Micropython-Stubber

Release 1.3.8

Jos Verlinde

Dec 02, 2021

CONTENTS:

CHAPTER

ONE

BOOST MICROPYTHON PRODUCTIVITY IN VSCODE

The intellisense and code linting that is so prevalent in modern editors, does not work out-of-the-gate for MicroPython projects. While the language is Python, the modules used are different from CPython , and also different ports have different modules and classes , or the same class with different parameters.

Writing MicroPython code in a modern editor should not need to involve keeping a browser open to check for the exact parameters to read a sensor, light-up a led or send a network request.

Fortunately with some additional configuration and data, it is possible to make the editors understand your flavor of MicroPython. even if you run a on-off custom firmware version.

In order to achieve this a few things are needed:

1. Stub files for the native / enabled modules in the firmware using PEP 484 Type Hints
2. Specific configuration of the VSCode Python extensions
3. Specific configuration of Pylint
4. Suppression of warnings that collide with the MicroPython principals or code optimization.

With that in place, VSCode will understand MicroPython for the most part, and help you to write code, and catch more errors before deploying it to your board.

Note that the above is not limited to VSCode and pylint, but it happens to be the combination that I use.

A lot of subs have already been generated and are shared on github or other means, so it is quite likely that you can just grab a copy to be productive in a few minutes.

For now you will need to *configure this by hand*, or use the `micropy cli` *tool*

1. The sister-repo [**MicroPython-stubs**][stubs-repo] contains [all stubs][all-stubs] I have collected with the help of others, and which can be used directly. That repo also contains examples configuration files that can be easily adopted to your setup.
2. A second repo [micropy-stubs repo][stubs-repo2] maintained by BradenM, also contains stubs but in a structure used and distributed by the *micropy-cli* tool. you should use micropy-cli to consume stubs in this repo.

The (stretch) goal is to create a VSCode add-in to simplify the configuration, and allow easy switching between different firmwares and versions.

1.1 Licensing

MicroPython-Stubber is licensed under the MIT license, and all contributions should follow this LICENSE.

CHAPTER

TWO

APPROACH TO COLLECTING STUB INFORMATION

The stubs are used by 3 components.

1. pylint
2. the VSCode Pylance Language Server
3. the VSCode Python add-in

These 3 tools work together to provide code completion/prediction, type checking and all the other good things. For this the order in which these tools use, the stub folders is significant, and best results are when all use the same order.

In most cases the best results are achieved by the below setup:

![stub processing order][]

1. **Your own source files**, including any libraries you add to your project. This can be a single libs folder or multiple directories. There is no need to run stubber on your source or libraries.
2. **The CPython common stubs**. These stubs are handcrafted to allow MicroPython script to run on a CPython system. There are only a limited number of these stubs and while they are not intended to be used to provide type hints, they do provide valuable information. Note that for some modules (such as the `gc`, `time` and `sys` modules) this approach does not work.
3. **Frozen stubs**. Most micropython firmwares include a number of python modules that have been included in the firmware as frozen modules in order to take up less memory. These modules have been extracted from the source code.
4. **Firmware Stubs**. For all other modules that are included on the board, [micropython-stubber] or [micropy-cli] has been used to extract as much information as available, and provide that as stubs. While there is a lot of relevant and useful information for code completion, it does unfortunately not provide all details regarding parameters that the above options may provide.

2.1 Stub collection process

- The **CPython common stubs** are periodically collected from the [micropython-lib][] or the [pycopy-lib][].
- The **Frozen stubs** are collected from the repos of [micropython][] + [micropython-lib][] and from the [loboris][] repo the methods to gather these differs per firmware family , and there are differences between versions how these are stored , and retrieved. where possible this is done per port and board, or if not possible the common configuration for has been included.
- the **Firmware stubs** are generated directly on a MicroPython board.

2.2 Firmware Stubs format and limitations

1. No function parameters are generated
2. No return types are generated
3. Instances of imported classes have no type (due to 2)
4. The stubs use the .py extension rather than .pyi (for autocomplete to work)
5. Due to the method of generation nested modules are included, rather than referenced. While this leads to somewhat larger stubs, this should not be limiting for using the stubs on a PC.
6.

2.3 Firmware naming convention

The firmware naming conventions is most relevant to provide clear folder names when selecting which stubs to use.

for stubfiles: {**firmware**}-{port}-{version}[-{build}]

for frozen modules : {firmware}-{version}-frozen

- ***firmware***: lowercase
  - micropython | loboris | pycopy | ...
- ***port***: lowercase , as reported by os.implementation.platform
  - esp32 | linux | win32 | esp32_lobo
- ***version*** : digits only , dots replaced by underscore, follow version in documentation rather than semver
  - 1_13
  - 1_9_4
- ***build***, only for nightly build, the build nr. extracted from the git tag
  - Nothing , for released versions
  - 103
  - N (short notation)

CHAPTER

THREE

USING STUBS

3.1 Manual configuration

The manual configuration, including sample configuration files is described in detail in the sister-repo [micropython-stubs][] section [using-the-stubs][]

3.2 Using micropy-cli

'micropy-cli' is command line tool for managing MicroPython projects with VSCode If you want a command line interface to setup a new project and configure the settings as described above for you, then take a look at : [micropy-cli]

```
pip install micropy-cli
micropy init
```

Braden has essentially created a front-end for using micropython-stubber, and the configuration of a project folder for pymakr.

micropy-cli maintains its own repository of stubs.

CHAPTER

FOUR

VSCODE AND PYLINT CONFIGURATION

The current configuration section describes how to use [Pylance].

> *To deliver an improved user experience, we've created Pylance as a brand-new language server based on Microsoft's Pyright static type checking tool. Pylance leverages type stubs (.pyi files) and lazy type inferencing to provide a highly-performant development experience. Pylance supercharges your Python IntelliSense experience with rich type information, helping you write better code, faster. The Pylance extension is also shipped with a collection of type stubs for popular modules to provide fast and accurate auto-completions and type checking.*

Some sections may still refer to the use of [Microsoft Python Language Server][mpls], which has been deprecated.

