Boot Mode Selection

This guide explains how to select the boot mode correctly and describes the boot log messages of ESP32.

Warning

The ESP32 has a 45k ohm internal pull-up/pull-down resistor at GPIO0 (and other pins). If you want to connect a switch button to enter the boot mode, this has to be a strong pull-down. For example a 10k resistor to GND.

Information about ESP32 strapping pins can also be found in the ESP32 Datasheet, section "Strapping Pins".

On many development boards with built-in USB/Serial, `esptool.py` can automatically reset the board into bootloader mode. For other configurations or custom hardware, you will need to check the orientation of some "strapping pins" to get the correct boot mode:

Select Bootloader Mode

GPIO0

The ESP32 will enter the serial bootloader when GPIO0 is held low on reset. Otherwise it will run the program in flash.

GPIO0 Input	Mode
Low/GND	ROM serial bootloader for esptool
High/VCC	Normal execution mode

GPIO0 has an internal pullup resistor, so if it is left unconnected then it will pull high.

Many boards use a button marked "Flash" (or "BOOT" on some Espressif development boards) that pulls GPIO0 low when pressed.

GPIO2

GPIO2 must also be either left unconnected/floating, or driven Low, in order to enter the serial bootloader.

In normal boot mode (GPIO0 high), GPIO2 is ignored.

Other Pins

As well as GPIO0 and GPIO2, the following pins influence the serial bootloader mode:

GPIO	Meaning
12 (MTDI)	If driven High, flash voltage (VDD_SDIO) is 1.8V not default 3.3V. Has internal pull-down, s
15 (MTDO)	If driven Low, silences boot messages printed by the ROM bootloader. Has an internal pull-

For more information, consult the ESP32 Datasheet, section "Strapping Pins".

Automatic Bootloader

`esptool.py` resets ESP32 automatically by asserting `DTR` and `RTS` control lines of the USB to serial converter chip, i.e., FTDI, CP210x, or CH340x. The `DTR` and `RTS` control lines are in turn connected to `GPIO0` and `EN` (`CHIP_PU`) pins of ESP32, thus changes in the voltage levels of `DTR` and `RTS` will boot the ESP32 into Firmware Download mode.

Note

When developing `esptool.py`, keep in mind `DTR` and `RTS` are active low signals, i.e., `True` = pin @ 0V, `False` = pin @ VCC.

As an example of auto-reset curcuitry implementation, check the schematic of the ESP32 DevKitC development board:

- The **Micro USB 5V & USB-UART** section shows the `DTR` and `RTS` control lines of the USB to serial converter chip connected to `GPIO0` and `EN` pins of the ESP module.
- Some OS and/or drivers may activate `RTS` and or `DTR` automatically when opening the serial port (true only for some serial terminal programs, not `esptool.py`), pulling them low together and holding the ESP in reset. If `RTS` is wired directly to `EN` then RTS/CTS "hardware flow control" needs to be disabled in the serial program to avoid this. An additional circuitry is implemented in order to avoid this problem - if both `RTS` and `DTR` are asserted together, this doesn't reset the chip. The schematic shows this specific circuit with two transistors and its truth table.
- If this circuitry is implemented (all Espressif boards have it), adding a capacitor between the `EN` pin and `GND` (in the 1uF-10uF range) is necessary for the reset circuitry to work reliably. This is shown in the **ESP32 Module** section of the schematic.
- The **Switch Button** section shows buttons needed for manually switching to bootloader.

Make the following connections for `esptool` to automatically enter the bootloader of an ESP32 chip:

ESP Pin	Serial Pin

ESP Pin	Serial Pin
EN	RTS
GPIO0	DTR

In Linux serial ports by default will assert RTS when nothing is attached to them. This can hold the ESP32 in a reset loop which may cause some serial adapters to subsequently reset loop. This functionality can be disabled by disabling `HUPCL` (ie `sudo stty -F /dev/ttyUSB0 -hupcl`).

(Some third party ESP32 development boards use an automatic reset circuit for `EN` & `GPIO0` pins, but don't add a capacitor on the `EN` pin. This results in unreliable automatic reset, especially on Windows. Adding a 1uF (or higher) value capacitor between `EN` pin and `GND` may make automatic reset more reliable.)

In general, you should have no problems with the official Espressif development boards. However, `esptool.py` is not able to reset your hardware automatically in the following cases:

- Your hardware does not have the `DTR` and `RTS` lines connected to `GPIO0` and `EN` (`CHIP_PU`)
- The `DTR` and `RTS` lines are configured differently
- There are no such serial control lines at all

Manual Bootloader

Depending on the kind of hardware you have, it may also be possible to manually put your ESP32 board into Firmware Download mode (reset).

- For development boards produced by Espressif, this information can be found in the respective getting started guides or user guides. For example, to manually reset a development board, hold down the **Boot** button (`GPIO0`) and press the **EN** button (`EN` (`CHIP_PU`)).
- For other types of hardware, try pulling `GPIO0` down.

