ver. 1.6.5-947-g39819f0, built on Jul 23, 2015

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`. 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.

Pin Functions

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.

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`.

Analog input

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 is 0 — 1.0V.

To read VCC voltage, 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.

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 `PWMRANGE`, which is equal to 1023 by default. PWM range may be changed by calling `analogWriteRange(new_range)`.

PWM frequency is 1kHz by default. Call `analogWriteFreq(new_frequency)` to change the frequency.

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.

Serial

`Serial` object works much the same way as on a regular Arduino. Apart from hardware FIFO (128 bytes for TX and RX) HardwareSerial has additional 256-byte TX and RX buffers. Both transmit and receive is interrupt-driven. Write and read functions only block the sketch execution when the respective FIFO/buffers are full/empty.

`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)`.

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.

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. 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.

WiFi(ESP8266WiFi library)

This is mostly similar to WiFi shield library. Differences include:

- `WiFi.mode(m)`: set mode to `WIFI_AP`, `WIFI_STA`, or `WIFI_AP_STA`.
- call `WiFi.softAP(ssid)` to set up an open network
- call `WiFi.softAP(ssid, password)` to set up a WPA2-PSK network (password should be at least 8 characters)
- `WiFi.macAddress(mac)` is for STA, `WiFi.softAPmacAddress(mac)` is for AP.
- `WiFi.localIP()` is for STA, `WiFi.softAPIP()` is for AP.
- `WiFi.RSSI()` doesn't work
- `WiFi.printDiag(Serial)` will print out some diagnostic info
- `WiFiUDP` class supports sending and receiving multicast packets on STA interface. When sending a multicast packet, replace `udp.beginPacket(addr, port)` with `udp.beginPacketMulticast(addr, port, WiFi.localIP())`. When listening to

multicast packets, replace `udp.begin(port)` with `udp.beginMulticast(WiFi.localIP(), multicast_ip_addr, port)`. You can use `udp.destinationIP()` to tell whether the packet received was sent to the multicast or unicast address. Also note that multicast doesn't work on softAP interface.

`WiFiServer`, `WiFiClient`, and `WiFiUDP` behave mostly the same way as with WiFi shield library. Four samples are provided for this library. You can see more commands here: http://www.arduino.cc/en/Reference/WiFi

Ticker

Library for calling functions repeatedly with a certain period. Two 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.

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 at 0x7b000 for storage.

Three examples included.

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).

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).

ESP-specific APIs

APIs related to deep sleep and watchdog timer are available in the `ESP` object, only available in Alpha version.

`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.)

`ESP.restart()` restarts the CPU.

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

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

Several APIs may be used to get flash chip info:

`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.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.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.

OneWire (from https://www.pjrc.com/teensy/td_libs_OneWire.html)

Library was adapted to work with ESP8266 by including register definitions into OneWire.h Note that if you already have OneWire library in your Arduino/libraries folder, it will be used instead of the one that comes with this package.

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 dicovery) queries. Currently the library only works on STA interface, AP interface is not supported. See attached example for details.

SSDP responder (ESP8266SSDP)

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

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. See attached example for details.

Servo

This library exposes the ability to control RC (hobby) servo motors. It will support upto 24 servos on any available output pin. By defualt 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 effected. 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.

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:

- 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
- DHT11 - Download latest v1.1.0 library and no changes are necessary. Older versions should initialize DHT as follows: `DHT dht(DHTPIN, DHTTYPE, 15)`

- 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 "NeoPixelAnimator" branch for ESP8266 to get 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.

ESP8266 Arduino Core

ESP8266 Arduino Core
ivan@esp8266.com

esp8266

Documentation for ESP8266 Arduino Core. Installation instructions, functions and classes reference.