4.1 Recommended order of the stubs in your config:

1. The src/libs folder(s)
2. The CPython common modules
3. The frozen modules offer more information that can be used in code completion, and therefore should be loaded before the firmware stubs.
4. The firmware stubs generated on or for your board

Announcing Pylance: Fast, feature-rich language support for Python in Visual Studio Code | Python (microsoft.com)

4.2 Relevant VSCode settings

Setting	Default	Description	ref
python.autoComplete.extraPaths	[]	Specifies locations of additional packages for which to load autocomplete data.	Autocomplete Settings
typeshedPaths	[]	Specifies paths to local typeshed repository clone(s) for the Python language server.	Git
python.linting.			Linting Settings
enabled	true	Specifies whether to enable linting in general.	
pylintEnabled	true	Specifies whether to enable Pylint.	

4.2.1 Pylance - pyright

[Pylance](Pylance - Visual Studio Marketplace) is replacing MPLS and provides the same and more functionality.

Setting	Default	Description
python.analysis.stubPath	Path-ings	Used to allow a user to specify a path to a directory that contains custom type stubs. Each package's type stub file(s) are expected to be in its own subdirectory.
python.analysis.autoSearchPaths	true	Used to automatically add search paths based on some predefined names (like `src`).
python.analysis.extraPaths	[]	Used to specify extra search paths for import resolution. This replaces the old `python.autoComplete.extraPaths` setting.

4.2.2 Sample configuration for Pylance

To update a project configuration from MPLS to Pylance is simple :

Open your VSCode settings file : `.vscode/settings.json`

- change the language server to Pylance `"python.languageServer": "Pylance",`
- remove the section: `python.autoComplete.typeshedPaths`
- remove the section : `python.analysis.typeshedPaths`
- optionally add : `"python.analysis.autoSearchPath": true,`

The result should be something like this :

```
{
     "python.languageServer": "Pylance",
     "python.analysis.autoSearchPath": true,
     "python.autoComplete.extraPaths": [
          "src/lib",
          "all-stubs/cpython_patch",
          "all-stubs/mpy_1_13-nightly_frozen/esp32/GENERIC",
          "all-stubs/esp32_1_13_0-103",
     ]
    "python.linting.enabled": true,
    "python.linting.pylintEnabled": true,
}
```

If you notice problems :

- The paths are case sensitive (which may not be apparent for your platform)
- To allow the config to be used cross platform you can use forward slashes `/`, *note that this is also accepted on Windows*
- If you prefer to use a backslash : in JSON notation the \ (backslash) MUST be escaped as \\ (double backslash)
- Remember to put the 'Frozen' module paths before the generated module paths.

References :

Pylance - Visual Studio Marketplace

microsoft/pyright: Static type checker for Python (github.com)

possible testing / diag :

pyright/command-line.md at master · microsoft/pyright (github.com)

4.3 pylint

Pylint needs 2 settings :

1. Specify **init-hook** to inform pylint where the stubs are stored. note that the `src` folder is already automagically included, so you do not need to add that.
2. disable some pesky warnings that make no sense for MicroPython, and that are caused by the stubs that have only limited information

File: .pylintrc

```
[MASTER]
# Loaded Stubs:  esp32-micropython-1.11.0
init-hook='import sys;sys.path[1:1] = ["src/lib", "all-stubs/cpython-core", "all-stubs/
↪mpy_1_12/frozen/esp32/GENERIC", "all-stubs/esp32_1_13_0-103",]'

disable = missing-docstring, line-too-long, trailing-newlines, broad-except, logging-
↪format-interpolation, invalid-name,
        no-method-argument, assignment-from-no-return, too-many-function-args,␣
↪unexpected-keyword-arg
        # the 2nd  line deals with the limited information in the generated stubs.
```

4.4 Microsoft Python Language Server settings - Deprecated

MPLS is being replaced by Pylance , and the below configuration is for reference only .

The language server settings apply when python.jediEnabled is false.

Setting	Default	Description	ref
python.jediEnabled	Default: *true*, must be set to FALSE	Indicates whether to use Jedi as the IntelliSense engine (true) or the Microsoft Python Language Server (false). Note that the language server requires a platform that supports .NET Core 2.1 or newer.	
python.analysis.			code analysis settings)
typeshedPaths	[]	Paths to look for typeshed modules on GitHub.	

Our long-term plan is to transition our Microsoft Python Language Server users over to Pylance and eventually deprecate and remove the old language server as a supported option

CHAPTER

FIVE

CREATE FIRMWARE STUBS

It is possible to create MicroPython stubs using the `createstubs.py` MicroPython script.

the script goes though the following stages

1. it determines the firmware family, the version and the port of the device, and based on that information it creates a firmware identifier (fwid) in the format : {family}-{port}-{version} the fwid is used to name the folder that stores the subs for that device.
    - micropython-pyboard-1_10
    - micropython-esp32-1_12
    - loboris-esp32_LoBo-3_2_4
2. it cleans the stub folder
3. it generates stubs, using a predetermined list of module names. for each found module or submodule a stub file is written to the device and progress is output to the console/repl.
4. a module manifest (`modules.json`) is created that contains the pertinent information determined from the board, the version of createstubs.py and a list of the successful generated stubs

Module duplication

Due to the module naming convention in micropython some modules will be duplicated , ie `uos` and `os` will both be included

5.1 Running the script

The createstubs.py script can either be run as a script or imported as a module depending on your preferences.

Running as a script is used on the linux or win32 platforms in order to pass a –path parameter to the script.

The steps are :

1. connect to your board
2. upload the script to your board [optional]
3. run/import the `createstubs.py` script
4. download the generated stubs to a folder on your PC
5. run the post-processor [optional, but recommended]

![createstubs-flow][]

Note: There is a memory allocation bug in MicroPython 1.30 that prevents createstubs.py to work. this was fixed in nightly build v1.13-103 and newer.

If you try to create stubs on this defective version, the stubber will raise *NotImplementedError*("MicroPython 1.13.0 cannot be stubbed")

5.2 Generating Stubs for a specific Firmware

The stub files are generated on a MicroPython board by running the script `createstubs.py`, this will generate the stubs on the board and store them, either on flash or on the SD card.

