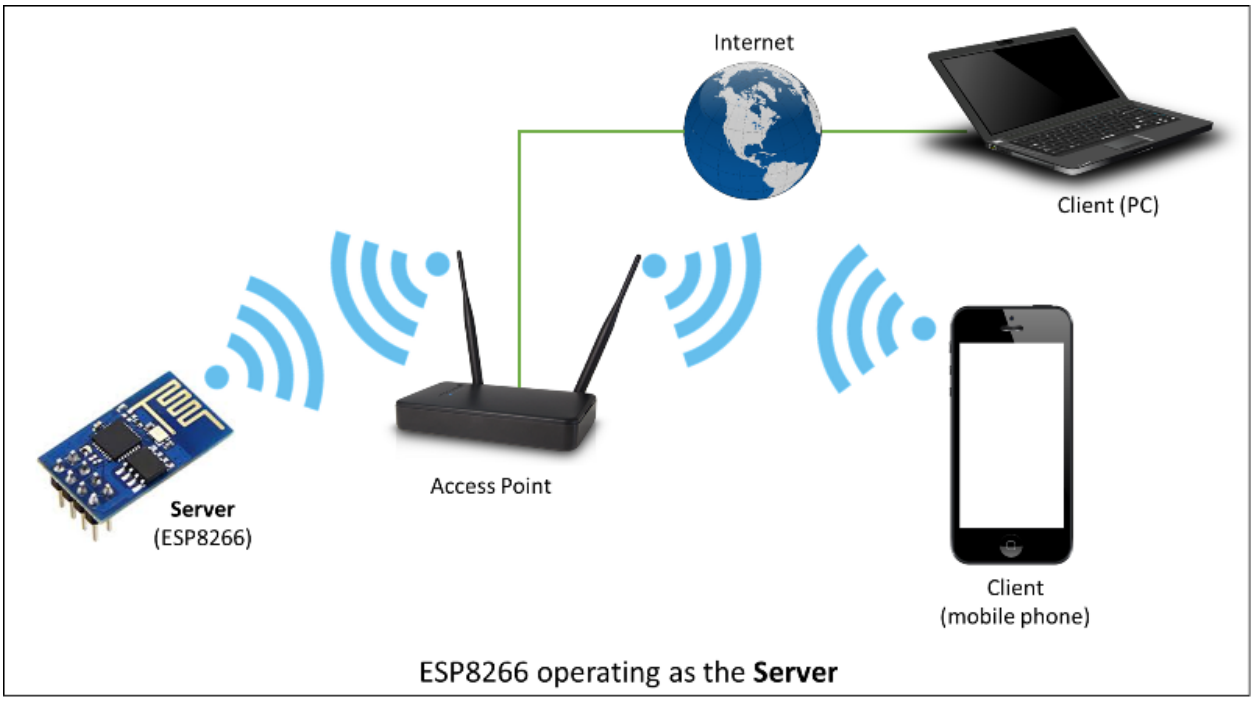
BearSSL::WiFiClientSecure contains more information on using and configuring TLS connections.

BearSSL::WiFiServerSecure discusses the TLS server mode available. Please read and understand the BearSSL::WiFiClientSecure first as the server uses most of the same concepts.

Check out the separate section with examples .

6.2.7 Server

The Server Class creates servers that provide functionality to other programs or devices, called clients.



Clients connect to sever to send and receive data and access provided functionality.

Check out separate section with examples / list of functions.

6.2.8 UDP

The UDP Class enables the User Datagram Protocol (UDP) messages to be sent and received. The UDP uses a simple "fire and forget" transmission model with no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

Check out separate section with examples / list of functions.

6.2.9 Generic

There are several functions offered by ESP8266's SDK and not present in Arduino WiFi library. If such function does not fit into one of classes discussed above, it will likely be in Generic Class. Among them is handler to manage Wi-Fi events like connection, disconnection or obtaining an IP, Wi-Fi mode changes, functions to manage module sleep mode, hostname to an IP address resolution, etc.

Check out separate section with examples / list of functions.

6.3 Diagnostics

There are several techniques available to diagnose and troubleshoot issues with getting connected to Wi-Fi and keeping connection alive.

6.3.1 Check Return Codes

Almost each function described in chapters above returns some diagnostic information.

Such diagnostic may be provided as a simple `boolean` type `true` or `false` to indicate operation result. You may check this result as described in examples, for instance:

```
Serial.printf("Wi-Fi mode set to WIFI_STA %s\n", WiFi.mode(WIFI_STA) ? "" : "Failed!");
```

Some functions provide more than just a binary status information. A good example is `WiFi.status()`.

```
Serial.printf("Connection status: %d\n", WiFi.status());
```

This function returns following codes to describe what is going on with Wi-Fi connection:

- 0 : `WL_IDLE_STATUS` when Wi-Fi is in process of changing between statuses
- 1 : `WL_NO_SSID_AVAIL`in case configured SSID cannot be reached
- 3 : `WL_CONNECTED` after successful connection is established
- 4 : `WL_CONNECT_FAILED` if connection failed
- 6 : `WL_CONNECT_WRONG_PASSWORD` if password is incorrect
- 7 : `WL_DISCONNECTED` if module is not configured in station mode

It is a good practice to display and check information returned by functions. Application development and troubleshooting will be easier with that.

6.3.2 Use printDiag

There is a specific function available to print out key Wi-Fi diagnostic information:

```
WiFi.printDiag(Serial);
```

A sample output of this function looks as follows:

```
Mode: STA+AP
PHY mode: N
Channel: 11
AP id: 0
Status: 5
Auto connect: 1
SSID (10): sensor-net
Passphrase (12): 123!$#0&*esP
BSSID set: 0
```

Use this function to provide snapshot of Wi-Fi status in these parts of application code, that you suspect may be failing.

6.3.3 Enable Wi-Fi Diagnostic

By default the diagnostic output from Wi-Fi libraries is disabled when you call `Serial.begin`. To enable debug output again, call `Serial.setDebugOutput(true)`. To redirect debug output to `Serial1` instead, call `Serial1.setDebugOutput(true)`. For additional details regarding diagnostics using serial ports please refer to *the documentation*.

Below is an example of output for sample sketch discussed in *Quick Start* above with `Serial.setDebugOutput(true)`:

```
Connectingscandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)
add 0
aid 1
cnt

connected with sensor-net, channel 6
dhcp client start...
chg_B1:-40
...ip:192.168.1.10,mask:255.255.255.0,gw:192.168.1.9
.
Connected, IP address: 192.168.1.10
```

The same sketch without `Serial.setDebugOutput(true)` will print out only the following:

```
Connecting....
Connected, IP address: 192.168.1.10
```

6.3.4 Enable Debugging in IDE

Arduino IDE provides convenient method to *enable debugging* for specific libraries.

6.4 What's Inside?

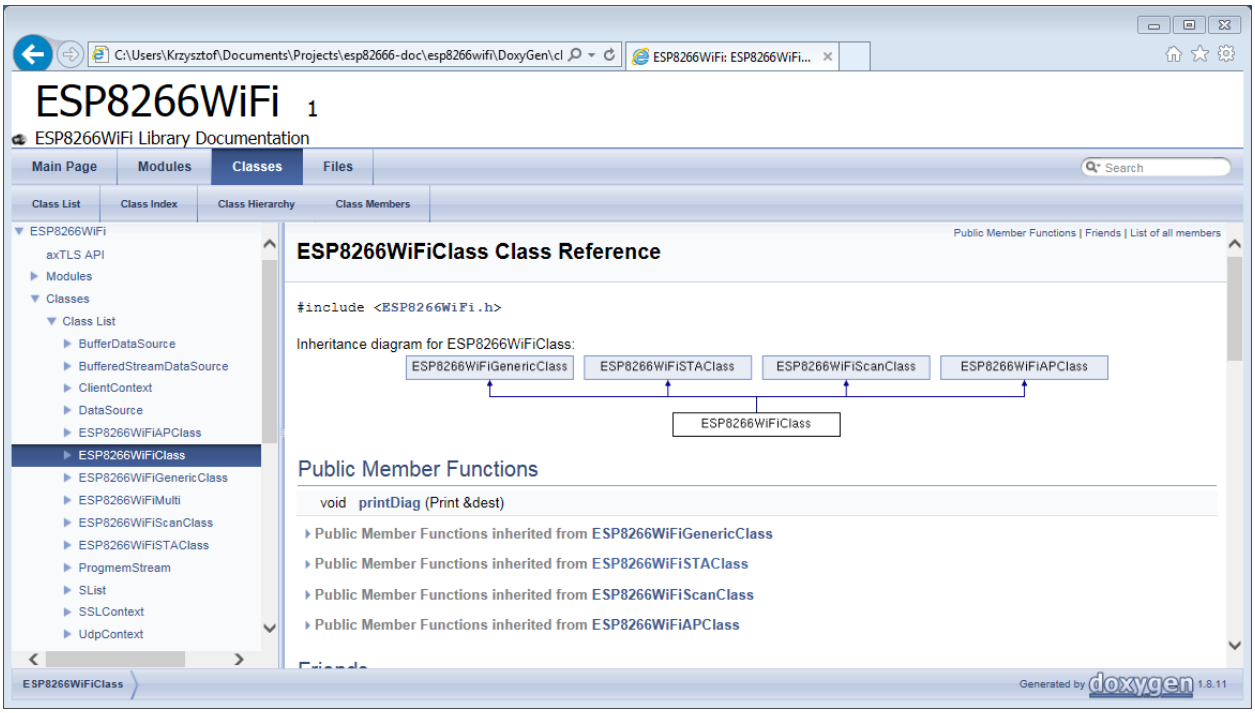
If you like to analyze in detail what is inside of the ESP8266WiFi library, go directly to the ESP8266WiFi folder of esp8266 / Arduino repository on the GitHub.

To make the analysis easier, rather than looking into individual header or source files, use one of free tools to automatically generate documentation. The class index in chapter *Class Description* above has been prepared in no time using great Doxygen, that is the de facto standard tool for generating documentation from annotated C++ sources.

The tool crawls through all header and source files collecting information from formatted comment blocks. If developer of particular class annotated the code, you will see it like in examples below.

If code is not annotated, you will still see the function prototype including types of arguments, and can use provided links to jump straight to the source code to check it out on your own. Doxygen provides really excellent navigation between members of library.

Several classes of ESP8266WiFi are not annotated. When preparing this document, Doxygen has been tremendous help to quickly navigate through almost 30 files that make this library.




```
wl_status_t ESP8266WiFiSTAClass::begin ( const char *   ssid,
                                         const char *   passphrase = NULL,
                                         int32_t        channel = 0,
                                         const uint8_t * bssid = NULL,
                                         bool           connect = true
                                       )
```

Start Wifi connection if passphrase is set the most secure supported mode will be automatically selected

Parameters

ssid	const char* Pointer to the SSID string.
passphrase	const char * Optional. Passphrase. Valid characters in a passphrase must be between ASCII 32-126 (decimal).
bssid	uint8_t[6] Optional. BSSID / MAC of AP
channel	Optional. Channel of AP
connect	Optional. call connect

Returns

Definition at line 97 of file ESP8266WiFiSTA.cpp.

bool ESP8266WiFiSTAClass::hostname (char * aHostname)

Set ESP8266 station DHCP hostname

Parameters

aHostname max length:32

Returns

ok

Definition at line 422 of file ESP8266WiFiSTA.cpp.

uint8_t WiFiUDP::begin (uint16_t port) virtual

Definition at line 77 of file WiFiUdp.cpp.

CHAPTER

SEVEN

OTA UPDATES

7.1 Introduction

OTA (Over the Air) update is the process of uploading firmware to an ESP module using a Wi-Fi connection rather than a serial port. Such functionality becomes extremely useful in case of limited or no physical access to the module.

OTA may be done using:

- *Arduino IDE*
- *Web Browser*
- *HTTP Server*

The Arduino IDE option is intended primarily for the software development phase. The other two options would be more useful after deployment, to provide the module with application updates either manually with a web browser, or automatically using an HTTP server.

In any case, the first firmware upload has to be done over a serial port. If the OTA routines are correctly implemented in the sketch, then all subsequent uploads may be done over the air.

By default, there is no imposed security for the OTA process. It is up to the developer to ensure that updates are allowed only from legitimate / trusted sources. Once the update is complete, the module restarts, and the new code is executed. The developer should ensure that the application running on the module is shut down and restarted in a safe manner. Chapters below provide additional information regarding security and safety of OTA updates.

7.1.1 Security Disclaimer

No guarantees as to the level of security provided for your application by the following methods is implied. Please refer to the GNU LGPL license associated for this project for full disclaimers. If you do find security weaknesses, please don't hesitate to contact the maintainers or supply pull requests with fixes. The MD5 verification and password protection schemes are already known to supply a very weak level of security.

7.1.2 Basic Security

The module has to be exposed wirelessly to get it updated with a new sketch. That poses a risk of the module being violently hacked and programmed with some other code. To reduce the likelihood of being hacked, consider protecting your uploads with a password, selecting certain OTA port, etc.

Check functionality provided with the ArduinoOTA library that may improve security:

```
void setPort(uint16_t port);
void setHostname(const char* hostname);
void setPassword(const char* password);
```

Certain basic protection is already built in and does not require any additional coding by the developer. ArduinoOTA and espota.py use Digest-MD5 to authenticate uploads. Integrity of transferred data is verified on the ESP side using MD5 checksum.

Make your own risk analysis and, depending on the application, decide what library functions to implement. If required, consider implementation of other means of protection from being hacked, like exposing modules for uploads only according to a specific schedule, triggering OTA only when the user presses a dedicated "Update" button wired to the ESP, etc.

7.1.3 Advanced Security - Signed Updates

While the above password-based security will dissuade casual hacking attempts, it is not highly secure. For applications where a higher level of security is needed, cryptographically signed OTA updates can be required. This uses SHA256 hashing in place of MD5 (which is known to be cryptographically broken) and RSA-2048 bit level public-key encryption to guarantee that only the holder of a cryptographic private key can produce signed updates accepted by the OTA update mechanisms.

Signed updates are updates whose compiled binaries are signed with a private key (held by the developer) and verified with a public key (stored in the application and available for all to see). The signing process computes a hash of the binary code, encrypts the hash with the developer's private key, and appends this encrypted hash (also called a signature) to the binary that is uploaded (via OTA, web, or HTTP server). If the code is modified or replaced in any way by anyone except the holder of the developer's private key, the signature will not match and the ESP8266 will reject the upload.

Cryptographic signing only protects against tampering with binaries delivered via OTA. If someone has physical access, they will always be able to flash the device over the serial port. Signing also does not encrypt anything but the hash (so that it can't be modified), so this does not protect code inside the device: if a user has physical access they can read out your program.

Securing your private key is paramount. The same private/public key pair that was used with the original upload must also be used to sign later binaries. Loss of the private key associated with a binary means that you will not be able to OTA-update any of your devices in the field. Alternatively, if someone else copies the private key, then they will be able to use it to sign binaries which will be accepted by the ESP.

Signed Binary Format

The format of a signed binary is compatible with the standard binary format, and can be uploaded to a non-signed ESP8266 via serial or OTA without any conditions. Note, however, that once an unsigned OTA app is overwritten by this signed version, further updates will require signing.

As shown below, the signed hash is appended to the unsigned binary, followed by the total length of the signed hash (i.e., if the signed hash was 64 bytes, then this uint32 data segment will contain 64). This format allows for extensibility (such as adding a CA-based validation scheme allowing multiple signing keys all based on a trust anchor). Pull requests are always welcome. (currently it uses SHA256 with RSASSA-PKCS1-V1_5-SIGN signature scheme from RSA PKCS #1 v1.5)

```
NORMAL-BINARY <SIGNATURE> <uint32 LENGTH-OF-SIGNATURE>
```

Signed Binary Prerequisites

OpenSSL is required to run the standard signing steps, and should be available on any UNIX-like or Windows system. As usual, the latest stable version of OpenSSL is recommended.

Signing requires the generation of an RSA-2048 key (other bit lengths are supported as well, but 2048 is a good selection today) using any appropriate tool. The following shell commands will generate a new public/private key pair. Run them in the sketch directory:

```
openssl genrsa -out private.key 2048
openssl rsa -in private.key -outform PEM -pubout -out public.key
```

Automatic Signing – Only available on Linux and Mac

The simplest way of implementing signing is to use the automatic mode, which presently is only possible on Linux and Mac due to some of the tools not being available for Windows. This mode uses the IDE to configure the source code to enable sigining verification with a given public key, and signs binaries as part of the standard build process using a given public key.

To enable this mode, just include *private.key* and *public.key* in the sketch *.ino* directory. The IDE will call a helper script (*tools/signing.py*) before the build begins to create a header to enable key validation using the given public key, and to actually do the signing after the build process, generating a *sketch.bin.signed* file. When OTA is enabled (ArduinoOTA, Web, or HTTP), the binary will automatically only accept signed updates.

When the signing process starts, the message:

```
Enabling binary signing
```

will appear in the IDE window before a compile is launched. At the completion of the build, the signed binary file well be displayed in the IDE build window as:

```
Signed binary: /full/path/to/sketch.bin.signed
```

If you receive either of the following messages in the IDE window, the signing was not completed and you will need to verify the *public.key* and *private.key*:

```
Not enabling binary signing
... or ...
Not signing the generated binary
```

Manual Signing of Binaries

Users may also manually sign executables and require the OTA process to verify their signature. In the main code, before enabling any update methods, add the following declarations and function call:

```
<in globals>
BearSSL::PublicKey signPubKey( ... key contents ... );
BearSSL::HashSHA256 hash;
BearSSL::SigningVerifier sign( &signPubKey );
...
<in setup()>
Update.installSignature( &hash, &sign );
```

The above snippet creates a BearSSL public key and a SHA256 hash verifier, and tells the Update object to use them to validate any updates it receives from any method.

Compile the sketch normally and, once a *.bin* file is available, sign it using the signer script:

```
<ESP8266ArduinoPath>/tools/signing.py --mode sign --privatekey <path-to-private.key> --
↪bin <path-to-unsigned-bin> --out <path-to-signed-binary>
```

Old And New Signature Formats

Up to version 2.5.2 of the core, the format of signatures was a little different. An additional signed binary with the extension legacy_sig is created. This file contains a signature in the old format and can be uploaded OTA to a device that checks for the old signature format.

To create a legacy signature, call the signing script with –legacy:

```
<ESP8266ArduinoPath>/tools/signing.py --mode sign --privatekey <path-to-private.key> --
↪bin <path-to-unsigned-bin> --out <path-to-signed-binary> --legacy <path-to-legacy-file>
```

7.2 Compression

The eboot bootloader incorporates a GZIP decompressor, built for very low code requirements. For applications, this optional decompression is completely transparent. For uploading compressed filesystems, the application must be built with *ATOMIC_FS_UPDATE* defined because, otherwise, eboot will not be involved in writing the filesystem.

No changes to the application are required. The *Updater* class and *eboot* bootloader (which performs actual application overwriting on update) automatically search for the *gzip* header in the uploaded binary, and if found, handle it.

