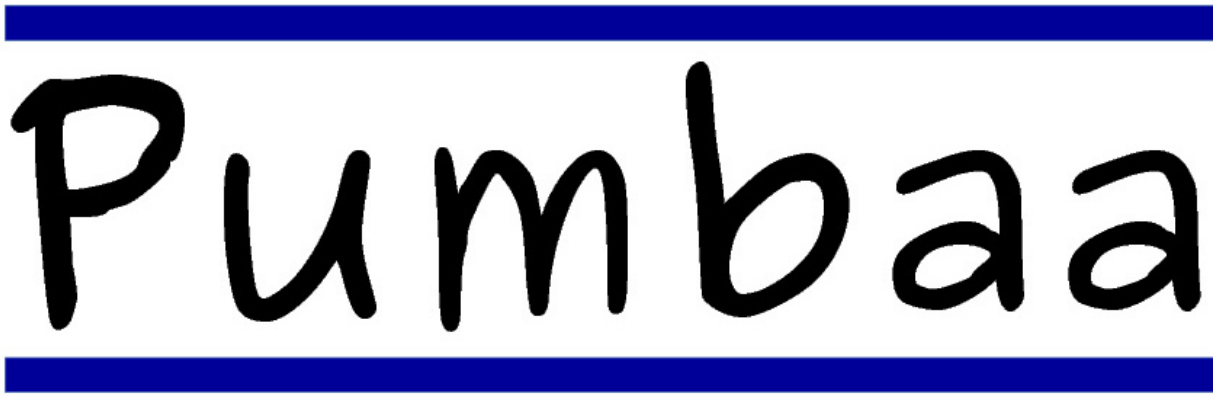


Pumbaa Documentation

Release 3.0.2

Erik Moqvist

Jul 19, 2017

Contents

Pumbaa is Python on top of Simba.

The implementation is a port of MicroPython, designed for embedded devices with limited amount of RAM and code memory.

Project homepage: https://github.com/eerimoq/pumbaa

CHAPTER 1

Videos

Measure the DAC output voltage on a Nano32 (ESP32). More videos are available on the *Videos* page.

Getting Started

Installation

There are three build systems available; *PlatformIO*, *Arduino IDE* and *Simba build system*. The *Simba build system* has more features than to the other two. It supports executing test suites, generating code coverage, profiling and more. Still, if you are familiar with *Arduino IDE* or *PlatformIO*, use that instead since it will be less troublesome.

PlatformIO

Install *Pumbaa* in PlatformIO.

1. Install the PlatformIO IDE.
2. Start the *PlatformIO IDE* and open *PlatformIO -> Project Examples* and select *pumbaa/blink*.
3. Click on *Upload* (the arrow image) in the top left corner.
4. The built-in LED blinks!
5. Done!

Arduino IDE

Install *Pumbaa* in the Arduino IDE 1.6.10 as a third party board using the Boards Manager.

1. Open *File -> Preferences*.

2. Add these URL:s to *Additional Boards Manager URLs* (click on the icon to the right of the text field) and press *OK*.

```
https://raw.githubusercontent.com/eerimoq/pumbaa-releases/master/arduino/sam/
↪package_pumbaa_sam_index.json
https://raw.githubusercontent.com/eerimoq/pumbaa-releases/master/arduino/esp32/
↪package_pumbaa_esp32_index.json
```

3. Open *Tools -> Board: ... -> Boards Manager...* and type *pumbaa* in the search box.
4. Click on *Pumbaa by Erik Moqivst version x.y.z* and click *Install* and press *Close*.
5. Open *Tools -> Board: ... -> Boards Manager...* and select one of the Pumbaa boards in the list.
6. Open *File -> Examples -> Pumbaa -> blink*.
7. Verify and upload the sketch to your device.
8. The built-in LED blinks!
9. Done!

Simba build system

The *Pumbaa* development environment can be installed on *Linux (Ubuntu 14)*.

1. Execute the one-liner below to install *Pumbaa*.

```
$ mkdir pumbaa && \
  cd pumbaa && \
  sudo apt install ckermit valgrind cppcheck cloc python python-pip doxygen git␣
↪lcov && \
  sudo apt install avrdude gcc-avr binutils-avr gdb-avr avr-libc && \
  sudo apt install bossa-cli gcc-arm-none-eabi && \
  sudo apt install make unrar autoconf automake libtool gcc g++ gperf \
                       flex bison texinfo gawk ncurses-dev libexpat-dev \
                       python-serial sed libtool-bin pmccabe help2man \
                       python-pyelftools unzip && \
  sudo pip install pyserial xpect readchar sphinx breathe sphinx_rtd_theme && \
  (git clone --recursive https://github.com/pfalcon/esp-open-sdk && \
   cd esp-open-sdk && \
   make) && \
  wget https://github.com/eerimoq/simba-releases/raw/master/arduino/esp32/tools/
↪xtensa-esp32-elf-linux$(getconf LONG_BIT)-1.22.0-59.tar.gz && \
  tar xf xtensa-esp32-elf-linux$(getconf LONG_BIT)-1.22.0-59.tar.gz && \
  rm xtensa-esp32-elf-linux$(getconf LONG_BIT)-1.22.0-59.tar.gz && \
  git clone --recursive https://github.com/eerimoq/pumbaa
```

2. Setup the environment.

```
$ cd pumbaa
$ source setup.sh
```

2. Build and upload the blink example to your device. Replace `<my-serial-port>` with your serial port name.

```
$ cd examples/blink
$ make -s BOARD=nano32 SERIAL_PORT=<my-serial-port> upload
```

3. The built-in LED blinks!
4. Done!

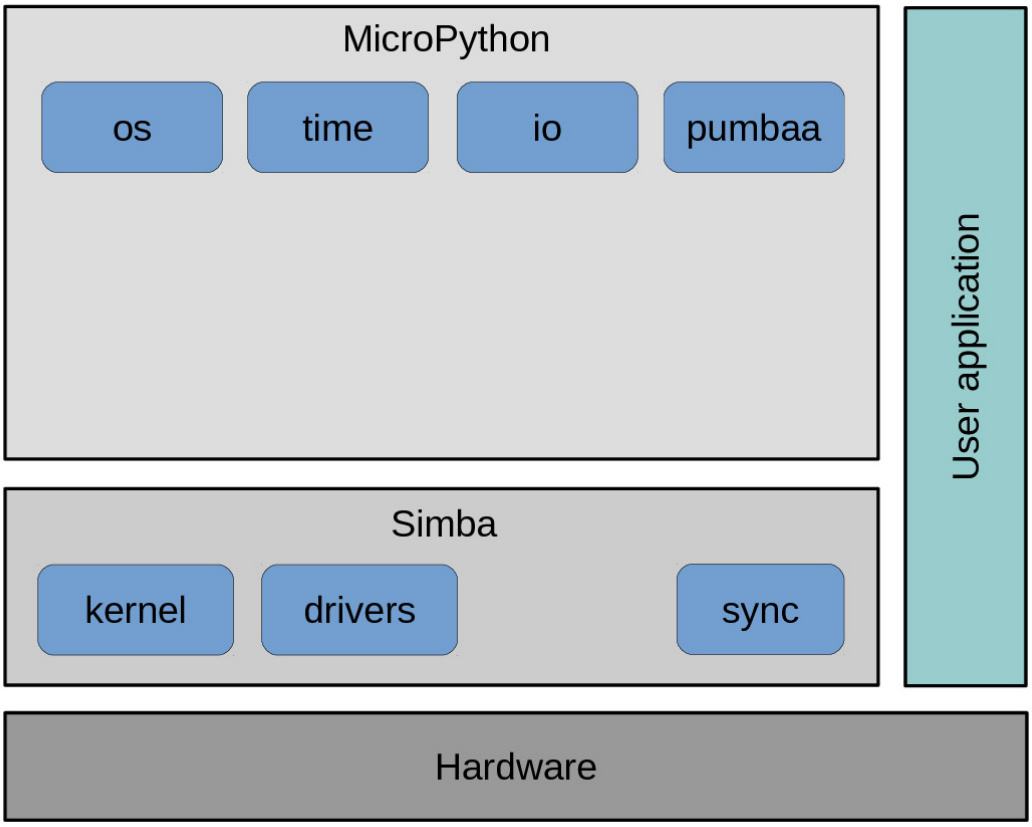
User Guide

This guide is intended for users of the Pumbaa Embedded Programming Platform.

The *Pumbaa* installation guide can be found on the *Getting Started* page.

Software architecture

Below is a picture of the *Pumbaa* software architecture. At the bottom is the hardware. On top of the hardware is the *Simba* operating system, that implementes all low level functionality; kernel, drivers, filesystems, networking, etc. *MicroPython* implements the Python 3 language and a many Python standard library modules. The user application on the right can be implemented in a mix of Python and C code depending of the requirements. Normally the whole application is implemnted in Python.



Contents:

Build system

PlatformIO

All Python source files in your project's `src/` folder will be uploaded to the board automatically as frozen modules.

See the PlatformIO website for more information.

Arduino IDE

The build system only allows a single Python script file, the Arduino sketch `.ino`.

See the Arduino website for more information.

Simba build system

The make variable `PYSRC` is list of all Python script files.

See the Simba documentation for more information.

Configuration

Standard Library

The *Library Reference* is configured at compile time using defines that starts with `CONFIG_PUMBAA_`. The default configuration includes most functionality, as most application wants that. If an application has special requirements, for example memory constraints, it has to be configured to remove unnecessaray functionality.

The underlying MicroPython source code can also be configured in compile time. Often these configuration valiables starts with `MICROPY_`.

Simba can be configured as described in the Simba documentation, with the only difference that your `config.h` must include `simba_config.h`, as in src/config.h. Put your defines before this include.

Search order

Highest priority first.

