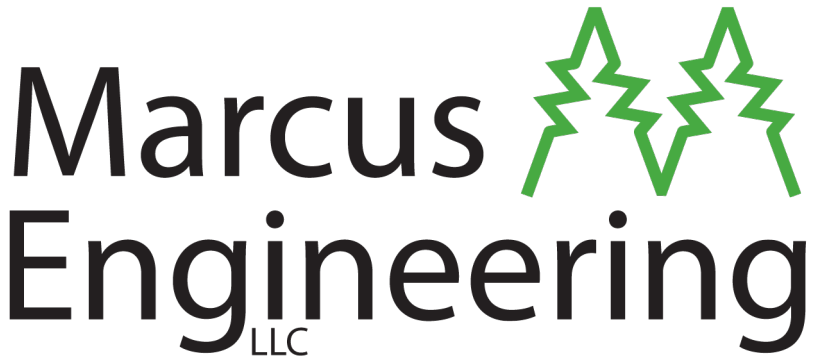

AGM Config Utility v1.0.0Documentation

Marcus Engineering, LLC

Feb 22, 2021

CONTENTS:

CHAPTER

ONE

OVERVIEW

The AGM Configuration Utility is used to interface with a collection of AGM electronic humidity sensors (Gen 1.1, Gen 2.1, and Gen 3.0). The application allows the user to read sensor manufacturing details, adjust settings, read device telemetry, test basic functionality, and offload measurements. The application is designed to be used for sensor provisioning, diagnostics, and end-of-life data download. The sensors communicate via IrDA, an infrared serial connection standard. A configuration tool is required to translate PC serial communications over IrDA at the different (and non-standard) baud rates supported by the sensors.

The application is written in Python 3.7 using the Qt5 framework.

This document details the underlying architecture and Python modules.

CHAPTER

TWO

USAGE

See the User Manual for details on application usage.

2.1 Building

Several methods for building executables, checking source formatting and building documentation can be performed with `setup.py`. Usage:

```
python setup.py <task>
```

The following tasks are supported:

- `build_executable`: Generates a monolithic executable on the current platform.
- `check_formatting`: Check source for PEP8 formatting errors using `flake8` and format according to `black`.
- `build_sphinx_latexhtml`: Generates this documentation.

2.2 Utilities

2.2.1 Write Hex

The `write_hex.py` utility writes out the provided hex file to sensor EEPROM, 8 bytes at a time.

Usage:

```
python write_hex.py <COM Port> <hex file>
```

Note that the upload may take some time. The communication scheme is limited to 8 bytes per transaction. Uploading an entire hex dump to a Gen 1 or Gen 2 sensor may take 60+ minutes to complete.

2.2.2 Parse Hex

The `parse_hex.py` utility reads in a sensor EEPROM hex dump and scrapes all data from it, including the logged battery voltages. Measurements are exported to CSVs. See *Download Logged Data* in the User Manual for more information on the export. Battery voltages are linearized between the sensor reading timestamps and assumed to have a one hour sampling rate prior to the initial sensor reading.

Usage:

```
python parse_hex.py <hex infile> <csv measurement outfile> <csv voltage outfile>
```

2.2.3 Reprogram

The `reprogram.py` utility provides a collection of actions to prepare a sensor for reprogramming, including clearing the first-time initialization flag, erasing debug and EEPROM data, updating the board version number, and disabling the RTC alarm so the unit power is uninterrupted during reprogramming.

Usage:

```
python reprogram.py <COM Port>
```

2.2.4 RTC Calibration

The `rtc_cal.py` utility provides a method for RTC calibration, particularly during design validation but can also be used to calibrate the RTC based on different silicon batches of crystals or RTCs. First enable the 16.384kHz square wave output and accurately measure the actual frequency. Then, perform calibration using the measured value, which sets the appropriate registers and offsets as defined in the Abracon AB08XX Application Manual (sec 5.9). Finally, disable the square wave output to prevent significant current draw during normal operation.

