ekmmeters Documentation

Release 0.1.0

EKM Metering

April 08, 2016

Contents

CHAPTER 1

Introduction

1.1 Getting Started

For most users, the most effective way to get started is by working through the examples with a live Omnimeter. Most of what this API does is very straightforward, and the examples are written to easily adapt.

It is generally quite difficult to come to terms with a new library (and perhaps a new language) while wrestling with serial connectivity issues. The trial version of EKM Dash is recommended as a way to easily insure your meters are properly connected.

The library offers two modes of data access: via dictionaries and via simpler assign* and extract* calls. Unless you are writing a dedicated device agent which must handle JSON or XML, the assign and extract methods are strongly recommended, both for ease of use and readability.

1.1.1 Goals

The purpose of this library is to provide a reliable and flexible interface to EKM v3 and v4 Omnnimeters. It is written to be compatible with existing EKM field naming conventions and to be accessible to both the casual user – who may not know Python well and simply wishes to script his or her meter – and the experienced user who is creating a device agent for an enterprise metering solution.

The library is written to completely encapsulate the required constants and enumerations for meter control, in a form which is friendly to intellisense style editors and Sphinx documentation. The adopted idiom is simple classes with static members for each categorically unique value.

1.1.2 PEP 8 Note

An implication of the use of EKM naming is that StudlyCase, used widely in existing EKM products, is employed in preference to all lower case function names in PEP 8. Function names keep this vocabulary intact in camelCase. In a (relatively few) cases, descriptive names or tabular presentation was held more important than line length, but in every case the library uses the PyCharm/IntelliJ 120 column lint default. These were implementation choices for an application-domain library, and no changes are planned at this time.

CHAPTER 2

Examples

All of these examples, and several more, can be found in the examples subdirectory of the github source.

To get started, you will want to make sure your meter is set up and you know the name of the port. If in doubt, download a trial copy of EKM Dash and insure that your meters are connected and talking.

Every example below is surrounded by a few lines of setup and teardown.

```
import os       # to delete example db before create
import random   # to generate example data
import time     # to support summarizer sample

from ekmmeters import *

my_port_name = "/dev/ttyS0"
my_meter_address = "300001162"

ekm_set_log(ekm_print_log)
port = SerialPort(my_port_name)

if (port.initPort() == True):
    my_meter = V4Meter(my_meter_address)
    my_meter.attachPort(port)
else:
    print "Cannot open port"
    exit()

# Example goes here

port.closePort()
```

All of the serial commands to the meter return True or False, and log the exceptions. In a long running agent, there is no user action (or programmatic action) requiring exception data: you can only retry until it is clear that the port is not talking to a meter. Generally failing calls will fall through a very short timeout.

Every method making a serial call accepts an optional password parameter (eight numeric characters in a string). The default, shipped with the meter, is “00000000”. Most systems urge setting passwords immediately. We don’t recommend that unless it is a feature on a mature system with a real level of security risk. EKM has no back door into your meter. If you reset and lose your password, it is gone. All of the examples below omit the password parameter and use the default.

2.1 Read

Both V3 and V4 Omnimeters are read with *request()*, which always returns a True or False. Request takes an optional termination flag which forces a "end this conversation" string to be sent to the meter. This flag is only used inside other serial calls: you can just ignore it and leave the default value of True.

The reads from your Omnimeter are returned with *request()* on both V3 and V4 Omnimeters. Omnimeters return data in 255 byte chunks. The supported V3 meter fields come back in one chunk (referred to in the EKM documentation as an A read), and the V4 Omnimeter uses two chunks (referred to as an AB read). The request method is the same on both meter versions.

```
if my_meter.request():
    my_read_buffer = my_meter.getReadBuffer()

    # you can also traverse the buffer yourself,
    #but this is the simplest way to get it all.

    json_str = my_meter.jsonRender(my_read_buffer)
    print json_str
```

2.2 Save to Database

A simple wrapper for Sqlite is included in the library. It is equally avaliable to V3 and V4 meter objects. The saved fields are a union of the v3 and v4 protocol, so additional Omnimeters of either version can be added without changing the table.

If you already have an ORM in place, such as SQLAlchemy, you should define an appropriate object and load it by traversing the read buffer. But for most simple cases, the following will suffice.

The method `insert()` tells the meter object to put the data away in an instantiated MeterDB.

The default behavior of a MeterDB object is built around portable-as-possible SQL: one create statement, which should only be called once, two index creates, an insert, and a drop. In this example we delete the Sqlite database entirely and call create each time.

```
os.remove("test.db")  # keep our example simple

my_db = SqliteMeterDB("test.db")
my_db.dbCreate()

arbitrary_iterations = 20

for i in range(arbitrary_iterations):
    if my_meter.request():
        my_meter.insert(my_db)
```

2.3 CT Ratio

The CT ratio tells the meter how to scale the input from an inductive pickup. It can be set on both V3 and V4 Omnimeters. Allowed values are shown under *CTRatio*.

The CT ratio is set with the method *setCTRatio()*. The field CT_Ratio is returned in every read request.

```
if my_meter.setCTRatio(CTRatio.Amps_800):
    if my_meter.request():
        ct_str = my_meter.getField(Field.CT_Ratio)
        print "CT is " + ct_str
```

2.4 Max Demand Period

The max demand period is a value in the set *MaxDemandPeriod*. The value can be set on both V3 and V4 omnimeters, and it is written with the method *setMaxDemandPeriod()*. The field Max_Demand_Period is returned in every read request.

```
if my_meter.setMaxDemandPeriod(MaxDemandPeriod.At_15_Minutes):
    if my_meter.request():
        mdp_str = my_meter.getField(Field.Max_Demand_Period)
        if mdp_str == str(MaxDemandPeriod.At_15_Minutes):
            print "15 Minutes"
        if mdp_str == str(MaxDemandPeriod.At_30_Minutes):
            print "30 Minutes"
        if mdp_str == str(MaxDemandPeriod.At_60_Minutes):
            print "60 Minutes"
```

2.5 Max Demand Reset Interval

In addition to setting the period for max demand, on V4 Omnimeters you can set an interval to force a reset.

Max demand reset interval is written using *setMaxDemandResetInterval()*, which can return True or False. It accepts values in the set *MaxDemandResetInterval*.

```
if my_meter.setMaxDemandResetInterval(MaxDemandResetInterval.Daily):
     print "Success"
```

2.6 Max Demand Reset Now

On both V3 and V4 Omnimeters, you can force an immediate reset with *setMaxDemandResetNow()*.

```
if my_meter.setMaxDemandResetNow():
     print "Success"
```

2.7 Pulse Output Ratio

On V4 Omnimeters, the pulse output ratio is set using *setPulseOutputRatio()*, which can return True or False. The value must be in the set *PulseOutput*. The field Pulse_Output_Ratio is is returned in every read request.

```
if my_meter.setPulseOutputRatio(PulseOutput.Ratio_5):
    if my_meter.request():
        po_str = my_meter.getField(Field.Pulse_Output_Ratio)
        print po_str
```

2.8 Pulse Input Ratio

On V4 Omnimeters, the pulse input ratios is set using *setPulseInputRatio()*, which can return True or False.

Each of the three pulse lines has an integer input ratio (how many times you must close the pulse circuit to register one pulse). The fields Pulse_Ratio_1, Pulse_Ratio_2 and Pulse_Ratio_3 are returned with every read request. The example below shows line one being set.

```
if my_meter.setPulseInputRatio(Pulse.Ln1, 55):
    if my_meter.request():
        pr_str = my_meter.getField(Field.Pulse_Ratio_1)
        print pr_str
```

2.9 Set Relay

On V4 Omnimeters, the relays toggle using the method *setRelay()*, which can return True or False.

The V4 Omnimeter has 2 relays, which can hold permanently or for a requested duration. The interval limits are in *RelayInterval*, the relay to select in *Relay*, and the requested state in *RelayState*.