Compress an application *.bin* file or filesystem package using any *gzip* available, at any desired compression level (*gzip -9* is recommended because it provides the maximum compression and uncompresses as fast as any other compressino level). For example:

```
gzip -9 sketch.bin  # Maximum compression, output sketch.bin.gz
<Upload the resultant sketch.bin.gz>
```

If signing is desired, sign the gzip compressed file *after* compression.

```
gzip -9 sketch.bin
<ESP8266ArduinoPath>/tools/signing.py --mode sign --privatekey <path-to-private.key> --
↪bin sketch.bin.gz --out sketch.bin.gz.signed
```

7.2.1 Updating apps in the field to support compression

If you have applications deployed in the field and wish to update them to support compressed OTA uploads, you will need to first recompile the application, then _upload the uncompressed *.bin* file once. Attempting to upload a *gzip* compressed binary to a legacy app will result in the Updater rejecting the upload as it does not understand the *gzip* format. After this initial upload, which will include the new bootloader and *Updater* class with compression support, compressed updates can then be used.

7.2.2 Safety

The OTA process consumes some of the ESP's resources and bandwidth during upload. Then, the module is restarted and a new sketch executed. Analyse and test how this affects the functionality of the existing and new sketches.

If the ESP is in a remote location and controlling some equipment, you should devote additional attention to what happens if operation of this equipment is suddenly interrupted by the update process. Therefore, decide how to put this equipment into a safe state before starting the update. For instance, your module may be controlling a garden watering system in a sequence. If this sequence is not properly shut down and a water valve is left open, the garden may be flooded.

The following functions are provided with the ArduinoOTA library and intended to handle functionality of your application during specific stages of OTA, or on an OTA error:

```
void onStart(OTA_CALLBACK(fn));
void onEnd(OTA_CALLBACK(fn));
void onProgress(OTA_CALLBACK_PROGRESS(fn));
void onError(OTA_CALLBACK_ERROR (fn));
```

7.2.3 OTA Basic Requirements

The flash chip size should be large enough to hold the old sketch (currently running) and the new sketch (OTA) at the same time.

Keep in mind that the file system and EEPROM, for example, need space too; see *Flash layout*.

```
ESP.getFreeSketchSpace();
```

can be used for checking the free space available for the new sketch.

For an overview of memory layout, where the new sketch is stored and how it is copied during the OTA process, see *Update process - memory view*.

The following chapters provide more details and specific methods for OTA updates.

7.3 Arduino IDE

Uploading modules wirelessly from Arduino IDE is intended for the following typical scenarios:

- during firmware development as a quicker alternative to loading over a serial port,
- for updating a small number of modules,
- only if modules are accessible on the same network as the computer with the Arduino IDE.

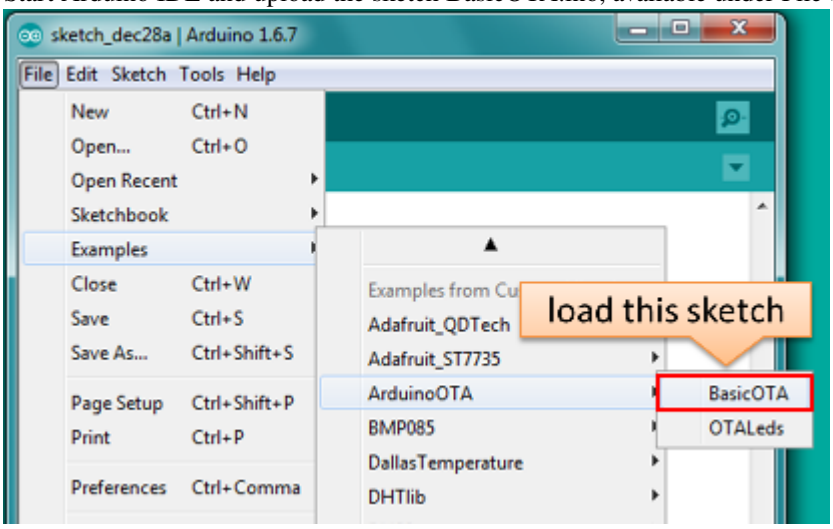
7.3.1 Requirements

- The ESP and the computer must be connected to the same network.

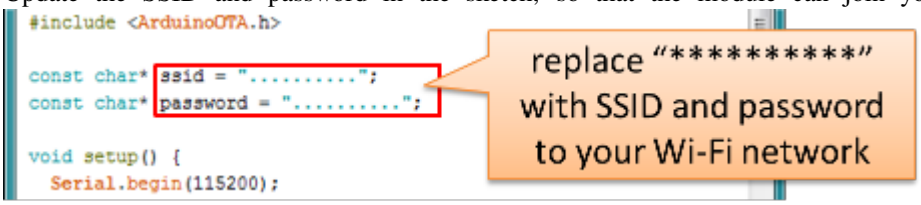
7.3.2 Application Example

Instructions below show configuration of OTA on a NodeMCU 1.0 (ESP-12E Module) board. You can use any other board that meets the *requirements* described above. This instruction is valid for all operating systems supported by the Arduino IDE. Screen captures have been made on Windows 7 and you may see small differences (like name of the serial port), if you are using Linux or MacOS.

1. Before you begin, please make sure that you have the following software installed:
   - Arduino IDE 1.6.7 or newer - https://www.arduino.cc/en/Main/Software
   - esp8266/Arduino platform package 2.0.0 or newer - for instructions follow https://github.com/esp8266/Arduino#installing-with-boards-manager
2. Now prepare the sketch and configuration for upload via a serial port.
   - Start Arduino IDE and upload the sketch BasicOTA.ino, available under File > Examples > ArduinoOTA

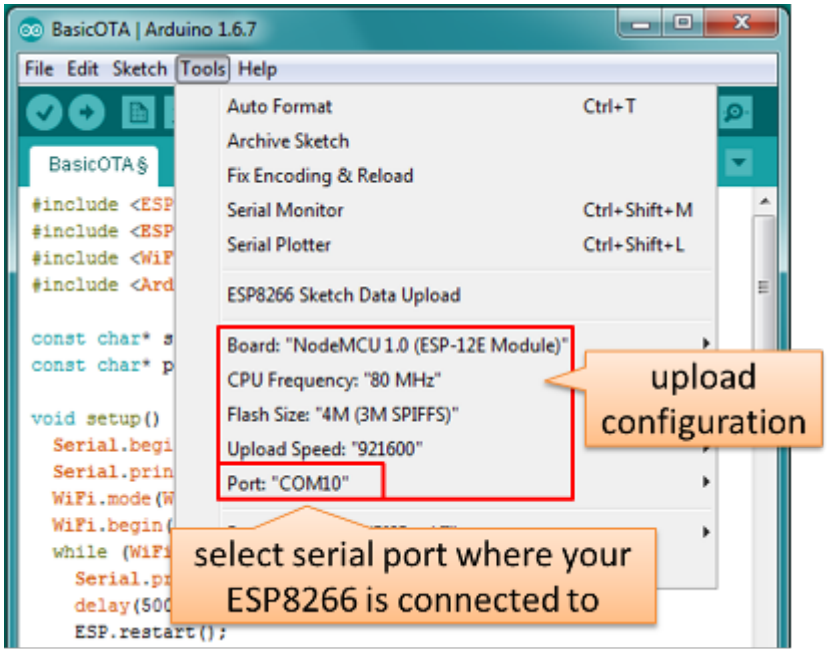


   - Update the SSID and password in the sketch, so that the module can join your Wi-Fi network



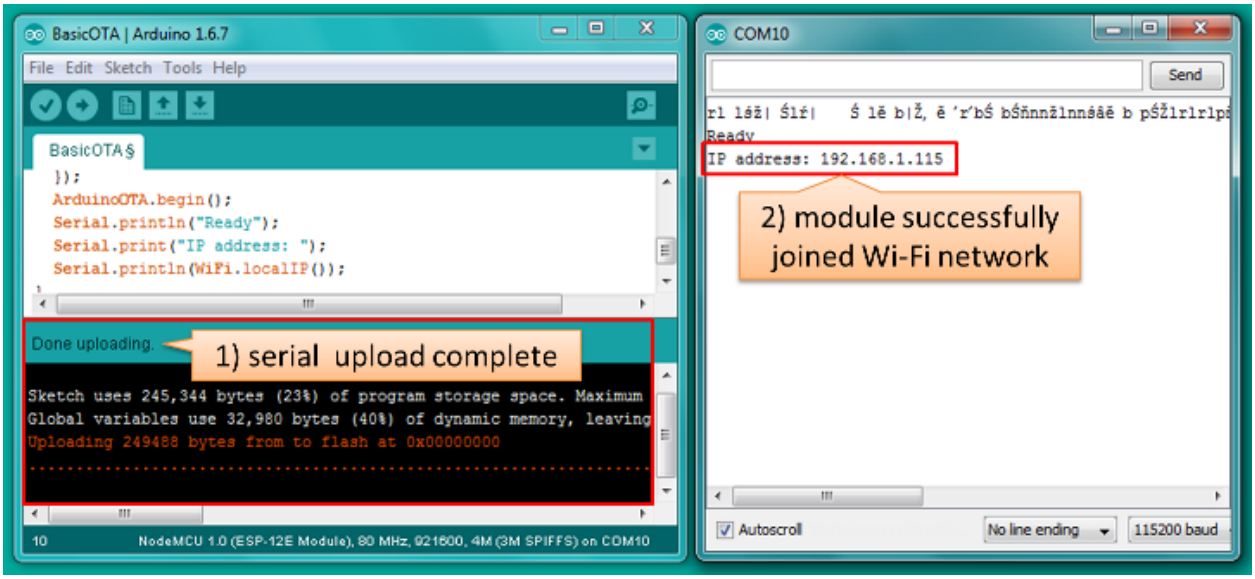
   - Configure upload parameters as below (you may need to adjust configuration if you are using a different

module):



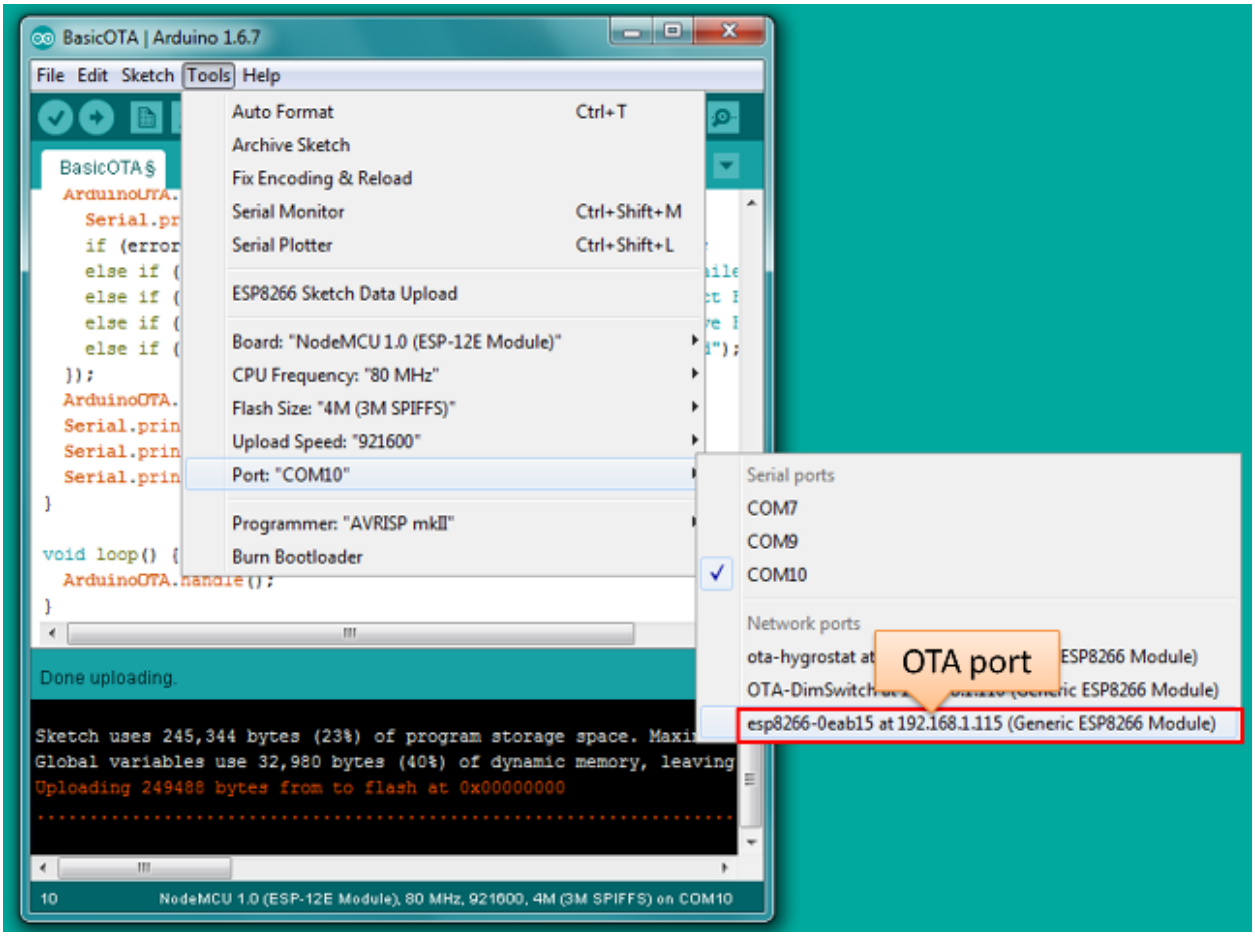
Note: Depending on version of platform package and board you have, you may see `Upload Using:` in the menu above. This option is inactive and it does not matter what you select. It has been left for compatibility with older implementation of OTA and finally removed in platform package version 2.2.0.

3. Upload the sketch (Ctrl+U). Once done, open Serial Monitor (Ctrl+Shift+M) and check if module has joined your Wi-Fi network:



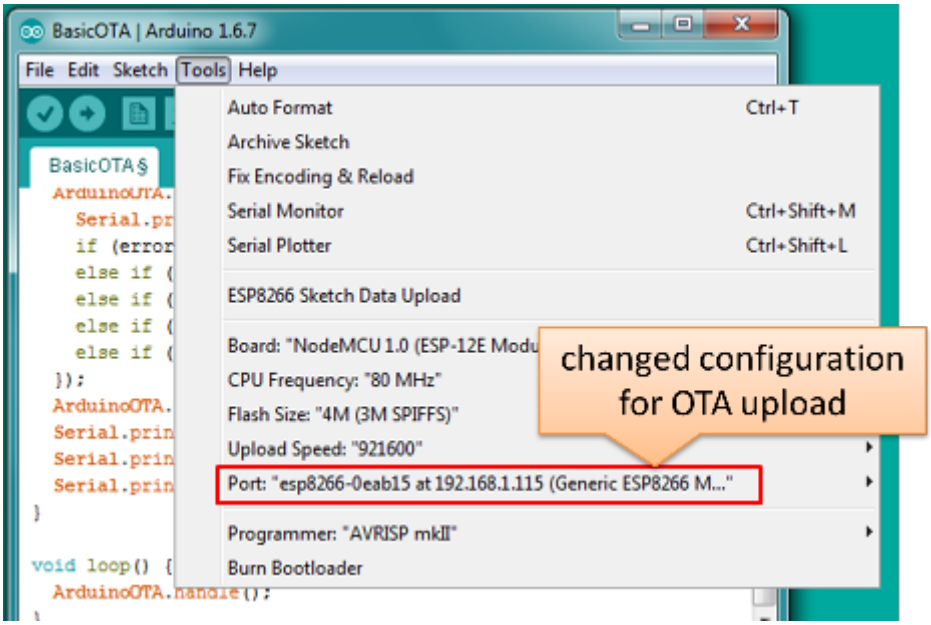
Note: The ESP module should be reset after serial upload. Otherwise, subsequent steps will not work. Reset may be done for you automatically after opening serial monitor, as visible on the screenshot above. It depends on how you have DTR and RTS wired from the USB-Serial converter to the ESP. If reset is not done automatically, then trigger it by pressing reset button or manually cycling the power. For more details why this should be done please refer to FAQ regarding `ESP.restart()`.

4. Only if the module is connected to network, after a couple of seconds, the esp8266-ota port will show up in Arduino IDE. Select port with IP address shown in the Serial Monitor window in previous step:


BasicOTA | Arduino 1.6.7
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ESP8266 Sketch Data Upload
Board: "NodeMCU 1.0 (ESP-12E Module)"
CPU Frequency: "80 MHz"
Flash Size: "4M (3M SPIFFS)"
Upload Speed: "921600"
Port: "COM10"
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM7
COM9
COM10
Network ports
OTA port
esp8266-0eab15 at 192.168.1.115 (Generic ESP8266 Module)
Done uploading.
Sketch uses 245,344 bytes (23%) of program storage space. Maxi
Global variables use 32,980 bytes (40%) of dynamic memory, leaving
Uploading 249488 bytes from to flash at 0x00000000
NodeMCU 1.0 (ESP-12E Module), 80 MHz, 921600, 4M (3M SPIFFS) on COM10

Note: If the OTA port does not show up, exit Arduino IDE, open it again and check if the port is there. If it is not, check your firewall and router settings. The OTA port is advertised using mDNS service. To check if the port is visible by your PC, you can use an application like Bonjour Browser.

5. Now get ready for your first OTA upload by selecting the OTA port:



Note: The menu entry `Upload Speed:` does not matter at this point as it concerns the serial port. Just left it unchanged.

6. If you have successfully completed all the above steps, you can upload (Ctrl+U) the same (or any other) sketch over OTA:

Note: To be able to upload your sketch over and over again using OTA, you need to embed OTA routines inside. Please use BasicOTA.ino as an example.