Usage:

```
python rtc_cal.py <COM Port>
```

CHAPTER

THREE

ARCHITECTURE

3.1 Sensor Module

The `Sensor` module includes functionality for connecting to and interfacing with a humidity logging sensor.

3.1.1 Connection

Connecting to a sensor is a two-step process. First, the `Sensor` must connect to the configuration tool via the `SerialPort` serial manager. Then, the tool must receive a valid response to a continuous series of IR start config bytes:

```
# Create serial port instance, but do not connect yet
port = SerialPort('COM1')
# Create sensor instance, which handles serial interactions
sensor = Sensor(port)
# Connect to the configuration tool and set up for low speed communication
sensor.connect()

# Wait for the sensor to connect, with a timeout
start = time.time()
while not sensor.connected() and time.time() - start < 1.0:
        time.sleep(0.01)

if sensor.connected():
        try:
                # Attempt a connection - a failure will throw a CommandException
                sensor.start_config_mode()
        except CommandException:
                print("Failed to start configuration mode")
```

Upon successfully entering configuration mode, the sensor begins issuing a heartbeat. The heartbeat executes every `HEARTBEAT_PERIOD` seconds and issues a *Read Hardware Version* command, which requires a minimal amount of power as it does not interact with any sensor peripherals.

Exit configuration mode with `exit_config_mode`, which will also terminate the heartbeat timer.

The sensor supports two baud rates: 9600 and 111,111. The speed can be selected with `set_high_speed()`. This will send a command and valid a response to set the device into the selected speed, then adjust the configuration tool speed accordingly. Any commands sent thereafter will be at the selected speed. This mode is recommended for downloading large amounts of data, otherwise it is not necessary.

3.1.2 Executing Commands

Once configured and connected, commands are executed by building and transmitting 13-byte command packets and receiving responses in formats dependent on the command. The vast majority of responses are of the same 13-byte format, while memory offloading commands will respond with payloads of 256 bytes.

The general packet format is a single command byte (`Command` enum), up to 11 payload bytes, and a CRC byte (CRC-8-MAXIM). The payload may contain an address, data, a boolean value, timestamps, etc. The payload is padded with `0xFF` to 11 bytes as needed. The CRC is calculated over the entire packet.

Simple commands are commands which have no payload and had a 13-byte response. This covers many commands involving reading sensor values, blinking LEDs, and more.

Several decorators are included on most commands on a case-by-case basis. The `@blocking` decorator uses a mutex to prevent other commands from executing, with a 1 second timeout that throws a `SensorTimeoutException`. The `@retry` decorator attempts `RETRY_COUNT` retries of the wrapped function if it throws a `CommandException`. Upon failing the final attempt, the exception is raised. The `@reqs_config_mode` decorator simply requires that the sensor currently be in configuration mode before continuing, and will throw a CommandException if it is not.

3.1.3 Memory Fields

The memory is divided into RAM (stored in the real-time clock since it is the only continuously powered IC on the sensor) and EEPROM. The various fields of data are organized in the *eeprom_1_2_fields.json* and *eeprom_3_fields.json* files. The RAM was initially used for storing debug information, but is currently unused as of firmware version 2. Each field contains a name, address, length in number of records, width in bits, and a description. These definitions are loaded when the sensor is initialized, defaulting to the Gen 1/2 fields. When the hardware version is read, the major version determines which EEPROM field definition to reload, which in turn determines how many bytes to download in an EEPROM dump. This runtime initialization of the fields allows for simple changes in organization of the data without affecting the majority of the application structure and for maintaining backwards compatibility.

As data is read from memory, it is stored in an internal representation. Individual fields can then be accessed through the memory `read_field` method. After data is read from EEPROM or RAM, it can be accessed, altered, or exported.

Sensor state and commands.

class sensor.**Command**(*value*)

Command bytes.