Simba build system

1. Command line as `CDEFS_EXTRA="<configuration variable>=<value>"`.
2. A file named `config.h` in the application root folder.
3. The default configuration file, src/pumbaa_config_default.h.

PlatformIO

1. The variable `build_flags` in `platformio.ini` as `build_flags = -D<configuration variable>=<value>`.
2. A file named `config.h` in the application source folder `src`.
3. The default configuration file, src/pumbaa_config_default.h.

Arduino IDE

1. A file (also called a *tab*) named `config.h` in the sketch.
2. The default configuration file, src/pumbaa_config_default.h.

Variables

All configuration variables are listed in src/pumbaa_config_default.h.

Environment setup

The first step is always to setup the *Pumbaa* environment. It's a simple matter of sourcing a setup-script in the pumbaa root folder.

This step only applies to the *Simba build system*, and not to the *Arduino IDE* or *PlatformIO*.

```
$ source setup.sh
```

Hello World application

Let's start with the *Pumbaa* "Hello World" application. It examplifies what an application is and how to build and run it.

It consists of two files; `main.py` and `Makefile`.

main.py

main.py is the main script of the application.

```
print("Hello world!")
```

Makefile

Makefile contains build configuration of the application.

```
NAME = hello_world
BOARD ?= linux

PUMBAA_ROOT ?= ../..
include $(PUMBAA_ROOT)/make/app.mk
```

Build and run

Compile, link and run it by typing the commands below in a shell:

```
$ cd examples/hello_world
$ make -s run
<build system output>
Hello world!
$
```

Cross-compile, link and then run on an Arduino Due:

```
$ cd examples/hello_world
$ make -s BOARD=arduino_due run
<build system output>
Hello world!
$
```

Developer Guide

This guide is intended for developers of the Pumbaa Embedded Programming Platform. Users are advised to read the *User Guide* instead.

Contents:

Releasing

Follow these steps to create a new release:

1. Write the new version in `VERSION.txt`. The version should hace the format `<major>.<minor>.<revision>`.

   Increment `<major>` for non-backwards compatible changes.

   Increment `<minor>` for new features.

   Increment `<revision>` for bug fixes.

2. Write the new version in `package.json`. This file is used by *PlatformIO 3* to find the current *Pumbaa* release.

3. Run the test suites and generate the documentation and other files.

   ```
   make -s -j8 test-all-boards
   make -s -j8 release-test
   ```

4. Commit the generated files.

5. Generate files for Arduino and PlatformIO releases. The generated archives and Arduino manifests are copied to the release repository.

   ```
   make -s release
   ```

6. Add, commit and push the Pumbaa Arduino releases in the release repository.

   ```
   (cd ../pumbaa-releases && \
    git add arduino/*/*.zip platformio/*.zip && \
    git commit && \
    git push origin master)
   ```

7. Start a http server used to download package manifests in the Arduino IDE.

```
(cd make/arduino && python -m SimpleHTTPServer)
```

8. Start the Arduino IDE and add these URL:s in Preferences.

```
http://localhost:8000/esp32/package_pumbaa_esp32_index.json
http://localhost:8000/sam/package_pumbaa_sam_index.json
```

9. Install all four packages and run the blink example for each one of them.
10. Commit the manifests, tag the commit with `<major>.<minor>.<revision>` and push.

```
git commit
git tag <major>.<minor>.<revision>
git push origin master
```

11. Add, commit and push the Pumbaa Arduino package manifests in the release repository.

```
(cd ../pumbaa-releases && \
 git add arduino/*/*.json && \
 git commit && \
 git push origin master)
```

12. Done.

Boards

The boards supported by *Pumbaa*.

Arduino Due

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/arduino_due.html

Drivers

Supported drivers for this board.

- *Adc*
- *Can*
- *Dac*
- *Ds18b20*
- *EepromI2C*
- *Exti*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*

- *Pin*
- *Sd*
- *Spi*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	223992	37320
default-configuration	352248	76334

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	1
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	1
CONFIG_PUMBAA_CLASS_DAC	1
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	0
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	1
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	0
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	0
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	1
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	0
CONFIG_PUMBAA_HEAP_SIZE	32768
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	0

Continued on next page

Table 1.1 – continued from previous page

Name	Value
CONFIG_PUMBAA_MODULE_SSL	0
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

Cygwin

ESP-01

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/esp01.html

Drivers

Supported drivers for this board.

- *Ds18b20*
- *EepromI2C*
- *esp_wifi*
- *Exti*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Spi*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	454573	63084
default-configuration	522992	78392

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	0
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	0
CONFIG_PUMBAA_CLASS_DAC	0
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	1
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	1
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	0
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	0
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	0
CONFIG_PUMBAA_HEAP_SIZE	24576
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	0
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

ESP-12E Development Board

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/esp12e.html

Drivers

Supported drivers for this board.

- *Ds18b20*
- *EepromI2C*
- *esp_wifi*
- *Exti*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Spi*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	454705	63152
default-configuration	530136	79256

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	0
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	0
CONFIG_PUMBAA_CLASS_DAC	0
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	1
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	1
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	1
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0

Continued on next page

Table 1.3 – continued from previous page

Name	Value
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	0
CONFIG_PUMBAA_HEAP_SIZE	24576
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	0
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

ESP32-DevKitC

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/esp32_devkitc.html

Drivers

Supported drivers for this board.

- *Adc*
- *Can*
- *Dac*
- *Ds18b20*
- *EepromI2C*
- *esp_wifi*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Spi*
- *Uart*
- *Ws2812*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	335860	87708
default-configuration	709116	189832

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	1
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	1
CONFIG_PUMBAA_CLASS_DAC	1
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	1
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	0
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	1
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	1
CONFIG_PUMBAA_EMACS	1
CONFIG_PUMBAA_HEAP_SIZE	65536
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	1
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

Linux

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/linux.html

Drivers

Supported drivers for this board.

- *Adc*
- *Can*
- *Dac*
- *Ds18b20*
- *EepromI2C*
- *Exti*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Sd*
- *Spi*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	784937	1603784
default-configuration	1623563	1824896

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	1
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	1
CONFIG_PUMBAA_CLASS_DAC	1

Continued on next page

Table 1.5 – continued from previous page

Name	Value
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	0
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	1
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	1
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	1
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	1
CONFIG_PUMBAA_HEAP_SIZE	(1024 * 1024)
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	1
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

Nano32

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/nano32.html

Drivers

Supported drivers for this board.

- *Adc*
- *Can*
- *Dac*
- *Ds18b20*
- *EepromI2C*

- *esp_wifi*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Spi*
- *Uart*
- *Ws2812*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	335752	87708
default-configuration	709004	189832

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	1
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	1
CONFIG_PUMBAA_CLASS_DAC	1
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	1
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	0
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	1
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1

Continued on next page

Table 1.6 – continued from previous page

Name	Value
CONFIG_PUMBAA_CLASS_WS2812	1
CONFIG_PUMBAA_EMACS	1
CONFIG_PUMBAA_HEAP_SIZE	65536
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	1
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

NodeMCU

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/nodemcu.html

Drivers

Supported drivers for this board.

- *Ds18b20*
- *EepromI2C*
- *esp_wifi*
- *Exti*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Spi*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	454717	63148
default-configuration	530208	79256

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	0
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	0
CONFIG_PUMBAA_CLASS_DAC	0
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	1
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	1
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	1
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0
CONFIG_PUMBAA_CLASS_SPI	1
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	0
CONFIG_PUMBAA_HEAP_SIZE	24576
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	1
CONFIG_PUMBAA_MODULE_SSL	0
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

Particle IO Photon

Pinout and general information

Simba documentation: http://simba-os.readthedocs.io/en/latest/boards/photon.html

Drivers

Supported drivers for this board.

- *Ds18b20*
- *EepromI2C*
- *Flash*
- *I2C*
- *I2CSoft*
- *Owi*
- *Pin*
- *Uart*

Memory usage

Below is the memory usage of two applications:

- The minimal-configuration application is configured to only include the bare minimum of functionality.
- The default-configuration application is built with the default configuration.

Application	Flash	RAM
minimal-configuration	219488	37652
default-configuration	276320	72786

Default configuration

Default Standard Library configuration.

Name	Value
CONFIG_PUMBAA_CLASS_ADC	0
CONFIG_PUMBAA_CLASS_BOARD	1
CONFIG_PUMBAA_CLASS_CAN	0
CONFIG_PUMBAA_CLASS_DAC	0
CONFIG_PUMBAA_CLASS_DS18B20	1
CONFIG_PUMBAA_CLASS_EEPROM_I2C	1
CONFIG_PUMBAA_CLASS_ESP_WIFI	0
CONFIG_PUMBAA_CLASS_EVENT	1
CONFIG_PUMBAA_CLASS_EXTI	0
CONFIG_PUMBAA_CLASS_FLASH	1
CONFIG_PUMBAA_CLASS_HTTP_SERVER	0
CONFIG_PUMBAA_CLASS_HTTP_SERVER_WEBSOCKET	0
CONFIG_PUMBAA_CLASS_I2C	1
CONFIG_PUMBAA_CLASS_I2C_SOFT	1
CONFIG_PUMBAA_CLASS_OWI	1
CONFIG_PUMBAA_CLASS_PIN	1
CONFIG_PUMBAA_CLASS_QUEUE	1
CONFIG_PUMBAA_CLASS_SD	0
CONFIG_PUMBAA_CLASS_SPI	0

Continued on next page

Table 1.8 – continued from previous page

Name	Value
CONFIG_PUMBAA_CLASS_TIMER	1
CONFIG_PUMBAA_CLASS_UART	1
CONFIG_PUMBAA_CLASS_WS2812	0
CONFIG_PUMBAA_EMACS	0
CONFIG_PUMBAA_HEAP_SIZE	32768
CONFIG_PUMBAA_MAIN_FRIENDLY_REPL	1
CONFIG_PUMBAA_MAIN_REBOOT_AT_EXIT	1
CONFIG_PUMBAA_MODULE_SELECT	1
CONFIG_PUMBAA_MODULE_SOCKET	0
CONFIG_PUMBAA_MODULE_SSL	0
CONFIG_PUMBAA_OS_FORMAT	1
CONFIG_PUMBAA_OS_SYSTEM	1
CONFIG_PUMBAA_PING	1
CONFIG_PUMBAA_SYS_LOCK	1
CONFIG_PUMBAA_SYS_REBOOT	1
CONFIG_PUMBAA_THRD	1

Examples

Below is a list of simple examples that are useful to understand the basics of *Pumbaa*.