Password Protection

Protecting your OTA uploads with password is really straightforward. All you need to do, is to include the following statement in your code:

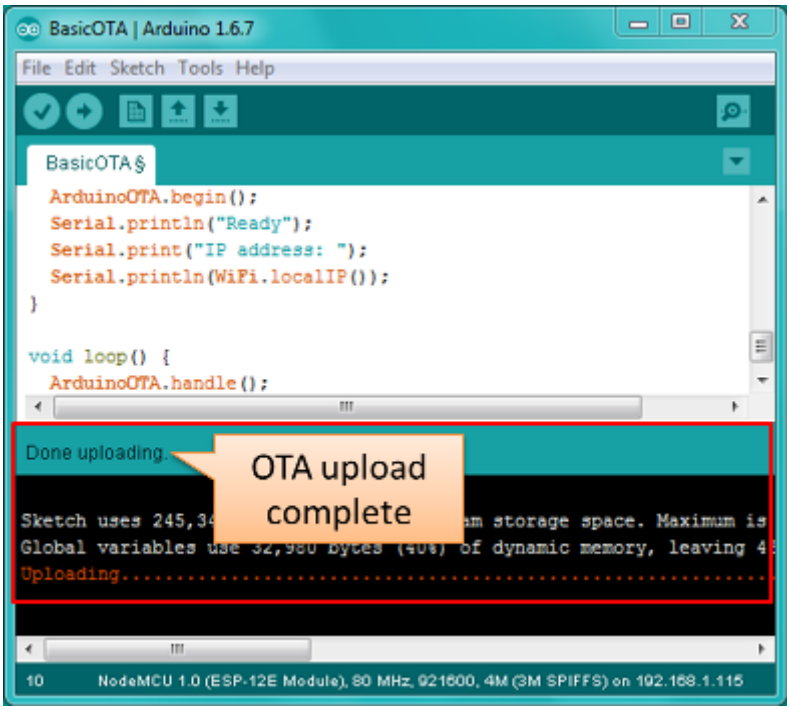
```
ArduinoOTA.setPassword((const char *)"123");
```

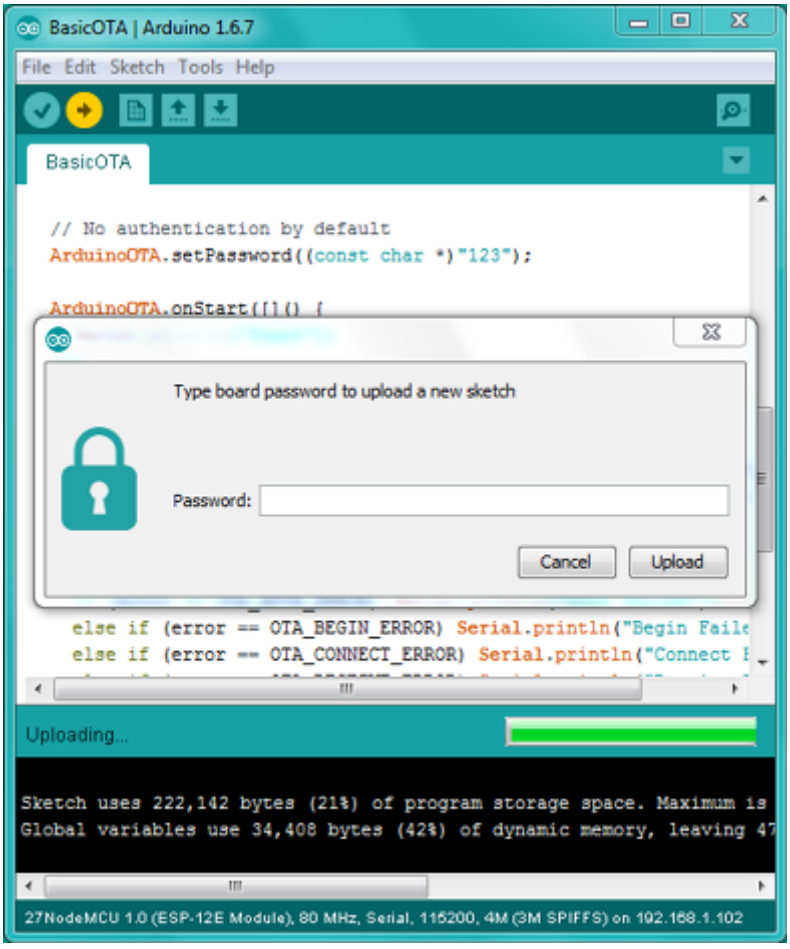
Where `123` is a sample password that you should replace with your own.

Before implementing it in your sketch, it is a good idea to check how it works using *BasicOTA.ino* sketch available under *File* > *Examples* > *ArduinoOTA*. Go ahead, open *BasicOTA.ino*, uncomment the above statement that is already there, and upload the sketch. To make troubleshooting easier, do not modify example sketch besides what is absolutely required. This is including original simple `123` OTA password. Then attempt to upload sketch again (using OTA). After compilation is complete, once upload is about to begin, you should see prompt for password as follows:

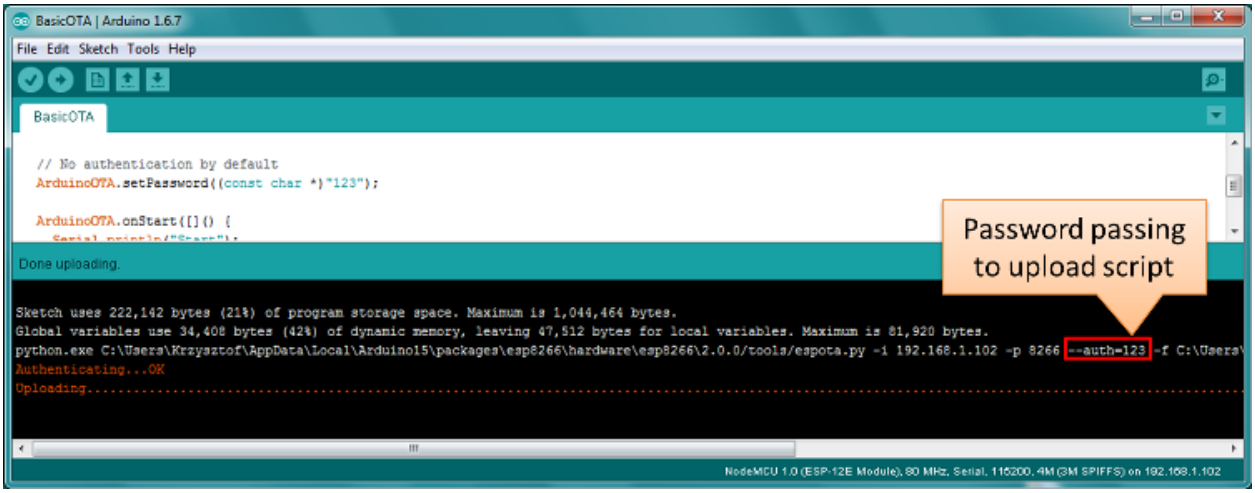
Enter the password and upload should be initiated as usual with the only difference being `Authenticating...OK` message visible in upload log.

You will not be prompted for a reentering the same password next time. Arduino IDE will remember it for you. You will see prompt for password only after reopening IDE, or if you change it in your sketch, upload the sketch and then try to upload it again.


BasicOTA | Arduino 1.6.7
File Edit Sketch Tools Help
BasicOTA §
ArduinoOTA.begin();
Serial.println("Ready");
Serial.print("IP address: ");
Serial.println(WiFi.localIP());
}
void loop() {
ArduinoOTA.handle();
Done uploading.
OTA upload complete
Uploading.....
10 NodeMCU 1.0 (ESP-12E Module), 80 MHz, 921600, 4M (3M SPIFFS) on 192.168.1.115


BasicOTA | Arduino 1.6.7
File Edit Sketch Tools Help
BasicOTA
// No authentication by default
ArduinoOTA.setPassword((const char *)"123");
Type board password to upload a new sketch
Password:
Cancel
Upload
Uploading...
Sketch uses 222,142 bytes (21%) of program storage space. Maximum is
Global variables use 34,408 bytes (42%) of dynamic memory, leaving 47
27NodeMCU 1.0 (ESP-12E Module), 80 MHz, Serial, 115200, 4M (3M SPIFFS) on 192.168.1.102

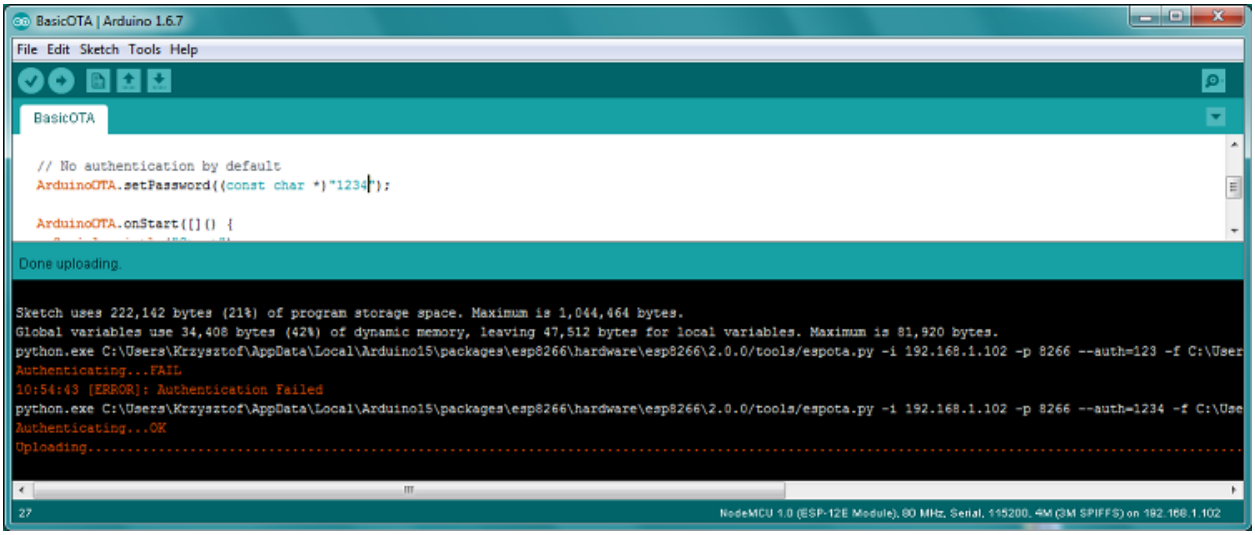
Please note, it is possible to reveal password entered previously in Arduino IDE, if IDE has not been closed since last upload. This can be done by enabling *Show verbose output during: upload* in *File > Preferences* and attempting to upload the module.



The picture above shows that the password is visible in log, as it is passed to *espota.py* upload script.

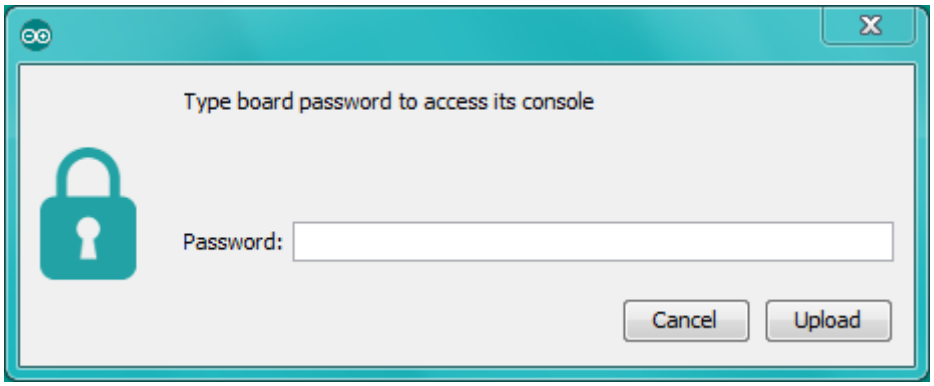
Another example below shows situation when password is changed between uploads.

When uploading, Arduino IDE used previously entered password, so the upload failed and that has been clearly reported by IDE. Only then IDE prompted for a new password. That was entered correctly and second attempt to upload has been successful.



Troubleshooting

If OTA update fails, first step is to check for error messages that may be shown in upload window of Arduino IDE. If this is not providing any useful hints, try to upload again while checking what is shown by ESP on serial port. Serial Monitor from IDE will not be useful in that case. When attempting to open it, you will likely see the following:



This window is for Arduino Yún and not yet implemented for esp8266/Arduino. It shows up because IDE is attempting to open Serial Monitor using network port you have selected for OTA upload.

Instead you need an external serial monitor. If you are a Windows user check out Termite. This is handy, slick and simple RS232 terminal that does not impose RTS or DTR flow control. Such flow control may cause issues if you are using respective lines to toggle GPIO0 and RESET pins on ESP for upload.

Select COM port and baud rate on external terminal program as if you were using Arduino Serial Monitor. Please see typical settings for Termite below:

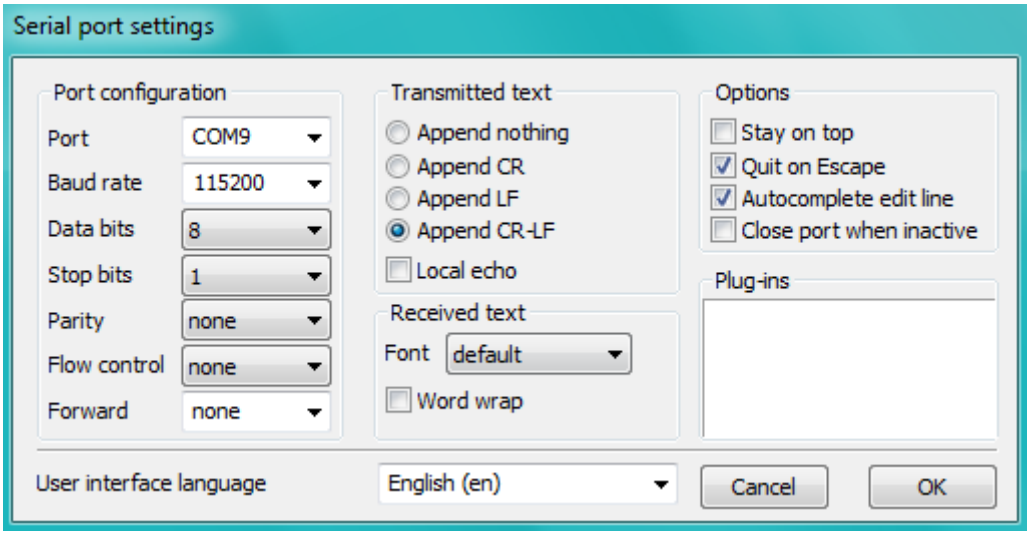
Then run OTA from IDE and look what is displayed on terminal. Successful *ArduinoOTA* process using BasicOTA.ino sketch looks like below (IP address depends on your network configuration):

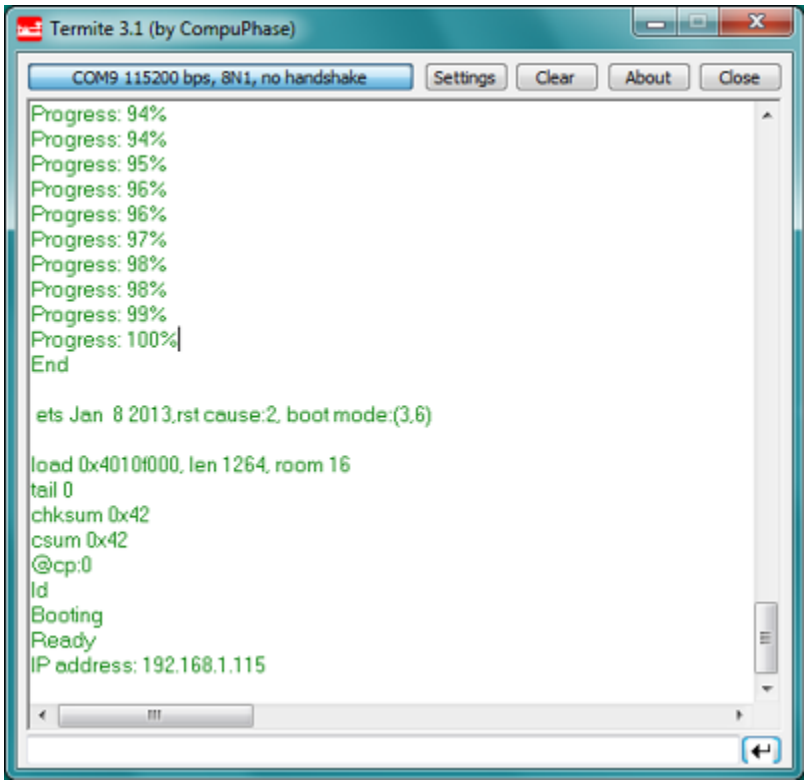
If upload fails you will likely see errors caught by the uploader, exception and the stack trace, or both.

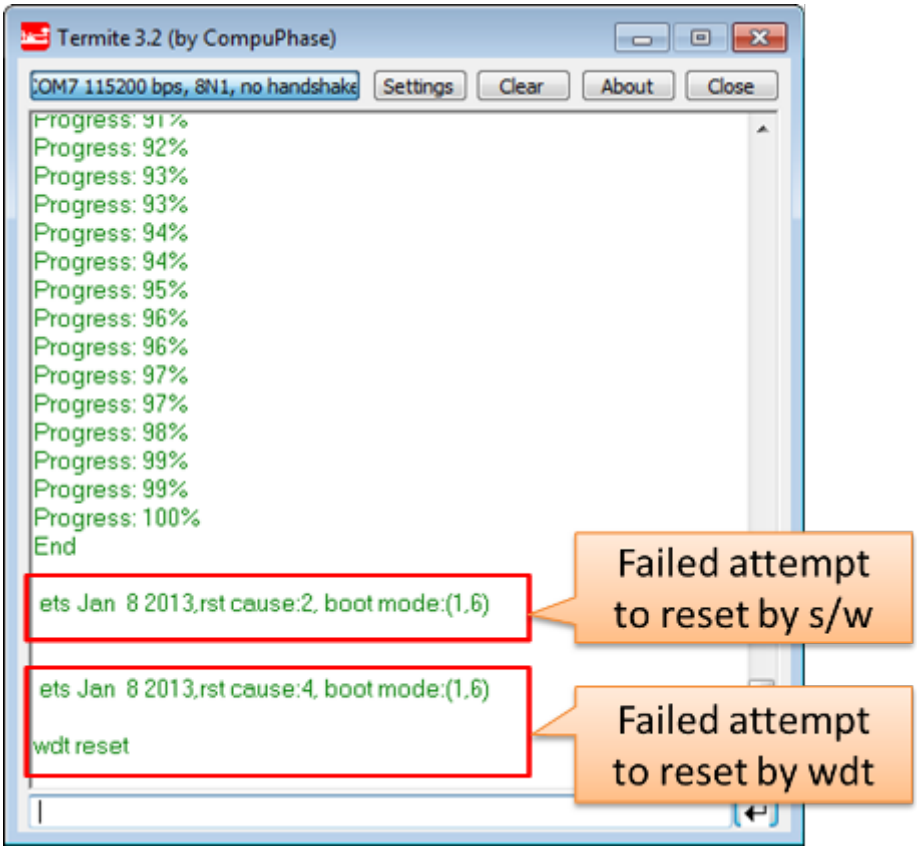
Instead of the log as on the above screen you may see the following:

If this is the case, then most likely ESP module has not been reset after initial upload using serial port.

The most common causes of OTA failure are as follows:


Serial port settings
Port configuration
Port
COM9
Baud rate
115200
Data bits
8
Stop bits
1
Parity
none
Flow control
none
Forward
none
Transmitted text
Append nothing
Append CR
Append LF
Append CR-LF
Local echo
Received text
Font
default
Word wrap
Options
Stay on top
Quit on Escape
Autocomplete edit line
Close port when inactive
Plug-ins
User interface language
English (en)
Cancel
OK


Termite 3.1 (by CompuPhase)
COM9 115200 bps, 8N1, no handshake
Settings
Clear
About
Close
Progress: 94%
Progress: 94%
Progress: 95%
Progress: 96%
Progress: 96%
Progress: 97%
Progress: 98%
Progress: 98%
Progress: 99%
Progress: 100%
End
ets Jan 8 2013,rst cause:2, boot mode:(3,6)
load 0x4010f000, len 1264, room 16
tail 0
chksum 0x42
csum 0x42
@cp:0
Id
Booting
Ready
IP address: 192.168.1.115



- not enough physical memory on the chip (e.g. ESP01 with 512K flash memory is not enough for OTA).
- too much memory declared for the filesystem so new sketch will not fit between existing sketch and the filesystem – see *Update process - memory view*.
- too little memory declared in Arduino IDE for your selected board (i.e. less than physical size).
- not resetting the ESP module after initial upload using serial port.

For more details regarding flash memory layout please check *File system*. For overview where new sketch is stored, how it is copied and how memory is organized for the purpose of OTA see *Update process - memory view*.

7.4 Web Browser

Updates described in this chapter are done with a web browser that can be useful in the following typical scenarios:

- after application deployment if loading directly from Arduino IDE is inconvenient or not possible,
- after deployment if user is unable to expose module for OTA from external update server,
- to provide updates after deployment to small quantity of modules when setting an update server is not practicable.

7.4.1 Requirements

- The ESP and the computer must be connected to the same network.

