The esp_tool Utility Program

Introduction

The esp_tool utility program is a command line utility written in C++ for interacting with an ESP8266 device (e.g. downloading code to it) and for processing ELF files (e.g. to extract sections from it for later downloading). The utility is compatible with Windows systems (XP and later) and Linux systems. The source code for the utility is provided along with a Windows (nmake) makefile and a Linux makefile. The code can probably be compiled for OS X systems but the author has not attempted to do so.

The code for esp_tool was inspired by Fredrik Ahlberg's esptool.py (https://github.com/themadinventor/esptool) but the author wanted a utility that could be run in the absence of a Python interpreter. Although the Python code could have been packaged into an executable program using py2exe the author didn't want to deal with the issues of redistributing the Microsoft C runtime DLL neede for that conversion. The code for esp_tool was also inspired by Christian Kippel's esptool-ck (https://github.com/igrr/esptool-ck) but that application is written in C whereas the esp_tool code is written in C++.

The esp_tool utility contains ideas not present in either of the abovementioned utilities at the time this was written, e.g. sparse combined image files and the ability to extract the Flash address from image filenames. Using a sparse combined image file can substantially reduce the time required to download an image to an ESP8266 due to avoiding the time needed to transmit all of the pad bytes representing the space between load image components. Further, a sparse combined image file simplifies downloading by having all of the image information in one file that loads to a known address. The ability to extract Flash load addresses from image filename simplifies the command options for loading multiple images and avoids possible errors in specifying the load address.

Invocation

The syntax for invoking esp_tool is as follows:

`esp_tool [<options>] [<operations>] [<files>]...`

where <*options*> is one or more of the options listed in the first table below, <*operations*> is one or more of the operations listed in the second table below and <*files*> represents the names of one or more files on which to operate. The square brackets surrounding each of these elements indicates that each is optional. Invoking esp_tool with no options results in the output of a brief summary of the invocation syntax.

When running on Windows systems, the <files> parameter on the command line may include filename specifications that incorporate the normal Windows wildcard characters * and ?. On Linux-based systems, the normal Linux filename wildcards may be used.

The command line options and operation flags are summarized in separate tables below and then described in detail.

esp_tool Options

Long Form	Short Form	Description
`--address=<addr>`	`-a<addr>`	Specify the address for later operations.
`--baud=<speed>`	`-b<speed>`	Specify the baud rate to use.
`--elf-file=<file>`	`-e<file>`	Specify an ELF file for later operations.
`--exit=<code>`	`-e<code>`	Specify the character code to exit monitor mode.
`--flash-freq=<freq>`	`-fs<freq>`	Specify the Flash clock speed.
`--flash-mode=<mode>`	`-fm<mode>`	Specify the Flash interface mode.
`--flash-parm=<value>`	`-fp<value>`	Specify the combine Flash parameters.
`--flash-size=<size>`	`-fs<size>`	Specify the Flash size.
`--log=<file>`	`-l<file>`	Specify a log file for device output.
`--monitor[=<speed>]`	`-m[<speed>]`	Request monitor mode after operations.

`--no-run`	`-r0`	Do not run the device after operations.
`--port=<port>`	`-p<port>`	Specify the serial port to use.
`--quiet`	`-q`	Suppress progress output.
`--reset=<mode>`	`-r<mode>`	Specify the reset protocol.
`--run`	`-r1`	Run the device after completing operations.
`--size=<size>`	`-s<size>`	Specify the size for later operations.

