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.

uPyEasy
Building Environment

2 Version History

Date	Version	Comments
10-03-2018	0.1	Initial draft, ESP32 only

3 Content

uPyEasy
Building Environment

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 STM32 Building Environment

8 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.

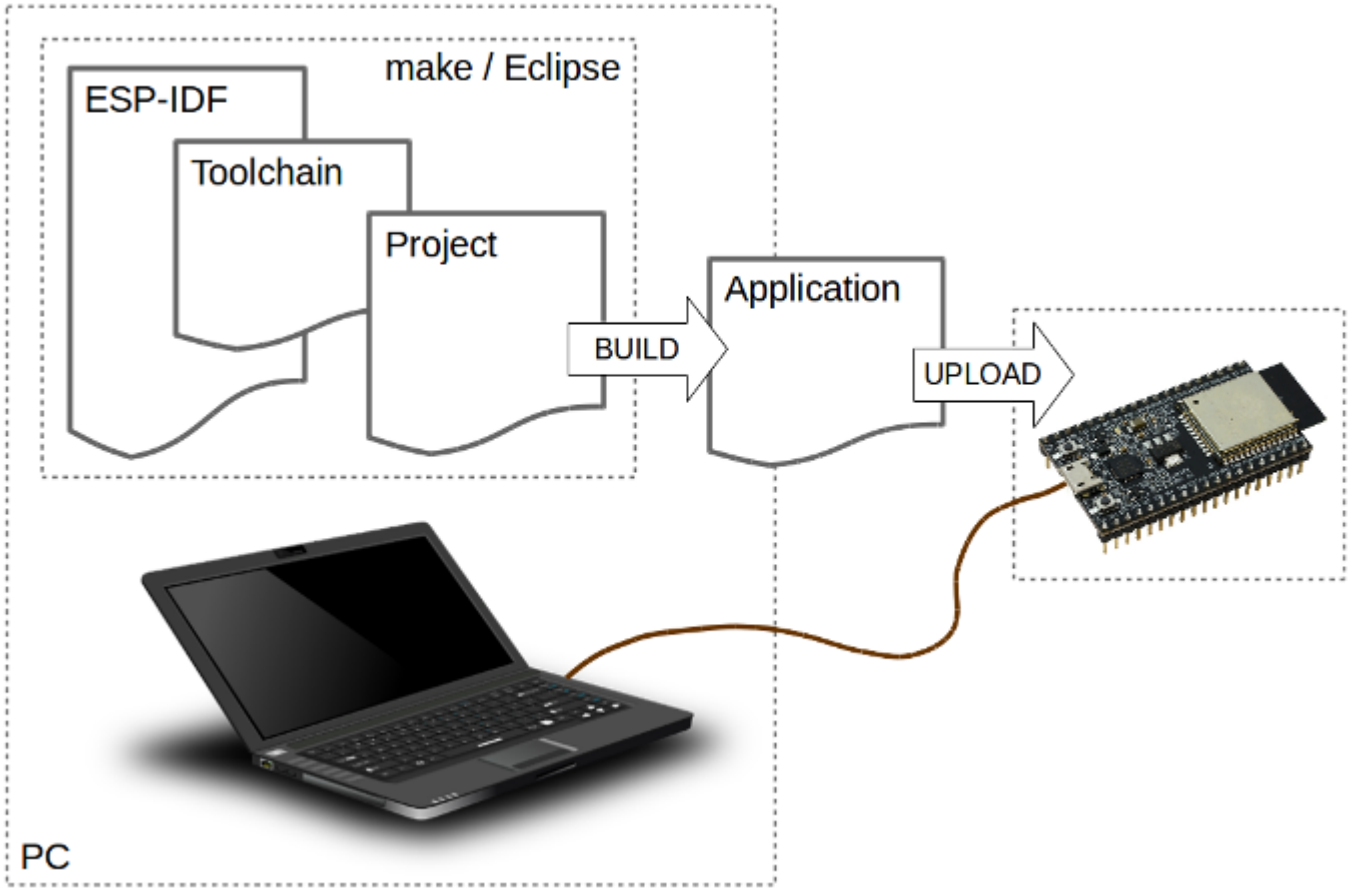
Espressif provides the basic hardware and software resources that help application developers to build their ideas around the ESP32 series hardware. The software development framework by Espressif is intended for rapidly developing Internet-of-Things (IoT) applications, with Wi-Fi, Bluetooth, power management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

8.1 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

8.2 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.

8.3 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`

8.4 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.

8.5 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
```

8.6 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 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)
```

8.7 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.

9 ESP8266

10 ARMBIAN