7.4.2 Implementation Overview

Updates with a web browser are implemented using `ESP8266HTTPUpdateServer` class together with `ESP8266WebServer` and `ESP8266mDNS` classes. The following code is required to get it work:

setup()

```
MDNS.begin(host);

httpUpdater.setup(&httpServer);
httpServer.begin();

MDNS.addService("http", "tcp", 80);
```

loop()

```
httpServer.handleClient();
```

7.4.3 Application Example

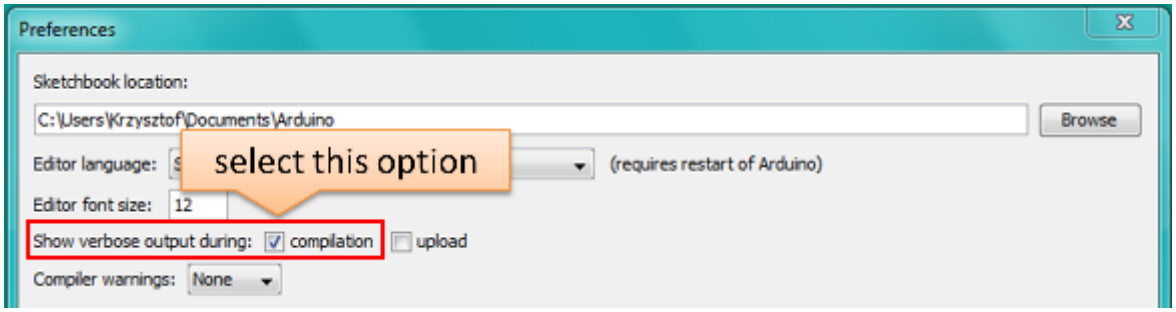
The sample implementation provided below has been done using:

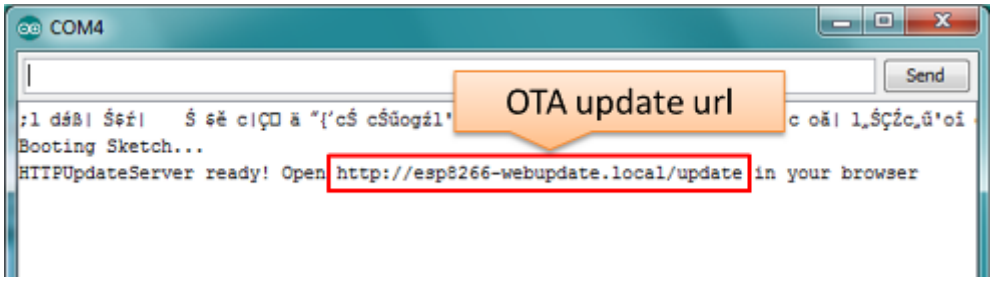
- example sketch WebUpdater.ino available in `ESP8266HTTPUpdateServer` library,
- NodeMCU 1.0 (ESP-12E Module).

You can use another module if it meets previously described *requirements*.

1. Before you begin, please make sure that you have the following software installed:
   - Arduino IDE and 2.0.0-rc1 (of Nov 17, 2015) version of platform package as described under https://github.com/esp8266/Arduino#installing-with-boards-manager
   - Host software depending on O/S you use:
     1. Avahi https://avahi.org/ for Linux
     2. Bonjour https://www.apple.com/support/bonjour/ for Windows
     3. Mac OSX and iOS - support is already built in / no any extra s/w is required
2. Prepare the sketch and configuration for initial upload with a serial port.
   - Start Arduino IDE and load sketch WebUpdater.ino available under File > Examples > ESP8266HTTPUpdateServer.
   - Update SSID and password in the sketch, so the module can join your Wi-Fi network.
   - Open File > Preferences, look for "Show verbose output during:" and check out "compilation" option.

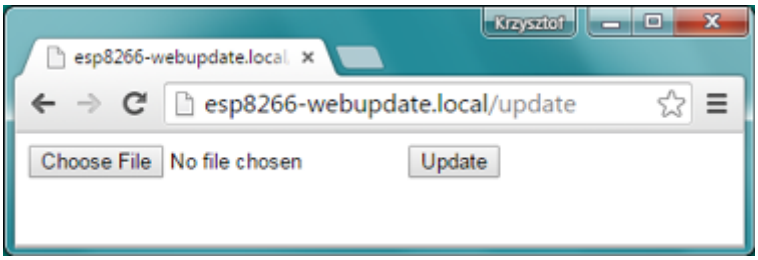
     Note: This setting will be required in step 5 below. You can uncheck this setting afterwards.
3. Upload sketch (Ctrl+U). Once done, open Serial Monitor (Ctrl+Shift+M) and check if you see the following message displayed, that contains url for OTA update.





Note: Such message will be shown only after module successfully joins network and is ready for an OTA upload. Please remember about resetting the module once after serial upload as discussed in chapter *Arduino IDE*, step 3.

4. Now open web browser and enter the url provided on Serial Monitor, i.e. `http://esp8266-webupdate.local/update`. Once entered, browser should display a form like below that has been served by your module. The form invites you to choose a file for update.



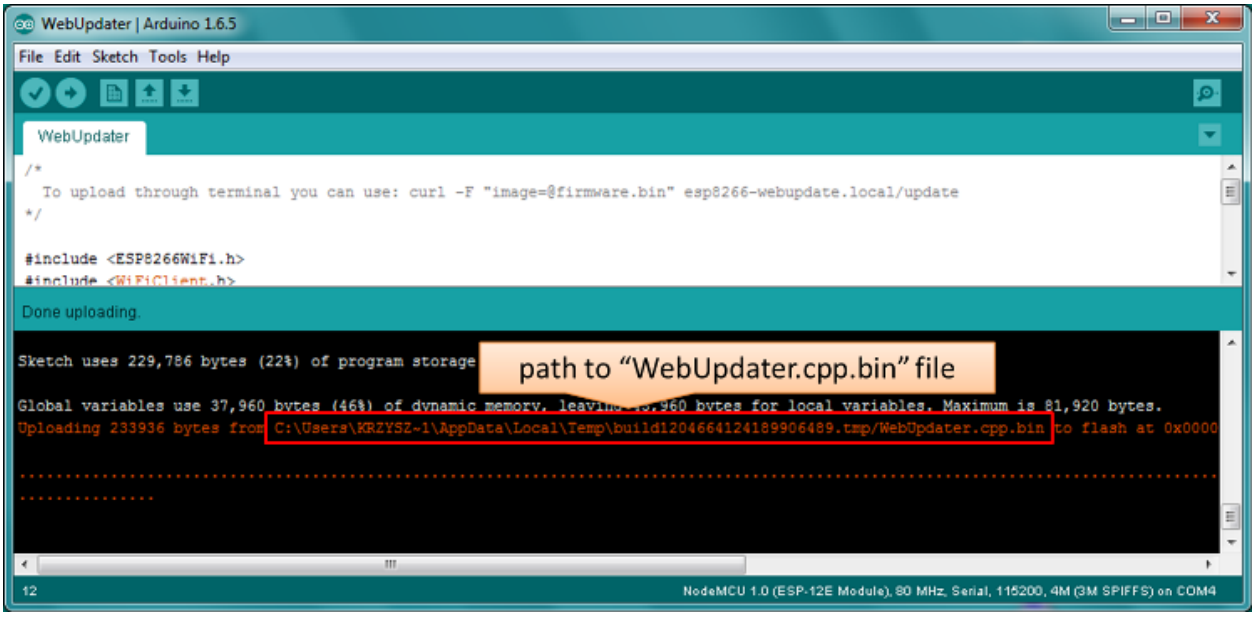
Note: If entering `http://esp8266-webupdate.local/update` does not work, try replacing `esp8266-webupdate` with module's IP address. For example, if your module IP is `192.168.1.100` then url should be `http://192.168.1.100/update`. This workaround is useful in case the host software installed in step 1 does not work. If still nothing works and there are no clues on the Serial Monitor, try to diagnose issue by opening provided url in Google Chrome, pressing F12 and checking contents of "Console" and "Network" tabs. Chrome provides some advanced logging on these tabs.

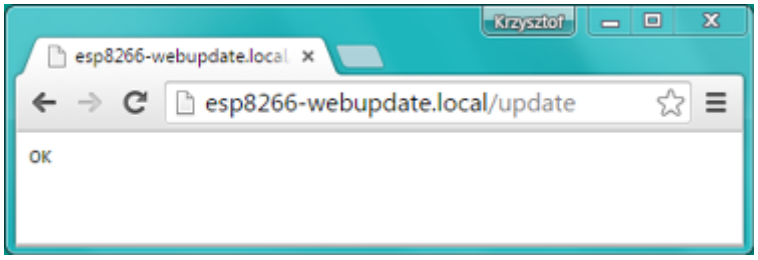
5. To obtain the file, navigate to directory used by Arduino IDE to store results of compilation. You can check the path to this file in compilation log shown in IDE debug window as marked below.

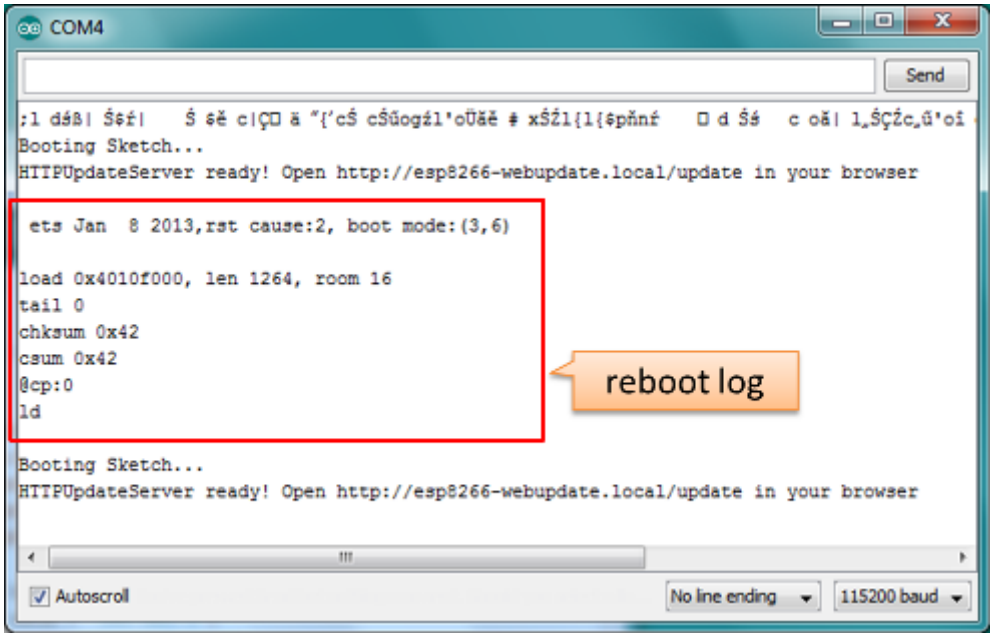
6. Now press "Choose File" in web browser, go to directory identified in step 5 above, find the file "WebUpdater.cpp.bin" and upload it. If upload is successful, you will see "OK" on web browser like below.

   Module will reboot that should be visible on Serial Monitor:

   Just after reboot you should see exactly the same message `HTTPUpdateServer ready! Open http://esp8266-webupdate.local/update in your browser` like in step 3. This is because module has been loaded again with the same code – first using serial port, and then using OTA.


WebUpdater | Arduino 1.6.5
File Edit Sketch Tools Help
WebUpdater
/*
To upload through terminal you can use: curl -F "image=@firmware.bin" esp8266-webupdate.local/update
*/
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
Done uploading.
Sketch uses 229,786 bytes (22%) of program storage
path to “WebUpdater.cpp.bin” file
Global variables use 37,960 bytes (46%) of dynamic memory, leaving
Uploading 233936 bytes from C:\Users\KRZYSZ~1\AppData\Local\Temp\build1204664124189906489.tmp/WebUpdater.cpp.bin to flash at 0x0000
12
NodeMCU 1.0 (ESP-12E Module), 80 MHz, Serial, 115200, 4M (3M SPIFFS) on COM4


Krzysztof
esp8266-webupdate.local
esp8266-webupdate.local/update
OK


COM4
Send
Booting Sketch...
HTTPUpdateServer ready! Open http://esp8266-webupdate.local/update in your browser
ets Jan 8 2013,rst cause:2, boot mode:(3,6)
load 0x4010f000, len 1264, room 16
tail 0
chksum 0x42
csum 0x42
@cp:0
ld
reboot log
Booting Sketch...
HTTPUpdateServer ready! Open http://esp8266-webupdate.local/update in your browser
Autoscroll
No line ending
115200 baud

Once you are comfortable with this procedure, go ahead and modify WebUpdater.ino sketch to print some additional messages, compile it, locate new binary file and upload it using web browser to see entered changes on a Serial Monitor.

You can also add OTA routines to your own sketch following guidelines in *Implementation Overview* above. If this is done correctly, you should be always able to upload new sketch over the previous one using a web browser.

In case OTA update fails dead after entering modifications in your sketch, you can always recover module by loading it over a serial port. Then diagnose the issue with sketch using Serial Monitor. Once the issue is fixed try OTA again.

7.5 HTTP Server

`ESP8266HTTPUpdate` class can check for updates and download a binary file from HTTP web server. It is possible to download updates from every IP or domain address on the network or Internet.

Note that by default this class closes all other connections except the one used by the update, this is because the update method blocks. This means that if there's another application receiving data then TCP packets will build up in the buffer leading to out of memory errors causing the OTA update to fail. There's also a limited number of receive buffers available and all may be used up by other applications.

There are some cases where you know that you won't be receiving any data but would still like to send progress updates. It's possible to disable the default behaviour (and keep connections open) by calling closeConnectionsOnUpdate(false).

7.5.1 Requirements

- web server

7.5.2 Arduino code

Simple updater

Simple updater downloads the file every time the function is called.

```
#include <ESP8266httpUpdate.h>

WiFiClient client;
ESPhttpUpdate.update(client, "192.168.0.2", 80, "/arduino.bin");
```

Advanced updater

Its possible to point the update function to a script on the server. If a version string argument is given, it will be sent to the server. The server side script can use this string to check whether an update should be performed.

The server-side script can respond as follows: - response code 200, and send the firmware image, - or response code 304 to notify ESP that no update is required.

```
#include <ESP8266httpUpdate.h>

WiFiClient client;
t_httpUpdate_return ret = ESPhttpUpdate.update(client, "192.168.0.2", 80, "/esp/update/
↪arduino.php", "optional current version string here");
switch(ret) {
```

(continues on next page)

(continued from previous page)

```
    case HTTP_UPDATE_FAILED:
        Serial.println("[update] Update failed.");
        break;
    case HTTP_UPDATE_NO_UPDATES:
        Serial.println("[update] Update no Update.");
        break;
    case HTTP_UPDATE_OK:
        Serial.println("[update] Update ok."); // may not be called since we reboot the␣
↪ESP
        break;
}
```

TLS updater

Please read and try the examples provided with the library.

7.5.3 Server request handling

Simple updater

For the simple updater the server only needs to deliver the binary file for update.

Advanced updater

For advanced update management a script (such as a PHP script) needs to run on the server side. On every update request, the ESP sends some information in HTTP headers to the server.

Example header data:

```
[User-Agent] => ESP8266-http-Update
[x-ESP8266-STA-MAC] => 18:FE:AA:AA:AA:AA
[x-ESP8266-AP-MAC] => 1A:FE:AA:AA:AA:AA
[x-ESP8266-free-space] => 671744
[x-ESP8266-sketch-size] => 373940
[x-ESP8266-sketch-md5] => a56f8ef78a0bebd812f62067daf1408a
[x-ESP8266-chip-size] => 4194304
[x-ESP8266-sdk-version] => 1.3.0
[x-ESP8266-version] => DOOR-7-g14f53a19
[x-ESP8266-mode] => sketch
```

With this information the script now can check if an update is needed. It is also possible to deliver different binaries based on the MAC address, as in the following example:

```
<?PHP

header('Content-type: text/plain; charset=utf8', true);

function check_header($name, $value = false) {
    if(!isset($_SERVER[$name])) {
        return false;
```

(continues on next page)

(continued from previous page)

```
    }
    if($value && $_SERVER[$name] != $value) {
        return false;
    }
    return true;
}

function sendFile($path) {
    header($_SERVER["SERVER_PROTOCOL"].' 200 OK', true, 200);
    header('Content-Type: application/octet-stream', true);
    header('Content-Disposition: attachment; filename='.basename($path));
    header('Content-Length: '.filesize($path), true);
    header('x-MD5: '.md5_file($path), true);
    readfile($path);
}

if(!check_header('User-Agent', 'ESP8266-http-Update')) {
    header($_SERVER["SERVER_PROTOCOL"].' 403 Forbidden', true, 403);
    echo "only for ESP8266 updater!\n";
    exit();
}

if(
    !check_header('x-ESP8266-STA-MAC') ||
    !check_header('x-ESP8266-AP-MAC') ||
    !check_header('x-ESP8266-free-space') ||
    !check_header('x-ESP8266-sketch-size') ||
    !check_header('x-ESP8266-sketch-md5') ||
    !check_header('x-ESP8266-chip-size') ||
    !check_header('x-ESP8266-sdk-version')
) {
    header($_SERVER["SERVER_PROTOCOL"].' 403 Forbidden', true, 403);
    echo "only for ESP8266 updater! (header)\n";
    exit();
}

$db = array(
    "18:FE:AA:AA:AA:AA" => "DOOR-7-g14f53a19",
    "18:FE:AA:AA:AA:BB" => "TEMP-1.0.0"
);

if(!isset($db[$_SERVER['x-ESP8266-STA-MAC']])) {
    header($_SERVER["SERVER_PROTOCOL"].' 500 ESP MAC not configured for updates', true,␣
↪500);
}

$localBinary = "./bin/".$db[$_SERVER['x-ESP8266-STA-MAC']].".bin";

// Check if version has been set and does not match, if not, check if
// MD5 hash between local binary and ESP8266 binary do not match if not.
// then no update has been found.
if((!check_header('x-ESP8266-sdk-version') && $db[$_SERVER['x-ESP8266-STA-MAC']] != $_
```

(continues on next page)

(continued from previous page)

```
↪SERVER['x-ESP8266-version'])
    || $_SERVER["x-ESP8266-sketch-md5"] != md5_file($localBinary)) {
    sendFile($localBinary);
} else {
    header($_SERVER["SERVER_PROTOCOL"].' 304 Not Modified', true, 304);
}

header($_SERVER["SERVER_PROTOCOL"].' 500 no version for ESP MAC', true, 500);
```

7.6 Stream Interface

The Stream Interface is the base for all other update modes like OTA, HTTP Server / client. Given a Stream-class variable *streamVar* providing *byteCount* bytes of firmware, it can store the firmware as follows:

```
Update.begin(firmwareLengthInBytes);
Update.writeStream(streamVar);
Update.end();
```

7.7 Updater class

Updater is in the Core and deals with writing the firmware to the flash, checking its integrity and telling the bootloader (eboot) to load the new firmware on the next boot.

The following *Updater <https://github.com/esp8266/Arduino/tree/master/cores/esp8266/Updater.h* methods could be used to be notified about OTA progress:

```
using THandlerFunction_Progress = std::function<void(size_t, size_t)>;
void onProgress(THandlerFunction_Progress); // current and total number of bytes

using THandlerFunction_Error = std::function<void(uint8_t)>;
void onStart(THandlerFunction_Error); // error code

using THandlerFunction = std::function<void()>;
void onEnd(THandlerFunction);
void onError(THandlerFunction);
```

7.7.1 Using RTC memory

The bootloader command will be stored into the first 128 bytes of user RTC memory, then it will be retrieved by eboot on boot. That means that user data present there will be lost (per discussion in #5330).

