--- T-H-I-N-X ---

WebIOPi Device Driver Guide

White Paper

Version 1.2 (November 2014)

ANDREAS RIEGG

© 2014 BY ANDREAS RIEGG [BRAIN (AT) T-H-I-N-X.NET]

WebIOPi © 2012 - 2014 BY ERIC PTAK [TROUCH (AT) TROUCH.COM]

Change log

Version	Date	Author	Changes
0.6	July 1st, 2014	Andreas Riegg	Initial version
0.8	Aug 20th, 2014	Andreas Riegg	Optimized chapter structure, completed content of several (sub)chapters, added some (partly incomplete) new sub chapters, added a FAQ chapter, updated example code to v1.1, added a change log table
1.0	Aug 28th, 2014	Andreas Riegg	Changed name of the document, completed the last sub chapters that were incomplete, added the "Device communication and control technology" chapter, went through the whole document and corrected typos and misunderstandable formulations, updated wildcard Clock REST mapping to "clock/*"
1.1	Sep 29th, 2014	Andreas Riegg	Added driver checklist, added hint on close(), added text and code for osrtc.py, added chapter on test and documentation, added swagger spec for RTC clock API
1.2	Nov 20th, 2014	Andreas Riegg	Added further explanations and examples on I2C and SPI class usage.

Content

Change log 2
Licensing & Copyright 5
Preface 5
Hardware abstraction layer(s) 6
    Introduction 6
    Basic device functionality abstraction 6
    Device connectivity and control abstraction 9
    Usage of Inheritance 10
Technical driver implementation 12
    Overview 12
    Driver code locations 12
    Coding style and conventions 13
    Device classes 18
    Device parameter handling 18
    Special coding patterns 22
    Logging handling 25
    Test and debugging handling 25
    Warning/error/exception handling 26
Technical driver integration 27
    Driver framework integration 27
    JScript and Device Monitor integration 28
    WebIOPi setup/installation integration 28
Device communication and control technology 30
    Bus class 30
    I2C communication 30
    SPI communication 33
    Serial communication 33
    1-wire communication 33
Driver test and documentation 36
    Test cases 36
    Test console 36
    Documentation 40
Driver quick checklist 41
Troubleshooting/FAQ 42

Appendices 43
Code of __init__.py for RTC Clock abstraction class..43
Code of dsrtc.py for Dallas/Maxim RTC chip drivers 48
Code of osrtc.py for system clock driver 52
Code (changes) of webiopi.js for RTC Clock abstraction class...54
Swagger API specification of RTC Clock REST API 56
References 61

Licensing & Copyright

This paper is ruled by the following Copyright statement:

T-H-I-N-X by Andreas Riegg is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License.

The source code in this paper is ruled by the following Copyright statement:


```
Copyright © 2014 Andreas Riegg
WebIOPi Copyright © 2012 - 2014 Eric Ptak


Licensed under the Apache License, Version 2.
 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```


Preface

Some of the concepts used here are based on a generic IoT things server concept. To avoid redundancies these aspects are not repeated here. So if you miss a deeper explanation for something, please lookup in this reference [THINX_2014].

The driver details here are documented for WebIOPi and the used Python language. However, all general concepts are valid for any other IoT device driver server and any other implementation language as long as it supports the most important object-oriented concepts.

All code examples provided here are extracted from the RTC category device drivers.

```
Text in framed courier font denotes code fragments.
```

Text in **`shaded courier font`** denotes references to source code elements.

Code fragments marked in red denote changes or additions.

Text in ***bold italics*** is preliminary and to be done or to be completed.

Hardware abstraction layer(s)

Introduction

A common design principle to reduce the amount of code that has to be developed but also to reduce the items that have to be learned and understood is the concept of abstraction. If abstraction is used in the context of an object oriented programming language like Python, one of the most important concept to implement this abstraction is (single and/or multiple) inheritance. The driver implementation framework of WebIOPi uses both concepts. Therefore it is necessary to first briefly explain WebIOPI abstraction in this chapter before starting to show plain driver coding rules in the following chapters. However, this chapter is not intended to provide a complete summer camp on abstraction concepts, it only points out the most important aspects and how they influence driver coding.

Basic device functionality abstraction

Categories and channels

This kind of abstraction is used to implement the TouchPoints concept explained in the abstract internet of things (IoT) resources document [THINX_2014]. The physical connection nature of TouchPoints is mainly modeled using device **categories** and device **channels**.

WebIOPi already implements the most important common device categories like **analog I/O**, **digital I/O** and a bunch of popular **sensors**. In the case of sensors, the physical type of sensor gets added sequentially to the category (like **sensor/temperature** or **sensor/pressure**). You find more examples for categories in the referenced document. The basic abstraction is achieved by putting the code for the different channels into different Python packages that are named according to the device category. On next level, some classes are defined that implement the most parts of the chip independent (and thus abstract) driver functionalities and partly by using also inheritance within those classes. The results of this step are the classes **`ADC`**, **`DAC`** and **`PWM`** for **analog** devices; the class **`GPIOPort`** for **digital** devices, the classes **`Pressure`**, **`Temperature`**, **`Luminosity`**, **`Distance`** and **`Humidity`** for **sensor** devices. For our RTC chips example, this leads to the new **clock** Package and a new **`Clock`** class. Typically, all the abstract code and the associated classes are contained within the **`__init__().py`** files of the device packages.

WebIOPi also implements the most common device channels. For analog I/O and digital I/O devices, this is just the port numbering scheme (either **sequential numbers (0, 1, 2, ...)** for separate I/O chips or **nominal numbers (4, 17, 25, ...)** for the native processor ports). For some sensors also channels exist like **/pressure/sea**. You find more examples for channels in the resources document. For our RTC chips, these channels are **datetime**, **date**, **time**, ...

As a bottom line, you find the implementation locations for basic device category abstraction via the names of the Python packages and locate the abstract classes defined in the **`__init__().py`** files of those packages. But there is no strict 1:1 relationship between the abstraction and the Python elements. The Python elements are more built in a way so that implementation is as simple as possible and to some extent also as a result of the historic evolution of WebIOPi.

Situations exist where a single chip has more than one category of functions. In such cases the driver should be put into the package that represents the main category of function. In the case of breakouts or shields that contain multiple chips with multiple categories of functions it is best to put the drivers into the shields package.

Public methods/interfaces of the abstract device functionality

While the internal detailed implementation structure for the abstract code should still be considered somewhat private and maybe also subject to be changed over time, the most important thing to users of the abstraction

framework is the public interface of the devices. These interfaces represent the functionality that is meant to be officially available for all remote users of the devices. Technically, these public interfaces are represented by public Python methods of the abstraction device classes plus some additional public methods of concrete device (sub)classes.

As Python does not have explicit public/private language decorators like e.g. Java, the Python convention using the pre/post double underscores is used (`"__xxxxx__"`). Additionally, if public methods retrieve or set some device (channel) value, the prefixes **get...** or **set...** are used.

Consequently, the opposite is also very important: All private methods of the abstraction classes and the concrete device classes are meant **NOT** to be used from outside at **ANY** time. And – they may change over time.

For our RTC chips, the corresponding public methods to retrieve values are `getDateTime()`, `getDateTimeString()`, `getDate()`, `getDateString()`, `getTime()`, `getTimeString()`, plus a whole bunch of methods to get single time and date slots like `getSec()` or `getMon()`.

In order to allow easy consistence checking and avoiding intense client roundtrips for updates, the setting of clock values has been restricted to complete date and time value (strings) only. Especially for dates it is very important to check for potential wrong values like e.g. trying to set the day of month February to 30 which would be wrong. Thus, no public methods are available to set single days, hours or seconds. Only methods to set a complete date (`setDate()`, `setDateString()`) or complete time (`setTime()`, `setTimeString()`) or a complete datetime (`setDateTime()`),`setDateTimeString()` are made public available.

Some of the public methods are coded to work with rich Python type objects as parameters (e.g. datetime.time class), others with just Python primitive type objects as parameters (e.g. integer). This difference plays a role for the REST mapping (see below).

Some devices implement a wild card interface (mapped to path ".../*") that answers the complete device state as JSON object. For RTC chips this is provided by `clockWildcard()`.

REST mappings

Having the public/private discussion of the previous chapter in mind, only public methods should get a REST mapping - but not all public methods really NEED a REST mapping. Remember, REST calls are typically done via a networked connection that introduces latencies which will increase round-trip times of calls very significant compared to local calls. Plus, the REST API defines the remote interface of each device and it is a good practice not to clutter this interface with "microscopic" externalization of every bit and chip register. And, finally, REST calls can occur in any sequence from any remote client and may be managed multi-threaded by a REST handler. As consequence, you can't heavily rely on the final order of execution of them within the server when called via a networked connection.

However, non REST-mapped public methods still make sense as they can be called much faster and in defined sequence local on the server and as they can be used to work more in Python-like style having richer Python objects as parameters (e.g. within the WebIOPi Python server scripts). Anyway, as they are not necessarily needed to get the REST stuff working, this can be still considered as an optional item and does not have to be implemented for every abstraction category.

WebIOPi REST mappings are added via the Python decorators `@request` and `@response`. Please look at some Python manual to understand what a decorator is and how it works.

IMPORTANT: Only devices that get created via a config file entry are being mapped to the REST API. Devices that are created manually within a Python server script are **NOT** mapped to the REST API. However, methods that carry the decorators remain fully available within the server scripts for manually created devices. For more details on this, see WebIOPi wiki and the chapter on class initialization within this manual.

IMPORTANT 2: Vice versa, devices created within a server script do **NOT** show up in the Device Monitor as showing them and accessing their attributes needs to have them available via the REST API.

The `@request` decorator allows specifying the HTTP/CoAP verb of the request (currently GET or POST) plus a pattern-like string that allows to set the URL path parameters for the mapped method. %-characters are used to handle additional replace-by or derive-from mappings for channels and/or values. This resembles highly the usage of the %-character within Python strings, so again check a Python manual for details on this. However, this is currently only implemented for primitive Python type objects like integer, float and string. It is not available for any kind of rich Python types. Together with the fact that WebIOPi POST calls only use path-encoded parameters (no header variables, no payload content) these are the reasons why some public methods carry a "...String" in its name and handle the parameter transfer using strings.

The `@response` decorator allows specifying the content (mime) type of the result plus also a string pattern to format the result value. The content type defaults to "text/plain", the formatting expression to "%s" which is just a string.

For our RTC chips, the REST mappings to retrieve some values (-> verb GET) are set to the path `/clock/datetime` for the method `getDateTimeString()`, to the path `/clock/date` for the method `getDateString()` and to the path `/clock/time` for the method `getTimeString()`. In order to make it possible to retrieve single value slots via REST without the need to do string pattern processing on the client, also mappings for all individual slots/channels like the path `/clock/seconds` for method `getSec()` or the path `/clock/year` for the method `getYrs()` have been implemented.

The REST mappings to update some values (-> verb POST) are set to the path `/clock/date` for the method `setDateString()`, to the path `/clock/time` for the method `setTimeString()` and to the path `/clock/datetime` for the method `setDateTimeString()`. Additionally, a mapping to set the day-of-week (dow) has been added using path `/clock/dow` for method `setDow()`. Otherwise, no possibility to set the dow value would have been available.

As already mentioned, no REST mappings exist to set individual date and time single value slots like e.g. minute.

And, for the reasons also discussed above, the public methods `getDateTime()`, `getDate()`, `getTime()`, `setDateTime()`, `setDate()`, `setTime()` do not have a REST mapping as they use rich Python types as parameters.

You can find a compact summary of the complete RTC Clock REST API in the chapter on Driver test and documentation within this document.

Device functionality abstraction contracts

In most cases, the abstract device functionality can only be implemented partly on the top level; other parts of the implementation have to be delegated to the concrete device classes on lower level(s). In order to orchestrate this, the specific parts are implemented and invoked on abstract level by using private methods that have an "empty" implementation (generating a `NotImplementedError` exception) on that hierarchy level. To avoid those exceptions (which would be a driver bug), every "empty" implementation has to be re-implemented on one of the lower class hierarchy levels. This kind of code design represents some sort of "agreement" nature and is therefore called an "abstraction contract".

For RTC chips, such "empty" methods exist for all single date and time value slots like `__getSec__()`, `__getMin__()`, `__setSec__()`, `__setMin__()`, etc. The reason for this is that the value slots have to be accessed via registers that may vary for different chips and/or chip vendors.

To some extent "more complex" public methods can be implemented on abstraction level using the simple abstract "primitives". This acts as some kind of default implementation. The RTC methods `__getDateTime__()`, `__setDateTime__()`, `__getDate__()`, `__setDate__()`, `__getTime__()` and `__setTime__()` are examples for this. They are all implemented using the primitives `__getSec__()`, `__getMin__()` etc. This leads to the fact that every subclass that implements all primitives also has those complex methods. Sometimes it is possible to speed up such default methods by reimplementing them in concrete device classes and use features like sequential register access. This will save a lot of basic device communication roundtrips and this will improve the speed of the default implementations. The reimplemented versions of all the methods above within the `DSclock` class are an example for this.

While this may look quite simple for the RTC chips because their internal register design is very similar even between different chip vendors , this is not so common for other categories of chips and is one of the most important abstraction tasks to be solved by WebIOPi driver developers. So, please do not underestimate this task and take care to think really severe about this.

On the other hand, if this job is done in a very good way (or already provided by WebIOPi), adding drivers for new chips of an already existing abstraction category is much easier and faster and speeds up driver development a lot. Just take a look at some of the driver classes that are derived from others and see how less code is needed in some cases (sometimes, just a few lines). This is a huge improvement compared to driver development with e.g. C language. Python device drivers will never be so fast like C ones (but the high C speed is not necessary anymore in most cases nowadays), but their overall footprint and textual readability is much superior compared to C.

Helper functions

Further reducing driver code footprint and achieving high driver quality is supported by providing helper functions from abstract classes to chip sub classes. These helper functions implement code that will be needed within all kinds of concrete chip classes and it really makes sense to implement this code only once. Additionally, these helper functions also should utilize (where appropriate) the private methods to get/set value slots to make them independent of concrete chips.

For RTC chips, some functions to do BCD calculations (`BcdBits2Int()`, `Int2BcdBits()`), some value checks for calendar features (`checkYear()`, `checkDow()`) and some formatting helpers (`DateValues2String()`, `String2DateValues()`, ...) are provided on abstract level. Notice a simple trick: In principle, the date and time strings submitted to methods like `setDateString()` would have also to be checked. However, this task is delegated to the Python classes `date` and `time` as when creating instances of them, these classes do also a value checking. This just saves code footprint.

Device connectivity and control abstraction

This part of WebIOPi implements the low-level hardware bus layer for all devices (I2C, SPI, Serial, 1-Wire, ...). All drivers for chips that are accessible via these interfaces have to inherit from **ONE** of these classes as a concrete chip can typically only implement one of these hardware interfaces. Within these bus classes all methods can be found to do low-level interaction with those buses (like reading and writing basic bus bytes) accompanied by some very helpful medium-level convenience methods (like reading and writing to I2C device registers). In some cases chips with identical base functionality are available in variants for different buses (mostly I2C and SPI). In such situations the inheritance of the different bus classes may be delayed to a deeper level of the driver class hierarchy. See the implementation of the MCP23XXX chips for an example how to do this. In some rare cases chips implement more than one interface within the same hardware package. If so, it is still recommended to develop a dedicated driver class for each protocol variant. You may add an interface indicator to the class name to indicate this like e.g. ..._I for I2C and ..._S for SPI.

As WebIOPi already contains classes for most of the common used device connectivity protocols an extension at this point is really very seldom needed for driver developers.

Some chips use special digital protocols (being not I2C or SPI) with a dedicated timing. It is more or less NOT possible to implement this kind of exact timing within WebIOPi on Python level by e.g. setting a digital I/O port to High and Low in order to realize this kind of protocol ("bitbanging" on Python level). It is even in some cases due to the multitasking nature of Linux also not possible to implement this on C kernel module level. If for some protocol a working kernel module exists, it may make sense to call this driver from within Python using the Python <-> C interface, but an explanation how to do this is out of the scope of this manual. In other cases, the kernel modules expose their functions to special paths of the Linux file system and it is possible to use that from within WebIOPi as long as no special timing constraints occur. Some of the 1-wire drivers use this kind of interfacing. If even this kind of Linux kernel module support does not exist, it is recommended to use a hardware that is able to implement such digital protocols with the required exact timing (the whole Arduino family is the hottest candidate for this) instead and interface from WebIOPi to these. Even a shield for the Raspberry Pi already exists that has such an Arduino-like CPU built in (look for "Gertduino"). Unfortunately, a direct interface from WebIOPi to Arduino does not exist, maybe it may be provided in some future release.

For RTC chips, nothing specific has been added here, they just use the existing `I2C` bus class.

Usage of Inheritance

Multiple inheritance

This mixes the abstract device functionality and the abstract device connectivity together resulting in concrete chip classes that act as final driver implementation for all supported chips. In other words, here the technical chip connectivity (its hardware bus interface) is "married" with the physical or logical function it implements. For chips with more than one functional category this principle is used multiple times (like Pressure AND Temperature for some sensors) resulting in a list of functionality super classes a chip class inherits from.

Some words to multiple inheritance here. It is well known that multiple inheritance may increase the complexity of code and that not all developers love it. However, it is used very slim within WebIOPi and that really helps a lot to keep the drivers source footprint very lean.

Python itself brings some rules for multiple inheritance. If a class inherits from super classes that implement exactly the same method, then the one from the first super class in the list is called so the sequence of the inherited classes has a meaning. Things are also different when Python classes inherit from class `Object`, but this is out of scope here as it is not used within WebIOPi today. On the other hand, the WebIOPi abstract device functionality classes avoid this problem by using methods names that are unique like e.g. `analogRead()` and `digitalRead()`. So if you add new device function abstraction classes please follow this "rule".

For RTC chips, things are straightforward; the device classes inherit from the bus class `I2C` and from the functionality class `Clock`.

Multiple inheritance via instance composition

Sometimes, using "native" multiple inheritance despite all benefits shown above would have too much implications that are not acceptable. Or a class would inherit things that make no sense on lower levels, but "negative" inheritance that deletes functionality, is not really possible. As already mentioned, there is also a philosophic discussion among developers of object-oriented languages that disputes on using multiple inheritance at all or not, but this is another topic that I will not stress here in more detail.

However, to avoid multiple inheritance for any reason, another pattern can be used. In principle, here a class creates objects of other classes within its body and delegates all specific calls to these other objects (instances).

This way of using multiple inheritance by composition is likely to be the method of choice for driver classes to be implemented for shields as they often are a really colorful mixture of many functionalities. The WebIOPi driver for PiFace goes into this direction as it does not inherit for the MCP23S17 class but creates an instance of it.

Device inheritance

Especially in the case of chip families from the same vendor that differ just in things like the number of ports, most of the functionalities are the same and only some small differences exist (e.g. 4 and 8 channels with otherwise identical functions). Then it makes sense to implement the identical functionality within a common super class of those chips and put the specific code into the sub classes. Sometimes it is even possible to parameterize the common code such that the only difference between different device classes is just the parameter set used within the `__init__()` method.

At the bottom line, this helps a lot to further reduce the overall driver code footprint for similar chips. Again, see the MCP23XXX implementation for an example on this. However, do not underestimate the challenge of this work. Sometimes it is intellectual more ambitious to reduce the amount of code instead of just extending it. Plus, writing methods that are configurable by parameters is a bit more burdening that just writing them for one dedicated case.

For Dallas/Maxim RTC chips, the generalized `DSclock` class has been defined as all the chips share many features like most of the time registers addresses and their bit allocations. The final chip classes (e.g. `DS3231`) inherit from `DSclock`. In the case of `DS1338` this is used twice as this class is derived from `DS1307`. Sometimes there are multiple variants possible that are all correct. The `DS1307` could also have been derived from `DS1338` without any problems as the differences between them are so small.

Abstraction framework contracts

They provide some generic introspection-like slots to allow generalized tools like the WebIOPi Device Monitor. Currently these are the methods `__str__()` and `__family__()`. The last helps to populate the Device Monitor. The result of `__family__()` is used to select the device category that will be shown for each device. The result of `__str__()` is used as device class name that will be shown when generating logging and debugging output. For chips that can have multiple addresses and can thus occur more than once within a WebIOPi server, the inclusion of their current bus address within the result is recommended. If a device implements more than one device category, `__family__()` can return a list of family entries (e.g. see class BMP085 as example). This will then lead to multiple category listings for a unique device in the Device Monitor. Consequently, the result of `__family__()` should go in synch to all abstract function classes a device inherits from.

For RTC chips, the class `Clock` implements `__family__()` and returns the string “Clock”. Additionally all chip classes implement their own `__str__()` providing a string with their name. As all Dallas/Maxim RTC chips have fixed I2C addresses the inclusion of the address in their name is unnecessary.

Technical driver implementation

Overview

To provide a driver, two main steps have to be accomplished. The first step is the coding of it in Python language and the second step is to make sure it gets correct integrated into the WebIOPi framework and setup/installation sequence. However, depending on the nature of the new device, not every single action is needed in every case. To sort out a bit the different cases that exist, here is a list of options; they are sorted in the order of increasing complexity:

1. The new driver just extends (sub classes) one of the existing devices. An example for this would be a new temperature chip that can be implemented as sub class of the existing TMPxxx ones.
2. The new driver fits well into the existing device abstraction structure but can't be implemented as direct sub class of an existing chip (or one of the intermediate chip family abstraction classes). An example for this would be if you develop a new driver for a pure digital I/O chip for a new chip vendor.
3. The new driver fits basically well into the existing device abstraction structure but lacks some small additional functionality on the abstraction level. An example for this would be if you develop a new driver for a new kind of sensor like an I2C chip measuring acceleration.
4. The new driver does not fit anywhere into the existing device abstraction structure because it addresses a kind of device category that is not present at the moment. An example for this would be if you develop a new driver for real-time clock chips (what I use as example here).
5. The new driver fits well into the existing device abstraction structure but has functionalities that belong to different abstraction categories. An example for this would be if you develop a new driver for a chip that may have analog I/O ports plus a temperature sensor added. Or, if you develop a dedicated driver for one of the extension shields that contain a whole bunch of chips.

Mixtures of the above cases are also possible but are omitted here to keep the list short enough.

The following content will refer to these different cases with their numbers.

Driver code locations

Python package(s)

If the decisions concerning the hardware abstraction and the device classes (or device class hierarchy) have been made, the first step for the Python coding is to select or define the needed Python sub package folder.

In case 4 you have to create a new Python sub package folder below the devices node. In all other cases this is not needed. The name of this folder has to reflect the kind of hardware abstraction it provides and is written in lowercase letters. As already mentioned, this leads within our RTC example to a new **clock** subpackage folder. Per Python rule, every subpackage folder MUST contain a file called `__init__().py` that is empty by default.

IMPORTANT: If you create new subpackages, some parts of WebIOPi have to be modified to incorporate this addition correct; details follow below in the integration chapter.

__init__().py

In the /devices subfolders, this is the source file that contains all code that makes up the hardware abstraction layer.

In case 4, you have to create it from scratch and put the new abstraction code in there. In most cases, this will be the complete code for your new abstraction class. In the RTC example, this is the new `Clock` class and all its methods.

In case 3, you will have to extend the abstraction code contained in `__init__().py` within existing the package with the things that miss. As this may influence the existing code and all chip drivers that inherit from this abstraction, this is a step that has to be decided, designed, coded and tested very very carefully!

In all other cases, nothing has to be changed here.

driver.py

In case 1, just identify the Python source file that contains the chip class you want to inherit from. Keep the name of the driver file unchanged and add the code to implement the new chip sub class.

In all other cases you have to create a new driver.py file from scratch. Select a suitable name for it that gives some hint about the chip type. If a driver file contains classes for more than one chip you can use (multiple) "x" characters at appropriate positions within the driver file name to indicate that. Add the code to implement your chip classes and put the driver file into the correct subpackage folder.

Especially if you create a new driver file make sure to fulfill all framework and abstraction contracts.

For our RTC example, a new driver file called `dsrtc.py` was created. As the Dallas/Maxim chip names vary their numbers a lot and also other chips exist, that are't RTCs but they also start with DS*, the usage of "x" within the diver file name was not possible so `dsrtc.py` was the best solution for this.

Sometimes chips provide more than one category of functionality (case 5). In this case put the driver file into the subpackage that denotes the MAIN feature set of the chip. You almost can't make anything wrong with this. As long as you fulfill all the abstraction and framework contracts correct and obey the driver integration rules, this will work at the end so don't worry too much about this. If it would be too "wild" for you, choose the way that drivers for shields are developed and put it into the /shields subpackage.

The information here is just basic. You find much more details and hints concerning naming conventions, using inheritance, coding style and concepts plus other details in a following chapter.

IMPORTANT: If you create new driver files, some parts of WebIOPi have to be modified to incorporate this addition correct; details follow again below in the integration chapter.

Coding style and conventions

Overall source file structure

General remarks

The driver's code is separated into different sections. The different sections can occur for every class within a driver. They group methods together that implement a dedicated part of functionality. As inheritance is used throughout the driver's implementation only a subset of code sections may exist for a specific class as all methods belonging to some section may be fully inherited from a super class. Each section is started by a comment like this

