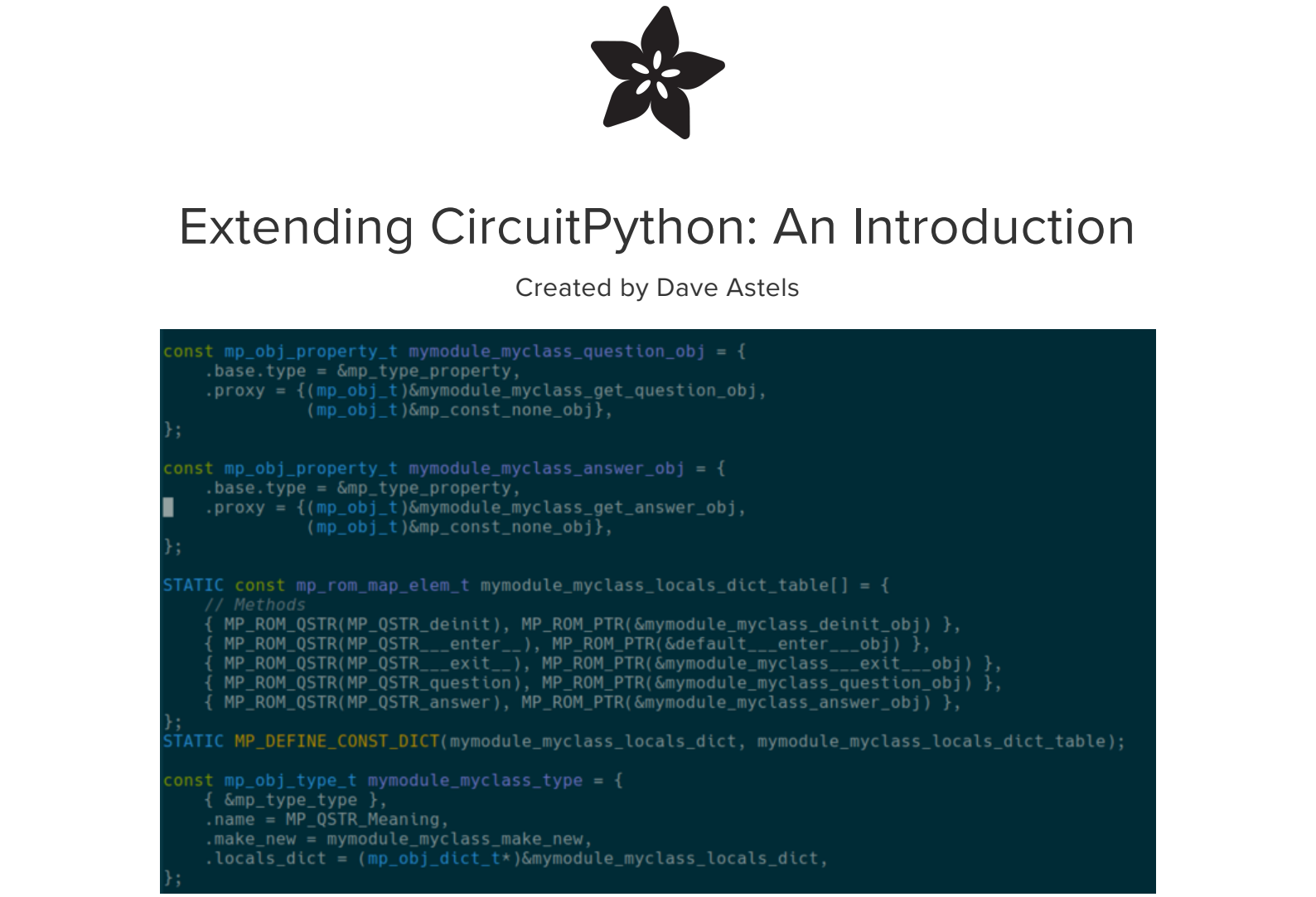


# Extending CircuitPython: An Introduction

Created by Dave Astels

```
const mp_obj_property_t mymodule_myclass_question_obj = {
    .base.type = &mp_type_property,
    .proxy = {(mp_obj_t)&mymodule_myclass_get_question_obj,
              (mp_obj_t)&mp_const_none_obj},
};

const mp_obj_property_t mymodule_myclass_answer_obj = {
    .base.type = &mp_type_property,
    .proxy = {(mp_obj_t)&mymodule_myclass_get_answer_obj,
              (mp_obj_t)&mp_const_none_obj},
};

STATIC const mp_rom_map_elem_t mymodule_myclass_locals_dict_table[] = {
    // Methods
    { MP_ROM_QSTR(MP_QSTR_deinit), MP_ROM_PTR(&mymodule_myclass_deinit_obj) },
    { MP_ROM_QSTR(MP_QSTR___enter__), MP_ROM_PTR(&default___enter___obj) },
    { MP_ROM_QSTR(MP_QSTR___exit__), MP_ROM_PTR(&mymodule_myclass___exit___obj) },
    { MP_ROM_QSTR(MP_QSTR_question), MP_ROM_PTR(&mymodule_myclass_question_obj) },
    { MP_ROM_QSTR(MP_QSTR_answer), MP_ROM_PTR(&mymodule_myclass_answer_obj) },
};
STATIC MP_DEFINE_CONST_DICT(mymodule_myclass_locals_dict, mymodule_myclass_locals_dict_table);

const mp_obj_type_t mymodule_myclass_type = {
    { &mp_type_type },
    .name = MP_QSTR_Meaning,
    .make_new = mymodule_myclass_make_new,
    .locals_dict = (mp_obj_dict_t*)&mymodule_myclass_locals_dict,
};
```

https://learn.adafruit.com/extending-circuitpython

Last updated on 2024-06-03 02:32:17 PM EDT

# Table of Contents

# Overview

As of CircuitPython 5.0.0, many details have changed, and this guide needs considersable updating.

