RasPyre Software Framework Documentation

Release 2.1.2

Jan Frederick Eick, Paul Debus, Guido Morgenthal

Jun 13, 2019

Contents

Contents:

CHAPTER 1

Quickstart

1.1 Requirements

To start using the RasPyre platform, you need the following:

- 1 or more Raspberry Pi Model 3B+ (including power source)
- an SD card with a minimum of 4GB capacity per Raspberry Pi
- at least one USB WiFi adapter using a Ralink RT5370 chip
- 1 or more MPU6050 Accelerometer (mounted on a breakout board, including wiring)

1.2 Software Setup (Raspberry Pi)

For easier installation you can obtain an SD card image based on the Raspbian operating system from https://github.com/msk-buw/raspyre/releases The image is preconfigured with a Linux Real Time Kernel and the RasPyre driver module for MPU6050 accelerometers.

Download the file `raspyre_sdcard_image.zip` to your computer and unpack the ZIP archive.

Write the resulting image file `raspyre_sdcard_image.img` to each SD card.

Note: If you need help with the process, refer to the system-specific guides from the Raspberry Pi foundation:

- Linux
- Mac OS
- Windows

For manual installation and configuration of the RasPyre components: see section *Manual Installation*.

1.3 Hardware Setup

Connect each MPU6050 accelerometer to each Raspberry Pi, that you wish to utilize as a sensor node, according to Fig. 1.1.

If you are using the Connector PiHat, connect the sensor with an 8pin Molex to 4pin JST-ZH-1.50mm cable as shown in Fig. 1.2.

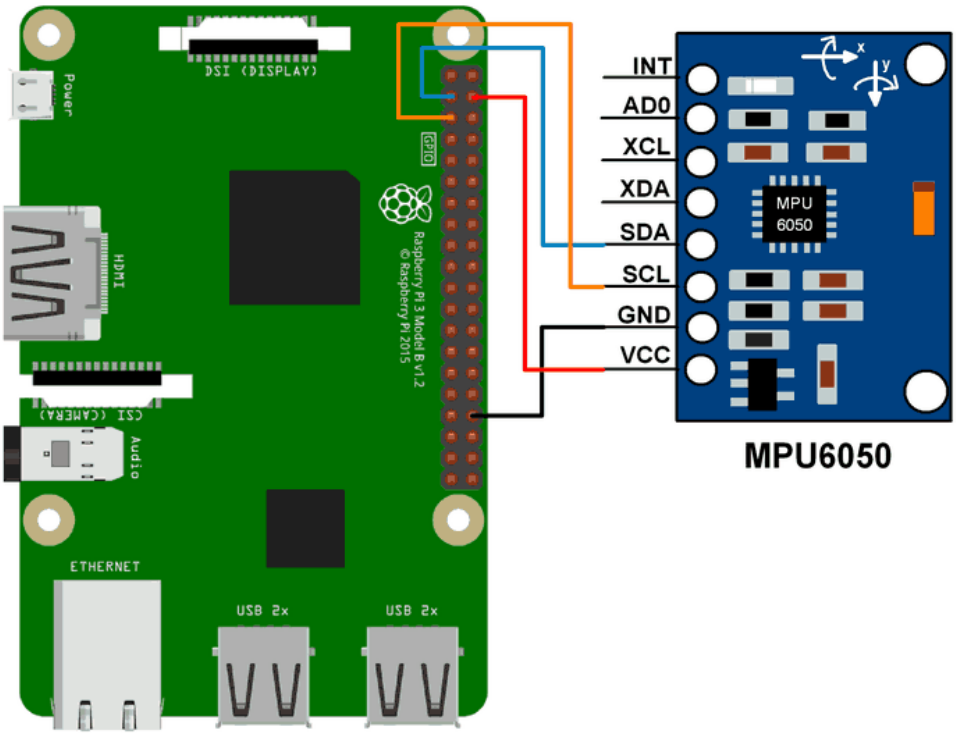

Fig. 1.1: Wiring diagram of the connection between the Raspberry Pi and an MPU6050 breakout board.

1.4 Software Setup (caSHMmere frontend)

1.4.1 Windows

Download the latest `cashmere_frontend_win_setup.exe` from https://github.com/msk-buw/raspyre/releases/. Run the setup wizard and install **caSHMere** to a location of your choice (Fig. 1.3). Finally, start **caSHMere** from the desktop link or from the start menu entry.

1.4.2 Linux

Download the latest `cashmere_frontend_linux.tar.gz` from https://github.com/msk-buw/raspyre/releases/. Extract the archive from the command line and run the extracted `cashmere` binary file.

```
tar xfz cashmere_frontend_linux.tar.gz
cd cashmere
./cashmere
```

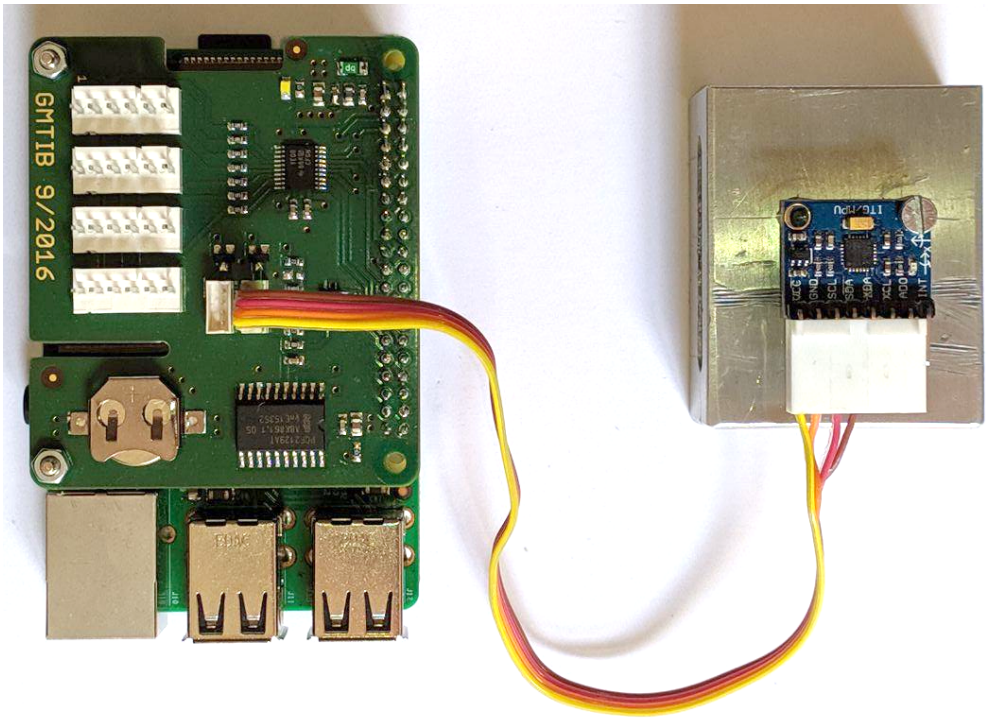

Fig. 1.2: Connection between the Raspberry Pi and an MPU6050 sensor with an 8pin Molex to 4pin JST-ZH-1.50mm cable.

