Reference

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) {...
```

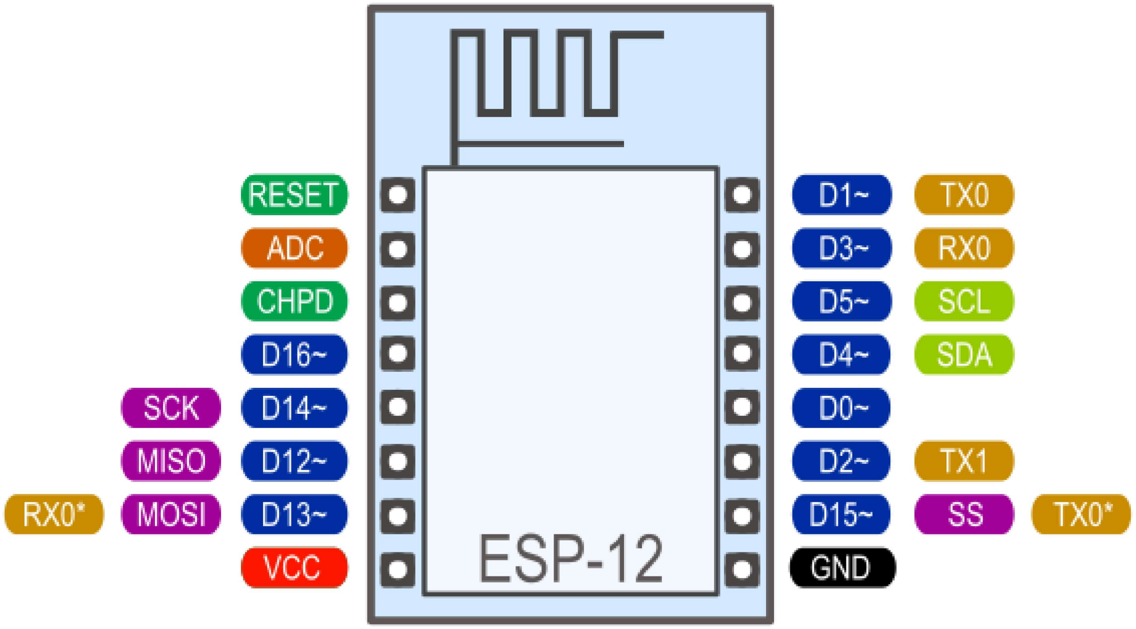
- Interrupts must not call `delay()` or `yield()`, or call any routines which internally use `delay()` or `yield()` either.
- Long-running (>1ms) tasks in interrupts will cause instabilty or crashes. WiFi and other portions of the core can become unstable if interrupts are blocked by a long-running interrupt. 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.
- Memory operations can be dangerous and should be avoided in interrupts. Calls to `new` or `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.).

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.



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`. ISRs need to have `IRAM_ATTR` before the function definition.

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.

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.

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

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.

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);
}
```

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

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..).