CMD_BLINK_GREEN = 205

CMD_BLINK_IR = 206

CMD_BLINK_RED = 204

CMD_EEPROM_OFFLOAD = 212

CMD_END_CONFIG = 187

CMD_FLASHLIGHT = 0

CMD_FLOODLIGHT = 16

CMD_HARD_RESET = 200

CMD_READ_BATTERY = 217

CMD_READ_CURR_READ = 221

CMD_READ_EEPROM_MULT = 193

CMD_READ_EEPROM_SINGLE = 192

CMD_READ_FW_COMMIT = 226

CMD_READ_FW_VERS = 215

CMD_READ_HUMIDITY = 208

CMD_READ_HUM_CAL = 224

CMD_READ_HW_VERS = 216

CMD_READ_RTC_CONFIG = 219

CMD_READ_RTC_MULT = 197

CMD_READ_RTC_SINGLE = 196

CMD_READ_SERIAL = 213

CMD_READ_TEMP = 207

CMD_READ_TEMP_CAL = 222

CMD_READ_TIME = 209

CMD_READ_TIMESTAMP = 210

CMD_READ_TOUCH = 211

CMD_RTC_OFFLOAD = 203

CMD_SET_SERIAL_SPEED = 218

CMD_SET_TIME = 202

CMD_SOFT_RESET = 201

CMD_START_CONFIG = 68

CMD_WRITE_EEPROM_MULT = 195

CMD_WRITE_EEPROM_SINGLE = 194

CMD_WRITE_HUM_CAL = 225

CMD_WRITE_RTC_CONFIG = 220

CMD_WRITE_RTC_MULT = 199

CMD_WRITE_RTC_SINGLE = 198

CMD_WRITE_SERIAL = 214

CMD_WRITE_TEMP_CAL = 223

exception sensor.**CommandException**

Exception for command related issues, such as invalid response.

class sensor.**Sensor**(*serial_port:* serialport.SerialPort)

abort_eeprom_dump() → None

Aborts any currently running EEPROM dump.

Returns None

Return type None

blink_green_led() → None
Blink the green LED for diagnostic testing.

Returns None

Return type None

blink_ir_led() → None
Blink the IR LED For diagnostic testing.

Returns None

Return type None

blink_red_led() → None
Blink the red LED for diagnostic testing.

Returns None

Return type None

cancel_heartbeat() → None
Cancels any currently running heartbeat.

config_mode: bool = False

connect() → None
Start the serial port thread, connecting to the configuration tool. If the SerialPort has already closed, a new one should be provided before connecting.

Returns None

Return type None

connected() → bool
Gets serial connection status of the configuration tool.

Returns True if serial connection established with config tool.

Return type bool

disconnect() → None
Close the serial connection to the configuration tool. The sensor should normally exit configuration mode before disconnection.

The heartbeat is also cancelled.

Returns None

Return type None

disconnected_handler: Optional[Callable[[Exception], None]] = None

dump_eeprom(*callback: Optional[Callable[[int, int], None]] = None*, *entire: bool = True*, *restart: bool = False*) → bytearray
Dump the entire EEPROM contents (512k bytes). An optional callback allows for progress updates (current and total bytes). If the eeprom dump is aborted, all data collected to that point is returned.

If entire is True, the full EEPROM range will be dumped. Otherwise, the number of measurements will be extracted, and the only the range up to the last measurement will be read.

If the dump was previously interrupted (either aborted or communication failed), the dump will continue from the last good address. To download the entire range again, restart should be True.

Regardless of whether the dump was interrupted, aborted, or completed successfully, the resulting bytes are stored in the internal representation for CSV export.

The returned data is from address 0 up to the final successful address.

Parameters

- **callback** (*Optional[Callable[[int, int], None]]*) – Optional callback for progress updates
- **entire** (*bool*) – Read the entire EEPROM range if True
- **restart** (*bool*) – Restart the dump from the beginning if previously interrupted

Returns EEPROM memory contents (up to 524,288 bytes)

Return type bytearray

dump_interrupted: bool = False

dump_last_addr: int = 0

eeprom: memory.SensorEEPROM

exit_config_mode() → None
Exits configuration mode.

Returns None

Return type None

export_eeprom_csv(*filename: str*) → None
Exports EEPROM data to a CSV.

The initial and final timestamps are taken from EEPROM.

