

Kolban's Book on ESP8266

NOVEMBER 2016

Neil Kolban

Table of Contents

Introduction.....	24
Overview.....	25
The ESP8266.....	26
Maturity.....	27
The ESP8266 specification.....	27
ESP8266 Modules.....	28
ESP-12.....	28
ESP-1.....	32
Adafruit HUZZAH.....	38
NodeMCU devKit.....	38
node.IT (aka ESP-210).....	40
SparkFun WiFi Shield – ESP8266.....	40
Espresso Lite.....	41
Wemos D1.....	41
Oak by digistump.....	41
Connecting to the ESP8266.....	41
WiFi Theory.....	42
AT Command Programming.....	44
Commands.....	45
Installing the latest AT command processor.....	51
Assembling circuits.....	52
USB to UART converters.....	52
Breadboards.....	54
Power.....	55
Multi-meter / Logic probe / Logic Analyzer.....	56
Sundry components.....	56
Physical construction.....	56
Recommended setup for programming ESP8266.....	56
Configuration for flashing the device.....	59
Programming.....	60
Boot mode.....	60
ESP8266 – Software Development Kit (SDK).....	61
Include directories.....	61
Compiling.....	62
Loading a program into the ESP8266.....	69
Programming environments.....	73
Compilation tools.....	73
ar.....	73
esptool.py.....	74
esptool-ck.....	76

gcc.....	78
gen_appbin.py.....	79
make.....	80
nodemcu-flasher.....	80
nm.....	82
objcopy.....	82
objdump.....	82
xxd.....	82
ESP8266 Linking.....	83
Flashing over the air – FOTA.....	84
Debugging.....	88
ESP-IDF logging.....	88
Logging to UART1.....	90
Run a Blinky.....	90
Dumping IP Addresses.....	91
Exception handling.....	91
Debugging and testing TCP and UDP connections.....	93
Android – Socket Protocol.....	94
Android – UDP Sender/Receiver.....	94
Windows – Hercules.....	94
Curl.....	94
Eclipse – TCP/MON.....	94
httpbin.org.....	97
ESP8266 Architecture.....	97
Custom programs.....	97
WiFi at startup.....	97
Working with WiFi – ESP8266.....	98
Scanning for access points.....	98
Defining the operating mode.....	99
Handling WiFi events.....	99
Station configuration.....	101
Connecting to an access point.....	101
Control and data flows when connecting as a station.....	102
Being an access point.....	103
The DHCP server.....	104
Current IP Address, netmask and gateway.....	105
WiFi Protected Setup – WPS.....	105
Working with TCP/IP.....	106
The espconn architecture.....	106
TCP.....	107
Sending and receiving TCP data.....	111
Flow control.....	113
TCP Error handling.....	113
UDP.....	114

Broadcast with UDP.....	116
Ping request.....	117
Name Service.....	117
Multicast Domain Name Systems.....	118
Installing Bonjour.....	119
Working with SNTP.....	122
ESP-NOW.....	123
GPIOs.....	124
Pullup and pull down settings.....	130
GPIO Interrupt handling.....	130
Expanding the number of available GPIOs.....	132
ESP_PCF8574 C library.....	136
PCF8574 JavaScript Library.....	137
Working with I2C.....	137
Working with SPI – Serial Peripheral Interface.....	139
Hardware SPI.....	141
The MetalPhreak/ESP8266_SPI_Driver.....	143
Working with serial.....	144
ESP8266 Task handling.....	146
Timers and time.....	147
Working with memory.....	148
Working with flash memory.....	152
Pulse Width Modulation – PWM.....	153
Analog to digital conversion.....	154
Sleep modes.....	156
Watchdog timer.....	157
Yielding control.....	158
Security.....	159
Mapping from Arduino.....	159
Spiffs File System.....	160
Partner TCP/IP APIs.....	161
TCP/IP Sockets.....	162
Handling errors.....	165
Sockets – accept().....	168
Sockets – bind().....	169
Sockets – close().....	169
Sockets – closesocket().....	169
Sockets – connect().....	169
Sockets – fcntl().....	170
Sockets – freeaddrinfo().....	170
Sockets – getaddrinfo().....	170
Sockets – gethostname().....	170
Sockets – getpeername().....	170
Sockets – getsockname().....	170

Sockets – getsockopt()	170
Sockets – htonl()	171
Sockets – htons()	171
Sockets – inet_ntop()	171
Sockets – inet_pton()	171
Sockets – ioctlsocket()	171
Sockets – listen()	171
Sockets – read()	171
Sockets – recv()	172
Sockets – recvfrom()	172
Sockets – select()	173
Sockets – send()	173
Sockets – sendto()	173
Sockets – setsockopt()	173
Sockets – shutdown()	174
Sockets – socket()	174
Sockets – write()	174
Socket data structures	175
Sockets – struct sockaddr	175
Sockets – struct sockaddr_in	175
Java Sockets	175
WebSockets	178
A WebSocket browser app	178
FreeRTOS WebSocket	180
Mongoose WebSocket	180
Web Servers	181
Mongoose	181
Programming using Eclipse	182
Installing the Eclipse Serial terminal	186
Web development using Eclipse	192
Programming using the Arduino IDE	193
Implications of Arduino IDE support	194
Installing the Arduino IDE with ESP8266 support	195
Tips for working in the Arduino environment	201
Initialize global classes in setup()	201
Invoking Espressif SDK API from a sketch	201
Exception handling	202
The SPIFFS file system	202
The mkspiffs command	202
The architecture of the Arduino IDE support	203
Building ESP Arduino apps using the Eclipse IDE	211
Reasons to consider using Eclipse over Arduino IDE	225
Notes on using the Eclipse Arduino package	226
Arduino ESP Libraries	227

The WiFi library.....	227
WiFi.begin.....	227
WiFi.beingSmartConfig.....	228
WiFi.beginWPSConfig.....	228
WiFi.BSSID.....	228
WiFi.BSSIDstr.....	228
WiFi channel.....	228
WiFi.config.....	229
WiFi.disconnect.....	229
WiFi.encryptionType.....	229
WiFi.gatewayIP.....	229
WiFi.getNetworkInfo.....	229
WiFi.hostByName.....	230
WiFi.hostname.....	230
WiFi.isHidden.....	230
WiFi.localIP.....	230
WiFi.macAddress.....	230
WiFi.mode.....	231
WiFi.printDiag.....	231
WiFi.RSSI.....	231
WiFi.scanComplete.....	231
WiFi.scanDelete.....	232
WiFi.scanNetworks.....	232
WiFi.smartConfigDone.....	232
WiFi.softAP.....	232
WiFi.softAPConfig.....	233
WiFi.softAPDisconnect.....	233
WiFi.softAPmacAddress.....	233
WiFi.softAPIP.....	233
WiFi.SSID.....	233
WiFi.status.....	233
WiFi.stopSmartConfig.....	234
WiFi.subnetMask.....	234
WiFi.waitForConnectResult.....	234
WiFiClient.....	234
WiFiClient.....	234
WiFiClient.available.....	234
WiFiClient.connect.....	235
WiFiClient.connected.....	235
WiFiClient.flush.....	235
WiFiClient.getNoDelay.....	235
WiFiClient.peek.....	235
WiFiClient.read.....	235
WiFiClient.remoteIP.....	235

WiFiClient.remotePort.....	236
WiFiClient.setLocalPortStart.....	236
WiFiClient.setNoDelay.....	236
WiFiClient.status.....	236
WiFiClient.stop.....	236
WiFiClient.stopAll.....	236
WiFiClient.write.....	236
WiFiServer.....	237
WiFiServer.....	237
WiFiServer.available.....	237
WiFiServer.begin.....	237
WiFiServer.getNoDelay.....	237
WiFiServer.hasClient.....	237
WiFiServer.setNoDelay.....	238
WiFiServer.status.....	238
WiFiServer.write.....	238
IPAddress.....	238
ESP8266WebServer.....	238
ESP8266WebServer.....	241
ESP8266WebServer.arg.....	241
ESP8266WebServer.argName.....	241
ESP8266WebServer.args.....	241
ESP8266WebServer.begin.....	241
ESP8266WebServer.client.....	241
ESP8266WebServer.handleClient.....	242
ESP8266WebServer.hasArg.....	242
ESP8266WebServer.method.....	242
ESP8266WebServer.on.....	242
ESP8266WebServer.onFileUpload.....	243
ESP8266WebServer.onNotFound.....	243
ESP8266WebServer.send.....	243
ESP8266WebServer.sendContent.....	243
ESP8266WebServer.sendHeader.....	243
ESP8266WebServer.setContentLength.....	243
ESP8266WebServer.streamFile.....	244
ESP8266WebServer.upload.....	244
ESP8266WebServer.uri.....	244
ESP8266mDNS library.....	244
MDNS.addService.....	244
MDNS.begin.....	244
MDNS.update.....	244
I2C – Wire.....	245
Wire.available.....	245
Wire.begin.....	245

Wire.beginTransmission.....	246
Wire.endTransmission.....	246
Wire.flush.....	246
Wire.onReceive.....	247
Wire.onReceiveService.....	247
Wire.onRequest.....	247
Wire.onRequestService.....	247
Wire.peek.....	247
Wire.pins.....	247
Wire.read.....	248
Wire.requestFrom.....	248
Wire.setClock.....	248
Wire.write.....	249
Ticker library.....	249
Ticker.....	249
attach.....	249
attach_ms.....	250
detach.....	250
once.....	250
once_ms.....	250
EEPROM library.....	250
EEPROM.begin.....	251
EEPROM.commit.....	251
EEPROM.end.....	251
EEPROM.get.....	251
EEPROM.getDataPtr.....	251
EEPROM.put.....	251
EEPROM.read.....	251
EEPROM.write.....	251
SPIFFS.....	252
SPIFFS.begin.....	252
SPIFFS.open.....	252
SPIFFS.openDir.....	252
SPIFFS.remove.....	252
SPIFFS.rename.....	253
File.available.....	253
File.close.....	253
File.flush.....	253
File.name.....	253
File.peek.....	253
File.position.....	253
File.read.....	253
File.seek.....	254
File.size.....	254

File.write.....	254
Dir.fileName.....	254
Dir.next.....	254
Dir.open.....	254
Dir.openDir.....	254
Dir.remove.....	255
Dir.rename.....	255
ESP library.....	255
ESP.deepSleep.....	255
ESP.eraseConfig.....	255
ESP.getBootMode.....	255
ESP.getBootVersion.....	255
ESP.getChipId.....	255
ESP.getCpuFreqMHz.....	255
ESP.getCycleCount.....	255
ESP.getFlashChipId.....	255
ESP.getFlashChipMode.....	255
ESP.getFlashChipRealSize.....	256
ESP.getFlashChipSize.....	256
ESP.getFlashChipSizeByChipId.....	256
ESP.getFlashChipSpeed.....	256
ESP.getFreeHeap.....	256
ESP.getFreeSketchSpace.....	256
ESP.getResetInfo.....	256
ESP.getResetInfoPtr.....	256
ESP.getSdkVersion.....	256
ESP.getSketchSize.....	256
ESP.getVcc.....	257
ESP.reset.....	257
ESP.restart.....	257
ESP.updateSketch.....	257
ESP.wdtDisable.....	257
ESP.wdtEnable.....	257
ESP.wdtFeed.....	257
String library.....	258
Constructor.....	258
String.c_str.....	258
String.reserve.....	258
String.length.....	258
String.concat.....	258
String.equalsIgnoreCase.....	258
String.startsWith.....	258
String.endsWith.....	259
String.charAt.....	259

String.setCharAt.....	259
String.getBytes.....	259
String.toCharArray.....	259
String.indexOf.....	259
String.lastIndexOf.....	259
String.substring.....	259
String.replace.....	259
String.remove.....	259
String.toLowerCase.....	259
String.toUpperCase.....	259
String.trim.....	260
String.toInt.....	260
String.toFloat.....	260
Programming with JavaScript.....	260
Smart.js.....	261
Smart.js GPIO.....	262
Setting up an HTTP server.....	263
Debugging.....	264
Espruino.....	265
Editing and deploying code.....	265
Working with variables.....	266
Booting Espruino.....	266
WiFi access.....	266
Writing network socket applications using Espruino.....	267
Writing a REST client using Espruino.....	268
Writing a Web Server using Espruino.....	269
Working with GPIO.....	271
Working with I2C and JavaScript.....	271
Debugging JavaScript.....	272
Editing JavaScript.....	272
Espruino ESP8266 Libraries.....	273
Core JavaScript capabilities.....	274
Running code at intervals.....	274
Working with GPIO.....	275
SPI.....	275
Key differences from JavaScript.....	276
Building Espruino.....	276
Programming with Lua.....	277
ESPlorer IDE.....	277
GPIO with Lua.....	277
WiFi with Lua.....	278
Networking with Lua.....	278
Programming with Basic.....	278
Integration with Web Apps.....	278

REST Services.....	278
REST protocol.....	279
ESP8266 as a REST client.....	279
Making a REST request using Mongoose.....	279
ESP8266 as a REST service provider.....	280
Tasker.....	280
AutoRemote.....	280
DuckDNS.....	282
Mobile apps.....	283
Blynk.....	283
Sample Snippets.....	283
Forming a TCP connection.....	283
Sample applications.....	284
Sample – Light an LED based on the arrival of a UDP datagram.....	284
Sample – Ultrasonic distance measurement.....	286
Sample – WiFi Scanner.....	289
Sample – Working with micro SD cards.....	289
Sample – Playing audio from an event.....	289
Sample – A changeable mood light.....	289
Sample – Bootstrapping networking.....	294
Sample Libraries.....	294
Function list.....	294
authModeToString.....	294
checkError.....	295
delayMilliseconds.....	295
dumpBSSINFO.....	295
dumpEspConn.....	295
dumpRestart.....	295
dumpState.....	295
errorToString.....	296
eventLogger.....	296
eventReasonToString.....	296
flashSizeAndMapToString.....	296
setAsGpio.....	296
setupBlink.....	296
toHex.....	297
Using FreeRTOS.....	297
The architecture of a task in FreeRTOS.....	298
Blocking and synchronization within RTOS.....	300
Lists within RTOS.....	300
ESP8266 – Building apps for RTOS.....	300
Consoles with RTOS.....	302
Debugging tips.....	303
Developing solutions on Linux.....	304

Building a Linux environment.....	304
API Reference.....	313
FreeRTOS API reference.....	313
eTaskGetState.....	313
pcTaskGetName.....	313
xEventGroupClear.....	314
xEventGroupCreate.....	314
xEventGroupSetBits.....	314
xEventGroupWaitBits.....	314
xTaskCreate.....	315
vTaskDelay.....	316
vTaskDelayUntil.....	316
vTaskDelete.....	316
xTaskGetCurrentTaskHandle.....	317
xTaskGetTickCount.....	317
vEventGroupDelete.....	317
vTaskList.....	317
vTaskPrioritySet.....	317
vTaskResume.....	317
xTaskResumeAll.....	317
vTaskResumeFromISR.....	317
vTaskSuspend.....	318
vTaskSuspendAll.....	318
xQueueCreate.....	318
vQueueDelete.....	318
xQueuePeek.....	318
xQueueReceive.....	318
xQueueSend.....	318
xQueueSendToBack.....	319
xQueueSendToFront.....	319
vSemaphoreCreateBinary.....	319
xSemaphoreCreateCounting.....	319
vSemaphoreGive.....	319
xSemaphoreGiveFromISR.....	319
vSemaphoreTake.....	319
pvPortMalloc.....	319
pvPortFree.....	319
List Processing.....	319
vListInitialise.....	319
vListInitialiseItem.....	319
vListInsert.....	320
vListInsertEnd.....	320
lwip Reference.....	320
Sockets.....	320

Timer functions.....	321
os_delay_us.....	321
os_timer_arm.....	321
os_timer_disarm.....	322
os_timer_setfn.....	322
system_timer_reinit.....	323
os_timer_arm_us.....	323
hw_timer_init.....	323
hw_timer_arm.....	323
hw_timer_set_func.....	323
System Functions.....	323
system_adc_read.....	323
system_deep_sleep_set_option.....	323
system_get_boot_mode.....	323
system_get_boot_version.....	324
system_get_chip_id.....	324
system_get_cpu_freq.....	324
system_get_flash_size_map.....	324
system_get_rst_info.....	325
system_get_userbin_addr.....	325
system_get_vdd33.....	325
system_init_done_cb.....	325
system_os_post.....	326
system_os_task.....	326
system_phys_set_rfoption.....	327
system_phys_set_max_tpw.....	327
system_phys_set_tpw_via_vdd33.....	327
system_print_meminfo.....	327
system_restart_enhance.....	328
system_rtc_clock_cali_proc.....	328
system_set_os_print.....	328
system_show_malloc.....	328
system_rtc_clock_cali_proc.....	329
system_uart_swap.....	329
system_soft_wdt_feed.....	329
system_soft_wdt_stop.....	329
system_soft_wdt_restart.....	330
system_uart_de_swap.....	330
system_update_cpu_freq.....	330
os_memset.....	330
os_memcmp.....	330
os_memcpy.....	331
os_malloc.....	331
os_calloc.....	331

os_realloc.....	331
os_zalloc.....	332
os_free.....	332
os_bzero.....	332
os_delay_us.....	333
os_printf.....	333
os_install_putc1.....	333
os_random.....	334
os_get_random.....	334
os_strlen.....	334
os_strcat.....	334
os strchr.....	335
os_strcmp.....	335
os strcpy.....	335
os strncmp.....	335
os strncpy.....	335
os sprintf.....	336
os strstr.....	336
SPI Flash.....	336
spi_flash_get_id.....	336
spi_flash_erase_sector.....	336
spi_flash_read.....	337
spi_flash_set_read_func.....	337
system_param_save_with_protect.....	337
spi_flash_write.....	337
system_param_load.....	338
WiFi – ESP8266.....	338
wifi_fpm_close.....	338
wifi_fpm_do_sleep.....	338
wifi_fpm_do_wakeup.....	338
wifi_fpm_get_sleep_type.....	338
wifi_fpm_open.....	338
wifi_fpm_set_sleep_type.....	338
wifi_fpm_set_wakeup_cb.....	338
wifi_get_channel.....	338
wifi_get_ip_info.....	338
wifi_get_macaddr.....	339
wifi_get_opmode.....	339
wifi_get_opmode_default.....	339
wifi_get_phy_mode.....	340
wifi_get_sleep_type.....	340
wifi_get_user_fixed_rate.....	340
wifi_get_user_limit_rate_mask.....	340
wifi_set_broadcast_if.....	340

wifi_get_broadcast_if.....	341
wifi_set_sleep_type.....	341
wifi_promiscuous_enable.....	341
wifi_promiscuous_set_mac.....	341
wifi_register_rfid_locp_recv_cb.....	341
wifi_register_send_pkt_freedom_cb.....	341
wifi_register_user_ie_manufacturer_recv_cb.....	341
wifi_rfid_locp_recv_close.....	341
wifi_rfid_locp_recv_open.....	341
wifi_send_pkt_freedom.....	341
wifi_set_channel.....	341
wifi_set_event_handle_cb.....	341
wifi_set_ip_info.....	342
wifi_set_macaddr.....	342
wifi_set_opmode.....	342
wifi_set_opmode_current.....	343
wifi_set_phy_mode.....	343
wifi_set_promiscuous_rx_cb.....	343
wifi_set_sleep_type.....	344
wifi_set_user_fixed_rate.....	344
wifi_set_user_ie.....	344
wifi_set_user_limit_rate_mask.....	344
wifi_set_user_rate_limit.....	344
wifi_set_user_sup_rate.....	344
wifi_status_led_install.....	345
wifi_status_led_uninstall.....	345
wifi_unregister_rfid_locp_recv_cb.....	346
wifi_unregister_send_pkt_freedom_cb.....	346
wifi_unregister_user_ie_manufacturer_recv_cb.....	346
WiFi Station.....	346
wifi_station_ap_change.....	346
wifi_station_ap_number_set.....	346
wifi_station_connect.....	346
wifi_station_dhcpc_start.....	347
wifi_station_dhcpc_status.....	347
wifi_station_dhcpc_stop.....	347
wifi_station_disconnect.....	347
wifi_station_get_ap_info.....	348
wifi_station_get_auto_connect.....	348
wifi_station_get_config.....	348
wifi_station_get_config_default.....	349
wifi_station_get_connect_status.....	349
wifi_station_get_current_ap_id.....	349
wifi_station_get_hostname.....	350

wifi_station_get_reconnect_policy.....	350
wifi_station_get_rssi.....	350
wifi_station_scan.....	350
wifi_station_set_auto_connect.....	351
wifi_station_set_cert_key.....	352
wifi_station_clear_cert_key.....	352
wifi_station_set_config.....	352
wifi_station_set_config_current.....	353
wifi_station_set_reconnect_policy.....	353
wifi_station_set_hostname.....	353
WiFi SoftAP.....	353
wifi_softap_dhcps_start.....	353
wifi_softap_dhcps_status.....	354
wifi_softap_dhcps_stop.....	354
wifi_softap_free_station_info.....	354
wifi_softap_get_config.....	355
wifi_softap_get_config_default.....	355
wifi_softap_get_dhcps_lease.....	356
wifi_softap_get_dhcps_lease_time.....	356
wifi_softap_get_station_info.....	356
wifi_softap_get_station_num.....	356
wifi_softap_reset_dhcps_lease_time.....	356
wifi_softap_set_config.....	357
wifi_softap_set_config_current.....	357
wifi_softap_set_dhcps_lease.....	357
wifi_softap_set_dhcps_lease_time.....	358
wifi_softap_dhcps_offer_option.....	358
WiFi WPS.....	358
wifi_wps_enable.....	358
wifi_wps_disable.....	359
wifi_wps_start.....	359
wifi_set_wps_cb.....	359
Upgrade APIs.....	359
system_upgrade_flag_check.....	359
system_upgrade_flag_set.....	360
system_upgrade_reboot.....	360
system_upgrade_start.....	360
system_upgrade_userbin_check.....	360
wifi_promiscuous_enable.....	360
wifi_promiscuous_set_mac.....	360
wifi_promiscuous_rx_cb.....	360
wifi_get_channel.....	361
wifi_set_channel.....	361
Smart config APIs.....	361

smartconfig_start.....	361
smartconfig_stop.....	361
SNTP API.....	361
snntp_setserver.....	361
snntp_getserver.....	361
snntp_setservername.....	362
snntp_getservername.....	362
snntp_init.....	362
snntp_stop.....	363
snntp_get_current_timestamp.....	363
snntp_get_real_time.....	363
snntp_set_timezone.....	363
snntp_get_timezone.....	364
Generic TCP/UDP APIs.....	364
espconn_delete.....	364
espconn_dns_setserver.....	364
espconn_gethostname.....	365
espconn_port.....	366
espconn_regist_sentcb.....	366
espconn_regist_recvcb.....	366
espconn_send.....	367
espconn_sendto.....	367
ipaddr_addr.....	367
IP4_ADDR.....	368
IP2STR.....	368
TCP APIs.....	368
espconn_abort.....	368
espconn_accept.....	369
espconn_get_connection_info.....	369
espconn_connect.....	370
espconn_disconnect.....	370
espconn_regist_connectcb.....	371
espconn_regist_disconcb.....	371
espconn_regist_reconcb.....	371
espconn_regist_write_finish.....	372
espconn_set_opt.....	373
espconn_clear_opt.....	373
espconn_regist_time.....	374
espconn_set_keepalive.....	374
espconn_get_keepalive.....	375
espconn_secure_accept.....	375
espconn_secure_ca_disable.....	375
espconn_secure_ca_enable.....	375
espconn_secure_set_size.....	375

espconn_secure_get_size.....	375
espconn_secure_delete.....	375
espconn_secure_connect.....	375
espconn_secure_send.....	376
espconn_secure_disconnect.....	376
espconn_tcp_get_max_con.....	376
espconn_tcp_set_max_con.....	376
espconn_tcp_get_max_con_allow.....	376
espconn_tcp_set_max_con_allow.....	376
espconn_recv_hold.....	376
espconn_recv_unhold.....	377
UDP APIs.....	377
espconn_create.....	377
espconn_igmp_join.....	377
espconn_igmp_leave.....	377
ping APIs.....	378
ping_start.....	378
ping_regist_recv.....	378
ping_regist_sent.....	378
mDNS APIs.....	379
espconn_mdns_init.....	379
espconn_mdns_close.....	379
espconn_mdns_server_register.....	379
espconn_mdns_server_unregister.....	379
espconn_mdns_getservername.....	379
espconn_mdns_setservername.....	379
espconn_mdns_set_hostname.....	380
espconn_mdns_get_hostname.....	380
espconn_mdns_disable.....	380
espconn_mdns_enable.....	380
GPIO – ESP32.....	380
gpio_config.....	380
gpio_get_level.....	382
gpio_input_get.....	382
gpio_input_get_high.....	382
gpio_intr_enable.....	382
gpio_intr_disable.....	382
gpio_isr_register.....	382
gpio_output_set.....	383
gpio_output_set_high.....	383
gpio_set_direction.....	383
gpio_set_intr_type.....	384
gpio_set_level.....	384
gpio_set_pull_mode.....	384

GPIO – ESP8266.....	385
PIN_PULLUP_DIS.....	386
PIN_PULLUP_EN.....	387
PIN_FUNC_SELECT.....	387
GPIO_ID_PIN.....	387
GPIO_OUTPUT_SET.....	387
GPIO_DIS_OUTPUT.....	388
GPIO_INPUT_GET.....	388
gpio_output_set.....	388
gpio_input_get.....	389
gpio_intr_handler_register.....	389
gpio_pin_intr_state_set.....	389
gpio_intr_pending.....	390
gpio_intr_ack.....	390
gpio_pin_wakeup_enable.....	390
gpio_pin_wakeup_disable.....	391
UART APIs.....	391
UART_CheckOutputFinished.....	391
UART_ClearIntrStatus.....	391
UART_ResetFifo.....	391
UART_SetBaudrate.....	391
UART_SetFlowCtrl.....	391
UART_SetIntrEna.....	391
UART_SetLineInverse.....	391
UART_SetParity.....	392
UART_SetPrintPort.....	392
UART_SetStopBits.....	392
UART_SetWordLength.....	392
UART_WaitTxFifoEmpty.....	392
uart_init.....	392
uart0_tx_buffer.....	393
uart0_sendStr.....	393
uart0_rx_intr_handler.....	393
I2C Master APIs.....	394
i2c_master_checkAck.....	394
i2c_master_getAck.....	394
i2c_master_gpio_init.....	394
i2c_master_init.....	394
i2c_master_readByte.....	395
i2c_master_send_ack.....	395
i2c_master_send_nack.....	395
i2c_master_setAck.....	395
i2c_master_start.....	395
i2c_master_stop.....	395

i2c_master_writeByte.....	395
SPI APIs.....	395
cache_flush.....	395
spi_lcd_9bit_write.....	396
spi_mast_byte_write.....	396
spi_byte_write_espslave.....	396
spi_slave_init.....	396
spi_slave_isr_handler.....	396
hspi_master_readwrite_repeat.....	396
spi_test_init.....	396
PWM APIs.....	396
pwm_init.....	396
pwm_start.....	397
pwm_set_duty.....	397
pwm_get_duty.....	397
pwm_set_period.....	398
pwm_get_period.....	398
get_pwm_version.....	398
set_pwm_debug_en(uint8 print_en).....	398
Bit twiddling.....	398
ESP Now.....	399
esp_now_add_peer.....	399
esp_now_deinit.....	399
esp_now_del_peer.....	399
esp_now_get_peer_key.....	399
esp_now_get_peer_role.....	399
esp_now_get_self_role.....	399
esp_now_init.....	399
esp_now_register_recv_cb.....	399
esp_now_register_send_cb.....	399
esp_now_send.....	399
esp_now_set_kok.....	399
esp_now_set_peer_role.....	399
esp_now_set_peer_key.....	399
esp_now_set_self_role.....	399
esp_now_unregister_recv_cb.....	399
esp_now_unregister_send_cb.....	399
SPIFFS.....	399
esp_spiffs_deinit.....	400
esp_spiffs_init.....	400
SPIFFS_check.....	401
SPIFFS_clearerr.....	401
SPIFFS_close.....	401
SPIFFS_closedir.....	402

SPIFFS_creat.....	402
SPIFFS_erase_deleted_block.....	402
SPIFFS_errno.....	402
SPIFFS_fflush.....	402
SPIFFS_format.....	402
SPIFFS_fremove.....	403
SPIFFS_fstat.....	403
SPIFFS_gc.....	403
SPIFFS_gc_quick.....	403
SPIFFS_info.....	403
SPIFFS_lseek.....	404
SPIFFS_mount.....	404
SPIFFS_mounted.....	405
SPIFFS_open.....	405
SPIFFS_open_by_dirent.....	406
SPIFFS_opendir.....	406
SPIFFS_read.....	407
SPIFFS_readdir.....	407
SPIFFS_remove.....	407
SPIFFS_rename.....	408
SPIFFS_stat.....	408
SPIFFS_unmount.....	408
SPIFFS_write.....	408
Lib-C.....	408
atoi.....	409
atol.....	409
bzero.....	409
calloc.....	409
free.....	409
malloc.....	409
memcmp.....	409
memcpy.....	409
memmove.....	409
memset.....	409
os_get_random.....	409
os_random.....	410
printf.....	410
puts.....	410
rand.....	410
realloc.....	410
snprintf.....	410
sprintf.....	410
strcat.....	410
strchr.....	410

strcmp.....	411
strcpy.....	411
strcspn.....	411
strdup.....	411
strlen.....	411
strncat.....	411
strncmp.....	411
strncpy.....	411
strrchr.....	411
strspn.....	411
strstr.....	412
strtok.....	412
strtok_r.....	412
strtol.....	412
zalloc.....	412
Data structures.....	412
esp_spiffs_config.....	412
station_config.....	412
struct softap_config.....	413
struct station_info.....	413
struct dhcps_lease.....	414
struct bss_info.....	414
struct ip_info.....	415
struct rst_info.....	415
struct espconn.....	416
esp_tcp.....	417
esp_udp.....	417
struct ip_addr.....	417
ipaddr_t.....	418
struct ping_option.....	418
struct ping_resp.....	418
struct mdns_info.....	419
enum phy_mode.....	419
GPIO_INT_TYPE.....	419
System_Event_t.....	420
espconn error codes.....	422
STATUS.....	422
Reference materials.....	424
C++ Programming.....	424
Simple class definition.....	424
Lambda functions.....	424
Ignoring warnings.....	424
Eclipse.....	425
ESPFS breakdown.....	425

EspFsInit.....	425
espFsOpen.....	425
espFsClose.....	425
espFsFlags.....	425
espFsRead.....	426
mkespfimage.....	426
ESPHTTPD breakdown.....	426
httpdInit.....	426
httpdGetMimetype.....	427
httpdUrlDecode.....	427
httpdStartResponse.....	427
httpdSend.....	427
httpdRedirect.....	428
httpdHeader.....	428
httpdGetHeader.....	428
httpdFindArg.....	428
httpdEndHeaders.....	428
Makefiles.....	429
Forums.....	431
Reference documents.....	431
Github.....	432
Github quick cheats.....	432
SDK.....	433
Single board computer comparisons.....	433
Heroes.....	434
Max Filippov – jcmvbkb – GCC compiler for Xtensa.....	434
Ivan Grokhotkov – igrr – Arduino IDE for ESP8266 development.....	434
jantje – Arduino Eclipse.....	435
Richard Sloan – ESP8266 Community owner.....	435
Mikhail Grigorev – CHERTS – Eclipse for ESP8266 development.....	435
Mmiscool – Basic Interpreter.....	436
Areas to Research.....	436

Introduction

Howdy Folks,

I've been working in the software business for over 30 years but until recently, hadn't been playing directly with Micro Processors. When I bought a Raspberry PI and then an Arduino, I'm afraid I got hooked. In my house I am surrounded by computers of all shapes, sizes and capacities ... any one of them with orders of magnitude more power than any of these small devices ... however, I still found myself fascinated.

When I stumbled across the ESP8266 in early 2015, it piqued my interest. I hadn't touched C programming in decades (I'm a Java man these days). As I started to read what was available in the way of documentation from the excellent community surrounding the device, I found that there were only small pockets of knowledge. The best source of information was (and still is) the official PDFs for the SDK from Espressif (the makers of the ESP8266) but even that is quite "light" on examples and background. As I studied the device, I started to make notes and my pages of notes continued to grow and grow.

This book (if we want to call it that) is my collated and polished version of those notes. Rather than keep them to myself, I offer them to all of us in the ESP8266 community in the hope that they will be of some value. My plan is to continue to update this work as we all learn more and share what we find in the community forums. As such, I will re-release the work at regular intervals so please check back at the book's home page for the latest.

As you read, make sure that you fully understand that there are undoubtedly inaccuracies, errors in my understanding and errors in my writing. Only by feedback and time will we be able to correct those. Please forgive the grammatical errors and spelling mistakes that my spell checker hasn't caught.

For questions or comments on the book, please post to this forum thread:

<http://www.esp8266.com/viewtopic.php?f=5&t=4326>

The home page for the book is:

<http://neilkolban.com/tech/esp8266/>

Please don't email me directly with technical questions. Instead, let us use the forum and ask and answer the questions as a great community of ESP8266 minded enthusiasts, hobbyists and professionals.



Neil Kolban
Texas, USA

Overview

A micro controller is an integrated circuit that is capable of running programs. There are many instances of those on the market today from a variety of manufacturers. The prices of these micro controllers keeps falling. In the hobbyist market, an open source architecture called "Arduino" that uses the Atmel range of processors has caught the imagination of countless folks. The boards containing these Atmel chips combined with a convention for connections and also a free set of development tools has lowered the entry point for playing with electronics to virtually nill. Unlike a PC, these processors are extremely low end with low amounts of ram and storage capabilities. They won't be replacing the desktop or laptop any time soon. For those who want more "oomph" in their processors, the folks over at Raspberry PI have developed a very cheap (~\$45) board that is based on the ARM processors that has much more memory and uses micro SD for persistent data storage. These devices run a variant of the Linux operating system. I'm not going to talk further about the Raspberry PI as it is in the class of "computer" as opposed to microprocessor.

These micro controller and architectures are great and there will always be a place for them. However, there is a catch ... and that is networking. These devices have an amazing set of capabilities including direct electrical inputs and outputs (GPIOs) and support for a variety of protocols including SPI, I2C, UART and more, however, none of them so far come with wireless networking included.

No question (in my mind) that the Arduino has captured everyone's attention. The Arduino is based on the Atmel chips and has a variety of physical sizes in its open hardware footprints. The primary micro controller used is the ATmega328. One can find instances of these raw processors on ebay for under \$2 with fully constructed boards containing them for under \$3. This is 10-20 times cheaper than the Raspberry PI. Of course, one gets dramatically less than the Raspberry PI so comparison can become odd ... however if what one wants to do is tinker with electronics or make some simple devices that connect to LEDs, switches or sensors, then the functional features needed become closer.

Between them, the Arduino and the Raspberry PI appear to have all the needs covered. If that were the case, this would be a very short book. Let us add the twist that we started with ... wireless networking. To have a device move a robot chassis or flash LED patterns or make some noises or read data from a sensor and beep when the temperature gets too high ... these are all great and worthy projects. However, we are all very much aware of the value of the Internet. Our computers are Internet connected, our phones are connected, we watch TV (Netflix) over the Internet, we play games over the Internet, we socialize (??) over the Internet ... and so on. The Internet has become

such a basic commodity that we would laugh if someone offered us a new computer or a phone that lacked the ability to go "on-line".

Now imagine what a micro controller with native wireless Internet could do for us? This would be a processor which could run applications as well as or better than an Arduino, which would have GPIO and hardware protocol support, would have RAM and flash memory ... but would have the killer new feature that it would also be able to form Internet connections. And that ... simply put ... is what the ESP8266 device is. It is an alternative microprocessor to the ones already mentioned but also has WiFi and TCP/IP (Transmission Control Protocol / Internet Protocol) support already built in. What is more, it is also not much more expensive than an Arduino. Searching ebay, we find ESP8266 boards under \$3.

The ESP8266

The ESP8266 is the name of a micro controller designed by Espressif Systems. Espressif is a Chinese company based out of Shanghai. The ESP8266 advertises itself as a self-contained WiFi networking solution offering itself as a bridge from existing micro controller to WiFi ... and ... is also capable of running self contained applications.

Volume production of the ESP8266 didn't start until the beginning of 2014 which means that, in the scheme of things, this is a brand new entry in the line-up of processors. And ... in our technology hungry world, new commonly equates to interesting. A couple of years after IC production, 3rd party OEMs are taking these chips and building "breakout boards" for them. If I were to hand you a raw ESP8266 straight from the factory, it is unlikely we would know what to do with one. They are very tiny and virtually impossible for hobbyists to attach wires to allow them to be plugged into breadboards. Thankfully, these OEMs bulk purchase the ICs, design basic circuits, design printed circuit boards and construct pre-made boards with the ICs pre-attached immediately ready for our use. It is these boards that capture our interest and that we can buy for a few dollars on ebay.

There are a variety of board styles available. The two that I am going to focus on have been given the names ESP-1 and ESP-12. It is important to note that there is only one ESP8266 processor and it is this processor that is found on ALL breakout boards. What distinguishes one board from another is the number of GPIO pins exposed, the amount of flash memory provided, the style of connector pins and various other considerations related to construction. From a programming perspective, they are all the same.

Maturity

The ESP8266 is a new device in the arena. It has been around since only the summer of 2014 but has already been shipping production volumes in the tens of millions.

Everybody and everything has to start somewhere. This means there is a whole new wealth of territory to be explored and new features and functions and usage patterns to be discovered. On the down side, it does not yet have the richness of tutorials, samples and videos that accompany other micro controller systems. Its documentation is not brilliant and some of the core questions on its usage are still being examined. How this sits with you is a function of your intent in tinkering in this area. If you want to follow the paths that have been followed many times before, other processors will be more attractive. However if you like a sense of adventure and getting in on the "ground floor" of a new arrival, the challenges that we (the ESP8266 community) are trying to solve may actively excite you rather than dissuade you.

It is also a major reason that folks like myself spend many, many hours studying and documenting what we find ... so others can hopefully build on what has been learned without re-inventing the wheel.

Could the excitement about ESP8266 processors fizzle? Yes ... these devices may just be a flash in the pan and a few years from now, the hobbyist won't give a second thought about them. But what I ask you is to approach the device with an open mind.

The ESP8266 specification

When we approach a new electronics device, we like to know about its specification.

Here are some of the salient points:

Voltage	3.3V
Current consumption	10uA – 170mA
Flash memory attachable	16MB max (512K normal)
Processor	Tensilica L106 32 bit
Processor speed	80-160MHz
RAM	32K + 80K
GPIOs	17 (multiplexed with other functions)
Analog to Digital	1 input with 1024 step (10 bit) resolution
802.11 support	b/g/n/d/e/i/k/r
Maximum concurrent TCP connections	5

The question of determining how long an ESP8266 can run on batteries is an interesting one. The current consumption is far from constant. When transmitting at full power, it

can consume 170mA but when in a deep sleep, it only need 10uA. That is quite a difference. This means that the runtime of an ESP8266 on a fixed current reservoir is not just a function of time but also of what it is doing during that time ... and that is a function of the program deployed upon it.

The ESP8266 is designed to be used with a partner memory module and this is most commonly flash memory. Most of the modules come with some flash associated with them. Realize that flash has a finite number of erases per page before something fails. They are rated at about 10,000 erases. This is not normally an issue for configuration change writes or daily log writes ... but if your application is continually writing new data extremely fast, then this may be an issue and your flash memory will fail.

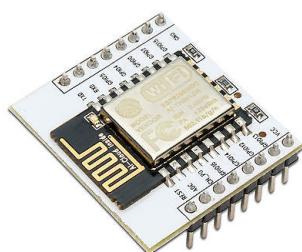
ESP8266 Modules

The ESP8266 integrated circuit comes in a small package, maybe five millimeters square. Obviously, unless you are a master solderer you aren't going to do much with that. The good news is that a number of vendors have created breakout boards that make the job much easier for you. Here we list some of the more common modules.

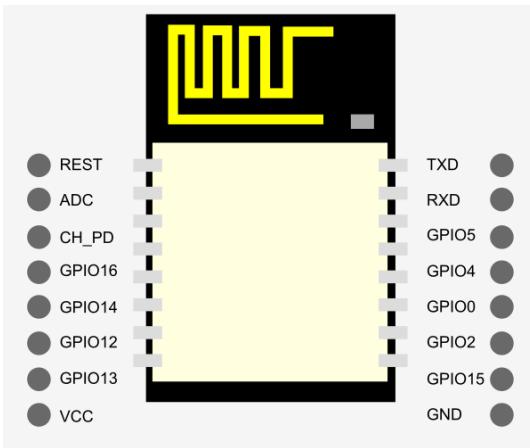
ESP-12

The current most popular and flexible configuration available today is called the ESP-12. It exposes the most GPIO pins for use. The basic ESP-12 module really needs its own expander module to make it breadboard and 0.1" strip board friendly.

Here is what an ESP-12 device looks like when mounted on a breadboard extender board:



The pin out of the extender board looks as follows:

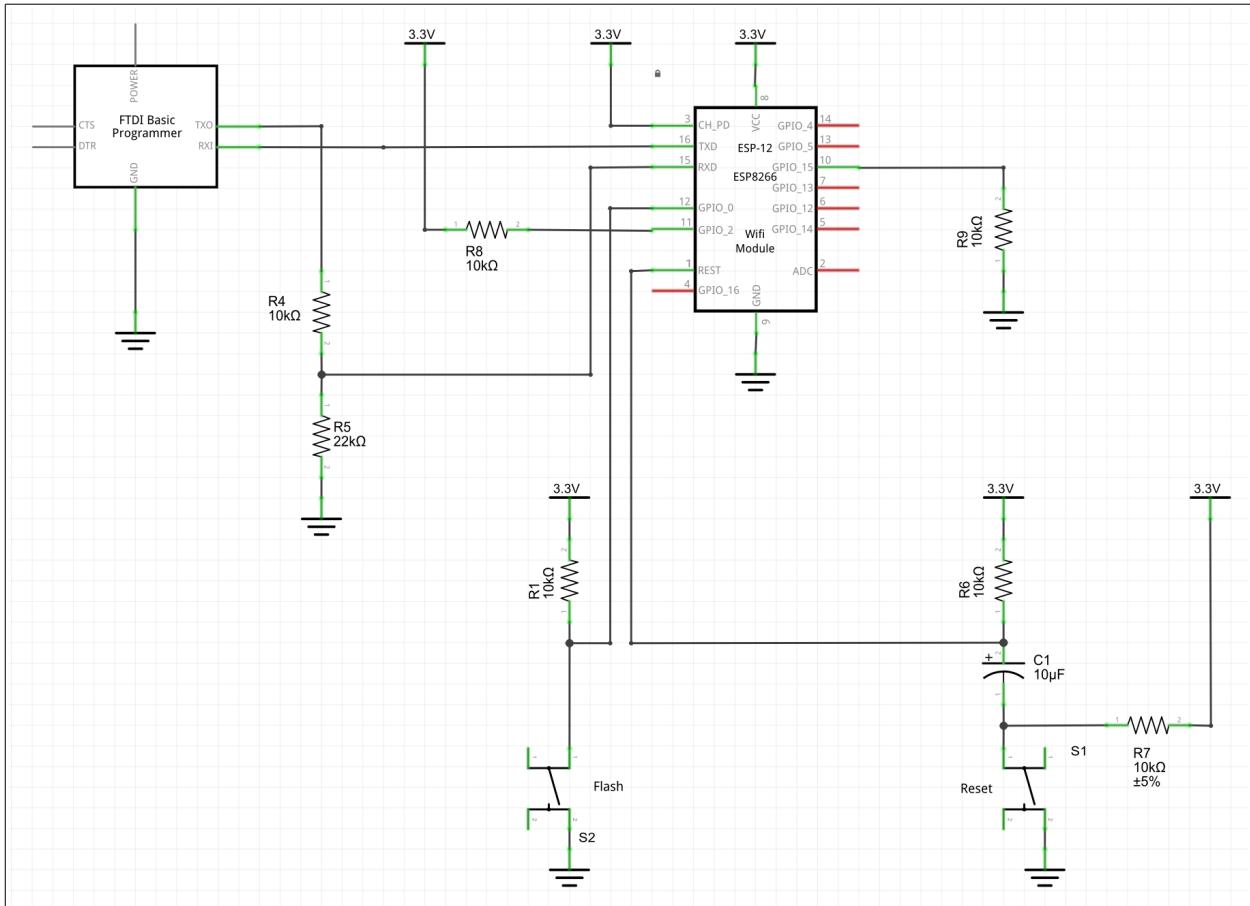


The ESP-12 has a blue surface mounted LED on the upper right. This LED flashes when there is UART traffic.

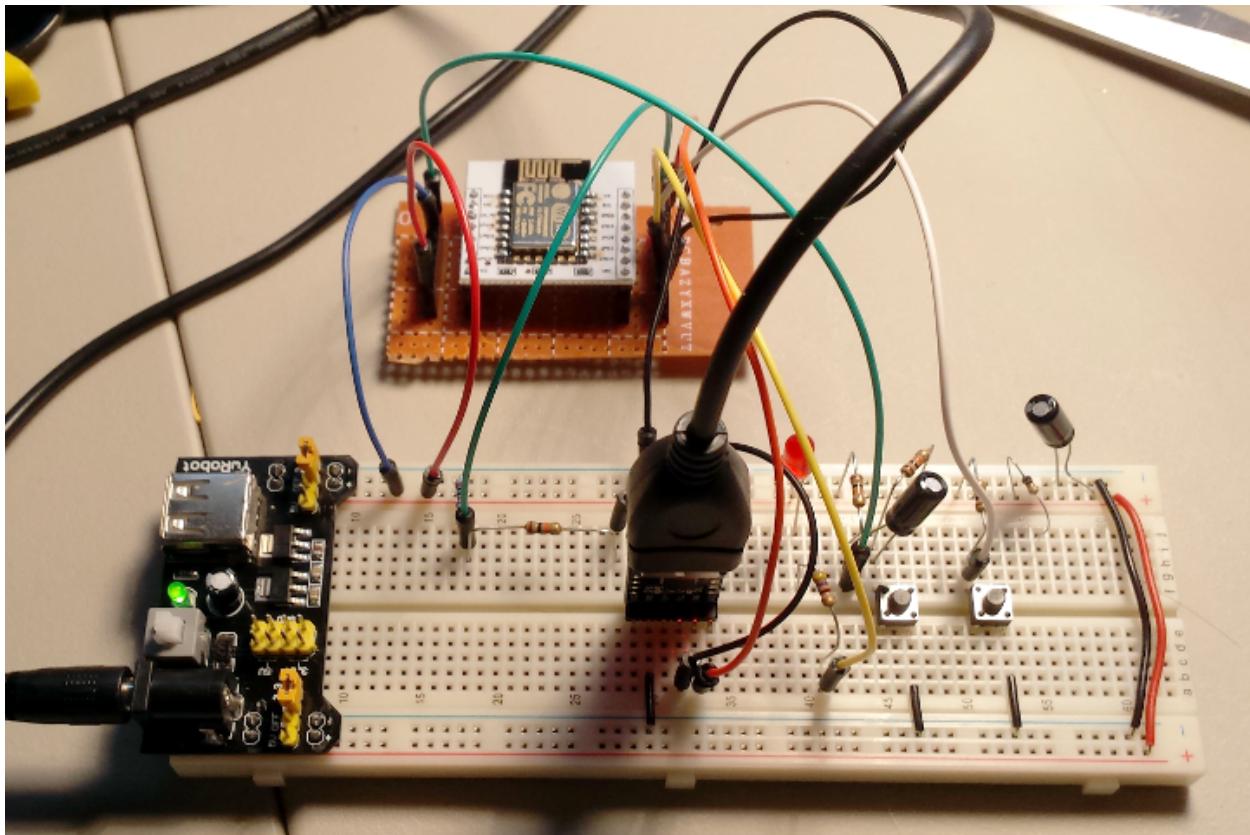
Here is a description of the various pins:

Name	Description
VCC	3.3V.
GPIO 13	Also used for SPI MOSI.
GPIO 12	Also used for SPI MISO.
GPIO 14	Also used for SPI Clock.
GPIO 16	
CH_PD	Chip enable. Should be high for normal operation. <ul style="list-style-type: none"> • 0 – Disabled • 1 – Enabled
ADC	Analog to digital input
REST	External reset. <ul style="list-style-type: none"> • 0 – Reset • 1 – Normal
TXD	UART 0 transmit.
RXD	UART 0 Receive.
GPIO 4	Regular GPIO.
GPIO 5	Regular GPIO.
GPIO 0	Should be high on boot, low for flash update.
GPIO 2	Should be high on boot.
GPIO 15	Should be low on boot and flash.
GND	Ground.

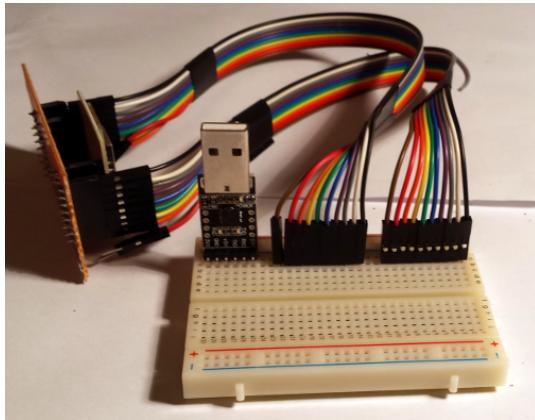
Here is a schematic for connecting an instance:



Next we see an image of this circuit built out on a breadboard.



If we just wish to use our breakout board, we have the following when mounted on a breadboard, we can have the following setup:



This gives us two sets of 8 pin connectors. The first set is:

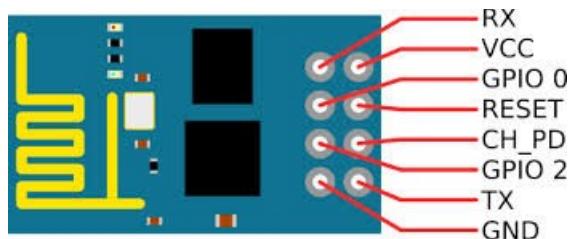
Set 1	
Pin	Color
GND	Orange
GPIO15	Yellow
GPIO2	Green
GPIO0	Blue
GPIO5	Purple
GPIO4	Grey
RXD	White
TXD	Black

The second set is:

Set 2	
Pin	Color
VCC	Orange
GPIO13	Yellow
GPIO12	Green
GPIO14	Blue
GPIO16	Purple
CH_PD	Grey
ADC	White
REST	Black

ESP-1

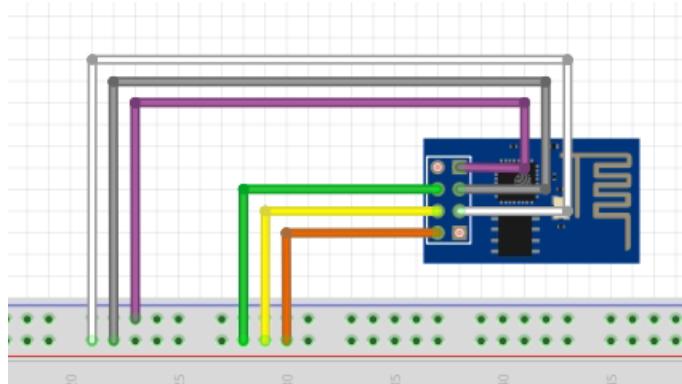
The ESP-1 board is an ESP8266 on an 8 pin board. It is not at all breadboard friendly but fortunately we can make adapters for it extremely easily. The ESP-1 was one of the first boards available and other than price, should all but be discounted. It only provides a small subset of the pin-outs provided by other boards. However, the ESP-1 was massively volume produced and still maintains one of the smallest physical footprints. If you only need a couple of I/Os or need a very tiny size, there may still be some value in this selection.



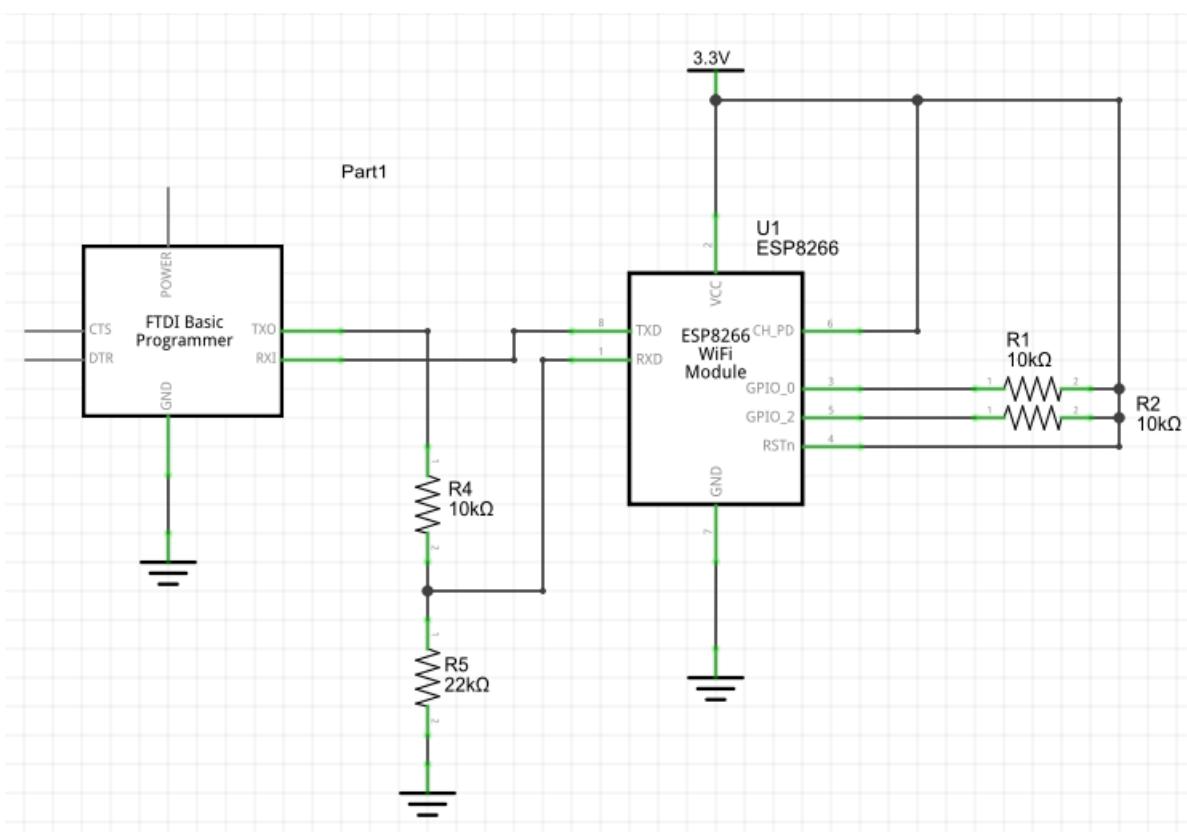
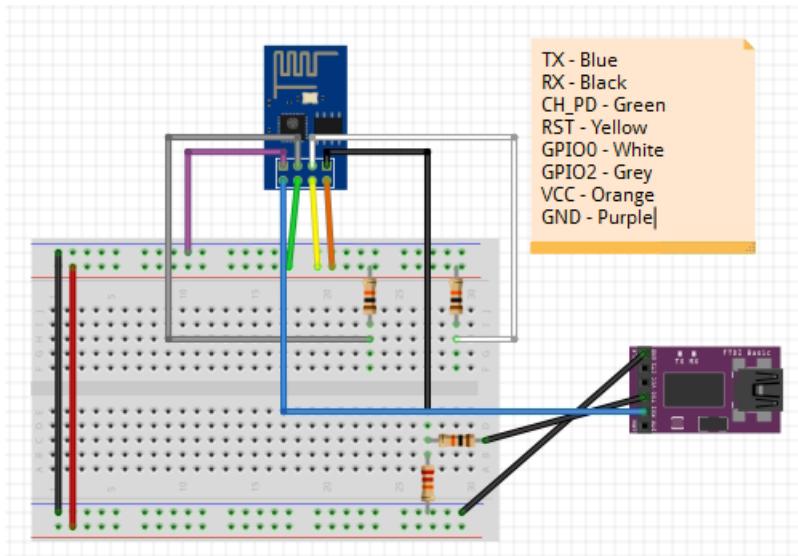
The pin out of the device is as follows:

Function	Color	Description
TX	Blue	Transmit
RX	Black	Receive. Always use a level converter for incoming data. This device is not 5V tolerant.
CH_PD	Green	Chip enable. Should be high for normal operation. <ul style="list-style-type: none"> • 0 – Disabled • 1 – Enabled
RST	Yellow	External reset. <ul style="list-style-type: none"> • 0 – Reset • 1 – Normal
GPIO 0	Grey	Should be high on boot, low for flash update.
GPIO 2	Grey	Should be high on boot.
VCC	Orange	3.3V
GND	Purple	Ground

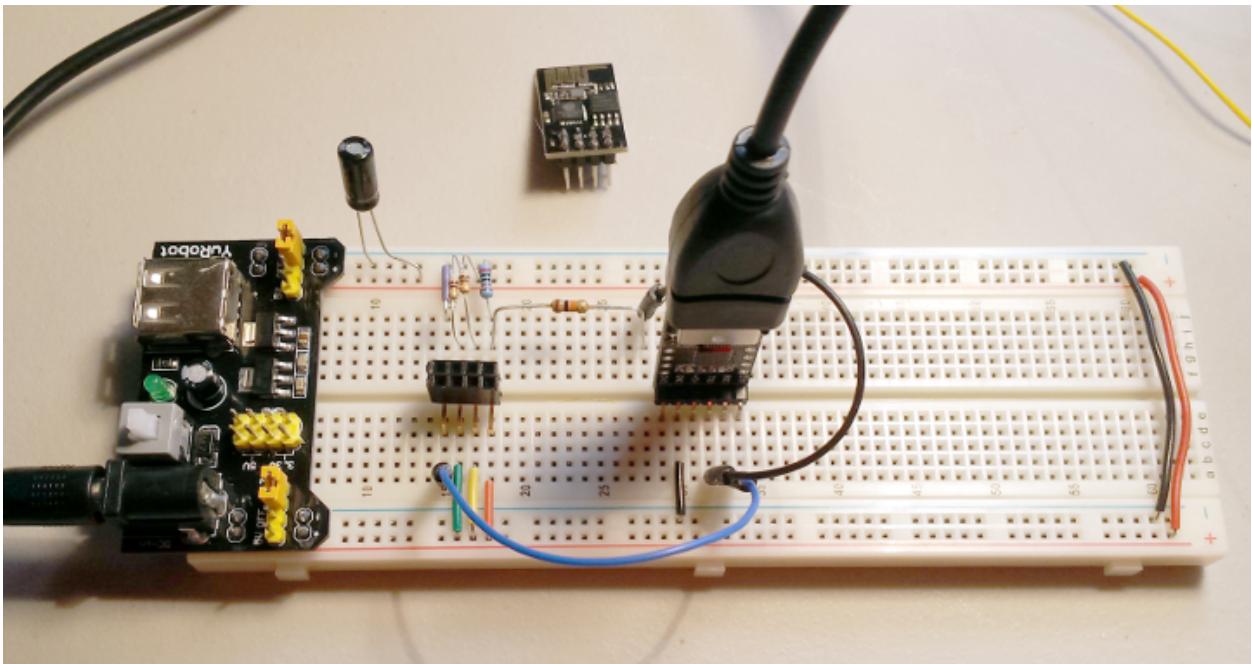
A simple circuit is shown below. Note that the TX and RX pins are shown **not** connected. Remember to **always** use a level converter for the RX pin into the device as it is **not** 5V tolerant.



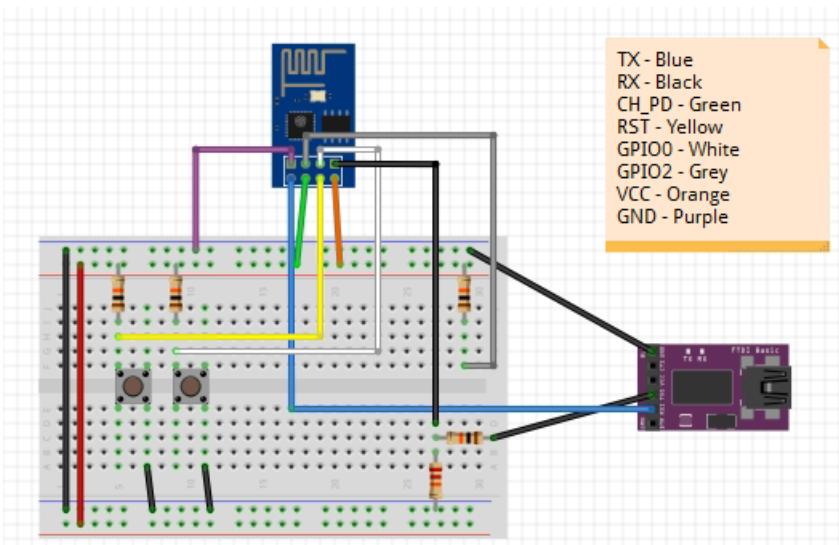
here is an alternate circuit:

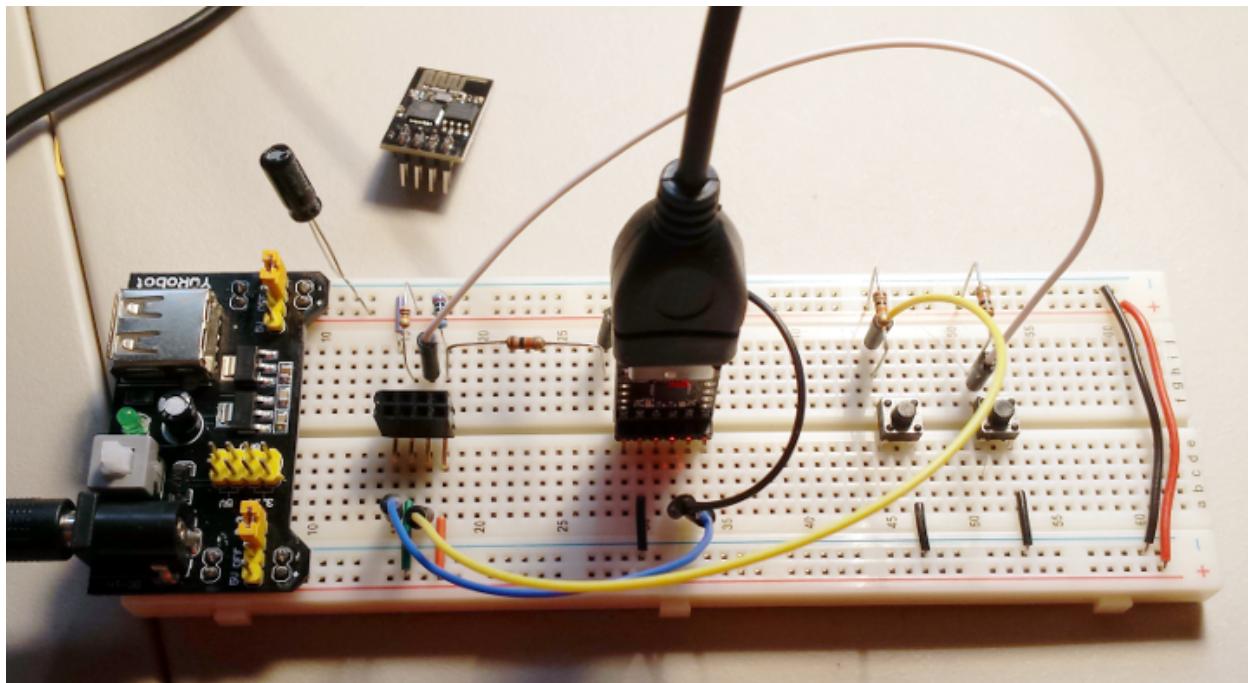
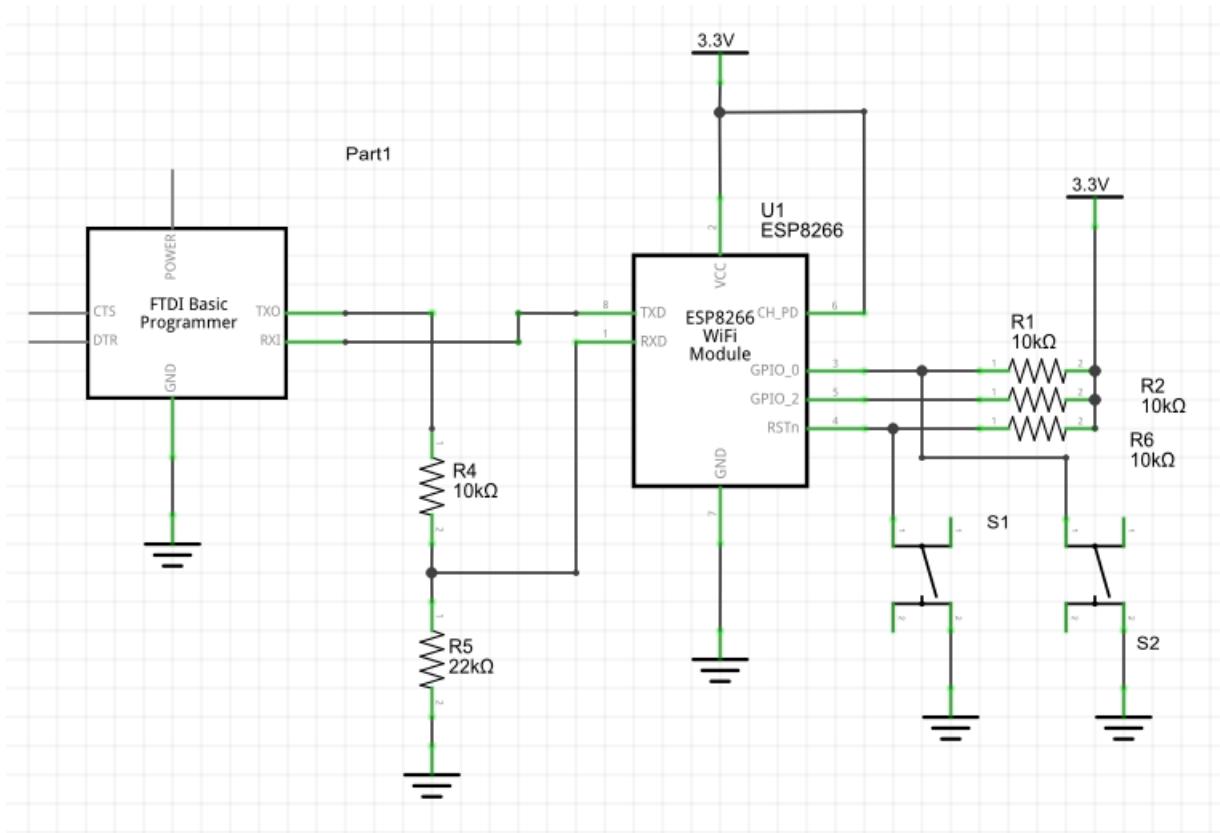


Here is the circuit on a breadboard that was demonstrated to work just fine.

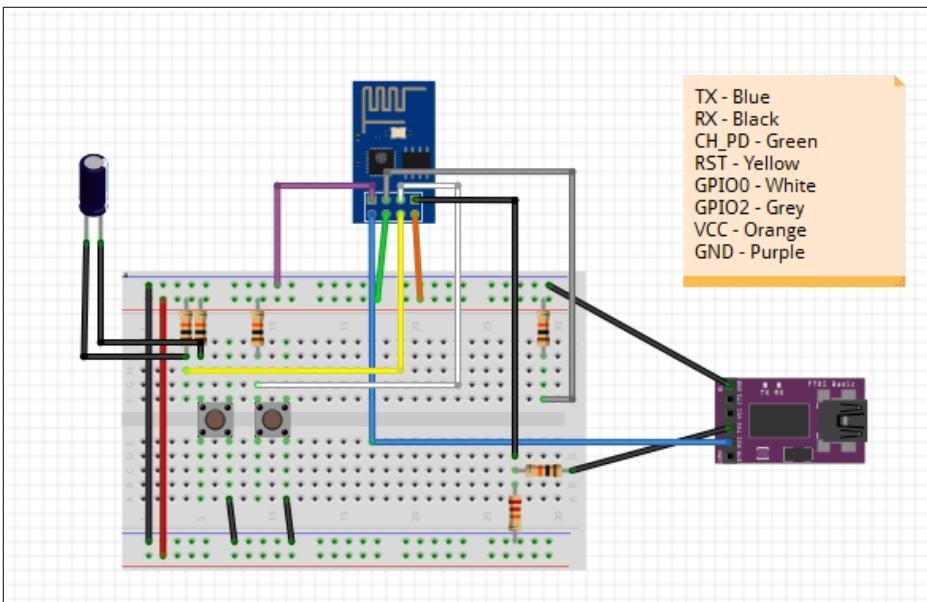
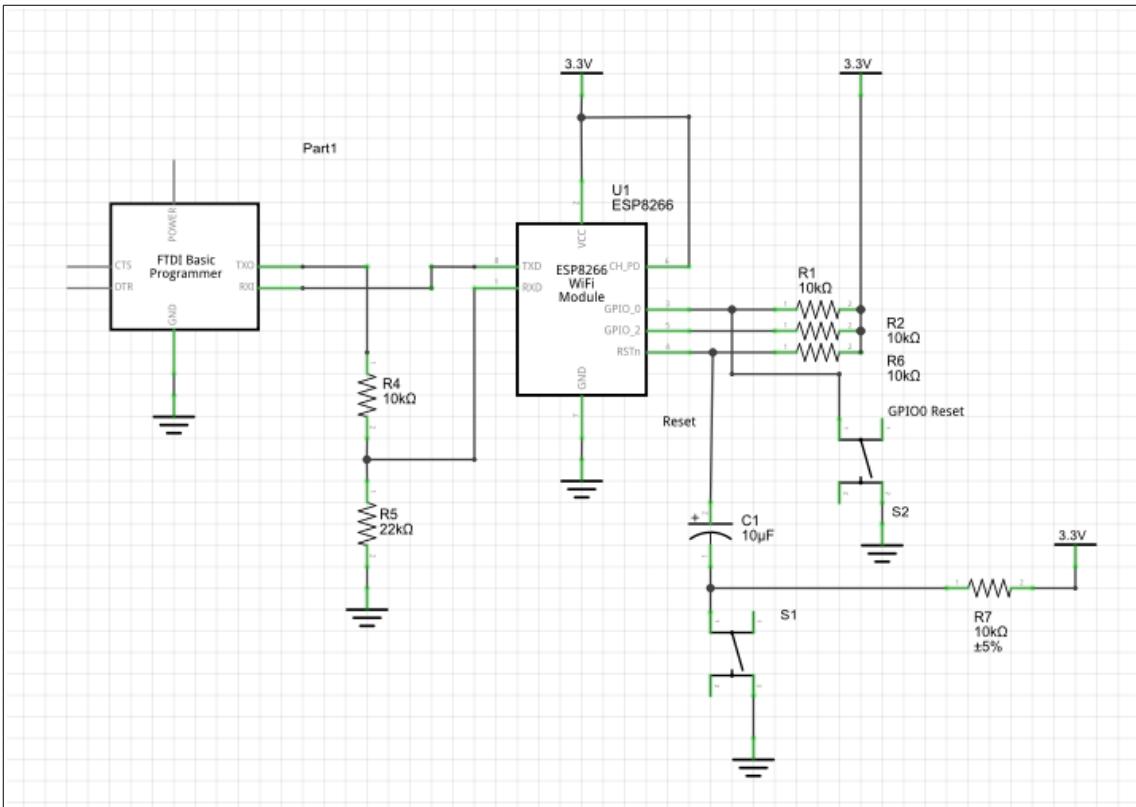


If we wish to add grounding buttons for RESET and GPIO 0, the following are some circuits:





When we press the reset button, it makes sense for that just to be a momentary press. Here is a circuit for that:

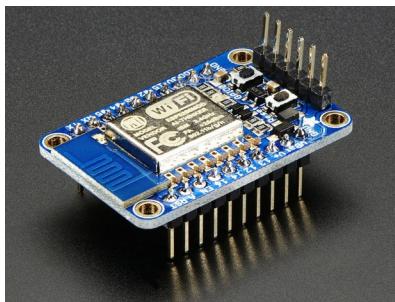


The default serial connection speed seems to be 115200.

See also:

- YouTube: [ESP8266 ESP-01 Pin details, Getting started](#)
- YouTube: [ESP8266 ESP-01 and USB to serial converter connections \(CP2102 Silicon Labs\)](#)

Adafruit HUZZAH



The Adafruit HUZZAH is a breakout board for the ESP8266. It is the most breadboard friendly of the solutions I have encountered so far.

See also:

- [Adafruit HUZZAH](#)

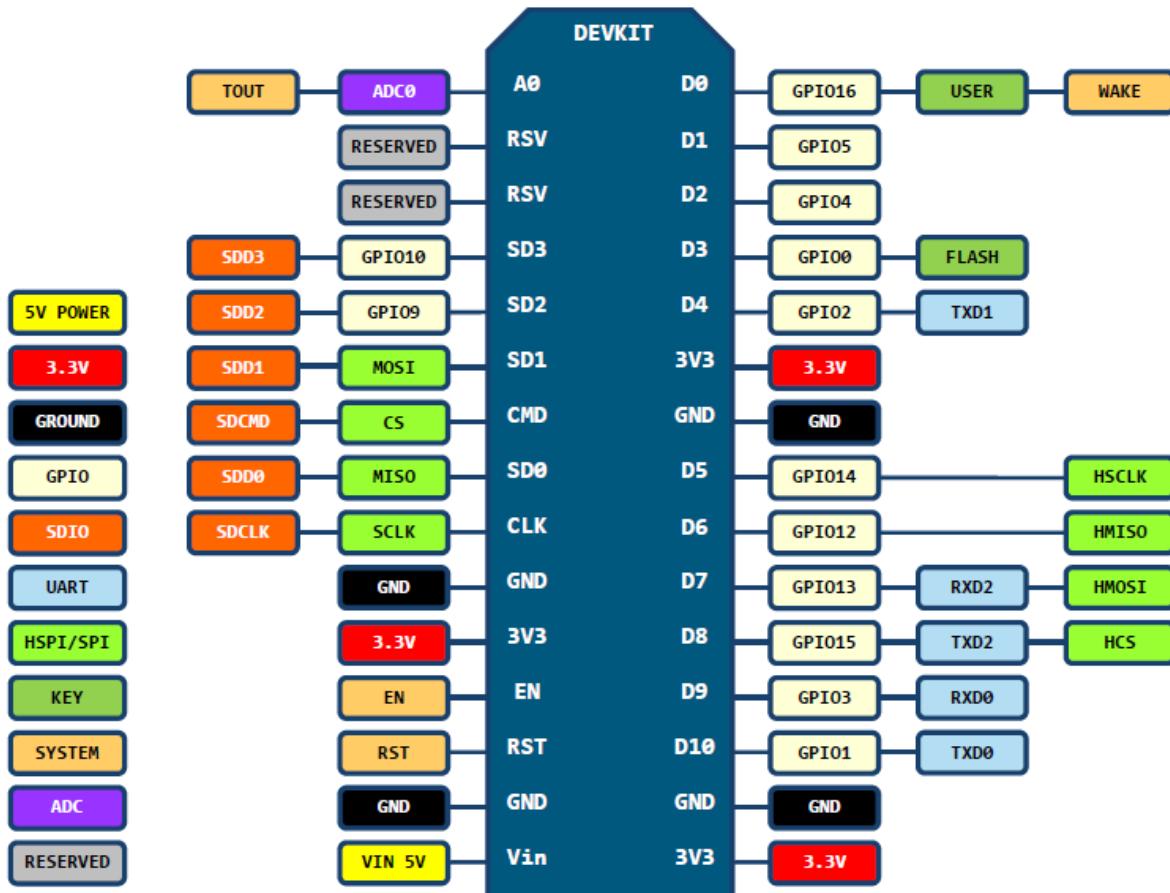
NodeMCU devKit

This module comes with a built in USB connector and a rich assortment of pin-outs. It is also immediately breadboard friendly (if one straddles two boards).



The pin mapping on this device is as follows:

PIN DEFINITION



D0(GPIO16) can only be used as gpio read/write, no interrupt supported, no pwm/i2c/ow supported.

There are currently two flavors of the NodeMCU board called v0.9 and v1.0. Here are the primary differences:

Function	NodeMCU v0.9	NodeMCU v1.0
USB	CH340 based	CP2012 based
ESP8266	ESP-12E	ESP-12E

When connected via USB on a Windows machine, the Serial connector shows as "USB-SERIAL CH340".

- ▼ Ports (COM & LPT)
- USB-SERIAL CH340 (COM5)

If for some reason the USB serial connector does not show up, search the internet for drivers for the CH340G for your PC's operating system.

This device is not especially single breadboard friendly (although it may leave one row of exposed pins) but fits beautifully on two breadboards that are side by side. I recommend adding a second USB→UART connected as follows:

- UART GND → GND
 - UART RX → GPIO2 (TXD1) [NodeMCU: D4]

In addition, a button between GND and RST will also provide a reset ability.

See also:

- [NodeMCU home page](#)
 - [GitHub: nodemcu/nodemcu-devkit-v1.0](#)
 - [Github: nodemcu/nodemcu-devkit](#)
 - [NodeMCU LUA docs](#)
 - [ESP8266: NodeMCU Dev Kit V1.0 Review](#)

node.IT (aka ESP-210)

See also:

- ESP-210

SparkFun WiFi Shield - ESP8266



SparkFun have produced a WiFi shield for the Arduino. This is an ESP8266 mounted on a well designed PCB that mates with the Arduino. This makes communicating with the ESP8266 via AT commands extremely easy with no wiring required. Simply push the shield board into the sockets of the Arduino and you are done.

See also:

- SparkFun WiFi Shield – ESP8266

Espresso Lite

See also:

- [Espresso lite](#)

Wemos D1

The Wemos D1 provides an Arduino Uno styled board complete with multiple power choices and female headers at both sides of the board. Should you be more comfortable working on an Arduino Uno sized platform, this makes a good candidate.

See also:

- [Wemos D1](#)

Oak by digistump

See also:

- [Oak by digistump](#)

Connecting to the ESP8266

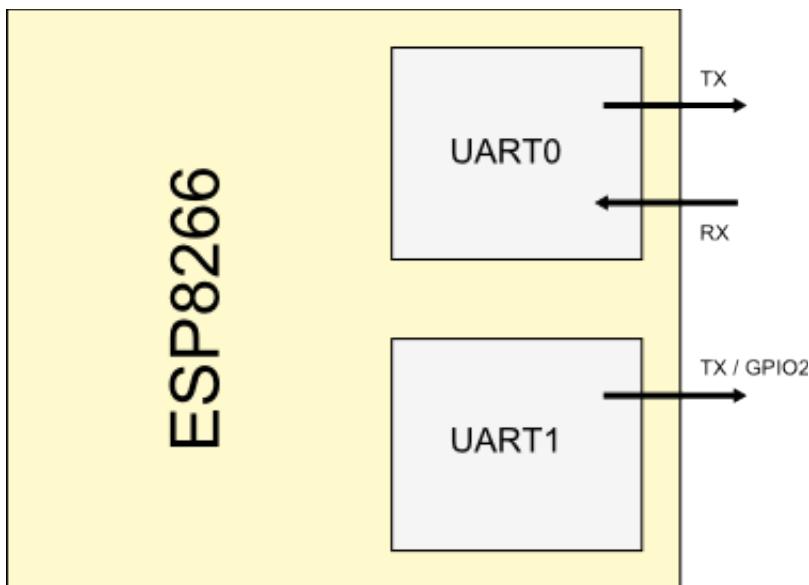
The ESP8266 is a WiFi device and hence we will eventually connect to it using WiFi protocols but some bootstrapping is required first. The device doesn't know what network to connect to, which password to use and other necessary parameters. This of course assumes we are connecting as a station, if we wish the device to be an access point or we wish to load our own applications into it, the story gets deeper. This implies that there is a some way to interact with the device other than WiFi and there is ... the answer is UART (Serial). The ESP8266 has a dedicated UART interface with pins labeled TX and RX. The TX pin is the ESP8266 transmission (outbound from ESP8266) and the RX pin is used to receive data (inbound into the ESP8266). These pins can be connected to a UART partner. By far the easiest and most convenient partner for us is a USB → UART converter. These are discussed in detail later in the book. For now let us assume that we have set those up. Through the UART, we can attach a terminal emulator to send keystrokes and have data received from the ESP8266 displayed as characters on the screen. This is used extensively when working with the AT commands. A second purpose of the UART is to receive binary data used to "flash" the flash memory of the device to record new applications for execution. There are a variety of technical tools at our disposal to achieve that task.

When we use a UART, we need to consider the concept of a baud rate. This is the speed of communication of data between the ESP8266 and its partner. During boot, the ESP8266 attempts to automatically determine the baud rate of the partner and match it. It assumes a default of 74880 and if you have a serial terminal attached, you will see a message like:

ets Jan 8 2013,rst cause:2, boot mode:(1,0)

if it is configured to receive at 74880.

The ESP8266 has a second UART associated with it that is output only. One of the primary purposes of this second UART is to output diagnostics and debugging information. This can be extremely useful during development and as such I recommend attaching **two** USB → UART converters to the device. The second UART is multiplexed with pin GPIO2.



See also:

- USB to UART converters
- AT Command Programming
- Loading a program into the ESP8266

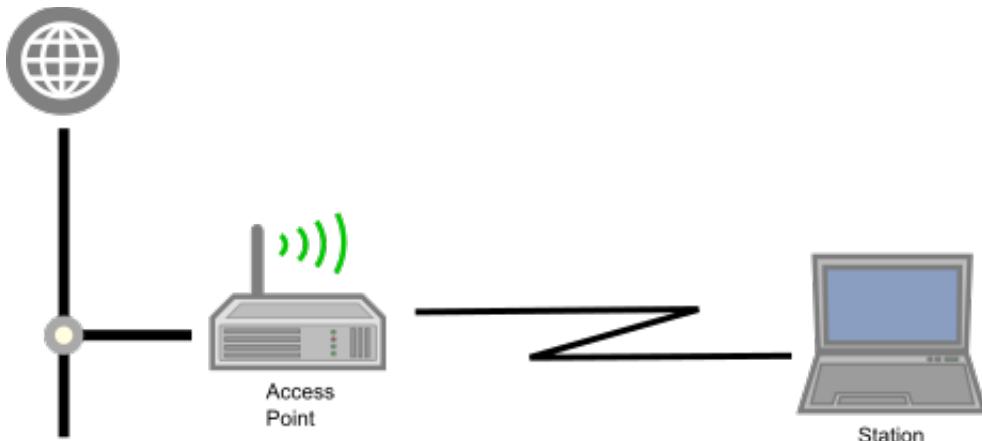
WiFi Theory

When working with a WiFi oriented device, it is important that we have at least some understanding of the concepts related to WiFi. At a high level, WiFi is the ability to participate in TCP/IP connections over a wireless communication link. WiFi is specifically the set of protocols described in the IEEE 802.11 Wireless LAN architecture.

Within this story, a device called a Wireless Access Point (access point or AP) acts as the hub of all communications. Typically it is connected to (or acts as) a TCP/IP router to the rest of the TCP/IP network. For example, in your home, you are likely to have a WiFi access point connected to your modem (cable or DSL). WiFi connections are then formed to the access point (through devices called stations) and TCP/IP traffic flows through the access point to the Internet.



The devices that connect to the access points are called "stations":



An ESP8266 device can play the role of an Access Point, a Station or both at the same time.

Very commonly, the access point also has a network connection to the Internet and acts as a bridge between the wireless network and the broader TCP/IP network that is the Internet.

A collection of stations that wish to communicate with each other is termed a Basic Service Set (BSS). The common configuration is what is known as an Infrastructure BSS. In this mode, all communications inbound and outbound from an individual station are routed through the access point.

A station must associate itself with an access point in order to participate in the story. A station may only be associated with a single access point at any one time.

Each participant in the network has a unique identifier called the MAC address. This is a 48bit value.

When we have multiple access points within wireless range, the station needs to know with which one to connect. Each access point has a network identifier called the BSSID (or more commonly just SSID). SSID is **service set identifier**. It is a 32 character value that represents the target of packets of information sent over the network.

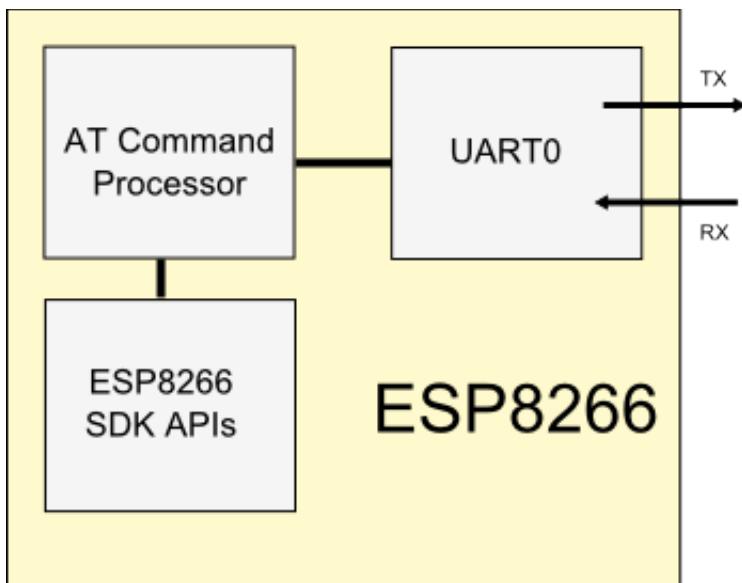
See also:

- Wikipedia – [Wireless access point](#)
- Wikipedia – [IEEE 802.11](#)
- Wikipedia – [WiFi Protected Access](#)
- Wikipedia – [IEEE 802.11i-2004](#)

AT Command Programming

The quickest and easiest way to get started with an ESP8266 is to access it via the AT command interface.

When we think about an ESP8266 device we find that it has a built in UART (Serial) connection. This means that it can both send and receive data using the UART protocol. We also know that the device can communicate with WiFi. What if we had an application that ran on the ESP8266 that took "instructions" received over the serial link, executed them and then returned a response? This would then allow us to use the ESP8266 without ever having to know the programming languages that are native to the device. This is exactly what a program that has so far been found to be pre-installed on the ESP8266 does for us. The program is called the "AT command processor" named after the format of the commands sent through the serial link. These commands are all prefixed with "AT" and follow (roughly) the style known as the "Hayes command set".



If we think of an application wishing to use the services of the ESP8266 as a client and the ESP8266 as a server capable of servicing those commands as a server, then the client sends strings of characters through the UART connection to the server and server responds with the outcome.

Espressif publish a complete set of AT command documentation which can be found in their forum page at:

- http://espressif.com/sites/default/files/documentation/4a-esp8266_at_instruction_set_en.pdf

There are two primary documents:

- ESP8266EX AT Instruction Set
- ESP8266EX AT Command Examples

Commands

When one has wired an ESP8266 to a serial converter, the next question will be "Is it working?". When we connect a serial monitor, the first command we can send is "AT" which should respond with a simple "OK".

An instruction passed to the device follows one of the following syntax options:

Type	Format	Description
Test	AT+<x>=?	Query the parameters and its range of values.
Query	AT+<x>?	Return the current value of the parameter.
Set	AT+<x>=<...>	Set the value of a parameter.
Execute	AT+<x>	Execute a command.

All "AT" instructions end with the "\r\n" pair.