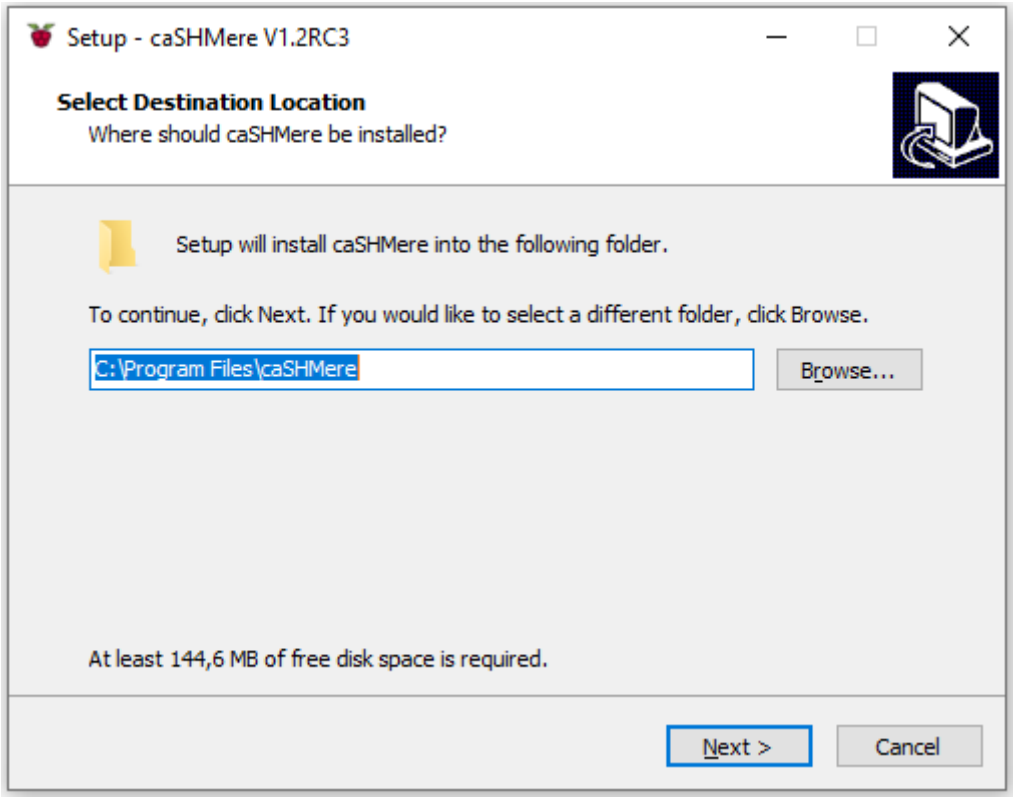

Fig. 1.3: Windows setup wizard to install **caSHMere**.

1.4.3 MacOS

Download the latest `cashmere_frontend_macos.dmg` from https://github.com/msk-buw/raspyre/releases/. Mount the DMG and double click on `caSHMere.app` or drag and drop the program into your Applications folder in Finder.

1.5 First Steps

1.5.1 Setting up the sensor network

Setup the raspyre nodes and supply each with a power source. Choose a portal node that can be reached via WiFi by plugging in the USB network adapter into a free USB port. Note: You can change the portal node at will by unplugging the USB network adapter and plugging it into a different node.

1.5.2 Network mesh

Allow ~30 seconds for the system to set up its network configuration correctly and for the nodes in the mesh network to exchange routing information. Connect your local machine to the open WiFi access point named **"RaspyreAP"**

1.5.3 Frontend software

Start up **caSHMere**. You will be greeted by a splash screen and the main interface is visible (Fig. 1.4).

The main interface is composed of three widgets. On the left is the central control widget to discover nodes in the sensor network and to send commands to individual nodes or groups of nodes. The right widget is dedicated to different widgets to visualize received sensor signals. The bottom area displays logging output during the runtime.

Note: **caSHMere** needs full network permissions to run properly. On Windows you should allow network access when you are presented the firewall prompt (Fig. 1.5).

1.5.4 Node discovery

Nodes are automatically discovered when you are connected to a portal node. Portal nodes are indicated in the list of nodes by a WiFi-signal icon.

Click "Refresh node list" to query the portal node for the routing information of the nodes in the sensor network. The list in the upper left corner should then be populated with reachable nodes and its IP addresses.

Note: Please allow up to 2 minutes for the routing information to be correct if you introduce additional nodes to the network during runtime.

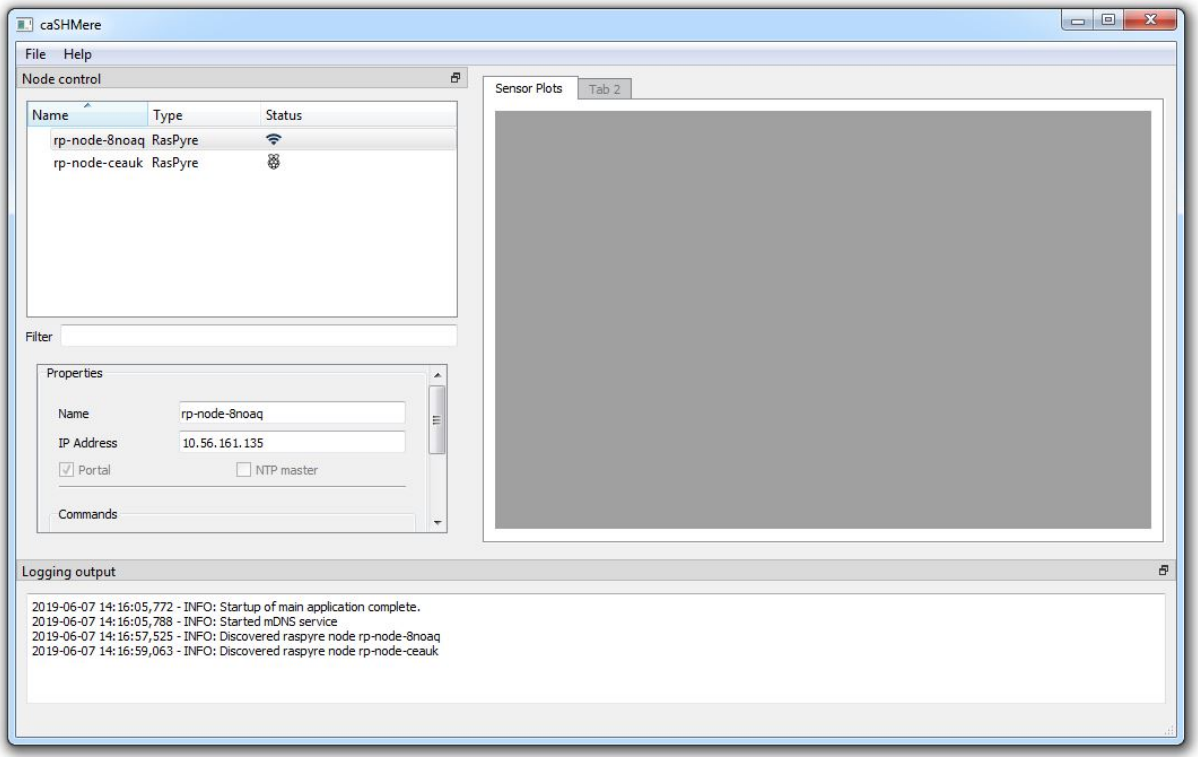

Fig. 1.4: **caSHMere** main interface after startup.