If hold-and-stay value is the zero interval. Using the hold constant, Min or 0 will switch the default state on or off (*RelayState*).

```
if my_meter.setRelay(RelayInterval.Hold,
                     Relay.Relay1,
                     RelayState.RelayOpen):

    if my_meter.setRelay(2, Relay.Relay1, RelayState.RelayClose):
        print "Complete"
```

2.10 Set Meter Time

On both V3 and V4 Omnimeters, meter time, which is used by the meter to calculate and store time of use tariffs, is set using the method `setTime()`, and returns True or False. The Meter_Time field is returned with every request. The method `splitEkmDate()` (which takes an integer) will break the date out into constituent parts.

In practice, it is quite difficult to corrupt the meter time, but if it becomes invalid, a request can return a '?' in one of the field positions. In that case your cast to int will throw a ValueException.

EKM meter time is stored in a proprietary year-first format requiring day of week. The API will strip off the century and calculate day of week for you.

Note the meter time is not the same as the timestamp at read, which every agent should capture. Your computer clock, which is calibrated to a time service, is more accurate. The API does not make any assumptions about how you will use Meter_Time, what time zones to employ, or the desirability of periodic corrections (though you can use this library to do all those things).

```
yy = 2023
mm = 11
dd = 22
hh = 15
min = 39
ss = 2

if (my_meter.setTime(yy, mm, dd, hh, min, ss)):
```

```
    if my_meter.request():
        time_str = my_meter.getField(Field.Meter_Time)
        dt = my_meter.splitEkmDate(int(time_str))
        print (str(dt.mm) + "-" +
               str(dt.dd) + "-" +
               str(dt.yy) + " " +
               str(dt.hh).zfill(2) + ":" +
               str(dt.minutes).zfill(2) + ":" +
               str(dt.ss).zfill(2))
    else:
        print "Request failed."
else:
    print "Set time failed."
```

2.11 Zero Resettable

The V4 fields Resettable_Rev_kWh_Tot and Resettable_kWh_Tot are zeroed with function *setZeroResettableKWH()*, which returns True or False.

```
if my_meter.setZeroResettableKWH():
    if my_meter.request():
        print my_meter.getField(Field.Resettable_Rev_kWh_Tot)
        print my_meter.getField(Field.Resettable_kWh_Tot)
```

2.12 Season Schedules

On both V3 and V4 Omnimeters, there are eight schedules, each with four tariff periods. Schedules can be assigned to seasons, with each season defined by a start day and month.

The season definitions are set with *setSeasonSchedules()*, which returns True or False. *setSeasonSchedules()* can use an internal meter buffer or a passed dictionary. Using the internal buffer and *assignSeasonSchedule()* is the simplest approach.

While you can pass an int, using *Seasons* and *Schedules* for the parameters is strongly recommended.

```
my_meter.assignSeasonSchedule(Seasons.Season_1, 1, 1, Schedules.Schedule_1)
my_meter.assignSeasonSchedule(Seasons.Season_2, 3, 21, Schedules.Schedule_2)
my_meter.assignSeasonSchedule(Seasons.Season_3, 6, 20, Schedules.Schedule_3)
my_meter.assignSeasonSchedule(Seasons.Season_4, 9, 21, Schedules.Schedule_8)

if my_meter.setSeasonSchedules():
    print "Success"
```

The method *assignSeasonSchedule()* will return False if the values are out of bounds (though this was omitted from the example above for simplicity).

You can also populate the season schedule using a dictionary, which simplifies loading a meter from passed JSON.

```
param_buf = OrderedDict()
param_buf["Season_1_Start_Month"] = 1
param_buf["Season_1_Start_Day"] = 1
param_buf["Season_1_Schedule"] = 1
param_buf["Season_2_Start_Month"] = 3
param_buf["Season_2_Start_Day"] = 21
param_buf["Season_2_Schedule"] = 2
```

```
param_buf["Season_3_Start_Month"] = 6
param_buf["Season_3_Start_Day"] = 20
param_buf["Season_3_Schedule"] = 3
param_buf["Season_4_Start_Month"] = 9
param_buf["Season_4_Start_Day"] = 21
param_buf["Season_4_Schedule"] = 4

if my_meter.setSeasonSchedules(param_buf):
    print "Completed"
```

2.13 Set Schedule Tariffs

On both V3 and V4 Omnimeters, a schedule is defined by up to four tariff periods, each with a start hour and minute. The meter will manage up to eight schedules.

Schedules are set one at a time via *setScheduleTariffs()*, returning True or False. The simplest way to set up the call is with *assignSeasonSchedule()*, which writes to the meter object internal buffer. The sets *Schedules* and *Tariffs* are provided for readability and convenience.

The following example creates one schedule with tariffs beginning at midnight (rate = 1), 5:30 am (rate = 2), noon (rate = 3), and 5:30 pm (rate 1).

```
my_meter.assignScheduleTariff(Schedules.Schedule_1, Tariffs.Tariff_1, 0,0,1)
my_meter.assignScheduleTariff(Schedules.Schedule_1, Tariffs.Tariff_2, 5,30,2)
my_meter.assignScheduleTariff(Schedules.Schedule_1, Tariffs.Tariff_3, 12,0,3)
my_meter.assignScheduleTariff(Schedules.Schedule_1, Tariffs.Tariff_4, 17,30,1)

if (my_meter.setScheduleTariffs()):
    print "Success"
```

Note that *assignSeasonSchedule()* should be tested for False in a production deployment.

You can also use the range(Extents.<name>) iterator to define all the schedules at once. The test below sets the first tariff and then steps hour and minute for the next three.

```
for schedule in range(Extents.Schedules):
    # create a random time and rate for the schedule
    min_start = random.randint(0,49)
    hr_start = random.randint(0,19)
    rate_start = random.randint(1,7)
    increment = 0
    for tariff in range(Extents.Tariffs):
        increment += 1
        my_meter.assignScheduleTariff(schedule, tariff,
                                      hr_start + increment,
                                      min_start + increment,
                                      rate_start + increment)
    my_meter.setScheduleTariffs()
```

If you are defining a schedule via JSON or XML, you can set the tariffs with a dictionary:

```
param_buf = OrderedDict()
param_buf["Schedule"] = 0
param_buf["Hour_1"] = 1
param_buf["Min_1"] = 11
param_buf["Rate_1"] = 1
param_buf["Hour_2"] = 2
```

```
param_buf["Min_2"] = 21
param_buf["Rate_2"] = 2
param_buf["Hour_3"] = 3
param_buf["Min_3"] = 31
param_buf["Rate_3"] = 3
param_buf["Hour_4"] = 4
param_buf["Min_4"] = 41
param_buf["Rate_4"] = 4

if my_meter.setScheduleTariffs(param_buf):
    print "Success"
```

2.14 Holiday Dates

On both V3 and V4 Omnimeters, a list of up to 20 holidays can be set to use a single schedule (which applies the relevant time of use tariffs to your holidays). The list of holiday dates is written with *setHolidayDates()*, which returns True or False.

Because the holiday list is relatively long, it is the only block without a set of helper constants: if you use *assignHolidayDate()* directly, the holiday is described by an integer from 0 to 19.