Parameters **filename** (*str*) – Full path to export file

Returns None

Return type None

export_eeprom_hex(*filename: str*) → None
Exports EEPROM hex dump. Data should have already been read from sensor with *dump_eeprom()*.

Parameters **filename** (*str*) – Full path to export file

Returns None

Return type None

get_measurement_count() → int
Returns the reading counter stored in EEPROM.

Returns Number of measurement records stored in EEPROM.

Return type int

hard_reset() → None
Perform a hard reset, clearing EEPROM counters and RTC mem and shutting down the device. This is equivalent to a factory reset.

Returns None

Return type None

heartbeat() → None
Confirms the sensor is still connected by sending a simple command up to RETRY_COUNT times.

If the heartbeat fails, the sensor is considered disconnected and is placed out of configuration mode.

Returns None

Return type None

heartbeat_timer: threading.Timer

high_speed: bool = False

port: str = ''

read_battery() → float
Reads the current battery voltage in volts.

Returns Battery voltage in volts

Return type float

read_debug_fields() → None
Dump all fields in EEPROM in the 'Debug' category. This is done by determining the full range of memory covering debug values, reading in that EEPROM and storing the results in the internal representation.

Returns None

Return type None

read_eeprom_byte(*addr: int*) → int
Reads a byte from EEPROM at the specified address.

Parameters **addr** (*int*) – EEPROM address (0 - 524,287)

Returns Value (byte) at address.

Return type int

read_eeprom_bytes(*addr: int*, *num: int = 1*) → bytearray
Reads N bytes from EEPROM starting at the specified address. The first read may not be aligned to 8 bytes, but all subsequent reads are to ensure the last read does not overflow.

Parameters

- **addr** (*int*) – EEPROM starting address (0 - 524,287)
- **num** (*int*) – Number of bytes to read.

Returns N bytes starting at address.

Return type bytearray

read_eeprom_field(*field_name: str*) → bytearray
Reads the specified field (determined by eeprom_fields.json) from EEPROM.

Parameters **field_name** (*str*) – Field name

Returns Data stored in eeprom

Return type bytearray

read_firmware_commit() → str
Reads a max 128-character firmware version control commit string.

Returns Commit string (max 128 char)

Return type str

read_firmware_version() → Tuple[int, int, int]
Reads the sensor firmware version as (major, minor, patch).

Returns Version as major, minor, patch.

Return type Tuple[int, int, int]

read_hardware_version() → Tuple[int, int]
Reads the sensor hardware (PCBA) version as (major, minor).

Returns Version as major, minor.

Return type Tuple[int, int]

read_humidity() → float
Read the current relative humidity.

Returns Relative humidity (0-100%).

Return type float

read_last_timestamp() → Tuple[int, int, int, int, int, int]
Read the latest measurement timestamp from EEPROM.

Returns EEPROM last measurement time in years, months, days, hours, minutes, and seconds.

Return type Tuple[int, int, int, int, int, int]

read_rtc_byte(*addr: int*) → int
Reads a byte from RTC memory at the specified address.

Parameters **addr** (*int*) – RTC memory address (0 - 255)

Returns Value (byte) at address.

Return type int

read_rtc_bytes(*addr: int*, *num: int = 1*) → bytearray
Reads N bytes from RTC memory starting at the specified address. If N exceeds the memory bounds, only data up to the last address will be returned.

Parameters

- **addr** (*int*) – RTC memory starting address (0 - 255)
- **num** (*int*) – Number of bytes to read

Returns N bytes starting at address.

Return type bytearray

read_rtc_config(*addr: int*) → int
Reads an RTC configuration register.

See AB08x5 RTC Family Application Manual for register details.

Parameters **addr** (*int*) – RTC configuration register address (0x00 - 0x3f)

Returns Byte stored in register

Return type int

read_sensor_version() → Tuple[int, int, int]
Reads the sensor hardware version as (major, minor, patch).