Normal and minified versions

The script is available in 2 versions :

1. The normal version, which includes logging, but also requires to logging module to be available.
2. A minified version, which requires less memory and only very basic logging. this is specially suited for low memory devices such as the esp8622 Both versions have the exact same functionality.

If your firmware does not include the `logging` module, you will need to upload this to your board, or use the minified version.

```
import createstubs
```

The generation will take a few minutes (2-5 minutes) depending on the speed of the board and the number of included modules.

As the stubs are generated on the board, and as MicroPython is highly optimized to deal with the scarce resources, this unfortunately does mean that the stubs lacks parameters details. So for these you must still use the documentation provided for that firmware.

5.3 Downloading the files

After the sub files have been generated , you will need to download the generated stubs from the micropython board and most likely you will want to copy and save them on a folder on your computer. if you work with multiple firmwares, ports or version it is simple to keep the stub files in a common folder as the firmware id is used to generate unique names

- ./stubs
  - /micropython-pyboard-1_10
  - /micropython-esp32-1_12
  - /micropython-linux-1_11
  - /loboris-esp32_LoBo-3_1_20
  - /loboris-esp32_LoBo-3_2_24

5.4 Custom firmware

The script tries to determine a firmware ID and version from the information provided in `sys.implementation` , `sys.uname()` and the existence of specific modules..

This firmware ID is used in the stubs , and in the folder name to store the subs.

If you need, or prefer, to specify a firmware ID you can do so by setting the firmware_id variable before importing createstubs For this you will need to edit the createstubs.py file.

The recommendation is to keep the firmware id short, and add a version as in the below example.

```
# almost at the end of the file
def main():
    stubber = Stubber(firmware_id='HoverBot v1.2.1')
    # Add specific additional modules to be stubbed
    stubber.add_modules(['hover','rudder'])
```

after this , upload the file and import it to generate the stubs using your custom firmware id.

5.5 The Unstubbables

There are a limited number of modules that cannot be stubbed by createstubs.py for a number of different reasons. Some simply raise errors , others my reboot the MCU, or require a specific configuration or state before they are loaded.

a few of the frozen modules are just included as a sample rather \t would not be very useful to generate stubs for these

the problematic category throw errors or lock up the stubbing process altogether:

```
 self.problematic=["upysh","webrepl_setup","http_client","http_client_ssl","http_server",
↪"http_server_ssl"]
```

the excluded category provides no relevant stub information

```
 self.excluded=["webrepl","_webrepl","port_diag","example_sub_led.py","example_pub_
↪button.py"]
```

`createstubs.py` will not process a module in either category.

Note that some of these modules are in fact included in the frozen modules that are gathered for those ports or boards

CHAPTER

SIX

CPYTHON AND FROZEN MODULES

6.1 Frozen Modules

It is common for Firmwares to include a few (or many) python modules as 'frozen' modules. 'Freezing' modules is a way to pre-process .py modules so they're 'baked-in' to MicroPython' s firmware and use less memory. Once the code is frozen it can be quickly loaded and interpreted by MicroPython without as much memory and processing time.

Most OSS firmwares store these frozen modules as part of their repository, which allows us to:

1. Download the *.py from the (github) repo using `git clone` or a direct download
2. Extract and store the 'unfrozen' modules (ie the *.py files) in a _Frozen folder. if there are different port / boards or releases defined , there may be multiple folders such as:
   - stubs/micropython_1_12_frozen
     - /esp32
       - /GENERIC
       - /RELEASE
       - /TINYPICO
     - /stm32
       - /GENERIC
       - /PYBD_SF2
3. generate typeshed stubs of these files. (the .pyi files will be stored alongside the .py files)
4. Include/use them in the configuration

ref: https://learn.adafruit.com/micropython-basics-loading-modules/frozen-modules

6.2 Collect Frozen Stubs (micropython)

This is run daily though the github action workflow : get-all-frozen in the micropython-stubs repo.

If you want to run this manually

- Check out repos side-by-side:
  - micropython-stubs
  - micropython-stubber
  - micropython

  - micropython-lib

- link repos using all_stubs symlink
- checkout tag / version in the micropython folder(for most accurate results should checkout micropython-lib for the same date)
- run `src/get-frozen.py`
- run `src/update-stubs.py`
- create a PR for changes to the stubs repo

6.3 Postprocessing

You can run postprocessing for all stubs by running either of the two scripts. There is an optional parameter to specify the location of the stub folder. The default path is `./all_stubs`

Powershell:

```
./scripts/updates_stubs.ps1 [-path ./mystubs]
```

or python

```
python ./src/update_stubs.py [./mystubs]
```

This will generate or update the `.pyi` stubs for all new (and existing) stubs in the `./all_stubs` or specified folder.

From version '1.3.8' the `.pyi` stubs are generated using `stubgen`, before that the `make_stub_files.py` script was used.

Stubgen is run on each 'collected stub folder' (that contains a `modules.json` manifest) using the options : `--ignore-errors --include-private` and the resulting `.pyi` files are stored in the same folder (`foo.py` and `foo.pyi` are stored next to each other).

In some cases `stubgen` detects duplicate modules in a 'collected stub folder', and subsequently does not generate any stubs for any `.py` module or script. then **Plan B** is to run stubgen for each separate `*.py` file in that folder. THis is significantly slower and according to the stubgen documentation the resulting stubs may of lesser quality, but that is better than no stubs at all.

Note: In several cases `stubgen` creates folders in inappropriate locations (reason undetermined), which would cause issues when re-running `stubgen` at a later time. to compensate for this behaviour the known-incorrect .pyi files are removed before and after stubgen is run see: `cleanup(modules_folder)` in `utils.py`

CHAPTER

SEVEN

REPO STRUCTURE

- *This and sister repos*
- *Structure of this repo*
- *Naming Convention and Stub folder structure*
- 2 python versions

7.1 This and sister repos

repo	Why	Where	example
micropython-stubber	needed to make stubs	in your source folder	develop/micropython-stubber
micropython	to collect frozen modules	submodule of micropython-stubber	develop/micropython-stubber/micropython
micropython-lib	to collect frozen modules	submodule of micropython-stubber	develop/micropython-stubber/micropython-lib
micropython-stubs	stores collected stubs	next to the `stubber`	develop/micropython-stubs

Note:

- recommended is to create a symlink from `develop/micropython-stubber\all-stubs` to `develop/micropython-stubs`

Note:

- For Git submodules please refer to https://git-scm.com/book/en/v2/Git-Tools-Submodules

7.2 Structure of this repo

The file structure is based on my personal windows environment, but you should be able to adapt that without much hardship to you own preference and OS.