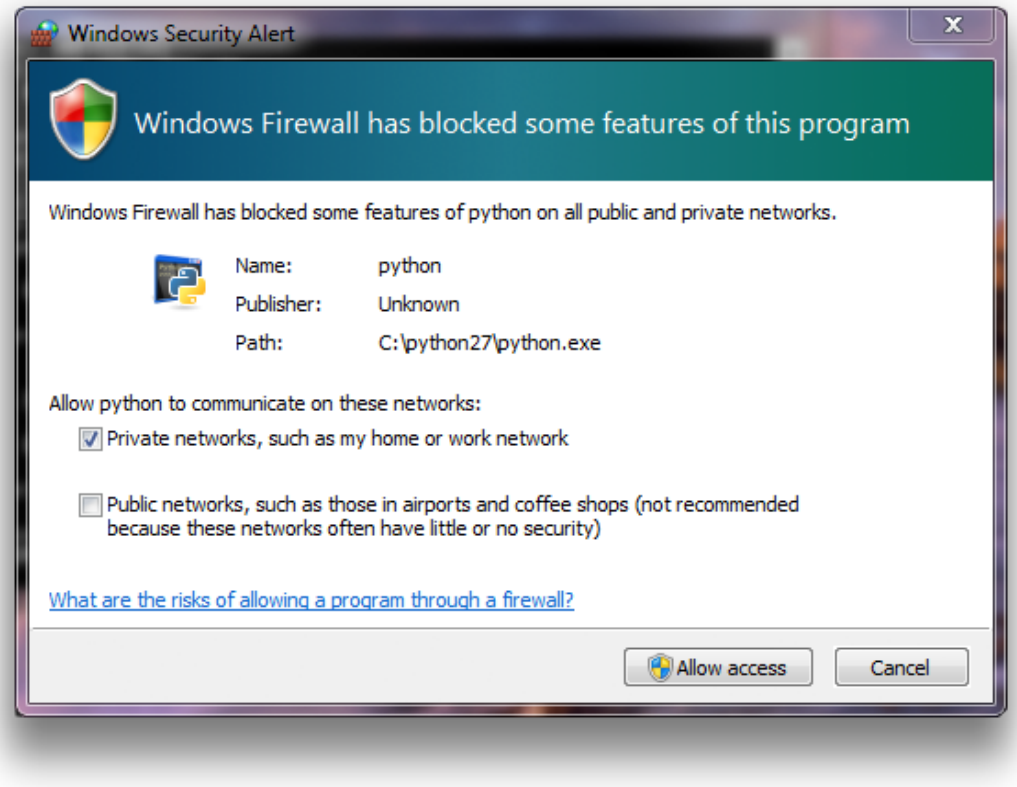

Fig. 1.5: Windows firewall prompt to allow network access of the application.

1.5.5 Sending commands

To send commands to individual nodes select a node from the list by left clicking. Activate the context menu by right clicking. From there you can choose different commands to send to the selected nodes. You can select several nodes as a group by holding down the Shift key. Individual nodes can be added to a selection group by holding down the Ctrl key while left clicking.

1.5.6 Time synchronization

To synchronize the time between the nodes in the mesh network you can individually assign a node the role of the master node and configure the remaining nodes to synchronize relative to this master node. A shortcut for this configuration task has been added to the interface. By selecting `Set reference time` in the context menu, the program tries to set the portal node as a master node and configure the the remaining nodes to use this server for synchronization. The reference time synchronization node is indicated in the node widget list with a clock icon (Fig. 1.6).

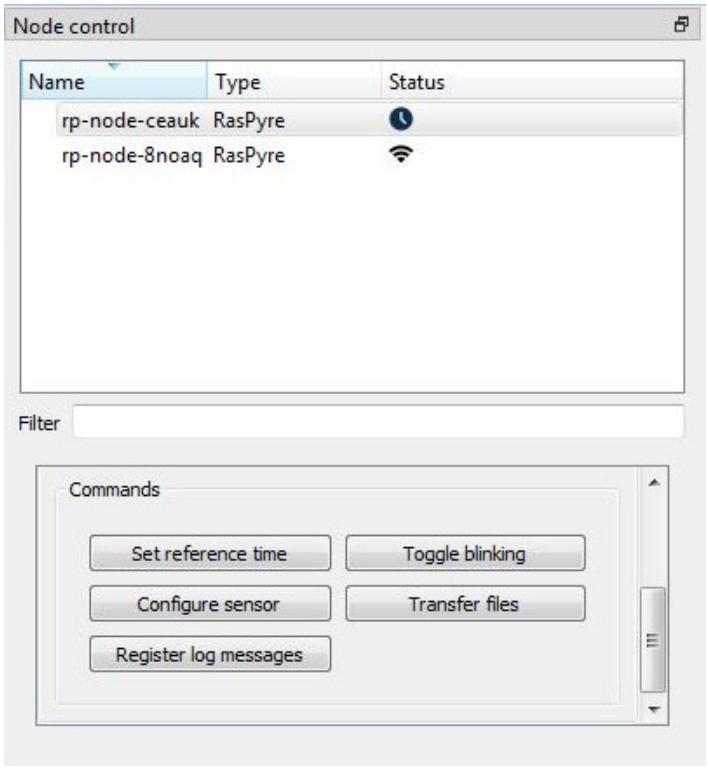

Fig. 1.6: Node list widget displaying node `rp-node-ceauk` as reference time node.

The logging widget provides additional information about the sent commands.

Note: Please allow ~2 minutes for the network to synchronize completely. You can visually inspect the synchronization by sending the command `Toggle blinking`. The selected nodes should blink synchronously after some time.

1.5.7 Configuration of attached sensor hardware

The configuration of installed sensor hardware is performed via the context menu as well. Select the nodes you wish to configure and select `Measurement Control/Add sensor` from the context menu. A dialog window will appear. Please refer to the individual module documentations for the details of configuration (Fig. 1.7)

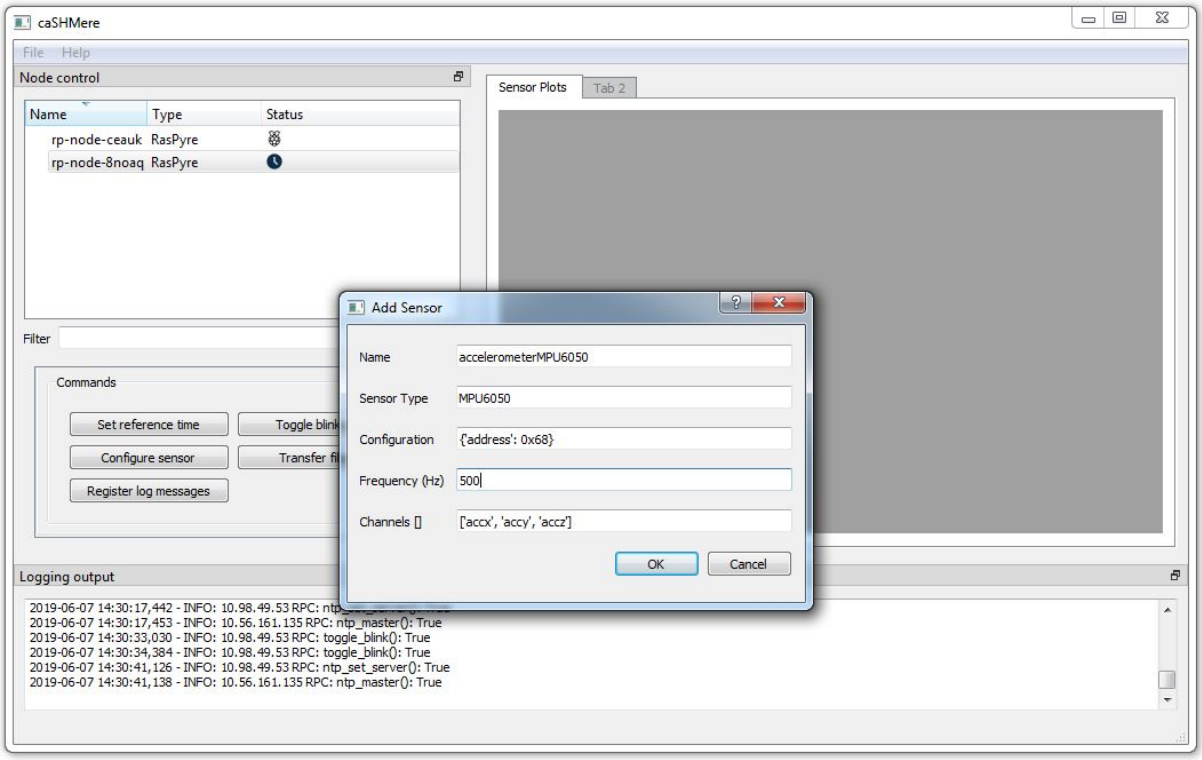

Fig. 1.7: Sensor configuration dialog to add a new `MPU6050` sensor instance.