EEPROM representation is also adjusted to match the returned hardware version. If the major version is 3, then the size is set to EEPROM_GEN_3_SIZE and the fields are adjusted to match eeprom_3_fields.json. Otherwise, the size is set to EEPROM_GEN_1_2_SIZE and the fields are adjusted to match eeprom_1_2_fields.json.

Returns Version as major, minor, patch.

Return type Tuple[int, int, int]

read_serial() → int
Reads the sensor 24-bit serial number.

Returns Serial number

Return type int

read_temp() → float
Read the current temperature from the humidity sensor.

Returns Temperature in Celsius

Return type float

read_time(*cmd: sensor.Command = <Command.CMD_READ_TIME: 209>*) → Tuple[int, int, int, int, int, int]
Read the current device time.

Returns Current device time in years, months, days, hours, minutes, and seconds.

Return type Tuple[int, int, int, int, int, int]

read_timestamp() → Tuple[int, int, int, int, int, int]
Read the EEPROM starting time.

Returns EEPROM starting time in years, months, days, hours, minutes, and seconds.

Return type Tuple[int, int, int, int, int, int]

read_touch_status() → bool
Read the current touch button status.

Returns True if pressed, False otherwise.

Return type bool

rxqueue: *queue.Queue*

send_flashlight_trigger() → None
Sends the flashlight trigger at a fixed interval for 1 second.

Returns None

Return type None

send_floodlight_trigger() → None
Sends the floodlight trigger at a fixed interval for 1 second.

Returns None

Return type None

serial: *serialport.SerialPort*

serial_exception_handler(*e: Exception*) → None
Handles exceptions arising from the SerialPort thread. If a disconnected handler has been set, it is passed a SensorSerialException.

Parameters **e** (*Exception*) – Exception object

Returns None

Return type None

set_disconnected_handler(*func: Callable[[Exception], None]*) → None
Sets the disconnect handler for the sensor. In the event that the serial connection or the heartbeat is lost, the disconnected handler will be called and the sensor will exit config mode and disconnect. The handler is passed the exception.

Parameters **func** (*Callable[[Exception], None]*) – Callback for serial port or heartbeat failures

Returns None

Return type None

set_high_speed(*on: bool*) → None
Sets the sensor baud rate to low speed (9,600 baud) or high speed (111,111 baud). Verifies the sensor is in the specified mode after switching.

Parameters **on** (*bool*) – High speed if True

Returns None

Return type None

set_serial_port(*serial_port:* serialport.SerialPort) → None
Updates the underlying serial port and queues, but does not connect.

Parameters **serial_port** (*SerialPort*) – New SerialPort reference

Returns None

Return type None

set_time(*year: int*, *month: int*, *day: int*, *hour: int*, *min: int*, *sec: int*) → None
Set the current RTC time.

Parameters

- **year** (*int*) – Year (2000-2099)
- **month** (*int*) – Month (1-12)
- **day** (*int*) – Day (1-31)
- **hour** (*int*) – Hour (0-23)
- **min** (*int*) – Minute (0-59)
- **sec** (*int*) – Second (0-59)

Returns None

Return type None

soft_reset() → None
Perform a soft reset, which resets the bad readings counter.

Returns None

Return type None

start_config_mode() → bool
Puts the device into configuration mode.

The command byte is sent repeatedly for 1 second and a response is checked. If received and valid, the device is placed in config mode.

Otherwise, a check is performed to see if the sensor is already in config mode. To do this, the serial number is requested in low and high speed modes.

Returns True if configuration mode is established

Return type bool

start_heartbeat() → None
Enables transmission of the heartbeat on a periodic threading.Timer.

stop_eeprom_dump: bool = False

txqueue: queue.Queue

write_eeprom_byte(*addr: int*, *data: int*) → None
Write a byte to EEPROM at the specified address.

Parameters

- **addr** (*int*) – EEPROM address (0 - 524,287)
- **data** (*int*) – Byte to write out.

Returns None

Return type None

write_eeprom_bytes(*addr: int*, *data: bytearray*) → None
Write a series of bytes to EEPROM starting at the specified address. The underlying write command always sends 8 bytes, so the starting address and data are required to be multiples of 8.