```
#---------- Code section ----------
```

to separate the sections also optically in the source file.

Heading comment

On top of the driver code sources a comment section is mandatory. This comment clarifies the copyright and its owner, some release information and also usage instructions.

Copyright and owner may seem a bit nasty and formal to some, but it is very important to state what licensing rules apply to the code below and who the intellectual owner of it is in case someone needs to contact the developer. The preferred source code license for WebIOPI is Apache 2.0 so please use it if you don't need any other license.

You don't need to document what is obvious from the source code. But it is a good behavior to document things that can't be derived directly from code like specific assumptions, things that are (currently) NOT supported and maybe other important restrictions.

The following code snippet shows an example of such a comment. You can use it as template (updating the copyright statement to your name).

```
#   Copyright 2014 Andreas Riegg - t-h-i-n-x.net
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#   Changelog
#
#   1.0    2014-06-10    Initial release.
#
#   Config parameters
#
#   - control         8 bit       Value of the control register
#
#   Usage remarks
#
#   - All chips have a fixed slave address of 0x68
#   - Memory feature of DS1307 and DS1338 and temperature reading from DS3231 are
#     currently not supported
#   - Setting alarms for DS1337 and DS3231 is currently not supported
#
```

Constants and definitions section

This code section contains the constants used by a class. Typical examples for constants are register addresses and default values, but bitmap masks used to handle sub-byte bits are also defined here.

Class initialization section and device instance creation

Here is the code that is used to initialize a new created object (called "instance") of a class. Primarily this is the place where the `__init__()` method of a class is defined. It is common behavior that this private method is automatically called by Python when a new instance is created. Within WebIOPI device classes, this instance creation occurs either automatically when a device class name is specified within the config-file

```
mydevice = DeviceClass
```

or when a device is created manually within a Python server script using a statement like this one:

```
mydevice = DeviceClass()
```

IMPORTANT: The meaning of `"mydevice"` is totally different in these cases although the same string is used often. In the config file case, aside the REST mapping stuff, WebIOPi internally keeps a list of all devices it creates and the string is used as key to access those device instances. In the manual creation case, it is the name of a Python variable within the script that does the creation. Additionally, devices that are created via config file can also be accessed within a server script using the `deviceInstance()` method of WebIOPi like

mydevice = webiopi.deviceInstance("mydevice").

The following code fragment shows a Python server script macro example for RTC chips that allows setting the time of any named (via config file) clock device to the current system time:

```
import webiopi
from datetime import datetime

@webiopi.macro
def tonow(clockDeviceName):
    clockDevice = webiopi.deviceInstance(clockDeviceName)
    if clockDevice != None:
        clockDevice.setDateTime(datetime.now())
```

The code uses **deviceInstance()** in a way that allows submitting an arbitrary clock device name as macro parameter.

Depending on the complexity of a chip, the **__init__()** method may contain a lot of things:

- Calling **__init__()** methods of super classes to initialize them correct also (this is not done by Python automatically and has to be done for EVERY super class in the case of multiple inheritance)
- Setting default values for parameters, doing string-to-value conversions and checking the validity of submitted parameters to this method
- Calling special initialization functions like e.g. waking up devices that are sleeping after powering them on
- Providing methods that allow (proactively) resetting a chip to a defined state

More details about **__init__()** will follow in subsequent sub chapters.

Abstraction framework contracts section

Here the methods are defined to fulfill the abstraction framework contracts. An implementation of **__str__()** is mandatory for device classes that will be used for concrete chips. It makes almost no sense for abstract super classes. An implementation of **__family__()** is mandatory for all basic abstract device category classes (for RTC chips the implementation returns "Clock"). And it is also mandatory if a device provides more than one TouchPoint category (e.g. Temperature AND Pressure), otherwise this will be inherited correct by all concrete sub classes.

If your driver class(es) do not inherit from one of the communication support classes (e.g. I2C) you have to implement a method called **close()** as it will be automatically be invoked when WebIOPi is terminated. By default, you can just do nothing like here:

```
    def close(self):
        return
```

However, this method is the best place to do some driver cleanup actions needed at WebIOPi shutdown. For our RTC example this is only needed for the class **OsClock** (and does nothing there).

XXXXX abstraction related methods section

Here all methods that realize the device category abstraction are defined and all methods are considered to be private. XXXXX gets replaced by the appropriate abstraction name.

For abstract (super) classes, all methods here are “empty” and raise an exception like explained above, a typical code pattern looks like this:

```
def __getXyz__(self):
    raise NotImplementedError
```

For final device classes, the real implementation is provided here.

For the RTC example, XXXXX is replaced by “Clock” and you find all the methods to handle the date and time values here.

Device configuration section

Most device chips allow some kind of configuration with parameters. Setting (and getting) those parameters needs commonly also specific methods as in most cases these parameters are also handled via register access. All methods that have to do with such chip configuration functions are in this section. Regularly, these methods are called during device initialization (using parameters from config file or parameters submitted via instance creation within Python server scripts). In more rare cases they may be also called during runtime (e.g. for setting different timing parameters for a sensor). Configuration methods that should not be called arbitrarily at runtime from external must be marked as private methods to clearly indicate this fact.

Device features section

Some chips allow the execution of a piece of code during runtime that triggers some special function. These functions are implemented as methods in this section. Common examples for such features are sleep and wakeup methods and the activation of some kind of chip calibration action.

The oscillator of some of the RTC chips can be started and stopped. For that, you find some `start()` and `stop()` methods here. Also, some of the oscillators are even in stop mode after power-on. In this case you will see a call of `start()` during `__init__()` so that the clock runs after its device object has been created by WebIOPi.

Local helpers section

All methods that are neither related to any of the framework contracts nor to any features/configuration are seen to be local helpers. They go into this section.

Naming conventions

General naming

The preferred basic naming convention for Python as well as for WebIOPi is CamelCase/camelCase. This applies mainly to variables and methods and partly to classes. Classes start with uppercase characters, methods and variables with lowercase characters.

Class naming

It’s mandatory for classes to start with uppercase characters. The remainder of the class name should be easy to understand but does not have to obey 100% to CamelCase. Best practice for driver class names is just to use the name of the chip (with uppercase letters and numbers) at least for those on the final chip class hierarchy level. It looks like those chip names are somewhat unique for the whole hardware vendor’s offerings so this is a good choice. If on chip level also abstract classes are used sometimes it makes sense to add “X” characters to parts of the chip names that vary.

For the RTC example, this leads to Dallas/Maxim chip classes `DS1307`, `DS1337`, `DS1338` and `DS3231`. Additionally, the class `OsClock` has been implemented that provides a read-only interface to the system clock. This class makes the standard Linux operating system clock available via the REST interface. As Python does not have an (official) interface to update this system clock, such functionality is not available and `set...()` calls will lead to a `NotImplementedError`.

Method naming

It is a common coding convention in Python to mark code elements that are private in some way by adding two underscores ("__") pre and post to the corresponding names. This has no technical implications it's just good developer style. Within WebIOPi drivers, this is used in the following way:

Public means that something is used from outside the WebIOPi device server kernel. This occurs typically in two "flavors":

1. Some device functionality is being mapped and used via the REST API. Then, all methods that are mapped to REST are automatically considered as public.
2. The device is being created and used just within a custom server script. Then, all methods that should be available on Python level within such scripts are also considered as public.

Cases 1 and 2 overlap in the sense that all methods from case 1 are thought to be also in case 2 but some methods that belong to case 2 do not belong to case 1. The reason for this is that not every public device method needs really to be published via REST as this may clutter the REST API or be dangerous due to the communication latencies. In other words, the REST API is typically a subset of the public device Python API as already mentioned earlier. Method names should be good readable and give hints what they do.

Constant naming and values

Constants (static variables) on every indention level use full uppercase spelling. If it makes sense for better readability, add underscores inline to those constants. If possible adhere as much as possible to the naming used in the specs for the chips. This allows better understanding of the relation between source code and chip spec.

To increase readability, some postfix patterns can be used optional for names like

- XXXXX — or
- XXXXX_ADDRESS — for constants the serve as address for something (e.g. chip registers)
- XXXXX_MASK — for bit patterns that serve as mask
- XXXXX_DEFAULT — for constants the denote default values
- XXXXX_FLAG — for constants the serve as indicator flag
- XXXXX_VALUE — for constants the serve as a special value for something
- XXXXX_ — for bits that are active-low

Constants need values. Again, for simplified mapping between driver code and chip spec, use the same number format for both locations.

- Addresses often have 8 bits and use hexadecimal notation ("0x00" to "0xFF")
- Single bit positions used as masks or flags can be specified using binary notation ("0b00100000") or by left-shifting 1 to the appropriate position ("1 << 5")
- Multiple bit positions used as masks should be specified in binary notation to recognize immediately their position ("0b00100110")
- Plain integer values should use simple integer notation
- Plain floating point values should use simple floating point notation

Device classes

For every individual chip type (or other devices) you need exactly one device class within the devices hierarchy of WebIOPi. Class names are bound to Python package names so it would be theoretically possible to have the same device class name in separate device packages. However, it is strongly recommended to avoid this completely as it will lead to problems latest when WebIOPi does its automatic driver class lookup search for config file device definitions. In order to avoid any further misunderstandings, please also make sure to obey this rule for abstract non-final device classes.

As already earlier mentioned, it is not necessary that the concrete chip classes are final leafs of the class hierarchy. For the RTC example you can look at the class **`DS1307`** to see such a constellation.

The most important thing is the correct handling of the calls to the **`__init__()`** methods from the super classes. Every **`__init__()`** has to be called individually but only ONCE. The standard pattern is to call **`__init__()`** of the super class within the **`__init__()`** of the current class. In the case of multiple inheritance, the **`__init__()`** of every super class has to be called. Another important aspect within **`__init__()`** is the device parameter handling which is covered in more detail in the following sub chapter.

Device parameter handling

Basic mechanism for configured parameters

The **`__init__()`** method of the driver classes is the main anchor point to handle the device parameters. All parameters that are within the **`__init__()`** definition (excluding "**`self`**") are considered to be "configurable" device parameters. In order to be able to address them consistent, named method arguments have to be used. Please choose the names meaningful so that it is easy to understand their function.

In the source file (**`driver.py`**), the basic code pattern for this is like this:

```
...
    class DeviceClass:
        def __init__(self, someParameter1, someParameter2, ...):
...
```

Here, 2 different parameters are defined. In principal, the number of parameters is unlimited, but for the sake of simplicity they should be not more than 4 – 5 at maximum.

As already mentioned sometimes a number of inheritance levels will be introduced for similar families of chips. This has also influence on the device parameter handling. Some parameters (notably the more general ones) will be introduced on higher levels of the device class hierarchy. Consequently they have to be received via named parameters within the **`__init__()`** methods of those classes. Other parameters will be only needed for individual chip classes on the lowers class hierarchy levels and these have to be introduced on theirs **`__init__()`** methods.

VERY IMPORTANT: When devices get created (via config file or within Python scripts, see below) only ONE **`__init__()`** method gets invoked directly, it's the one of the used (concrete) device class no matter where it sits in the device class hierarchy. For that reason, EVERY parameter that should be "configurable" for that individual device class has to show up in its **`__init__()`** method even when its used only in super classes or gets "inherited" from them. In order to make it available to the super classes code it has to be "propagated" up via explicit calls to **`__init__()`** of super classes and being resent as parameter of those calls. A very good example of this pattern is the handling of the slave parameter for I2C type chips.

In order to avoid any confusion it is a best practice to add an explaining comment for every "officially" configurable parameter to the header comment of a device driver.

Default values

Every device parameter needs a default value. You could set this value somewhere in the code of `__init__()` but it is better to do this directly in the method definition with the standard Python mechanism to achieve this. In the example below, `someParameter1` gets assigned a default value of `someDefaultvalue1`:

```
...
    class DeviceClass:
        def __init__(self, someParameter1=someDefaultValue1, someParameter2, ...):
...
```

Please choose the default values in a way so that a fresh instance of such a device class can work correct under normal conditions. In this context it is a good practice to look at the chip spec and choose the "software" default values in the same way as the chip gets configured by its "hardware" defaults when just being powered up. This can avoid surprises for chip users that just read the hardware spec.

Also take into account that any other code (maybe via WebIOPi, maybe a kernel module, maybe any other software) may have controlled the chip before and may have left it in some way "modified" or "dirty". If appropriate, set the chips to a fully defined level during `__init__()`.

Parameter value checking

Every device parameter should be checked for correctness. The natural place to do this for configurable parameters is the `__init__()` method. This check consists of 3 logical steps:

- Configurable parameters that are set via config will be delivered as strings to `__init__()`. In most cases, this is not the appropriate type. For numbers use `toint()`, for booleans use `str2bool()` to convert them to Python types that fit.
- Every parameter has a range of allowed values (mainly numbers). Test the converted value against this allowed range. This is also some kind of security measure to avoid problems caused by submitting "rogue" parameters values.
- If the checked value violates the allowed range, raise a `ValueError` exception. In the text message of the exception, name the parameter that is wrong, note the received value and give advice on the expected range.

"Standard" parameters

Some device parameters are being introduced through the device connectivity and control abstraction super classes and get inherited automatically:

- For I2C, this is **slave** being the 7 bit I2C address (e.g. 0x68)
- For SPI, this is **chip** being the selection of the SPI_ CE0 or SPI_CE1 signal (e.g. 0)
- For Serial, this is **device** being the Linux serial device "name" (e.g. "/dev/ttyAMA0")and **baudrate** being the serial baud rate (e.g. 9600)

IMPORTANT: Other parameters for SPI (e.g. mode) and via inheritance from class Bus (e.g. busName) exist, but they are currently not intended for public use even for driver developers, so please ignore and leave them untouched for now.

Usage via config file

All named parameters for a device can be set from the WebIOPi config file by adding statements like this to it:

```
...

[DEVICES]
...
myDevice = DeviceClass someParameter1:value1 someParameter2:value2
...
```

The name of the parameter in the config statement (e.g. `someParameter1`) has to match exactly (case-sensitive) the name of the parameter in the `__init__()` method of the device class. No blanks are allowed around the colon. WebIOPi provides some convenience methods to handle numerical or logical parameter values:

- For integer values this means that you can use three possible textual ways to provide an integer like “100” being a plain integer, “0x68” being a hexadecimal value and “0b010101” being a binary value.
- For boolean values this means that you can use either “1”, “true”, “True”, “yes”, or “Yes” to provide a Python `True` value. All other strings will be considered as Python `False`, even something like “nonsense”.

For details, see module `types.py` in package `utils`.

IMPORTANT: In order to provide this feature for any device, you have to invoke the methods `toint()` or `str2bool()` on every parameter of your device that can be set via config options within `__init__()`.

Usage in Python server scripts

Device instances can also be created via Python server scripts in the standard way to create Python objects using a class constructor like here:

```
...
myDevice = DeviceClass(someParameter1=value1, someParameter2=value2)
...
```

Like standard Python, you can choose to provide values for the named device parameters you need. All the other values will be set to their default values (if you provide them correct as highly recommended).

It is also possible to create devices by just providing the values:

```
...
myDevice = DeviceClass(value1, value2)
...
```

In this case the parameters will be set according to their sequence of occurrence in the `__init__()` method. In principle, this works also, however, you can avoid unexpected behavior by using only named parameters when calling the constructor. Problems will happen when the number and/or sequence of the parameters may change with updates of the device drivers.

IMPORTANT: Keep in mind, if you create device instances in this way they will not get a REST mapping. They will be only available locally within the server script (or global on WebIOPI Python server level if defined as global variables).

Non-configured parameters

Devices can have parameters that can be changed but this should not happen via config file or class constructors for any reason. In this case, such parameters should be left out of the `__init__()` method. For them, public device methods can be defined that allow this change. These methods should be placed into the device configuration section (see above) of a driver. Currently it is unusual to provide a REST mapping for them. This may change in future.

RTC example

The following code fragment from the class **DSclock** of our RTC example shows some of the above concepts:

```
...
class DSclock(I2C, Clock):
    def __init__(self, control):
        I2C.__init__(self, 0x68)
        Clock.__init__(self)
        if control != None:
            con = toint(control)
            if not con in range(0, 0xFF + 1):
                raise ValueError("control value [%d] out of range [%d..%d]" % (con, 0x00,
0xFF))
            self.__setCon__(con)
        else:
            self.__setCon__(self.CON_DEFAULT)
...
```

- The **__init__()** method introduces the configurable parameter **control**. Its purpose is to allow setting the value of a dedicated control register that is implemented in all supported Dallas/Maxim RTC chips. It's allowed value is 8 bits unsigned (very common for chip registers) which means it is considered as a positive numeric value.
- The value of **control** is checked first if it is not equal to **None**. If yes, it gets converted from string to an integer.
- Then, the integer value is checked to be in the allowed bounds. 8 bit numbers range from 0 ... 255 or 0x00 ... 0xFF (keep in mind, the Python range statement does not include the upper bound value)
- If the value violates the range, a **ValueException** is thrown including a text message that states the name of the value, the submitted value and the allowed bounds.
- It the value adheres the range, the private method **__setCon__()** to set the appropriate register with the new value for control is invoked.
- Finally, if the value for **control** is **None**, it is set to its default value **CON_DEFAULT** (which is provided as constant class variable).

The code above shows abstract code. The device **DSclock** gets never created. The code below shows a concrete device class (**DS3231**) that can be used as real device:

```
...
class DS3231(DSclock):

    CON         = 0x0E       # Control register address
    CON_MASK    = 0b11011111 # Control register mask
    CON_DEFAULT = 0b00011100 # Control register default value


#---------- Class initialization ----------

    def __init__(self, control=None):
        DSclock.__init__(self, control)

...
```

- First, some class variables are defined. This allows having a different value for **CON_DEFAULT** for each concrete chip. This is important as in fact every DSxxxx RTC chip has a control register, but the bits and also the default values are different. Same applies to the register address itself being **CON** and also the allocated bits being captured by the mask **CON_MASK**.

- Then, the `__init__()` method again carries the control parameter. This allows to really setting it. If unset, it is explicit set to `None` which will trigger the usage of the default value in the abstract super class.
- Finally, the `__init__()` method of the super class is invoked and the received value for control is propagated to the super class `__init__()` method where it gets further handled as already explained.

This kind of mechanism can (and has to) be used repeatedly for every device class hierarchy level, see class `DS1338` for an example.

Special coding patterns

Bit manipulation

In many cases chip features are implemented using (8 bit = 1 Byte) registers. If just the whole register is read or written, this is quite simple. However, the situation gets more complicated if just one bit or a set of bits has to be read or updated. We can find situations where

- some bit(s) are don't care or just undefined, but they must get a defined state especially when read
- some bit(s) have to be always 1 or 0, especially when written
- just some bit(s) have to be read, but reading is only possible for the whole register
- just some bit(s) have to be updated and all other one(s) have to keep their current value(s), but writing is only possible to the whole register

The following example shows this as a schematic pattern using the most generic case possible. 5 out of 8 bits have to be kept as they are, 3 out of 8 bits have to be updated to a specific value and the byte that is received to update tries to change more bits than allowed. These abbreviations apply:

- **V** marks a bit value/position that has to keep its current value (-> remain unchanged)
- **C** marks a bit value/position that has to change its current value (-> is to be updated)
- **U** marks a bit value/position that has been received intended for update (-> would be updated, but partly violates positions to be unchanged)
- **X** marks a bit value/position that just does not care
- **0, 1** marks a bit value/position that acts as or results from masking patterns
- The values of each **V**, **C**, **U** and **X** are boolean types and maybe 1 or 0, but their actual value **MUST** be without any influence for selecting the algorithm

So, we have this:

```
currentValue        VVCCVVCV
updateBits          XXUUXXUU        <- bit 0 is "illegal"
intendedResult      VVUUVVUV
```

The generic algorithm to achieve this is:

0. Read the `currentValue` from the register.
1. Define a change mask. This mask contains 1's on bit positions that can be changed and 0's on bit positions that have to keep their current value:

```
changeMask          00110010
```

2. Calculate the bit pattern that has to be kept with the operation `currentValue & ~changeMask (~ = not):`

```
currentValue        VVCCVVCV
~changeMask         11001101        <- &

toKeep              VV00VV0V
```

3. Calculate the bit pattern that contains the needed and allowed updates with the operation `updateBits & changeMask`:

```
updateBits          XXUUXXUU
changeMask          00110010        <- &

maskedUpdate        00UU00U0
```

4. Finally, calculate the new value for the register with the operation `toKeep | maskedUpdate`:

```
toKeep              VV00VV0V
maskedUpdate        00UU00U0        <- |

newValue            VVUUVVUV
```

5. This way, `newValue` is equal to the `intendedResult` . Finally, write `newValue` to the register.

If you put this into one formula, this results in:

`newValue = [(currentValue & ~changeMask) | (updateBits & changeMask)]`

This generic formula works in every case. However, some simplifications exist:

- If `updateBits` has 0's for sure on bit positions that have to be kept unchanged (e.g. because you use constant values for updates that ensure this), then the formula reduces to

  `newValue = [(currentValue & ~changeMask) | updateBits]`

  as the second &-operation does not change any bit in this case.
- If just one bit position is to be changed to a fixed 1 or 0 value (e.g. set a single GPIO port output to 1 or 0), this can be also achieved by an if-else branch and two simplified formulas:

```
def updateValue(self, changePosition, changeValue):
    changeMask = 1 << changePosition
        currentValue = self.__readValue__()
        if changeValue:
            currentValue |= changeMask
        else:
            currentValue &= ~changeMask
    self.__writeValue__(currentValue)
```

The method **`__setSec__()`** of class **`DS1307`** is an example for the generic formula. Updating the seconds register has to keep the value of the **`CH`** bit (bit 7) unchanged.

The **`start()`** and **`stop()`** methods within **`dsrtc.py`** are an example for the second simplified method as they toggle just one bit to a defined value. They have not an if-else branch but the two variants are separated to two different methods setting the bit to 0 in one method and to 1 in the corresponding other one.

Performance optimization

Driver code is code that can influence server performance a lot, especially if it is called very frequent. For that reason some care must be taken to ensure good overall performance. Plain Python code runs typically very fast if written by skilled persons so it will be in most cases not the reason for bottlenecks.

However, driver code accesses hardware components and here some pitfalls are waiting. Every chip has to be connected with some of the hardware interfaces. See the chapters on hardware interface and control abstraction. While some of these interfaces are very fast (e.g. SPI) and have no big impact on performance even when called very frequent, some others are slow (e.g. serial or low speed I2C) or very slow (1-wire) and they need special attendance.

The best possibility to improve performance for slow devices is to reduce the number of interface calls to a minimum. While some calls need to touch the real chip (exactly its TouchPoints) in every case (e.g. reading the sensed state of a GPIO input port), some others do not (e.g. reading a GPIO port if it is currently configured as input or output). As long as ONLY WebIOPi and a unique device instance are used to change such states, it is possible to cache such information in the driver code (e.g. some drivers have a FUNCTIONS list for digital I/O ports). This can reduce the number of hardware interface calls a lot and is especially helpful in cases where a lot of state (e.g. for wildcard * REST calls) is needed.

IMPORTANT: If the "unique instance per every chip" rule is violated or other programs outside WebIOPi simultaneously access the chips, this will lead to wrong readings. However, if such things occur you will run into trouble sooner or later anyway, so this should be obeyed in any situation. For this reason care is also needed when initializing chips. Some other code (even an abnormally terminated WebIOPi process) may have left the chips in an undefined or non-default state. Best rule is to set every chip to a defined state when initializing its instance in the driver code.

Another possibility to improve performance is multi-register access which is possible for many chips. Let's consider the following situation. Some registers have to be read; maybe we have 4 of them. Standard solution would be to submit 4 single register read calls. While this works stable in every case there is a shorter solution. Many chips have the feature that they increase the read address pointer automatically after every read and answer the value of the next register in the next read cycle. This means that you can read multiple registers that are in sequential order with one read command at once; here it would be one read call that gets 4 registers at once. This is (much) faster than repeated single read calls with manually increased register addresses. The same mechanism applies to write calls.

In our RTC example, this can be observed for the method `__getDateTime__()`. The default implementation on clock package level just reads every value with a separate call (`__getYrs__()`, `getMon__()`, ...) and puts the results together into one datetime object. Without any optimization, this would result in 6 single I2C calls. The optimized version within the `DSclock` class implements this with one I2C call that reads 7 registers at once (throwing the dow value away). The same is true for `__setDateTime__()` with the downside that the current value for dow has to be read before which reduces the speed improvement effect a bit. But this can't be avoided as the dow register is in the middle of the sequential registers space. Well, two 3 register writes could also be used, but the difference would be minimal.

Sleeping unnecessary long in driver code is also a potential bottleneck for performance. In some cases, operations within in chip take some time (e.g. something needs to be sampled and converted before the result is ready). If this process takes always the same time, no optimization is possible. However, in many situations the time needed differs and some status bit to indicate the end of conversion is available from the chip. In such a case it is better to loop over the status indicator bit (but only for a limited time of e.g. max. 100 ms) and leave the loop as soon as the conversion is complete.

Within plain Python code additional optimization steps are possible but they are not highest priority. An example for this is something what is called “make most used cases fast”. If you have an algorithm that may take some computing time it could be possible to have different implementations for different complexity levels. If you put if-elif-elif-else branches (maybe even hierarchically cascaded) around this it will be possible to check for the simplest cases and then execute them with simplified algorithms. Especially if the most complex case occurs only very seldom this can increase performance significantly. In the RTC example, the implementation of the conversion method `BcdBits2Int()` shows an example for this.

Logging handling

WebIOPi comprises a basic logging facility. This logging uses a 1:1 delegation to standard Python logging. It writes logging messages to a log file (located at `/var/log/webiopi` by default). You can modify the location of this log file by submitting/changing the –l parameter of the WebIOPi start command:

```
$ sudo webiopi [-h] [-c config] [-l log] [-s script] [-d] [port]
```

Within the code of a driver, logging messages with “info” level can be generated by using:

```
...
from webiopi.utils.logger import info
...
    info("Some logging info text")
...
```

As these logging messages are permanently written to the log file, please use them very sparsely. In most cases, drivers do not need to write info level logging messages. If you need to be able to follow some activities of your driver, use the debugging facility instead.

Test and debugging handling

WebIOPi also comprises a basic debugging facility. This debugging uses a 1:1 delegation to standard Python debug logging. It writes debugging messages to the console and/or log file. To use this in a console, you must:

1. Run WebIOPi in foreground mode from a console. If it is run as daemon, you will not be able to see the debugging messages (they will be written to the log file).
   IMPORTANT: If you setup WebIOPi to automatically start as daemon at boot time, you have to stop this daemon first or disable the automatic daemon boot time startup. If you forget this, you may see messages indicating that IP addresses are already in use which indicates that another WebIOPI process is already running.

2. Enable the debugging output by setting the –d option of the WebIOPi start command:

   ```
   $ sudo webiopi [-h] [-c config] [-l log] [-s script] [-d] [port]
   ```

3. Add the following code snippet to your driver code at appropriate locations:

   ```
   ...
   from webiopi.utils.logger import debug
   ...
       debug("MyDriver: Some debug info text")
   ...
   ```

In order to get some hint where a debugging message results from, it is a good practice to add some small piece of location hint to the debugging message (“MyDriver:” in the example above).

Warning/error/exception handling

In the same way like info and debug messages, you can also create warning, error and exception messages. See **`logger.py`** for details and syntax (which is very similar to the debugging message above). However, this feature is not often used within drivers for now, so you may just ignore it at the moment.

Sometimes it makes good sense to check the parameter values submitted to drivers in order to raise the robustness of the drivers and also to simplify finding errors that result from driver parameters. If you look at the code snippet from the time abstraction layer

```
...
    def checkDow(self, dow):
        if not dow in range(1,8):
            raise ValueError("dow [%d] out of range [%d..%d]" % (dow, 1, 7))
...
```

you see that the if-statement checks whether the value of parameter **`dow`** is within the allowed range of 1 to 7. If not it raises a **`ValueError`** exception with a text that explains the error. Please create meaningful error messages as good hints will allow faster bug fixing for WebIOPi users.

As a rule of robustness, every parameter submitted to drivers or via public and/or REST calls has to be checked in this way. You may do this inline or refactor it to separate methods like in the example above. For minimizing code footprint, the parameters should be checked on the driver class hierarchy level where they are introduced first to make sure that sub classes do not have to implement the checks again.

Typically, exceptions like the one above within drivers will not stop the WebIOPi server completely. In most cases, you will see a message on the console (if run in foreground mode), you will see a log entry in the log file (if run in background mode) and you will receive an HTTP/CoAP error response.

Technical driver integration

Driver framework integration

Update __init__() on package level

If you create a new driver source file within an existing device sub package containing (a) new device class(es) (case 2, 3, 4, 5) you have to update the DRIVERS dictionary within `__init__()` on package level. This is necessary so that the automatic device class lookup for config file device entries will be able to locate your new devices.

The structure of the DRIVERS dictionary is as follows: It contains list entries that have the names of the driver source files omitting ".py" as keys. We are on UNIX, so please take care of correct upper/lower case naming. The items of the list entries are string lists (resulting overall in a keyed matrix) that represent the class names of the device classes, again this is case-sensitive. Each device class that is intended to be used as device entry in the config file MUST be named here. Each device class MUST have the correct key in the sense that it must be exactly in the list that belongs to the Python source file in which this class is defined. Abstract device classes should not show up here and make sure that the keys AND the class name strings do not have any duplicates here or somewhere else within WebIOPi.

If you not only create (a) new source file(s) but also a new sub package (case 4) and thus also a new `__init__.py` for this package, you have to do the same as above. The only difference is that you now have to create a complete new DRIVERS dictionary preferably at the end of `__init__.py` like the existing ones.

```
...
DRIVERS = {}
DRIVERS["dsrtc"] = ["DS1307", "DS1337", "DS1338", "DS3231"]
DRIVERS["mcprtc"] = ["MCP7490"]
DRIVERS["osrtc"] = ["OsClock"]
...
```

The sequence of the class name string entries does not matter. Even if you have a driver source file just containing one new device class you MUST create a dictionary entry for that like for the MCP7490 chip above.

NOTE: The code for the MCP7490 chip is not included in this guide to keep it short. But it is contained in the whole RTC drivers package.

Update manager.py

If you create a new sub package directory below the /devices folder (most notably only in case 4), you have to add exactly the name of this folder to `"manager.py"` so that the automatic device class lookup for config file device entries still works. For our RTC example, this means extending the corresponding import statement with the entry for "clock"

```
...
from webiopi.devices import serial, digital, analog, sensor, shield, clock
...
```

and adding the new import also to the PACKAGES list

```
...
PACKAGES = [serial, digital, analog, sensor, shield, clock]
...
```

As both statements just represent lists, the position within the list does not matter.

JScript and Device Monitor integration

Update webiopi.js

If you create a new sub package directory below the /device folder (most notably only in case 4), you have to extend webiopi.js with code to support the new device category. This is necessary so that the Device Monitor shows your devices and that others can build customized web pages for their project.

Updating of webiopi.js comprises 5 logical steps for each device category. In the case of our RTC example, this means to add the support for the new clock device category. You find the code that shows the modifications of webiopi.js to do steps 1 – 4 in the appendices. Step 5 is not provided for clock devices so far.

These steps are:

1. Add the device category to the list of devices known by JScript. This is done by adding the device category as named in the `__family__()` contract to the method `WebIOPi.prototype.newDevice`. Just copy the existing code and add a new if-branch to this method. For RTC, a branch for `"Clock"` has been added.
2. Provide the code that allows creating the new device and setting some properties. This is done by adding a function to JScript with the name of the device category. In case of RTC this is the function `Clock`.
3. Provide the code to show some very basic generic UI (at least value reading) entry in the Device Monitor. To achieve this, the new JScript prototype object must define the methods `toString` and `refreshUI`. Within `refreshUI`, the HTML code to populate the UI elements has to be provided plus any method call that retrieves the value to show from the device via REST.
4. Provide more JScript methods that mirror the REST mappings of your device into a JScript element. This is done by creating JScript methods that do a HTTP request for the mappings and put this into a JScript method that is named closely like the corresponding public Python methods. For our RTC example, these are the methods like `Clock.prototype.getDateTime` etc.
5. Provide a `refreshUI` method that allows higher generic interaction with your device (mainly also updating values for TouchPoints that can be updated).

Steps 1 – 3 are mandatory; steps 4 and 5 are optional.

Device Monitor population

If webiopi.js is updated at least with steps 1 – 3 from above, then the Device Monitor will be automatically updated with generic entries for your new devices, no additional manual steps are involved. Even more, the result of the `__family__()` contract rules how much generic UI elements will be shown for each device as already mentioned.

Update WebIOPI doc-root

The webiopi.js file that will be used at runtime is the one that is found in the doc-root folder of WebIOPi. You can either set this folder direct by creating a corresponding entry in the config file or by relying on the default settings of WebIOPi after setup. Take care; if you update webiopi.js in the htdocs folder that's created during un-taring WebIOPI installation, you have to call "setup.sh" after every change to update the default htdocs location (see next chapter for details). If you "code-run-debug-change" your new version of webiopi.js intensively I recommend to use the config file option.

WebIOPi setup/installation integration

Update setup.py

If you create a new sub package directory below the /device folder (most notably only in case 4), you have to add this also to `setup.py` in order that it gets pre-compiled and added to the Python packages correct. This means extending the corresponding packages list like this:

```
...
      packages          = ["webiopi",
                           "webiopi.utils",
                           "webiopi.clients",
...
                           "webiopi.devices.clock",
                           "webiopi.devices.shield"
                           ],
...
```

Executing setup

The simplest way to activate all changes is just to go to the WebIOPi root directory that got created when un-taring the setup archive and just execute “setup” again. However, this repeats many steps that are unnecessary when just adding or modifying some Python code. Much leaner is the execution of setup without updating all Raspbian distribution libraries. This can be achieved by executing setup in a reduced fashion as

```
”sudo ./setup.sh skip-apt”
```

This still does many things that are somehow redundant, but as it does not update the Raspbian repositories this command finishes typically within 2-3 seconds which is just ok for the most developers. It also compiles the native C interfaces of WebIOPi when run for the first time but does not repeat this when running again. Plus, it takes automatically take care of Python 2 / 3 setup for WebIOPi. So, for the most developers, this would be the standard option. In this case, also the default doc-root folder gets updated which is important when you change webiopi.js.

This is not the end of the story. You can reduce the time for integrating Python changes further by doing just the Python setup. This can be done by going to the Python root directory of WebIOPi (which is below the basic WebIOPi root directory) and execute

```
”sudo python setup.py install”
```

However, this is now specific to the Python(s) version used and does only incorporate Python changes, nothing else. The above is valid for Python 2, in case of Python 3 this command would have to be

```
”sudo python3 setup.py install”
```

WebIOPi has a mechanism to look for the Python version(s) available on your Pi and adjusting its start command (“webiopi”) to the correct one. If both versions are available it defaults to version 3. If you know what Python version is used by your WebIOPi configuration you can select the correct setup from above and speed up your code-compile-test-fix sequence by doing this just for one Python version and avoiding all the extra stuff of setup.sh . However, be warned! Currently, WebIOPi is meant to be fully compatible with Python 2 AND 3. So if you develop a driver that you want to release to the public please make sure the driver is really tested to be fully compatible with both versions of Python!

Device communication and control technology

Bus class

The basic class for device communication in WebIOPi is the class Bus. This class manages the Linux file device system access for a set of chip connection protocols. For basic byte wise communication it provides the methods `readByte()`, `readBytes()`, `writeByte()` and `writeBytes()`. Please keep in mind that all byte values that are read or written have to be maximum 8 bits.

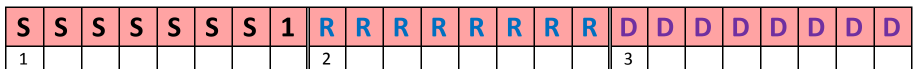
I2C communication

WebIOPi has all necessary services to automatically set up I2C communication; it even loads the needed kernel modules if they are not loaded before. The I2C bus class `I2C` does the selection of the correct I2C bus and provides some convenience methods for communication. Almost all I2C chip communication goes via chip registers. Normally, you would have to specify the register address first and then the "payload" bytes for read and/or write. The helper methods `readRegister()`, `readRegisters()`, `writeRegister()` and `writeRegisters()` make this simpler by giving the possibility to have the register address as parameter and do the byte plumbing for you. The methods with the "s" at the end allow subsequent reading/writing of many bytes and are suited for the auto increment or sequential read register feature of many I2C chips. So it is recommended to use these helper methods when communicating with I2C devices instead of using the very basic `readBytes()` and `writeBytes()` of class `Bus`.

Let's look at some examples from the class DS1307 within our RTC chips drivers. For this, the following rules apply: Each cell contains a single bit. Vertical double lines denote byte boundaries. **Red** cell bits are sent from the master to the slave (master is "writing"), **green** cell bits are sent back from the slave to the master (master is "reading"). Bits marked as **"S"** denote the I2C 7-bit slave address bits, bits marked as **"R"** denote the chip register address and bits marked as **"D"** denote just data bits. Bits with **"0"** or **"1"** denote a fixed value. For simplicity the I2C protocol start, acknowledge and stop bits are not shown. As these are automatically managed by the underlying I2C kernel drivers this removes no important information.

writeRegister()

The control register's value of the DS1307 chip can be set by writing the new value to the register at address 0x07. The chip specification describes that like this:



S	S	S	S	S	S	S	1	R	R	R	R	R	R	R	R	D	D	D	D	D	D	D	D
1								2								3							

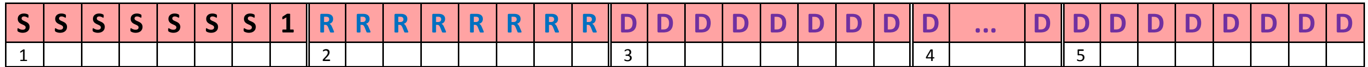
Three bytes are sent to the slave. The first byte contains the seven slave address bits plus the RW-bit set to 1 for write. The second byte contains the chip register address to which the write operation goes to. The last byte contains the byte that will be written in the register. In Python code, this looks like this (the code parts that handle bit masking are omitted):

```
...
    CON          = 0x07        # Control register address
...
    def __setCon__(self, value):
        self.writeRegister(self.CON, value)
...
```

The first parameter of `writeRegister()` is the address of the chip register and the second parameter is the data byte, that has to be written to it.

writeRegisters()

All time register values (seconds, minutes, hours) of the DS1307 chip can be set by writing the new values to the registers at address 0x00, 0x01 and 0x02. This operation can be done in one communication sequence as the register pointer, to which a given data byte is written to, is automatically incremented by 1 for each byte being transmitted. The chip specification describes that like this:



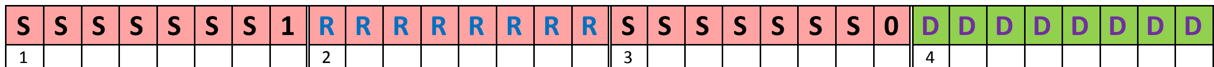
Five bytes are sent to the slave. The first byte contains the seven slave address bits plus the RW-bit set to 1 for write. The second byte contains the chip register address to which the write operation goes to. The third, fourth and last bytes contain the bytes that will be written into the registers. In Python code, this looks like this (the code parts that handle bit masking are omitted):

```
...
    SEC   = 0x00   # Seconds register coded as 2-digit BCD
...
    def __setTime__(self, aTime):
        data = bytearray(3)
        data[0] = self.Int2BcdBits(aTime.second)
        data[1] = self.Int2BcdBits(aTime.minute)
        data[2] = self.Int2BcdBits(aTime.hour)
        self.writeRegisters(self.SEC, data)
...
```

The first parameter of **`writeRegisters()`** is the address of the first chip register and the second parameter is the data bytearray, that has to be written to the subsequent registers.

readRegister()

The second's value of the RTC chip can be read from the register at address 0x00. Just submitting an I2C read operation means in the most cases that the value of the current addressed chip register will be answered. The current register address will be the value that has been set after the last read. As this can be any address this will be not appropriate in practice. To circumvent this, it is a common pattern to do an empty write to a chip register first (setting this way the register read pointer to a defined value) and then starting the read command. The chip specification describes that like this:



Three bytes are sent to the slave and one byte is received back from the slave. The first byte contains the seven slave address bits plus the RW-bit set to 1 for write. The second byte contains the chip register address to which the write operation goes to. The third byte contains the seven slave address bits plus the RW-bit set to 0 for read. The last byte contains the byte that was read from the register. In Python code, this looks like this (the code parts that handle bit masking are omitted):

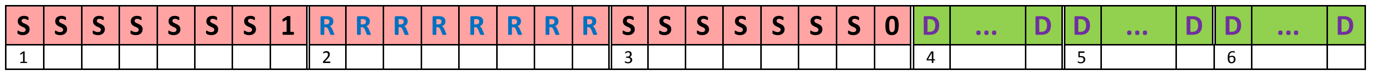
```
...
    SEC   = 0x00   # Seconds register coded as 2-digit BCD
...
    def __getSec__(self):
        data = self.readRegister(self.SEC)
        return self.BcdBits2Int(data)
...
```

The only parameter of **`readRegister()`** is the address of the chip register that has to be read from.

readRegisters()

All date register values (days, months, years) of the DS1307 chip can be retrieved by reading the values of the registers at address 0x04, 0x05 and 0x06. This operation can be done in one communication sequence as the

register pointer, from which a given data byte is read, is automatically incremented by 1 for each byte being retrieved. Here, the same problem with the current register pointer as for readRegister() applies so the same pattern to circumvent this is used. The chip specification describes that like this:



Three bytes are sent to the slave and three bytes are received back from the slave. The first byte contains the seven slave address bits plus the RW-bit set to 1 for write. The second byte contains the chip register address to which the write operation goes to. The third byte contains the seven slave address bits plus the RW-bit set to 0 for read. The fourth, fifth and the last byte contain the bytes that were read from the registers. In Python code, this looks like this (the code parts that handle bit masking are omitted):

```
...
    DAY   = 0x00   # Day of month register coded as 2-digit BCD
...
    def __getDate__(self):
        data = self.readRegisters(self.DAY, 3)
        day     = self.BcdBits2Int(data[0])
        month   = self.BcdBits2Int(data[1])
        year    = self.BcdBits2Int(data[2]) + 2000
        return date(year, month, day)
...
```

The first parameter of **readRegisters()** is the address of the first chip register and the second parameter is the number of data bytes, that have to be read from the subsequent registers.

NOTE: The main purpose of sequential operations is performance optimization as they utilize much less I2C cycles when multiple register values are needed to manipulate.

IMPORTANT: The repeated read and write methods work only for registers which are really subsequent or when the register address pointer is incremented automatically in a way so that the appropriate register positions are addressed. It is not possible to omit some registers somewhere in between the sequence. Sometimes, the automatic increment mode has to be activated explicitly by setting some chip configuration bit. For the RTC chips used in the examples here this is not necessary as they provide this feature by default.

REPETITION: The I2C slave address you specify is the 7 bit form of the address. While most chip spec documents also use this 7 bit format some do not and this may cause confusion. If you are in doubt what address a chip uses, connect it to the I2C as the only device and run the I2C command tool **"sudo i2cdetect -y 1"** that will produce the following output:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- 6f
70: -- -- -- -- -- -- -- --
```

This example shows the output if two RTC chips (one from Dallas/Maxim on 0x68, another from Microchip on 0x6F) are present on the I2C bus

NOTE: You will see this output even when WebIOPi is not started and no driver for those chips is available from it.

NOTE 2: The parameter of the detection command at the very end is the number of the I2C bus. For rev. 1 boards, it has to be 0.

SPI communication

WebIOPi has all necessary services to automatically set up SPI communication; it even loads the needed kernel modules if they are not loaded before. The SPI bus class `SPI` does the selection of the correct SPI_CE chip selection signal and provides a special "reading" method that comes as a combined write/read communication sequence called `xfer()`. This method plays the same role as the methods `readRegister(s)()` for I2C. `xfer()` has the special restriction that it uses as input parameter an array of bytes and that the size of the response array (also of bytes) has exactly the same size as the input array. So if you want to read 4 bytes but only have an input of 2 bytes, you must extend the input array with two additional don't care bytes (e.g. 0x00) at the end so that both sizes match.

The writing operations can simply be done using the inherited methods `writeByte()` **and** `writeBytes()` of class `Bus`.

HINT: Some SPI chips offer an additional addressing feature (e.g. MCP23Sxx) but this has nothing to do with the `chip` parameter of the class `SPI`.

As the RTC chips do not implement the SPI protocol no example can be provided here. You may look at the class MCP3X0X in the package .../devices/analog to see an example.

Serial communication

WebIOPi has all necessary services to do serial communication. By standard, it uses the RX/TX pins of the board which is the logical device `"/dev/ttyAMA0"` on Linux level. As these pins are normally used by a Linux console you have to free them from that. Please look at the WebIOPi wiki how to do that. The serial bus class `Serial` does the selection of the correct serial device and allows the setting of the baud rate. All other features of the serial connection are hard wired into this class, you may change them but at your own risk. The device parameter `device` is the logical name of the used serial device on Linux level. If you try to use other serial devices (e.g. via USB) you can get this working by using the Linux name for those alternative serial devices as value for the `device` parameter. The baudrate can be changed with the `baudrate` device parameter. The WebIOPi wiki has a very detailed documentation and example about serial communication.

1-wire communication

Kernel module integration

WebIOPi does not have its own full device drivers for 1-wire masters and slaves (at the moment, but maybe it will never have). It relies on the availability of Linux kernel modules for the chips and hooks onto them. As a consequence, if there are no kernel modules available for any slave you want to use, there is no way to get this 1-wire slave controlled within WebIOPi. WebIOPi also relies on using GPIO 4 and the kernel module to use this pin as bitbanged 1-wire master (`"w1-gpio"`).

The 1-wire bus class `OneWire` does the job of loading the necessary slave kernel modules and provides some lookup for recognized slaves. All the rest of the slave kernel module communication is done within the individual slave classes. They use a special mechanism that gets provided by the slave kernel modules. If loading of the modules was successful, you can look into the system directory `"/sys/bus/w1/devices"` to see all detected slaves and their slave addresses in the form of subdirectories (one per slave) named `"/sys/bus/w1/devices/ff-ssssssssssss"` (ff = family code, ssssssssssss = unique 1-wire slave address).

Within these slave subdirectories you will notice a couple of "files". However, these are not real files but some kind of logical paths that are mirrored by the slave modules into the Linux file system and can be read and/or written to by standard file commands (e.g. via Python) and that's exactly what the WebIOPi 1-wire slave device classes do. To some extent the kernel modules mirror the slave chip registers into the file system, but also other things. However, this is not really good documented. If you want to write a WebIOPi 1-wire slave driver class, you have to look at the C source code of the kernel modules to find out what and how the chip registers and functions get mirrored and maybe how individual bits are allocated. You need good C skills to get this done.

1-wire master DS2482-x integration

The solution with GPIO 4 used as 1-wire master does work. However, it is not very reliable, can cause problems with parasite powered slaves and is definitely NOT suited for 1-wire connections that use long wires and have many slave chips on the bus. It is strongly recommended to use hardware 1-wire masters in these cases and this will avoid a lot of error detection hassle.

One of the most popular chips for 1-wire masters is the DS2482-100 and -800. WebIOPi does not have a Python driver for it, but you can integrate it via its kernel module like the other 1-wire devices. To get this working you have to load the DS2482 kernel module BEFORE WebIOPi is being started. The shell commands for loading are these:

```
sudo echo ds2482 0x18 | sudo tee -a /sys/bus/i2c/devices/i2c-1/new_device
sudo modprobe wire
sudo modprobe ds2482
```

You need to have the Raspberry Pi I2C kernel modules enabled first like for all other WebIOPi I2C devices, nothing special here.

IMPORTANT: If you rely on the automatic loading of the I2C kernel modules by WebIOPi you have to do this manually as the I2C kernel modules have to be running BEFORE WebIOPi is started to get the shell commands above to work! The first line creates a new I2C device for the DS2482 on I2C slave address `0x18`. If your chip uses another address, change the value to yours. The command assumes your chip is connected to I2C bus 1 (which is standard since rev. 2 boards). For old rev. 1 boards, change `i2c-1` to `i2c-0`. It does not matter if you have the 1-channel (-100) or 8 channel (-800) version, the command is just the same.