There are a lot more examples and unit tests on Github that shows how to use most of the *Pumbaa* modules.

Blink

About

Turn a LED on and off periodically once a second. This example illustrates how to use digital pins and sleep a thread.

Source code

```
#
# @section License
#
# The MIT License (MIT)
#
# Copyright (c) 2016-2017, Erik Moqvist
#
# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy,
# modify, merge, publish, distribute, sublicense, and/or sell copies
# of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
```

```
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# This file is part of the Pumbaa project.
#


import time
import board
from drivers import Pin

LED = Pin(board.PIN_LED, Pin.OUTPUT)

while True:
    LED.toggle()
    time.sleep(0.5)
```

The source code can also be found on Github in the examples/blink folder.

Build and run

Build and upload the application.

```
$ cd examples/blink
$ make -s BOARD=<board> upload
```

Ds18b20

About

Read and print the room temperature measured with a DS18B20 sensor.

Source code

```
#
# @section License
#
# The MIT License (MIT)
#
# Copyright (c) 2016-2017, Erik Moqvist
#
# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy,
# modify, merge, publish, distribute, sublicense, and/or sell copies
# of the Software, and to permit persons to whom the Software is
```

```
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# This file is part of the Pumbaa project.
#

import binascii
import board
from drivers import Ds18b20, Owi

OWI = Owi(board.PIN_GPIO17)
DS18B20 = Ds18b20(OWI)

# Search for devices on the OWI bus.
print('Number of sensors:', OWI.search())

while True:
    # Taking a new temperature sample.
    DS18B20.convert()

    for device_id in DS18B20.get_devices():
        print('Device id: {}, Temperature: {}'.format(
            binascii.hexlify(device_id),
            DS18B20.get_temperature(device_id)))
```

The source code can also be found on Github in the examples/ds18b20 folder.

Build and run

Build and upload the application.

```
$ cd examples/ds18b20
$ make -s BOARD=<board> upload
```

Hello World

About

This application prints “Hello world!” to standard output.

Source code

```
#
# @section License
#
# The MIT License (MIT)
#
# Copyright (c) 2016-2017, Erik Moqvist
#
# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy,
# modify, merge, publish, distribute, sublicense, and/or sell copies
# of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# This file is part of the Pumbaa project.
#


print('Hello world!')
```

The source code can also be found on Github in the examples/hello_world folder.

Build and run

Build and run the application.

```
$ cd examples/hello_world
$ make -s BOARD=<board> run
...
Hello world!
$
```

Interactive

About

This application is a Python interpreter!

When the application starts it tries to run the script `main.py` from the file system. After the script ends the *Python* interactive interpreter is started.

The serial port baudrate is 38400.

Example script

Here is an example of how to write a script `main.py` using the interpreter.

1. Start the serial monitor.
2. Create `main.py` and write `print('Hello World!\n')` to it. This file will be executed everytime the board starts.

```
MicroPython v1.8.3-88-gf98bb2d on 2016-09-17; Arduino Due with SAM3X8E
Type "help()" for more information.
>>> with open("main.py", "w") as f:
...     f.write("print('Hello World!\n')")
>>>
```

3. Restart the board and you'll see `Hello World!` on the screen!

```
Hello World!

MicroPython v1.8.3-88-gf98bb2d on 2016-09-17; Arduino Due with SAM3X8E
Type "help()" for more information.
>>>
```

4. Done!

The example can be found on Github in the examples/interactive folder.

Build and run

Build and run the application.

```
$ cd examples/interactive
$ make -s BOARD=arduino_due run
...
MicroPython v1.8.3-88-gf98bb2d on 2016-09-17; Arduino Due with SAM3X8E
Type "help()" for more information.
>>>
```

Select

About

Setup three channels, add them to a *poll object* and wait for events to occur.

Three channels are polled:

1. An *event channel* that waits for a button to be pressed.
2. A *queue channel* that the string "foo" is written to in the script.
3. A *socket channel* waiting for UDP packets.

NOTE: Change the UDP configuration to match your setup.

Source code

```
#
# @section License
#
# The MIT License (MIT)
#
# Copyright (c) 2016-2017, Erik Moqvist
#
# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy,
# modify, merge, publish, distribute, sublicense, and/or sell copies
# of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# This file is part of the Pumbaa project.
#


import select
import socket
from sync import Event, Queue
from drivers import Exti
import board

BUTTON_PIN = board.PIN_GPIO0
UDP_ADDRESS = '192.168.1.103'
UDP_PORT = 30303

button = Event()
exti = Exti(BUTTON_PIN, Exti.FALLING, button, 0x1)

queue = Queue()

udp = socket.socket(type=socket.SOCK_DGRAM)
udp.bind((UDP_ADDRESS, UDP_PORT))

poll = select.poll()
poll.register(button)
poll.register(queue)
poll.register(udp)

queue.write('foo')
```

```
while True:
    [(channel, eventmask)] = poll.poll()

    if channel is button:
        button.read(0x1)
        print("button")
    elif channel is queue:
        print("queue:", queue.read(3))
    elif channel is udp:
        print("udp:", udp.recv(1024))
```

The source code can also be found on Github in the examples/select folder.

Build and run

Build and upload the application.

```
$ cd examples/blink
$ make -s BOARD=esp12e CDEFS_EXTRA="CONFIG_START_NETWORK_INTERFACE_WIFI_SSID=ssid␣
↪CONFIG_START_NETWORK_INTERFACE_WIFI_PASSWORD=password" run
...
queue: b'foo'
```

At this point the application is waiting for an event to occur. Send a UDP packet to it from your PC using Python.

```
>>> import socket
>>> udp = socket.socket(type=socket.SOCK_DGRAM)
>>> udp.sendto('bar', ('192.168.1.103', 30303))
```

The written packet is received by the application and printed.

```
udp: b'bar'
```

TCP Server

About

Create a listening TCP socket waiting for a client to connect. The server reads one byte at a time from the socket and writes it back to the client.

Source code

```
#
# @section License
#
# The MIT License (MIT)
#
# Copyright (c) 2016-2017, Erik Moqvist
#
# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy,
```

```
# modify, merge, publish, distribute, sublicense, and/or sell copies
# of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# This file is part of the Pumbaa project.
#

import time
import socket
from drivers import esp_wifi

SSID = 'Qvist2'
PASSWORD = 'maxierik'
IP = '192.168.0.7'
PORT = 9000

esp_wifi.set_op_mode(esp_wifi.OP_MODE_STATION)
esp_wifi.station_init(SSID, PASSWORD)
esp_wifi.station_connect()

while esp_wifi.station_get_status() != 'got-ip':
    print('Waiting for WiFi connection...')
    time.sleep(2)

listener = socket.socket()
listener.bind((IP, PORT))
listener.listen(1)

while True:
    print('Listening for a client to connect.')
    client, address = listener.accept()
    print('Accepted client with address', address)

    while True:
        data = client.recv(1)
        print(data)
        if not data:
            print('Socket closed.')
            break
        client.send(data)
```

The source code can also be found on Github in the examples/tcp_server folder.

Build and run

Build and upload the application.

```
$ cd examples/tcp_server
$ make -s BOARD=<board> upload
```

Library Reference

Here is a list of builtin modules in *Pumbaa*.

Standard Library modules

Python Standard Library modules. Only a subset of the cPython module functionality is implemented in MicroPython.

`array` — Array

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

---

class `array.`**`array`**(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

If given a list or string, the *initializer* is passed to the new array's *fromlist()*, *frombytes()*, or *fromunicode()* method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the *extend()* method.

`array.`**`append`**(*x*)

Append a new item with value *x* to the end of the array.

`array.`**`extend`**(*iterable*)

Append items from *iterable* to the end of the array. If *iterable* is another array, it must have exactly the same type code; if not, *TypeError* will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

binascii — Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations.

---

binascii.**a2b_base64**(*string*)
Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

binascii.**b2a_base64**(*data*)
Convert binary *data* to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The newline is added because the original use case for this function was to feed it a series of 57 byte input lines to get output lines that conform to the MIME-base64 standard. Otherwise the output conforms to RFC 3548.

binascii.**crc32**(*data*[, *value*])
Compute CRC-32, the 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
>>> binascii.crc32(b"hello world")
222957957
>>> crc = binascii.crc32(b"hello")
>>> binascii.crc32(b" world", crc)
222957957
```

binascii.**hexlify**(*data*)
Return the hexadecimal representation of the binary *data*. Every byte of data is converted into the corresponding 2-digit hex representation. The returned bytes object is therefore twice as long as the length of data.

binascii.**unhexlify**(*hexstr*)
Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of `hexlify()`. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise an `Error` exception is raised.

cmath — Mathematical functions for complex numbers

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

---

cmath.**phase**(*x*)
Return the phase of *x* (also known as the argument of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range *[-π, π]*, and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592
>>> phase(complex(-1.0, -0.0))
-3.14159
```

cmath.**polar**(*x*)
Return the representation of *x* in polar coordinates. Returns a pair `(r, phi)` where r is the modulus of *x* and phi is the phase of x. `polar(x)` is equivalent to `(abs(x), phase(x))`.

cmath.**rect**(*r*, *phi*)
Return the complex number x with polar coordinates *r* and *phi*. Equivalent to `r * (math.cos(phi) + math.sin(phi)*1j)`.

cmath.**exp**(*x*)
Return the exponential value `e**x`.

cmath.**log**(*x*)
Returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞, continuous from above.

cmath.**sqrt**(*x*)
Return the square root of *x*. This has the same branch cut as `log()`.

cmath.**cos**(*x*)
Return the cosine of *x*.

cmath.**sin**(*x*)
Return the sine of *x*.

cmath.**e**()
The mathematical constant e, as a float.

cmath.**pi**()
The mathematical constant π, as a float.

