uPyEasy
Building Environment

1 Executive Summary

MicroPython is a software implementation of the Python 3 programming language, written in C, that is optimized to run on a microcontroller. MicroPython is a full Python compiler and runtime that runs on the micro-controller hardware. The user is presented with an interactive prompt (the REPL) to execute supported commands immediately. Included are a selection of core Python libraries, MicroPython includes modules which give the programmer access to low-level hardware.

MicroPython supports a number of ARM based architectures and has since been run on Arduino, ESP8266, ESP32, and Internet of things hardware.

uPyEasy (microPyEasy) is a free and open source MCU firmware for the Internet of things (IoT) and originally developed by the Lisa Esselink. It runs on any MicroPython platform. The name "uPyEasy" by default refers to the firmware rather than the hardware which runs. At a low-level the uPyEasy firmware works the same as the NodeMCU firmware and also provides a very simple operating system on the micropython platform. The main difference between uPyEasy firmware and NodeMCU firmware is that the former is designed as a high-level toolbox that just works out-of-the-box for a pre-defined set of sensors and actuators. Users simply hook-up and read/control over simple web requests without having to write any code at all themselves, including firmware upgrades using OTA (Over The Air) updates.

The uPyEasy firmware can be used to turn any micropython platform into simple multifunction sensor and actuator devices for home automation platforms. Once the firmware is loaded on the hardware, configuration of uPyEasy is entirely web interface based.uPyEasy firmware is primarily used on micropython modules/hardware as a wireless WiFi sensor device with added sensors for temperature, humidity, barometric pressure, LUX, etc. The uPyEasy firmware also offers some low-level actuator functions to control relays.

The firmware is built on the micropython core which in turn uses many open source projects. Getting started with uPyEasy takes a few basic steps. In most cases micropython modules come with AT or NodeMCU LUA firmware, and you need to replace the existing firmware with the uPyEasy firmware by flashing the hardware with a python based flash tool to use it.

uPyEasy is initially re-using the webinterface from ESPEasy, but not the code since ESPEasy is C-based and uPyEasy is python based. Many of the concepts used in uPyEasy, like asynchronicity and a database to store information, are different from ESPEasy.

2 Version History

Date	Version	Comments
10-03-2018	0.1	Initial draft, ESP32 only
17-3-2018	0.2	Added generic build setup

3 Content

4 Purpose

The uPyEasy firmware can be used to turn the micropython module into an easy multifunction sensor device for Home Automation solutions like Domoticz. Configuration of uPyEasy is entirely web based, so once you've got the firmware loaded, you don't need any other tool besides a common web browser.

Where ESPEasy is designed for the ESP8266 SOC (System On a Chip), is uPyEasy designed for the micropython platform. Currently the micropython platform is supporting dozens of SOC’s, and so does uPyEasy.

Due to it’s rapid Agile development cycles, is uPyEasy primarily developed on the unix platform, while the STM32F405RGT, Pyboard, ESP32, ESP8266 and Pi platforms are heavily used for testing purposes.

5 Scope

The scope of uPyEasy:

- MicroPython based
- Multiple platforms (basically all platforms supported by micropython)
- Lan/WLan
- Support for GPIO, I2C, SPI, UART, I2S and CAN interface protocol
- Support for a SD card
- Support of psRam
- OTA firmware updates

6 Goals, Objectives, and Constraints

6.1 Goals

- Written in MicroPython
- Multiple platforms (basically all platforms supported by micropython)
- Basic testing/development done on:
  - Linux
  - STM32F405RGT
  - PyBoard
  - ESP32
  - ESP8266
- Async webserver
- Threads when possible
- Template based webpages
- Lan/WLan transparency
- Support for GPIO, I2C, SPI, UART, I2S and CAN interface protocol
- SSL support both for webserver and client
- Maximum reuse of libraries
- Rules in python
- Override in plugins (custom plugins overrule standard plugins)
- Plugin drag 'n drop (limited in smaller soc's due to memory constraints)
- Support of ANY library that is supported in micropython
- Support of viper speed
- Dynamic support of SD card
- Support of psRam
- Use of FileDB to store config and other values
- Bootloader supporting OTA firmware updates (Yaota8266, 247KB), meaning 1MB esp's will have 750KB free at most when using OTA.
- i18n support

6.2 Constraints

To prevent that uPyEasy is spread thin over to many platforms for to many purposes, leading to uncessary low quality and development times, it's necessary to limit the purpose using this document scope.

The scope of this document is:

- Only ***Linux*** building platform is supported
- Only ***four*** MicroPython ports are used:
  - STM32
  - ESP32
  - ESP8266
  - ARMBIAN

7 Generic Building Environment

MicroPython implements the entire Python 3.4 syntax (including exceptions, `with`, `yield from`, etc., and additionally `async`/`await` keywords from Python 3.5). The following core datatypes are provided: `str` (including basic Unicode support), `bytes`, `bytearray`, `tuple`, `list`, `dict`, `set`, `frozenset`, `array.array`, `collections.namedtuple`, classes and instances. Builtin modules include `sys`, `time`, and `struct`, etc. Select ports have support for `_thread` module (multithreading). Note that only a subset of Python 3 functionality is implemented for the data types and modules.