What	Details	Where
stub root	symlink to connect the 2 sister-repos	all_stubs
firmware stubber	MicroPython	board/createstubs.py
minified firmware stubber	MicroPython	minified/createstubs.py
PC based scripts	CPython	src/*
PC based scripts	CPython	process.py
pytest tests		test/*

7.3 Naming Convention and Stub folder structure

What	Why	Where
stub root	connect the 2 repos	all_stubs
cpython stubs for micropython core	adapt for differences between CPython and MicroPython	stubs/cpython-core
generated stub files	needed to use stubs	stubs/{firmware}-{port}-{version}-frozen
Frozen stub files	better code intellisense	stubs/{firmware}-{version}-frozen

Note: I found that, for me, using submodules caused more problems than it solved. So instead I link the two main repo's using a *symlink*.

Note: I in the repo tests I have used the folders `TESTREPO-micropython` and `TESTREPO-micropython-lib` to avoid conflicts with any development that you might be doing on similar `micropython` repos at the potential cost of a little disk space.

```
cd /develop

git clone  https://github.com/josverl/micropython-stubber.git
git clone  https://github.com/josverl/micropython-stubs.git
git clone  https://github.com/micropython/micropython.git
git clone  https://github.com/micropython/micropython.git
```

7.4 Create a symbolic link

To create the symbolic link to the `../micropython-stubs/stubs` folder the instructions differ slightly for each OS/ The below examples assume that the micropython-stubs repo is cloned 'next-to' your project folder. please adjust as needed.

7.4.1 Windows 10

Requires `Developer enabled` or elevated powershell prompt.

```
# target must be an absolute path, resolve path is used to resolve the relative path to␣
↪absolute
New-Item -ItemType SymbolicLink -Path "all-stubs" -Target (Resolve-Path -Path ../
↪micropython-stubs/stubs)
```

or use mklink in an (elevated) command prompt

```
rem target must be an absolute path
mklink /d all-stubs c:\develop\micropython-stubs\stubs
```

7.4.2 Linux/Unix/Mac OS

```
# target must be an absolute path
ln -s /path/to/micropython-stubs/stubs all-stubs
```

CHAPTER

EIGHT

STUBS

Initially I also stored all the generated subs in the same repo. That turned out to be a bit of a hassle and since then I have moved all the stubs to the micropython-stubs repo

Below are the most relevant stub sources referenced in this project.

8.1 Firmware and libraries

8.1.1 MicroPython firmware and frozen modules *[MIT]*

https://github.com/micropython/micropython

https://github.com/micropython/micropython-lib

8.1.2 Pycopy firmware and frozen modules *[MIT]*

https://github.com/pfalcon/pycopy

https://github.com/pfalcon/pycopy-lib

8.1.3 LoBoris ESP32 firmware and frozen modules *[MIT, Apache 2]*

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo

8.2 Included custom stubs

Github repo	Contributions	License
pfalcon/micropython-lib	CPython backports	MIT
dastultz/micropython-pyb	a pyb.py file for use with IDEs in developing a project for the Pyboard	Apache 2

8.2.1 Stub source: MicroPython-lib > CPython backports *[MIT, Python]*

While micropython-lib focuses on MicroPython, sometimes it may be beneficial to run MicroPython code using CPython, e.g. to use code coverage, debugging, etc. tools available for it. To facilitate such usage, micropython-lib also provides re-implementations ("backports") of MicroPython modules which run on CPython. https://github.com/pfalcon/micropython-lib#cpython-backports

8.2.2 micropython_pyb *[Apache 2]*

This project provides a pyb.py file for use with IDEs in developing a project for the Pyboard. https://github.com/dastultz/micropython-pyb

---

CHAPTER

NINE

REFERENCES

9.1 Inspiration

9.1.1 Thonny - MicroPython _cmd_dump_api_info *[MIT License]*

The `createstubs.py` script to create the stubs is based on the work of Aivar Annamaa and the Thonny crew. It is somewhere deep in the code and is apparently only used during the development cycle but it showed a way how to extract/generate a representation of the MicroPython modules written in C

While the concepts remain, the code has been rewritten to run on a micropython board, rather than on a connected PC running CPython. Please refer to : Thonny code sample

9.1.2 MyPy Stubgen

MyPy stubgen is used to generate stubs for the frozen modules and for the `*.py` stubs that were generated on a board.

9.1.3 make_stub_files *[Public Domain]*

https://github.com/edreamleo/make-stub-files

This script `make_stub_files.py` makes a stub (.pyi) file in the output directory for each source file listed on the command line (wildcard file names are supported).

The script does no type inference. Instead, the user supplies patterns in a configuration file. The script matches these patterns to: The names of arguments in functions and methods and The text of return expressions. Return expressions are the actual text of whatever follows the "return" keyword. The script removes all comments in return expressions and converts all strings to "str". This preprocessing greatly simplifies pattern matching.