`collections` — High-performance container datatypes

---

collections.**namedtuple**(*typename*, *field_names*)
Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with *typename* and *field_names*) and a helpful `__repr__()` method which lists the tuple contents in a name=value format.

The *field_names* are a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`. Alternatively, *field_names* can be a sequence of strings such as `['x', 'y']`.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

class collections.**OrderedDict**([*items*])
Return an instance of a dict subclass, supporting the usual *dict* methods. An OrderedDict is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

popitem(*last=True*)
The `popitem()` method for ordered dictionaries returns and removes a `(key, value)` pair. The pairs are returned in LIFO order if last is true or FIFO order if false.

`hashlib` — Secure hashes and message digests

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA256.

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use `sha256()` to create a SHA-256 hash object. You can now feed this object with bytes-like objects (normally bytes) using the `update()` method. At any point you can ask it for the digest of the concatenation of the data fed to it so far using the `digest()` or `hexdigest()` methods.

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}
 ↪Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0fK\x94\x06
 ↪'
```

`io` — Core tools for working with streams

The io module provides Python's main facilities for dealing with various types of I/O. There are two main types of I/O: text I/O and binary I/O. These are generic categories, and various backing stores can be used for each of them.

`io.`**open**(*name*, *mode='r'*)
Open a file named *name*. *mode* is a combination of the characters `'rwbta'`.

class `io.`**FileIO**(...)
This is type of a file open in binary mode, e.g. using `open(name, "rb")`. You should not instantiate this class directly.

class `io.`**TextIOWrapper**(...)
This is type of a file open in text mode, e.g. using `open(name, "rt")`. You should not instantiate this class directly.

class `uio.`**StringIO**([*string*])

class `uio.`**BytesIO**([*string*])
In-memory file-like objects for input/output. `StringIO` is used for text-mode I/O (similar to a normal file opened with `"t"` modifier). `BytesIO` is used for binary-mode I/O (similar to a normal file opened with `"b"` modifier). Initial contents of file-like objects can be specified with string parameter (should be normal string for `StringIO` or bytes object for `BytesIO`).

read(*size=-1*)
Read up to *size* bytes fro mthe file. If *size* is negative or `None`, read until end of file.

readall()
Read until end of file.

readline()
Read a line.

write(*b*)
Write *b* to the file.

seek(*offset*[, *whence*])
Move the file cursor *offset* bytes relative to beginning of the file (`0`), current position (`1`) or end of the file (`2`). The default value of *whence* is `0`.

flush()
Flush all buffers.

close()
Close the file.

getvalue()
Get the current contents of the underlying buffer which holds data.

`json` — JSON encoder and decoder

JSON (JavaScript Object Notation) is a lightweight data interchange format inspired by JavaScript object literal syntax.

`json.`**dumps**(*obj*)
Serialize *obj* to a JSON formatted str using this conversion table.

`json.`**loads**(*s*)
Deserialize *s* (a str instance containing a JSON document) to a Python object using this conversion table.

`math` — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

`math.`**ceil**(*x*)
Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__()`, which should return an Integral value.

`math.`**copysign**(*x*, *y*)
Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns -1.0.

`math.`**fabs**(*x*)
Return the absolute value of *x*.

`math.`**floor**(*x*)
Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__()`, which should return an Integral value.

`math.`**fmod**(*x*, *y*)
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer n such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising

1e100. For this reason, function fmod() is generally preferred when working with floats, while Python's x % y is preferred when working with integers.

math.**frexp**(*x*)
Return the mantissa and exponent of *x* as the pair (m, e). m is a float and e is an integer such that x == m * 2**e exactly. If *x* is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1. This is used to "pick apart" the internal representation of a float in a portable way.

math.**isinf**(*x*)
Return True if *x* is a positive or negative infinity, and False otherwise.

math.**isnan**(*x*)
Return True if *x* is a NaN (not a number), and False otherwise.

math.**ldexp**(*x*, *i*)
Return x * (2**i). This is essentially the inverse of function *frexp()*.

math.**modf**(*x*)
Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

math.**trunc**(*x*)
Return the Real value *x* truncated to an Integral (usually an integer). Delegates to x.__trunc__().

math.**log**(*x*[, *base*])
With one argument, return the natural logarithm of *x* (to base e).

With two arguments, return the logarithm of *x* to the given *base*, calculated as log(x)/log(base).

math.**pow**(*x*, *y*)
Return *x* raised to the power *y*. If both *x* and *y* are finite, *x* is negative, and *y* is not an integer then pow(x, y) is undefined, and raises ValueError.

Unlike the built-in ** operator, math.pow() converts both its arguments to type float. Use ** or the built-in pow() function for computing exact integer powers.

math.**sqrt**(*x*)
Return the square root of *x*.

math.**acos**(*x*)
Return the arc cosine of *x*, in radians.

math.**asin**(*x*)
Return the arc sine of *x*, in radians.

math.**atan**(*x*)
Return the arc tangent of *x*, in radians.

math.**atan2**(*y*, *x*)
Return atan(y/x), in radians. The result is between -π and π. The vector in the plane from the origin to point (x, y) makes this angle with the positive X axis. The point of atan2() is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, atan(1) and atan2(1, 1) are both pi/4, but atan2(-1, -1) is -3*pi/4.

math.**cos**(*x*)
Return the cosine of *x* radians.

math.**sin**(*x*)
Return the sine of *x* radians.

math.**tan**(*x*)
Return the tangent of *x* radians.

math.**degrees**(*x*)
Convert angle *x* from radians to degrees.

math.**radians**(*x*)
Convert angle *x* from degrees to radians.

math.**pi**()
The mathematical constant π = 3.141592..., to available precision.

math.**e**()
The mathematical constant e = 2.718281..., to available precision.

os — Miscellaneous operating system interfaces

This module provides a portable way of using operating system dependent functionality.

---

os.**uname**()
Returns information identifying the current operating system. The return value is an object with five attributes:

- sysname - operating system name
- nodename - name of machine on network (implementation-defined)
- release - operating system release
- version - operating system version
- machine - hardware identifier

os.**getcwd**()
Return a string representing the current working directory.

os.**listdir**(*path='.'*)
Return a list containing the names of the entries in the directory given by path. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory.

os.**stat**()
Return a *stat_result* object for this entry.

os.**system**(*command*)
Returns the output of given file system command. Raises `OSError` if the command is missing or fails to execute.

os.**format**(*path*)
Format file system at given path. All data in the file system will be lost.

random — Generate pseudo-random numbers

This module implements pseudo-random number generators for various distributions.

---

random.**seed**(*a=None*)
Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).

If *a* is an int, it is used directly.

random.**getrandbits**(*k*)
Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*[, *step*])
Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

The positional argument pattern matches that of `range()`. Keyword arguments should not be used because the function may use them in unexpected ways.

```
>>> random.randrange(10)            # Integer from 0 to 9
7
>>> random.randrange(0, 101, 2)     # Even integer from 0 to 100
26
```

random.**randint**(*a*, *b*)
Return a random integer N such that *a* <= N <= *b*. Alias for `randrange(a, b+1)`.

random.**choice**(*seq*)
Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises IndexError.

```
>>> random.choice('abcdefghij')     # Single random element
'c'
```

random.**random**()
Return the next random floating point number in the range `[0.0, 1.0]`.

```
>>> random.random()                 # Random float x, 0.0 <= x < 1.0
0.374448
```

random.**uniform**(*a*, *b*)
Return a random floating point number N such that *a* <= N <= *b* for *a* <= *b* and *b* <= N <= *a* for *b* < *a*.

The end-point value *b* may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

```
>>> random.uniform(1, 10)           # Random float x, 1.0 <= x < 10.0
1.180014
```

`select` — Waiting for I/O completion

This module provides access to the *poll()* function available in most operating systems. It only works for the Simba channels; *events*, *queues* and *sockets*.

---

select.**poll**()
Returns a polling object, which supports registering and unregistering channels, and then polling them for I/O events.

class select.**poll**
The poll class.

