Kamstrup OmniPower wm-bus metering system

Release beta

Team 3, E5PRO5 2020, Aarhus Uni., School of Engineering

Nov 24, 2020

TABLE OF CONTENTS

This documentation covers the system to read the Kamstrup OmniPower 1-phase meter over wm-bus. The implementation uses an iM871-A transceiver to read wm-bus messages. Processed measurements are sent upstream using MQTT.

CHAPTER

ONE

RUNNING THE METERING SYSTEM

1.1 Metering system main event loop

Synopsis This script is main loop which handles the system flow

Authors Steffen, Thomas, Janus

Latest update 17 November 2020

Version 0.92

Version history

- **Ver. 0.1**: Build main loop with queue and Mqtt startup.
- **Ver. 0.9**: Implement mqtt to get command from ReCalc, dispatcher, and mqtt to send data to ReCalc.
- **Ver. 0.91**: Implement (1) gw-id from settings into topics, (2) mqtt pub rc check (log on err), (3) Use DEBUG instead of print.
- **Ver. 0.92**: Move functions out of __main__ section to document them.

1.1.1 Starting and stopping the system

- **Full system is started using shell script *start_stop_system.sh*.**
  - Start: *./start_stop_system.sh start*
  - Stop: *./start_stop_system.sh stop*
- Driver will be running as daemon and will create a FIFO as driver/IM871A_pipe.
- This script will visibly run in terminal (debug).

1.1.2 Stopping the system over MQTT

- **Send message with *topic: STOP* to end program. Example:**
  - *mosquitto_pub -h <INSERT IP> -p 1883 -t STOP -m 'anything' -u <INSERT USER> -P <INSERT PWD>*

1.1.3 Data flows

- Config flow: ReCalc command (mqtt) -> Update Dispatcher
- Data flow: Driver -> (FIFO) -> C1-parser -> Dispatcher -> Handler (OmniPower) -> Mqtt publish data
- Error logging flow: On errors -> Logger -> SysLog

1.1.4 Testing

- **Test message to start monitoring our OmniPower (fill in username and password):**
  - *mosquitto_pub -h <INSERT IP> -t “v2/706462169/sensors/set” -m ‘[{“deviceId”: “32666857”,”manufacturerKey”: “kam”,”encryptionKey”: “9A25139E3244CC2E391A8EF6B915B697”, “manufacturerDeviceKey”: “OmniPower1”}]’ -u <INSERT USER> -P <INSERT PWD>*
- **Test to receive published data (fill in username and password):**
  - *mosquitto_sub -h <INSERT IP> -t “#” -u <INSERT USER> -P <INSERT PWD>*

1.1.5 Handling data from ReCalc

Monitored list is built based on serial numbers from ReCalc messages:

- Keeps object (dispatcher) to keep track of monitored meters.
- No method (or way) to report if a serial numbers is invalid.
- Invalid serial numbers will be monitored, but no data will ever be sent.
- Consider expanding ReCalc Cloud API to receive messages about invalid commands.
- Currently only able to send config messages for OmniPower devices.

1.1.6 Future implementation of more meters

- Expand the api.config_json() function to take as arguments the name/type/manufacturer of a device.
- Return a config string based on these properties.
- Base configs on yaml files or other files that can be updated on the fly without restarting code.

1.2 Main event loop

`run.run_system.`**`run_system`**`()`

Implements the main loop to run the entire system.

1.3 Support functions

`run.run_system.`**`on_command_callback`**`(`*client*, *userdata*, *message*`)`

On every message from ReMoni ReCalc, the message is put into an atomic, threadsafe queue. Atomic deque, dq, is from global scope from __main__ section.

`run.run_system.`**`end_loop`**`()`

Function to cleanly exit loop and end threads, disconnect. From __main__ section: FIFO queue, fifo; Mqtt subscriber, recalc; Mqtt publisher.

`run.run_system.`**`DEBUG`**`(`*message: str*`)`

Prints out if global DEBUG_ON set to True.

CHAPTER

TWO

IMPLEMENTATION OF IM871-A DRIVER

2.1 Generic driver class for WM-Bus USB-dongle IM871A

Platform Python 3.5.10 on Linux

Synopsis This module implements a class for communication with IM871A module.

Authors Steffen Breinbjerg, Thomas Serup

Date 21 October 2020

2.1.1 Version history

- Ver 1.0: Set up driver.
- Ver 1.1: Implemented seperate 'open pipe' handler. Added pipe path as 2.nd argument.
- Ver 1.2: Implemented CRC-16 check.
- Ver 1.3: Logging exceptions to syslog instead of printing to console.
- Ver 1.4: No longer takes USB port as argument. Function for handling port is located in 'utils/Search_for_dongle'.