A more common use case will see all holidays stored and set at once. The range(Extents.Holidays) idiom can be used to fill the holiday table:

```
for holiday in range(Extents.Holidays):
    day = random.randint(1,28)
    mon = random.randint(1,12)
    my_meter.assignHolidayDate(holiday, mon, day)

my_meter.setHolidayDates()
```

As with the other settings commands, a dictionary can be passed to *setHolidayDates()* for JSON and XML support.

```
param_buf = OrderedDict()
param_buf["Holiday_1_Month"] = 1
param_buf["Holiday_1_Day"] = 1
param_buf["Holiday_2_Month"] = 2
param_buf["Holiday_2_Day"] = 3
param_buf["Holiday_3_Month"] = 4
param_buf["Holiday_3_Day"] = 4
param_buf["Holiday_4_Month"] = 4
param_buf["Holiday_4_Day"] = 5
param_buf["Holiday_5_Month"] = 5
param_buf["Holiday_5_Day"] = 4
param_buf["Holiday_6_Month"] = 0
param_buf["Holiday_6_Day"] = 0
param_buf["Holiday_7_Month"] = 0
param_buf["Holiday_7_Day"] = 0
param_buf["Holiday_8_Month"] = 0
param_buf["Holiday_8_Day"] = 0
param_buf["Holiday_9_Month"] = 0
param_buf["Holiday_9_Day"] = 0
param_buf["Holiday_10_Month"] = 0
param_buf["Holiday_10_Day"] = 0
param_buf["Holiday_11_Month"] = 0
param_buf["Holiday_11_Day"] = 0
```

```
param_buf["Holiday_12_Month"] = 0
param_buf["Holiday_12_Day"] = 0
param_buf["Holiday_13_Month"] = 0
param_buf["Holiday_13_Day"] = 0
param_buf["Holiday_14_Month"] = 0
param_buf["Holiday_14_Day"] = 0
param_buf["Holiday_15_Month"] = 0
param_buf["Holiday_15_Day"] = 0
param_buf["Holiday_16_Month"] = 0
param_buf["Holiday_16_Day"] = 0
param_buf["Holiday_17_Month"] = 0
param_buf["Holiday_17_Day"] = 0
param_buf["Holiday_18_Month"] = 0
param_buf["Holiday_18_Day"] = 0
param_buf["Holiday_19_Month"] = 0
param_buf["Holiday_19_Day"] = 0
param_buf["Holiday_20_Month"] = 1
param_buf["Holiday_20_Day"] = 9

if my_meter.setHolidayDates(param_buf):
    print "Set holiday dates success."
```

2.15 LCD Display

A V4 Omnimeter alternates through up to 40 display items. There are 42 possible display fields, defined in *LCDItems*.

The simplest way to set display items is with the *setLCDCmd()* call, which takes a list of *LCDItems* and returns True or False.

```
lcd_items = [LCDItems.RMS_Volts_Ln_1, LCDItems.Line_Freq]
if my_meter.setLCDCmd(lcd_items):
    print "Meter should now show Line 1 Volts and Frequency."
```

While most meter commands with more than a few of parameters use a dictionary to organize the data (simplifying serialization over the wire), the LCD display items are a single list of 40 integers. A JSON or XML call populated by integer codes is not a good thing. You can translate the name of any value in *LCDItems* to a corresponding integer with *lcdString()*.

```
lcd_items = [my_meter.lcdString("RMS_Volts_Ln_1"),
             my_meter.lcdString("Line_Freq")]

if my_meter.setLCDCmd(lcd_items):
    print "Meter should now show Line 1 Volts and Frequency."
```

2.16 Read Settings

The tariff data used by the Omnimeter (both V3 and V4) amounts to a small relational database, compressed into fixed length lists. There are up to eight schedules, each schedule can track up to four tariff periods in each day, and schedules can be assigned to holidays, weekends, and seasons. The running kWh and reverse kWh for each tariff period is returned with every read, and can be requested for each of the last six recorded months.

The simplest way get the data is all at once, with `readSettings()`, which returns True or False. As it combines 5 read commands, `readSettings()` takes longer than most other API calls.

The data is easy to get but harder to walk. If you do not want to manage offsets and position, you can use the "for <item> in range(Extents.<items>" iteration style, below. Since the lists on the meter are always the same length, you can use the code below as it is, and put your own storage or send function at the bottom of each loop.

We start by reading all the settings tables out the meter object buffers.

```
if my_meter.readSettings():

    # print header line
    print("Schedule".ljust(15) + "Tariff".ljust(15) +
          "Date".ljust(10) + "Rate".ljust(15))

    # There are eight schedules and four tariffs to traverse.  We can
    # safely get indices for extractScheduleTariff -- which returns a
    # single tariff as a tuple -- using the idiom
    # of range(Extents.<item_type>)

    for schedule in range(Extents.Schedules):

        for tariff in range(Extents.Tariffs):

            schedule_tariff = my_meter.extractScheduleTariff(schedule, tariff)

            # and now we can print the returned tuple in a line
            print (("Schedule_" + schedule_tariff.Schedule).ljust(15) +
                   ("kWh_Tariff_" + schedule_tariff.Tariff).ljust(15) +
                   (schedule_tariff.Hour+":"+
                    schedule_tariff.Min).ljust(10) +
                   (schedule_tariff.Rate.ljust(15)))
```

Continuing the traversal of data returned from readSettings(), we get per month data:

```
# print header line
print("Month".ljust(7) + "kWh_Tariff_1".ljust(14) + "kWh_Tariff_2".ljust(14) +
       "kWh_Tariff_3".ljust(14) + "kWh_Tariff_4".ljust(14) +
       "kWh_Tot".ljust(10) + "Rev_kWh_Tariff_1".ljust(18) +
       "Rev_kWh_Tariff_2".ljust(18) + "Rev_kWh_Tariff_3".ljust(18) +
       "Rev_kWh_Tariff_4".ljust(18) + "Rev_kWh_Tot".ljust(11))

# traverse the provided six months:
for month in range(Extents.Months):

     # extract the data for each month
     md = my_meter.extractMonthTariff(month)

     # and print the line
     print(md.Month.ljust(7) + md.kWh_Tariff_1.ljust(14) +
               md.kWh_Tariff_2.ljust(14) + md.kWh_Tariff_3.ljust(14) +
               md.kWh_Tariff_4.ljust(14) + md.kWh_Tot.ljust(10) +
               md.Rev_kWh_Tariff_1.ljust(18) + md.Rev_kWh_Tariff_2.ljust(18) +
               md.Rev_kWh_Tariff_3.ljust(18) + md.Rev_kWh_Tariff_4.ljust(18) +
               md.Rev_kWh_Tot.ljust(10))
```

And continue to list the 20 holidays and their assigned schedule, plus the assigned weekend schedule.

```
# print the header
print("Holiday".ljust(12) + "Date".ljust(20))

# traverse the defined holidays
for holiday in range(Extents.Holidays):
```

```
    # get the tuple ffor each individual holiday
    holidaydate = my_meter.extractHolidayDate(holiday)

    # and print the line
    print(("Holiday_" + holidaydate.Holiday).ljust(12) +
          (holidaydate.Month + "-" + holidaydate.Day).ljust(20))

# the schedules assigned to the above holidays, and to weekends
holiday_weekend_schedules = my_meter.extractHolidayWeekendSchedules()
print "Holiday schedule = " + holiday_weekend_schedules.Holiday
print "Weekend schedule = " + holiday_weekend_schedules.Weekend
```

Without the print statements – assuming you are just pulling the meter data out into your own storage or display, and you can write my_save_tariff(), my_save_month(), my_save_holidays() and my_save_holiday_weekend() functions – the extraction traversal is much shorter. (Please note that unlike every other example on this page, the code below isn't runnable — the my_save functions are just placeholders for your own database writes or display calls).

```
for schedule in range(Extents.Schedules):
    for tariff in range(Extents.Tariffs):
        my_tariff_tuple = my_meter.extractScheduleTariff(schedule, tariff)
        my_save_tariff(my_tariff_tuple)  # handle the tupe printed above

for month in range(Extents.Months):
    my_months_tuple = my_meter.extractMonthTariff(month)
    my_save_month(my_months_tuple) # handle the tuple printed above

for holiday in range(Extents.Holidays):
    holidaydate = my_meter.extractHolidayDate(holiday)
    my_save_holidays(holidaydate.Month, holidaydate.Day)

holiday_weekend_schedules = my_meter.extractHolidayWeekendSchedules()
my_save_holiday_weekend(holiday_weekend_schedules.Holiday,
                        holiday_weekend_schedules.Weekend)
```