The next two commands load the basic module driver for 1-wire devices (`"wire"`) and then the kernel module for the DS2482 chip (`"ds2482"`). If you have connected your 1-wire slaves already to the master channel(s), you can look again into the system directory `"/sys/bus/w1/devices"` to see all detected slaves and their slave addresses. If you don't see any slave directories there, something went wrong and you will not be able to control them from WebIOPi. Keep in mind, the kernel modules for your slaves will be loaded automatically by WebIOPi, but the master has to successful detect the slaves before this will work. So take the time and do the check above at least once after you did a new wiring of your slaves.

You can submit the commands above manually from a shell, but this is not desired if you want to run a server unattended and get all done automatically at boot time. To get this setup, you can

- put the commands above into a shell script (e.g. `ds2482.sh`) located in your home directory (e.g. `/home/pi`)
- make the file executable (`chmod 755`) and
- add the following line to the END of your `/etc/rc.local` file: `"sudo bash /home/pi/ds2482.sh"`

From now on the kernel modules will be loaded upon each reboot and WebIOPi will be able to control them. You may be able to achieve this also by integrating the module loading into the standard list of modules Linux loads at startup but I did not test this.

HINT: The slave address for 1-wire devices that you have to use in the config file are the names of the subdirectories that you can look up as explained above. So at least for the first time you should do this step to get the slave addresses and maybe write them down for later use.

Driver test and documentation

Test cases

It is a good developer practice to intensively test the code that gets produced and this naturally also applies to WebIOPi device driver code. You can test the code at the end or do this continuously which is standard when you do agile software development (which is my preferred way to develop software). Whatever process you use, make sure the following tests have been done for your drivers:

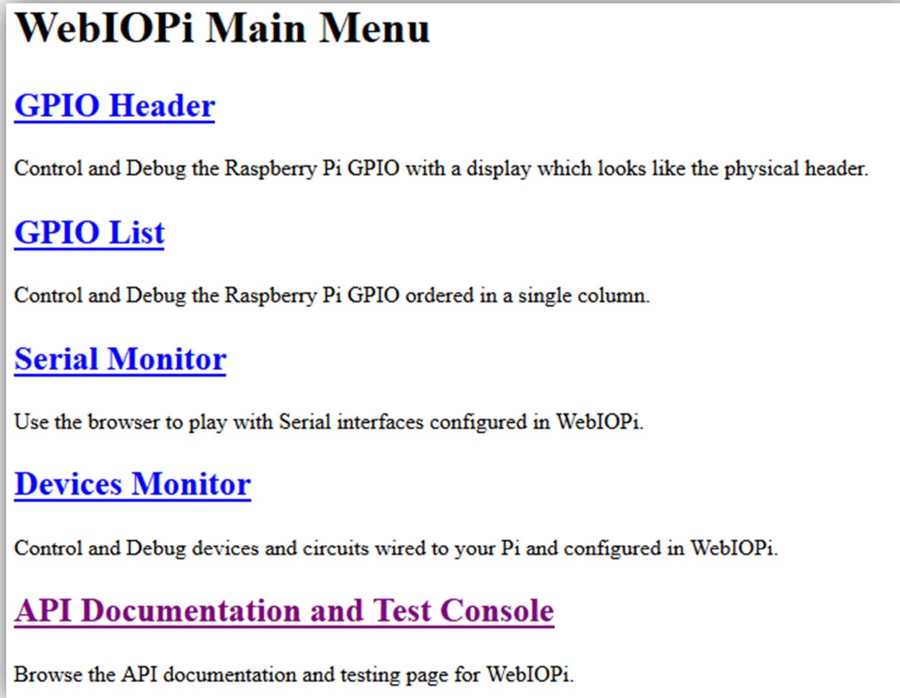
- Driver code should compile without errors or severe warnings (obviously)
- The minimum test requirement is that every main statement is at least used/executed once during the tests (C0 - Statement Coverage Test).
- If you have branches and loops within your code make sure that every branch and loop gets entered at least one time and that all parts of them are really used at least one time (e.g. via different values for the branch/loop parameters) (C1 - Branch Coverage Test).
- More sophisticated testing levels (Cx) exist but this may be a lot of work.
- Test all REST calls across the (local) network with both legal and illegal paths and parameters (especially for HTTP/CoAP **POST**) [Keep in mind, the REST API may be exposed to everyone in the internet world, so make at least sure that submitted illegal calls do not harm something]:
    - If you expect integer numbers e.g. between 0 and 3 for a parameter, try also negative numbers, values > 3 and also very large negative and positive values and to see what happens. Try in this case also float and arbitrary strings (not being legal integers) as parameters and see what happens.
    - Same mechanism applies for float parameters.
    - If you expect strings as parameters, try also strings that would be legal (positive/negative) integers or floats and also strings that are very long and contain special characters.
    - If you expect boolean parameters that are “coded” e.g. with “0”, “1”, “yes”, “no”, etc. try also strings that violate the allowed character range and see what happens.
- Test all public non-REST calls by e.g. invoking them within custom Python server scripts (macros). Apply the same logic to submitted parameters as already explained for the REST calls. Additionally, do some tests that submit wrong object types for Python parameters and see what happens.

Test console

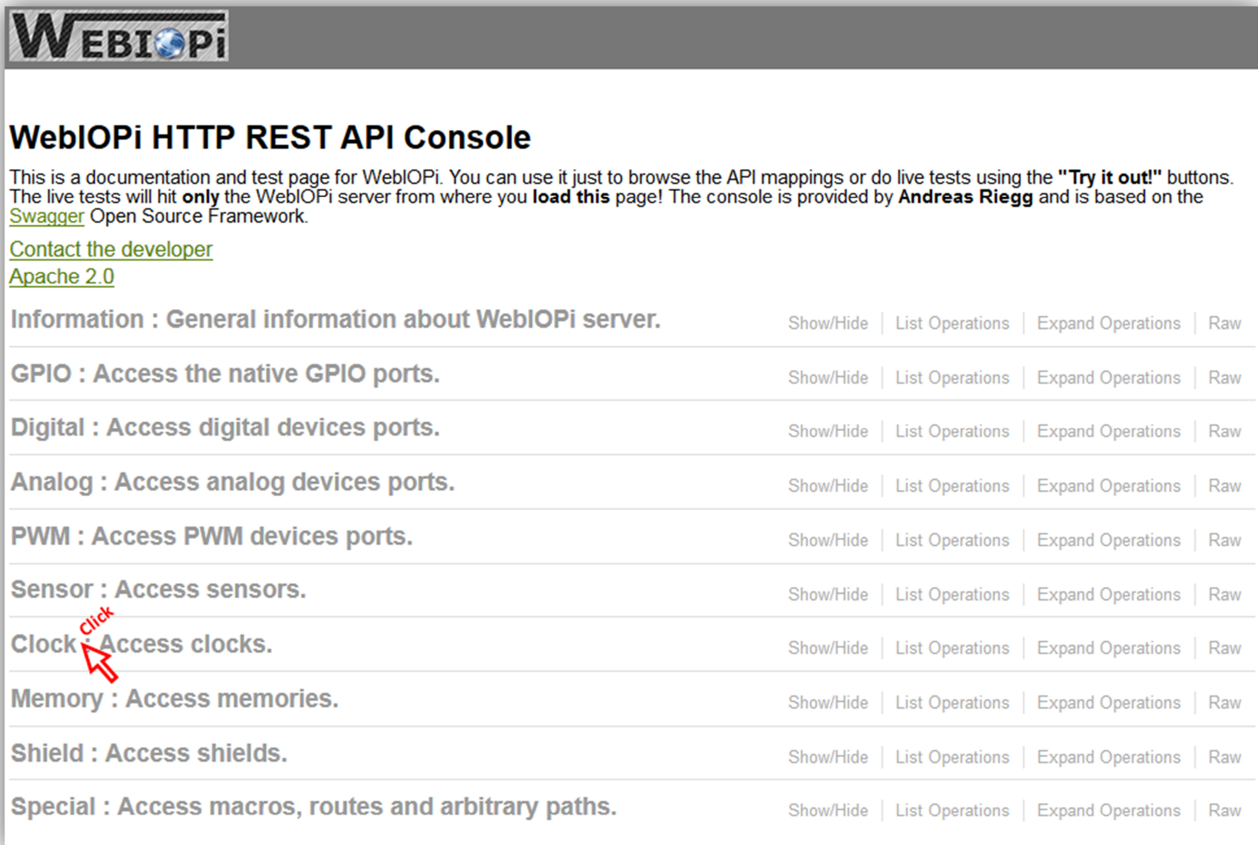
In general, REST API testing can be done with any tool that allows submitting HTTP GET/POST calls and shows the results. This also applies to non-REST API testing when done via Python custom server script macros. It's quite easy to test HTTP REST calls with a browser, you just can type-in the URL and see the result. However, this applies only to HTTP GET calls. To test POST (or any other) calls, you have to use some kind of console tool like curl or HTTP helper tools that e.g. come as a browser extension that allow you to specify the HTTP verb to be used (I often use the Firefox Plugin HttpRequester).

However, it may be a bit cumbersome to type in all the REST calls (and optionally suffer from typos). Some help is available in the form of a Swagger-based HTTP REST API Console. This console is an optional addon for WebIOPi that has been configured to be easy usable for WebIOPi. You find more details on Swagger here [SWAGGER_2014] and can get the resources from here [THINXSWAG_2014].

If you install the addon and call the default homepage of WebIOPi, you will get this modified menu:



The last link will lead you to the Swagger-based test console. The following screenshots show the test console focusing on the RTC Clock APIs. The first image just shows the home page of the console:



The whole console contains clickable links that can be used for navigation. If you click on the text “Clock” this will open the list of all API’s belonging to the Clock abstraction. The values in curly brackets are parameters of the HTTP request. They have to be entered by the user of the test console with appropriate values.

Clock : Access clocks. Click

Show/Hide | List Operations | Expand Operations | Raw

Method	Path	Description
GET	/devices/{clockDeviceName}/clock/*	Returns the full clock state of a configured clock device
GET	/devices/{clockDeviceName}/clock/datetime	Reads the current datetime of a configured clock device
GET	/devices/{clockDeviceName}/clock/date	Reads the current date of a configured clock device
POST	/devices/{clockDeviceName}/clock/date/{dateValue}	Sets and returns the date value of a configured clock device
GET	/devices/{clockDeviceName}/clock/time	Reads the current time of a configured clock device
POST	/devices/{clockDeviceName}/clock/time/{timeValue}	Sets and returns the time value of a configured clock device
GET	/devices/{clockDeviceName}/clock/second	Reads the current second value of a configured clock device
GET	/devices/{clockDeviceName}/clock/minute	Reads the current minute value of a configured clock device
GET	/devices/{clockDeviceName}/clock/hour	Reads the current hour value of a configured clock device
GET	/devices/{clockDeviceName}/clock/dow	Reads the current day of week value of a configured clock device
POST	/devices/{clockDeviceName}/clock/dow/{dowValue}	Sets and returns the day of week value of a configured clock device
GET	/devices/{clockDeviceName}/clock/day	Reads the current day value of a configured clock device
GET	/devices/{clockDeviceName}/clock/month	Reads the current month value of a configured clock device
GET	/devices/{clockDeviceName}/clock/year	Reads the current year value of a configured clock device

If you click on the text “GET” this will open the details of the API’s belonging to the URL “/devices/{clockDeviceName}/clock/*”:

Clock : Access clocks.

Show/Hide | List Operations | Expand Operations | Raw

GET /devices/{clockDeviceName}/clock/* — Returns the full clock state of a configured clock device

Implementation Notes

The full state of the clock device.

Response Class

Model | Model Schema — Click

```
ClockState {
  isodate (string): Date value in ISO format,
  isotime (string): Time value in ISO format,
  isodow (integer): Day of week value in ISO format
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
clockDeviceName	myclock	Clock device name as defined in WebIOPi	path	string

Try it out!

GET /devices/{clockDeviceName}/clock/datetime — Reads the current datetime of a configured clock device

You get some textual information about the purpose of the API, the provided results and the parameters of the call.

The selected API has a parameter of type path with the name “clockDeviceName”. In order to test the API you have to enter a valid device name here. As already mentioned earlier, WebIOPi device names are the names that are given to the devices in the config file.

Sometimes calls have more than one parameter; in this case the parameters list has an entry for each parameter. If you click additionally on “Model schema” you can change the result pattern to a formal sample of the result (for result types “application/json”):

If you finally click on “Try it out!” you submit a real live call to the WebIOPi server. This call goes (hard-wired) only to the server from where the console page has been loaded from. The frame extends further and you can notice the generated URL that was really sent as well as some details about the returned HTTP response:

Taking all this together the Swagger-based console gives both live documentation and basic testing/experimenting facilities in one place.

Documentation

The most important usage hints for a driver should be documented direct in the header comment of a driver source file. In order to be complete it is helpful to provide additional content that can be used to update the WebIOPi wiki pages for the new drivers.

Driver quick checklist

This list gives a compact overview and sequence for all steps to provide a driver. Remember, not every step is required for each chip, some points are optional.

- ☐ Create new /devices sub package (if needed)
- ☐ Create new `__init__.py` with abstract class(es) (if needed)
- ☐ Create new `yourdriver.py` source file(s)
- ☐ Provide copyright and license header for each source file
- ☐ Define new device class(es)
- ☐ Inherit from one/some of the hardware abstraction classes and from one/none of the device communication classes
- ☐ Implement `__init__()` with **ALL** config parameters
- ☐ Implement `__str__()`
- ☐ Implement `__family__()` for hardware abstraction base classes and/or if multiple abstraction functionalities are provided
- ☐ Implement `close()` if not inherited
- ☐ Provide public methods and REST mappings using the Python decorators `@request` and `@response`
- ☐ Update "`DRIVERS`" dictionary in `__init__.py` of devices sub package
- ☐ Update `manager.py`
- ☐ Update `setup.py`
- ☐ Update `webiopi.js`
- ☐ Call `setup.sh` to activate changes and extensions
- ☐ Test your driver code and API
- ☐ Document your driver API
- ☐ Provide test console Swagger API spec (optional)

Troubleshooting/FAQ

Q: I have changed some Python source code of WebIOPi. Is it sufficient just to restart WebIOPi to make the changes active?

A: No, this is not sufficient. WebIOPi uses the Python modules compilation/installation mechanism. For that reason you have to call **`"sudo ./setup.sh skip-apt"`** again before the changes will become active.

Q: If I want to make changes to WebIOPi source code, which source files should I modify? It looks that there are several directories where I can locate some WebIOPi Python sources.

A: When you un-tar the WebIOPi setup file on your server as the first step to install WebIOPi, you will get a /python subdirectory below the directory where you do the un-tar. This is the ONLY source location where you can do the changes. All other locations (one for each Python major version) that seem to contain Python source code are being used during the Python module setup process. Files at those locations have been copied from the original location and will be overwritten. Changes made there have no effect and will be deleted with the next setup. Please keep your fingers off them.

Q: It looks to me that the device parameters from the WebIOPi config file do not get recognized correct, what may be wrong?

A: Please obey the correct syntax of config file for the [DEVICES] section:

```
devicename = DeviceClass parameter1:value1 parameter2:value2
```

one line per device, one blank between parameter/value pairs, no blanks around the colon

---

Q: Chip X has the same (abstract) functionality as chip Y, but WebIOPi just provides a driver for chip X. They both use the same bus (e.g. I2C). Why can't I just use driver X for chip Y?

A: WebIOPi chip drivers (as well as any other chip drivers) have to map the external (abstract) functionality to the internal chip command and register structure and protocol. In most cases, the internal commands and chip registers are structured in different ways for different chip suppliers. For that it is very unlikely that drivers for chip X will work for chip Y.

Q: I'm using some additional hardware with WebIOPi and it seems that my system gets somewhat instable, loses the network connection or WebIOPi stops responding (maybe after some days). The server suffers from unintended shutdowns and reboots. In extreme cases I even ended up with a corrupted SD card. Why makes WebIOPi my system so volatile?

A: WebIOPi does not make your system unstable. But, in most cases, you use WebIOPi in combination with additional hardware and run it 24/7 (this is what WebIOPi is intended to do with it) - and here lies the potential reason for these problems. Most Raspberry Pi's computers are very sensitive to hiccups in their 5V power supply, notably short power under voltages and/or peaks in power consumption. Such hiccups can result from insufficient or bad quality power supplies, from short power peaks produced by some of your additional hardware shields and chips and from momentary power underruns of your electrical power supplier ("brown outs") or comparable instabilities in your mains supply. Before searching for some software tweaks and other miracles to overcome this, first try good quality power supplies and/or power your shield from external separate supplies. In many cases this solves the problem.

Appendices

Code of __init__.py for RTC Clock abstraction class

```
#   Copyright 2014 Andreas Riegg - t-h-i-n-x.net
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#   Changelog
#
#   1.0    2014-06-04    Initial release.
#   1.1    2014-08-17    Added public NON-REST methods that work with
#                        naive Python datetime objects and restructured code
#   1.2    2014-09-23    Added driver lookup for class OsClock
#
#   Usage remarks
#
#   - Date formatting is handled according to ISO8601:2004 (aka ISO calendar).
#   - Public NON-REST methods have real Python datetime objects as parameters
#   - All datetime objects are handeled in naive mode as (most) RTC chips do
#     not handle timezione infos. It's up to the user to decide what time gets
#     stored in the chips. It's recommended to use UTC.
#   - Setting date via REST is only possible as full date string to be able
#     to check calendar consistency
#   - Setting time via REST is handled the same way for consistency and
#     simplification
#

from webiopi.decorators.rest import request, response
from webiopi.utils.types import toint, M_JSON
from datetime import datetime, date, time


class Clock():

    def __init__(self):
        return

#---------- Abstraction framework contracts ----------

    def __family__(self):
        return "Clock"


#---------- Clock abstraction REST implementation ----------

    @request("GET", "clock/*")
    @response(contentType=M_JSON)
    def clockWildcard(self):
        return  {"isotime": self.getTimeString(),
                 "isodate": self.getDateString(),
                 "isodow": self.getDow()}
        # {
        # "isotime" = "12:34:56",
        # "isodate" = "2014-05-13",
        # "isodow" = 3
        # }

    @request("GET", "clock/datetime")
    @response("%s")
    def getDateTimeString(self):
        return self.__getDateTimeString__()

    @request("GET", "clock/date")
    @response("%s")
```

```
    def getDateString(self):
        return self.__getDateString__()

    @request("POST", "clock/date/%(value)s")
    @response("%s")
    def setDateString(self, value):
        year, month, day = self.String2DateValues(value)
        newDate = date(year, month, day)
        self.setDate(newDate)
        return self.getDateString()

    @request("GET", "clock/time")
    @response("%s")
    def getTimeString(self):
        return self.__getTimeString__()

    @request("POST", "clock/time/%(value)s")
    @response("%s")
    def setTimeString(self, value):
        hour, minute, second = self.String2TimeValues(value)
        newTime = time(hour, minute, second)
        self.setTime(newTime)
        return self.getTimeString()

    @request("GET", "clock/second")
    @response("%d")
    def getSec(self):
        return self.__getSec__()

    @request("GET", "clock/minute")
    @response("%d")
    def getMin(self):
        return self.__getMin__()

    @request("GET", "clock/hour")
    @response("%d")
    def getHrs(self):
        return self.__getHrs__()

    @request("GET", "clock/dow")
    @response("%d")
    def getDow(self):
        return self.__getDow__()

    @request("POST", "clock/dow/%(value)d")
    @response("%d")
    def setDow(self, value):
        self.checkDow(value)
        self.__setDow__(value)
        return self.getDow()

    @request("GET", "clock/day")
    @response("%d")
    def getDay(self):
        return self.__getDay__()

    @request("GET", "clock/month")
    @response("%d")
    def getMon(self):
        return self.__getMon__()

    @request("GET", "clock/year")
    @response("%d")
    def getYrs(self):
        return self.__getYrs__()


#---------- Clock abstraction NON-REST implementation ----------

    def getDateTime(self):
        return self.__getDateTime__()

    def setDateTime(self, aDatetime):
        return self.__setDateTime__(aDatetime)

    def getDate(self):
        return self.__getDate__()

    def setDate(self, aDate):
        return self.__setDate__(aDate)
```

```
    def getTime(self):
        return self.__getTime__()

    def setTime(self, aTime):
        return self.__setTime__(aTime)


#---------- Clock abstraction contracts ----------

    def __getSec__(self):
        raise NotImplementedError

    def __setSec__(self, value):
        raise NotImplementedError

    def __getMin__(self):
        raise NotImplementedError

    def __setMin__(self, value):
        raise NotImplementedError

    def __getHrs__(self):
        raise NotImplementedError

    def __setHrs__(self, value):
        raise NotImplementedError

    def __getDay__(self):
        raise NotImplementedError

    def __setDay__(self, value):
        raise NotImplementedError

    def __getMon__(self):
        raise NotImplementedError

    def __setMon__(self, value):
        raise NotImplementedError

    def __getYrs__(self):
        raise NotImplementedError

    def __setYrs__(self, value):
        raise NotImplementedError

    def __getDow__(self):
        raise NotImplementedError

    def __setDow__(self, value):
        raise NotImplementedError


#---------- Clock default implementations ----------
# Rely on reading and writing the base digits, all or some may be reimplemented
# in subclasses for performance improvements by e.g. sequential register access.
# If all methods here that have the __set prefix are reimplemented without using
# any of the __set digit primitives (e.g. __setYrs__()), then these __set methods
# are not mandatory to be reimplemented in order to avoid the NotImplementedError
# as they will never be called (but __setDow__() IS still needed).

    def __getDateTime__(self):
        return datetime(self.__getYrs__(),
                        self.__getMon__(),
                        self.__getDay__(),
                        self.__getHrs__(),
                        self.__getMin__(),
                        self.__getSec__())

    def __setDateTime__(self, aDatetime):
        self.checkYear(aDatetime.year)
        self.__setYrs__(aDatetime.year)
        self.__setMon__(aDatetime.month)
        self.__setDay__(aDatetime.day)
        self.__setHrs__(aDatetime.hour)
        self.__setMin__(aDatetime.minute)
        self.__setSec__(aDatetime.second)

    def __getDate__(self):
        return date(self.__getYrs__(),
```

```
                    self.__getMon__(),
                    self.__getDay__())

    def __setDate__(self, aDate):
        self.checkYear(aDate.year)
        self.__setYrs__(aDate.year)
        self.__setMon__(aDate.month)
        self.__setDay__(aDate.day)

    def __getTime__(self):
        return time(self.__getHrs__(),
                    self.__getMin__(),
                    self.__getSec__())

    def __setTime__(self, aTime):
        self.__setHrs__(aTime.hour)
        self.__setMin__(aTime.minute)
        self.__setSec__(aTime.second)


#---------- Clock abstraction helpers ----------

    def __getDateTimeString__(self):
        theDateTime = self.__getDateTime__()
        return self.Strings2DateTimeString(
            self.DateValues2String(theDateTime.year, theDateTime.month, theDateTime.day),
            self.TimeValues2String(theDateTime.hour, theDateTime.minute, theDateTime.second))

    def __getDateString__(self):
        theDate = self.__getDate__()
        return self.DateValues2String(theDate.year, theDate.month, theDate.day)

    def __getTimeString__(self):
        theTime = self.__getTime__()
        return self.TimeValues2String(theTime.hour, theTime.minute, theTime.second)


#---------- BCD Conversions ----------

    def BcdBits2Int(self, bits):
        if (bits < 0):
            raise NotImplementedError
        elif (bits <= 0xFF):
            # most used case, make it faster
            return (
                    (((bits >> 4)  & 0x0F) * 10) +
                    (bits & 0x0F)
                   )
        elif (bits <= 0xFFFFFFFF):
            # 32 bits, being 8 digits, should be enough ...
            return (
                    (((bits >> 28) & 0x0F) * 10000000) +
                    (((bits >> 24) & 0x0F) * 1000000) +
                    (((bits >> 20) & 0x0F) * 100000) +
                    (((bits >> 16) & 0x0F) * 10000) +
                    (((bits >> 12) & 0x0F) * 1000) +
                    (((bits >> 8)  & 0x0F) * 100) +
                    (((bits >> 4)  & 0x0F) * 10) +
                    (bits & 0x0F)
                   )
        else:
            raise NotImplementedError

    def Int2BcdBits(self, value):
        valueString = "%d" % value
        bcdBits = 0
        digits = len(valueString)
        for i in range(digits):
            bcdBits += int(valueString[i]) << (4 * (digits - 1 - i))
        return bcdBits


#---------- Value checks ----------

    def checkYear(self, year):
        # Most RTC chips handle leap years only correct for 21st century
        # so limit allowed year value to this range
        if not year in range(2000,2100):
            raise ValueError("year [%d] out of range [%d..%d]" % (year, 2000, 2099))
```

```
    def checkDow(self, dow):
        if not dow in range(1,8):
            raise ValueError("dow [%d] out of range [%d..%d]" % (dow, 1, 7))


#---------- Clock values formatting ----------

    def DateValues2String(self, year, month, day):
        #"2014-05-13"
        return "%04d-%02d-%02d" % (year, month, day)

    def TimeValues2String(self, hour, minute, second):
        #"12:34:56"
        return "%02d:%02d:%02d" % (hour, minute, second)

    def Strings2DateTimeString(self, dateString, timeString):
        #"2014-05-13T12:34:56"
        return "%sT%s" % (dateString, timeString)

    def String2DateValues(self, string):
        values = string.split('-')
        year = toint(values[0])
        month = toint(values[1])
        day = toint(values[2])
        return (year, month, day)

    def String2TimeValues(self, string):
        values = string.split(':')
        hour = toint(values[0])
        minute = toint(values[1])
        if len(values) > 2:
            second = toint(values[2])
        else:
            second = 0
        return (hour, minute, second)


#---------- Driver lookup ----------

DRIVERS = {}
DRIVERS["dsrtc"] = ["DS1307", "DS1337", "DS1338", "DS3231"]
DRIVERS["mcprtc"] = ["MCP7940"]
DRIVERS["osrtc"] = ["OsClock"]
```

Code of dsrtc.py for Dallas/Maxim RTC chip drivers

```
#
#   Copyright 2014 Andreas Riegg - t-h-i-n-x.net
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#   Changelog
#
#   1.0    2014-06-26    Initial release.
#   1.1    2014-08-17    Updated to match v1.1 of Clock implementation
#
#   Config parameters
#
#   - control         8 bit       Value of the control register
#
#   Usage remarks
#
#   - All chips have a fixed slave address of 0x68
#   - Driver uses sequential register access to speed up performance
#   - Temperature reading from DS3231 is currently not supported
#   - Setting alarms for DS1337 and DS3231 is currently not supported
#

from webiopi.utils.types import toint
from webiopi.devices.i2c import I2C
from webiopi.devices.clock import Clock
from webiopi.decorators.rest import request, response
from datetime import datetime, date, time


class DSclock(I2C, Clock):

#---------- Constants and definitions ----------

    # Common I2C registers for all DSxxxx clock chips
    SEC  = 0x00  # Seconds register coded as 2-digit BCD
    MIN  = 0x01  # Minutes register coded as 2-digit BCD
    HRS  = 0x02  # Hours register coded as 2-digit BCD
    DOW  = 0x03  # Day of week register coded as 1-digit BCD
    DAY  = 0x04  # Day of month register coded as 2-digit BCD
    MON  = 0x05  # Months register coded as 2-digit BCD
    YRS  = 0x06  # Years register coded as 2-digit BCD

    # Bit masks for common registers for all DSxxxx clock chips
    SEC_MASK = 0b01111111
    MIN_MASK = 0b01111111
    HRS_MASK = 0b00111111
    DOW_MASK = 0b00000111
    DAY_MASK = 0b00111111
    MON_MASK = 0b00011111
    YRS_MASK = 0b11111111


#---------- Class initialization ----------

    def __init__(self, control):
        I2C.__init__(self, 0x68)
        Clock.__init__(self)
        if control != None:
            con = toint(control)
            if not con in range(0, 0xFF + 1):
                raise ValueError("control value [%d] out of range [%d..%d]" % (con, 0x00, 0xFF))
            self.__setCon__(con)
        else:
            self.__setCon__(self.CON_DEFAULT)
```

```
#---------- Clock abstraction related methods ----------

    def __getSec__(self):
        data = self.readRegister(self.SEC)
        return self.BcdBits2Int(data & self.SEC_MASK)

    def __getMin__(self):
        data = self.readRegister(self.MIN)
        return self.BcdBits2Int(data & self.MIN_MASK)

    def __getHrs__(self):
        data = self.readRegister(self.HRS)
        return self.BcdBits2Int(data & self.HRS_MASK)

    def __getDay__(self):
        data = self.readRegister(self.DAY)
        return self.BcdBits2Int(data & self.DAY_MASK)

    def __getMon__(self):
        data = self.readRegister(self.MON)
        return self.BcdBits2Int(data & self.MON_MASK)

    def __getYrs__(self):
        data = self.readRegister(self.YRS)
        return 2000 + self.BcdBits2Int(data & self.YRS_MASK)

    def __getDow__(self):
        data = self.readRegister(self.DOW)
        return self.BcdBits2Int(data & self.DOW_MASK)

    def __setDow__(self, value):
        self.writeRegister(self.DOW, self.Int2BcdBits(value & self.DOW_MASK))