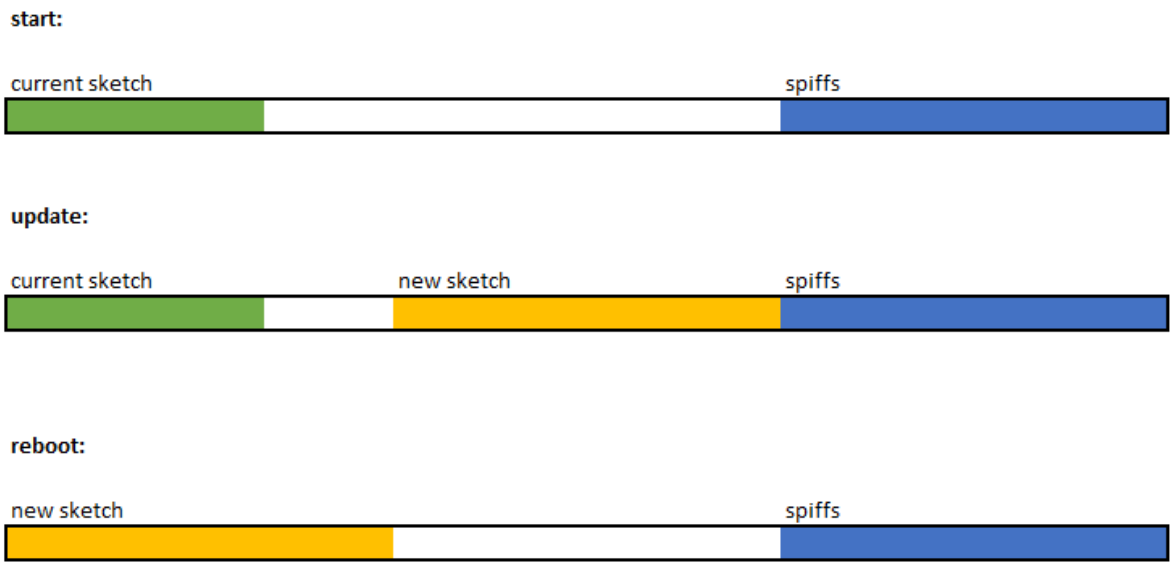
7.7.2 Flash mode and size

For uncompressed firmware images, the Updater will change the flash mode bits if they differ from the flash mode the device is currently running at. This ensures that the flash mode is not changed to an incompatible mode when the device is in a remote or hard to access area. Compressed images are not modified, thus changing the flash mode in this instance could result in damage to the ESP8266 and/or flash memory chip or your device no longer be accessible via OTA, and requiring re-flashing via a serial connection (per discussion in #7307).

7.7.3 Update process - memory view

- The new sketch will be stored in the space between the old sketch and the spiff.
- on the next reboot, the "eboot" bootloader checks for commands.
- the new sketch is now copied "over" the old one.
- the new sketch is started.

By default, OTA filesystem updates overwrite the target flash directly. This can lead to the file system being corrupted if there is a power outage during the update process. In order to use the same two step process that is used for OTA application firmware updates, set the *ATOMIC_FS_UPDATE* flag. Note that you will need to have enough unused space for the new filesystem image to be stored, hence is why this is not the default behaviour.



CHAPTER

EIGHT

GUIDE TO PROGMEM ON ESP8266 AND ARDUINO IDE

8.1 Intro

PROGMEM is a Arduino AVR feature that has been ported to ESP8266 to ensure compatibility with existing Arduino libraries, as well as, saving RAM. On the esp8266 declaring a string such as `const char * xyz = "this is a string"` will place this string in RAM, not flash. It is possible to place a String into flash, and then load it into RAM when it is needed. On an 8bit AVR this process is very simple. On the 32bit ESP8266 there are conditions that must be met to read back from flash.

On the ESP8266 PROGMEM is a macro:

```
#define PROGMEM   ICACHE_RODATA_ATTR
```

`ICACHE_RODATA_ATTR` is defined by:

```
#define ICACHE_RODATA_ATTR  __attribute__((section(".irom.text")))
```

Which places the variable in the .irom.text section in flash. Placing strings in flash requires using any of the methods above.

Declare a global string to be stored in flash.

```
static const char xyz[] PROGMEM = "This is a string stored in flash";
```

8.2 Declare a flash string within code block.

For this you can use the PSTR macro. Which are all defined in pgmspace.h

```
#define PGM_P       const char *
#define PGM_VOID_P  const void *
#define PSTR(s) (__extension__({static const char __c[] PROGMEM = (s); &__c[0];}))
```

In practice:

```
void myfunction(void) {
PGM_P xyz = PSTR("Store this string in flash");
const char * abc = PSTR("Also Store this string in flash");
}
```

The two examples above will store these strings in flash. To retrieve and manipulate flash strings they must be read from flash in 4byte words. In the Arduino IDE for esp8266 there are several functions that can help retrieve strings from flash that have been stored using PROGMEM. Both of the examples above return `const char *`. However use of these pointers, without correct 32bit alignment you will cause a segmentation fault and the ESP8266 will crash. You must read from the flash 32 bit aligned.

8.3 Functions to read back from PROGMEM

Which are all defined in pgmspace.h

```
int memcmp_P(const void* buf1, PGM_VOID_P buf2P, size_t size);
void* memccpy_P(void* dest, PGM_VOID_P src, int c, size_t count);
void* memmem_P(const void* buf, size_t bufSize, PGM_VOID_P findP, size_t findPSize);
void* memcpy_P(void* dest, PGM_VOID_P src, size_t count);
char* strncpy_P(char* dest, PGM_P src, size_t size);
char* strcpy_P(dest, src)
char* strncat_P(char* dest, PGM_P src, size_t size);
char* strcat_P(dest, src)
int strncmp_P(const char* str1, PGM_P str2P, size_t size);
int strcmp_P(str1, str2P)
int strncasecmp_P(const char* str1, PGM_P str2P, size_t size);
int strcasecmp_P(str1, str2P)
size_t strnlen_P(PGM_P s, size_t size);
size_t strlen_P(strP)
char* strstr_P(const char* haystack, PGM_P needle);
int printf_P(PGM_P formatP, ...);
int sprintf_P(char *str, PGM_P formatP, ...);
int snprintf_P(char *str, size_t strSize, PGM_P formatP, ...);
int vsnprintf_P(char *str, size_t strSize, PGM_P formatP, va_list ap);
```

There are a lot of functions there but in reality they are `_P` versions of standard c functions that are adapted to read from the esp8266 32bit aligned flash. All of them take a `PGM_P` which is essentially a `const char *`. Under the hood these functions all use, a process to ensure that 4 bytes are read, and the request byte is returned.

This works well when you have designed a function as above that is specialised for dealing with PROGMEM pointers but there is no type checking except against `const char *`. This means that it is totally legitimate, as far as the compiler is concerned, for you to pass it any `const char *` string, which is obviously not true and will lead to undefined behaviour. This makes it impossible to create any overloaded functions that can use flash strings when they are defined as `PGM_P`. If you try you will get an ambiguous overload error as `PGM_P == const char *`.

Enter the __FlashStringHelper... This is a wrapper class that allows flash strings to be used as a class, this means that type checking and function overloading can be used with flash strings. Most people will be familiar with the `F()` macro and possibly the FPSTR() macro. These are defined in WString.h:

```
#define FPSTR(pstr_pointer) (reinterpret_cast<const __FlashStringHelper *>(pstr_pointer))
#define F(string_literal) (FPSTR(PSTR(string_literal)))
```

So `FPSTR()` takes a PROGMEM pointer to a string and casts it to this `__FlashStringHelper` class. Thus if you have defined a string as above `xyz` you can use `FPSTR()` to convert it to `__FlashStringHelper` for passing into functions that take it.

```
static const char xyz[] PROGMEM = "This is a string stored in flash";
Serial.println(FPSTR(xyz));
```

The F() combines both of these methods to create an easy and quick way to store an inline string in flash, and return the type __FlashStringHelper. For example:

```
Serial.println(F("This is a string stored in flash"));
```

Although these two functions provide a similar function, they serve different roles. FPSTR() allows you to define a global flash string and then use it in any function that takes __FlashStringHelper. F() allows you to define these flash strings in place, but you can't use them anywhere else. The consequence of this is sharing common strings is possible using FPSTR() but not F(). __FlashStringHelper is what the String class uses to overload its constructor:

```
String(const char *cstr = nullptr); // constructor from const char *
String(const String &str); // copy constructor
String(const __FlashStringHelper *str); // constructor for flash strings
```

This allows you to write:

```
String mystring(F("This string is stored in flash"));
```

How do I write a function to use __FlashStringHelper? Simples: cast the pointer back to a PGM_P and use the _P functions shown above. This an example implementation for String for the concat function.

```
unsigned char String::concat(const __FlashStringHelper * str) {
    if (!str) return 0; // return if the pointer is void
    int length = strlen_P((PGM_P)str); // cast it to PGM_P, which is basically const␣
↪char *, and measure it using the _P version of strlen.
    if (length == 0) return 1;
    unsigned int newlen = len + length;
    if (!reserve(newlen)) return 0; // create a buffer of the correct length
    strcpy_P(buffer + len, (PGM_P)str); //copy the string in using strcpy_P
    len = newlen;
    return 1;
}
```

8.4 How do I declare a global flash string and use it?

```
static const char xyz[] PROGMEM = "This is a string stored in flash. Len = %u";

void setup() {
    Serial.begin(115200); Serial.println();
    Serial.println( FPSTR(xyz) ); // just prints the string, must convert it to␣
↪FlashStringHelper first using FPSTR().
    Serial.printf_P( xyz, strlen_P(xyz)); // use printf with PROGMEM string
}
```

8.5 How do I use inline flash strings?

```
void setup() {
    Serial.begin(115200); Serial.println();
    Serial.println( F("This is an inline string")); //
    Serial.printf_P( PSTR("This is an inline string using printf %s"), "hello");
}
```

8.6 How do I declare and use data in PROGMEM?

```
const size_t len_xyz = 30;
const uint8_t xyz[] PROGMEM = {
  0x53, 0x61, 0x79, 0x20, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20,
  0x74, 0x6f, 0x20, 0x4d, 0x79, 0x20, 0x4c, 0x69, 0x74, 0x74,
  0x6c, 0x65, 0x20, 0x46, 0x72, 0x69, 0x65, 0x6e, 0x64, 0x00};

 void setup() {
     Serial.begin(115200); Serial.println();
     uint8_t * buf = new uint8_t[len_xyz];
     if (buf) {
      memcpy_P(buf, xyz, len_xyz);
      Serial.write(buf, len_xyz); // output the buffer.
     }
 }
```

8.7 How do I declare some data in PROGMEM, and retrieve one byte from it.

Declare the data as done previously, then use `pgm_read_byte` to get the value back.

```
const size_t len_xyz = 30;
const uint8_t xyz[] PROGMEM = {
  0x53, 0x61, 0x79, 0x20, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20,
  0x74, 0x6f, 0x20, 0x4d, 0x79, 0x20, 0x4c, 0x69, 0x74, 0x74,
  0x6c, 0x65, 0x20, 0x46, 0x72, 0x69, 0x65, 0x6e, 0x64, 0x00
};

void setup() {
  Serial.begin(115200); Serial.println();
  for (int i = 0; i < len_xyz; i++) {
    uint8_t byteval = pgm_read_byte(xyz + i);
    Serial.write(byteval); // output the buffer.
  }
}
```

8.8 How do I declare Arrays of strings in PROGMEM and retrieve an element from it.

It is often convenient when working with large amounts of text, such as a project with an LCD display, to setup an array of strings. Because strings themselves are arrays, this is actually an example of a two-dimensional array.

These tend to be large structures so putting them into program memory is often desirable. The code below illustrates the idea.

```
// Define Strings
const char string_0[] PROGMEM = "String 0";
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";

// Initialize Table of Strings
const char* const string_table[] PROGMEM = { string_0, string_1, string_2, string_3,␣
↪string_4, string_5 };

char buffer[30]; // buffer for reading the string to (needs to be large enough to take␣
↪the longest string

void setup() {
  Serial.begin(9600);
  Serial.println("OK");
}

void loop() {
  for (int i = 0; i < 6; i++) {
    strcpy_P(buffer, (char*)pgm_read_dword(&(string_table[i])));
    Serial.println(buffer);
    delay(500);
  }
}
```

8.9 In summary

It is easy to store strings in flash using `PROGMEM` and `PSTR` but you have to create functions that specifically use the pointers they generate as they are basically `const char *`. On the other hand `FPSTR` and `F()` give you a class that you can do implicit conversions from, very useful when overloading functions, and doing implicit type conversions. It is worth adding that if you wish to store an `int`, `float` or pointer these can be stored and read back directly as they are 4 bytes in size and therefore will be always aligned!

Hope this helps.

CHAPTER

NINE

USING GDB TO DEBUG APPLICATIONS

ESP applications can be debugged using GDB, the GNU debugger, which is included with the standard IDE installation. This note will only discuss the ESP specific steps, so please refer to the main GNU GDB documentation.

Note that as of 2.5.0, the toolchain moved from the ESPRESSIF patched, closed-source version of GDB to the main GNU version. The debugging formats are different, so please be sure to use only the latest Arduino toolchain GDB executable.

9.1 CLI and IDE Note

Because the Arduino IDE doesn't support interactive debugging, the following sections describe debugging using the command line. Other IDEs which use GDB in their debug backends should work identically, but you may need to edit their configuration files or options to enable the remote serial debugging required and to set the standard options. PRs are happily accepted for updates to this document with additional IDEs!

9.2 Preparing your application for GDB

Applications need to be changed to enable GDB debugging support. This change will add 2-3KB of flash and around 700 bytes of IRAM usage, but should not affect operation of the application.

In your main `sketch.ino` file, add the following line to the top of the application:

```
#include <GDBStub.h>
```

And in the `void setup()` function ensure the serial port is initialized and call `gdbstub_init()`:

```
Serial.begin(115200);
gdbstub_init();
```

Rebuild and reupload your application and it should run exactly as before.

9.3 Starting a Debug Session

Once your application is running, the process to attach a debugger is quite simple: . Close the Arduino Serial Monitor . Locate Application.ino.elf File . Open a Command Prompt and Start GDB . Apply the GDB configurations . Attach the Debugger . Debug Away!

9.3.1 Close the Arduino Serial Monitor

Because GDB needs full control of the serial port, you will need to close any Arduino Serial Monitor windows you may have open. Otherwise GDB will report an error while attempting to debug.

9.3.2 Locate Application.ino.elf File

In order for GDB to debug your application, you need to locate the compiled ELF format version of it (which includes needed debug symbols).

Under Linux these files are stored in `/tmp/arduino_build_*` and the following command will help locate the right file for your app:

```
find /tmp -name "*.elf" -print
```

Under Windows these files are stored in `%userprofile%\AppData\Local\Temp\arduino_build_*` and the following command will help locate the right file for your app:

```
dir %userprofile%\appdata\*.elf /s/b
```

Note the full path of ELF file that corresponds to your sketch name, it will be needed later once GDB is started.

9.3.3 Open a Command Prompt and Start GDB

Open a terminal or `CMD` prompt and navigate to the proper ESP8266 toolchain directory.

Linux

```
~/.arduino15/packages/esp8266/tools/xtensa-lx106-elf-gcc/2.5.0-4-b40a506/bin/xtensa-
↪lx106-elf-gdb
```

Windows (Using Board Manager version)

```
%userprofile%\AppData\Local\Arduino15\packages\esp8266\tools\xtensa-lx106-elf-gcc\2.5.0-
↪3-20ed2b9\bin\xtensa-lx106-elf-gdb.exe
```

Windows (Using Git version)

```
%userprofile%\Documents\Arduino\hardware\esp8266com\esp8266\tools\xtensa-lx106-elf\bin\
↪xtensa-lx106-elf-gdb.exe
```

Please note the proper GDB name is “xtensa-lx106-elf-gdb”. If you accidentally run “gdb” you may start your own operating system’s GDB, which will not know how to talk to the ESP8266.

9.3.4 Apply the GDB Configurations

At the `(gdb)` prompt, enter the following options to configure GDB for the ESP8266 memory map and configuration:

```
set remote hardware-breakpoint-limit 1
set remote hardware-watchpoint-limit 1
set remote interrupt-on-connect on
set remote kill-packet off
set remote symbol-lookup-packet off
set remote verbose-resume-packet off
mem 0x20000000 0x3feffffff ro cache
mem 0x3ff00000 0x3fffffff rw
mem 0x40000000 0x400fffff ro cache
mem 0x40100000 0x4013ffff rw cache
mem 0x40140000 0x5fffffff ro cache
mem 0x60000000 0x60001fff rw
set serial baud 115200
```

Now tell GDB where your compiled ELF file is located:

```
file /tmp/arduino_build_257110/sketch_dec26a.ino.elf
```

9.3.5 Attach the Debugger

Once GDB has been configured properly and loaded your debugging symbols, connect it to the ESP with the command (replace the ttyUSB0 or COM9 with your ESP's serial port):

```
target remote /dev/ttyUSB0
```

or

```
target remote \\.\COM9
```

At this point GDB will send a stop the application on the ESP8266 and you can begin setting a breakpoint (`break loop`) or any other debugging operation.

9.4 Example Debugging Session

Create a new sketch and paste the following code into it:

```
#include <GDBStub.h>

void setup() {
  Serial.begin(115200);
  gdbstub_init();
  Serial.printf("Starting...\n");
}

void loop() {
  static uint32_t cnt = 0;
  Serial.printf("%d\n", cnt++);
```

(continues on next page)

(continued from previous page)

```
  delay(100);
}
```

Save it and then build and upload to your ESP8266. On the Serial monitor you should see something like

```
1
2
3
....
```

Now close the Serial Monitor.

Open a command prompt and find the ELF file:

```
earle@server:~$ find /tmp -name "*.elf" -print
/tmp/arduino_build_257110/testgdb.ino.elf
/tmp/arduino_build_531411/listfiles.ino.elf
/tmp/arduino_build_156712/SDWebServer.ino.elf
```

In this example there are multiple `elf` files found, but we only care about the one we just built, `testgdb.ino.elf`.