By writing four functions to bridge to your own storage or display, you can put away all the non-request meter data fairly simply. Getting the bufffers directly as dictionaries requires individual handling of all repeating fields, and appropriate handling of both schedule blocks and both month blocks stored on the meter. The following example will print all the fields handled by the traversals above, using directly requested buffers.

```
if my_meter.readSettings():

    months_fwd_blk = my_meter.getMonthsBuffer(ReadMonths.kWh)
    months_rev_blk = my_meter.getMonthsBuffer(ReadMonths.kWhReverse)
    sched_1 = my_meter.getSchedulesBuffer(ReadSchedules.Schedules_1_To_4)
    sched_2 = my_meter.getSchedulesBuffer(ReadSchedules.Schedules_5_To_8)
    holiday_blk = my_meter.getHolidayDatesBuffer()

    print my_meter.jsonRender(months_fwd_blk)
    print my_meter.jsonRender(months_rev_blk)
    print my_meter.jsonRender(sched_1)
    print my_meter.jsonRender(sched_2)
    print my_meter.jsonRender(holiday_blk)
```

The readSettings() function itself breaks out to *readScheduleTariffs()*, *readMonthTariffs()* and *readHolidayDates()*. If you take this approach you will need to call *readMonthTariffs()* twice, with ReadMonths.kWh and ReadMonths.kWhReverse, and call *readScheduleTariffs()* twice as well, with parameters ReadSchedules.Schedules_1_To_4 and ReadSchedules.Schedules_5_To_8.

2.17 Meter Observer

This library is intended for programmers at all levels. Most users seeking to summarize their data or generate notifications can do so simply in the main polling loop. However, sometimes only an observer pattern will do. This is a very simple implementation and easily learned, but nothing in this example is necessary for mastery of the API.

Each meter object has a chain of 0 to n observer objects. When a request is issued, the meter calls the subclassed update() method of every observer object registered in its chain. All observer objects descend from MeterObserver, and require an override of the Update method and constructor.

Given that most applications will poll tightly on Meter::request(), why would you do it this way? An observer pattern might be appropriate if you are planning on doing a lot of work with the data for each read over an array of meters, and want to keep the initial and read handling results in a single class If you are writing a set of utilities, subclassing MeterObserver can be convenient. The update method is exception wrapped: a failure in your override will not block the next read.

All of that said, the right way is the course the way which is simplest and clearest for your project.

Using the examples set_notify.py and set_summarize.py (from the github source) is the most approachable way to explore the pattern. All the required code is below, but it may be more rewarding to run from and modify the already typed examples.

We start by moddifying the skeleton we set up at the beginning of this page. with a request loop at the *bottom* of the file, right before closing the serial port. It is a simple count limited request loop, and is useful when building software against this library.

```
ekm_set_log(ekm_no_log)  # comment out to restore

poll_reads = 120   # counts to iterate
print "Starting " + str(poll_reads) + " read poll."
read_cnt = 0  # read attempts
fail_cnt = 0  # consecutive failed reads
while (read_cnt < poll_reads):
   read_cnt += 1
   if not my_meter.request():
      fail_cnt += 1
      if fail_cnt > 3:
         print ">3 consecutive fails. Please check connection and restart"
         exit()
else:
   fail_cnt = 0
```

The notification observer example requires that your meter have pulse input line one hooked up, if only as two wires you can close. To create a notification observer, start by subclassing MeterObserver immediately before the snippet above. The constructor sets a startup test condition and initializes the last pulse count used for comparison.

```
class ANotifyObserver(MeterObserver):

 def __init__(self):

     super(ANotifyObserver, self).__init__()
     self.m_startup = True
     self.m_last_pulse_cnt = 0

 def Update(self, def_buf):

     pulse_cnt = def_buf[Field.Pulse_Cnt_1][MeterData.NativeValue]

     if self.m_startup:
```

```
            self.m_last_pulse_cnt = pulse_cnt
            self.m_startup = False
        else:
            if self.m_last_pulse_cnt < pulse_cnt:
                self.doNotify()
                self.m_last_pulse_cnt = pulse_cnt

    def doNotify(self):
        print "Bells!  Alarms!  Do that again!"
```

Note that our Update() override gets the numeric value directly, using MeterData.NativeValue. It could as easily return MeterData.StringValue, and cast. The first update() sets the initial comparison value. Subsequent update() calls compare the pulse count and check to see if there is a change. The doNotify() method is our triggered event, and can of course do anything Python can.

And finally – right before dropping into our poll loop, we instantiate our subclassed MeterObserver, and register it in the meter's observer chain. We also put the pulse count on the LCD, and set the input ratio to one so every time we close the pulse input, we fire our event.

```
my_observer = ANotifyObserver()
my_meter.registerObserver(my_observer)

my_meter.setLCDCmd([LCDItems.Pulse_Cn_1])
my_meter.setPulseInputRatio(Pulse.Ln1, 1)
```

This example is found in full in the github examples directory for ekmmeters, as set_notify.py. A second example, set_summarize.py, provides a MeterObserver which keeps a voltage summary over an arbitrary number of seconds, passed in the constructor. While slightly longer than the example above, it does not require wiring the meter pulse inputs.

CHAPTER 3

Unit Tests

A very basic unit test framwork is avaliable in the Github project directory. It represents the minimal level of compliance for check-in.

To use the unit tests, you must update unittest.ini with your installed V3 and V4 meter addresses. Both are required. The serial port name is OS appropriate. If you are using an FTDI dongle, list ports to see what appears on insertion.

The unit tests also parameterize whether each serial commmand should pause after completion, and for how long. The default is 100ms, and you should not need to change it. Keep in mind your command is being sent as part of a series of 9600 baud exchanges and the meter processor is quite slow: 100ms to insure serial buffer and interrupt handling completes is a very small relative price. In a well behaved serial driver in a flawless environment, this number is not necessary. If it is useful, the fact will likely only show up on repeated serial runs, where the meter is putting the most stress it can on the UART and any oddness in the combination of drivers and hardware will show up. At the 100ms default, the unit tests complete in about 2 minues and 45 seconnds, most of which is blocking exchanges with the meter. At 50ms, the time drops to about 2 minutes and ten seconds. In most cases this is neither a necessary or desirable optimization target.

If you are learning the library, you will find very good coverage of library functionality in the examples, and example code is an adaptation appropriate skeleton. Unit test logic is built around unittests2, which provides the framework for execution and largely dictates the pattern of error and port handling.

If you do choose to run the unit tests, you will need the ConifigParser, random and unittest2 packages.

CHAPTER 4

Meter Class

The Meter class is the base class for V3Meter and V4Meter. It is never called directly. It encapsulates the next data to send, the last data read, and all of the possible serial commands.

class ekmmeters.**Meter**(*meter_address='000000000000'*)
Abstract base class. Encapuslates serial operations and buffers.

initParamLists()
Initialize all short in-object send buffers to zero.

getReadBuffer()
Required override to fetch the read serial block.

Returns Every supported field (A or A+B, includes all fields)

Return type *SerialBlock*

request(*send_terminator=False*)
Required override, issue A or A+B reads and square up buffers.

Parameters **send_terminator** (*bool*) – Send termination string at end of read.

Returns True on successful read.

Return type bool

serialPostEnd()
Required override, issue termination string to port.

setMaxDemandPeriod(*period*, *password='00000000'*)
Serial call to set max demand period.

Parameters

- **period** (*int*) – : as int.
- **password** (*str*) – Optional password.

Returns True on completion with ACK.

Return type bool

setMaxDemandResetInterval(*interval*, *password='00000000'*)
Serial call to set max demand interval.

Parameters

- **interval** (*int*) – *MaxDemandResetInterval* as int.
- **password** (*str*) – Optional password.