Note: It was found that the stubs / prototypes of some functions with complex arguments were not handled correctly, resulting in incorrectly formatted stubs (.pyi)Therefore this functionality has been replaced by MyPy `stubgen`

9.2 Documentation on Type hints

- Type hints cheat sheet
- PEP 3107 – Function Annotations
- PEP 484 – Type Hints
- Optional Static Typing for Python
- TypeShed
- SO question

CHAPTER

TEN

CHANGELOG

10.1 documentation

- Add Sphinxs documentaion
  - changelog
  - automatic API documentation for
    - createsubs.py (board)
    - scripts to run on PC / Github actions
- Publish documentation to readthedocs

10.2 createstubs - version 1.4

- createstubs.py
  - improvements to handle nested classes to be able to create stubs for lvgl. this should also benefit other more complex modules.
- added `stub_lvgl.py` helper script

10.3 createstubs.py - version 1.3.16

- createstubs.py
  - fix for micropython v1.16
  - skip _test modules in module list
  - black formatting
  - addition of **init** methods (based on runtime / static)
  - class method decorator
  - additional type information for constants using comment style typing
  - detect if running on MicroPython or CPython
  - improve report formatting to list each module on a separate line to allow for better comparison
- workflows

  - move to ubuntu 20.04
    - move to test/tools/ubuntu_20_04/micropython_v1.xx
  - run more tests in GHA
- postprocessing
  - minification adjusted to work with `black`
  - use `mypy.stubgen`
  - run per folder
    - verify 1:1 relation .py-.pyi
    - run `mypy.stubgen` to generate missing .pyi files
  - publish test results to GH
- develop / repo setup
  - updated dev requirements (requirements-dev.txt)
  - enable developing on GitHub codespaces
  - switched to using submodules to remove external dependencies how to clone : `git submodule init` `git submodule update`
  - added black configuration file to avoid running black on minified version
  - switched to using .venv on all platforms
  - added and improved tests
    - test coverage increased to 82%
  - move to test/tools/ubuntu_20_04/micropython_v1.xx
    - for test (git workflows)
    - for tasks
  - make use of CPYTHON stubs to alle makestubs to run well on CPYTHON
    - allows pytest, and debugging of tests
  - add tasks to :
    - run createstubs in linux version

CHAPTER

ELEVEN

TO-DO (PROVISIONAL)

11.1 working on it

11.1.1 read RST files

- add prototypes from RST ref: https://github.com/python/mypy/blob/master/mypy/stubdoc.py

11.1.2 documentation

- how to run post-processing
- how the debug setup works

11.1.3 stubber :

- document - that gc and sys modules are somehow ignored by pylint and will keep throwing errors
- add mpy information to manifest
- use 'nightly' naming convention in createstubs.py
- change firmware naming

11.1.4 frozen stubs

- add simple readme.md ?

11.1.5 Stub augmentation/ merging typeinformation from copied / generated type-rich info

https://libcst.readthedocs.io/en/latest/tutorial.html

- add prototypes from Source ? check if https://github.com/python/mypy/blob/master/mypy/stubgenc.py might be useful
- test to auto-merge common prototypes by stubber ie. add common return types to make_stub_files.cfg
- resolve import time issues

11.1.6 SYS en GC

#pylint: disable=no-member ## workaround for sys and gc

Module ‘sys’ has no ‘print_exception’ member Module ‘gc’ has no ‘mem_free’ member Module ‘gc’ has no ‘threshold’ member Module ‘gc’ has no ‘mem_free’ member Module ‘gc’ has no ‘mem_alloc’ member { “resource”: “/c:/develop/MyPython/ESP32-P1Meter/src/main.py”, “owner”: “python”, “code”: “no-member”, “severity”: 8, “message”: “Module ‘gc’ has no ‘mem_free’ member”, “source”: “pylint”, “startLineNumber”: 33, “startColumn”: 22, “endLineNumber”: 33, “endColumn”: 22 }

11.1.7 Webrepl

Unable to import ‘webrepl’ can include in common modules C:\develop\MyPython\micropython\extmod\webrepl\webrepl.py

CHAPTER

TWELVE

DEVELOPING

12.1 Windows 10

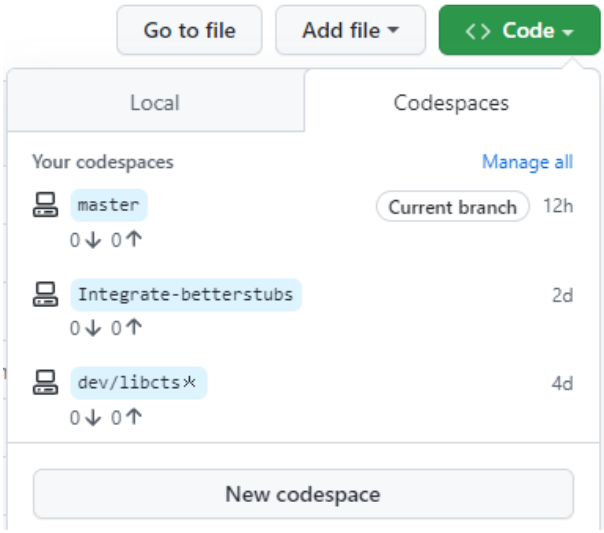
I use Windows 10 and use WSL2 to run the linux based parts. if you develop on other platform, it is quite likely that you may need to change some details. if that is needed , please update/add to the documentation and send a documentation PR.

- clone
- create python virtual environment (optional)
- install requirements-dev
- setup sister repos
- run test to verify setup

12.2 Github codespaces

Is is also possible to start a pre-configure development environment in GitHub Codespaces this is probably the fastest and quickest way to start developing.

Note that Codespaces is currently in an extended beta.



12.3 Wrestling with two pythons

This project combines CPython and MicroPython in one project. As a result you may/will need to switch the configuration of pylint and VSCode to match the section of code that you are working on. This is caused by the fact that pylint does not support per-folder configuration

to help switching there are 2 different .pylintrc files stored in the root of the project to simplify switching.

Similar changes will need to be done to the .vscode/settings.json

If / when we can get pylance to work with the micropython stubs , this may become simpler as Pylance natively supports multi-root workspaces, meaning that you can open multiple folders in the same Visual Studio Code session and have Pylance functionality in each folder.

12.4 Minification

if you make changes to the createstubs.py script , you should also update the minified version by running `python process.py minify` at some point.

if you forget to do this there is a github action that should do this for you and create a PR for your branch.