Command	Description
AT	Returns OK
AT+RST	Restart the ESP8266.
AT+GMR	Returns firmware version for both the AT command processor and the SDK in use. Currently, the response returned looks like: AT version:0.21.0.0 SDK version:0.9.5
AT+GSLP=<time>	Put the device into a deep sleep for a time in milliseconds. It will wake up after this period.
ATE [0 1]	Echo AT commands. <ul style="list-style-type: none"> • ATE0 – Echo commands off • ATE1 – Echo commands on
AT+RESTORE	Restore the defaults of settings in flash memory.
AT+UART_CUR=<baudrate>, <databits>, <stopbits>, <parity>, <flow control>	The databits can be 5, 6, 7 or 8. parity can be 0=none, 1=odd, 2=even flowcontrol can be: 0 – disable 1 – enable RTS 2 – enable CTS 3 – enable both RTS and CTS
AT+UART_DEF=<baudrate>, <databits>, <stopbits>, <parity>, <flow control>	
AT+SLEEP?	
AT+SLEEP=<sleep mode>	
AT+RFPOWER=<TX power>	
AT+RFVDD?	
AT+RFVDD=<VDD33>	
AT+RFVDD	
WIFI	
AT+CWMODE_CUR=<mode>	Sets the current mode of operation. <ul style="list-style-type: none"> • 1 – Station mode • 2 – AP mode • 3 – AP + Station mode

AT+CWMODE_CUR?	Get the current mode of operation.
AT+CWMODE_CUR=?	Get the list of available modes.
AT+CWMODE_DEF=<mode>	Sets the current mode of operation. <ul style="list-style-type: none"> • 1 – Station mode • 2 – AP mode • 3 – AP + Station mode
AT+CWMODE_DEF?	Get the current mode of operation.
AT+CWMODE_DEF=?	Get the list of available modes.
AT+CWJAP_CUR=<ssid>, <password>[, <bssid>]	Join the WiFi network (JAP = Join Access Point).
AT+CWJAP_CUR?	Get the current connection info.
AT+CWJAP_DEF=<ssid>, <password>[, <bssid>]	Join the WiFi network (JAP = Join Access Point).
AT+CWJAP_DEF?	Get the current connection info.
AT+CWLAP	List the "List Access Points". The response is: + CWLAP: <ecn>, <ssid>, <rssi>, <mac>, <ch> where: <ul style="list-style-type: none"> • ecn <ul style="list-style-type: none"> ◦ 0 – OPEN ◦ 1 – WEP ◦ 2 – WPA_PSK ◦ 3 – WPA2_PSK ◦ 4 – WPA_WPA2_PSK • ssid – SSID of AP • rssi – Signal strength • mac – MAC address • ch – Channel
AT+CWLAP=<ssid>, <mac>, <ch>	List a filtered set of access points.
AT+CWQAP	Disconnect from AP.
AT+CWSAP_CUR?	Configuration of softAP mode
AT+CWSAP_CUR=<ssid>, <pwd>, <chl>, <ecn>	
AT+CWSAP_DEF?	Configuration of softAP mode

AT+CWSAP_DEF=<ssid>, <pwd>, <chl>, <ecn>	
AT+CWLIF	List of IPs connected in softAP mode
AT+CWDHCP_CUR?	
AT+CWDHCP_CUR=<mode><en>	Enable or disable DHCP. <ul style="list-style-type: none"> • mode <ul style="list-style-type: none"> ◦ 0 – softAP ◦ 1 – station ◦ 2 – softAP + station • en <ul style="list-style-type: none"> ◦ 0 – Enable ◦ 1 – Disable
AT+CWDHCP_DEF?	
AT+CWDHCP_DEF=<mode><en>	Enable or disable DHCP. <ul style="list-style-type: none"> • mode <ul style="list-style-type: none"> ◦ 0 – softAP ◦ 1 – station ◦ 2 – softAP + station • en <ul style="list-style-type: none"> ◦ 0 – Enable ◦ 1 – Disable
AP+CWAUTOCONN=<enable>	
AT+CIPSTAMAC_CUR?	Set/get MAC address of station.
AT+CIPSTAMAC_CUR=<mac>	Set/get MAC address of station.
AT+CIPSTAMAC_DEF?	Set/get MAC address of station.
AT+CIPSTAMAC_DEF=<mac>	Set/get MAC address of station.
AT+CIPAPMAC_CUR?	Set/get MAC address of softAP.
AT+CIPAPMAC_CUR=<mac>	Set/get MAC address of softAP.
AT+CIPAPMAC_DEF?	Set/get MAC address of softAP.
AT+CIPAPMAC_DEF=<mac>	Set/get MAC address of softAP.
AT+CIPSTA_CUR=<iP>	Set the ip address of station.
AT+CIPSTA_CUR?	Get the IP address of station. For example: +CIPSTA:"0.0.0.0"

AT+CIPSTA_DEF=<IP>	Set the ip address of station.
AT+CIPSTA_DEF?	Get the IP address of station. For example: +CIPSTA:"0.0.0.0"
AT+CIPAP_CUR?	Set the ip address of softAP.
AT+CIPAP_CUR=<IP>[,<gat eway>, <netmask>]	Set the ip address of softAP.
AT+CIPAP_DEF?	Set the ip address of softAP.
AT+CIPAP_DEF=<IP>[,<gat eway>, <netmask>]	Set the ip address of softAP.
AT+CIFSR	Returns the IP address and gateway IP address.

TCP/IP networking

AT+CIPSTATUS	<p>Information about connection. Response format is:</p> <p>STATUS: <stat> + CIPSTATUS: <id>, <type>, <addr>, <port>, <tetype></p> <ul style="list-style-type: none"> • stat <ul style="list-style-type: none"> ◦ 2 – Got IP ◦ 3 – Connected ◦ 4 – Disconnected • id – Id of the connection • type – TCP or UDP • addr – IP address • port – Port number • tetype <ul style="list-style-type: none"> ◦ 0 – ESP8266 runs as client ◦ 1 – ESP8266 runs as server
AT+CIPSTART=<type>,<addr>, <port>[, <local port>, <mode>]	<p>Start a connection when CIPMUX=0.</p> <ul style="list-style-type: none"> • type – TCP or UDP • addr – Remote IP address • port – Remote port • local port – For UDP only • mode – For UDP only <ul style="list-style-type: none"> ◦ 0 – destination peer entity of UDP is fixed ◦ 1 – destination peer entity may change once ◦ 2 – destination peer entity may change
AT+CIPSTART=<id>,<type>, <addr>,<port>[, <local port>, <mode>]	<p>Start a connection when CIPMUX=1.</p> <ul style="list-style-type: none"> • id – 0-4 value of connection

<port>[,<local port>,<mode>]	<ul style="list-style-type: none"> • type – TCP or UDP • addr – Remote IP address • port – Remote port • local port – For UDP only • mode – For UDP only <ul style="list-style-type: none"> ◦ 0 – destination peer entity of UDP is fixed ◦ 1 – destination peer entity may change once ◦ 2 – destination peer entity may change
AT+CIPSTART=?	???
AT+CIPSEND=<length>	Send length characters.
AT+CIPCLOSE	Close a connection.
AT+CIFSR	Get the local IP address.
AT+CIPMUX=<mode>	Enable multiple connections. <ul style="list-style-type: none"> • 0 – Single connection. • 1 – Multiple connections.
AT+CIPMUX?	Returns the current value for CIPMUX. <ul style="list-style-type: none"> • 0 – Single connection. • 1 – Multiple connections.
AT+CIPSERVER=<mode> [,<port>]	Configure as a TCP server. If no port is supplied, default is 333. A server may only be created when CIPMUX=1 (allow multiple connections). <ul style="list-style-type: none"> • mode <ul style="list-style-type: none"> ◦ 0 – Delete server (needs a restart after) ◦ 1 – Create server
AT+CIPMODE=<mode>	Set the transfer mode. <ul style="list-style-type: none"> • 0 – Normal mode. • 1 – Unvarnished mode.
AT+CIPSTO=<time>	Set server timeout. A value in the range of 0 – 7200 seconds.
AT+CIUPDATE	???

See also:

- YouTube – [ESP8266 Tutorial AT Commands](#)

Installing the latest AT command processor

The latest AT command processor can always be downloaded in binary form from the Espressif web site. Always view the README that comes with the files and follow the instructions contained within. In order to load the firmware images, you will need a

flashtool. In addition, you will need the files supplied with download. As of v0.50, the files needed and the addresses to be loaded into are:

File	Address
nonboot/eagle.flash.bin	0x00000
nonboot/eagle.irom0text.bin	0x40000
blank.bin	0x3E000
blank.bin	0x7E000

These instructions are for 512K flash chips.

See also:

- Espressif – [Download of AT command processor – V0.50](#)

Assembling circuits

Since the ESP8266 is an actual electronic component, some physical assembly is required. This book will not attempt to cover non-ESP8266 electronics as that is a very big and broad subject in its own right. However, what we will do is describe some of the components that we have found extremely useful while building ESP8266 solutions.

USB to UART converters

You can't program an ESP8266 without supplying it data through a UART. The easiest way to achieve this is through the use of a USB to UART converter. I use the devices that are based upon the CP2102 which can be found cheaply on ebay for under \$2 each. Another popular brand are the devices from Future Technology Devices International (FTDI). You will want at least two. One for programming and one for debugging. I suggest buying more than two just in case ...



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

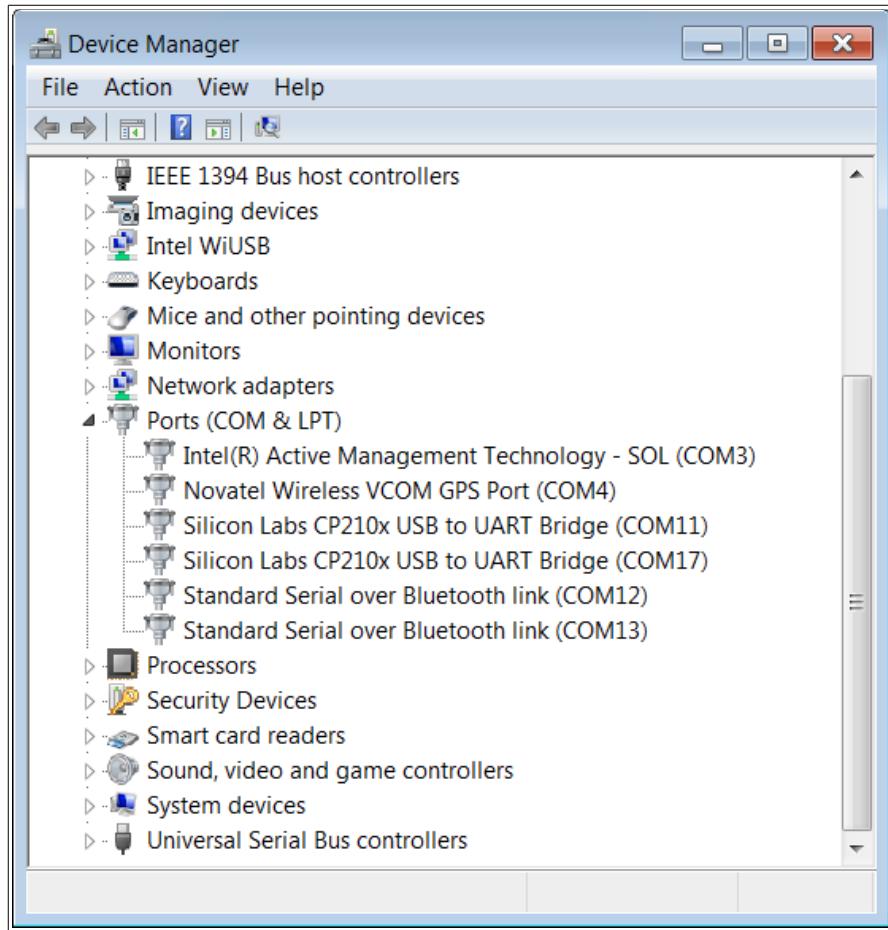
type of connector cable I recommend. Get them with as short a cable length as possible. 12-24 inches should be preferred.



When we plug in a USB → UART into a Windows machine, we can learn the COM port that the new serial port appears upon by opening the Windows Device Manager. There are a number of ways of doing this, one way is to launch it from the DOS command window with:

```
mmc devmgmt.msc
```

Under the section called Ports (COM & LPT) you will find entries for each of the COM ports. The COM ports don't provide you a mapping that a particular USB socket is hosting a particular COM port so my poor suggestion is to pull the USB from each socket one by one and make a note of which COM port disappears (or appears if you are inserting a USB).

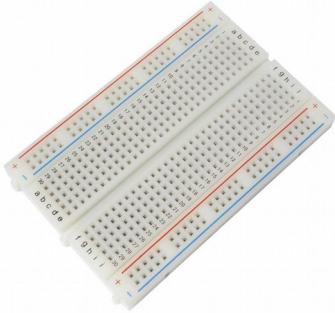


See also:

- Connecting to the ESP8266
- Working with serial

Breadboards

I find I can never have too many breadboards. I suggest getting a few full size and half size boards along with some 24 AWG connector wire and a good pair of wire strippers. Keep a trash bin close by otherwise you will find yourself knee deep in stripped insulation and cut wire parts before you know it. I also recommend some Dupont male-male pre-made wires. Ribbon cable can also be useful.



Power

We need electricity to get these devices working. I choose the MB102 breadboard attachable power adapters. These can be powered from an ordinary wall-wart (mains adapter) or from USB. It appears that the plug for wall-wart power is 2.1mm and center positive however I strongly suggest that you read your specific supplier's data sheets very carefully. There is also a potential concern that the barrel socket is wired in parallel with the USB input which could mean that if you attach a high voltage input (eg. 12V) while also having a USB source connected, you may very well damage your USB device. The devices have a master on/off power switch plus a jumper to set 3.3V or 5V outputs. You can even have one breadboard rail be 3.3V and the other 5V ... but take care not to apply 5V to your ESP8266. By having two power rails, one at 3.3V and the other at 5V, you can power both the ESP8266 and devices/circuits that require 5V.



When the ESP8266 starts to transmit over wireless, that can draw a lot of current which can cause ripples in your power supply. You may also have other sensors or devices connected to your supply as well. These fluctuations in the voltage can cause problems. It is strongly recommended that you place a 10 micro farad capacitor between +ve and -ve as close to your ESP8266 as you can. This will provide a reservoir of power to even out any transient ripples. This is one of those tips that you ignore at your peril. Everything may work just fine without the capacitor ... until it doesn't or until you start getting intermittent problems and are at a loss to explain them. Let me put it this way, for the few cents it costs and the zero harm it does, why not?

Multi-meter / Logic probe / Logic Analyzer

When your circuit doesn't work and you are staring at it wondering what is wrong, you will be thankful if you have a multi-meter and a logic probe. If your budget will stretch, I also recommend a USB based logic analyzer such as those made by Saleae. These allow you to monitor the signals coming into or being produced by your ESP8266. Think of this as the best source of debugging available to you.

See also:

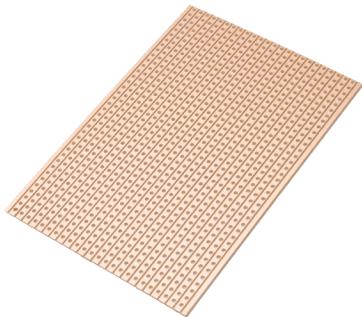
- [Saleae logic analyzers](#)

Sundry components

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

Physical construction

When you have breadboarded your circuit and written your application, there may come a time where you wish to make your solution permanent. At that point, you will need a soldering iron, solder and some strip-board. I also recommend some female header sockets so that you don't have to solder your ESP8266s directly into the circuits. Not only does this allow you to reuse the devices (should you desire) but in the unfortunate event that you fry one, it will be easier to replace.

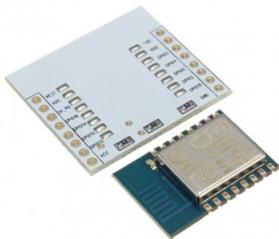


Recommended setup for programming ESP8266

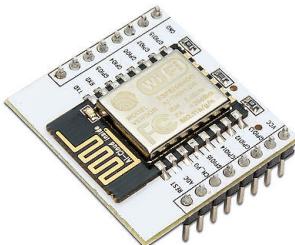
Obviously in order to program an ESP8266, you will actually need to obtain an ESP8266 but it isn't that easy. The actual ESP8266 itself is a tiny integrated circuit and you are unlikely to be able to use it directly. Instead, you will buy one of the many styles of breakout boards that already exist. The common ones are the ESP-1 which exposes 2 GPIO pins and the ESP-12 which exposes 9. I recommend the ESP-12 as it is only marginally more expensive for the extra pins exposed.



You will also need a mounting board as the ESP-12 by itself doesn't have connector pins. You can commonly buy both the ESP-12 and the mounting board together at the same time. However, check carefully, the mounting boards can be bought separately and you need to validate that when you order and assume you are getting both that you are not *just* buying the mounting boards without the ESP8266. You will be disappointed.



The ESP-12 is then soldered onto the mounting board so you will need a soldering iron and some fine grained hand control. The soldering is not the easiest in the world as the pins are extremely close together. For this reason and for others, I'd suggest buying multiple ESP-12s and mounting boards instead of just one. It is also not difficult to fry your ESP-12 if you get some wiring wrong. Once assembled, it should look as follows:



Mine never look this "clean" when build as my solder resin seems to discolor the original attractive white base of the mounting board. However, looks aren't important.

Assuming you now have a mounted ESP-12 with pins, your next question will be "now what"? This is where you will want a few breadboards and connector wire. You could use dupont connectors with female sockets attached to the ESP-12 and male pins on the other to attach to your breadboard but you will find that wires inevitably come loose at the worse possible times. You can mount the ESP-12 to a breadboard but I tend to find that there is not enough space for connector wires underneath it.

Once secured, I recommend **two** USB → UART connectors. Why two? One dedicated for flashing the device and one for debugging.

For power, I recommend using MB102 breadboard power supplies however, make sure that you set the jumper cables to be 3.3V. You will ruin your ESP8266 if you try and power it at 5V.

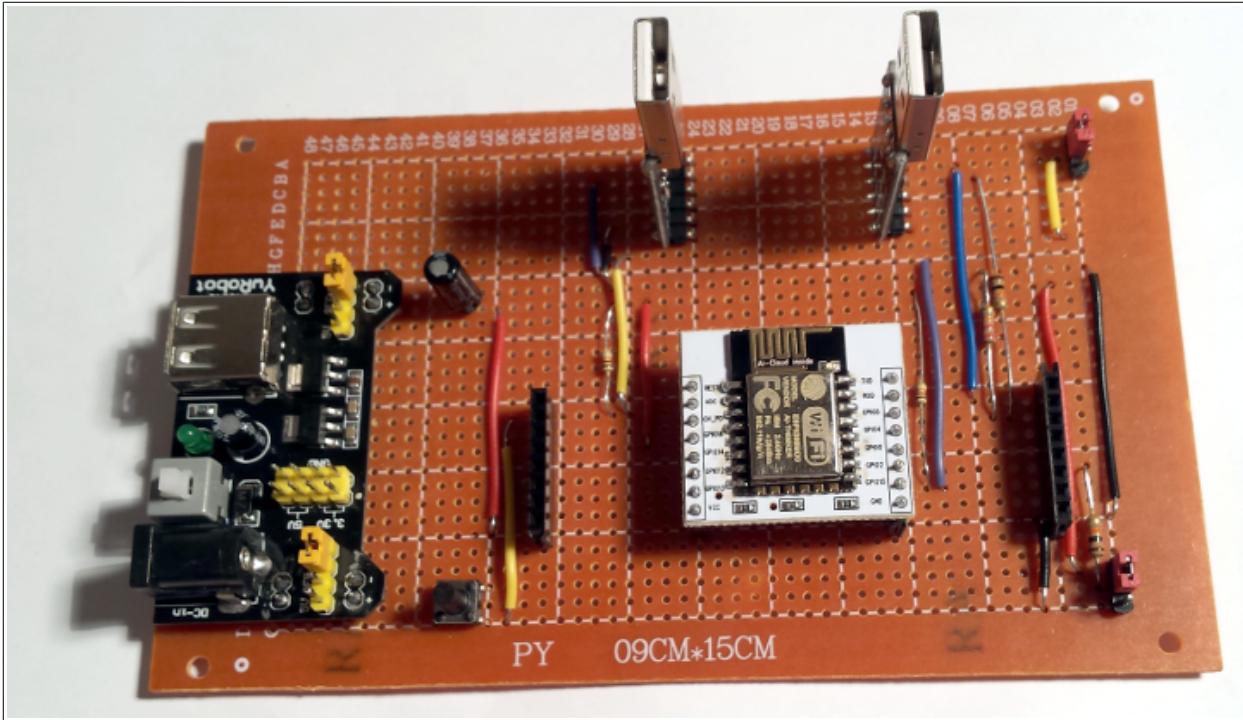
Once it is all wired up, you will need a PC with two open USB ports.

Parts list

- Breadboards – 2 half size – \$3.50 for 2
- ESP-12 plus mounting boards – 3 sets – \$3.80 each – \$11.40
- CP2102 USB → UARTs – 2 pieces - \$3.10
- USB male to female extenders – 2 pieces – \$1.00 each – \$2.00
- 24 AWG wire – 5 meters for \$1.12
- 8pin 2.54mm stackable long legged female headers – 10 pieces for \$3.95
- Red diffuse LEDs – A handful – \$1.00
- Resistors – Some 10K, some 20K, some 330Ohm – A handful – \$1.00
- Capacitors – Some 10 micro farad – \$1.00

All told, it comes to about \$30 + some shipping. I buy all my components through ebay from Chinese suppliers that give me the price/quality I am looking for. The name of the game though is patience. Once you order it usually takes 2-3 weeks for the parts to arrive so be patient and use the time to watch you-tube videos on electronics projects and the relevant community forums.

Eventually, you are likely going to want to build a permanent circuit for your development. On a strip board the circuit I built looks like:



Configuration for flashing the device

Later on in the book you will find that when it comes time to flash the device with your new applications, you will have to set some of the GPIO pins to be low and then reboot. This is the indication that it is now ready to be flashed. Obviously, you can build a circuit that you use for flashing your firmware and then place the device in its final circuit but you will find that during development, you will want to flash and test pretty frequently. This means that you will want to use jumper wires and to allow you to move the links of pins on your breadboards from their "flash" position to their "normal use" position.

Programming

The ESP8266 allows you to write applications that can run natively on the device. You can compile C language code and deploy it to the device through a process known as flashing. In order for your applications to do something useful, they have to be able to interact with the environment. This could be making network connections or sending/receiving data from attached sensors, inputs and outputs. In order to make that happen, the ESP8266 contains a core set of functions that we can loosely think of as the operating system of the device. The services of the operating system are exposed to be called from your application providing a contract of services that you can leverage. These services are fully documented. In order to successfully write applications for deployment, you need to be aware of the existence of these services. They become indispensable tools in your tool chest. For example, if you need to connect to a WiFi access point, there is an API for that. To get your current IP address, there is an API for that and to get the time since the device was started, there is an API for that. In fact, there are a LOT of APIs available for us to use. The good news is that no-one is expecting us to memorize all the details of their use. Rather it is sufficient to broadly know that they exist and have somewhere to go when you want to look up the details of how to use them.

To sensibly manage the number and variety of these exposed APIs, we can collect sets of them together in meaningful groups of related functions. This gives us yet another and better way to manage our knowledge and learning of them.

The primary source of knowledge on programming the ESP8266 is the ESP8266 SDK API Guide. Direct links to all the relevant documents can be found at Reference documents.

See also:

- [Espressif Systems](#) – Manufacturers of the ESP8266
- [Espressif Bulletin Board System](#) – Place for SDKs, docs and forums

Boot mode

When the ESP8266 boots, the values of the pins known as `MTDO`, `GPIO0` and `GPIO2` are examined. The combination of the high or low values of these pins provide a 3 bit number with a total of 8 possible values from 000 to 111. Each value has a possible meaning interpreted by the device when it boots.

Value [15-0-2]	Decimal Value	Meaning
000	0	Remapping ... details unknown.
001	1	Boot from the data received from UART0. Also includes flashing the flash memory for subsequent normal starts.
010	2	Jump start
011	3	Boot from flash memory
100	4	SDIO low speed V2
101	5	SDIO high speed V1
110	6	SDIO low speed V1
111	7	SDIO high speed V2

From a practical perspective, what this means is that if we wish the device to run normally, we want to boot from flash with the pins having values 011 while when we wish to flash the device with a new program, we want to supply 001 to boot from UART0.

Note that `MTDO` is also known as `GPIO15`.

ESP8266 - Software Development Kit (SDK)

Include directories

The C programming language uses a text based pre-processor to include data in the compilation. The C pre-processor has the ability to include additional C source files that, by convention, are called header files and end with the ".h" prefix. Within these files we commonly find definitions of data types and function prototypes that are used during compilation. The ESP8266 SDK provides a directory called "include" which contains the include files supplied by Espressif for use with the ESP8266. The list of header files that we may use are as described in the following table:

File	Notes
at_custom.h	Definitions for custom extensions to the AT command handler.
c_types.h	C language definitions.
eagle_soc.h	Low level definitions and macros. Heavily related to bit twiddling at the CPU level. No idea why the file is called "eagle".
espconn.h	TCP and UDP definitions. This has pre-reqs of c_types.h and ip_addr.h.
espnow.h	Functions related to the esp now support.
ets_sys.h	Unknown.
gpio.h	Definitions for GPIO interactions.
ip_addr.h	IP address definitions and macros.
mem.h	Definitions for memory manipulation and access.
os_type.h	OS type definitions.
osapi.h	Includes a user supplied header called "user_config.h".
ping.h	Definitions for the ping capability.
pwm.h	Definitions for PWM.
queue.h	Queue and list definitions.
smartconfig.h	Definitions for smart config.
sntp.h	Definitions for SNTP.
spi_flash.h	Definitions for flash.
upgrade.h	Definitions for upgrades.
user_interface.h	Definitions for OS and WiFi. I have no explanation for why this file is named "user_interface" as there is obviously no UI involved with ESP8266s.

Compiling

Application code for an ESP8266 program is commonly written in C. Before we can deploy an application, we must compile the code into binary machine code instructions. Before that though, let us spend a few minutes thinking about the code.

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

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

For our ESP8266 story, this means that if we can find (which we can) a set of C compiler tools that take C source and generate Xtensa binary, we can use CDT to build our programs.

To make things more interesting though, we need to realize that C is not the only language we can use for building ESP8266 applications. We can also use C++ and assembly. You may be surprised that I mention assembly as that is as low level as we can possibly get however there are odd times when we need just that (thankfully rarely) ... especially when we realize that we are pretty much programming directly to the metal. The Arduino libraries (for example) have at least one assembly language file.

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

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

To perform the compilations, we need a set of development tools.

My personal preference is the package for Eclipse which has everything pre-built and ready for use. However, these tools can also be downloaded from the Internet as open source projects on a piece by piece basis.

The macro `LOCAL` is a synonym for the C language keyword "`static`".

From reading the docs, no published example of how to compile was found. However, when one uses the Eclipse open source project, one can see the Makefiles that are used and this exposes examples of compilation.

A typical compilation looks like:

```
17:57:16 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f
C:/Users/IBM_ADMIN/Documents/RaspberryPi/ESP8266/EclipseDevKit/WorkSpace/k_blinky/Make
file all
CC user/user_main.c
AR build/app_app.a
LD build/app.out
-----
Section info:

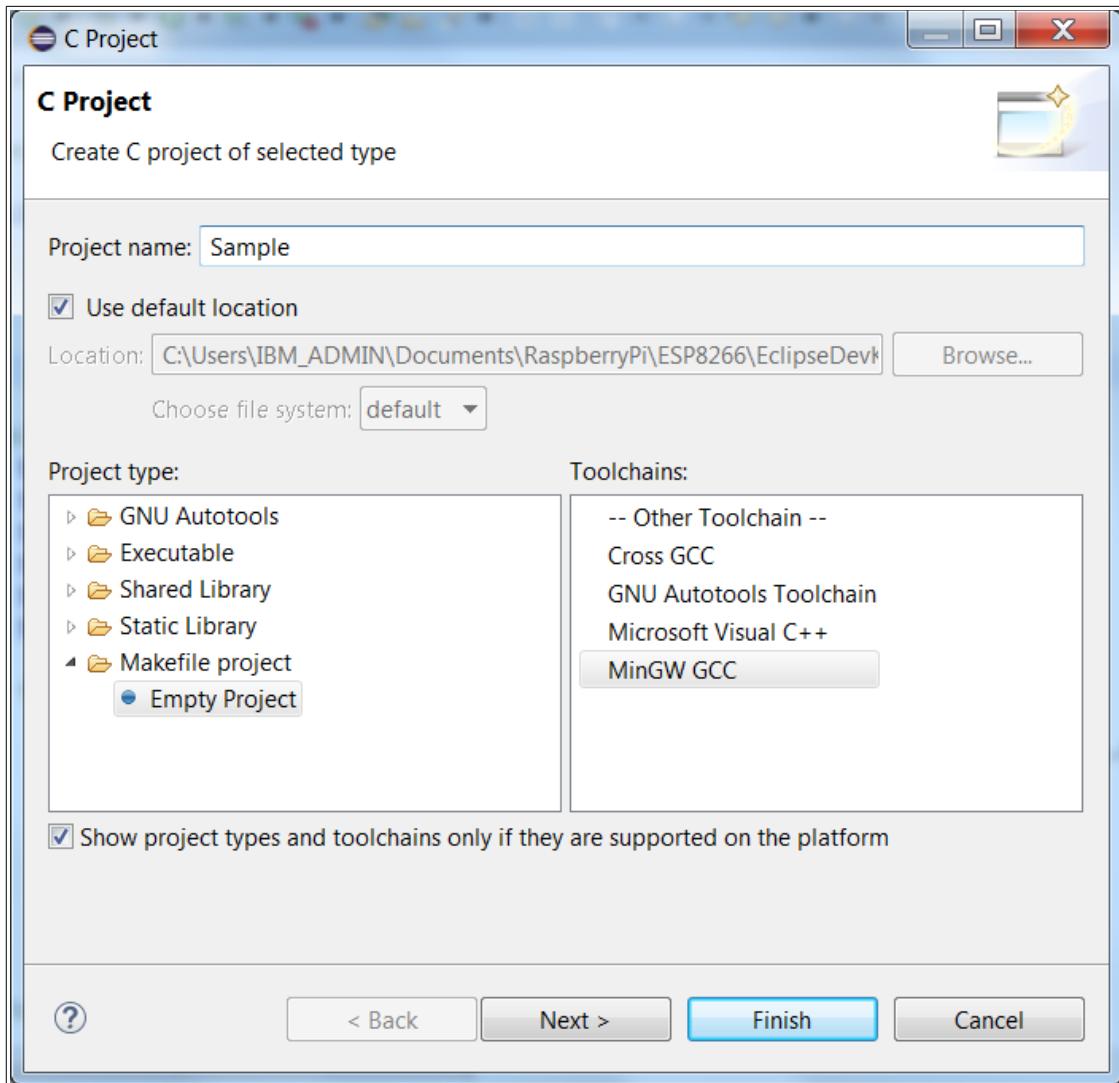
build/app.out:      file format elf32-xtensa-le

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .data     0000053c 3ffe8000 3ffe8000 000000e0 2**4
              CONTENTS, ALLOC, LOAD, DATA
 1 .rodata   00000878 3ffe8540 3ffe8540 00000620 2**4
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .bss      00009130 3ffe8db8 3ffe8db8 00000e98 2**4
              ALLOC
 3 .text     00006f22 40100000 40100000 00000e98 2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .irom0.text 00028058 40240000 40240000 00007dc0 2**4
              CONTENTS, ALLOC, LOAD, READONLY, CODE
-----
Section info:
  Section|          Description| Start (hex) | End (hex) |Used space
-----
  data| Initialized Data (RAM)| 3FFE8000| 3FFE853C| 1340
  rodata| ReadOnly Data (RAM)| 3FFE8540| 3FFE8DB8| 2168
  bss| Uninitialized Data (RAM)| 3FFE8DB8| 3FFF1EE8| 37168
  text| Cached Code (IRAM)| 40100000| 40106F22| 28450
  irom0_text| Uncached Code (SPI)| 40240000| 40268058| 163928
Total Used RAM : 40676
Free RAM : 41244
Free IRam : 4336
-----
Run objcopy, please wait...
objcopy done
Run gen_appbin.exe
No boot needed.
Generate eagle.flash.bin and eagle.irom0text.bin successfully in folder firmware.
eagle.flash.bin----->0x00000
eagle.irom0text.bin---->0x40000
Done

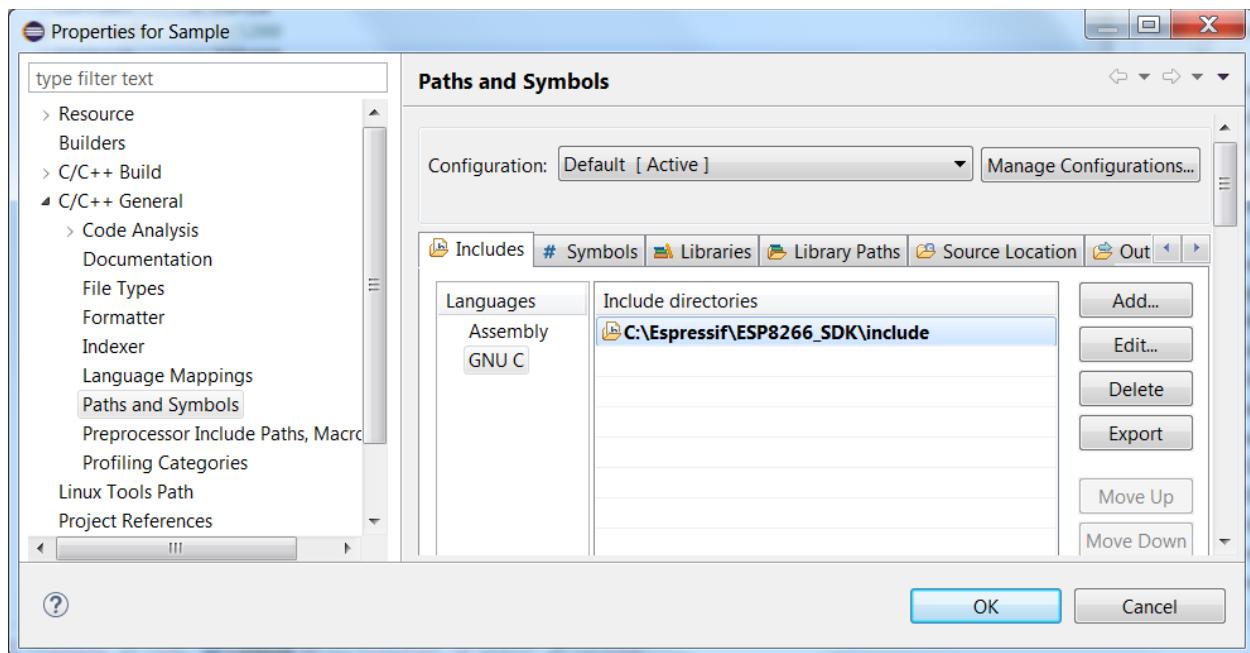
17:57:19 Build Finished (took 3s.141ms)
```

We can build solutions using the pre-supplied Makefiles but, personally, I don't like mystery so here is a recipe for building a solution from scratch.

1. Create a new project from File > New > C Project
2. Select a Makefile project



3. Add the ESP8266 include directory



4. Create the folders called "user" and "include"

```
Sample
  Includes
    include
    user
```

5. Create the file called "user_config.h" in include.

```
Sample
  Includes
  include
    user_config.h
  user
```

6. Create the C file called "user_main.c" in user.

```
Sample
  Includes
  include
    user_config.h
  user
    user_main.c
```

7. Create a Makefile

```
# Base directory for the compiler
XTENSA_TOOLS_ROOT ?= c:/Espressif/xtensa-lx106-elf/bin
SDK_BASE      ?= c:/Espressif/ESP8266_SDK
SDK_TOOLS     ?= c:/Espressif/utils
ESPPORT       = COM18
#ESPBAUD      = 115200
ESPBAUD        = 230400

# select which tools to use as compiler, librarian and linker
CC           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
AR           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-ar
LD           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
OBJCOPY      := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-objcopy
OBJDUMP      := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-objdump
ESPTOOL      ?= $(SDK_TOOLS)/esptool.exe

# compiler flags using during compilation of source files
TARGET        = myApp
CFLAGS        = -Os -g -O2 -std=gnu90 -Wpointer-arith -Wundef -Werror -Wl,-EL -fno-\
inline-functions -nostdlib -mlongcalls -mtext-section-literals -mno-serialize-volatile \
-D_ets_ -DICACHE_FLASH
MODULES       = user
BUILD_BASE    = build
FW_BASE       = firmware
SDK_LIBDIR   = lib
SDK_LDDIR    = ld

#
# Nothing to configure south of here.
#
# linker flags used to generate the main object file
LDFLAGS       = -nostdlib -Wl,--no-check-sections -u call_user_start -Wl,-static
# libraries used in this project, mainly provided by the SDK
LIBS          = c gcc hal phy pp net80211 lwip wpa main

#
# linker script used for the above linkier step
LD_SCRIPT     = eagle.app.v6.ld

flashimageoptions = --flash_freq 40m --flash_mode qio --flash_size 4m
SDK_LIBDIR    := $(addprefix $(SDK_BASE)/, $(SDK_LIBDIR))
LD_SCRIPT     := $(addprefix -T$(SDK_BASE)/$(SDK_LDDIR)/, $(LD_SCRIPT))
LIBS          := $(addprefix -l, $(LIBS))
APP_AR        := $(addprefix $(BUILD_BASE)/, $(TARGET)_app.a)
TARGET_OUT    := $(addprefix $(BUILD_BASE)/, $(TARGET).out)
BUILD_DIRS    = $(addprefix $(BUILD_BASE)/, $(MODULES)) $(FW_BASE)
SRC           = $(foreach moduleDir, $(MODULES), $(wildcard $(moduleDir)/*.c))
# Replace all x.c with x.o
OBJS          = $(patsubst %.c, $(BUILD_BASE)/%.o, $(SRC))

all: checkdirs $(TARGET_OUT)
    echo "Image file built!"
```

```

# Build the application archive.
# This is dependent on the compiled objects.
$(APP_AR): $(OBJS)
    $(AR) -cru $(APP_AR) $(OBJS)

# Build the objects from the C source files
$(BUILD_BASE)/%.o : %.c
    $(CC) $(CFLAGS) -I$(SDK_BASE)/include -Iinclude -c $< -o $@

# Check that the required directories are present
checkdirs: $(BUILD_DIRS)

# Create the directory structure which holds the builds (compiles)
$(BUILD_DIRS):
    mkdir --parents --verbose $@

$(TARGET_OUT): $(APP_AR)
    $(LD) -L$(SDK_LIBDIR) $(LD_SCRIPT) $(LDFLAGS) -Wl,--start-group $(LIBS) \
$(APP_AR) -Wl,--end-group -o $@
    $(OBJDUMP) --headers --section=.data \
        --section=.rodata \
        --section=.bss \
        --section=.text \
        --section=.irom0.text $@
    $(OBJCOPY) --only-section .text --output-target binary $@ eagle.app.v6.text.bin
    $(OBJCOPY) --only-section .data --output-target binary $@ eagle.app.v6.data.bin
    $(OBJCOPY) --only-section .rodata --output-target binary $@ \
eagle.app.v6.rodata.bin
    $(OBJCOPY) --only-section .irom0.text --output-target binary $@ \
eagle.app.v6.irom0text.bin
    $(SDK_TOOLS)/gen_appbin.exe $@ 0 0 0 0
    mv eagle.app.flash.bin $(FW_BASE)/eagle.flash.bin
    mv eagle.app.v6.irom0text.bin $(FW_BASE)/eagle.irom0text.bin
    rm eagle.app.v6.*

#
# Flash the ESP8266
#
flash: all
    $(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) write_flash $(flashimageoptions) \
0x00000 $(FW_BASE)/eagle.flash.bin 0x40000 $(FW_BASE)/eagle.irom0text.bin

#
# Clean any previous builds
#
clean:
# Remove forceably and recursively
    rm --recursive --force --verbose $(BUILD_BASE) $(FW_BASE)

flashId:
    $(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) flash_id

readMac:
    $(ESPTOOL) --port $(ESPPORT) --baud $(ESPBAUD) read_mac

```

```
imageInfo:  
$(ESPTOOL) image_info $(FW_BASE)/eagle.flash.bin
```

8. Add Make targets for at least all and flash

See also:

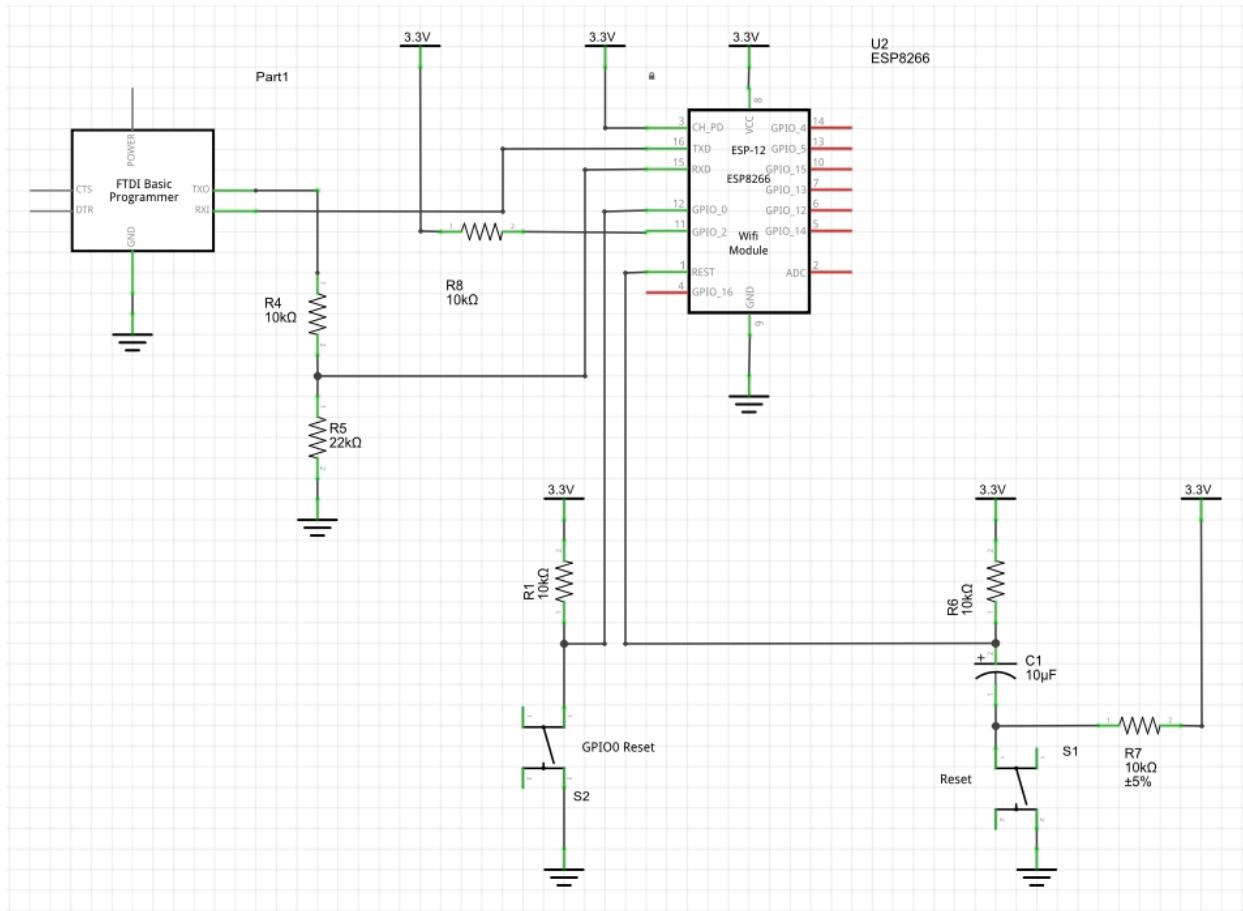
- Programming using Eclipse

Loading a program into the ESP8266

Once the program has been compiled, it needs to be loaded into the ESP8266. This task is called "flashing". In order to flash the ESP8266, it needs to be placed in a mode where it will accept the new incoming program to replace the old existing program. The way this is done is to reboot the ESP8266 either by removing and reapplying power or by bringing the REST pin low and then high again. However, just rebooting the device is not enough. During start-up, the device examines the signal value found on `GPIO0`. If the signal is low, then this is the indication that a flash programming session is about to happen. If the signal on `GPIO0` is high, it will enter its normal operation mode. Because of this, it is recommended not to let `GPIO0` float. We don't want it to accidentally enter flashing mode when not desired. A pull-up resistor of 10k is perfect.

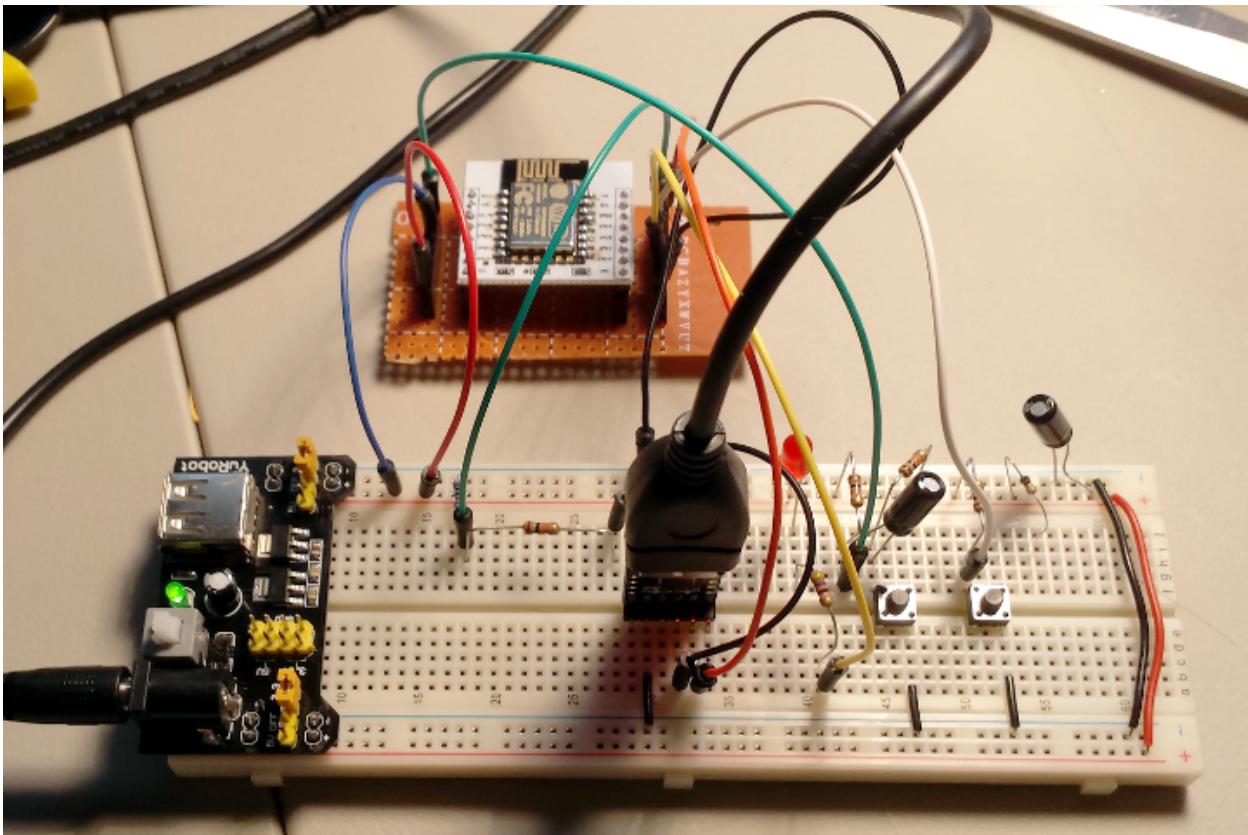
We can build a circuit which includes a couple of buttons. One for performing a reset and one for bringing `GPIO0` low. Pressing the reset button by itself will reboot the device. This alone is already useful. However if we are holding the "`GPIO0` low" button while we press reset, then we are placed in flash mode.

Here is an example schematic diagram illustrating an ESP-12 including the buttons:



Notice that there is a voltage divider from the output of the USB to UART converter TX pin. The thinking behind this is to handle the case where the output TX voltage is greater than the desired 3.3V wanted on the RX input of the ESP8266. Is this required? The belief is that it is **not** required if you are sure that the output TX voltage will be 3.3V. This appears to be the case for the CP2102 range of USB to UARTs however I am have no knowledge on other devices. What I can claim is that having a voltage divider that reduces 5V to 3.3V still results in a usable output level voltage to indicate a high signal when fed with a 3.3V actual output. I don't know how close I am coming to the minimum RX input voltage on the ESP8266 indicating a high.

When built out on a breadboard, it may look as follows:



This however suffers from the disadvantage that it requires us to manually press some buttons to load a new application. This is not a horrible situation but maybe we have alternatives?

When we are flashing our ESP8266s, we commonly connect them to USB->UART converters. These devices are able to supply UART used to program the ESP8266. We are familiar with the pins labeled RX and TX but what about the pins labeled RTS and DTR ... what might those do for us?

RTS which is "Ready to Send" is an output from the UART to inform the downstream device that it may now send data. This is commonly connected to the partner input CTS which is "Clear to Send" which indicates that it is now acceptable to send data. Both RTS and CTS are active low.

DTR which is "Data Terminal Ready" is used in flow control.

When flashing the device using the Eclipse tools and recipes the following are the flash commands that are run (as an example) and the messages logged:

```
22:34:17 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f C:/Users/User1/WorkSpace/k_blinky/Makefile flash
c:/Espressif/utils/esptool.exe -p COM11 -b 115200 write_flash -ff 40m -fm qio -fs 4m
0x000000 firmware/eagle.flash.bin 0x400000 firmware/eagle.irom0text.bin
```

```

Connecting...
Erasing flash...
head: 8 ;total: 8
erase size : 16384

Writing at 0x00000000... (3 %)
Writing at 0x00000400... (6 %)
...
Writing at 0x00007000... (96 %)
Writing at 0x00007400... (100 %)
Written 30720 bytes in 3.01 seconds (81.62 kbit/s)...
Erasing flash...
head: 16 ;total: 41
erase size : 102400

Writing at 0x00040000... (0 %)
Writing at 0x00040400... (1 %)
...
Writing at 0x00067c00... (99 %)
Writing at 0x00068000... (100 %)
Written 164864 bytes in 16.18 seconds (81.53 kbit/s)...

Leaving...

```

22:34:40 Build Finished (took 23s.424ms)

As an example of what the messages look like if we **fail** to put the ESP8266 into flash mode, we have the following:

```

13:47:09 **** Build of configuration Default for project k_blinky ****
mingw32-make.exe -f C:/Users/User1/WorkSpace/k_blinky/Makefile flash
c:/Espressif/utils/esptool.exe -p COM11 -b 115200 write_flash -ff 40m -fm qio -fs 4m
0x000000 firmware/eagle.flash.bin 0x40000 firmware/eagle.irom0text.bin
Connecting...
Traceback (most recent call last):
  File "esptool.py", line 558, in <module>
    File "esptool.py", line 160, in connect
Exception: Failed to connect
C:/Users/User1/WorkSpace/k_blinky/Makefile:313: recipe for target 'flash' failed
mingw32-make.exe: *** [flash] Error 255

13:47:14 Build Finished (took 5s.329ms)

```

The tool called `esptool.py` provides an excellent environment for flashing the device but it can also be used for "reading" what is currently stored upon it. This can be used for making backups of the applications contained within before re-flashing them with a new program. This way, you can always return to what you had before over-writing. For example, on Unix:

```
esptool.py --port /dev/ttyUSB0 read_flash 0x00000 0xFFFF backup-0x00000.bin  
esptool.py --port /dev/ttyUSB0 read_flash 0x10000 0x3FFFF backup-0x10000.bin
```

See also:

- USB to UART converters
- Recommended setup for programming ESP8266
- Working with memory
- [What is a UART?](#)
- esptool.py
- esptool-ck

Programming environments

We can program the ESP8266 using the Espressif supplied SDK on Windows using Eclipse. A separate chapter on setting up that environment is supplied. We also have the ability to program the ESP8266 using the Arduino IDE. This is potentially a game changing story and it too been given its own important chapter.

See also:

- Programming using Eclipse
- Programming using the Arduino IDE

Compilation tools

There are a number of tools that are essential when building C based ESP8266 applications.

ar

The archive tool is used to package together compiled object files into libraries. These libraries end with ".a" (archive). A library can be named when using a linker and the objects contained within will be used to resolve externals.

Some of the most common flags used with this tool include:

- **-c** – Create a library
- **-r** – Replace existing members in the library
- **-u** – Update existing members in the library

The syntax of the command is:

```
ar -cru libraryName member.o member.o ... .
```

See also:

- GNU – [ar](#)
- [nm](#)

esptool.py

This tool is an open source implementation used to flash the ESP8266 through a serial port. It is written in Python. Versions have been seen to be available as windows executables that appear to have been generated ".EXE" files from the Python code suitable for running on Windows without a supporting Python runtime installation.

- **-p port | --port port** – The serial port to use
- **-b baud | --baud baud** – The baud rate to use for serial
- **-h** – Help
- **{command} -h** – Help for that command
- **load_ram {filename}** – Download an image to RAM and execute
- **dump_mem {address} {size} {filename}** – Dump arbitrary memory to disk
- **read_mem {address}** – Read arbitrary memory location
- **write_mem {address} {value} {mask}** – Read-modify-write to arbitrary memory location
- **write_flash** – Write a binary blob to flash
 - **--flash_freq {40m, 26m, 20m, 80m} | -ff {40m, 26m, 20m, 80m}** – SPI Flash frequency
 - **--flash_mode {qio, qout, dio, dout} | -fm {qio, qout, dio, dout}** – SPI Flash mode
 - **--flash_size {4m, 2m, 8m, 16m, 32m, 16m-c1, 32m-c1, 32m-c2} | -fs {4m, 2m, 8m, 16m, 32m, 16m-c1, 32m-c1, 32m-c2}** – SPI Flash size in Mbit
 - **{address} {fileName}** – Address to write, file to write ... repeatable
- **run** – Run application code in flash
- **image_info {image file}** – Dump headers from an application image. Here is an example output:

```
Entry point: 40100004
3 segments
```

```
Segment 1: 25356 bytes at 40100000
Segment 2: 1344 bytes at 3ffe8000
Segment 3: 924 bytes at 3ffe8540
```

```
Checksum: 40 (valid)
```

- **make_image** – Create an application image from binary files
 - **--segfile SEGFILE, -f SEGFILE** – Segment input file

- --segaddr SEGADDR, -a SEGADDR – Segment base address
 - --entrypoint ENTRYPOINT, -e ENTRYPOINT – Address of entry point
 - output
- elf2image – Create an application image from ELF file
 - --output OUTPUT, -o OUTPUT – Output filename prefix
 - --flash_freq {40m,26m,20m,80m}, -ff {40m,26m,20m,80m} – SPI Flash frequency
 - --flash_mode {qio,qout,dio,dout}, -fm {qio,qout,dio,dout} – SPI Flash mode
 - --flash_size {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2}, -fs {4m,2m,8m,16m,32m,16m-c1,32m-c1,32m-c2} – SPI Flash size in Mbit
 - --entry-symbol ENTRY_SYMBOL, -es ENTRY_SYMBOL – Entry point symbol name (default 'call_user_start')
- read_mac – Read MAC address from OTP ROM. Here is an example output:

```
MAC AP: 1A-FE-34-F9-43-22
MAC STA: 18-FE-34-F9-43-22
```

- flash_id – Read SPI flash manufacturer and device ID. Here is an example output:


```
head: 0 ;total: 0
erase size : 0
Manufacturer: c8
Device: 4014
```
- read_flash – Read SPI flash content
 - address – Start address
 - size – Size of region to dump
 - filename – Name of binary dump
- erase_flash – Perform Chip Erase on SPI flash. This is an especially useful command if one ends up someone bricking the device as it should reset the device to its defaults.

See also:

- esptool-ck
- nodemcu-flasher
- Loading a program into the ESP8266
- Working with memory
- Github: [themadinventor/esptool](https://github.com/themadinventor/esptool)

esptool-ck

Another tool that is also called `esptool-ck`. The naming of these tools being so similar is starting to become uncomfortable.

- `-eo <filename>` – Open an ELF object.
- `-es <section> <filename>` – Read the named section from the object and writes to the named file.
- `-ec` – Closes the ELF file.
- `-bo <filename>` – Prepares a firmware file for the ESP.
- `-bm <qio|qout|dio|dout>` – Set the flash chip interface mode.
- `-bz <512K|256K|1M|2M|4M|8M|16M|32M>` – Set the flash chip size.
- `-bf <40|26|20|80>` – Set the flash chip frequency.
- `-bs <section>` – Read the ELF section and write to the firmware image.
- `-bc` – Close the firmware image.
- `-v` – Increase the verbosity of output (`-v, -vv, -vvv`)
- `-q` – Disable most of the output
- `-cp <device>` – Serial device (eg. `COM1`)
- `-cd <board>` – Select the reset method for resetting the board.
 - `none`
 - `ck`
 - `wifio`
 - `nodemcu`
- `-cb <baudrate>` – Select the baud rate to use.
- `-ca <address>` – Address of flash memory as the target of the upload.
- `-cf <filename>` – Upload the named file to flash.

Here, for example, is a command to flash a NodeMCU devKit board:

```
esptool -cp COM15 -cd nodemcu -cb 115200 -ca 0x00000 -cf myApp_0x00000.bin
```

Here is a clean log of an Arduino IDE upload:

```
esptool v0.4.5 - (c) 2014 Ch. Klippel <ck@atelier-klippel.de>
    setting board to ck
    setting baudrate from 115200 to 115200
    setting port from COM1 to COM11
```



```
setting serial port timeouts to 1 ms
setting serial port timeouts to 1000 ms
flush complete
```

Binaries corresponding to releases of the tool can be found under the releases section:

<https://github.com/igrr/esptool-ck/releases>

See also:

- esptool.py
- nodemcu-flasher
- Github: <https://github.com/igrr/esptool-ck>

gcc

The open source GNU Compiler Collection includes compilers for C and C++. If we look carefully at the flags that are supplied for compiling and linking code for the ESP8266 we find the following:

Compiling

- `-c` – Compile the code to a `.o` object file.
- `-Os` – Optimize code generation for size.
- `-O2` – Optimize for performance which code result in larger code size. For example, instead of making a function call, code could be in-lined.
- `-ggdb` – Generate debug code that can be used by the `gdb` debugger..
- `-std=gnu90` – Dialect of C supported.
- `-Werror` – Make all warnings errors.
- `-Wno-address` – Do not warn about suspicious use of memory addresses.
- `-Wpointer-arith` – Warn when pointer arithmetic is attempted that depends on `sizeof`.
- `-Wundef` – Warn when an identifier is found in a `#if` directive that is not a macro.
- `-fno-inline-functions` – Do not allow functions to be replaced with in-line code.
- `-mlongcalls` – Translate direct assembly language calls into indirect calls.
- `-mtext-section-literals` – Allow literals to be intermixed with the text section.
- `-mno-serialize-volatile` – Special instructions for volatile definitions.

Linking:

- `-nostdlib` – Don't use standard C or C++ system startup libraries

See also:

- [GCC – The GNU Compiler Collection](#)

gen_appbin.py

The syntax of this tool is:

```
gen_appbin.py app.out boot_mode flash_mode flash_clk_div flash_size
```

- **flash_mode**
 - 0 – QIO
 - 1 – QOUT
 - 2 – DIO
 - 3 – DOUT
- **flash_clk_div**
 - 0 – 80m / 2
 - 1 – 80m / 3
 - 2 – 80m / 4
 - 0xf – 80m / 1
- **flash_size_map**
 - 0 – 512 KB (256 KB + 256 KB)
 - 1 – 256 KB
 - 2 – 1024 KB (512 KB + 512 KB)
 - 3 – 2048 KB (512 KB + 512 KB)
 - 4 – 4096 KB (512 KB + 512 KB)
 - 5 – 2048 KB (1024 KB + 1024 KB)
 - 6 – 4096 KB (1024 KB + 1024 KB)

The following files are expected to exist:

- eagle.app.v6.irom0text.bin
- eagle.app.v6.text.bin
- eagle.app.v6.data.bin
- eagle.app.v6.rodata.bin

The output of this command is a new file called `eagle.app.flash.bin`.

make

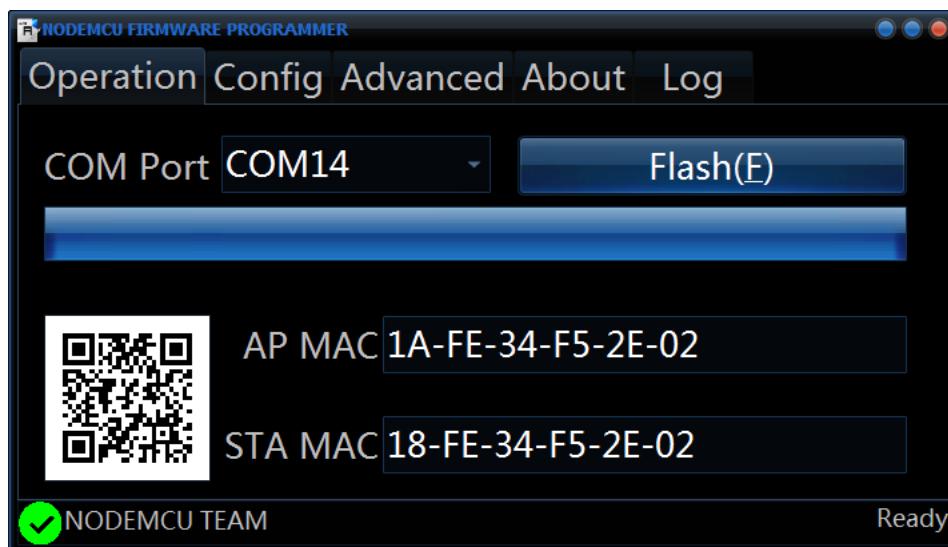
Make is a compilation engine used to track what has to be compiled in order to build your target application. Make is driven by a Makefile. Although powerful and simple enough for simple C projects, it can get complex pretty quickly. If you find yourself studying Makefiles written by others, grab the excellent GNU make documentation and study it deeply.

nodemcu-flasher

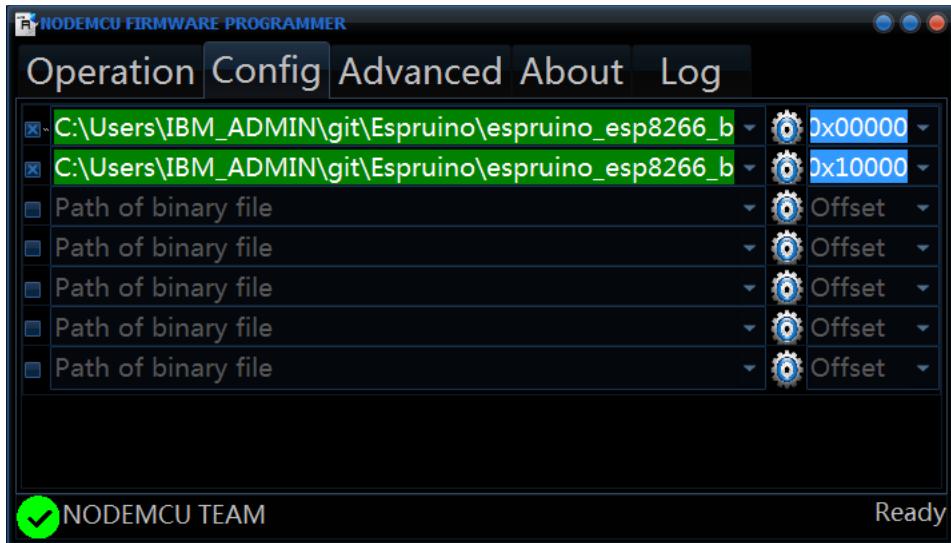
This tool is another instance of an ESP8266 flasher. Unlike some of the other tools available, this one is GUI based. From within the tool one can select all the options that one might expect including one or more files to flash, the serial connection and information and more.

Once entered, one can click the "Flash" button and flashing begins with an attractive progress bar.

The following is what the tool looks like after completing a flash:



Here is what it looks like within its flash file selection tab:



And finally, here are the communication settings:



Although visually attractive, it seems to have a big drawback. It feels much slower to flash than some of the other tools. This, of course, assumes that one attempts to flash at the same baud rate.

However, even with this slight weakness, it is still one of the easiest to use flasher tools available and appears to be perfect for the casual flasher. If I were to recommend a tool to be used by someone who only needed to install an app on their ESP8266 infrequently, this would probably be it.

See also:

- esptool.py
- esptool-ck
- GitHub: [nodemcu/nodemcu-flasher](#)
- YouTube: [ESP8266 How To Flash NodeMcu Firmware](#)
- [Flashing the NodeMCU firmware on the ESP8266 \(Windows\) - Guide](#)

nm

List symbols from object files.

Useful flags:

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

See also:

- ar
- GNU – [nm](#)

objcopy

See also:

- GNU – [objcopy](#)

objdump

The command is `xtensa-lx106-elf-objdump` located in

C:\Espressif\xtensa-lx106-elf\bin.

Some of the more important flags are:

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

See also:

- Wikipedia – [objdump](#)
- GNU – [objdump](#)
- man page – [objdump\(1\)](#)

xxd

This is a deceptively simple but useful tool. What it does is dump binary data contained within a file in a formatted form. One powerful use of it is to take a binary file and produce a C language data structure that represents the content of the file. This means that you can take binary data and include it in your applications. A copy of `xxd.exe` is distributed with the SDK supplied by Espressif in the `tools` folder.

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

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

ESP8266 Linking

When the C and C++ source files that constitute your project have been compiled to their object files, it is time to link them with libraries to finalize the executable to be deployed. Here is an example of a linking command used to build an executable.

```
xtensa-lx106-elf-gcc
  -g
  -Os
  -nostdlib
  -Wl,--no-check-sections -u call_user_start
  -Wl,-static
    "-L??/tools/sdk//lib"
    "-L??tools/sdk//ld"
    "-Teagle.flash.512k.ld"
  -Wl,-wrap,system_restart_local
  -Wl,-wrap,register_chipv6_phy
  -o "Release/Test_ESP_RESTClient.elf"
  -Wl,--start-group
  x.o
  y.o
  z.o
  -lm
  -lgcc
  -lhal
  -lphy
  -lnet80211
  -llwip
  -lwpa
  -lmain
  -lpp
  -lsmartconfig
  -lwps
  -lcrypto
-Wl,--end-group
"-LRelease"
```

Notice that some libraries are used when linking. Many of these libraries are supplied with the Espressif SDK.

Library name	Description
at	
crypto	
espnow	
hal	
json	
lwip	
lwip_536	
main	
net80211	
phy	
pp	
pwm	
smartconfig	
ssc	
ssl	
upgrade	
wpa	
wps	

See also:

- Working with memory
- The GNU Linker
- An Introduction to the GNU – Compiler and Linker

Flashing over the air - FOTA

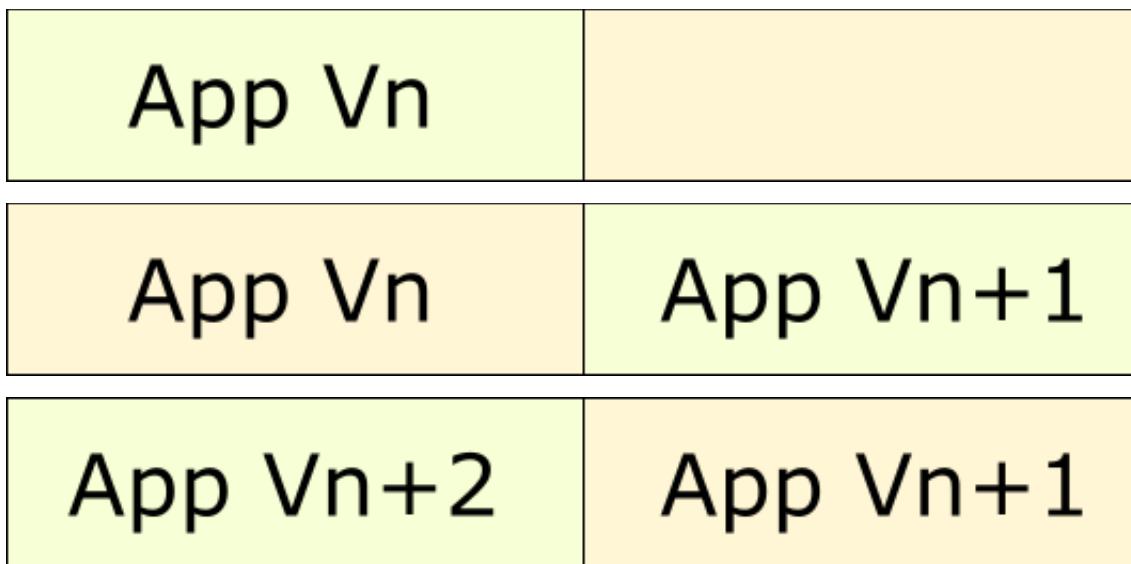
Imagine that you have built a fantastic ESP8266 application that is contained within an embedded device. You ship it to your customers and all is great. Suddenly, you start to get reports that it is periodically failing. You diagnose the error and to your horror, you find that there was coding mistake ... but thankfully, it is easily and quickly fixed. You now find that you have a problem. Your embedded device doesn't have a UART exposed and, even if it did, your customers would be up in arms if they had to plug it into a computer. Worse, it would become a support nightmare to walk consumers through the mechanics of achieving a reload when we made no assumptions about the technical skills of the end consumer.

To resolve this issue, we introduce the concept of flashing (or more specifically re-flashing) an application over the internet through a WiFi connection. This notion is called "Flashing Over The Air" or FOTA.

Generically, it works as follows.

Your ESP8266 has an amount of flash memory available to it. Let us divide that memory into two equal halves.

When an ESP8266 ships, your application will be loaded into the 1st half of flash and will ignore the second half. From time to time, it will "call home" via the Internet and ask if there is a replacement set of firmware (a new version). If there is, then it will download that new firmware into the 2nd half of flash. If that fully succeeds, the device will reboot and start running the new firmware from the 2nd half of flash ... it will now ignore the 1st half. A subsequent replacement of the firmware with yet another version will be loaded into the 1st half and the story repeats.



Effectively, we are thus able to flip-flop between two versions. With this high level theory under our belts, let us now dig a little deeper. Obviously, if we follow this story, we see that we have effectively reduced the amount of flash available to host our programs by half. That doesn't sound good ... why do we do this? The answer is actually quite simple. If we consider the reality that WiFi connections can fail, Internet access can be lost and power can simply be removed from a device, we can end up in the situation where an upload of new code into flash actually fails before it completes. This would leave us with broken code in the flash area. If we tried to "in-place" replace our existing application and such a failure occurred, we would have "bricked" the device. It is likely that a replaced application that was only half loaded wouldn't even boot. To circumvent that issue, the ESP8266 contains a flag which defines which of the two possible halves of firmware is currently the one used to execute the program. Only when a new version of the firmware has been validated as having been successfully loaded is the flag switched to the other half. If an error occurs during the replacement upload, then the flag is not switched and no harm will have been done since we had effectively ignored the second half of flash in the first place.

Let us now go even deeper. Espressif provides code called "boot" which is responsible for booting an ESP8266. When an ESP8266 is powered on, it is this boot code that gets control. It is boot which determines how the remainder of the power-on of the device will proceed. When we flash an ESP8266, we should provide both the boot application and our own application logic. From an address space perspective, the boot program is loaded into flash address `0x00000` for 4KBytes. Our application will be loaded from address `0x01000` onwards.

Since the ESP8266 can have a variety of flash sizes, we examine each of these in turn.

512KB flash

Content	Address	Size
Boot	<code>0x0 0000 – 0x0 0FFF</code>	4KB
App 1 (user1.bin)	<code>0x0 1000 – 0x3 BFFF</code>	236KB
User params	<code>0x3 C000 – 0x3 FFFF</code>	16KB
Do not use	<code>0x4 0000 – 0x4 0FFF</code>	4KB
App 2 (user2.bin)	<code>0x4 1000 – 0x7 BFFF</code>	236KB
System params	<code>0x7 C000 – 0x7 FFFF</code>	16KB

1024KB Flash

Content	Address	Size
Boot	<code>0x0 0000 – 0x0 0FFF</code>	4KB
App 1 (user1.bin)	<code>0x0 1000 – 0x7 BFFF</code>	492KB
User params	<code>0x7 C000 – 0x7 FFFF</code>	16KB
Do not use	<code>0x8 0000 – 0x8 0FFF</code>	4KB
App 2 (user2.bin)	<code>0x8 1000 – 0xF BFFF</code>	492KB
System params	<code>0xF C000 – 0xF FFFF</code>	16KB

2048KB Flash – Option 1

Content	Address	Size
Boot	0x00 0000 – 0x00 0FFF	4KB
App 1 (user1.bin)	0x00 1000 – 0x07 BFFF	492KB
User params	0x07 C000 – 0x07 FFFF	16KB
Do not use	0x08 0000 – 0x08 0FFF	4KB
App 2 (user2.bin)	0x08 1000 – 0x0F BFFF	492KB
User Data	0x0F C000 – 0x1F BFFF	1008KB
System params	0x1F C000 – 0x1F FFFF	16KB

2048KB Flash – Option 2

Content	Address	Size
Boot	0x00 0000 – 0x00 0FFF	4KB
App 1 (user1.bin)	0x00 1000 – 0x07 BFFF	1004KB
User params	0x0F C000 – 0x0F FFFF	16KB
Do not use	0x10 0000 – 0x10 0FFF	4KB
App 2 (user2.bin)	0x10 1000 – 0x1F BFFF	1004KB
System params	0x1F C000 – 0x1F FFFF	16KB

4096KB Flash – Option 1

Content	Address	Size
Boot	0x00 0000 – 0x00 0FFF	4KB
App 1 (user1.bin)	0x00 1000 – 0x07 BFFF	492KB
User params	0x07 C000 – 0x07 FFFF	16KB
Do not use	0x08 0000 – 0x08 0FFF	4KB
App 2 (user2.bin)	0x08 1000 – 0x0F BFFF	492KB
User Data	0x0F C000 – 0x3F BFFF	3072KB
System params	0x3F C000 – 0x3F FFFF	16KB

4096KB Flash – Option 2

Content	Address	Size
Boot	0x00 0000 – 0x00 0FFF	4KB
App 1 (user1.bin)	0x00 1000 – 0x07 BFFF	1004KB
User params	0x0F C000 – 0x0F FFFF	16KB
Do not use	0x10 0000 – 0x10 0FFF	4KB
App 2 (user2.bin)	0x10 1000 – 0x1F BFFF	1004KB
User Data	0x1f C000 – 0x3F BFFF	2048KB
System params	0x3F C000 – 0x3F FFFF	16KB

See also:

- Espressif document: 99C – ESP8266 – OTA Upgrade

Debugging

When writing programs, we may find that they don't always run as expected.

Performing debugging on an SOC can be difficult since we have no readily available source level debuggers.

ESP-IDF logging

The ESP-IDF framework provides a logging set of features. These logging items can then be inserted in your own application for diagnosing problems or capturing traces.

To use the logging functions, we must include `"esp_log.h"`.

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

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

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

By default, when logging is requested, the output is sent to the primary serial stream. However, we can over-ride that destination by using the function called

`esp_log_set_vprintf()`. This takes as a parameter a reference to a C function that has the same syntax as `vprintf`. Specifically:

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

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

- `ESP_LOG_NONE`

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

The logged output is of the format:

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

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

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

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

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

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

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

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

The signature of this function is:

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

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

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

- Error: Reference source not found

Logging to UART1

We can insert diagnostic statements using `os_printf()`. This causes the text and data associated with these functions to be written to UART1 (GPIO 2). If we attach a USB → UART device in the circuit, we can then look at the data logged. In my development environment I always have two USB → UART devices in play. One to flash new applications and one to use for diagnostic output.

The OS is also able to write debugging information. By default this is on but can be switched off with a call to `system_set_os_print()`.

See also:

- USB to UART converters
- Working with serial
- `system_set_os_print`

Run a Blinky

Physically looking at an ESP8266 there isn't much to see that tells you all is working well within it. There is a power light and a network transmission active light ... but that's about it. A technique that I recommend is to always have your device perform execute a "blinking led" which is commonly known as a "Blinky". This can be achieved by connecting a GPIO pin to a current limiting resistor and then to an LED. When the GPIO signal goes high, the LED lights. When the GPIO signal goes low, the LED becomes dark. If we define a timer callback that is called (for example) once a second and toggles the GPIO pin signal value each invocation, we will have a simple blinking LED. You will be surprised how good a feeling it will give simply knowing that *something* is alive within the device each time you see it blink.

The cost of running the timer and changing the I/O value to achieve a blinking should not be a problem during development time so I wouldn't worry about side effects of doing this. Obviously for a published application, you may not desire this and can simply remove it.

However, although this is a trivial circuit, it has a lot of uses during development. First, you will always know that the device is operating. If the LED is blinking, you know the device has power and logic processing control. If the light stops blinking, you will know that something has locked up or you have entered an infinite loop.

Another useful purpose for including the Blinky is to validate that you have entered flash mode when programming the device. If we understand that the device can boot up in

normal or flash mode and we boot it up in flash mode, then the Blinky will stop executing. This can be useful if you are using buttons or jumpers to toggle the boot mode as it will provide evidence that you are *not* in normal mode. On occasion I have mis-pressed some control buttons and was quickly able to realize that something was wrong before even attempting to flash it as the Blinky was still going.

Here is some simple code for setting up a Blinky. In this example we use GPIO4 as the LED driver. First, the code we place in `user_init`:

```
PIN_FUNC_SELECT(PERIPH_IO_MUX_GPIO4_U, FUNC_GPIO4);
os_timer_disarm(&blink_timer);
os_timer_setfn(&blink_timer, (os_timer_func_t *)blink_cb, (void *)0);
os_timer_arm(&blink_timer, 1000, 1);
```

This assumes a global called `blink_timer` defined as:

```
LOCAL os_timer_t blink_timer;
```

The callback function in this example is called `blink_cb` and looks like:

```
LOCAL void ICACHE_FLASH_ATTR blink_cb(void *arg)
{
    led_state = !led_state;
    GPIO_OUTPUT_SET(4, led_state);
}
```

The global variable called `led_state` contains the current state of the LED (1=on, 0=off):

```
LOCAL uint8_t led_state=0;
```

Dumping IP Addresses

Being a WiFi and TCP/IP device, you would imagine that the ESP8266 works a lot with IP addresses and you would be right. We can generate a string representation of an IP address using:

```
os_printf(IPSTR, IP2STR(pIpAddrVar))
```

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

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

which may be more useful in certain situations.

See also:

- `ipaddr_t`

Exception handling

At run-time, things may not always work as expected and an exception can be thrown. For example, you might attempt to access storage at an invalid location or write to read only memory or perform a divide by zero.

ESP8266

When such an occurrence happens, the device will reboot itself but not before writing some diagnostics to UART1. Diagnostics may look like:

```
Fatal exception (28):  
epc1=0x40243182, epc2=0x00000000, epc3=0x00000000, excvaddr=0x00000050,  
depc=0x00000000
```

The codes are as follows:

- `exccause` – Code describing the cause
- `epc1` – Exception program counter
- `excvaddr` – Virtual address that caused the most recent fetch, load or store exception. For example, if a write to memory occurs and that memory is not RAM an exception will be thrown and the value here will be the address that was attempted to be written.

The primary exception codes are:

Code	Code	Cause name
0	0x00	IllegalInstructionCause
1	0x01	SyscallCause
2	0x02	InstructionFetchErrorCause
3	0x03	LoadStoreErrorCause
4	0x04	Level1InterruptCause
5	0x05	AllocaCause
6	0x06	IntegerDivideByZeroCause
7	0x07	Reserved
8	0x08	PrivilegedCause
9	0x09	LoadStoreAlignmentCause
10	0x0a	Reserved
11	0x0b	Reserved
12	0x0c	InstrPIFDataErrorCause
13	0x0d	LoadStorePIFDataErrorCause
14	0x0e	InstrPIFAddrErrorCause
15	0x0f	LoadStorePIFAddrErrorCause
16	0x10	InstTLBMissCause
17	0x11	InstTLBMultiHitCause
18	0x12	InstFetchPrivilegeCause
19	0x13	Reserved
20	0x14	InstFetchProhibitedCause

21	0x15	Reserved
22	0x16	Reserved
23	0x17	Reserved
24	0x18	LoadStoreTLBMissCause
25	0x19	LoadStoreTLBMultiHitCause
26	0x1a	LoadStorePrivilegeCause
27	0x1b	Reserved
28	0x1c	LoadProhibitedCause
29	0x1d	StoreProhibitedCause
30	0x1e	Reserved
31	0x1f	Reserved
32-39	0x20-0x27	CoprocessorDisabled
40-63	0x28-0x3f	Reserved

If we know the location of the exception, we can analyze the executable (`app.out`) to figure out what piece of code caused the problem. For example:

```
xtensa-lx106-elf-objdump -x app.out -d
```

Using a debugger (GDB)

GDB is the GNU Debugger and is an excellent tool for debugging compiled C source code. However, it is primarily designed to debug OS hosted applications such as those compiled for Windows or Linux and didn't have much applicability to the ESP8266. This was until Espressif released their GDB stub.

The version of the debugger must be the `xtensa-lx106-elf-gdb` tool.

To prepare for using the tool, one must compile with the following additional flags:

- `-ggdb`
- `-Og`

In addition, we must initialize GDB with a call to `gdbstub_init()` somewhere early in the start up code in our application. Finally, we link in the `gdbstub` library.

See also:

- Github: [espressif/esp-gdbstub](https://github.com/espressif/esp-gdbstub)

Debugging and testing TCP and UDP connections

When working with TCP/IP, you will likely want to have some applications that you can use to send and receive data so that you can be sure the ESP8266 is working. There

are a number of excellent tools and utilities available and these vary by platform and function.

Android - Socket Protocol

The Socket Protocol is a free Android app available from the Google play app store.

See:

- <https://play.google.com/store/apps/details?id=aprisco.app.android>

Android - UDP Sender/Receiver

The UDP Sender/Receiver is another free Android app available from the Google play app store. What makes this one interesting is its ability to be a UDP (as opposed to TCP) sender and receiver. See:

- <https://play.google.com/store/apps/details?id=com.jca.udpsendreceive>

Windows - Hercules

Hercules is an older app for Windows that still seems to work just fine on the latest releases. It looks a little old in the tooth now but still seems to get the job done just fine.

See:

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

Curl

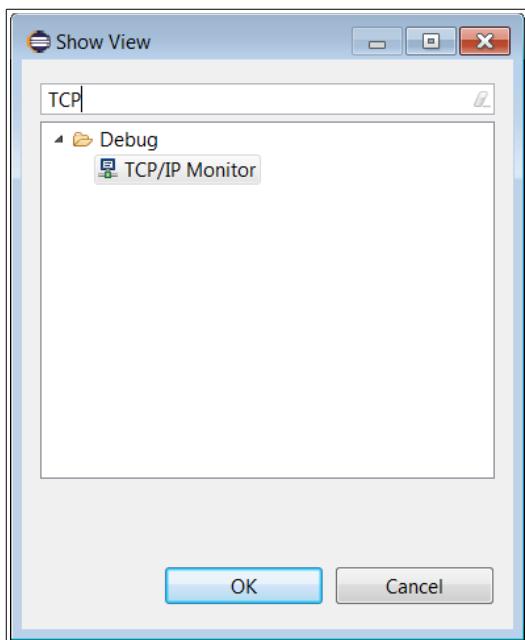
Curl is powerful and comprehensive command line tool for performing any and all URL related commands. It can transmit HTTP requests of all different formats and receive their responses. It has a bewildering set of parameters available to it which is both a blessing and curse. You can be pretty sure that if it can be done, Curl can do it ... however be prepared to wade through a lot of documentation.

See also:

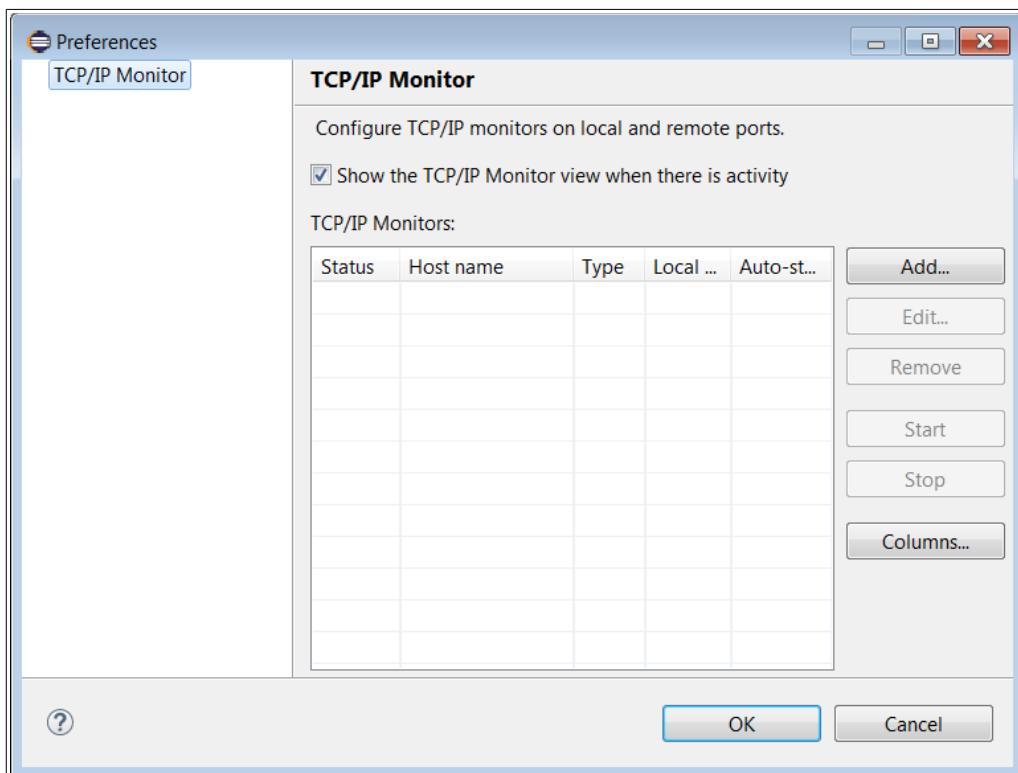
- Curl

Eclipse - TCP/MON

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

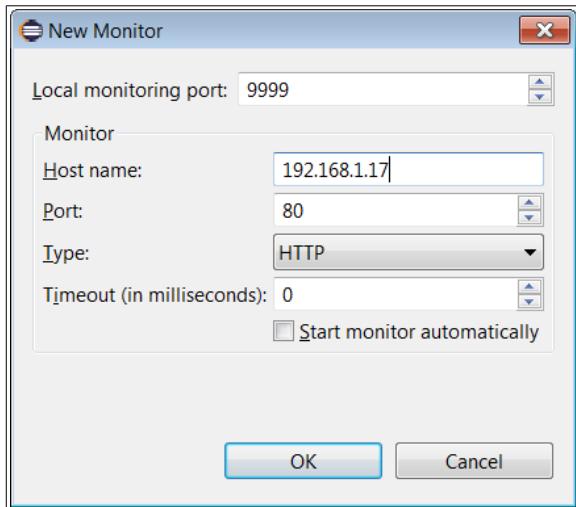


If you can't find it in the view finder, the chances are high that you haven't installed "Eclipse Web Developer Tools". Once launched, open its properties pane:



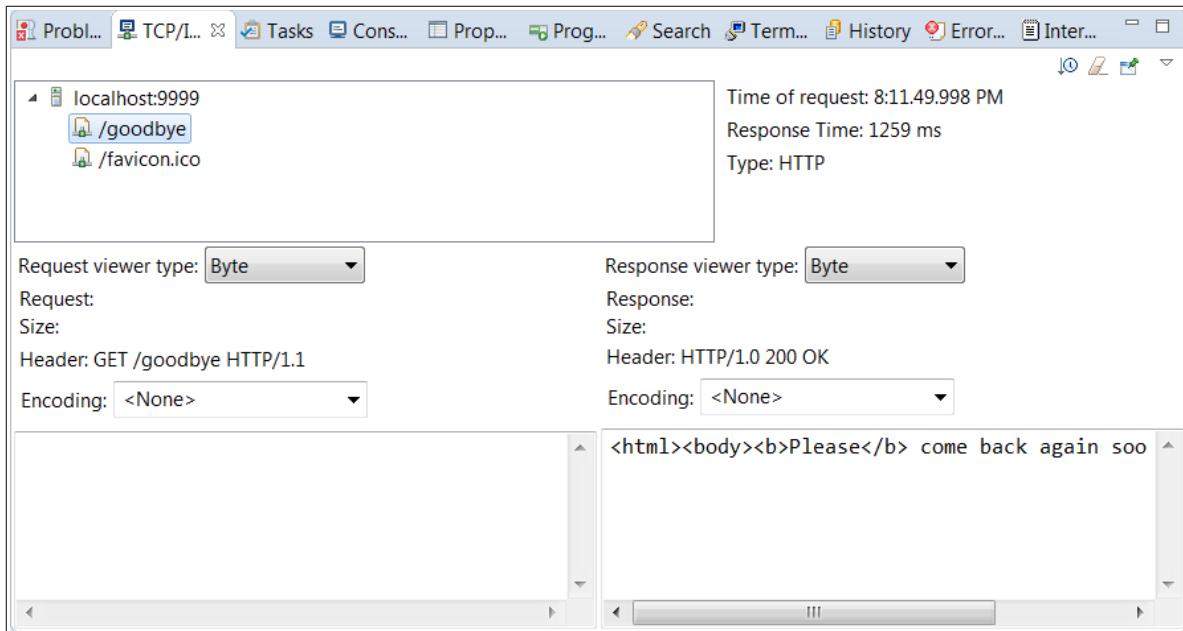
From there you can add local listeners. These will be TCP/IP listeners that listen on a local port where Eclipse is running. During configuration, you specify another IP address and port number. When TCP traffic now arrives at the listener on which TCP/IP

Monitor is watching, it will forward that traffic to the partner while at the same time logging it to the TCP/IP Monitor screen.