Returns True on completion with ACK.

Return type bool

setMeterPassword(*new_pwd*, *pwd='00000000'*)
Serial Call to set meter password. USE WITH CAUTION.

Parameters

- **new_pwd** (*str*) – 8 digit numeric password to set
- **pwd** (*str*) – Old 8 digit numeric password.

Returns True on completion with ACK.

Return type bool

jsonRender(*def_buf*)
Translate the passed serial block into string only JSON.

Parameters **def_buf** (SerialBlock) – Any *SerialBlock* object.

Returns JSON rendering of meter record.

Return type str

splitEkmDate(*dateint*)
Break out a date from Omnimeter read.

Note a corrupt date will raise an exception when you convert it to int to hand to this method.

Parameters **dateint** (*int*) – Omnimeter datetime as int.

Returns

Named tuple which breaks out as followws:

yy	Last 2 digits of year
mm	Month 1-12
dd	Day 1-31
weekday	Zero based weekday
hh	Hour 0-23
minutes	Minutes 0-59
ss	Seconds 0-59

Return type tuple

getMeterAddress()
Getter for meter object 12 character address.

Returns 12 character address on front of meter

Return type str

registerObserver(*observer*)
Place an observer in the meter update() chain.

Parameters **observer** (MeterObserver) – Subclassed MeterObserver.

unregisterObserver(*observer*)
Remove an observer from the meter update() chain.

Parameters **observer** (MeterObserver) – Subclassed MeterObserver.

getSchedulesBuffer(*period_group*)
Return the requested tariff schedule *SerialBlock* for meter.

Parameters **period_group** (*int*) – A *ReadSchedules* value.

Returns The requested tariff schedules for meter.

Return type *SerialBlock*

getHolidayDatesBuffer()
Get the meter *SerialBlock* for holiday dates.

getMonthsBuffer(*direction*)
Get the months tariff SerialBlock for meter.

Parameters **direction** (*int*) – A *ReadMonths* value.

Returns Requested months tariffs buffer.

Return type *SerialBlock*

setMaxDemandResetNow(*password='00000000'*)
Serial call zero max demand (Dash Now button)

Parameters **password** (*str*) – Optional password

Returns True on completion with ACK.

Return type bool

setTime(*yy*, *mm*, *dd*, *hh*, *minutes*, *ss*, *password='00000000'*)
Serial set time with day of week calculation.

Parameters

- **yy** (*int*) – Last two digits of year.
- **mm** (*int*) – Month 1-12.
- **dd** (*int*) – Day 1-31
- **hh** (*int*) – Hour 0 to 23.
- **minutes** (*int*) – Minutes 0 to 59.
- **ss** (*int*) – Seconds 0 to 59.
- **password** (*str*) – Optional password.

Returns True on completion and ACK.

Return type bool

setCTRatio(*new_ct*, *password='00000000'*)
Serial call to set CT ratio for attached inductive pickup.

Parameters

- **new_ct** (*int*) – A *CTRatio* value, a legal amperage setting.
- **password** (*str*) – Optional password.

Returns True on completion with ACK.

Return type bool

assignScheduleTariff(*schedule*, *tariff*, *hour*, *minute*, *rate*)
Assign one schedule tariff period to meter bufffer.

Parameters

- **schedule** (*int*) – A *Schedules* value or in range(Extents.Schedules).

- **tariff** (*int*) – *Tariffs* value or in range(Extents.Tariffs).
- **hour** (*int*) – Hour from 0-23.
- **minute** (*int*) – Minute from 0-59.
- **rate** (*int*) – Rate value.

Returns True on completed assignment.

Return type bool

setScheduleTariffs(*cmd_dict=None*, *password='00000000'*)
Serial call to set tariff periodds for a schedule.

Parameters

- **cmd_dict** (*dict*) – Optional passed command dictionary.
- **password** (*str*) – Optional password.

Returns True on completion and ACK.

Return type bool

assignSeasonSchedule(*season*, *month*, *day*, *schedule*)
Define a single season and assign a schedule

Parameters

- **season** (*int*) – A *Seasons* value or in range(Extent.Seasons).
- **month** (*int*) – Month 1-12.
- **day** (*int*) – Day 1-31.
- **schedule** (*int*) – A *LCDItems* value or in range(Extent.Schedules).

Returns True on completion and ACK.

Return type bool

setSeasonSchedules(*cmd_dict=None*, *password='00000000'*)
Serial command to set seasons table.

If no dictionary is passed, the meter object buffer is used.

Parameters

- **cmd_dict** (*dict*) – Optional dictionary of season schedules.
- **password** (*str*) – Optional password

Returns True on completion and ACK.

Return type bool

assignHolidayDate(*holiday*, *month*, *day*)
Set a singe holiday day and month in object buffer.

There is no class style enum for holidays.

Parameters

- **holiday** (*int*) – 0-19 or range(Extents.Holidays).
- **month** (*int*) – Month 1-12.
- **day** (*int*) – Day 1-31

Returns True on completion.

Return type bool

setHolidayDates(*cmd_dict=None*, *password='00000000'*)
Serial call to set holiday list.

If a buffer dictionary is not supplied, the method will use the class object buffer populated with assignHolidayDate.

Parameters

- **cmd_dict** (*dict*) – Optional dictionary of holidays.
- **password** (*str*) – Optional password.

Returns True on completion.

Return type bool

setWeekendHolidaySchedules(*new_wknd*, *new_hldy*, *password='00000000'*)
Serial call to set weekend and holiday *Schedules*.

Parameters

- **new_wknd** (*int*) – *Schedules* value to assign.
- **new_hldy** (*int*) – *Schedules* value to assign.
- **password** (*str*) – Optional password..

Returns True on completion and ACK.

Return type bool

readScheduleTariffs(*tableset*)
Serial call to read schedule tariffs buffer

Parameters **tableset** (*int*) – *ReadSchedules* buffer to return.

Returns True on completion and ACK.

Return type bool

extractScheduleTariff(*schedule*, *tariff*)
Read a single schedule tariff from meter object buffer.

Parameters

- **schedule** (*int*) – A *Schedules* value or in range(Extent.Schedules).
- **tariff** (*int*) – A *Tariffs* value or in range(Extent.Tariffs).

Returns True on completion.

Return type bool

readMonthTariffs(*months_type*)
Serial call to read month tariffs block into meter object buffer.

Parameters **months_type** (*int*) – A *ReadMonths* value.

Returns True on completion.

Return type bool

extractMonthTariff(*month*)
Extract the tariff for a single month from the meter object buffer.

Parameters **month** (*int*) – A *Months* value or range(Extents.Months).

Returns

The eight tariff period totals for month. The return tuple breaks out as follows:

kWh_Tariff_1	kWh for tariff period 1 over month.
kWh_Tariff_2	kWh for tariff period 2 over month
kWh_Tariff_3	kWh for tariff period 3 over month
kWh_Tariff_4	kWh for tariff period 4 over month
kWh_Tot	Total kWh over requested month
Rev_kWh_Tariff_1	Rev kWh for tariff period 1 over month
Rev_kWh_Tariff_3	Rev kWh for tariff period 2 over month
Rev_kWh_Tariff_3	Rev kWh for tariff period 3 over month
Rev_kWh_Tariff_4	Rev kWh for tariff period 4 over month
Rev_kWh_Tot	Total Rev kWh over requested month

Return type tuple

readHolidayDates()
Serial call to read holiday dates into meter object buffer.

Returns True on completion.

Return type bool

extractHolidayDate(*setting_holiday*)
Read a single holiday date from meter buffer.

Parameters **setting_holiday** (*int*) – Holiday from 0-19 or in range(Extents.Holidays)

Returns

Holiday tuple, elements are strings.

Holiday	Holiday 0-19 as string
Day	Day 1-31 as string
Month	Monty 1-12 as string

Return type tuple

extractHolidayWeekendSchedules()
extract holiday and weekend `Schedule` from meter object buffer.