```
>>> from sync import Queue
>>> poll = select.poll()
>>> queue = Queue()
>>> poll.register(queue)
>>> poll.poll(1.0)              # Timeout since no data is available in the queue.
[]
>>> queue.write(b'hello')
>>> poll.poll()                 # Data is available in the queue.
[(<0x20034050>, 1)]
>>> poll.poll()[0][0].read(5)   # Read the available data.
b'hello'
```

register(*channel*[, *eventmask*])
Register given *channel* with the polling object. Future calls to the *poll()* method will then check whether the channel has any pending I/O events.

eventmask is an optional bitmask describing the type of events you want to check for, and can be currently only be `POLLIN`.

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

unregister(*channel*)
Remove given *channel* being tracked by a polling object.

Attempting to remove a file descriptor that was never registered causes a *KeyError* exception to be raised.

modify(*channel*, *eventmask*)
Modifies an already registered channel. This has the same effect as *register(channel, eventmask)*. Attempting to modify a channel that was never registered causes an *IOError* exception with errno *ENOENT* to be raised.

poll([*timeout*])
Polls the set of registered channels, and returns a possibly-empty list containing `(channel, event)` 2-tuples for the descriptors that have events or errors to report. An empty list indicates that the call timed out and no channel had any events to report. If *timeout* is given, it specifies the length of time in seconds which the system will wait for events before returning. If *timeout* is omitted, negative, or `None`, the call will block until there is an event for this poll object.

`select.`**POLLIN**
There is data to read.

`select.`**POLLHUP**
Hung up.

`socket` — Low-level networking interface

This module provides access to the BSD socket interface.

`socket.`**AF_INET**
This constant represent the address (and protocol) family, used for the first argument to *`socket.socket()`*.

`socket.`**SOCK_STREAM**

`socket.`**SOCK_DGRAM**
These constants represent the socket types, used for the second argument to *`socket.socket()`*.

socket.**getaddrinfo**(*host*, *port*[, *family*[, *socktype*[, *proto*[, *flags*]]]])

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as 'http', a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass NULL to the underlying C API.

The function returns a list of 5-tuples with the following structure:

```
(family, socktype, proto, canonname, sockaddr)
```

In these tuples, *family*, *socktype*, *proto* are all integers and are meant to be passed to *socket.socket()*. *canonname* will be a string representing the canonical name of the host if AI_CANONNAME is part of the flags argument; else canonname will be empty. sockaddr is a tuple describing a socket address, whose format depends on the returned family (a (address, port) 2-tuple for AF_INET, a (address, port, flow info, scope id) 4-tuple for AF_INET6), and is meant to be passed to the socket.connect() method.

The following example fetches address information for a hypothetical TCP connection to example.org on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, 0, 0, socket.IPPROTO_TCP)
[(2, 1, 6, '', ('93.184.216.34', 80))]
```

socket.**socket**([*family*[, *type*[, *proto*]]])

Create a new socket using the given address *family*, socket *type* and protocol number. The address family should be *socket.AF_INET*. The socket *type* should be *socket.SOCK_STREAM* (the default), *socket.SOCK_DGRAM*. The protocol number is usually zero and may be omitted in that case.

socket.**inet_aton**(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a string four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

If the IPv4 address string passed to this function is invalid, `socket.error` will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6.

socket.**inet_ntoa**(*packed_ip*)

Convert a 32-bit packed IPv4 address (a string four characters in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, `socket.error` will be raised. `inet_ntoa()` does not support IPv6.

class socket.**SocketType**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without flags argument instead.

accept()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

bind(*address*)

Bind the socket to address. The socket must not already be bound. The format is of *address* is `(ip_address, port)`.

close()
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

Note `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

connect(*address*)
Connect to a remote socket at address. The format is of *address* is `(ip_address, port)`.

listen(*backlog*)
Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

recv(*bufsize*)
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*.

recvfrom(*bufsize*)
Receive data from the socket. The return value is a pair `(string, address)` where string is a string representing the data received and address is the address of the socket sending the data.

recv_into(*buffer*[, *nbytes*])
Receive up to nbytes bytes from the socket, storing the data into a buffer rather than creating a new string. If nbytes is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received.

recvfrom_into(*buffer*[, *nbytes*])
Receive data from the socket, writing it into *buffer* instead of creating a new string. The return value is a pair `(nbytes, address)` where nbytes is the number of bytes received and address is the address of the socket sending the data.

send(*string*)
Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

sendall(*string*)
Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from string until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

sendto(*string*, *address*)

sendto(*string*, *flags*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. The optional flags argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of address depends on the address family — see above.)

shutdown(*how*)
Shut down one or both halves of the connection.

`ssl` — TLS/SSL wrapper for socket objects

Wrap sockets in TLS/SSL to encrypt the transport channel.

Warning: This module may lead to a false sense of security, as it is implemented by a TLS/SSL novice, me. Use with care!

Server side example:

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS)
>>> context.load_cert_chain("server.crt", keyfile="server.key")

>>> listener_sock = socket.socket()
>>> listener_sock.bind(('127.0.0.1', 10023))
>>> listener_sock.listen(5)

>>> client_sock, _ = listener_sock.accept()
>>> ssl_client_sock = context.wrap_socket(client_sock, server_side=True)

>>> ssl_client_sock.recv(5)
b'hello'
>>> ssl_client_sock.send(b'goodbye')
>>> ssl_client_sock.close()
>>> client_sock.close()
```

Client side example:

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS)
>>> context.load_verify_locations(cafile="server.crt")

>>> server_sock = socket.socket()
>>> server_sock.connect(('127.0.0.1', 10023))
>>> ssl_server_sock = context.wrap_socket(server_sock)

>>> ssl_server_sock.send(b'hello')
>>> ssl_server_sock.recv(7)
'goodbye'
>>> ssl_server_sock.close()
>>> server_sock.close()
```

---

class `ssl.`**`SSLContext`**(*protocol=ssl.PROTOCOL_TLS*)

Initialize given SSL context. A SSL context contains settings that lives longer than a socket.

`load_cert_chain`(*certfile*, *keyfile=None*)

Load given certificate chain into the context.

`load_verify_locations`(*cafile*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*.

`set_verify_mode`(*mode*)

Whether to try to verify other peers' certificates. Set *mode* to *CERT_NONE* to skip the verification, and *CERT_REQUIRED* to enable verification.

By default, server side sockets does not verify the client's certificate, while client side sockets do verify the server's certificate.

Load CA certificates with *load_verify_location()*.

`wrap_socket`(*sock*, *server_side=False*)

Wrap a normal TCP socket *sock* in this SSL context.

Performs the SSL handshake.

CERT_NONE
Do not verify the peer certificate.

CERT_REQUIRED
Verify the peer certificate.

class ssl.**SSLSocket**
This is a Python type object that represents the SSL socket.

close()
Close the SSL socket.

recv(*bufsize*)
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*.

send(*string*)
Send data *string* to the socket. The socket must be connected to a remote socket. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

get_server_hostname()
Returns the hostname of the server as a string.

cipher()
Returns the three-tuple with connection cipher information. For example `('TLS-RSA-WITH-AES-256-GCM-SHA384', 'TLSv1.1', -1)`

struct — Interpret strings as packed binary data

---

sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

---

sys.**exit**([*arg*])
Exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.

sys.**print_exception**()

sys.**path**
A list of strings that specifies the search path for modules.

sys.**argv**
The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not).

sys.**version**
A string containing the version number of the Python interpreter. Do not extract version information out of it, rather, use *version_info* and the functions provided by the platform module.

sys.**version_info**
A tuple containing the three components of the version number: major, minor and micro. The *version_info* value corresponding to the Python version 3.4 is (3, 4, 0).

sys.**implementation**
An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

name is the implementation's identifier, e.g. 'micropython'. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

version is a named tuple, in the same format as *sys.version_info*. It represents the version of the Python implementation. This has a distinct meaning from the specific version of the Python language to which the currently running interpreter conforms, which *sys.version_info* represents. For example, for Micropython 1.8 *sys.implementation* might be *(1, 8, 0)*, whereas *sys.version_info* would be *(3, 4, 0)*.

sys.**platform**
This string contains a platform identifier that can be used to append platform-specific components to *sys.path*, for instance.

sys.**byteorder**
An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

sys.**modules**
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail.

_thread — Low-level threading API

This module provides low-level primitives for working with multiple threads - multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided.

A thread created by this module is a Simba thread. The classes `sync.Event` and `sync.Queue` can be used for synchronization in addition to the simple locks.

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])
Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits.

_thread.**exit**()
Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

_thread.**allocate_lock**()
Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.**get_ident**()
Return the thread identifier of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.**stack_size**([*size*])
Return the thread stack size used when creating new threads.

class _thread.**LockType**

acquire([*waitflag*])
Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for

existence). If the integer *waitflag* argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is `True` if the lock is acquired successfully, `False` if not.

release()
Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

locked()
Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

time — Time access and conversions

This module provides various time-related functions.

---

time.**sleep**(*secs*)
Suspend execution of the calling thread for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the *sleep()* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

time.**sleep_ms**(*msecs*)
Same as *sleep()* but sleep for *msecs* number of milliseconds.

time.**sleep_us**(*usecs*)
Same as *sleep()* but sleep for *usecs* number of microseconds.

time.**time**()
Return the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

zlib — Compression compatible with gzip

Decompress compressed data.

---

zlib.**decompress**(*data*)
Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data.

MicroPython modules

MicroPython specific modules.

gc — Garbage Collector management

---

gc.**enable**()
Enable automatic garbage collection.

gc.**disable**()
Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using `gc.collect()`.

gc.**collect**()
Run a garbage collection.

gc.**mem_alloc**()
Returns the number of bytes allocated on the heap.

gc.**mem_free**()
Returns the number of bytes available on the heap.

micropython — Access and control MicroPython internals

---

micropython.**mem_info**([*verbose*])
Print information about currently used memory. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out the entire heap indicating which blocks are used and which are free.

micropython.**qstr_info**([*verbose*])
Print information about currently interned strings. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.**alloc_emergency_exception_buf**(*size*)
Allocate *size* bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to put it at the start of your main script (eg main.py) and then the emergency exception buffer will be active for all the code following it.

Pumbaa modules

This part of the Library Reference contains Pumbaa specific modules.

kernel — Kernel

The kernel package is the heart in *Simba*. It implements the thread scheduler.

Simba documentation: kernel

---

kernel.**sys_lock**()
Take the system lock. Turns off interrupts.

kernel.**sys_unlock**()
Release the system lock. Turns on interrupts.

kernel.**sys_reboot**()
Reboot the system. Sets all registers to their known, default values and restarts the application. Also known as a soft reset.

kernel.**thrd_yield**()
Put the currently executing thread on the ready list and reschedule.