esp_tool Operation Flags

Long Form	Short Form	Description
`--dump-mem`	`-od`	Copy a memory region from the device to a file.
`--elf-info`	`-os`	Output section information from an ELF file.
`--erase-flash[=<size>]`	`-oe[<size>]`	Erase all or a portion of Flash.
`--extract[=<image>]`	`-ox[<image>]`	Extract sections from an ELF file, create images.
`--flash-id`	`-of`	Read Flash identification information.
`--image-info`	`-oi`	Output information from an ELF file.
`--padded=<file>`	`-cp<file>`	Create a padded combined image file.
`--padded+=<file>`	`-cp+<file>`	Append to an existing padded combined image file.
`--read-flash`	`-or`	Copy a Flash region from the device to a file.
`--read-mac`	`-om`	Read the MAC addresses of the device.
`--section=<section>`	`-os<section>`	Extract data from sections of an ELF file.
`--sparse=<file>`	`-cs<file>`	Create a sparse combined image file.
`--sparse+=<file>`	`-cs+<file>`	Append to an existing sparse combined image file.
`--write-flash`	`-ow`	Write one or more files to Flash.

Command Line Parameters in Detail

In the following discussion, numeric values referred to may be specified in decimal or, with a prefix (0x or 0X), in hexadecimal. In either case, the value may be immediately followed by the letter K (upper or lower case) to indicate a multiplier of 1024.

`--address=<addr>` **`-a<addr>`**

This option specifies an address for a later operation. Example: `--address=0x40000`

`--baud=<speed>` **`-b<speed>`**

This option specifies the baud rate for the serial port connected to the device. The baud rate value may be any baud rate supported by the serial port. The default speed is 115,200 baud. Example: `--baud=230400`

`--dump-mem` **`-od`**

This operation flag requests that a region of memory of the device be read and its content output to the subsequently named file. The address at which to begin reading is given by the `--address` or `-a` option and the number of bytes to read is given by the `--size` or `-s` option.

`--elf-file=<elf>` **`-e<elf>`**

This option specifies an ELF format file to be processed later. See the `--elf-info`, `--section` and `--extract` operation flags.

--elf-info **-os**

This operation flag causes section information from a previously specified ELF file to be output to stdout. In addition to the section names, load address and size information for each section is also output.

--erase-flash*[=<size>]* **-oe*[<size>]***

This operation flag specifies that all of or part of Flash memory on the device should be erased. If the optional size modifier is not present the entire Flash chip is erased. If the size modifier is given, Flash memory beginning at a previously specified address is erased. The address given is rounded down to a 1K block boundary and the given size is rounded up to an integral multiple of 1K. Example: `--erase=0x1000`

--exit=<*code*> **-x<*code*>**

This option specifies the numeric value of the character code that will cause an exit from the device monitoring function (see `--monitor`). Example: `--exit=0x0c`

--extract*[=<image>]* **-ox*[<image>]***

This operation flag requests that image files be created by extracting certain sections from a previously named ELF file. The image files created will have names based on the ELF filename but with the extension removed and a load address indicator and the extension `.bin` added. As an example, if the ELF file is named `myApp.elf`, the two generated image files will be like `myApp_0x00000.bin` and `myApp_0x40000.bin`.

The first image created will contain the `.text`, `.data` and `.rodata` sections from the ELF file and will have the form of a standard boot image with a Flash address of 0x00000 and will incorporate the Flash parameters. The second image created will be a raw binary image containing the `.irom0.text` section from the ELF file. The Flash address of the second section is typically `0x40000` but it may be different depending on how the application was built.

If one of the combining options was given prior to the `--extract` operation flag, the generated image files are then combined as requested by the combining option. In this case, the optional <*image*> parameter may be provided to specify an additional image file that will be combined with the extracted images.

--flash-freq=<*freq*> **-ff<*freq*>**

This option specifies the clock frequency of the Flash chip on the device. The frequency is specified using one of the designators `20M`, `26M`, `40M` or `80M` (case insensitive). The default Flash frequency is `40M`, representing 40MHz. Example: `--flash-freq=26M`

--flash-id **-of**

This operation flag causes Flash chip identification information to be read from the device and output to stdout.