2.1.2 Link Modes

IM871A is able to run in different modes. Default mode is S2.

Mode	Argument	Description
S1	s1	Stationary, one way communication
S1-m	s1m	S1 with shorter header
S2	s2	Stationary, bidirectional communication
T1	t1	Frequent transmit, one way communication
T2	t2	Frequent transmit, bidirectional communication
C1, Telegram Format A	c1a	Compact, one way communication. No fixed length
C1, Telegram Format B	c1b	Compact, one way communication. Fixed length
C2, Telegram Format A	c2a	Compact, bidirectional communication. No fixed length
C2, Telegram Format B	c2b	Compact, bidirectional communication. Fixed length

class `driver.DriverClass.`**`IM871A`**(*program_path*, *logOnDestruct=True*)

Implementation of a driver class for IM871A USB-dongle. Takes 1 argument1: - The path to where to put the pipe, e.g. the program directory.

close()
: Close the connection to IM871A, and the pipe.

open() → bool
: Opens the port if port has been closed. It opens with the path given when instantiating the class. Also open the pipe.

open_pipe() → bool
: Open up the pipe. Blocks until pipe is opened at the other end.

ping() → bool
: Ping the WM-Bus module to check if it's alive.

read_data() → bool
: Read single dataframe from meters sending with the specified link mode. Send data into 'named pipe' (USBx_pipe). Removes the WM-Bus frame before sending data to pipe.

reset_module() → bool
: Reset the WM-Bus module. The reset will be performed after approx. 500ms.

setup_linkmode(*mode: str*) → bool
: Setup link mode for communication with meter. Takes the link mode as argument. If no Link Mode is set, default is 'S2'

CHAPTER

THREE

OMNIPOWER IMPLEMENTATION

3.1 Parse Kamstrup OmniPower wm-bus telegrams

platform Python 3.5.10 on Linux, OS X

synopsis Implements parsing functionality for C1 telegrams and log handling for data series

author Janus Bo Andersen

date 28 October 2020

3.1.1 Version history

- Ver 1.0: Set up parser and decryption. Janus.
- Ver 2.0: Implement CRC16, timezone. Janus.
- Ver 2.1: More robust exception handling, parse ELL-SN. Janus
- Ver 2.2: Utilize new MeterMeasurement.is_empty() in validation during parsing. Janus

3.1.2 Overview

- This module implements parsing for the Kamstrup OmniPower meter, single-phase.
- The meter sends wm-bus C1 (compact one-way) telegrams.
- Telegrams on wm-bus are little-endian, i.e. LSB first.
- The meter sends 1 long and 7 short telegrams, and then repeats.
- Long telegrams include data record headers (DRH) and data, that is DIF/VIF codes + data.
- Short telegrams only include data.

3.1.3 Telegram fields

In a telegram C1 telegram, the data fields are:

#	Byte#	Bytes	M-bus field	Description	Expected value (little-endian)
0	0	1	L	Telegram length	0x27 (39 bytes, short frame), or 0x2D (45 bytes, long frame)
1	1	1	C	Control field (type and purpose of message)	0x44 (SND_NR)
2	2-3	2	M	Manufacturer ID (official ID code)	0x2D2C (KAM)
3	4-7	4	A	Address (meter serial number)	0x57686632 (big-endian:32666857)
4	8	1	Ver.	Version number of the wm-bus firmware	0x30
5	9	1	Medium	Type / medium of meter	0x02 (Electricity)
6	10	1	CI	Control Information	0x8D (Extended Link Layer 2)
7	11	1	CC	Communication Control	0x20 (Slow response sync.)
8	12	1	ACC	Access field	Varies
9	13-16	4	AES CTR / SN	AES counter (Session number)	Varies, see below
10	17-39 17-45	23 29	Data	Contains AES-encrypted data frame, varying for short and long frames	Encrypted data
11		2	CRC16	CRC16 check	

The fields 0-9 of the telegram can be unpacked using the little-endian format *<BBHIBBBBB*, where

- < marks little-endian,
- *B* is an unsigned 1 byte (char),
- *H* is an unsigned 2 byte (short),
- *I* is an unsigned 4 byte (int)

3.1.4 AES counter / Session number

The AES_CTR / Extended Link Layer SN field (ELL-SN) is is structured as per EN 13757-4, p. 54.

The example shows ELL-SN value of 0x01870320 (little-endian) -> 0x20038701 (big-endian). Bit readout:

Byte:	3	2	1	0
Hex:	0x20	0x03	0x87	0x01
Binary:	0010 0000	0000 0011	1000 0111	0000 0001

Should give the following slicing and interpretation:

Bits:	31-29	28-04	03-00
Field:	ENC	Time	Session
Example:	001	0 0000 0000 0011 1000 0111 0000	0001
Hex:	0x1	0x3870 (14448)	0x1

- ENC (Encryption): 0 -> No encryption, 1 -> AES-CTR mode, higher -> reserved.