This function is often called periodically from low priority work heavy threads to give higher priority threads the chance to execute.

kernel.**thrd_join**(*thrd*)
Wait for given thread to terminate.

kernel.**thrd_self**()
Get current thread's id.

kernel.**thrd_set_name**(*name*)
Set the name of the current thread to *name*.

kernel.**thrd_get_name**()
Returns the name of the current thread.

kernel.**thrd_get_by_name**(*name*)
Returns the identifier of given thread.

kernel.**thrd_set_log_mask**(*thrd*, *mask*)
Set the log mask of given thread.

kernel.**thrd_get_log_mask**()
Get the log mask of the current thread.

kernel.**thrd_set_prio**(*thrd*, *prio*)
Set the priority of given thread.

kernel.**thrd_get_prio**()
Get the priority of the current thread.

kernel.**thrd_set_global_env**(*name*, *value*)
Set the value of given environment variable. The pointers to given name and value are stored in the current global environment array.

kernel.**thrd_get_global_env**(*name*)
Get the value of given environment variable in the global environment array.

kernel.**thrd_set_env**(*name*, *value*)
Set the value of given environment variable. The pointers to given name and value are stored in the current threads' environment array.

kernel.**thrd_get_env**(*name*)
Returns the value of given environment variable. If given variable is not found in the current threads' environment array, the global environment array is searched. Returns `None` if the variable is missing.

class kernel.**Timer**(*timeout*, *event=None*, *mask=0x1*, *callback=None*, *flags=0*)
Instantiate a timer object from given arguemts. The timer expires *timeout* seconds after the timer has been started. When the timer expires given *callback* is called from interrupt context and *mask* is written to given *event channel*. Set *flags* to *PERIODIC* to create a periodic timer.

Simba documentation: kernel/timer

start()
Start the timer.

stop()
Stop the timer. If the timer is stopped before it has expired it will never exipre. This function has no effect if the timer has already expired.

PERIODIC
Pass this flag to *Timer* for periodic timers.

sync — Thread synchroniztion

Thread synchronization refers to the idea that multiple threads are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action.

Simba documentation: sync

---

class sync.**Event**
Initialize given event object.

Simba documentation: sync/event

read(*mask*)
Wait for an event to occur and return a mask of all active events.

write(*mask*)
Write given event(s) to the channel.

size()
Get the number of event(s) set in the channel.

class sync.**Queue**
Initialize given queue object.

Simba documentation: sync/queue

read(*size*)
Reads up to *size* number of bytes from the queue and returns them as a string. Raises an exception on error.

write(*string*)
Write given *string* to the queue. Returns the number of bytes written. Raises an exception on error.

size()
Get the number of bytes available to read.

drivers — Device drivers

This module contains device drivers.

The following classes are defined:

- *class Pin* – Digital pins
- *class Exti* – External interrupts
- *class Adc* – Analog to digital convertion
- *class Dac* – Digital to analog convertion
- *class Spi* – Serial peripheral interface
- *class Can* – Controller Area Network
- *class I2C* – I2C

- *class* *I2CSoft* – Software I2C
- *class* *Owi* – Onewire
- *class* *Ds18b20* – DS18B20 temperature
- *class* *Sd* – Secure Digital memory
- *class* *esp_wifi* – Espressif WiFi
- *class* *Uart* – Universal Asynchronous Receiver/Transmitter
- *class* *Flash* – Flash memory
- *class* *Ws2812* – WS2812 Neo Pixels
- *class* *EepromI2C* – I2C EEPROM

Simba documentation: drivers

---

class drivers.**Pin**(*device*, *mode*)

Create a pin object with given *device* and *mode*. The *device* is selected among the pins available in the *board* module. *mode* must be either *INPUT* or *OUTPUT*.

```
>>> led = Pin(board.PIN_LED, Pin.OUTPUT)
>>> led.write(1)
>>> led.toggle()
```

Simba documentation: drivers/pin

read()

Read the current pin value and return it as an integer. Returns 0 if the pin is low and 1 if the pin is high.

write(*value*)

Write the logic level *value* to the pin. *value* must be an object that can be converted to an integer. The value is either 0 or 1, where 0 is low and 1 is high.

toggle()

Toggle the pin output value (high/low).

set_mode(*mode*)

Set the pin mode to given mode *mode*. The mode must be either *INPUT* or *OUTPUT*.

INPUT

Input pin mode.

OUTPUT

Output pin mode.

class drivers.**Exti**(*device*, *trigger*, *channel=None*, *data=None*, *callback=None*)

Create an object handling interrupts on given *device*. *trigger* may be a combination of *RISING*, *FALLING* or *BOTH*. When an interrupt occurs given *callback* is called from interrupt context and *data* is written to given event or queue channel *channel*.

Event channel example.

```
>>> event = Event()
>>> exti = Exti(board.EXTI_D3, Exti.FALLING, event, 0x1)
>>> exti.start()
>>> event.read(0x1)        # Wait for an interrupt to occur.
>>> exti.stop()
```

Queue channel example.

```
>>> queue = Queue()
>>> exti = Exti(board.EXTI_D4, Exti.RISING, queue, b'1')
>>> exti.start()
>>> queue.read(1)         # Wait for an interrupt to occur.
b'1'
>>> exti.stop()
```

Simba documentation: drivers/exti

start()
Start the interrupt handler.

stop()
Stop the interrupt handler.

RISING
Trigger an interrupt on rising edges.

FALLING
Trigger an interrupt on falling edges.

BOTH
Trigger an interrupt on both rising and falling edges.

class drivers.**Adc**(*device*, *pin_device*, *reference*, *sampling_rate*)
Instansiate an Adc object with given *device* and *pin_device* devices. *reference* is the voltage reference and *sampling_rate* is the sampling frequency.

Here is an example of how to create a ADC driver object and convert an analog signal level to three digital samples with a sampling rate of 1 kHz.

```
>>> a0 = Adc(board.PIN_ADC0, board.PIN_A0, Adc.REFERENCE_VCC, 1000)
>>> a0.convert(3)
b'\x00\x01\x00\x02\x00\x03'
>>> array.array('h', a0.convert(3))
array('h', [1, 2, 3])
```

The equivalent asynchronous example.

```
>>> a0 = Adc(board.PIN_ADC0, board.PIN_A0, Adc.REFERENCE_VCC, 1000)
>>> a0.async_convert(3)
>>> array.array('h', a0.async_wait())
array('h', [1, 2, 3])
```

Simba documentation: drivers/adc

convert(*number_of_samples*)
Start a synchronous convertion of an analog signal to digital samples. This is equivalent to *async_convert()* + *async_wait()*, but in a single function call. Returns a bytes object where each sample is 2 bytes.

async_convert(*number_of_samples*)
Start an asynchronous convertion of analog signal to digital samples. Call *async_wait()* to wait for the convertion to complete.

async_wait()
Wait for an asynchronous convertion started with *async_convert()* to complete. Returns a bytes object where each sample is 2 bytes.

REFERENCE_VCC
Use VCC as reference.

class drivers.**Dac**(*devices*, *sampling_rate*)
Instansiate a Dac object. *devices* is either a list of DAC pin devices or a single DAC pin device. The DAC pin devices can be found in the *board* module, often named PIN_DAC0 and PIN_DAC1.

Here is an example of how to create a DAC driver and convert digital samples to an analog signal.

```
>>> dac = Dac(board.PIN_DAC0)
>>> dac.convert(b'\x01\x02\x03\x04')
```

Simba documentation: drivers/dac

convert(*samples*)
Start a synchronous convertion of digital samples to an analog signal. This function returns when all samples have been converted.

async_convert(*samples*)
Start an asynchronous convertion of digital samples to an analog signal. This function only blocks if the hardware is not ready to convert more samples. Call *async_wait()* to wait for an asynchronous convertion to finish.

async_wait()
Wait for an ongoing asynchronous convertion to finish.

class drivers.**Spi**(*device*, *slave_select*, *mode=MODE_MASTER*, *speed=SPEED_250KBPS*, *polarity=0*, *phase=0*)
Create a Spi object. Select the SPI device with *device* and slave select pin with *slave_select*. *mode* in one of *MODE_MASTER* and *MODE_SLAVE*. *speed* is only used by the master. *polarity* is the bus idle logic level. *phase* controls if sampling are done on falling or rising clock edges.

Here is an example of how to create a SPI driver and write 4 bytes to the slave.

```
>>> spi = Spi(board.SPI_0, board.PIN_D3)
>>> spi.start()
>>> spi.select()
>>> spi.write(b'\x01\x02\x03\x04')
>>> spi.deselect()
>>> spi.stop()
```

Simba documentation: drivers/spi

start()
Configures the SPI hardware with the settings of this object.

stop()
Deconfigures the SPI hardware if given driver currently ownes the bus.

take_bus()
In multi master application the driver must take ownership of the SPI bus before performing data transfers. Will re-configure the SPI hardware if configured by another driver.

give_bus()
In multi master application the driver must give ownership of the SPI bus to let other masters take it.

select()
Select the slave by asserting the slave select pin.

deselect()
Deselect the slave by de-asserting the slave select pin.

transfer(*write_buffer*[, *size*])
Simultaniuos read/write operation over the SPI bus. Writes data from *write_buffer* to the bus. The *size* argument can be used to transfer fewer bytes than the size of *write_buffer*. Returns the read data as a bytes object.

The number of read and written bytes are always equal for a transfer.

transfer_into(*read_buffer*, *write_buffer*[, *size*])
Same as *transfer()*, but the read data is written to *read_buffer*.

read(*size*)
Read *size* bytes from the SPI bus. Returns the read data as a bytes object.

read_into(*buffer*[, *size*])
Same as *read()*, but the read data is written to *buffer*.

write(*buffer*[, *size*])
Write *size* bytes from *buffer* to the SPI bus. Writes all data in *buffer* if *size* is not given.