--flash-mode=<*mode*> **-fm<*mode*>**

This option specifies the interface mode of the Flash chip on the device. The interface mode is specified using one of the designators `QIO`, `QOUT`, `DIO` or `DOUT` (case insensitive). The default Flash interface mode is `QIO`. Example: `--flash-mode=DOUT`

`--flash-parm=<val>` **`-fp<val>`**

This option specifies the parameters of the Flash chip as a 16-bit composite value as follows:

Flash Parameter Value Components

Bits	Description	Values
15-12	Flash size	0 = 512KB, 1 = 256KB, 2 = 1MB, 3 = 2MB, 4 = 4MB, 5 = 8MB, 6 = 16MB, 7 = 32MB
11-8	Flash frequency	0 = 40MHz, 1 = 26MHz, 2 = 20MHz, 3 = 80MHz
7-4	not used	
3-0	Flash mode	0 = QIO, 1 = QOUT, 2 = DIO, 3 = DOUT

The default Flash parameter value is `0x0000`. Example: `--flash-parm=0x3000`

`--flash-size=<size>` **`-fs<size>`**

This option specifies the size of the Flash chip on the device. The size is specified using one of the designators `512K`, `1M`, `2M`, `4M`, `8M`, `16M` or `32M` (case insensitive) where the suffix letter connotes kilobytes or megabytes. The default Flash size is `512K`, representing 512K bytes. Example: `--flash-size=2M`

`--help` **`-h`**

This operation flag causes a summary of the command line parameters to be output stdout and then the program exits.

`--image-info` **`-oi`**

This operation flag causes information about a subsequently named standard boot image file or combined image file to be output to stdout. The combined image may be either a padded image or a sparse image.

`--log=<file>` **`-l<file>`**

This option specifies a file to which device output will be written if the utility enters monitor mode (see `--monitor`). The log file will only be created if monitor mode is entered and if the named file exists it will be silently overwritten.

`--monitor[=<speed>]` **`-m[<speed>]`**

This option requests that monitor mode be entered after all operations are complete. In monitor mode, characters typed on the keyboard will be sent to the device and characters received from the device will be sent to stdout. If requested, characters received will also be written to a log file (see `--log`).

If the optional speed is specified, the serial port will be changed to that speed upon entering monitor mode. Otherwise, the default speed or that previously specified with the `--baud` or `-b` option will be used.

`--no-run` **`-r0`**

This option requests that the device not be caused to run after completing all operations.

--padded=<*file*> -cp<*file*>

This operation flag specifies that subsequently named image files should be combined and written to the named file with padding added to align each image with its starting address. The padding consists of bytes with the value 0xff. If the named file exists, it is silently overwritten. Example: --padded=appImage.bin

--padded+=<*file*> -cp+<*file*>

This operation flag specifies that subsequently named image files should be combined and appended to the named file with padding added to align each image with its starting address. If the named file does not exist, an error is reported. Example: --padded+=appImage.bin

--port=<*port*> -p<*port*>

This option specifies the serial port to use for communication with the device. On Windows platforms, the <*port*> is specified by a numeric value (e.g. 2) or by the designator `COM`*n* where n is a numeric value. On Linux platforms the <*port*> is specified using the device name, e.g. `/dev/ttyS1`. The default is `COM1` on Windows platforms and `/dev/ttyS0` on Linux platforms.

--quiet -q

This option request quiet operation mode in which information about operation progress is suppressed.

--read -or

This operation flag specifies that Flash memory on the device beginning at the previously indicated address and continuing for the previously specified size should be read and the content written to the subsequently named file.

--read-mac -om

This operation flag causes the station and AP MAC addresses of the device be read and output to stdout.

--reset=<*mode*> -r<*mode*>

This option specifies a reset protocol to use to prepare the device for communication. The supported reset protocols are given in the table below, the protocol names are case insensitive. The default is not to use a reset protocol.

Reset Protocols

Name	Description
`none`	No protocol employed, manual reset required.
`auto`	DTR capacitively coupled to RST, RTS controls GPIO0.
`dtrOnly`	DTR capacitively coupled to RST and controls GPIO0 directly.
`ck`	RTS controls RST, DTR controls GPIO0.
`wifio`	DTR capacitively coupled to RST, TxD controls GPIO0 via a PNP transistor.
`nodeMCU`	DTR and RTS control RST and GPIO0 via cross-coupled NPN transistors.