For example, here TCP/IP Monitor is listening on 192.168.1.2 (localhost) which is where Eclipse is running. It is listening on port 9999. When TCP/IP traffic arrives at that address, it will be sent onwards to 192.168.1.17 (which happens to be my ESP8266 device) to port 80.

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



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

httpbin.org

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

ESP8266 Architecture

To start thinking about writing applications for the ESP8266, we need to understand the high level architecture of the device.

Custom programs

Custom programs are applications that you can write and are the core focus of this book. These programs can be written in C or C++ and then compiled into the binary files. The programs are expected to have "well known" functions defined within that serve as architected entry points and callbacks.

Programmers write a C language file with a suggested name of "`user_main.c`". Contained within is a function with the signature:

```
void user_init(void)
```

This provides the initial entry into application code. It is called once during startup. While executing within this function, realize that not all of the environment is yet operational. If you need a fully functioning environment, register a callback function that will be invoked when the environment is 100% ready. This callback function can be registered with a call to `system_init_done_cb`.

RF initialization must also be provided via:

```
void user_rf_pre_init(void)
```

When running in user code, we need to be sensitive that the primary purpose of the device is network communications. Since these are handled in the software, when user code gets control, that simply means that networking code doesn't. Since we only have one thread of control, we can't be in two places at once. The recommended duration to spend in user code at a single sitting is less than 10msecs.

See also:

- `system_init_done_cb`

WiFi at startup

The ESP8266 stores WiFi start-up information in flash memory. This allows it to perform its functions at start-up without having to ask the user for any special or additional information. In my opinion, this is more trouble than it is worth. If I am going to write an ESP8266 application, I want to control when, how and to what it will connect

or be an access point. Thankfully, there is a function called `wifi_station_set_auto_connect()` and its partner called `wifi_station_get_auto_connect()`. These allow us to override the auto connection functions when we are a station.

Working with WiFi - ESP8266

The ESP8266 can either be a station in the network, an access point for other devices or both. This is a fundamental consideration and we will want to choose how the device behaves early on in our design. Once we have chosen what we want, we set a global mode property which indicates which of the operational modes our device will perform (station, access point or station AND access point).

Scanning for access points

If the ESP8266 is performing the role of a station we will need to connect to an access point. We can request a list of the available access points against which we can attempt to connect. We do this using the `wifi_station_scan()` function. This function takes a callback function pointer as one of its parameters. This callback will be invoked when the scan has completed. The callback is necessary because it can take some time (a few seconds) for the scan to be performed and we can't afford to block operation of the system as a whole until complete. The scan callback function receives a linked list of BSS structures. Contained within a BSS structure are:

- The SSID for the network
- The BSSID for the access point
- The channel
- The signal strength
- ... others

For example:

```
void scanCB(void *arg, STATUS status) {
    struct bss_info *bssInfo;
    bssInfo = (struct bss_info *)arg;
    // skip the first in the chain ... it is invalid
    bssInfo = STAILQ_NEXT(bssInfo, next);
    while(bssInfo != NULL) {
        os_printf("ssid: %s\n", bssInfo->ssid);
        bssInfo = STAILQ_NEXT(bssInfo, next);
    }
}

//...
```

```

{
    // Ensure we are in station mode
    wifi_set_opmode_current(STATION_MODE);

    // Request a scan of the network calling "scanCB" on completion
    wifi_station_scan(NULL, scanCB);
}

```

Note the use of the `STAILQ_NEXT()` macro to navigate to the next entry in the list. The end of the list is indicated when this returns `NULL`.

See also:

- Sample – WiFi Scanner
- `struct bss_info`
- `STATUS`

Defining the operating mode

The ESP8266 can execute as a WiFi Station, a WiFi access point or both a station and an access point. These are considered the three possible global operating modes. The operating mode that is used when the device boots is retained in flash memory but can be changed with a call to `wifi_set_opmode()`. This will change the current mode as well as record the mode to be used on next restart. To merely change the mode without changing the next boot mode, we can use `wifi_set_opmode_current()`. To retrieve the current mode, we can use `wifi_get_opmode()` and to retrieve the mode used on boot, we can use `wifi_get_opmode_default()`. Quite why we have the option to change the current mode without saving it in flash memory is a mystery. Presumably there is some occasion when such a feature was needed and thus exposed but what ever that reason may be is not obvious.

See also:

- `wifi_get_opmode`
- `wifi_get_opmode_default`

Handling WiFi events

During the course of operating as a WiFi device, certain events may occur that ESP8266 needs to know about. These may be of importance or interest to the applications running within it. Since we don't know when, or even if, any events will happen, we can't have our application block waiting for them to occur. Instead what we should do is define a callback function that will be invoked should an event actually occur. The function called `wifi_set_event_handler_cb()` does just that. It registers a function that will be called when the ESP8266 detects certain types of WiFi related events. The registered function is invoked and passed a rich data structure that includes the type of event and associated data corresponding to that event. The types of events that cause the callback to occur are:

- We connected to an access point
- We disconnected from an access point
- The authorization mode changed
- We got a DHCP issued IP address
- A station connected to us when we are in Access Point mode
- A station disconnected from us when we are in Access Point mode

Here is an example of an event handler function that simply logs the name of the event that was seen:

```
void eventHandler(System_Event_t *event) {switch(event->event) {
    case EVENT_STAMODE_CONNECTED:
        os_printf("Event: EVENT_STAMODE_CONNECTED\n");
        break;
    case EVENT_STAMODE_DISCONNECTED:
        os_printf("Event: EVENT_STAMODE_DISCONNECTED\n");
        break;
    case EVENT_STAMODE_AUTHMODE_CHANGE:
        os_printf("Event: EVENT_STAMODE_AUTHMODE_CHANGE\n");
        break;
    case EVENT_STAMODE_GOT_IP:
        os_printf("Event: EVENT_STAMODE_GOT_IP\n");
        break;
    case EVENT_SOFTAPMODE_STACONNECTED:
        os_printf("Event: EVENT_SOFTAPMODE_STACONNECTED\n");
        break;
    case EVENT_SOFTAPMODE_STADISCONNECTED:
        os_printf("Event: EVENT_SOFTAPMODE_STADISCONNECTED\n");
        break;
    default:
        os_printf("Unexpected event: %d\n", event->event);
        break;
}}
```

The callback function can be registered in `user_init()` as follows:

```
wifi_set_event_handler_cb(eventHandler);
```

We are limited to what we should do in an event handler callback. Specifically, it appears that we should not try and form new connections. Instead, we should post a task that we are now able to do additional work.

See also:

- `System_Event_t`

Station configuration

When we think of an ESP8266 as a WiFi Station, we will realize that at any one time, it can only be connected to one access point. Putting it another way, there is no meaning in saying that the device is connected to **two** or more access points at the same time.

The identity of the access point to which we wish to be associated is known as the "station_config" and is modeled as the C structure called "struct station_config". Contained within that structure are two very important fields called "ssid" and "password". The `ssid` field is the SSID of the access point to which we will connect. The `password` field is the clear text value of the password that will be used to authenticate our device to the target access point to allow connection.

When booted, the ESP8266 remembers the last `station_config` we set. We can explicitly set the `station_config` data using the function `wifi_station_set_config()`. This will set the current configuration **and** save it for later retrieval after a reboot. If we only wish to set the current station config and **not** have the information persisted, we can use the `wifi_station_set_config_current()`.

We should not try and perform any WiFi operations until the device is fully initialized. We know we are initialized by registering a callback using the `system_init_done_cb()` function.

For example:

```
void initDone() {  
    wifi_set_opmode_current(STATION_MODE);  
    struct station_config stationConfig;  
    strncpy(stationConfig.ssid, "myssid", 32);  
    strncpy(stationConfig.password, "mypassword", 64);  
    wifi_station_set_config(&stationConfig);  
}
```

See also:

- `system_init_done_cb`
- `wifi_station_get_config_default`
- `wifi_station_set_config_current`
- `station_config`

Connecting to an access point

Once the ESP8266 has been set up with the station configuration details which includes the SSID and password, we are ready to perform a connection to the target access point. The function `wifi_station_connect()` will form the connection. Realize that this is not instantaneous and you should not assume that immediately following this command you are connected. Nothing in the ESP8266 blocks and as such neither does the call to this function. Some time later, we will actually be connected. We will see two callback events fired. The first is `EVENT_STAMODE_CONNECTED` indicating that we have

connected to the access point. The second event is `EVENT_STAMODE_GOT_IP` which indicates that we have been assigned an IP address by the DHCP server. Only at that point can we truly participate in communications. If we are using static IP addresses for our device, then we will only see the connected event.

There is one further consideration associated with connecting to access points and that is the idea of automatic connection. There is a boolean flag that is stored in flash that indicates whether or not the ESP8266 should attempt to automatically connect to the last used access point. If set to true, then after the device is started and without you having to code any API calls, it will attempt to connect to the last used access point. This is a convenience that I prefer to switch off. Usually, I want control in my device to determine when I connect. We can enable or disable the auto connect feature by making a call to `wifi_station_set_auto_connect()`.

See also:

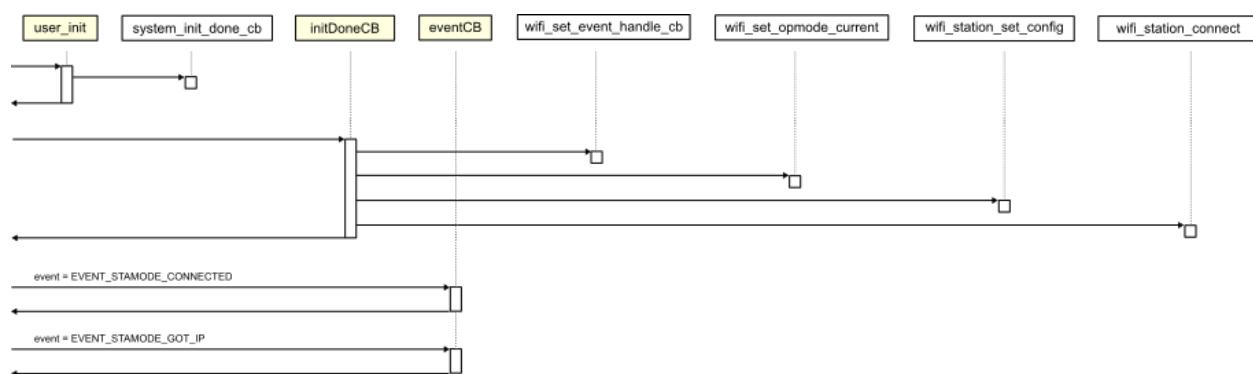
- Handling WiFi events

Control and data flows when connecting as a station

We are now at the stage where we can draw a sequence flow of the parts. Some functions you are responsible and must supply including:

- `user_init` – Entry point into the application
- `initDoneCB` – Callback when initialization has been completed
- `eventCB` – Callback when a WiFi related event is detected

The other functions we are responsible for calling. We will consider this part of the sequence completed when we have an indication that we have a valid IP address.



Being an access point

So far we have only considered the ESP8266 as a WiFi station to an existing access point but it also has the ability to **be** an access point to other WiFi devices (stations) including other ESP8266s.

In order to be an access point, we need to define the SSID that allows other devices to distinguish our network. This SSID can be flagged as hidden if we don't wish it to be scanned. In addition, we will also have to supply the authentication mode that will be used when a station wishes to connect with us. This is used to allow authorized stations and disallow non-authorized ones. Only stations that know our password will be allowed to connect. If we are using authentication, then we will also have to choose a password that the connecting stations will have to know and supply to successfully connect.

The first task in being an access point is to flag the ESP8266 as such using the `wifi_set_opmode()` or `wifi_set_opmode_current()` functions and pass in the flag that requests we be either a dedicated access point or an access point **and** a station.

Here is a snippet of code that can be used to setup an ESP8266 as an access point:

```
// Define our mode as an Access Point
wifi_set_opmode_current(SOFTAP_MODE);

// Build our Access Point configuration details
os_strcpy(config.ssid, "ESP8266");
os_strcpy(config.password, "password");
config.ssid_len = 0;
config.authmode = AUTH_OPEN;
config.ssid_hidden = 0;
config.max_connection = 4;
wifi_softap_set_config_current(&config);
```

When a remote station connects to the ESP8266 as an access point, we will see a debug message written to UART1 that may look similar to:

```
station: f0:25:b7:ff:12:c5 join, AID = 1
```

This contains the MAC address of the new station joining the network. When the station disconnects, we will see a corresponding debug log message that may be:

```
station: f0:25:b7:ff:12:c5 leave, AID = 1
```

From within the ESP8266, we can determine how many stations are currently connected with a call to `wifi_softap_get_station_num()`. If we wish to find the details of those stations, we can call `wifi_softap_get_station_info()` which will return a linked list of `struct station_info`. We have to explicitly release the storage allocated by this call with an invocation of `wifi_softap_free_station_info()`.

Here is an example of a snippet of code that lists the details of the connected stations:

```

uint8 stationCount = wifi_softap_get_station_num();
os_printf("stationCount = %d\n", stationCount);
struct station_info *stationInfo = wifi_softap_get_station_info();
if (stationInfo != NULL) {
    while (stationInfo != NULL) {
        os_printf("Station IP: %d.%d.%d.%d\n", IP2STR(&(stationInfo->ip)));
        stationInfo = STAILQ_NEXT(stationInfo, next);
    }
    wifi_softap_free_station_info();
}

```

When an ESP8266 acts as an access point, this allows other devices to connect to it and form a WiFi connection. However, it appears that two devices connected to the same ESP8266 acting as an access point can not directly communicate between each other. For example, imagine two devices connecting to an ESP8266 as an access point. They may be allocated the IP addresses 192.168.4.2 and 192.168.4.3. We might imagine that 192.168.4.2 could ping 192.168.4.3 and visa versa but that is not allowed. It appears that the only direct network connection permitted is between the newly connected stations and the access point (the ESP8266) itself.

This seems to limit the applicability of the ESP8266 as an access point. The primary intent of the ESP8266 as an access point is to allow mobile devices (eg. your phone) to connect to the ESP8266 and have a conversation with an application that runs upon it.

See also:

- [wifi_softap_set_config_current](#)
- [wifi_softap_get_station_num](#)

The DHCP server

When the ESP8266 is performing the role of an access point, it is likely that you will want it to also behave as a DHCP server so that connecting stations will be able to be automatically assigned IP addresses and learn their subnet masks and gateways.

The DHCP server can be started and stopped within the device using the APIs called `wifi_softap_dhcps_start()` and `wifi_softap_dhcps_stop()`. The current status (started or stopped) of the DHCP server can be found with a call to `wifi_softap_dhcps_status()`.

The default range of IP addresses offered by the DHCP server is 192.168.4.1 upwards. The first address becomes assigned to the ESP8266 itself. It is important to realize that this address range is **not** the same address range as your LAN where you may be working. The ESP8266 has formed its own network address space and even though they may appear with the same sorts of numbers (192.168.x.x) they are isolated and independent networks. If you start an access point on the ESP8266 and connect to it

from your phone, don't be surprised when you try and ping it from your Internet connected PC and don't get a response.

See also:

- `wifi_softap_dhcps_stop`

Current IP Address, netmask and gateway

Should we need it, we can query the OS environment for the current IP address, netmask and gateway. The values of these are commonly set for us by a DHCP server when we connect to an access point. The function called `wifi_get_ip_info()` returns our current information while the function called `wifi_set_ip_info()` allows us to set our addresses.

When we connect to an access point and have chosen to use DHCP, when we are allocated an IP address, an event is generated that can be used as an indication that we now have a valid IP address.

To correctly setup static IP addresses, in the `init_done` callback, call `wifi_station_dhcpc_stop()` to disable the DHCP client running in the ESP8266. After this call `wifi_station_connect()` to start the access point connection phase. When the event arrives that indicates we are connected to an access point as a station (`EVENT_STAMODE_CONNECTED`), we can call `wifi_set_ip_info()` and pass in the IP address, gateway and netmask that we wish to use. Note that when we use a static IP address, we will not receive the callback event that indicates we have received an IP address (`EVENT_STAMODE_GOT_IP`) as we already have it.

See also:

- Handling WiFi events
- `wifi_station_dhcpc_stop`
- `struct ip_info`

WiFi Protected Setup - WPS

The ESP8266 supports WiFi Protected Setup in station mode. This means that if the access point supports it, the ESP8266 can connect to the access point without presenting a password. Currently only the "push button mode" of connection is implemented. Using this mechanism, a physical button is pressed on the access point and, for a period of two minutes, any station in range can join the network using the WPS protocols. An example of use would be the access point WPS button being pressed and then the ESP8266 device calling `wifi_wps_enable()` and then `wifi_wps_start()`. The ESP8266 would then connect to the network.

See also:

- `wifi_wps_enable`

- `wifi_wps_start`
- `wifi_set_wps_cb`
- [Simple Questions: What is WPS \(WiFi Protected Setup\)](#)
- Wikipedia: [WiFi Protected Setup](#)

Working with TCP/IP

TCP/IP is the network protocol that is used on the Internet. It is the protocol that the ESP8266 natively understands and uses with WiFi as the transport. Books upon books have already been written about TCP/IP and our goal is not to attempt to reproduce a detailed discussion of how it works, however, there are some concepts that we will try and capture.

First, there is the IP address. This is a 32bit value and should be unique to every device connected to the Internet. A 32bit value can be thought of as four distinct 8bit values ($4 \times 8 = 32$). Since we can represent an 8bit number as a decimal value between 0 and 255, we commonly represent IP addresses with the notation

`<number>.<number>.<number>.<number>` for example 173.194.64.102. These IP addresses are not commonly entered in applications. Instead a textual name is typed such as "`google.com`" ... but don't be misled, these names are an illusion at the TCP/IP level. All work is performed with 32bit IP addresses. There is a mapping system that takes a name (such as "`google.com`") and retrieves its corresponding IP address. The technology that does this is called the "Domain Name System" or DNS.

When we think of TCP/IP, there are actually three distinct protocols at play here. The first is IP (Internet Protocol). This is the underlying transport layer datagram passing protocol. Above the IP layer is TCP (Transmission Control Protocol) which provides the illusion of a connection over the connectionless IP protocol. Finally there is UDP (User Datagram Protocol). This too lives above the IP protocol and provides datagram (connectionless) transmission between applications. When we say TCP/IP, we are **not** just talking about TCP running over IP but are in fact using this as a shorthand for the core protocols which are IP, TCP and UDP and additional related application level protocols such as DNS, HTTP, FTP, Telnet and more.

The espconn architecture

Because we are not allowed to block control in the ESP8266 for any length of time, we must register callback functions which will be invoked when some long duration action has completed or an asynchronous events occurs. For example, when we wish to receive an incoming network connection, we can't simply wait for that connection to arrive. Instead, we register a connection callback function and then return control back to the OS. When the connection eventually arrives in the future, the callback function that we previously registered is invoked on our behalf.

The following table lists the callback functions that the ESP8266 provides supporting TCP connections and events.

Register Function	Callback	Description
espconn_regist_connectcb	espconn_connect_callback	TCP connected successfully
espconn_regist_disconcb	espconn_disconnect_callback	TCP disconnected successfully
espconn_regist_reconcb	espconn_reconnect_callback	Error detected or TCP disconnected
espconn_regist_sentcb	espconn_sent_callback	Sent TCP or UDP data
espconn_regist_recvcb	espconn_recv_callback	Received TCP or UDP data
espconn_regist_write_finish	espconn_write_finish_callback	Write data into TCP-send-buffer

See also:

- [espconn_regist_connectcb](#)
- [espconn_regist_disconcb](#)
- [espconn_regist_reconcb](#)
- [espconn_regist_sentcb](#)
- [espconn_regist_recvcb](#)
- [espconn_regist_write_finish](#)

TCP

A TCP connection is a bi-directional pipe through which data can flow in both directions. Before the connection is established, one side is acting as a server. It is passively listening for incoming connection requests. It will simply sit there for as long as needed until a connection request arrives. The other side of the connection is responsible for initiating the connection and it actively asks for a connection to be formed. Once the connection has been constructed, both sides can send and receive data. In order for the "client" to request a connection, it must know the address information on which the server is listening. This address is composed of two distinct parts. The first part is the IP address of the server and the second part is the "port number" for the specific listener. If we think about a PC, you may have many applications on it, each of which can receive an incoming connection. Just knowing the IP address of your PC is not sufficient to address a connection to the correct application. The combination of IP address plus port number provides all the addressing necessary.

As an analogy to this, think of your cell phone. It is passively sitting there until someone calls it. In our story, it is the listener. The address that someone uses to form a connection is your phone number which is comprised of an area code plus the remainder. For example, a phone number of (817) 555-1234 will reach a particular phone. However the area code of 817 is for Fort Worth in Texas ... calling that by itself is not sufficient to reach an individual ... the full phone number is required.

No we will look at how an ESP8266 can set itself up as a listener for an incoming TCP/IP connection.

We start by introducing an absolutely vital data structure that is called "struct espconn". This data structure contains much of the "state" of our connection and is passed into most of our TCP APIs.

We initialize it by setting a number of its fields:

- `type` – This is the type of connection we are going to use. Since we want to use a TCP connection as opposed to a UDP connection, we supply `ESPCONN_TCP` as the value.
- `state` – The state of the connection will change over time but we initialize it to have an initial empty state by supplying `ESPCONN_NONE`.

For example:

```
struct espconn conn1;

void init() {
    conn1.type = ESPCONN_TCP;
    conn1.state = ESPCONN_NONE;
}
```

Now we introduce another structure called "esp_tcp". This structure contains TCP specific settings. For our story, this is where we supply the port number which our TCP connection will listen upon for client connections. This is supplied in the property called "local_port".

```
esp_tcp tcp1;

void init() {
    tcp1.local_port = 25867;
}
```

Within the `struct espconn` data type, there is a field called "proto" which is a pointer to a protocol specific data structure. For a TCP connection, this will be a pointer to an "esp_tcp" instance ... and this is where we get to glue the story together. The full code becomes:

```
struct espconn conn1;
esp_tcp tcp1;

void init() {
    tcp1.local_port = 25867;
    conn1.type = ESPCONN_TCP;
    conn1.state = ESPCONN_NONE;
    conn1.proto.tcp = &tcp1;
}
```

We can now start our server listening for incoming TCP connections using `espconn_accept()`. This takes the `struct espconn` as input which is used to indicate on what port we should listen (among other things). Here is an example:

```
espconn_accept(&conn1);
```

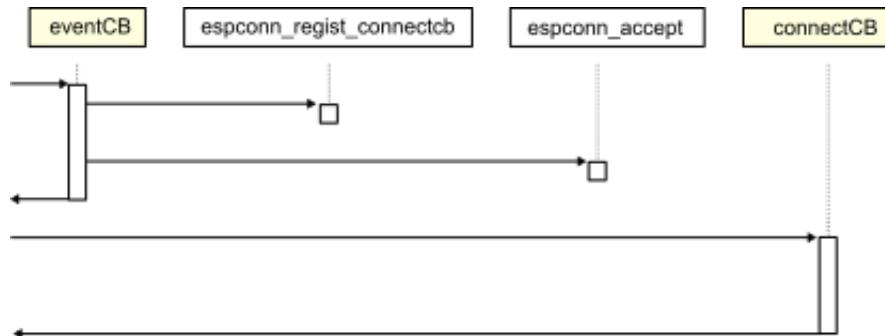
After calling this, the ESP8266 will now be passively listening for incoming TCP connections on the port specified in the `local_port` field. It is important to note that your code does not block waiting for an incoming request. Somewhere in the heart of the ESP8266 it now knows to accept connections on that port. The next question is a simple one ... what happens when a connection eventually arrives?

The answer to that is part of the core architecture of the device and revolves around the notion of callbacks. In your own application code, it is your responsibility to register a callback function that will be invoked when the connection arrives. This is where the `espconn_regist_connectcb()` function comes into play. This function registers a user supplied callback function that will be called when a connection arrives.

```
void connectCB(void *arg) {
    struct espconn *pNewEspConn = (struct espconn *)arg;
    ... Do something with the new connection
}

{
    ...
    espconn_regist_connectcb(&conn1, connectCB);
    espconn_accept(&conn1);
}
```

Seen as a sequence flow diagram, we can see the relationships between some of the components. We assume that in the event callback when we have been allocated an IP address, we then register that we are interested in connections and that we are willing to accept incoming new connections. Then, at some time in the future, we receive a new connection request and the connection callback is invoked.



The content of the `struct espconn` passed into the callback will include the remote IP address of the partner that connected with us. We can use that information for logging or for authorization. For example, if the IP address is not one we wish to allow, we can disconnect at this point using `espconn_disconnect()`. Realize that this data structure represents the **new** connection with the partner that just invoked up and is **not** the same as `struct espconn` that was used to register that we wanted to accept new connections. A **new** `struct espconn` will be passed in for each new connection formed.

This covers the ESP8266 receiving incoming connection requests, but what if it should desire to form a connection outbound to a remote TCP application? To perform an outbound connection request we can use the `espconn_connect()` call. Prior to making this call, we must set up the TCP structure. The field `remote_port` must contain the port number of the application partner to which we wish to connect. In addition the `remote_ip` field must contain the IP address of the machine hosting the partner. The `local_port` must be assigned an unused local port using `espconn_port()`. The `local_ip` must also be completed using the local IP address. Just like the receiving an inbound connection, making an outbound connection will result in an invocation to the connection callback when the connection is established. Once the connection has been formed, once again, the two ends of the connection will be peers of each other. It is **vital** to realize that just issuing an `espconn_connect()` does **not** result in an immediate connection. Instead, only after the `connectCB` has been received can we actually use the connection.

For example:

```
struct espconn conn1;
esp_tcp tcp1;

void init() {
    tcp1.remote_port = 25867;
    tcp1.remote_ip = ipAddress;
    tcp1.local_port = espconn_port();
    struct ip_info ipconfig;
    wifi_get_ip_info(STATION_IF, &ipconfig);
    os_memcpy(tcp1.local_ip, &ipconfig.ip, 4);

    conn1.type = ESPCONN_TCP;
    conn1.state = ESPCONN_NONE;
    conn1.proto.tcp = &tcp1;
}
```

If the partner in our conversation should close the connection, we will be informed of that through the function we register with `espconn_regist_disconcb()`. The state field of the `struct espconn` will contain `CLOSE`. Detection the graceful shutdown of a partner allows us to perform logic that we may need such as releasing resources or persisting data.

If a TCP connection is formed and no traffic flows over the connection for at least 10 seconds (default), then the connection is automatically closed from the ESP8266 end. The idle connection timeout property can be set with the `espconn_regist_time()` function.

The ESP8266 support a maximum of 5 concurrent TCP connections.

See also:

- `espconn_accept`

- espconn_connect
- espconn_disconnect
- espconn_regist_connectcb
- espconn_regist_disconcb
- espconn_regist_time
- struct espconn
- esp_tcp

Sending and receiving TCP data

At this point, let us now assume that we have a connection between an ESP8266 and a partner application. Having a connection is great but now we need to have a conversation. Information and data needs to flow in one or both directions. There are two considerations... we may receive data from the partner or we may wish to send data to the partner. It is important to note that in TCP, a connection is bidirectional. Once the connection has been established, either party can send data at any time. There is no concept of one party having exclusive sending or receiving rights. The choice of who is the receiver and who is the transmitter is purely up to the design of the application.

For example, imagine we had a project to turn on an LED at an ESP8266 when it receives a "1" character and turn it off when it receives a "0" character. In that story, the ESP8266 would be exclusively a receiver and, simply by our choices, need not transmit data. The partner would be exclusively a transmitter.

Now let us consider a second example. In this case the ESP8266 is connected to a temperature sensor and every few seconds it sends the current temperature to the partner. In that story, the ESP8266 is exclusively a transmitter and the partner only a receiver.

Finally, we can image an ESP8266 connected to multiple sensors. It receives commands from the partner as input which it interprets. Based on the received data, the correct sensor is chosen, its value read and the results transmitted back. In this story, the ESP8266 is at first a receiver and then becomes a transmitter while the partner is the opposite.

To receive data from a partner, we register a callback function using `espconn_regist_recvcb()`. We pass in the `struct espconn` that was supplied in the connected callback that identifies our connection. This registered callback function is invoked when new data becomes available from the partner. The callback function is passed a buffer containing the data and an indicator of how much data was received.

The following is an example of logging data that is received over the network:

```
void recvCB(void *arg, char *pData, unsigned short len) {
    struct espconn *pEspConn = (struct espconn *)arg;
    os_printf("Received data!! - length = %d\n", len);
    int i=0;
```

```

    for (i=0; i<len; i++) {
        os_printf("%c", pData[i]);
    }
    os_printf("\n");
} // End of recvCB

```

The function called `recvCB()` is registered as a callback when data is available for the connection. With this in mind, we can start running some experiments and the results will be interesting.

If we send data, we see the callback being invoked as expected. However, as the size of the data transmitted, which is received by the ESP8266, increases, at about 1460 bytes, a strange thing happens. Instead of `recvCB()` being called once, we see it being called twice. The first time it gets the first 1460 bytes and the second time it gets what remains. This is repeated for increments of 1460 byte transmission sizes. For example, if we send 5000 bytes, `recvCB()` is called 4 times. The first three times with 1460 bytes of data and the last with 620 bytes giving a total of 5000.

Why would this be? Part of the answer is that the ESP8266 has only a very small amount of RAM available to it and needs to be able to support parallel connections. As such, it can apparently throttle the data being sent from the sender until space is available to process it.

It can't be stressed enough the importance of this concept. Data sent from the server over a TCP connection is "streamed" to the ESP8266. There is no concept of a unit of data transmission. Instead data sent in the pipe at the sender will arrive at the ESP8266 but it may very well arrive at different rates. The order of the transmitted data is preserved (obviously). In principle, making two transmissions at the sender of 5 bytes each could result in one receive at the ESP8266 of 10 bytes or just as easily one receive of 1 byte and one receive of 9 bytes. Don't make **any** assumptions about the bracketing of TCP data.

To transmit data to a partner we use the function called `espconn_send()`.

This command takes the `struct espconn` which identifies which connection to send data through. The function also takes a pointer to a buffer of data and the length of the data to send. A vital consideration is that the data to be sent is not sent immediately. When we call `espconn_send()` what we are doing is handing off a buffer of data to be transmitted at some time in the future. We anticipate this will be a few milliseconds but it could be longer. We must honor the contract. When the ESP8266 does successfully transmit the data, a callback will be made to a function that was registered with the `espconn_register_sentcb()`. Only after having seen a confirmation that the last transmission request has been completed should we execute another `espconn_send()` request.

When we ask for data to be transmitted, we provide a pointer to a buffer that contains the data. It is important to realize that we must maintain that data until after we are sure its content has been sent. For example, we can't request a transmission and then immediately dispose off or change the buffer. What we hand off to the OS is a pointer to a buffer and until the OS tells us that it has finished consuming it, we must maintain its integrity.

See also:

- `espconn_regist_recvcb`
- `espconn_send`

Flow control

Consider the notion of an ESP8266 in communication with a partner and the partner is sending 5K of data per second. Now imagine that the ESP8266 is only processing 1K of data per second. As you can see, something will go wrong quite quickly. We will overwhelm the ESP8266 with too much data. What we really want is to institute a flow control mechanism such that the sender of the data is told to throttle back its delivery of data to a rate that the ESP8266 can accommodate. The send may choose to buffer data at its end or else may be told to not send so much data per unit of time by pushing back to the original transmitting logic.

See also:

- `espconn_recv_hold`
- `espconn_recv_unhold`

TCP Error handling

When a connection is formed between two partners it is essential that we realize that there isn't an actual dedicated underlying connection between them. Instead, there is only a logical connection that appears to be present over the datagram oriented protocol of IP. What this might mean is that if one end of the connection abnormally ends, the other end won't immediately know about it. As an example, if in the real world I make a phone call to you then your phone indicates to you that we have a connection. If the battery on my phone dies the telephone network detects that and drops the connection. Your phone also hangs up and you know we are no longer in communication. In the TCP world, that doesn't happen. If my "TCP" phone dies, your "TCP" phone isn't told that mine is gone. You may be left sitting there indefinitely listening to silence and waiting for me to say something.

To resolve that situation, TCP introduces a concept called "keep-alive". The notion is very simple. With keep-alive, the two partners periodically exchange a heartbeat communication with each other. As long as they each hear the heartbeat of the other, they are both still present. However, if one side of the connection is lost, the heartbeat

request will be sent but no response will arrive at which point, the one sending the heartbeat will assume that the partner has gone and we can take appropriate cleanup and shutdown actions.

There is an API available to us to control the keep-alive settings. It is called `espconn_set_keepalive()`. It has a number of properties including:

- How long should we wait since the last time we heard from the partner before sending a heartbeat?
- If no response, how long between subsequent heartbeats?
- How many times should we send a heartbeat until we declare the partner dead?

It is recommended that if keep-alive processing is to be used then the keep-alive settings be made in the callback handler of the connect callback. The keep-alive option must also be explicitly enabled using the `espconn_set_opt()` call prior to setting the keep-alive properties.

If the partner connection is lost, we can detect that by registering a callback function with `espconn_reconnect_callback()`.

See also:

- `espconn_set_keepalive`
- `espconn_get_keepalive`
- `espconn_set_opt`
- `espconn_clear_opt`

UDP

If we think of TCP as forming a connection between two parties similar to a telephone call, then UDP is like sending a letter through the postal system. If I were to send you a letter, I would need to know your name and address. Your address is needed so that the letter can be delivered to the correct house while your name ensure that it ends up in your hands as opposed to someone else who may live with you. In TCP/IP terms, the address is the IP address and the name is the port number.

With a telephone conversation, we can exchange as much or as little information as we like. Sometimes I talk, sometimes you talk ... but there is no maximum limit on how much information we can exchange in one conversation. With a letter however, there are only so many pages of paper that will fit in the envelopes I have at my disposal.

The notion of the mail analogy is how we might choose to think about UDP. The acronym stands for User Datagram Protocol and it is the notion of the datagram that is akin to the letter. A datagram is an array of bytes that are transmitted from the sender to the receiver as a unit. The maximum size of a datagram using UDP is 64KBytes. No connection need be setup between the two parties before data starts to flow. However,

there is a down side. The sender of the data will not be made aware of a receiver's failure to retrieve the data. With TCP, we have handshaking between the two parties that lets the sender know that the data was received and, if not, can automatically retransmit until it has been received or we decide to give up. With UDP, and just like a letter, when we send a datagram, we lose sight of whether or not it actually arrives safely at the destination.

If we wish to receive incoming datagrams, we must register what port number we are interested in receiving them upon. We achieve that through the poorly named `espconn_create()` function. This function causes the ESP8266 to start listening for incoming datagrams on the local port defined in the `struct espconn`. After calling this function, you should then call `espconn_regist_recvcb()` to register a callback function that will be invoked when a datagram arrives.

Here is a high level example of setting up a UDP listener once an IP address has been allocated:

```
struct espconn conn1;
esp_udp udp1;

void setupUDP() {
    sint8 err;
    conn1.type = ESPCONN_UDP;
    conn1.state = ESPCONN_NONE;
    udp1.local_port = 25867;
    conn1.proto.udp = &udp1;

    err = espconn_create(&conn1);
    err = espconn_regist_recvcb(&conn1, recvCB);
} // End of setupUDP
```

Should we wish to stop the ESP8266 from listening for datagrams, we can call the function called `espconn_delete()`.

Now is a good time to come back to IP addresses and port numbers. We should start to be aware that on a PC, only one application can be listening upon any given port. For example, if my application is listening on port 25867, then no other application can also be listening on that same port ... not your application nor another copy instance of mine. When an incoming connection or datagram arrives at a machine, it has arrived because the IP address of the sent data matches the IP address of the device at which it arrived. We then route within the device based on port numbers. And here is where I want to clarify a detail. We route within the machine based on the **pair** of both protocol and port number.

So for example, if a request arrives at a machine for port 25867 over a TCP connection, it is routed to the TCP application watching port 25867. If a request arrives at the same machine for port 25867 over UDP, it is routed to the UDP application watching port 25867. What this means is that we **can** have two applications listening on the same

port but on different protocols. Putting this more formally, the allocation space for port numbers is a function of the protocol and it is not allowed for two applications to simultaneously reserve the same port within the same protocol allocation space. Although I used the story of a PC running multiple applications, in our ESP8266 the story is similar even though we just run one application on the device. If your single application should need to listen on multiple ports, don't try and use the same port with the same protocol as the second function call will find the first one has already allocated the port. This is a detail that I am happy for you to forget as you will rarely come across it but I wanted to catch it here for completeness.

Now let us look at what it takes to send a datagram. Similar to other functions, we need a `struct espconn` control block. This must be configured to use UDP and name the remote IP address and port. Once populated, we can then initialize the data structure with a call to `espconn_create()` and now we are ready to send data. We use the `espconn_send()` function. When we have sent all our data, we can conclude with an `espconn_delete()` to release the resources that the ESP8266 maintains for data sending.

Here is an example:

```
struct espconn sendResponse;
esp_udp udp;

void sendDatagram(char *datagram, uint16 size) {
    sendResponse.type = ESPCONN_UDP;
    sendResponse.state = ESPCONN_NONE;
    sendResponse.proto.udp = &udp;
    IP4_ADDR((ip_addr_t *)sendResponse.proto.udp->remote_ip, 192, 168, 1, 7);
    sendResponse.proto.udp->remote_port = 9876; // Remote port
    err = espconn_create(&sendResponse);
    err = espconn_send(&sendResponse, "hi123", 5);
    err = espconn_delete(&sendResponse);
}
```

See also:

- `espconn_create`
- `espconn_delete`
- `espconn_send`
- `espconn_regist_recvcb`
- `espconn_regist_sentcb`
- `struct espconn`

Broadcast with UDP

One of the features available to us with UDP is the concept of broadcast. This is the notion that a sender of data can build a datagram and transmit it such that all the devices on the same subnet can receive a copy of it. Receivers choose a UDP port and start listening upon it just as they normally would. A transmitting application transmits a

message on the same port but with an IP address where the host part of the IP address is all binary ones. For example, if we have a netmask of 255.255.255.0 and our network is 192.168.1.x, then transmitting on the IP address 192.168.1.255 will be a broadcast. A special IP address of 255.255.255.255 represents a broadcast on our local network.

For the ESP8266, there is an API called `wifi_set_broadcast_if()` which determines which interfaces will be available for broadcast. The choices are the station, the access point or both the station and access point. A corresponding API called `wifi_get_broadcast_if()` can be used to retrieve the current broadcast configuration state.

See also:

- `wifi_set_broadcast_if`
- `wifi_get_broadcast_if`

Ping request

At the TCP/IP level, a device with an IP address can "ping" another device with an IP address. What this means is that messages are transmitted between them that allows them to know that they have a route through the network to each other. If the destination is either not running or no route is available, we will also be informed that there was a failure.

The ESP8266 provides a structure called `struct ping_option` that contains the details of a ping request. This is passed in as a parameter to the function called `ping_start()` which initiates the ping. Before calling this function, the target IP address and the number of ping requests should be set within the `struct ping_option`.

Two callback functions can be registered with `ping_regist_recv()` and `ping_regist_sent()`. The first is called when a ping response is received and the other is called when a ping request is sent.

See also:

- `ping_start`
- `ping_regist_recv`
- `ping_regist_sent`
- `struct ping_option`

Name Service

On the Internet, server machines can be found by their Domain Name Service (DNS) names. This is the service that resolves a human readable representation of a machine such as "google.com" into the necessary IP address value (eg. 216.58.217.206). In order for this transformation to happen, the ESP8266 needs to know the IP address of one or more DNS servers that it will then use to perform the name to IP address

mapping. If we are using DHCP then nothing else need be done as the DHCP server automatically provides the DNS server addresses. However, if we should not be using DHCP, then we need to instruct the ESP8266 of the locations of the DNS servers manually. We can do this using the `espconn_dns_setserver()` function. This takes an array of one or two IP addresses as input and from that point onwards, these servers will be used for DNS resolution. If two addresses are supplied and the first is unresponsive, the second will be used.

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

Once we have define the nameservers, we can look up the address of a hostname using the `espconn_gethostbyname()` function. The return code for this call should be carefully examined. We may have the address immediately because of a cache or we may need to perform a network request and provide a callback for later retrieval. If the later, the `ipAddr` is returned as NULL ... however, your DNS provider may choose to provide an IP address of a search engine and hence you'll get an address back ... but not the one to the host you expected!!

See also:

- `espconn_dns_setserver`
- `espconn_gethostbyname`
- Wikipedia: [Domain Name System](#)
- Google: [Public DNS](#)

Multicast Domain Name Systems

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

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

Using the Multicast Domain Name System (mDNS) an ESP8266 can attempt to resolve a hostname of a machine on the local network to its IP address. It does this by broadcasting a packet asking for the machine with that identity to respond.

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

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

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

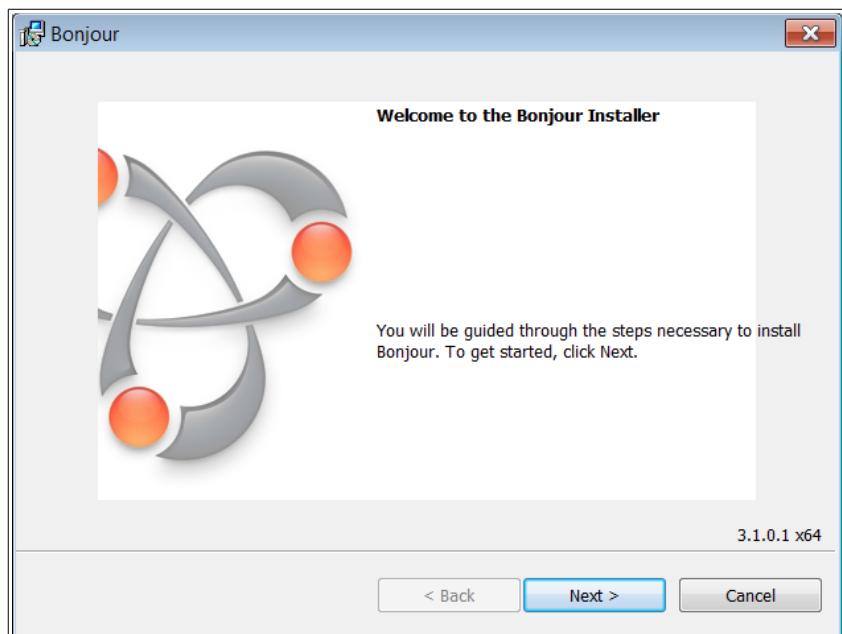
See also:

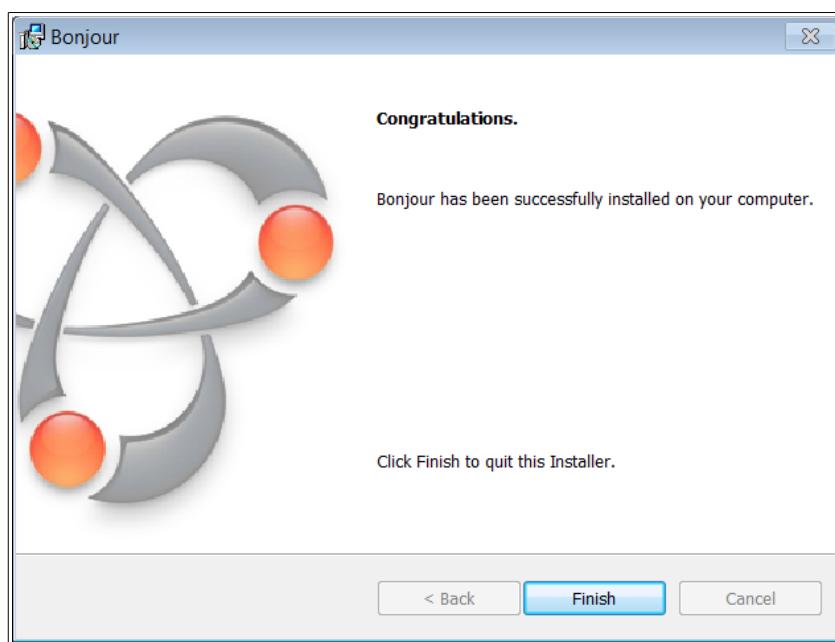
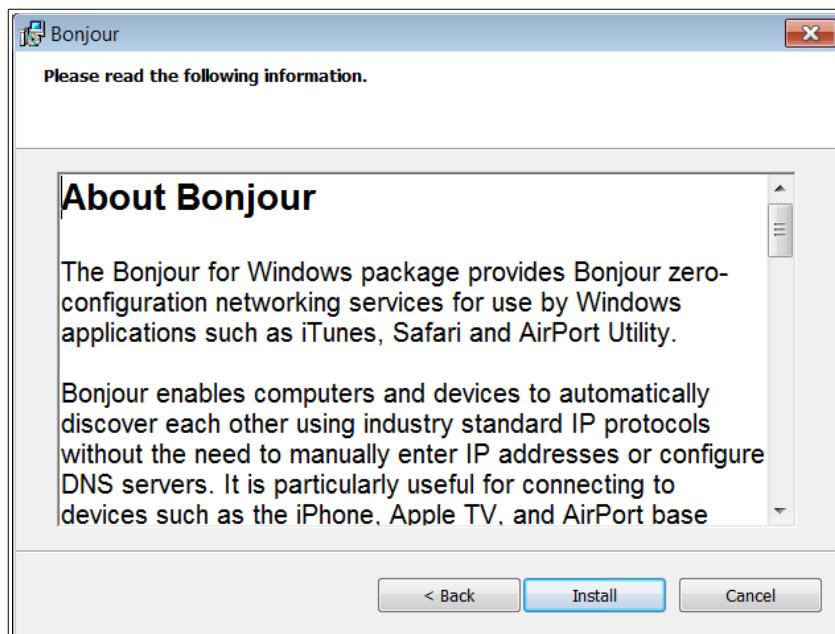
- Wikipedia – [Multicast DNS](#)
- IETF RFC 6762: [Multicast DNS](#)
- [Multicast DNS](#)
- [New DNS Technologies in the Lan](#)
- [Avahi](#) – Implementation of mDNS ... source project for Unix machines
- Adafruit – [Bonjour \(Zeroconf\) Networking for Windows and Linux](#)
- [chrome.mdns](#) – API description for Chrome API for mDNS
- Android – [ZeroConf Browser](#)
- espconn_mdns_init
- espconn_mdns_close
- espconn_mdns_server_register
- espconn_mdns_server_unregister
- espconn_mdns_get_servername
- espconn_mdns_set_servername
- espconn_mdns_set_hostname
- espconn_mdns_get_hostname
- espconn_mdns_disable
- espconn_mdns_enable

Installing Bonjour

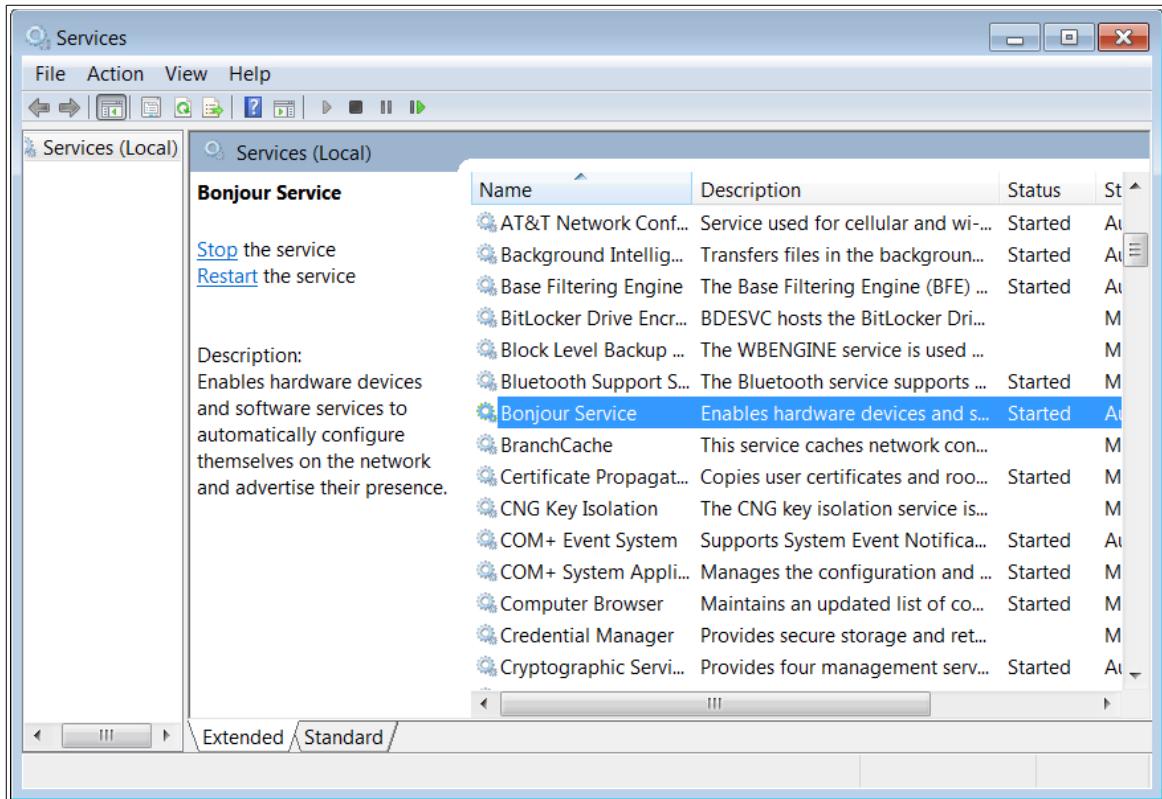
Launch the Bonjour installer:







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



Working with SNTP

SNTP is the Simple Network Time Protocol and allows a device connected to the Internet to learn the current time. In order to use this, you must know of at least one time server located on the Internet. The US National Institute for Science and Technology (NIST) maintains a number of these which can be found here:

- <http://tf.nist.gov/tf-cgi/servers.cgi>

Other time servers can be found all over the globe and I encourage you to Google search for your nearest or country specific server.

Once you know the identity of a server by its hostname or IP address, you can call either of the functions called `sntp_setservername()` or `sntp_setserver()` to declare that we wish to use that time server instance. The ESP8266 can be configured with up to three different time servers so that if one or two are not available, we might still get a result.

The ESP8266 must also be told the local timezone in which it is running. This is set with a call to `sntp_set_timezone()` which takes the number of hours offset from UTC. For example, I am in Texas and my timezone offset becomes "-5".

With these configured, we can start the SNTP service on the ESP8266 by calling `sntp_init()`. This will cause the device to determine its current time by sending packets over the network to the time servers and examining their responses. It is important to note that immediately after calling `sntp_init()`, you will not yet know what the current time may be. This is because it may take a few seconds for the ESP8266 to send the time requests and get their responses and this will all happen asynchronously to your current commands and won't complete till sometime later.

When ready, we can retrieve the current time with a call to `sntp_get_current_timestamp()` which will return the number of seconds since the 1st of January 1970 UTC. We can also call the function called `sntp_get_real_time()` which will return a string representation of the time.

See also:

- `sntp_setserver`
- `sntp_setservername`
- `sntp_init`
- `sntp_set_timezone`
- `sntp_get_current_timestamp`
- `sntp_get_real_time`
- [IETF RFC5905: Network Time Protocol Version 4: Protocol and Algorithms Specification](#)

ESP-NOW

The concept of ESP-NOW is to achieve a private protocol between sets of ESP8266s. Think of it loosely as a controller/slave relationship where we have one controller and potentially multiple slaves. The slaves form "persistent" connections to the controller. What this means is that when a slave ESP8266 is powered on, it is virtually immediately able to transmit to the controller. Compare this with the notion of the ESP8266 powering on, connecting to the access point and then connecting to the master device. These flows take time while ESP-NOW is much faster from a startup.

To start, an ESP8266 which wishes to participate in using this protocol will invoke `esp_now_init()`. Should it no longer wish to be part of this kind of network, it can call `esp_now_deinit()`. Before communication can proceed, the device will have to add the peers in the network. This is achieved through a call to `esp_now_add_peer()`.

Each ESP8266 device in the network will declare itself as having a role of either a slave or a controller through a call to `esp_now_set_self_role()`.

A corresponding `esp_now_delete_peer()` can be used to forget about a previously registered peer. When ready to transmit data, a call can be made to `esp_now_send()` supplying the address of the recipient as well as the data to be transmitted. The maximum amount of data that can be currently sent as a unit is 256 bytes.

Two callbacks are available which are invoked when either a new message has been transmitted or a new message has been received.

See also:

- `esp_now_init`
- `esp_now_send`
- `esp_now_register_recv_cb`

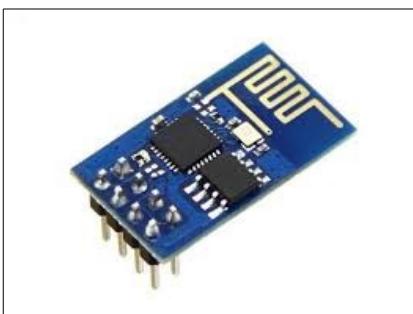
GPIOs

The ESP8266 has 17 GPIO pins. When we think of a GPIO we must realize that at any one time, each instance has two operational modes. It can either be an input or an output. When it is an input, we can read a value from it and determine the logic level of the signal present at the physical pin. When it is an output, we can write a logic level to it and that will appear as a physical output.

Remember to distinguish between the ESP8266 integrated circuit which is a tiny device:



which differs from the various models of breakout board such as the ESP-1:



which has 8 pins exposed, 4 of which are GPIO. The module called ESP-12:

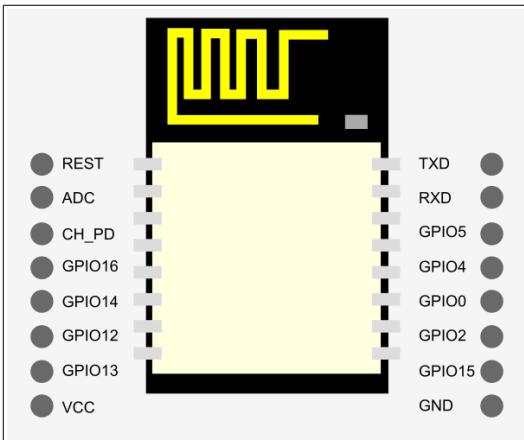


which has 16 pins exposed, 11 of which are GPIO.

For GPIO, here are the exposed mappings:

Pin	ESP-1	ESP-12
GPIO 0	•	•
GPIO 1	•	•
GPIO 2	•	•
GPIO 3	•	•
GPIO 4		•
GPIO 5		•
GPIO 6		
GPIO 7		
GPIO 8		
GPIO 9		
GPIO 10		
GPIO 11		
GPIO 12		•
GPIO 13		•
GPIO 14		•
GPIO 15		•
GPIO 16		•
Totals	4	11

It is also good to remind ourselves of the pin-outs of the device.



As you can see there is no obvious pattern to the layout of the pins and as such you must take great care when wiring up a circuit. It is easy to make a mistake.

Another vital consideration when working with GPIOs is voltage. The ESP8266 is a 3.3V device. You need to be extremely cautious if you are working with 5V (or above) partner MCUs or sensors. Unfortunately devices like the Arduino are typically 5V as are USB → UART converters and many sensors. This means you are as likely as not to be working in a mixed voltage environment. Under no circumstances should you think you can power the ESP8266 with a direct voltage of more than 3.3V. Obviously, you can convert higher voltages down to 3.3V but never try and connect a greater voltage directly. Another subtler consideration is when using GPIOs for signal input and supply greater than 3.3V as a high signal value. I strongly suggest not doing it. Some folks may claim you can "get away with it" and if you experiment it may (seem) to work but you are taking an unnecessary risk for no obviously good reason. If it works ... then it will work till it doesn't at which point it will be too late and you may cook your device.

In my own experiments, I have accidentally over-powered ESP8266s, reverse voltage powered ESP8266s and applied too high a voltage as input. In each case the result was a dead chip and in a few cases, attempting to see if it still worked by applying normal voltage resulted in the device not only not working but getting so hot to the touch it burned my fingers.

Because accidents happen when building GPIO based circuits, I recommend buying more ESP8266 instances than you need. That way if you do happen to find yourself needing a second (or third or fourth) you will have them at your disposal.

Prior to making use of any ESP8266 GPIO functions, you must call the supplied `gpio_init()` function. What this actually does is unknown however the rules say call it and call it we must.

The way that the ESP8266 thinks of GPIOs is as though each GPIO was a bit in a 16bit array.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

(We will come back to how 17 GPIOs maps to 16 bits at a later time)

One array contains an indication of whether or not the GPIO is input or output. We will call this the direction array. A second array indicates the values of the GPIOs. For input GPIOs, the value is the value on the pin. For output GPIOs, the value is the value to be written to the pin. We will call this the value array.

A function is supplied by the ESP8266 called `gpio_output_set()`. This function takes **four** 16 bit values to be used as masks against the two 16 bit arrays.

The first mask is called the "`set_mask`". A `1` value in the set mask sets the corresponding bit value to be `1` in the value array.

The second mask is called the "`clear_mask`". A `1` value in the clear mask sets the corresponding bit value to be `0` in the value array.

Notice that in both cases, if the masks have a `0` value, the original values are unchanged.

The third mask is called the "`enable_output`" mask. A `1` value in the enable output mask sets the corresponding GPIO to be in output mode.

The fourth mask is called the "`enable_input`" mask. A `1` value in the enable input mask sets the corresponding GPIO to be in input mode.

Take care not to set a GPIO to be both input and output or to have a value of both `1` and `0`. The results will be undefined.

Constants are defined for each of the bit positions. Those constants are:

- `BIT0 – 2^0`
- `BIT1 – 2^1`
- `...`
- `BIT31 – 2^31`

So, for example. If we want to set GPIO 5 to be input, we might code:

```
gpio_output_set(0, 0, 0, BIT5);
```

to set GPIO 4 to be output and have a high value, we might code:

```
gpio_output_set(BIT4, 0, BIT4, 0);
```

to set GPIO 0 and 1 to both be output and the first to be 1 and the second to be 0:

```
gpio_output_set(BIT0, BIT1, BIT0 | BIT1, 0);
```

If we wish to retrieve the values of the GPIOs, we can use the `gpio_input_get()` method. This returns a bit mask containing all the bits.

We have some helper macros that are available. These are useful wrappers around `gpio_output_set()` and `gpio_input_get()`.

- `GPIO_OUTPUT_SET(GPIO_NUMBER, value)` – Sets the corresponding GPIO to be output and sets its value.
- `GPIO_DIS_OUTPUT(GPIO_NUMBER)` – Sets the corresponding GPIO to be input (disabled output).
- `GPIO_INPUT_GET(GPIO_NUMBER)` – Gets the value of the input GPIO

Since pins on an ESP8266 can serve multiple purposes, we must first declare what function that pin will have. To do this, we use a macro which sets the function of the logical pin:

```
PIN_FUNC_SELECT(pinName, functionUsage)
```

For example, to define GPIO2 as a GPIO pin and set its value, we might code:

```
PIN_FUNC_SELECT(PERIPH_IO_MUX_GPIO2_U, FUNC_GPIO2);  
GPIO_OUTPUT_SET(2, 1);
```

Here is the complete table of mappings.

Pin Name	Function 1	Function 2	Function 3	Function 4	Physical pin	Devices
MTDI_U	MTDI	I2SI_DATA	HSPIQ MISO	GPIO12	10	12
MTCK_U	MTCK	I2SI_BCK	HSPID MOSI	GPIO13	12	12
MTMS_U	MTMS	I2SI_WS	HSPICLK	GPIO14	9	12
MTDO_U	MTDO	I2SO_BCK	HSPICS	GPIO15	13	12
U0RXD_U	U0RXD	I2SO_DATA		GPIO3	25	1, 12
U0TXD_U	U0TXD	SPICS1		GPIO1	26	1, 12
SD_CLK_U	SD_CLK	SPICLK		GPIO6	21	
SD_DATA0_U	SD_DATA0	SPIQ		GPIO7	22	
SD_DATA1_U	SD_DATA1	SPIID		GPIO8	23	
SD_DATA2_U	SD_DATA2	SPIHD		GPIO9	18	
SD_DATA3_U	SD_DATA3	SPIWP		GPIO10	19	
SD_CMD_U	SD_CMD	SPICS0		GPIO11	20	
GPIO0_U	GPIO0	SPICS2			15	1, 12
GPIO2_U	GPIO2	I2SO_WS	U1TXD		14	1, 12
GPIO4_U	GPIO4	CLK_XTAL			16	12
GPIO5_U	GPIO5	CLK_RTC			24	12

The following are the keys to some of the values in the table:

- Devices column
 - 1=ESP-1
 - 12=ESP-12

Here are the GPIO pins by mapping:

GPIO	Pin Name	NodeMCU	Notes	Risk
GPIO0	GPIO0_U	D3	Pin controls state of ESP8266 at boot. Caution when used as an output pin.	Red
GPIO1	U0TXD_U	D10	Pin is commonly used for flashing the device.	Orange
GPIO2	GPIO2_U	D4	Used for UART1 output and, as such, is likely to be used during development time for debugging. Written to when flashed with new firmware.	Orange
GPIO3	U0RXD_U	D9	Pin is commonly used for flashing the device.	Orange
GPIO4	GPIO4_U	D2	Only use is as a GPIO.	Green
GPIO5	GPIO5_U	D1	Only use is as a GPIO.	Green
GPIO6	SD_CLK_U		Not exposed on current devices.	Red
GPIO7	SD_DATA0_U		Not exposed on current devices.	Red
GPIO8	SD_DATA1_U		Not exposed on current devices.	Red
GPIO9	SD_DATA2_U	SD2	Not exposed on current devices.	Red
GPIO10	SD_DATA3_U	SD3	Not exposed on current devices.	Red
GPIO11	SD_CMD_U		Not exposed on current devices.	Red
GPIO12	MTDI_U	D6		Green
GPIO13	MTCK_U	D7		Green
GPIO14	MTMS_U	D5		Green
GPIO15	MTDO_U	D8	Used to control UART0 RTS and hence may have an influence on firmware flashing since the firmware data arrives via UART0.	Yellow
GPIO16	???	D0	???	Yellow

The maximum output current from a GPIO pin is only 12mA.

Given a choice, if you are using GPIO0, use it as an input pin as opposed to an output pin. The reason for this is that when you are developing solutions, you need to bring GPIO0 low to place the ESP8266 into flash mode where it reads new programs from the UART. This means that you will be changing the input signal to GPIO0. If you use the pin as an output, there is the possibility that when you change your wiring to bring it low or press a button to bring it low, if the signal is high at that time, you will short the circuit.

However, if the pin is input then that won't be a problem. Ideally, avoid using GPIO0 altogether and leave it specifically for bootstrapping the device in different modes.

See also:

- `PIN_FUNC_SELECT`
- `GPIO_OUTPUT_SET`
- `GPIO_DIS_OUTPUT`
- `GPIO_INPUT_GET`
- `gpio_output_set`
- `gpio_input_get`

Pullup and pull down settings

We commonly think of an input GPIO pin as having either a high or low signal supplied to it. This means that it is connected to +ve or ground. But what if it is connected to neither? In this case, the pin is considered to be in a floating state. There are times where we wish to define an unconnected pin as being high or low. An unconnected pin that is to be considered high is termed "pulled up" while an unconnected pin that is to be considered low is termed "pulldown". This comes from the physical hardware practice of attaching resistors to pull up or pull down the signal when it otherwise would be floating.

In the ESP8266 SDK, we can define a GPIO as being pulled-up by using the macro called `PIN_PULLUP_EN` and we can define the GPIO as no longer being pulled up using the macro `PIN_PULLUP_DIS`.

GPIO Interrupt handling

If we consider that the signal on a pin can move from high to low or from low to high, such a change might be something our application would be interested in knowing. To determine when such a change happens, we can continually poll the value and detect a transition change. However this is not the best solution for a number of reasons. First, we have to busily perform working checking whether a value has changed. Secondly, there will be a latency from the time the event happens to the time when we check. Thirdly, it is possible to completely miss a signal change if the duration of the change is short. For example, if we check the value of a pin and find it high and then immediately after it goes low and then high again, the next time we poll we will still see the pin high and never have known that it was ever low for a short period.

The solution to all these problems is the notion of an interrupt. An interrupt is similar to your doorbell at your house. Without a door bell (or listening for someone knocking) you would have to periodically check to see if there is anyone at the door. This wastes your time for the majority of instances where there is no-one there and also makes sure that when there is someone there, you attend to them in a timely fashion.

In the land of ESP8266s, we can define an interrupt callback function that will be called when a pin changes its signal value. We can also define what constitutes a reason for invoking the callback. We can configure the callback handler (technically called an interrupt handler) on a pin by pin basis.

First, let us consider the interrupt callback function. This is registered with a call to `gpio_intr_handler_register()`.

We can enable or disable interrupt handling on a global level. A call to `ETS_GPIO_INTR_ENABLE` enables interrupt handling while a call to `ETS_GPIO_INTR_DISABLE` disables global interrupt handling.

To enable an interrupt for a specific pin, we use the function called `gpio_pin_intr_state_set()`. This allows us to set the reason that an interrupt might occur. The reasons include:

- Disable – don't call an interrupt on a signal change.
- PosEdge – Call the interrupt handler on a change from low to high.
- NegEdge – Call the interrupt handler on a change from high to low.
- AnyEdge – Call the interrupt handler on either a change from low to high or a change from high to low.
- Hi – Call the interrupt handler while the signal is high.
- Lo – Call the interrupt handler while the signal is low.

Notes: When I went to implement an interrupt handler in a project, I found that theory and practice didn't meet. As of now, the only story I have got working for an interrupt handler looks as follows:

```
static void intrHandlerCB(
    uint32 interruptMask, //!< A mask indicating which GPIOs have changed.
    void *arg             //!< Optional argument.
) {
    uint32 gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
    os_printf("status: 0x%x\n", gpio_status);
    gpio_intr_ack(interruptMask);
    int pin;
    for (pin=0; pin<16; pin++) {
        if ((interruptMask & (1<<pin)) != 0) {
            // Do something
            gpio_pin_intr_state_set(GPIO_ID_PIN(pin), GPIO_PIN_INTR_ANYEDGE);
        }
    }
}
```

See also:

- `gpio_intr_handler_register`
- `gpio_pin_intr_state_set`

- gpio_intr_pending
- gpio_intr_ack

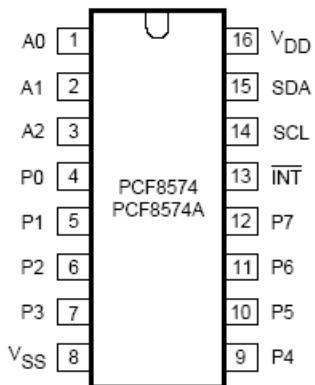
Expanding the number of available GPIOs

Although the ESP devices only have a limited number of GPIO pins, that needn't be a restriction for us. We have the capability to expand the number of GPIOs available to us through some relatively inexpensive integrated circuits. One of these is called the PCF8574. (The PFC8574A is the same but has a different set of addresses).

This is an I²C device and hence works over only two wires. Using this IC we supply a 3 bit address (000-111) that is used to select the slave address of the device. Since each address has 8 IOs and we can have up to 8 devices, this means a total of 64 additional pins.

It appears that the device will use a pull-up resistor for a high and bring the pin to ground for low. This means that if we want to use any of the pins for input, we should set their write mode to high first. This will allow either a high or low input signal to be detected. It would appear that if we set the output signal to be low and then fed a raw high signal into the device, we would have a short.

Here is the pin diagram for the device:



Here is a description of the pins:

Symbol	Pin	Description
A0-A2	1, 2, 3	Addressing
P0-P7	4, 5, 6, 7, 9, 10, 11, 12	Bi directional I/O
INT	13	Interrupt output
SCL	14	Serial Clock Line
SDA	15	Serial Data Line
V _{DD}	16	Supply Voltage (2.5V – 6V)
Vss (Ground)	8	Ground

The address that the slave device can be found upon is configurable via the A0–A2 pins.
It appears at the following address:

PCF8574

0	1	0	0	A2	A1	A0
---	---	---	---	----	----	----

PCF8574A

0	1	1	1	A2	A1	A0
---	---	---	---	----	----	----

The pins A0–A2 must not float.

This results in the following table:

A2	A1	A0	Address PCF8574	Address PCF8574A
0	0	0	0x20	0x38
0	0	1	0x21	0x39
0	1	0	0x22	0x3a
0	1	1	0x23	0x3b
1	0	0	0x24	0x3c
1	0	1	0x25	0x3d
1	1	0	0x26	0x3e
1	1	1	0x27	0x3f

Here is an example program that drives LEDs to create a Cylon effect.

```
#include <Wire.h>
#include <Ticker.h>
// SDA - Yellow - 4
// CLK - White - 5

#define SDA_PIN 4
```

```

#define CLK_PIN 5

Ticker ticker;
int counter = 0;
int dir = 1;

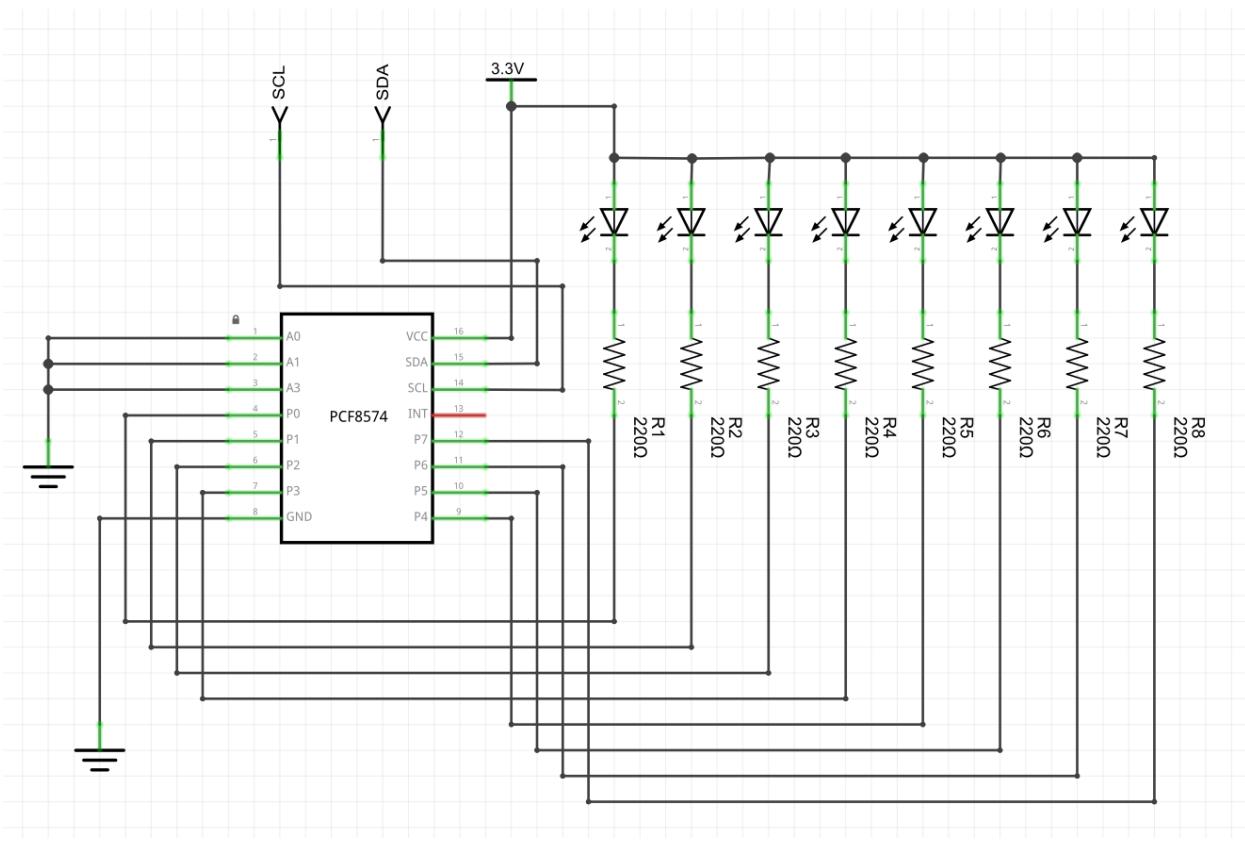
void timerCB() {
    Wire.beginTransmission(0x20);
    Wire.write(~((uint8_t)1<<counter));
    Wire.endTransmission();
    counter += dir;
    if (counter == 8) {
        counter = 6;
        dir = -1;
    } else if (counter == -1) {
        counter = 1;
        dir = 1;
    }
}

void setup()
{
    Wire.begin(SDA_PIN,CLK_PIN);
    ticker.attach(0.1, timerCB);
}

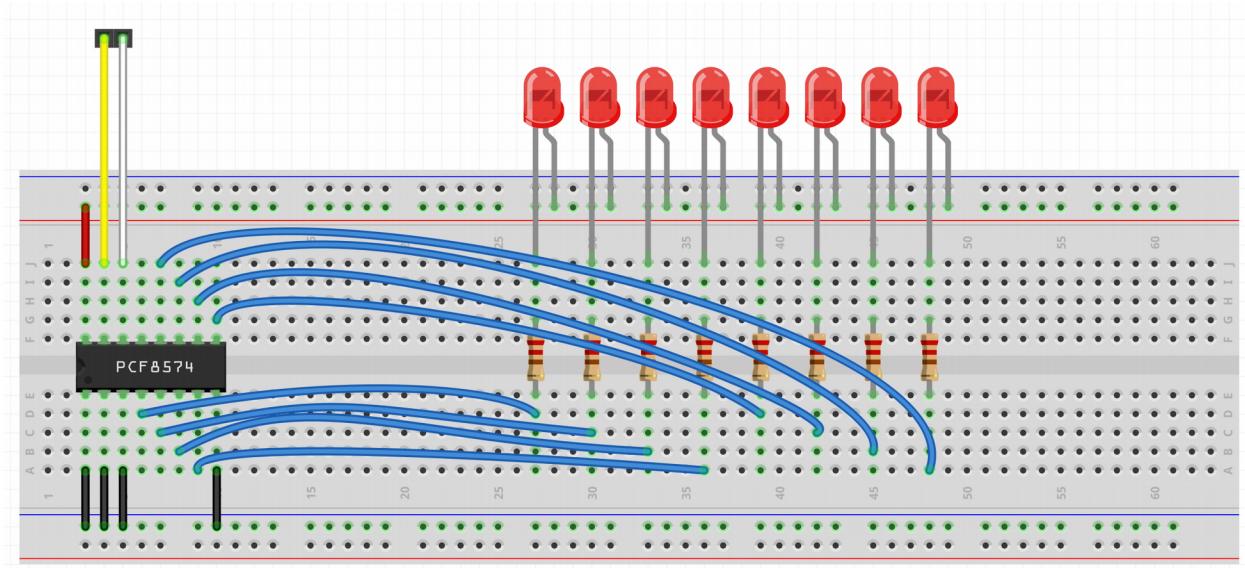
void loop()
{
}

```

The corresponding circuit is:



And on a breadboard:



See also:

- [A video tutorial on this topic](#)
- [A class for PCF8574 \(RobTillaart/Github\)](#)
- [8BIT IO EXPANDER \(PCF8574\)](#)

- skywodd – [PCF8574 Arduino library](#)
- [Datasheet](#) – NXP
- [Product page](#) – TI
- Working with I2C
-

ESP_PCF8574 C library

A class called `ESP_PCF8574` has been written to use the Arduino libraries. The library can be found on [Github](#).

It provides the following methods:

ESP_PCF8574.begin

Begin the PCF8574 control.

```
void begin(uint8_t address, uint8_t sda, uint8_t clk)
```

The `address` parameter is the I2C address of the PCF8574 ... typically `0x20 – 0x27`.

The `sda` and `clk` parameters are the pin numbers used for I2C SDA and CLK.

ESP_PCF8574.getBit

Retrieve the input of a given bit.

```
bool getBit(uint8_t bit)
```

If we think of the 8 GPIOs supplied by the PCF8574 as being 8 bits of data, this method retrieves the value of data on a given input pin.

ESP_PCF8574.getByte

Retrieve all 8 bits of input.

```
uint8_t getByte()
```

If we think of the 8 GPIOs supplied by the PCF8574 as being 8 bits of data, this method retrieves the value of all the inputs.

ESP_PCF8574.setBit

Set the value of a given output pin.

```
void setBit(uint8_t bit, bool value)
```

Set the value of a given output pin.

ESP_PCF8574.setByte

Set the value of all output pins.

```
void setByte(uint8_t value)
```

Set the value of all output pins.

PCF8574 JavaScript Library

A JavaScript library has been built for interfacing the PCF8574 from a JavaScript application.

Invoking from JavaScript is simplicity itself. Since the PCF8574 is merely a simple I2C board, using the I2C interface is all that is needed.

For example:

```
var sda = NodeMCU.D1; // Yellow
var scl = NodeMCU.D2; // White
I2C1.setup({scl: scl, sda: sda});
I2C1.writeTo(0x20, value);
```

Assuming that the address for our PCF8574 is 0x20.

Working with I2C

The I2C interface is a serial interface technology for accessing devices. It has two signal lines called SDA (Data) and SCL (Clock). The ESP8266 can act as a master and the devices connected downstream act as slaves. Up to 127 distinct slaves are theoretically attachable. Each slave device has a unique address and the master decides which slave is to receive data or be allowed to speak next.

All the slaves connected use an "open drain" connection to the bus. This means that when they connect, their attachment is either open circuit or ground as an output. Because of this it is impossible for there to be an electrical conflict as it would be impossible for one device to assert a high signal while another tried to assert a low signal. The presence of a logical high signal occurs when the current slave device goes open circuit. This means that we need pull-up resistors on the lines such that when no-one is actively asserting a low signal, they are pulled-up to a logical high signal. A resistor value of 4.7KΩ is recommended.

The start of a transmission is indicated when the SCL is left high and SDA is pulled low. This informs all the slave devices that an address is about to be issued. When the address is seen by all the slaves, only one of them should match and the other devices ignore the request.

The address of a slave follows the initial start indication and is comprised of 7 bits with most significant bit first. Following the 7 bit address is a final 8th bit that indicates whether this is a read or a write request. A value of 1 indicates a read from the slave while a value of 0 indicates a write from the master.

Immediately after the 8 bits of address, comes the acknowledgment bit. This bit is **not** transmitted from the master to the slave but is instead transmitted from the slave to the master. Be sure you understand that as when looking at diagrams showing data on the SDA wire, those diagrams typically do not show the origination of the data, only their sequence. The turn around time from the last bit of the 8 bit address/direction data sent from the master to the acknowledgment bit sent from the slave happens without missing any clock cycles so has to be fast. A value of 0 in the acknowledgment states that the slave will process or respond. A value of 1 in the acknowledgment states that no-one is responding or the slave is not present.

Following this addressing frame comes the data frame or frames. For a master write request, the master will send 8 bits of data and expect a single bit of acknowledgment. For a read from the slave, the slave will send 9 bits of data (8 data bits and an acknowledgment).

The master will finally send an end of communication (or stop) indication which is a transition to high on the clock with NO corresponding transition to low and **then** a transition from low to high on the SDA line.

To use I2C, we first transmit a start request using `i2c_master_start()`. We follow this by the address and the read/write flag. Since an address is 7 bits and the read/write flag is one bit, this totals 8 bits and hence we can write a byte:

```
i2c_master_writeByte((address << 1) | readOrWrite);
```

Next we can read and check the acknowledgment flag using:

```
if (i2c_master_checkAck() == true) {  
    ...  
}
```

And from here we can either end or execute the next part of read or write.

When we wish to end, we execute `i2c_master_stop()`.

We can send an address query for each of the possible device addresses and see if we get an acknowledgment. If we do, then we have an I2C device at that address. This can be used to create a map of devices.

Here is an example application that does just that:

```
#include <ets_sys.h>  
#include <osapi.h>  
#include <os_type.h>  
#include <gpio.h>
```

```

#include <user_interface.h>
#include <espconn.h>
#include <mem.h>
#include "driver/uart.h"
#include "driver/i2c_master.h"

void user_rf_pre_init(void) {
}

os_timer_t scanTimer;

void scanTimerCB(void *pArg) {
    os_printf("--- Examining ---\n");
    uint8_t i;
    for (i=1; i<127; i++) {
        i2c_master_start();
        i2c_master_writeByte(i << 1);

        if (i2c_master_checkAck()) {
            os_printf("Found device at: 0x%2x\n", i);
        }
        i2c_master_stop();
    }
    os_printf("Done!\n");
}

} // End of timerCallback

void init() {
    i2c_master_gpio_init();
    os_timer_setfn(&scanTimer, scanTimerCB, NULL);
    os_timer_arm(&scanTimer, 10000, 1);
}

void user_init(void) {
    uart_init(BIT_RATE_115200, BIT_RATE_115200);
    system_init_done_cb(init);
} // End of user_init

```

See also:

- [I2C Bus](#)
- Sparkfun – [Tutorial: I2C](#)
- I2C Master APIs
- wget http://<IP address> --quiet --output-document=-

Working with SPI - Serial Peripheral Interface

SPI is a serial protocol used to communicate between masters and slaves. All slaves connect to the same bus but only the slave with its `ss` pin low is allowed to transmit. SPI is a full duplex protocol. What this means is that while data is being pushed out from the master to the slave, the slave is simultaneously sending data back to the master. The `MOSI` pin contains the serial data from the master to the slave while the `MISO` pin contains the data from the slave to the master.

Typically three pins:

- MISO – **M**aster **I**n, **S**lave **O**ut – Sending data to the master from the slave
- MOSI – **M**aster **O**ut, **S**lave **I**n – Sending data to the slave from the master
- SCK (SCLK) – Serial Clock – Synchronizes data from the master/slave relationship

There is also an additional signal:

- SS (CSN (Chip Select NOT), NSS) – **S**lave **S**elect – Used to enable/disable the slave so that there can be multiple slaves. SS low means slave is the active slave.

Since this is a serial protocol and we will receive data in bytes, we need to be cognizant of whether or not data will arrive LSB first or MSB first. There will be an option to control this.

For the clock, we will be latching data and we will need to know what edges and settings are important. There will be a clock mode option to control this. In SPI there are two attributes called phase and polarity. Phase (CPHA) is whether we are latching data on high or low and Polarity (CPOL) is whether high or low means that the clock is idle.

CPOL=0 means clock is default low, CPOL=1 means clock is default high.

When CPOL=0, then the following are the values for CPHA

CPHA=0 means data is captured on clock rising edge, CPHA=1 means data is captured on clock falling edge.

When CPOL=1, then the following are the values for CPHA

CPHA=0 means data is captured on clock falling edge, CPHA=1 means data is captured on clock rising edge.

SPI wraps these two flags into four defined and named modes:

Mode	Clock Polarity – CPOL	Clock Phase – CPHA
SPI_MODE0	0 (Clock default low)	0
SPI_MODE1	0 (Clock default low)	1
SPI_MODE2	1 (Clock default high)	0
SPI_MODE3	1 (Clock default high)	1

Also for the clock, what speed are we will need to know what speed the data is to be moved. There will be a clock control speed option to control this.

Hardware SPI

The ESP8266 has hardware SPI support that is controlled by setting registers and using SDK supplied macros. There are certain physical pins that are reserved for the hardware SPI. These are:

GPIO	NodeMCU	Name	Function
GPIO12	D6	HMISO	MISO
GPIO13	D7	HMOXI	MOSI
GPIO14	D5	HSCLK	CLK
GPIO15	D8	HCS	CS

To begin with, we will think about the clock. This is the speed at which the ESP8266 toggles the communication stream with the SPI partner. We can set this to be the same speed as the processor clock (80MHz) by writing to a peripheral register.

```
WRITE_PERI_REG(SPI_CLOCK(HSPI), SPI_CLK_EQU_SYSCLK)
```

This instructs the ESP8266 that the hardware SPI clock speed should be equal to the system clock speed which is normally 80MHz.

If this speed is too fast, we can change the clock speed with a divisor called `SPI_CLKDIV_PRE`. This divides the clock speed by a value. If you want to divide by X then supply a value of X-1.

For example, to divide the clock speed by 10, provide a value of 9.

Now we have a raw clock speed. However, there is more. There is a setting called `SPI_CLKCNT_N` that defines how many of these raw clock ticks should constitute one SPI clock cycle. Remember that an SPI clock cycle starts high, transitions to low and then returns to high again. The `SPI_CLKCNT_N` defines how many raw clock cycles correspond to SPI clock cycle. For example, specifying a value of 9 (desired value is n+1) means that 10 raw clock cycles will be one SPI clock cycle. Realize that this is a divisor of the raw clock cycle.

There are other settings that effect clock and these are called `SPI_CLKCNT_H` and `SPI_CLK_CNT_L`. Taken together, these define the number of clock cycles that the clock is high vs low. This shapes the clock signal. The values here are encoded to mean the number of raw clock cycles where the SPI clock is high vs low. The difference between them gives the result. For example, setting `SPI_CLKCNT_H=6` and `SPI_CLKCNT_L=1` results in a difference of 5 meaning that 5 raw clock cycles will be high and the remaining clock cycles (5) will be low.