12.5 Testing

MicroPython-Stubber has a number of tests written in Pytest

see below overview

folder	what	how	used where
board	createstubs.pynormal & minified	runs createstubs.py on micropython-linux ports	WSL2 and github actions
checkout_repo	simple_git moduleretrieval of frozen modules	does not use mocking but actually retrieves different firmware versions locally using git or dowNloads modules for online	local windows
common	all other tests	common	local + github action

Note: Also see *test documentation*

Platform detection to support pytest In order to allow both simple usability om MicroPython and testability on Full Python, createstubs does a runtime test to determine the actual platform it is running on while importing the module This is similar to using the `if __name__ == "__main__":` preamble If running on MicroPython, then it starts stubbing

```
if isMicroPython():
    main()
```

Testing on micropython linux port(s) in order to be able to test `createstubs.py`, it has been updated to run on linux, and accept a –path parameter to indicate the path where the stubs should be stored.

12.6 github actions

12.6.1 pytests.yml

This workflow will :

- test the workstation scripts
- test the createstubs.py script on multiple micropython linux versions
- test the minified createstubs.py script on multiple micropython linux versions

12.6.2 run minify-pr.yml

This workflow will :

- create a minified version of createstubs.py
- run a quick test on that
- and submit a PR to the branch -minify

CHAPTER

THIRTEEN

TESTING

A significant number of tests have been created in pytest.

- The tests are located in the `tests` folder.
- The `tests/data` folder contains folders with subs that are used to verify the correct working of the minification modules
- debugging the tests only works if –no-cov is specified for pytest

13.1 testing & debugging createstubs.py

- the `tests\mocks` folder contains mock-modules that allow the micropython code to be run in CPython. This is used by the unit tests that verify `createstubs.py` and it minified version.
- in order to load / debug the test the python path needs to include the cpython_core modules (Q&D)
- mocking cpython_core/os is missing the implementation attribute so that has been added (Q&D)

13.2 platform detection

In order to allow both simple usability om MicroPython and testability on *full* Python, createstubs does a runtime test to determine the actual platform it is running on while importing the module

This is similar to using the `if __name__ == "__main__":` preamble

```
if isMicroPython():
    main()
```

This allows pytest test running on full Python to import `createstubs.py` and run tests against individual methods, while allowing the script to run directly on import on a MicroPython board.

Note: Some test are platform dependent and have been marked to only run on linux or windows

13.3 Code Coverage

Code coverage is measured and reported in the `coverage/index.html` report. This report is not checked in to the repo, and therefore is only

CHAPTER

FOURTEEN

API REFERENCE

This page contains auto-generated API reference documentation[1].

14.1 stub_lvgl

Helper module to create stubs for the lvgl modules. Note that the stubs can be very large, and it may be best to directly store them on an SD card if your device supports this.

14.1.1 Module Contents

Functions

main()	Create stubs for the lvgl modules using the lvlg version number.

stub_lvgl.**main**()
Create stubs for the lvgl modules using the lvlg version number.

14.2 main

14.2.1 Module Contents

Functions

countdown()	

main.**countdown**()

[1] Created with sphinx-autoapi

14.3 createstubs

Create stubs for (all) modules on a MicroPython board Copyright (c) 2019-2021 Jos Verlinde

14.3.1 Module Contents

Classes

Stubber	Generate stubs for modules in firmware

Functions

resetWDT()	
show_help()	
read_path() → str	get --path from cmdline. [unix/win]
isMicroPython() → bool	runtime test to determine full or micropython
main()	

Attributes

stubber_version	
ENOENT	
MAX_CLASS_LEVEL	

createstubs.**stubber_version = 1.4.1**

createstubs.**ENOENT = 2**

createstubs.**MAX_CLASS_LEVEL = 2**

createstubs.**resetWDT**()

class createstubs.**Stubber**(*path: str = None*, *firmware_id: str = None*)
Generate stubs for modules in firmware

Parameters

- **path** (*str*) –
- **firmware_id** (*str*) –

static _info()
collect base information on this runtime

get_obj_attributes(*self*, *obj: object*)
extract information of the objects members and attributes

Parameters **obj** (*object*) –

add_modules(*self*, *modules: list*)
Add additional modules to be exported

Parameters **modules** (*list*) –

create_all_stubs(*self*)
Create stubs for all configured modules

create_module_stub(*self*, *module_name: str*, *file_name: str = None*)
Create a Stub of a single python module

Parameters

- **module_name** (*str*) –
- **file_name** (*str*) –

write_object_stub(*self*, *fp*, *object_expr: object*, *obj_name: str*, *indent: str*, *in_class: int = 0*)
Write a module/object stub to an open file. Can be called recursive.

Parameters

- **object_expr** (*object*) –
- **obj_name** (*str*) –
- **indent** (*str*) –
- **in_class** (*int*) –

property flat_fwid(*self*)
Turn _fwid from 'v1.2.3' into '1_2_3' to be used in filename

clean(*self*, *path: str = None*)
Remove all files from the stub folder

Parameters **path** (*str*) –

report(*self*, *filename: str = 'modules.json'*)
create json with list of exported modules

Parameters **filename** (*str*) –

ensure_folder(*self*, *path: str*)
Create nested folders if needed

Parameters **path** (*str*) –

static get_root() → str
Determine the root folder of the device

Return type str

createstubs.**show_help**()

createstubs.**read_path**() → str
get –path from cmdline. [unix/win]

Return type str

createstubs.**isMicroPython**() → bool
runtime test to determine full or micropython

Return type bool

createstubs.**main**()

14.4 get_mpy

Collect modules and python stubs from MicroPython source projects (v1.12 +) and stores them in the all_stubs folder
The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

14.4.1 Module Contents

Functions

freeze_as_mpy(path, script=None, opt=0)	
freeze_as_str(path)	
freeze(path, script=None, opt=0)	Freeze the input, automatically determining its type. A .py script
freezedry(path, script)	copy the to-be-frozen module to the destination folder to be stubbed
include(manifest)	Include another manifest.
convert_path(path)	Perform variable substitution in path
get_frozen(stub_path: str, version: str, mpy_path: str = None, lib_path: str = None)	get and parse the to-be-frozen .py modules for micropython to extract the static type information
get_frozen_folders(stub_path: str, mpy_path: str, lib_path: str, version: str)	get and parse the to-be-frozen .py modules for micropython to extract the static type information
get_target_names(path: str) → tuple	get path to port and board names from a path
get_frozen_manifest(manifests, stub_path: str, mpy_path: str, lib_path: str, version: str)	get and parse the to-be-frozen .py modules for micropython to extract the static type information