MicroPython can execute scripts in textual source form or from precompiled bytecode, in both cases either from an on-device filesystem or "frozen" into the MicroPython executable.

See the repository http://github.com/micropython/pyboard for the MicroPython board (PyBoard), the officially supported reference electronic circuit board.

Major components in this repository:

- py/ -- the core Python implementation, including compiler, runtime, and core library.
- mpy-cross/ -- the MicroPython cross-compiler which is used to turn scripts into precompiled bytecode.
- ports/unix/ -- a version of MicroPython that runs on Unix.
- ports/stm32/ -- a version of MicroPython that runs on the PyBoard and similar STM32 boards (using ST's Cube HAL drivers).
- ports/minimal/ -- a minimal MicroPython port. Start with this if you want to port MicroPython to another microcontroller.
- tests/ -- test framework and test scripts.
- docs/ -- user documentation in Sphinx reStructuredText format. Rendered HTML documentation is available at http://docs.micropython.org (be sure to select needed board/port at the bottom left corner).

Additional components:

- ports/bare-arm/ -- a bare minimum version of MicroPython for ARM MCUs. Used mostly to control code size.
- ports/teensy/ -- a version of MicroPython that runs on the Teensy 3.1 (preliminary but functional).
- ports/pic16bit/ -- a version of MicroPython for 16-bit PIC microcontrollers.
- ports/cc3200/ -- a version of MicroPython that runs on the CC3200 from TI.
- ports/esp8266/ -- an experimental port for ESP8266 WiFi modules.
- extmod/ -- additional (non-core) modules implemented in C.
- tools/ -- various tools, including the pyboard.py module.

- examples/ -- a few example Python scripts.

The subdirectories above may include READMEs with additional info.

"make" is used to build the components, or "gmake" on BSD-based systems. You will also need bash, gcc, and Python (at least 2.7 or 3.3).

8 Unix Building Environment

- DEB-based systems
- FreeBSD-based systems
- RPM-based systems
- Pacman-based systems
- Gentoo-based systems
- Mac systems

8.1.1 Debian, Ubuntu, Mint, and variants

The following packages will need to be installed before you can compile and run MicroPython:

- build-essential
- libreadline-dev
- libffi-dev
- git
- pkg-config (required at least in ubuntu 14.04)

To install these packages, use the following command:

```
sudo apt-get install build-essential libreadline-dev libffi-dev git pkg-config
```

Then, clone the repository to your local machine:

```
git clone --recurse-submodules https://github.com/micropython/micropython.git
```

Change directory to the Unix build directory:

```
cd ./micropython/ports/unix
```

And then make the executable

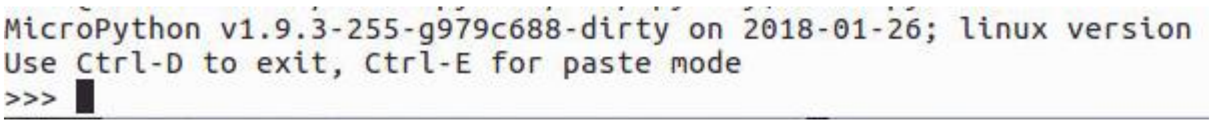
```
make axtls
```

```
make
```

At that point, you will have a functioning micropython executable, which may be launched with the command:

```
./micropython
```

At this point the REPL (Read, Eval, Print, Loop) command is shown:



```
MicroPython v1.9.3-255-g979c688-dirty on 2018-01-26; linux version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> 
```

8.1.2 FreeBSD

(Release 9.2 tested)

Ensure that you have git, GCC, gmake, python3, and bash packages installed:

[as root] pkg_add -r git gcc gmake python3 bash

Clone the git repository to your local machine:

git clone https://github.com/micropython/micropython.git

Change directory to the Unix build directory:

cd ./micropython/ports/unix

Edit main.c, replacing "malloc.h" with "stdlib.h", then:

gmake

This will generate the 'py' executable, which may be executed by:

./micropython

8.1.3 Fedora, CentOS, and Red Hat Enterprise Linux and variants

For Fedora 'micropython' is available in the Fedora Package Collection.

sudo dnf -y install micropython

Additional details can be found in the Fedora Developer Portal.