Parameters

- **addr** (*int*) – EEPROM starting address (0 - 524,280)
- **data** (*bytearray*) – Data to write out
- **num** (*int*) – Number of bytes to write

Returns None

Return type None

write_eeprom_field(*field_name: str*, *data: bytearray*) → None
Writes the specified field (determined by eeprom_fields.json) to EEPROM.

Parameters

- **field_name** (*str*) – Field name
- **data** (*bytearray*) – Data to write out

Returns None

Return type None

write_hardware_version(*major: int*, *minor: int*) → None
Updates the sensor hardware (PCBA) version by writing out to EEPROM.

Parameters

- **major** – Major board version
- **minor** – Minor board version

Returns None

write_rtc_byte(*addr: int*, *data: int*) → None
Write a byte to RTC memory at the specified address.

Parameters

- **addr** (*int*) – RTC memory address (0 - 255)
- **data** (*int*) – Byte to write out.

Returns None

Return type None

write_rtc_bytes(*addr: int*, *data: bytearray*) → None
Write a series of bytes to RTC memory starting at the specified address. The underlying write command always sends 8 bytes, so the starting address and data are required to be multiples of 8.

Parameters

- **addr** (*int*) – RTC RAM starting address (0 - 524,287)
- **data** (*bytearray*) – Data to write out (up to 253 bytes)
- **num** (*int*) – Number of bytes to write

Returns None

Return type None

write_rtc_config(*addr: int*, *data: int*) → None
Writes to an RTC configuration register.

See AB08x5 RTC Family Application Manual for register details. No assumptions are made about the response.

Parameters

- **addr** (*int*) – RTC configuration register address (0x00 - 0x3f)
- **data** (*int*) – Byte to write out

Returns None

Return type None

write_serial(*sn: int*) → int
Sets the sensor serial number.

Parameters **sn** (*int*) – Serial number (20-bit, unsigned)

Returns Serial number written

Return type int

exception sensor.**SensorException**
Generic sensor exception.

exception sensor.**SensorModeException**
Exception for attempting to send command when config mode is not established.

exception sensor.**SensorSerialException**
Exception for serial connection issues.

exception sensor.**SensorTimeoutException**
Exception for sensor timeouts.

sensor.**blocking**(*func: Callable[[...], T]*) → Callable[[...], T]
Makes the wrapping function blocking to prevent multiple commands executing simultaneously.

Parameters **func** (*Callable[.., T]*) – Function to wrap.

Returns Wrapped function.

Return type Callable[.., T]

Raises SensorTimeoutException: Fails to acquire lock in time (sensor busy)

sensor.**create_cmd_packet**(*cmdbyte:* sensor.Command, *payload:* *bytearray* = *bytearray(b'')*) → bytearray

Generates a command packet from a command byte and an optional payload.

If no payload is given, the command byte with no CRC will be returned. If a payload is given, the packet will take the following form:

<CMD (1)><PAYLOAD (10:0)><CRC>

If the payload is greater than 11 bytes, it is truncated. If the payload is less than 11 bytes, it is padded with 0xFF. If a payload is present, a CRC-8 is calculated from the command byte, payload,

and any padded bytes.

Parameters

- **cmdbyte** (Command) – Command byte
- **payload** (*bytearray*) – Data to send with command

Returns Command packet with command byte and optional payload and CRC

Return type bytearray

sensor.**reqs_config_mode**(*func: Callable[[...], T]*) → Callable[[...], T]

Require configuration mode for the decorated function - raises a SensorModeException if config mode has not been established.

Parameters **func** (*Callable[.., T]*) – Function to wrap.

Returns Wrapped function.

Return type Callable[.., T]

Raises SensorModeException: Command cannot execute because sensor is not in config mode.

sensor.**retry**(*func: Callable[[...], T]*) → Callable[[...], T]

Decorator for retry attempts (RETRY_COUNT). If all retries fail, a CommandException is raised.

Parameters **func** (*Callable[.., T]*) – Function to wrap.

Returns Wrapped function.