Function	Bits	Mask	Shift
SPI_CLK_EQU_SYSCLK	31	N/A	SPI_CLK_EQU_SYSCLK
SPI_CLKDIV_PRE	30:18	SPI_CLKDIV_PRE	SPI_CLKDIV_PRE_S
SPI_CLKCNT_N	17:12	SPI_CLKCNT_N	SPI_CLKCNT_N_S
SPI_CLKCNT_H	11:6	SPI_CLKCNT_H	SPI_CLKCNT_H_S
SPI_CLKCNT_L	5:0	SPI_CLKCNT_L	SPI_CLKCNT_L_S

There are three registers that control data input and output. These are called SPI_USER, SPI_USER1 and SPI_USER2.

SPI_USER contains the following flags:

- SPI_CS_SETUP – When enabled, the chip select line is pulled low a few cycles before transmission allowing the partner SPI device to become ready for a transmission.
- SPI_CS_HOLD – When enabled, the chip select line remains low for a few cycles after transmission allowing the partner SPI device to continue for a little.

Function	Bit
SPI_USR_COMMAND	31
SPI_USR_ADDR	30
SPI_USR_DUMMY	29
SPI_USR_MISO	28
SPI_USR_MOSI	27
SPI_USR_MOSI_HIGHTPART	25
SPI_USR_MISO_HIGHTPART	24
SPI_SIO	16
SPI_FWRITE_QIO	15
SPI_FWRITE_DIO	14
SPI_FWRITE_QUAD	13
SPI_FWRITE_DUAL	12
SPI_WR_BYTE_ORDER	11
SPI_RD_BYTE_ORDER	10
SPI_CK_OUT_EDGE	7
SPI_CK_I_EDGE	6
SPI_CS_SETUP	5
SPI_CS_HOLD	4
SPI_FLASH_MODE	2

`SPI_USER1` contains four settings for the number of bits in various SPI settings. These are:

- `SPI_USR_ADDR_BITLEN` – How many bits in the SPI address.
- `SPI_USR_MOSI_BITLEN` – How many bits in the MOSI data.
- `SPI_USR_MISO_BITLEN` – How many bits in the MISO data.
- `SPI_USR_DUMMY_CYCLELEN` – How many bits in the dummy cycles.

An additional register called `SPI_ADDR` contains the address.

Function	Bits	Mask	Shift
<code>SPI_USR_ADDR_BITLEN</code>	31:26	<code>SPI_USR_ADDR_BITLEN</code>	<code>SPI_USR_ADDR_BITLEN_S</code>
<code>SPI_USR_MOSI_BITLEN</code>	25:17	<code>SPI_USR_MOSI_BITLEN</code>	<code>SPI_USR_MOSI_BITLEN_S</code>
<code>SPI_USR_MISO_BITLEN</code>	16:8	<code>SPI_USR_MISO_BITLEN</code>	<code>SPI_USR_MISO_BITLEN_S</code>
<code>SPI_USR_DUMMY_CYCLELEN</code>	7:0	<code>SPI_USR_DUMMY_CYCLELEN</code>	<code>SPI_USR_DUMMY_CYCLELEN_S</code>

Function	Bits	Mask	Shift
<code>SPI_USR_COMMAND_BITLEN</code>	31:28	<code>SPI_USR_COMMAND_BITLEN</code>	<code>SPI_USR_COMMAND_BITLEN_S</code>
<code>SPI_USR_COMMAND_VALUE</code>	15:0	<code>SPI_USR_COMMAND_VALUE</code>	<code>SPI_USR_COMMAND_VALUE_S</code>

The details of these registers and macros defining constants can be found in the `spi_register.h` file supplied in the drivers include directory.

See also:

- [Hardware SPI Clock Registers](#)
- [Hardware SPI \(HSPI\) Command & Data Registers](#)
- Wikipedia – [Serial Peripheral Interface Bus](#)

[The MetalPhreak/ESP8266_SPI_Driver](#)

Interacting with hardware SPI can be challenging. Fortunately, an open source project on GitHub has provided an easy to use solution to the problem. The project is known as `MetalPhreak/ESP8266_SPI_Driver` which is composed of a C source file and a couple of header files.

The source exposes the following functions:

- `void spi_init(uint8 spi_no)`
- `void spi_init_gpio(uint8 spi_no, uint8 sysclk_as_spiclk)`
- `void spi_clock(uint8 spi_no,
 uint16 prediv,
 uint8 cntdiv)`

To calculate the effective clock rate, take the CPU frequency (80MHz) and then divide it by preDiv. This gives a base frequency. Next we divide that by

- void spi_tx_byte_order(uint8 spi_no, uint8 byte_order)
- void spi_rx_byte_order(uint8 spi_no, uint8 byte_order)
- uint32 spi_transaction(uint8 spi_no,
 uint8 cmd_bits,
 uint16 cmd_data,
 uint32 addr_bits,
 uint32 dout_bits,
 uint8 dout_data,
 uint32 din_bits,
 uint32 dummy_bits)

In addition, the following macros are also provided:

- spi_busy(spi_no)
- spi_txd(spi_no, bits, data)
- spi_tx8(spi_no, data)
- spi_tx16(spi_no, data)
- spi_tx32(spi_no, data)
- spi_rxd(spi_no, bits)
- spi_rx8(spi_no)
- spi_rx16(spi_no)
- spi_rx32(spi_no)

In the previous functions, SPI and HSPI are the allowable values for spi_no.

See also:

- GitHub: [MetalPhreak/ESP8266_SPI_Driver](#)

Working with serial

ESP8266

There are two UARTs in the system known as UART0 and UART1. UART0 has its own dedicated TX and RX pins while UART1 is multiplexed with GPIO2. UART1 is output only and hence only has a TX line.

The serial interface to the ESP8266 can be initialized with a call to the function `uart_init()`.

For example

```
uart_init(BIT_RATE_115200, BIT_RATE_115200);
```

To write a string to the serial port, we can then use `os_printf()`. This has the same format as `printf` but writes to the serial port.

In order to work with UART, you must include the `uart.c`, `uart.h` and `uart_register.h` files from `examples/driver_lib`. In your application, you must then include `"driver/uart.h"`.

To transmit data using UART0, we have the function called `uart0_tx_buffer()` which accepts a pointer to data and a length and transmits it.

Within the SDK there is a transmission buffer. Because UART transmission is typically a slow operation, applications that wish to transmit data have their data stored in the TX buffer which is then drained by transmission over time. Data written to the UART is assured to be written in the order in which it was supplied. Should the TX buffer become full, no new data can be accepted.

Similarly, data received by the ESP8266 over UART is placed in a receive buffer. The application running on the ESP8266 has to receive the data in a timely fashion. If the RX buffer becomes full there is no place to put new data and that new data will be discarded. In UART terms, both the TX and RX buffers are termed "FIFO" which means first-in-first-out. The buffers are 128 bytes each (128 bytes for TX and a second 128 byte buffer for RX).

To find out how many bytes are on the various queues, we have to go pretty low level. For example, the number of bytes on the TX queue is given by:

```
(READ_PERI_REG(UART_STATUS(uart_no)) >> UART_TXFIFO_CNT_S) & UART_TXFIFO_CNT;
```

while the number of bytes on the RX queue is given by:

```
(READ_PERI_REG(UART_STATUS(UART0)) >> UART_RXFIFO_CNT_S) & UART_RXFIFO_CNT;
```

See also:

- Connecting to the ESP8266
- USB to UART converters
- Error: Reference source not found
- Error: Reference source not found
- Error: Reference source not found
- `os_printf`

ESP8266 Task handling

Imagine we wish to have a task performed for us asynchronously. What we might want to do is post that we wish this to happen and then go on with our business. When we are done and have relinquished control back to the OS, we assume that the task will eventually start executing. This is the function provided by the task functions of the ESP8266. There are two functions of interest to us. The first is called `system_os_task()` which sets up a task processor.

When we wish to post that a task is eligible to start, we can use the second function called `system_os_post()` which posts a message.

The task function that we registered will then be "invoked" at some point after the post request and will be given the parameters supplied in the post. The priority identifies the relative priority of two posts that have been issued. The one with the highest priority will execute first.

It is important to note that only **three** priorities are allowed which are 0, 1 and 2 with 0 having the lowest priority. It is also important to note that there can only be **one** handler for each task registration by priority. So if we execute `system_os_task()` twice using the same priority in both cases, only the last one is remembered and will be executed when a task of that priority is posted.

With this background, what is the purpose of this function set? Why would we care about it?

Note: The Arduino library for ESP uses the priority 0 task set.

Here is a sample of a simple task handler.

```
void taskHandler(os_event_t *event) {
    switch(event->sig) {
        case 1:
            break;
        case 2:
            break;
    }
}

os_event_t *taskQueue;
taskQueue = (os_event_t *)malloc(sizeof(os_event_t) * TASK_QUEUE_LEN);
system_os_task(taskHandler, USER_TASK_PRIO_1 taskQueue, TASK_QUEUE_LEN);

system_os_post(USER_TASK_PRIO_1, 1, (os_param_t)"Hello");
```

See also:

- Error: Reference source not found
- Error: Reference source not found

Timers and time

Within our code, we may wish to delay for a period of time. We can use the `os_delay_us()` function to suspend processing for a given period measured in microseconds. There are 1000 microseconds in a millisecond and a 1000 milliseconds in a second.

We can configure a timer to be called on a periodic basis with a callback granularity of milliseconds. A data structure called `os_timer_t` holds the state of the timer.

We can define the user function to be called when the timer fires using the `os_timer_setfn()` function. Note that we can only set the callback function when the timer is disarmed.

When ready, we can arm the timer so that it starts ticking and fires when ready. We do this using the `os_timer_arm()` function.

The repeat flag indicates whether the timer should restart after it has fired.

We can suspend or cancel the firing of the timer using `os_timer_disarm()`.

Here is an example:

```
os_timer_t myTimer;

void timerCallback(void *pArg) {
    os_printf("Tick!");
} // End of timerCallback

void user_init(void) {
    uart_init(BIT_RATE_115200, BIT_RATE_115200);
    os_timer_setfn(&myTimer, timerCallback, NULL);
    os_timer_arm(&myTimer, 1000, 1);
} // End of user_init
```

Another aspect of working with time is time calculations and measurement. The function `system_get_time()` returns a 32 bit unsigned integer (unit32) value which is the microseconds since the device booted. This value will roll over after 71 minutes.

What if we need a granularity of time smaller than a microsecond? Hopefully you won't need this often ... however some solutions such as working with the WS2812 LEDs do in fact require ultra fine precision.

One possible solution is to drop down to assembly language programming. There is a special register managed by the ESP8266 which is called "ccount" which measures cycles of operation. The value of this is incremented each time an operational cycle completes.

The value of this register can be retrieved with the following C code:

```

static inline uint32_t getCycleCount() {
    uint32_t ccount;
    __asm__ __volatile__("rsr %0,ccount":"=a" (ccount));
    return ccount;
}

```

This fragment uses the in-line assembler to transform an assembly language statement into its corresponding operational instruction.

See also:

- Error: Reference source not found
- os_timer_arm
- os_timer_disarm
- os_timer_setfn

Working with memory

When working in C, you have to think in terms of computer memory. The amount of available RAM is likely to be less than 45KBytes.

We can allocate memory using `os_malloc()` or `os_zalloc()`. The first function allocates and returns memory and the second does exactly the same but zeros the memory before returning. When your logic no longer needs the memory, it can return it back to the heap with `os_free()`. To determine how much heap size is available, we can call `system_get_free_heap_size()`. Once we have the memory pointer, we can start to manipulate it through a series of memory commands. The `os_memset()` command will set a block of memory to a specific value. The `os_memcpy()` will copy a block of memory to a different block. The `os_bzero()` function will set the values of a block of memory to zero.

Memory on the ESP8266 is made up of a number of components. We have:

- data
- rodata
- bss
- heap

The values of these can be found through the `system_print_meminfo()` function.

When the ESP8266 needs to read an instruction from memory in order to execute it, that instruction can come from one of two places. The instruction can be in flash memory (also called `irom`) or it can be in RAM (also called `iram`). It takes less time for the processor to retrieve the instruction from RAM than it does from flash. It is believed that an instruction fetch from flash takes four times longer than the same instruction

fetched from RAM. However, on the ESP8266 there is far less RAM than there is flash. What this means is that you are far more likely to run out of RAM way before you run out of flash. When writing normal applications, we shouldn't fixate on having instructions in RAM rather than flash for the performance benefit. The execution speeds of the ESP8266 are so fast that if the cost of retrieving an instruction from RAM is blindingly fast then retrieving an instruction from slower flash is **still** blindingly fast.

There are however certain classes of instructions that we might wish to place in RAM rather than flash. Examples of these are interrupt handlers where the time spent in these should always be as short as possible and also function that write to flash.

When we define C functions, we can add an attribute by the name of `ICACHE_FLASH_ATTR`. What this does is place this function in the flash memory address space as opposed to RAM. Specifically, flagging a function with `ICACHE_FLASH_ATTR` tags it as being in the `".irom0.text"` section of code.

Note: From a raw technical perspective, `ICACHE_FLASH_ATTR` is a `#define` that maps to:

```
attribute__((section(".irom0.text")))
```

One of the areas we have not yet discussed is how memory is populated and used when an ESP8266 boots. There are two files commonly uploaded into flash. The first is at offset `0x00000` and this contains the data that will be loaded into RAM by the ROM based boot-loader. This data contains a series of sections and addresses in RAM of where the data will be loaded. The second binary file is commonly loaded into flash at `0x40000`. It contains the binary data of a section called `.irom0.text` which contains code. The RAM loaded code must match where this flash stored data is addressed.

For those interested in low level details, the format of the memory written to the flash files has been decoded. It is believed that the low memory looks like:

```
struct rom_header {
    uint8 magic;
    uint8 sect_count;
    uint8 flags1;
    uint8 flags2;
    uint32 entry_addr;
};
```

The `magic` property is a constant of `0xe9`. The `sect_count` contains the number of sections to load into ram. This will not include the `irom0.text` section. The `flags1` and `flags2` are used to indicate the flash size, clock rate and IO mode. Finally, `entry_addr` is the entry point to start executing user supplied code.

Immediately following the header, are section entries (there should be `sect_count` of them) where each entry is:

```

struct sect_header {
    uint32 address;
    uint32 length;
}

```

The addresses should be within the `.iram` address space starting at `0x40100000`. After each of the headers is a check-sum value. A check-sum is calculated from each of the sections and validated that it matches what is supposed to be present.

The whole of flash is also mapped to address space `0x40200000`.

The second file contains the `irom.text` section data. By default, the address space for this section is `0x40240000` which means that it should be written to flash at `0x40000`.

An excellent map of ESP8266 memory is being maintained on the ESP8266 [Wiki](#).

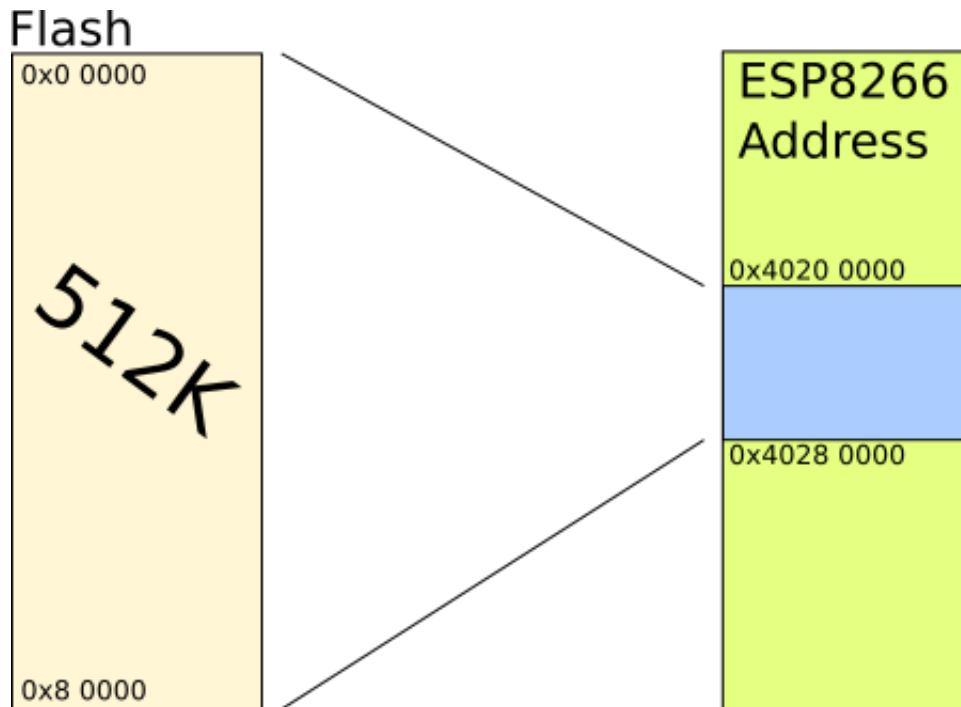
The core essence of it is here:

Address	Size	Notes
<code>0x0000 0000 <</code>		Can't be read.
<code>0x2000 0000 <</code>		Unmapped
<code>0x3FF0 0000 <</code>		Memory mapped I/O.
<code>0x3FF1 0000 <</code>		Unmapped.
<code>0x3FFE 8000 <</code>	<code>80K – 81920 (0x14000)</code>	User data RAM (dram)
<code>0x3FFF C000 <</code>	<code>16K – 16384 (0x4000)</code>	ETS system data RAM.
<code>0x4000 0000 <</code>		Internal ROM
<code>0x4010 0000 <</code>	<code>32K – 32768 (0x8000)</code>	Instruction RAM (iram/sram)
<code>0x4010 8000 <</code>		Unmapped or unknown
<code>0x4020 0000 <</code>	<code>Max 1024K (1M) – 1048576 (0x100000)</code>	SPI Flash
<code>0x4030 0000 <</code>		Unmapped or unknown

Now comes a discussion that took me a long, long time to comprehend. It is the relationship between the flash memory and the address space of the ESP8266. If we examine the previous table, we seem to see that the Flash memory is mapped into the ESP8266 address space at address `0x4020 0000`. That is key and vital to understand. If we read from this address onwards, we are actually reading from flash memory.

Let us think about the ESP-12 which has 512K of flash memory. In hex, this is `0x80000` bytes. This means an address range of `0x4020 0000` to `0x4027 FFFF`. Now let us consider the ESP flashing tools such as “`esptool`”. This also accepts an address into which to load flash ... but how does that address relate to run-time address space of the

ESP8266? The answer is that if we write to address `0x0 0000` of flash, at run-time it will appear at `0x4020 0000`. So in effect what we have is the following:



Now we answer one more puzzle. When we link together the object files and produce a binary image ... part of the role of the linker is to come up with the final address layout where the executable will finally reside. In practice, when we run the linker, we supply a linker control file called "`eagle.app.v6.ld`". This is supplied by Espressif. If we look in the `default` file, we find the following at the start of the file:

```
MEMORY
{
    dport0_0_seg : org = 0x3FF00000, len = 0x10
    dram0_0_seg : org = 0x3FFE8000, len = 0x14000
    iram1_0_seg : org = 0x40100000, len = 0x8000
    irom0_0_seg : org = 0x40240000, len = 0x32000
}
```

Now ... look at this closely ... because it took me forever to understand what I am about to tell you. Look at the address at where the `irom0` segment begins. It starts at `0x4024 0000` ... this is 256K **into** the 512K address range! Putting this another way, our executables can't use the whole address with these default settings. Why did Espressif set this? The answer is believed to be because they want to provide Over The Air (OTA) upgrade capability and wish to allow you to have a running copy of your app **and** an in-flight new copy be loaded. This means that effectively, 256K for the current version and 256K for the new version being loaded. If we tried to use the whole address space and during a refresh something went wrong, we have nothing to fall back to.

If you, like me, wanted to use the whole address space, the answer is simple. Change the corresponding entry in the "eagle.app.v6.ld" file.

One last wrinkle. It is believed that the last 16K of flash should be reserved for Espressif SDK storage for things like the last used SSID and password. Don't assume that you can use that range.

By default, the following items in an ELF file go to the 0x0 0000 flash location

- .text
- .data
- .rodata
- .iram0.text

The following sections go to the 0x4 0000 flash location

- .text

See also:

- os_memset
- os_memcpy
- os_memcmp
- os_malloc
- os_zalloc
- os_free
- Error: Reference source not found
- Error: Reference source not found
- [The ESP8266 Boot Process](#)
- esptool.py
- esptool-ck
- gen_appbin.py

Working with flash memory

Flash memory provides a non-volatile repository of information that survives a power cycle of the device.

Data contained within flash is stored in units of sectors which are 4096 bytes in size. To write data we can call `spi_flash_write`. To read data we call `spi_flash_read`.

Since writing to flash is performed in units of 4096 bytes, we can not change a single byte by just over-writing it, instead we must retrieve the whole sector, erase the sector and then write back the sector with the changed content. This can take some time to complete and because of this, we may find that a failure is more likely to occur (eg. a loss of power). If a failure occurs after we have erased a sector or during the re-write of the sector, it should immediately become apparent that we will result in an overall corruption of data.

Data reads and writes have to be 4 bytes aligned within flash.

The ESP8266 has to be instructed about the size of the flash memory available to it. Attempting to use flash memory addresses that differ from the expected size of flash memory available can result in unexpected results.

When using `esptool.py`, the `--flash_size` flag can be supplied. For `esptool-ck`, the corresponding flag is `-bz`.

See also:

- `spi_flash_get_id`
- `spi_flash_erase_sector`
- `spi_flash_read`
- `spi_flash_set_read_func`
- `system_param_save_with_protect`
- Cesanta: [ESP8266, flash and alignment](#)
- `system_param_load`

Pulse Width Modulation - PWM

The idea behind pulse width modulation is that we can think of regular pulses of output signals as encoding information in how long the signal is kept high. Let us imagine that we have a period of 1HZ (one thing per second). Now let us assume that we raise the output voltage to a level of 1 for $\frac{1}{2}$ of a second at the start of the period. This would give us a square wave which starts high, lasts for 500 milliseconds and then drops low for the next 500 milliseconds. This repeats on into the future. The duration that the pulse is high relative to the period allows us to encode an analog value onto digital signals. If the pulse is 100% high for the period then the encoded value would be 1.0. If the pulse is 100% low for the period, then the encoded value would be 0.0. If the pulse is on for "n" milliseconds (where n is less than 1000), then the encoded value would be $n/1000$.

Typically, the length of a period is not a second but much, much smaller allowing us to output many differing values very quickly. The ratio of the on signal to the period is called the "duty cycle". This encoding technique is called "Pulse Width Modulation" or "PWM".

There are a variety of purposes for PWM. Some are output data encoders. One commonly seen purpose is to control the brightness of an LED. If we apply maximum voltage to an LED, it is maximally bright. If we apply $\frac{1}{2}$ the voltage, it is about $\frac{1}{2}$ the brightness. By applying a fast period PWM signal to the input of an LED, the duty cycle becomes the brightness of the LED. The way this works is that either full voltage or no voltage is applied to the LED but because the period is so short, the "average" voltage over time follows the duty cycle and even though the LED is flickering on or off, it is so fast that our eyes can't detect it and all we see is the apparent brightness change.

For the ESP8266, the period of the PWM can range from 1 millisecond to 10 milliseconds. This is a frequency of 1KHz to 100Hz. The resolution of the duty cycle is down to 45 nanoseconds which is 14 bits of resolution data. The device provides support for up to 8 PWM channels where each channel can be associated with its own pin and duty cycle. The period is the same for all PWM channels.

To start using the ESP8266 PWM support, a call to `pwm_init()` is needed which sets up which pins are to be used for PWM and for which channels. A call to this function also sets up an initial period and duty cycle. A call to `pwm_start()` can then be made to start the PWM outputs. The period of PWM as a whole and duty cycles for each channel can be changed using the `pwm_set_period()` and `pwm_set_duty()` functions.

The ESP8266 PWM functions utilize the hardware timer. As such you can have PWM support or utilize the hardware timer for your uses ... but not both.

To utilize the ESP8266 SDK PWM functions, you must link your application with `libpwm.a`.

See also:

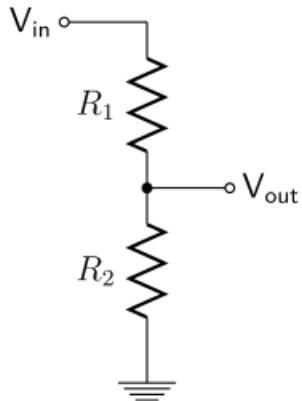
- Wikipedia: [Pulse-width modulation](#)
- [Espressif Sample](#)
- `pwm_init`
- `pwm_start`
- `pwm_set_duty`
- `pwm_get_duty`
- `pwm_set_period`
- `pwm_get_period`

Analog to digital conversion

Analog to digital conversion is the ability to read a voltage level from a pin between 0 and some maximum value and convert that analog voltage into a digital representation. Varying the voltage applied to the pin will change the value read. The ESP8266 has an analog to digital converter built into it with a resolution of 1024 distinct values. What that means is that 0 volts will produce a digital value of 0 while the maximum voltage will produce a digital value of 1023 and voltage ranges between these will produce a correspondingly scaled digital value.

To read the digital value of the analog voltage, the function called `system_adc_read()` should be called. The pin on the physical ESP8266 from which the voltage is read is called `TOUT` and serves no other purpose.

The input range on the pin is from 0V to 1V. This implies that the input voltage to the ADC can not be the maximum voltage used to power the ESP8266 itself (3.3V). So we will need to use a voltage divider circuit.



The formula to map these out is:

$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

Since we know V_{out} is going to 1V and V_{in} is 3V and we choose R_2 to be 10K, we find:

$$R_1 = \frac{R_2 \cdot V_{in}}{V_{out}} - R_2$$

and for our values:

$$R_1 = \frac{10000 \cdot 3.3}{1.0} - 10000 = 23000$$

A common 22K resistor will work well.

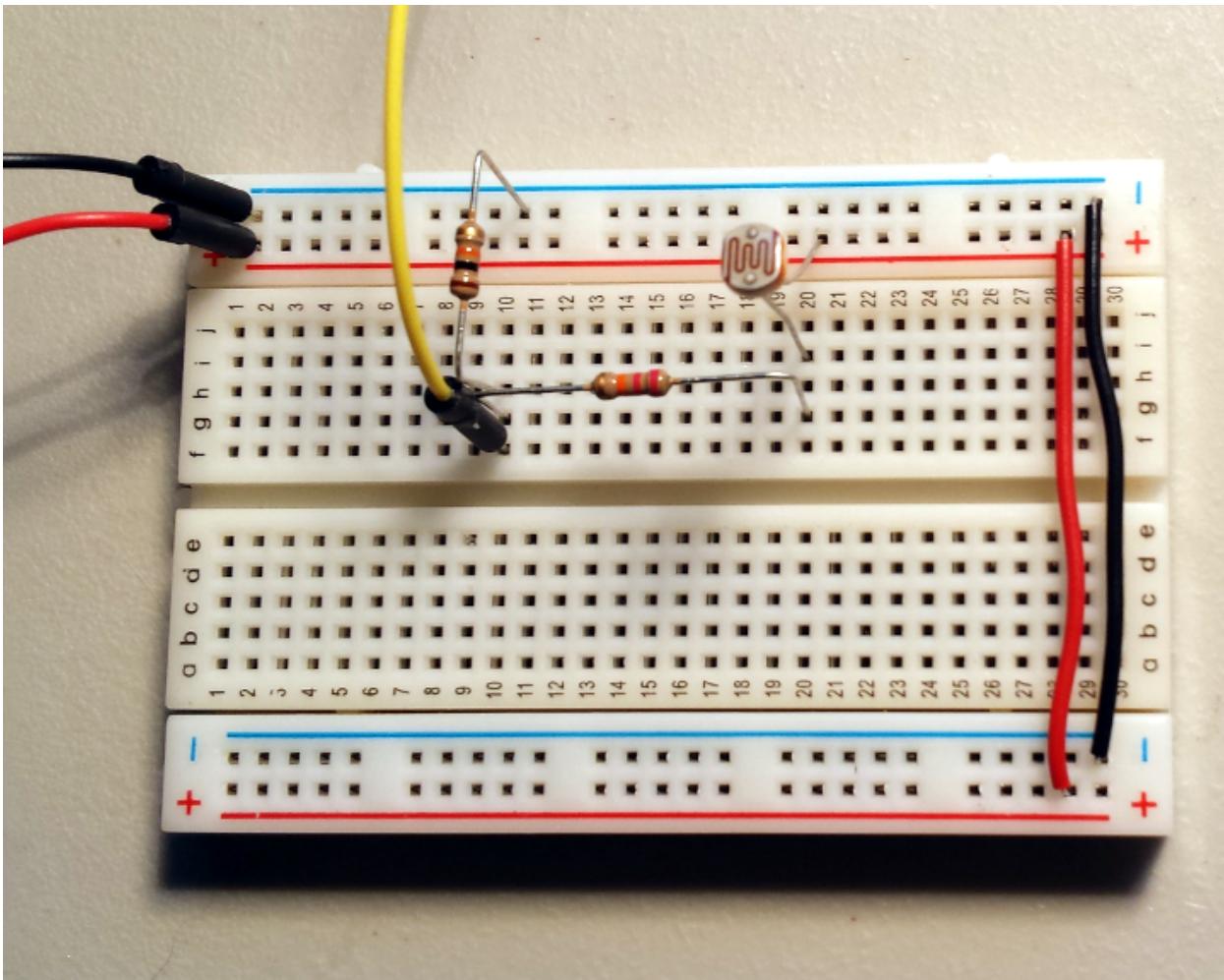
Here is an example. What this example does is print the value read from the ADC every second.

```
os_timer_t myTimer;

void timerCallback(void *pArg) {
    uint16 adcValue = system_adc_read();
    os_printf("adc = %d\n", adcValue);
} // End of timerCallback

void user_init(void) {
    uart_init(BIT_RATE_115200, BIT_RATE_115200);
    os_timer_setfn(&myTimer, timerCallback, NULL);
    os_timer_arm(&myTimer, 1000, 1);
} // End of user_init
```

If we build out on a breadboard a circuit which includes a light dependent resistor such as the following:



Then when we change the amount of light falling on the resistor, we can see the values change as data is written in the output log. This can be used to trigger an action (for example) when it becomes dark.

Open question: What is the sample rate of the ADC?

See also:

- Error: Reference source not found
- Wikipedia: [Voltage divider](#)

Sleep modes

If the ESP8266 device is constantly on, then it is constantly consuming current. If the power source is unlimited, then this need not necessarily be an issue however when running on batteries or other finite supply, we may need to minimize consumption. One way to achieve this is to suspend the operation of the device when not in use. When the device is suspended, the notion is that consumption will be reduced. There are three defined sleep modes. These are called modem-sleep, light-sleep, deep-sleep.

By looking at the following table we can get a sense of the abilities in each of these three modes:

Function	Modem	Light	Deep
WiFi	off	off	off
System Clock	on	off	off
Real Time Clock	on	on	on
CPU	on	pending	off
Current consumption	15mA	0.5mA	20µA

The modem-sleep can only be used when the ESP8266 is in station mode connected to an access point. The application of this mode is when the ESP8266 needs to still perform work but minimizes the amount of wireless transmissions.

The light-sleep mode is the same as modem-sleep but in this case the clocks will be suspended.

In deep-sleep mode, the device is really asleep. Neither CPU nor WiFi activities take place. The device is to all intents and purposes off ... with one exception ... it can wake up at a specified regular interval.

To enter deep sleep mode, we can call `system_deep_sleep()`. This can be supplied with a suspension time. The device will go to sleep and after the interval has elapsed, the device will wake up again. In addition to having a timer, we can also awake from a deep sleep by toggling the value of a signal on a pin.

We can control which mode the device is in by calling `wifi_set_sleep_type()`.

Watchdog timer

The ESP8266 is a single threaded processor. This means it can only do one thing at a time as there are no parallel threads that can be executed concurrently with each other. An implication of this is that when the OS gives control to your application, it doesn't get control back until such time as you explicitly relinquish it. However, this can cause problems. The ESP8266 is primarily a WiFi and TCP/IP device that expects to be able to receive and transmit data as well as respond to asynchronous events within a timely manner. As an example, if your ESP8266 device is connected to an access point and the access point wants to validate that you are still connected, it may transmit a packet to you and expect a response. You have no control over when that will happen. If your own application program has control over the execution at the time when the request arrives, that request will not be responded to until after you return control back to the OS. Meanwhile, the access point may be expecting a response within some predetermined time period and, if does not receive a reply within that interval, may

assume that you have disconnected. To prevent such circumstances your application code has to return control back to the OS in a timely manner. It is recommended that your code return control within 50 milliseconds of gaining control. If you take longer, you run the risk of requests to your device timing out.

If your own code fails to return control back to the OS, the OS must assume that things are going wrong. As such, it has a timer that we call the "watchdog". When control is given to your own code, the watchdog timer starts ticking. If you have not returned control back to the OS by the time the watchdog timer reaches zero, it takes matters into its own hands. Explicitly what it does is reboot the device. This may sound like a pretty drastic action but the thinking is that it is better to do this and hope that whatever was blocked is now unblocked than just sit there "dead".

Reports claim that the watchdog timer may be about 1 second (1000 milliseconds). However, in my tests, I find that the timer fires at about 3.2 seconds (3200 msecs).

A function called `system_soft_wdt_stop()` stops the watchdog timer ... or at least one of them. There appears to be **two** timers. One is in software, the other in hardware. This function stops the software timer. It can be restarted with `system_soft_wdt_restart()` ... however, a second timer called the hardware watchdog timer will fire after about 8 seconds and doesn't appear able to be trapped. A new call introduced in SDK 1.3 is called `system_soft_wdt_feed()`. Unfortunately the documentation on this is exceedingly poor. The best reports we have so far on what it does for is that when we call it, the watchdog time is reset to its starting point and starts ticking down again. I'm not quite sure of the value of this given that we already have API to stop and then restart the timer. Hopefully in the future we can gain additional knowledge to clear up any mysteries that may be lurking within.

See also:

- [system_soft_wdt_stop](#)
- [system_soft_wdt_restart](#)
- Error: Reference source not found

Yielding control

We have just been describing the notion of having to return control back to the OS in order for it to perform its house keeping duties. The way we do this is simply to return from the callback that the OS invoked us upon. If we think about how an ESP8266 program works, we will see that in order for us to relinquish control back to the OS, the OS must have called us in the first place. Therefore it makes sense for us to return control at a later point. However, if we return control, we (obviously) loose all the state (variables) that were in existence when we returned.

Now we get to introduce a concept called "yielding". The idea behind yielding is that instead of our application returning control back to the OS, what we can arrange to do is return to the OS while at the same time maintaining the context of where we are within the current execution. When the OS completes a round of housekeeping, what it can then do is "return" back to where ever we were when we requested a yield to occur.

This is tricky stuff to implement but fortunately Ivan Gorkhotov has achieved this task for us and we can leverage what he already built.

To use this:

1. Include cont.h
2. Create a global of "cont_t g_cont __attribute__ ((aligned (16)));"
3. In user_init called "cont_init(&g_cont);"
4. Register a system_os_task() processor.

When we wish to schedule some code for execution, post a task.

```
void esp_schedule() {
    system_os_post(TASK_PRIORITY, 0, 0);
}

static void taskHandler(os_event_t *events) {
    cont_run(&g_cont, someFunction);
    if(cont_check(&g_cont) != 0) {
        os_printf("Overflow detected\r\n");
        abort();
    }
}
```

Security

The ESP8266 has the ability to store the password used to connect to the access point in memory. This means that if one were to physically compromise the device (i.e. steal it) then they could, in principle, dump the flash memory and retrieve your password. You could choose not to cache the password in the clear in flash but instead have your applications "decode" an encoded version that is saved in the flash memory ... this would prevent an obvious retrieval through a simple memory grab. The encoding scheme could be a simple XOR against a magic number (either hard-coded or your own MAC address).

Mapping from Arduino

Without argument, the Arduino has become the most successful microprocessor programming environment to-date. There are tons and tons of existing sketches in

existence and let us not forget about the wealth of libraries. Tools and utilities exist to compile and run Arduino sketches on ESP8266s. What if instead we wanted to port those Arduino sketches to native ESP8266 code? Can we find mappings between the Arduino APIs and the corresponding ESP8266 APIs?

Arduino	ESP8266
digitalWrite(pin, value)	GPIO_OUTPUT_SET(pin, value)
digitalRead(pin)	GPIO_INPUT_GET(pin)
delay(ms)	os_delay_us(ms * 1000) Note: ms <= 65535
delayMicroseconds(us)	os_delay_us(us)
millis()	system_get_time() / 1000

From a functional perspective, here are some comparisons between an Arduino and an ESP8266:

	ESP8266	Arduino (Uno)
GPIOs	17 (Fewer typically exposed)	14 (20 including analog)
Analog input	1	6
PWM channels	8	6
Clock speed	80MHz	16MHz
Processor	Tensilica	Atmel
SRAM	45KBytes	2KBytes
Flash	512Kb or more (separate)	32KB (on chip)
Operating Voltage	3.3V	5V
Max current per I/O	12mA	40mA
UART (hardware)	1 ½	1
Networking	Built-in	Separate
Documentation	Poor	Excellent
Maturity	Early	Mature

Note: Because the Arduino has no native networking, no further comparisons of network capability were included above. Do remember that, at this time, when one is using an ESP8266, the chances are high it is because you **need** network access.

Spiffs File System

The SPI Flash File System (SPIFFS) is a file system mechanism intended for embedded devices. An implementation is supplied with the FreeRTOS ESP8266 SDK.

What is the page size of ESP8266 flash? What is the block size of ESP8266 flash?

When a SPIFFS API call is made, a zero or positive response indicates success while a value < 0 indicates an error. The nature of the error can be retrieved through the `SPIFFS_errno()` call.

The SPIFFS implementation does not directly access the flash memory. Instead, a functional area called a hardware abstraction layer ("hal") provides this service. A SPIFFS integration requires that three functions be created that have the following signatures:

```
s32_t (*spiffs_read)(u32_t addr, u32_t size, u8_t *dst)  
s32_t (*spiffs_write)(u32_t addr, u32_t size, u8_t *src)  
s32_t (*spiffs_erase)(u32_t addr, u32_t size)
```

If they succeed, the return code should be `SPIFFS_OK(0)`. On an ESP8266, these will map to the SPI flash APIs.

The SPIFFS file system could be hierarchical in nature such that it contains both directories and files but it seems that in reality it is not. There is only one directory called the root. The root directory is `"/"`. To determine the members of a directory, we can open a directory for reading with the `SPIFFS_opendir()` API and, when we are finished, close the reading operation with a `SPIFFS_closedir()` API call. We can walk through the directory entries with calls to `SPIFFS_readdir()`.

For example:

```
spiffs_DIR spiffsDir;  
SPIFFS_opendir(&fs, "/", &spiffsDir);  
struct spiffs_dirent spiffsDirEnt;  
while(SPIFFS_readdir(&spiffsDir, &spiffsDirEnt) != 0) {  
    printf("Got a directory entry: %s\n", spiffsDirEnt.name);  
}  
SPIFFS_closedir(&spiffsDir);
```

To create a file, we can use the `SPIFFS_open()` API by supplying a `SPIFFS_CREAT` flag.

See also:

- Github: [pellepl/spiffs](https://github.com/pellepl/spiffs)

Partner TCP/IP APIs

If the ESP8266 can act as one end of a TCP/IP connection, something else has to act as the other (of course, there is nothing to prevent two ESP8266s from communicating between themselves). Here we look into some technologies that allow partners to interact with the ESP8266 over the TCP/IP protocol.

For the TCP/IP protocol, the programming API originally developed for the Unix platform and written in C was called "sockets". The notion of a socket is that it logically represents an endpoint of a network connection. A sender of data sends data through the socket and the receiver of data receives data through the socket. The implementation of the "socket" itself is provided by the libraries but the logical notion of the socket remains. You will find yourself working with an "instance" of a socket and you should think of it as an opaque data type that refers to a communication link.

Sockets remains the primary API and is present in the majority of languages. Here we discuss some of the variants for some of the more common languages.

TCP/IP Sockets

The sockets API is a programming interface for working with TCP/IP networking. It is probably the most familiar API for network programming. TCP/IP network flows come in two flavors ... connection oriented over TCP and datagram oriented over UDP. The sockets API provides distinct patterns of calls for both styles.

For TCP, a server is built by:

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

For a TCP client, we build by:

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

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

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

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

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

When we are creating a server side socket, we want it to listen for incoming connection requests. To do this, we need to tell the socket which TCP/IP port number it should be listening upon. On a given device, only one application at a time can be using any given port number. If we want to associate a port number with an application, such as our server application in this case, we perform a task called "binding" which binds (or assigns) the port number to the socket which in turn is owned by the application.

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

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

```
listen(socket, backlog)
```

The backlog is the number of connection requests that the run-time will listen for and accept before they are handed off to the application for handling. The way to think about this is imagine that you are the application and you can only do one thing at a time. For example, you can only be talking to one person at a time on the phone. Now imagine you have a secretary who is handling your incoming calls. When a call arrives

and you are not busy, the secretary hands off the call to you. Now imagine that you are busy. At that time, the secretary answers the phone and asks the caller to wait. When you free up, she hands you the waiting call. Now let us assume that you are still busy when yet another client calls. She also tells this caller to wait. We are starting to build a queue of callers. And this is where the backlog concept comes into play. The backlog instructs the run-time how many calls can be received and asked to wait. If more calls arrive than our backlog will allow, the run-time rejects the call immediately. Not only does this prevent run-away resource consumption at the server, it also can be used as an indication to the caller that it may be better served trying elsewhere.

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

The API call looks like:

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

The return from `accept()` is a new socket that represents the connection between the requesting client and the server. It is vital to realize that this is distinct from the server socket we created earlier which we bound to our server listening port. That socket is still alive and well and exists to continue to service further client connections. The newly returned socket is the connection for the conversation that was initiated by this single client. Like all TCP connections, the conversation is symmetric and bi-directional.

This means that there is now no longer the notion of a client and server ... both parties can send and receive as they would like at any time.

See also:

- Wikipedia – [Berkeley Sockets](#)
- [Beej's Guide to Network Programming](#)

Handling errors

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

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

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

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

Symbol	Value	Description
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Arg list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Device or resource busy

EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func
ERANGE	34	Math result not representable
EDEADLK	35	Resource deadlock would occur
ENAMETOOLONG	36	File name too long
ENOLCK	37	No record locks available
ENOSYS	38	Function not implemented
ENOTEMPTY	39	Directory not empty
ELOOP	40	Too many symbolic links encountered
EWOULDBLOCK EAGAIN	41	Operation would block
ENOMSG	42	No message of desired type
EIDRM	43	Identifier removed
ECHRNG	44	Channel number out of range
EL2NSYNC	45	Level 2 not synchronized
EL3HLT	46	Level 3 halted
EL3RST	47	Level 3 reset
ELNRNG	48	Link number out of range
EUNATCH	49	Protocol driver not attached
ENOCSI	50	No CSI structure available
EL2HLT	51	Level 2 halted
EBADE	52	Invalid exchange
EBADR	53	Invalid request descriptor

EXFULL	54	Exchange full
ENOANO	55	No anode
EBADRQC	56	Invalid request code
EBADSLT	57	Invalid slot
EBFONT	59	Bad font file format
ENOSTR	60	Device not a stream
ENODATA	61	No data available
ETIME	62	Timer expired
ENOSR	63	Out of streams resources
ENONET	64	Machine is not on the network
ENOPKG	65	Package not installed
EREMOTE	66	Object is remote
ENOLINK	67	Link has been severed
EADV	68	Advertise error
ESRMNT	69	Srmount error
ECOMM	70	Communication error on send
EPROTO	71	Protocol error
EMULTIHOP	72	Multihop attempted
EDOTDOT	73	RFS specific error
EBADMSG	74	Not a data message
EOVERFLOW	75	Value too large for defined data type
ENOTUNIQ	76	Name not unique on network
EBADFD	77	File descriptor in bad state
EREMCHG	78	Remote address changed
ELIBACC	79	Can not access a needed shared library
ELIBBAD	80	Accessing a corrupted shared library
ELIBSCN	81	.lib section in a.out corrupted
ELIBMAX	82	Attempting to link in too many shared libraries
ELIBEXEC	83	Cannot exec a shared library directly
EILSEQ	84	Illegal byte sequence
ERESTART	85	Interrupted system call should be restarted
ESTRPIPE	86	Streams pipe error
EUSERS	87	Too many users
ENOTSOCK	88	Socket operation on non-socket
EDESTADDRREQ	89	Destination address required
EMSGSIZE	90	Message too long
EPROTOTYPE	91	Protocol wrong type for socket

ENOPROTOOPT	92	Protocol not available
EPROTONOSUPPORT	93	Protocol not supported
ESOCKTNOSUPPORT	94	Socket type not supported
EOPNOTSUPP	95	Operation not supported on transport endpoint
EPFNOSUPPORT	96	Protocol family not supported
EAFNOSUPPORT	97	Address family not supported by protocol
EADDRINUSE	98	Address already in use
EADDRNOTAVAIL	99	Cannot assign requested address
ENETDOWN	100	Network is down
ENETUNREACH	101	Network is unreachable
ENETRESET	102	Network dropped connection because of reset
ECONNABORTED	103	Software caused connection abort
ECONNRESET	104	Connection reset by peer
ENOBUFS	105	No buffer space available
EISCONN	106	Transport endpoint is already connected
ENOTCONN	107	Transport endpoint is not connected
ESHUTDOWN	108	Cannot send after transport endpoint shutdown
ETOOMANYREFS	109	Too many references: cannot splice
ETIMEDOUT	110	Connection timed out
ECONNREFUSED	111	Connection refused
EHOSTDOWN	112	Host is down
EHOSTUNREACH	113	No route to host
EALREADY	114	Operation already in progress
EINPROGRESS	115	Operation now in progress
ESTALE	116	Stale NFS file handle
EUCLEAN	117	Structure needs cleaning
ENOTNAM	118	Not a XENIX named type file
ENAVAIL	119	No XENIX semaphores available
EISNAM	120	Is a named type file
EREMOTEIO	121	Remote I/O error
EDQUOT	122	Quota exceeded
ENOMEDIUM	123	No medium found
EMEDIUMTYPE	124	Wrong medium type

Sockets - accept()

Accept an incoming request.

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen)
```

Here we can block waiting for an incoming connection request from the server socket. We will return immediately if there is a client connection awaiting acceptance. The address of the client is returned to us along with its length.

If we try and accept too many concurrent sockets, the ESP8266 may return ENFILE to indicate that we have an overflow.

The return is the accepted socket connection to the client or -1 if an error.

Sockets - bind()

Associate a socket with an address.

```
bind(int s, const struct sockaddr *name, socklen_t namelen)
```

The name parameter is the socket address to be bound to the socket. The namelen provides the length of the address. If the sin_addr.s_addr is htonl(INADDR_ANY) then we are being a server listening on any incoming IP address.

A return of < 0 on error.

Here is an example of us defining ourselves as a server:

```
struct sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddr.sin_port = htons(80);

int rc = bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
```

Sockets - close()

Close the corresponding socket.

```
close(int s)
```

Close the socket.

Sockets - closesocket()

Close the corresponding socket.

```
closesocket(int s)
```

Close the socket.

Sockets - connect()

Connect to a server.

```
connect(int s, const struct sockaddr *partnerAddr, socklen_t addrlen)
```

Connect the socket to a partner. The address of the partner is supplied in the `partnerAddr` field. This is a client initiated call and is expected to connect with a listening server.

Sockets - `fcntl()`

Perform control functions.

```
fcntl(int s, int cmd, int val)
```

We can set control functions on sockets here.

Command	Value	Description
<code>F_SETFL</code>	<code>O_NONBLOCK</code>	Set the socket non blocking.

Sockets - `freeaddrinfo()`

Sockets - `getaddrinfo()`

Sockets - `gethostbyname()`

Sockets - `getpeername()`

Retrieve the address associated with the partner/peer to which the socket is connected.

```
getpeername(int s,  
           struct sockaddr *peerAddr,  
           socklen_t *namelen)
```

Note that `namelen` must be primed with the size of the available address buffer.

Sockets - `getsockname()`

Retrieve the current local address to which the socket is bound.

```
getsockname(int s,  
           struct sockaddr *name,  
           socklen_t *namelen)
```

Note that `namelen` must be primed with the size of the available address buffer.

Sockets - `getsockopt()`

```
getsockopt(int s,  
          int level,  
          int optname,
```

```
void *optval,  
socklen_t *optlen)
```

An important example of using this function is to retrieve the last error associated with the socket. The following code fragment illustrates this:

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

Sockets - htonl()

Convert a host formatted long integer to network byte order.

Sockets - htons()

Convert a host formatted short integer to network byte order.

Sockets - inet_ntop()

Sockets - inet_pton()

Sockets - ioctlsocket()

```
ioctl(int s, long cmd, void *argp)
```

Sockets - listen()

Start listening for incoming connections.

```
listen(int s, int backlog)
```

If we are bound as a server, we will start to listen for incoming connections requests on the socket. The `backlog` parameter defines how many sockets we can keep a handle to before we accept them.

A return value < 0 means an error.

Sockets - read()

Receive data from a partner.

```
ssize_t read(int s, void *mem, size_t len)
```

Similar to the recv() function.

See also:

- Sockets – recv()
- Sockets – recvfrom()

Sockets - recv()

Receive data from a partner.

```
ssize_t recv(int s,  
            void *mem,  
            size_t len,  
            int flags)
```

This function returns the number of bytes actually received. A value of -1 indicates an error. A value of zero indicates the partner having closed the connection.

The flags is the boolean combination of:

- MSG_CMSG_CLOEXEC
- MSG_DONTWAIT – Indicate that we don't want to block waiting for data. If there is no data immediately available for us to receive, we return -1 to indicate an error and the error code is "EAGAIN".
- MSG_ERRQUEUE
- MSG_OOB
- MSG_PEEK
- MSG_TRUNC
- MSG_WAITALL

See also:

- Sockets – read()
- Sockets – recvfrom()
- [man\(2\) - recv](#)

Sockets - recvfrom()

```
recvfrom(int s,  
        void *mem,  
        size_t len,  
        int flags,  
        struct sockaddr *from,  
        socklen_t *fromlen)
```

See also:

- Sockets – read()

- Sockets – recv()

Sockets - select()

Check for data available for reading or writing.

```
select(int maxfdp1,
       fd_set *readset,
       fd_set *writeset,
       fd_set *exceptset,
       struct timeval *timeout)
```

On some Linux systems, to use selected, one would include <sys/select.h>. In ESP-IDF that is not present but does not appear to be needed.

Sockets - send()

Send a set of bytes down the socket to the partner.

```
ssize_t send(int s, const void *dataptr, size_t size, int flags)
```

The data pointed to by dataptr for size bytes is transmitted.

See also:

- [man\(2\) – send](#)

Sockets - sendto()

Send data to a UDP partner.

```
sendto(int s,
       const void *dataptr,
       size_t size,
       int flags,
       const struct sockaddr *to,
       socklen_t tolen)
```

Sockets - setsockopt()

```
setsockopt (int s,
            int level,
            int optname,
            const void *optval,
            socklen_t optlen)
```

The options that are anticipated to be available are:

- **TCP_NODELAY** – Disable the Nagle algorithm.
- **SO_KEEPALIVE** – Enable liveness pinging.

Sockets - shutdown()

Shutdown parts of a socket.

```
shutdown(int s, int how)
```

Shutdown all or part of the socket.

The socket is shutdown based on the how parameter which may be one of:

- SHUT_RD – No further receives are allowed.
- SHUT_WR – No further writes are allowed.
- SHUT_RDWR – No further reads or writes are allowed.

Sockets - socket()

Create a new socket for the specific domain, type and protocol.

```
socket(int domain, int type, int protocol)
```

Domain can be one of:

- AF_INET – TCP/IP
- Others ...

Type can be one of:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

Protocol can be one of:

- IPPROTO_IP
- IPPROTO_TCP
- IPPROTO_UDP

A common usage pattern is:

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

Returns a new socket descriptor or a value < 0 on error.

Sockets - write()

```
write(int s, const void *dataptr, size_t size)
```

Socket data structures

Sockets - struct sockaddr

Sockets - struct sockaddr_in

- `sin_family` – `AF_INET`
- `sin_port`
- `struct in_addr sin_addr` – This structure has a member called `s_addr` which is an IP address. Special values have special meanings. For example `INADDR_ANY` is any address.

Java Sockets

The sockets API is the defacto standard API for programming against TCP/IP. My programming language of choice is Java and it has full support for sockets. What this means is that I can write a Java based application that leverages sockets to communicate with the ESP*. I can send and receive data through quite easily.

In Java, there are two primary classes that represents sockets, those are `java.net.Socket` which represents a client application which will form a connection and the second class is `java.net.ServerSocket` which represents a server that is listening on a socket awaiting a client connection. Since the ESP* can be either a client or a server, both of these Java classes will come into play.

To connect to an ESP* running as a server, we need to know the IP address of the device and the port number on which it is listening. Once we know those, we can create an instance of the Java client with:

```
Socket clientSocket = new Socket(ipAddress, port);
```

This will form a connection to the ESP*. Now we can ask for both an `InputStream` from which to receive partner data and an `OutputStream` to which we can write data.

```
InputStream is = clientSocket.getInputStream();
OutputStream os = clientSocket.getOutputStream();
```

When we are finished with the connection, we should call `close()` to close the Java side of the connection:

```
clientSocket.close();
```

It really is as simple as that. Here is an example application:

```
package kolban;

import java.io.OutputStream;
import java.net.Socket;
```

```

import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.DefaultParser;
import org.apache.commons.cli.Options;

public class SocketClient {
    private String hostname;
    private int port;

    public static void main(String[] args) {
        Options options = new Options();
        options.addOption("h", true, "hostname");
        options.addOption("p", true, "port");
        CommandLineParser parser = new DefaultParser();
        try {
            CommandLine cmd = parser.parse(options, args);

            SocketClient client = new SocketClient();
            client.hostname = cmd.getOptionValue("h");
            client.port = Integer.parseInt(cmd.getOptionValue("p"));
            client.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void run() {
        try {
            int SIZE = 65000;
            byte data[] = new byte[SIZE];
            for (int i = 0; i < SIZE; i++) {
                data[i] = 'X';
            }
            Socket s1 = new Socket(hostname, port);
            OutputStream os = s1.getOutputStream();
            os.write(data);
            s1.close();
            System.out.println("Data sent!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
} // End of class
// End of file

```

To configure a Java application as a socket server is just as easy. This time we create an instance of the `SocketServer` class using:

```
SocketServer serverSocket = new SocketServer(port)
```

The port supplied is the port number on the machine on which the JVM is running that will be the endpoint of remote client connection requests. Once we have a

`ServerSocket` instance, we need to wait for an incoming client connection. We do this using the blocking API method called `accept()`.

```
Socket partnerSocket = serverSocket.accept();
```

This call blocks until a client connect arrives. The returned `partnerSocket` is the connected socket to the partner which can be used in the same fashion as we previously discussed for client connections. This means that we can request the `InputStream` and `OutputStream` objects to read and write to and from the partner. Since Java is a multi-threaded language, once we wake up from `accept()` we can pass off the received partner socket to a new thread and repeat the `accept()` call for other parallel connections. Remember to `close()` any partner socket connections you receive when you are done with them.

So far, we have been talking about TCP oriented connections where once a connection is opened it stays open until closed during which time either end can send or receive independently from the other. Now we look at datagrams that use the UDP protocol.

The core class behind this is called `DatagramSocket`. Unlike TCP, the `DatagramSocket` class is used both for clients and servers.

First, let us look at a client. If we wish to write a Java UDP client, we will create an instance of a `DatagramSocket` using:

```
DatagramSocket clientSocket = new DatagramSocket();
```

Next we will "connect" to the remote UDP partner. We will need to know the IP address and port that the partner is listening upon. Although the API is called "connect", we need to realize that no connection is formed. Datagrams are connectionless so what we are actually doing is associating our client socket with the partner socket on the other end so that **when** we actually wish to send data, we will know where to send it to.

```
clientSocket.connect(ipAddress, port);
```

Now we are ready to send a datagram using the `send()` method:

```
DatagramPacket data = new DatagramPacket(new byte[100], 100);
clientSocket.send(data);
```

To write a UDP listener that listens for incoming datagrams, we can use the following:

```
DatagramSocket serverSocket = new DatagramSocket(port);
```

The port here is the port number on the same machine as the JVM that will be used to listen for incoming UDP connections.

To wait for an incoming datagram, call `receive()`.

```
DatagramPacket data = new DatagramPacket(new byte[100], 100);
clientSocket.receive(data);
```

If you are going to use the Java Socket APIs, read the JavaDoc thoroughly for these classes as there are many features and options that were not listed here.

See also:

- [Java tutorial: All About Sockets](#)
- [JDK 8 JavaDoc](#)

WebSockets

WebSockets is both an API and a protocol introduced in HTML5. Simply put, if we imagine an HTTP server sitting waiting for incoming HTTP requests, we can convert a current request into a socket connection between the server and the browser such that either end can send data to be received by its partner.

Here we see a raw request to upgrade an HTTP connection to a WebSocket connection:

```
GET / HTTP/1.1
Host: 192.168.1.10
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: file://
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Sec-WebSocket-Key: saim6TzFH+zVb4qY2nrh0Q==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

See also:

- html5rocks – [Introducing WebSockets: Bringing Sockets to the Web](#)
- [The WebSocket protocol – RFC6455](#)
- [The WebSocket API](#)

A WebSocket browser app

The highest likelihood is that you will be running your ESP8266 as a WebSocket server. This would imply that you are going to have browser hosted applications that will be connecting to a WebSocket server as clients and from there, you will likely be writing some WebSocket client code ... if nothing else then for unit testing your server.

Because of that, we will now spend some time talking about what is involved in writing a WebSocket client application.

Let us assume that we will be writing JavaScript hosted in the browser. We start by creating an instance of a WebSocket object passing in the URL to the WebSocket server:

```
var ws = new WebSocket("ws://<somehost>[:<someport>]");
```

The WebSocket API is mostly event driven and there are a number event types of interest to us:

- `open` – Invoked when the connection to the WebSocket server has been established and we are now ready to send or receive data. We can define this with the "`onopen`" property of the WebSocket as a function reference.
- `message` – Receive a message from the server. We can define this with the "`onmessage`" property of the WebSocket as a function reference.
- `error` – Receive an indication that an error was detected. We can define this with the "`onerror`" property of the WebSocket as a function reference.
- `close` – Receive an indication that a request to close the connection was detected. We can define this with the "`onclose`" property of the WebSocket as a function reference.

Event handlers can be registered either with an "`on<Event>`" mechanism or with an `addEventListener()` call.

There are two methods defined on a WebSocket object. Those are:

- `send` – Send data to a WebSocket server
- `close` – Close the connection to a WebSocket server. The `close()` method takes two parameters:
 - `close code` – An integer close code describing the reason for the close.
 - 1000 – `CLOSE_NORMAL`
 - 1001 – `CLOSE_GOING_AWAY`
 - `status message` – A string describing the close reason.

Finally, there are a few attributes:

- `readyState` – The state of the WebSocket connection. Values include:
 - `WebSocket.CONNECTING`
 - `WebSocket.OPEN`
 - `WebSocket.CLOSING`
 - `WebSocket.CLOSED`

- `bufferedAmount` – Amount of data that is buffered pending transmission to the WebSocket server
- `protocol` – The WebSocket server selected protocol being used.

FreeRTOS WebSocket

The FreeRTOS SDK distributes an implementation of the noPoll open source project.

To use noPoll in your app, you must include `"nopoll.h"`.

The context is very important and should be created at the start and disposed of at the end. For example:

```
noPollCtx *ctx = nopoll_ctx_new();
// Do something ...
nopoll_ctx_unref();
```

Once a context has been created, we can then register ourselves as a server:

```
noPollCon *listener = nopoll_listener_new(ctx, "0.0.0.0", "1234");
nopoll_ctx_set_on_msg(ctx, listener_on_message_handler, NULL);
nopoll_loop_wait(ctx, 0);
```

When a message is received, the registered function (`listener_on_message_handler`) will be invoked:

```
void listener_on_message_handler(
    noPollCtx *ctx,
    noPollConn *conn,
    noPollMsg *msg,
    noPollPtr *userData) {
    // Do something
    nopoll_conn_send_text(conn, "Thanks", 5);
}
```

An example WebSocket application is supplied by Espressif in the FreeRTOS SDK for ESP8266. The sample can be found in `/examples/websocket_demo`.

See also:

- [noPoll: OpenSource WebSocket toolkit](#)
- [The WebSocket protocol – RFC6455](#)
- [The WebSocket API](#)

Mongoose WebSocket

Using Cesanta's Mongoose libraries, we can setup a WebSocket server. After setting up a binding for incoming network requests we can call

`mg_set_protocol_http_websocket()`. This will attach an event handler to the network protocol level to handle events associated with WebSockets. Specifically, these are the following events we are interested in:

- MG_EV_WEBSOCKET_HANDSHAKE_REQUEST
- MG_EV_WEBSOCKET_HANDSHAKE_DONE
- MG_EV_WEBSOCKET_FRAME

When an MG_EV_WEBSOCKET_HANDSHAKE_REQUEST is received, the data contains a parsed HTTP request as a struct http_message.

The struct http_message contains:

- message
- method
- uri
- proto
- resp_code
- resp_status_msg
- query_string
- header_names
- header_values
- body

When an MG_EV_WEBSOCKET_FRAME is received, the data contains a reference to a struct websocket_message. The struct websocket_message contains:

- data – The data passed from the partner.
- size – The size of the passed data.
- flags – Flags (unknown).

Web Servers

A Web Server is a software component that listens for incoming HTTP requests from Web browsers. There are many implementations of Web Servers that can run within an ESP8266 environment.

Mongoose

Mongoose provides an API that one can use to build a rich and powerful HTTP server. At a high level, we call `mg_mgr_init()` to initialize the environment. Next we bind it to a handler using `mg_bind()`. Finally, we poll the server for work using `mg_mgr_poll()`.

```

void setupMongoose() {
    struct mg_mgr mgr;
    printf("Starting mongoose setup\n");
    mg_mgr_init(&mgr, NULL);
    printf("Successfully initied\n");
    mg_bind(&mgr, "80", evHandler);
    printf("Successfully bound\n");
    while(1) {
        mg_mgr_poll(&mgr, 1000);
    }
}

```

When a request arrives from a browser, we consider that an event which is handed off to an event handler. The signature of an event handler is:

```

void eventHandler(
    struct mg_connection *nc,
    int ev,
    void *evData)

```

Where:

- `nc` – The connection that received the event.
- `ev` – The type of event that triggered the callback.
- `evData` – Data associated with the event.

Thus far we will receive callbacks for socket connections. If we wish, we can now register that we wish to parse the incoming data as an HTTP request. We do that by making a call to `mg_set_protocol_http_websocket()`.

When setup as an HTTP server, an incoming browser request will appear with the event type of `MG_EV_HTTP_REQUEST`. The `evData` passed in will be an instance of `struct http_message` from which the nature of the request can be determined.

See also:

- Github: [cesanta/mongoose](#)
- [Mongoose Developer Centre](#)

Programming using Eclipse

Eclipse is a popular open source framework primarily used for hosting application development tools. Although primarily geared for building Java applications, it also has first class C and C++ support.

ESP8266

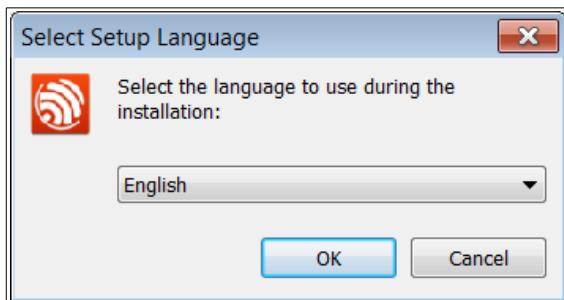
A project for building ESP8266 applications using Eclipse can be found here:

- <http://www.esp8266.com/viewtopic.php?f=9&t=820>

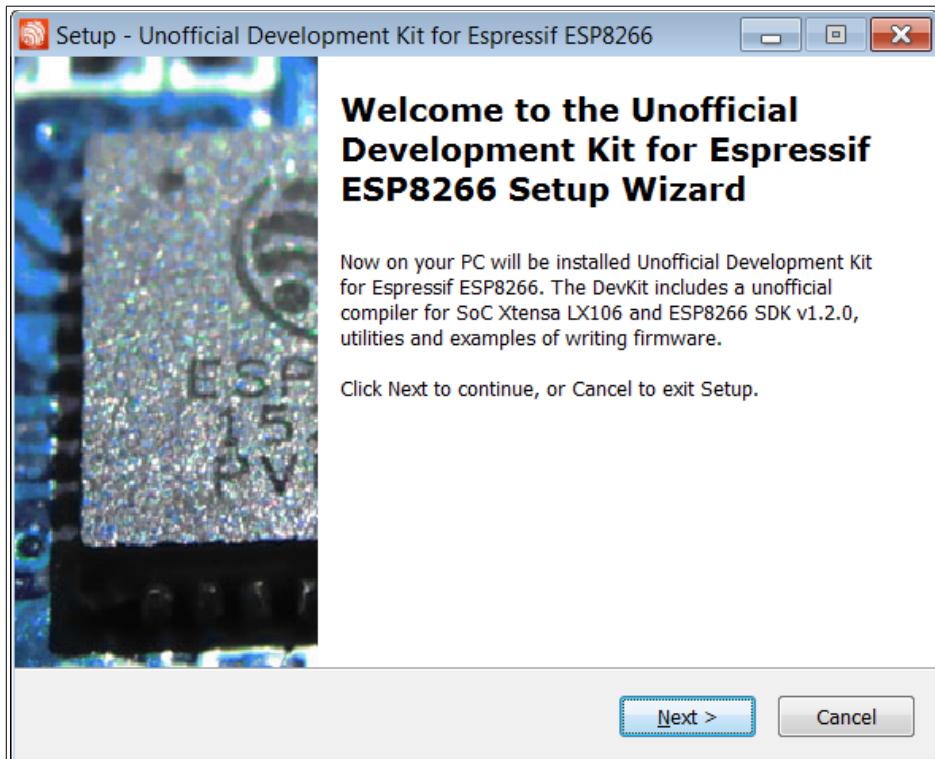
Do not include spaces in any of the path parts pointing to the workspace. Here are some notes on installing this project ... however, always read the documentation accompanying the project.

Download the `Espressif-ESP8266-DevKit-vxxx-x86`. This is a large download of approx 125MBytes.

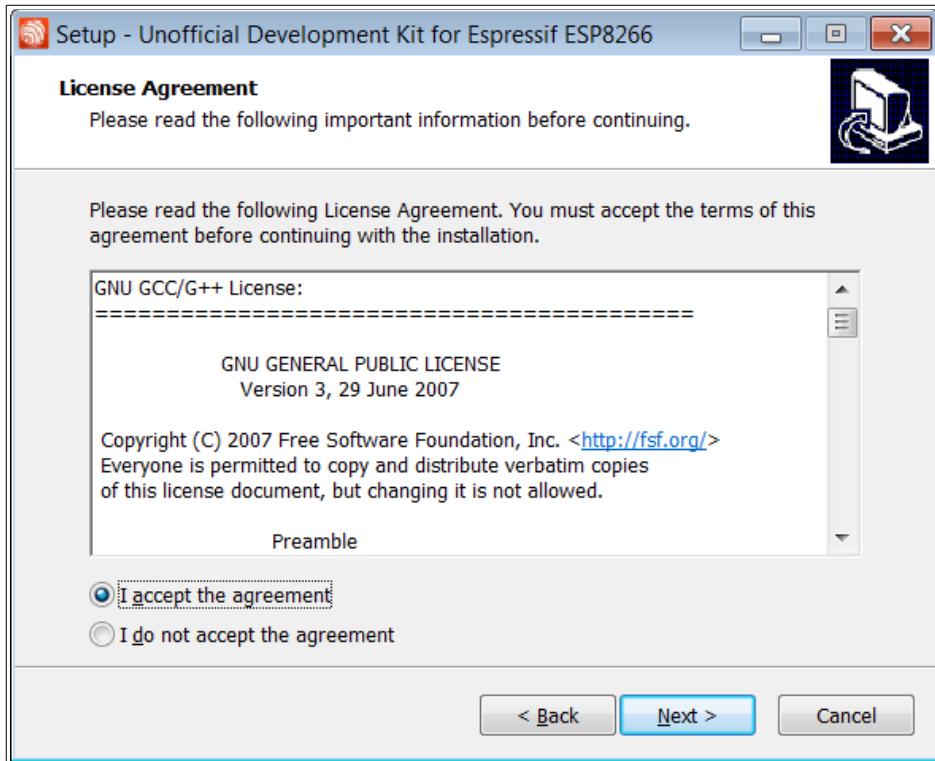
Run the installer. It will ask you for your choice of installation language.



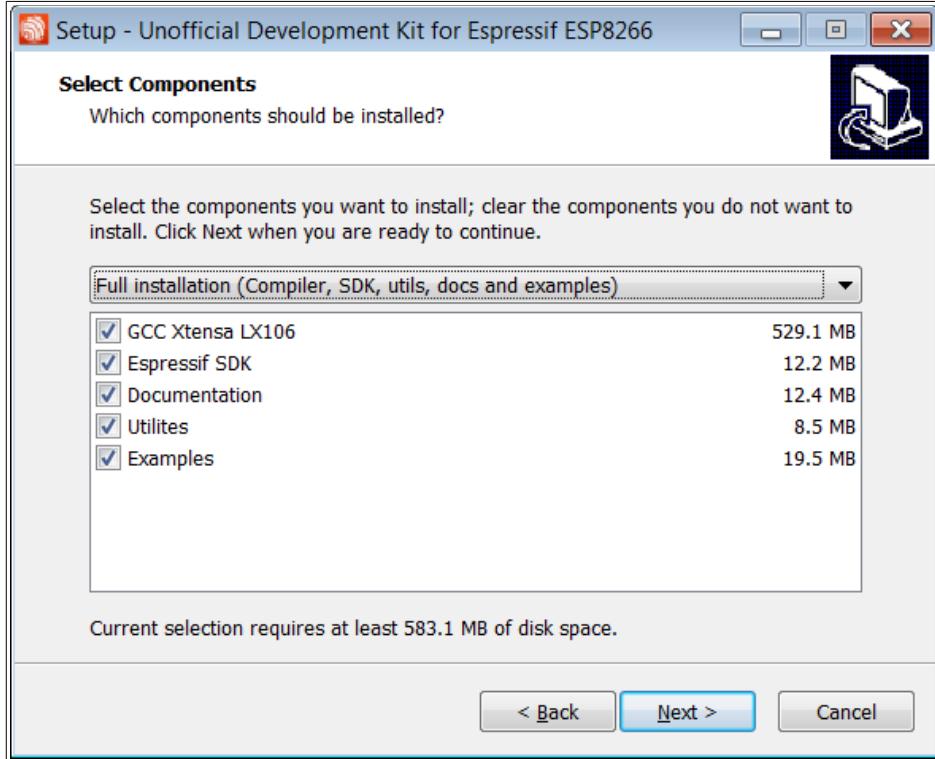
Next comes the splash screen:



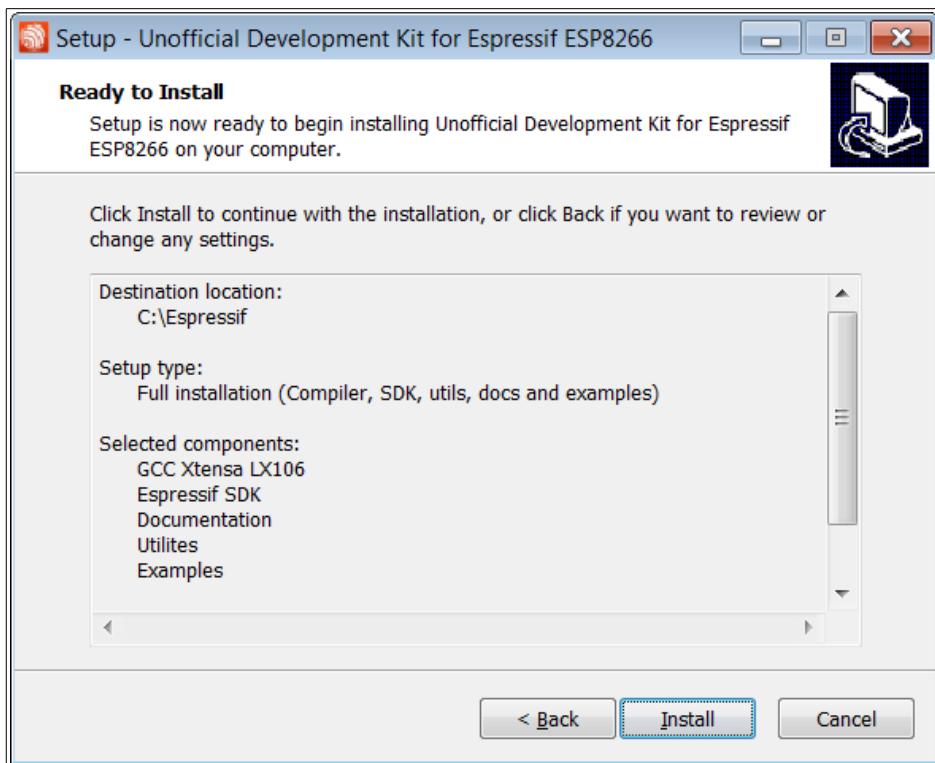
Next comes the license agreement:



Now the selection of which components to install:



Finally a confirmation dialog to review what you have selected.



The result of this will be a new directory structure at C:\Espressif\.

There are other dependencies that you will need which are listed at the link above.
These include:

- A Java runtime environment. I use the latest Java 8 from Oracle.
- Eclipse environment with C/C++ developer tools. I use the latest "Mars" release.
- MinGW – Unix tools and utilities that execute on Windows.
- MinGW installation helper – A cache and list of the MinGW packages that need to be installed for correct operation.

The Makefiles supplied with the package are key. They have been crafted to provide the easiest compiles. The targets contained within the Makefiles include:

- all – Compile all the code but do not flash.
- clean – Clean any previous builds.
- flash – Compile the code if needed and then flash.
- flashboot
- flashinit

- `flashonefile`

There are some flags that are used with the Makefile that you can edit. These include:

- `VERBOSE=1` – Enable verbosity which includes debug information. Specifically the compilation commands are shown.

See also:

- [Eclipse.org](#)
- [Eclipse C/C++ Development Tooling \(CDT\)](#)
- [Primary forum thread](#)

Installing the Eclipse Serial terminal

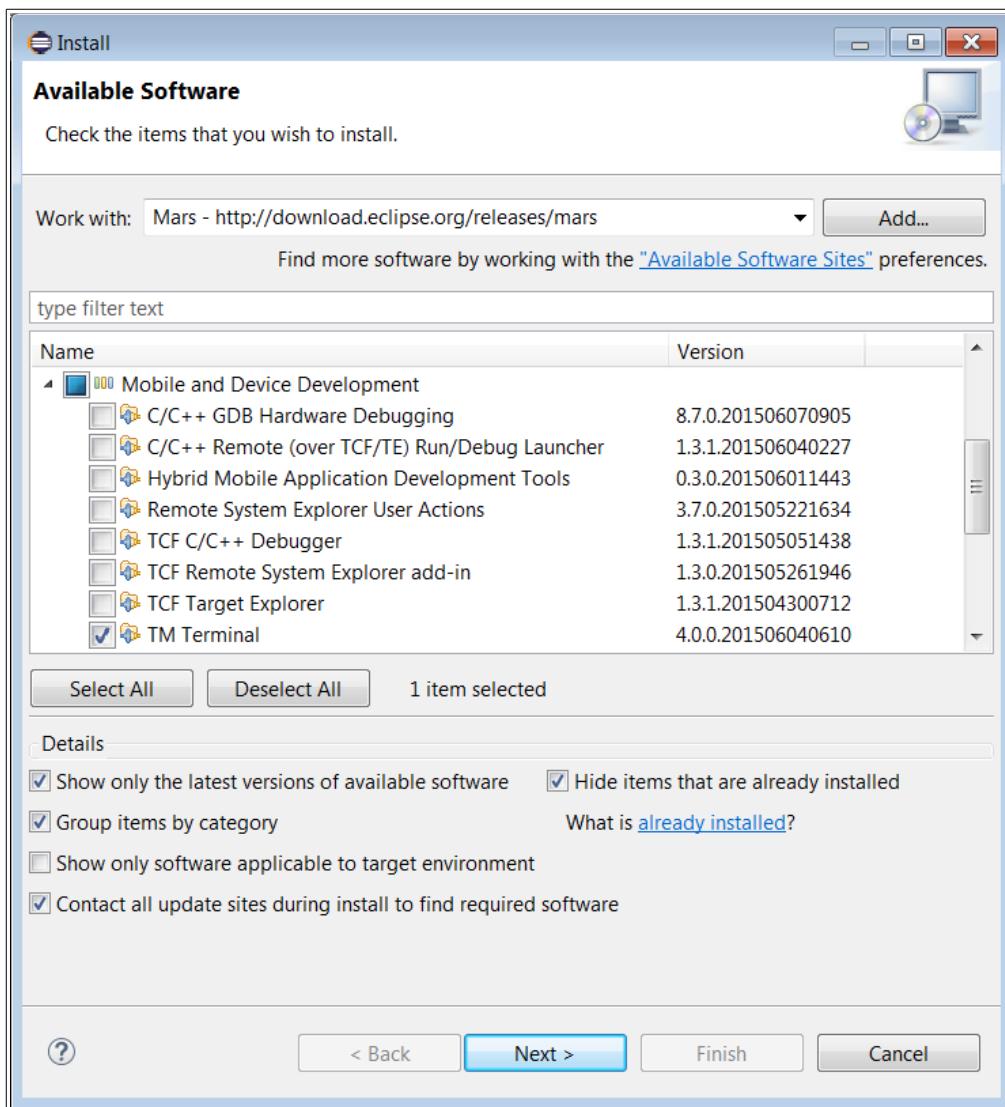
Although there are many excellent serial terminals available as stand-alone Windows applications, an alternative is the Eclipse Terminal which also has serial support. This allows a serial terminal to appear as a view within the Eclipse IDE. It does not come installed by default but the steps to add are not complex.

First start Eclipse (I use the Mars release).

Go to Help > Install new software.

Select the eclipse download repository.

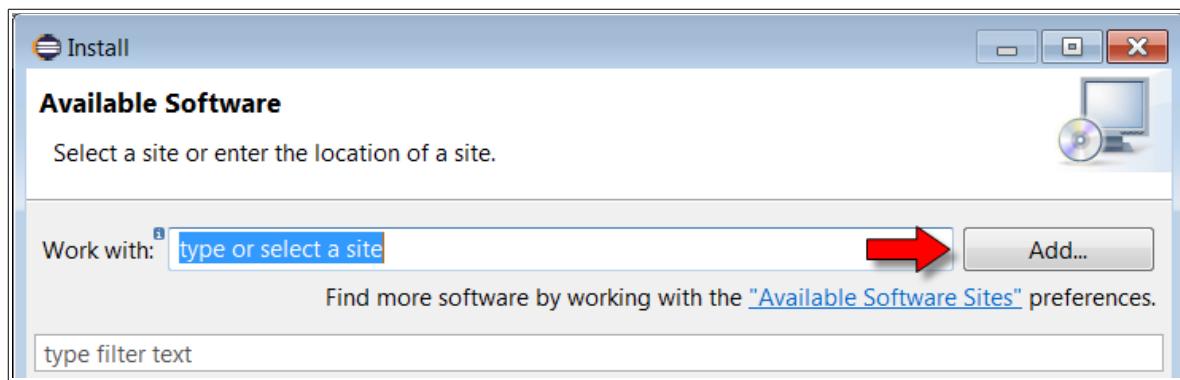
Select Mobile and Device development > TM Terminal.



Step through the following sections and when prompted to restart, accept yes.

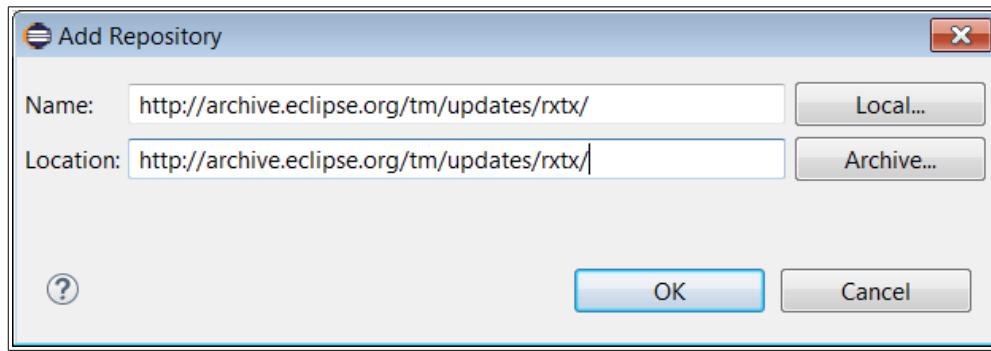
We are not ready to use it yet, we must add serial port support into Eclipse.

Go back to Help > Install new software and add a new repository

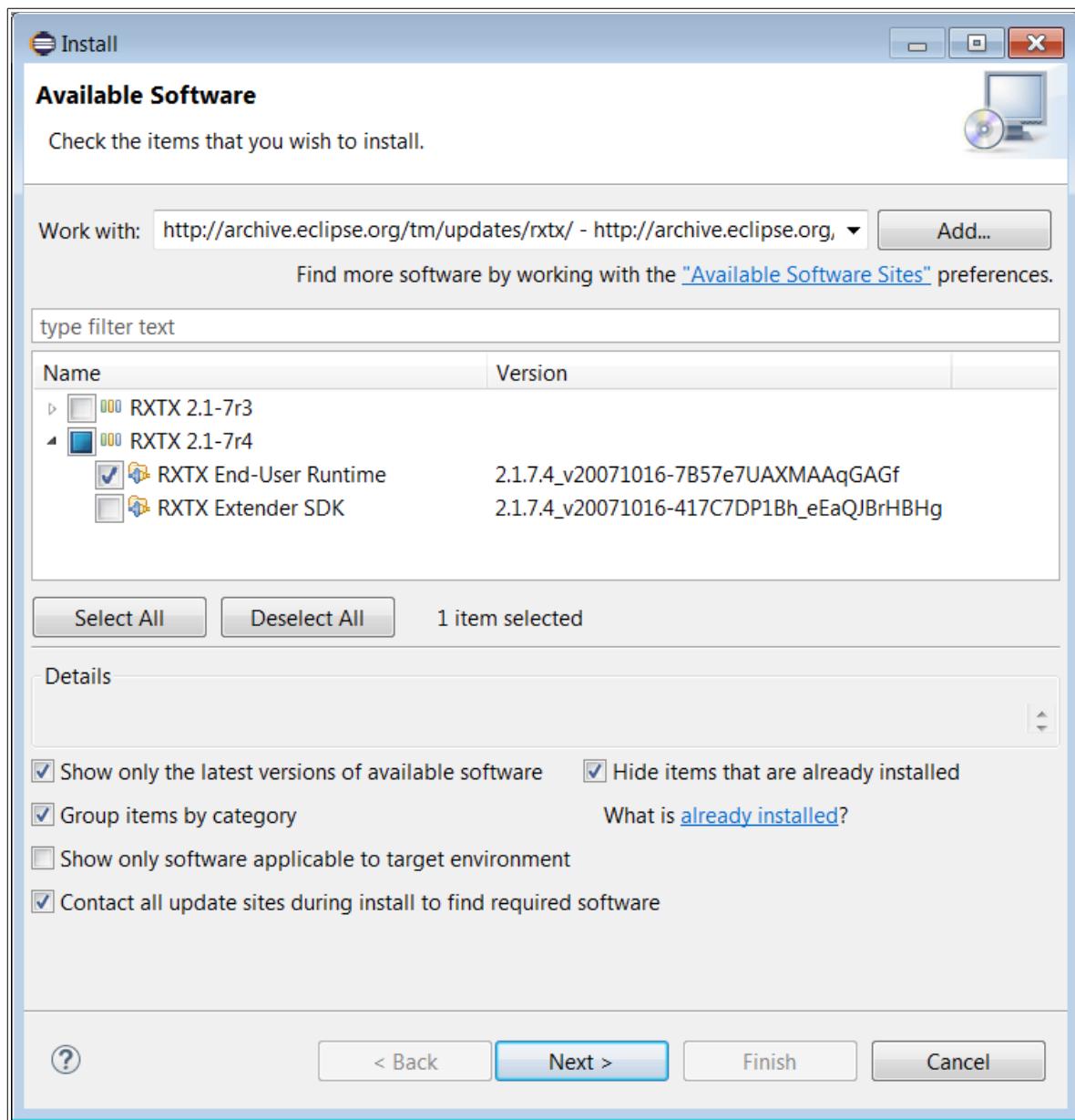


The repository URL is:

- <http://archive.eclipse.org/tm/updates/rxtx/>

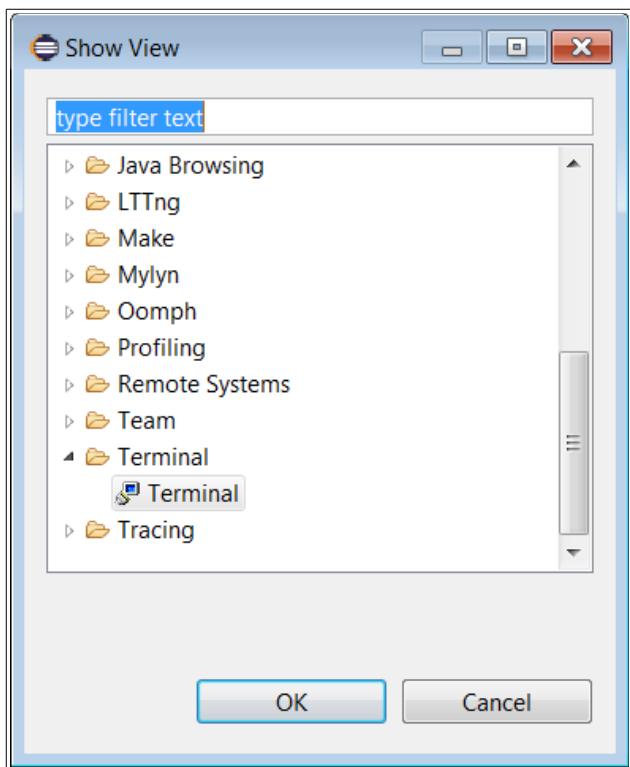


Now we can select the Serial port run-time support library:

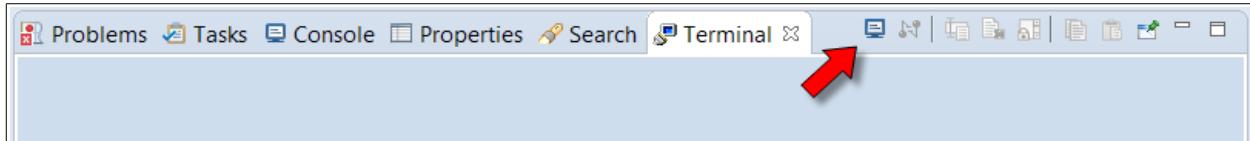


Follow through the further navigation screens and restart Eclipse when prompted.