Open up the proper ESP8266-specific GDB

```
earle@server:~$ ~/.arduino15/packages/esp8266/hardware/xtensa-lx106-elf/bin/xtensa-lx106-
↪elf-gdb
GNU gdb (GDB) 8.2.50.20180723-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=xtensa-lx106-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
    <https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <https://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

We're now at the GDB prompt, but nothing has been set up for the ESP8266 and no debug information has been loaded. Cut-and-paste the setup options:

```
(gdb) set remote hardware-breakpoint-limit 1
(gdb) set remote hardware-watchpoint-limit 1
(gdb) set remote interrupt-on-connect on
(gdb) set remote kill-packet off
(gdb) set remote symbol-lookup-packet off
(gdb) set remote verbose-resume-packet off
(gdb) mem 0x20000000 0x3fefffff ro cache
(gdb) mem 0x3ff00000 0x3fffffff rw
(gdb) mem 0x40000000 0x400fffff ro cache
```

(continues on next page)

(continued from previous page)

```
(gdb) mem 0x40100000 0x4013ffff rw cache
(gdb) mem 0x40140000 0x5fffffff ro cache
(gdb) mem 0x60000000 0x60001fff rw
(gdb) set serial baud 115200
(gdb)
```

And tell GDB where the debugging info ELF file is located:

```
(gdb) file /tmp/arduino_build_257110/testgdb.ino.elf
Reading symbols from /tmp/arduino_build_257110/testgdb.ino.elf...done.
```

Now, connect to the running ESP8266:

```
(gdb)     target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
0x40000f68 in ?? ()
(gdb)
```

Don't worry that GDB doesn't know what is at our present address, we broke into the code at a random spot and we could be in an interrupt, in the ROM, or elsewhere. The important bit is that we're now connected and two things will now happen: we can debug, and the app's regular serial output will be displayed on the GDB console..

Continue the running app to see the serial output:

```
(gdb) cont
Continuing.
74
75
76
77
...
```

The app is back running and we can stop it at any time using `Ctrl-C`:

```
113
^C
Program received signal SIGINT, Interrupt.
0x40000f68 in ?? ()
(gdb)
```

At this point we can set a breakpoint on the main `loop()` and restart to get into our own code:

```
(gdb) break loop
Breakpoint 1 at 0x40202e33: file /home/earle/Arduino/sketch_dec26a/sketch_dec26a.ino,␣
↪line 10.
(gdb) cont
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.
bcn_timout,ap_probe_send_start

Breakpoint 1, loop () at /home/earle/Arduino/sketch_dec26a/sketch_dec26a.ino:10
10  void loop()
(gdb)
```

Let's examine the local variable:

```
(gdb) next
loop () at /home/earle/Arduino/sketch_dec26a/sketch_dec26a.ino:13
13      Serial.printf("%d\n", cnt++);
(gdb) print cnt
$1 = 114
(gdb)
```

And change it:

```
$2 = 114
(gdb) set cnt = 2000
(gdb) print cnt
$3 = 2000
(gdb)
```

And restart the app and see our changes take effect:

```
(gdb) cont
Continuing.
2000
Breakpoint 1, loop () at /home/earle/Arduino/sketch_dec26a/sketch_dec26a.ino:10
10  void loop() {
(gdb) cont
Continuing.
2001
Breakpoint 1, loop () at /home/earle/Arduino/sketch_dec26a/sketch_dec26a.ino:10
10  void loop() {
(gdb)
```

Looks like we left the breakpoint on loop(), let's get rid of it and try again:

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) cont
Continuing.
2002
2003
2004
2005
2006
....
```

At this point we can exit GDB with `quit` or do further debugging.

9.5 ESP8266 Hardware Debugging Limitations

The ESP8266 only supports a single hardware breakpoint and a single hardware data watchpoint. This means only one breakpoint in user code is allowed at any time. Consider using the `thb` (temporary hardware breakpoint) command in GDB while debugging instead of the more common `break` command, since `thb` will remove the breakpoint once it is reached automatically and save you some trouble.

Because of the single hardware breakpoint limitation, you must pay careful attention to the output from `gdb` when you set a breakpoint. If your breakpoint expression matches multiple locations, as in this example:

```
(gdb) break loop
Breakpoint 1 at 0x40202c84: loop. (2 locations)
```

Then you will be unable to `continue`:

```
(gdb) cont
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.
Warning:
Cannot insert hardware breakpoint 1.
Could not insert hardware breakpoints:
You may have requested too many hardware breakpoints/watchpoints.
```

You can resolve this situation by deleting the previous breakpoint and using a more specific breakpoint expression:

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) break mysketch.ino:loop
Breakpoint 2 at 0x40202c84: file .../mysketch.ino, line 113.
```

CHAPTER

TEN

MMU - ADJUST THE RATIO OF ICACHE TO IRAM

10.1 Overview

The ESP8266 has a total of 64K of instruction memory, IRAM. This 64K of IRAM is composed of one dedicated 32K block of IRAM and two 16K blocks of IRAM. The last two 16K blocks of IRAM are flexible in the sense that it can be used as a transparent cache for external flash memory. These blocks can either be used for IRAM or an instruction cache for executing code out of flash, ICACHE.

The code generated for a sketch is divided up into two groups, ICACHE and IRAM. IRAM offers faster execution. It is used for interrupt service routines, exception handling, and time-critical code. The ICACHE allows for the execution of up to 1MB of code stored in flash. On a cache miss, a delay occurs as the instructions are read from flash via the SPI bus.

There is 98KB of DRAM space. This memory can be accessed as byte, short, or a 32-bit word. Access must be aligned according to the data type size. A 16bit short must be on a multiple of 2-byte address boundary. Likewise, a 32-bit word must be on a multiple of 4-byte address boundary. In contrast, data access in IRAM or ICACHE must always be a full 32-bit word and aligned. We will discuss a non32-bit exception handler for this later.

10.2 Option Summary

10.2.1 The Arduino IDE Tools menu option, `MMU` has the following selections:

1. `32KB cache + 32KB IRAM (balanced)`
    - This is the legacy ratio.
    - Try this option 1st.
2. `16KB cache + 48KB IRAM (IRAM)`
    - With just 16KB cache, execution of code out of flash may be slowed by more cache misses when compared to 32KB. The slowness will vary with the sketch.
    - Use this if you need a little more IRAM space, and you have enough DRAM space.
3. `16KB cache + 48KB IRAM and 2nd Heap (shared)`
    - This option builds on the previous option and creates a 2nd Heap made with IRAM.
    - The 2nd Heap size will vary with free IRAM.
    - This option is flexible. IRAM usage for code can overflow into the additional 16KB IRAM region, shrinking the 2nd Heap below 16KB. Or IRAM can be under 32KB, allowing the 2nd Heap to be larger than 16KB.
    - Installs a Non-32-Bit Access handler for IRAM. This allows for byte and 16-bit aligned short access.

- This 2nd Heap is supported by the standard `malloc` APIs.
- Heap selection is handled through a `HeapSelect` class. This allows a specific heap selection for the duration of a scope.
- Use this option, if you are still running out of DRAM space after you have moved as many of your constant strings/data elements that you can to PROGMEM.

4. `16KB cache + 32KB IRAM + 16KB 2nd Heap (not shared)`
   - Not managed by the `umm_malloc` heap library
   - If required, non-32-Bit Access for IRAM must be enabled separately.
   - Enables a 16KB block of unmanaged IRAM memory
   - Data persist across reboots, but not deep sleep.
   - Works well for when you need a simple large chunk of memory. This option will reduce the resources required to support a shared 2nd Heap.

MMU related build defines and possible values. These values change as indicated with the menu options above:

#define	balanced	IRAM	shared (IRAM and Heap)	not shared (IRAM and Heap)
MMU_ IRAM_SIZE	0x8000	0xC000	0xC000	0x8000
MMU_IC ACHE_SIZE	0x8000	0x4000	0x4000	0x4000
MMU_ IRAM_HEAP	–	–	defined, e nablesu mm_malloc	–
MMU _SEC_HEAP	–	**	**	0 x40108000
MMU_SEC_ HEAP_SIZE	–	**	**	0x4000

** This define is to an inline function that calculates the value, based on unused code space, requires `#include <mmu_iram.h>`.

10.2.2 The Arduino IDE Tools menu option, `Non-32-Bit Access` has the following selections:

- `Use pgm_read macros for IRAM/PROGMEM`
- `Byte/Word access to IRAM/PROGMEM (very slow)`
  - This option adds a non32-bit exception handler to your build.
  - Handles read/writes to IRAM and reads to ICACHE.
  - Supports short and byte access to IRAM
  - Not recommended for high-frequency access data, use DRAM if you can.
  - Expect it to be slower than DRAM, each character access, will require a complete save and restore of all 16+ registers.
  - Processing an exception uses 256 bytes of stack space just to get started. The actual handler will add a little more.
  - This option is implicitly enabled and required when you select MMU option `16KB cache + 48KB IRAM and 2nd Heap (shared)`.

IRAM, unlike DRAM, must be accessed as aligned full 32-bit words, no byte or short access. The pgm_read macros are an option; however, the store operation remains an issue. For a block copy, ets_memcpy appears to work well as long as the byte count is rounded up to be evenly divided by 4, and source and destination addresses are 4 bytes aligned.

A word of caution, I have seen one case with the new toolchain 10.1 where code that reads a 32-bit word to extract a byte was optimized away to be a byte read. Using `volatile` on the pointer stopped the over-optimization.

To get a sense of how memory access time is effected, see examples `MMU48K` and `irammem` in `ESP8266`.

NON-OS SDK v3.0.0 and above have builtin support for Non-32-Bit Access. Selecting `Byte/Word access to IRAM/PROGMEM` will override the builtin version with ours. However, there is no known reason to do this other than debugging.

10.3 Miscellaneous

10.3.1 For calls to `umm_malloc` with interrupts disabled.

- `malloc` will always allocate from the `DRAM` heap when called with interrupts disabled.
  - `realloc` with a NULL pointer will use `malloc` and return a `DRAM` heap allocation. Note, calling `realloc` with interrupts disabled is **not** officially supported. You are on your own if you do this.
- If you must use IRAM memory in your ISR, allocate the memory in your init code. To reduce the time spent in the ISR, avoid non32-bit access that would trigger the exception handler. For short or byte access, consider using the inline functions described in section "Performance Functions" below.

10.3.2 How to Select Heap

The `MMU` selection `16KB cache + 48KB IRAM and 2nd Heap (shared)` allows you to use the standard heap API function calls (`malloc`, `calloc`, `free`, ...). to allocate memory from DRAM or IRAM. This selection can be made by instantiating the class `HeapSelectIram` or `HeapSelectDram`. The usage is similar to that of the `InterruptLock` class. The default/initial heap source is DRAM. The class is in `umm_malloc/umm_heap_select.h`.

```
...
  char *bufferDram;
  bufferDram = (char *)malloc(33);
  char *bufferIram;
  {
      HeapSelectIram ephemeral;
      bufferIram = (char *)malloc(33);
  }
...
  free(bufferIram);
  free(bufferDram);
...
```

`free` will always return memory to the correct heap. There is no need for tracking and selecting before freeing.

`realloc` with a non-NULL pointer will always resize the allocation from the original heap it was allocated from. When the supplied pointer is NULL, then the current heap selection is used.

Low-level primitives for selecting a heap. These are used by the above Classes:

- `umm_get_current_heap_id()`
- `umm_set_heap_by_id( ID value )`

- Possible ID values
  - `UMM_HEAP_DRAM`
  - `UMM_HEAP_IRAM`

Also, an alternate stack select method API is available. This is not as easy as the class method; however, for some small set of cases, it may provide some additional control:

- `ESP.setIramHeap()` Pushes current heap ID onto a stack and sets Heap API for an IRAM selection.
- `ESP.setDramHeap()` Pushes current heap ID onto a stack and sets Heap API for a DRAM selection.
- `ESP.resetHeap()` Restores previously pushed heap. ### Identify Memory

These always inlined functions can be used to determine the resource of a pointer:

```
bool mmu_is_iram(const void *addr);
bool mmu_is_dram(const void *addr);
bool mmu_is_icache(const void *addr);
```

10.3.3 Performance Functions

While these always inlined functions, will bypass the need for the exception handler reducing execution time and stack use, it comes at the cost of increased code size.

These are an alternative to the `pgm_read` macros for reading from IRAM. When compiled with 'Debug Level: core' range checks are performed on the pointer value to make sure you are reading from the address range of IRAM, DRAM, or ICACHE.

```
uint8_t mmu_get_uint8(const void *p8);
uint16_t mmu_get_uint16(const uint16_t *p16);
int16_t mmu_get_int16(const int16_t *p16);
```

While these functions are intended for writing to IRAM, they will work with DRAM. When compiled with 'Debug Level: core', range checks are performed on the pointer value to make sure you are writing to the address range of IRAM or DRAM.

```
uint8_t mmu_set_uint8(void *p8, const uint8_t val);
uint16_t mmu_set_uint16(uint16_t *p16, const uint16_t val);
int16_t mmu_set_int16(int16_t *p16, const int16_t val);
```

CHAPTER

ELEVEN

BOARDS

11.1 Generic ESP8266 Module

These modules come in different form factors and pinouts. See the page at ESP8266 community wiki for more info: ESP8266 Module Family.

Usually these modules have no bootstrapping resistors on board, insufficient decoupling capacitors, no voltage regulator, no reset circuit, and no USB-serial adapter. This makes using them somewhat tricky, compared to development boards which add these features.

In order to use these modules, make sure to observe the following:

- **Provide sufficient power to the module.** For stable use of the ESP8266 a power supply with 3.3V and >= 250mA is required. Using the power available from USB to Serial adapter is not recommended, these adapters typically do not supply enough current to run ESP8266 reliably in every situation. An external supply or regulator alongwith filtering capacitors is preferred.
- **Connect bootstrapping resistors** to GPIO0, GPIO2, GPIO15 according to the schematics below.
- **Put ESP8266 into bootloader mode** before uploading code.

11.2 Serial Adapter

There are many different USB to Serial adapters / boards. To be able to put ESP8266 into bootloader mode using serial handshaking lines, you need the adapter which breaks out RTS and DTR outputs. CTS and DSR are not useful for upload (they are inputs). Make sure the adapter can work with 3.3V IO voltage: it should have a jumper or a switch to select between 5V and 3.3V, or be marked as 3.3V only.

Adapters based around the following ICs should work:

- FT232RL
- CP2102
- CH340G

PL2303-based adapters are known not to work on Mac OS X. See https://github.com/igrr/esptool-ck/issues/9 for more info.

11.3 Minimal Hardware Setup for Bootloading and Usage

PIN	Resistor	Serial Adapter
VCC		VCC (3.3V)
GND		GND
TX or GPIO2*		RX
RX		TX
GPIO0	PullUp	DTR
Reset*	PullUp	RTS
GPIO15*	PullDown	
CH_PD	PullUp	

- Note
- GPIO15 is also named MTDO
- Reset is also named RSBT or REST (adding PullUp improves the stability of the module)
- GPIO2 is alternative TX for the boot loader mode
- **Directly connecting a pin to VCC or GND is not a substitute for a PullUp or PullDown resistor, doing this can break upload management and the serial console, instability has also been noted in some cases.**

11.4 ESP to Serial

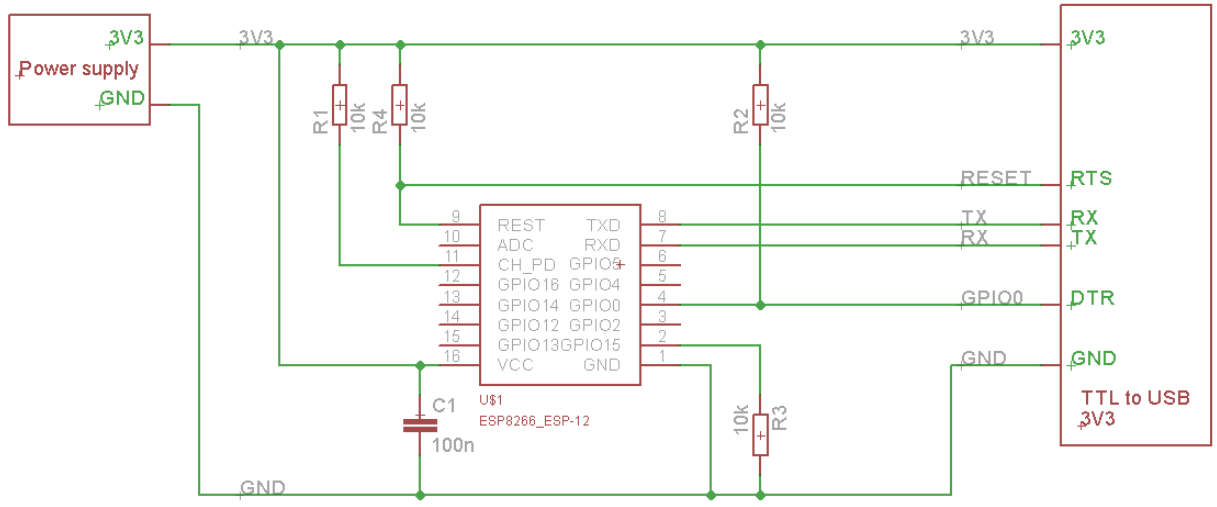


Fig. 1: ESP to Serial

11.4.1 Minimal Hardware Setup for Bootloading only

ESPxx Hardware

PIN	Resistor	Serial Adapter
VCC		VCC (3.3V)
GND		GND
TX or GPIO2		RX
RX		TX
GPIO0		GND
Reset		RTS*
GPIO15	PullDown	
CH_PD	PullUp	

- Note
- if no RTS is used a manual power toggle is needed

11.4.2 Minimal Hardware Setup for Running only

ESPxx Hardware

PIN	Resistor	Power supply
VCC		VCC (3.3V)
GND		GND
GPIO0	PullUp	
GPIO15	PullDown	
CH_PD	PullUp	

11.5 Minimal

11.6 Improved Stability

11.7 Boot Messages and Modes

The ESP module checks at every boot the Pins 0, 2 and 15. based on them its boots in different 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)

at startup the ESP prints out the current boot mode example:

```
rst cause:2, boot mode:(3,6)
```

note: - GPIO2 is used as TX output and the internal Pullup is enabled on boot.

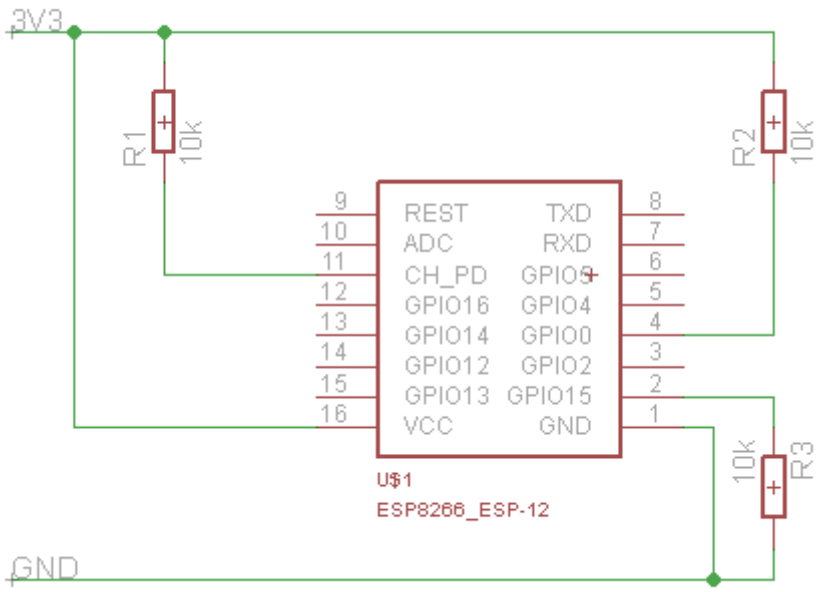


Fig. 2: ESP min

11.7.1 rst cause

Number	Description
0	unknown
1	normal boot
2	reset pin
3	software reset
4	watchdog reset

11.7.2 boot mode

the first value respects the pin setup of the Pins 0, 2 and 15.

Number	GPIO15	GPIO0	GPIO2	Mode
0	0V	0V	0V	Not valid
1	0V	0V	3.3V	Uart
2	0V	3.3V	0V	Not valid
3	0V	3.3V	3.3V	Flash
4	3.3V	0V	0V	SDIO
5	3.3V	0V	3.3V	SDIO
6	3.3V	3.3V	0V	SDIO
7	3.3V	3.3V	3.3V	SDIO

note: - number = ((GPIO15 << 2) | (GPIO0 << 1) | GPIO2);

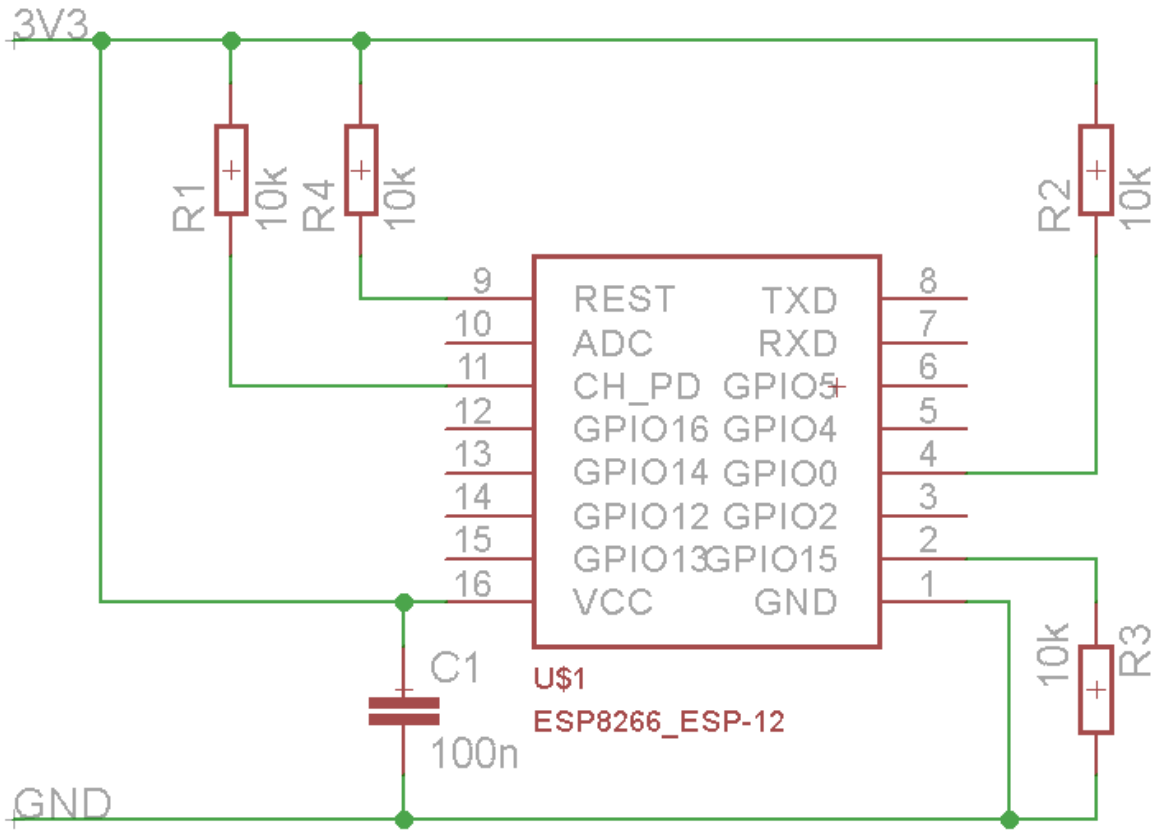


Fig. 3: ESP improved stability

11.8 Generic ESP8285 Module

ESP8285 (datasheet) is a multi-chip package which contains ESP8266 and 1MB flash. All points related to bootstrapping resistors and recommended circuits listed above apply to ESP8285 as well.

Note that since ESP8285 has SPI flash memory internally connected in DOUT mode, pins 9 and 10 may be used as GPIO / I2C / PWM pins.

11.9 Lifely Agrumino Lemon v4

Procuct page https://www.lifely.cc

This Board "Lifely Agrumino Lemon" is based with WT8266-S1 core with WiFi 2,4Ghz and 2MB of Flash. Power Micro usb power cable, Lir2450 rechargeable battery (or not rechargeable)or with JST connector in the back board Max 6 Vin Libraries and examples Download libraries from: Official Arduino Ide, our website https://www.lifely.cc or https://github.com/lifely-cc/ Full pinout and PDF for setup here https://www.lifely.cc our libraries is OpenSource

11.10 ESPDuino (ESP-13 Module)

TODO

11.11 Adafruit Feather HUZZAH ESP8266

The Adafruit Feather HUZZAH ESP8266 is an Arduino-compatible Wi-Fi development board powered by Ai-Thinker's ESP-12S, clocked at 80 MHz at 3.3V logic. A high-quality SiLabs CP2104 USB-Serial chip is included so that you can upload code at a blistering 921600 baud for fast development time. It also has auto-reset so no noodling with pins and reset button pressings. A 3.7V Lithium polymer battery connector is included, making it ideal for portable projects. The Adafruit Feather HUZZAH ESP8266 will automatically recharge a connected battery when USB power is available.

Product page: https://www.adafruit.com/product/2821

11.12 WiFi Kit 8

The Heltec WiFi Kit 8 is an Arduino-compatible Wi-Fi development board powered by Ai-Thinker's ESP-12S, clocked at 80 MHz at 3.3V logic. A high-quality SiLabs CP2104 USB-Serial chip is included so that you can upload code at a blistering 921600 baud for fast development time. It also has auto-reset so no noodling with pins and reset button pressings. A 3.7V Lithium polymer battery connector is included, making it ideal for portable projects. The Heltec WiFi Kit 8 will automatically recharge a connected battery when USB power is available.

Product page: https://github.com/Heltec-Aaron-Lee/WiFi_Kit_series

11.13 Invent One

The Invent One is an Arduino-compatible Wi-Fi development board powered by Ai-Thinker's ESP-12F, clocked at 80 MHz at 3.3V logic. It has an onboard ADC (PCF8591) so that you can have multiple analog inputs to work with. More information can be found here: https://blog.inventone.ng

Product page: https://inventone.ng

11.14 XinaBox CW01

The XinaBox CW01(ESP8266) is an Arduino-compatible Wi-Fi development board powered by an ESP-12F, clocked at 80 MHz at 3.3V logic. The CW01 has an onboard RGB LED and 3 xBUS connection ports.

Product page: https://xinabox.cc/products/CW01

11.15 ESPresso Lite 1.0

ESPresso Lite 1.0 (beta version) is an Arduino-compatible Wi-Fi development board powered by Espressif System's own ESP8266 WROOM-02 module. It has breadboard-friendly breakout pins with in-built LED, two reset/flash buttons and a user programmable button . The operating voltage is 3.3VDC, regulated with 800mA maximum current. Special distinctive features include on-board I2C pads that allow direct connection to OLED LCD and sensor boards.

11.16 ESPresso Lite 2.0

ESPresso Lite 2.0 is an Arduino-compatible Wi-Fi development board based on an earlier V1 (beta version). Re-designed together with Cytron Technologies, the newly-revised ESPresso Lite V2.0 features the auto-load/auto-program function, eliminating the previous need to reset the board manually before flashing a new program. It also feature two user programmable side buttons and a reset button. The special distinctive features of on-board pads for I2C sensor and actuator is retained.

11.17 Phoenix 1.0

Product page: http://www.espert.co

11.18 Phoenix 2.0

Product page: http://www.espert.co

11.19 NodeMCU 0.9 (ESP-12 Module)

11.19.1 Pin mapping

Pin numbers written on the board itself do not correspond to ESP8266 GPIO pin numbers. Constants are defined to make using this board easier:

```
static const uint8_t D0   = 16;
static const uint8_t D1   = 5;
static const uint8_t D2   = 4;
static const uint8_t D3   = 0;
static const uint8_t D4   = 2;
static const uint8_t D5   = 14;
static const uint8_t D6   = 12;
static const uint8_t D7   = 13;
static const uint8_t D8   = 15;
static const uint8_t D9   = 3;
static const uint8_t D10  = 1;
```

If you want to use NodeMCU pin 5, use D5 for pin number, and it will be translated to 'real' GPIO pin 14.

11.20 NodeMCU 1.0 (ESP-12E Module)

This module is sold under many names for around $6.50 on AliExpress and it's one of the cheapest, fully integrated ESP8266 solutions.

It's an open hardware design with an ESP-12E core and 4 MB of SPI flash.

According to the manufacturer, "with a micro USB cable, you can connect NodeMCU devkit to your laptop and flash it without any trouble". This is more or less true: the board comes with a CP2102 onboard USB to serial adapter which just works, well, the majority of the time. Sometimes flashing fails and you have to reset the board by holding down FLASH + RST, then releasing FLASH, then releasing RST. This forces the CP2102 device to power cycle and to be re-numbered by Linux.

The board also features a NCP1117 voltage regulator, a blue LED on GPIO16 and a 220k/100k Ohm voltage divider on the ADC input pin. The ESP-12E usually has a led connected on GPIO2.

Full pinout and PDF schematics can be found here

11.21 Olimex MOD-WIFI-ESP8266(-DEV)

This board comes with 2 MB of SPI flash and optional accessories (e.g. evaluation board ESP8266-EVB or BAT-BOX for batteries).

The basic module has three solder jumpers that allow you to switch the operating mode between SDIO, UART and FLASH.

The board is shipped for FLASH operation mode, with jumpers TD0JP=0, IO0JP=1, IO2JP=1.

Since jumper IO0JP is tied to GPIO0, which is PIN 21, you'll have to ground it before programming with a USB to serial adapter and reset the board by power cycling it.

UART pins for programming and serial I/O are GPIO1 (TXD, pin 3) and GPIO3 (RXD, pin 4).

You can find the board schematics here

11.22 SparkFun ESP8266 Thing

Product page: https://www.sparkfun.com/products/13231

11.23 SparkFun ESP8266 Thing Dev

Product page: https://www.sparkfun.com/products/13711

11.24 SparkFun Blynk Board

Product page: https://www.sparkfun.com/products/13794

11.25 SweetPea ESP-210

TODO

11.26 LOLIN(WEMOS) D1 R2 & mini

Product page: https://www.wemos.cc/

11.27 LOLIN(WEMOS) D1 ESP-WROOM-02

No real product pages. See: https://www.instructables.com/How-to-Use-Wemos-ESP-Wroom-02-D1-Mini-WiFi-Module-/ or https://www.arduino-tech.com/wemos-esp-wroom-02-mainboard-d1-mini-wifi-module-esp826618650-battery/

11.28 LOLIN(WEMOS) D1 mini (clone)

Clone variant of the LOLIN(WEMOS) D1 mini board, with enabled flash-mode menu, DOUT selected by default.

Product page of the preferred official board: https://www.wemos.cc/

11.29 LOLIN(WEMOS) D1 mini Pro

Product page: https://www.wemos.cc/

11.30 LOLIN(WEMOS) D1 mini Lite

11.30.1 Parameters in Arduino IDE:

- Card: "WEMOS D1 Mini Lite"
- Flash Size: "1M (512K FS)"
- CPU Frequency: "80 Mhz"

11.30.2 Power:

- 5V pin : 4.7V 500mA output when the board is powered by USB ; 3.5V-6V input
- 3V3 pin : 3.3V 500mA regulated output
- Digital pins : 3.3V 30mA.

11.30.3 links:

- Product page: https://www.wemos.cc/
- Board schematic: https://wiki.wemos.cc/_media/products:d1:sch_d1_mini_lite_v1.0.0.pdf
- ESP8285 datasheet: https://www.espressif.com/sites/default/files/0a-esp8285_datasheet_en_v1.0_20160422.pdf
- Voltage regulator datasheet: http://pdf-datasheet.datasheet.netdna-cdn.com/pdf-down/M/E/6/ME6211-Microne.pdf

11.31 LOLIN(WeMos) D1 R1

Product page: https://www.wemos.cc/

11.32 ESPino (ESP-12 Module)

ESPino integrates the ESP-12 module with a 3.3v regulator, CP2104 USB-Serial bridge and a micro USB connector for easy programming. It is designed for fitting in a breadboard and has an RGB Led and two buttons for easy prototyping.

For more information about the hardware, pinout diagram and programming procedures, please see the datasheet.

Product page: http://www.espino.io/en

11.33 ThaiEasyElec's ESPino

ESPino by ThaiEasyElec using WROOM-02 module from Espressif Systems with 4 MB Flash.

We will update an English description soon. - Product page: http://thaieasyelec.com/products/wireless-modules/wifi-modules/espino-wifi-development-board-detail.html - Schematics: www.thaieasyelec.com/downloads/ETEE052/ETEE052_ESPino_Schematic.pdf - Dimensions: http://thaieasyelec.com/downloads/ETEE052/ETEE052_ESPino_Dimension.pdf - Pinouts: http://thaieasyelec.com/downloads/ETEE052/ETEE052_ESPino_User_Manual_TH_v1_0_20160204.pdf (Please see pg. 8)

11.34 WifInfo

WifInfo integrates the ESP-12 or ESP-07+Ext antenna module with a 3.3v regulator and the hardware to be able to measure French telemetry issue from ERDF powering meter serial output. It has a USB connector for powering, an RGB WS2812 Led, 4 pins I2C connector to fit OLED or sensor, and two buttons + FTDI connector and auto reset feature.

For more information, please see WifInfo related blog entries, github and community forum.

11.35 Arduino

TODO

11.36 4D Systems gen4 IoD Range

gen4-IoD Range of ESP8266 powered Display Modules by 4D Systems.

2.4", 2.8" and 3.2" TFT LCD with uSD card socket and Resistive Touch. Chip Antenna + uFL Connector.

Datasheet and associated downloads can be found on the 4D Systems product page.

The gen4-IoD range can be programmed using the Arduino IDE and also the 4D Systems Workshop4 IDE, which incorporates many additional graphics benefits. GFX4d library is available, along with a number of demo applications.

- Product page: https://4dsystems.com.au/products/iot-display-modules

11.37 Digistump Oak

The Oak requires an *Serial Adapter* for a serial connection or flashing; its micro USB port is only for power.

To make a serial connection, wire the adapter's **TX to P3**, **RX to P4**, and **GND** to **GND**. Supply 3.3v from the serial adapter if not already powered via USB.

To put the board into bootloader mode, configure a serial connection as above, connect **P2 to GND**, then re-apply power. Once flashing is complete, remove the connection from P2 to GND, then re-apply power to boot into normal mode.

11.38 WiFiduino

Product page: https://wifiduino.com/esp8266

11.39 Amperka WiFi Slot

Product page: http://wiki.amperka.ru/wifi-slot

11.40 Seeed Wio Link

Wio Link is designed to simplify your IoT development. It is an ESP8266 based open-source Wi-Fi development board to create IoT applications by virtualizing plug-n-play modules to RESTful APIs with mobile APPs. Wio Link is also compatible with the Arduino IDE.

Please DO NOTICE that you MUST pull up pin 15 to enable the power for Grove ports, the board is designed like this for the purpose of peripherals power management.

Product page: https://www.seeedstudio.com/Wio-Link-p-2604.html

11.41 ESPectro Core

ESPectro Core is ESP8266 development board as the culmination of our 3+ year experience in exploring and developing products with ESP8266 MCU.

Initially designed for kids in mind, everybody should be able to use it. Yet it's still hacker-friendly as we break out all ESP8266 ESP-12F pins.

More details at https://shop.makestro.com/product/espectrocore/

11.42 Schirmilabs Eduino WiFi

Eduino WiFi is an Arduino-compatible DIY WiFi development board using an ESP-12 module

Product page: https://schirmilabs.de/?page_id=165

11.43 ITEAD Sonoff

ESP8266 based devices from ITEAD: Sonoff SV, Sonoff TH, Sonoff Basic, and Sonoff S20

These are not development boards. The development process is inconvenient with these devices. When flashing firmware you will need a Serial Adapter to connect it to your computer.

> Most of these devices, during normal operation, are connected to *wall power (AKA Mains Electricity)*. **NEVER** try to flash these devices when connected to *wall power*. **ALWAYS** have them disconnected from *wall power* when connecting them to your computer. Your life may depend on it!

When flashing you will need to hold down the push button connected to the GPIO0 pin, while powering up with a safe 3.3 Volt source. Some USB Serial Adapters may supply enough power to handle flashing; however, it many may not supply enough power to handle the activities when the device reboots.

More product details at the bottom of https://www.itead.cc/wiki/Product/

11.44 DOIT ESP-Mx DevKit (ESP8285)

DOIT ESP-Mx DevKit - This is a development board by DOIT, with a DOIT ESP-Mx module (datasheet) using a ESP8285 Chip. With the DOIT ESP-Mx module, GPIO pins 9 and 10 are not available. The DOIT ESP-Mx DevKit board has a red power LED and a blue LED connected to GPIO16 and is active low to turn on. It uses a CH340C, USB to Serial converter chip.

ESP8285 (datasheet) is a multi-chip package which contains ESP8266 and 1MB flash.

CHAPTER

TWELVE

FAQ

The purpose of this FAQ / Troubleshooting is to respond to questions commonly asked in Issues section and on ESP8266 Community forum.

Where possible we are going right to the answer and provide it within one or two paragraphs. If it takes more than that, you will see a link to "Read more" details.

Please feel free to contribute if you believe that some frequent issues are not covered below.

12.1 I am getting "espcomm_sync failed" error when trying to upload my ESP. How to resolve this issue?

This message indicates issue with uploading ESP module over a serial connection. There are couple of possible causes, that depend on the type of your module, if you use separate USB to serial converter.

Read more.

12.2 Why esptool is not listed in "Programmer" menu? How do I upload ESP without it?

Do not worry about "Programmer" menu of Arduino IDE. It doesn't matter what is selected in it — upload now always defaults to using esptool.

Ref. #138, #653 and #739.

12.3 My ESP crashes running some code. How to troubleshoot it?

The code may crash because of s/w bug or issue with your h/w. Before entering an issue report, please perform initial troubleshooting.

Read more.

12.4 How can I get some extra KBs in flash ?

- Using `*printf()` with floats is enabled by default. Some KBs of flash can be saved by using the option `--nofloat` with the boards generator:

  `./tools/boards.txt.py --nofloat --boardsgen`

- Use the debug level option `NoAssert-NDEBUG` (in the Tools menu)

Read more.

12.5 About WPS

From release 2.4.2 and ahead, not using WPS will give an extra ~4.5KB in heap.

In release 2.4.2 only, WPS is disabled by default and the board generator is required to enable it:

`./tools/boards.txt.py --allowWPS --boardsgen`

Read more.

For platformIO (and maybe other build environments), you will also need to add the build flag: -D NO_EXTRA_4K_HEAP

This manual selection is not needed starting from 2.5.0 (and in git version). WPS is always available, and not using it will give an extra ~4.5KB compared to releases until 2.4.1 included.

12.6 This Arduino library doesn't work on ESP. How do I make it work?

You would like to use this Arduino library with ESP8266 and it does not perform. It is not listed among libraries verified to work with ESP8266.

Read more.

12.7 In the IDE, for ESP-12E that has 4M flash, I can choose 4M (1M FS) or 4M (3M FS). No matter what I select, the IDE tells me the maximum code space is about 1M. Where does my flash go?

The reason we cannot have more than 1MB of code in flash has to do with a hardware limitation. Flash cache hardware on the ESP8266 only allows mapping 1MB of code into the CPU address space at any given time. You can switch mapping offset, so technically you can have more than 1MB total, but switching such "banks" on the fly is not easy and efficient, so we don't bother doing that. Besides, no one has so far complained about 1MB of code space being insufficient for practical purposes.

The option to choose 3M or 1M filesystem is to optimize the upload time. Uploading 3MB takes a long time so sometimes you can just use 1MB. Other 2MB of flash can still be used with `ESP.flashRead` and `ESP.flashWrite` APIs if necessary.

12.8 I have observed a case when ESP.restart() doesn't work. What is the reason for that?

You will see this issue only if serial upload was not followed by a physical reset (e.g. power-on reset). For a device being in that state `ESP.restart` will not work. Apparently the issue is caused by one of internal registers not being properly updated until physical reset. This issue concerns only serial uploads. OTA uploads are not affected. If you are using `ESP.restart`, the work around is to reset ESP once after each serial upload.

Ref. #1017, #1107, #1782

12.9 How to resolve "Board generic (platform esp8266, package esp8266) is unknown" error?

This error may pop up after switching between staging and stable esp8266 / Arduino package installations, or after upgrading the package version Read more.

12.10 How to clear TCP PCBs in time-wait state ?

This is not needed anymore:

PCBs in time-wait state are limited to 5 and removed when that number is exceeded.

Ref. https://github.com/d-a-v/esp82xx-nonos-linklayer/commit/420960dfc0dbe07114f7364845836ac333bc84f7

For reference:

Time-wait PCB state helps TCP not confusing two consecutive connections with the same (s-ip,s-port,d-ip,d-port) when the first is already closed but still having duplicate packets lost in internet arriving later during the second. Artificially clearing them is a workaround to help saving precious heap.