- Time: Minute counter since 01/01/2013 (?), or since meter started, requires RTC set on meter. So time measurement was taken about 10 days after the meter was started.
- Session: Incremented by meter for each transmission, unless using partial/fractured frames.

Parsing. The whole ELL-SN field is read out and masks are used to extract fields.

3.1.5 Telegram examples

Encrypted short telegrams:

L	C	M	A	Ver	Med	CI	CC	ACC	AES CTR	Encrypted payload	CRC 16
27	44	2D 2C	5768 6632	30	02	8D	20	2E	2187 0320	D3A4F149 B1B8F578 3DF7434B 8A66A557 86499ABE 7BAB59	xxxx
27	44	2d 2c	5768 6632	30	02	8d	20	63	60dd 0320	c42b87f4 6fc048d4 2498b44b 5e34f083 e93e6af1 617631	3d9c
27	44	2d 2c	5768 6632	30	02	8d	20	8e	11de 0320	188851bd c4b72dd3 c2954a34 1be369e9 089b4eb3 858169	494e

Encrypted long telegrams:

L	C	M	A	Ver	Med	CI	CC	ACC	AES CTR	Encrypted payload	CRC 16
2D	44	2D 2C	5768 6632	30	02	8D	20	64	61DD 0320	38931d14 b405536e 0250592f 8b908138 d58602ec a676ff79 e0caf0b1 4d	0e7d

3.1.6 Decryption

- The wireless m-bus on OmniPower uses AES-128 Mode CTR (if enabled, otherwise no encryption).
- See EN 13757-4:2019, p. 54, as ELL (Ext. Link-Layer) with ENC = 0x1 => AES-CTR.
- A decryption prefix (initial counter block) is built from some of the fields.
- See table 54 on p. 55 of EN 13757-4:2019.

It can be packed using the format *<HIBBBIB*.

M	A	Ver	Med	CC	AES_CTR	FN	BC
2D2C	57686632	30	02	20	21870320	0000	00

- AES Prefix (initialization vector): Fields M, . . . , AES_CTR, FN.
- FN: frame number (frame # sent by meter within same session number, in case of multi-frame transmissions).
- AES Counter: BC
- BC: Block counter (encryption block number, counts up for each 16 byte block decrypted within the telegram).

3.1.7 Decrypted payload examples

The interpretation of the fields in the OmniPower is

Field	Kamstrup name	Data fmt (DIF)	Value type (VIF/E)	VIF/E meaning	DIF VIF/E
Data 1	A+	32-bit uint	Energy, 10^1 Wh	Consumption from grid, accum.	04 04
Data 2	A-	32-bit uint	Energy, 10^1 Wh	Production to grid, accum.	04 84 3C
Data 3	P+	32-bit uint	Power, 10^0 W	Consumption from grid, instantan.	04 2B
Data 4	P-	32-bit uint	Power, 10^0 W	Production to grid, instantan.	04 AB 3C

Transport layer control information fields (TPL-CI), ref. EN 13757-7:2018, p. 17, introduce Application Layer (APL) as:

- 0x78 with full frames (Response from device, full M-Bus frame)
- 0x79 with compact frames (Response from device, M-Bus compact frame)

Data integrity check (CRC16)

The first 2 bytes (16 bits) of a payload is always the CRC16 value of the sent message. This value must be checked versus CRC16 calculated on the received payload.

Decrypted short telegram payload

CRC16	TPL-CI	Data fmt. sign.	CRC16 data	Data 1	Data 2	Data 3	Data 4
1170	79	138C	4491	CE000000	00000000	03000000	00000000

Measurement data starts at byte 7, and can easily be extracted using *<IIII* little-endian format.

In this example, 206 10^1 Wh (2.06 kWh) have been consumed, and the current power draw is 3 10^0 W (0.003 kW).

Decrypted long telegram payload

In this kind of telegram, the DRHs are included.

CRC16	TPL-CI	DIF/VIF 1	Data 1	DIF/VIF/VIFE 2	Data 2	DIF/VIF 3	Data 3	DIF/VIF 4	Data 4
9831	78	04 04	D7000000	04 84 3C	00000000	04 2B	03000000	04 AB 3C	00000000

Extraction is slightly more complex, requiring either a longer parsing pattern or perhaps a regex.

In this example, 215 10^1 Wh (2.15 kWh) have been consumed, and the current power draw is 3 10^0 W (0.003 kW).