Example circuits for each of the reset protocols is provided later in this document.

--run -r1

This option requests that the device be caused to run after completing all operations. This is the default behavior.

--sections=<*sect*>
--section=<*sect*> -os<*sect*>

This operation flag specifies one or more names of sections in a separately named ELF file to be extracted to an image file subsequently named. If multiple section names are given they are separated by commas. Note that no spaces are allowed in the list of section names. Example: `--section=.text,.data,.rodata`

--size=<*size*> -s<*size*>

This option specifies a size for a later operation. The size may be specified as a decimal value or, with a prefix (`0x` or `0X`), as a hexadecimal value. Example: `--size=32768`

--sparse=<*file*> -cs<*file*>

This operation flag specifies that subsequently named image files should be combined and written to the named file in the form of a sparse image, meaning that the empty space between images is not explicitly represented in the combined image. If the named file exists, it is silently overwritten. Although not required, it is recommended that a special extension such as `.esp` be used for sparse image files. Example: --sparse=appImage.esp

--sparse+=<*file*> -cs+<*file*>

This operation flag specifies that subsequently named image files should be combined and appended to the named file which is expected to be a previously created sparse combined image.
Example: --sparse+=appImage.esp

--write -ow

This operation flag specifies that subsequently named image files should be written to Flash on the device. Generally speaking, for each image file named, the Flash address to which to write the image should be specified before the filename (but see the later discussion on automatic address extraction). Since the default address is zero, if the first file to be written to Flash should be located at address zero it isn't strictly necessary to specify the load address explicitly in that particular case. This is the default operation so if no other operations are performed it isn't necessary to specify the write operation.

Command Line Arguments from the Environment

When `esp_tool` begins running it checks for an environment variable named `ESP_TOOL`. If found, the string value of that variable is processed as command line parameters. This is useful, for example, to set certain options that will often be needed, e.g. the serial port and speed.

Normally, the command line parameters in the environment variable value will be delimited by whitespace (space or tab characters). If, for some reason, you want have space or tab characters in a parameter you must surround the entire sequence with a matching pair of single quote or double quote characters so that the parameter begins and ends with the quote character.

Filenames on the Command Line

Some of the operations require a name of one or more files on which to perform the operation. Depending on the operation, the named file may be created (silently overwriting an existing file of the same name) or the named file may be opened for reading or appending (in which case it an error for the file not to exist).

For image files to be written to Flash and for image files to be combined, the load address for the image may be extracted from the filename. For example, you might have an image file named `app_0x00000.bin`. If you add the `@` prefix to the name on the command line, esp_tool will examine the filename looking for a `0x` prefix (case

insensitive) and then extract the hexadecimal value that follows as the load address for that image. Note that the @ prefix is stripped from the filename before further processing. The drawback to this is, of course, that esp_tool can't handle filenames that begin with an at sign.

Combined Image Files

It is often useful to utilize combined image files to avoid the necessity of specifying the load address and filename for multiple images. The combined image files that can be produced by and processed by esp_tool are of two types. The first type is the padded combined image that is common to other similar tools. When building a padded combined image that encapsulates two or more images, the "empty" space between image components if filled with 0xff bytes. Although this type of combined image file serves the purpose of simplifying downloading, it can have the drawback of requiring much longer to Flash the image to the device than would be the case if the individual images were downloaded separately.

The second type of combined image file that is, at this writing, unique to esp_tool is the sparse combined image file. The sparse combined image file realizes the advantages of a single image file with the faster Flash time of using multiple separate image files. A sparse combined image file consists of a four-byte header followed by one or more data segments. The header has the content described in the table below.

Sparse Combined Image File Header

Offset	Length	Description
0	3	Sparse image signature: 'e', 's', 'p'
3	1	Number of image data segments contained in the combined image.