If you want a more recent release or if you are running CentOS then follow those steps. The required packages can be installed with:

```
sudo yum install git gcc readline-devel libffi-devel
```

Clone the git repository to your local machine:

```
git clone https://github.com/micropython/micropython.git
```

Change directory to the Unix build directory:

```
cd ./micropython/ports/unix
```

And then make the executable

```
make axtls
```

```
make
```

At that point, you will have a functioning micropython executable, which may be launched with the command:

```
./micropython
```

8.1.4 ArchLinux

The following packages will need to be installed before you can compile and run MicroPython:

- gcc or gcc-multilib
- readline
- git

To install these packages, use the following command:

```
pacman -S gcc readline git
```

Then, clone the repository to your local machine:

```
git clone https://github.com/micropython/micropython.git
```

Change directory to the Unix build directory:

cd micropython/ports/unix

And then make the executable

make

At that point, you will have a functioning micropython executable, which may be launched with the command:

./micropython

8.1.5 Gentoo Linux

emerge dfu-util app-mobilephone/dfu-util-0.9::gentoo

8.1.6 Mac OSX

Dependencies are listed on MicroPython on Mac OSX

The XCode and Command Line Developer Tools package will need to be installed before you can compile and run MicroPython:

xcode-select --install

Then, clone the repository to your local machine:

git clone https://github.com/micropython/micropython.git

Change directory to the git repo and fetch all the submodules:

git submodule update --init

Change directory to the Unix build directory:

cd micropython/ports/unix

Make the dependencies:

make axtls

And then make the executable

```
make
```

At that point, you will have a functioning micropython executable, which may be launched with the command:

```
./micropython
```

9 STM32 Building Environment

This directory contains the port of MicroPython to ST's line of STM32Fxxx microcontrollers. It is based on the STM32Cube HAL library and currently supports: STM32F401, STM32F405, STM32F411, STM32F429, STM32F746.

The officially supported boards are the line of pyboards: PYBv1.0 and PYBv1.1 (both with STM32F405), and PYBLITEv1.0 (with STM32F411). See micropython.org/pyboard for further details.

Other boards that are supported include ST Discovery and Nucleo boards. See the boards/ subdirectory, which contains the configuration files used to build each individual board.

9.1 Linux (general)

This page covers installing the ARM toolchain needed to build firmware for the pyboard (or other ARM processors).

One source of the ARM toolchain (for all platforms) is from: https://launchpad.net/gcc-arm-embedded

For all of the linux variants, you should download the "Linux installation tarball". There is a link over on the right side of the page. The actual URL changes periodically (about once every 3 months).

```
mkdir ~/stm
cd ~/stm
wget https://launchpad.net/gcc-arm-embedded/4.8/4.8-2014-q2-
update/+download/gcc-arm-none-eabi-4_8-2014q2-20140609-linux.tar.bz2
tar xf gcc-arm-none-eabi-4_8-2014q2-20140609-linux.tar.bz2
```

and then add `${HOME}/stm/gcc-arm-none-eabi-4_8-2014q2/bin` to your PATH. You might do this by adding the following line to your `~/.bashrc` file:

```
export PATH="${PATH}:${HOME}/stm/gcc-arm-none-eabi-4_8-2014q2/bin"
```

and then it will be added automatically each time you login. You can also just execute the command directly from your bash shell and it will work until you exit your shell.

Note: This is a 32-bit toolchain, so if you're using a 64-bit version of linux, you may need to also install the 32-bit version of libc.

For Ubuntu 14.04, I did:

```
sudo apt-get install libc6:i386
```

#Using Symbolic Debugging Symbolics can be useful for module or hardware integration testing. The hardware link to the processor is often manufacturer specific, and the debugging software needs a driver to that software. For STM32 https://github.com/micropython/micropython/wiki/Symbolic-Debugging-for-STM32

9.2 Build instructions

Before building the firmware for a given board the MicroPython cross-compiler must be built; it will be used to pre-compile some of the built-in scripts to bytecode. The cross-compiler is built and run on the host machine, using:

```
$ make -C mpy-cross
```

This command should be executed from the root directory of this repository. All other commands below should be executed from the ports/stm32/ directory.

An ARM compiler is required for the build, along with the associated binary utilities. The default compiler is `arm-none-eabi-gcc`, which is available for Arch Linux via the package `arm-none-eabi-gcc`, for Ubuntu via instructions here, or see here for the main GCC ARM Embedded page. The compiler can be changed using the `CROSS_COMPILE` variable when invoking `make`.

To build for a given board, run:

```
$ make BOARD=PYBV11
```

The default board is PYBV10 but any of the names of the subdirectories in the `boards/` directory can be passed as the argument to `BOARD=`. The above command should produce binary images in the `build-PYBV11/` subdirectory (or the equivalent directory for the board specified).