3.2 The C1 Telegram class

class `meter.OmniPower.`**`C1Telegram`**(*telegram: bytes*)

Implements capture of data fields for a C1 telegram from OmniPower

`decrypt_using`(*meter: meter.OmniPower.OmniPower*) → bool

Decrypts a telegram and inserts it into telegram (self) Uses a meter object and its key to perform decryption. Requires instantiated OmniPower meter with valid AES-key.

3.3 The OmniPower class

class `meter.OmniPower.`**`OmniPower`**(*name: str = 'Kamstrup OmniPower one-phase', meter_id: str = '32666857', manufacturer_id: str = '2C2D', medium: str = '02', version: str = '30', aes_key: str = '9A25139E3244CC2E391A8EF6B915B697'*)

Implementation of our OmniPower single-phase meter Passed values are hex encoded as string, e.g. '2C2D' for value 0x2C2D.

`add_measurement_to_log`(*measurement: meter.MeterMeasurement.MeterMeasurement*) → bool

Pushes a new measurement to the tail end of the log

`decrypt`(*telegram: meter.OmniPower.C1Telegram*) → bytes

Decrypt a telegram. Returns decrypted bytes. Raises CrcCheckException if CRCs do not match after decryption.

Requires:

- the prefix from the telegram (telegram.prefix), and
- the encryption key stored in the meter object.

Decrypts the data stored in field telegram.encrypted

`dump_log_to_json`() → str

Returns a JSON string of all measurement frames in log, with an incremented number for each observation.

`extract_measurement_frame`(*telegram: meter.OmniPower.C1Telegram*) → *meter.MeterMeasurement.MeterMeasurement*

Requires that the telegram is already decrypted, otherwise returns empty measurement frame.

`is_this_my`(*telegram: meter.OmniPower.C1Telegram*) → bool

Check whether a given telegram is from this meter by comparing meter setting to telegram

`process_telegram`(*telegram: meter.OmniPower.C1Telegram*) → bool

Does entire processing chain for a telegram, including adding to log. Returns True if processing is OK and added to log OK. Otherwise False.

classmethod `unpack_long_telegram_data`(*data: bytes*) → Tuple[int, ...]

Long C1 telegrams contain DIF/VIF information and field data values

classmethod `unpack_short_telegram_data`(*data: bytes*) → Tuple[int, ...]

Short C1 telegrams only contain field data values, no information about DIF/VIF

3.4 Exception classes

class `meter.OmniPower.`**`TelegramParseException`**(*exception_message: str*)

Use this to raise an exception if a bytestream telegram fails to parse into C1 format.

class `meter.OmniPower.`**`AesKeyException`**(*exception_message: str*)

Use this to raise an exception when an AES key is missing or wrong length.

CHAPTER

FOUR

IMPLEMENTATION OF GENERIC MEASUREMENTS

4.1 Generic class for measurements and measurement frames

platform Python 3.5.10 on Linux, OS X

synopsis This module implements classes for generic measurements taken from a meter.

authors Janus Bo Andersen, Jakob Aaboe Vestergaard

date 13 October 2020

Changelog: 03 Nov 2020: Added is_empty() method to MeterMeasurement. Janus.

4.2 The Measurement class

class `meter.MeterMeasurement.`**`Measurement`**(*value: float*, *unit: str*)

Single physical measurement. A single measurement of a physical quantity pair, consisting of a value and a unit.

4.3 The MeterMeasurement class

class `meter.MeterMeasurement.`**`MeterMeasurement`**(*meter_id: str*, *timestamp: datetime.datetime*)

A single measurement collection based on one frame from the meter. Will contain multiple measurements of physical quantities taken at the same time.

`add_measurement`(*name: str*, *measurement: meter.MeterMeasurement.Measurement*) → None

Store a new measurement in the collection.

`as_dict`() → dict

Serializes and outputs the Measurement frame as a structured dict.

`is_empty`()

Must return True if no measurements have been added, otherwise False

`json_dump`() → str

Returns a JSON formatted string of all data in frame.

CHAPTER

FIVE

MQTT AND API IMPLEMENTATION

5.1 MQTT clients and related functionality

5.1.1 MQTT-class for communication between Gateway and ReCalc/Cloud at Re-Moni

Platform Python 3.5.10 on Linux

Synopsis This module implements a class for MQTT Client

Authors Steffen Breinbjerg, Janus Bo Andersen

Latest update 18 November 2020

Version 1.2

- **Ver. 1.0**: Setup MQTT class with loaded settings.
- **Ver. 1.1**: Implement support functions for return codes and better printout for on_connect.
- **Ver. 1.2**: Implemented TLS and better handling of reason codes.