The dialog is pre-filled with a default configuration for the MPU6050 sensor.

- The name field can be freely chosen for later easier identification of the generated time series records.
- The sensor type selects the installed driver module to use for the specific sensor hardware.
- The configuration field is a serialized JSON string holding the individual sensor parameters. Please refer to the documentation of the sensor module. For the MPU6050 sensor, the only configurable parameter is the address of the I2C cable. You can either select address 0x68 or 0x69 according to the connected cable.
- The frequency field defines the polling frequency for the specified sensor task.
- The channels field consists of a list of channel identifiers which are to be polled during the measurement. Please refer to the documentation of the sensor module for a list of valid channels. For the MPU6050 sensor, the acceleration axes are already selected, denoted by “accx”, “accy”, “accz”.

The newly added sensor will be indicated in the node widget list, below the selected RasPyre node and indicated with a red icon to show that it is not currently measuring (Fig. 1.8).

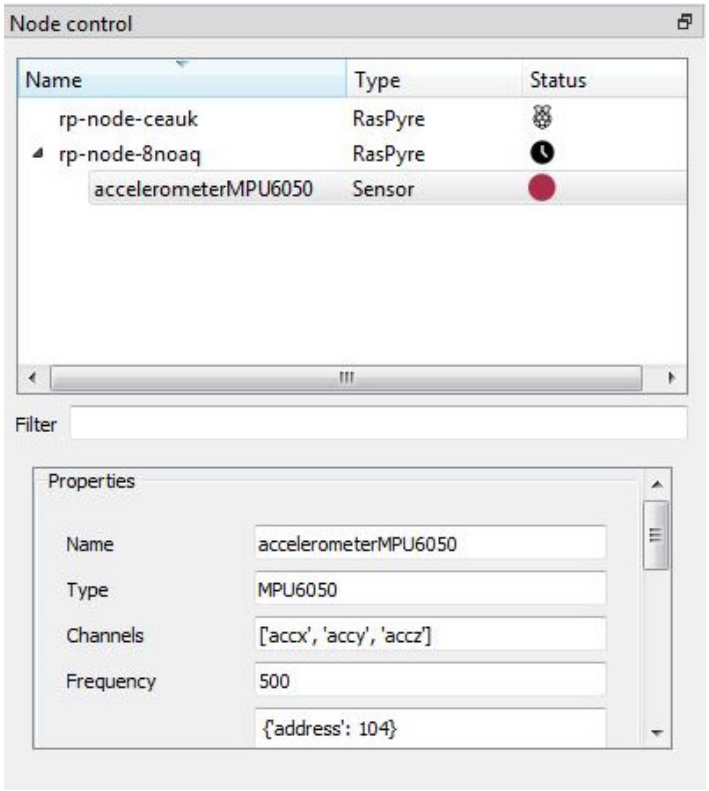

Fig. 1.8: Configured sensor below the RasPyre node it is attached to.

1.5.8 Start and stop a measurement

To start a measurement select nodes which are properly configured for their sensor hardware and select in the context menu `Measurement control/Start measurement`. You will be prompted to provide a name for the measurement (Fig. 1.9). The nodes will start the measurement task and record time series on their local storage. Additionally the sensor signal is published on a network socket.

The node list widget indicates a running measurement with a green light icon next to the configured sensor (Fig. 1.10).

To stop a measurement select `Measurement control/Stop measurement` from the context menu of your selected nodes.

1.5.9 Transfer recorded time series data

To download recorded time series from individual nodes select `File Manager` from the context menu. The node will be queried for its recorded measurement files and they will be displayed in the list of the dialog (Fig. 1.11).

Select the file you wish to transfer and click the Download button. Select the destination where you wish to save the downloaded time series file. If the checkbox "Convert to CSV" is ticked, the transferred binary file will automatically converted to a CSV file. The original binary will be deleted after successful conversion.

Each time series file is named after the following scheme: `hostname__measurementname_sensorname_timestamp.bin`

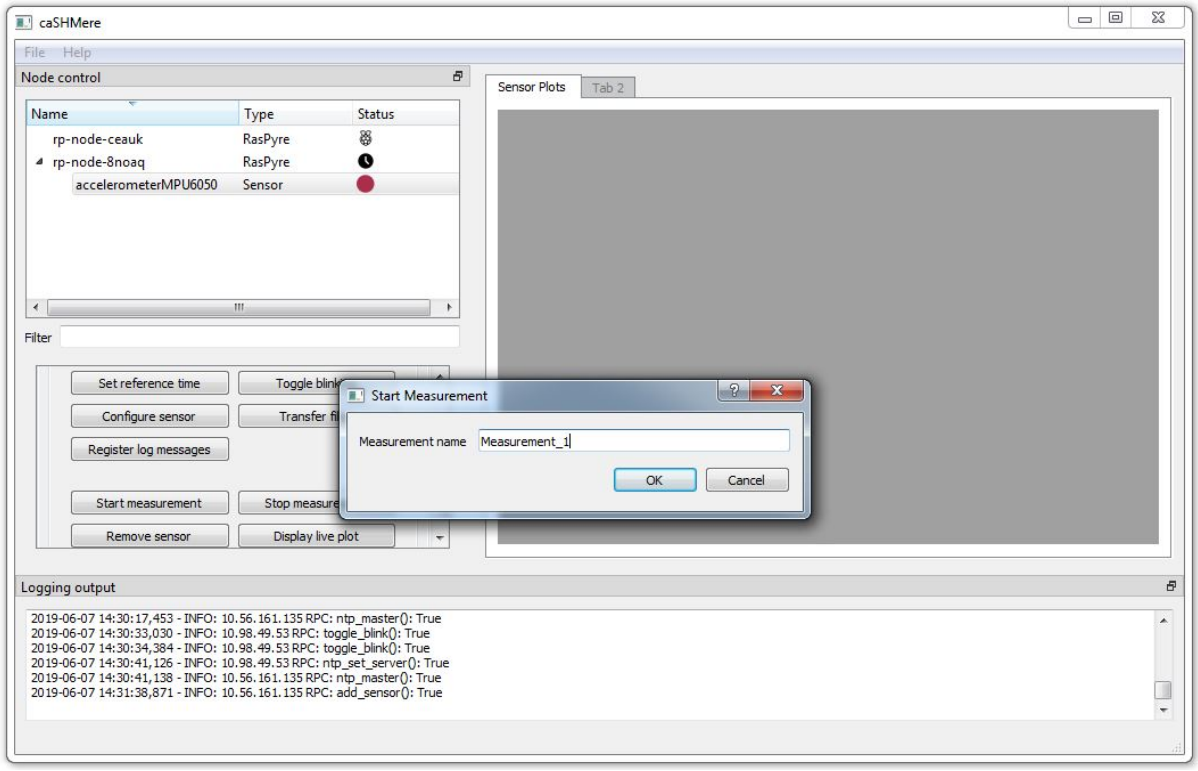

Fig. 1.9: Dialog to start a measurement for the selected nodes.