You must then get your board/microcontroller into DFU mode. On the pyboard connect the 3V3 pin to the P1/DFU pin with a wire (they are next to each other on the bottom left of the board, second row from the bottom) and then reset (by pressing the RST button) or power on the board. Then flash the firmware using the command:

```
$ make BOARD=PYBV11 deploy
```

This will use the included `tools/pydfu.py` script. You can use instead the `dfu-util` program (available here) by passing `USE_PYDFU=0`:

```
$ make BOARD=PYBV11 USE_PYDFU=0 deploy
```

If flashing the firmware does not work it may be because you don't have the correct permissions. Try then:

```
$ sudo make BOARD=PYBV11 deploy
```

Or using `dfu-util` directly:

```
$ sudo dfu-util -a 0 -d 0483:df11 -D build-PYBV11/firmware.dfu
```

9.2.1 Flashing the Firmware with stlink

ST Discovery or Nucleo boards have a builtin programmer called ST-LINK. With these boards and using Linux or OS X, you have the option to upload the `stm32` firmware using the `st-flash` utility from the stlink project. To do so, connect the board with a mini USB cable to its ST-LINK USB port and then use the make target `deploy-stlink`. For example, if you have the STM32F4DISCOVERY board, you can run:

```
$ make BOARD=STM32F4DISC deploy-stlink
```

The `st-flash` program should detect the USB connection to the board automatically. If not, run `lsusb` to determine its USB bus and device number and set the `STLINK_DEVICE` environment variable accordingly, using the format `<USB_BUS>:<USB_ADDR>`. Example:

```
$ lsusb
[...]
Bus 002 Device 035: ID 0483:3748 STMicroelectronics ST-LINK/V2
$ export STLINK_DEVICE="002:0035"
$ make BOARD=STM32F4DISC deploy-stlink
```

9.2.2 Flashing the Firmware with OpenOCD

Another option to deploy the firmware on ST Discovery or Nucleo boards with a ST-LINK interface uses OpenOCD. Connect the board with a mini USB cable to its ST-LINK USB port and then use the make target `deploy-openocd`. For example, if you have the STM32F4DISCOVERY board:

```
$ make BOARD=STM32F4DISC deploy-openocd
```

The `openocd` program, which writes the firmware to the target board's flash, is configured via the file `ports/stm32/boards/openocd_stm32f4.cfg`. This configuration should work for all boards based on a STM32F4xx MCU with a ST-LINKv2 interface. You can override the path to this configuration by setting `OPENOCD_CONFIG` in your Makefile or on the command line.

9.3 Accessing the board

Once built and deployed, access the MicroPython REPL (the Python prompt) via USB serial or UART, depending on the board. For the pyboard you can try:

```
$ picocom /dev/ttyACM0
```

uPyEasy
Building Environment

10 ESP32 Building Environment

This chapter is intended to help users set up the software environment for development of applications using hardware based on the Espressif ESP32. Through a simple example we would like to illustrate how to use ESP-IDF (Espressif IoT Development Framework), including the menu based configuration, compiling the ESP-IDF and firmware download to ESP32 boards.

Introduction

ESP32 integrates Wi-Fi (2.4 GHz band) and Bluetooth 4.2 solutions on a single chip, along with dual high performance cores, Ultra Low Power co-processor and several peripherals. Powered by 40 nm technology, ESP32 provides a robust, highly integrated platform to meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.

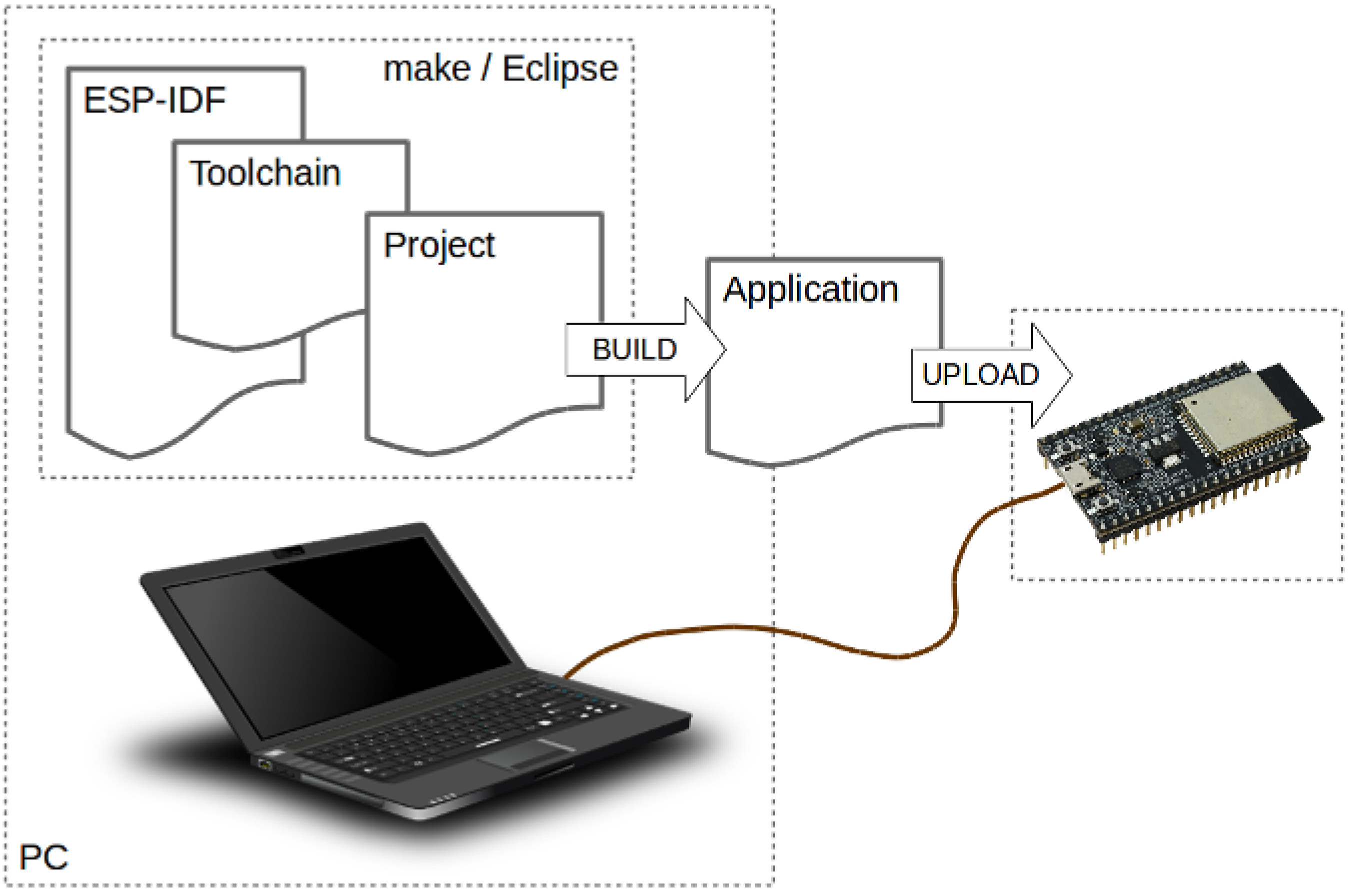
Espressif provides the basic hardware and software resources that help application developers to build their ideas around the ESP32 series hardware. The software development framework by Espressif is intended for rapidly developing Internet-of-Things (IoT) applications, with Wi-Fi, Bluetooth, power management and several other system features.