Returns

Holiday and weekend `Schedule` values, as strings.

Holiday	`Schedule` as string
Weekend	`Schedule` as string

Return type tuple

readSettings()
Recommended call to read all meter settings at once.

Returns True if all subsequent serial calls completed with ACK.

Return type bool

readCmdMsg()
Getter for message set by last command.

Returns Last set message, if exists.

Return type str

clearCmdMsg()
Zero out the command message result hint string

4.1 SerialBlock Class

Serial block is a simple subclass of OrderedDictionary. The subclassing is primarily cautionary.

class ekmmeters.**SerialBlock**
Simple subclass of collections.OrderedDict.

Key is a *Field* and value is *MeterData* indexed array.

The *MeterData* points to one of the following:

SizeValue	Integer. Equivalent to struct char[SizeValue]
TypeValue	A *FieldType* value.
ScaleValue	A *ScaleType* value.
StringValue	Printable, scaled and formatted content.
NativeValue	Converted, scaled value of field native type.
CalculatedFlag	If True, not part of serial read, calculated.
EventFlag	If True, state value

CHAPTER 5

V3Meter Class

class ekmmeters.**V3Meter**(*meter_address='000000000000'*)

Subclass of Meter and interface to v3 meters.

attachPort(*serial_port*)

Attach required *SerialPort*.

Parameters **serial_port** (SerialPort) – Serial port object, does not need to be initialized.

request(*send_terminator=False*)

Required request() override for v3 and standard method to read meter.

Parameters **send_terminator** (*bool*) – Send termination string at end of read.

Returns CRC request flag result from most recent read

Return type bool

getReadBuffer()

Return *SerialBlock* for last read.

Appropriate for conversion to JSON or other extraction.

Returns A read.

Return type *SerialBlock*

insert(*meter_db*)

Insert to *MeterDB* subclass.

Please note MeterDB subclassing is only for simplest-case.

Parameters **meter_db** (MeterDB) – Instance of subclass of MeterDB.

getField(*fld_name*)

Return *Field* content, scaled and formatted.

Parameters **fld_name** (*str*) – A *Field* value which is on your meter.

Returns String value (scaled if numeric) for the field.

Return type str

CHAPTER 6

V4Meter Class

class ekmmeters.**V4Meter**(*meter_address='000000000000'*)

Commands and buffers for V4 Omnnimeter.

attachPort(*serial_port*)

Required override to attach the port to the meter.

Parameters **serial_port** (SerialPort) – Declared serial port. Does not need to be initialized.

request(*send_terminator=False*)

Combined A and B read for V4 meter.

Parameters **send_terminator** (*bool*) – Send termination string at end of read.

Returns True on completion.

Return type bool

requestA()

Issue an A read on V4 meter.

Returns True if CRC match at end of call.

Return type bool

getReadBuffer()

Return the read buffer containing A and B reads.

Appropriate for JSON conversion or other processing in an agent.

Returns A *SerialBlock* containing both A and B reads.

Return type *SerialBlock*

getField(*fld_name*)

Return *Field* content, scaled and formatted.

Parameters **fld_name** (*str*) – A *:class:~ekmmeters.Field* value which is on your meter.

Returns String value (scaled if numeric) for the field.

Return type str

lcdString(*item_str*)

Translate a string to corresponding LCD field integer

Parameters **item_str** (*str*) – String identical to LcdItems entry.

Returns LcdItems integer or 0 if not found.

Return type int

setLCDCmd(*display_list*, *password='00000000'*)
Single call wrapper for LCD set."

Wraps `setLcd()` and associated init and add methods.

Parameters

- **display_list** (*list*) – List composed of *LCDItems*
- **password** (*str*) – Optional password.

Returns Passthrough from `setLcd()`

Return type bool

setRelay(*seconds*, *relay*, *status*, *password='00000000'*)
Serial call to set relay.

Parameters

- **seconds** (*int*) – Seconds to hold, ero is hold forever. See *RelayInterval*.
- **relay** (*int*) – Selected relay, see *Relay*.
- **status** (*int*) – Status to set, see *RelayState*
- **password** (*str*) – Optional password

Returns True on completion and ACK.

Return type bool

setPulseInputRatio(*line_in*, *new_cnst*, *password='00000000'*)
Serial call to set pulse input ratio on a line.

Parameters

- **line_in** (*int*) – Member of *Pulse*
- **new_cnst** (*int*) – New pulse input ratio
- **password** (*str*) – Optional password

Returns:

setZeroResettableKWH(*password='00000000'*)
Serial call to zero resettable kWh registers.

Parameters **password** (*str*) – Optional password.

Returns True on completion and ACK.

Return type bool

setPulseOutputRatio(*new_pout*, *password='00000000'*)
Serial call to set pulse output ratio.

Parameters

- **new_pout** (*int*) – Legal output, member of *PulseOutput* .
- **password** (*str*) – Optional password

Returns True on completion and ACK

Return type bool

CHAPTER 7

SerialPort Class

class ekmmeters.**SerialPort**(*ttyport*, *baudrate=9600*, *force_wait=0.1*)

Wrapper for serial port commands.

It should only be necessary to create one SerialPort per real port.

Object construction sets the class variables. The port is opened with initPort(), and any serial exceptions will thrown at that point.

The standard serial settings for v3 and v4 EKM meters are 9600 baud, 7 bits, 1 stop bit, no parity. The baud rate may be reset but all timings and test in this library are at 9600 baud. Bits, stop and parity may not be changed.

initPort()

Required initialization call, wraps pyserial constructor.

getName()

Getter for serial port name

Returns name of serial port (ex: 'COM3', '/dev/ttyS0')

Return type string

closePort()

Passthrough for pyserial port close().

write(*output*)

Passthrough for pyserial Serial.write().

Parameters **output** (*str*) – Block to write to port

setPollingValues(*max_waits*, *wait_sleep*)

Optional polling loop control

Parameters

- **max_waits** (*int*) – waits
- **wait_sleep** (*int*) – ms per wait

getResponse(*context=''*)

Poll for finished block or first byte ACK. :param context: internal serial call context.

Returns Response, implict cast from byte array.

Return type string

CHAPTER 8

MeterObserver Class

class ekmmeters.**MeterObserver**

Unenforced abstract base class for implementations of the observer pattern.

To use, you must override the constructor and update().

update(*definition_buffer*)

Called by attached *Meter* on every *request()*.

Parameters **definition_buffer** (SerialBlock) – SerialBlock for request

CHAPTER 9

MeterDB Class

The MeterDb and SqliteMeterDB classes are designed as a helper for the simplest use case: read a meter and save the measurements. A SQL Server descendant class for Iron Python can also be found in the examples.

If you are using this library to write data to an existing program with an ORM such as SQLAlchemy, do not use these classes: extract the data and load to an appropriate object.

Most users of MeterDB will employ an existing subclass (like SqliteMeterDB). Overriding MeterDB is specifically for simple use cases, where overriding between one and five queries (create, insert, drop, and 2 index creates) is more approachable than setting up or learning an ORM.

class ekmmeters.**MeterDB**(*connection_string*)

Base class for single-table reads database abstraction.

setConnectString(*connection_string*)

Setter for connection string. :param connection_string: Connection string.

mapTypeToSql(*fld_type='None'*, *fld_len=0*)

Translate FieldType to portable SQL Type. Override if needful. :param fld_type: *FieldType* in serial block. :type fld_type: int :param fld_len: Binary length in serial block

Returns Portable SQL type and length where appropriate.

Return type string

fillCreate(*qry_str*)

Return query portion below CREATE. :param qry_str: String as built.

Returns Passed string with fields appended.

Return type string

sqlCreate()

Reasonably portable SQL CREATE for defined fields. :returns: Portable as possible SQL Create for all-reads table.

