Adafruit CircuitPython Register

Release 0.1

Feb 06, 2018

Contents

This library provides a variety of data descriptor class for Adafruit CircuitPython that makes it really simple to write a device drivers for a I2C and SPI register based devices. Data descriptors act like basic attributes from the outside which makes using them really easy to use.

CHAPTER 1

API

1.1 Module Reference

1.1.1 I2C

`i2c_bit` - Single bit registers

`i2c_bit` - Single bit registers

class ROBit(*register_address*, *bit*)

Single bit register that is read only. Subclass of *RWBit*.

Values are `bool`

Parameters

- **register_address** (*int*) – The register address to read the bit from
- **bit** (*type*) – The bit index within the byte at `register_address`

class RWBit(*register_address*, *bit*)

Single bit register that is readable and writeable.

Values are `bool`

Parameters

- **register_address** (*int*) – The register address to read the bit from
- **bit** (*type*) – The bit index within the byte at `register_address`

`i2c_bits` - Multi bit registers

`i2c_bits` - Multi bit registers

class ROBits(*num_bits*, *register_address*, *lowest_bit*)

Multibit register (less than a full byte) that is read-only. This must be within a byte register.

Values are `int` between 0 and 2 ** `num_bits` - 1.

Parameters

- **num_bits** (*int*) – The number of bits in the field.
- **register_address** (*int*) – The register address to read the bit from
- **lowest_bit** (*type*) – The lowest bits index within the byte at `register_address`

class RWBits(*num_bits*, *register_address*, *lowest_bit*)

Multibit register (less than a full byte) that is readable and writeable. This must be within a byte register.

Values are `int` between 0 and 2 ** `num_bits` - 1.

Parameters

- **num_bits** (*int*) – The number of bits in the field.
- **register_address** (*int*) – The register address to read the bit from
- **lowest_bit** (*type*) – The lowest bits index within the byte at `register_address`

`i2c_struct` - Generic structured registers based on `struct`

`i2c_struct` - Generic structured registers based on `struct`

class Struct(*register_address*, *struct_format*)

Arbitrary structure register that is readable and writeable.

Values are tuples that map to the values in the defined struct. See struct module documentation for struct format string and its possible value types.

Parameters

- **register_address** (*int*) – The register address to read the bit from
- **struct_format** (*type*) – The struct format string for this register.

class UnaryStruct(*register_address*, *struct_format*)

Arbitrary single value structure register that is readable and writeable.

Values map to the first value in the defined struct. See struct module documentation for struct format string and its possible value types.

Parameters

- **register_address** (*int*) – The register address to read the bit from
- **struct_format** (*type*) – The struct format string for this register.

`i2c_bcd_datetime` - Binary Coded Decimal date and time register

`i2c_bcd_datetime` - Binary Coded Decimal date and time register

class BCDDateTimeRegister(*register_address*, *weekday_first=True*, *weekday_start=1*)

Date and time register using binary coded decimal structure.

The byte order of the register must* be: second, minute, hour, weekday, day (1-31), month, year (in years after 2000).

- Setting weekday_first=False will flip the weekday/day order so that day comes first.

Values are `time.struct_time`

Parameters

- **register_address** (*int*) – The register address to start the read
- **weekday_first** (*bool*) – True if weekday is in a lower register than the day of the month (1-31)
- **weekday_start** (*int*) – 0 or 1 depending on the RTC's representation of the first day of the week

`i2c_bcd_alarm` - Binary Coded Decimal alarm register

`i2c_bcd_alarm` - Binary Coded Decimal alarm register

class BCDAlarmTimeRegister(*register_address*, *has_seconds=True*, *weekday_shared=True*, *weekday_start=1*)

Alarm date and time register using binary coded decimal structure.

The byte order of the registers must* be: [second], minute, hour, day, weekday. Each byte must also have a high enable bit where 1 is disabled and 0 is enabled.

- If weekday_shared is True, then weekday and day share a register.
- If has_seconds is True, then there is a seconds register.

Values are a tuple of (`time.struct_time`, `str`) where the struct represents a date and time that would alarm. The string is the frequency:

- "secondly", once a second (only if alarm has_seconds)
- "minutely", once a minute when seconds match (if alarm doesn't seconds then when seconds = 0)
- "hourly", once an hour when `tm_min` and `tm_sec` match
- "daily", once a day when `tm_hour`, `tm_min` and `tm_sec` match
- "weekly", once a week when `tm_wday`, `tm_hour`, `tm_min`, `tm_sec` match
- "monthly", once a month when `tm_mday`, `tm_hour`, `tm_min`, `tm_sec` match

Parameters

- **register_address** (*int*) – The register address to start the read
- **has_seconds** (*bool*) – True if the alarm can happen minutely.
- **weekday_shared** (*bool*) – True if weekday and day share the same register

- **weekday_start** (*int*) – 0 or 1 depending on the RTC's representation of the first day of the week (Monday)

1.1.2 SPI

Coming soon!

CHAPTER 2

Creating a driver

Creating a driver with the register library is really easy. First, import the register modules you need from the available modules:

```
from adafruit_register import i2c_bit
from adafruit_bus_device import i2c_device
```

Next, define where the bit is located in the device's memory map:

```
class HelloWorldDevice:
    """Device with two bits to control when the words 'hello' and 'world' are lit."""

    hello = i2c_bit.RWBit(0x0, 0x0)
    """Bit to indicate if hello is lit."""

    world = i2c_bit.RWBit(0x1, 0x0)
    """Bit to indicate if world is lit."""
```

Lastly, we need to add an `i2c_device` member of type `I2CDevice` that manages sharing the I2C bus for us. Make sure the name is exact, otherwise the registers will not be able to find it. Also, make sure that the i2c device implements the `busio.I2C` interface.

```
def __init__(self, i2c, device_address=0x0):
    self.i2c_device = i2c_device.I2CDevice(i2c, device_address)
```

Thats it! Now we have a class we can use to talk to those registers:

```
import busio
from board import *

with busio.I2C(SCL, SDA) as i2c:
    device = HelloWorldDevice(i2c)
    device.hello = True
    device.world = True
```

CHAPTER 3

Adding register types

Adding a new register type is a little more complicated because you need to be careful and minimize the amount of memory the class will take. If you don't, then a driver with five registers of your type could take up five times more extra memory.

First, determine whether the new register class should go in an existing module or not. When in doubt choose a new module. The more finer grained the modules are, the fewer extra classes a driver needs to load in.

Here is the start of the *RWBit* class:

```python
class RWBit:
    """
    Single bit register that is readable and writeable.

    Values are `bool`

    :param int register_address: The register address to read the bit from
    :param type bit: The bit index within the byte at ``register_address``
    """
    def __init__(self, register_address, bit):
        self.bit_mask = 1 << bit
        self.buffer = bytearray(2)
        self.buffer[0] = register_address
```

The first thing done is writing an RST formatted class comment that explains the functionality of the register class and any requirements of the register layout. It also documents the parameters passed into the constructor (`__init__`) which configure the register location in the device map. It does not include the device address or the i2c object because its shared on the device class instance instead. That way if you have multiple of the same device on the same bus, the register classes will be shared.

In `__init__` we only use two member variable because each costs 8 bytes of memory plus the memory for the value. And remember this gets multiplied by the number of registers of this type in a driver! Thats why we pack both the register address and data byte into one bytearray. We could use two byte arrays of size one but each MicroPython object is 16 bytes minimum due to the garbage collector. So, by sharing a byte array we keep it to the 16 byte minimum instead of 32 bytes. Each `memoryview` also costs 16 bytes minimum so we avoid them too.

Another thing we could do is allocate the `bytearray` only when we need it. This has the advantage of taking less memory up front but the cost of allocating it every access and risking it failing. If you want to add a version of `Foo` that lazily allocates the underlying buffer call it `FooLazy`.

Ok, onward. To make a data descriptor we must implement `__get__` and `__set__`.

```
def __get__(self, obj, objtype=None):
    with obj.i2c_device:
        obj.i2c_device.write(self.buffer, end=1, stop=False)
        obj.i2c_device.readinto(self.buffer, start=1)
    return bool(self.buffer[1] & self.bit_mask)

def __set__(self, obj, value):
    with obj.i2c_device:
        obj.i2c_device.write(self.buffer, end=1, stop=False)
        obj.i2c_device.readinto(self.buffer, start=1)
        if value:
            self.buffer[1] |= self.bit_mask
        else:
            self.buffer[1] &= ~self.bit_mask
        obj.i2c_device.write(self.buffer)
```

As you can see, we have two places to get state from. First, `self` stores the register class members which locate the register within the device memory map. Second, `obj` is the driver class that uses the register class which must by definition provide a `I2CDevice` compatible object as `i2c_device`. This object does two thing for us:

1. Waits for the bus to free, locks it as we use it and frees it after.
2. Saves the device address and other settings so we don't have to.

Note that we take heavy advantage of the `start` and `end` parameters to the i2c functions to slice the buffer without actually allocating anything extra. They function just like `self.buffer[start:end]` without the extra allocation.

Thats it! Now you can use your new register class like the example above. Just remember to keep the number of members to a minimum because the class may be used a bunch of times.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index