CircuitPython (like MicroPython from which it derives) is implemented in C. While fundamentally different and far more complex than a typical Arduino sketch, it's still the same language; if you are comfortable work in C/C++ in the context of Arduino, CircuitPython internals should look somewhat familiar.  While it's true that we can do pretty much anything in Python, there are a variety of reasons we might want to implement a module in C as part of the virtual machine.

1. We're running out of RAM for our Python code
2. We have some code that needs to run faster
3. We have some C code that we want to use.
4. We want finer, more direct control over some aspects of the hardware.
5. We have some functionality that is so fundamental, that we always want it available. `digitalio` is a prime example of this.

## How-To

To make this happen, not only do we need to implement a Python module in C, we have to connect it to the runtime as well as add it to the build system.

In this guide we will explore how to do this. We'll start with a simple example to show what's needed, and follow that up with a more practical example of an input pin debouncer. Both of these examples are generic CircuitPython extensions, requiring no port specific code. We'll explore that in a future guide.

There isn't much documentation to be found on how to extend the runtime. The best approach after working thought this guide is to look around at the other modules defined this way and see how things are done there.

You will need to be set up and comfortable building CircuitPython for whatever board(s) you have. There's a guide to get up to speed on that (https://adafru.it/Bfu).

The code in this guide has been updated to work with latest CircuitPython 4.0 from github as of mid January 2018 and will not work on prior versions.

---

# A Simple Example

There are three places in the VM codebase that we'll be working: the implementation, the connection into the virtual machine, and integration into the build.

We'll start by exploring this with a very simple example.

## `shared-module`

This is where the implementation goes. We need to add a directory here named for our module: `mymodule`. In this directory we need to add an `__init__.c` file that contains module level functions. We also need to add a header and source file for each class in the module. In this case that means `MyClass.h` and `MyClass.c`.

### `__init__.c`

As mentioned, the definition of any module level functions go here. This example has none, so it's empty, with a "this space left intentionally blank" comment.

```
// No module functions
```

### `MyClass.h`

In this file we define the structure that holds the instance variables of our class. In this simple example, all we need to do is keep track of whether the instance has been disposed of. Normally there will be other ways of telling this state, and we won't need something specific just for it.

The other variable is required: `base` provides storage for the basic information for a Python class.

```
#ifndef MICROPY_INCLUDED_MYMODULE_MYCLASS_H
#define MICROPY_INCLUDED_MYMODULE_MYCLASS_H

#include "py/obj.h"

typedef struct {
  mp_obj_base_t base;
  bool deinited;
} mymodule_myclass_obj_t;


#endif // MICROPY_INCLUDED_MYMODULE_MYCLASS_H
```

### `MyClass.c`

This file contains the methods of `MyClass`.

It begins by including the corresponding header file as well as some basic runtime support. Then there's the standard constructor that is used to initialize new instances. This example is simple and the constructor takes no arguments. This will typically NOT be the case. The next example shows a constructor with arguments.

```
#include "py/runtime.h"
#include "MyClass.h"

void shared_module_mymodule_myclass_construct(mymodule_myclass_obj_t* self) {
  self-&gt;deinited = 0;
}
```

The `deinit` related methods handle disposal of instances as well as determining whether an instance has been disposed of. In classes that aren't this trivial, one will usually have a way to identify a valid object inherent to the object itself.

```
bool shared_module_mymodule_myclass_deinited(mymodule_myclass_obj_t* self) {
  return self-&gt;deinited;
}

void shared_module_mymodule_myclass_deinit(mymodule_myclass_obj_t* self) {
  self-&gt;deinited = 1;
}
```

The remaining functions implement the class methods and properties.

This example is simple: just two read-only properties that return a fixed value, and nothing has parameters.

The `question` property is defined in `shared_module_mymodule_myclass_get_question`. This naming convention is the convention used. Just Do It. Sticking with the established conventions is the safest way to go. You never know when it's depended on. The CircuitPython runtime is complex enough that you don't want to take chances.

Note that these function return native C types. Conversion to CircuitPython runtime types are done in the interface functions below.

```
const char * shared_module_mymodule_myclass_get_question(mymodule_myclass_obj_t*
self) {
  return "Tricky...";
}

mp_int_t shared_module_mymodule_myclass_get_answer(mymodule_myclass_obj_t* self) {
  return 42;
}
```

## `shared-bindings`

In this directory we place the interface for our module. It's the plumbing that connects the implementation to the CircuitPython runtime. Start by adding a directory named after the module as we did for the implementation, e.g. `mymodule`. We need a couple general files, as well a pair for each class. So at a minimum we need the following.