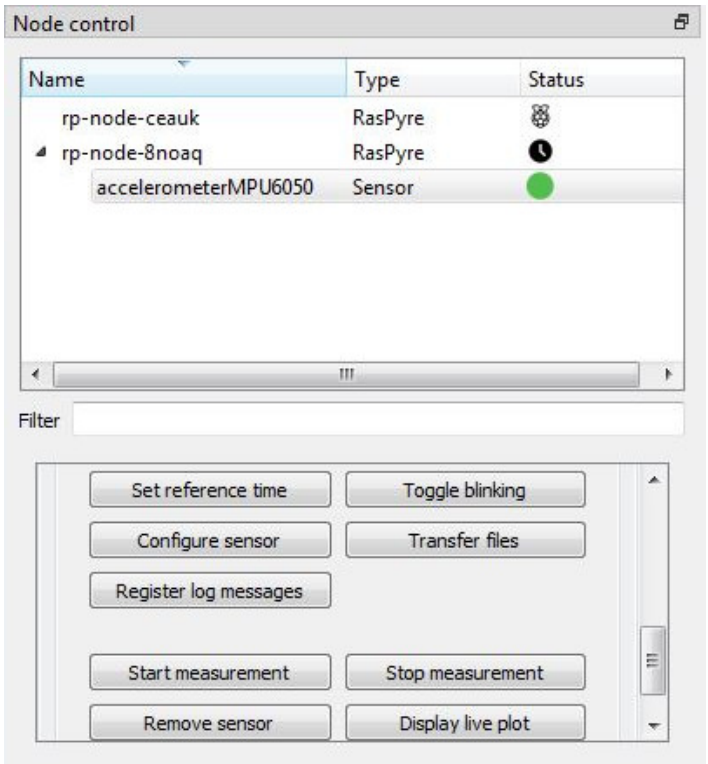

Fig. 1.10: Sensor node indicator for a running measurement.

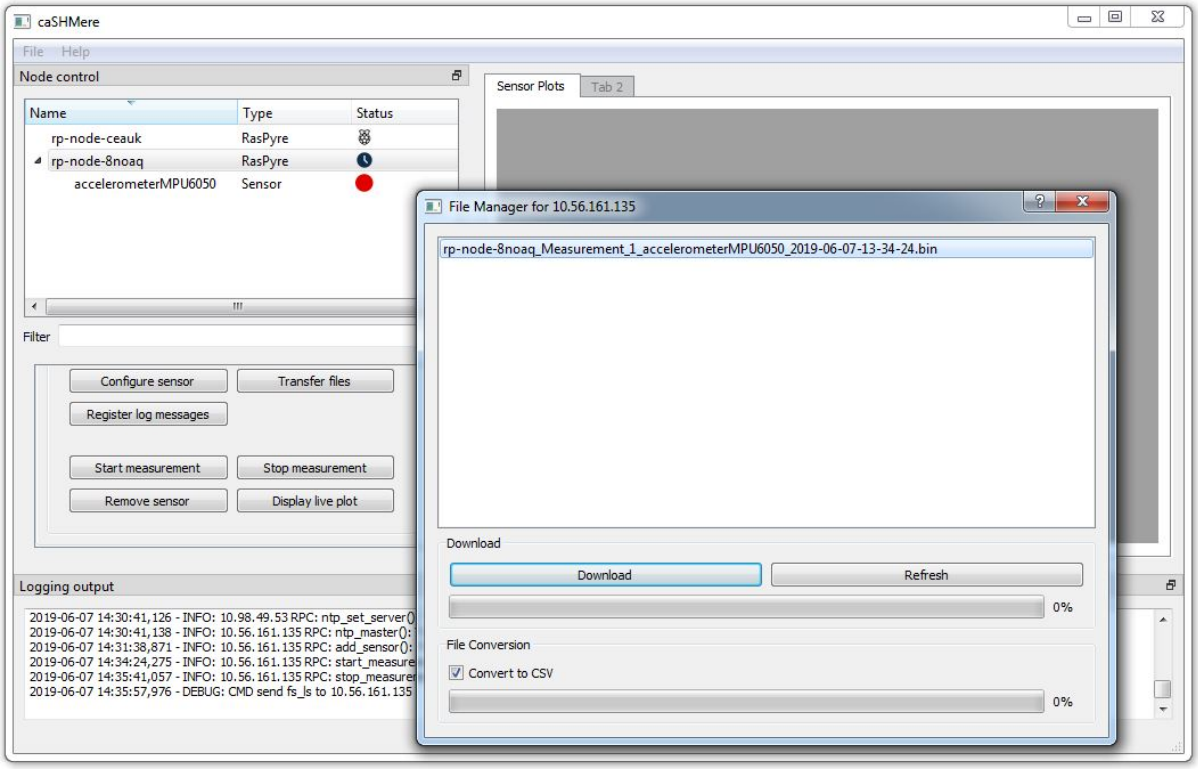

Fig. 1.11: File transfer dialog showing the stored measurement files on the RasPyre node.

1.5.10 Plot signal during measurement

During a running measurement the acquired signal data can by visualized live by utilizing the plot widgets.

Click the button labeled `Create plot widget`. A dialog will appear to configure to which signal to subscribe.

The fields a pre-filled with default information for the case you wish to subscribe to a signal from a MPU6050 sensor (Fig. 1.12).

- The Address field denotes the ZMQ-network address to which you wish to subscribe. Each measurement publishes its signal on the port 5556. Enter the information in the following form: `tcp://IP_ADDRESS:5556`
- The channels field consists of a list of the channels in the acquired network packets. Please refer to the documentation of the individual sensor module for specifics.
- The datatypes field string denotes the datatypes of the channels. In the given example, the individual channels are decoded as double datatype.
- The units field consists of a list of the units for each individual channel. In the example, the time channel is interpreted as a 64bit datetime timestamp.

After successful time synchronization and configuration a new plot widget will appear in the right area of the interface (Fig. 1.13).

By grabbing the right edge of the plot window with the left mouse button, you can drag the FFT plotting area into the plot. Tick the checkbox `Calculate FFT` to calculate a Fast Fourier Transform for the selected signal and visualize it (Fig. 1.14)

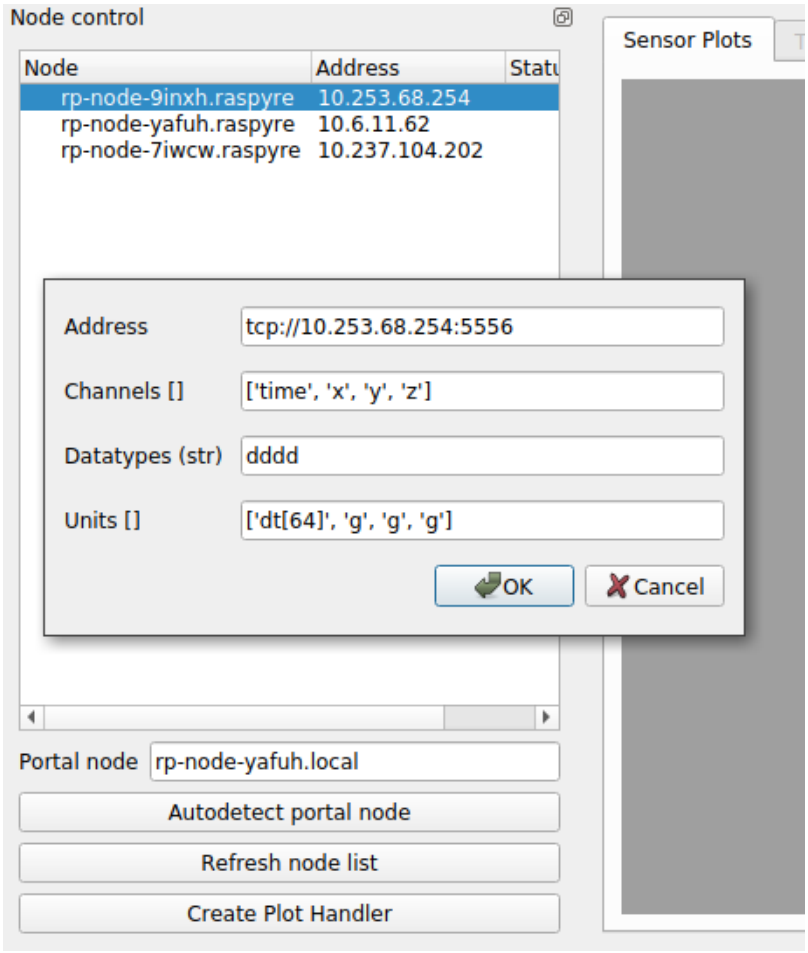

Fig. 1.12: Plot creation dialog.