5.1.2 MqttClient class

`class mqtt.MqttClient.`**`MqttClient`**(*name*, *on_message*, *on_publish*, *param_settings='mqtt_local'*)

This class wraps a Paho MQTT client and sets it up using a profile from settings/secrets.yaml. On instantiation, all setup steps run up to and including connect.

`__init__`(*name*, *on_message*, *on_publish*, *param_settings='mqtt_local'*)

Handles all setup and connection when object is initialized.

Parameters

- **name** (*str*) – Client ID (must be unique)
- **on_message** (*function_ptr*) – On message callback function
- **on_publish** (*function_ptr*) – On publish callback function
- **param_settings** (*str*) – Profile from secrets.yaml to use

`disconnect`()

Gracefully disconnects from server.

`loop_forever`()

Will block the program, and only handle callback functions and disconnects.

loop_start()
Start loop in new thread. This thread will handle disconnects to MQTT broker.

loop_stop()
Stops the loop thread.

static on_connect(*client: paho.mqtt.client.Client*, *userdata*, *flags*, *rc*)
on_connect callback rc argument value meaning:

- 0: Connection successful
- 1: Connection refused - incorrect protocol version
- 2: Connection refused - invalid client identifier
- 3: Connection refused - server unavailable
- 4: Connection refused - bad username or password
- 5: Connection refused - not authorised 6-255: Currently unused.

publish(*topic*, *payload*)

Parameters

- **topic** (*str*) – Topic to publish to
- **payload** (*str*) – Message payload

Returns (result, mid)

Return type

tuple

- result: 0 on success, 4 if no connection
- mid is message id for tracked message

subscribe(*topic*, *qos*)

Parameters

- **topic** (*str*) – Topic to subscribe to.
- **qos** (*int*) – Quality of service level (0, 1, 2).

Returns (result, mid)

Return type

tuple

- result: 0 on success, 4 if no connection
- mid is message id for tracked message

5.1.3 Supporting functions

`mqtt.MqttClient.`**`connection_rc_str`**`(`*rc: Tuple*`)` → str

Returns text description of return codes for a connection attempt to server. Based on return code spec. from Paho MQTT. Alternatively, use mqtt.connack_string().

`mqtt.MqttClient.`**`publish_rc_str`**`(`*rc: Tuple*`)` → str

Returns text description of return codes for publishing a message. Based on return code spec. from Paho MQTT. Alternatively, use mqtt.error_string().

`mqtt.MqttClient.`**`publish_rc_bool`**`(`*rc: Tuple*`)` → bool

Returns True if message was sent correctly, otherwise False. Based on return code spec. from Paho MQTT. The rc tuple is (reason_code, message_id). Alternatively, use mqtt.error_string().

5.2 API implementation

5.2.1 API compliant messages

Synopsis Implements functionality to send and receive correctly formatted messages as spec'ed by Re-Moni API v2.

Authors Jakob, Steffen, Janus

Last update 18 Nov. 2020.

`mqtt.api.`**`build_api_message_from_log_obj`**`(`*m:* meter.MeterMeasurement.MeterMeasurement`)` → List[Any]

Due to bug in ReCalc, this currently only returns a list of Python dicts. In the future, should return the same dumped to JSON.

`mqtt.api.`**`config_json`**`()` → str

Returns a JSON-formatted string to config OmniPower in ReCalc API.

CHAPTER

SIX

UTILITIES AND SUPPORT FUNCTIONS

6.1 CRC16 for wm-bus

6.1.1 CRC16 for wm-bus

synopsis CRC16 calculator for EN 13757

author Janus Bo Andersen

date October 2020

Overview:

- This function performs the CRC16 algorithm.
- The result can be used to confirm data integrity of received payload in a wm-bus message.
- Wm-bus follows the CRC16 standard outlined in EN 13757.

A *CrcCheckException* class is also implemented, which is used to raise exceptions if a CRC check fails.

The IM871-A transceiver removes the outer CRC16 (last two bytes of a message) and replaces it with its own, which follows another standard, CRC16-CCITT. So the outer CRC16 can not be checked with the function in this module.

CRC16 EN 13757:

- CRC16 uses a generator polynomial, g(x), described in EN 13757-4.
- See p. 42 for data-link layer CRC, and an example with a C1 telegram on p. 84.
- See p. 58 for transport layer CRC polynomial.

g(x) = x^16 + x^13 + x^12 + x^11 + x^10 + x^8 + x^6 + x^5 + x^2 + 1

In binary (excluding x^16 as it is shifted out anyway), this g(x) is represented as

Byte 1	Byte 2	Hex value
0011 1101	0110 0101	0x3D65
MSbit = x^15	LSbit = x^0 = 1	

See EN 13757-4, table 43, p. 50 for expected structure of ELL for a CI=0x8D telegram. PayloadCRC is included in the encrypted part of telegram.