Boot Log

Boot Mode Message

After reset, the second line printed by the ESP32 ROM (at 115200bps) is a reset & boot mode message:

```
ets Jun  8 2016 00:22:57
rst:0x1 (POWERON_RESET),boot:0x3 (DOWNLOAD_BOOT(UART0/UART1/SDIO_REI_REO_V2))
```

`rst:0xNN (REASON)` is an enumerated value (and description) of the reason for the reset. A mapping between the hex value and each reason can be found in the ESP-IDF source under RESET_REASON enum. The value can be read in ESP32 code via the get_reset_reason() ROM

function.

`boot:0xNN (DESCRIPTION)` is the hex value of the strapping pins, as represented in the GPIO_STRAP register.

The individual bit values are as follows:

- `0x01` - GPIO5
- `0x02` - MTDO (GPIO15)
- `0x04` - GPIO4
- `0x08` - GPIO2
- `0x10` - GPIO0
- `0x20` - MTDI (GPIO12)

If the pin was high on reset, the bit value will be set. If it was low on reset, the bit will be cleared.

A number of boot mode strings can be shown depending on which bits are set:

- `DOWNLOAD_BOOT(UART0/UART1/SDIO_REI_REO_V2)` or `DOWNLOAD(USB/UART0)` - ESP32 is in download flashing mode (suitable for esptool)
- `SPI_FAST_FLASH_BOOT` - This is the normal SPI flash boot mode.
- Other modes (including `SPI_FLASH_BOOT`, `SDIO_REI_FEO_V1_BOOT`, `ATE_BOOT`) may be shown here. This indicates an unsupported boot mode has been selected. Consult the strapping pins shown above (in most cases, one of these modes is selected if GPIO2 has been pulled high when GPIO0 is low).

Note

`GPIO_STRAP` register includes GPIO 4 but this pin is not used by any supported boot mode and be set either high or low for all supported boot modes.

Later Boot Messages

Later output from the ROM bootloader depends on the strapping pins and the boot mode. Some common output includes:

Early Flash Read Error

```
flash read err, 1000
```

This fatal error indicates that the bootloader tried to read the software bootloader header at address 0x1000 but failed to read valid data. Possible reasons for this include:

- There isn't actually a bootloader at offset 0x1000 (maybe the bootloader was flashed to the

wrong offset by mistake, or the flash has been erased and no bootloader has been flashed yet.)
- Physical problem with the connection to the flash chip, or flash chip power.
- Flash encryption is enabled but the bootloader is plaintext. Alternatively, flash encryption is disabled but the bootloader is encrypted ciphertext.
- Boot mode accidentally set to `HSPI_FLASH_BOOT`, which uses different SPI flash pins. Check GPIO2 (see above).
- VDDSDIO has been enabled at 1.8V (due to MTDI/GPIO12, see above), but this flash chip requires 3.3V so it's browning out.

Software Bootloader Header Info

```
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
```

This is normal boot output based on a combination of eFuse values and information read from the bootloader header at flash offset 0x1000:

- `configsip: N` indicates SPI flash config:
- 0 for default SPI flash
- 1 if booting from the HSPI bus (due to eFuse configuration)
- Any other value indicates that SPI flash pins have been remapped via eFuse (the value is the value read from eFuse, consult espefuse docs to get an easier to read representation of these pin mappings).
- `SPIWP:0xNN` indicates a custom `WP` pin value, which is stored in the bootloader header. This pin value is only used if SPI flash pins have been remapped via eFuse (as shown in the `configsip` value). All custom pin values but WP are encoded in the configsip byte loaded from eFuse, and WP is supplied in the bootloader header.
- `clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00` Custom GPIO drive strength values for SPI flash pins. These are read from the bootloader header in flash. Not currently supported.
- `mode: AAA, clock div: N`. SPI flash access mode. Read from the bootloader header, correspond to the `--flash_mode` and `--flash_freq` arguments supplied to `esptool.py write_flash` or `esptool.py elf2image`.
- `mode` can be DIO, DOUT, QIO, or QOUT. *QIO and QOUT are not supported here*, to boot in a Quad I/O mode the ROM bootloader should load the software bootloader in a Dual I/O mode and then the ESP-IDF software bootloader enables Quad I/O based on the detected flash chip mode.
- `clock div: N` is the SPI flash clock frequency divider. This is an integer clock divider value from an 80MHz APB clock, based on the supplied `--flash_freq` argument (ie 80MHz=1, 40MHz=2, etc). The ROM bootloader actually loads the software bootloader at a lower frequency than the flash_freq value. The initial APB clock frequency is equal to the crystal frequency, so with a 40MHz crystal the SPI clock used to load the software bootloader will be half the configured value (40MHz/2=20MHz). When the software bootloader starts it sets the APB clock to 80MHz causing the SPI clock frequency to match the value set when flashing.

Software Bootloader Load Segments

```
load:0x3fff0008,len:8
load:0x3fff0010,len:3680
load:0x40078000,len:8364
load:0x40080000,len:252
entry 0x40080034
```

These entries are printed as the ROM bootloader loads each segment in the software bootloader image. The load address and length of each segment is printed.

You can compare these values to the software bootloader image by running `esptool.py --chip esp32 image_info /path/to/bootloader.bin` to dump image info including a summary of each segment. Corresponding details will also be found in the bootloader ELF file headers.

If there is a problem with the SPI flash chip addressing mode, the values printed by the bootloader here may be corrupted.

The final line shows the entry point address of the software bootloader, where the ROM bootloader will call as it hands over control.