Attributes

log
FAMILY
path_vars
stub_dir
mpy_path

get_mpy.**log**

get_mpy.**FAMILY = micropython**

get_mpy.**path_vars**

get_mpy.**stub_dir**

exception get_mpy.**FreezeError**
Bases: Exception

Common base class for all non-exit exceptions.

get_mpy.**freeze_as_mpy**(*path*, *script=None*, *opt=0*)

get_mpy.**freeze_as_str**(*path*)

get_mpy.**freeze**(*path*, *script=None*, *opt=0*)
Freeze the input, automatically determining its type. A .py script will be compiled to a .mpy first then frozen, and a .mpy file will be frozen directly.

path must be a directory, which is the base directory to search for files from. When importing the resulting frozen modules, the name of the module will start after *path*, ie *path* is excluded from the module name.

If *path* is relative, it is resolved to the current manifest.py. Use $(MPY_DIR), $(MPY_LIB_DIR), $(PORT_DIR), $(BOARD_DIR) if you need to access specific paths.

If *script* is None all files in *path* will be frozen.

If *script* is an iterable then freeze() is called on all items of the iterable (with the same *path* and *opt* passed through).

If *script* is a string then it specifies the filename to freeze, and can include extra directories before the file. The file will be searched for in *path*.

opt is the optimisation level to pass to mpy-cross when compiling .py to .mpy. (ignored in this implementation)

get_mpy.**freezedry**(*path*, *script*)
copy the to-be-frozen module to the destination folder to be stubbed

get_mpy.**include**(*manifest*)
Include another manifest.

The manifest argument can be a string (filename) or an iterable of strings.

Relative paths are resolved with respect to the current manifest file.

get_mpy.**convert_path**(*path*)
Perform variable substitution in path

get_mpy.**get_frozen**(*stub_path: str*, *version: str*, *mpy_path: str = None*, *lib_path: str = None*)

get and parse the to-be-frozen .py modules for micropython to extract the static type information

- requires that the MicroPython and Micropython-lib repos are checked out and available on a local path
- repos should be cloned side-by-side as some of the manifests refer to micropython-lib scripts using a relative path

Parameters

- **stub_path** (*str*) –
- **version** (*str*) –
- **mpy_path** (*str*) –
- **lib_path** (*str*) –

get_mpy.**get_frozen_folders**(*stub_path: str*, *mpy_path: str*, *lib_path: str*, *version: str*)
get and parse the to-be-frozen .py modules for micropython to extract the static type information locates the to-be-frozen files in modules folders - 'ports/<port>/modules/*.py' - 'ports/<port>/boards/<board>/modules/*.py'

Parameters

- **stub_path** (*str*) –

- **mpy_path** (*str*) –
- **lib_path** (*str*) –
- **version** (*str*) –

get_mpy.**get_target_names**(*path: str*) → tuple
get path to port and board names from a path

Parameters **path** (*str*) –

Return type tuple

get_mpy.**get_frozen_manifest**(*manifests*, *stub_path: str*, *mpy_path: str*, *lib_path: str*, *version: str*)
get and parse the to-be-frozen .py modules for micropython to extract the static type information locates the to-be-frozen files through the manifest.py introduced in MicroPython 1.12 - manifest.py is used for board specific and daily builds - manifest_release.py is used for the release builds

Parameters

- **stub_path** (*str*) –
- **mpy_path** (*str*) –
- **lib_path** (*str*) –
- **version** (*str*) –

get_mpy.**mpy_path = ./micropython**

14.5 get_lobo

Collect modules and python stubs from the Loboris MicroPython source project and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

14.5.1 Module Contents

Functions

get_frozen(stub_path=None, *, repo=None, version='3.2.24')	Loboris frozen modules

Attributes

FAMILY
PORT
log

get_lobo.**FAMILY = loboris**

get_lobo.**PORT = esp32_lobo**

get_lobo.**log**

get_lobo.**get_frozen**(*stub_path=None*, *, *repo=None*, *version='3.2.24'*)
Loboris frozen modules

14.6 update_stubs

Collect modules and python stubs from other projects and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

14.6.1 Module Contents

update_stubs.**log**

update_stubs.**stub_path**

14.7 utils

14.7.1 Module Contents

Functions

clean_version(version: str, build: bool = False)	omit the commit hash from the git tag
stubfolder(path: str) → str	return path in the stub folder
flat_version(version: str)	Turn version from 'v1.2.3' into '1_2_3' to be used in filename
cleanup(modules_folder: pathlib.Path)	Q&D cleanup
generate_pyi_from_file(file: pathlib.Path) → bool	Generate a .pyi stubfile from a single .py module using mypy/stubgen
generate_pyi_files(modules_folder: pathlib.Path) → bool	generate typeshed files for all scripts in a folder using mypy/stubgen
manifest(family=None, machine=None, port=None, platform=None, sysname=None, nodename=None, version=None, release=None, firmware=None) → dict	create a new empty manifest dict
make_manifest(folder: pathlib.Path, family: str, port: str, version: str) → bool	Create a *module.json* manifest listing all files/stubs in this folder and subfolders.
generate_all_stubs()	just create typeshed stubs
read_exclusion_file(path: pathlib.Path = None) → List[str]	Read a .exclusion file to determine which files should not be automatically re-generated
should_ignore(file: str, exclusions: List[str]) → bool	Check if a file matches a line in the exclusion list.

Attributes

log

STUB_FOLDER

utils.**log**

utils.**STUB_FOLDER = ./all-stubs**

utils.**clean_version**(*version: str*, *build: bool = False*)
omit the commit hash from the git tag

Parameters

- **version** (*str*) –
- **build** (*bool*) –

utils.**stubfolder**(*path: str*) → str
return path in the stub folder

Parameters **path** (*str*) –

Return type str

utils.**flat_version**(*version: str*)
Turn version from 'v1.2.3' into '1_2_3' to be used in filename

Parameters **version** (*str*) –

utils.**cleanup**(*modules_folder: pathlib.Path*)
Q&D cleanup

Parameters **modules_folder** (*pathlib.Path*) –

utils.**generate_pyi_from_file**(*file: pathlib.Path*) → bool
Generate a .pyi stubfile from a single .py module using mypy/stubgen

Parameters **file** (*pathlib.Path*) –

Return type bool

utils.**generate_pyi_files**(*modules_folder: pathlib.Path*) → bool
generate typeshed files for all scripts in a folder using mypy/stubgen

Parameters **modules_folder** (*pathlib.Path*) –

Return type bool

utils.**manifest**(*family=None*, *machine=None*, *port=None*, *platform=None*, *sysname=None*, *nodename=None*, *version=None*, *release=None*, *firmware=None*) → dict
create a new empty manifest dict

Return type dict

utils.**make_manifest**(*folder: pathlib.Path*, *family: str*, *port: str*, *version: str*) → bool
Create a *module.json* manifest listing all files/stubs in this folder and subfolders.

Parameters

- **folder** (*pathlib.Path*) –
- **family** (*str*) –

- **port** (*str*) –
- **version** (*str*) –

Return type bool

utils.**generate_all_stubs**()
just create typeshed stubs

utils.**read_exclusion_file**(*path: pathlib.Path = None*) → List[str]
Read a .exclusion file to determine which files should not be automatically re-generated in .GitIgnore format

Parameters **path** (*pathlib.Path*) –

Return type List[str]

utils.**should_ignore**(*file: str*, *exclusions: List[str]*) → bool
Check if a file matches a line in the exclusion list.

Parameters

- **file** (*str*) –
- **exclusions** (*List[str]*) –

Return type bool

14.8 get_cpython

Download or update the micropyton compatibility modules from pycopy and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

14.8.1 Module Contents

Functions

get_core(requirements, stub_path=None)	Download MicroPython compatibility modules

Attributes

log
family

get_cpython.**log**

get_cpython.**family = common**

get_cpython.**get_core**(*requirements*, *stub_path=None*)
Download MicroPython compatibility modules

14.9 get_all_frozen

Collect modules and python stubs from other projects and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

1) get cpython core modules
2) get micropython frozen modules for the CURRENT checked out version
3) get Loboris frozen modules (no longer maintained)
4) Generate/update type hint files (pyi) for all stubs.

14.9.1 Module Contents

Functions

get_all(mpy_path=MPY_PATH)	get all frozen modules for the current version of micropython

Attributes

log
STUB_FOLDER
MPY_PATH

get_all_frozen.**log**

get_all_frozen.**STUB_FOLDER = ./all-stubs**

get_all_frozen.**MPY_PATH = ./micropython**

get_all_frozen.**get_all**(*mpy_path=MPY_PATH*)
get all frozen modules for the current version of micropython

14.10 basicgit

simple Git module, where needed via powershell

14.10.1 Module Contents

Functions

_run_git(cmd: str, repo: str = None, expect_stderr=False)	run a external (git) command in the repo's folder and deal with some of the errors
get_tag(repo: str = None) → Union[str, None]	get the most recent git version tag of a local repo"
checkout_tag(tag: str, repo: str = None) → bool	get the most recent git version tag of a local repo"
fetch(repo: str) → bool	fetches a repo
pull(repo: str, branch='master') → bool	pull a repo origin into master

basicgit.**_run_git**(*cmd: str*, *repo: str = None*, *expect_stderr=False*)
run a external (git) command in the repo's folder and deal with some of the errors

Parameters

- **cmd** (*str*) –
- **repo** (*str*) –

basicgit.**get_tag**(*repo: str = None*) → Union[str, None]
get the most recent git version tag of a local repo" repo should be in the form of : repo = "./micropython"

returns the tag or None

Parameters **repo** (*str*) –

Return type Union[str, None]

basicgit.**checkout_tag**(*tag: str*, *repo: str = None*) → bool
get the most recent git version tag of a local repo" repo should be in the form of : repo = "../micropython/.git"

returns the tag or None

Parameters

- **tag** (*str*) –
- **repo** (*str*) –

Return type bool

basicgit.**fetch**(*repo: str*) → bool
fetches a repo repo should be in the form of : repo = "../micropython/.git"

returns True on success

Parameters **repo** (*str*) –

Return type bool

basicgit.**pull**(*repo: str*, *branch='master'*) → bool
pull a repo origin into master repo should be in the form of : repo = "../micropython/.git"

returns True on success

Parameters **repo** (*str*) –

Return type bool

14.11 downloader

Download files from a public github repo

14.11.1 Module Contents

Functions

download_file(url: str, module: str, folder: str = './')	dowload a file from a public github repo
download_files(repo, frozen_modules, savepath)	dowload multiple files from a public github repo

downloader.**download_file**(*url: str*, *module: str*, *folder: str = './'*)
dowload a file from a public github repo

Parameters

- **url** (*str*) –
- **module** (*str*) –
- **folder** (*str*) –

downloader.**download_files**(*repo*, *frozen_modules*, *savepath*)
dowload multiple files from a public github repo

14.12 add_class_init

14.12.1 Module Contents

Functions

add_init_methods(filename) → int	Add (missing) __init__ methods to a class using a regex

Attributes

empty_classdef
re_classdef
repl_classdef
x

add_class_init.**empty_classdef = (?P<indent1>
?)class\s*(?P<class>\s*.+\s*):(?P<LF>\r?\n)(?P<indent2> +)''\r?\n**

add_class_init.**re_classdef**

add_class_init.**repl_classdef = \g<indent1>class \g<class>:\g<LF>\g<indent2>def
__init__(self):\g<LF>\g<indent2> ...**

add_class_init.**add_init_methods**(*filename*) → int

Add (missing) __init__ methods to a class using a regex this assumes the (incorrect) classdef format that has been used by stubbers prior to version 1.4.0 and updates that to add the init.

Return type int

add_class_init.**x**

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

P

R

S

U

W

X