Return type string

sqlInsert(*def_buf*, *raw_a*, *raw_b*)

Reasonably portable SQL INSERT for from combined read buffer. :param def_buf: Database only serial block of all fields. :type def_buf: SerialBlock :param raw_a: Raw A read as hex string. :type raw_a: str :param raw_b: Raw B read (if exists, otherwise empty) as hex string.

Returns SQL insert for passed read buffer

Return type str

sqlIdxMeterTime()
Reasonably portable Meter_Address and Time_Stamp index SQL create. :returns: SQL CREATE INDEX statement.

Return type str

sqlIdxMeter()
Reasonably portable Meter_Address index SQL create. :returns: SQL CREATE INDEX statement.

Return type str

sqlDrop()
Reasonably portable drop of reads table. :returns: SQL DROP TABLE statement.

Return type str

dbInsert(*def_buf*, *raw_a*, *raw_b*)
Call overridden dbExec() with built insert statement. :param def_buf: Block of read buffer fields to write. :type def_buf: SerialBlock :param raw_a: Hex string of raw A read. :type raw_a: str :param raw_b: Hex string of raw B read or empty.

dbCreate()
Call overridden dbExec() with built create statement.

dbDropReads()
Call overridden dbExec() with build drop statement.

dbExec(*query_str*)
Required override for MeterDB subclass, run a query. :param query_str: SQL Query to run.

class ekmmeters.**SqliteMeterDB**(*connection_string='default.db'*)
MeterDB subclass for simple sqlite database

dbExec(*query_str*)
Required override of dbExec() from MeterDB(), run query. :param query_str: query to run

renderJsonReadsSince(*timestamp*, *meter*)
Simple since Time_Stamp query returned as JSON records.

Parameters

- **timestamp** (*int*) – Epoch time in seconds.
- **meter** (*str*) – 12 character meter address to query

Returns JSON rendered read records.

Return type str

renderRawJsonReadsSince(*timestamp*, *meter*)
Simple Time_Stamp query returned as JSON, with raw hex string fields.

Parameters

- **timestamp** (*int*) – Epoch time in seconds.
- **meter** (*str*) – 12 character meter address to query

Returns JSON rendered read records including raw hex fields.

Return type str

CHAPTER 10

Logs and Exceptions

10.1 Logging

The ekmmeters module uses module level logging via predefined callback. It is off by default.

Simple print logging is turned on with:

```
ekm_set_log(ekm_print_log)
```

We *strongly* recommend leaving this on while getting started.

The logging is turned off with:

```
ekm_set_log(ekm_no_log)
```

You can send the output to file or syslog or other destination with a custom callback. A custom callback must be of the form:

```
def my_logging_function(string_out):
    # simplest case: print or log
    print(string_out)
```

The callback is set – exactly as above – via set_log(function_name):

```
set_log(my_logging_function)
```

10.2 Exceptions

Every logging and exception system suggests its own strategy. In this library, serial sets and reads, which rely on a connected meter, will trap exceptions broadly, return False, and log the result. Most other methods will simply allow exceptions to percolate up, for best handling by the caller without reinterpretation.

CHAPTER 11

Constants

Every protocol has a range of irreducible and necessary values. This library uses a simple static class variable, in the approximate style of the C++ enum, to encapsulate and describe the full set of required values.

11.1 Settings

Values used primarily (but not exclusively) for serial settings parameters.

class ekmmeters.**MaxDemandResetInterval**

As passed in *setMaxDemandResetInterval()*. V4 Omnimeters.

Off	0
Monthly	1
Weekly	2
Daily	3
Hourly	4

class ekmmeters.**MaxDemandPeriod**

As passed in *setMaxDemandPeriod()*. V3 and V4 Omnimeters.

At_15_Minutes	1
At_30_Minutes	2
At_60_Minutes	3

class ekmmeters.**LCDItems**

As passed in addLcdItem(). V4 Omnimeters.

kWh_Tot	1
Rev_kWh_Tot	2
RMS_Volts_Ln_1	3
RMS_Volts_Ln_2	4
RMS_Volts_Ln_3	5
Amps_Ln_1	6
Amps_Ln_2	7
Amps_Ln_3	8
RMS_Watts_Ln_1	9
RMS_Watts_Ln_2	10
RMS_Watts_Ln_3	11
RMS_Watts_Tot	12

Continued on next page

Table 11.1 – continued from previous page

Power_Factor_Ln_1	13
Power_Factor_Ln_2	14
Power_Factor_Ln_3	15
kWh_Tariff_1	16
kWh_Tariff_2	17
kWh_Tariff_3	18
kWh_Tariff_4	19
Rev_kWh_Tariff_1	20
Rev_kWh_Tariff_2	21
Rev_kWh_Tariff_3	22
Rev_kWh_Tariff_4	23
Reactive_Pwr_Ln_1	24
Reactive_Pwr_Ln_2	25
Reactive_Pwr_Ln_3	26
Reactive_Pwr_Tot	27
Line_Freq	28
Pulse_Cnt_1	29
Pulse_Cnt_2	30
Pulse_Cnt_3	31
kWh_Ln_1	32
Rev_kWh_Ln_1	33
kWh_Ln_2	34
Rev_kWh_Ln_2	35
kWh_Ln_3	36
Rev_kWh_Ln_3	37
Reactive_Energy_Tot	38
Max_Demand_Rst	39
Rev_kWh_Rst	40
State_Inputs	41
Max_Demand	42

class ekmmeters.**CTRatio**

As passed in *setCTRatio()*. V3 and V4 Omnimeters.

Amps_100	100
Amps_200	200
Amps_400	400
Amps_600	600
Amps_800	800
Amps_1000	1000
Amps_1200	1200
Amps_1500	1500
Amps_2000	2000
Amps_3000	3000
Amps_4000	4000
Amps_5000	5000

class ekmmeters.**Pulse**

As passed to *setPulseInputRatio()*. V4 Omnimeters.

Simple constant to clarify call.

In1	1
In2	2
In3	3

class ekmmeters.**PulseOutput**
As passed to *setPulseOutputRatio()*. V4 Omnimeters.

Ratio_1	Ratio_40
Ratio_2	Ratio_50
Ratio_4	Ratio_80
Ratio_5	Ratio_100
Ratio_8	Ratio_200
Ratio_10	Ratio_400
Ratio_16	Ratio_800
Ratio_20	Ratio_1600
Ratio_25	

class ekmmeters.**Relay**
Relay specified in *setRelay()*. V4 Omnimeters.

Relay1	OUT1 on V4 Meter
Relay2	OUT2 on V4 Meter

class ekmmeters.**RelayState**
Relay state in *setRelay()*. V4 Omnimeters.

RelayOpen	0
RelayClosed	1

class ekmmeters.**RelayInterval**
Relay interval in *setRelay()*. V4 Omnimeters.

Max	9999 seconds
Min	0, parameter limit
Hold	0 (lock relay state)

11.2 Serial Block

Values established when a SerialBlock is initialized.

class ekmmeters.**MeterData**
Each *SerialBlock* value is an array with these offsets. All Omnimeter versions.

SizeValue	0
TypeValue	1
ScaleValue	2
StringValue	3
NativeValue	4
CalculatedFlag	5
EventFlag	6

class ekmmeters.**Field**
Union of all V3A and V4AB Fields Returned.

Use these values to directy get read data with Meter::getField() or in directy traversal of *SerialBlock*.