What You Need

To develop applications for ESP32 you need:

- PC loaded with Linux operating system
- Toolchain to build the Application for ESP32
- ESP-IDF that essentially contains API for ESP32 and scripts to operate the Toolchain
- A text editor to write programs (Projects) in C, e.g. Eclipse
- The ESP32 board itself and a USB cable to connect it to the PC

uPyEasy
Building Environment



Development of applications for ESP32

Preparation of development environment consists of two steps:

1. Setup of **Toolchain**
2. Getting of **ESP-IDF** from GitHub

Having environment set up, you are ready to start the most interesting part - the application development. This process may be summarized in four steps:

1. Configuration of a **Project** and writing the code
2. Compilation of the **Project** and linking it to build an **Application**
3. Flashing (uploading) of the **Application** to **ESP32**
4. Monitoring / debugging of the **Application**

Supported features include:

- REPL (Python prompt) over UART0.

- 16k stack for the MicroPython task and 96k Python heap.
- Many of MicroPython's features are enabled: unicode, arbitrary-precision integers, single-precision floats, complex numbers, frozen bytecode, as well as many of the internal modules.
- Internal filesystem using the flash (currently 2M in size).
- The machine module with GPIO, UART, SPI, software I2C, ADC, DAC, PWM, TouchPad, WDT and Timer.
- The network module with WLAN (WiFi) support.

10.1 Setting up the ESP-IDF

There are two main components that are needed to build the firmware:

- The Xtensa cross-compiler that targets the CPU in the ESP32 (this is different to the compiler used by the ESP8266)
- The Espressif IDF (IoT development framework, aka SDK)

The ESP-IDF changes quickly and MicroPython only supports a certain version. The git hash of this version can be found by running make without a configured ESPIDF. Then you can fetch only the given esp-idf using the following command:

```
$ git clone https://github.com/espressif/esp-idf.git
$ git checkout <Current supported ESP-IDF commit hash>
$ git submodule update –recursive
```

10.2 Setting up the toolchain

Install Prerequisites

To compile with ESP-IDF you need to get the following packages:

- CentOS 7:

  ```
  sudo yum install git wget make ncurses-devel flex bison gperf python
  pyserial
  ```

- Ubuntu and Debian:

```
sudo apt-get install git wget make libncurses-dev flex bison gperf
python python-serial
```

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-
pyserial
```

Toolchain Setup

ESP32 toolchain for Linux is available for download from Espressif website:

- for 64-bit Linux:

  https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz

- for 32-bit Linux:

  https://dl.espressif.com/dl/xtensa-esp32-elf-linux32-1.22.0-80-g6c4433a-5.2.0.tar.gz

1. Download this file, then extract it in `~/esp` directory:

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-
5.2.0.tar.gz
```

2. The toolchain will be extracted into `~/esp/xtensa-esp32-elf/` directory.

   To use it, you will need to update your `PATH` environment variable in `~/.profile` file. To make `xtensa-esp32-elf` available for all terminal sessions, add the following line to your `~/.profile` file:

```
export PATH="$PATH:$HOME/esp/xtensa-esp32-elf/bin"
```

   Alternatively, you may create an alias for the above command. This way you can get the toolchain only when you need it. To do this, add different line to your `~/.profile` file:

```
alias get_esp32='export PATH="$PATH:$HOME/esp/xtensa-esp32-elf/bin"'
```

   Then when you need the toolchain you can type `get_esp32` on the command line and the toolchain will be added to your `PATH`.