### `__init__.h`

Even if you don't have anything for this file, it has to be present as shown below.

```
#ifndef MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE___INIT___H
#define MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE___INIT___H

#include "py/obj.h"

// Nothing now.

#endif  // MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE___INIT___H
```

### `__init__.c`

This file takes care of hooking up the globals provided by the module. In this case it's just the name of the module and the class.

```
#include &lt;stdint.h&gt;

#include "py/obj.h"
#include "py/runtime.h"

#include "shared-bindings/mymodule/__init__.h"
#include "shared-bindings/mymodule/MyClass.h"

STATIC const mp_rom_map_elem_t mymodule_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_mymodule) },
    { MP_ROM_QSTR(MP_QSTR_MyClass), MP_ROM_PTR(&amp;mymodule_myclass_type) },
};

STATIC MP_DEFINE_CONST_DICT(mymodule_module_globals, mymodule_module_globals_table);

const mp_obj_module_t mymodule_module = {
    .base = { &amp;mp_type_module },
    .globals = (mp_obj_dict_t*)&amp;mymodule_module_globals,
};
```

### `MyClass.h`

The class header here declares the functions from the implementation. They are declared `extern` which tells the compiler that they are defined elsewhere and will be available when all the files are linked together.

```
#ifndef MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE_MYCLASS_H
#define MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE_MYCLASS_H

#include "shared-module/mymodule/MyClass.h"

extern const mp_obj_type_t mymodule_myclass_type;

extern void shared_module_mymodule_myclass_construct(mymodule_myclass_obj_t* self);
extern void shared_module_mymodule_myclass_deinit(mymodule_myclass_obj_t* self);
extern bool shared_module_mymodule_myclass_deinited(mymodule_myclass_obj_t* self);
extern char * shared_module_mymodule_myclass_get_question(mymodule_myclass_obj_t*
self);
extern mp_int_t shared_module_mymodule_myclass_get_answer(mymodule_myclass_obj_t*
self);

#endif // MICROPY_INCLUDED_SHARED_BINDINGS_MYMODULE_MYCLASS_H
```

### `MyClass.c`

As usual, we start with some includes: one for your class, and a handful of runtime support headers.

```
#include &lt;stdint.h&gt;
#include &lt;string.h&gt;
#include "lib/utils/context_manager_helpers.h"
#include "py/objproperty.h"
#include "py/runtime.h"
#include "py/runtime0.h"
#include "shared-bindings/mymodule/MyClass.h"
#include "shared-bindings/util.h"
```

Now we come to the life-cycle functions, including the constructor support.

```
//| .. currentmodule:: mymodule
//|
//| :class:`MyClass` -- The great question (and answer to it) of life, the
universe, and everything.
//|
=============================================================================
//|
//| Provides the great question (and answer to it) of life, the universie, and
everything.
//|
//| .. class:: MyClass()
//|
//|   Create an object.

STATIC mp_obj_t mymodule_myclass_make_new(const mp_obj_type_t *type, size_t n_args,
size_t n_kw, const mp_obj_t *pos_args) {
  mp_arg_check_num(n_args, n_kw, 0, 0, true);
    mymodule_myclass_obj_t *self = m_new_obj(mymodule_myclass_obj_t);
    self-&gt;base.type = &amp;mymodule_myclass_type;
    shared_module_mymodule_myclass_construct(self);
    return MP_OBJ_FROM_PTR(self);
}

//|   .. method:: deinit()
//|
//|      Deinitializes the Meaning and releases any hardware resources for reuse.
//|
STATIC mp_obj_t mymodule_myclass_deinit(mp_obj_t self_in) {
  shared_module_mymodule_myclass_deinit(self_in);
  return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_1(mymodule_myclass_deinit_obj,
mymodule_myclass_deinit);

//|   .. method:: __enter__()
//|
//|      No-op used by Context Managers.
//|
//  Provided by context manager helper.

//|   .. method:: __exit__()
//|
//|      Automatically deinitializes the hardware when exiting a context. See
//|      :ref:`lifetime-and-contextmanagers` for more info.
//|
STATIC mp_obj_t mymodule_myclass_obj___exit__(size_t n_args, const mp_obj_t *args) {
  shared_module_mymodule_myclass_deinit(args[0]);
  return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(mymodule_myclass___exit___obj, 4, 4,
mymodule_myclass_obj___exit__);
```

Now the actual properties. These simply call the implementation functions.

```
//|   .. attribute:: question
//|
//|     The question of life, the universe and everything
//|
STATIC mp_obj_t mymodule_myclass_obj_get_question(mp_obj_t self_in) {
  char *str = shared_module_mymodule_myclass_get_question(self_in);
  return mp_obj_new_str(str, strlen(str));
}
MP_DEFINE_CONST_FUN_OBJ_1(mymodule_myclass_get_question_obj,
```

```
mymodule_myclass_obj_get_question);

//|    .. attribute:: answer
//|
//|      The answer to the question of life, the universe and everything
//|
STATIC mp_obj_t mymodule_myclass_obj_get_answer(mp_obj_t self_in) {
  return mp_obj_new_int(shared_module_mymodule_myclass_get_answer(self_in));
}
MP_DEFINE_CONST_FUN_OBJ_1(mymodule_myclass_get_answer_obj,
mymodule_myclass_obj_get_answer);
```