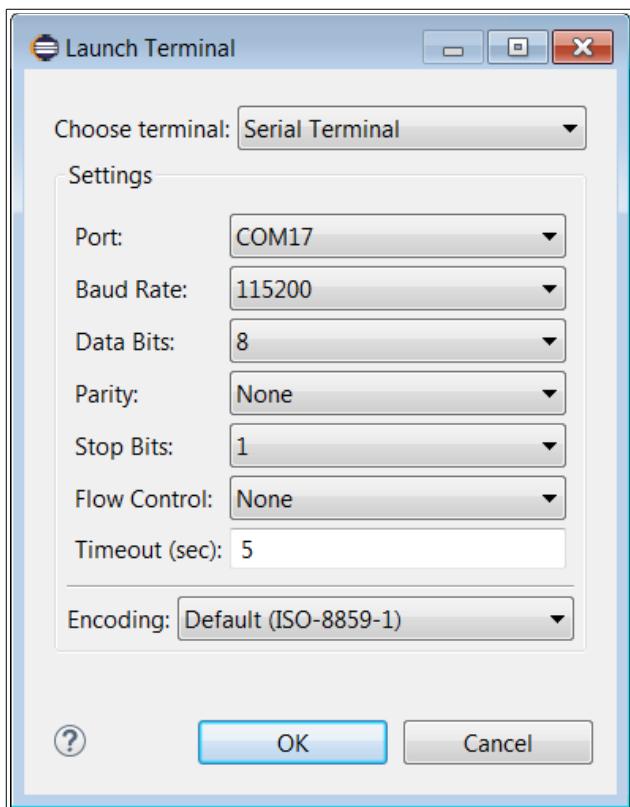
We now have terminal support installed and are ready to use it. From Windows > Show View > Other we will find a new category called "Terminal".



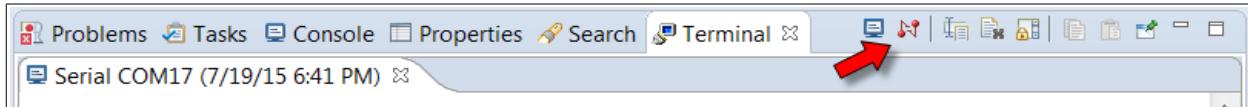
Opening this adds a Terminal view to our perspective. There is a button that will allow us to open a new terminal instance that is shown in the following image:



Clicking this brings up the dialog asking us for the type of terminal and the properties. For our purposes, we wish to choose a serial terminal. Don't forget to also set the port and baud rate to match what your ESP8266 uses.



After clicking OK, after a few seconds we will see that we are connected and a new disconnect icon appears:



And now the terminal is active. For my purposes, I connect this terminal to UART1 of the ESP8266 for debugging while leaving UART0 for flashing new copies of my application. Here is an example of what my typical window looks like:

```
Problems Tasks Console Properties Search Terminal
Serial COM17 (7/19/15 6:41 PM)
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
EfæÄÄb¥Ä{Fó ùmode : sta(18:fe:34:f1:ce:53) + softAP(1a:fe:34:f1:ce:53) !
add if0
add if1
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
bcn 100
InitDone!
del if0
mode : softAP(1a:fe:34:f1:ce:53)
bcn 0
del if1
usl
sul 0 0
add if1
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
bcn 100
```

You can invert the colors to produce a white on black visualization which many users prefer.

Web development using Eclipse

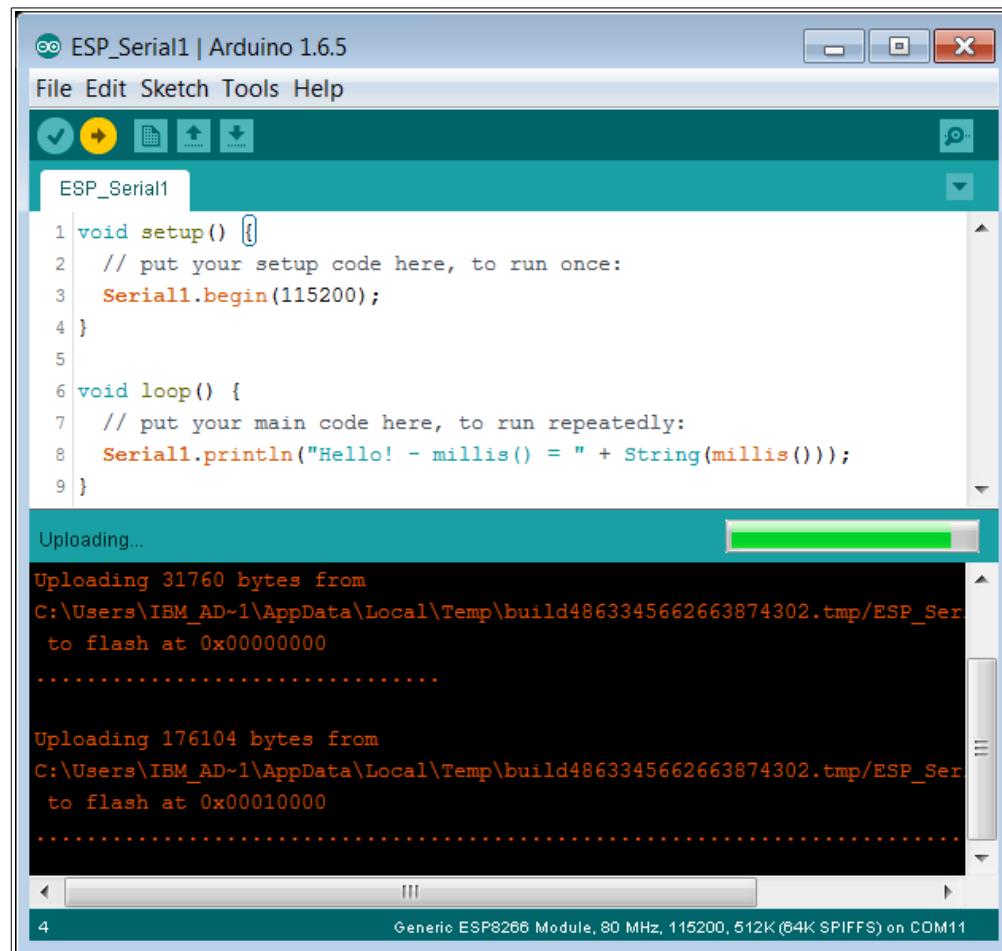
Eclipse also provides a first class web development environment for writing and testing web apps including HTML pages. It is suggested that the Eclipse Web Developer Tools be installed.

Programming using the Arduino IDE

Long before there was an ESP8266, there was the Arduino. A vitally important contribution to the open source hardware community and the entry point for the majority of hobbyists into the world of home built circuits and processors.

One of the key attractions about the Arduino is its relative low complexity allowing everyone the ability to build something quickly and easily. The Integrated Development Environment (IDE) for the Arduino has always been free of charge for download from the Internet. If a professional programmer were to sit down with it, they would be shocked at its apparent limited capabilities. However, the subset of function it provides compared to a "full featured" IDE happen to cover 90% of what one wants to achieve. Combine that with the intuitive interface and the Arduino IDE is a force to be reckoned with.

Here is what a simple program looks like in the Arduino IDE:



```
ESP_Serial1 | Arduino 1.6.5
File Edit Sketch Tools Help
ESP_Serial1
1 void setup() {
2   // put your setup code here, to run once:
3   Serial1.begin(115200);
4 }
5
6 void loop() {
7   // put your main code here, to run repeatedly:
8   Serial1.println("Hello! - millis() = " + String(millis()));
9 }

Uploading...
Uploading 31760 bytes from
C:\Users\IBM_AD~1\AppData\Local\Temp\build4863345662663874302.tmp\ESP_Ser
to flash at 0x00000000
.....
Uploading 176104 bytes from
C:\Users\IBM_AD~1\AppData\Local\Temp\build4863345662663874302.tmp\ESP_Ser
to flash at 0x00010000
.....
4 Generic ESP8266 Module, 80 MHz, 115200, 512K (64K SPIFFS) on COM11
```

In Arduino parlance, an application is termed a "sketch". Personally, I'm not a fan of that phrase but I'm sure research was done to learn that this is the least intimidating name

for what would otherwise be called a C language program and that would intimidate the least number of people.

The IDE has a button called "Verify" which, when clicked, compiles the program. Of course, this will also have the side-effect that it will verify that the program compiles cleanly ... but compilation is what it does. A second button is called "Upload" that, when clicked, what it does is deploy the application to the Arduino.

In addition to providing a C language editor plus tools to compile and deploy, the Arduino IDE provides pre-supplied libraries of C routines that "hide" complex implementation details that might otherwise be needed when programming to the Arduino boards. For example, UART programming would undoubtedly have to set registers, handle interrupts and more. Instead of making the poor users have to learn these technical APIs, the Arduino folks provided high level libraries that could be called from the sketches with cleaner interfaces which hide the mechanical "gorp" that happens under the covers. This notion is key ... as these libraries, as much as anything else, provide the environment for Arduino programmers.

Interesting as this story may be, you may be asking how this relates to our ESP8266 story? Well, a bunch of talented individuals have built out an Open Source project on Github that provides a "plug-in" or "extension" to the Arduino IDE tool (remember, that the Arduino IDE is itself free). What this extension does is allow one to write sketches in the Arduino IDE that leverage the Arduino library interfaces which, at compile and deployment time, generate code that will run on the ESP8266. What this effectively means is that we can use the Arduino IDE and build ESP8266 applications with the minimum of fuss.

Implications of Arduino IDE support

The ability to treat an ESP8266 as though it were "like" an Arduino is a notion that I haven't been able to fully absorb yet. ESP8266 is a Tensilica CPU unlike the Arduino which is an ATmega CPU. Espressif have created dedicated and architected API in the form of their SDK for directly exposed ESP8266 APIs. The Arduino libraries for ESP8266 seem to map their intent to these exposed APIs. For these reasons and similar, one might argue that the Arduino support is an unnecessary facade on top of a perfectly good environment and by imposing an "alien" technology model on top of the ESP8266 native functions, we are masking access to lower levels of knowledge and function. Further, thinking of the ESP8266 as though it were an Arduino can lead to design problems. For example, the ESP8266 needs regular control in order to handle WiFi and other internal actions. This conflicts with the Arduino model where the programmer can do what he wants within the loop function for as long as he wants.

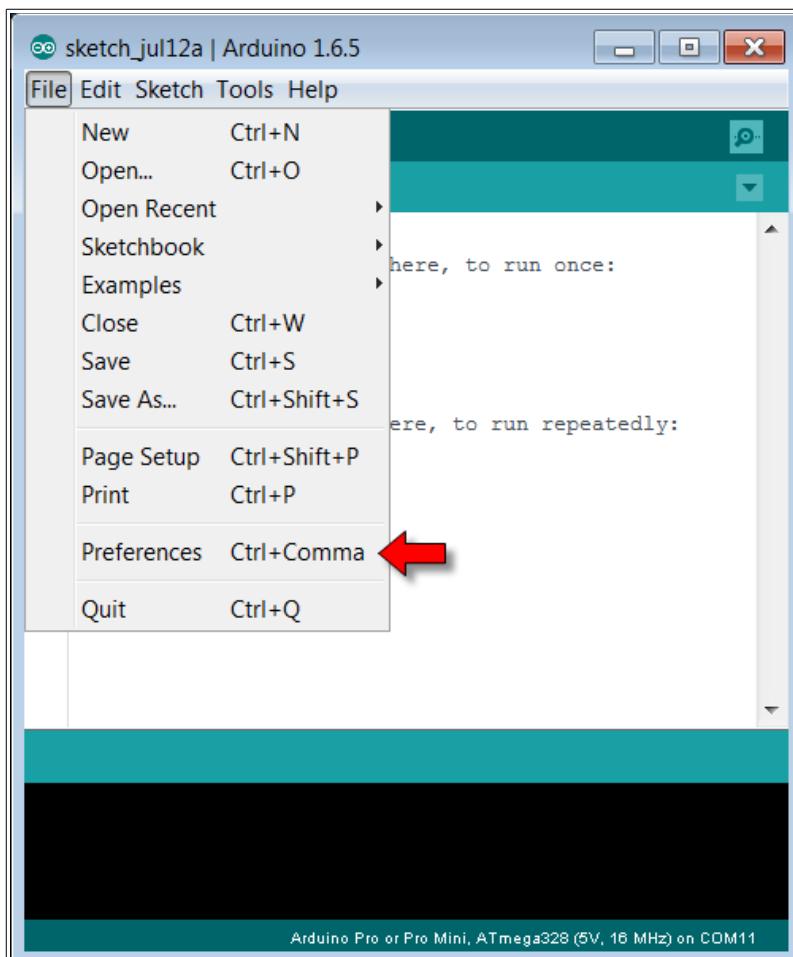
The flip side is that the learning curve to get something running on an Arduino has been shown to be extremely low. It doesn't take long at all to get a blinky light going on a breadboard. With that train of thought, why should users of the ESP8266 be penalized for having to install and learn more complex tool chains and syntax to achieve the same result with more ESP8266 oriented tools and techniques? The name of the game should be to allow folks to tinker with CPUs and sensors without having to have university degrees in computing science or electrical engineering and if the price one pays to get there is to insert a "simple to use" illusion then why not? If I build a paper airplane and throw it out my window ... I may get pleasure from that. A NASA rocket scientist shouldn't scoff at my activities or lack of knowledge of aerodynamics ... the folded paper did its job and I achieved my goal. However, if my job was to put a man on the moon, the ability to visualize the realities of the technology at the "realistic" level becomes extremely important.

Installing the Arduino IDE with ESP8266 support

To assemble this environment, one must download a current version of the Arduino IDE. This will be about 140 Mbytes.

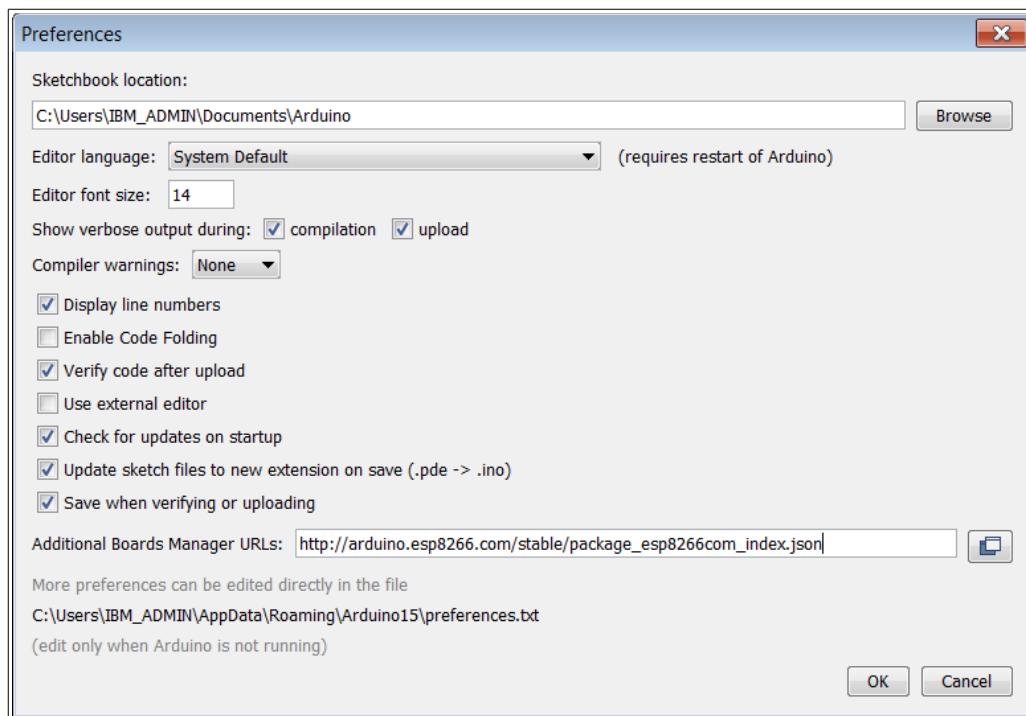
I download the ZIP file version and then extract its content.

Next, we launch the Arduino IDE and open the `Preferences` dialog:

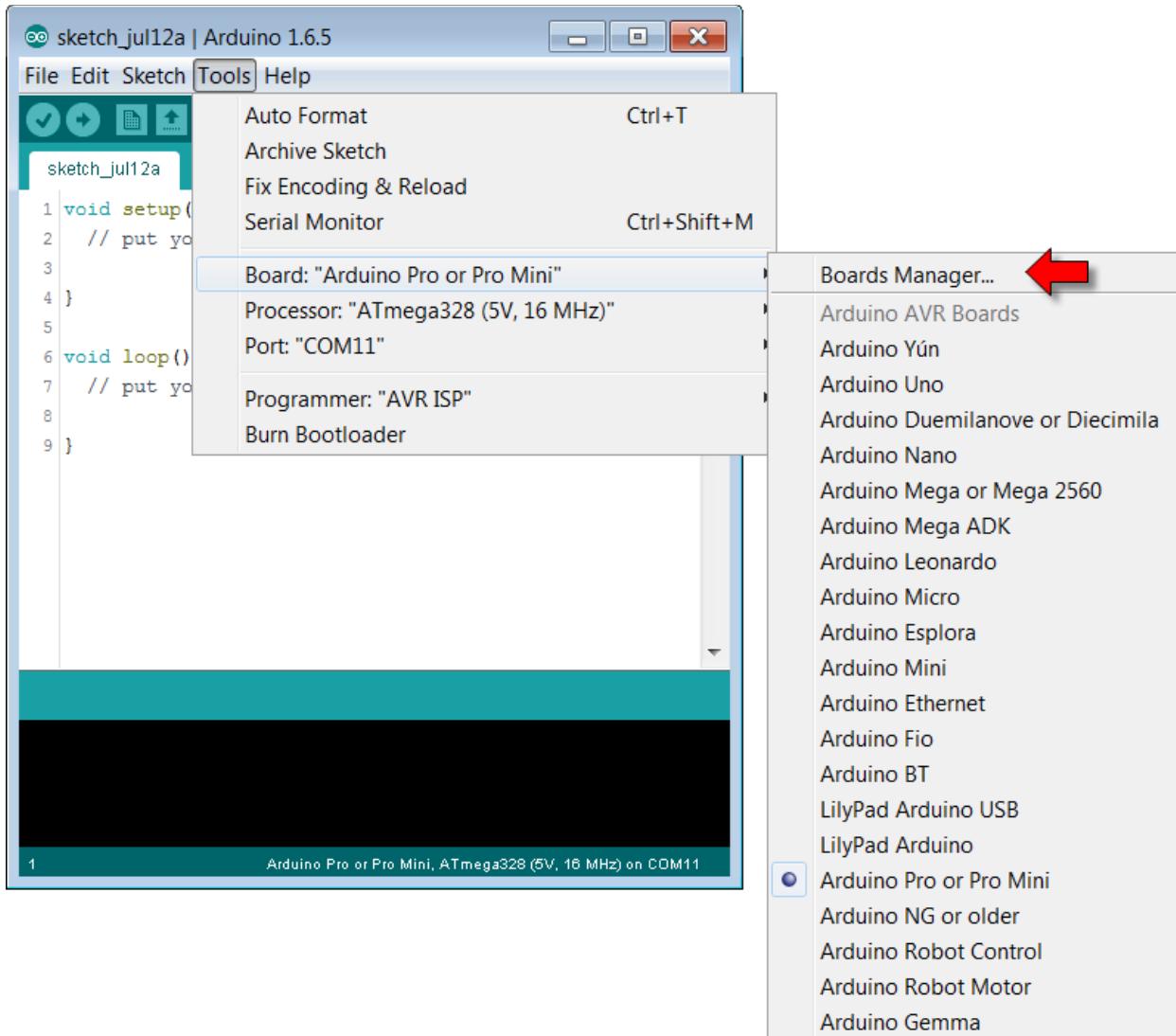


In the Additional Boards Manager URLs field enter the URL for the ESP8266 package which is:

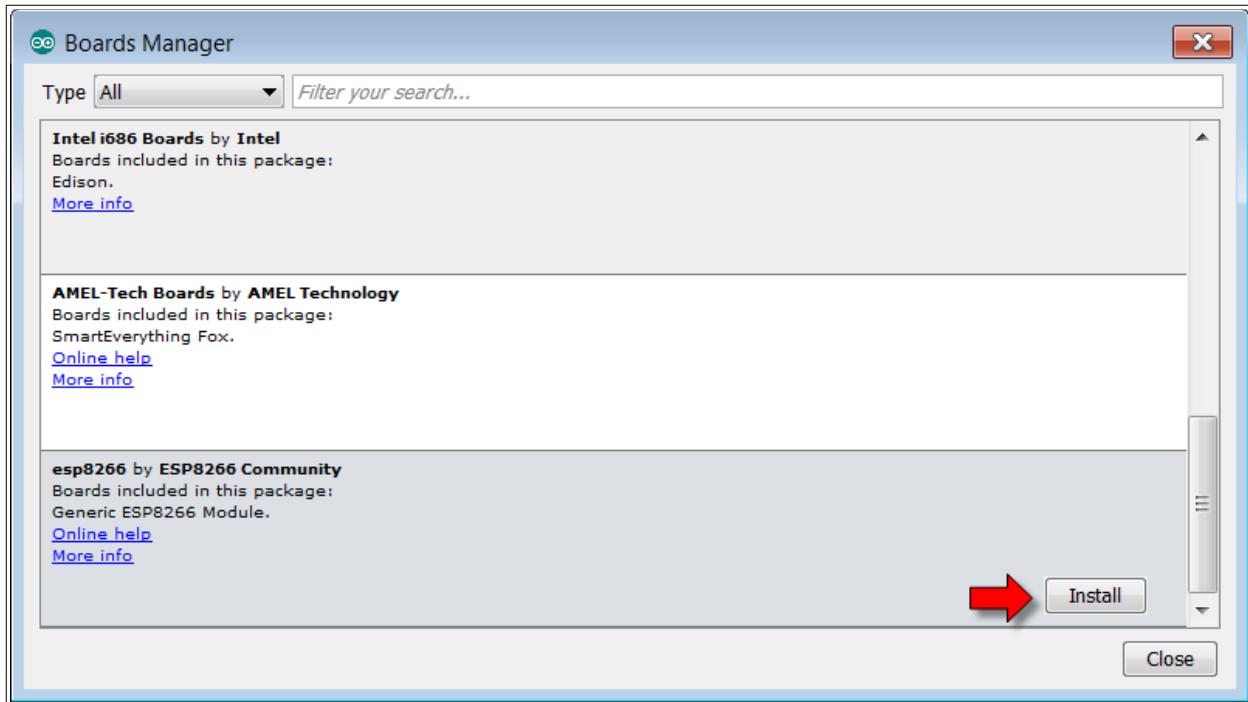
http://arduino.esp8266.com/stable/package_esp8266com_index.json



Select the Boards Manager from the Tools > Board menu:

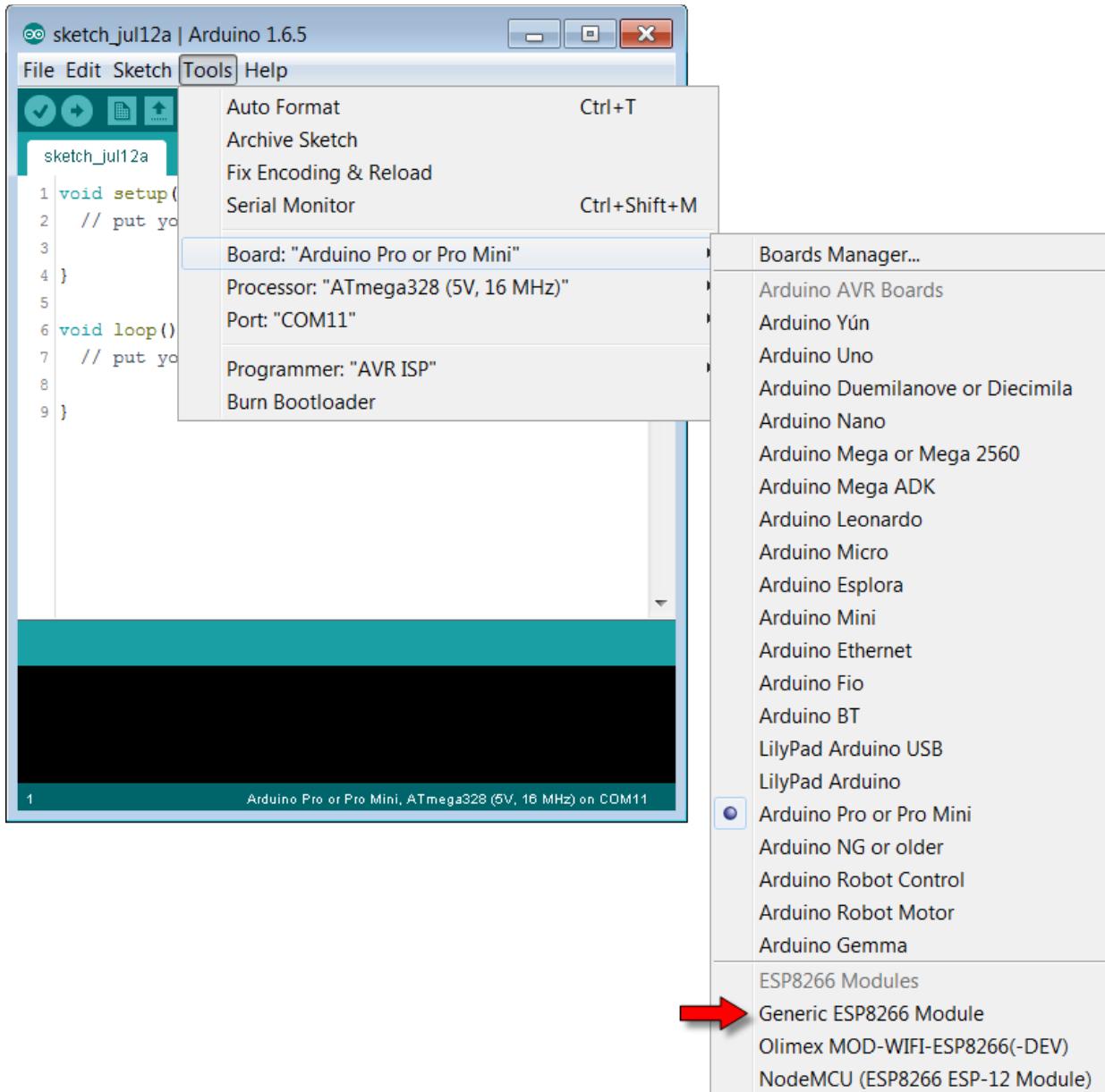


Install the ESP8266 support:



This will contact the Internet and download the artifacts necessary for ESP8266 support.

Once completed, in the Arduino IDE Board selections, you will find the "Generic ESP8266 Module":



Now we are ready to start building, compiling and running sketches.

A simple and sample sketch I recommend for testing is:

```
void setup() {  
    Serial1.begin(115200);  
}  
  
void loop() {  
    Serial1.println("Hello! - millis() = " + String(millis()));  
}
```

When run, a loop of messages will appear on the UART1 output saying hello and the number of milliseconds since last boot. As much as anything, this will validate that the

environment has been setup correctly, you can compile a program and that deployment to the ESP8266 is successful.

See also:

- Github: [esp8266/Arduino](#)
- [Arduino IDE](#)

Tips for working in the Arduino environment

Remember that the Arduino environment is two things. First, an actual application that you install on your machine providing the Arduino IDE. Second, a set of libraries that model those available to an actual Arduino device which are mapped to ESP* capabilities. With that in mind, here are some hints and tips that I find useful when writing Arduino sketches for an ESP* environment.

Initialize global classes in `setup()`

Within an Arduino sketch, we have a pre-supplied function called `setup()` that is called only once during ESP8266 boot-up. Within this function, you perform one time initialization functions. In C++, we have the ability to create class instances globally.

For example:

```
 MyClass myClass(123);

void setup() {
    // Some code here
}
```

instead of this, use the following:

```
 MyClass *myClass;

void setup() {
    myClass = new MyClass(123);
    // Some code here
}
```

This of course changes your variable's data type. It went from being an instance of `MyClass` to being a pointer to an instance of `MyClass` which means that you might have to change other aspects of your program ... but the reason for this is that in the first case, the constructor for your `MyClass` instance ran outside of the `setup()` and we can't say what state the environment might have been in at that point. Within the `setup()` code, we have a reasonable expectation of the environment context.

Invoking Espressif SDK API from a sketch

[ESP8266](#)

There is nothing to prevent you from invoking Espressif SDK API from within your sketch. You must include any include files that are necessary. Here is an example of including "user_interface.h".

```
extern "C" {
    #include "user_interface.h"
}
```

Notice the bracketing with the C++ construct that causes the content to appear as though it were being defined in a C program.

Exception handling

When an exception is detected in the code, the code halts. Typically we see the following logged to the serial port when this happens:

```
ets Jan  8 2013,rst cause:2, boot mode:(1,7)
ets Jan  8 2013,rst cause:4, boot mode:(1,7)
wdt reset
```

Unfortunately, this tells us absolutely nothing about the location or cause of the issue.

The SPIFFS file system

The mkspiffs command

A tool has been made available that builds a "spiffs" file system binary from a directory structure found on disk. The command is called "mkspiffs" and has its own github project.

The full syntax of the command is:

```
mkspiffs {-c <pack_dir>|-l|-i} [-b <number>] [-p <number>] [-s <number>] [-]
[-version] [-h] <image_file>
```

Where the parameters are:

- **-c <pack_dir> or --create <pack_dir>** – Create a spiffs image file from examination of a directory to be packed into the spiffs image.
- **-l or --list** – List the content of an existing image file.
- **-i or --visualize** – Visualize the spiffs image.
- **-b <number> or --block <number>** – The File System size blocks size in bytes.
- **-p <number> or --page <number>** – The File System page size in bytes.
- **-s <number> or --size <number>** – The File System image size in bytes.
- **-- or --ignore_rest** – Ignore the remaining arguments.

- `--version` – Display the version information.
- `-h` or `--help` – Display usage/help information
- `<image_file>` – The file to contain (or already contains) the spiffs image.

See also:

- Github: [igrr/mkspiffs](https://github.com/igrr/mkspiffs)

The architecture of the Arduino IDE support

The Arduino IDE for ESP8266 uses the concept of a "board manager". The thinking behind this was that with the growing number of Arduino related boards out there, all with different capabilities and subtleties, the act of adding support for a new type of device (board) should be made generic and easier. To that end, support was added in 1.6.4 and beyond for the board manager JSON file. This file describes the content of a new board and where to "get" the parts necessary for building applications.

For the ESP8266 Arduino IDE, the board JSON file can be found at:

http://arduino.esp8266.com/stable/package_esp8266com_index.json

If we examine the content of this file in conjunction with the specification of the Arduino IDE package file format, we learn a lot of interesting things.

Here is the file as of 2015-08-03 ...

```
{
  "packages": [ {
    "name": "esp8266",
    "maintainer": "ESP8266 Community",
    "websiteURL": "https://github.com/esp8266/Arduino",
    "email": "ivan@esp8266.com",
    "help": {
      "online": "http://arduino.esp8266.com/versions/1.6.5-947-g39819f0/doc/reference.html"
    },
    "platforms": [ {
      "name": "esp8266",
      "architecture": "esp8266",
      "version": "1.6.5-947-g39819f0",
      "category": "ESP8266",
      "url": "http://arduino.esp8266.com/versions/1.6.5-947-g39819f0/esp8266-1.6.5-947-g39819f0.zip",
      "archiveFileName": "esp8266-1.6.5-947-g39819f0.zip",
      "checksum": "SHA-256:79a395801a94c77f4855f3921b9cc127d679d961ec207e7fb89f90754123d66a",
      "size": "2295584",
      "help": {
        "online": "http://arduino.esp8266.com/versions/1.6.5-947-g39819f0/doc/reference.html"
      }
    }
  }
}
```

```

},
"boards": [
{
  "name":"Generic ESP8266 Module"
},
{
  "name":"Olimex MOD-WIFI-ESP8266 (-DEV) "
},
{
  "name":"NodeMCU 0.9 (ESP-12 Module)"
},
{
  "name":"NodeMCU 1.0 (ESP-12E Module)"
},
{
  "name":"Adafruit HUZZAH ESP8266 (ESP-12) "
},
{
  "name":"SweetPea ESP-210"
}
],
"toolsDependencies": [ {
  "packager":"esp8266",
  "name":"esptool",
  "version":"0.4.5"
},
{
  "packager":"esp8266",
  "name":"xtensa-lx106-elf-gcc",
  "version":"1.20.0-26-gb404fb9"
} ]
} ],
"tools": [ {
  "name":"esptool",
  "version":"0.4.5",
  "systems": [
    {
      "host":"i686-mingw32",
      "url":"https://github.com/igrr/esptool-ck/releases/download/0.4.5/esptool-0.4.5-win32.zip",
      "archiveFileName":"esptool-0.4.5-win32.zip",
      "checksum":"SHA-256:1b0a7d254e74942d820a09281aa5dc2af1c8314ae5ee1a5abb0653d0580e531b",
      "size":"17408"
    },
    {
      "host":"x86_64-apple-darwin",
      "url":"https://github.com/igrr/esptool-ck/releases/download/0.4.5/esptool-0.4.5-osx.tar.gz",
      "archiveFileName":"esptool-0.4.5-osx.tar.gz",
      "checksum":"SHA-256:924d31c64f4bb9f748e70806dafbabb15e5eb80afcdde33715f3ec884be1652d",
      "size":"11359"
    }
  ]
}
]
}

```

```

{
    "host": "i386-apple-darwin",
    "url": "http://arduino.esp8266.com/esptool-0.4.5-1-gfaa5794-osx.tar.gz",
    "archiveFileName": "esptool-0.4.5-1-gfaa5794-osx.tar.gz",
    "checksum": "SHA-
256:722142071f6cf4d8c02dea42497a747e06abf583d86137a6a256b7db71dc61f6",
    "size": "20751"
},
{
    "host": "x86_64-pc-linux-gnu",
    "url": "https://github.com/igrr/esptool-ck/releases/download/0.4.5/esptool-
0.4.5-linux64.tar.gz",
    "archiveFileName": "esptool-0.4.5-linux64.tar.gz",
    "checksum": "SHA-
256:4ce799e13fdbd89f8a8f08a08db77dc3b1362c4486306fe1b3801dee80cfaf3203",
    "size": "12789"
},
{
    "host": "i686-pc-linux-gnu",
    "url": "https://github.com/igrr/esptool-ck/releases/download/0.4.5/esptool-
0.4.5-linux32.tar.gz",
    "archiveFileName": "esptool-0.4.5-linux32.tar.gz",
    "checksum": "SHA-
256:4aa81b97a470641771cf371e5d470ac92d3b177adbe8263c4aae66e607b67755",
    "size": "12044"
}
],
},
{
    "name": "xtensa-lx106-elf-gcc",
    "version": "1.20.0-26-gb404fb9",
    "systems": [
        {
            "host": "i686-mingw32",
            "url": "http://arduino.esp8266.com/win32-xtensa-lx106-elf-gb404fb9.tar.gz",
            "archiveFileName": "win32-xtensa-lx106-elf-gb404fb9.tar.gz",
            "checksum": "SHA-
56:1561ec85cc58cab35cc48bfdb0d0087809f89c043112a2c36b54251a13bf781f",
            "size": "153807368"
        },
        {
            "host": "x86_64-apple-darwin",
            "url": "http://arduino.esp8266.com/osx-xtensa-lx106-elf-gb404fb9-2.tar.gz",
            "archiveFileName": "osx-xtensa-lx106-elf-gb404fb9-2.tar.gz",
            "checksum": "SHA-
256:0cf150193997bd1355e0f49d3d49711730035257bc1aee1aaad619e56b9e4e6",
            "size": "35385382"
        },
        {
            "host": "i386-apple-darwin",
            "url": "http://arduino.esp8266.com/osx-xtensa-lx106-elf-gb404fb9-2.tar.gz",
            "archiveFileName": "osx-xtensa-lx106-elf-gb404fb9-2.tar.gz",
            "checksum": "SHA-
256:0cf150193997bd1355e0f49d3d49711730035257bc1aee1aaad619e56b9e4e6",
            "size": "35385382"
        }
    ]
}

```

```

        },
        {
            "host":"x86_64-pc-linux-gnu",
            "url":"http://arduino.esp8266.com/linux64-xtensa-lx106-elf-
gb404fb9.tar.gz",
            "archiveFileName":"linux64-xtensa-lx106-elf-gb404fb9.tar.gz",
            "checksum":"SHA-
256:46f057fdb8b320889a26167daf325038912096d09940b2a95489db92431473b7",
            "size":"30262903"
        },
        {
            "host":"i686-pc-linux-gnu",
            "url":"http://arduino.esp8266.com/linux32-xtensa-lx106-elf.tar.gz",
            "archiveFileName":"linux32-xtensa-lx106-elf.tar.gz",
            "checksum":"SHA-
256:b24817819f0078fb05895a640e806e0aca9aa96b47b80d2390ac8e2d9ddc955a",
            "size":"32734156"
        }
    ]
}
]
}
}

```

Breaking this down, we have one package in this file which has the following sections:

- `name` – **esp8266** – The name of the package itself
- `maintainer` – **ESP8266 Community** – Who maintains the package
- `websiteURL` – Where to go to find more about this package
- `email` – Who to email to find out more
- `help` – Where to go for on-line help
- `platforms` – The set of platforms on which this board runs
- `tools` – The details of required tools

For the platforms, we describe the details of each platform ... currently there is only one:

- `name` – **esp8266**
- `architecture` – **esp8266**
- `version` – The version ID of this package/platform
- `category` – **ESP8266**
- `url` – Where to download this platform
- `archiveFileName` – The name of the file

- `checksum` – A hash that can be used against the file to see if it has been tampered with
- `size` – The size in bytes of the file
- `help` – Where to read the docs for this platform
- `boards` – A list of boards that are associated with the platform.
- `toolDependencies` – The names of additional tools/components that are required

For the tools, this is a list of tools needed for the package. Each tool has the following:

- `name` – The logical name of the tool
- `version` – The version of the tool
- `systems` – A list of entries which define where to download the tool for a variety of platforms including Windows, Linux and OSx.

With this information and a copy of the file, you should be able to see how some of the pieces fit together.

When a package is installed, it is created in the directory:

```
C:\Users\<User>\AppData\Roaming\Arduino15\packages
```

For our ESP8266 story, the package is `esp8266` and hence all the files can be found in:

```
C:\Users\<User>\AppData\Roaming\Arduino15\packages
```

we will call this the root.

Beneath the root we will find two directories:

- `hardware`
- `tools`

The `tools` directory contains the root of our tools needed for execution ... these are the C compiler and the upload tool.

The `hardware` folder contains the rest of our information.

Specifically the following folders:

- `bootloaders` – a mystery ...
- `cores` – Core header and source files providing the code always linked with our sketches. This is the primary set of wrappers for the Arduino libraries.
- `tools` – The Espressif SDK

- `libraries` – The default libraries for our package
- `variants` – Header files that differ by variant of board selected

And the following files:

- `boards`
- `platform`
- `programmers`

Within the Arduino IDE we can switch on verbose settings which results in additional details being logged during compilation or upload. From these we can learn more about what happens.

If we examine a typical compilation statement, we find the following.

```
xtensa-lx106-elf-gcc
-D_ets_
-DICACHE_FLASH
-U__STRICT_ANSI__
-Itools/sdk//include
-c
-g
-x assembler-with-cpp
-MMD
-DF_CPU=80000000L
-DARDUINO=10605
-DARDUINO_ESP8266_ESP01
-DARDUINO_ARCH_ESP8266
-DESP8266
-Icores\esp8266
-Ivariants\generic\cores\esp8266\cont.S
-o cont.S.o

xtensa-lx106-elf-gcc
-D_ets_
-DICACHE_FLASH
-U__STRICT_ANSI__
-Itools/sdk//include
-c
-Os
-g
-Wpointer-arith
-Wno-implicit-function-declaration
-Wl,-EL
-fno-inline-functions
-nostdlib
-mlongcalls
-mtext-section-literals
-falign-functions=4
-MMD
-std=gnu99
```

```

-DF_CPU=80000000L
-DARDUINO=10605
-DARDUINO_ESP8266_ESP01
-DARDUINO_ARCH_ESP8266
-DESP8266
-Icores\esp8266
-Ivariants\generic
cores\esp8266\cont_util.c
-o cont_util.c.o

```

The contents of the core directory are the artifacts that are linked with your Arduino sketches.

spiffs	
abi.cpp	
Arduino.h	Primary include file for applications.
binary.h	Binary definitions for the range 0-255 up to 8 bits.
cbuf.h	Circular buffer.
Client.h	
cont.h	
cont.S	
cont_util.c	
core_esp8266_eboot_command.c	
core_esp8266_flash_utils.c	
core_esp8266_i2s.c	
core_esp8266_main.cpp	The main entry point into the ESP application.
core_esp8266_noniso.c	
core_esp8266_phy.c	
core_esp8266_postmortem.c	
core_esp8266_si2c.c	
core_esp8266_sigma_delta.c.unused	
core_esp8266_timer.c	
core_esp8266_wiring.c	
core_esp8266_wiring_analog.c	
core_esp8266_wiring_digital.c	
core_esp8266_wiring_pulse.c	
core_esp8266_wiring_pwm.c	
core_esp8266_wiring_shift.c	
debug.cpp	
debug.h	
eboot_command.h	
Esp.cpp	

Esp.h	
esp8266_peri.h	
flash_utils.h	
HardwareSerial.cpp	
HardwareSerial.h	
i2s.h	
IPAddress.cpp	
IPAddress.h	
libc_replacements.c	
pgmspace.cpp	
pgmspace.h	
Print.cpp	
Print.h	
Printable.h	
Server.h	
sigma_delta.h	
stdlib_noniso.h	
Stream.cpp	
Stream.h	
Tone.cpp	
twi.h	
Udp.h	
Updater.cpp	
Updater.h	
user_config.h	
Wcharacter.h	
wiring_private.h	
Wmath.cpp	
Wstring.cpp	
Wstring.h	

When we look at how an application is uploaded, we see a command similar to the following:

```
esptool.exe -vv -cd ck -cb 115200 -cp COM11 -ca 0x00000 -cf ESP_I2CScanner.cpp.bin
```

Building ESP Arduino apps using the Eclipse IDE

Now our heads are really going to hurt ... there is no easy way to get through this ... but the story is important and the results are great.

So far we have seen that we can build ESP programs using a C compiler and the Espressif SDK. We have also seen that we can build these programs within an Eclipse environment ... also against the Espressif SDK. We have just examined the notion of building programs using the Arduino IDE which provides mappings to many of the Arduino libraries implemented for ESP. Now we are going to return to using Eclipse but this time as an alternative to the Arduino IDE but still using the Arduino libraries to build "Arduino flavored" ESP programs.

The key to this story is the excellent Open Source Eclipse tooling for Arduino building found here:

<http://eclipse.baeyens.it/index.shtml>

This set of plug-ins to the Eclipse framework leverages an existing Arduino environment such as the one that we have just built which includes the ESP support. The plug-ins interrogate the Arduino IDE setup and provide the build and editing tools used there.

Before going any further, it is vital that you get the ESP Arduino IDE working by following all the instructions necessary to build solutions using that environment. That is a prerequisite for getting the Eclipse environment working.

We have two choices for getting the Eclipse environment working. The first is to download a fully prepared Eclipse environment that includes the C development tools **and** the plug-ins for Arduino support. This is the easiest ... however it is also likely that you have an existing Eclipse framework already installed that you may wish to re-use or extend. Eclipse is meant to be an extensible environment which provides a framework into which additional plug-ins can be added as needed. In that pattern, we can download the latest Eclipse framework (Mars) and then add in the relevant plug-ins.

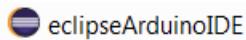
Here is an example of getting ready using a pre-built Eclipse download.

First download a current overnight build ... and extract its content. Note that the maintainer is distributing files in tar.gz format. I use [7-Zip](#) to decompress. The download size is about 170MBytes so make sure you download sooner than later. Make sure that you download the correct version of the Eclipse environment that corresponds to the version of Java you have installed. For example, if you have 32bit Java installed, don't download the 64bit version of Eclipse. If you do and attempt to launch Eclipse, you will get errors that you will have to dig into only to find buried in logs that there is an incompatibility. If you do make the error, the message you receive might look like:

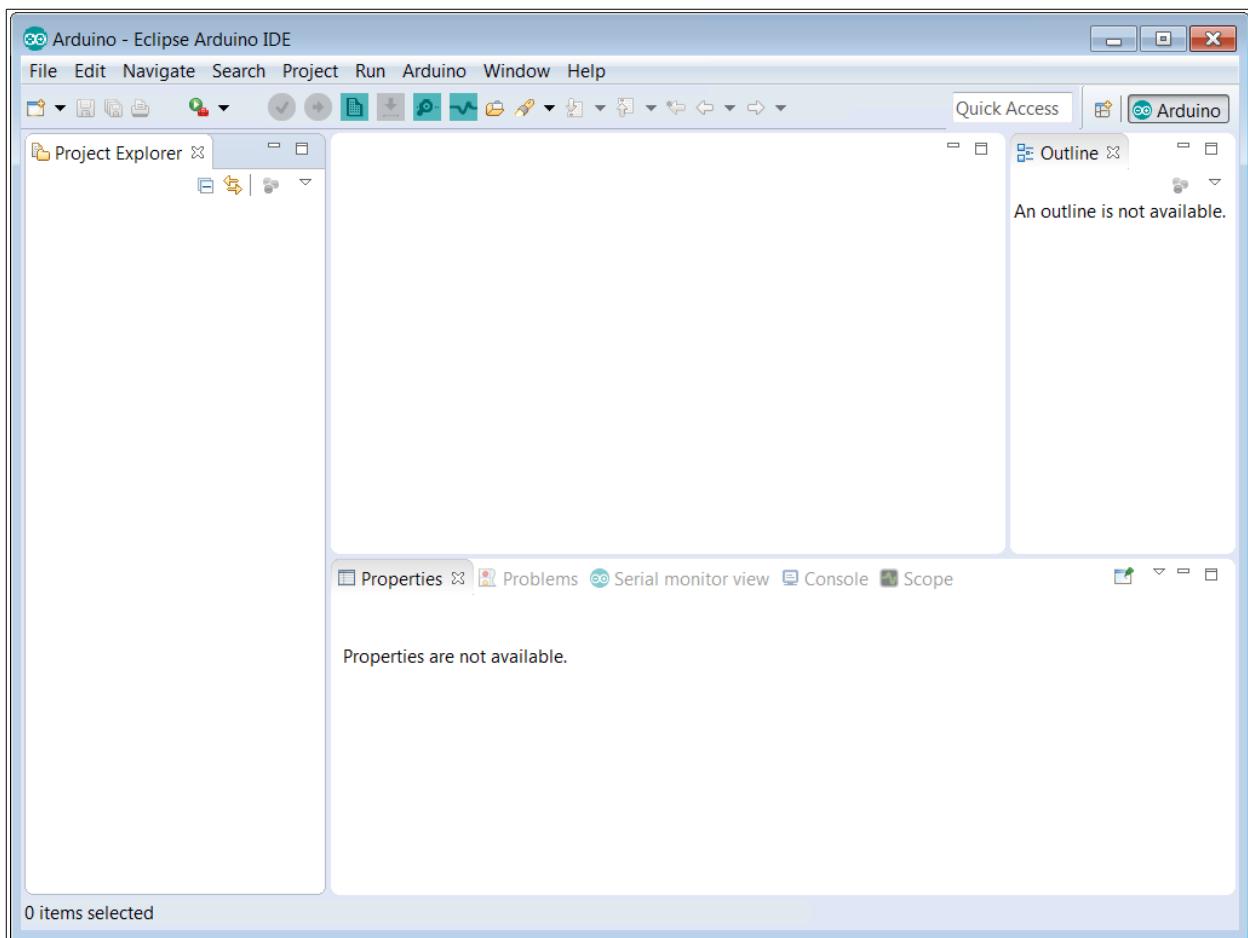


As you can see, it isn't readily apparent what went wrong.

When you are ready to launch, start the program called "`eclipseArduinoIDE`".



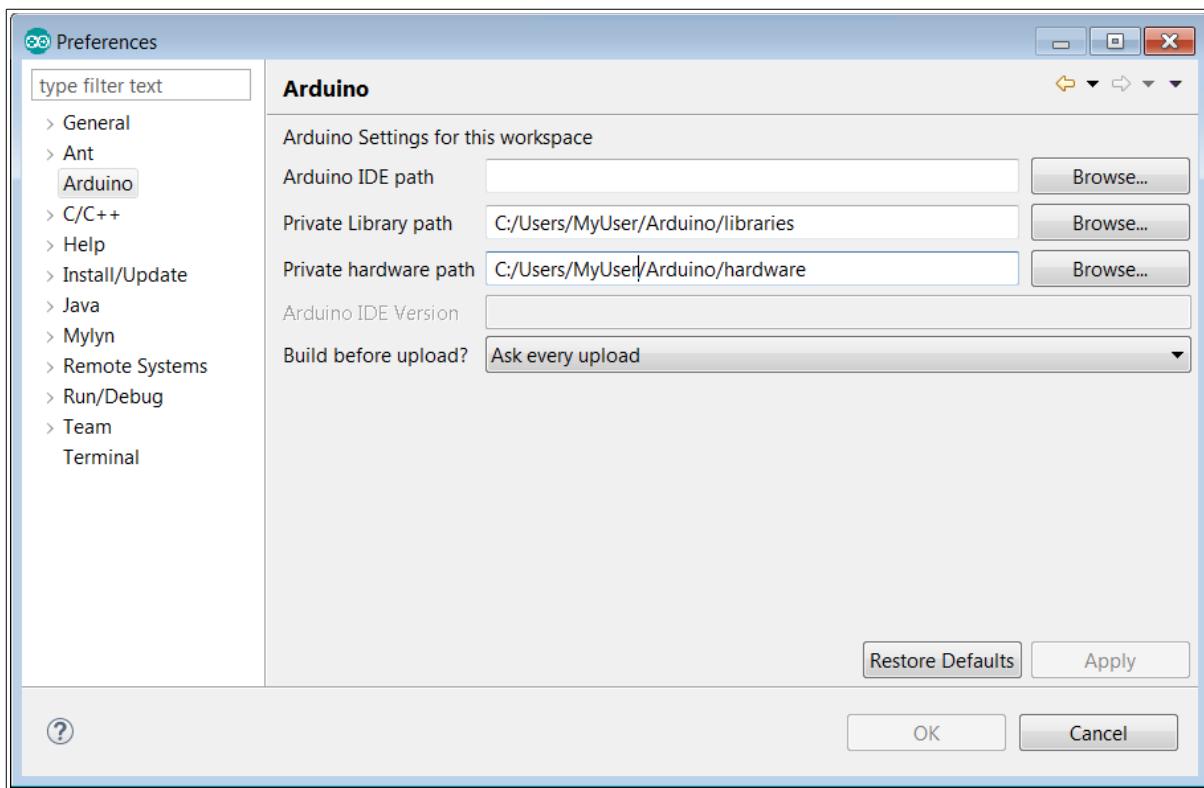
If all has gone well, we will see the following:



Now it is time to configure the Eclipse environment for to learn about our Arduino environment. The recipes that follow are used to overcome some bugs so may change over time ...

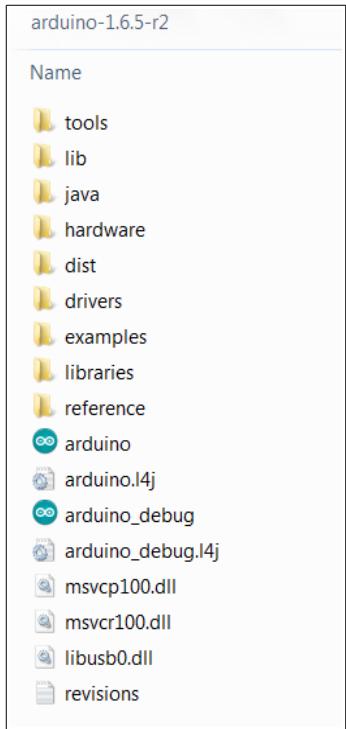
First, we need to tell Eclipse where it can find our Arduino environment.

Open up Preferences and select Arduino:



We need to change some of the settings.

For the Arduino IDE path, point to the root directory where your Arduino IDE is installed. This should be the directory which contains the following:



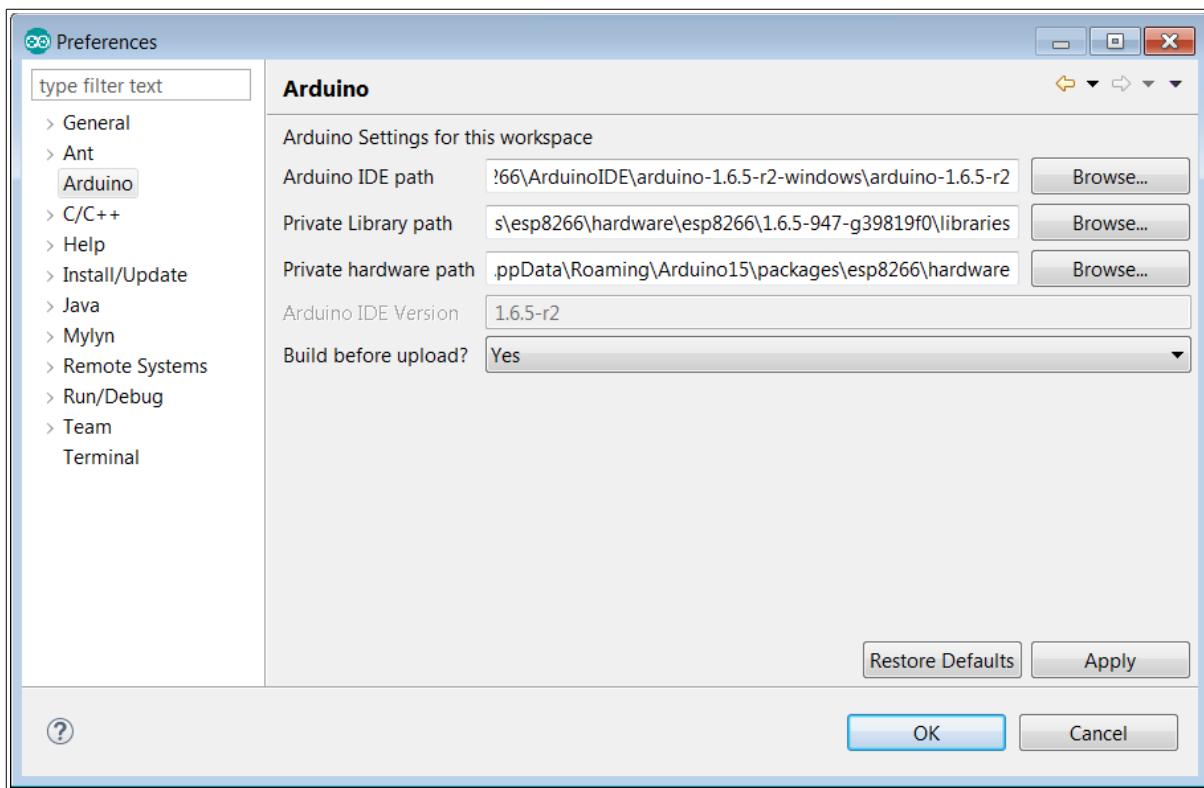
The next part is a little trickier. We need to supply values for both "Private Library path" and "Private hardware path". These directories are the directories for the Arduino ESP package and **NOT** the native Arduino. You will find these at the following directories:

```
C:\Users\<Your  
Userid>\AppData\Roaming\Arduino15\packages\esp8266\hardware\esp8266\<Your  
Version>\libraries
```

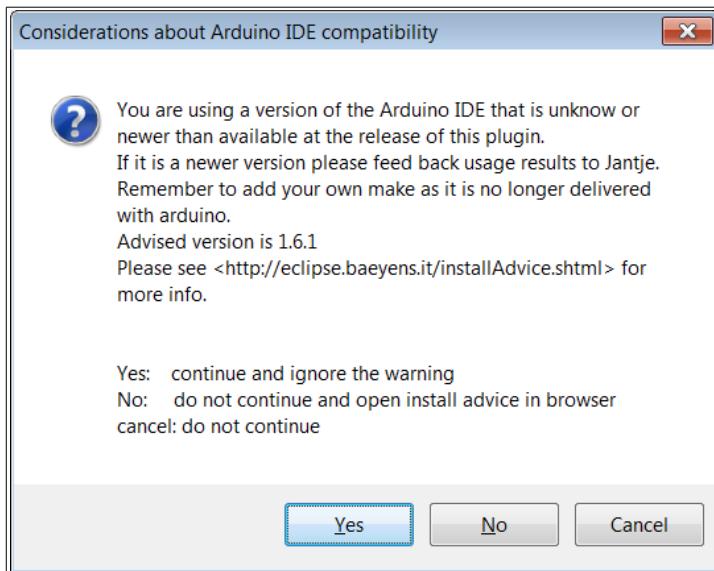
```
C:\Users\<Your Userid>AppData\Roaming\Arduino15\packages\esp8266\hardware
```

I also recommend you change your "Build before upload" to be "Yes" at this point.

After entering, your screen might look like.

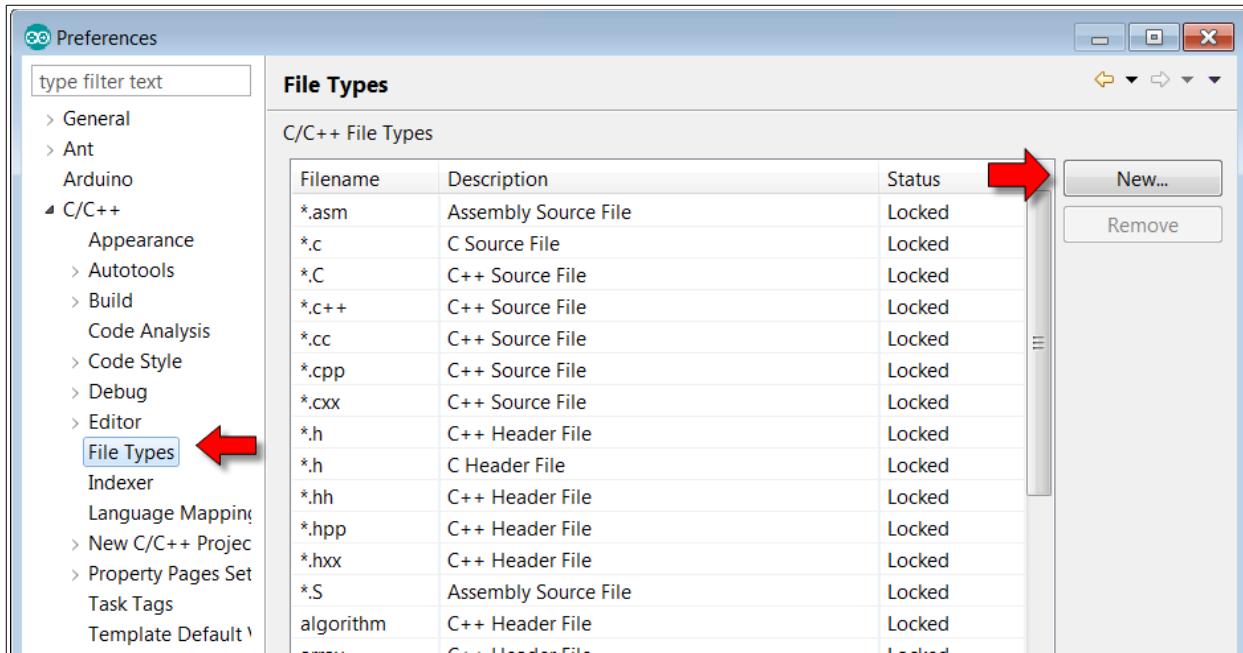


When applying the changes you made here, you may see the following warning a few times:

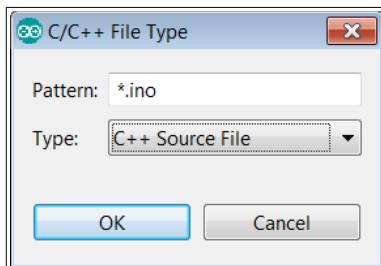


Click "Yes" to ignore the warnings and move on.

Next, we have to add "`*.ino`" as a new C++ file type ... we do this again in preferences:



In the resulting dialog, add the following:



We are getting close. Now we have a few final one-time tweaks to make. Find the file called "platform.txt" which is located at:

```
C:\Users\<Your  
Userid>\AppData\Roaming\Arduino15\packages\esp8266\hardware\esp8266\<Version>\platform  
.txt
```

Edit this file with your text editor and find the line which reads:

```
tools.esptool.upload.pattern="{path}/{cmd}" {upload.verbose} -cd {upload.resetmethod}  
-cb {upload.speed} -cp "{serial.port}" -ca 0x00000 -cf "{build.path}/  
{build.project_name}.bin"
```

and change it to be:

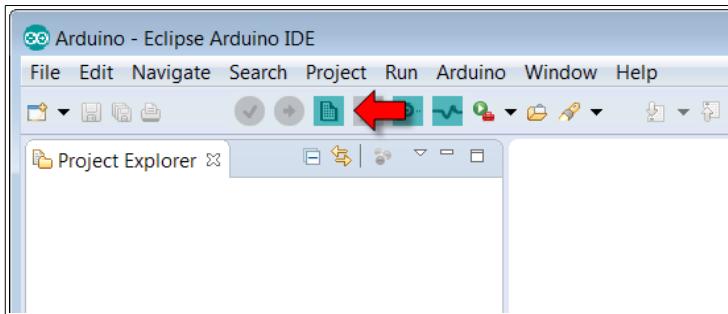
```
tools.esptool.upload.pattern="{path}/{cmd}" -vv -cd {upload.resetmethod} -cb  
{upload.speed} -cp "{serial.port}" -ca 0x00000 -cf "{build.path}/  
{build.project_name}.bin"
```

(This is we change "{upload.verbose}" to "-vv")

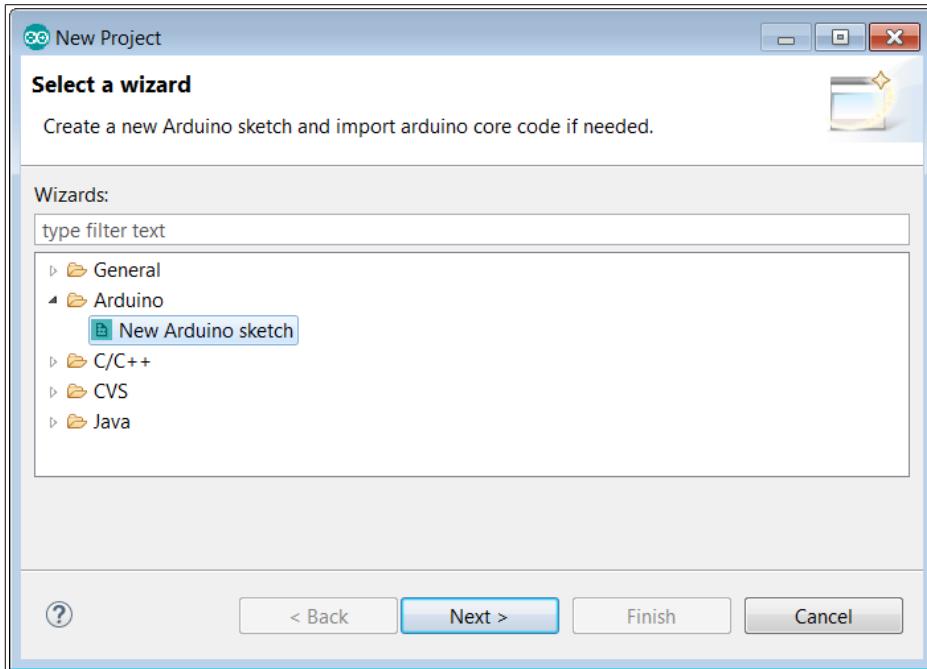
Save the file.

We are now ready to build and use our environment.

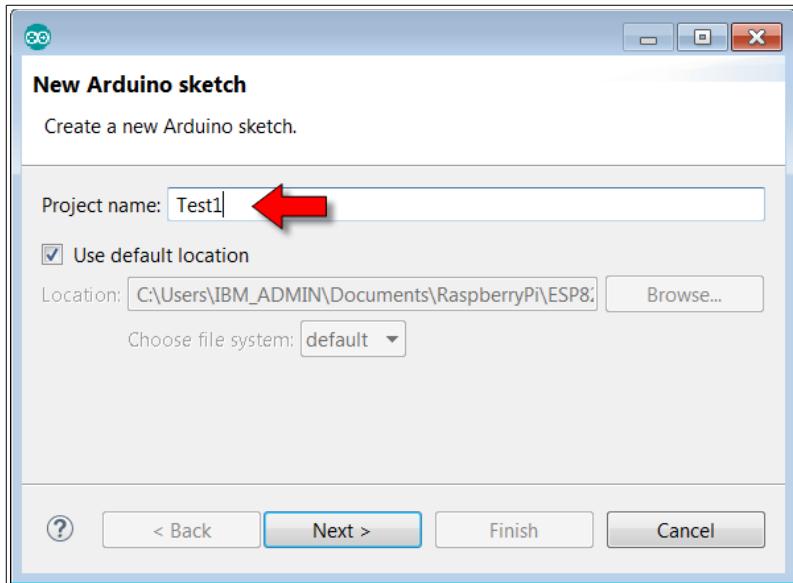
From the main window, create a new "Sketch".



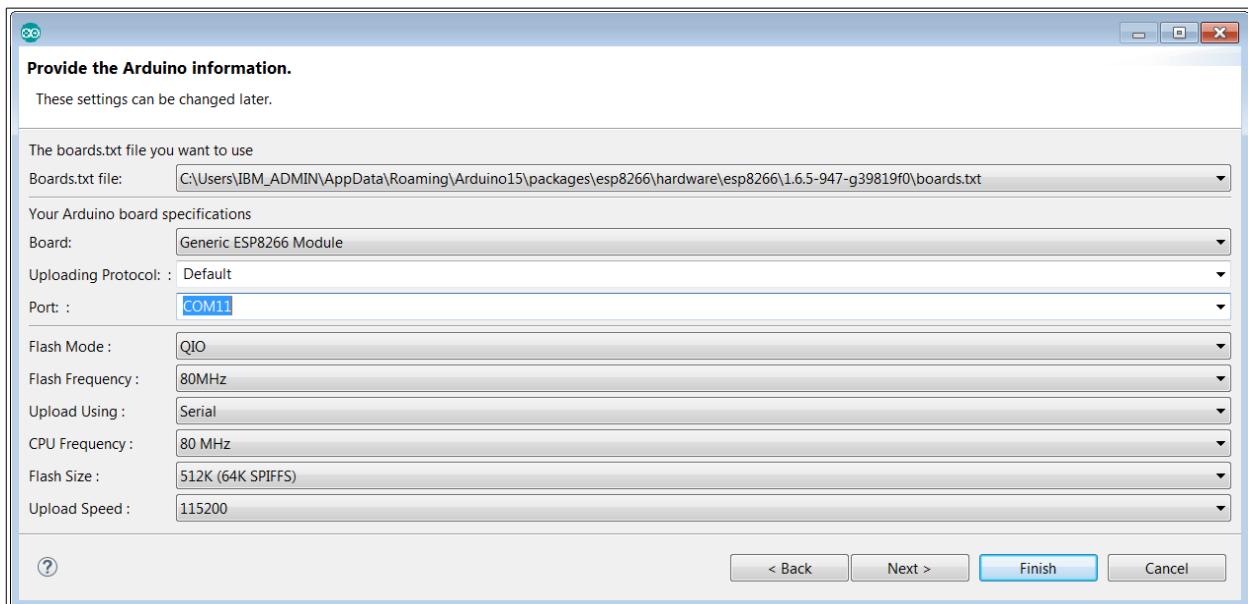
Note that we can also do this through the standard Eclipse > New Project



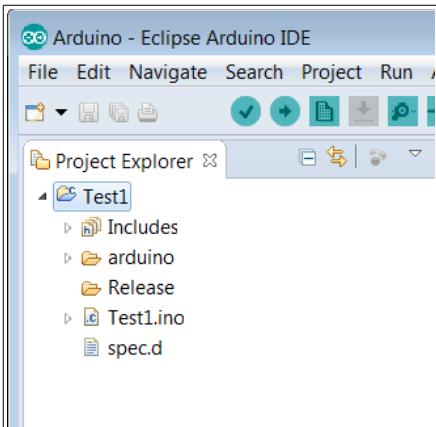
The first page of the project creation wizard is the name we wish to give our project:



Next we supply some core settings. Take your time over these. The one that will likely differ for you is the "Port:" which is the serial port used to flash your device:



At the completion of the wizard (assuming you took the rest of the options as defaults) ... you will have a project that looks like:



Edit the `Test1.ino` C++ source file and add your code.

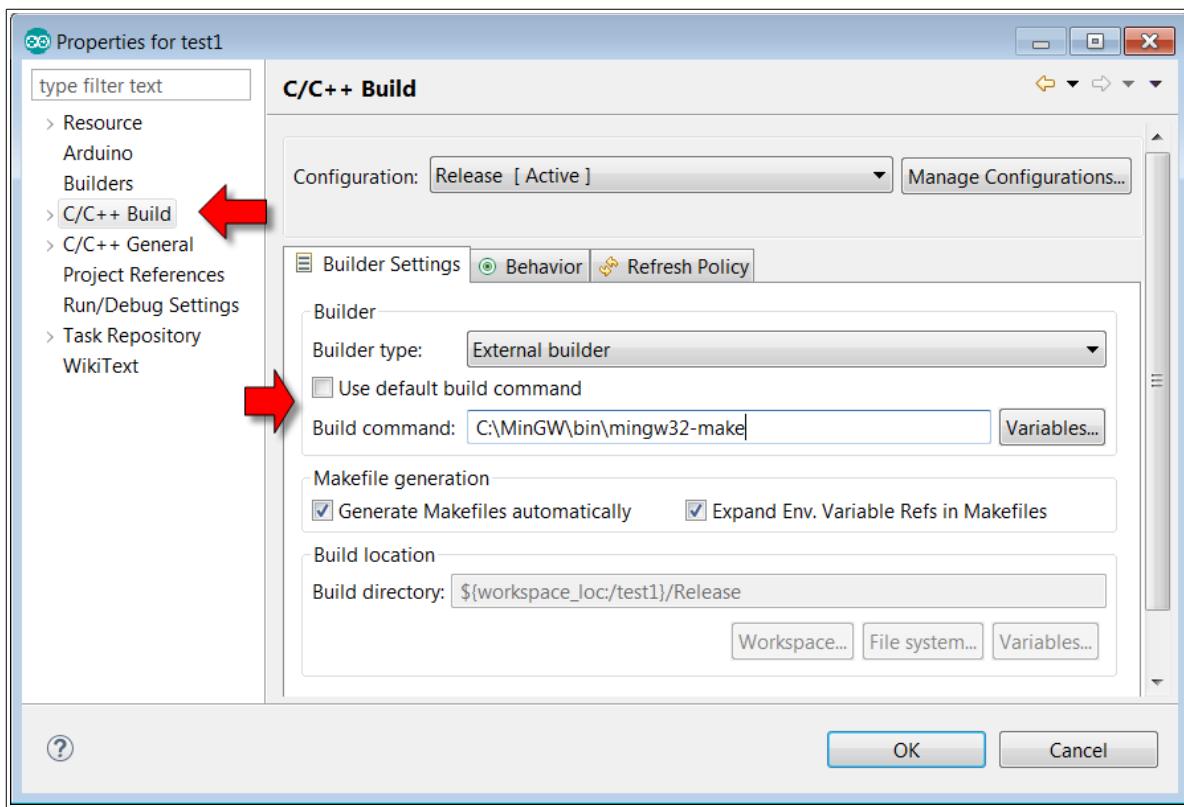
```
//The setup function is called once at startup of the sketch
void setup()
{
    Serial1.begin(115200);
    Serial1.println("It worked!");
    // Add your initialization code here
}

// The loop function is called in an endless loop
void loop()
{
    //Add your repeated code here
}
```

Here you will see your pay-off. You are now editing your source in the professional C/C++ editor that is part of Eclipse. This includes entry assist, syntax checking and highlighting (and more).

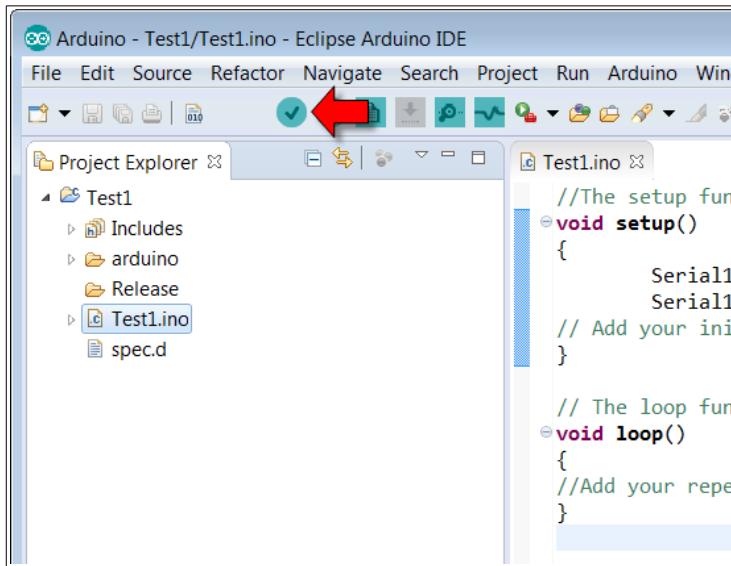
Before you can compile your project, you need to change the project specific settings to tell the project where to find your `make` program. In my environment I am using "mingw32-make". You can see where to make the changes in the following screen shot.

Note: There has to be a better way than this ... but I wanted to get this recipe out rather than hold everyone up while I tinkered with this small nugget:



And finally, we can compile our program.

Click the "Verify" icon:



Now the compilation will take place. For your first build of this project, **all** the source code of all the libraries will be built. Future builds will just compile what has changed. On my machine the build took 51 seconds:

The screenshot shows the Eclipse CDT Build Console window. It displays a table of compilation statistics and a message indicating the build was finished. A red arrow points to the build time.

```

CDT Build Console [Test1]
.xt.prop._ZNK4cbuf4roomEv      72      0
.xt.prop._ZN4cbuf4readEv       60      0
.xt.prop._ZN4cbuf5writeEc     36      0
.xt.prop._ZTV6Stream          12      0
.xt.prop._ZTV14HardwareSerial 12      0
.xt.lit._ZN5Print5writeEPKc   8       0
.xt.prop._ZN5Print5writeEPKc  48      0
.xt.prop._ZTV5Print           12      0
.debug_loc                     16268    0
Total                           527405

'Finished building target: Test1'
'

20:13:53 Build Finished (took 51s.74ms)

```

However, when I now edit my project file (`Test1.ino`) and recompile, the re-build only takes 4 seconds as only the files that have changed need to be recompiled.

The screenshot shows the Eclipse CDT Build Console window. It displays a message indicating the build was finished. A red arrow points to the build time.

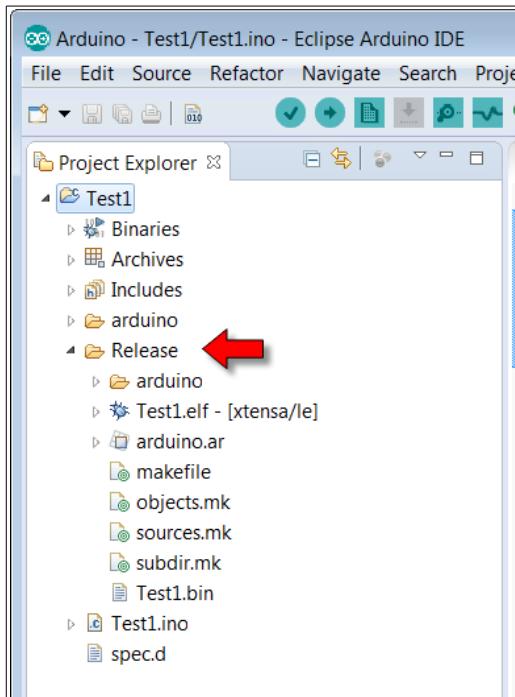
```

'Finished building target: Test1'
'

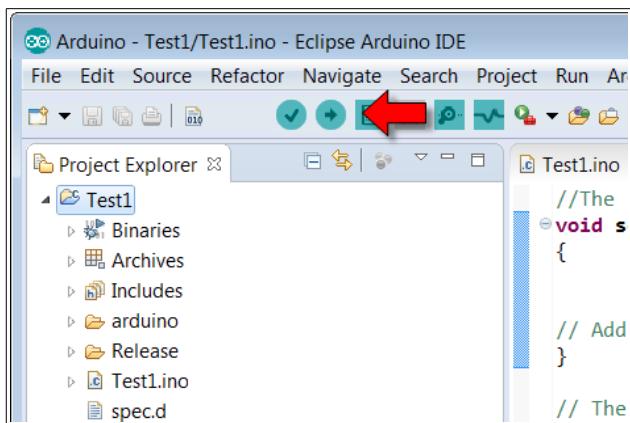
20:14:44 Build Finished (took 4s.880ms)

```

Following a build, a new directory called "Release" can be found which contains all the artifacts that were compiled. If you want to force a re-build of all, simply delete "Release".



Now that you have build the program, you can upload it with the upload icon:



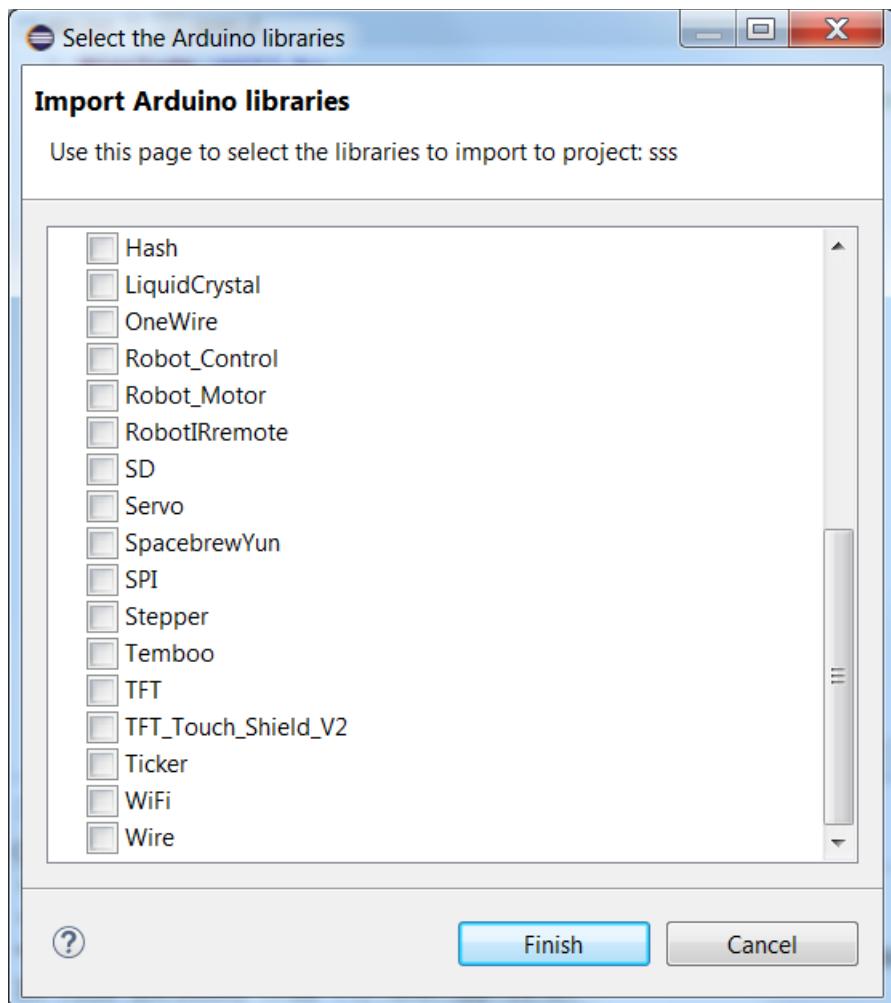
This will upload the executable. Unfortunately, there isn't much to see while the upload happens so sit back and be patient. If you have a USB → UART connector attached to UART1 of the ESP8266, you will see the upload progress ... but that is not essential (though I recommend it).

And ... that is it. You are now building ESP8266 applications using Arduino libraries in an Eclipse environment.

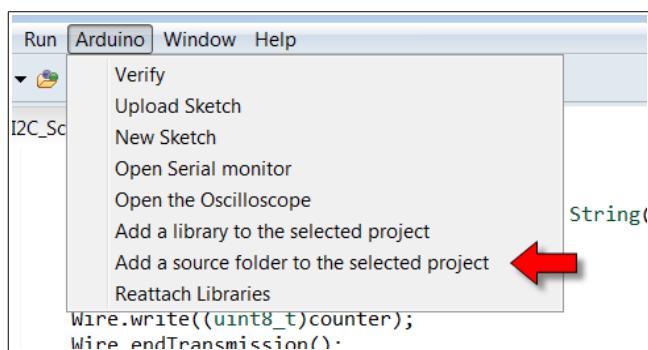
If we wish to add one of the supplied libraries, we can select the library to include with:

```
Arduino > Add a library to the selected project
```

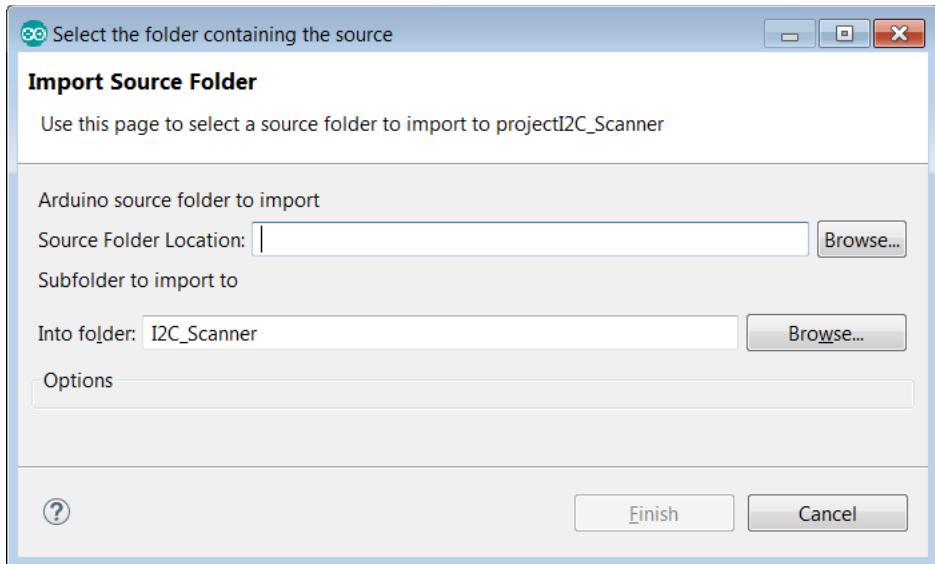
From there, we can select one of the libraries:



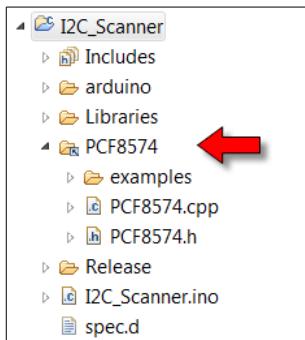
We can add external libraries that exist a source files on the file system. From the Arduino menu, select "Add a source folder to the selected project".



A dialog will be presented where one can select the directory to be included in the build.



Once added it becomes a linked directory in the project and the contents of the directory will be compiled and linked.



Note: The preceding recipe was based upon the Arduino IDE 1.6.5-r2, the stable version of Arduino for ESP8266 as of 2015-07-23 and the 2015-08-05 build of Arduino Eclipse.

See also:

- Github: [esp8266/Arduino](#)
- [Arduino – Eclipse](#)

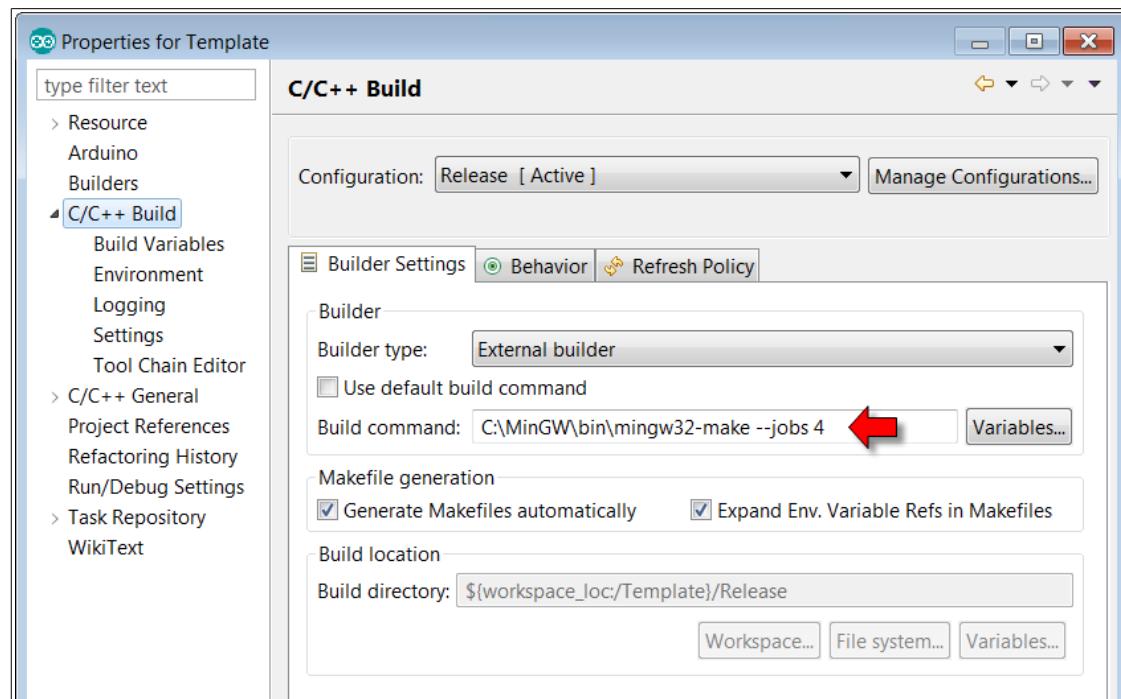
Reasons to consider using Eclipse over Arduino IDE

As previously mentioned, there is no question that the Arduino IDE is much more friendly and consumable than the professional Eclipse environment for folks new to the area. There doesn't appear to be anything that one can't build using the Arduino IDE that would mean one would have to switch to Eclipse. So why then would one ever consider using Eclipse?

There is a trade-off between ease of use and richness of function. For example, Eclipse has built in syntax assistance, error checking, code cross references, refactoring and much more. None of these things are "essential" but any one of them can be considered to make a programming job easier if and when needed. If I need to rename a variable, in Arduino IDE I have to manually find and replace each occurrence. In Eclipse, I can re-factor the variable using a built-in wizard and the IDE does the work for me. As another example, if I can't remember the syntax for a method, in Arduino IDE I would go to the web and look it up while in Eclipse I could type the name of the method and hover my mouse over it and the tooling will show me the possible options for the parameters.

Notes on using the Eclipse Arduino package

- Do **not** create Eclipse projects that have spaces in their names. This confuses the compiler.
- If you are compiling on a multi-core machine, you can cause the compilations of the source files to progress in parallel using by adding the "--jobs <num>" parameter to your `make` command. This will cause `make` to execute some number of jobs in parallel. For example, if you have a 4 core machine, setting `num` to be 4 might be a good start. This flag works with the GNU `make` tool.



- At times we wish to write sketches that work on both a real Arduino and an ESP8266 but the code has to be slightly different. We can include code for both

architectures by using the #defines called ARDUINO_ARCH_ESP8266 and/or ESP8266. Using these we can #if/#endif sections based on the architecture we are compiling against.