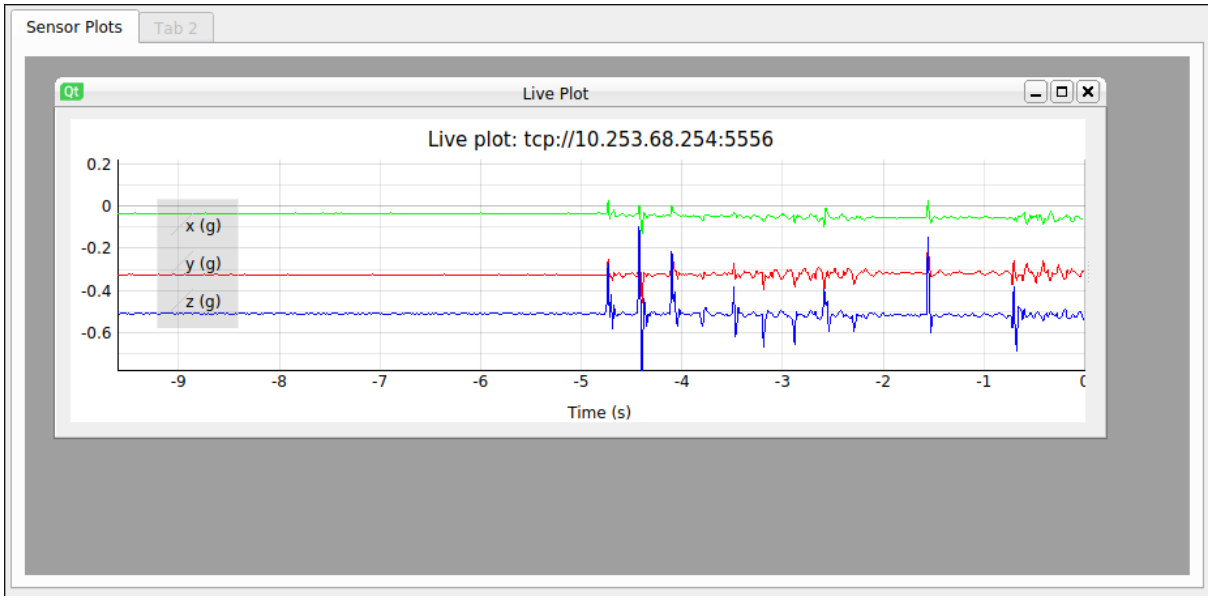

Fig. 1.13: Newly created plot widget in MDI area.

You can utilize the left mouse button in the plot window to drag the signal along the axes and the right mouse button to adjust the scaling of the plot area. If you wish to stop plotting the signal just close the sub window inside the right area of the interface.

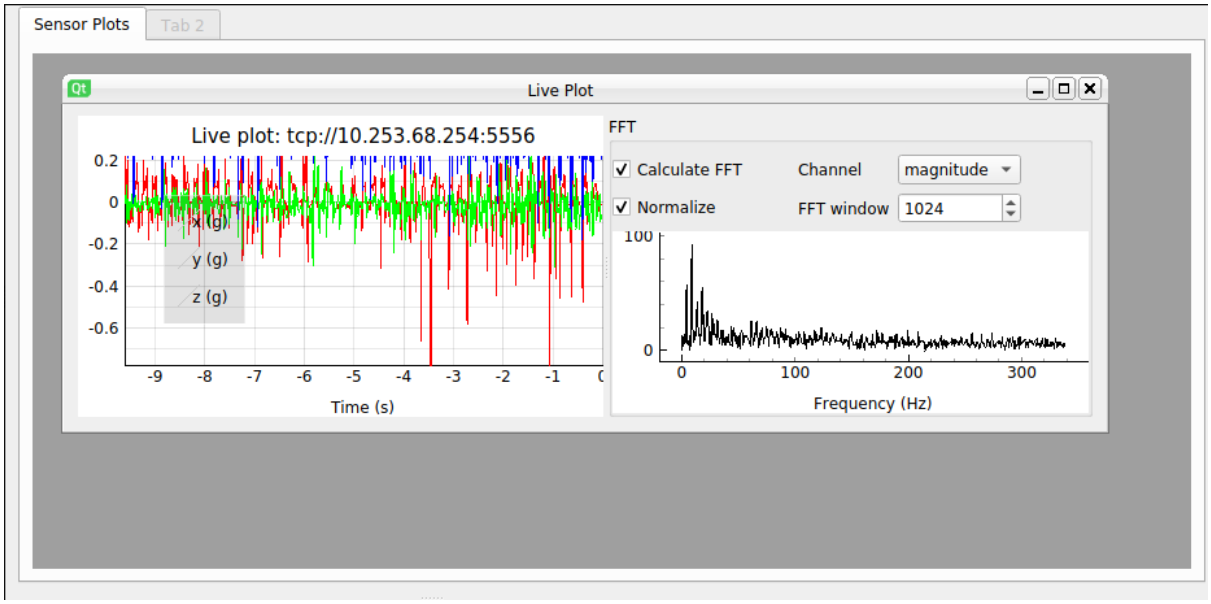

Fig. 1.14: Plot widget displaying additional FFT plot.

CHAPTER 2

Manual Installation

If you decide to configure your RasPyre nodes manually you need to perform several steps.

Note: Please ensure that you are running a Linux distribution with a configured real time kernel.

2.1 Software Installation

Install a suitable Python interpreter (3.5+).

On a Debian based system:

```
# apt-get install python3 python3-dev python3-pip
```

Install the OLSRd routing daemon:

```
# apt-get install olsrd
```

Install the RasPyre python package:

```
# python3 -m pip install raspyre
```

Install one or more sensor drivers (e.g. the `raspyre-mpu6050` package):

```
# python3 -m pip install raspyre-mpu6050
```

2.2 Network configuration

Configure your network adapters, so that your meshing interface is identified by `mesh0`. Configure the `udev` subsystem, that the WiFi adapter that should provide the portal access point is renamed to `ap0`.

Start up the OLSR daemon. A sample configuration file can be found the in the `conf` directory of the RasPyre distribution.

2.3 RPC daemon configuration

Configure the RasPyre RPC service to start up as a daemon. A systemd unit file is provided in the `conf` directory:

```
[Unit]
Description=Raspyre RPC Server Backend
After=network.target

[Service]
WorkingDirectory=/home/pi
ExecStart=/usr/local/bin/raspyre-rpcserver --logfile /home/pi/raspyre-rpc.
↪log /home/pi/data/ --verbose
User=root
LimitRTPRIO=90
LimitRTTIME=infinity

[Install]
WantedBy=multi-user.target
```

Note: Ensure that the process is run with proper rights to request real time priority CPU scheduling up to priority 90.

CHAPTER 3

raspyre

3.1 raspyre package

3.1.1 Subpackages

raspyre.rpc package

Submodules

raspyre.rpc.blink module

class raspyre.rpc.blink.**BlinkProcess**
Bases: multiprocessing.context.Process

run()
Method to be run in sub-process; can be overridden in sub-class

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

raspyre.rpc.functions module

This module implements the functions exposed by the Rapsyre RPC interface.