Finally there's code that defines the class locals. This usually serves to bind the method names to the interface functions defines above.

```
const mp_obj_property_t mymodule_myclass_question_obj = {
    .base.type = &amp;mp_type_property,
    .proxy = {(mp_obj_t)&amp;mymodule_myclass_get_question_obj,
              (mp_obj_t)&amp;mp_const_none_obj},
};

const mp_obj_property_t mymodule_myclass_answer_obj = {
    .base.type = &amp;mp_type_property,
    .proxy = {(mp_obj_t)&amp;mymodule_myclass_get_answer_obj,
              (mp_obj_t)&amp;mp_const_none_obj},
};

STATIC const mp_rom_map_elem_t mymodule_myclass_locals_dict_table[] = {
    // Methods
    { MP_ROM_QSTR(MP_QSTR_deinit), MP_ROM_PTR(&amp;mymodule_myclass_deinit_obj) },
    { MP_ROM_QSTR(MP_QSTR___enter__), MP_ROM_PTR(&amp;default___enter___obj) },
    { MP_ROM_QSTR(MP_QSTR___exit__),
MP_ROM_PTR(&amp;mymodule_myclass___exit___obj) },
    { MP_ROM_QSTR(MP_QSTR_question),
MP_ROM_PTR(&amp;mymodule_myclass_question_obj) },
    { MP_ROM_QSTR(MP_QSTR_answer), MP_ROM_PTR(&amp;mymodule_myclass_answer_obj) },
};
STATIC MP_DEFINE_CONST_DICT(mymodule_myclass_locals_dict,
mymodule_myclass_locals_dict_table);

const mp_obj_type_t mymodule_myclass_type = {
    { &amp;mp_type_type },
    .name = MP_QSTR_Meaning,
    .make_new = mymodule_myclass_make_new,
    .locals_dict = (mp_obj_dict_t*)&amp;mymodule_myclass_locals_dict,
};
```

## `ports/atmel-samd`

We'll need to edit two files to hook our new module into the build:

### `Makefile`

We need to add  the c files we added to `shared-module` to the `SRC_SHARED_MODULE` list:

`mymodule/__init__.c \`
`mymodule/MyClass.c \`

Note the reverse slash at the end of the lines. This is C's line continuation which is needed when defining a multi-line macro.

### `mpconfigport.h`

There are two places in this file that need an addition.

First we need to add our new module.  Look for a comment very similar to

`// extra built in modules to add to the list of known ones`

Add a line to the list immediately following it, similar to the rest. The difference in what you add will be that it mentions your new module:

`extern const struct _mp_obj_module_t mymodule_module;`

The second thing to do is add your module to the `EXTRA_BUILTIN_MODULES` macro, with a line like the others there:

`{ MP_OBJ_NEW_QSTR(MP_QSTR_mymodule), (mp_obj_t)&mymodule_module }, \`

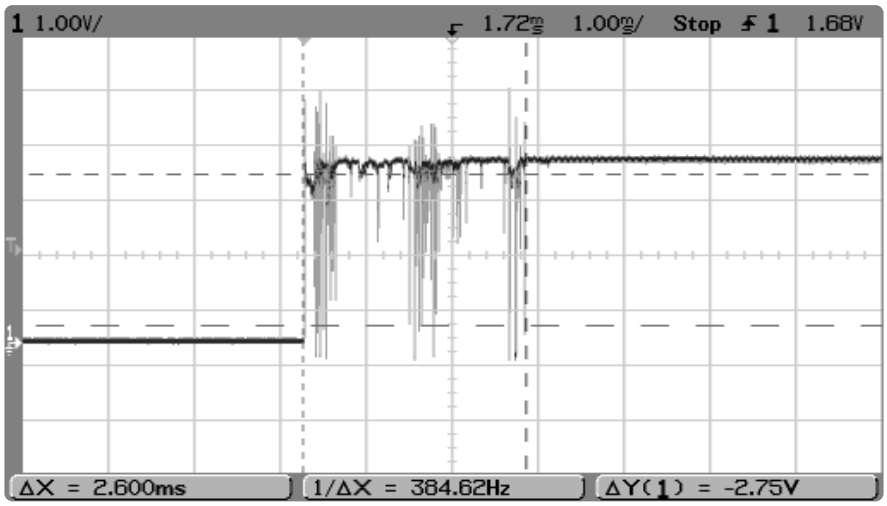
Don't forget that reverse slash at the end of the line.

## In Action

Now build CircuitPython for your board and try it out:

```
&gt;&gt;&gt; import mymodule
&gt;&gt;&gt; dir(mymodule)
['__class__', '__name__', 'MyClass']
&gt;&gt;&gt; dir(mymodule.MyClass)
['__class__', '__enter__', '__exit__', '__name__', 'answer', 'deinit', 'question']
&gt;&gt;&gt; m = mymodule.MyClass()
&gt;&gt;&gt; m.answer
42
&gt;&gt;&gt; m.question
'Tricky...'
```