Algorithm rules:

- Treats data most-significant bit first
- Final CRC shall be complemented
- Multi-byte data is transmitted LSB first
- CRC is transmitted MSB first

Math background:

- CRC uses a finite field F=[0, 1], so we do subtraction using XOR.
- CRC is the final remainder from repeated long division of message by polynomial,when no further division is possible.
- The output CRC is complemented by XOR with 0xFFFF.

Algorithm implementation comments:

The implemented algorithm uses Python's ability for 'infinite' width of integers. That is slightly inefficient, and can't be ported to C code on an embedded device. But it is significantly easier to debug and understand than byte-wise algorithms or lookup tables.

6.1.2 CRC16 functions

`utils.crc16_wmbus.`**`crc16_wmbus`**(*message: bytes*) → bytes

Takes a bytes object with a message (ascii encoded hex values). Returns the CRC16 value for the message encoded in a bytes object.

Example: f(b'79138C4491CE000000000000000300000000000000') -> b'1170'.

`utils.crc16_wmbus.`**`crc16_check`**(*payload: bytes*) → bool

Takes a payload and splits into CRC16-field and message. Computes CRC16 on the message and compares to CRC16-field. Return True if match. Raises CrcCheckException if no match.

6.1.3 CRC check exception

`class` `utils.crc16_wmbus.`**`CrcCheckException`**(*crc_recv: bytes, crc_calc: bytes, exception_message: str*)

Use this to raise an exception when a CRC16 check has failed.

6.2 CRC16 for IM871-A

6.2.1 Implementation of CRC16 for IM871-A (CCITT)

synopsis CRC16 CCITT implementation based on Steffen's C implementation.

authors Steffen and Janus.

date 29 Oct 2020.

- See IMST's WMBUS_HCL_Spec_V1.6.pdf.

- CRC computation starts from the Control Field and ends with the last octet of the Payload Field.
- IM871A uses CRC16-CCITT Polynomial G(x) = 1 + x^5 + x^12 + x^16.

6.2.2 CRC16 function

`utils.crc16_im871a.`**`crc16_im871a_check`**(*m: bytes*) → bool

Confirm CRC16 integrity of a full bytestring received from IM871-A. Argument must be the entire message from IM871-A. Function returns TRUE when the check sum matches the expected CRC16 value.

`utils.crc16_im871a.`**`crc16_im871a_calc`**(*m: bytes*) → bytes

Compute CRC16 (CCITT) for a message received serially from IM871-A. Argument must:

- NOT contain the first field, e.g. 0xA5.
- Start and contain the control field, e.g. 0x8203.
- Contain full payload, e.g. 0x2744…2637.
- NOT contain the trailing 2 bytes of expected CRC16 value, e.g. 0xC1AB.

Full message example: a5 8203 27442d2c5768663230028d20cd12340720519df247ff65e751662a300bc4e5c67da86477f0182637 c1ab.

6.3 Timezone handling

6.3.1 Zulu timezone handling

synopsis Stamp measurements with Zulu time (UTC), and easy future-proof management of timestamps as required by ReMoni.

author Janus Bo Andersen.

date October 2020.

description Output dates in ISO 8601 format, i.e. output 2020-10-25T10:08:00Z for 25th October 2020 at 10:08:00 (HH:MM:SS) in UTC time.

Note that ISO 8601 allows using +00:00 instead of Z. ReMoni prefers Z for this implementation.

- UTC class implementation based on: https://docs.python.org/3.5/library/datetime.html
- Zulu time definition based on: https://en.wikipedia.org/wiki/ISO_8601

6.3.2 ZuluTime class

`class` `utils.timezone.`**`ZuluTime`**

Very explicit Zulu-time class to implement Zulu time as UTC time. Implements required methods on abstract base class *tzinfo*.

`dst`(*dt*)

No Daylight savings time.

`tzname`(*dt*)

Name of timezone.

`utcoffset`(*dt*)

UTC is zero time offset from UTC, of course.

6.3.3 Print datetime object with ISO 8601 format

`utils.timezone.`**`zulu_time_str`**`(`*timestamp: datetime.datetime*`)` → str

Print a timestamp with ISO format like 2020-10-25T10:08:00Z

6.4 Logging to syslog

`utils.log.`**`log_error`**`(`*message*`)` → None

Function for sending logging message to syslog file. Use this one for error messages.

`utils.log.`**`log_info`**`(`*message*`)` → None

Function for sending logging message to syslog file. Use this one for info messages.

6.5 Automatic search for IM871-A dongle device

Experimental: Search for IMST IM871a WMbus dongle. Looking at udev rules in Linux: Default OS placed in /lib/udev/rules.d - These cannot be changed. If user want to create custom udev rules this should be placed in /etc/udev/rules.d

Rules placed in /etc have priority over /lib.