Meter_Address	12 character Mfr ID’

Continued on next page

Table 11.2 – continued from previous page

Time_Stamp	Epoch in ms at read
Model	Meter model
Firmware	Meter firmware
kWh_Tot	Meter power total
kWh_Tariff_1	Power in timeslot 1
kWh_Tariff_2	Power in timeslot 2
kWh_Tariff_3	Power in timeslot 3
kWh_Tariff_4	Power in timeslot 4
Rev_kWh_Tot	Meter rev. total
Rev_kWh_Tariff_1	Rev power in timeslot 1
Rev_kWh_Tariff_2	Rev power in timeslot 2
Rev_kWh_Tariff_3	Rev power in timeslot 3
Rev_kWh_Tariff_4	Rev power in timeslot 4
RMS_Volts_Ln_1	Volts line 1
RMS_Volts_Ln_2	Volts line 2
RMS_Volts_Ln_3	Volts line 3
Amps_Ln_1	Current line 1
Amps_Ln_2	Current line 2
Amps_Ln_3	Current line 3
RMS_Watts_Ln_1	Instantaneous watts line 1
RMS_Watts_Ln_2	Instantaneous watts line 2
RMS_Watts_Ln_3	Instantaneous watts line 3
RMS_Watts_Tot	Instantaneous watts 1 + 2 + 3
Cos_Theta_Ln_1	Prefix in `CosTheta`
Cos_Theta_Ln_2	Prefix in `CosTheta`
Cos_Theta_Ln_3	Prefix in `CosTheta`
Max_Demand	Demand in period
Max_Demand_Period	*`MaxDemandPeriod`*
Meter_Time	*`setTime()`* and *`splitEkmDate()`*
CT_Ratio	*`setCTRatio`*
Pulse_Cnt_1	Pulse Count Line 1
Pulse_Cnt_2	Pulse Count Line 2
Pulse_Cnt_3	Pulse Count Line 3
Pulse_Ratio_1	*`setPulseInputRatio()`*
Pulse_Ratio_2	*`setPulseInputRatio()`*
Pulse_Ratio_3	*`setPulseInputRatio()`*
State_Inputs’	*`StateIn`*
Power_Factor_Ln_1	EKM Power Factor
Power_Factor_Ln_2	EKM Power Factor
Power_Factor_Ln_3	EKM Power Factor
Reactive_Energy_Tot	Total VAR
kWh_Ln_1	Line 1 power
kWh_Ln_2	Line 2 power
kWh_Ln_3	Line 3 power
Rev_kWh_Ln_1	Line 1 reverse power
Rev_kWh_Ln_2	Line 2 reverse power
Rev_kWh_Ln_3	Line 3 revers power
Resettable_kWh_Tot	*`setZeroResettableKWH()`*
Resettable_Rev_kWh_Tot	*`setZeroResettableKWH()`*
Reactive_Pwr_Ln_1	VAR Line 1

Continued on next page

Table 11.2 – continued from previous page

Reactive_Pwr_Ln_2	VAR Line 2
Reactive_Pwr_Ln_3	VAR Line 3
Reactive_Pwr_Tot	VAR Total
Line_Freq	Freq. Hz.
State_Watts_Dir	*DirectionFlag*
State_Out	*StateOut*
kWh_Scale	*ScaleKWH*
RMS_Watts_Max_Demand	Power peak in period
Pulse_Output_Ratio	*PulseOutput*
Net_Calc_Watts_Ln_1	RMS_Watts with Direction
Net_Calc_Watts_Ln_2	RMS_Watts with Direction
Net_Calc_Watts_Ln_3	RMS_Watts with Direction
Net_Calc_Watts_Tot	RMS_Watts with Direction
Status_A	Reserved diagnostic.
Status_B	Reserved diagnostic.
Status_C	Reserved diagnostic.

Power_Factor is the only power factor measurement supported by upstring EKM products. The original Cos Theta value is provided as an API-only feature.

class ekmmeters.**ScaleType**
Scale type defined in SerialBlock. V4 Omnimeters.

These values are set when a field is defined a SerialBlock. A Div10 or Div100 results in immediate scaling, otherwise the scaling is perfformed per the value in Field.kWh_Scale as described in ScaleKWH.

KWH	*ScaleKWH*
No	Do not scale
Div10	Scale 10^-1
Div100	Scale 10^-2

class ekmmeters.**FieldType**
Every SerialBlock element has a field type. V3 and V4 Omnimeters.

Data arrives as ascii. Field type determines disposition. The destination type is Python.

NoType	Not type assigned, invalid
Hex	Implicit hex string
Int	Implicit int
Float	Implicit float
String	Leave as string, terminate
PowerFactor	EKM L or C prefixed pf

11.3 Meter

Values used to select meter object buffers to operate on. V4 and V3 Omnimeters.

class ekmmeters.**ReadSchedules**
For *readScheduleTariffs()* and *getSchedulesBuffer()*. V3 and V4.

Schedules_1_To_4	1st 4 blocks tariffs and schedules
Schedules_5_To_8	2nd 4 blocks tariffs and schedules

class ekmmeters.**ReadMonths**
As passed to *readMonthTariffs()* and *getMonthsBuffer()*. V3 and V4.

Use to select the forward or reverse six month tariff data.

kWh	Select forward month tariff data
kWhReverse	Select reverse month tariff data

11.4 Data

Values which only appear in a read. V4 Omnimeters.

class ekmmeters.**DirectionFlag**
On V4, State_Watts_Dir mask shows RMS_Watts direction on line 1-3.

The Direction flag is used to generate Calc_Net_Watts field on every read. Each word in constant is the direction of the corresponding at the moment of read. Ex ForwardReverseForward means RMS_Watts lines one and three are positive, and line two is negtive.

ForwardForwardForward	1
ForwardForwardReverse	2
ForwardReverseForward	3
ReverseForwardForward	4
ForwardReverseReverse	5
ReverseForwardReverse	6
ReverseReverseForward	7
ReverseReverseReverse	8

class ekmmeters.**ScaleKWH**
Scaling or kWh values controlled by Fields.kWh. V4 Omnimeters.

If MeterData.ScaleValue is ScaleType.KWH, Fields.kWh_Scale one of these.

NoScale	0	no scaling
Scale10	1	scale 10^-1
Scale100	2	scale 10^-2
EmptyScale	-1	Reserved

class ekmmeters.**StateIn**
State of each pulse line at time of read. V4 Omnimeters.

HighHighHigh	0
HighHighLow	1
HighLowHigh	2
HighLowLow	3
LowHighHigh	4
LowHighLow	5
LowLowHigh	6
LowLowLow	7

class ekmmeters.**StateOut**
Pulse output state at time of read. V4 Omnimeters.

OffOff	1
OffOn	2
OnOff	3
OnOn	4

11.5 Traversal

Values primarily (but not exclusively) used for extraction from or assignment to serial buffers. V3 and V4 Omnimeters.

class ekmmeters.**Extents**
Traversal extents to use with for range(Extent) idiom. V3 and V4 Omnimeters.

Use of range(Extent.Entity) as an iterator insures safe assignnment without off by one errors.

Seasons	4
Holidays	20
Tariffs	4
Schedules	8
Months	6

class ekmmeters.**Seasons**
As passed to *assignSeasonSchedule()*. V3 and V4 Omnimeters.

assign* methods use a zero based index for seasons. You may set a season using one of these constants or fill and iterate over range(Extents.Seaons).

Season_1	0
Season_2	1
Season_3	2
Season_4	3

class ekmmeters.**Months**
As passed to *extractMonthTariff()*. V3 and V4 Omnimeters.

Month_1	0
Month_2	1
Month_3	2
Month_4	3
Month_5	4
Month_6	5

class ekmmeters.**Tariffs**
As passed to *assignScheduleTariff()*. V3 and V4 Omnimeters.

Tariff_1	0
Tariff_2	1
Tariff_3	2
Tariff_4	3

class ekmmeters.**Schedules**
Allowed schedules. V3 and V4 Omnimeters.

Schedules on the meter are zero based, these apply to most passed schedule parameters.

Schedule_1	0
Schedule_2	1
Schedule_3	2
Schedule_4	3
Schedule_5	4
Schedule_6	5
Schedule_7	6
Schedule_8	7

CHAPTER 12

Index and Search

- genindex
- search

S

T

U

V

W