# A Debouncer Module




Let's do something more involved now that we know the general layout. A debouncer is a good next step. Here's a CircuitPython implementation that's been used in a few of the author's guides.

```
import time
import digitalio

class Debouncer(object):
    """Debounce an input pin"""

    DEBOUNCED_STATE = 0x01
    UNSTABLE_STATE = 0x02
    CHANGED_STATE = 0x04

    def __init__(self, pin, interval=0.010):
        """Make am instance.
           :param int pin: the pin (from board) to debounce
           :param int mode: digitalio.Pull.UP or .DOWN (default is no pull up/down)
           :param int interval: bounce threshold in seconds (default is 0.010, i.e.
10 milliseconds)
        """
        self.state = 0x00
        self.pin = digitalio.DigitalInOut(pin)
        self.pin.direction = digitalio.Direction.INPUT
        self.pin.pull = digitalio.Pull.UP
        if self.pin.value:
            self.__set_state(Debouncer.DEBOUNCED_STATE | Debouncer.UNSTABLE_STATE)
        self.previous_time = 0
        if interval is None:
            self.interval = 0.010
        else:
            self.interval = interval

    def __set_state(self, bits):
        self.state |= bits

    def __unset_state(self, bits):
        self.state &amp;= ~bits

    def __toggle_state(self, bits):
        self.state ^= bits

    def __get_state(self, bits):
```

```
        return (self.state &amp;amp; bits) != 0

    def update(self):
        """Update the debouncer state. Must be called before using any of the
properties below"""
        self.__unset_state(Debouncer.CHANGED_STATE)
        current_state = self.pin.value
        if current_state != self.__get_state(Debouncer.UNSTABLE_STATE):
            self.previous_time = time.monotonic()
            self.__toggle_state(Debouncer.UNSTABLE_STATE)
        else:
            if time.monotonic() - self.previous_time &amp;gt;= self.interval:
                if current_state != self.__get_state(Debouncer.DEBOUNCED_STATE):
                    self.previous_time = time.monotonic()
                    self.__toggle_state(Debouncer.DEBOUNCED_STATE)
                    self.__set_state(Debouncer.CHANGED_STATE)

    @property
    def value(self):
        """Return the current debounced value of the input."""
        return self.__get_state(Debouncer.DEBOUNCED_STATE)

    @property
    def rose(self):
        """Return whether the debounced input went from low to high at the most
recent update."""
        return self.__get_state(self.DEBOUNCED_STATE) and
self.__get_state(self.CHANGED_STATE)

    @property
    def fell(self):
        """Return whether the debounced input went from high to low at the most
recent update."""
        return (not self.__get_state(self.DEBOUNCED_STATE)) and
self.__get_state(self.CHANGED_STATE)
```

## Implementation

We'll start with the implementation in `shared-module/debounce`. As before we need to add an `__init__.c` file that contains module level functions (i.e. there's nothing in it) . Then we need the class implementation files. First the header that defines the data side of the Python class: `Debouncer.h`

```
#ifndef MICROPY_INCLUDED_DEBOUNCE_DEBOUNCER_H
#define MICROPY_INCLUDED_DEBOUNCE_DEBOUNCER_H

#include "shared-bindings/digitalio/DigitalInOut.h"
#include "py/obj.h"

typedef struct {
  mp_obj_base_t base;
  uint8_t state;
  digitalio_digitalinout_obj_t pin;
  uint64_t previous_time;
  uint64_t interval;
} debounce_debouncer_obj_t;


#endif // MICROPY_INCLUDED_DEBOUNCE_DEBOUNCER_H
```

We'll go through the C file (`Debouncer.c`) a section at a time. First, the includes, constants, and external declarations. The latter is the way we can refer to a function or variable that is defined elsewhere, but which we want to use in this file. We don't have to identify where it is, the linker will figure that out during the build. In this case, we want the function that implements Python's `time.monotonic` function.

```
#include "common-hal/microcontroller/Pin.h"
#include "shared-bindings/digitalio/Pull.h"
#include "shared-bindings/digitalio/DigitalInOut.h"
#include "py/runtime.h"
#include "supervisor/shared/translate.h"
#include "Debouncer.h"


#define DEBOUNCED_STATE (0x01)
#define UNSTABLE_STATE (0x02)
#define CHANGED_STATE (0x04)


extern uint64_t common_hal_time_monotonic(void);
```

Next, we have the equivalent of the private methods in the Python version (prefixed by `__`). In C, we simply have to define them here and not make them available up to the runtime.

```
void set_state(debounce_debouncer_obj_t* self, uint8_t bits)
{
  self-&gt;state |= bits;
}

void unset_state(debounce_debouncer_obj_t* self, uint8_t bits)
{
  self-&gt;state &amp;= ~bits;
}

void toggle_state(debounce_debouncer_obj_t* self, uint8_t bits)
{
  self-&gt;state ^= bits;
}

uint8_t get_state(debounce_debouncer_obj_t* self, uint8_t bits)
{
  return (self-&gt;state &amp; bits) != 0;
}
```

Next, let's turn to the constructor and lifecycle functions. This time the constructor takes two parameters: the pin to debounce, and the debounce interval (how long to let the input settle before accepting it's value).

```
void shared_module_debounce_debouncer_construct(debounce_debouncer_obj_t* self,
                                                mcu_pin_obj_t* pin, mp_int_t interval)
{
  digitalinout_result_t result =
common_hal_digitalio_digitalinout_construct(&amp;self-&gt;pin, pin);
  if (result != DIGITALINOUT_OK) {
    return;
  }
  common_hal_digitalio_digitalinout_switch_to_input(&amp;self-&gt;pin, PULL_UP);
```

```
  self-&gt;state = 0x00;
  if (common_hal_digitalio_digitalinout_get_value(&amp;self-&gt;pin)) {
    set_state(self, DEBOUNCED_STATE | UNSTABLE_STATE);
  }
  self-&gt;interval = interval;
}

bool shared_module_debounce_debouncer_deinited(debounce_debouncer_obj_t* self) {
  return common_hal_digitalio_digitalinout_deinited(&amp;self-&gt;pin);
}

void shared_module_debounce_debouncer_deinit(debounce_debouncer_obj_t* self) {
  if (shared_module_debounce_debouncer_deinited(self)) {
    return;
  }
  common_hal_digitalio_digitalinout_deinit(&amp;self-&gt;pin);
}
```

Notice that we have something related to the implementation to tell when the instance is valid: the pin. The `deinited` function checks for a valid pin, and the `deinit` function releases the pin and invalidates it.