MODE_MASTER
SPI master mode.

MODE_SLAVE
SPI slave mode.

SPEED_8MBPS

SPEED_4MBPS

SPEED_2MBPS

SPEED_1MBPS

SPEED_500KBPS

SPEED_250KBPS

SPEED_125KBPS
SPI bus speed. Only used if the driver is configured as master.

class drivers.**Can**(*device*, *speed=SPEED_500KBPS*)
Create a Can object. Select CAN device and speed with *device* and *speed*.

Here is an example of how to create a CAN driver, write a frame and then read a frame.

```
>>> can = Can(board.CAN_0)
>>> can.start()
>>> can.write(0x123, b'\x01\x02')
>>> can.read()
(id=0x32, data=b'\x34\x35\x36', flags=0)
>>> can.stop()
```

Simba documentation: drivers/can

start()
Starts the CAN device.

stop()
Stops the CAN device.

read()
Read a frame from the CAN bus and return it as a named tuple with three items; *id*, *data* and *flags*. *id* is the frame id as an integer. *flags* contains information about the frame format, and possibly additional information in the future. *data* is a bytes object of up to 8 bytes of read frame data.

write(*id*, *data*[, *flags*])
Write a frame with given *id* and *data* to the CAN bus. *id* is an integer and *data* is a bytes object of up to 8 bytes. Set *FLAGS_EXTENDED_FRAME* in *flags* to write an extended frame (29 bits frame id), otherwise a standard frame is written.

SPEED_500KBPS
CAN bus speed.

FLAGS_EXTENDED_FRAME
Extended frame flag. A 29 bits frame id will be sent/received.

class drivers.**I2C**(*device*, *baudrate=BAUDRATE_100KBPS*, *address=-1*)
Create an I2C object. Select the I2C device with *device*. The bus baudrate *baudrate* is one of *BAUDRATE_1MBPS*, *BAUDRATE_400KBPS* and *BAUDRATE_100KBPS*. *address* is the slave address when this driver is a slave (only master is suported in the current version of the driver).

Here is an example of how to create a I2C obeject and scan the bus to find conected devices.

```
>>> i2c = I2C(0)
>>> i2c.start()
>>> i2c.scan()
[87, 104]
>>> i2c.stop()
```

Simba documentation: drivers/i2c

start()
Start the I2C driver. Configures the hardware.

stop()
Stop the I2C driver. Resets the hardware.

read(*address*, *size*)
Read *size* bytes from slave with address *address*.

read_into(*address*, *buffer*[, *size*])
Read `len(buffer)` bytes from slave with address *address* into *buffer*. Give the argument *size* to read fewer bytes than `len(buffer)`.

write(*address*, *buffer*[, *size*])
Write the buffer *buffer* to slave with address *address*.

scan()
Scan the bus and return a list of all found slave addresses.

BAUDRATE_1MBPS

BAUDRATE_400KBPS

BAUDRATE_100KBPS
I2C bus baudrate. Only used if the driver is configured as master.

class drivers.**I2CSoft**(*scl*, *sda*, *baudrate=50000*, *max_clock_stretching_sleep_us=1000000*, *clock_stretching_sleep_us=10000*)
Create an I2CSoft object. Select the I2C SCL and SDA pins with *scl* and *sda*. The bus baudrate is selected using the *baudrate* argument. *max_clock_stretching_sleep_us* and *clock_stretching_sleep_us* are timing configuration parameters.

Here is an example of how to create a I2CSoft obeject and scan the bus to find conected devices.

```
>>> i2c = I2CSoft(board.PIN_D3, board.PIN_D4)
>>> i2c.start()
```

```
>>> i2c.scan()
[87, 104]
>>> i2c.stop()
```

Simba documentation: drivers/i2c_soft

start()
Start the I2C soft driver. Configures the hardware.

stop()
Stop the I2C soft driver. Resets the hardware.

read(*address*, *size*)
Read *size* bytes from slave with address *address*.

read_into(*address*, *buffer*[, *size*])
Read `len(buffer)` bytes from slave with address *address* into *buffer*. Give the argument *size* to read fewer bytes than `len(buffer)`.

write(*address*, *buffer*[, *size*])
Write the buffer *buffer* to slave with address *address*.

scan()
Scan the bus and return a list of all found slave addresses.

class drivers.**Owi**(*pin_device*)
Create an Owi object with *pin_device* as the One Wire bus pin.

Here is an example of how to use the Owi class.

```
>>> owi = Owi(board.PIN_D3)
>>> owi.reset()
>>> owi.search()
2
>>> owi.get_devices()
[b'12345678', b'abcdefgh']
>>> owi.read(b'12345678', 3)
b'\x00\x01\x02'
>>> owi.write(b'12345678', b'\x00')
1
```

Simba documentation: drivers/owi

reset()
Send reset on One Wire bus.

search()
Search for devices on the One Wire bus. The device id of all found devices are stored and returned by *get_devices()*.

get_devices()
Returns a list of all devices found in the latest call to *search()*.

read(*device_id*, *size*)
Read *size* bytes from device with id *device_id*.

write(*device_id*, *buffer*[, *size*])
Write buffer *buffer* to device with id *device_id*. Give *size* to write fewer bytes than the buffer size.

class drivers.**Ds18b20**(*owi*)
Create a Ds18b20 object.

Here is an example of how to use the Ds18b20 class.

```
>>> owi = Owi(board.PIN_D3)
>>> owi.search()
>>> ds18b20 = Ds18b20(owi)
>>> ds18b20.get_devices()
[b'(2345678']
>>> ds18b20.convert()
>>> ds18b20.get_temperature(b'(2345678')
20.5
```

Simba documentation: drivers/ds18b20

convert()
Start temperature convertion on all sensors. A convertion takes about one second to finish.

get_devices()
Returns a list of all DS18B20 devices found by the latest call to *Owi.search()*.

get_temperature(*device_id*)
Get the temperature for given device identity. Reads the latest converted sample for the device with id *device_id*. Call *convert()* before calling this function to get the current temperature.

class drivers.**Sd**(*spi*)
Create a Sd object with given SPI driver.

Here is an example of how to create a SD and read the CID.

```
>>> sd = Sd(spi)
>>> sd.start()
>>> print(sd.read_cid())
(mid=2, oid=b'TM', pnm=b'SA04G', prv=22, psn=-681299654, mdt=60416, crc=107)
>>> sd.stop()
```

Simba documentation: drivers/sd

start()
Configures the SD card driver. This resets the SD card and performs the initialization sequence.

stop()
Deconfigures the SD card driver.

read_cid()
Read card CID register and return it. The CID contains card identification information such as Manufacturer ID, Product name, Product serial number and Manufacturing date.

The return value is an object with 7 attributes:

- mid - manufacturer ID
- oid - OEM/Application ID
- pnm - Product name
- prv - Product revision
- psn - Product serial number
- mdt - Manufacturing date
- crc - CRC7 checksum

read_csd()
Read card CSD register and return it. The CSD contains that provides information regarding access to the card's contents.

The return value is an object with 29 attributes for version 1 cards and 24 attributes for version 2 cards:

- ...

read_block(*block*)
Read given block from SD card and returns it as a bytes object.

read_block_into(*block*, *buffer*)
Same as *read_block()*, but the read data is written to *buffer*.

write_block(*block*, *buffer*)
Write *buffer* to given block.

class drivers.**esp_wifi**
This class is a singleton and can not be instanciated. It configures the Espressif WiFi stack.

An example of how to connect to a WiFi network:

```
>>> esp_wifi.set_op_mode(esp_wifi.OP_MODE_STATION)
>>> esp_wifi.station_init('ssid', 'password')
>>> esp_wifi.station_connect()
>>> esp_wifi.station_get_status()
'got-ip'
>>> esp_wifi.station_get_ip_info()
(address='192.168.0.5', netmask='255.255.255.0', gateway='192.168.0.1')
```

An example of how to setup a SoftAP:

```
>>> esp_wifi.set_op_mode(esp_wifi.OP_MODE_SOFTAP)
>>> esp_wifi.softap_init('ssid', 'password')
>>> esp_wifi.softap_get_ip_info()
(address='192.168.4.1', netmask='255.255.255.0', gateway='192.168.4.1')
```

Simba documentation: drivers/esp_wifi

set_op_mode(*mode*)
Set the WiFi operating mode to *mode*. *mode* is one of *OP_MODE_STATION*, *OP_MODE_SOFTAP*, *OP_MODE_STATION_SOFTAP*.

get_op_mode()
Returns the current WiFi operating mode.

set_phy_mode(*mode*)
Set the WiFi physical mode (802.11b/g/n) to one of *PHY_MODE_11B*, *PHY_MODE_11G* and *PHY_MODE_11N*.

get_phy_mode()
Returns the physical mode (802.11b/g/n).

softap_init(*ssid*, *password*)
Initialize the WiFi SoftAP interface with given *ssid* and *password*.

softap_set_ip_info(*info*)
Set the ip address, netmask and gateway of the WiFi SoftAP. The info object *info* is a three items tuple of address, netmask and gateway strings in IPv4 format.

softap_get_ip_info()
Returns a three items tuple of the SoftAP ip address, netmask and gateway.

softap_get_number_of_connected_stations()
Returns the number of stations connected to the SoftAP.

softap_get_station_info()
Returns the information of stations connected to the SoftAP, including MAC and IP addresses.

softap_dhcp_server_start()
Enable the SoftAP DHCP server.

softap_dhcp_server_stop()
Disable the SoftAP DHCP server. The DHCP server is enabled by default.

softap_dhcp_server_status()
Returns the SoftAP DHCP server status.

station_init(*ssid*, *password*[, *info*])
Initialize the WiFi station.

station_connect()
Connect the WiFi station to the Access Point (AP). This function returns before a connection has been established. Call *station_get_status()* periodically until it retuns `got-ip` to ensure the WiFi station has been assigned an IP the the WiFi Access Point DHCP server.

station_disconnect()
Disconnect the WiFi station from the AP.

station_set_ip_info(*info*)
Set the ip address, netmask and gateway of the WiFi station. The info object *info* is a three items tuple of address, netmask and gateway strings in IPv4 format.

station_get_ip_info()
Returns the station ip address, netmask and gateway.

station_set_reconnect_policy(*policy*)
Set whether the station will reconnect to the AP after disconnection. Set *policy* to `True` to automatically reconnect and `False` otherwise.

station_get_reconnect_policy()
Check whether the station will reconnect to the AP after disconnection.

station_get_status()
Get the connection status of the WiFi station.

station_dhcp_client_start()
Enable the station DHCP client.

station_dhcp_client_stop()
Disable the station DHCP client.

station_dhcp_client_status()
Get the station DHCP client status.