Specific for serial communication rules OS uses 60-serial.rules where one line define location and name for the link to tty PORT used for specific device.

ENV{.ID_PORT}==”?*”, SYMLINK+=”serial/by-id/$env{ID_BUS}-$env{ID_SERIAL} -
if$env{ID_USB_INTERFACE_NUM}-port$env{.ID_PORT}”

Looking at the line its clear that the link is placed in /dev/serial/by-id. Furthermore the name of the link is given by the ID_BUS, ID_SERIAL ,ID_USB_INTERFACE_NUM, ID_PORT. At the moment it’s believed but not confirmed that ID_USB_INTERFACE_NUM and ID_PORT is subject to change if different port is assigned.

This method first confirms the directory /dev/serial/by-id exist, afterwards looks for iM871a in any link name. If any the method will return the absolute path. If Several links contains “iM871a” the method will return path to first encountered link.

CHAPTER SEVEN

UNIT TESTS

7.1 Tests for driver implementation

Tests for IM871-A driver. Uses mocked tests when not on Gateway. On Gateway, tests run using hardware peripheral.

Future dev.:

- Try spec=True to investigate if MagicMock can correctly spec the patched classes and functions.
- E.g. when patching serial.Serial, the MagicMock should appear to have all relevant methods.
- This could eliminate some need for spec'ing our own test doubles (fakes).

`test.test_DriverClass.`**`IM871A_pipe`**`()`
: Fixture that returns a path to use for the pipe

`class` `test.test_DriverClass.`**`PatchSerial`**`(`*read_raw_return*`)`
: Fakes the serial.Serial object. Avoids attempts of write/read operations from /dev/tty devices on local machines.

  `close``()`

  `read``(`*num_bytes*`)`

  `write``(`*message_bytes*`)`

`class` `test.test_DriverClass.`**`PipeWriter`**
: Fakes the builtin method used to open a pipe and write to it. This avoids pipes being opened on local machines.

  `close``()`

  `flush``()`

  `write``(`*message*`)`

`test.test_DriverClass.`**`input_data`**`()`
: Fixture that returns raw usb data, processed data and processed data with errors

`test.test_DriverClass.`**`is_on_gateway`**`()`
: Returns true if this is the gateway

`test.test_DriverClass.`**`patched_driver`**`(`*mock_obj_im871a_port*, *mock_obj_fifo*, *mock_obj_serial_conn*, *mock_open*`)`
: Make a driver fixture where we've patched (bypass/override) the port.Serial dependency with our own fake object. The port.Serial-function (serial.Serial) is used when the driver tries to establish serial connection. Patch os.mkfifo to prevent making FIFO (pipe) in the local OS during object construction. Set up that we test on one specific test vector.

`test.test_DriverClass.`**`test_CRC_check_fails_RPi`**`(`*IM871A_pipe*, *input_data*`)`
: Tests if a unsuccesfull CRC-check returns false

`test.test_DriverClass.`**`test_CRC_check_succes_RPi`**(*IM871A_pipe*, *input_data*)
: Tests if a succesfull CRC-check returns true

`test.test_DriverClass.`**`test_close`**(*patched_driver*)

`test.test_DriverClass.`**`test_constructor_destructor`**(*patched_driver*)
: Test construction and destruction of a driver object is OK.

`test.test_DriverClass.`**`test_linkmodes_RPi`**(*IM871A_pipe*)
: Testing all the linkmodes possible with IM871A

`test.test_DriverClass.`**`test_object_instatiated_true_RPi`**(*IM871A_pipe*)
: Testing if an object of the type IM871A can be instantiated

`test.test_DriverClass.`**`test_read_data`**(*patched_driver*)
: Already Patch out the pipe dependency to an instance of local PipeWriter-type object that we can easily read.

`test.test_DriverClass.`**`test_read_data_RPi`**(*IM871A_pipe*, *input_data*)
: Testing if it's possible to read from the pipe

`test.test_DriverClass.`**`test_setup_linkmode`**(*patched_driver*)

`test.test_DriverClass.`**`test_usb_essentials_RPi`**(*IM871A_pipe*)
: Testing if it is possible to open the pipe and reset the USB-module

`test.test_DriverClass.`**`test_vectors`**()
: Build test set of raw serial data and ascii (processed) data.

7.2 Tests for OmniPower implementation

Tests for the functionality of OmniPower implementation.

`test.test_OmniPower.`**`bad_payload_list`**()
: Sets up a mangled telegram.

`test.test_OmniPower.`**`bad_telegrams_list`**()
: Sets up a list of bad telegrams that must cause exceptions at various places.