The `update` function does the work in this code.

When it's called, the function first grabs the current time and clears the changed state. Then it checks to see if the the pin is different from last time. If so, it restarts the settling timer and updates the state (to be checked against next time). If the pin hasn't changed since last time, it checks to see if the settling timeout has expired. If so, it's done unless the pin is now different than the most recent debounced state. If the pin is different, the time and debounced state are updated, and the change is noted. This change flag is used in the `rose` and `fell` properties which key off a change in the pin's debounced value.

```
void shared_module_debounce_debouncer_update(debounce_debouncer_obj_t* self) {
  if (shared_module_debounce_debouncer_deinited(self)) {
    return;
  }
  uint64_t now = common_hal_time_monotonic();
  unset_state(self, CHANGED_STATE);
  bool current_state = common_hal_digitalio_digitalinout_get_value(&amp;self-
&gt;pin);
  if (current_state != get_state(self, UNSTABLE_STATE)) {
    self-&gt;previous_time = now;
    toggle_state(self, UNSTABLE_STATE);
  } else {
    if (now - self-&gt;previous_time &gt;= self-&gt;interval) {
      if (current_state != get_state(self, DEBOUNCED_STATE)) {
        self-&gt;previous_time = now;
        toggle_state(self, DEBOUNCED_STATE);
        set_state(self, CHANGED_STATE);
      }
    }
  }
}
```

Finally, we have the property getter functions:
`value` - the most recent debounced state
`rose` - whether the debounced state is high and got there during the most recent call to `update`
`fell` - whether the debounced state is low and got there during the most recent call to `update`

```
bool shared_module_debounce_debouncer_get_value(debounce_debouncer_obj_t* self) {
  return get_state(self, DEBOUNCED_STATE);
}

bool shared_module_debounce_debouncer_get_rose(debounce_debouncer_obj_t* self) {
  return get_state(self, DEBOUNCED_STATE) &amp;&amp; get_state(self, CHANGED_STATE);
}

bool shared_module_debounce_debouncer_get_fell(debounce_debouncer_obj_t* self) {
  return !get_state(self, DEBOUNCED_STATE) &amp;&amp; get_state(self,
CHANGED_STATE);
}
```

## Interface

Again, there are no exposed method level functions, so `shared-bindings/debounce/__init__.h` is simply:

```
#ifndef MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE___INIT___H
#define MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE___INIT___H

#include "py/obj.h"

// Nothing now.

#endif  // MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE___INIT___H
```

The file `shared-bindings/debounce/__init__.c` ties in the Debouncer class:

```
#include &lt;stdint.h&gt;

#include "py/obj.h"
#include "py/runtime.h"

#include "shared-bindings/debounce/__init__.h"
#include "shared-bindings/debounce/Debouncer.h"

STATIC const mp_rom_map_elem_t debounce_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_debounce) },
    { MP_ROM_QSTR(MP_QSTR_Debouncer), MP_ROM_PTR(&amp;debounce_debouncer_type) },
};

STATIC MP_DEFINE_CONST_DICT(debounce_module_globals, debounce_module_globals_table);

const mp_obj_module_t debounce_module = {
    .base = { &amp;mp_type_module },
    .globals = (mp_obj_dict_t*)&amp;debounce_module_globals,
};
```

`shared-bindings/debounce/Debouncer.h` declares the exposed functions in the implementation (above) as `extern`. Remember this just tells the compiler that these functions will be available; it's up to the linker to find them and make the connections.

```
#ifndef MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE_DEBOUNCER_H
#define MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE_DEBOUNCER_H

#include "shared-module/debounce/Debouncer.h"

extern const mp_obj_type_t debounce_debouncer_type;

extern void shared_module_debounce_debouncer_construct(debounce_debouncer_obj_t*
self, mcu_pin_obj_t* pin, mp_int_t interval);
extern void shared_module_debounce_debouncer_deinit(debounce_debouncer_obj_t* self);
extern bool shared_module_debounce_debouncer_deinited(debounce_debouncer_obj_t*
self);
extern void shared_module_debounce_debouncer_update(debounce_debouncer_obj_t* self);
extern bool shared_module_debounce_debouncer_get_fell(debounce_debouncer_obj_t*
self);
extern bool shared_module_debounce_debouncer_get_rose(debounce_debouncer_obj_t*
self);
extern bool shared_module_debounce_debouncer_get_value(debounce_debouncer_obj_t*
self);

#endif // MICROPY_INCLUDED_SHARED_BINDINGS_DEBOUNCE_DEBOUNCER_H
```

As before (but more complex this time) the interface source file (`shared-bindings/debounce/Debouncer.c`) does the job of plumbing the implementation code into the CircuitPython runtime. This is still fairly straightforward as only the constructor has parameters.

```
#include &lt;stdint.h&gt;

#include "lib/utils/context_manager_helpers.h"
#include "py/objproperty.h"
#include "py/runtime.h"
#include "py/runtime0.h"
#include "shared-bindings/microcontroller/Pin.h"
#include "shared-bindings/debounce/Debouncer.h"
#include "shared-bindings/util.h"

//| .. currentmodule:: debounce
//|
//| :class:`Debouncer` -- Debounce an input pin
//|
//|
//| Debouncer cleans up an input pin and provides value/rise/fall properties.
//|
//| .. class:: Debouncer(pin, mode, interval)
//|
//|   Create a Debouncer object associated with the given pin. It tracks the value
of the pin over time,
//|   allowing it to settle before acknowledging transitions.
//|
//|   :param ~microcontroller.Pin pin: Pin to debounce
//|   :param ~int interval: debounce interval in milliseconds
//|
//|   For example::
//|
//|     import debounce
//|     import board
```

```
//|
//|     deb = bebounce.Debounce(board.D10, 10)
//|     while True:
//|         deb.update()
//|         if deb.fell
//|             print("Pressed")
//|
STATIC mp_obj_t debounce_debouncer_make_new(const mp_obj_type_t *type, size_t
n_args, const mp_obj_t *pos_args, mp_map_t *kw_args) {
    enum { ARG_pin, ARG_interval };
    static const mp_arg_t allowed_args[] = {
        { MP_QSTR_pin_, MP_ARG_REQUIRED | MP_ARG_OBJ },
        { MP_QSTR_interval, MP_ARG_INT },
    };
    mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
    mp_arg_parse_all(n_args, pos_args, kw_args, MP_ARRAY_SIZE(allowed_args),
allowed_args, args);

    assert_pin(args[ARG_pin].u_obj, false);
    mcu_pin_obj_t* pin = MP_OBJ_TO_PTR(args[ARG_pin].u_obj);

    mp_int_t interval = 10;
    if (n_args == 2) {
      interval = args[ARG_interval].u_int;
    }

    debounce_debouncer_obj_t *self = m_new_obj(debounce_debouncer_obj_t);
    self-&gt;base.type = &amp;debounce_debouncer_type;

    shared_module_debounce_debouncer_construct(self, pin, interval);

    return MP_OBJ_FROM_PTR(self);
}


//|   .. method:: deinit()
//|
//|     Deinitializes the debouncer and releases any hardware resources for reuse.
//|
STATIC mp_obj_t debounce_debouncer_deinit(mp_obj_t self_in) {
    debounce_debouncer_obj_t *self = MP_OBJ_TO_PTR(self_in);
    shared_module_debounce_debouncer_deinit(self);
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_1(debounce_debouncer_deinit_obj,
debounce_debouncer_deinit);

//|   .. method:: update()
//|
//|     Do an update cycle it it's time to.
//|
STATIC mp_obj_t debounce_debouncer_obj_update(size_t n_args, const mp_obj_t *args) {
    (void)n_args;
    shared_module_debounce_debouncer_update(args[0]);
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(debounce_debouncer_update_obj, 1, 1,
debounce_debouncer_obj_update);


//|   .. method:: __exit__()
//|
//|     Automatically deinitializes the hardware when exiting a context. See
//|     :ref:`lifetime-and-contextmanagers` for more info.
//|
STATIC mp_obj_t debounce_debouncer_obj___exit__(size_t n_args, const mp_obj_t
*args) {
    (void)n_args;
    shared_module_debounce_debouncer_deinit(args[0]);
```

```
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(debounce_debouncer___exit___obj, 4, 4,
debounce_debouncer_obj___exit__);


//|   .. attribute:: value
//|
//|     The current debounced value
//|
STATIC mp_obj_t debounce_debouncer_obj_get_value(mp_obj_t self_in) {
    debounce_debouncer_obj_t *self = MP_OBJ_TO_PTR(self_in);
    raise_error_if_deinited(shared_module_debounce_debouncer_deinited(self));

    return mp_obj_new_bool(shared_module_debounce_debouncer_get_value(self));
}
MP_DEFINE_CONST_FUN_OBJ_1(debounce_debouncer_get_value_obj,
debounce_debouncer_obj_get_value);


//|   .. attribute:: rose
//|
//|     Whether the input transitioned low to high since last update
//|
STATIC mp_obj_t debounce_debouncer_obj_get_rose(mp_obj_t self_in) {
    debounce_debouncer_obj_t *self = MP_OBJ_TO_PTR(self_in);
    raise_error_if_deinited(shared_module_debounce_debouncer_deinited(self));

    return mp_obj_new_bool(shared_module_debounce_debouncer_get_rose(self));
}
MP_DEFINE_CONST_FUN_OBJ_1(debounce_debouncer_get_rose_obj,
debounce_debouncer_obj_get_rose);


//|   .. attribute:: value
//|
//|     Whether the input transitioned high to low since last update
//|
STATIC mp_obj_t debounce_debouncer_obj_get_fell(mp_obj_t self_in) {
    debounce_debouncer_obj_t *self = MP_OBJ_TO_PTR(self_in);
    raise_error_if_deinited(shared_module_debounce_debouncer_deinited(self));

    return mp_obj_new_bool(shared_module_debounce_debouncer_get_fell(self));
}
MP_DEFINE_CONST_FUN_OBJ_1(debounce_debouncer_get_fell_obj,
debounce_debouncer_obj_get_fell);


const mp_obj_property_t debounce_debouncer_value_obj = {
    .base.type = &amp;mp_type_property,
    .proxy = {(mp_obj_t)&amp;debounce_debouncer_get_value_obj,
              (mp_obj_t)&amp;mp_const_none_obj},
};

const mp_obj_property_t debounce_debouncer_rose_obj = {
    .base.type = &amp;mp_type_property,
    .proxy = {(mp_obj_t)&amp;debounce_debouncer_get_rose_obj,
              (mp_obj_t)&amp;mp_const_none_obj},
};

const mp_obj_property_t debounce_debouncer_fell_obj = {
    .base.type = &amp;mp_type_property,
    .proxy = {(mp_obj_t)&amp;debounce_debouncer_get_fell_obj,
              (mp_obj_t)&amp;mp_const_none_obj},
};

STATIC const mp_rom_map_elem_t debounce_debouncer_locals_dict_table[] = {
    // Methods
    { MP_ROM_QSTR(MP_QSTR_deinit), MP_ROM_PTR(&amp;debounce_debouncer_deinit_obj) },
```

```
    { MP_ROM_QSTR(MP_QSTR___enter__), MP_ROM_PTR(&amp;default___enter___obj) },
    { MP_ROM_QSTR(MP_QSTR___exit__),
MP_ROM_PTR(&amp;debounce_debouncer___exit___obj) },
    { MP_ROM_QSTR(MP_QSTR_update), MP_ROM_PTR(&amp;debounce_debouncer_update_obj) },
    { MP_ROM_QSTR(MP_QSTR_value), MP_ROM_PTR(&amp;debounce_debouncer_value_obj) },
    { MP_ROM_QSTR(MP_QSTR_rose), MP_ROM_PTR(&amp;debounce_debouncer_rose_obj) },
    { MP_ROM_QSTR(MP_QSTR_fell), MP_ROM_PTR(&amp;debounce_debouncer_fell_obj) },
};
STATIC MP_DEFINE_CONST_DICT(debounce_debouncer_locals_dict,
debounce_debouncer_locals_dict_table);

const mp_obj_type_t debounce_debouncer_type = {
    { &amp;mp_type_type },
    .name = MP_QSTR_Debouncer,
    .make_new = debounce_debouncer_make_new,
    .locals_dict = (mp_obj_dict_t*)&amp;debounce_debouncer_locals_dict,
};
```