Note

If you have `/bin/bash` set as login shell, and both `.bash_profile` and `.profile` exist, then update `.bash_profile` instead.

3. Log off and log in back to make the `.profile` changes effective. Run the following command to verify if `PATH` is correctly set:

   ```
   printenv PATH
   ```

   You are looking for similar result containing toolchain's path at the end of displayed string:

   ```
   $ printenv PATH
   /home/user-name/bin:/home/user-
   name/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
   :/bin:/usr/games:/usr/local/games:/snap/bin:/home/user-name/esp/xtensa-
   esp32-elf/bin
   ```

   Instead of `/home/user-name` there should be a home path specific to your installation.

Permission issues /dev/ttyUSB0

With some Linux distributions you may get the `Failed to open port /dev/ttyUSB0` error message when flashing the ESP32. This can be solved by adding the current user to the dialout group.

Arch Linux Users

To run the precompiled gdb (xtensa-esp32-elf-gdb) in Arch Linux requires ncurses 5, but Arch uses ncurses 6.

Backwards compatibility libraries are available in AUR for native and lib32 configurations:

- https://aur.archlinux.org/packages/ncurses5-compat-libs/
- https://aur.archlinux.org/packages/lib32-ncurses5-compat-libs/

Before installing these packages you might need to add the author's public key to your keyring as described in the "Comments" section at the links above.

Alternatively, use crosstool-NG to compile a gdb that links against ncurses 6.

The Espressif ESP-IDF instructions above only install pyserial for Python 2, so if you're running Python 3 or a non-system Python you'll also need to install `pyserial` (or `esptool`) so that the Makefile can flash the board and set parameters:

```
$ pip install pyserial
```

Once everything is set up you should have a functioning toolchain with prefix xtensa-esp32-elf- (or otherwise if you configured it differently) as well as a copy of the ESP-IDF repository. You will need to update your `PATH` environment variable to include the ESP32 toolchain. For example, you can issue the following commands on (at least) Linux:

```
$ export PATH=$PATH:$HOME/esp/crosstool-NG/builds/xtensa-esp32-elf/bin
```

You can put this command in your `.profile` or `.bash_login`.

You then need to set the `ESPIDF` environment/makefile variable to point to the root of the ESP-IDF repository. You can set the variable in your PATH, or at the command line when calling make, or in your own custom `makefile`. The last option is recommended as it allows you to easily configure other variables for the build. In that case, create a new file in the esp32 directory called `makefile` and add the following lines to that file:

```
ESPIDF = <path to root of esp-idf repository>
#PORT = /dev/ttyUSB0
#FLASH_MODE = qio
#FLASH_SIZE = 4MB
#CROSS_COMPILE = xtensa-esp32-elf-

include Makefile
```

Be sure to enter the correct path to your local copy of the IDF repository (and use `$(HOME)`, not tilde, to reference your home directory). If your filesystem is case-insensitive then you'll need to use `GNUmakefile` instead of `makefile`. If the Xtensa cross-compiler is not in your path you can use the `CROSS_COMPILE` variable to set its location. Other options of interest are `PORT` for the serial port of your esp32 module, and `FLASH_MODE` (which may need to be `dio` for some modules) and `FLASH_SIZE`. See the Makefile for further information.

10.3 uPyEasy Modules

The uPyEasy modules from the source code directory `letscontrolit/uPyEasy/src` must be copied to the directory `ports/esp32/modules/upyeasy` . The modules in the source code directory `letscontrolit/uPyEasy/modules` must be copied to the directory `ports/esp32/modules`

10.4 Building the firmware

The MicroPython cross-compiler must be built to pre-compile some of the built-in scripts to bytecode. This can be done by (from the root of this repository):

```
$ make -C mpy-cross
```

The ESP32 port has a dependency on Berkeley DB, which is an external dependency (git submodule). You'll need to have git initialize that module using the commands:

```
$ git submodule init lib/berkeley-db-1.xx
$ git submodule update
```

Then to build MicroPython for the ESP32 run:

```
$ cd ports/esp32
$ make
```

This will produce binary firmware images in the `build/` subdirectory (three of them: bootloader.bin, partitions.bin and application.bin).

To flash the firmware you must have your ESP32 module in the bootloader mode and connected to a serial port on your PC. Refer to the documentation for your particular ESP32 module for how to do this. The serial port and flash settings are set in the `Makefile`, and can be overridden in your local `makefile`; see above for more details.