#---------- Clock default re-implementations ----------
# Speed up performance by sequential register access

    def __getDateTime__(self):
        data = self.readRegisters(self.SEC, 7)
        second = self.BcdBits2Int(data[0] & self.SEC_MASK)
        minute = self.BcdBits2Int(data[1] & self.MIN_MASK)
        hour   = self.BcdBits2Int(data[2] & self.HRS_MASK)
        day    = self.BcdBits2Int(data[4] & self.DAY_MASK)
        month  = self.BcdBits2Int(data[5] & self.MON_MASK)
        year   = self.BcdBits2Int(data[6] & self.YRS_MASK) + 2000
        return datetime(year, month, day, hour, minute, second)

    def __setDateTime__(self, aDatetime):
        self.checkYear(aDatetime.year)
        dow = self.__getDow__() # preserve current dow value
        data = bytearray(7)
        data[0] = self.Int2BcdBits(aDatetime.second & self.SEC_MASK)
        data[1] = self.Int2BcdBits(aDatetime.minute & self.MIN_MASK)
        data[2] = self.Int2BcdBits(aDatetime.hour   & self.HRS_MASK)
        data[3] = self.Int2BcdBits(dow              & self.DOW_MASK)
        data[4] = self.Int2BcdBits(aDatetime.day    & self.DAY_MASK)
        data[5] = self.Int2BcdBits(aDatetime.month  & self.MON_MASK)
        data[6] = self.Int2BcdBits((aDatetime.year - 2000)  & self.YRS_MASK)
        self.writeRegisters(self.SEC, data)

    def __getDate__(self):
        data = self.readRegisters(self.DAY, 3)
        day    = self.BcdBits2Int(data[0] & self.DAY_MASK)
        month  = self.BcdBits2Int(data[1] & self.MON_MASK)
        year   = self.BcdBits2Int(data[2] & self.YRS_MASK) + 2000
        return date(year, month, day)

    def __setDate__(self, aDate):
        self.checkYear(aDate.year)
        data = bytearray(3)
        data[0] = self.Int2BcdBits(aDate.day   & self.DAY_MASK)
        data[1] = self.Int2BcdBits(aDate.month & self.MON_MASK)
        data[2] = self.Int2BcdBits((aDate.year - 2000)  & self.YRS_MASK)
        self.writeRegisters(self.DAY, data)

    def __getTime__(self):
        data = self.readRegisters(self.SEC, 3)
        second = self.BcdBits2Int(data[0] & self.SEC_MASK)
        minute = self.BcdBits2Int(data[1] & self.MIN_MASK)
        hour   = self.BcdBits2Int(data[2] & self.HRS_MASK)
```

```
        return time(hour, minute, second)

    def __setTime__(self, aTime):
        data = bytearray(3)
        data[0] = self.Int2BcdBits(aTime.second & self.SEC_MASK)
        data[1] = self.Int2BcdBits(aTime.minute & self.MIN_MASK)
        data[2] = self.Int2BcdBits(aTime.hour   & self.HRS_MASK)
        self.writeRegisters(self.SEC, data)


#---------- Local helpers ----------

    def __getCon__(self):
        data = self.readRegister(self.CON)
        return (data & self.CON_MASK)

    def __setCon__(self, value):
        self.writeRegister(self.CON, (value & self.CON_MASK))


class DS1307(DSclock):

    CON         = 0x07       # Control register address

    CON_MASK    = 0b10010011 # Control register mask
    CON_DEFAULT = 0b10000011 # Control register default value

    CH          = 0b10000000 # Clock halt bit value/mask


#---------- Class initialization ----------

    def __init__(self, control=None):
        DSclock.__init__(self, control)
        # Clock is stopped by default upon poweron, so start it
        self.start()


#---------- Abstraction framework contracts ----------

    def __str__(self):
        return "DS1307"


#---------- Clock abstraction related methods ----------

    def __setSec__(self, value):
        # Keep CH bit unchanged
        secValue = self.Int2BcdBits(value)
        secRegisterData = self.readRegister(self.SEC)
        self.writeRegister(self.SEC, (secRegisterData & ~self.SEC_MASK) | (secValue & self.SEC_MASK))


#---------- Device features ----------

    @request("POST", "run/start")
    def start(self):
        # Clear CH bit, keep sec value unchanged
        secRegisterData = self.readRegister(self.SEC)
        self.writeRegister(self.SEC, (secRegisterData & ~self.CH))

    @request("POST", "run/stop")
    def stop(self):
        # Set CH bit, keep sec value unchanged
        secRegisterData = self.readRegister(self.SEC)
        self.writeRegister(self.SEC, (secRegisterData | self.CH))


class DS1338(DS1307):

    CON_MASK    = 0b10110011 # Control register mask
    CON_DEFAULT = 0b10110011 # Control register default value


#---------- Class initialization ----------

    def __init__(self, control=None):
        DS1307.__init__(self, control)
```

```
#---------- Abstraction framework contracts ----------

    def __str__(self):
        return "DS1338"


class DS1337(DSclock):

    CON         = 0x0E       # Control register address
    CON_MASK    = 0b10011111 # Control register mask
    CON_DEFAULT = 0b00011000 # Control register default value
    EOSC_       = 0b10000000 # Enable oscillator active low bit value/mask


#---------- Class initialization ----------

    def __init__(self, control=None):
        DSclock.__init__(self, control)


#---------- Abstraction framework contracts ----------

    def __str__(self):
        return "DS1337"


#---------- Device features ----------

    @request("POST", "run/start")
    def start(self):
        # Clear EOSC_ bit
        conRegisterData = self.__getCon__()
        self.__setCon__(conRegisterData & ~self.EOSC_)

    @request("POST", "run/stop")
    def stop(self):
        # Set EOSC_ bit
        conRegisterData = self.__getCon__()
        self.__setCon__(conRegisterData | self.EOSC_)


class DS3231(DSclock):

    CON         = 0x0E       # Control register address
    CON_MASK    = 0b11011111 # Control register mask
    CON_DEFAULT = 0b00011100 # Control register default value


#---------- Class initialization ----------

    def __init__(self, control=None):
        DSclock.__init__(self, control)


#---------- Abstraction framework contracts ----------

    def __str__(self):
        return "DS3231"
```

Code of osrtc.py for system clock driver

```
#
#   Copyright 2014 Andreas Riegg - t-h-i-n-x.net
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#   Changelog
#
#   1.0    2014-09-02    Initial release.
#
#   Usage remarks
#
#   - The driver just uses the system clock as available within standard
#     Python. As standard Python does not support setting the system time,
#     this functionality is not available.
#
#   Implementation remarks
#
#   - All __set... methods are not reimplemented, so calling them will lead
#     to the intended NotImplementedError. Dow uses ISO format (1..7).


from webiopi.devices.clock import Clock
from datetime import datetime


class OsClock(Clock):

#---------- Class initialisation ----------

    def __init__(self):
        Clock.__init__(self)


#---------- Abstraction framework contracts ----------

    def __str__(self):
        return "OsClock"

    def close(self):
        return


#---------- Clock abstraction related methods ----------

    def __getSec__(self):
        return (datetime.now()).second

    def __getMin__(self):
        return (datetime.now()).minute

    def __getHrs__(self):
        return (datetime.now()).hour

    def __getDay__(self):
        return (datetime.now()).day

    def __getMon__(self):
        return (datetime.now()).month

    def __getYrs__(self):
        return (datetime.now()).year

    def __getDow__(self):
        return (datetime.now()).isoweekday()
```

```
#---------- Clock default re-implementations ----------

    def __getDateTime__(self):
        return datetime.now()

    def __getDate__(self):
        return (datetime.now()).date()

    def __getTime__(self):
        return (datetime.now()).time()
```

Code (changes) of webiopi.js for RTC Clock abstraction class

```
...

WebIOPi.prototype.newDevice = function(type, name) {
        if (type == "ADC") {
                return new ADC(name);
        }

        if (type == "DAC") {
                return new DAC(name);
        }

...

        if (type == "Clock") {
                return new Clock(name);
        }

        return undefined;
}

...

function Clock(name) {
        this.name = name;
        this.url = "/devices/" + name + "/clock";
        this.refreshTime = 1000;
}

Clock.prototype.toString = function() {
        return this.name + ": Clock";
}

Clock.prototype.getDateTime = function(callback) {
        $.get(this.url + "/datetime", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getDate = function(callback) {
        $.get(this.url + "/date", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getTime = function(callback) {
        $.get(this.url + "/time", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getSec = function(callback) {
        $.get(this.url + "/second", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getMin = function(callback) {
        $.get(this.url + "/minute", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getHrs = function(callback) {
        $.get(this.url + "/hour", function(data) {
                callback(this.name, data);
        });
}

Clock.prototype.getDow = function(callback) {
        $.get(this.url + "/dow", function(data) {
                callback(this.name, data);
        });
}
```

```
Clock.prototype.getDay = function(callback) {
	$.get(this.url + "/day", function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.getMon = function(callback) {
	$.get(this.url + "/month", function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.getYrs = function(callback) {
	$.get(this.url + "/year", function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.setDate = function(value, callback) {
	$.post(this.url + "/date/" + value, function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.setTime = function(value, callback) {
	$.post(this.url + "/time/" + value, function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.setDow = function(value, callback) {
	$.post(this.url + "/dow/" + value, function(data) {
		callback(this.name, data);
	});
}

Clock.prototype.refreshUI = function() {
	var clock = this;
	var element = this.element;
	if ((element != undefined) && (element.header == undefined)) {
		element.header = $("<h3>" + this + "</h3>");
		element.append(element.header);
	}

	this.getDateTime(function(name, data){
		if (element != undefined) {
			element.header.text(clock + ": " + data);
		}
		setTimeout(function(){clock.refreshUI()}, clock.refreshTime);
	});
}
```

Swagger API specification of RTC Clock REST API


```
{
   "$copyright" : "(C)Copyright 2014 Andreas Riegg",
   "$license" : "Licensed under the Apache License, Version 2.0",
   "apiVersion" : "0.7.x",
   "apis" : [
      {
         "description" : "",
         "operations" : [
            {
               "method" : "GET",
               "nickname" : "clockWildcard",
               "notes" : "The full state of the clock device.",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "application/json" ],
               "summary" : "Returns the full clock state of a configured clock device",
               "type" : "ClockState"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/*"
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "GET",
               "nickname" : "getDateTimeString",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Reads the current datetime of a configured clock device",
               "type" : "string"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/datetime"
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "GET",
               "nickname" : "getDateString",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Reads the current date of a configured clock device",
               "type" : "integer"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/date"
```

```
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "POST",
               "nickname" : "setDateString",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  },
                  {
                     "allowMultiple" : false,
                     "description" : "Desired date value of a clock device in ISO format (YYYY-MM-DD),
2000 <= year < 2100",
                     "name" : "timeValue",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Sets and returns the date value of a configured clock device",
               "type" : "string"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/date/{dateValue}"
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "GET",
               "nickname" : "getTimeString",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Reads the current time of a configured clock device",
               "type" : "integer"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/time"
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "POST",
               "nickname" : "setTimeString",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  },
                  {
                     "allowMultiple" : false,
                     "description" : "Desired time value of a clock device in ISO format (HH:MM[:SS])",
                     "name" : "timeValue",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
```

```
            "produces" : [ "text/plain" ],
            "summary" : "Sets and returns the time value of a configured clock device",
            "type" : "string"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/time/{timeValue}"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "GET",
            "nickname" : "getSec",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Reads the current second value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/second"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "GET",
            "nickname" : "getMin",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Reads the current minute value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/minute"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "GET",
            "nickname" : "getHrs",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Reads the current hour value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/hour"
   },
   {
      "description" : "",
      "operations" : [
         {
```

```
            "method" : "GET",
            "nickname" : "getDow",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Reads the current day of week value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/dow"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "POST",
            "nickname" : "setDow",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               },
               {
                  "allowMultiple" : false,
                  "description" : "Desired day of week value of a clock device",
                  "name" : "dowValue",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "integer"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Sets and returns the day of week value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/dow/{dowValue}"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "GET",
            "nickname" : "getDay",
            "parameters" : [
               {
                  "allowMultiple" : false,
                  "description" : "Clock device name as defined in WebIOPi",
                  "name" : "clockDeviceName",
                  "paramType" : "path",
                  "required" : true,
                  "type" : "string"
               }
            ],
            "produces" : [ "text/plain" ],
            "summary" : "Reads the current day value of a configured clock device",
            "type" : "integer"
         }
      ],
      "path" : "/devices/{clockDeviceName}/clock/day"
   },
   {
      "description" : "",
      "operations" : [
         {
            "method" : "GET",
            "nickname" : "getMon",
            "parameters" : [
```

```
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Reads the current month value of a configured clock device",
               "type" : "integer"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/month"
      },
      {
         "description" : "",
         "operations" : [
            {
               "method" : "GET",
               "nickname" : "getYrs",
               "parameters" : [
                  {
                     "allowMultiple" : false,
                     "description" : "Clock device name as defined in WebIOPi",
                     "name" : "clockDeviceName",
                     "paramType" : "path",
                     "required" : true,
                     "type" : "string"
                  }
               ],
               "produces" : [ "text/plain" ],
               "summary" : "Reads the current year value of a configured clock device",
               "type" : "integer"
            }
         ],
         "path" : "/devices/{clockDeviceName}/clock/year"
      }
   ],
   "basePath" : "",
   "models" : {
      "ClockState" : {
         "description" : "WebIOPi clock",
         "id" : "ClockState",
         "properties" : {
            "isodate" : {
               "description" : "Date value in ISO format",
               "type" : "string"
            },
            "isotime" : {
               "description" : "Time value in ISO format",
               "type" : "string"
            },
            "isodow" : {
               "description" : "Day of week value in ISO format",
               "type" : "integer"
            }
         },
         "required" : [ "isodate", "isotime", "isodow" ],
         "type" : "object"
      }
   },
   "resourcePath" : "",
   "swaggerVersion" : "1.2"
}
```

References

- [WEBIOPI_2014]: WebIOPi project homepage (http://code.google.com/p/webiopi/)
- [THINXNET_2014]: T-H-I-N-X.NET resources homepage (http://www.t-h-i-n-x.net)
- [THINX_2014]: Generic IoT Abstract Resources Model document, available on the t-h-i-n-x.net pages (http://www.t-h-i-n-x.net/6.html)
- [SWAGGER_2014]: The Swagger Project to define a standard, language-agnostic interface to REST APIs (https://github.com/wordnik/swagger-spec)
- [THINXSWAG_2014]: T-H-I-N-X.NET WebIOPi resources page containing the Swagger API addon for WebIOPi (http://www.t-h-i-n-x.net/3.html)