`test.test_OmniPower.`**`good_telegrams_list`**()
: Sets up a list of good telegrams.

`test.test_OmniPower.`**`omnipower_base`**()
: Creates an good OmniPower object with no data in log.

`test.test_OmniPower.`**`omnipower_setup`**(*omnipower_base*)
: Sets up an omnipower test fixture with at least one telegram stored in log.

`test.test_OmniPower.`**`omnipower_with_no_aes_key`**(*omnipower_base*)
: Creates a good OmniPower object with empty AES key.

`test.test_OmniPower.`**`test_Omnipower_longtelegram`**(*omnipower_base*)
: Assure Omnipower class can process long telegrams

`test.test_OmniPower.`**`test_Omnipower_noAESkey`**(*omnipower_base*)
: Assure that Omnipower class can't decrypt with wrong or no AES-key

`test.test_OmniPower.`**`test_Omnipower_notmytelegram`**(*omnipower_base*)
: Assure Omnipower class rejects a telegram from an unknown sensor

`test.test_OmniPower.`**`test_Omnipower_shorttelegram`**(*omnipower_base*)
: Assure Omnipower class can process long telegrams

test.test_OmniPower.**test_c1telegram_must_raise_exception**(*bad_telegrams_list*)
Test that C1 Telegram initialized with bad bytestream raises exception.

test.test_OmniPower.**test_decrypt_must_raise_aes_key_error**(*omnipower_with_no_aes_key*, *good_telegrams_list*)
If AES key is not OK, decrypt must raise an AesKeyException.

test.test_OmniPower.**test_decrypt_must_raise_crc_check_error**(*omnipower_base*, *bad_payload_list*)
If the payload has been modified or mangled, CRC16 check must fail, and a CrcCheckException is raised, which passes through .decrypt().

test.test_OmniPower.**test_decrypt_using_must_return_false_for_bad_key**(*omnipower_with_no_aes_key*, *good_telegrams_list*)
Decrypt_using is the telegram that attempts to decrypt itself using a meter object. If the AES key in the meter object is bad, it cannot be used for decryption. Then decrypt_using must return False to signify failed operation. Test strategy: Good telegram + bad AES key -> AesKeyException.

test.test_OmniPower.**test_decrypt_using_must_return_false_for_bad_payload**(*omnipower_base*, *bad_payload_list*)
Decrypt_using is the telegram that attempts to decrypt itself using a meter object. If the payload is bad, the meter object cannot successfully validate CRC16. Then decrypt_using must return False to signify failed operation. Test strategy: Bad payload + good AES key -> CrcCheckException.

test.test_OmniPower.**test_extract_measurement_frame_returns_empty_if_tlg_not_decrypted**(*omnipo*
*good_*
Expect OmniPower's extract_measurement_frame to return an empty object if the telegram has not been decrypted, so there is no data to extract. Per spec., it must then return an empty object. Test strategy, used the implemented method .is_empty() to test this.

test.test_OmniPower.**test_json_full_log**(*omnipower_setup*)
Test that a full log of MeterMeasurement objects dumped to JSON can all be recovered correctly.

test.test_OmniPower.**test_process_telegram_returns_false_if_not_parsable**(*omnipower_base*, *good_telegrams_list*)
Expect OmniPower's process_telegram to return False if the telegram cannot be parsed / handled by OmniPower. Reasons:

- Not sent from this meter
- decrypt_using returns false (tested above)
- empty frame returned (tested above)
- add_measurement_to_log fails (not tested)

7.3 Tests for MeterMeasurement implementation

Tests for the functionality of MeterMeasurement implementation.

test.test_MeterMeasure.**MeasureFix**()
Setup-fixture for Measurement-class

test.test_MeterMeasure.**initialized_measurement_frame**()
Sets up a measurement frame with ID and fixed Zulu timestamp, but no data.

test.test_MeterMeasure.**keys**()
Setup-fixture for keys

test.test_MeterMeasure.**omnipower_setup**()
Sets up an omnipower test fixture with at least one telegram stored in log.

`test.test_MeterMeasure.`**`test_add_measurement`**(*MeasureFix*, *keys*)
Test the “add_measurement”-method from MeterMeasurement

`test.test_MeterMeasure.`**`test_as_dict`**(*MeasureFix*, *keys*)
Test the “as_dict” method from MeterMeasurement

`test.test_MeterMeasure.`**`test_json_single_measurement`**(*omnipower_setup*)
Test that a single MeterMeasurement dumped to JSON can be recovered correctly.

`test.test_MeterMeasure.`**`test_meter_measurement_returns_empty`**(*initialized_measurement_frame*)
A MeterMeasurement with no data added must return True on is_empty() method.

CHAPTER EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX