MicroPython Development Documentation Documentation

Release 1.0

Radomir Dopieralski

Nov 12, 2017

Contents

Contents:

CHAPTER 1

Introduction

This is an unofficial MicroPython development guide. It was written to make it easier for new developers to start contributing code to the MicroPython project. It's not comprehensive, and at times might even be wrong, but it is some starting point.

CHAPTER 2

Directory Structure

The MicroPython repository is divided into a number of directories, each with its own function. Some of them include:

2.1 Docs, Logo and Examples

The user documentation for all ports lives in the `docs` directory, with some example code in the `examples` directory. The `logo` directory contains the MicroPython logo in various formats.

2.2 Tests

All tests live in the `tests` directory.

2.3 Python Code

The `py` directory contains the bulk of code for the compiler, runtime environment and core library of MicroPython. This is where we will be looking for API functions and macros.

2.4 Port-specific Code

The directories such as `esp8266`, `cc3200`, `pic16bit`, `teensy`, `stmhal`, `bare-arm`, `qemu-arm`, `unix` and `windows` contain code and tools specific to particular ports of MicroPython. If you are working on a feature to be added for a specific hardware, this is probably the best place to put your files.

2.5 Minimal Port

The `minimal` directory contains the minimum of files that a new port needs. You can use this as a template for starting new ports of MicroPython.

2.6 Common Parts

The `extmod` directory contains source of MicroPython modules that may are not part of the core library, but are useful for multiple ports. The `drivers` directory has code for libraries that communicate with various additional hardware, such as displays or sensors.

2.7 Tools and Utilities

The `tools` directory contains various tools useful for working with MicroPython. The `mpy-cross` directory has code for the MicroPython cross-compiler, which can be used to generate frozen bytecode modules.

CHAPTER 3

Adding a Module

When using MicroPython, you have even bigger chance to need your own library written in C than in other Python implementations. You might want to use one of the existing libraries for your platform, get access to some peripherals or other features of the hardware that is not exposed to Python by default, or even just do something faster and with smaller memory overhead. To do that, you will need to extend the existing firmware with your own C code, and this article is going to show you how to do it, using the ESP8266 port as the example.

Note: you can use any naming-scheme you like and do not have to name your variables or data-type in any specific way. If you plan on extending a Micropython port and create pull requests, you should however use the same naming scheme of micropython. See the ESP8266 port for examples.

3.1 Adding Your Own Source File

In order to add your own MicroPython module written in C, you need to create a new C file and add references to it to several files, so that the file is picked up during the compilation.

First of all, you need to add it to the `Makefile`, to the list of source files in the `SRC_C` variable. Make sure to follow the same format as the other files in there. The order doesn't matter much.

```
SRC_C = \
        main.c \
        system_stm32.c \
        stm32_it.c \
        ...
        mymodule.c
        ...
```

The second file you will need to add to is `esp8266.ld`, which is the map of memory used by the compiler. You have to add it to the list of files to be put in the `.irom0.text` section, so that your code goes into the instruction read-only memory (iROM). If you fail to do that, the compiler will try to put it in the instruction random-access memory (iRAM), which is a very scarce resource, and which can get overflown if you try to put too much there.

Now just create an empty `mymodule.c` file, and run the compilation to see that it is now included in the firmware.

3.2 Creating a Python Module

We have our file, but it doesn't actually do anything. It's empty, and there is no new python module that we could import. Time to change that.

From the C side, modules in MicroPython are simply structs with a certain structure. Open the `mymodule.c` file and put this code inside:

```
#include "py/nlr.h"
#include "py/obj.h"
#include "py/runtime.h"
#include "py/binary.h"
#include "portmodules.h"

STATIC const mp_map_elem_t mymodule_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_mymodule) },
};

STATIC MP_DEFINE_CONST_DICT (
    mp_module_mymodule_globals,
    mymodule_globals_table
);

const mp_obj_module_t mp_module_mymodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&mp_module_mymodule_globals,
};
```

What does this code do? It just defines a python module, using `mp_obj_module_t` type, and then initializes some of its fields, such as the base type, the name, and the dictionary of globals for that module. In that dictionary, it defines one variable, `__name__`, with the name of our module in it. That's it.

Now, for this module to actually be available for import, we need to add it to `mpconfigport.h` file to `MICROPY_PORT_BUILTIN_MODULES`:

```
extern const struct _mp_obj_module_t mp_module_mymodule;

#define MICROPY_PORT_BUILTIN_MODULES \
    { MP_OBJ_NEW_QSTR(MP_QSTR_umachine), (mp_obj_t)&machine_module }, \
    ...
    { MP_OBJ_NEW_QSTR(MP_QSTR_mymodule), (mp_obj_t)&mp_module_mymodule }, \
```

Now you can try compiling the firmware and flashing it to your board. Then you can run `import mymodule` and see it imported.

3.3 Adding a Function

Now let's add a simple function to that module. Edit `mymodule.c` again and add this code right after the includes:

```
#include <stdio.h>

STATIC mp_obj_t mymodule_hello(void) {
    printf("Hello world!\n");
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_0(mymodule_hello_obj, mymodule_hello);
```

This creates a function object `mymodule_hello_obj` which takes no arguments, and when called, executes the C function `mymodule_hello`. Also note, that our function has to return something (as every Python function returns an mp_obj_t-struct) – so we return `None`. Now we need to actually add that function object to our module:

```
STATIC const mp_map_elem_t mymodule_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_mymodule) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_hello), (mp_obj_t)&mymodule_hello_obj },
};
```

Micropython uses the QSTR-macros to define constant strings. This is used to identify strings and store only unique ones for preserving memory (as it is very limited on the PyBoard-Hardware). Your port has a file `qstrdefsport.h`. In our case add `Q(hello)` to the list (on a new line). This will define the string `hello` for Micropython. Failing to do so will result in a missing file on compilation.