```
// no need for #include
struct tcp_pcb;
extern struct tcp_pcb* tcp_tw_pcbs;
extern "C" void tcp_abort (struct tcp_pcb* pcb);

void tcpCleanup (void) {
  while (tcp_tw_pcbs)
    tcp_abort(tcp_tw_pcbs);
}
```

Ref. #1923

12.11 Why is there a board generator and what about it ?

The board generator is a python script originally intended to ease the Arduino IDE's *boards.txt* configuration file about the multitude of available boards, especially when common parameters have to be updated for all of them.

This script is also used to manage uncommon options that are currently not available in the IDE menu.

Read more.

12.12 My WiFi won't reconnect after deep sleep using `WAKE_RF_DISABLED`

When you implement deep sleep using `WAKE_RF_DISABLED`, this forces what appears to be a bare metal disabling of WiFi functionality, which is not restored using `WiFi.forceSleepWake()` or `WiFi.mode(WIFI_STA)`. If you need to implement deep sleep with `WAKE_RF_DISABLED` and later connect to WiFi, you will need to implement an additional (short) deep sleep using `WAKE_RF_DEFAULT`.

Ref. #3072

12.13 My WiFi was previously automatically connected right after booting, but isn't anymore

This was WiFi persistence. Starting from version 3 of this core, WiFi is indeed off at boot and is powered on only when starting to be used with the regular API.

Read more at former WiFi persistent mode.

12.14 How to resolve "undefined reference to `flashinit`" error ?

Please read *flash layout* documentation entry.

12.15 How to specify global build defines and options?

By using a uniquely named *.h* file, macro definitions can be created and globally used. Additionally, compiler command-line options can be embedded in this file as a unique block comment.

Read more.

CHAPTER

THIRTEEN

EXCEPTION CAUSES (EXCCAUSE)

EXCCAUSE Code	Cause Name	Cause Description
0	IllegalInstructionCause	Illegal instruction
1	SyscallCause	SYSCALL instruction
2	InstructionFetchErrorCause	Processor internal physical address or data error during instruction fetch
3	LoadStoreErrorCause	Processor internal physical address or data error during load or store
4	Level1InterruptCause	Level-1 interrupt as indicated by set level-1 bits in the INTERRUPT register
5	AllocaCause	MOVSP instruction, if caller's registers are not in the register file
6	IntegerDivideByZeroCause	QUOS, QUOU, REMS, or REMU divisor operand is zero
7	Reserved for Tensilica	
8	PrivilegedCause	Attempt to execute a privileged operation when CRING != 0
9	LoadStoreAlignmentCause	Load or store to an unaligned address
10..11	Reserved for Tensilica	
12	InstrPIFDateErrorCause	PIF data error during instruction fetch
13	LoadStorePIFDataErrorCause	Synchronous PIF data error during LoadStore access
14	InstrPIFAddrErrorCause	PIF address error during instruction fetch
15	LoadStorePIFAddrErrorCause	Synchronous PIF address error during LoadStore access
16	InstTLBMissCause	Error during Instruction TLB refill
17	InstTLBMultiHitCause	Multiple instruction TLB entries matched
18	InstFetchPrivilegeCause	An instruction fetch referenced a virtual address at a ring level less than CRIN
19	Reserved for Tensilica	
20	InstFetchProhibitedCause	An instruction fetch referenced a page mapped with an attribute that does not
21..23	Reserved for Tensilica	
24	LoadStoreTLBMissCause	Error during TLB refill for a load or store
25	LoadStoreTLBMultiHitCause	Multiple TLB entries matched for a load or store
26	LoadStorePrivilegeCause	A load or store referenced a virtual address at a ring level less than CRING
27	Reserved for Tensilica	
28	LoadProhibitedCause	A load referenced a page mapped with an attribute that does not permit loads
29	StoreProhibitedCause	A store referenced a page mapped with an attribute that does not permit
30..31	Reserved for Tensilica	
32..39	CoprocessornDisabled	Coprocessor n instruction when cpn disabled. n varies 0..7 as the cause varies
40..63	Reserved	

Infos from Xtensa Instruction Set Architecture (ISA) Reference Manual

CHAPTER

FOURTEEN

DEBUGGING

14.1 Introduction

Since 2.1.0-rc1 the core includes a Debugging feature that is controllable over the IDE menu.

The new menu points manage the real-time Debug messages.

14.1.1 Requirements

For usage of the debugging a Serial connection is required (Serial or Serial1).

The Serial Interface need to be initialized in the `setup()`.

Set the Serial baud rate as high as possible for your Hardware setup.

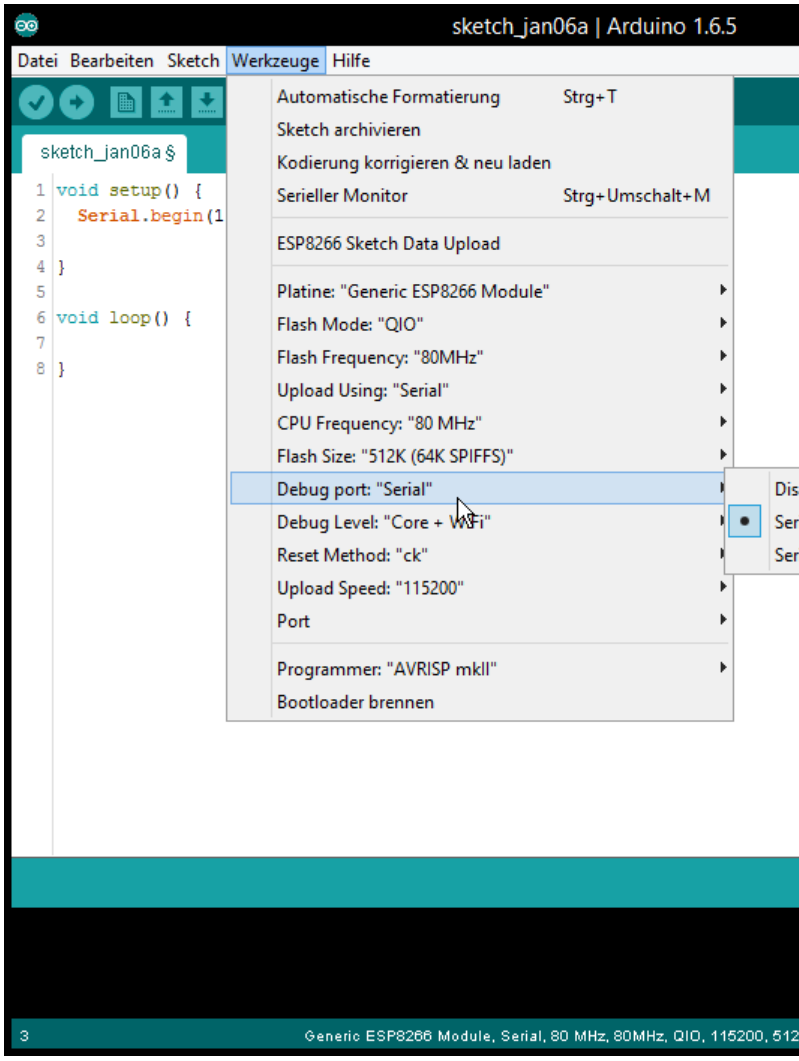
Minimum sketch to use debugging:

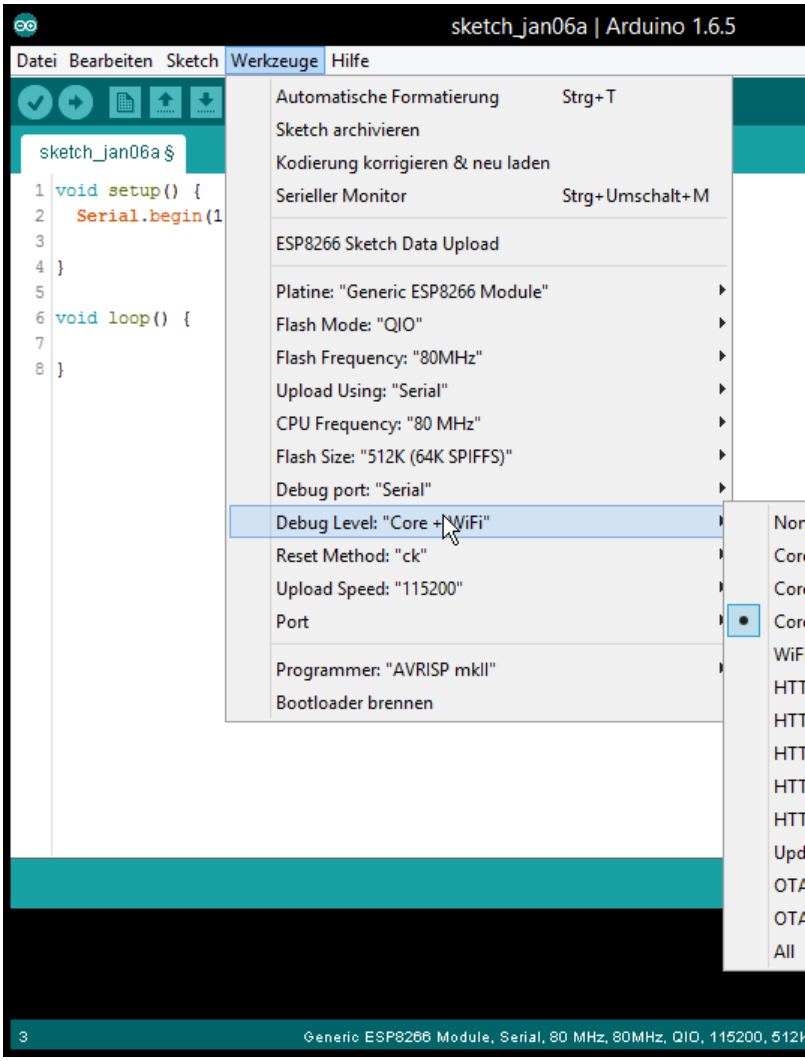
```
void setup() {
    Serial.begin(115200);
}

void loop() {
}
```

14.1.2 Usage

1. Select the Serial interface for the Debugging messages:





2. Select which type / level you want debug messages for:
3. Check if the Serial interface is initialized in `setup()` (see *Requirements*)
4. Flash sketch
5. Check the Serial Output

14.2 Information

It work with every sketch that enables the Serial interface that is selected as debug port.

The Serial interface can still be used normal in the Sketch.

The debug output is additional and will not disable any interface from usage in the sketch.

14.2.1 For Developers

For the debug handling uses defines.

The defined are set by command line.

Debug Port

The port has the define `DEBUG_ESP_PORT` possible value: - Disabled: define not existing - Serial: Serial - Serial1: Serial1

Debug Level

All defines for the different levels starts with `DEBUG_ESP_`

a full list can be found here in the boards.txt

Example for own debug messages

The debug messages will be only shown when the Debug Port in the IDE menu is set.

```
#ifdef DEBUG_ESP_PORT
#define DEBUG_MSG(...) DEBUG_ESP_PORT.printf( __VA_ARGS__ )
#else
#define DEBUG_MSG(...)
#endif

void setup() {
    Serial.begin(115200);

    delay(3000);
    DEBUG_MSG("bootup...\n");
}

void loop() {
    DEBUG_MSG("loop %d\n", millis());
    delay(1000);
}
```

CHAPTER

FIFTEEN

STACK DUMPS

15.1 Introduction

If the ESP crashes the Exception Cause will be shown and the current stack will be dumped.

Example:

```
Exception (0): epc1=0x402103f4 epc2=0x00000000 epc3=0x00000000 excvaddr=0x00000000␣
↪depc=0x00000000

ctx: sys
sp: 3ffffc10 end: 3fffffb0 offset: 01a0

>>>stack>>>
3ffffdb0:  40223e00 3fff6f50 00000010 60000600
3ffffdc0:  00000001 4021f774 3fffc250 4000050c
3ffffdd0:  400043d5 00000030 00000016 ffffffff
3ffffde0:  400044ab 3fffc718 3ffffed0 08000000
3ffffdf0:  60000200 08000000 00000003 00000000
3ffffe00:  0000ffff 00000001 04000002 003fd000
3ffffe10:  3fff7188 000003fd 3fff2564 00000030
3ffffe20:  40101709 00000008 00000008 00000020
3ffffe30:  c1948db3 394c5e70 7f2060f2 c6ba0c87
3ffffe40:  3fff7058 00000001 40238d41 3fff6ff0
3ffffe50:  3fff6f50 00000010 60000600 00000020
3ffffe60:  402301a8 3fff7098 3fff7014 40238c77
3ffffe70:  4022fb6c 40230ebe 3fff1a5b 3fff6f00
3ffffe80:  3ffffec8 00000010 40231061 3fff0f90
3ffffe90:  3fff6848 3ffed0c0 60000600 3fff6ae0
3ffffea0:  3fff0f90 3fff0f90 3fff6848 3fff6d40
3ffffeb0:  3fff28e8 40101233 d634fe1a fffeffff
3ffffec0:  00000001 00000000 4022d5d6 3fff6848
3ffffed0:  00000002 4000410f 3fff2394 3fff6848
3ffffee0:  3fffc718 40004a3c 000003fd 3fff7188
3ffffef0:  3fffc718 40101510 00000378 3fff1a5b
3fffff00:  000003fd 4021d2e7 00000378 000003ff
3fffff10:  00001000 4021d37d 3fff2564 000003ff
3fffff20:  000003fd 60000600 003fd000 3fff2564
3fffff30:  ffffff00 55aa55aa 00000312 0000001c
3fffff40:  0000001c 0000008a 0000006d 000003ff
3fffff50:  4021d224 3ffecf90 00000000 3ffed0c0
```

(continues on next page)

(continued from previous page)

```
3fffff60:  00000001 4021c2e9 00000003 3fff1238
3fffff70:  4021c071 3ffecf84 3ffecf30 0026a2b0
3fffff80:  4021c0b6 3fffdab0 00000000 3fffdcb0
3fffff90:  3ffecf40 3fffdab0 00000000 3fffdcc0
3fffffa0:  40000f49 40000f49 3fffdab0 40000f49
<<<stack<<<
```

The first number after `Exception` gives the cause of the reset. a full list of all causes can be found *here* the hex after are the stack dump.

15.1.1 Decode

It's possible to decode the Stack to readable information. You can get a copy and read about the Esp Exception Decoder tool.

For a troubleshooting example using the Exception Decoder Tool, read FAQ: My ESP Crashes.

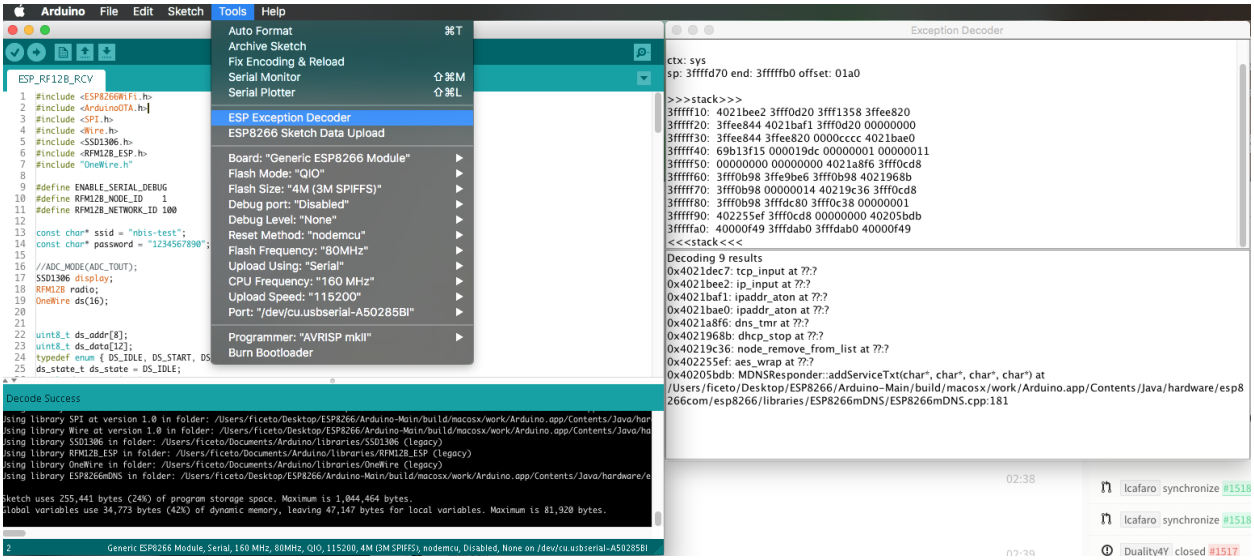


Fig. 1: ESP Exception Decoder

CHAPTER

SIXTEEN

USING ECLIPSE WITH ARDUINO ESP8266

16.1 What to Download

- arduino IDE
- Eclipse IDE for C/C++ Developers
- Java

16.2 Setup Arduino

See the Readme

16.3 Setup Eclipse

- step 1
- step 2
- go to Window –> preferences –> Arduino
- add as private hardware path the Part to the ESP8266

example private hardware path

```
Windows: C:\Users\[username]\AppData\Roaming\Arduino15\packages\esp8266\hardware
Linux: /home/[username]/.arduino15/packages/esp8266/hardware
```

16.4 Eclipse won't build

if eclipse dont find the path to the Compiler add to the platform.txt after:

```
version=1.6.4
```

this:

```
runtime.tools.xtensa-lx106-elf-gcc.path={runtime.platform.path}/../../../tools/xtensa-
↪lx106-elf-gcc/1.20.0-26-gb404fb9
runtime.tools.esptool.path={runtime.platform.path}/../../../tools/esptool/0.4.4
```

Note: - the path may changed, check the current version. - each update over the Arduino IDE will remove the fix - may not needed in future if Eclipse Plugin get an Update