You will also need to have user permissions to access the /dev/ttyUSB0 device. On Linux, you can enable this by adding your user to the `dialout` group, and rebooting or logging out and in again.

```
$ sudo adduser <username> dialout
```

If you are installing MicroPython to your module for the first time, or after installing any other firmware, you should first erase the flash completely:

```
$ make erase
```

To flash the MicroPython firmware to your ESP32 use:

```
$ make deploy
```

This will use the `esptool.py` script (provided by ESP-IDF) to download the binary images.

10.5 Getting a Python prompt

You can get a prompt via the serial port, via UART0, which is the same UART that is used for programming the firmware. The baudrate for the REPL is 115200 and you can use a command such as:

```
$ picocom -b 115200 /dev/ttyUSB0
```

10.6 Configuring the WiFi and using the board

The ESP32 port is designed to be (almost) equivalent to the ESP8266 in terms of the modules and user-facing API. There are some small differences, notably that the ESP32 does not automatically connect to the last access point when booting up. But for the most part the documentation and tutorials for the ESP8266 should apply to the ESP32 (at least for the components that are implemented).

See http://docs.micropython.org/en/latest/esp8266/esp8266/quickref.html for a quick reference, and http://docs.micropython.org/en/latest/esp8266/esp8266/tutorial/intro.html for a tutorial.

The following function can be used in your own code to connect to a WiFi access point (you can either pass in your own SSID and password, or change the defaults so you can quickly call `wlan_connect()` and it just works):

```
def wlan_connect(ssid='MYSSID', password='MYPASS'):
    import network
    wlan = network.WLAN(network.STA_IF)
    if not wlan.active() or not wlan.isconnected():
        wlan.active(True)
        print('connecting to:', ssid)
        wlan.connect(ssid, password)
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

Note that some boards require you to configure the WiFi antenna before using the WiFi. On Pycom boards like the LoPy and WiPy 2.0 you need to execute the following code to select the internal antenna (best to put this line in your boot.py file):

```
import machine
antenna = machine.Pin(16, machine.Pin.OUT, value=0)
```

10.7 Boot scripts

On boot, MicroPython EPS32 port executes `_boot.py` script from internal frozen modules. It mounts filesystem in FlashROM, or if it's not available, performs first-time setup of the module and creates the filesystem. This part of the boot process is considered fixed, and not available for customization for end users (even if you build from source, please refrain from changes to it; customization of early boot process is available only to advanced users and developers, who can diagnose themselves any issues arising from modifying the standard process).

Once the filesystem is mounted, `boot.py` is executed from it. The standard version of this file is created during first-time module set up and has commands to start a WebREPL daemon (disabled by default, configurable with `webrepl_setup` module), etc. This file is customizable by end users (for example, you may want to set some parameters or add other services which should be run on a module start-up). But keep in mind that incorrect modifications to boot.py may still lead to boot loops or lock ups, requiring to reflash a module from scratch. (In particular, it's recommended that you use either `webrepl_setup` module or manual editing to configure WebREPL, but not both).

As a final step of boot procedure, `main.py` is executed from filesystem, if exists. This file is a hook to start up a user application each time on boot (instead of going to REPL). For small test applications, you may name them directly as `main.py`, and upload to module, but instead it's recommended to keep your application(s) in separate files, and have just the following in `main.py`:

```
import upyeasy
upyeasy.main()
```

This will allow to keep the structure of your application clear, as well as allow to install multiple applications on a board, and switch among them.

uPyEasy already has a changed `initsetup.py` file in the modules directory which will automatically create the correct boot.py file in which these command are listed:

```
import upyeasy
upyeasy.main()
```

If that is not the case, please alter the boot.py file manually.

10.8 Troubleshooting

- Continuous reboots after programming: Ensure FLASH_MODE is correct for your board (e.g. ESP-WROOM-32 should be DIO). Then perform a `make clean`, rebuild, redeploy.

11 ESP8266

This is an experimental port of MicroPython for the WiFi modules based on Espressif ESP8266 chip.

WARNING: The port is experimental and many APIs are subject to change.

Supported features include:

- REPL (Python prompt) over UART0.
- Garbage collector, exceptions.
- Unicode support.
- Builtin modules: gc, array, collections, io, struct, sys, esp, network, many more.
- Arbitrary-precision long integers and 30-bit precision floats.
- WiFi support.
- Sockets using modlwip.
- GPIO and bit-banging I2C, SPI support.
- 1-Wire and WS2812 (aka Neopixel) protocols support.
- Internal filesystem using the flash.
- WebREPL over WiFi from a browser (clients at https://github.com/micropython/webrepl).
- Modules for HTTP, MQTT, many other formats and protocols via https://github.com/micropython/micropython-lib .

Work-in-progress documentation is available at http://docs.micropython.org/en/latest/esp8266/ .

11.1 Build instructions

The tool chain required for the build is the OpenSource ESP SDK, which can be found at https://github.com/pfalcon/esp-open-sdk. Clone this repository and run `make` in its directory to build and install the SDK locally. Make sure to add toolchain bin directory to your PATH. Read esp-open-sdk's README for additional important information on toolchain setup.

Add the external dependencies to the MicroPython repository checkout:

```
$ git submodule update --init
```

See the README in the repository root for more information about external dependencies.

The MicroPython cross-compiler must be built to pre-compile some of the built-in scripts to bytecode. This can be done using:

```
$ make -C mpy-cross
```

Then, to build MicroPython for the ESP8266, just run:

```
$ cd ports/esp8266
$ make axtls
$ make
```

This will produce binary images in the `build/` subdirectory. If you install MicroPython to your module for the first time, or after installing any other firmware, you should erase flash completely:

```
esptool.py --port /dev/ttyXXX erase_flash
```

Erase flash also as a troubleshooting measure, if a module doesn't behave as expected.

To flash MicroPython image to your ESP8266, use:

```
$ make deploy
```

This will use the `esptool.py` script to download the images. You must have your ESP module in the bootloader mode, and connected to a serial port on your PC. The default serial port is `/dev/ttyACM0`, flash mode is `qio` and flash size is `detect` (auto-detect based on Flash ID). To specify other values, use, eg (note that flash size is in megabits):

```
$ make PORT=/dev/ttyUSB0 FLASH_MODE=qio FLASH_SIZE=32m deploy
```

The image produced is `build/firmware-combined.bin`, to be flashed at 0x00000.

512KB FlashROM version

The normal build described above requires modules with at least 1MB of FlashROM onboard. There's a special configuration for 512KB modules, which can be built with `make 512k`. This configuration is **NOT** supported for uPyEasy!

11.2 First start

Be sure to change ESP8266's WiFi access point password ASAP, see below.

Serial prompt

You can access the REPL (Python prompt) over UART (the same as used for programming).

- Baudrate: 115200

Run `help()` for some basic information.

WiFi

Initially, the device configures itself as a WiFi access point (AP).

- ESSID: MicroPython-xxxxxx (x's are replaced with part of the MAC address).
- Password: micropythoN (note the upper-case N).
- IP address of the board: 192.168.4.1.
- DHCP-server is activated.
- Please be sure to change the password to something non-guessable immediately. `help()` gives information how.

WebREPL

Python prompt over WiFi, connecting through a browser.

- Hosted at http://micropython.org/webrepl.
- GitHub repository https://github.com/micropython/webrepl. Please follow the instructions there.

upip

The ESP8266 port comes with builtin `upip` package manager, which can be used to install additional modules (see the main README for more information):

```
>>> import upip
>>> upip.install("micropython-pystone_lowmem")
[...]
>>> import pystone_lowmem
>>> pystone_lowmem.main()
```

Downloading and installing packages may requite a lot of free memory, if you get an error, retry immediately after the hard reset.

11.3 Documentation

More detailed documentation and instructions can be found at http://docs.micropython.org/en/latest/esp8266/ , which includes Quick Reference, Tutorial, General Information related to ESP8266 port, and to MicroPython in general.

11.4 Troubleshooting

While the port is in beta, it's known to be generally stable. If you experience strange bootloops, crashes, lockups, here's a list to check against:

- You didn't erase flash before programming MicroPython firmware.
- Firmware can be occasionally flashed incorrectly. Just retry. Recent esptool.py versions have --verify option.
- Power supply you use doesn't provide enough power for ESP8266 or isn't stable enough.
- A module/flash may be defective (not unheard of for cheap modules).

Please consult dedicated ESP8266 forums/resources for hardware-related problems.

Additional information may be available by the documentation links above.

12 ARMBIAN

On a Ubuntu 14.04LTS this worked. First remove the arm-none-eabi that comes with Ubuntu 14.04LTS and install the gcc-arm-embedded version.

sudo apt-get remove binutils-arm-none-eabi gcc-arm-none-eabi

sudo add-apt-repository ppa:terry.guo/gcc-arm-embedded

sudo apt-get update

sudo apt-get install gcc-arm-none-eabi

(as of 2015Sept19 pulled in amd64 4.9.3.2015q2-1trusty1)

If needed to remove

sudo apt-get remove gcc-arm-none-eabi

For teensy script add-memzip.sh need

sudo apt-get install realpath

Assuming micropython has been installed via git in current directory

cd git\micropython\stmhal

make

(completes but didn't test)

cd ../teensy

make

(completes but didn't test)

See Running Scripts in https://github.com/micropython/micropython/wiki/Board-Teensy3.1)

Discussion on design https://forum.pjrc.com/threads/24794-MicroPython-for-Teensy-3-1/)

cd ../minimal

make - FAILS

12.1.1 Fedora

Install packages arm-none-eabi-gcc, arm-none-eabi-binutils and arm-none-eabi-newlib to be able to build stmhal port. Install dfu-util for flashing the firmware.