Each image data segment, representing the content of a constituent image file, consists of an 8-byte image data header followed by the binary data from the image file padded with zero bytes if necessary to be an integral multiple of 4 bytes in length. The image data header consists of the image load address followed by the (padded) image length, both four bytes in little endian form.

Examples of Use

esp_tool --elf-file=myApp.elf --extract

This command extracts standard sections from the ELF file myApp.elf and produces two image files, like myApp_0x00000.bin and myApp_0x40000.bin.

esp_tool --elf-file=myApp.elf --padded=myApp.bin --extract

This command produces two image files as described in the preceding example and then combines them with padding to produce the image file myApp.bin.

esp_tool --elf-file=myApp.elf --sparse=myApp.esp --extract

This command is like the immediately preceding example except that it produces a sparse combined image.

esp_tool --port=COM2 myApp_0x00000.bin -a0x4000 myApp_0x40000.bin

This command attempts to connect to the device over serial port COM2 and, if successful, downloads the two named files to the device. Note that it wasn't necessary to specify the address for the first image file because the address defaults to 0. Also note that since no reset protocol was specified a manual reset with GPIO0 grounded would be necessary so that the connection could be established.

```
esp_tool --port=/dev/ttyUSB0 @myApp_0x00000.bin @myApp_0x40000.bin
```

This command has the same effect as the immediately preceding example but it uses automatic address extraction to obviate the need to specify the load addresses explicitly. Note, too, that this example illustrates the use of a Linux device driver name to specify the serial port.

```
esp_tool --port=/dev/ttyUSB0 --reset=auto myApp_0x00000.esp
```

This example illustrates using a combined image file that is probably sparse. Note, also, the use of the `auto` reset protocol that requires connection of DTR to reset and RTS to GPIO0.

```
esp_tool --port=COM2 --dump-mem -a0x40000000 --size=64K rom0.bin
```

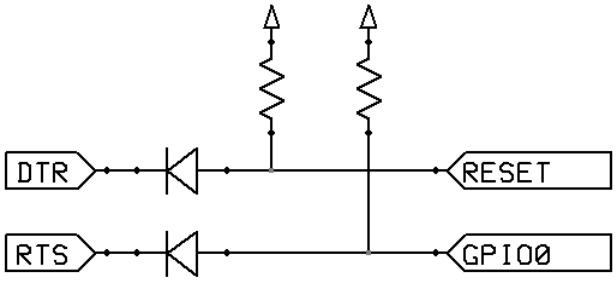
This example dumps the internal ROM to a file. Note the use of the K suffix on the size parameter.

Reset Circuits

Some ESP8266 devices have special circuitry that allows the serial port DTR and/or RTS signals to be used to get the device into the bootloader. There are several known methods available on different boards but the effect of all of the is the same, i.e. to hold GPIO0 low while resetting the device. The following text describes the known methods and provides stylized circuitry for each of them. Although pullup resistors are shown on the RESET and GPIO0 lines in the diagrams below, it is important to note that any particular board may have these present already.

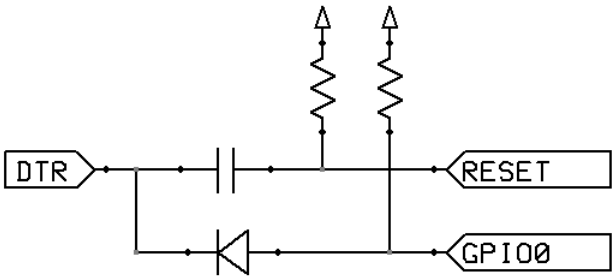
Auto Reset Mode

In this reset mode, the DTR signal is connected to the RESET pin (either directly or via a diode) while the RTS signal is connected to GPIO0 (again, either directly ore via a diode). An example circuit for this mode is shown below.