- As of 2015-08-08, attempting to use native SDK function in the Arduino Eclipse environment does not work if your source files are "*.ino". It appears that "user_include.h" is included automatically with C++ function name mangling in effect.
- I recommend renaming any "*.ino" files in your project to "*.cpp".

Arduino ESP Libraries

The WiFi library

The Arduino has a WiFi library for use with its WiFi shield. A library with a similar interface has been supplied for the Arduino environment for the ESP8266.

To use the ESP8266 WiFi library you must include its header:

```
#include <ESP8266WiFi.h>
```

To be a station and connect to an access point, execute a call to WiFi.begin(ssid, password). Now we need to poll WiFi.status(). When this returns WL_CONNECTED, then we are connected to the network.

To set up an access point, we would call WiFi.softAP() supplying the ssid and password information.

Here is an example of us connecting as a station:

```
WiFi.mode(WIFI_STA);
WiFi.begin(SSID, PASSWORD);
if (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial1.println("Failed");
    return;
}
WiFi.printDiag(Serial1);
// We are now connected as a station
```

See also:

- WiFiClient
- WiFiServer
- [Arduino WiFi library](#)

WiFi.begin

Start a WiFi connection as a station.

```
int begin(
    const char *ssid,
```

```
const char *passPhrase=NULL,
int32_t channel=0,
uint8_t bssid[6]=NULL)

int begin(
char *ssid,
char *passPhrase=NULL,
int32_t channel=0,
uint8_t bssid[6]=NULL)
```

Begin a WiFi connection as a station. The `ssid` parameter is mandatory but the others can be left as default. The return value is our current connection status.

[WiFi.beingSmartConfig](#)

`bool beginSmartConfig()`

[WiFi.beginWPSConfig](#)

`bool beginWPSConfig()`

[WiFi.BSSID](#)

Retrieve the current BSSID.

```
uint8_t BSSID()
uint8_t *BSSID(uint8_t networkItem)
```

Retrieve the current BSSID.

[WiFi.BSSIDstr](#)

Retrieve the current BSSID as a string representation.

```
String BSSIDstr()
String BSSIDstr(uint8_t networkItem)
```

Retrieve the current BSSID as a string representation.

[WiFi channel](#)

Retrieve the current channel.

```
int32_t channel()
int32_t channel(uint8_t networkItem)
```

Retrieve the current channel.

WiFi.config

Set the WiFi connection configuration.

```
void config(IPAddress local_ip, IPAddress gateway, IPAddress subnet)
void config(IPAddress local_ip, IPAddress gateway, IPAddress subnet, IPAddress dns)
```

Set the configuration of the WiFi using static parameters. This disables DHCP.

WiFi.disconnect

Disconnect from an access point.

```
int disconnect(bool wifiOff = false)
```

Disconnect from the current access point.

WiFi.encryptionType

Return the encryption type of the scanned WiFi access point.

```
uint8_t encryptionType(uint8_t networkItem)
```

Return the encryption type of the scanned WiFi access point.

The values are one of:

- ENC_TYPE_NONE
- ENC_TYPE_WEP
- ENC_TYPE_TKIP
- ENC_TYPE_CCMP
- ENC_TYPE_AUTO

WiFi.gatewayIP

Get the IP address of the station gateway.

```
IPAddress gatewayIP()
```

Retrieve the IP address of the station gateway.

WiFi.getNetworkInfo

Retrieve all the details of the specified scanned networkItem.

```
bool getNetworkInfo(uint8_t networkItem,
                    String &ssid,
                    uint8_t &encryptionType,
                    int32_t &RSSI,
                    uint8_t *&BSSID,
```

```
    uint32_t &channel,  
    bool &isHidden)
```

Retrieve all the details of the specified scanned networkItem.

WiFi.hostByName

Lookup a host by a name.

```
int hostByName(const char *hostName, IPAddress &result)
```

Look up a host by name and get its IP address. This function returns 1 on success and 0 on failure.

WiFi.hostname

Retrieve and set the hostname used by this station.

```
String hostname()  
bool hostname(char *hostName)  
bool hostname(const char *hostName)  
bool hostname(String hostName)
```

WiFi.isHidden

Determine if the scanned network item is flagged as hidden.

```
bool isHidden(uint8_t networkItem)
```

Determine if the scanned network item is flagged as hidden.

WiFi.localIP

Get the station IP address.

```
IPAddress localIP()
```

Get the IP address for the station. There is a separate IP address if the ESP is an access point.

See also:

- WiFi.softAPIP

WiFi.macAddress

Get the station interface MAC address.

```
uint_t *macAddress(uint8_t *mac)  
String macAddress()
```

Get the station interface MAC address.

WiFi.mode

Set the operating mode.

```
void mode(WiFiMode mode)
```

Set the operating mode of the WiFi. This is one of:

- `WIFI_OFF` – Switch off WiFi
- `WIFI_STA` – Be a WiFi station
- `WIFI_AP` – Be a WiFi access point
- `WIFI_AP_STA` – Be both a WiFi station and a WiFi access point

See also:

- Defining the operating mode

WiFi.printDiag

Log the state of the WiFi connection.

```
void printDiag(Print &dest)
```

Log the state of the WiFi connection. We can pass in either Serial or Serial1 as an argument to log the data to the Serial port. An example of output is as shown next:

```
Mode: STA
PHY mode: N
Channel: 7
AP id: 0
Status: 5
Auto connect: 0
SSID (7): yourSSID
Passphrase (8): yourPassword
BSSID set: 0
```

Note that the status value is the result of a `wifi_station_get_connect_status()` call.

WiFi.RSSI

Retrieve the RSSI (Received Signal Strength Indicator) value of the scanned network item.

```
int32_t RSSI(uint8_t networkItem)
```

Retrieve the RSSI value of the scanned network item.

WiFi.scanComplete

Determine the status of a previous scan request.

```
int8_t scanComplete()
```

If the result is ≥ 0 then this is the number of WiFi access points found. Otherwise, the value is less than 0 and the codes are:

- SCAN_RUNNING – A scan is currently in progress.
- SCAN_FAILD – A scan failed.

See also:

- WiFi.scanDelete
- WiFi.scanDelete

WiFi.scanDelete

Delete the results from a previous scan.

```
void scanDelete()
```

Delete the results from a previous scan. A request to scan the network results in the allocation of memory. This call releases that memory.

See also:

- WiFi.scanComplete
- WiFi.scanDelete

WiFi.scanNetworks

Scan the access points in the environment.

```
int8_t scanNetworks(bool async = false)
```

Scan the access points in the environment. We can either perform this synchronous or asynchronous. On a synchronous call, the result is the number of access points found.

See also:

- WiFi.scanComplete
- WiFi.scanDelete

WiFi.smartConfigDone

```
bool smartConfigDone()
```

WiFi.softAP

Setup an access point.

```
void softAP(const char *ssid)
void softAP(const char *ssid,
            const char *passPhrase,
            int channel=1,
            int ssid_hidden=0)
```

The `ssid` is used to advertize our network. The `passPhrase` is the password a station must supply in order to be authorized to access.

WiFi.softAPConfig

```
void softAPConfig(IPAddress local_ip, IPAddress gateway, IPAddress subnet)
```

WiFi.softAPdisconnect

```
int softAPdisconnect(bool wifiOff=false)
```

WiFi.softAPmacAddress

Get the MAC address of the access point interface.

```
uint8_t *softAPmacAddress(uint8_t *mac)
```

Get the MAC address of the access point interface.

WiFi.softAPIP

Get the IP address of the access point interface.

```
IPAddress softAPIP()
```

Return the IP address of the access point interface. There is a separate IP for the station.

See also:

- `WiFi.localIP`

WiFi.SSID

Retrieve the SSID.

```
char *SSID()  
const char *SSID(uint8_t networkItem)
```

Here we retrieve the SSID of the current station or the SSID of the scanned network id.

WiFi.status

Retrieve the current WiFi status.

```
wl_status_t status()
```

The status returned will be one of:

- `WL_IDLE_STATUS (0)`

- WL_NO_SSID_AVAIL (1)
- WL_SCAN_COMPLETED (2)
- WL_CONNECTED (3)
- WL_CONNECT_FAILED (4)
- WL_CONNECTION_LOST (5)
- WL_DISCONNECTED (6)

WiFi.stopSmartConfig

void stopSmartConfig()

WiFi.subnetMask

IPAddress subnetMask()

WiFi.waitForConnectResult

Wait until the WiFi connection has been formed or failed.

```
uint8_t waitForConnectResult()
```

If we are a station, then block waiting for us to become connected or failed. The return code is the status. Specifically, this function watches the status to see when it becomes something other than WL_DISCONNECTED. Perhaps a more positive form of this function would be:

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

WiFiClient

This library provides TCP connections to a partner. A separate class provides UDP communications.

To use this library, you must include "ESP8266WiFi.h".

We create an instance of this class and then connect to a partner using the `connect()` method.

WiFiClient

WiFiClient.available

Return the amount of data available to be read.l15summe

```
int available()
```

Return the amount of data available to be read.

WiFiClient.connect

Connect to the given host at the given port using TCP.

```
int connect(const char* host, uint16_t port)
int connect(IPAddress ip, uint16_t port)
```

Connect to the given host at the given port using TCP. This function returns 0 on a failure.

WiFiClient.connected

Determine if we are connected to a partner.

```
uint8_t connected()
```

Return true if connected and false otherwise.

WiFiClient.flush

```
void flush()
```

WiFiClient.getNoDelay

```
bool getNoDelay()
```

WiFiClient.peek

```
int peek()
```

WiFiClient.read

Read data from the partner.

```
int read()
int read(uint8_t *buf, size_t size)
```

Read data from the partner. These functions read either a single byte or a sequence of bytes from the partner.

WiFiClient.remoteIP

Retrieve the remote IP address of the connection.

```
IPAddress remoteIP()
```

Retrieve the remote IP address of the connection.

[WiFiClient.remotePort](#)

Return the remote port being used in an existing connection.

```
uint16_t remotePort()
```

Return the remote port being used in an existing connection.

[WiFiClient.setLocalPortStart](#)

Set the initial port for allocating local ports for connections.

```
void setLocalPortStart(uint16_t port)
```

Set the initial port for allocating local ports for connections.

[WiFiClient.setNoDelay](#)

```
void setNoDelay(bool nodelay)
```

[WiFiClient.status](#)

```
uint8_t status()
```

[WiFiClient.stop](#)

Disconnect a client.

```
void stop()
```

Disconnect a client.

[WiFiClient.stopAll](#)

Stop all the connections formed by this WiFi client.

```
void stopAll()
```

[WiFiClient.write](#)

Write data to the partner.

```
size_t write(uint8_t b)
size_t write(const uint8_t *buf, size_t size)
size_t write(T& source, size_t unitSize);
```

Write data to the partner. The first function writes one byte, while the second function writes an array of characters.

WiFiServer

WiFiServer

Create an instance of a Server listening on the supplied port.

```
WiFiServer(uint16_t port)
```

Create an instance of a Server listening on the supplied port. Interestingly, it appears that once we create a server instance within an ESP8266, there is no way to stop it running.

WiFiServer.available

Retrieve a WiFiClient object that can be used for communications.

```
WiFiClient available(byte* status)
```

Retrieve the corresponding WiFiClient.

See also:

- WiFiClient

WiFiServer.begin

Start listening for incoming connections.

```
void begin()
```

Start listening for incoming connections. Until this method is called, the ESP8266 doesn't accept incoming connections. Interestingly, once called, there is no obvious way to stop listening. The port used for the incoming connections is the one supplied when the WiFiServer object was constructed.

WiFiServer.getNoDelay

WiFiServer.hasClient

Return true if we have a client connected.

```
bool hasClient()
```

WiFiServer.setNoDelay

WiFiServer.status

WiFiServer.write

WARNING!! This method is not implemented.

```
size_t write(uint8_t b)
size_t write(const uint8_t *buffer, size_t size)
```

Although present on the interface, this method is not yet implemented.

IPAddress

A representation of an IPAddress. This class has some operator overrides:

[i] – Get the ith byte of the address. I should be 0-3.

ESP8266WebServer

The ESP8266WebServer class provides the core implementation of an HTTP server. This is software that responds to browser requests. To use this class we create an instance of an ESP8266WebServer object specifying the TCP port number on which it will listen. The default port for browsers is port 80 so this is a good choice.

Once the object has been created, we define one or more callback functions that will be invoked when a browser connection is received. The function called `on()` is used to register these. These callback functions are keyed on the URL path requested by the browser. For example, if our ESP8266 is running at IP address 192.168.1.2 and the browser URL is:

```
http://192.168.1.2/index.html
```

The the URL path will be "/index.html".

If we wish to send a response to a request at that URL, we would register a callback function using that path as a key.

For example:

```
myServer.on("/index.html", myFunction);
```

where `myFunction` is a C function with the signature:

```
void myFunction()
```

The callback function, when called, can use the ESP8266WebServer object to execute a `send()` method call to send a response.

If a browser request arrives for a URL path that is not explicitly handled, a call to a callback function registered with the `onNotFound()` method is invoked. This can serve as a catch-all for processing.

When a URL contains query properties of the form "`x=y`", the number, names and values of these properties are available in the `args()`, `argName()` and `arg()` functions. Note that URL encoding is not currently supported so data can not yet contain URL invalid characters.

When a request from a browser is received, one wants to send back a response and the way to achieve that is through an invocation of the `send()` method. This takes the response code to the browser (200 for OK), the MIME encoding type and the payload of the data as parameters.

When you send a request from a browser to a Web server, anticipate an extra HTTP GET request that wishes to retrieve a file called "`/favicon.ico`" which is used to specify an icon that represents the web site being accessed. To handle this, we might wish to add a handler function that looks as follows:

```
webServer.on("/favicon.ico", []() {
    webServer.send(404, "text/plain", "");
});
```

This registers a handler for the icon file and sends back a 404 (not found) response to the browser. Notice the use of an in-line anonymous function in C++. Your choice to use this style of coding is your own. Personally, I prefer explicitly declared functions.

Here is an example of a Web server app:

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>

const char *ssid="mySsid";
const char *password="myPassword+";

ESP8266WebServer webServer(80);

void logDetails();

void testHandler() {
    Serial1.println("testHandler");
    logDetails();
    webServer.send(200, "text/plain", "Here is our response: " + String(millis()));
}

void notFoundHandler() {
    Serial1.println("Not Found Handler");
    logDetails();
}
```

```

String methodToString(HTTPEMethod method) {
    switch(method) {

        case HTTP_GET:
            return "GET";
        case HTTP_POST:
            return "POST";
        case HTTP_PUT:
            return "PUT";
        case HTTP_PATCH:
            return "PATCH";
        case HTTP_DELETE:
            return "DELETE";
    }
    return "Unknown";
}

void logDetails() {
    Serial1.println("URL is: " + webServer.uri());
    Serial1.println("HTTP Method on request was: " +
methodToString(webServer.method()));

    // Print how many properties we received and then print their names
    // and values.
    Serial1.println("Number of query properties: " + String(webServer.args()));
    int i;
    for (i=0; i<webServer.args(); i++) {
        Serial1.println(" - " + webServer.argName(i) + " = " + webServer.arg(i));
    }
}

void setup() {
    Serial1.begin(115200);
    Serial1.println("Starting ...");
    WiFi.begin(ssid, password);
    // Wait for connection
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial1.print(".");
    }
    Serial1.println("");
    Serial1.print("Connected to ");
    Serial1.println(ssid);
    Serial1.print("IP address: ");
    Serial1.println(WiFi.localIP());
    webServer.on("/test", testHandler);
    webServer.on("/favicon.ico", []() {
        webServer.send(404, "text/plain", "");
    });

    webServer.onNotFound(notFoundHandler);
    webServer.begin();
    Serial1.println("We have started the Web Server");
}

```

```
void loop() {
    webServer.handleClient();
}
```

See also:

- Wikipedia: [Favicon](#)
- `ESP8266WebServer.on`
- `ESP8266WebServer.send`

[ESP8266WebServer](#)

Construct an instance of a WebServer object.

```
ESP8266WebServer::ESP8266WebServer(int port)
```

Construct an instance of the class. The `port` number supplied is the port that will be listened upon for incoming browser requests.

[ESP8266WebServer.arg](#)

Retrieve the value of the argument.

```
String arg(int index)
```

For a property passed on a query string, here we can retrieve the corresponding value.

[ESP8266WebServer.argName](#)

Retrieve the name name of the argument.

```
String argName(int index)
```

For a property passed on a query string, here we can retrieve the corresponding name.

[ESP8266WebServer.args](#)

Retrieve the number of properties passed on a query string supplied with the browser query.

```
int args()
```

[ESP8266WebServer.begin](#)

Start listening for incoming browser connections.

```
void begin()
```

[ESP8266WebServer.client](#)

`WiFiClient client()`

ESP8266WebServer.handleClient

Handle a client (browser) request.

```
void handleClient()
```

This is the function that is to be periodically called to process incoming browser requests. For example, this is the function that is placed in the body of the loop().

ESP8266WebServer.hasArg

Return true if the named property was supplied.

```
bool hasArg(const char* name)
```

The name parameter is the name of the property that may have been supplied as a property in a query string.

ESP8266WebServer.method

Retrieve the method supplied by the original browser request.

```
HTTPMethod method()
```

The HTTPMethod can be one of:

- HTTP_GET
- HTTP_POST
- HTTP_PUT
- HTTP_PATCH
- HTTP_DELETE

ESP8266WebServer.on

Register callbacks to process browser requests.

```
void on(const char *uri,
        ESP8266WebServer::THandlerFunction handler)
void ESP8266WebServer::on(const char *uri,
                           HTTPMethod method,
                           ESP8266WebServer::THandlerFunction handler)
```

Register a callback function for a URI and method. The first variant of the function will handle a matching URI for all methods while the second allows us to handle callbacks for the same URI part but different HTTP methods.

The handler function has a signature of:

```
void (handler *)()
```

ESP8266WebServer.onFileUpload

ESP8266WebServer.onNotFound

Register a callback when no specific handler otherwise exists.

```
void onNotFound(THandlerFunction fn)
```

If no callback has been explicitly registered for an incoming URL request, this callback function will be invoked as a catch-all.

ESP8266WebServer.send

Send a response to the browser.

```
void send(int code, const char *contentType, const String &content)
```

Send data to the browser. This is the primary response entry point.

The `code` parameter is the HTTP response code. The code value of 200 means OK.

The `contentType` parameter is the MIME content of the payload. The `content` parameter is the actual content to send.

ESP8266WebServer.sendContent

Send a string to the browser. This is a lower level interface and using the `send()` method is the correct way to send app data.

```
void sendContent(const String &content)
```

Send a string to the browser. The string passed in `content` is used to transmit the data.

ESP8266WebServer.sendHeader

Send an HTTP header.

```
void sendHeader(const String& name, const String& value, bool first)
```

Add an HTTP header to the output stream sent to the browser. The `name` parameter is the name of the header while `value` is the value of the header. The `first` parameter says whether or not the header will be added at the front of the list of headers or at the end.

ESP8266WebServer.setContentLength

Set the length of the content to be sent.

```
void setContentLength(size_t contentLength)
```

ESP8266WebServer.streamFile

Stream the content of a file.

```
size_t streamFile(T &file, const String &contentType)
```

ESP8266WebServer.upload

HTTPUpload& upload()

ESP8266WebServer.uri

Retrieve the current URI that was supplied by the browser.

```
String uri()
```

This is of primary value in a request callback handler where we can determine the relative URI path.

ESP8266mDNS library

Advertise ourselves via Multicast DNS. The library called "ESP8266mDNS" must be added to the project and the include called "ESP8266mDNS.h" must be included.

Examining this library, it appears to **not** use the ESP8266 SDK functions for mDNS. That appears odd.

See also:

- Multicast Domain Name Systems

MDNS.addService

```
void addService(char *service, char *proto, uint16_t port);
```

```
void addService(const char *service, const char *proto, uint16_t port)
```

```
void addService(String service, String proto, uint16_t port)
```

MDNS.begin

Begin responding to mDNS requests.

```
bool begin(const char* hostName);
```

```
bool begin(const char* hostName, IPAddress ip, uint32_t ttl=120)
```

Note that the version of the function with more than the hostname is implemented by ignoring the other parameters. The function returns true on success.

MDNS.update

```
void update();
```

I2C - Wire

The `Wire` class provides I2C support. In order to use this class, import "`Wire.h`" into your sketch. When we use this class, a global instance called "`Wire`" is made available to us. One wire is called SCL which provides the clock while the other wire is called SDA and is the data bus. On the Arduino, the library supports being either a master or a slave however in the current implementation, only being a master is supported.

To use this class, first we define which pins should be used and then start the service.

```
Wire.begin(SDApin, SCLpin);
```

To send data, we begin a transmission using `beginTransmission()`:

```
Wire.beginTransmission(deviceAddress);
```

now we can write some data ...

```
Wire.write(value);
```

and finally complete the transmission:

```
Wire.endTransmission();
```

if we wish to receive data from the slave, we can call `requestFrom()`:

```
Wire.requestFrom(deviceAdress, size, true);
```

and data can be read using the `available()` and `read()` functions.

See also:

- Working with I2C
- Arduino – [Wire Library](#)

Wire.available

Determine the number of bytes available to read.

```
int available(void)
```

Determine the number of bytes available to read.

See also:

- `Wire.read`
- `Wire.requestFrom`

Wire.begin

Initialize the wire library.

```
void begin(int SDApin, int SCLpin)
void begin()
void begin(uint8_t address)
void begin(int address)
```

Initialize the wire library. When an address is supplied, we are a slave otherwise we are a master. We can also specify the pins to be used for SDA and SCL. If we are a master and no pins are supplied, we will use the default pins.

WARNING!! – It appears that there is **NO** support for actually being a slave and only a master is supported at this time.

WARNING!! – In the current code, the `address` parameter is ignored!!

See also:

- `Wire.pins`

Wire.beginTransmission

Beging a transmission block to a slave.

```
void beginTransmission(uint8_t address)
void beginTransmission(int address)
```

Begin the notion of sending a transmission to a slave device with the supplied address. Further calls to `write()` will queue data to be transmitted which is finally executed with a call to `endTransmission()`.

Wire.endTransmission

End the bracketing of a transmission.

```
uint8_t endTransmission(void) // Defaults to sendStop = true
uint8_t endTransmission(uint8_t sendStop)
```

End the bracketing of a transmission and perform the actual transmit. The return codes are:

- 0 – Transmitted correctly
- 2 – Received NACK on transmit of address
- 3 – Received NACK on transmit of data
- 4 – line busy

Wire.flush

Discard any un-read or un-written data.

```
void TwoWire::flush(void)
```

Discard any un-read or un-written data. A call to `available()` will return 0 and a call to `endTransmission()` will transmit no data.

Wire.onReceive

A callback when we are in the role of a slave and receive a transmission from a master.

```
void onReceive( void (*function)(int numBytes) )
```

A callback when slave receives a transmission from a master.

WARNING!! – This function is not implemented.

Wire.onReceiveService

Not implemented.

```
void onReceiveService(uint8_t* inBytes, int numBytes)
```

Not implemented.

WARNING!! – This function is not implemented.

Wire.onRequest

A callback invoked when we are in the role of a slave and a master requests data from us.

```
void onRequest(void (*function)(void))
```

A callback invoked when we are in the role of a slave and a master requests data from us.

WARNING!! – This function is not implemented.

Wire.onRequestService

Not implemented.

```
void onRequestService(void)
```

Not implemented.

WARNING!! – This function is not implemented.

Wire.peek

Peek at the next byte.

```
int peek(void)
```

Peek at the next byte if one is available. A return of -1 if there is no byte available.

Wire.pins

WARNING!! - This function has been deprecated in favor of `begin(sda, scl)`.

Define the default pins for SDA and SCL.

```
void pins(int sda, int scl)
```

Define the default pins for SDA and SCL.

See also:

- [Wire.begin](#)

[Wire.read](#)

Read a single byte.

```
int read(void)
```

Read a single byte from the bus. A value of -1 is returned if there is no byte available.

See also:

- [Wire.available](#)
- [Wire.requestFrom](#)

[Wire.requestFrom](#)

Request data from a slave.

```
size_t requestFrom(uint8_t address, size_t size, bool sendStop)
uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
uint8_t requestFrom(uint8_t address, uint8_t quantity)
uint8_t requestFrom(int address, int quantity)
uint8_t requestFrom(int address, int quantity, int sendStop)
```

Request data from a slave. This method should be called when we are playing the role of a master. The `address` parameter defines the slave address for the device that should respond. If the `sendStop` is true, a stop message is transmitted releasing the I2C bus. If `sendStop` is false, a restart message is transmitted preventing another bus master from taking control.

The `quantity` parameter states how many bytes we wish to receive.

The return value is the number of bytes that were received.

See also:

- [Wire.read](#)
- [Wire.available](#)

[Wire.setClock](#)

Set the clock frequency.

```
void setClock(uint32_t frequency)
```

Set the clock frequency. Always call `setClock()` AFTER a call to `begin()`.

Wire.write

Write one or more bytes to the slave.

```
size_t write(uint8_t data)
size_t write(const uint8_t *data, size_t quantity)
```

Write one or more bytes to the slave.

Ticker library

This library sets up callback functions that are called after a period of time. To use this library you must include "Ticker.h". For example:

```
#include <Ticker.h>

void timerCB() {
    Serial1.println("Tick ...");
}

void setup()
{
    Serial1.begin(115200);
    ticker.attach(5, timerCB);
    Serial1.println("Ticker attached");
}
```

Ticker

An instance of a Ticker object. Commonly this is created as a global such as:

```
Ticker myTicker;
```

attach

Attach a callback function to the ticker.

```
void attach(float seconds,
           callback_t callback)
void attach(float seconds,
           void (*callback)(TArg),
           TArg arg)
```

Attach a callback function to the ticker such that the callback is invoked each period of seconds. Note that seconds is a float so we can specify values such as 0.1 to indicate a callback every 1/10th of a second (100 milliseconds).

The `callback_t` is defined as:

```
void (*callback_t)(void)
```

attach_ms

Attach a callback function to the ticker.

```
void attach_ms(uint32_t milliseconds,
               callback_t callback)
void attach_ms(uint32_t milliseconds,
               void (*callback)(TArg), TArg arg)
```

Attach a callback function to the ticker such that the callback is invoked each period of milliseconds. Only one attachment can be made to a timer.

detach

Detach a ticker from the timer.

```
void detach()
```

Detach a callback function from the timer. No further callbacks will occur.

once

Attach a callback function to the timer for a one-shot firing.

```
void once(float seconds,
          callback_t callback)
void once(float seconds,
          void (*callback)(TArg),
          TArg arg)
```

Attach a callback function to the timer for a one-shot firing. Note that seconds is a float so we can specify values such as 0.1 to indicate a callback every 1/10th of a second (100 milliseconds).

once_ms

Attach a callback function to the timer for a one-shot firing.

```
void once_ms(uint32_t milliseconds,
             callback_t callback)
void once_ms(uint32_t milliseconds,
             void (*callback)(TArg),
             TArg arg)
```

Attach a callback function to the timer for a one-shot firing.

EEPROM library

This library allows us to store and retrieve data from storage that persists across a device restart. A singleton object called EEPROM is pre-supplied for use.

EEPROM.begin

Begin the process of writing or reading from EEPROM. The size is the amount of storage we wish to work with.

```
void begin(size_t size)
```

EEPROM.commit

The changes to the data are committed to EEPROM. A return of `true` indicates success while a return of `false` indicates a failure.

```
bool commit()
```

EEPROM.end

Commits the changes to the data and then releases any local storage. No further reads or writes should be attempted until after the next `begin()` call.

```
void end()
```

EEPROM.get

Read a data structure from storage.

```
T &get(int address, T &t)
```

EEPROM.getDataPtr

Retrieve a pointer to the storage we are going to read or write.

```
uint8_t *getDataPtr()
```

EEPROM.put

Put a data structure to storage.

```
const T &put(int address, const T &t)
```

EEPROM.read

Read a byte from storage.

```
uint8_t read(int address)
```

EEPROM.write

Write a byte to storage.

```
void write(int address, uint8_t value)
```

SPIFFS

FS is the File System library which provides the ability to read and write files from within the Arduino ESP environment. But wait ... read and write files to where? There are no "drives" on an ESP8266. The data for the files is read and written to an area of flash memory and since flash is relatively small in size (4MBytes or so max) then that is an upper bound of maximum size of the cumulative files ... however, this is still more than enough for many usage patterns such as saving state, logs or configuration information.

SPIFFS.begin

Begin working with the SPIFFS file system.

```
bool begin()
```

Returns true of success and false otherwise.

SPIFFS.open

Open the named file.

```
File open(const char *path, const char *mode)
File open(const String &path, const char *mode)
```

The mode defines how we wish to access the file. The options are:

- r – Read the file. The file must exist.
- w – Write to the file. Truncate the file if it exists.
- a – Append to the file.
- r+ – Read and write the file.
- w+ – Read and write the file.
- a+ – Read and write the file.

See also:

- File.close

SPIFFS.openDir

Open a directory.

```
Dir openDir(const char *path)
Dir openDir(const String &path)
```

SPIFFS.remove

Remove/delete a file from the file system.

```
bool remove(const char *path)
bool remove(const String &path)
```

SPIFFS.rename

Rename a file.

```
bool rename(const char *pathFrom, const char *pathTo)
bool rename(const String &pathFrom, const String &pathTo)
```

File.available

Return the number of bytes that are available within the file from the current file position to its maximum size.

```
int available()
```

File.close

Close a previously opened file.

```
void close()
```

No further reading nor writing should be attempted to be performed.

File.flush

Flush the file.

```
void flush()
```

File.name

Retrieve the name of the file.

```
const char *name()
```

File.peek

Peek at the next byte of data in the file without consuming it.

```
int peek()
```

File.position

Retrieve the current file pointer position.

```
size_t position()
```

File.read

Read data from the file.

```
int read()  
size_t read(uint8_t *buf, size_t size)
```

Read either a single byte of data or a buffer of data from the file.

File.seek

Change the current file pointer position.

```
bool seek(uint32_t pos, SeekMode mode)
```

The mode can be one of:

- SeekSet – Change the file pointer position to the absolute value.
- SeekCur – Change the file pointer position to be relative to the current position.
- SeekEnd – Change the file pointer position to be relative to the end of the file.

File.size

Retrieve the maximum size of the file.

```
size_t size()
```

File.write

Write data to the file.

```
size_t write(uint8_t c)  
size_t write(uint8_t *buf, size_t size)
```

Write either a single byte or a buffer of bytes into the file at the current file pointer position.

Dir.fileName

Retrieve the name of the file.

```
String fileName()
```

Dir.next

```
bool next()
```

Dir.open

```
File open(const char *mode)  
File open(String &path, const char *mode)
```

Dir.openDir

```
Dir openDir(const char *path)  
Dir openDir(String &path)
```

Dir.remove

Dir.rename

ESP library

A class has been provided called ESP that provides ESP8266 environment specific functions. You must realize that using these functions will result in your applications not being portable to the Arduino (if that is a desire).

ESP.deepSleep

void deepSleep(uint32_t time_us, WakeMode mode)

ESP.eraseConfig

bool eraseConfig()

ESP.getBootMode

uint8_t getBootMode()

ESP.getBootVersion

uint8_t getBootVersion()

ESP.getChipId

uint32_t getChipId()

ESP.getCpuFreqMHz

uint8_t getCpuFreqMHz()

ESP.getCycleCount

uint32_t getCycleCount()

ESP.getFlashChipId

uint32_t getFlashChipId()

ESP.getFlashChipMode

FlashMode_t getFlashChipMode()

ESP.getFlashChipRealSize

uint32_t getFlashChipRealSize()

ESP.getFlashChipSize

uint32_t getFlashChipSize()

ESP.getFlashChipSizeByChipId

uint32_t getFlashChipSizeByChipId()

ESP.getFlashChipSpeed

uint32_t getFlashChipSpeed()

ESP.getFreeHeap

uint32_t getFreeHeap()

ESP.getFreeSketchSpace

uint32_t getFreeSketchSpace()

ESP.getResetInfo

String getResetInfo()

ESP.getResetInfoPtr

struct rst_info * getResetInfoPtr()

ESP.getSdkVersion

Retrieve the string representation of the SDK being used.

const char *getSdkVersion()

ESP.getSketchSize

uint32_t getSketchSize()

ESP.getVcc

uint16_t getVcc()

ESP.reset

void EspClass::reset()

ESP.restart

void EspClass::restart()

ESP.updateSketch

bool updateSketch(Stream& in,
uint32_t size,
bool restartOnFail,
bool restartOnSuccess)

ESP.wdtDisable

void wdtdisable()

ESP.wdtEnable

void wdteable(uint32_t timeout_ms)

ESP.wdtFeed

void wdtdfeed()

String library

Although it is believed that this library may be identical to the Arduino String library, I believe it is so essential to understand that I am going to list the methods again.

String

Constructor

```
String(const char *cstr = "");  
String(const String &str)  
String(char c)  
String(unsigned char, unsigned char base = 10)  
String(int, unsigned char base = 10)  
String(long, unsigned char base = 10)  
String(unsigned long, unsigned char base = 10)  
String(float, unsigned char decimalPlaces = 2)  
String(double, unsigned char decimalPlaces = 2)
```

Create an instance of the String class seeded with various data type initializers.

String.c_str

Retrieve a C string representation.

```
const char *c_str()
```

String.reserve

```
unsigned char reserve(unsigned int size)
```

String.length

Return the length of the string.

```
unsigned int length()
```

Return the length of the string.

String.concat

```
unsigned char concat(const String &str)  
unsigned char concat(const char *cstr)  
unsigned char concat(char c)  
unsigned char concat(unsigned char c)  
unsigned char concat(int num)  
unsigned char concat(unsigned int num)  
unsigned char concat(long num)  
unsigned char concat(unsigned long num)  
unsigned char concat(float num)  
unsigned char concat(double num)
```

String.equalsIgnoreCase

```
unsigned char equalsIgnoreCase(const String &s) const;
```

String.startsWith

Determine whether or not this string starts with another string.

```
unsigned char startsWith(const String &prefix)
unsigned char startsWith(const String &prefix, unsigned int offset)

String.endsWith
unsigned char endsWith(const String &suffix)

String.charAt
char charAt(unsigned int index)

String.setCharAt
void setCharAt(unsigned int index, char c)

String.getBytes
void getBytes(unsigned char *buf, unsigned int bufsize, unsigned int index = 0)

String.toCharArray
void toCharArray(char *buf, unsigned int bufsize, unsigned int index = 0)
```

String.indexOf

Find the position of a string or character within the current string.

```
int indexOf(char ch)
int indexOf(char ch, unsigned int fromIndex)
int indexOf(const String &str)
int indexOf(const String &str, unsigned int fromIndex)
```

Find the position of a string or character within the current string. If the match is not found, -1 is returned otherwise the position of the start of the match is returned.

String.lastIndexOf

```
int lastIndexOf(char ch)
int lastIndexOf(char ch, unsigned int fromIndex)
int lastIndexOf(const String &str)
int lastIndexOf(const String &str, unsigned int fromIndex)
```

String.substring

Retrieve a substring from within the current string.

```
String substring(unsigned int beginIndex)
String substring(unsigned int beginIndex, unsigned int endIndex)
```

Retrieve a substring from within the current string.

String.replace

```
void replace(char find, char replace)
void replace(const String& find, const String& replace)
```

String.remove

```
void remove(unsigned int index)
void remove(unsigned int index, unsigned int count)
```

String.toLowerCase

```
void toLowerCase(void)
```

String.toUpperCase

```
void toUpperCase(void)
```

String.trim
void trim(void)

String.toInt
longToInt(void)

String.toFloat
float toFloat(void)

Programming with JavaScript

The JavaScript is a high level interpreted language. Some of its core constructs are loose typing, object oriented, support of lambda functions, support of closures and, most importantly, has become **the** language of the web. If one is writing a browser hosted application, then it is a certainty that it will be written in JavaScript. But what of non-browser environments? For a while now JavaScript has been eating into server side code through projects such as Node.js. As a language for running server code, it has a significant set of features to realize this capability. Specifically, it supports an event driven architecture paradigm. In JavaScript, we can register functions to be called back upon events being detected. These callbacks can be defined as simple in-line functions on what to do. In these made up examples, we illustrate this:

```
httpServer.on("/path1", function() {  
    // Do something for /path1  
    httpServer.send(response);  
});  
  
or  
  
socket.accept(port, function(newSocket) {  
    newSocket.on("receive", function(data) {  
        print("We received new data: " + data);  
        newSocket.send("We got the data", function() {  
            newSocket.close();  
        });  
    });  
});
```

And if we can implement a good JavaScript model, it maps excellently to the ESP8266 model of the world which is itself event driven via callbacks. This mapping won't be easy ... but plans are afoot.

Espruino is an open source project to provide a JavaScript run-time for embedded devices. It has been implemented for the ARM Cortex M3/M4 processors and others.

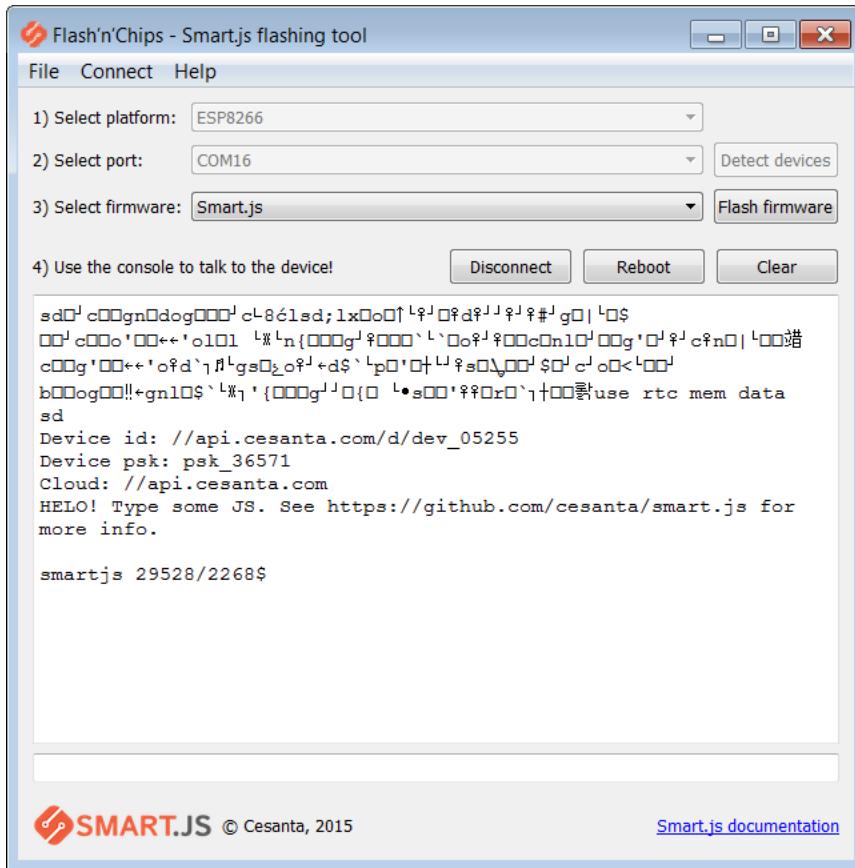
The question now is ... can it be used for the ESP8266? An active project is attempting to do just that.

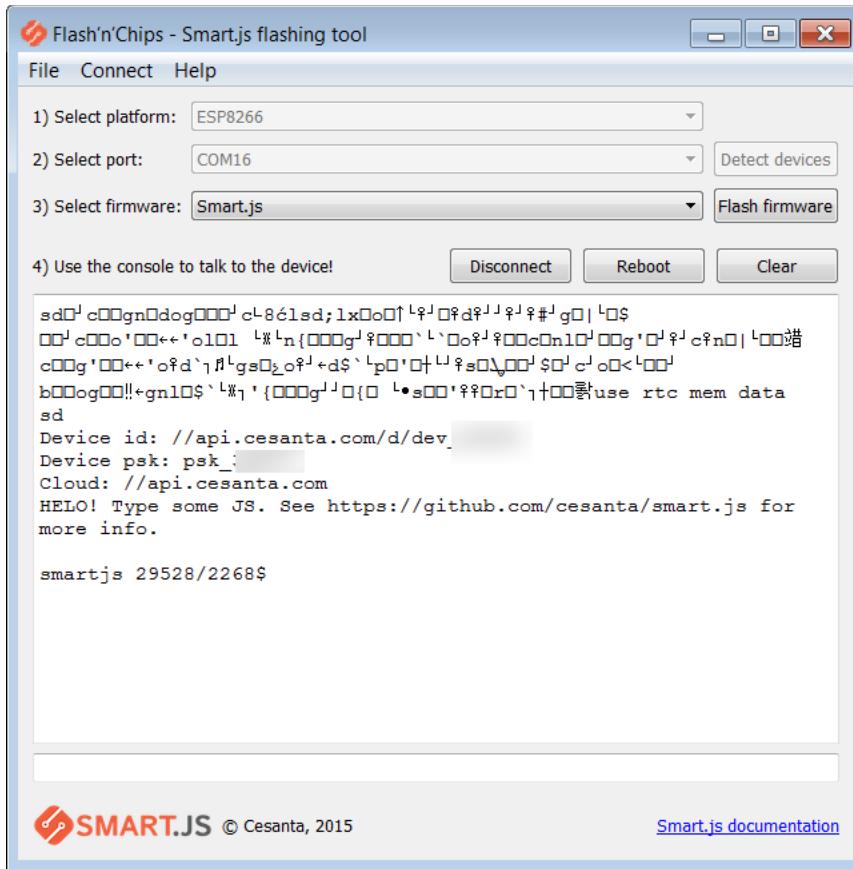
See also:

- [Espruino](#)

Smart.js

Smart.js is an implementation of JavaScript for a variety of embedded platforms. It provides native support for most of the common I/O functions including GPIO, SPI, I2C, PWM. It has networking support through WiFi and HTTP server. It supports a file system.





To configure WiFi, run:

```
wifi.setup(ssid, password)
```

To list files ... run `File.list('.')`

To determine which version, run `version`

See also:

- Github: [cesanta/smarty.js](https://github.com/cesanta/smarty.js)
- [Smart.js home page](#)
- [Smart.js Developer Centre](#)

Smart.js GPIO

Smart.js has some sophisticated GPIO capabilities. Before reading or writing from a pin, we must first declare the mode of access. We can do this with `GPIO.setmode(pin, mode, pullup)`. The pin parameter is the pin we are defining, mode is the I/O mode which is encoded as:

- 0 – Input and output

- 1 – Input
- 2 – Output
- 3 – Interrupts

The pullup parameter defines any pullup characteristics:

- 0 – floating
- 1 – pull-up
- 2 – pull-down

Once the mode has been set, we can read from a pin via `GPIO.read(pin)` or write to a pin with `GPIO.write(pin, value)`.

If we wish to use interrupts, we can register a function to be called when a value of a pin changes. This is set with `GPIO.setisr(pin, type, function)`. The type parameter is an encoding of the type of signal change that will trigger the interrupt.

- 0 – Interrupts disabled
- 1 – Positive edge
- 2 – Negative edge
- 3 – Any edge
- 4 – Low
- 5 – High
- 6 – Button

The function that is called has the signature `function(pin, level)`.

Related to GPIO is the notion of PWM. We can define a PWM using `PWM.set(pin, period, duty)`. Both the period and duty are supplied in microseconds.

Setting up an HTTP server

Smart.js provides an implementation of an Http server. A pre-supplied class called "Http" is available. First we create an instance of an Http server using the `createServer()` method.

```
var httpServ = Http.createServer(function(req, res) {});
```

The creation of a server does not immediately start it listening, instead the `listen()` method on the Http server should be called.

```
httpServ.listen(80);
```

When a browser connects to the server the function will be called with parameters of both a request and response object.

The request object contains the following properties:

- `url` – The relative URL part of the request

The response object contains the following methods:

- `writeHead(status, headers)`
- `write(data)`
- `end()`

Debugging

When running Smart.js JavaScript programs, we may wish to do some debugging. We can perform logging using `print()` or `Debug.print()`.

From a memory and other resources standpoint, the command `GC.stat()` returns an object that contains details about the available resources. This includes:

- `owned_max`
- `owned`
- `astsize`
- `funcncell`
- `funcnfree`
- `propncell`
- `propnfree`
- `objncell`
- `objfree`
- `struse`
- `strres`
- `jsfree`
- `jssize`
- `sysfree`
-

Espruino

Espruino is an open source implementation of JavaScript that runs on a variety of platforms including the ESP8266. The web page for ESP8266 support can be found here:

<http://www.espruino.com/EspruinoESP8266>

A build of Espruino for the ESP8266 is available as part of the master set of builds available here:

<http://www.espruino.com/Download>

Here is a recipe for downloading and installing a copy of Espruino onto an ESP8266 using Linux as a base.

```
$ mkdir espruino
$ cd espruino
$ wget http://www.espruino.com/files/espruino_1v87.zip
$ unzip espruino_1v87.zip
$ cd espruino_1v87_esp8266
$ esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash \
  --flash_freq 80m --flash_mode qio --flash_size 32m \
  0x0000 "boot_v1.4(b1).bin" \
  0x1000 espruino_esp8266_user1.bin \
  0x37E000 blank.bin
```

When I ran this, the result was the following:

```
esptool.py v1.2-dev
Connecting...
Running Cesanta flasher stub...
Flash params set to 0x004f
Writing 4096 @ 0x0... 4096 (100 %)
Wrote 4096 bytes at 0x0 in 0.4 seconds (88.7 kbit/s)...
Writing 466944 @ 0x1000... 466944 (100 %)
Wrote 466944 bytes at 0x1000 in 41.0 seconds (91.0 kbit/s)...
Writing 4096 @ 0x37e000... 4096 (100 %)
Wrote 4096 bytes at 0x37e000 in 0.4 seconds (88.6 kbit/s)...
Leaving...
```

If `esptool.py` is not installed, it can be added with:

```
$ sudo apt-get install python-pip
$ pip install esptool
```

Editing and deploying code

Espruino has an elegant code editor and deployment tool that is based on Chrome. It allows us to write our JavaScript and load it into the Espruino run-time. We can also use the `dump()` method to ask Espruino what it has installed as a program. This can be copied back to an editor.

Not only is the Espruino web editor available for execution in Chrome, it is available as an application that can be locally installed. Assuming you have a local copy of Node.js installed, we can run:

```
$ sudo npm install -g espruino-web-ide  
$ espruino-web-ide
```

This will launch a local copy of the development tools.

Working with variables

An object called "global" can be used to list the global data in scope.

Booting Espruino

When Espruino boots, it will continue from where it was in the code when "save()" was executed. The effect of running "save()" is to cause the state of the program to be written to flash and then loaded on next boot. If we need to perform initialization upon boot, we use the `E.on("init", function() { ... })` event handler. For example:

```
E.on("init", function() {  
    // Code here ...  
});
```

A method called "reset()" will reset the state of Espruino without performing a hardware reset.

WiFi access

A module called "Wifi" contains the majority of WiFi access functions. To use this we bring in the module with:

```
var wifi = require("Wifi");
```

To start listening as an access point, we can use the `startAP()` method. For example:

```
var wifi = require("Wifi");  
wifi.startAP("ESP8266", {  
    "authMode": "open"  
}, function(err) {  
    console.log("AP now started: " + err);  
});
```

We can interrogate the status of the Access Point using `getAPDetails()`. For example:

```
wifi.getAPDetails(function(data) {  
    // process data  
});
```

The structure returned might look like:

```
{  
  "status": "enabled",  
  "authMode": "open",  
  "hidden": false, "maxConn": 4,  
  "ssid": "ESP8266",  
  "password": "",  
  "savedSsid": null,  
  "stations": [  
    {  
      "ip": "192.168.4.2",  
      "mac": "00:36:76:21:97:a3"  
    }  
  ]  
}
```

Note that "stations" is an array of stations that are connected to the access point.

If we want to know the IP address of the access point we can call `wifi.getAPIP()`. For example:

```
wifi.getAPIP(function(data) {  
  console.log(data);  
});
```

An example of what might be returned would be:

```
{  
  "ip": "192.168.4.1",  
  "netmask": "255.255.255.0",  
  "gw": "192.168.4.1",  
  "mac": "1a:fe:34:f5:2e:ec"  
}
```

The alternative to being an access point is being a station. If we choose that option then we get to connect to an existing WiFi network as a station participant. We do this by calling `wifi.connect()`. This takes the identity of the network to connect to, options and a callback function. For example:

```
wifi.connect("mySsid", {  
  password: "myPassword",  
}, function(err) {  
  console.log("Connect completed ... err is: " + err);  
});
```

Note that the options and the callback functions are optional.

Writing network socket applications using Espruino

The network library provided by the Espruino environment can be accessed using:

```
var net = require("net")
```

This returns a network object that exposes operations necessary to interact with the network. The methods exposed by this object include:

- `createServer` – Create a server instance
- `connect` – Connect to a partner

```
net.connect("http://1.2.3.4:80", function(connection) {  
    // Connection is a connection object to a partner.  
});
```

The connection object (which is called a "Socket" by Espruino) is always a reference to a specific connection between the ESP8266 and a partner over the network. The connection object has the following methods:

- `available` – The number of bytes available to read. This will always return 0 if a data listener has been registered.
- `read` – Read some number of bytes.
- `write` – Write some number of bytes.
- `end` – Terminate the connection.
- `on("data")` – Register a callback to be invoked when data becomes available.

Writing network HTTP applications using Espruino

Writing a REST client using Espruino

A REST client is one which sends REST requests which are basically HTTP requests. At a high level, the following example illustrates how to achieve this:

```
var http = require("http");  
var getRequest = http.get({  
    host: "192.168.1.9",  
    port: 1817,  
    path: "/sendmessage?message=1"  
}, function(response) {  
    // Handle response here  
});
```

There are two key parameters to the `get()` method. The first is an options JavaScript object that configures the details of the REST request. This object includes:

- `host` – The IP address of the target of the REST request
- `port` – The port number to which the request will be sent
- `path` – The relative URL part
- `method` – Unknown

- `headers` – JavaScript object representing the headers to send.

The second parameter is a callback function that will be invoked when the response to the REST request is received. The callback function has the signature:

```
callback(response)
```

where `response` is an instance of the Espruino `HTTPCRS` object. This object has the following methods:

- `available` – Return the number of available bytes for reading.
- `event close` – Called when the connection is closed.
- `event data` – Called when there is data available.
- `pipe` – Do something with a pipe.
- `read` – Read available data.

In addition, the response object has a property called `headers` which are the HTTP response headers returned from the server. These will include the usual HTTP properties which may include:

- Content-Type
- Date
- Connection
- Content-Length

The return from the `http.get()` function call is an object of type `HTTPCRQ`.

Writing a Web Server using Espruino

We can also set our environment up to be a Web Server that can process incoming HTTP client requests. To do this we need to access the `http` class using `require("http")`. This class has a method on it to create an instance of a Web Server. The method has the signature:

```
createServer(handlerFunction)
```

The handler callback function is invoked when each new client request arrives. In the majority of cases the client will be a browser but it could just as easily be another application making REST request. The callback handler takes two parameters as input. The first is an object representing the request (`httpSRq`) and the second an object representing the response (`httpSRs`)

The `createServer` method returns an instance of an HTTP server object (`httpSrv`). We can then start that HTTP server listening with a call to `listen()` which takes the port number the server should listen upon.

For example:

```
var http = require("http");
var httpServer = http.createServer(function(request, response) { ... });
httpServer.listen(80);
```

Once the server starts listening, incoming HTTP requests will cause the handler function to be invoked passing in an object representing the request and an object that can be used to formulate the response.

The request object passed to the callback function has the following properties:

- `url` – The local path of the incoming request.
- `method` – The HTTP method of the incoming request.
- `headers` – The headers contained in the data.

Since the `url` parameter can be made up of query options, it might be nice to be able to extract those. Here is a fragment that will do just that:

```
var partsOfUrl = request.url.split("?");
if (partsOfUrl.length > 1) {
  var options = partsOfUrl[1].split('&');
  var optionsObj = {};
  for (var i=0; i<options.length; i++) {
    var splitEquals = options[i].split('=');
    optionsObj[splitEquals[0]] = splitEquals[1];
  }
  print("Final obj: " + JSON.stringify(optionsObj));
}
```

If the incoming request is an HTTP POST or PUT command, we can have optional payload data sent as part of the request from the client. We access that data by registering a data event callback on the request object:

```
request.on("data", function(receivedData) {
  // Process data
});
```

Let us pause here for a moment to consider what is happening. Data is arriving from the caller as a stream TCP socket. This means that the data will arrive in order but not necessarily all once. If the client wishes to send in 100 bytes of payload data in the request, we may receive all 100 bytes as a single chunk or we might just as easily receive 100 callbacks of 1 byte each. As such, it is our responsibility to ensure that we have received all the data we need before processing. If we follow this notion, we will also realize that the `request.available()` and `request.read()` methods should be understood properly. Calling `request.available()` tells us what data has been received

so far ... and does not indicate how much actual data may be eventually received. Similarly, the `request.read()` method returns data that has been received but does not block waiting for new/additional data to arrive.

An event is made available to determine when the client has sent all the required data. This is through the `request.on("close", function() ...)` mechanism. We can register a close callback handler to be informed when the client has closed the connection. At this point, we can read all the data through `available()` and `read()` because we know there will be no further data arriving.

To return a response, we can invoke the `writeHead(statusCode, headers)` method to set a status code and headers for the return. To write content in the response we can use:

```
response.write(data);
```

To test, we need to send an HTTP request. Assuming we are not testing with a browser, we can use:

```
wget http://<IP address> --quiet --output-document=
```

The `--quiet` flag switches off app chatter while `--output-document` writes the received data to `stdout`.

To send a request containing data, we can use:

```
wget http://<IP address> --post-data "<some data>" --quiet -output-document=
```

Working with GPIO

Espruino provides GPIO support through a number of global methods. These include `"pinMode()"` to set the mode of a pin, `"getPinMode()"` to get the mode of a pin, `"digitalWrite()"` to set the logic level, `"digitalRead()"` to get the logic level, `"digitalPulse()"` to pulse a logic level.

Working with I2C and JavaScript

The software I2C mechanism is available from the `I2C` class which is a built-in. To use, we can leverage an instance of an `I2C` object. Espruino pre-creates one called `I2C1`. From there, we can use the `setup()` method upon it. The `setup()` method takes an object as input which has properties of:

- `scl` – The pin to be used for a clock (default is 14)
- `sda` – The pin to be used for SDA (default is 2)
- `bitrate` – The bitrate of communication (default is 50000)

To write to the I2C device, we can invoke the `writeTo()` method. This has the signature:

```
writeTo(address, data, ...)
```

Where `address` is the address of the I2C slave device and `data` is the data to transmit.

To read data, we can use the `readFrom()` method. This has the signature:

```
readFrom(address, quantity)
```

Where `address` is the address of the I2C slave device and `quantity` is the number of bytes to read. The result is an array of bytes.

Convenience constants are available called "`HIGH`" and "`LOW`".

See also:

- Working with I2C

Debugging JavaScript

There are a number of ways we can debug the JavaScript code.

The first is through the `dump()` statement. This will log the current interpreter state.

The `trace()` statement can be used to dump the variables including their types. It can take a variable name.

The `global` variable is a scope qualifier. Using `global["xFF"]` will access the Espruino "hidden" variables.

Editing JavaScript

Personally, I prefer to use the Eclipse programming environment for all my work. We can install the JavaScript development tools to provide us a nice JavaScript editor. This can be installed through the normal Eclipse plug-in installation mechanisms. Simply search for JavaScript in the installable components. Once installed and editing the script, you get all kinds of JavaScript language support including entry assist and a program outline:

```

1 var WiFiEvent = {
2   STAMODE_CONNECTED: 0,
3   STAMODE_DISCONNECTED: 1,
4   STAMODE_AUTHMODE_CHANGE: 2,
5   STAMODE_GOT_IP: 3,
6   SOFTAPMODE_STACONNECTED: 4,
7   SOFTAPMODE_STADISCONNECTED: 5
8 };
9
10 var ssid = "ESPTEST";
11 var password = XXXXXXXXXX;
12
13 ESP8266WiFi.onWiFiEvent(function(event, details) {
14   print("WiFi Event Callback invoked!");
15   print("event=" + event + ", details=" + JSON.stringify(details));
16   print("ipInfo: " + JSON.stringify(ESP8266WiFi.getIPInfo()));
17   if (event == WiFiEvent.STAMODE_CONNECTED) {
18     print("We are connected and our IP address is " + ESP8266WiFi.getIP()
19   }
20 });
21
22 ESP8266WiFi.connect(ssid, password);
23

```

The Outline panel on the right shows the following structure:

- WiFiEvent : {}
 - STAMODE_CONNECTED : Number
 - STAMODE_DISCONNECTED : Number
 - STAMODE_AUTHMODE_CHANGE : Number
 - STAMODE_GOT_IP : Number
 - SOFTAPMODE_STACONNECTED : Number
 - SOFTAPMODE_STADISCONNECTED : Number
 - ssid : String
 - password : String

Espruino ESP8266 Libraries

There is an ESP8266 specific library than can be accessed from the require statement that looks like:

```
var esp8266 = require("ESP8266");
```

There are a number of methods exposed on the returned object including:

- **crc32** – Create a 32 bit CRC.
- **deepSleep** – Make the ESP8266 enter "deep sleep" mode. Effectively a timed reboot.
- **dumpSocketInfo** – Debug socket information by writing it to the log.
- **getFreeFlash** – Return the amount of free flash storage. Deprecated.
- **getResetInfo** – Retrieve the reason for the last reset/reboot.
- **getState** – Retrieve the state of the device
 - **sdkVersion** – Version of SDK
 - **cpuFrequency** – MHZ of CPU speed
 - **freeHeap** – Amount of free heap storage
 - **maxCon** – Maximum number of connections
 - **flashMap** – How flash is mapped
 - **flashKB** – Configured flash size
 - **flashChip** – Manufacturer of flash chip

here is an example of output:

```
{  
  "sdkVersion": "1.5.0",  
  "cpuFrequency": 160, "freeHeap": 10096, "maxCon": 10,  
  "flashMap": "4MB:512/512",  
  "flashKB": 4096,  
  "flashChip": "0xe0 0x4016"  
}
```

- `logDebug` – Enable or disable debug logging.
- `neopixelWrite` – Write to a string of NeoPixels.
- `ping` – Ping the given IP address.
- `printLog` – Print the debug log to the console.
- `readLog`
- `reboot` – Reboot the device.
- `setCPUFreq` – Set the CPU frequency. Deprecated.
- `setLog` – Set the log mode
 - 0 – off
 - 1 – in memory
 - 2 – in memory and UART0
 - 3 – in memory and UART1

Core JavaScript capabilities

The JavaScript language provided by Espruino is covered in detail by the Espruino documentation. However, here are some of the core nuggets that I find extremely useful.

See also:

- Espruino software reference

Running code at intervals

We can define a timer that will fire either once (`setTimeout()`) or periodically (`setInterval()`) and call a function.

The syntax for this is:

```
setTimeout(function, delay, [args, ...])  
setInterval(function, period, [args, ...])
```

Where delay and period are times measured in milliseconds. Optional arguments can also be supplied which are passed to the function. Both these functions return an id value that can be used to cancel the request before it happens. The function to do this is called `clearInterval()`.

```
clearInterval(id)
```

We can also change the interval on a periodic callback with the `changeInterval()` function.

```
changeInterval(id, newPeriod)
```

Working with GPIO

We can define an object of type `Pin` to represent a GPIO pin and then either set its value or read its value. For example, here is a simple blinky:

```
var ledOn = false;
var ledPin = new Pin(4);
setInterval(function() {
    digitalWrite(ledPin, ledOn);
    ledOn = !ledOn;
}, 1000);
```

SPI

SPI is a wire protocol used to drive SPI compliant interface components. Espruino has a module called `SPI` that provides us access to those capabilities. First we create an SPI port using:

```
var mySPI = new SPI();
```

Next we can configure this port using the `setup()` function. The parameter to `setup` is an object which contains:

- `sck` – The Pin to use for the clock.
- `miso` – The pin to use for master in/slave out.
- `mosi` – The pin to use for master out/slave in.
- `baud` (optional) – Defaults to 100000.
- `mode` (optional) – Defaults to 0.
- `order` (optional) – Defaults to "msb".

Finally, we can call the `write()` function to write data. Alternatively we can call `send()`.

Here is an example. The MAX7219 is a powerful little IC that is able to drive an 8x8 matrix of LEDs. Being an SPI device, it uses three SPI signals:

- CS – Low to select the MAX7219 for SPI communication.
- MOSI – The data line over which serial data will flow.
- CLK – The clock line controlling reception of new bits of data.

If we look at an ESP8266, we might choose to map these to the following ESP8266 pins:

Function	Pin	NodeMCU	Color
CS	GPIO 12	D6	Green
MOSI	GPIO 13	D7	
CLK	GPIO 14	D5	Blue

Key differences from JavaScript

Although Espruino is an excellent implementation, it does have some distinctions from a ECMAScript standards. Some things are subtle and unlikely to be stumbled across while others are bigger. Here are some examples:

- Calling functions before their declaration are not supported.
- Declaring a variable as const doesn't make it so.

Building Espruino

To build Espruino from the source tree:

```
$ git clone https://github.com/espruino/Espruino.git
$ cd Espruino
$ export ESP8266_BOARD=1
$ export FLASH_4MB=1
$ export ESP8266_SDK_ROOT=/esp8266/sdk/ESP8266_NONOS_SDK
$ export PATH=$PATH:/pot/xtensa-lx106-elf/bin
$ export ESPHOSTNAME=espruino:88
```

Create the SDK dir:

```
$ wget
http://espressif.com/sites/default/files/sdks/esp8266\_nonos\_sdk\_v2.0.0\_16\_08\_10.zip
$ wget
http://espressif.com/sites/default/files/sdks/esp8266\_nonos\_sdk\_v2.0.0\_patch\_16\_08\_09.zip
$ mkdir src
$ unzip esp8266_nonos_sdk_v2.0.0_16_08_10.zip -d sdk
$ unzip esp8266_nonos_sdk_v2.0.0_patch_16_08_09.zip -d sdk/ESP8266_NONOS_SDK/lib
```

Programming with Lua

Lua is a powerful scripting language that is available on ESP8266 environments. The most popular implementation of Lua for the ESP8266 is known as the NodeMCU Lua firmware and is available at its github repository. Some folks interchange the phrase NodeMCU for the Lua firmware itself so take care to make sure that you understand the context involved.

Builds of the firmware can be downloaded directly as can the source.

Once you have a copy of the firmware you can flash it using your favorite flash tools.

We won't be describing the Lua language itself in this book. There are excellent books already written on Lua and also references and tutorials can be found on the Internet. Rather, let us look at the specifics of running Lua on an ESP8266.

Assuming you have flashed an ESP8266 with Lua, you will need to connect a serial terminal to it in order to interact with it. The serial baud rate should be 9600.

See also:

- Github: [nodemcu/nodemcu-firmware](#)
- [NodeMCU firmware Wiki](#)
- [Lua 5.1 Reference Manual](#)
- [lua.org](#)
- [NodeMCU ua Forum on ESP8266.com](#)
- [nodemcu-unofficial-faq](#)

ESPlorer IDE

The ESPlorer IDE is a development environment for building Lua applications for the ESP8266.

See also:

- [Esplorer home page](#)
- eBook: [Getting Started with the ESPlorer IDE](#)
- GitHub: [4refr0nt/ESPlorer](#)

GPIO with Lua

Lua has the concept of 13 logical pins identified by 0-12. These pins map to the GPIO pins of an ESP8266 as follows:

Lua pin number	ESP8266 Pin	NodeMCU devKit	Lua pin number	ESP8266 Pin	NodeMCU devKit
0	GPIO16	D0	7	GPIO13	D7
1	GPIO5	D1	8	GPIO15	D8
2	GPIO4	D2	9	GPIO3	D9
3	GPIO0	D3	10	GPIO1	D10
4	GPIO2	D4	11	GPIO9	SD2

5	GPIO14	D5	12	GPIO10	SD3
6	GPIO12	D6			

The GPIO pins known as GPIO6, GPIO7 and GPIO8 on the ESP8266 are not exposed.

Before reading or writing from a GPIO pin, we need to inform Lua of the mode of that pin. Our choices are input, output or interrupt. For input, we can also declare whether the input is pull-up or floating.

The syntax of the statement is:

```
gpio.mode(pin, mode, pullup)
```

Once the mode has been set, if it is input, we can call `gpio.read()` to read the value from the pin and `gpio.write()` to write a value to the pin.

If we wish to be triggered by an interrupt on the pin (input) we can use `gpio.trig()`.

WiFi with Lua

Networking with Lua

Programming with Basic

See also:

- [ESP8266 Basic](#)

Integration with Web Apps

REST Services

The notion of distributed computing dates back many decades. The idea that one computer could perform a service on behalf of another is a classic concept. The thinking is that work could be distributed across systems, data could be centralized or dedicated systems could perform specialized roles. Over the years, many forms of distributed computing have been tried. These include socket servers, remote procedure calls (RPC), Systems Network Architecture (SNA), Distributed Computing Environment (DCE), Web Services and others.

Today (2015), the current incumbent of distributed computing protocols and technology is REST. REST is a simple protocol that leverages the existing Hyper Text Transport Protocol (HTTP) used as the transport between browsers and web servers. This protocol was built to allow a browser to request data from a remote file system hosted by a web server. It provides HTTP "commands" which include GET, POST, PUT and others. The notion behind REST is more of an accident than a design. REST re-purposes HTTP as a communication conduit from a client to a server where a client makes a REST request and the server offers up a REST service. From the network

perspective, it "looks" just like a browser/Web Server interaction but both ends choose to agree on the formation and interpretation of the communication.

When we add an ESP8266 into the mix, our desire is two-fold. We want the ESP8266 to be able to be a client to external REST service providers and we want the ESP8266 to be the target of clients making REST requests. From the partner perspective, it should be unaware that it is interacting with the ESP8266 as compared to any other computing device.

REST protocol

The REST protocol is built on top of HTTP.

See also:

- [RFC7230 – HTTP/1.1 – Message Syntax and Routing](#)
- [HTTP: The Protocol Every Web Developer Must Know – Part 1](#)

ESP8266 as a REST client

For the ESP8266 to be a REST client, it must build and transmit HTTP requests to the service provider. This will include building HTTP headers, transmitting the data in a form expected by the provider (eg. JSON, XML or other textual representation) and handling the response from the provider which may include interpreting the received payload.

To transmit a REST request is composed of two parts. First it opens a TCP connection to the partner and then transmits the HTTP compliant data down that connection. The first part is easy, the second part is more of a challenge. We could read and understand the HTTP spec and build the request part by part but this would have to be done for each project that wishes to use REST client technology. What would be better is if we had a library that "knows" how to make well formed REST requests and we simply leveraged its existing functions.

Making a REST request using Mongoose

Using the Mongoose APIs, we can quite easily send a REST request and work with the response. The high level story is to initialize Mongoose with `mg_mgr_init()`, request a connection to the REST service provider with `mg_connect()`, associate the connection as being HTTP oriented and then start processing events. The first event to return will be an `MG_EV_CONNECT` event indicating that we are now network connected. From there we can use `mg_printf()` to send the REST request. When the REST partner responds, we will get an `MG_EV_HTTP_REPLY` event and we have completed our request/response pairing.

ESP8266 as a REST service provider

For an ESP8266 to be a REST service provider, basically means that it has to play the role of a Web Server and respond to Web Server requests. However, unlike a simple Web Server which simply retrieves and sends file content as a function of the path on the URL, it is likely that the REST service provider will perform some computation when an HTTP client request arrives. For example, if we attached a temperature sensor to the GPIOs of the ESP8266, when a REST request arrives, the ESP8266 could read the current temperature value and send the encoded result back as the response to the request.

Being a Web Server basically means listening on a TCP port and when connections arrive, interpreting the data received as HTTP protocol. This would be a lot of work on a project by project basis but thankfully there are a number of pre-written libraries that perform this task for us and all we need concern ourselves with is examination of any parameters passed with the request and performing the logic we wish performed when ever a new request is received.

A library such as `ESP8266WebServer` would be perfect for this task.

See also:

- [ESP8266WebServer](#)

Tasker

Tasker is an Android application that automates and scripts tasks to be executed on an Android device. Using Tasker we can create a task which is defined as a sequence of commands and actions to be executed. Next we can create a Profile which maps an event, that when detected, executes a task. Although this is useful, how does that relate to an ESP8266? Imagine that the event that occurs is an ESP8266 sending a message to your phone. With that notion, an ESP8266 can, effectively, trigger anything that one might be able to do with such a phone. For example, it might make a phone call, send an SMS message or capture a photograph.

See also:

- [Tasker home page](#)
- YouTube: [Tasker 101 Tutorials](#)

AutoRemote

Following on from our discussion of Tasker above, we now have an admission. It appears that Tasker does **not** have the ability to listen for incoming TCP/IP based events and messages. However, because Tasker is extensible and developers can

write plug-ins for it, Tasker can be augmented. One such augmentation is the AutoRemote plugin. Using that plugin, a TCP/IP message can then be sent and received by AutoRemote which can then act as a source of events for Tasker.

With AutoRemote configured as a Tasker plugin, we can configure it to listen for HTTP requests. This causes AutoRemote to listen on TCP port number 1817. The data it is listening for is an HTTP request. For example:

```
http://<phone ip>:1817/sendmessage?message=1
```

With both Tasker and AutoRemote installed, it will still not be listening for incoming WiFi messages over a local WiFi environment unless we are Internet connected. We must run a Tasker Task called "AutoRemote WiFi".

For example, in Tasker:

1. Create a new profile triggered by Event → System → Device Boot
2. Create a New Task associated with the profile
3. Add an action from Plugin → AutoRemote → Wifi
4. In the configuration for the action, check "Wifi Service"

What this will do is start the Wifi Service whenever the device (Android) boots.

Unfortunately, AutoRemote has a serious drawback. It doesn't allow Tasker to send a response back in the original REST request that might contain data that could be used. For example, if we wish to use AutoRemote to send a request that returned the current GPS location, that is simply not possible.

When an AutoRemote request arrives, it sets a number of variables within the Tasker environment that can be used as parameters to Tasker tasks. These include:

- %armessage
- %arpar()
- %arcomm
- %artime
- %arfiles
- %arsenderbtmac
- %arsenderid
- %arsenderlocalip
- %arsendername
- %arsenderpublicip

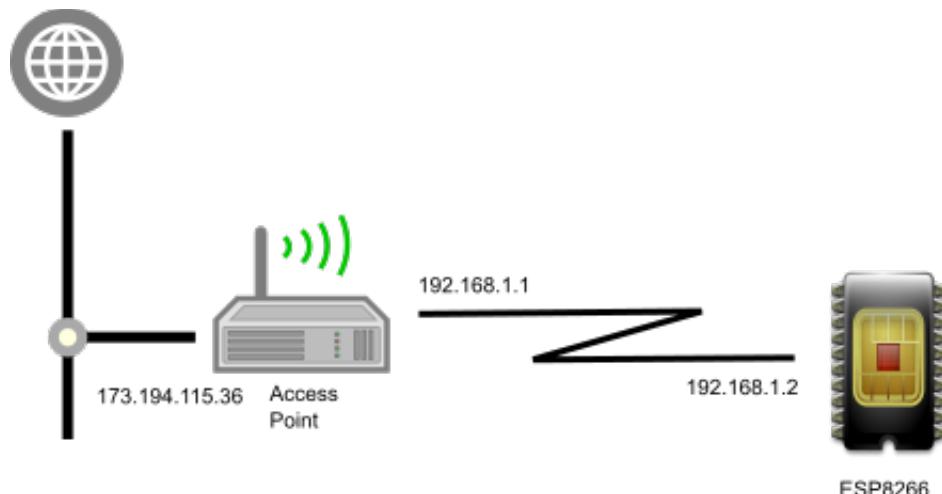
- %arsendertype
- %arvia
 - wifi

See also:

- [AutoRemote home page](#)

DuckDNS

I anticipate that in most folks houses there is a WiFi access point that either directly or through a modem, connects to the Internet. Since the WiFi access point offers a local network to which the ESP8266 can join, we now see that the ESP8266 can reach the outside world through the access point. However, what about the reverse? What if we want a client on the Internet to reach our ESP8266. How could we achieve that?



If we look at the above diagram (all IP address made up), we see that the ESP8266 knows its own IP address as 192.168.1.2. However, this can't be "shared" with the Internet as that is a local address and not a global IP address. What would need to be shared is the IP address of the access point as seen on the Internet.

One way to achieve that is through the use of a service provider such as DuckDNS. This free service allows you to register a name. Your device (usually a PC) periodically sends a request to the DuckDNS web site saying "Hello ... I am here!". The return address of that request is always the IP address of your access point connected to the Internet and hence DuckDNS learns your external address. Later, someone (perhaps a third party) can ask "What is the IP address" of the name you registered and that address is made available. Essentially, DuckDNS acts as a real-time broker of logical names to IP addresses.

If you are concerned that "some scary person" can learn the IP address of your access point ... then don't use DuckDNS. However, for the majority of us, our modem/router/access point prevents incoming traffic from reaching us and essentially blocks anything we don't want. But wait ... won't this also block requests to the ESP8266? The answer is "yes it will" which is why you have to define port-forwarding. Port forwarding a function of your modem/router/access point that says that when a request arrives for a given port location, automatically forward it to an IP address on your local network ... for example, the network address of your ESP8266.

<https://www.duckdns.org/update?domains=XXX&token=XXX&ip=>

Mobile apps

Blynk

See also:

- [Blynk home page](#)

Sample Snippets

There are times when all we need is a snippet of code that we can copy to achieve a task. Here we present a set of such snippets that may of use simply by copying and pasting them.

Forming a TCP connection

Here we see a snippet of code that can be used to make a TCP/IP connection.

```
#define REMOTE_PORT 80
#define REMOTE_IP "216.58.218.206"
struct espconn conn1;
esp_tcp tcp1;

void connectCB(void *arg) {
    os_printf("We have connected\n");
}

void errorCB(void *arg, sint8 err) {
    os_printf("We have an error: %d\n", err);
}

void makeConnection() {
    conn1.type = ESPCONN_TCP;
    conn1.state = ESPCONN_NONE;
    conn1.proto.tcp = &tcp1;
    conn1.proto.tcp->remote_port = REMOTE_PORT;
    *((uint32 *)conn1.proto.tcp->remote_ip) = ipaddr_addr(REMOTE_IP);
    espconn_regist_connectcb(&conn1, connectCB);
    espconn_regist_reconcb(&conn1, errorCB);
    espconn_connect(&conn1);
    os_printf("We have asked for a connection!");
}
```

Sample applications

Reading and reviewing sample applications is good practice. It allows you to study what others have written and see if you can understand each of the statements and the program flow as a whole.

Sample - Light an LED based on the arrival of a UDP datagram

In this sample we will have the ESP8266 become a WiFi station and connect. It will start to listen for incoming datagrams and if the first byte of received data is the character "1", it will light an LED. If the character is "0", it will extinguish the LED.

Here is the full code of the application with commentary following:

```
#include <ets_sys.h>
#include <osapi.h>
#include <os_type.h>
#include <gpio.h>
#include <user_interface.h>
#include <espconn.h>
#include <mem.h>
#include "driver/uart.h"

#define LED_GPIO 15

LOCAL struct espconn conn1;
LOCAL esp_udp udp1;

LOCAL void recvCB(void *arg, char *pData, unsigned short len);
LOCAL void eventCB(System_Event_t *event);
LOCAL void setupUDP();
LOCAL void initDone();

LOCAL void recvCB(void *arg, char *pData, unsigned short len) {
    struct espconn *pEspConn = (struct espconn *)arg;
    os_printf("Received data!! - length = %d\n", len);
    if (len == 0 || (pData[0] != '0' && pData[0] != '1')) {
        return;
    }
    int v = (pData[0] == '1');
    GPIO_OUTPUT_SET(LED_GPIO, v);
} // End of recvCB

LOCAL void initDone() {
    wifi_set_opmode_current(STATION_MODE);
    struct station_config stationConfig;
    strncpy(stationConfig.ssid, "myssid", 32);
    strncpy(stationConfig.password, "password", 64);
    wifi_station_set_config_current(&stationConfig);
    wifi_station_connect();
```

```

} // End of initDone

LOCAL void setupUDP() {
    conn1.type = ESPCONN_UDP;
    conn1.state = ESPCONN_NONE;
    udp1.local_port = 25867;
    conn1.proto.udp = &udp1;
    espconn_create(&conn1);
    espconn_regist_recvcb(&conn1, recvCB);
    os_printf("Listening for data\n");
} // End of setupUDP

LOCAL void eventCB(System_Event_t *event) {
    switch (event->event) {
    case EVENT_STAMODE_GOT_IP:
        os_printf("IP: %d.%d.%d.%d\n", IP2STR(&event->event_info.got_ip.ip));
        setupUDP();
        break;
    }
} // End of eventCB

void user_rf_pre_init(void) {
}

void user_init(void) {
    uart_init(BIT_RATE_115200, BIT_RATE_115200);

    // Set GPIO15 as a GPIO pin
    PIN_FUNC_SELECT(PERIPH_IO_MUX_MTDO_U, FUNC_GPIO15);

    // Call "initDone" when the ESP8266 has initialized
    system_init_done_cb(initDone);
    wifi_set_event_handler_cb(eventCB);
} // End of user_init

```

Control starts in the `user_init()` function where we setup the UART baud. In this example, we have chosen GPIO15 as our output pin so we map the function of the physical pin called "`MTDO_U`" to the logical function of "`GPIO15`". We register a function called `initDone()` to be called when initialization of the device is complete and we also register a function called `eventCB()` to be called when WiFi events arrive indicating a change of state.

With these items having been setup, we return control back to the OS. We expect to be called back through `initDone()` when the device is fully ready for work. In `initDone()` we define ourselves as a Wifi Station and name the access point with its password that we wish to use. Finally we ask for a connection to the access point.

If all goes well, we will be connected to the access point and then be allocated an IP address. Both of these will result in events being generated which will cause us to wake up in `eventCB()`. The only event we are interested in seeing is the allocation of the IP

address. When we are notified of that, we call the function called `setupUDP()` to initialize our UDP listening environment.

In `setupUDP()`, we create a `struct espconn` control block defined for UDP and configured to listen on our chosen port of 25867. We also register a receive callback to the function `recvCB()`. This will be called when new data arrives. At this point, all our setup is completed and we have a device connected to the WiFi network listening on UDP port 25867 for datagrams.

When a datagram arrives, we wake up in `recvCB()` having been passed in the datagram data. We check that we actually have data and that it is good ... if not, we end the callback straight away.

Finally, we look at the first character of the data and, based on its value, change the output value of the GPIO. The physical GPIO is wired to an LED and a resistor.

If a character of '1' is transmitted, the output of `GPIO15` goes high and the LED lights. If the character value is '0', the output of `GPIO15` goes low, and the LED is extinguished.

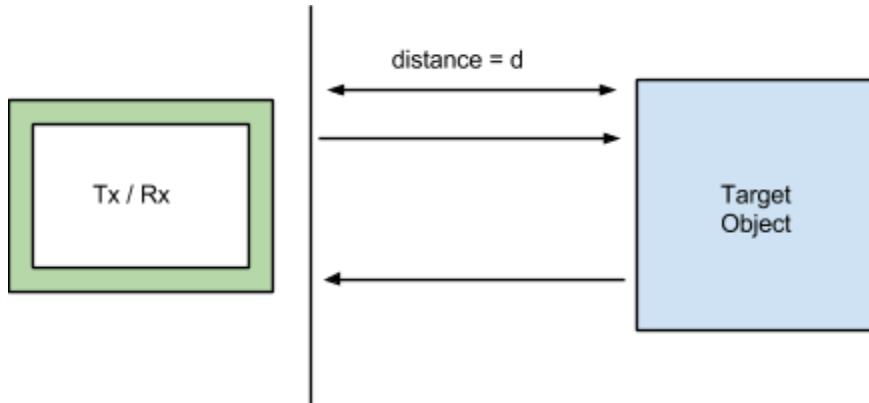
Sample - Ultrasonic distance measurement

The HC SR-04 is an ultrasonic distance measurement sensor.



Send a minimum of a 10us pulse to `Trig` (low to high to low). Later, `Echo` will go low/high/low. The time that `Echo` is high is the time it takes the sonic pulse to reach a back-end and bounce back.

Speed of sound is 340.29 m/s ($340.29 * 39.3701$ inches/sec). Call this V_{sound} .



If T_{echo} is the time for echo response then $d = (T_{echo} * V_{sound}) / 2$.

Also the equation for expected T_{echo} lengths is given by:

$$T_{echo} = 2d/V_{sound}$$

For example:

Distance	Time
1cm	$2 * 0.01 / 340 = 0.058 \text{ msec} = 59 \text{ usecs}$
10cm	$2 * 0.1 / 340 = 0.59 \text{ msec} = 590 \text{ usecs}$
1m	$2 * 1 / 340 = 5.9 \text{ msec} = 5900 \text{ usecs (5.9 msec)}$

Because the `Echo` response is a 5V signal, it is vital to reduce this to 3.3V for input into the ESP8266. A voltage divider will work. The pins on the device are:

- `VCC` – The input voltage is 5V.
- `Trig` – Pulse (low to high) to trigger a transmission ... minimum of 10usecs.
- `Echo` – Pulses low to high to low when an echo is received. Warning, this is a 5V output.
- `Gnd` – Ground.

To drive this device, we need to utilize two pins on the ESP8266 that we will logically call `Trig` and `Echo`. In my design, I set `Trig` to be GPIO4 and `Echo` to be GPIO5.

Our design for the application will not include any networking but it should be straightforward to add it as needed. We will setup a timer that fires once a second which is how often we wish to take a measurement. When the timer wakes up, we will pulse `Trig` from low to high and back to low holding high for 10 microseconds. We will now record the time and start polling the `Echo` pin waiting for it to go high. When it does, we will record the time again and subtracting one from the other will tell us how

long it took the sound to bounce back. From that we can calculate the distance to an object. If no response is received in 20 msecs, we will assume that there was no object to detect. We will then log the result to the Serial console.

An example program that performs this design is shown next:

```
#define TRIG_PIN 4
#define ECHO_PIN 5

os_timer_t myTimer;

void user_rf_pre_init(void) {
}

void timerCallback(void *pArg) {
    os_printf("Tick!\n");
    GPIO_OUTPUT_SET(TRIG_PIN, 1);
    os_delay_us(10);
    GPIO_OUTPUT_SET(TRIG_PIN, 0);
    uint32 val = GPIO_INPUT_GET(ECHO_PIN);
    while(val == 0) {
        val = GPIO_INPUT_GET(ECHO_PIN);
    }
    uint32 startTime = system_get_time();
    val = GPIO_INPUT_GET(ECHO_PIN);
    while(val == 1 && (system_get_time() - startTime) < (20 * 1000)) {
        val = GPIO_INPUT_GET(ECHO_PIN);
    }
    if (val == 0) {
        uint32 delta = system_get_time() - startTime;
        // Calculate the distance in cm.
        uint32 distance = 340.29 * 100 * delta / (1000 * 1000 * 2);
        os_printf("Distance: %d\n", distance);
    } else {
        os_printf("No echo!\n");
    }
} // End of timerCallback

void user_init(void) {
    uart_init(BIT_RATE_115200, BIT_RATE_115200);
    // Setup ultrasonics pins as GPIO
    setAsGpio(TRIG_PIN);
    setAsGpio(ECHO_PIN);
    setupBlink(15);
    // Set the trigger pin to be default low
    GPIO_OUTPUT_SET(TRIG_PIN, 0);
    os_timer_setfn(&myTimer, timerCallback, NULL);
    os_timer_arm(&myTimer, 1000, 5);
} // End of user_init
```

Once this has been written and tested, we will make a second pass at the puzzle but this time using an interrupt to trigger the response to the echo.

See also:

- GPIOs

Sample - WiFi Scanner

A WiFi scanner is an application which periodically scans for available WiFi networks and shows them to the user. In our design, we will scan periodically and remember the set of networks we find. When we perform re-scans, we will check to see if each of the networks located is a network we have previously seen and, if not, list it to the user. We will also keep a "last seen" time for each network and if a network has not been seen for a minute, then we will forget about it such that if it appears again, we will once more list it to the user.

To illustrate our design, we will break the solution into a number of parts. The first part will be to register a callback function that is called every 30 seconds. This callback will be responsible for requesting a WiFi scan using `wifi_station_scan()`. This takes a callback function which itself will be invoked when the scan is complete.

When the scan completes, we will have a new list of detected networks. We will walk this list and for each network detected, determine if we have seen it before. If we have, we will update the last seen time. If not, we will add it to the list of previously seen networks and log it to the user.

A second timer callback will run once a minute and will walk the list of previously seen networks. If any of them are older than a minute, we will remove them.

See also:

- Scanning for access points

Sample - Working with micro SD cards

A micro SD card is a small portable storage device that can host gigabytes of data. Through the use of an adapter, a micro SD card can be leveraged in conjunction with an ESP8266 providing read and write access to data that persists across ESP8266 restarts.

Sample - Playing audio from an event

In this sample, we wish an event to be detected by the ESP8266 which, when it happens, causes an audio file to be played.

Sample - A changeable mood light

NeoPixels are LEDs that are driven by a single data line of high speed signaling. Most NeoPixels have a +ve and ground voltage source as well as a data line for input and a

data line for output. The output of one NeoPixel can be fed into the input of the next one to produce a string of such LEDs. The input data to the LED is a stream of 24 bits of encoded data which should be interpreted as 8 bits for the red channel, 8 bits for the green channel and 8 bits for the blue channel. Each channel can thus have a luminance value of between 0 and 255. By mixing the values for each of the channels together, you can color an LED to any color you may choose. After sending in a stream of 24 bits, if we send in a second stream of 24 bits quickly after the first stream, the second stream is "pushed" through to the next LED in the chain. This can be repeated as far as desired. If we pause sending in data, the current values are "latched" into place and each LED remembers its own value.

The timings of the data signals for these LEDs can be quite tricky but fortunately great minds have already built fantastic libraries for driving them correctly so we need not concern ourselves with these low level timings and can instead concentrate on devising interesting projects and purposes to which the LEDs can be placed. There are a number of different types of these LEDs with the most common ones being known as WS2811, WS2812 or PL9823.

Within the Espruino JavaScript environment, a method called `neopixelWrite()` can be found. This takes two parameters. The first is the ESP8266 GPIO pin that will be used as the source of the signals to the LEDs. It is to this pin that the LEDs should be wired. The pin used for data output from the ESP8266 to the NeoPixels should be set in GPIO output mode. For example:

```
pinMode(pin, "output");
```

The second parameter is an array of integers. The values of the array should be supplied in groups of 3 corresponding to the 3 channels of red, green and blue. For example, if we had one NeoPixel connected to GPIO4 on an ESP8266 and we wanted to set it to all red, we might code:

```
neopixelWrite(new Pin(4), [255, 0, 0]);
```

If we wanted the next pixel to be green while the first is red, we might write:

```
neopixelWrite(new Pin(4), [255, 0, 0, 0, 255, 0]);
```

Again, there is no obvious limit to the number of LEDs we can string together.

Now that we see that we can set the brightness and color of an LED, let us look at how we might design some code to do something. Let us imagine that we had a string of 16 LEDs and wanted to make them the same color ... we might define a function as follows:

```
function colorLeds(red, green, blue) {  
  var data = [];  
  for (var i=0; i<16; i++) {  
    data.push(green);  
    data.push(red);  
  }  
  neopixelWrite(new Pin(4), data);  
}
```

```

        data.push(blue);
    }
    esp8266.neopixelWrite(NodeMCU.D2, data);
}

```

If we call this function with the correct red, green and blue values, it will set the LEDs string correctly.

Now let us go one step further. Imagine that we received a network REST request that described the color that we want the LEDs to show. A complete application may be:

```

var esp = require("ESP8266");
var NodeMCU = {
    // D0: new Pin(16),
    D1 : new Pin(5),
    D2 : new Pin(4),
    D3 : new Pin(0),
    D4 : new Pin(2),
    D5 : new Pin(14),
    D6 : new Pin(12),
    D7 : new Pin(13),
    D8 : new Pin(15),
    D9 : new Pin(3),
    D10 : new Pin(1)
};

pinMode(NodeMCU.D2, "output");

function colorLeds(red, green, blue) {
    var data = [];
    for (var i=0; i<16; i++) {
        data.push(green);
        data.push(red);
        data.push(blue);
    }
    esp.neopixelWrite(NodeMCU.D2, data);
}

function beServer() {
    var http = require("http");
    var httpServer = http.createServer(function(request, response) {
        print(request);
        var partsOfUrl = request.url.split "?";
        if (partsOfUrl.length > 1) {
            var options = partsOfUrl[1].split '&';
            var optionsObj = {};
            for (var i=0; i<options.length; i++) {
                var splitEquals = options[i].split '=';
                optionsObj[splitEquals[0]] = splitEquals[1];
            }
            print("Final obj: " + JSON.stringify(optionsObj));
            if (optionsObj.color !== null) {
                var red = parseInt(optionsObj.color.substr(0,2), 16);
                var green = parseInt(optionsObj.color.substr(2,2), 16);
                var blue = parseInt(optionsObj.color.substr(4,2), 16);
            }
        }
    });
}

```

```

        print("red: " + red + ", green: " + green + ", blue: " + blue);
        colorLeds(red, green, blue);
    }
}
print("Result url = " + url);
response.writeHead(200, {
    "Access-Control-Allow-Origin": "*"
});
response.end("");
}); // End of on new browser request

httpServer.listen(80);
print("Now being an HTTP server!");
} // End of beServer

var ssid      = "ssid";
var password = "password";

// Connect to the access point
var wifi = require("wifi");
print("Connecting to access point.");
wifi.connect(ssid, password, null, function(err, ipInfo) {
    if (err) {
        print("Error connecting to access point.");
        return;
    }
    var ESP8266 = require("ESP8266");
    print("Connect says that we are now connected!!!");
    print("Starting web server at http://" + ESP8266.getAddressAsString(ipInfo.ip)
+ ":80");
    beServer();
});

```

When this application runs, it connects to the local WiFi access point and then starts listening for incoming REST requests. A rest request is expected to have a query parameter at the end with the format `color=value` where value is encoded as 6 hex characters corresponding to the color. Finally, we can write a web page that will present a color picker and, when we pick a color, send a REST request to the ESP8266 to illuminate the LEDs appropriately. Here is a sample web page to achieve this task:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Set LED colors</title>

<link
    href="http://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.11.2/jquery-ui.min.css"
    rel="stylesheet" type="text/css" />
<script
src="http://cdnjs.cloudflare.com/ajax/libs/require.js/2.1.15/require.min.js"></script>
<link rel='stylesheet' href='spectrum.css' />
<script>
    require

```

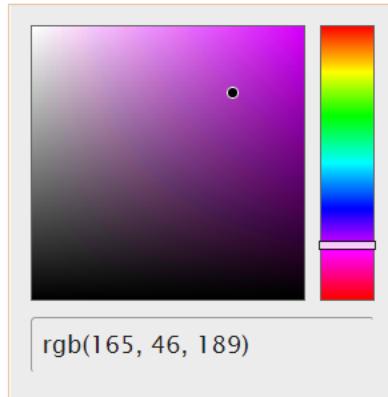
```

.config({
  baseUrl : "src",
  paths : {
    "jquery" : "http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min",
    "jquery-ui" : "http://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.11.2/jquery-
i.min",
  },
  shim : {
    "jquery-ui" : {
      deps : [ "jquery" ],
      exports : 'jQueryUI'
    }
  }
})
// End of shims
});

require([ "jquery", "spectrum", "jquery-ui" ], function($) {
$(function() {
  var allowHttp = true;
  $("#flat").spectrum({
    flat : true,
    preferredFormat : "rgb",
    move : function(color) {
      if (allowHttp) {
        allowHttp = false;
        $.ajax({
          url : "http://192.168.1.10",
          data : {
            color : color.toHex()
          },
          success: function() {
            allowHttp = true;
          },
          error: function() {
            allowHttp = true;
          }
        });
      }
    },
    showInput : true,
    showButtons : false
  });
}); // End of on load
}); // End of require
</script>
</head>
<body>
  <div id="flat" style="width:500px; height: 500px;"></div>
</body>
</html>

```

The end result as seen on the web page looks as follows:



Selecting a new color causes the data to sent to the ESP8266 which colors the LEDs appropriately with the over-all end result being the ability to change the mood light of the LED string.

Sample - Bootstrapping networking

Imagine that you are given a fantastic ESP8266 application and you install it on your device. The likelihood is that it will use WiFi based networking. Now comes the interesting question of just how do you bootstrap the device?

Sample Libraries

There are times when commonly used functions can be captured and reused over and over. This section describes just such a set of functions which have been collected. The source for these functions has been placed in Github at <location to be provided>.

The functions, when compiled, are placed in a library called `libcommon.a`. This can then be linked within your Makefile so that unresolved references to these functions can be satisfied.

A header file called "`common.h`" is all that one needs to add into your own applications.

Function list

`authModeToString`

Given an `AUTH_MODE`, return a string representation of the mode.

```
char *authModeToString(AUTH_MODE mode)
```

checkError

Check a return code for an error.

```
void checkError(sint8 err)
```

Check the `err` code for an error and if it is one, log it.

delayMilliseconds

Delay for a period of milliseconds.

```
void delayMilliseconds(uint32 milliseconds)
```

The `milliseconds` parameters is the number of milliseconds to delay before returning.

dumpBSSINFO

Dump an instance of struct `bss_info` to the log.

```
void dumpBSSINFO(struct bss_info *bssInfo)
```

dumpEspConn

Dump to the log a decoded representation of the `struct espconn`.

```
void dumpEspConn(struct espconn *pEspConn)
```

dumpRestart

Dump the restart information to the log.

```
void dumpRestart()
```

See also:

- Exception handling

dumpState

Dump the WiFi station state to the log.

```
void dumpState()
```

See also:

- Error: Reference source not found

errorToString

Given an error code, return a string representation of it.

```
char *errorToString(sint8 err)
```

eventLogger

Write a WiFi event to the log.

```
void eventLogger(System_Event_t *event)
```

We can register this function as a callback for a WiFi event. Write the event data to the log.

See also:

- Handling WiFi events

eventReasonToString

Convert an event reason to a string representation.

```
char *eventReasonToString(int reason)
```

Some of the WiFi event callbacks can return a reason value that is an encoding of the reason that something failed. This function returns a string representation of the int value code.

flashSizeAndMapToString

Return a string representation of the flash size and map.

```
char *flashSizeAndMapToString()
```

setAsGpio

Set a pin to be used as a GPIO.

```
void setAsGpio(uint8 pin)
```

Set the GPIO supplied as `pin` to be GPIO function.

See also:

- GPIOs
- GPIOs

setupBlink

Setup a blinking LED on the given pin.

```
void setupBlink(uint8 blinkPin)
```

The `blinkPin` parameter is the pin to use for blinking.

`toHex`

Convert an array of bytes to a hex string.

```
uint8 *toHex(uint8 *ptr, int size, uint8 *buffer)
```

Convert the bytes pointed to by `ptr` for `size` bytes into a hex string. The `buffer` parameter will be where the result will be stored. It must be $2 * \text{size} + 1$ bytes in length (or more). Each byte is 2 hex characters plus a single byte NULL terminator at the end. The function returns the start of the buffer.

Using FreeRTOS

When we think of a modern computer, we quickly realize that it has an operating system of some sort. Common examples of these are Microsoft Windows or Linux. The purpose of an operating system is to provide an interface between software applications and the underlying hardware infrastructure. If it wasn't for an operating system, each application would likely have to perform its own similar implementation of such functions which would be a waste. Why not write it once and provide an abstraction layer upon which higher level functions (such as applications) can be built. The capabilities of operating systems on PCs are very similar. They handle memory management, hardware I/O (reading from keyboards and mice and driving graphics cards), task management (multiple programs running concurrently), disk and file system interactions and much more. Early operating systems provided basic functions while today's operating systems have become richer and richer to the point where they may no longer be considered as just operating systems. Since when did an operating system need to provide Freecell or Minesweeper?

If we rewind the clock and start again and look to the core aspects of an operating system, we come to today's FreeRTOS. FreeRTOS is an open source operating system that provides very basic functions to higher level applications ... again ... the core notion of the purpose of an operating system in the first place. However, FreeRTOS is designed for embedded systems such as the ESP8266. It is orders of magnitude simpler than other operating systems such as Linux but this is by design.

FreeRTOS has been ported to a wide variety of hardware platforms including the Xtensa CPUs used in the ESP8266. When compiled, it results in a library that is under 5K Bytes in size.

The core functions it provides are:

- memory management

- task management
- API synchronization

See also:

- Free RTOS home page
- [Study of an operating system: FreeRTOS](#)
- Github: [espressif/ESP32_RRTOS_SDK](#)

The architecture of a task in FreeRTOS

Let us start with the notion of a task. A task is a piece of work that we wish to perform. If you wish, you can think of this as a C language function. For example:

```
int add(int a, int b) {
    return a + b;
}
```

could be considered a task ... although this would be ridiculously simple. Generically, think of a task as the execution of a piece of C code that you have authored. We normally think of code running from its start all the way through to its end ... however, this is not necessarily the most efficient way to proceed. Consider the idea of an application which wishes to send some data over the network. It may wish to send a megabyte of data ... however it may also find that it can only send 100K at a time before it has to wait for the transmitted data to be delivered. In that story, it would send 100K and wait for the transmission to complete, send the next 100K and wait for that transmission to complete and so on. But what of those periods of time where the code is waiting for a previous transmission to complete? What is the CPU doing at those times?

The chances are that it is doing nothing but monitoring for the flag that states that the transmission has completed. This is a waste. In theory the CPU could be performing other work (assuming that there is in fact other work that could be performed). If there is indeed other work available, we could "context switch" between these work items such that when one blocks waiting for something to happen, control could be passed to another to do something useful.

If we call each piece of work "a task", that is the value of a task in FreeRTOS. The task represents a piece of work to be performed but instead of assuming that the work will quickly go from start to end, we are declaring that there may be times within the work where it can relinquish control to other work (tasks).

With this in mind, we should think about how a task is created. There is an API provided by FreeRTOS called "`xTaskCreate()`" which creates an instance of a task.

Here it is important to realize that a task is a logical abstraction. There isn't anything specific provided in the CPU that knows what a task is. Instead, it is the operating system (FreeRTOS in our case) that is providing the model of the task for us.

If we think deeply about a task, we can conceive of the task having a state. At any given time, either a task is running or it is not running. A task that is running is one that is actively using the CPU (i.e. not waiting for anything else to happen). A task that is not running is one that doesn't have the CPU. For example, if we created two tasks, one of them would be running and the other not running. If the one that is running reaches a point where it can no longer perform meaningful work, it will relinquish CPU control and become not running. The other task then has the opportunity to become running.

Going even deeper, when a task is not running, it may be "not running" for a particular reason ... such as:

- Blocked waiting for something to complete
- Suspended by the user
- Ready to run such that when the task that is running is no-longer running, this task is eligible to become running

In FreeRTOS we define a task as a C function that takes a `void *` parameter. For example,

```
void myTask(void *myParameters)
```

might be a signature for a task function.

A task function is expected to run forever. Should it need to end, it should clean itself up before returning by invoking `vTaskDelete()`.

When a task relinquishes control back to the OS, the OS then may have a choice between multiple tasks as to which one should become running. This selection process is called "scheduling". FreeRTOS uses the concept of a "priority" to determine which task to run next. Each task that is ready to run is considered a potential candidate and the one that has the highest priority will become the one that is running.

When coding directly to FreeRTOS in non-ESP8266 environment, one would normally have to make a call to `vTaskStartScheduler()` to ensure that the task scheduler is operational. This should not be attempted in the ESP8266 environment as the internals of the ESP8266 environment have already registered other tasks and already started the scheduler.

There are a number of timer related functions within FreeRTOS that work on the notion of "ticks" where a tick is a unit of time. In the ESP8266 FreeRTOS the tick interval is 1/100th of a second (10 msecs).

Blocking and synchronization within RTOS

With the notion of parallel processing tasks within RTOS, we must have a mechanism to synchronize actions between tasks. For example, imagine a task that produces data and a second task that consumes data produced by the first. The producing task must have a mechanism that describes that data has been produced and the consuming task must have a mechanism to block waiting for data to be produced.

One way to achieve this is through the notion of an "event group". Think of an event group as a set of flags that can have the value "0" or "1". A task can set the value of a flag and a second task can be configured to wait (block) until a flag transitions from "0" to "1". What this means is that there is an asynchronous and loosely coupled communication through the use of these flags. From an implementation perspective, RTOS provides a data type called an "event group handle" that is implemented by the opaque data type called "`EventGroupHandle_t`". An instance of this is created through a call to `xEventGroupCreate()`. We should assume that that an event group handle can contain a maximum of 8 distinct flags that are identified as 0 through 7. We can set the flags within an event group using `xEventGroupSetBits()` and clear flags using `xEventGroupClearBits()`. Should we need to obtain the values of an event group, we can call `xEventGroupGetBits()`. Simply toggling bits isn't that useful, but we get into the core of the story with the `xEventGroupWaitBits()` function call. When invoked, it causes the caller to be blocked until a named bit or bits becomes set.

Lists within RTOS

FreeRTOS provides list processing functions.

ESP8266 - Building apps for RTOS

Espressif distribute an SDK for building RTOS apps for the ESP8266. This SDK is available from Github. My personal choice for retrieving the SDK is to use the latest version of Eclipse and use its in-built Git retrieval tools.

The FreeRTOS version supplied by Espressif appears to be v7.5.2. The latest available from FreeRTOS themselves appears to be v8.2.3.

Eclipse also provides an environment for C program compilation. The suggested C→Object compilation flags are:

Parameter/flag	Meaning
-g	
-Wpointer-arith	
-Wundef	
-Werror	
-Wl, -El	
-fno-inline-functions	
-nostdlib	
-mlongcalls	
-mtext-section-literals	
-ffunction-sections	
-fdatasections	

For linking we use the following linker flags:

-L\$(SDK_PATH)/lib	
-Wl, --gc-sections	
-nostdlib	
-T\$(LD_FILE)	
-Wl, --no-check-sections	
-u call_user_start	
-Wl, -static	
-Wl, --start-group	
-lminic	
-lgcc	
-lhal	
-lphy	
-lpp	
-linet80211	
-lwpa	
-lcrypto	
-lmain	
-lfreertos	
-llwip	

When configuring the CDT in Eclipse, the following settings are suggested:

- C/C++ Build → Environment – ESP8266_SDK_ROOT = <Path to RTOS SDK>

- C/C++ General → Paths and Symbols – GNU C – {ESP8266_SDK_ROOT}/include/espressif
- C/C++ General → Paths and Symbols – GNU C – {ESP8266_SDK_ROOT}/include/lwip
- C/C++ General → Paths and Symbols – GNU C – {ESP8266_SDK_ROOT}/include/lwip/ipv4

Having compiled the source code into object files and then linked the object files into an ELF formatted executable, what remains is to split this executable into the sections to be loaded into ESP8266 flash memory at different locations. This will result in two files for loading into flash. One file will be the data that will eventually be loaded into RAM at runtime while the other will be available as addressable flash memory.

We can use `esptool_ck` for this task:

```
esptool_ck -eo bin.elf -bo app_0x00000.bin -bs .text -bs .data -bs .rodata -bs
.irom0.text -bc -ec
esptool_ck -eo $bin.elf -es .irom0.text ap_0x10000.bin -ec
```

After flashing to the ESP8266, we may find that the following boot messages are produced:

```
pp_task_hdl : 3ffef4e0, prio:13, stack:512
pm_task_hdl : 3ffefdb0, prio:1, stack:176
tcpip_task_hdl : 3ffef260, prio:10,stack:512
idle_task_hdl : 3ffef300,prio:0, stack:176
tim_task_hdl : 3fff1428, prio:2,stack:256
xPortStartScheduler
frc2_timer_task_hdl:3fff1938, prio:12, stack:512

OS SDK ver: 1.3.0(68c9e7b) compiled @ Nov 2 2015 18:53:21
phy ver: 484, pp ver: 9.9

SDK version:1.3.0(68c9e7b)
```

See also:

- Github: [espressif/ESP8266_RRTOS_SDK](#)
-

Consoles with RTOS

The default debug stream is written to UART0 at a baud rate of 74880.

Debugging tips

If things get weird, erase all the flash of your device and start again.

In FreeRTOS, to cause debugging to be written to UART1, the following can be used:

```
UART_ConfigTypeDef uart_config;
uart_config.baud_rate = BIT_RATE_115200;
uart_config.data_bits = UART_WordLength_8b;
uart_config.parity = USART_Parity_None;
uart_config.stop_bits = USART_StopBits_1;
uart_config.flow_ctrl = USART_HardwareFlowControl_None;
uart_config.UART_RxFlowThresh = 120;
uart_config.UART_InverseMask = UART_None_Inverse;
UART_ParamConfig(UART1, &uart_config);
UART_SetPrintPort(UART1);
```

Developing solutions on Linux

When working in a Linux environments, there are certain tips and techniques which might be useful/valuable.

- When connecting to an SP8266 board using a USB→UART connector, the device may show up under `/dev/ttyUSB0`. If we examine the permissions upon this file, we may find that it is configured as:

```
crw-rw---- root dialout
```

This means that it is accessible by root and users in the `dialout` group. If you wish to flash the ESP8266 through this device, your userid should thus be a member of this group. To add your user to the group, the following Linux command may be used:

```
sudo usermod -a -G dialout <yourUserId>
```

after making the change, you must log out and log back in again.

- A useful terminal client is `GtkTerm`. This tool provides a terminal viewer that can be used to monitor the USB→UART connector to view log and debug messages. It creates a configuration file in `$HOME/.gtktermrc` that can be edited to change the default serial port (eg. `/dev/ttyUSB0`) as well as changing the baud rate to your desired value.
- Another good terminal client is `screen`. `Screen` is a full screen terminal emulator.
- Yet another terminal client is the classic `cu` command. Again, very easy to use. An example of use would be:

```
$ cu --line /dev/ttyUSB0 --speed 115200
```

To quit a `cu` session, enter "`~.`".

Building a Linux environment

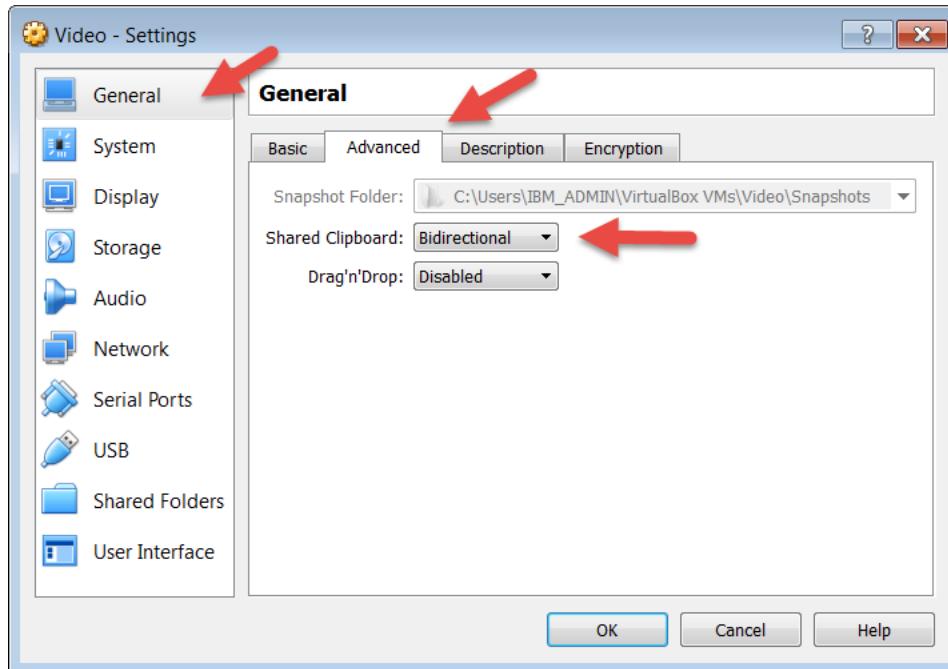
If you don't run Linux natively you may wish to consider running Oracle VirtualBox to host a Linux environment on your Windows or Mac machine. Oracle VirtualBox is an Open Source implementation of an operating system virtualization product. One can download VirtualBox from here:

<https://www.virtualbox.org/>

In my tests, I ran Ubuntu 15.10.

I define a disk size of at least 20GBytes and 2GBytes of RAM. If you have multiple cores, you may want to define those as being available.

After building an image, make sure that you enable the ability to copy and paste between the host OS and the guest OS.



Also make sure that the VirtualBox guest tools are installed.

There are some packages that you really can't do without including:

- git

You can install new packages with:

```
sudo apt-get install <package>
```

Once you have a Linux OS installed, next we want to build a compilation environment.

The popular pfalcon/esp-open-sdk is what we will illustrate.

Before starting the remainder of our build, we must ensure that a number of optional component to Linux are installed as they are mandatory for building the toolchain. The following command may be run to install the set of components we need:

```
sudo apt-get install make unrar autoconf automake libtool-bin gcc g++ gperf \
flex bison texinfo gawk ncurses-dev libexpat-dev python python-serial sed \
git unzip bash help2man wget bzip2
```

First we must execute the command to download the github based project:

```
git clone --recursive https://github.com/pfalcon/esp-open-sdk.git
```

Once downloaded, we can build the solution with:

```
make STANDALONE=y
```

Note that the build needs network access and will access external websites including:

- www.mpfr.org

The build/compilation will take about an hour to complete.

At the conclusion, a set of new directories can be found. Among these are:

- xtensa-lx106-elf/bin – The compiled tools including gcc, objcopy and gdb.
- sdk – A symbolic link to the latest Espressif SDK

We want to add some environment variables into our user's profile:

- Add the `xtensa-lx106-elf/bin` to the PATH
- Export `ESP8266_SDK_ROOT` to the root of the SDK

You will also want to install an `esptool-ck` into your local bin folder.

You will also want to add your userid to the group called dialout.

Here is a sample Makefile for Linux:

```

PROJ_NAME=test1
COMPRT=/dev/ttyUSB0
OBJS=user_main.o uart.o
#
CC=xtensa-lx106-elf-gcc
OBJS=user_main.o uart.o
APP=a.out
ESPTOOL_CK=esptool
CCFLAGS= -Wimplicit-function-declaration -fno-inline-functions -mlongcalls -mtext-section-literals \
-mnno-serialize-volatile -I$(ESP8266_SDK_ROOT)/include -I. -D__ETS__ -DICACHE_FLASH -DXTENSA -DUSE_US_TIMER

LDFLAGS=-nostdlib \
-L$(ESP8266_SDK_ROOT)/lib -L$(ESP8266_SDK_ROOT)/ld -T$(ESP8266_SDK_ROOT)/ld/eagle.app.v6.ld \
-Wl,--no-check-sections -u call_user_start -Wl,--static -Wl,--start-group \
-lc -lgcc -lhal -lphy -lpp -lnet80211 -llwip -lwpa -lmain -ljson -lupgrade -lssl \
-lpwm -lsmartconfig -Wl,--end-group

all: $(PROJ_NAME)_0x00000.bin $(PROJ_NAME)_0x40000.bin

a.out: $(OBJS)
       $(CC) -o a.out $(LDFLAGS) $(OBJS)

$(PROJ_NAME)_0x00000.bin: a.out
       $(ESPTOOL_CK) -eo $< -bo $@ -bs .text -bs .data -bs .rodata -bs .iram0.text -bc -ec || true

$(PROJ_NAME)_0x40000.bin: a.out
       $(ESPTOOL_CK) -eo $< -es .irom0.text $@ -ec || true

.c.o:
       $(CC) $(CCFLAGS) -c $<

clean:
       rm -f a.out *.o *.bin

flash: all
       $(ESPTOOL_CK) -cp $(COMPRT) -cd nodemcu -cb 115200 -ca 0x00000 -cf $(PROJ_NAME)_0x00000.bin
       $(ESPTOOL_CK) -cp $(COMPRT) -cd nodemcu -cb 115200 -ca 0x40000 -cf $(PROJ_NAME)_0x40000.bin

```

And also a simple "hello world" application:

```
#include "osapi.h"
#include "user_interface.h"
#include "uart.h"

#include "espmissingincludes.h"

void uart_init_2(UartBautRate uart0_br, UartBautRate uart1_br);

void systemInitDoneCB() {
    os_printf("Hello World\n");
}

void user_init() {
    uart_init_2(115200, 115200);
    system_init_done_cb(systemInitDoneCB);
}
```

Next we install Java 8. Download from Oracle and extract into /usr/java. For example tar zxvf file. Update JAVA_HOME and PATH.

Now that we have a compilation environment, chances are high we will want an IDE my favorite is Eclipse.

<https://eclipse.org/downloads/>

Run the Eclipse installer

eclipseinstaller by Oomph

type filter text

Eclipse IDE for Java Developers

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven integration and WindowBuilder

Eclipse IDE for Java EE Developers

Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.

Eclipse IDE for C/C++ Developers

An IDE for C/C++ developers with Mylyn integration.

Eclipse IDE for PHP Developers

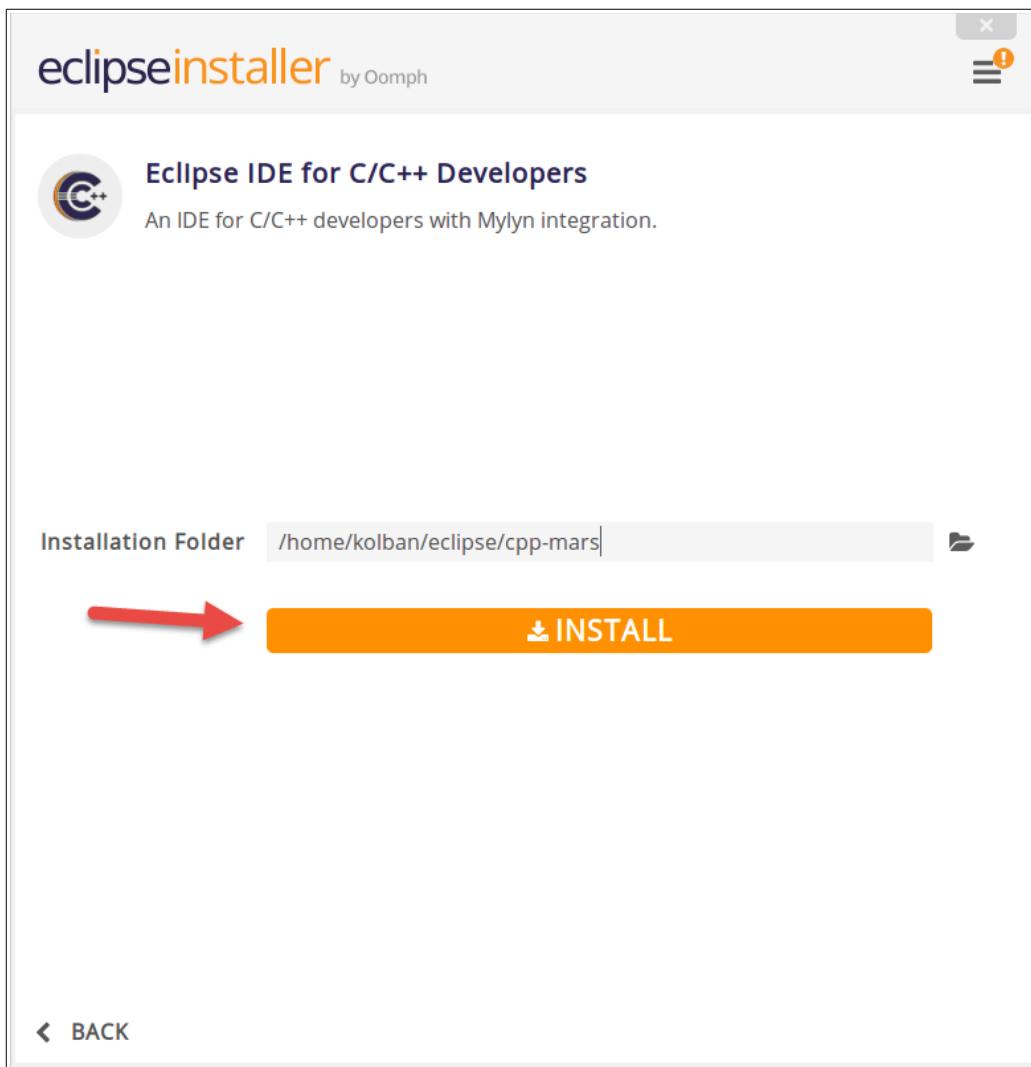
The essential tools for any PHP developer, including PHP language support, Git client, Mylyn and editors for JavaScript, HTML, CSS and XML.

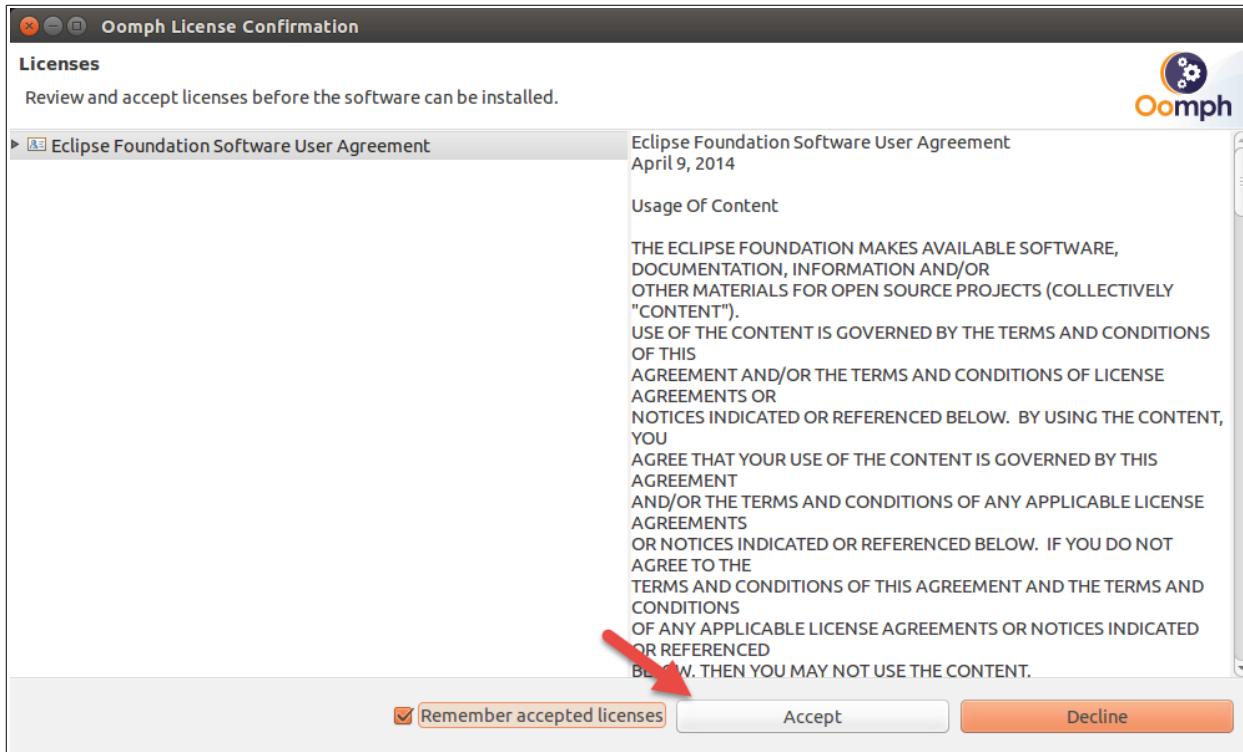
Eclipse IDE for Eclipse Committers

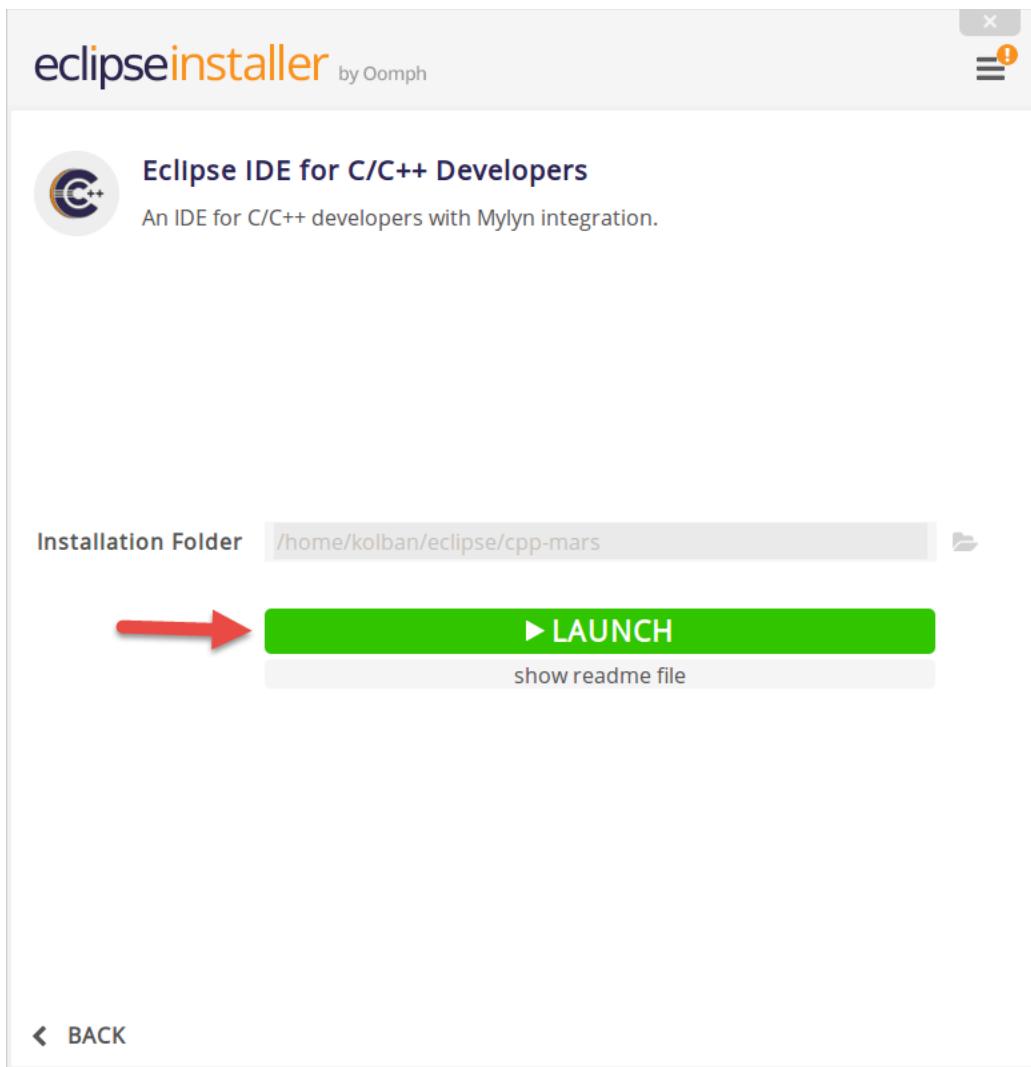
Package suited for development of Eclipse itself at Eclipse.org; based on the Eclipse Platform adding PDE, Git, Marketplace Client, source code and developer...

Eclipse DSL Tools

The essential tools for Java and DSL developers, including a Java Model IDE, a DSL

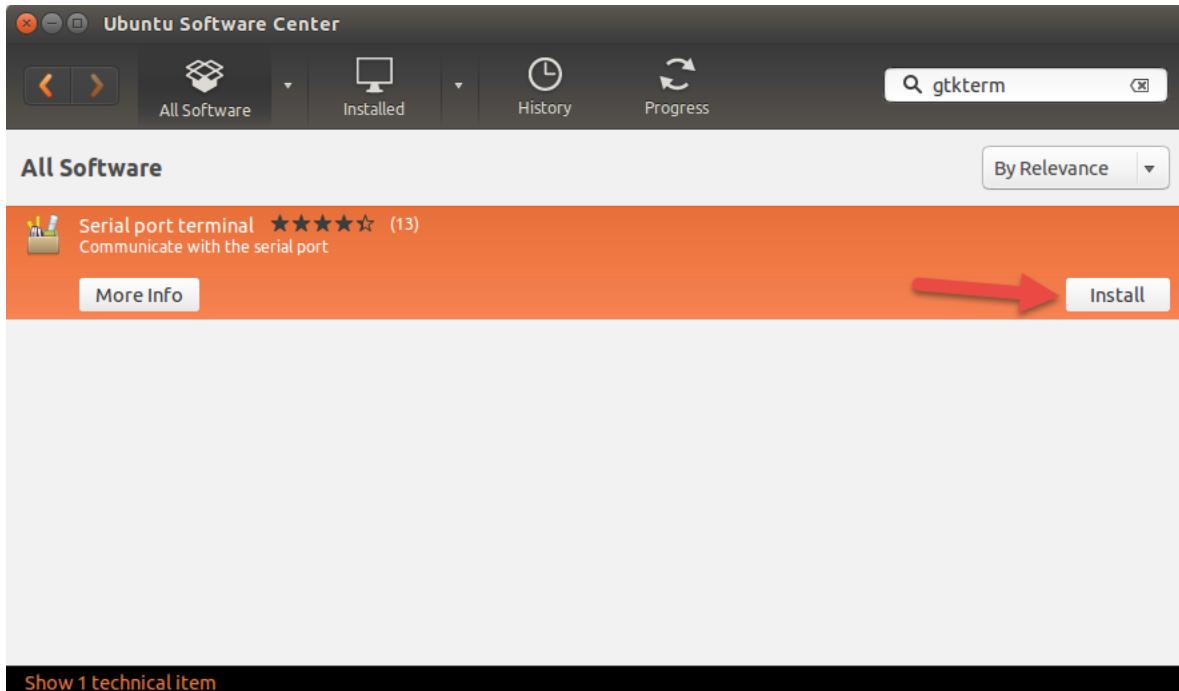






After having launched Eclipse, we want to install the updates. However, for Mars, there won't be any as it is so new.

Install GtTerm



Launch it once to create the `~/.gtktermrc` file. Edit that file and change:

- `port = /dev/ttyUSB0`
- `speed = 115200`

Add the current user to the group dialout

```
sudo usermod -a -G dialout <yourUserid>
```

You will need to logout and login again for this to take effect. You can validate that your userid has the correct group by running the `id` command from a shell:

```
$ id  
uid=1000(kolban) gid=1000(kolban) groups=1000(kolban),4(adm),20(dialout),24(cdrom),  
27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare)
```

Download [igrr/esptool-ck](#) as `esptool` from Github releases. Ensure that it is in your `bin` folder and that `bin` folder is on your path.

Now we are finally ready to build an app. We can download a sample for this story from Github at the following URL:

<https://github.com/nkolban/Sample-ESP8266-App.git>

This sample assumes that the xtensa tools are on your `PATH` and that the environment variable `ESP8266_SDK_ROOT` is properly defined.

I would also install:

- [Chrome](#)

See also:

- [Oracle Virtual Box](#)
- [Ubuntu downloads](#)

API Reference

Now we have a mini reference to the syntax of many of the ESP8266 exposed APIs. Do not use this reference exclusively. Please also refer to the published Espressif SDK Programming Guide.

Some acronyms and other names are used in the naming of APIs and may need some explanation to fully appreciate them:

- dhcpc – DHCP client
- dhcps – DHCP server
- softap – Access point implemented in software
- wps – WiFi Protected Setup
- sntp – Simple Network Time Protocol
- mdns – Multicast Domain Name System
- uart – Universal asynchronous receiver/transmitter
- pwm – Pulse width modulation

FreeRTOS API reference

See also:

- Using FreeRTOS

eTaskGetState

Retrieve the state of a task.

```
eTaskState eTaskGetState(TaskHandle_t xTask)
```

pcTaskGetName

Get the name of the task.

```
char *pcTaskGetTaskName(TaskHandle_t xTaskToQuery)
```

xEventGroupClear

Clear one or more bits in an event group.

```
EventBits_t xEventGroupClearBits(EventGroupHandle_t eventGroup,  
                                const EventBits_t bitsToClear)
```

- `eventGroup` – The event group that contains the bits to be cleared.
- `bitsToClear` – The set of bits to be cleared within the event group.

Includes:

- `<freertos/event_groups.h>`

xEventGroupCreate

Create a new FreeRTOS event group.

```
EventGroupHandle_t xEventGroupCreate()
```

Includes:

- `<freertos/event_groups.h>`

xEventGroupSetBits

Set one or more bits in an event group.

```
EventBits_t xEventGroupSetBits(EventGroupHandle_t eventGroup,  
                               const EventBits_t bitsToSet)
```

- `eventGroup` – The event group that contains the bits to be set.
- `bitsToSet` – The set of bits to be set within the event group.

Includes:

- `<freertos/event_groups.h>`

xEventGroupWaitBits

Block waiting for one or more "bits" to become set.

```
EventBits_t xEventGroupWaitBits(  
    const EventGroupHandle_t eventGroup,  
    const EventBits_t bitsToWaitFor,  
    const BaseType_t clearOnExit,  
    const BaseType_t waitForAllBits,  
    TickType_t ticksToWait)
```

Calling this function can cause the caller to block until bits are set or a timeout is met.
Bits can be set through `xEventGroupSetBits()`.

- `eventGroup` – The event group that references the bits to be watched.
- `bitsToWaitFor` – The set of bits within the event group that we are waiting upon.
- `clearOnExit` – Should the bits that we are waiting on be cleared automatically when set and this method returns?
- `waitForAllBits` – Should we unblock on the first bit that is set that we are watching for or alternatively should we wait on all the bits being set?
- `ticksToWait` – How many ticks should we wait for before returning? This provides a timeout (if needed). The RTOS variable "`portMAX_DELAY`" can be used to specify that we wish to wait indefinitely.

Includes:

- `<freertos/event_groups.h>`

xTaskCreate

Create a new instance of a task.

```
BaseType_t xTaskCreate(
    pdTASK_CODE pvTaskCode,
    const signed portCHAR *pcName,
    unsigned portSHORT usStackDepth,
    void *pvParameters,
    unsigned portBASE_TYPE uxPriority,
    xTaskHandle *pxCreatedTask)
```

- `pvTaskCode` – Pointer to the task function. In C programming, we can simply supply the name of a function or, as has been seen in some samples, the address of the name of the function. Apparently these equate to items that are close enough to be used interchangablly.
- `pcName` – Debugging name of the task.
- `usStackDepth` – Size of the stack for the task.
- `pvParameters` – Parameters for the task instance. This may be NULL.
- `uxPriority` – Priority of the task instance.
- `xTaskHandle` – Reference to the newly created task instance. This may be passed in as NULL if no task handle is required to be returned.

When a created task is invoked and then decides to end via a return, it is essential that the task call `vTaskDelete(NULL)` before it completes. Calling this is an indication to

FreeRTOS that the task is finished and need no longer be considered for context switching.

See also:

- [vTaskDelete](#)
- [xTaskCreate](#)

vTaskDelay

Delay a task for a specified number of ticks.

```
void vTaskDelay(const TickType_t xTicksToDelay)
```

The constant called `portTICK_PERIOD_MS` provides the number of ticks in a millisecond. If we wished to delay for 1 second, we could then supply `1000 / portTICK_PERIOD_MS`.

See also:

- [vTaskDelay](#)

vTaskDelayUntil

Delay a task until a specified absolute time.

```
void vTaskDelayUntil(const TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement)
```

This function blocks a task until some absolute time in the future.

- `pxPreviousWakeTime` – The base time which the increment will be relative from.
- `xTimeIncrement` – The time in ticks which, when added to the `pxPreviousWakeTime`, will be the time that the task is ready to run again.

See also:

- [vTaskDelayUntil](#)

vTaskDelete

Delete an instance of a task.

```
void vTaskDelete(TaskHandle_t pxTask)
```

This function will delete an instance of a task. If the `pxTask` handle is `NULL` then the current task will be deleted.

See also:

- [xTaskCreate](#)
- [vTaskDelete](#)

xTaskGetCurrentTaskHandle

Get the current task handle.

xTaskGetTickCount

Get the current tick count.

```
portTickType xTaskGetTickCount()
```

Return the number of ticks that have occurred since the task scheduler was started.

vEventGroupDelete

Delete an event group.

```
void vEventGroupDelete(EventGroupHandle_t eventGroup)
```

Includes:

- `<freertos/event_groups.h>`

vTaskList

```
void vTaskList(char *pcWriteBuffer)
```

NOT AVAILABLE

vTaskPrioritySet

```
void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority)
```

See also:

- [vTaskPrioritySet](#)

vTaskResume

```
void vTaskResume(TaskHandle_t pxTaskToResume)
```

xTaskResumeAll

See also:

- [vTaskResumeAll](#)

vTaskResumeFromISR

```
void xTaskResumeFromISR(TaskHandle_t pxTaskToResume)
```

vTaskSuspend

```
void vTaskSuspend(TaskHandle_t pxTaskToSuspend)
```

vTaskSuspendAll

See also:

- [vTaskSuspendAll](#)

xQueueCreate

Create a queue for holding items.

```
xQueueHandle xQueueCreate(
    unsigned portBASE_TYPE uxQueueLength,
    unsigned portBASE_TYPE uxItemSize)
```

- `uxQueueLength` – The maximum number of items that the queue can contain.
- `uxItemSize` – The size in bytes reserved for each element in the queue.

See also:

- [xQueueCreate](#)

vQueueDelete

```
void vQueueDelete(xQueueHandle xQueue)
```

xQueuePeek

```
portBASE_TYPE xQueuePeek(
    xQueueHandle xQueue,
    void *pvBuffer,
    portTickType xTicksToWait)
```

xQueueReceive

```
portBASE_TYPE xQueueReceive(
    xQueueHandle xQueue,
    void *pvBuffer,
    portTickType xTicksToWait)
```

xQueueSend

```
portBASE_TYPE xQueueSend(
    xQueueHandle xQueue,
    const void * pvItemToQueue,
    portTickType xTicksToWait)
```

xQueueSendToBack

```
portBASE_TYPE xQueueSendToBack(  
    xQueueHandle xQueue,  
    const void * pvItemToQueue,  
    portTickType xTicksToWait);
```

xQueueSendToFront

```
portBASE_TYPE xQueueSendToFront(  
    xQueueHandle xQueue,  
    const void * pvItemToQueue,  
    portTickType xTicksToWait)
```

vSemaphoreCreateBinary

xSemaphoreCreateCounting

vSemaphoreGive

xSemaphoreGiveFromISR

vSemaphoreTake

pvPortMalloc

pvPortFree

List Processing

vListInitialise

Initialize a list.

```
void vListInitialise(xList * const pxList)
```

The `pxList` is a list that should be initialized.

vListInitialisItem

Initialize an item for insertion into a list.

```
void vListInitialiseItem(xListItem * const pxItem)
```

Initialize an item that can be added to a list.

vListInsert

Insert an item into a list.

```
void vListInsert(xList * const pxList, xListItem * const pxNewListItem)
```

vListInsertEnd

Insert an item at the end of a list

```
void vListInsertEnd( xList * const pxList, xListItem * const pxNewListItem)
```

Iwip Reference

Sockets

- accept
- bind
- shutdown
- closesocket
- connect
- getsocketname
- getpeername
- setsockopt
- getsockopt
- listen
- recv
- recvfrom
- send
- sendto
- socket
- select

- ioctlsocket
- read
- write
- close
- fcntl

Timer functions

Timer functions allow us to register functions that will be executed at a time in the future or periodically after time passes. We also group functions that manipulate or retrieve time values in this set.

`os_delay_us`

Delay for microseconds.

```
void os_delay_us(uint16 us)
```

Delay for a maximum interval of 65535 microseconds.

Includes:

- `osapi.h`

See also:

- Timers and time
- `system_set_os_print`

`os_timer_arm`

Enable a millisecond granularity timer.

```
void os_timer_arm(
    os_timer_t *pTimer,
    uint32_t milliseconds,
    bool repeat)
```

Arm a timer such that it starts ticking and fires when the clock reaches zero.

The `pTimer` parameter is a pointer to a timer control structure.

The `milliseconds` parameter is the duration of the timer measured in milliseconds.

The `repeat` parameter is whether or not the timer will restart once it has reached zero.

Includes:

- `osapi.h`

See also:

- Timers and time

- `os_timer_disarm`
- `os_timer_setfn`

`os_timer_disarm`

Disarm/Cancel a previously armed timer.

```
void os_timer_disarm(os_timer_t *pTimer)
```

Stop a previously started timer which was started by a call to `os_timer_arm()`.

The `pTimer` parameter is a pointer to a timer control structure.

Includes:

- `osapi.h`

See also:

- Timers and time
- `os_timer_arm`
- `os_timer_setfn`

`os_timer_setfn`

Define a function to be called when the timer fires

```
void os_timer_setfn(
    os_timer_t *pTimer,
    os_timer_func_t *pFunction,
    void *pArg)
```

Define the callback function that will be called when the timer reaches zero.

The `pTimer` parameters is a pointer to the timer control structure.

The `pFunction` parameters is a pointer to the callback function.

The `pArg` parameter is a value that will be passed into the called back function.

The callback function should have the signature:

```
void (*functionName)(void *pArg)
```

The `pArg` parameter is the value registered with the callback function.

Includes:

- `osapi.h`

See also:

- Timers and time
- `os_timer_arm`
- `os_timer_disarm`

system_timer_reinit

Used to set a micro second timer

os_timer_arm_us

Enable a micro second timer

hw_timer_init

Initialize a hardware timer

hw_timer_arm

Set the trigger delay

hw_timer_set_func

Set the timer callback

System Functions

system_adc_read

Read the A/D converter value.

```
uint16 system_adc_read()
```

Read the value of the analog to digital converter. The granularity is 1024 discrete steps.

See also:

- Analog to digital conversion

system_deep_sleep_set_option

Define what the chip will do when it next wakes up.

```
bool system_deep_sleep_set_option(uint8 option)
```

system_get_boot_mode

Get the current boot mode

```
uint8 system_get_boot_mode()
```

The return value indicates the current boot mode and will be one of:

- `SYS_BOOT_ENHANCE_MODE` – 0
- `SYS_BOOT_NORMAL_MODE` – 1

On my devices, the value being returned is "0".

`system_get_boot_version`

The version of the boot loader.

```
uint8 system_get_boot_version()
```

The current value returned through testing of my devices is "5".

`system_get_chip_id`

Get the id of the chip

```
long system_get_chip_id()
```

For example: 0xf94322

`system_get_cpu_freq`

Get the current CPU frequency

```
int system_get_cpu_freq()
```

Returns the CPU frequency in MHz. The value will either be 80 or 160.

`system_get_flash_size_map`

Get current flash size and map

```
enum flash_size_map system_get_flash_size_map()
```

The value returned is an enum which has the following definitions:

- `FLASH_SIZE_4M_MAP_256_256`
- `FLASH_SIZE_2M`
- `FLASH_SIZE_8M_MAP_512_512`
- `FLASH_SIZE_16M_MAP_512_512`
- `FLASH_SIZE_32M_MAP_512_512`
- `FLASH_SIZE_16M_MAP_1024_1024`
- `FLASH_SIZE_32M_MAP_1024_1024`

See also:

- Loading a program into the ESP8266

system_get_rst_info

Information about the current startup.

```
struct rst_info* system_get_rst_info()
```

Retrieve information about the current device startup.

Includes:

- `user_interface.h`

See also:

- Exception handling
- `struct rst_info`

system_get_userbin_addr

Get the address of user bin

```
uint32 system_get_userbin_addr()
```

The current value returned on my devices is `0x0`.

system_get_vdd33

Measure voltage

Unknown ... but related to analog to digital conversion.

See also:

- Analog to digital conversion
- Error: Reference source not found

system_init_done_cb

Register a function to be called when system initialization is complete

```
void system_init_done_cb(init_done_cb_t callbackFunction)
```

This function is designed only be called in `user_init()`. It will register a function to be called one time after the ESP8266 has been initialized. The `init_done_cb_t` defines a function:

```
void (*functionName) (void)
```

Includes:

- `user_interface.h`

See also:

- Custom programs

`system_os_post`

Post a message to a task.

```
bool system_os_post(uint8 priority,  
                    os_signal_t signal,  
                    os_param_t parameter)
```

Post a message to a task. The task will not run immediately but will run as soon as it can.

The `priority` field is the priority of the task request. Three values are defined –

`USER_TASK_PRIO_0`, `USER_TASK_PRIO_1` and `USER_TASK_PRIO_2`.

The `signal` parameter is used by the task handler to determine who should process the signal. It is actually a `uint32_t`.

The `parameter` parameter is used to pass in optional data to the handler.

The return is `true` on success and `false` on failure.

Includes:

- `user_interface.h`

See also:

- ESP8266 Task handling

`system_os_task`

Setup a task for execution at a later time.

```
bool system_os_task(os_task_t task,  
                    uint8 priority,  
                    os_event_t *queue,  
                    uint queueLength)
```

The "os_task_t" is a pointer to a task handler function which has the signature:

```
void (*functionName)(os_event_t *event)
```

This function is defined to be a task handler that will receive all the different post notifications of the same priority level.

The `os_event_t` is a structure which contains:

- `os_signal_t` signal

- `os_param_t param`

Both of these are unsigned 32bit integers.

The `priority` field is the priority of the task request. Three values are defined:

- `USER_TASK_PRIO_0`
- `USER_TASK_PRIO_1`
- `USER_TASK_PRIO_2`

The return is `true` on success and `false` on failure.

Includes:

- `user_interface.h`

See also:

- `ESP8266 Task handling`

`system_phys_set_rfoption`

Enable the RF after waking up from a sleep (or not)

`system_phys_set_max_tpw`

Set the maximum transmission power

`system_phys_set_tpw_via_vdd33`

Set the transmission power as a function of voltage

`system_print_meminfo`

Print memory information

```
void system_print_meminfo()
```

Memory information for diagnostics is written to the output stream which is commonly `UART1`. The format of the data looks as follows:

```
data    : 0x3ffe8000 ~ 0x3ffe853c, len: 1340
rodata: 0x3ffe8540 ~ 0x3ffe8af0, len: 1456
bss    : 0x3ffe8af0 ~ 0x3fff1c18, len: 37160
heap   : 0x3fff1c18 ~ 0x3fffc000, len: 41960
```

The `.data` section is where global and static local initialized variables are kept.

The `.rodata` section is where read-only global and static data is kept.

The `.bss` is where un-initialized global and local static data is kept.

The `.heap` is where the heap of the program can be found.

See also:

- [Wikipedia – .bss](#)
- [Wikipedia – Data segment](#)

`system_restart_enhance`

Restarts the system in enhanced boot mode

`system_rtc_clock_cali_proc`

Clock calibration.

```
uint32 system_rtc_clock_cali_proc(void)
```

Retrieve the real time clock calibration. This is the wall clock duration of a clock cycle measured in micro seconds. The 16 bit number returned has bits 11-0 representing the value after the decimal point. We can multiply the value returned here against the number of cycles since a previous restart and determine an elapsed wall clock value.

`system_set_os_print`

Turn on or off logging.

```
void system_set_os_print(unint8 onOff)
```

A value of 0 switches it off while a value of 1 switches it on. It was initially thought that this controlled OS level logging however it seems to control **all** logging via `os_printf()`.

Includes:

- `user_interface.h`

See also:

- `os_printf`
- `os_install_putc1`
- Logging to UART1

`system_show_malloc`

Debug potential memory leak issues.

```
void system_show_malloc()
```

This API should also be enabled by explicitly defining `MEMLEAK_DEBUG`.

The documentation on this function in the SDK programming guide provides a number of warnings and caveats that are not yet fully understood so use with caution.

•

system_RTC_clock_cali_proc

Clock calibration.

```
uint32 system_RTC_clock_cali_proc(void)
```

Retrieve the real time clock calibration. This is the wall clock duration of a clock cycle measured in micro seconds. The 16 bit number returned has bits 11-0 representing the value after the decimal point. We can multiply the value returned here against the number of cycles since a previous restart and determine an elapsed wall clock value.

system_uart_swap

Swap serial UARTs.

When an ESP8266 starts up, it uses certain pins for UART0 control. Specifically, it needs pins for the functions TX, RX, CTS and RTS. By calling this function, the physical pins used for UART0 are switched around.

Function	Default	Swapped
U0TXD	U0TXD	MTDO
U0RXD	U0RXD	MTCK
U0CTS	MTCK	U0RXD
U0RTS	MTDO	U0TXD

system_soft_wdt_feed

Feed the software watchdog.

```
void system_soft_wdt_feed()
```

Feed the software watchdog. The function is only of value when the software watchdog is enabled. If we need to perform looping within our code, we need to call this function periodically so that we don't starve the WiFi runtime. The motif here is starve and food and hence the notion of a watchdog timer that checks that we don't spend too long away from WiFi ... so we must feed the dog. Interesting metaphors.

However ... experiments are showing that it doesn't seem to actually DO anything. The mystery of its purpose continues. See: <http://bbs.espressif.com/viewtopic.php?f=7&t=1055>

system_soft_wdt_stop

Disable the software watchdog.

```
void system_soft_wdt_stop()
```

Stop the software watchdog. It is recommended not to stop this timer for too long (8 seconds or less) otherwise the hardware watchdog will force a reset.

See also:

- Watchdog timer

system_soft_wdt_restart

Restart the software watchdog.

```
void system_soft_wdt_restart()
```

Restart the software watchdog following a previous call to stop it.

See also:

- Watchdog timer

system_uart_de_swap

Go back to original UART.

system_update_cpu_freq

Set the CPU frequency

```
void system_update_cpu_freq(int freq)
```

Set the CPU frequency. Either 80 or 160.

os_memset

Set the values of memory

```
void os_memset(void *pBuffer, int value, size_t size)
```

Set the memory pointed to by `pBuffer` to the `value` for `size` bytes.

Includes:

- `osapi.h`

See also:

- Working with memory
- `os_memcpy`

os_memcmp

Compare two regions of memory.

```
int os_memcmp(uint8 *ptr1, uint8 *ptr2, int size)
```

Compare two regions of memory. The return is 0 if they are equal.

Includes:

- `osapi.h`

os_memcpy

Copy the values of memory.

```
void os_memcpy(void *destination, void *source, size_t size)
```

Copy the memory from the buffer pointed to by `source` to the buffer pointed to by `destination` for the number of bytes specified by `size`.

Includes:

- `osapi.h`

See also:

- Working with memory
- `os_memset`

os_malloc

Allocate storage from the heap.

```
void *malloc(size_t size)
```

Allocate `size` bytes from the heap and return a pointer to the allocated storage.

Includes:

- `mem.h`

See also:

- Working with memory
- `os_zalloc`
- `os_free`

os_calloc

Allocate storage for a set of elements.

```
void *calloc(size_t num, size_t size)
```

Here we allocate `num` instances of `size` sized objects in contiguous memory.

Includes:

- `mem.h`

os_realloc

Reallocate a previously obtained chunk of memory with a new size.

```
void *os_realloc(void *buf, size_t newSize)
```

Includes:

- mem.h

os_zalloc

Allocate storage from the heap and zero its values.

```
void *os_zalloc(size_t size)
```

Allocate `size` bytes from the heap and return a pointer to the allocated storage. Before returning, the storage area is zeroed.

Includes:

- mem.h

See also:

- Working with memory
- os_malloc
- os_free

os_free

Release previously allocated storage back to the heap.

```
void os_free(void *pBuffer)
```

Release the storage previously allocated by `os_malloc()` or `os_zalloc()` back to the heap.

Includes:

- mem.h

See also:

- Working with memory
- os_malloc
- os_zalloc

os_bzero

Set the values of memory to zero.

```
void os_bzero(void *pBuffer, size_t size)
```

Sets the data pointer to by `pBuffer` to zero for `size` bytes.

Includes:

- osapi.h

See also:

- Working with memory

os_delay_us

Delay for microseconds.

```
void os_delay_us(uint16 us)
```

Delay for a maximum interval of 65535 microseconds.

Includes:

- osapi.h

See also:

- Timers and time
- system_set_os_print

os_printf

Print a string to UART.

```
void os_printf(char *format, ...)
```

The format flags that are known to work include:

- %d – display an integer
- %ld – display a long integer
- %lu – display a long unsigned integer
- %x – display as a hex number
- %s – display as a string
- "\n" – display a newline (includes a prefixed carriage return)

Note that there is **no** %f to print a float or double.

The output text is sent to the function registered with `os_install_putc1()`. By default, this is UART0 but can be changed to UART1 by setting the `uart1_write_char()` function.

Includes:

- osapi.h

See also:

- Debugging
- os_install_putc1
- system_set_os_print

os_install_putc1

Register a function print a character

```
void os_install_putc1(void (*pFunc)(char c));
```

Register a function that will be called by output functions such as `os_printf()` that will log output. For example, this can be used to write to the serial ports. When a call is made to the supplied `uart_init()` method, the writing function is set to write to UART1.

Includes:

- `osapi.h`

See also:

- `os_printf`
- `system_set_os_print`

os_random

```
unsigned long os_random()
```

Includes:

- `osapi.h`

os_get_random

```
int os_get_random(unsigned char *buf, size_t len)
```

Includes:

- `osapi.h`

os_strlen

Get the length of a string.

```
int os_strlen(char *string)
```

Return the length of the null terminated string.

Includes:

- `osapi.h`

os_strcat

Concatenate two strings together.

```
char *os_strcat(char *str1, char *str2)
```

Concatenate the null terminated sting pointed to by `str1` with the string pointed to by `str2` and store the result at `str1`.

Includes:

- `osapi.h`

os_strchr

Includes:

- osapi.h

os_strcmp

Compare two strings.

```
int os_strcmp(char *str1, char *str2)
```

Compare the null terminated string pointed to by `str1` with the null terminated string pointed to by `str2`. If `str1 < str2` then the return is `< 0`. If `str1 > str2` then the return is `> 0` otherwise they are equal and the return is `0`.

Includes:

- osapi.h

os_strcpy

Copy one string to another.

```
char *os_strcpy(char *dest, char *src)
```

Copy the null terminated string pointed to by `src` to the memory located at `dest`.

Includes:

- osapi.h

os_strncmp

Includes:

- osapi.h

os_strncpy

Copy one string to another but be sensitive to the amount of memory available in the target buffer.

```
char *os_strncpy(char *dest, char *source, size_t sizeOfDest)
```

Understand that the resulting string in `dest` may **not** be null terminated.

Includes:

- osapi.h

os_sprintf

```
sprintf(char * buffer, char *format, ...)
```

The format is not as rich as normal `sprintf()` in a C library. For example, no float or double support.

Includes:

- `osapi.h`

os_strstr

Includes:

- `osapi.h`

SPI Flash

The SPI Flash apis allow us to read, write and erase sectors contained within flash memory. Note that there is a specific document from Espressif that covers the SPI Flash functions exclusively.

spi_flash_get_id

Get the ID info of SPI flash

```
uint32 spi_flash_get_id(void)
```

This is a bit encoded value that represents information about the flash chip being used in conjunction with the ESP8266.

The esptool also includes a command (`flash_id`) that can be used to retrieve and display the flash information.

Includes:

- `spi_flash.h`

See also:

- `esptool.py`

spi_flash_erase_sector

Erase a flash sector. Each sector is 4k in size.

```
SpiFlashOpResult spi_flash_erase_sector(uint16 sec)
```

The `sec` parameter is the sector number (a sector is 4096 bytes in size).

Includes:

- [spi_flash.h](#)

See also:

[spi_flash_read](#)

Read data from flash

```
SpiFlashOpResult spi_flash_read(uint32 src_addr, uint32 des_addr, uint32 size)
```

The `src_addr` parameter is the address in flash that will be read. The `des_addr` is the address in memory which will be written. The `size` parameter is the size of data to be read.

Includes:

- [spi_flash.h](#)

See also:

[spi_flash_set_read_func](#)

```
void spi_flash_set_read_func(user_spi_flash_read read)
```

Includes:

- [spi_flash.h](#)

See also:

[system_param_save_with_protect](#)

Memory saving

```
bool system_param_save_with_protect(uint16 start_sec, void *param, uint16 len)
```

Includes:

- [spi_flash.h](#)

See also:

[spi_flash_write](#)

Write data to flash

```
SpiFlashOpResult spi_flash_write(uint32 destAddr, uint32 *srcAddr, uint32 size)
```

The `destAddr` is the address in flash which is to be written. The `srcAddr` is the source address in memory from where the new data is to be taken. The `size` parameter is the size of the data to be written.

Includes:

- `spi_flash.h`

See also:

`system_param_load`

Read data saved with flash protection

```
bool system_param_load(uint16 start_sec, uint16 offset, void *param, uint16 len)
```

Includes:

- `spi_flash.h`

See also:

WiFi - ESP8266

`wifi_fpm_close`

`wifi_fpm_do_sleep`

`wifi_fpm_do_wakeup`

`wifi_fpm_get_sleep_type`

`wifi_fpm_open`

`wifi_fpm_set_sleep_type`

`wifi_fpm_set_wakeup_cb`

`wifi_get_channel`

`wifi_get_ip_info`

Retrieve the current IP info about the station.

```
bool wifi_get_ip_info(
    uint8 if_index,
    struct ip_info *info)
```

The `if_index` parameter defines the interface to retrieve. Two values are defined:

- `STATION_IF` – 0 – The station interface
- `SOFTAP_IF` – 1 – The Soft Access Point interface

The `info` parameter is populated with details of the current ip address, netmask and gateway.

Includes:

- `user_interface.h`

See also:

- Current IP Address, netmask and gateway
- `struct ip_info`

wifi_get_macaddr

Get the MAC address.

```
bool wifi_get_macaddr(uint8 if_index, uint8 *macaddr)
```

A MAC address is 6 bytes.

Includes:

- `user_interface.h`

wifi_get_opmode

Get the operating mode of the WiFi

```
uint8 wifi_get_opmode()
```

Return the current operating mode of the device.

There are four values defined:

- `NULL_MODE` – Null mode. (0)
- `STATION_MODE` – Station mode. (1)
- `SOFTAP_MODE` – Soft Access Point (AP) mode. (2)
- `STATIONAP_MODE` – Station + Soft Access Point (AP) mode. (3)

Includes:

- `user_interface.h`

See also:

- Defining the operating mode
- `wifi_get_opmode_default`

wifi_get_opmode_default

Get the default operating mode

```
uint8 wifi_get_opmode_default()
```

Return the default operating mode of the device following startup.

There are three values defined:

- **STATION_MODE** – Station mode
- **SOFTAP_MODE** – Soft Access Point (AP) mode
- **STATIONAP_MODE** – Station + Soft Access Point (AP) mode

Includes:

- `user_interface.h`

See also:

- Defining the operating mode
- `wifi_get_opmode`
- `wifi_set_opmode`
- `wifi_set_opmode_current`

wifi_get_phy_mode

Get the physical level WiFi mode.

```
enum phy_mode wifi_get_phys_mode();
```

This is used to retrieve the IEEE 802.11 network type such a b/g/n.

Includes:

- `user_interface.h`

See also:

- `enum phy_mode`
-

wifi_get_sleep_type

Includes:

- `user_interface.h`

wifi_get_user_fixed_rate

```
int wifi_get_user_fixed_rate(uint8 *enable_mask, uint8 *rate)
```

wifi_get_user_limit_rate_mask

```
uint8 wifi_get_user_limit_rate_mask()
```

wifi_set_broadcast_if

```
bool wifi_set_broadcast_if(uint8 interface)
```

Includes:

- `user_interface.h`

See also:

- Broadcast with UDP

`wifi_get_broadcast_if`

`uint8 wifi_get_broadcast_if()`

Includes:

- `user_interface.h`

See also:

- Broadcast with UDP

`wifi_set_sleep_type`

Includes:

- `user_interface.h`

`wifi_promiscuous_enable`

`wifi_promiscuous_set_mac`

`wifi_register_rfid_locp_recv_cb`

`wifi_register_send_pkt_freedom_cb`

`wifi_register_user_ie_manufacturer_recv_cb`

`wifi_rfid_locp_recv_close`

`wifi_rfid_locp_recv_open`

`wifi_send_pkt_freedom`

`wifi_set_channel`

`wifi_set_event_handle_cb`

Define a callback function to sense WiFi events.

```
void wifi_set_event_handler_cb(wifi_event_handler_cb_t callbackFunction)
```

Registers a function to be called when an event is detected by the WiFi subsystem.

The signature of the registered callback function is:

```
void (*functionName) (System_Event_t *event)
```

Includes:

- `user_interface.h`

See also:

- Handling WiFi events
- `System_Event_t`

wifi_set_ip_info

Set the interface data for the device.

```
bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)
```

The `if_index` parameter defines the interface to retrieve. Two values are defined:

- `STATION_IF` – 0 – The station interface
- `SOFTAP_IF` – 1 – The Soft Access Point interface

The `info` parameter is a pointer to a `struct ip_info` that contains the values we wish to set.

Includes:

- `user_interface.h`

See also:

- Current IP Address, netmask and gateway
- `struct ip_info`

wifi_set_macaddr

Set the MAC address.

```
bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)
```

A MAC address is 6 bytes.

Includes:

- `user_interface.h`

wifi_set_opmode

Set the operating mode of the WiFi including saving to flash.

```
bool wifi_set_opmode(uint8 opmode)
```

There are three values defined:

- `STATION_MODE` – Station mode
- `SOFTAP_MODE` – Soft Access Point (AP) mode

- **STATIONAP_MODE** – Station + Soft Access Point (AP) mode

Includes:

- `user_interface.h`

See also:

- Defining the operating mode
- `wifi_get_opmode`
- `wifi_get_opmode_default`

wifi_set_opmode_current

Set the operating mode of the WiFi but don't save to flash.

```
bool wifi_set_opmode_current(uint8 opmode)
```

There are three values defined:

- **STATION_MODE** – Station mode
- **SOFTAP_MODE** – Soft Access Point (AP) mode
- **STATIONAP_MODE** – Station + Soft Access Point (AP) mode

Includes:

- `user_interface.h`

See also:

- Defining the operating mode
- `wifi_get_opmode`
- `wifi_get_opmode_default`

wifi_set_phy_mode

Set the physical level WiFi mode.

```
bool wifi_set_phy_mode(enum phy_mode mode)
```

This is used to set the IEEE 802.11 network type such a b/g/n.

Includes:

- `user_interface.h`

See also:

- `enum phy_mode`

wifi_set_promiscuous_rx_cb

`wifi_set_sleep_type`

`wifi_set_user_fixed_rate`

`int wifi_set_user_fixed_rate(uint8 enable_mask, uint8 rate)`

The enable mask can be one of:

- `FIXED_RATE_MASK_NONE`
- `FIXED_RATE_MASK_STA`
- `FIXED_RATE_MASK_AP`
- `FIXED_RATE_MASK_ALL`

The rate can be one of:

- `PHY_RATE_6`
- `PHY_RATE_9`
- `PHY_RATE_12`
- `PHY_RATE_18`
- `PHY_RATE_24`
- `PHY_RATE_36`
- `PHY_RATE_48`
- `PHY_RATE_54`

`wifi_set_user_ie`

`wifi_set_user_limit_rate_mask`

`bool wifi_set_user_limit_rate_mask(uint8 enable_mask)`

`wifi_set_user_rate_limit`

`bool wifi_set_user_rate_limit(uint8 mode, uint8 ifidx, uint8 max, uint8 min)`

`wifi_set_user_sup_rate`

`int wifi_set_user_sup_rate(uint8 min, uint8 max)`

- `RATE_11B5M`
- `RATE_11B11M`
- `RATE_11B1M`
- `RATE_11B2M`
- `RATE_11G6M`

- RATE_11G12M
- RATE_11G24M
- RATE_11G48M
- RATE_11G54M
- RATE_11G9M
- RATE_11G18M
- RATE_11G36M

wifi_status_led_install

Associate a GPIO pin with the WiFi status LED.

```
void wifi_status_led_install(
    uint8 gpio_id,
    uint32 mux_name,
    uint8 gpio_func)
```

When WiFi traffic flows, we may wish a status LED to flicker or blink indicating flowing traffic. This function allows us to specify a GPIO that should be pulsed to indicate WiFi traffic.

The `gpio_id` parameter is the numeric pin number.

The `mux_name` is the name of the multiplexer logical name.

The `gpio_func` is the function to be enabled for that multiplexer.

Includes:

- `user_interface.h`

See also:

- `wifi_status_led_uninstall`

wifi_status_led_uninstall

Disassociate a status LED from a GPIO pin.

```
void wifi_status_led_uninstall()
```

Disassociates a previous association setup with a call to `wifi_status_led_install()`.

Includes:

- `user_interface.h`

See also:

- `wifi_status_led_install`

```
wifi_unregister_rfid_locp_recv_cb  
wifi_unregister_send_pkt_freedom_cb  
wifi_unregister_user_ie_manufacturer_recv_cb
```

WiFi Station

The following APIs relate to the ESP* device acting as a station and connecting to an external access point.

wifi_station_ap_change

Change the connection to another access point

```
bool wifi_station_ap_change(uint newApId)
```

Includes:

- user_interface.h

wifi_station_ap_number_set

Number of stations that will be cached

```
bool wifi_station_ap_number_set(uint8 ap_number)
```

Includes:

- user_interface.h

wifi_station_connect

Connect the station to an access point.

```
bool wifi_station_connect()
```

If we are already connected to a different access point then we first need to disconnect from it using `wifi_station_disconnect()`. There is also an auto connect attribute which can be used to allow the device to attempt to connect to the last access point seen when it is powered on. This can be set with the `wifi_station_set_auto_connect()` function.

Includes:

- user_interface.h

See also:

- Connecting to an access point
-

wifi_station_dhcpc_start

Start the DHCP client.

```
bool wifi_station_dhcpc_start()
```

If DHCP is enabled, then the IP, netmask and gateway will be retrieved from the DHCP server while if disabled, we will be using static values.

Includes:

- user_interface.h

See also:

- Current IP Address, netmask and gateway
- wifi_station_dhcpc_stop

wifi_station_dhcpc_status

Get the DHCP client status

```
enum dhcp_status wifi_station_dhcpc_status()
```

One of:

- DHCP_STOPPED
- DHCP_STARTED

Includes:

- user_interface.h

wifi_station_dhcpc_stop

Stop the DHCP client

```
bool wifi_station_dhcpc_stop()
```

If DHCP is enabled, then the IP, netmask and gateway will be retrieved from the DHCP server while if disabled, we will be using static values.

Includes:

- user_interface.h

See also:

- Current IP Address, netmask and gateway

wifi_station_disconnect

Disconnect the station from an access point.

```
bool wifi_station_disconnect()
```

We should presume that we have previously connected via a `wifi_station_connect()`. We can determine our current connection status through `wifi_station_get_connect_status()`.

The return is `true` on success and `false` on an `error`.

Includes:

- `user_interface.h`

`wifi_station_get_ap_info`

Get the information of access points cached

```
uint8 wifi_station_get_ap_info(struct station_config configs[])
```

Includes:

- `user_interface.h`

`wifi_station_get_auto_connect`

Determine whether or not the ESP will auto connect to the last access point on boot.

```
uint8 wifi_station_get_auto_connect()
```

Determine whether or not the device will attempt to auto-connect to the last access point on restart. A value if 0 means it will not while non 0 means it will.

Includes:

- `user_interface.h`

See also:

- `wifi_station_set_auto_connect`

`wifi_station_get_config`

Get the current station configuration

```
bool wifi_station_get_config(struct station_config *config)
```

Retrieve the current station configuration settings.

Includes:

- `user_interface.h`

See also:

- Station configuration
- `station_config`
- `wifi_station_set_config`

- `wifi_station_set_config_current`

`wifi_station_get_config_default`

Get the default station configuration

Includes:

- `user_interface.h`

See also:

- Station configuration

`wifi_station_get_connect_status`

Get the connection status of the station.

```
uint8 wifi_station_get_connect_status()
```

The result is an enum with the following possible values:

Enum name	Value
STATION_IDLE	0
STATION_CONNECTING	1
STATION_WRONG_PASSWORD	2
STATION_NO_AP_FOUND	3
STATION_CONNECT_FAIL	4
STATION_GOT_IP	5
Not in station mode	255

Includes:

- `user_interface.h`

See also:

- `WiFi.printDiag`

`wifi_station_get_current_ap_id`

Get the current access point id

```
uint8 wifi_station_get_current_ap_id()
```

Includes:

- `user_interface.h`

wifi_station_get_hostname

Get the DHCP hostname of the WiFi device.

```
char* wifi_station_get_hostname()
```

Includes:

- user_interface.h

wifi_station_get_reconnect_policy

wifi_station_get_rssi

Get the received signal strength indication (rssi).

```
sint8 wifi_station_get_rssi()
```

Get the received signal strength indication (rssi).

Includes:

- user_interface.h

wifi_station_scan

Scan for available access points

```
bool wifi_station_scan(  
    struct scan_config *config,  
    scan_done_cb_t callbackFunction)
```

We can scan the WiFi frequencies looking for access points. We must be in station mode in order to execute the command. When the function is executed, we provide a callback function that will be asynchronously invoked at some time in the future with the results.

The `scan_config` structure contains:

- `uint8 *ssid`
- `uint8 *bssid`
- `uint8 channel`
- `uint8 show_hidden`

If we supply this structure, then only access points that match are returned.

The `scan_config` parameter can be `NULL` in which case no filtering will be performed and all access points will be returned.

The `scan_done_cb_t` is a function with the following structure:

```
void (*functionName)(void *arg, STATUS status)
```

The `arg` parameter is a pointer to a `struct bss_info`.

It is important to note that the **first** entry in the chain must be skipped over as it is the head of the list.

To get the next entry, we can use `STAILQ_NEXT(pBssInfoVar, next)`.

The `AUTH_MODE` is an enum

Enum name	Value
AUTH_OPEN	0
AUTH_WEP	1
AUTH_WPA_PSK	2
AUTH_WPA2_PSK	3
AUTH_WPA_WPA2_PSK	4

`STATUS` is an enum containing:

Enum name	Value
OK	0
FAIL	1
PENDING	2
BUSY	3
CANCEL	4

On success, the function returns true and false on a failure.

The name of this function is peculiar. Given that it appears to locate access points and not stations, I believe a more appropriate name would have been

```
wifi_access_point_scan()
```

Includes:

- `user_interface.h`

See also:

- Scanning for access points
- `struct bss_info`
- `STATUS`

wifi_station_set_auto_connect

Set whether or not the ESP will auto connect to the last access point on boot.

```
bool wifi_station_set_auto_connect(uint8 setValue)
```

Set whether or not the device will attempt to auto-connect to the last access point on restart. A value of 0 means it will not while a non 0 value means it will. If called in `user_init()`, the setting will be effective immediately. If called elsewhere, the setting will take effect on next restart.

Includes:

- `user_interface.h`

See also:

- `wifi_station_get_auto_connect`

wifi_station_set_cert_key

Set certificate and private key for connecting to WPA2-Enterprise access point.

```
bool wifi_station_set_cert_key(  
    uint8 *client_cert, int client_cert_len,  
    uint8 *private_key, int private_key_len,  
    uint8 *private_key_passwd, int private_key_passwd_len)
```

wifi_station_clear_cert_key

Release resources and clear status after connecting to a WPA2-Enterprise access point.

```
void wifi_station_clear_cert_key(void)
```

wifi_station_set_config

Set the configuration of the station.

```
bool wifi_station_set_config(struct station_config *config)
```

This function can only be called when the device mode includes Station support. Specifically, the details of which access point to interact with are supplied here. The details are persisted across a restart of the device.

A return value of true indicates success and a value of false indicates failure.

Includes:

- `user_interface.h`

See also:

- Station configuration
- `station_config`

- `wifi_station_get_config`
- `wifi_station_get_config_default`

`wifi_station_set_config_current`

Set the configuration of the station but don't save to flash.

```
bool wifi_station_set_config_current(struct station_config *config)
```

This function can only be called when the device mode includes Station support. Specifically, the details of which access point to interact with are supplied here. The details are not persisted across a restart of the device.

A return value of true indicates success and a value of false indicates failure.

Includes:

- `user_interface.h`

See also:

- Station configuration
- `station_config`
- `wifi_station_get_config`
- `wifi_station_get_config_default`

`wifi_station_set_reconnect_policy`

What should happen when the ESP gets disconnected from the AP

```
bool wifi_station_set_reconnect_policy(bool set)
```

Includes:

- `user_interface.h`

`wifi_station_set_hostname`

Set the DHCP hostname of the WiFi device.

```
bool wifi_station_set_hostname(char *name)
```

Includes:

- `user_interface.h`
-

WiFi SoftAP

The following APIs relate to the ESP* device acting as an access point to external stations.

`wifi_softap_dhcps_start`

Start the DHCP server service.

```
bool wifi_softap_dhcps_start()
```

Start the DHCP server service inside the device.

Includes:

- `user_interface.h`

See also:

- The DHCP server
- `wifi_softap_dhcps_stop`
- `wifi_softap_dhcps_offer_option`

wifi_softap_dhcps_status

Return the status of the DHCP server service.

```
enum dhcp_status wifi_softap_dhcps_status()
```

Retrieve the status of the DHCP server service. The returned value will be one of:

- `DHCP_STOPPED`
- `DHCP_STARTED`

Includes:

- `user_interface.h`

See also:

- The DHCP server
- `wifi_softap_dhcps_stop`
- `wifi_softap_dhcps_offer_option`

wifi_softap_dhcps_stop

Stop the DHCP server service.

```
bool wifi_softap_dhcps_stop()
```

Stop the DHCP server service inside the device.

Includes:

- `user_interface.h`

See also:

- The DHCP server
- `wifi_softap_dhcps_offer_option`

wifi_softap_free_station_info

Release the data associated with a `struct station_info`.

```
void wifi_softap_free_station_info()
```

Following a call to `wifi_softap_get_station_info()` we may have data returned to us. The data was allocated by the OS and we must return it with this function call. Note that this function does **not** take in the data that was returned.

Includes:

- `user_interface.h`

See also:

- Being an access point
-

`wifi_softap_get_config`

Retrieve the current softAP configuration details.

```
bool wifi_softap_get_config(struct softap_config *pConfig)
```

When called, the `struct softap_config` pointed to be `pConfig` will be filled in with the details of the current softAP configuration. The details returned are those actually in use and may differ from the ones saved for default.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- `user_interface.h`

See also:

- `struct softap_config`
- `wifi_softap_get_config_default`
- `wifi_softap_set_config_current`

`wifi_softap_get_config_default`

Retrieve the default softAP configuration details.

```
bool wifi_softap_get_config_default(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be filled in with the details of the default softAP configuration. The details returned are those used at boot and may be different from the ones currently in use.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- `user_interface.h`

See also:

- `struct softap_config`
- `wifi_softap_set_config_current`

wifi_softap_get_dhcps_lease

wifi_softap_get_dhcps_lease_time

Get the DHCP server lease time value.

```
uint32 wifi_softap_get_dhcps_lease_time()
```

Return the number of minutes that a server DHCP lease IP address will be held.

wifi_softap_get_station_info

Return the details of all connected stations.

```
struct station_info *wifi_softap_get_station_info()
```

The return data is a linked list of `struct station_info` data structures.

Includes:

- `user_interface.h`

See also:

- Being an access point
- `wifi_softap_get_station_num`

wifi_softap_get_station_num

Return the count of stations currently connected.

```
uint8 wifi_softap_get_station_num()
```

Returns the number of stations currently connected. The maximum number of connections on an ESP8266 is 4 but we can reduce this in the softAP configuration if needed.

Includes:

- `user_interface.h`

See also:

- Being an access point

wifi_softap_reset_dhcps_lease_time

Reset the DHCP server lease time to the default value.

```
bool wifi_softap_reset_dhcps_lease_time()
```

Reset the DHCP server lease time to the default value which is currently 120 minutes.

wifi_softap_set_config

Set the current and default softAP configuration.

```
bool wifi_softap_set_config(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be used as the details of the default and current softAP configuration.

A value of 1 will be returned on success and 0 otherwise.

Includes:

- `user_interface.h`

See also:

- `struct softap_config`
- `wifi_softap_get_config_default`
- `wifi_softap_set_config_current`

wifi_softap_set_config_current

Set the default softAP configuration.

```
bool wifi_softap_set_config_current(struct softap_config *config)
```

When called, the `struct softap_config` pointed to be `pConfig` will be used as the details of the current softAP configuration but will not be saved as default.

Includes:

- `user_interface.h`

See also:

- `struct softap_config`
- `wifi_softap_get_config_default`
- `wifi_softap_set_config`

wifi_softap_set_dhcps_lease

Define the IP address range that will be leased by this DHCP server.

```
bool wifi_softap_set_dhcps_lease(struct dhcps_lease *pLease)
```

The `pLease` parameter is a pointer to a `struct dhcps_lease` which contains an IP address range of IP addresses that will be leased by this DHCP server. The difference between the upper and lower bound of the IP addresses must be 100 or less. This function will not take effect until the DHCP server is stopped and restarted (assuming it is already running).

Includes:

- `user_interface.h`

See also:

- The DHCP server
- `wifi_softap_dhcps_stop`
- `wifi_softap_dhcps_offer_option`
- `struct dhcps_lease`

`wifi_softap_set_dhcps_lease_time`

Set the DHCP server lease time.

```
bool wifi_softap_set_dhcps_lease_time(uint32 minutes)
```

Set how long a DHCP IP address lease is good for. the default is 120 minutes. The parameter is the number of minutes that the lease should be held. It has an allowable range of 1-2880.

`wifi_softap_dhcps_offer_option`

Set DHCP server options.

```
bool wifi_softap_set_dhcps_offer_option(uint8 level, void *optarg)
```

Currently, the `level` parameter can only be `OFFER_ROUTER` with `optarg` being a bit mask with values:

- 0b0 – Disable router information.
- 0b1 – Enable router information.

Includes:

- `user_interface.h`

See also:

- `wifi_softap_dhcps_stop`

WiFi WPS

`wifi_wps_enable`

```
bool wifi_wps_enable(WPS_TYPE_t wps_type)
```

The type parameter can be one of the following:

- `WPS_TYPE_DISABLE` – Unsupported
- `WPS_TYPE_PBC` – Push Button Configuration – Supported
- `WPS_TYPE_PIN` – Unsupported
- `WPS_TYPE_DISPLAY` – Unsupported
- `WPS_TYPE_MAX` – Unsupported

See also:

- WiFi Protected Setup – WPS

wifi_wps_disable

```
bool wifi_wps_disable()
```

See also:

- WiFi Protected Setup – WPS

wifi_wps_start

```
bool wifi_wps_start()
```

See also:

- WiFi Protected Setup – WPS

wifi_set_wps_cb

```
bool wifi_set_wps_cb(wps_st_cb_t callback)
```

The signature of the callback function is:

```
void (*functionName)(int status)
```

The status parameter will be one of:

- WPS_CB_ST_SUCCESS
- WPS_CB_ST_FAILED
- WPS_CB_ST_TIMEOUT

See also:

- WiFi Protected Setup – WPS

Upgrade APIs

system_upgrade_flag_check

Retrieve the upgrade status flag.

```
uint8 system_upgrade_flag_check()
```

The returned value will be one of:

- UPGRADE_FLAG_IDLE
- UPGRADE_FLAG_START
- UPGRADE_FLAG_FINISH

system_upgrade_flag_set

Set the upgrade status flag.

```
void system_upgrade_flag_set(uint8 flag)
```

The flag can be one of:

- UPGRADE_FLAG_IDLE
- UPGRADE_FLAG_START
- UPGRADE_FLAG_FINISH

system_upgrade_reboot

Reboot the ESP8266 and run the new firmware.

```
void system_upgrade_reboot()
```

system_upgrade_start

Start downloading the new firmware from the server.

```
bool system_upgrade_start(struct upgrade_server_info *server)
```

The server parameter is a structure ...

system_upgrade_userbin_check

Determine which of the two possible firmware images can be upgraded.

```
uint8 system_upgrade_userbin_check()
```

The result will be either UPGRADE_FW_BIN1 or UPGRADE_FW_BIN2.

Sniffer APIs

wifi_promiscuous_enable

```
void wifi_promiscuous_enable(uint8 promiscuous)
```

wifi_promiscuous_set_mac

```
void wifi_promiscuous_set_mac(const uint8_t *address)
```

wifi_promiscuous_rx_cb

```
void wifi_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)
```

`wifi_get_channel`

`wifi_set_channel`

Smart config APIs

`smartconfig_start`

`bool smartconfig_start(sc_callback_t cb, uint8 log)`

`smartconfig_stop`

`bool smartconfig_stop(void)`

SNTP API

Handle Simple Network Time Protocol request.

`sntp_setserver`

Set the address of an SNTP server.

`void sntp_setserver(unsigned char index, ip_addr_t *addr)`

Set the address of one of the three possible SNTP servers to be used.

The `index` parameter must be either 0, 1 or 2 and specifies which of the SNTP server slots is to be set.

The `addr` parameter is the IP address of the SNTP server to be recorded.

Includes:

- `sntp.h`

See also:

- Working with SNTP

`sntp_getserver`

Retrieve the IP address of the SNTP server.

`ip_addr_t sntp_getserver(unsigned char index)`

Retrieve the IP address of a previously registered SNTP server.

The `index` parameter is the index of the SNTP server to be retrieved. It may be either 0, 1 or 2.

Includes:

- `sntp.h`

See also:

- Working with SNTP

`sntp_setservername`

Set the hostname of a target SNTP server.

```
void sntp_setservername(unsigned char index, char *server)
```

Specify an SNTP server by its hostname.

The `index` parameter is the index of an SNTP server to be set. It may be either 0, 1 or 2.

The `server` parameter is a NULL terminated string that names the host that is an SNTP server.

See also:

- Working with SNTP

`sntp_getservername`

Get the hostname of a target SNTP server.

```
char *sntp_getservername(unsigned char index)
```

Retrieve the hostname of a specific SNTP server that was previously registered.

The `index` parameter is the index of an SNTP server that was previously set. It may be either 0, 1 or 2.

The return from this function is a NULL terminated string.

Includes:

- `sntp.h`

See also:

- Working with SNTP

`sntp_init`

```
void sntp_init()
```

Initialize the SNTP functions.

Includes:

- `sntp.h`

See also:

- Working with SNTP

sntp_stop

```
void sntp_stop()
```

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_current_timestamp

Get the current timestamp as an unsigned 32 bit value representing the number of seconds since January 1st 1970 UTC.

```
uint32 sntp_get_current_timestamp()
```

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_real_time

```
char *sntp_get_real_time(long t)
```

????

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_set_timezone

Set the current local timezone.

```
bool sntp_set_timezone(sint8 timezone)
```

Invoking this function declares our local timezone as a signed offset in hours from UTC. It should only be called when the SNTP functions are not running as for example after a call to `sntp_stop()`.

The `timezone` parameter is a time zone in the range -11 to 13.

The return value is true on success and false otherwise.

Includes:

- sntp.h

See also:

- Working with SNTP

sntp_get_timezone

Get the current timezone.

```
sint8 sntp_get_timezone()
```

Retrieve the current value for the timezone as previously set with a call to `sntp_set_timezone()`.

Includes:

- sntp.h

See also:

- Working with SNTP

Generic TCP/UDP APIs

espconn_delete

Delete a control block structure.

```
sint8 espconn_delete(struct espconn *espconn)
```

The device maintains data and storage for each conversation (TCP and UDP). When these conversations are finished and we no longer are going to communicate with the partners, we can indicate that by calling this function which will release the internal storage. It is anticipated that failure to do this will result in memory leaks.

Return code of 0 on success otherwise the code indicates the error:

- ESPCONN_ARG – Illegal argument

This API undoes the effect of `espconn_create` or `espconn_accept`.

See also:

- UDP
- `espconn_create`
- `espconn_accept`

espconn_dns_setserver

Set the default DNS server.

```
void espconn_dns_setserver(char numdns, ip_addr_t *dnsservers)
```

The `numdns` is the number of DNS servers supplied which must be 1 or 2. No more than 2 DNS servers may be supplied. This function should not be called if DHCP is being used.

The `dnsservers` parameter is an array of 1 or 2 IP addresses.

See also:

- Name Service

espconn_gethostbyname

```
err_t espconn_gethostbyname(struct espconn *espconn,
    const char *hostname,
    ip_addr_t *addr,
    dns_found_callback found)
```

The parameters are:

- `espconn` – Care and understanding are needed when examining this parameter. Since it is a `struct espconn`, we would immediately think it has something to do with communications and is somehow used to control the `espconn_gethostbyname()` function. The answer is much much simpler. It is ignored. Yup ... the operation of `gethostbyname()` does **not** depend on this parameter at all. It however does show up in one more place. When the callback function is invoked as a result of having finished the `gethostbyname` ... the `arg` parameter to the callback is set to be the value of this `espconn` parameter. So in reality, it would have been perhaps better to define the data type of this first parameter to be a "`void *`" as basically that is how it used.
- `hostname` – The name of the host to lookup.
- `addr` – The address of a storage area where the IP address will be placed **only** if it has recently been queried before and is held in cache. The address found here is valid if `ESPCONN_OK` is returned.
- `found` – A callback function that will be invoked when the address has been resolved. The callback will be invoked only if `ESPCONN_INPROGRESS` is returned.

The `dns_found_callback` is a function with the following signature:

```
void (*functionName)(const char *name, ip_addr_t *ipAddr, void *arg)
```

where the `arg` parameter is a pointer to a `struct espconn`, the `name` is the hostname being sought and the `ipAddr` is the address of the IP address used to store the result.

When a host name cannot be found, the `ipAddr` is returned as NULL ... however, your DNS provider may choose to provide an IP address of a search engine and hence you'll get an address back ... but not the one to the host you expected!!

Return code of 0 on success otherwise the code indicates the error:

- `ESPConn_OK` – Succeeded.
- `ESPConn_INPROGRESS` – Indicates that we don't have a cache and we need to lookup.
- `ESPConn_ARG` – Illegal argument.

See also:

- Name Service
- lwIP – [DNS](#)

`espconn_port`

`uint32 espconn_port()`

`espconn_regist_sentcb`

Register a callback function that will be called when data has been sent.

```
sint8 espconn_regist_sentcb(  
    struct espconn *espconn,  
    espconn_sent_callback sent_cb)
```

The format of the callback function is:

```
void (*functionName)(void *arg)
```

The `arg` parameter is a pointer to a `struct espconn` that describes the connection.

See also:

- Sending and receiving TCP data
- `struct espconn`

`espconn_regist_recvcb`

Register a function to be called when data becomes available on the TCP connection or UDP datagram.

```
sint8 espconn_regist_recvcb(  
    struct espconn *espconn,  
    espconn_recv_callback recv_cb)
```

The format of the callback function is:

```
void (*functionName)(void *arg, char *pData, unsigned short len)
```

Where `args` is a pointer to a `struct espconn`, `pData` is a pointer to the data received and `len` is the length of the data received.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- Sending and receiving TCP data
- UDP
- `espconn_create`
- `struct espconn`

`espconn_send`

Send data through the connection to the partner.

```
sint8 espconn_send(  
    struct espconn *pEspconn,  
    uint8 *pBuffer,  
    uint16 length)
```

The `pEspconn` parameter identifies the connection through which to transmit the data.

The `pBuffer` parameter points to a data buffer to be transmitted.

The `length` parameter supplies the length of the data in bytes that is to be transmitted.

Note that the data need not be transmitted immediately. We can be notified when the data has been transmitted by a callback to the function registered with `espconn_regist_sentcb()`.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_MEM` (-1) – Out of memory
- `ESPCONN_ARG` (-12) – Illegal argument

See also:

- Sending and receiving TCP data
- UDP
- `espconn_regist_sentcb`

`espconn_sendto`

```
sin16 espconn_sendto(struct espconn *espconn, uint8 *psent, uint16 length)
```

`ipaddr_addr`

Build a TCP/IP address from a dotted decimal string representation.

```
uint32 ipaddr_addr(char *addressString)
```

Return an IP address (4 byte) value from a dotted decimal string representation supplied in the `addressString` parameter. Note that the `uint32` type is **not** assignable to the addresses in an `esp_tcp` or `esp_udp` structure. Instead we have to use a local variable and then copy the content. For example:

```
uint32 addr = ipaddr_addr(server);
memcpy(m_tcp.remote_ip, &addr, 4);
```

IP4_ADDR

Set the value of a variable to an IP address from its decimal representation.

```
IP4_ADDR(struct ip_addr * addr, a, b, c, d)
```

The `addr` parameter is a pointer to storage to hold an IP address. This may be an instance of `struct ip_addr`, a `uint32, uint8[4]`. It must be cast to a pointer to a `struct ip_addr` if not already of that type.

The parameters `a, b, c` and `d` are the parts of an IP address if it were written in dotted decimal notation.

Includes:

- `ip_addr.h`

See also:

- `struct ip_addr`

IP2STR

Generate four int values used in a `os_printf` statement

```
IP2STR(ip_addr_t *address)
```

This is a macro which takes a pointer to an IP address and returns four comma separated decimal values representing the 4 bytes of an IP address. This is commonly used in code such as:

```
os_printf("%d.%d.%d.%d\n", IP2STR(&addr));
```

TCP APIs

espconn_abort

Force the termination of a TCP/IP connection.

```
sint8 espconn_abort(struct espconn *espconn)
```

This API should **not** be called in any `espconn` callback functions.

A return code of 0 indicates success.

`espconn_accept`

Listen for an incoming TCP connection.

```
sint8 espconn_accept(struct espconn *espconn)
```

After calling this function, the ESP8266 starts listening for incoming connections. Any callback functions registered with `espconn_register_connectcb()` will be invoked when new connections arrive.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_MEM` – Out of memory
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_ARG` – Illegal argument

Note: After some thought, I think I really don't like the name of this. What this function does is cause the ESP8266 to start listening on a local port for new incoming requests. Essentially making the ESP8266 a server. When we study the sockets API, we find that the equivalent function call to achieve this task is called `listen`. So my suggested/recommended new name for this function would be `espconn_listen`.

So where then did the `accept` name come from? The answer is that in sockets API there is a partner function called `accept`. When executed against a socket that has previously had `listen` called against it, what it does is block until a partner actually attempts to connect. In the ESP8266, there is no equivalent. Instead, after `espconn_accept` is called, the ESP8266 immediately starts listening and when a partner connects, we wake up in the `connect` callback. So ... is `espconn_accept` a sockets `listen()` call or a sockets `accept()` call? My mind says that it is MUCH closer to a `listen()` call.

See also:

- `TCP`
- `espconn_register_connectcb`
- `espconn_delete`

`espconn_get_connection_info`

```
sint8 espconn_get_connection_info(
    struct espconn *espconn,
    remot_info **pcon_info,
    uint8 typeFlags)
```

The `espconn` is a pointer to the TCP control block.

The `pcon_info` parameter is the partner info.

The `typeFlags` defines what kind of partner we are getting information about:

- 0 – regular partner
- 1 – SSL partner

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

`espconn_connect`

Connect to a remote application using TCP.

```
sint8 espconn_connect(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN RTE` (-4) – Routing problem
- `ESPCONN MEM` (-1) – Out of memory
- `ESPCONN ISCONN` (-15) – Already connected
- `ESPCONN ARG` (-12) – Illegal argument

Realize that after making this call, we may still fail to connect. This is an asynchronous call which will be performed at a later time. If there is a failure at that point, we will find that the callback registered with `espconn_regist_reconcb()` will be invoked.

When the connection has been established, any registered callback made with `espconn_regist_connect()` will be invoked.

See also:

- TCP
- `espconn_disconnect`
- `espconn_regist_connectcb`
- `espconn_regist_disconcb`
- `espconn_regist_reconcb`

`espconn_disconnect`

Disconnect a TCP connection.

```
sint8 espconn_disconnect(struct espconn *espconn)
```

Disconnect a TCP connection that was previously formed with `espconn_connect()` or `espconn_accept()`. When the disconnect has succeeded, we will see a callback to the function registered with `espconn_regist_disconcb()`.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN ARG` – Illegal argument

See also:

- TCP
- espconn_accept
- espconn_connect
- espconn_regist_disconcb

espconn_regist_connectcb

Register a function that will be called when a TCP connection is formed.

```
sint8 espconn_regist_connectcb(
    struct espconn *espconn,
    espconn_connect_callback connect_cb)
```

Return code of 0 on success otherwise the code indicates the error:

- ESPCONN_ARG – Illegal argument

The callback function should have the following signature:

```
void (*functionName)(void *arg)
```

Where the arg parameter is a pointer to an struct espconn instance.

Question: Is this a NEW struct espconn or the original one?

See also:

- The espconn architecture
- espconn_accept
- espconn_connect

espconn_regist_disconcb

Register a function that will be called back after a TCP disconnection.

```
sint8 espconn_regist_disconcb(
    struct espconn *espconn,
    espconn_connect_callback discon_cb)
```

The signature of the disconnect callback function is the same as the connect callback:

```
void (*functionName)(void *arg)
```

where arg is a struct espconn pointer.

See also:

- TCP
- The espconn architecture
- espconn_accept
- espconn_connect
- espconn_disconnect

espconn_regist_reconcb

Register a function that will be called when an error is detected.

```
sint8 espconn_regist_reconcb(
    struct espconn *espconn,
    espconn_reconnect_callback recon_cb)
```

This callback is invoked when an error is detected. For example when attempting to connect to a partner which isn't listening. It is likely that the name of this function was simply badly chosen. See:

<http://bbs.espressif.com/viewtopic.php?f=66&t=1063>

The signature of the callback function is:

```
void (*functionName)(void *arg, sint8 err)
```

The `arg` parameter is a pointer to a `struct espconn`.

The `err` parameter is one of the following:

- `ESPCONN_TIMEOUT` (-3)
- `ESPCONN_ABRT` (-8)
- `ESPCONN_RST` (-9)
- `ESPCONN_CLSD` (-10)
- `ESPCONN_CONN` (-11) – Failed connecting to a partner
- `ESPCONN_HANDSHAKE` (-28)
- `ESPCONN_PROTO_MSG` ??

Question: What does it mean to the connection status if we receive an error indication? Should we then try and disconnect or are we already disconnected? See:

<http://www.esp8266.com/viewtopic.php?f=9&t=5864>

See also:

- The `espconn` architecture
- TCP
- `espconn_accept`
- `espconn_connect`
- `struct espconn`

`espconn_regist_write_finish`

Register a callback function to be invoked when data has been successfully transmitted to the partner.

```
sint8 espconn_regist_write_finish(struct espconn *espconn,
    espconn_connect_callback write_finish_cb);
```

The signature of the callback is:

```
void (*functionName)(void *arg)
```

The `arg` parameter is a pointer to a `struct espconn`.

See also:

- The `espconn` architecture
- `espconn_send`

`espconn_set_opt`

Define which options to turn on for a connection.

```
sint8 espconn_set_opt(  
    struct espconn *espconn,  
    uint8 opt)
```

This function should be called in an `espconn_connect_callback`. The `espconn` parameter is the control block for the connection that is to be modified.

The `opt` parameter is a bit encoding of flags that are to be set on. The `opt` parameter is an enum of type `espconn_option`:

Enum Name	Value
ESPCONN_REUSEADDR	0x01
ESPCONN_NODELAY	0x02
ESPCONN_COPY	0x04
ESPCONN_KEEPALIVE	0x08

Bits that are not set on are left unchanged from their current existing values.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- `espconn_clear_opt`
- `espconn_set_keepalive`
- `espconn_get_keepalive`

`espconn_clear_opt`

Define which options to turn off for a connection.

```
sint8 espconn_clear_opt(  
    struct espconn *espconn,  
    uint8 opt)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

The opt value is an enum of type `espconn_option`:

Enum Name	Value
<code>ESPCONN_REUSEADDR</code>	0x01
<code>ESPCONN_NODELAY</code>	0x02
<code>ESPCONN_COPY</code>	0x04
<code>ESPCONN_KEEPALIVE</code>	0x08

See also:

- TCP Error handling
- `espconn_set_opt`
- `espconn_set_keepalive`
- `espconn_get_keepalive`

`espconn_regist_time`

Define an idle connection timeout value.

```
sint8 espconn_regist_time(
    struct espconn *espconn,
    uint32 interval,
    uint8 typeFlag)
```

If a connection is idle for a period of time, the ESP8266 is configured to automatically close the connection. It appears that the default is 10 seconds.

The `espconn` parameter describes the connection that is to have its timeout changed.

The `interval` parameter defines the timeout interval in seconds. The maximum value is 7200 seconds (2 hours).

The `typeFlag` parameter can be 0 to indicate that all connections are to be changed or 1 to set just this connection.

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument

See also:

- TCP

`espconn_set_keepalive`

```
sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void *optArg)
```

espconn_get_keepalive

```
sint8 espconn_get_keepalive(struct espconn *espconn, uint8 level, void *optArg)  
???
```

espconn_secure_accept

Listen for an incoming SSL TCP connection

```
sint8 espconn_secure_accept(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- ESPCONN_MEM – Out of memory
- ESPCONN_ISCONN – Already connected
- ESPCONN_ARG – Illegal argument

espconn_secure_ca_disable

```
bool espconn_secure_ca_disable(uint8 level)
```

espconn_secure_ca_enable

```
bool espconn_secure_ca_enable(uint8 level, uint16 flash_sector)
```

espconn_secure_set_size

espconn_secure_get_size

espconn_secure_delete

Delete an SSL connection when running as an SSL based server.

```
sint8 espconn_secure_delete(struct espconn *espconn)
```

A return code of 0 indicates success.

espconn_secure_connect

Form an SSL connection to a partner.

```
sint8 espconn_secure_connect(struct espconn *espconn)
```

Form an SSL connection to a partner.

Return code of 0 on success otherwise the code indicates the error:

- ESPCONN_MEM – Out of memory

- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_ARG` – Illegal argument

`espconn_secure_send`

Send data through a secure connection.

```
sint8 espconn_secure_send(struct espconn *espconn, uint8 *pBuf, uint16 length)
```

Send data through a secure connection.

`espconn_secure_disconnect`

Secure TCP disconnection.

```
sint8 espconn_secure_disconnect(struct espconn *espconn)
```

Secure TCP disconnection.

Do not call this function from within an ESP callback function.

`espconn_tcp_get_max_con`

Return the maximum number of concurrent TCP connections.

```
uint8 espconn_tcp_get_max_con()
```

`espconn_tcp_set_max_con`

Set the maximum number of concurrent TCP connections

```
sint8 espconn_tcp_set_max_con(uint8 num)
```

`espconn_tcp_get_max_con_allow`

Get the maximum number of TCP clients allowed to connect inbound.

.

`espconn_tcp_set_max_con_allow`

Set the maximum number of TCP clients allowed to connect inbound.

`espconn_recv_hold`

Suspend receiving TCP data.

```
sint8 espconn_recv_hold(struct espconn *espconn)
```

Suspend receiving new data over TCP. To resume receiving data, one can use the `espconn_recv_unhold` function call.

- `espconn_recv_unhold`

espconn_recv_unhold

Unblock receiving TCP data.

```
sint8 espconn_recv_unhold(struct espconn *espconn)
```

Resume receiving new data over TCP. This method should be used in conjunction with `espconn_recv_hold` which suspends receipt of data.

See also:

- `espconn_recv_hold`

UDP APIs

espconn_create

Create a UDP control block in preparation for sending datagrams.

```
sint8 espconn_create(struct espconn *espconn)
```

Return code of 0 on success otherwise the code indicates the error:

- `ESPCONN_ARG` – Illegal argument
- `ESPCONN_ISCONN` – Already connected
- `ESPCONN_MEM` – Out of memory

See also:

- `UDP`
- `espconn_regist_sentcb`
- `espconn_regist_recvcb`
- `espconn_send`
- `espconn_delete`
- `espconn_connect`

espconn_igmp_join

Join a multicast group.

espconn_igmp_leave

Leave a multicast group.

ping APIs

ping_start

```
bool ping_start(struct ping_option *ping_opt)
```

Includes:

- ping.h

See also:

- Ping request
- struct ping_option

ping_regist_recv

```
bool ping_regist_recv(struct ping_option *ping_opt, ping_recv_function ping_recv)
```

Register a function that will be called when a ping is received. The signature of the function is:

```
void (*functionName)(void* pingOpt, void *pingResp)
```

The parameters passed in are pingOpt which is a pointer to the struct ping_option and pingResp which is a pointer to a struct ping_resp.

Includes:

- ping.h

See also:

- Ping request
- struct ping_option
- struct ping_resp

ping_regist_sent

```
bool ping_regist_sent(struct ping_option *ping_opt, ping_sent_function ping_sent)
```

Register a function that will be called when a ping is sent. The signature of the function is:

```
void (*functionName)(void* pingOpt, void *pingResp)
```

The parameters passed in are pingOpt which is a pointer to the struct ping_option and pingResp which is a pointer to a struct ping_resp.

Includes:

- ping.h

See also:

- Ping request
- struct ping_option

mDNS APIs

See also:

- Multicast Domain Name Systems

espconn_mdns_init

Initialize mDNS on the ESP8266.

```
void espconn_mdns_init(struct mdns_info *info)
```

The structure of type `struct mdns_info` contains vital initialization information and must be completed before calling this function.

See also:

- `struct mdns_info`

espconn_mdns_close

Close mDNS support.

```
void espconn_mdns_close()
```

Close mDNS support. This can be used following a call to `espconn_mdns_init()`.

See also:

- `espconn_mdns_init`

espconn_mdns_server_register

Register the mDNS server.

```
void espconn_mdns_server_register()
```

espconn_mdns_server_unregister

Unregister the mDNS server.

```
void espconn_mdns_server_unregister()
```

espconn_mdns_get_servername

Get the mDNS server name.

```
char *espconn_mdns_get_servername()
```

espconn_mdns_set_servername

Set the mDNS server name.

```
char *espconn_mdns_set_servername()
```

espconn_mdns_set_hostname

Set the mDNS hostname.

```
void espconn_mdns_set_hostname(char *name)
```

espconn_mdns_get_hostname

Get the mDNS hostname

```
char *espconn_mdns_get_hostname()
```

espconn_mdns_disable

Disable mDNS.

```
void espconn_mdns_disable()
```

See also:

- [espconn_mdns_enable](#)

espconn_mdns_enable

Enable mDNS

```
void espconn_mdns_enable()
```

See also:

- [espconn_mdns_disable](#)

GPIO - ESP32

The gpio functions in the ESP32 are provided through the ESP-IDF. One must include the "driver/gpio.h" header.

gpio_config

ESP32

```
esp_err_t gpio_config(gpio_config_t *pGPIOConfig)
```

The `gpio_config_t` data structure contains:

```
    uint64_t pin_bit_mask  
    gpio_mode_t mode  
    gpio_pullup_t pull_up_en  
    gpio_pulldown_t pull_down_en  
    gpio_int_type_t intr_type
```

The `pin_bit_mask` defines which pins we are configuring. Constants are defined to assist us here. For example, if we are configuring GPIO34 and GPIO16 we can set the `pin_bit_mask` to `GPIO_Pin_16 | GPIO_Pin_34` which is the boolean "or" of the two constant values.

The `mode` is used to set the mode of all of the pins we are configuring. The allowable values are:

- `GPIO_MODE_INPUT`
- `GPIO_MODE_OUTPUT`
- `GPIO_MODE_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT`

The `pull_up_en` enables an internal pull-up resistor. The allowable values are:

- `GPIO_PULLUP_ENABLE`
- `GPIO_PULLUP_DISABLE`

The `pull_down_en` enables an internal pull-down resistor. The allowable values are:

- `GPIO_PULLDOWN_ENABLE`
- `GPIO_PULLDOWN_DISABLE`

The `intr_type` configures how interrupts are handled for the pin. The allowable values are:

- `GPIO_INTR_DISABLE`
- `GPIO_INTR_POSEDGE`
- `GPIO_INTR_NEGEDGE`
- `GPIO_INTR_ANYEDGE`
- `GPIO_INTR_LOW_LEVEL`
- `GPIO_INTR_HIGH_LEVEL`

gpio_get_level

ESP32

Retrieve the signal level on the pin.

```
int gpio_get_level(gpio_num_t gpioNum)
```

Get the signal level on the specified pin. Either 0 or 1.

gpio_input_get

ESP32

Retrieve a bitmask of the values of the first 32 GPIOs (0-31).

```
uint32_t gpio_input_get()
```

gpio_input_get_high

ESP32

Retrieve a bitmask of the values of the last 10 GPIOs (32-39).

```
uint32_t gpio_input_get_high()
```

gpio_intr_enable

ESP32

Enable interrupts on the specified pin.

```
esp_err_t gpio_intr_enable(gpio_num_t gpioNum)
```

gpio_intr_disable

ESP32

Disable interrupts on the specified pin.

```
esp_err_t gpio_intr_disable(gpio_num_t gpioNum)
```

gpio_isr_register

ESP32

Register an interrupt handler.

```
esp_err_t gpio_isr_register(uint32_t gpioIntr, void (*fn)(void *), void *arg)
```

gpio_output_set

ESP32

Set GPIOs that need to be input, output, high or low on bulk

```
void gpio_output_set(  
    uint32_t setMask,  
    uint32_t clearMask,  
    uint32_t enableMask,  
    uint32_t disableMask)
```

Work with the first 32 GPIOs (0-31) specifying which ones are input, which ones are output, which ones should be left alone and which ones should be set high/low. This API lets us set a batch of GPIOs in one operation.

gpio_output_set_high

ESP32

Set GPIOs that need to be input, output, high or low on bulk

```
void gpio_output_set_high(  
    uint32_t setMask,  
    uint32_t clearMask,  
    uint32_t enableMask,  
    uint32_t disableMask)
```

Work with the last 10 GPIOs (32-39) specifying which ones are input, which ones are output, which ones should be left alone and which ones should be set high/low. This API lets us set a batch of GPIOs in one operation.

gpio_set_direction

ESP32

Set the direction of a pin.

```
esp_err_t gpio_set_direction(gpio_num_t gpioNum, gpio_mode_t mode)
```

The `mode` is used to set the mode of the pin we are configuring. The allowable values are:

- `GPIO_MODE_INPUT`
- `GPIO_MODE_OUTPUT`
- `GPIO_MODE_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT_OD`
- `GPIO_MODE_INPUT_OUTPUT`

gpio_set_intr_type

ESP32

Set the interrupt type of a pin.

```
esp_err_t gpio_set_intr_type(gpio_num_t gpioNum, gpio_int_type_t intrType)
```

The `intr_type` configures how interrupts are handled for the pin. The allowable values are:

- `GPIO_INTR_DISABLE`
- `GPIO_INTR_POSEDGE`
- `GPIO_INTR_NEGEDGE`
- `GPIO_INTR_ANYEDGE`
- `GPIO_INTR_LOW_LEVEL`
- `GPIO_INTR_HIGH_LEVEL`

gpio_set_level

ESP32

Set the level of a pin.

```
esp_err_t gpio_set_level(gpio_num_t gpioNum, uint32_t level)
```

Set the level of an output pin. The level should be either 0 or 1.

gpio_set_pull_mode

ESP32

Set the pullup/pulldown mode of the pin.

```
esp_err_t gpio_set_pull_mode(gpio_num_t gpioNum, gpio_pull_mode_t pull)
```

The allowable values for pull are:

- `GPIO_PULLUP_ONLY`
- `GPIO_PULLDOWN_ONLY`
- `GPIO_PULLUP_PULLDOWN`
- `GPIO_FLOATING`

GPIO - ESP8266

Pin names are:

- PERIPHS_IO_MUX_GPIO0_U
- PERIPHS_IO_MUX_GPIO2_U
- PERIPHS_IO_MUX_MTDI_U
- PERIPHS_IO_MUX_MTCK_U // GPIO 13
- PERIPHS_IO_MUX_MTMS_U // GPIO 14

Pin Name	Function 1	Function 2	Function 3	Function 4	Physical pin
MTDI_U	MTDI	I2SI_DATA	HSPIQ MISO	GPIO12	10
MTCK_U	MTCK	I2SI_BCK	HSPID MOSI	GPIO13	12
MTMS_U	MTMS	I2SI_WS	HSPICLK	GPIO14	9
MTDO_U	MTDO	I2SO_BCK	HSPICS	GPIO15	13
U0RXD_U	U0RXD	I2SO_DATA		GPIO3	25
U0TXD_U	U0TXD	SPICS1		GPIO1	26
SD_CLK_U	SD_CLK	SPICLK		GPIO6	21
SD_DATA0_U	SD_DATA0	SPIQ		GPIO7	22
SD_DATA1_U	SD_DATA1	SPID		GPIO8	23
SD_DATA2_U	SD_DATA2	SPIHD		GPIO9	18
SD_DATA3_U	SD_DATA3	SPIWP		GPIO10	19
SD_CMD_U	SD_CMD	SPICS0		GPIO11	20
GPIO0_U	GPIO0	SPICS2			15
GPIO2_U	GPIO2	I2SO_WS	U1TXD		14
GPIO4_U	GPIO4	CLK_XTAL			16
GPIO5_U	GPIO5	CLK_RTC			24

Pin functions are:

- FUNC_GPIO0
- FUNC_GPIO12
- FUNC_GPIO13
- FUNC_GPIO14
- FUNC_GPIO15
- FUNC_U0RTS
- FUNC_GPIO3

- FUNC_U0TXD
- FUNC_GPIO1
- FUNC_SDCLK
- FUNC_SPICLK
- FUNC_SDDATA0
- FUNC_SPIQ
- FUNC_U1TXD
- FUNC_SDDATA1
- FUNC_SPID
- FUNC_U1RXD
- FUNC_SDATA1_U1RXD
- FUNC_SDDATA2
- FUNC_SPIHD
- FUNC_GPIO9
- FUNC_SDDATA3
- FUNC_SPIWP
- FUNC_GPIO10
- FUNC_SDCMD
- FUNC_SPICS0
- FUNC_GPIO0
- FUNC_GPIO2
- FUNC_U1TXD_BK
- FUNC_U0TXD_BK
- FUNC_GPIO4
- FUNC_GPIO5
- LED_GPIO_FUNC

PIN_PULLUP_DIS

Disable pin pull-up

PIN_PULLUP_DIS(PIN_NAME)

See also:

- [GPIOs](#)

PIN_PULLUP_EN

Enable pin pull-up

PIN_PULLUP_EN(PIN_NAME)

See also:

- [GPIOs](#)

PIN_FUNC_SELECT

Set the function of a specific pin.

PIN_FUNC_SELECT(PIN_NAME, FUNC)

See also:

- [GPIOs](#)

GPIO_ID_PIN

Get the id of a logical pin.

GPIO_ID_PIN(pinNum)

Convert a logical pin number into the identity of a pin. This is an interesting function as `GPIO_ID_PIN(x)` is coded to equal "`x`". The question now becomes whether or not one still needs to code `GPIO_ID_PIN()` when accessing GPIO functions.

GPIO_OUTPUT_SET

Set the output value of a specific pin.

GPIO_OUTPUT_SET(GPIO_NUMBER, value)

This is a helper macro that invokes `gpio_output_set()`. Take care when passing in a value that is part of an expression such as `pData=='1'`. The value is evaluated a number of times so should not have side-effects. There is also a current bug related to operator precedence ... it is strongly recommended to place the value in extra parenthesis when coding. For example:

```
GPIO_OUTPUT_SET(GPIO_NUMBER, (pData=='1'))
```

Includes:

- `gpio.h`

See also:

- [GPIOs](#)

GPIO_DIS_OUTPUT

Set the pin to be input (disabled output).

`GPIO_DIS_OUTPUT(GPIO_NUMBER)`

This is a helper macro that invokes `gpio_output_set()`.

Includes:

- `gpio.h`

See also:

- [GPIOs](#)

GPIO_INPUT_GET

Read the value of the pin.

`GPIO_INPUT_GET(GPIO_NUMBER)`

This is a helper macro that invokes `gpio_input_get()`.

Includes:

- `gpio.h`

See also:

- `gpio_input_get`

gpio_output_set

Change the values of GPIO pins in one operation.

```
void gpio_output_set(  
    uint32 set_mask,  
    uint32 clear_mask,  
    uint32 enable_output,  
    uint32 enable_input)
```

The parameters are:

- `set_mask` – Bits with a "1" are set high, bits with a "0" are left unchanged.
- `clear_mask` – Bits with a "1" are set low, bits with a "0" are left unchanged
- `enable_output` – Bits with a "1" are set to output
- `enable_input` – Bits with a "1" are set to input

Includes:

- `gpio.h`

See also:

- `GPIOs`

`gpio_input_get`

Get the values of the GPIOs.

```
uint32 gpio_input_get()
```

Retrieve the values from the GPIOs and return a bitmask of their values.

Includes:

- `gpio.h`

See also:

- `GPIOs`

`gpio_intr_handler_register`

Register a callback function that will be invoked when a GPIO interrupt occurs.

```
void gpio_intr_handler_register(
    gpio_intr_handler_fn_t callbackFunction,
    void *arg)
```

The signature of the handler function must be:

```
void (*functionName)(uint32 interruptMask, void *arg)
```

Includes:

- `gpio.h`

See also:

- `GPIO Interrupt handling`

`gpio_pin_intr_state_set`

```
void gpio_pin_intr_state_set(
    uint32 pinId,
    GPIO_INT_TYPE intr_state)
```

The `pinId` is the GPIO pin id value returned from `GPIO_ID_PIN(num)`.

The `intr_state` parameter defines what triggers the interrupt.

Includes:

- `gpio.h`

See also:

- GPIO Interrupt handling
- GPIOs
- GPIO_INT_TYPE

gpio_intr_pending

Obtain the set of pending interrupts

```
uint32 gpio_intr_pending()
```

Includes:

- gpio.h

See also:

- GPIO Interrupt handling

gpio_intr_ack

Flag a set of interrupts as having been handled. This should be called from an interrupt handler function.

```
void gpio_intr_ack(uint32 ack_mask)
```

Includes:

- gpio.h

gpio_pin_wakeup_enable

Define that the device can wakeup from light-sleep mode when an IO interrupt occurs.

```
void gpio_pin_wakeup_enable(  
    uint32 pin,  
    GPIO_INT_TYPE intr_state)
```

The `pin` parameter defines the pin number used to wake the device.

The `intr_state` defines which type of transition will wake the device. The choices are:

- GPIO_PIN_INTR_LOLEVEL
- GPIO_PIN_INTR_HILEVEL

Includes:

- gpio.h

See also:

- GPIOs
- GPIO_INT_TYPE

```
gpio_pin_wakeup_disable  
void gpio_pin_wakeup_disable()
```

Includes:

- gpio.h

UART APIs

These functions have to be compiled in from the uart files in `driver.lib`.

UART_CheckOutputFinished

```
bool UART_CheckOutputFinished(uint8 uart_no, uint32 time_out_us)
```

UART_ClearIntrStatus

```
void UART_ClearIntrStatus(uint8 uart_no, uint32 clr_mask);
```

UART_ResetFifo

```
void UART_ResetFifo(uint8 uart_no);
```

UART_SetBaudrate

Set the baud rate.

```
void UART_SetBaudrate(uint8 uart_no, uint32 baud_rate)
```

Set the baud rate used by the UART. The `uart_no` identifies the UART to set (0 or 1) and the `baud_rate` is the desired baud rate. UARTs have definitions of UART0 and UART1.

UART_SetFlowCtrl

```
void UART_SetFlowCtrl(uint8 uart_no, UART_HwFlowCtrl flow_ctrl, uint8 rx_thresh)
```

UART_SetIntrEna

```
void UART_SetIntrEna(uint8 uart_no, uint32 ena_mask)
```

UART_SetLineInverse

```
void UART_SetLineInverse(uint8 uart_no, UART_LineLevelInverse inverse_mask)
```

UART_SetParity

Set the parity.

```
void UART_SetParity(uint8 uart_no, UartParityMode parity_mode)
```

Set the parity used by the UART. The `uart_no` identifies the UART to set (0 or 1) and the `parity_mode` defines what to use.

UART_SetPrintPort

Set the output terminal.

```
void UART_SetPrintPort(uint8 uart_no)
```

Set the output terminal. Set the UART to be used when writing debug via `os_printf()`.

UARTs have definitions of UART0 and UART1.

UART_SetStopBits

Set how long the stop bits should be.

```
void UART_SetStopBits(uint8 uart_no, UartStopBitsNum bit_num)
```

Set how long the stop bits should be. The `num` identifies the number of stop bits to use.

UART_SetWordLength

Set the number of bits in a transmission unit.

```
void UART_SetWordLength(uint8 uart_no, UartBitsNum4Char len)
```

Set the number of bits in a transmission unit. The `uart_no` identifies the UART to set (0 or 1) and the `len` parameter defines how many bits.

UART_WaitTxFifoEmpty

Wait for the TX buffer to empty.

```
void UART_WaitTxFifoEmpty(uint8 uart_no, uint32 time_out_us)
```

Wait for the TX buffer to empty. The `uart_no` identifies the UART to set (0 or 1) and the `time_out_us` specifies how long to wait before giving up. The value is supplied in microseconds.

uart_init

```
void uart_init(UartBautRate uart0BaudRate, UartBautRate uart1BaudRate)
```

There appears to be a typo in the data type ... but likely we will be stuck with that now.
The `UartBautRate` is an enum that contains:

- `BIT_RATE_9600`
- `BIT_RATE_19200`
- `BIT_RATE_38400`
- `BIT_RATE_57600`
- `BIT_RATE_74880`
- `BIT_RATE_115200`
- `BIT_RATE_230400`
- `BIT_RATE_460800`
- `BIT_RATE_921600`

See also:

- Working with serial

`uart0_tx_buffer`

Transmit a buffer of data via UART0.

```
void uart0_tx_buffer(uint8 *buffer, uint16 length)
```

Transmit the data pointed to by the `buffer` for the given `length`.

See also:

- Working with serial

`uart0_sendStr`

Transmit a string of data via UART0.

```
void uart0_sendStr(const char *str)
```

Transmit a string of data via UART0. The string to send is supplied in the `str` parameter.

`uart0_rx_intr_handler`

Handle the receiving of data via UART0.

```
void uart0_rx_intr_handler(void *parameter)
```

The parameter is a pointer to a `RcvMsgBuff` structure. My best guess on how to use this function is to create it in `user_main.c` and its mere existence will cause it to be invoked at the appropriate time. Looking at the sample supplied, we see that it needs a detailed low level implementation.

See also:

- Working with serial

I2C Master APIs

These functions have to be compiled in from the `i2c_master` files in `driver_lib`.

See also:

- Working with I2C

`i2c_master_checkAck`

Retrieve the ack from the data bus and return true or false.

```
bool i2c_master_checkAck()
```

Retrieve the ack from the data bus and return true or false.

`i2c_master_getAck`

Retrieve the ack from the data bus and return its value.

```
uint8 i2c_master_getAck()
```

Retrieve the ack from the data bus and return its value. It isn't clear why this function might be exposed as well as the `i2c_master_checkAck()`.

`i2c_master_gpio_init`

Configure the GPIOs and then call `i2c_master_init()`.

```
void i2c_master_gpio_init()
```

Configure the GPIOs and then call `i2c_master_init()`.

`i2c_master_init`

Initialize I2C functions.

```
void i2c_master_init()
```

Initialize I2C functions.

i2c_master_readByte
uint8 i2c_master_readByte()

i2c_master_send_ack

void i2c_master_send_ack()

i2c_master_send_nack

void i2c_master_send_nack()

i2c_master_setAck

Set ack to I2C bus as level value.

void i2c_master_setAck(uint8 level)

Set ack to I2C bus as level value.

i2c_master_start

Set I2C to send state.

void i2c_master_start()

Set I2C to send state.

i2c_master_stop

Set I2C to stop sending state.

void i2c_master_stop()

Set I2C to stop sending state.

i2c_master_writeByte

void i2c_master_writeByte(uint8 wrdata)

SPI APIs

These functions have to be compiled in from the SPI files in driver_lib.

cache_flush

`spi_lcd_9bit_write`

`spi_mast_byte_write`

`spi_byte_write_espslave`

`spi_slave_init`

`spi_slave_isr_handler`

`hspi_master_readwrite_repeat`

`spi_test_init`

PWM APIs

`pwm_init`

Initialize PWM.

```
void pwm_init(
    uint32 period,
    uint32 *duty,
    uint32 num_pwm_channels,
    uint32 (*pin_info_list)[3])
```

The `period` parameter is the PWM period. The value is measured in microseconds with a minimum value of 1000 giving a 1KHz period (there are 1000 periods of 1000 microseconds in a second).

The `duty` parameter is the duty ratio of each PWM channel.

The `num_pwm_channels` is the number of PWM channels being defined. There can be up to `PWM_CHANNEL_NUM_MAX` channels. Currently this is defined as 8.

The `pin_info_list` is a pointer to an array of `num_pwm_channels * 3` instances of `uint32`s that provides the PWM pin mappings. The parameters per PWM channel are:

- GPIO register
- IO reuse of corresponding pin
- GPIO number

For example:

```

uint32 pinInfoList[][3] = {
    {PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12, 12},
    {PERIPHS_IO_MUX_MTDO_U, FUNC_GPIO15, 15},
    {PERIPHS_IO_MUX_MTCK_U, FUNC_GPIO13, 13}
};

```

See also:

- Pulse Width Modulation – PWM
- `pwm_set_duty`
- `pwm_set_period`
- `pwm_start`

pwm_start

```
void pwm_start()
```

After configuring the parameters for PWM, this function should be called.

See also:

- Pulse Width Modulation – PWM

pwm_set_duty

```
void pwm_set_duty(uint32 duty, uint8 channel)
```

The resolution of a duty step is 45 nanoseconds. Here we can set the number of duty steps in a cycle. For example, imagine we have a period of 1KHz. This means that 1 cycle is 1000 microseconds. If we want the duty cycle to be 50%, then the output has to be high for 500 microseconds. 500 microseconds is 11111 units of 45 nanoseconds and that would become the duty value. Formulaic-ally, the duty ratio is $(duty * 45) / (period * 1000)$.

The `duty` parameter supplies the number of 45 nanosecond intervals that the output will be high in one period.

```
duty = 1000000 / 0.045 / frequency
```

The `channel` parameter specifies which of the PWM channels is being changed.

After changing the duty value, a call to `pwm_start()` is required to recalculate the values.

See also:

- Pulse Width Modulation – PWM
- `pwm_get_duty`
- `pwm_init`

pwm_get_duty

```
uint32 pwm_get_duty(uint8 channel)
```

Get the duty value of the specified channel.

See also:

- Pulse Width Modulation – PWM
- `pwm_get_duty`
- `pwm_init`

`pwm_set_period`

Set the period for PWM operations.

```
void pwm_set_period(uint32 period)
```

The `period` parameter is the PWM period. The value is measured in microseconds with a minimum value of 1000 giving a 1KHz period (there are 1000 periods of 1000 microseconds in a second).

See also:

- Pulse Width Modulation – PWM
- `pwm_get_period`
- `pwm_init`

`pwm_get_period`

```
uint32 pwm_get_period()
```

Get the current setting of the PWM period.

See also:

- Pulse Width Modulation – PWM
- `pwm_set_period`
- `pwm_init`

`get_pwm_version`

```
uint32 get_pwm_version()
```

See also:

- Pulse Width Modulation – PWM

`set_pwm_debug_en(uint8 print_en)`

Used to enable or disable debug print.

Bit twiddling

- `BIT(b)` – The 2^b value

ESP Now

`esp_now_add_peer`
`esp_now_deinit`
`esp_now_del_peer`
`esp_now_get_peer_key`
`esp_now_get_peer_role`
`esp_now_get_self_role`
`esp_now_init`
`esp_now_register_recv_cb`
`esp_now_register_send_cb`
`esp_now_send`

The maximum amount of data that can be sent as a unit is 256 bytes.

`esp_now_set_kok`
`esp_now_set_peer_role`
`esp_now_set_peer_key`
`esp_now_set_self_role`
`esp_now_unregister_recv_cb`
`esp_now_unregister_send_cb`

SPIFFS

When an API call is made to SPIFFS, it can possibly set an error code that can be retrieved by a call to `SPIFFS_errno()`. The returned value can be one of the following:

Symbol	Value	Meaning
<code>SPIFFS_OK</code>	0	
<code>SPIFFS_ERR_NOT_MOUNTED</code>	-10000	
<code>SPIFFS_ERR_FULL</code>	-10001	
<code>SPIFFS_ERR_NOT_FOUND</code>	-10002	
<code>SPIFFS_ERR_END_OF_OBJECT</code>	-10003	
<code>SPIFFS_ERR_DELETED</code>	-10004	

SPIFFS_ERR_NOT_FINALIZED	-10005	
SPIFFS_ERR_NOT_INDEX	-10006	
SPIFFS_ERR_OUT_OF_FILE_DESC	-10007	
SPIFFS_ERR_FILE_CLOSED	-10008	
SPIFFS_ERR_FILE_DELETED	-10009	
SPIFFS_ERR_BAD_DESCRIPTOR	-10010	
SPIFFS_ERR_IS_INDEX	-10011	
SPIFFS_ERR_IS_FREE	-10012	
SPIFFS_ERR_INDEX_SPAN_MISMATCH	-10013	
SPIFFS_ERR_DATA_SPAN_MISMATCH	-10014	
SPIFFS_ERR_INDEX_REF_FREE	-10015	
SPIFFS_ERR_INDEX_REF_LU	-10016	
SPIFFS_ERR_INDEX_REF_INVALID	-10017	
SPIFFS_ERR_INDEX_FREE	-10018	
SPIFFS_ERR_INDEX_REF_LU	-10019	
SPIFFS_ERR_INDEX_INVALID	-10020	
SPIFFS_ERR_NOT_WRITABLE	-10021	
SPIFFS_ERR_NOT_READABLE	-10022	
SPIFFS_ERR_CONFLICTING_NAME	-10023	
SPIFFS_ERR_NOT_CONFIGURED	-10024	
SPIFFS_ERR_NOT_A_FS	-10025	
SPIFFS_ERR_MOUNTED	-10026	
SPIFFS_ERR_ERASE_FAIL	-10027	
SPIFFS_ERR_MAGIC_NOT_POSSIBLE	-10028	
SPIFFS_ERR_NO_DELETED_BLOCKS	-10029	
SPIFFS_ERR_INTERNAL	-10050	
SPIFFS_ERR_TEST	-10100	
???	-10072	Possibly attempt to create a file that already exists. Could also mean "no error".

See also:

- [Spiffs File System](#)

[**esp_spiffs_deinit**](#)

[**esp_spiffs_init**](#)

Initialize SPIFFS.

```
sint32 esp_spiffs_init(struct esp_spiffs_config *config)
```

The config parameter is a structure defining the initialization information for SPIFFS.

It contains:

- phys_size
- phys_addr
- phys_erase_block
- log_block_size
- log_page_size
- fd_buf_size
- cache_buf_size

An example configuration might be:

```
struct esp_spiffs_config config;
config.phys_size = FS1_FLASH_SIZE;
config.phys_addr = FS1_FLASH_ADDR;
config.phys_erase_block = SECTOR_SIZE;
config.log_block_size = LOG_BLOCK;
config.log_page_size = LOG_PAGE;
config.fd_buf_size = FD_BUF_SIZE * 2;
config.cache_buf_size = CACHE_BUF_SIZE;
```

A return code of 0 means success.

SPIFFS_check

Runs a consistency check on given filesystem.

```
s32_t SPIFFS_check(spiffs *fs)
```

SPIFFS_clearerr

Clears last error.

```
void SPIFFS_clearerr(spiffs *fs)
```

SPIFFS_close

Closes a filehandle. If there are pending write operations, these are finalized before closing.

```
void SPIFFS_close(spiffs *fs, spiffs_file filehandle)
```

Close the filehandle that was previously opened with a call to SPIFFS_open().

See also:

- SPIFFS_open

SPIFFS_closedir

Closes a directory stream.

```
s32_t SPIFFS_closedir(spiffs_DIR *spiffsDir)
```

The directory stream should have been previously opened with a call to `SPIFFS_opendir()`.

See also:

- `SPIFFS_opendir`
- `SPIFFS_readdir`

SPIFFS_create

Create a specific file.

```
s32_t SPIFFS_create(spiffs *fs, char *path, spiffs_mode mode)
```

One normally uses `SPIFFS_open()` to create a file.

SPIFFS_erase_deleted_block

Erase deleted blocks in the file system.

```
s32_t SPIFFS_erase_deleted_block(spiffs *fs)
```

SPIFFS_errno

Get the last error code.

```
s32_t SPIFFS_errno(spiffs *fs)
```

Retrieve the last error code.

SPIFFS_fflush

Flush all write operations from cache to the file system.

```
s32_t SPIFFS_fflush(spiffs *fs, spiffs_file filehandle)
```

SPIFFS_format

Formats the entire file system.

```
s32_t SPIFFS_format(spiffs *fs);
```

All data will be lost. The filesystem must not be mounted when calling this. NB: formatting is awkward. Due to backwards compatibility, `SPIFFS_mount` MUST be called

prior to formatting in order to configure the filesystem. If SPIFFS_mount succeeds, SPIFFS_unmount must be called before calling SPIFFS_format. If SPIFFS_mount fails, SPIFFS_format can be called directly without calling SPIFFS_unmount first.

SPIFFS_fremove

Remove a file by its file handle.

```
s32_t SPIFFS_fremove(spiffs *fs, spiffs_file filehandle)
```

Remove a file by its file handle.

SPIFFS_fstat

Get the status of a file by a file handle.

```
s32_t SPIFFS_fstat(spiffs *fs,  
                    spiffs_file filehandle,  
                    spiffs_stat *spiffsStat)
```

The spiffs_stat contains:

- obj_id
- size – The size of the content of the file.
- type
- name – The name of the file.

SPIFFS_gc

Perform an explicit garbage collection.

```
s32_t SPIFFS_gc(spiffs *fs, u32_t size)
```

Invoke the garbage collection to ensure that there is enough space for size bytes.

SPIFFS_gc_quick

Perform an explicit garbage collection.

```
s32_t SPIFFS_gc_quick(spiffs *fs, u16_t max_free_pages)
```

SPIFFS_info

Return the amount of storage in total and amount actually used.

```
s32_t SPIFFS_info(spiffs *fs, u32_t *total, u32_t *used)
```

The total parameter is the total number of bytes in the file system. The used parameter is the amount of space used.

SPIFFS_lseek

Move the read/write offset for the file.

```
s32_t SPIFFS_lseek(spiffs *fs, spiffs_file filehandle, s32_t offset, int whence)
```

- `fs` – The file system that owns the file.
- `filehandle` – An open handle to the file.
- `offset` – An amount to move within the file.
- `whence` – The direction of movement:
 - `SPIFFS_SEEK_SET` – Move to specific location.
 - `SPIFFS_SEEK_CUR` – Move relative to the current location.
 - `SPIFFS_SEEK_END` – Move relative to the end of the file.

SPIFFS_mount

Initializes the file system dynamic parameters and mounts the filesystem. If `SPIFFS_USE_MAGIC` is enabled the mounting may fail with `SPIFFS_ERR_NOT_A_FS` if the flash does not contain a recognizable file system. In this case, `SPIFFS_format` must be called prior to remounting.

```
s32_t SPIFFS_mount(  
    spiffs *fs,  
    spiffs_config *config,  
    u8_t *work,  
    u8_t *fd_space, u32_t fd_space_size,  
    void *cache, u32_t cache_size,  
    spiffs_check_callback check_cb_f);
```

- `fs` – The file system struct.
- `config` – the physical and logical configuration of the file system.
- `work`
- `fd_space`
- `fd_space_size` – Example 32*4.
- `cache`
- `cache_size` – Example (128 + 32) * 8.
- `check_cb_f`

The `spiffs_config` structure contains:

- `hal_read_f` – physical read function. This is a function with the signature:

`s32_t func(u32_t addr, u32_t size, u8_t *dst)`

- `hal_write_f` – physical write function. This is a function with the signature:

`s32_t func(u32_t addr, u32_t size, u8_t *src)`

- `hal_erase_f` – physical erase function. This is a function with the signature:

`s32_t func(u32_t addr, u32_t size)`

- `phys_size` – physical size of the spi flash.
- `phys_addr` – physical offset in spi flash used for spiffs, must be on block boundary.
- `phys_erase_block` – physical size when erasing a block.
- `log_block_size` – logical size of a block, must be on physical block size boundary and must never be less than a physical block. Example 4*1024.
- `log_page_size` – logical size of a page, must be at least `log_block_size / 8`. Example 128.

SPIFFS_mounted

Checks whether the file system is mounted.

`u8_t SPIFFS_mounted(spiffs *fs)`

Returns 0 if **not** mounted.

SPIFFS_open

Open a file.

```
spiffs_file SPIFFS_open(
    spiffs *fs,
    char *path,
    spiffs_flags flags,
    spiffs_mode mode)
```

Open a file. This can also include the creation of the file when it is opened.

- `fs` – The file system to open.
- `path` – The path to the file to open.
- `flags` – Control flags for opening the file. A combination of:
 - `SPIFFS_APPEND`
 - `SPIFFS_CREAT`

- SPIFFS_DIRECT
- SPIFFS_RDONLY
- SPIFFS_RDWR
- SPIFFS_TRUNC
- SPIFFS_WRONLY
- mode – The mode for the open. Ignored in this release.

See also:

- SPIFFS_close

SPIFFS_open_by_dirent

Open a file by its directory entry.

```
spiffs_file SPIFFS_open_by_dirent(
    spiffs *fs,
    struct spiffs_dirent *spiffsDirEnt,
    spiffs_flags flags,
    spiffs_mode mode)
```

Open a file.

- fs – The file system to open.
- spiffsDirEnt – The path to the file to open.
- flags – Control flags for opening the file. A combination of:
 - SPIFFS_APPEND
 - SPIFFS_DIRECT
 - SPIFFS_RDONLY
 - SPIFFS_RDWR
 - SPIFFS_TRUNC
 - SPIFFS_WRONLY

SPIFFS_opendir

Open a directory stream for the directory name specified.

```
spiffs_DIR *SPIFFS_opendir(
    spiffs *fs,
    char *directoryName,
    spiffs_DIR *spiffsDir)
```

- fs – The SPIFFS file system to be worked against.

- `directoryName` – The name of the directory to be read.
- `spiffsDir` – The directory structure to be populate.

See also:

SPIFFS_read

Read data from a file.

`s32_t SPIFFS_read(spiffs *fs, spiffs_file filehandle, void *buf, s32_t len)`

Read data from a file and place in a buffer.

SPIFFS_readdir

Read the directory.

```
struct spiffs_dirent *SPIFFS_readdir(
    spiffs_DIR *spiffsDir,
    struct spiffs_dirent *spiffsDirEnt)
```

Read the directory specified by `spiffsDir` which had previously been opened with `SPIFFS_opendir()`.

A `struct spiffs_dirent` contains:

- `obj_id`
- `name`
- `type`
- `size`
- `pix`

See also:

- Spiffs File System
- `SPIFFS_opendir`
- `SPIFFS_closedir`
- `SPIFFS_open_by_dirent`

SPIFFS_remove

Remove a file by name.

`s32_t SPIFFS_remove(spiffs *fs, char *path)`

SPIFFS_rename

Rename a file.

```
s32_t SPIFFS_rename(spiffs *fs, char *old, char *newPath)
```

SPIFFS_stat

Get the status of a file by path.

```
s32_t SPIFFS_stat(  
    spiffs *fs,  
    char *path,  
    spiffs_stat *spiffsStat)
```

- `fs` – The file system holding the file.
- `path` – The path to the file.
- `spiffsStat` – The stats data of the file.

The `spiffs_stat` contains:

- `obj_id`
- `size`
- `type`
- `name`

SPIFFS_unmount

Unmount a file system.

```
void SPIFFS_unmount(spiffs *fs)
```

SPIFFS_write

Write data into an open file.

```
s32_t SPIFFS_write(  
    spiffs *fs,  
    spiffs_file filehandle,  
    void *buf, s32_t len)
```

Lib-C

The FreeRTOS environment provides a set of C runtime library routines that are defined in "esp_libc.h".

atoi
int atoi(const char *s)

atol
long atol(const char *s)

bzero
void bzero(void *s, size_t n)

calloc
void *calloc(size_t c, size_t n)

free
void free(void *p)

malloc
void *malloc(size_t n)

memcmp
int memcmp(const void *m1, const void *m2, size_t n)

memcpy
void *memcpy(void *dst, const void *src, size_t n)

memmove
void *memmove(void *dst, const void *src, size_t n)

memset
void *memset(void *dst, int c, size_t n)

os_get_random
int os_get_random(unsigned char *buf, size_t len)

os_random
unsigned long os_random(void)

printf
int printf(const char *format, ...)

Need to include "stdio.h".

puts
int puts(const char *str)

rand
Generate a random number.

int rand()

Return a random number. Note that the result is an integer which is signed.

realloc
void *realloc(void *p, size_t n)

snprintf
int snprintf(char *buf, unsigned int count, const char *format, ...)

sprintf
int sprintf(char *out, const char *format, ...)

strcat
char *strcat(char *dst, const char *src)

strchr
char *strchr(const char *s, int c)

strcmp

```
int strcmp(const char *s1, const char *s2)
```

strcpy

```
char *strcpy(char *dst, const char *src)
```

strcspn

```
size_t strcspn(const char *s, const char *reject)
```

strdup

```
char *strdup(const char *s)
```

strlen

Return the length of a null terminated string.

```
size_t strlen(const char *s)
```

Return the length of a null terminated string.

strncat

```
char *strncat(char *dst, const char *src, size_t count)
```

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n)
```

strncpy

```
char *strncpy(char *dst, const char *src, size_t n)
```

strrchr

```
char *strrchr(const char *s, int c)
```

strspn

```
size_t strspn(const char *s, const char *accept)
```

strstr

```
char *strstr(const char *s1, const char *s2)
```

strtok

```
char *strtok(char *s, const char *delim)
```

strtok_r

```
char *strtok_r(char *s, const char *delim, char **ptrptr)
```

strtol

```
long strtol(const char *str, char **endptr, int base)
```

zalloc

```
void *zalloc(size_t n)
```

Data structures

esp_spiffs_config

- `phys_size` – Physical size of the SPI Flash.
- `phys_addr` – Physical offset in SPI flash used for spiffs. Must be on a block boundary.
- `phys_erase_block` – Physical size when erasing a block.
- `log_block_size` – Logical size of a block. Must match the physical size of a block.
- `log_page_size` – Logical size of a page.
- `fd_buf_size` – File descriptor memory area size.
- `cache_buf_size` – The cache buffer size.

station_config

A description of a station configuration. Contains the following fields:

- `uint8 ssid[32]` – The SSID of the access point.
- `uint8 password[64]` – The password to access the access point.
- `uint8 bssid_set` – Flag to indicate whether or not to use the `bssid` property. A value of 1 means to use and a value of 0 means to not use.

- `uint8 bssid[6]` – If several access points have the same SSID, BSSID can contain a MAC address to indicate which of the access points to connect to.

See also:

- Station configuration
- `wifi_station_get_config_default`
- `wifi_station_set_config_current`

struct softap_config

Configuration control structure for softAP.

- `uint8 ssid[32]`
- `uint8 password[64]`
- `uint8 ssid_len` – The length of the SSID. If 0, then the ssid is null terminated.
- `uint8 channel` – The channel to be used for communication. Values are 1 to 13.
- `uint8 authmode` – The authentication mode required. The choices are:
 - `AUTH_OPEN`
 - `AUTH_WPA2_PSK`
 - `AUTH_WPA_PSK`
 - `AUTH_WPA_WPA2_PSK`

`AUTH_WEP` is not supported.

- `uint8 ssid_hidden` – Whether or not this SSID is hidden. A value of 1 makes it hidden.
- `uint8 max_connection` – The maximum number of station connections. The maximum and default is 4.
- `uint16 beacon_interval` – The beacon interval in milliseconds. Values are 100 – 60000.

See also:

- `wifi_softap_get_config`
- `wifi_softap_get_config_default`
- `wifi_softap_set_config_current`

struct station_info

This structure provides information on the stations connected to an ESP8266 while it is an access point. It is a linked list with properties:

- `uint8 bssid[6]` – The ???

- `struct ipaddr ip` – The IP address of the connected station

To get the next entry, we can use `STAILQ_NEXT(pStationInfo, next)`.

See also:

- Being an access point

struct dhcps_lease

This structure is used by the `wifi_softap_dhcps_lease()` function to define the start and end range of available IP addresses.

The fields contained within are:

- `struct ip_addr start_ip`
- `struct ip_addr end_ip`

Includes:

- `user_interface.h`

See also:

- The DHCP server

struct bss_info

This structure contains:

- `STAILQ_ENTRY(bss_info) next`
- `uint8 bssid[6]`
- `uint8 ssid[32]`
- `uint8 channel`
- `sint8 rssi` – The received signal strength indication
- `AUTH_MODE authmode`
- `uint8 is_hidden`
- `sint16 freq_offset`

To get the next entry, we can use `STAILQ_NEXT(pBssInfoVar, next)`.

The `AUTH_MODE` is an enum

- `AUTH_OPEN` – No authentication. No challenge on any station connect.
- `AUTH_WEP = 1`
- `AUTH_WPA_PSK = 2`

- AUTH_WPA2_PSK = 3
- AUTH_WPA_WPA2_PSK = 4

See also:

- Scanning for access points

struct ip_info

This structure defines information about an interface possessed by the ESP8266. It contains the following fields:

- struct ip_addr ip – The IP address of the interface.
- struct ip_addr netmask – The netmask used by the interface.
- struct ip_addr gw – The IP address of the gateway used by the interface.

See also:

- struct ip_addr
- IP4_ADDR

struct rst_info

Information about the current boot/restart

This structure contains:

- uint32 reason
- uint32 exccause
- uint32 epc1
- uint32 epc2
- uint32 epc3
- uint32 excvaddr
- uint32 depc

The `reason` field is an enum with the following values:

- 0 – Default restart – Normal start-up on power up
- 1 – Watch dog timer – Hardware watchdog reset
- 2 – Exception – An exception was detected
- 3 – Software watch dog timer – Software watchdog reset
- 4 – Soft restart

- 5 – Deep sleep wake up

See also:

- Exception handling
- Error: Reference source not found

struct espconn

This data structure is the representation of a connection between the ESP8266 and a partner. It contains the "control blocks" and identification information ... however it is important to note that it is not always an opaque piece of data.

- `enum espconn_type type` – The type can be one of
 - `ESPCONN_INVALID`
 - `ESPCONN_TCP` – Identifies this connection as being of type TCP.
 - `ESPCONN_UDP` – Identifies this connection as being of type UDP.
- `enum espconn_state` – The state can be one of
 - `ESPCONN_NONE` – The state for an initial connection.
 - `ESPCONN_WAIT`
 - `ESPCONN_LISTEN`
 - `ESPCONN_CONNECT`
 - `ESPCONN_WRITE`
 - `ESPCONN_READ`
 - `ESPCONN_CLOSE`
- `union {`
 - `esp_tcp *tcp`
 - `esp_udp *udp`
- `} proto` – This field is a union of `tcp` and `udp` meaning that only one of them should ever be used for an instance of this data structure. If the data structure is used for TCP then the `tcp` property should be used while for UDP, the `udp` property should be used.
- `void *reverse` – In the comments, this is flagged as a field *reserved* for user code. It is possible the name chosen (`reverse`) is actually a typo in the header file!!
- Other fields ... there are other fields in the structure but they are not meant to be read or written to by user applications. Ignore them. Using their values is undefined and may have unexpected effects.

See also:

- TCP
- esp_tcp
- esp_udp

esp_tcp

- uint8 local_ip[4] – The local IP address
- int local_port – The local port
- uint8 remote_ip[4] – The remote IP address
- int remote_port – The remote port
- Other fields ... there are other fields in the structure but they are not meant to be read or written to by user applications. Ignore them. Using their values is undefined and may have unexpected effects.

See also:

- struct espconn

esp_udp

This data structure is used in the `proto` property of the `struct espconn` control block.

- int remote_port – The local IP address
- int local_port – The local port
- uint8 local_ip[4] – The remote IP address
- uint8 remote_ip[4] – The remote port

See also:

- struct espconn
- UDP

struct ip_addr

A representation of an IP address.

It contains the following field:

- uint32 addr – The actual 4 byte IP address.

Includes:

- ip_addr.h

See also:

- ipaddr_addr

- IP4_ADDR
- ipaddr_t

ipaddr_t

A **typedef** for `struct ipaddr`.

See also:

- `struct ip_addr`

struct ping_option

The fields contained within the structure are:

- `uint32 count` – The number of times to transmit a ping
- `uint32 ip` – The IP address that is the target of the ping
- `uint32 coarse_time`
- `recv_function recv_function`
- `sent_function sent_function`
- `void *reverse;`

Includes:

- `ping.h`

See also:

- Ping request
- `ping_start`
- `ping_regist_recv`
- `ping_regist_sent`

struct ping_resp

The fields contained within the structure are:

- `uint32 total_count`
- `uint32 resp_time`
- `uint32 seqno`
- `uint32 timeout_count`
- `uint32 bytes`
- `uint32 total_bytes`
- `uint32 total_time`

- `sint8 ping_err` – An indication of whether or not an error occurred. A value of 0 means no error.

Includes:

- `ping.h`

See also:

- `Ping request`
- `ping_start`
- `ping_regist_recv`
- `ping_regist_sent`

struct mdns_info

- `char *host_name`
- `char *server_name`
- `uint16 server_port`
- `unsigned long ipAddr` – This should be the IP address being offered.
- `char *txt_data[10]` – An array of options of the form "`name = value`".

See also:

- Multicast Domain Name Systems

enum phy_mode

The 802.11 physical mode to be used or being used.

- `PHY_MODE_11B`
- `PHY_MODE_11G`
- `PHY_MODE_11N`

GPIO_INT_TYPE

These are the possible triggers for an interrupt. This is an enum defined as follows:

- `GPIO_PIN_INTR_DISABLE` – Interrupts are disabled.
- `GPIO_PIN_INTR_POSEDGE` – Interrupt on a positive edge transition.
- `GPIO_PIN_INTR_NEGEDGE` – Interrupt on a negative edge transition.
- `GPIO_PIN_INTR_ANYEDGE` – Interrupt on any edge transition.
- `GPIO_PIN_INTR_LOLEVEL` – Interrupt when low.
- `GPIO_PIN_INTR_HILEVEL` – Interrupt when high.

See also:

- `gpio_pin_wakeup_enable`

System_Event_t

The event type contains:

- `uint32 event` – The type of event that occurred. Can be
 - `EVENT_STAMODE_CONNECTED` (0) – We have successfully connected to an access point.
 - `uint8[32] event_info.connected.ssid` – The SSID of the access point.
 - `uint8 ssid_len`
 - `uint8[6] bssid`
 - `event_info.connected.channel` – The channel used to connect to the access point.
 - `EVENT_STAMODE_DISCONNECTED` (1)
 - `uint8[6] event_info.disconnected.bssid`
 - `uint8[32] event_info.disconnected.ssid`
 - `uint8 ssid_len`
 - `uint8 event_info.disconnected.reason` – The reason is one of the following:
 - `REASON_UNSPECIFIED = 1`
 - `REASON_AUTH_EXPIRE = 2`
 - `REASON_AUTH_LEAVE = 3`
 - `REASON_ASSOC_EXPIRE = 4`
 - `REASON_ASSOC_TOOMANY = 5`
 - `REASON_NOT_AUTHED = 6`
 - `REASON_NOT_ASSOCED = 7`
 - `REASON_ASSOC_LEAVE = 8`
 - `REASON_ASSOC_NOT_AUTHED = 9`
 - `REASON_DISASSOC_PWRCAP_BAD = 10`
 - `REASON_DISASSOC_SUPCHAN_BAD = 11`
 - `REASON_IE_INVALID = 13`

- REASON_MIC_FAILURE = 14
- REASON_4WAY_HANDSHAKE_TIMEOUT = 15
- REASON_GROUP_KEY_UPDATE_TIMEOUT = 16
- REASON_IE_IN_4WAY_DIFFERS = 17
- REASON_GROUP_CIPHER_INVALID = 18
- REASON_PAIRWISE_CIPHER_INVALID = 19
- REASON_AKMP_INVALID = 20
- REASON_UNSUPP_RSN_IE_VERSION = 21
- REASON_INVALID_RSN_IE_CAP = 22
- REASON_802_1X_AUTH_FAILED = 23
- REASON_CIPHER_SUITE_REJECTED = 24
- REASON_BEACON_TIMEOUT = 200
- REASON_NO_AP_FOUND = 201
- EVENT_STAMODE_AUTHMODE_CHANGE (2)
 - event_info.auth_change.old_mode
 - event_info.auth_change.new_mode
- EVENT_STAMODE_GOT_IP (3)
 - event_info.got_ip.ip
 - event_info.got_ip.mask
 - event_info.got_ip.gw
- EVENT_SOFTAPMODE_STACONNECTED (4)
 - event_info.sta_connected.mac
 - event_info.sta_connected.aid
- EVENT_SOFTAPMODE_STADISCONNECTED (5)
 - event_info.sta_disconnected.mac
 - event_info.sta_disconnected.aid
- EVENT_STAMODE_DHCP_TIMEOUT
- EVENT_SOFTAPMODE_PROBEREQRECVED
- Event_Info_u event_info

This is a C Union containing data that is available as a function of the event type.

- Event_StaMode_Connected_t connected

- Event_StaMode_Disconnected_t disconnected
- Event_StaMode_AuthMode_Change_t auth_change
- Event_StaMode_Got_IP_t got_ip
- Event_SoftAPMode_StaConnected_t sta_connected
- Event_SoftAPMode_StaDisconnected_t sta_disconnected

See also:

- Error: Reference source not found

espconn error codes

Constant	Value
ESPCONN_OK	0
ESPCONN_MEM	-1
ESPCONN_TIMEOUT	-3
ESPCONN_RTE	-4
ESPCONN_INPROGRESS	-5
ESPCONN_ABRT	-8
ESPCONN_RST	-9
ESPCONN_CLSD	-10
ESPCONN_CONN	-11
ESPCONN_ARG	-12
ESPCONN_ISCONN	-15
ESPCONN_HANDSHAKE	-28
ESPCONN_PROTO_MSG	-61

STATUS

This is an enum defined as follows:

Enum Name	Value
OK	0
FAIL	1
PENDING	2
BUSY	3
CANCEL	4

See also:

- Error: Reference source not found

Reference materials

There is a wealth of information available on the ESP8266 from a variety of sources.

C++ Programming

Simple class definition

Sample class header

```
#ifndef MyClass_h
#define MyClass_h

class MyClass {
public:
    MyClass();
    static void myStaticFunc();
    void myFunc();
};

#endif
```

Sample class source

```
#include <MyClass.h>
MyClass::MyClass() {
    // Constructor code here ...
}
String MyClass::myStaticFunc() {
    // Code here ...
}
void MyClass::myFunc() {
    // Code here ...
}
```

Lambda functions

Modern C++ has introduced lambda functions. These are C++ language functions that don't have to be pre-declared but can instead be declared "inline". The functions have no names associated with them but otherwise behave just like other functions.

See also:

- [Lambda functions](#)

Ignoring warnings

From time to time, your code may issue compilation warnings that you wish to suppress. One way to achieve this is through the use of the C compile `#pragma` directive.

For example:

```
#pragma GCC diagnostic ignored "-Wformat"
```

See also:

- [GCC Diagnostic Pragmas](#)

Eclipse

Although not technically an ESP8266 story, I feel an understanding of the major components of Eclipse will do no harm.

See also:

- [Eclipse mars documentation](#)

ESPFS breakdown

The ESPFS is a library which stores "files" within the flash of the ESP8266 and allows an application to read them. It is part of the ESPHTTPD project.

EspFsInit

```
EspFsInitResult espFsInit(char *flashAddress)
```

Initialize the environment pointing to where the file data can be found. The return will be one of:

- `ESPFS_INIT_RESULT_OK`
- `ESPFS_INIT_RESULT_NO_IMAGE`
- `ESPFS_INIT_RESULT_BAD_ALIGN`

espFsOpen

```
EspFsFile *espFsOpen(char *fileName)
```

Open the file specified by the file name and return a structure that is the "handle" to the file or NULL if the file can not be found.

espFsClose

```
void espFsClose(EspFsFile *fileHandle)
```

Close the file that was previously opened by a call to `espFsOpen()`. No further reads should be performed.

espFsFlags

```
int espFsFlags(EspFsFile *fileHandle)
```

espFsRead

```
int espFsRead(EspFsFile *fileHandle, char *buffer, int length)
```

Read up to length bytes from the file and store them at the memory location pointed to by buffer. The actual number of bytes read is returned by the function call.

mkespfimage

This is not a function but a command which builds the binary data of the files to be placed in flash memory.

```
mkespfimage [-c compressor] [-l compression_level]
```

- -C
 - 0 – None
 - 1 – Heatshrink
- -l
 -

ESPHTTPD breakdown

The ESPHTTPD library provides an implementation of an HTTP server running on an ESP8266. In order to use this, we may wish to understand it better.

httpdInit

```
void httpdInit(HttpdBuiltInUrl *fixedUrls, int port)
```

Initialize the HTTP server running in the ESP. The `port` parameter is the port number that the ESP will listen upon for incoming browser requests. The default port number used by browsers is 80.

The `HttpdBuiltInUrl` is a typedef that provides mapping to URLs available on the HTTP server. The fields contained within are:

- `char *url` – The url to match.
- `cgiSendCallback cgiCb` – The callback function to call when matched.
- `const void *cgiArg` – Parameters to pass into the callback function.

It is vital that the last element in the array have NULLs for all attributes. This serves as a termination record. Here is an example definition for a minimal set of built in URLs:

```
HttpdBuiltInUrl builtInUrls[] = {  
    {NULL, NULL, NULL}  
};
```

The `cgiSendCallback` is a function with the following signature:

```
int (* functionName) (HttpdConnData *connData)
```

Includes:

- httpd.h

httpdGetMimetype

```
char *httpdGetMimeType(char *url)
```

Examine the `url` passed in and by looking at its file type, determine the MIME type of the data. If no file type is found, then the default MIME type is "text/html".

Includes:

- httpd.h

httpdUrlDecode

```
int httpdUrlDecode(char *val, int valLen, char *ret, int retLen)
```

Decode a URL according to URL decoding rules. The encoded url is supplied in `val` with a length of `valLen` bytes. The resulting decoded url string will be stored at `ret` with a maximum length of `retLen`. The actual length is returned by the function call itself.

Includes:

- httpd.h

httpdStartResponse

```
void httpdStartResponse(HttpdConnData *conn, int code)
```

Start sending the response data down the TCP connection to the browser. The `code` value is the primary browser response code.

Includes:

- httpd.h

httpdSend

```
int httpdSend(HttpdConnData *conn, const char *data, int len)
```

Send data to the browser through the TCP connection. The data is supplied as `data` and the `len` parameters is the number of bytes to write. If `len == -1`, then data is assumed to be a NULL terminated string.

Includes:

- httpd.h

httpdRedirect

```
void httpdRedirect(HttpdConnData *conn, char *newUrl)
```

Send an HTTP redirect instruction to the browser. The `newUrl` is the URL we wish the browser to use.

Includes:

- `httpd.h`

httpdHeader

```
void httpdHeader(HttpdConnData *conn, const char *field, const char *val)
```

Send an HTTP header. The name of the header is supplied in the `field` parameter and its value supplied in the `val` parameter.

Includes:

- `httpd.h`

httpdGetHeader

```
int httpdGetHeader(HttpdConnData *conn, char *header, char *ret, int retLen)
```

Search the browser supplied data header looking for a header that matches the `header` parameter. If found, return the header value at the buffer pointed to by `ret` which must be at least `retLen` bytes long.

Includes:

- `httpd.h`

httpdFindArg

```
int httpdFindArg(char *line, char *arg, char *buff, int buffLen)
```

Given a line of text, look for a parameter of the form "`name=value`" within the line. If the name matches our passed in `name`, then return the `value`.

Includes:

- `httpd.h`

httpdEndHeaders

```
void httpdEndHeaders(HttpdConnData *conn)
```

Conclude the output of headers to the output stream.

Includes:

- httpd.h

Makefiles

Books have been written on the language and use of Makefiles and our goal is not to attempt to rewrite those books. Rather, here is a cheaters guide to beginning to understand how to read them.

A general rule in a make file has the form:

```
target: prereqs ...
       recipe ...
```

Variables are defined in the form:

```
name=value
```

We can use the value of a variable with either `$(name)` or `${name}`.

Another form of definition is:

```
name:=value
```

Here, the value is locked to its value at the time of definition and will not be recursively expanded.

Some variables have well defined meanings:

Variable	Meaning
CC	C compiler command
AR	Archiver command
LD	Linker command
OBJCOPY	Object copy command
OBJDUMP	Object dump command

We can use the value of a previously defined variable in other variable definitions. For example:

```
XTENSA_TOOLS_ROOT ?= c:/Espressif/xtensa-lx106-elf/bin
CC           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
```

defines the C compiler as an absolute path based on the value of a previous variable.

Special expansions are:

- `$@` - The name of the target
- `$<` - The first prereq

Comments are lines that start with an `#` character.

Wildcards are:

- * - All characters
- ? - One character
- [...] - A set of characters

Make can be invoked recursively using

```
make -C <directoryName>
```

Imagine we wanted to build a list of source files by naming directories and the list of source files then becomes all the ".c" files, in those directories? How can we achieve that?

```
SRC_DIR = dir1 dir2
SRC := $(foreach sdir, $(SRC_DIR), $(wildcard $(sdir)/*.c))
OBJ := $(patsubst %.c, $(BUILD_BASE)/%.o, $(SRC))
```

The puzzle

Imagine a directory structure with

```
a
  a1.c
  a2.c
b
  b1.c
  b2.c
```

goal is to compile these to

```
build
  a
    a1.o
    a2.o
  b
    b1.o
    b2.o
```

We know how to compile x.c → x.o

```
MODULES=a b
BUILD_BASE=build
BUILD_DIRS=$(addprefix $(BUILD_BASE) /,$(MODULES))
SRC=$(foreach dir, $(MODULES), $(wildcard $(dir)/*.c))
# Replace all x.c with x.o
OBJS=$(patsubst %.c,%.o,$(SRC))

all:
    echo $(OBJS)
    echo $(wildcard $(OBJS)/*.c)
    echo $(foreach dir, $(OBJS), $(wildcard $(dir)/*.c))
    echo "SRC: " $(SRC)

test: checkdirs $(OBJS)
    echo "Compiled " $(SRC)
```

```

.c.o:
    echo "Compiling $(basename $<)"
    $(CC) -c $< -o build/${addsuffix .o, $(basename $<) }

checkdirs: $(BUILD_DIRS)

$(BUILD_DIRS):
    mkdir -p $@

clean:
    rm -f $(BUILD_DIRS)

```

Makefiles also have interesting commands:

- \$(shell <shell command>) - Run a shell commands
- \$(info "text"), \$(error "text"), \$(warning "text") – Generate output from make

See also:

- [GNU make](#)
- [Makefile cheat sheet](#)

Forums

There are a couple of excellent places to ask questions, answer other folks questions and read about questions and answers of the past.

- [Espressif ESP8266 BBS](#) – A moderated forum run by Espressif. The primary source for SDK downloads and the source of much of the core materials.
- [ESP8266 Community Forum](#) – A set of fora dedicated to the ESP8266 run for and by the ESP8266 user community.
- [ESP32 Community Forum](#) – The ESP32 community forum where **all** discussions of ESP32 are happening.

Reference documents

Espressif distributes PDF and Excel spreadsheets containing core information about the ESP8266. These can be downloaded freely from the web.

- [ESP8266 Technical Reference – v1.2](#)
- [ESP8266 FAQs](#)
- [ESP8266 SDK Getting Started Guide – v2.3](#)
- [ESP8266 Non-OS SDK API Reference – v2.0](#)
- [ESP-WROOM-32 Datasheet – 2016-08-22](#)
- [ESP32 Technical Reference Manual – 2016-09-23](#)
- [ESP32_RTO_SDK](#)

Old documents

- [0A-ESP8266 Datasheet v4.3](#)
- [0B-ESP8266 Hardware User Guide v1.1](#)
- [0C-ESP8266 WROOM WiFi Module Datasheet v0.3](#)
- [0D-ESP8266 Pin List Release 2014-11-15](#)

- [2A-ESP8266 IOT SDK User Manual V1.4](#)
- [2B-ESP8266 SDK IOT Demo V1.3](#)
- [2C-ESP8266 SDK Programming Guide V1.5](#)
- [4A-ESP8266 AT Instruction Set V1.5](#)
- [4B-ESP8266 AT Command Examples V1.3](#)
- 4C-ESP8266 AT upgrade example
- [8A-ESP8266 Interface GPIO v0.5](#)
- [8B-ESP8266 Interface GPIO Registers Release 2014-11-15](#)
- [8C-ESP8266 Interface I2C v1.0](#)
- [8D-ESP8266 Interface PWM v1.1](#)
- [8E-ESP8266 Interface UART v0.2](#)
- [8F-ESP8266 Interface UART Registers v0.1](#)
- [8G-ESP8266 Interface Infrared Remote Control v0.3](#)
- [8H-ESP8266 Interface SDIO SPI Mode v0.1-2](#)
- [8I-ESP8266 Interface SPI-WiFi Passthrough 1 – interrupt mode v0.1](#)
- [8J-ESP8266 Interface SPI-WiFi Passthrough 2 – interrupt mode v1.0](#)
- [8K-ESP8266 Sniffer Introduction v0.3](#)
- [8L-ESP8266 Interface SPI Registers Release 2014-11-18](#)
- [8M-ESP8266 Interface Timer Registers Release 2014-11-18](#)
- [8N-ESP8266 SPI Reference v1.0](#)
- [8O-ESP8266 SPI Overlap & Display Application Guide v0.1](#)
- [8P-ESP8266 I2S Module Description v1.0](#)
- [8Q-ESP8266 HSPI Host Multi-device API v1.0](#)
- 9A-ESP8266 FRC Timer Introduction (not yet published)
- [9B-ESP8266 Sleep Mode Function Description v1.0](#)
- [20A-ESP8266 RTOS SDK Programming Guide V1.3.0](#)
- [20B-ESP8266 RTOS SDK API Reference V1.3.0](#)
- [30A-ESP8266 Mesh User Guide V1.0](#)
- [99A-ESP8266 Flash RW Operation v0.2](#)
- 99B-ESP8266 Timer (not yet published)
- [99C-ESP8266 OTA Upgrade v1.6](#)

Here are similar reference documents for the ESP32

- [ESP32 RTOS SDK API Reference v1.1.0](#)

Github

There are a number of open source projects built on top of and around the ESP8266 that can be found on Github. Here is a list of links to some of these projects that are very well worth having a look:

- [EspressifApp](#)
- [jantje/arduinon-eclipse-plugin](#)
- [eriksl/esp8266-universal-io-bridge](#)
- [CHERTS/esp8266-devkit](#)
- ESPHTTPD project
 - [Spritelm/esphttpd](#)
 - [Spritelm/libesphttpd](#)

Github quick cheats

When working with open source projects, there are times when we would like to perform some tasks that involve multiple commands. Here we try and capture some of the more interesting ones that are used in ESP8266 projects from time to time.

```
git remote -v  
git remote add upstream <URL>  
git fetch upstream  
git merge upstream/master
```

See also:

- [Simple guide to forks in GitHub and Git](#)

SDK

The Software Development Kit (SDK) is published by Espressif and is required to build C based applications. It contains vital documentation in the form of PDF that don't appear to be available elsewhere.

- [ESP8266 SDK v1.4.0](#)

esp-open-sdk

cygwin packages

bison

flex

texinfo

wget

patch

libtool

automake

gettext-devel

Perform git clone --recursive <Repo> from within shell.

Single board computer comparisons

There are a number of single board computers on the market. Although the ESP8266 is usually not considered one of these, a lot of folks are using it as such. Let us put up a table and contrast the ESP8266 against these computers:

Device	CPU	RAM	Flash	Wifi	GPIO	OS	Cost
ESP8266	80MHz	80K	512K	Y	9	FreeRTOS	\$4
ESP32	160MHz	512K	Var	Y	?	FreeRTOS	??
Arduino	20MHz	2K	32K	N	?	N/A	\$2
Pi Zero	1GHz	512MB	SD	N	?	Linux	\$5
Omega	400MHz	64MB	16MB	Y	18	Linux	\$19
Omega 2							
C.H.I.P.	1GHz	512MB	4GB	Y	8+	Linux	\$9

Heroes

Within the ESP8266 user community there are individuals that I consider to have pushed the boundaries of knowledge further or have developed tools that dramatically improve working with the devices. I want to take a few moments and call out these good folks without whom all our ESP8266 travels would be harder:

Max Filippov - jcmvbkbc - GCC compiler for Xtensa

- Web site: Github – <https://github.com/jcmvbkbc>



A compiler for C based on GCC that compiles to Xtensa binary for flashing. It is doubtful that any useful work could be performed without this contribution.

Ivan Grokhotkov - igrr - Arduino IDE for ESP8266 development

- Web site: Github – <https://github.com/esp8266/Arduino>



An implementation of technology that allows one to develop ESP8266 applications using the Arduino IDE as well as libraries that map Arduino functions to ESP8266 equivalents or near equivalents.

jantje - Arduino Eclipse

- Web site: [Arduino Eclipse](#)
- Web site: Github – [jantje/arduino-eclipse-plugin](#)



Although not technically for just the ESP8266, the Arduino Eclipse project is important. It provides the ability to build Arduino applications using Eclipse which has a superior development environment for more experienced programmers. Combine this with the Arduino ESP8266 project and we have a fantastic environment at our disposal.

Richard Sloan - ESP8266 Community owner

- Web site: <http://www.esp8266.com/index.php>

Without question, anyone who touches the ESP8266 should visit this community web site. The forum found there is extremely rich in knowledge and heavily trafficked. Folks new to the ESP8266 and experts alike are all welcomed. Simply come along, and join the fun. Richard is also the owner of the [themindfactory.com](#).

Mikhail Grigorev - CHERTS - Eclipse for ESP8266 development

- Web site: [Project Unofficial Development Kit for Espressif ESP8266](#)
- Web site: Github – [CHERTS/esp8266-devkit](#)



An extraordinarily well polished set of artifacts and instructions for building ESP8266 C applications within the Eclipse development environment.

Mmiscool - Basic Interpreter

- Website: <http://www.esp8266basic.com>
- Website: [forums](#)



A Basic interpreter/environment for writing applications in the Basic programming language. The author devotes a lot of time and energy into continued development and provides very fast response to user's questions.

Areas to Research

- Hardware timers ... when do they get called?
- If I define functions in a library called libcommon.a, what is added to the compiled application when I link with this library? Is it everything in the library or just the object files that are referenced?
- What is the memory map/layout of the ESP8266?
- How much RAM is installed and available for use?
- Document the information contained here ...
<http://bbs.espressif.com/viewtopic.php?p=3066#p3066>
- What is SSDP and how does it relate to the SSDP libraries?
- Study Device Hive - <http://devicehive.com/>
- Document using Visual Micro debugger with Visual Studio.
<http://www.visualmicro.com/>
- Power management
- MQTT support
- Research the semantics of a wifi_station_connect() when we are already connected.
- Drive an Arduino as a slave to an ESP8266.