Now when you compile and flash the firmware, you will be able to import the module and call the function inside it.

3.4 Function Arguments

The `MP_DEFINE_CONST_FUN_OBJ_0` macro that we used to define our function is a shortcut for defining a function with no arguments. We can also define a function that takes a single argument with `MP_DEFINE_CONST_FUN_OBJ_1` – the C function then needs to take an argument of type `mp_obj_t`:

```
STATIC mp_obj_t mymodule_hello(mp_obj_t what) {
    printf("Hello %s!\n", mp_obj_str_get_str(what));
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_1(mymodule_hello_obj, mymodule_hello);
```

This function will use the C-function `printf` to output a string. The parameter `what` will be turned into a string by the `mp_obj_str_get_str`-function (i.e. by Micropython).

Note that the `mp_obj_str_get_str` function will automatically raise the right exception on the python side if the argument we gave it is not a python string. This is very convenient.

It's also possible to define functions with variable number of arguments, or even with keyword arguments – you can easily find examples of that in the modules already included in MicroPython. I will not be covering this in detail.

3.5 Classes

A class is a C-struct with certain fields, quite similar to a module:

```
// creating the table of global members
STATIC const mp_rom_map_elem_t mymodule_hello_locals_dict_table[] = { };
STATIC MP_DEFINE_CONST_DICT(mymodule_hello_locals_dict,
                            mymodule_hello_locals_dict_table);

// create the class-object itself
const mp_obj_type_t mymodule_helloObj_type = {
    // "inherit" the type "type"
    { &mp_type_type },
     // give it a name
    .name = MP_QSTR_helloObj,
     // give it a print-function
    .print = mymodule_hello_print,
```

```
    // give it a constructor
    .make_new = mymodule_hello_make_new,
    // and the global members
    .locals_dict = (mp_obj_dict_t*)&mymodule_hello_locals_dict,
};
```

It needs two functions: one for creating the class and allocating all the memory it needs, and one for printing the objects of that class (similar to python's `__repr__`). Let's add them near the top of our file:

```
// this is the actual C-structure for our new object
typedef struct _mymodule_hello_obj_t {
    // base represents some basic information, like type
    mp_obj_base_t base;
    // a member created by us
    uint8_t hello_number;
} mymodule_hello_obj_t;
```

We define a C-struct, which holds the class data and one additional field `hello_number`. Next we need a function to print the object and a constructor:

```
mp_obj_t mymodule_hello_make_new( const mp_obj_type_t *type,
                                  size_t n_args,
                                  size_t n_kw,
                                  const mp_obj_t *args ) {
    // this checks the number of arguments (min 1, max 1);
    // on error -> raise python exception
    mp_arg_check_num(n_args, n_kw, 1, 1, true);
    // create a new object of our C-struct type
    mymodule_hello_obj_t *self = m_new_obj(mymodule_hello_obj_t);
    // give it a type
    self->base.type = &mymodule_hello_type;
    // set the member number with the first argument of the constructor
    self->hello_number = mp_obj_get_int(args[0])
    return MP_OBJ_FROM_PTR(self);
}


STATIC void mymodule_hello_print( const mp_print_t *print,
                                  mp_obj_t self_in,
                                  mp_print_kind_t kind ) {
    // get a ptr to the C-struct of the object
    mymodule_hello_obj_t *self = MP_OBJ_TO_PTR(self_in);
    // print the number
    printf ("Hello(%u)", self->hello_number);
}
```

Now we need to add our object to the module, by adding it into the global member dictionary of our module:

```
STATIC const mp_map_elem_t mymodule_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_mymodule) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_hello), (mp_obj_t)&mymodule_hello_obj },
    { MP_OBJ_NEW_QSTR(MP_QSTR_helloObj), (mp_obj_t)&mymodule_helloObj_obj },
};
```

Note that both the function `mymodule_hello_obj` added earlier to our module and the class `mymodule_hello_obj` are passed as `mp_obj_t` to the globals table of the module.

3.6 Adding Methods

Methods in MicroPython are just functions in the class's locals dict. You add them the same way as you do to modules, just remember that the first argument is a pointer to the data struct:

```
STATIC mp_obj_t mymodule_hello_increment(mp_obj_t self_in) {
    mymodule_hello_obj_t *self = MP_OBJ_TO_PTR(self_in);
    self->hello_number += 1;
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_1(mymodule_hello_increment_obj,
                          mymodule_hello_increment);
```

Also, don't forget to add them to the locals dict:

```
STATIC const mp_rom_map_elem_t mymodule_hello_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_inc), MP_ROM_PTR(&mymodule_hello_increment_obj) },
}
```

3.7 Using our module in Micropython

Now we can use the module in Micropython after rebuilding our port. For example you can write a Python snippet like this:

```
import mymodule;

mymodule.hello ();
a = mymodule.hellObj ( 12 );
print (mymodule);
mymodule.inc();
print (mymodule);
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search