The functions are grouped in classes and should be invoked into different prefix namespaces. e.g.: class MeasurementHandler consists of functions to handle the Measurement Processes of the Pi's GPIO interface and should reside in a namespace that identifies this behaviour as such.

class raspyre.rpc.functions.**IPContextFilter**
Bases: logging.Filter

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

exception raspyre.rpc.functions.**RaspyreDirectoryInvalid**
Bases: Exception

exception raspyre.rpc.functions.**RaspyreDirectoryNotFound**
Bases: Exception

exception raspyre.rpc.functions.**RaspyreFileInvalid**
Bases: Exception

class raspyre.rpc.functions.**RaspyreService**(*data_directory*, *configuration_directory*)
Bases: object

PROCESS_TIMEOUT = 3

add_sensor(*sensorname*, *sensortype*, *config*, *frequency*, *axis*)
This function adds a sensor to the current setup. Each installed raspyre-sensor-driver package can be used to instantiate a sensor for measurement usage (e.g. raspyre-mpu6050, raspyre-ads1115) Example call:

```
>>> add_sensor(sensorname="S1_left_bridge", sensor_type="MPU6050",
               config={address=0x69},
               frequency=100, axis=['accx', 'accy', 'accz'])
```

Parameters

- **sensorname** – Unique String to identify sensor
- **sensortype** – String specifying the sensor driver package
- **config** – Dictionary of sensor configuration data. The dictionary keys are passed to the initialization method of the specified sensor driver package
- **frequency** – Polling frequency for the measurement
- **axis** – List of parameters to be polled from the sensor

Returns True

Return type Boolean

clear_sensors()
This function removes all configured sensors from the current setup.

Returns True

Return type Boolean

configuration_restore(*sensorname*, *path*)
This function restores a sensor from a given configuration file.

Parameters

- **sensorname** – Unique String identifying the sensor
- **path** – File path relative to the configuration_directory

Returns True

Return type Boolean

configuration_save(*sensorname*, *path*)
This function saves the configuration state of a sensor.

Parameters

- **sensorname** – String of sensor name
- **path** – Path relative to the configuration directory

Returns True

Return type Boolean

debug_log_msg()

fs_ls(*path='.'*)
This function lists the contents of the data storage directory. It returns a list of 2 lists. The first list contains directories of the queried path, the second list contains the file names.

Parameters **path** – path to be queried relative to the data directory

Returns list of 2 lists with [[directories], [files]]

Return type list of lists

fs_mkdir(*path*)
This function creates a directory in the specified path below the data storage directory.

Parameters **path** – Path relative to the data directory

Returns True

Return type Boolean

fs_mv(*src*, *dst*)
This function renames src to dst. If dst is a directory an error will be raised. If dst is a file, it will be silently replaced.

Parameters

- **src** – Path relative to the data directory
- **dst** – Path relative to the data directory

Returns True

Return type Boolean

fs_rm(*path*)
This function removes the specified file from the file system.

Parameters **path** – Path relative to the data directory

Returns True

Return type Boolean

fs_rmdir(*path*, *recursive=False*)
This function removes a directory relative to the data storage

Parameters

- **path** – Path relative to the data directory - not the data directory itself

- **recursive** – Boolean flag indicating recursive deletion

Returns True

Return type Boolean

fs_stat(*path*)
This function returns the POSIX information of a stat system call. Please refer to `stat()`

Parameters **path** – Path relative to the data directory

Returns True

Return type Boolean

get_dns_info()

get_extra(*extra*)
This function is reserved for future usage

Parameters **extra** – Dictionary

Returns True

Return type Boolean

get_info()
This function returns the internal sensor dictionary.

Returns Dictionary of current configuration

Return type Dictionary

get_network_nodes()
FIXME: This function is not implemented

Returns empty List

Return type List

get_status()

get_system_date()
This function returns a string representation of the current system time.

Returns String of current datetime

Return type String

is_measuring(*sensorname*)
This function returns True if the specified sensor is currently used by a measurement process.

Parameters **sensorname** – String of sensor name

Returns True|False

Return type Boolean

list_files()
DEPRECATED! This function lists the filenames in the data directory. ATTENTION: This function is DEPRECATED and will be removed in a later version. Please use fs_ls()

Returns List of file names

Return type List of Strings

ntp_master()

ntp_set_server(*ip_str*)

ntp_sync()

ping()
This function simply returns True. It is used as simple connectivity checking function.

Returns True

Return type Boolean

remove_sensor(*sensorname*)
Removes the sensor specified by its name from the current setup.

Parameters **sensorname** – String identifying sensor

Returns True

Return type Boolean

set_extra(*extra={}*)
This function is reserved for future usage.

Parameters **extra** – Dictionary

Returns True

Return type Boolean

set_network_logger(*host*, *loglevel=10*)

set_system_date(*date*)
This function sets the current system date. NOTICE: This function does not modify any modified realtime clock!

Parameters **date** – String of date. The parameter is passed to the system's date operation thus accepts its format strings. Please refer to the Linux manual page date(1).

Returns True

Return type Boolean

start_blink()

start_measurement(*measurementname*, *sensornames=None*)
This function starts a measurement process for the specified sensors.

Parameters

- **measurementname** – String describing the measurement
- **sensornames** – None [all sensors], String [one specific sensor], List of Strings (optional)

Returns True

Return type Boolean

start_ntp()

stop_blink()

stop_measurement(*sensornames=None*)
This function stops a currently running measurement.

Parameters **sensornames** – None [all sensors], String [one specific sensor], List of Strings (optional)

Returns True

Return type Boolean

stop_ntp()

toggle_blink()

update_sensor(*sensorname*, *config*)
FIXME: This function updates the configuration of a given sensor.

Parameters

- **sensorname** – String identifying the sensor
- **config** – Dictionary of changes configuration parameters. Each key value pair is passed to the sensor's updateConfiguration()

Returns True

Return type Boolean

raspyre.rpc.functions.**get_ip_address**(*ifname*)

raspyre.rpc.handler module

class raspyre.rpc.handler.**HandlerProcess**(*sensor*, *sensor_name*, *config*, *frequency*, *axis*, *mmap_file*, *buffer_size*, *data_dir*, *csv=False*, *chunked=False*, *chunk_minutes=10*)
Bases: multiprocessing.context.Process

run()
Method to be run in sub-process; can be overridden in sub-class

setMeasurementName(*measurement_name*)

shutdown()

raspyre.rpc.mplog module

class raspyre.rpc.mplog.**MultiProcessingLog**(*name*, *mode*, *maxsize*, *rotate*)
Bases: logging.Handler

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

receive()

send(*s*)

setFormatter(*fmt*)
Set the formatter for this handler.

raspyre.rpc.pollingprocess module

class raspyre.rpc.pollingprocess.**PollingProcess**(*sensor*, *sensor_name*, *config*, *frequency*, *axis*, *data_dir*, *mmap_file*, *buffer_size*, *chunked=False*, *chunk_minutes=10*)
Bases: multiprocessing.context.Process

PROCESS_PRIORITY = 90

run()
Method to be run in sub-process; can be overridden in sub-class

setMeasurementName(*measurement_name*)

shutdown()

class raspyre.rpc.pollingprocess.**Sched_Param**
Bases: _ctypes.Structure

sched_priority
Structure/Union member

class raspyre.rpc.pollingprocess.**Timespec**
Bases: _ctypes.Structure

tv_nsec
Structure/Union member

tv_sec
Structure/Union member

raspyre.rpc.server module

Raspyre-RPCServer

This module is used to create a XMLRPC-Server for the Raspyre SHM platform.

class raspyre.rpc.server.**RequestHandler**(**args*, *directory=None*, ***kwargs*)
Bases: xmlrpc.server.SimpleXMLRPCRequestHandler, http.server.SimpleHTTPRequestHandler

do_GET()
Serve a GET request.

rpc_paths = ('/RPC2', '/')

send_head()
Common code for GET and HEAD commands.

This sends the response code and MIME headers.