Notice how each function has a form similar to the following:

```
STATIC mp_obj_t debounce_debouncer_obj_get_rose(mp_obj_t self_in) {
    debounce_debouncer_obj_t *self = MP_OBJ_TO_PTR(self_in);
    raise_error_if_deinited(shared_module_debounce_debouncer_deinited(self));

    return mp_obj_new_bool(shared_module_debounce_debouncer_get_rose(self));
}
MP_DEFINE_CONST_FUN_OBJ_1(debounce_debouncer_get_rose_obj,
debounce_debouncer_obj_get_rose);
```

Particularly, notice the final line: `MP_DEFINE_CONST_FUN_OBJ_1` is what does the plumbing.

The final part of the file defines what is known about the Python class (notice we use plain C to define the Python classes):

```
STATIC const mp_rom_map_elem_t debounce_debouncer_locals_dict_table[] = {
    // Methods
    { MP_ROM_QSTR(MP_QSTR_deinit), MP_ROM_PTR(&amp;debounce_debouncer_deinit_obj) },
    { MP_ROM_QSTR(MP_QSTR___enter__), MP_ROM_PTR(&amp;default___enter___obj) },
    { MP_ROM_QSTR(MP_QSTR___exit__),
MP_ROM_PTR(&amp;debounce_debouncer___exit___obj) },
    { MP_ROM_QSTR(MP_QSTR_update), MP_ROM_PTR(&amp;debounce_debouncer_update_obj) },
    { MP_ROM_QSTR(MP_QSTR_value), MP_ROM_PTR(&amp;debounce_debouncer_value_obj) },
    { MP_ROM_QSTR(MP_QSTR_rose), MP_ROM_PTR(&amp;debounce_debouncer_rose_obj) },
    { MP_ROM_QSTR(MP_QSTR_fell), MP_ROM_PTR(&amp;debounce_debouncer_fell_obj) },
};
STATIC MP_DEFINE_CONST_DICT(debounce_debouncer_locals_dict,
debounce_debouncer_locals_dict_table);

const mp_obj_type_t debounce_debouncer_type = {
    { &amp;mp_type_type },
    .name = MP_QSTR_Debouncer,
    .make_new = debounce_debouncer_make_new,
    .locals_dict = (mp_obj_dict_t*)&amp;debounce_debouncer_locals_dict,
};
```

Changes need to be made to `Makefile` and `mpconfigport.h` similar to those on the previous page, except that they reference the `debounce` module.

# Going On From Here


The CircuitPython runtime provides a wealth of support for adding modules in C. As mentioned, documentation is sparse and the best way to learn what you need is to explore the modules that are already present.

This guide provides an overview of what's involved in writing a port independant module in C. You are not limited in what you can add in this way. CircuitPython provides what you need to interface with most hardware you will want, but puts limits on the performance of your code. There are times you will want to use algorithms that benefit from very high performance. Implementing them in C and exposing them as Python modules is a way to get that level of performance.

There is also a way to add modules that rely on specific board capabilities. We'll explore that in a future quide.