OP_MODE_NULL

OP_MODE_STATION

OP_MODE_SOFTAP

OP_MODE_STATION_SOFTAP
WiFi operating modes.

PHY_MODE_11B

PHY_MODE_11G

PHY_MODE_11N
WiFi physical modes.

class drivers.**Uart**(*device*, *baudrate=115200*)
Create a Uart object. Select UART device and baudrate with *device* and *baudrate*.

Here is an example of how to create a UART driver, write data and then read data.

Instances of this class can be polled by the select module.

```
>>> uart = Uart(1)
>>> uart.start()
>>> uart.write(b'1234')
>>> uart.read(4)
b'5678'
>>> buf = bytearray(4)
>>> uart.read_into(buf, 3)
3
>>> buf
bytearray(b'901\x00')
>>> uart.stop()
```

Simba documentation: drivers/uart

start()
Starts the UART device. Configures the hardware.

stop()
Stops the UART device.

read(*size*)
Read *size* bytes from the UART. Returns the read data.

read_into(*buffer*[, *size*])
Read *size* bytes from the UART into given buffer *buffer*. Returns number of bytes read.

write(*buffer*[, *size*])
Write data in *buffer* to the UART. *size* may be used to send fewer bytes than the size of *buffer*. Returns number of bytes written.

size()
Returns the number of bytes in the input buffer.

class drivers.**Flash**(*device*)
Create a Flash object. Select flash device index with *device*. It is normally given as 0.

The flash address given to the instance methods is either relative to the flash start address or an absolute physical address in the CPU memory map. This is board dependent and it is not documented anywhere.

Here is an example of how to create a flash driver and use the erase, read and write methods.

```
>>> flash = Flash(0)
>>> flash.read(0x300000, 4)
b'5678'
>>> flash.erase(0x300000, 4)
>>> flash.write(0x300000, b'1234')
4
>>> buf = bytearray(8)
>>> flash.read_into(0x300000, buf, 4)
4
>>> buf
bytearray(b'1234\x00\x00\x00\x00')
```

Simba documentation: drivers/flash

read(*address*, *size*)
Read *size* bytes from given address *address* in the flash. Returns the read data.

read_into(*address*, *buffer*[, *size*])
Read *size* bytes from given address *address* in flash into given buffer *buffer*. Returns number of bytes read.

write(*address*, *buffer*[, *size*])
Write data in *buffer* to given address *address* in flash. *size* may be used to write fewer bytes than the size of *buffer*. Returns number of bytes written.

erase(*address*, *size*)
Erase *size* bytes in flash starting at given addess *address*.

class drivers.**Ws2812**(*pin_devices*)
Create a Ws2812 object.

Here is an example of how to create a Ws2812 driver and control a LED strip of 30 pixles.

```
>>> ws2812 = Ws2812(board.PIN_GPIO18)
>>> ws2812.write(30 * b'\xff\x00\x00')
```

Simba documentation: drivers/ws2812

write(*buffer*[, *number_of_pixels*])
Write GRB data from buffer *buffer* to the LED strip. Writes all data in *buffer* if *size* is not given.

class drivers.**EepromI2C**(*i2c*, *address*, *size*)
Create a I2C EEPROM object. *address* is the EEPROM I2C address, and *size* is the EEPROM size in bytes.

Here is an example of how to create an I2C EEPROM obeject and transfer data to and from the EEPROM using it.

```
>>> i2c = I2C(0)
>>> i2c.start()
>>> eeprom = EepromI2C(i2c, 0x57, 32768)
>>> eeprom.write(0, b'Hello World!')
>>> eeprom.read(0, 12)
b'Hello World!'
```

Simba documentation: drivers/eeprom_i2c

read(*address*, *size*)
Read *size* bytes from EEPROM address *address*.

read_into(*address*, *buffer*[, *size*])
Read `len(buffer)` bytes from EEPROM address *address* into *buffer*. Give the argument *size* to read fewer bytes than `len(buffer)`.

write(*address*, *buffer*[, *size*])
Write the buffer *buffer* to EEPROM address *address*.

inet — Internet networking

Internet network configuration and protocol implementation.

Simba documentation: inet

class inet.**HttpServer**(*ip_address*, *port*, *routes*, *on_no_route*)

Create a HTTP server object. The HTTP server opens a socket and binds to given *ip_address* and *port*. *routes* is a list of route tuples, where each route tuple contains a path and a callback function. *on_no_route* is called when a request is received for a path that is not found in *routes*.

Here is an example of a HTTP server with one route, `'/index.html'`. Enter `http://192.168.0.7:8000/index.html` in your webbrowser to get the index page.

```
>>> from inet import HttpServer
>>> def on_no_route(_, request):
>>>     return (request.path + ' not found.',
>>>             HttpServer.RESPONSE_CODE_404_NOT_FOUND,
>>>             HttpServer.CONTENT_TYPE_TEXT_PLAIN)

>>> def on_request_index(_, request):
>>>     return ('<html><body>Hello from Pumbaa!</body></html>', )

>>> routes = [('/index.html', on_request_index)]
>>> http_server = HttpServer('192.168.0.7', 80, routes, on_no_route)
>>> http_server.start()
```

Simba documentation: inet/http_server

start()

Start the HTTP server.

stop()

Stop the HTTP server.

wrap_ssl(*context*)

Wrap given HTTP server in SSL, to make it "secure". *context* is created with the SSL module.

This function must be called before *start()*.

class inet.**HttpServerWebSocket**(*connection*, *request*)

Create a HTTP server WebSocket object with given *connection* and *request*.

An example of how to use the HTTP server WebSocket class in a HTTP server route callback.

```
>>> from inet import HttpServerWebSocket
>>> def on_websocket_echo(connection, request):
>>>     ws = HttpServerWebSocket(connection, request)
>>>     while True:
>>>         message = ws.read()
>>>         print('Message:', message)
>>>         if not message:
>>>             break
>>>         ws.write(message)
```

read()

Read a message from the remote endpoint.

write(*buffer*)

Write *buffer* to the remote endpoint.

inet.**ping_host_by_ip_address**(*address*, *timeout*)

Ping host by IPv4 address *address*. Send an echo request packet to the host and wait for the echo reply packet. Only the ICMP header is transmitted, no extra payload data is added to the packet. Returns the round trip time in milliseconds.

Raises an *OSError* exception if no response is received within *timeout* seconds after the request is sent.

```
>>> inet.ping_host_by_ip_address("192.168.0.5", 2)
10
>>> inet.ping_host_by_ip_address("192.168.0.7", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError:
```

Simba documentation: inet/ping

`text` — Text parsing, editing and colorization

Simba documentation: text

`text`.**`emacs`**([*path*])
Start the Emacs text editor. Automatically opens file at *path* if given.

`board` — Board devices

The board module is board unique. This page shows the general structure of those modules.

`PIN_<ID>`
Pin devices.

`EXTI_<ID>`
External interrupt devices.

`PWM_<ID>`
PWM devices.

`ADC_<ID>`
ADC devices.

`DAC_<ID>`
DAC devices.

`FLASH_<ID>`
Flash devices.

`SPI_<ID>`
SPI devices.

License

The MIT License (MIT)

Copyright (c) 2014-2017, Erik Moqvist

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Videos

#7 Pumbaa: Emacs text editor on Nano32 (ESP32)!

Write a Python script in Emacs on a Nano32 (ESP32) and import it.

#6 Pumbaa: DAC ramp on Nano32 (ESP32)!

Measure the DAC output voltage on a Nano32 (ESP32).

#5 Pumbaa: Dual board CAN blink on Nano32 (ESP32) and Arduino Due.

Blink a LED on Nano32 by sending a CAN frame from an Arduino Due.

#4 Pumbaa: Room temperature (DS18B20).

Read and print the room temperature measured with a DS18B20 sensor.

#3 Pumbaa: Gource of the Pumbaa repository.

Gource visualizes the Pumbaa Git repository file tree over time. In this project the source, test and documentation was written simultaneously, a perfect school book example of software development.

#2 Pumbaa: Queue class unit testing on Nano32!

Unit testing is an important part of most embedded applications. This video shows how easy it is to run a test suite on the target hardware, in this case a Nano32 board.

In the video; the test suite can be seen in the left window and the compilation, upload and execution in the right window.

To execute the same test suite on native linux the `BOARD` variable shall be set to `linux`. Simple as that.

#1 Pumbaa: Blink on Nano32!

The very first Pumbaa video demonstrates the classic blink application. It's run on a Nano32 board that has a ESP32 MCU.

CHAPTER 2

Features

- MicroPython 3 language.
- *Python Standard Library* modules.
- *MicroPython modules* modules.
- A thin *Python wrapper* for Simba modules.

See the *Library Reference* for a full list of features.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

O

P

R

S

T

U

W

Z