Return type Callable[.., T]

Raises CommandException: Command failed after exhausting all retries.

sensor.**valid_response**(*packet: bytearray*, *cmd:* sensor.Command, *length: int* = *13*) → bool

Determine whether a packet is the appropriate length, contains the right command, and has a correct CRC.

Parameters

- **packet** (*bytearray*) – Packet to check
- **cmd** (Command) – Expected command
- **length** (*int*) – Expected packet length (include command and CRC)

Returns True if valid

Return type bool

3.2 SerialPort Module

The `SerialPort` module is used to communicate with a serial port in a separate thread. Python queues are used to pass data read and written to the port in a thread-safe manner. `SerialPort` supports changing the IrDA baud rate from standard speed (9,600 baud) to high speed (111,111 baud). A 13-byte command string ("SETHIGHSPEEDx") is transmitted to the sensor, where it is discarded as invalid. The configuration tool will respond by adjusting the IrDA baud rate after transmitting the message. The underlying serial implementation is through the `pyserial` package.

If the serial port is disconnected for any reason, an exception is thrown and the thread closes. The exception can be handled by passing an exception handler function prior to attempting a connection. The `Sensor` class does this when initialized or when the serial port is changed.

Serial port connection and communication.

class serialport.**SerialPort**(*port: str*, *txqueue: queue.Queue = <queue.Queue object>*, *rxqueue: queue.Queue = <queue.Queue object>*, *baud: int = 115200*)

Serial port class.

Provides a threaded serial port manager using read and write queues. Serial port communicates with a configuration tool that supports changing baud rate with special commands.

baud: int **= 0**

close() → None

Stop the thread and end communication.

Returns None

Return type None

connected: bool **= False**

exception: Exception

exception_handler: Optional[Callable[[Exception], None]]

high_speed: bool **= False**

is_open() → bool

Determine the status of the serial connection.

Returns True is serial port is open.

Return type bool

port: str **= ''**

run() → None

Open serial port and send and write data from queues.

Returns None

Return type None

rxqueue: queue.Queue

serial_port: serial.serialwin32.Serial

set_high_speed(*enabled: bool = True*) → None

Reconfigure for high or low speed communication.

Parameters **enabled** (*bool*) – True to enable high speed, False to enable low speed.

Returns None.

Return type None

sig_handler(*signum: signal.Signals*, *frame: frame*) → None
Handle thread signals and stop execution.

Parameters

- **signum** (*signal.Signals*) – Signal type
- **frame** (*FrameType*) – Stack frame summary

Returns None

Return type None

stop: bool = False

txqueue: queue.Queue

serialport.**list_ports**() → List[str]
Lists available serial ports.

Source: https://stackoverflow.com/questions/12090503/listing-available-com-ports-with-python Claims that it is successfully tested on Windows 8.1 x64, Windows 10 x64, Mac OS X 10.9.x / 10.10.x / 10.11.x and Ubuntu 14.04 / 14.10 / 15.04 / 15.10 with both Python 2 and Python 3.

Returns List of available serial port reference strings

Return type List[str]

3.3 Config Tool GUI

The GUI framework used is Qt via the PySide2 package, which is the official Python binding for the Qt5.12+ framework. Each section of the UI is constructed separately. Each command sent to the sensor is executed in its own thread to prevent blocking the UI thread as most commands have the potential to take seconds to respond if retries are necessary.

CHAPTER

FOUR

CHANGELOG

- **v1.0.0: Update implementation for v2 firmware for first UI release.**
  - Remove RAM implementations for v2 firmware architecture
  - Update memory definitions for all sensors
  - Remove IR LED blink command from UI
  - Add sensor active diagnostic
  - Add sensor firmware commit string
  - Add RTC calibration script
  - Add serial and firmware commit to CSV export
  - Add battery voltage export to hex parse utility
  - Update documentation
  - Minor bugfixes
- **v0.1.1: Minor bugfixes**
  - Update config mode entry on hard reset
  - Fix logging messages on config mode entry fallback
- v0.1.0: Initial release for customer review

PYTHON MODULE INDEX

S

INDEX