ESP8266 universal I/O bridge

PREFACE

General

The ESP8266 universal I/O bridge is a project that attempts to make all of the I/O on the ESP8266 available over the (wireless) network. This more or less assumes the use of an (possibly always-on) server that frequently contacts the ESP8266 to fetch the current data or to send control commands. It is not intended to program automated actions or to automatically upload results "to the cloud".

Currently the available I/O's are: all of the built-in GPIO's (such as digital input, digital output or PWM), I2C (emulated by software bit-banging), the ADC (analog input) and the UART. External GPIO's (well-known I2C I/O expanders) are currently being implemented. It also features external displays, using SAA1064 or Hitachi HD44780 type LCD displays (4x20).

The software listens at tcp port 24, and that is where the configuration and commands should be entered. Type telnet <ip_address> 24 and type ? for help. No need for flashing when the configuration changes, just change the config and write it.

There is also an very bare bones http server on board, which currently only shows the I/O status, but may be extended quite easily in the future. Use the http interface simply by pointing your browser to your ESP's IP address and add the port number 24 to it.

If the requirements are met, the OTA-version can be used, which means that updates can be programmed over the network instead of using the UART.

IO

All I/O pins can be configured to work as plain digital input, plain digital output, "timer" mode (this means trigger once, either manually or at startup, or toggle continuously) or "pwm" mode (16 bits PWM mode, running at 330 Hz, suitable for driving lighting, maybe servo motors as well, not tested). The ADC input and the RTC GPIO are also supported.

Pins are organised in devices of at most 16 pins. Each pin must be configured to a mode it will be used in. That goes even for pins that can only have one function (e.g. the adc input). If a pin is not configured (i.e. set to mode “disabled”), it won't be used and not even initialised, thereby keeping it's original function (e.g. UART) or remaining floating (HiZ).

Two devices are always present, device 0 = the internal GPIO's 0-15 and device 1 = the other I/O pins: the RTC GPIO and the ADC input. Other devices can be added using I2C I/O expanders. Currently supported are MCP23017 and PCF8574.

UART bridge

The UART pins (TXD/RXD) are available and are bridged to the ESP8266's ip address at port 23, unless they are re-assigned as GPIO pins.

The UART bridge accepts connections on tcp port 23, gets all data from it, sends it to the UART (serial port) and the other way around. This is the way to go to make your non-networking microcontroller WiFi-ready. If you add an RS-232C buffer (something like a MAX232 or similar), you can even make your non-networking peripherals like printers etc. available over the wireless lan.

The UART driver is heavily optimised and is completely interrupt driven, which makes it very efficient.

I2C

The ESP8266 does not have a hardware I2C module (as opposed to most microcontrollers), so the protocol needs to be implemented using a bit-banging software emulation. Espressif supplies code that does exactly that, but it's rubbish. So I wrote my own protocol handler from scratch and it already proved to be quite robust.

All off the internal GPIO pins can be selected to work as I2C/SMBus pins (SDA+SCL). You can use the "raw" I2C send and receive commands to send/receive arbitrary commands/data to arbitrary slaves. For a number of I2C sensors there is built-in support, which allows you to read them out directly, where a temperature etc. is given as result.

External displays

Currently the SAA1064 is supported, it's a 4x7 led display multiplexer, controlled over I2C. Recently added is LCD text displays using the well-known Hitachi HD44780 LCD controller. See the command reference, display section, on how to connect these display and how to configure them.

Support for Orbital Matrix I2C-controlled VFD/LCD screens is planned.

The system consists of multiple "slots" of messages that will be shown in succession. You can set a timeout on a message and it will be deleted automatically after that time. If no slots are left, it will show the current time (from RTC). Use 0 as timeout to not auto-expire slots.

GETTING STARTED

General

You can either download the software and flash it ("precompiled image") from GitHub or compile the software yourself. In both cases you will need to use a suitable flashing device (CP210x or similar USB to UART converter) and a tool for flashing. I am using esptool.py for that and the Makefile also expects it to be present. It's not required though, you can use any flashing tool and do the flashing manually. The flashing process itself has been described at numerous places, I am not going to repeat it here.

Image types

There are two types of images:

name	type	update using	files	flash size requirements
PLAIN	plain / normal	UART flash mechanism	IROM image, IRAM image	4 Mbit
OTA	over the air updating	OTA flash mechanism (network)	rboot, rboot config, image	16 Mbit

The plain images have very little requirements. They will run in less than 256 kbytes, so a 4 Mbit flash chip that's found on most "simple" ESP8266 break-out-boards will suffice.

The over-the-air ("OTA") upgradable image needs 1 Mbyte each because of the address mapping/banking mechanism of the ESP8266 used. This means the usual 4 Mbit flash chip won't be sufficient, you will need a 16 Mbit flash chip at least. Some break-out-boards already have this amount of flash memory, others can be upgraded. I have had success with the Winbond W25Q16DVSSIG and W25Q16DVSNIG. They're almost the same, the first is 208 mil, the second is 150 mil. That is important, because they need to fit on the pads of the PCB. The ESP-201 for instance, requires a 150 mil flash chip, the ESP-01 as well, but newer issues of the ESP-01, which come with 8 Mbits of flash instead of 4 Mbits, actually require a 208 mil IC.

The default image for the Makefile is always OTA. So if you're going to flash, config, etc, a plain image, always include **`IMAGE=plain`** on the make command line or add it in the Makefile. Also, always **`make clean`** if you switch from plain to ota image and v.v.

If you're going to use the pre-compiled images, you can skip the next section "building the software".

Building the software

For building the software you'll need to get and install the opensdk building environment. Get it here: http://github.com/pfalcon/esp-open-sdk. You can use the latest version and you can also make it install the latest sdk from Espressif, there are no known issues there. Change the Makefile to point SDKROOT to the root of the opendsk directory.

The build process also uses the ESPTOOL2 tool from Richard Burton. The Makefile will fetch and build it automatically in a make session if you do `git submodule init` and `git submodule update` first.

Now you can start build the "plain" (as opposed to "ota", "over the air" flash) version. Type `make IMAGE=plain` and wait for completion. The process will yield two files: espiobridge-plain-iram-0x000000.bin and espiobridge-plain-irom-0x010000.bin, just like the precompiled images. They can be flashed as usual.

If you're going to build the OTA image, use `make IMAGE=ota` (or leave out the `IMAGE=ota` part, it's default). The build process uses part of RBOOT by Richard Burton in addition to the ESPTOOL2 tool. If you properly typed the above `git` commands, the submodule will be present and RBOOT will be built automatically during the build process. After the build has finished, you will have (a.o.) these files: espiobridge-rboot-boot.bin: the rboot binary, rboot-config.bin: the rboot configuration (no need to make one yourself), espiobridge-rboot-image: the actual Universal I/O bridge firmware. These files can be flashed like the precompiled OTA images. See further down the page for more detailed description.

There will also be a binary called "otapush" which is compiled with your host compiler. It's the program that needs to be run to push new firmware to the ESP8266. It's tested on Linux, but it's quite simple, so I guess it will work on any more-or-less POSIX-compliant operating system. The protocol is very simple anyway, but uses CRC and MD5 for data protection. There is no risk that any "flipped bit" during transfer or flashing will give erroneous flash images.

Flashing precompiled images or self built images

The "plain" image consists of two files: espiobridge-plain-iram-0x000000.bin and espiobridge-plain-irom-0x010000.bin. These can be flashed to address 0x000000 and address 0x010000 respectively, using your flash tool of choice.

The "ota" image consists of three files:

file	use	flash to address
espiobridge-rboot-boot.bin	the rboot binary	0x000000
rboot-config.bin	the rboot configuration (no need to make one yourself)	0x001000
espiobridge-rboot-image	the actual Universal I/O bridge firmware	0x002000

Or use `make flash` but for that you'll need to have esptool.py installed and you need to adjust ESPTOOL in the Makefile. The `make flash` command will also flash default and blank configuration sectors. They're not used by the I/O bridge though, it's just to make the SDK code happy.

Configuring WLAN

Obviously, the I/O bridge needs to know the WLAN SSID and password to connect. You could configure them using the telnet connection to port 24, but for that you'd need the I/O bridge already connected. Chicken-and-egg...

In the past there have been different methods to configure WLAN parameters from scratch (typing them from the UART at startup, creating a default config and flash it), but none of these were satisfactory.

The current method of "booting up" is using the access point mode, see below for details.

When the software boots, it will first check if there is a valid configuration present. If there is no valid configuration, several configuration options are initialised to their default. For the WLAN SSID and password, they are "esp" and "espespesp" respectively, for both client and access point mode.

Step 2. If no valid configuration is present, assume client mode. Otherwise start up in the mode configured.

Step 3. If in client mode, try to connect. If no connection+IP is obtained within 30 seconds, switch to access point mode.

Summarised this means that if your config is empty or invalid, so it can't connect, just wait half a minute until the default "esp" SSID shows up, connect to it using passwd "espespesp", telnet to address 192.168.4.1 at port 24 and setup the config. Don't forget to write the config!

How to switch between client and access point mode

The default is to start in client mode. Use the "wlan-mode" (wm) command to switch to access point mode or v.v. Use "ap" for access point mode or "client" for client mode. After you entered the wm command, the switch will be made immediately but it will not be written to flash yet, for safety.

After switching to access point mode, wait for the SSID (default: esp) to show up and connect using the passwd (default: espespesp). Your current connection will not be disconnected, you need to do so yourself. You will get an IP address of 192.168.4.2. Use telnet to connect to the ESP8266 at port 24, at ip address 192.168.4.1. The first time you try, you will get a "connection refused" error. That is normal (unfortunately). Just try again and it will work. Now write the config to make the change permanent or e.g. reset to return to client mode (or use the wlan-mode command for that).

After switching to client mode, disconnect, wait for the connection to the access point to establish and then you can use telnet again to log in. If you want to make this mode permanent, write the config.

USING THE I/O BRIDGE

Configuring and using the UART bridge

- Attach your microcontroller's UART lines or RS232C's line driver lines to the ESP8266, I think enough has been written about how to do that.
- Start a telnet session to port 24 of the ip address, type help and <enter>.
- You will now see all commands.
- Use the commands starting with uart to setup the UART. After that, issue the config-write command to save and use the reset command to restart.
- After restart you will have a transparent connection between tcp port 23 and the UART; tcp port 24 always remains available for control.

Configuring and using IO pins

The I/O pins are organised in devices of at most 16 pins. Each pin of each device must be configured to a mode it will be used in. That goes even for pins that can only have one function (e.g. the ADC input). If a pin is not configured (i.e. set to mode “disabled”), it won't be used and not even initialised, thereby keeping it's original function (e.g. UART) or remaining floating (HiZ).

Device 0 always consists of the 16 “normal” built-in GPIO's. All can be used, except for 6 to 11, which is used for the flash memory interface, they aren't allowed to be configured for GPIO. In total eight pins can be configured for PWM operation, which is generically called “analog output” mode, because using a simple RC filter, it can be used as analog output.

Device 1 always consists of the two “special” pins. Pin 0 is the “extra” GPIO, also known as the RTC GPIO. This is a simple I/O pin that can only do digital input and digital output, so PWM is not supported, but timer mode is supported. Pin 1 is the ADC input (analog input). The returned value is normalised to a 16 bit value (0-65535), but the precision won't be 16 bits, expect about 9-10 bits.

Higher numbered devices will represent externally connected I2C I/O expanders, like the MCP23017 (16 I/O pins) and PCF8574 (8 I/O pins). Each pin of these needs to be configured in the same way as the GPIO's from device 0 and device 1 (internal). These devices support digital input, digital output, counter and timer operation (using software emulation on the ESP8266 itself). PWM is not supported.

The following I/O pin modes are supported (depending on each device and pin), using the io-mode command:

I/O function	mode name for im command etc.	function	available on device
disabled	disabled	pin isn't touched	all devices
digital input	inputd	digital two state input	most devices
counter	counter	count (downward) edges on digital input	most devices
digital output	outputd	digital two state output	most devices
timer	timer	trigger digital output once or repeatedly	most devices
analog input	inputa	analog input, value 0 – 65535	ADC pin on device 1 only
analog output	outputa	analog output (or PWM), value 0 – 65535	GPIO pins on device 0 only
i2c	i2c	set pin to i2c sda or scl mode	GPIO pins on device 0 only

For each pin some flags can be (indepently) set or cleared, using the io-set-flag and io-clear-flag commands:

I/O additional feature	flags name for isf and icf etc.	function	available on mode
autostart	autostart	**digital output**: set the output to on after start (otherwise it's set to off)	digital output, timer, analog output
		timer: trigger automatically after start, otherwise wait for explicit trigger	
		analog output: trigger the modulation feature, otherwise wait for explicit trigger.	
repeat	repeat	repeat the trigger after one cycle	timer, analog output
pull-up	pullup	activate internal (weak) pull-up resistors	digital input, counter
reset on read	reset-on-read	reset the counter when it's read	counter

Configuring and using I2C

To use I2C, first configure two GPIO's as sda and scl lines using the **im .. mode i2c sda** and **im .. mode i2c scl <delay>** commands. The <delay> needs to **5** to able to communicate with generic I2C devices, it makes the bus run at just below 100 kHz. If you double the speed of the CPU, increase this value accordingly. Some devices can run at much higher speeds (Fastmode, Fastmode+, etc.), in that case, the delay can be lower and the bus speed will increas, but make sure it's never faster than the slowest device on the bus. Some devices may proof difficult to communicate with. In that case, an additional delay might help.

Now use the raw I2C read and write commands to communicate. Don't forget to set the target slave address first. Most devices accept a register pointer as first data byte and then the values to write to this register. Similarly for a read from a certain register, write one byte (the register pointer) and then read one or more bytes.

The repeated-start-condition feature is not implemented. It has never been proven to be a real requirement, so I left it out.

Besides the well-known GPIOs, it's possible to use GPIO0 and GPIO2 (boot selection) for I2C pins, just as GPIO1 and GPIO3 (normally connected to the UART) using a proper pull-up resistor. This will come handy if you only have access to an ESP-01, that only has these GPIO's available.

The selected GPIO's are set to open drain mode without any pull-up, so you will have to add them yourself. The proper value of the resistors should be calculated but 4.7 kOhm usually works just fine.

Currently supported I2C sensors are:

- digipicco (temperature and humidity)
- lm75 (and compatible sensors, at two different addresses) (temperature)
- ds1621/ds1631/ds1731 (temperature)
- bmp085 (temperature and pressure) (untested for now)
- htu21 (temperature and humdity)
- am2321 (temperature and humidity)
- tsl2550 (light intensity)
- tsl2560 (light intensity)
- bh1750 (light intensity).

Configuring and using displays

During startup, available displays are probed. There is no need to configure them manually. For the SAA1064, It does need a properly functioning I2C bus, though, so make sure it works. Hitaci-type LCD's are controlled over I2C I/O expanders, so they can't be detected automatically, make sure the configuration is active and correct and it will be detected as such.

All displays have 8 slots for messages that can be set using the **`ds`** (display-set) command: Use **`dd`** (display-dump) to show all detected displays, you may want to add a verbosity value (**`0-2`**) to see more detail. Finally the **`db`** (display-bright) command controls the brightness of the display. Valid values are **0** (off), **`1, 2, 3, 4`** (max).

Note the SAA1064 runs at 5V or higher. It cannot be connected to the ESP8266 directly, it must have it's own 5V power supply and may or may not need I2C level shifters. I am using level shifters, but it may not be necessary, YMMV.

COMMAND REFERENCE

General, status and informational commands

command	alternate (long) command	parameters	description
`ccd`	`current-config-dump`		Show currently used config (not saved to flash).
`cd`	`config-dump`		Show config from flash (active after next restart).
`ctp`	`command-tcp-port`	*tcp_port*	Set / show the tcp port the command interface listens to.
`ctt`	`command-tcp-timeout`	*timeout_in_seconds*	Set / show the disconnect timeout on the command interface tcp port (0 = no timeout).
`cw`	`config-write`		Write current config to flash.
`gss`	`gpio-status-set`	*io pin*	Select the io device and pin to trigger when general status changes (currently on reception of a command). Set io and pin to -1 to disable this feature. On trigger, a value of -1 is written to this pin, so generally you'd want to use a timer “down” mode pin for this purpose (so the pin is reset after some time).
`nd`	`ntp-dump`		Show current ntp status (if configured).
`ns`	`ntp-set`	*ntp-server-ip timezone-hours*	Setup ntp server. Allow it to run for a few minutes before the correct time is acquired.
`?`	`help`		Show all commands and usage in brief.
`q`	`quit`		Close the current command connection. If idle for over 30 seconds, the connection is closed anyway.
`r`	`reset`		Reset/reboot the device. Necessary to activate some change (like PWM), after having written the config using cw.
`rs`	`rtc-set`	*hh*:*mm*	Set internal clock, works even without NTP but may run out of sync on the long term. If NTP is used, the internal clock is synchronised every minute.
`s or u`	`set or unset`	*flag_name*	Set or unset generic operation flags. Use set or unset without arguments to get a list of currently supported flags.
`S`	`stats`		Retrieve running statistics.
`wac`	`wlan-ap-configure`	*ssid passwd channel*	Reconfigure WLAN identification for access point mode. Write config and reboot to effectuate.
`wcc`	`wlan-client-configure`	*ssid passwd*	Reconfigure WLAN identification for client mode. Write config and reboot to effectuate.
`ws`	`wlan-scan`		Scan for SSID's. Issue this command first, then wait a few seconds and then issue wlan-list.
`wl`	`wlan-list`		List SSID's found with wlan-scan.
`wm`	`wlan-mode`	`client|ap`	Set wlan mode. This command will be effective immediately. See text for details.

I/O configuration and related commands

command	alternate (long) command / submode	parameters	description
`im`	`io-mode`	*io_device io_pin mode mode_parameters*	Configure pin mode.

	mode=**inputd**	**inputd**	Set pin to digital input mode.
	mode=**counter**	**counter** ***debounce***	Set pin to digital input mode, count downward transitions. Set debounce in milliseconds.
	mode=**outputd**	**outputd**	Set pin to digital output mode.
	mode=**timer**	**timer** ***direction delay***	Set pin to timer mode. Direction is up or down and specifies the default state and triggered state (down: default up, triggered down, up: default down, triggerend up). Delay is time before state is set to default after trigger.
	mode=**inputa**		Set pin to analog input mode.
	mode=**outputa (1)**		Select analog output (or PWM). Default value is 0.
	mode=**outputa (2)**	***default_value***	Select analog output (or PWM). Default value set to argument (0-65535).
	mode=**outputa (3)**	***lower_bound upper_bound delay***	Select analog output (or PWM). After (auto-)trigger start to modulate between lower_bound and upper_bound at specified speed (ms / 10000).
	mode=**i2c**	**sda**	Configure built-in GPIO for I2C/sda use.
		scl ***bus_delay***	Configure built-in GPIO for I2C/scl use. Set bus_delay to 5 for standard 100 khz bus, lower value is higher bus speed and v.v.
	mode=lcd	**rs** ***io pin***	Set I/O and pin for R/S signaling. Can be any digital capable pin. Required.
		rw ***io pin***	Set I/O and pin for R/W signaling. Can be any digital capable pin. Not required, is only always set to low (write).
		e ***io pin***	Set I/O and pin for E signaling. Can be any digital capable pin. Required.
		d0 ***io pin***	Set I/O and pin for the D0 data line. Can be any digital capable pin. Required for 8 bit operation, not required for 4 bit operation.
		d1 ***io pin***	Set I/O and pin for the D1 data line. Can be any digital capable pin. Required for 8 bit operation, not required for 4 bit operation.
		d2 ***io pin***	Set I/O and pin for the D2 data line. Can be any digital capable pin. Required for 8 bit operation, not required for 4 bit operation.
		d3 ***io pin***	Set I/O and pin for the D3 data line. Can be any digital capable pin. Required for 8 bit operation, not required for 4 bit operation.
		d4 ***io pin***	Set I/O and pin for the D4 data line.

			Can be any digital capable pin. Required for both 4 bit and 8 bit operation.
		d5 ***io pin***	Set I/O and pin for the D5 data line. Can be any digital capable pin. Required for both 4 bit and 8 bit operation.
		d6 ***io pin***	Set I/O and pin for the D6 data line. Can be any digital capable pin. Required for both 4 bit and 8 bit operation.
		d7 ***io pin***	Set I/O and pin for the D7 data line. Can be any digital capable pin. Required for both 4 bit and 8 bit operation.
		bl ***io pin***	Set I/O and pin for controlling the backlight. Can be an analog output for dimming or a digital output (on/off only). Optional.
ir	**io-read**	***io_device io_pin***	Read pin. Source of the value depends on the mode of the pin. Counter e.g. returns the current counter value. Outputs can also be read, they return the actual pin state, not that of the output latch.
iw	**io-write**	***io_device io_pin value***	Write pin. Semantics differ for various pin modes.
	mode=**inputd**		Not implemented.
	mode=**counter**		Write counter value directly.
	mode=**outputd**		Write value. 0 is off, any other value is on
	mode=**timer**		Control counter output. 0 is set pin to default, other value is trigger timer (set to active and start timer).
	mode=**inputa**		Not implemented.
	mode=**outputa**		Write analog output. If -1 is given and the this is a auto-modulating analog output (see mode=outputa (3)), start modulating. Otherwise just set the current value of the output.
	mode=**i2c**		Not implemented.
isf or **icf**	**io-set-flag or io-clear_flag**	***io_device io_pin flag***	Set or clear flags for a pin.
	flag=**autostart**		Automatically start this pin at start. For a digital output pin this means set to “on” instead of default “off”. For a timer or a modulated analog output this means trigger automatically at start. Otherwise these types of pins need to be triggered automatically by writing to them.
	flag=**repeat**		For a timer or a modulated analog output this mean repeat after the

			first cycle. Otherwise stop after one cycle.
	flag=**pullup**		For a digital input or counter, activate the built-in weak pull-up.
	flag=**reset_on_read**		For a timer, when it's read, also reset it automatically.

UART configuration and related commands

command	alternate (long) command	parameters	description
btp	**bridge-tcp-port**	***tcp_port***	Set / show the tcp port the UART is listening to.
btt	**bridge-tcp-timeout**	***timeout_in_seconds***	Set / show the UART tcp port disconnect timeout (0 = no timeout)
ub	**uart-baud**	***baud_rate***	Set baud rate.
ud	**uart-data**	***data_bits***	Set data bits (6, 7 or 8).
us	**uart-stop**	***stop_bits***	Set stop bits (1 or 2).
up	**uart-parity**	***parity***	Set parity (none/even/odd).

I2C related commands

command	alternate (long) command	parameters	description
i2a	**i2c-address**	***address***	Set I2C slave's address to use for other commands. Specify hex number from 0 to 7a. Don't include the r/w bit and don't include 0x.
i2r	**i2c-read**	***number_of_bytes_to_read***	Read this amount of bytes from the current I2C slave.
i2w	**i2c-write**	***byte_to_write, ...***	Write these bytes to the current I2C slave. Bytes can be specified in either decimal or hexadecimal, in which case they should be prefixed with 0x.
i2rst	**i2c-reset**		Reset the I2C bus. This also attempts to convince “stuck” slaves to release the bus, it may or may not work.

I2C sensor related commands

command	alternate (long) command	parameters	description
isd	**i2c-sensor-dump**	***verbosity***	List all detected sensors. Use the id's from this list to query the value (using isr). If verbosity is 0 or missing, only show found sensors. If it's 1, show all known sensors. If it's 2, show all known sensors including verbose error reporting.
isr	**i2c-sensor-read**	***sensor_id***	Read a sensor.
isc	**i2c-sensor-calibrate**	***sensor_id factor offset***	Calibrate a sensor. Specify a value (float) to multiply the value by (**factor**) and a value to be added (or subtracted...) from the value (**offset**). The calibration is saved in flash together with the config using config-write. If the parameters are left out, list all current calibrations.

Display control related commands

command	alternate (long) command	parameters	description
db	display-brightness	***brightness_value***	Set / show display brightness. 0 = off, 1 = 25%,

command	alternate (long) command	parameters	description
			2 = 50%, 3 = 75%, 4 = 100%.
dd	display-dump		Show all available information on the connected display.
ddm	display-default-message	*message*	Set the default message, i.e. the message that is shown when the user hasn't set any slots (yet).
dft	display-flip-timeout	*timeout*	Set / show the timeout in seconds between slot “flips”, i.e. how long each message slot is shown.
ds	display-set	*slot timeout tag text*	Set a message in a slot. After timeout seconds, the slot is released or use 0 to have a message linger forever. The tag is show on row 0 of the display, along with the current time. Use “-” as the tag to skip the tag completely and use all four rows for your message. Then finally the message which can have spaces and newlines inside. Use %%%% as message content to have the default message (= display type + time).

OTA commands

Note: these commands are not meant to be used directly. They're meant to be used in concerto with a host-side application that uploads a new image. See otapush.c for an example.

command	alternate (long) command	parameters	description
ow	ota-write	*image_length*	Start a firmware upload cycle. Specify the total amount of bytes that will be sent.
ov	ota-verify	*image_length*	Start a firmware verify cycle. Specify the total amount of bytes that will be sent.
os	ota-send	*chunk_length crc bytes...*	Send the next chunk. A chunk should be 256, 512 or 1024 bytes in size. If a transer using 1024 bytes doesn't succeed, try lower values. Also send the CRC32 of the chunk and then the data as raw binary content. Don't add newlines. Repeat until all data is sent or call ota-finish to either finish (when done) or cancel (when not complete). Every time a complete flash sector has been received, it will be written to flash and verified to both the received data and the received CRC32. If the chunk went well, the word “ACK” followed by the chunk ordinal will be replied.
of	ota-finish	*md5_sum*	Finish up. Send the md5_sum over the complete file as hex string. If everything went well, the string “VERIFY_OK” or “WRITE_OK” will be replied (depending on the requested operation).
oc	ota-commit		If everything went well, toggle the current boot bank and reset. Replies “OTA commit slot” + the new slot number if all went well.

See here for latest news and join the discussion: http://www.esp8266.com/viewtopic.php?t=3959, old topic: http://www.esp8266.com/viewtopic.php?&t=3212.