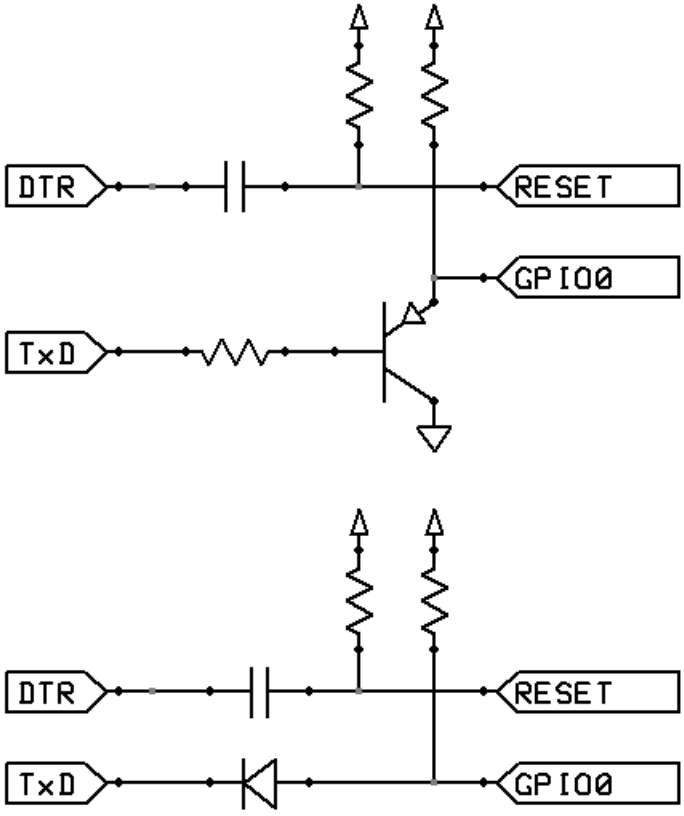
DTR Only Reset Mode

In this reset mode, the DTR signal is connected to the RESET pin via a diode and also connected to GPIO0 (either directly ore via a diode). This is useful with some types of USB-Serial adapters that do not break out the RTS signal. An example circuit for this mode is shown below.



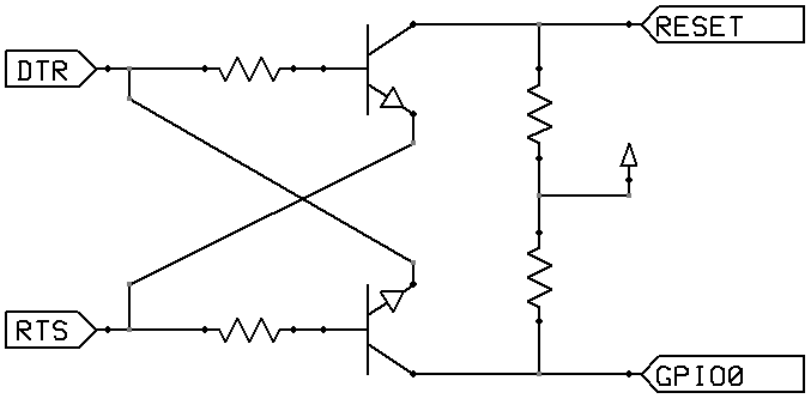
Wifio Reset Mode

This is another reset mode that is useful when RTS is not available. Here, the DTR signal is connected to RESET via a capacitor and GPIO0 is held low via a transistor by transmitting a serial break, effectively holding the TxD line (the RxD input to the ESP8266) low for a period of time, typically about 250mS. The simpler alternative shown second below seems to work just as well..



NodeMCU Reset Mode

In this reset mode, implemented on the NodeMCU Development board, both DTR and RTS are used but they connect to RESET and GPIO0 through a pair of cross-connected NPN transistors. This allows either RESET or GPIO0 to be pulled low but only when DTR and RTS are in opposite logic states.



Disclaimer

The author makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein or the computer program it describes. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The author reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the author or other parties is expressed or implied.