Return value is either a file object (which has to be copied to the outputfile by the caller unless the command was HEAD, and must be closed by the caller under all circumstances), or None, in which case the caller has nothing further to do.

server_version = 'RaspyreRPC/0.4'

class raspyre.rpc.server.**ThreadedXMLRPCServer**(*addr*, *requestHandler=<class 'xmlrpc.server.SimpleXMLRPCRequestHandler'>*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)
Bases: xmlrpc.server.SimpleXMLRPCServer, socketserver.ThreadingMixIn

class raspyre.rpc.server.**VerboseFaultXMLRPCServer**(*addr*, *requestHandler=<class 'xmlrpc.server.SimpleXMLRPCRequestHandler'>*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)
Bases: xmlrpc.server.SimpleXMLRPCServer

raspyre.rpc.server.**handle_exception**(*exc_type*, *exc_value*, *exc_traceback*)

raspyre.rpc.server.**main**()

raspyre.rpc.server.**run_rpc_server**(*datadir*, *address='0.0.0.0'*, *port=8000*, *logfile=None*, *configdir=None*, *verbose=False*)

raspyre.rpc.writer module

raspyre.rpc.writer.**generate_binary_header**(*date_float*, *metadata*, *fmt*, *units*, *column_names*)

raspyre.sensors package

Submodules

raspyre.sensors.mockup module

A mockup sensor to test framework functionality. It returns random values. The only configurable parameter is "sps" that defines the samples per second the sensor produces. It implements the Sensor interface provided by the raspyre framework and can be used as a reference.

class raspyre.sensors.mockup.**Mockup**(*sps*)
Bases: *raspyre.sensor.Sensor*

getAttributes()

getConfig()
Returns a dictionary of all configuration parameters the sensor has with their values.

getRecord(**args*)
Returns a Record object containing the requested values. The Parameters to the function specify the attributes that will be measured.

sensor_attributes = {'x': ('g', 'd'), 'y': ('g', 'd'), 'z': ('g', 'd')}

updateConfig(***kwargs*)
Pass a list of parameter names and values. The parameters of the sensor will be changed accordingly

raspyre.sensors.mockup.**build**(***kwargs*)

3.1.2 Submodules

3.1.3 raspyre.converter module

Raspyre Converter CSV Version: 0.1 Binary Version: 0.4

This tool converts Raspyre files between CSV format and binary format. If the input file is binary, the output will be CSV and vice versa. If the output file parameter is omitted, the input name will be used with a new file ending .csv or .bin The tool does support converting entire folders if the input file is a folder containing only valid measurement files.

exception raspyre.converter.**RaspyreReaderException**
Bases: Exception

raspyre.converter.**convert_binary_to_csv**(*source*, *target*)

raspyre.converter.**convert_csv_to_binary**(*source*, *target*)

raspyre.converter.**main**()

3.1.4 raspyre.helpers module

Some helper functions for frequent tasks: write Records to file create Pandas Dataframe from Records (not jet tested) change the name of the wifi the Pi hosts (only if the pi is configured accordingly)

raspyre.helpers.**changeWifiName**(*name*, *path='/home/pi/wifi.conf'*, *reboot=False*)
This function changes the SSID the Wifi created by the Pi uses. This SSID is in a config file. The default is the file wifi.conf in the home directory For the changes to take effect, the system has to be rebooted. This can be done with the reboot option.

raspyre.helpers.**rec2DF**(*records*)
Convert a list of Records into a Pandas DataFrame with column headers according to the attributes of the first object. NOT TESTED. Only works if all records have the same attribtues

raspyre.helpers.**rec2File**(*file*, *records*, *append=True*, *header=True*, *delimiter=', '*)
This function writes a list of Records into a file. You can choose whether to append or override the file if it allready exists and decide, if you want the attribute names asheader displayed. Also you can specify a column delimiter that defaults to ", ". The header is proceeded with a # sign to be marked as not data. The first column is the timestamp, after that the columns are ordered alphabetically

3.1.5 raspyre.record module

The Record object stores an arbitrary number of measured values. All records have the "time" attribute in common, that has the time of creation of the object in system time. Record objects can be sorted by this property. Element acces is done as in dictionarys.

class raspyre.record.**Record**(*values=None*)
Bases: object

add(*key*, *value*)

get(*key*)

3.1.6 raspyre.sensor module

class raspyre.sensor.**Sensor**
Bases: object

Base class for all concrete sensor implementations.

The derived classes have to call the base class constructor at the beginning of their own constructor. This is minimal interface the derived classes have to implement to comply with the frameworks requirements. Further expansion of the interface may be used to add functionality for special uses, but may not be used for the basic functionality.

Note: Each derived class must provide a class variable *sensor_attributes* that maps attribute strings to a datatype format character.

Example: A class for a sensor with attributes for 2 acceleration axes of type double and one temperature field of type integer would need to provide the *sensor_attributes* as follows:

```
sensor_attributes = { 'acceleration_1': 'd', 'acceleration_2': 'd',
↪'temperature': 'i' }
```

See also:

Format Characters

---

getAttributes()

getConfig()
Returns a dictionary of all configuration parameters the sensor has with their values.

getRecord(**args*)
Returns a Record object containing the requested values. The Parameters to the function specify the attributes that will be measured.

sensor_attributes = {}

struct_fmt(*attributes*)
Returns a struct format string representing the datatypes of the attributes parameter.

Parameters **attributes** (*list of strings*) – a list of a subset of the sensor's *sensor_attributes*

Returns a lift of format characters

Return type list of characters

Example

```
>>> from raspyre.sensors.mockup.mockup import Mockup
>>> sensor = Mockup(sps=100)
>>> sensor.struct_fmt(['y', 'x'])
['d', 'd']
```

See also:

Format Strings

units(*attributes*)

updateConfig(***kwargs*)
Pass a list of parameter names and values. The parameters of the sensor will be changed accordingly

3.1.7 raspyre.sensorbuilder module

This module provides the functions to create a sensor from a kwargs parameter.

raspyre.sensorbuilder.**createSensor**(*sensor_type*, ***kwargs*)

3.1.8 raspyre.storage module

This module provides the functionality to read and write data files. Supported for reading are all (including old) binary and csv formats, whereas for writing, we only support the newest binary and csv formats.

Also the functionality to resample data is included. Goal is to reduce the size of the stored data for longterm measurements. The following levels are available and can be derived from the file extension:

Level Interval Blocksize

rm01 sampling rate variable rm02 1 second 1 hour rm03 1 second 1 day rm04 1 minute 1 day rm05 1 minute 1 week rm06 1 minute 1 month rm07 1 hour 1 month

The filename consists of a descriptive name, the timestamp for the beginning of the file and the file extension that specifies the level of sampling. The file contains a timestamp as the first column followed by a number of value columns. The firs line of the file contains the name of the columns.

class raspyre.storage.**BinReader**(*filename*)
Bases: *raspyre.storage.Reader*

data()

parseHeader()

class raspyre.storage.**CSVReader**(*filename*)
Bases: *raspyre.storage.Reader*

data()
returns a generator for all the data rows in the csv file. :returns: tuple – the parsed values from the data row

parseHeader()
parses a CSV header, assuming all lines starting with #<space> currently supported versions: 0.1

exception raspyre.storage.**RaspyreFileFormatException**
Bases: Exception

class raspyre.storage.**Reader**(*filename*)
Bases: object

class raspyre.storage.**Writer**(*filename*, *binary=True*)
Bases: object

close()

writeHeader(*header*)

writeRow(*row*)

raspyre.storage.**build_binary_header**(*date_float*, *metadata*, *fmt*, *units*, *column_names*)

raspyre.storage.**build_csv_header**(*date*, *metadata*, *fmt*, *units*, *column_names*)
build csv header for the file writer version 0.2

raspyre.storage.**cleanCSVLine**(*line*)

cleans a CSV header line from leading # and whitespaces and trailing

and whitespaces used for header parsing :param line: string to be cleaned

raspyre.storage.**getReader**(*filename*)

raspyre.storage.**process_files**(*in_folder*, *out_folder*, *level*)
This function processes all the files in the in_folder such that it resamples them to the required sampling rate and stores them in the according blocksize in new files in the out_foler. It has the ability to see, which data has already been resampled, so that it can be called multiple times with the same arguments and only updates the new data.

Python Module Index

r

Index

H

I

L

M

N

P

R

S

T

U

V

W