
datalogd

Release 0.2.3

Jan 03, 2023

1	User Documentation	3
1.1	Quick Start	3
1.1.1	Installation	3
1.1.1.1	Windows	4
1.1.2	Configuration	4
1.1.2.1	Configuration Files	4
1.1.2.2	Command Line Parameters	5
1.2	Plugins	5
1.2.1	Data Sources	5
1.2.2	Data Sinks	6
1.2.3	Data Filters	6
1.2.4	Connecting	7
1.2.5	Running	7
1.3	Connection Graph	7
1.3.1	Node Attributes	8
1.4	Data Logging Recipes	9
1.4.1	Random Data to CSV File	9
1.4.2	System Temperatures to Matplotlib Plot	11
1.4.3	Temperatures to InfluxDB with Grafana Visualisation on Raspberry Pi	12
1.4.3.1	Hardware Setup	13
1.4.3.2	Software Setup	13
1.4.3.2.1	InfluxDB	13
1.4.3.2.2	Grafana	13
1.4.3.2.3	Recipe	14
1.4.3.2.4	Visualisation	14
1.4.4	Arduino Temperatures to InfluxDB and Grafana Display on Windows	15
1.4.4.1	Hardware	16
1.4.4.2	Recipe	21
1.4.4.3	Software	21
2	API Documentation	23
2.1	datalogd package	23
2.1.1	Subpackages	25
2.1.1.1	datalogd.plugins package	25
2.1.1.1.1	Submodules	25
3	Indices and tables	41

Python Module Index	43
Index	45

datalogd is a data logging daemon service which uses a source/filter/sink plugin architecture to allow extensive customisation and maximum flexibility. There are no strict specifications or requirements for data types, but typical examples would be readings from environmental sensors such as temperature, humidity, voltage or the like.

The user guide and API documentation can be read online at [Read the Docs](#). Source code is available on [GitLab](#).

Custom data sources, filters, or sinks can be created simply by extending an existing *DataSource*, *DataFilter*, or *DataSink* python class and placing it in a plugin directory.

Data sources, filters, and sinks can be arbitrarily connected together with a connection digraph described using the [DOT graph description language](#).

Provided data source plugins include:

- *LibSensorsDataSource* - (Linux) computer motherboard sensors for temperature, fan speed, voltage etc.
- *SerialDataSource* - generic data received through a serial port device. Arduino code for acquiring and sending data through its USB serial connection is also included.
- *RandomWalkDataSource* - testing or demonstration data source using a random walk algorithm.
- *ThorlabsPM100DataSource* - laser or light power measurement using the Thorlabs PM100 power meter.
- *PicoTC08DataSource* - thermocouple or other sensor measurements using the Pico Technologies TC-08 USB data logger.

Provided data sink plugins include:

- *PrintDataSink* - print to standard out or standard error streams.
- *FileDataSink* - write to a file.
- *LoggingDataSink* - simple output to console using python logging system.
- *CSVDataFilter* - format data as a table of comma separated values.
- *InfluxDB2DataSink* - InfluxDB 2.x database system specialising in time-series data.
- *MatplotlibDataSink* - create a plot of data using matplotlib.

Provided data filter plugins include:

- *KeyValDataFilter* - selecting or discarding data entries based on key-value pairs.
- *TimeStampDataFilter* - adding timestamps to data.
- *AggregatorDataFilter* - aggregating multiple data readings into a fixed-size buffer.

1.1 Quick Start

This guide will run through the installation and configuration of the data logging daemon service.

1.1.1 Installation

Note: `pip3` and `python3` are used here because currently `pip` and `python` refer to `python2` versions on some common Linux distributions such as Ubuntu. On Windows, or distributions like Arch where `python3` is the default, `pip` and `python` may be used instead.

Note: These instructions assume a system-wide install, which requires root or administrator privileges. On Linux, either first switch to a root login with `sudo -i`, or prefix all commands with `sudo`. Alternatively, a user-level install can be performed by using the `--user` flag on `pip` or `systemd` commands, for example `pip3 install --user ...`, or `systemctl --user ...`.

Ensure the `pip` python package manager is installed. For example:

```
# Debian, Ubuntu etc.
apt install python3-pip
# Arch Linux
pacman -Sy python-pip
```

Install using `pip`:

```
pip3 install --upgrade datalogd
```

Some plugins require additional packages. These will be listed when the plugin is attempted to be loaded. The optional dependencies can be also be installed with `pip`, for example:

```
pip3 install --upgrade pyserial pyserial-asyncio PySensors influxdb matplotlib
```

The executable should now be available. This will show the available *command line parameters*:

```
datalogd --help
```

On Linux operating systems, a [systemd service file](#) will be installed and enabled to run on startup. Automatic configuration to start on alternate operating systems (such as Windows) is not yet implemented, and therefore must be done manually. Once the configuration file has been prepared, the service can be started with `systemctl start datalog`.

1.1.1.1 Windows

There are several ways of getting the service to run automatically at startup. This is one example which can be configured as a standard user without admin privileges.

- Get the `datalogd.xml` file and save it somewhere.
- Open Task Scheduler (windows key, type `taskschd.msc`, enter).
- Click Action->Import Task..., find and select the `datalogd.xml` file.
- Click Change User or Group... button, type your user name, click Check Names, then OK, and OK.

1.1.2 Configuration

The default configuration has no function, and so will not run. Configuring the daemon is performed by either creating or editing a configuration file, or passing parameters on the command line.

1.1.2.1 Configuration Files

To obtain the location of the default configuration files, run with the `--show-config-dirs` command line option.

```
datalogd --show-config-dirs
```

```
INFO:main:Default configuration file locations are:
/etc/xdg/datalogd/datalogd.conf
/root/.config/datalogd/datalogd.conf
```

All config files will be read, with any options in the later files overriding the earlier ones. Note also that a custom config file may be specified on the *command line*, and will be read last. Configuration specified as *command line parameters* will override any configuration read from files.

A configuration file should take the form of:

```
[datalogd]

plugin_paths = []

connection_graph =
  digraph {
    source [class=NullDataSource];
    sink [class=NullDataSink];
    source -> sink;
  }
```


The options are:

- `plugin_paths` - path(s) to directories containing custom source/filter/sink plugins. See the [Plugins](#) section for details on creating custom plugins.
- `connection_graph` - declaration of plugin nodes, parameters, and the connections between them. See the [Connection Graph](#) section for details on the connection graph syntax.

1.1.2.2 Command Line Parameters

```
datalogd --help
```

```
usage: datalogd [-h] [-c FILE] [-p DIR [DIR ...]] [-g GRAPH_DOT] [--show-config-dirs]
```

Run the data logging daemon service.

optional arguments:

```
-h, --help            show this help message and exit
-c FILE, --configfile FILE
                        Path to configuration file.
-p DIR [DIR ...], --plugindir DIR [DIR ...]
                        Directories containing additional plugins.
-g GRAPH_DOT, --graph-dot GRAPH_DOT
                        Connection graph specified in DOT format.
--show-config-dirs    Display the default locations of configuration files, then
↳exit.
```

1.2 Plugins

This section describes how to create plugins, specify the connections between them, and then run the complete data logging pipeline.

Plugins are python classes which extend one of the base plugin types. The class can be defined in any python source code file, and multiple plugin classes may be defined in a single file. The directory containing plugin source code can be specified in a configuration file, or in command line parameters.

1.2.1 Data Sources

Data source classes must extend `DataSource`. In addition, their class name must have the suffix “DataSource”.

The following code is a simple example of a functional data source plugin. It sends the string “Hello, World!” to any connected sinks once every 10 seconds:

Listing 1: plugin_demos/helloworld_datasource.py

```
import asyncio
from datalogd import DataSource

# Must extend datalogd.DataSource, and class name must have DataSource suffix
class HelloWorldDataSource(DataSource):
    def __init__(self, sinks=[]):
        # Do init of the superclass (DataSource), connect any specified sinks
        super().__init__(sinks=sinks)
```

(continues on next page)

(continued from previous page)

```
# Queue first call of update routine
asyncio.get_event_loop().call_soon(self.say_hello)

def say_hello(self):
    "Send ``Hello, World!`` to connected sinks, then repeat every 10 seconds."
    self.send("Hello, World!")
    asyncio.get_event_loop().call_later(10, self.say_hello)
```

Note the use of the `asyncio` event loop to schedule method calls, as opposed to a separate loop/sleep thread or similar.

Data sources will have their `close()` method called when the application is shutting down so that any resources (devices, files) used can be released.

1.2.2 Data Sinks

Data sinks accept data using their `receive()` method, and do something with it, such as committing it to a database. They must extend `DataSink`, and must use the suffix “DataSink”.

A simple example data sink which prints received data in uppercase to the console is:

Listing 2: plugin_demos/shout_datasink.py

```
from datalogd import DataSink

# Must extend datalogd.DataSink, and class name must have DataSink suffix
class ShoutDataSink(DataSink):
    # Override the receive() method to do something useful with received data
    def receive(self, data):
        "Accept ``data`` and shout it out to the console."
        print(str(data).upper())
```

Data sinks will have their `close()` method called when the application is shutting down so that any resources (devices, files) used can be released.

1.2.3 Data Filters

Data filters are in fact both `DataSources` and `DataSinks`, and thus share the functionality of both. They must extend `DataFilter`, and must use the suffix “DataFilter”.

The `NullDataFilter` is the most simple example of a filter, which accepts data and passes it unchanged to any connected sinks. A slightly more functional filter which performs some processing on the data is:

Listing 3: plugin_demos/helloworld_datafilter.py

```
from datalogd import DataFilter

# Must extend datalogd.DataFilter, and class name must have DataFilter suffix
class HelloWorldDataFilter(DataFilter):
    # Override the receive() method to do something useful with received data
    def receive(self, data):
        # Send modified data to all connected sinks
        self.send(str(data).replace("Hello", "Greetings"))
```

1.2.4 Connecting

To connect the above source, filter and sink to a complete “Hello, World!” data logger, the connection graph could be specified as:

```
digraph {
    source [class=HelloWorldDataSource];
    filter [class=HelloWorldDataFilter];
    sink   [class=ShoutDataSink];
    source -> filter -> sink;
}
```

1.2.5 Running

See the *Command Line Parameters* section for details on specifying configuration from the command line.

```
$ datalogd --plugindir plugin_demos --graph 'digraph{a[class=HelloWorldDataSource];
↳b[class=HelloWorldDataFilter];c[class=ShoutDataSink];a->b->c;}'
```

```
INFO:main:Initialising DataLogDaemon.
INFO:DataLogDaemon:Loaded config from: /etc/xdg/datalogd/datalogd.conf
INFO:pluginlib:Loading plugins from standard library
INFO:pluginlib:Adding plugin_demos as a plugin search path
INFO:DataLogDaemon:Detected source plugins: NullDataSource, LibSensorsDataSource,
↳RandomWalkDataSource, SerialDataSource, HelloWorldDataSource
INFO:DataLogDaemon:Detected filter plugins: NullDataFilter, AggregatorDataFilter,
↳CSVDataFilter, KeyValDataFilter, TimeStampDataFilter, HelloWorldDataFilter
INFO:DataLogDaemon:Detected sink plugins: NullDataSink, FileDataSink,
↳InfluxDBDataSink, LoggingDataSink, MatplotlibDataSink, PrintDataSink, ShoutDataSink
INFO:DataLogDaemon:Initialising node a:HelloWorldDataSource()
INFO:DataLogDaemon:Initialising node b:HelloWorldDataFilter()
INFO:DataLogDaemon:Initialising node c:ShoutDataSink()
INFO:DataLogDaemon:Connecting a:HelloWorldDataSource -> b:HelloWorldDataFilter
INFO:DataLogDaemon:Connecting b:HelloWorldDataFilter -> c:ShoutDataSink
INFO:main:Starting event loop.
GREETINGS, WORLD!
GREETINGS, WORLD!
GREETINGS, WORLD!
^CINFO:main:Stopping event loop.
```

1.3 Connection Graph

Specifying the plugins, their parameters, and the connections between them is performed using the *DOT* graph description language.

The default connection graph is valid, but not useful, as it simply connects a *NullDataSource* to a *NullDataSink*:

```
1 digraph {
2   source [class=NullDataSource];
3   sink   [class=NullDataSink];
4   source -> sink;
5 }
```

The purpose of each line is:

1. `digraph` declares that this is a directed graph. This is required, as the flow of data is not bi-directional.
2. This line declares a `NullDataSource` plugin named “source”. `source` is a unique label for this node, which can be any string such as “src” or “a”. Inside the square brackets are the attributes for the node. The only required attribute is `class`, which specifies the python class name of the plugin to use for the node. Additional attributes are passed to the `__init__()` method of the plugin.
3. This line declares a `NullDataSink` plugin named `sink`.
4. This line adds `sink` as a receiver of data from `source`. Connections between nodes are indicated with `->`.

A more complicated (but just as useless!) connection graph is:

```
digraph {
  a [class=NullDataSource];
  b [class=NullDataSource];
  c [class=NullDataFilter];
  d [class=NullDataSink];
  e [class=NullDataSink];
  a -> c -> d;
  b -> c -> e;
}
```

This graph has two data sources and two sinks, connected together with a common filter. Any data produced by either of the sources will be fed to both of the sinks.

1.3.1 Node Attributes

Some plugins may accept (or require) additional parameters during initialisation. These are provided by attributes of the graph node which describe the plugin. The `RandomWalkDataSource` is one such plugin. It generates demonstration data using a random walk algorithm, and the parameters of the algorithm can be specified using node attributes.

```
digraph {
  a [class=RandomWalkDataSource, interval=1.5, walkers="[[100, 2.5], [50, 1], [0, 10]]
  ↪"];
  b [class=LoggingDataSink];
  a -> b;
}
```

The value of `interval` specifies how often the algorithm should run, and the value of `walkers` describes how many random walkers should be used and their starting and increment values.

Any attribute values which contain DOT punctuation (space, comma, `[]`, `{}` etc) must be enclosed in double quotes, as seen for the `walkers` attribute. The enclosed string will then be interpreted as JSON to determine its type and value, however, any double quotes in the JSON must be replaced with single quotes, so as to not conflict with the double quotes of the DOT language.

Note: To force interpretation of an attribute as a string, enclose the value in an additional set of single quotes. The quotes will be removed during parsing of the DOT. For example, `id="1.23e4"` will be a float, while `id="'1.23e4'"` will be a string.

1.4 Data Logging Recipes

1.4.1 Random Data to CSV File

This generates random data, adds a timestamp, formats as CSV, and writes to a file. The default settings for the *FileDataSink* is to flush out to the file every 10 seconds.

Listing 4: recipes/randomwalk_timestamp_csv_file.config

```
[datalogd]
connection_graph =
  digraph {
    a [class=RandomWalkDataSource];
    b [class=TimeStampDataFilter];
    d [class=CSVDataFilter, labels=None];
    s [class=FileDataSink, filename="randomwalk.csv"];
    a -> b -> d -> s;
  }
```

```
$ datalogd -c recipes/randomwalk_timestamp_csv_file.config
```

```
INFO:main:Initialising DataLogDaemon.
INFO:DataLogDaemon:Loaded config from: /etc/xdg/datalogd/datalogd.conf, recipes/
↳randomwalk_timestamp_csv_file.config
INFO:pluginlib:Loading plugins from standard library
INFO:DataLogDaemon:Detected source plugins: NullDataSource, LibSensorsDataSource,
↳RandomWalkDataSource, SerialDataSource
INFO:DataLogDaemon:Detected filter plugins: NullDataFilter, AggregatorDataFilter,
↳CSVDataFilter, KeyValDataFilter, TimeStampDataFilter
INFO:DataLogDaemon:Detected sink plugins: NullDataSink, FileDataSink,
↳InfluxDBDataSink, LoggingDataSink, MatplotlibDataSink, PrintDataSink
INFO:DataLogDaemon:Initialising node a:RandomWalkDataSource()
INFO:DataLogDaemon:Initialising node b:TimeStampDataFilter()
INFO:DataLogDaemon:Initialising node d:CSVDataFilter(labels=None)
INFO:DataLogDaemon:Initialising node s:FileDataSink(filename=randomwalk.csv)
INFO:DataLogDaemon:Connecting a:RandomWalkDataSource -> b:TimeStampDataFilter
INFO:DataLogDaemon:Connecting b:TimeStampDataFilter -> d:CSVDataFilter
INFO:DataLogDaemon:Connecting d:CSVDataFilter -> s:FileDataSink
INFO:main:Starting event loop.
```

```
$ cat randomwalk.csv
```

```
2020-04-17 22:02:51.887071+09:30,-1.0,2020-04-17 22:02:51.887071+09:30,-2.0
2020-04-17 22:02:52.888208+09:30,0.0,2020-04-17 22:02:52.888208+09:30,0.0
2020-04-17 22:02:53.889423+09:30,-1.0,2020-04-17 22:02:53.889423+09:30,-2.0
2020-04-17 22:02:54.889818+09:30,-1.0,2020-04-17 22:02:54.889818+09:30,-4.0
2020-04-17 22:02:55.891065+09:30,-2.0,2020-04-17 22:02:55.891065+09:30,-6.0
2020-04-17 22:02:56.892261+09:30,-3.0,2020-04-17 22:02:56.892261+09:30,-8.0
2020-04-17 22:02:57.893478+09:30,-4.0,2020-04-17 22:02:57.893478+09:30,-10.0
2020-04-17 22:02:58.894673+09:30,-3.0,2020-04-17 22:02:58.894673+09:30,-12.0
2020-04-17 22:02:59.895897+09:30,-2.0,2020-04-17 22:02:59.895897+09:30,-14.0
...
```

There are no labels for the column headers, and the file will grow infinitely large with time. It might be better to aggregate the data and update the file with only the latest data. Note the use of `mode="w"` and `flush_interval=None`,

which causes the file to be opened, written, and closed on each receipt of data. In this way, each block of aggregated data will overwrite the old one.

Listing 5: recipes/randomwalk_timestamp_aggregator_csv_file.config

```
[datalogd]
connection_graph =
  digraph {
    a [class=RandomWalkDataSource];
    b [class=TimeStampDataFilter];
    c [class=AggregatorDataFilter, buffer_size=3600, send_every=10];
    d [class=CSVDataFilter];
    s [class=FileDataSink, filename="randomwalk.csv", mode="w", flush_interval=None];
    a -> b -> c -> d -> s;
  }
```

```
$ datalogd -c recipes/randomwalk_timestamp_aggregator_csv_file.config
```

```
INFO:main:Initialising DataLogDaemon.
INFO:DataLogDaemon:Loaded config from: /etc/xdg/datalogd/datalogd.conf, recipes/
↳randomwalk_timestamp_aggregator_csv_file.config
INFO:pluginlib:Loading plugins from standard library
INFO:DataLogDaemon:Detected source plugins: NullDataSource, LibSensorsDataSource,
↳RandomWalkDataSource, SerialDataSource
INFO:DataLogDaemon:Detected filter plugins: NullDataFilter, AggregatorDataFilter,
↳CSVDataFilter, KeyValDataFilter, TimeStampDataFilter
INFO:DataLogDaemon:Detected sink plugins: NullDataSink, FileDataSink,
↳InfluxDBDataSink, LoggingDataSink, MatplotlibDataSink, PrintDataSink
INFO:DataLogDaemon:Initialising node a:RandomWalkDataSource()
INFO:DataLogDaemon:Initialising node b:TimeStampDataFilter()
INFO:DataLogDaemon:Initialising node c:AggregatorDataFilter(buffer_size=3600, send_
↳every=30)
INFO:DataLogDaemon:Initialising node d:CSVDataFilter()
INFO:DataLogDaemon:Initialising node s:FileDataSink(filename=randomwalk.csv)
INFO:DataLogDaemon:Connecting a:RandomWalkDataSource -> b:TimeStampDataFilter
INFO:DataLogDaemon:Connecting b:TimeStampDataFilter -> c:AggregatorDataFilter
INFO:DataLogDaemon:Connecting c:AggregatorDataFilter -> d:CSVDataFilter
INFO:DataLogDaemon:Connecting d:CSVDataFilter -> s:FileDataSink
INFO:main:Starting event loop.
```

```
$ cat randomwalk.csv
```

```
timestamp,analog_randomwalk0,timestamp,analog_randomwalk1
2020-04-17 22:27:36.264577+09:30,1.0,2020-04-17 22:27:36.264577+09:30,0.0
2020-04-17 22:27:37.265669+09:30,1.0,2020-04-17 22:27:37.265669+09:30,2.0
2020-04-17 22:27:38.266249+09:30,2.0,2020-04-17 22:27:38.266249+09:30,0.0
2020-04-17 22:27:39.267433+09:30,3.0,2020-04-17 22:27:39.267433+09:30,-2.0
2020-04-17 22:27:40.268600+09:30,2.0,2020-04-17 22:27:40.268600+09:30,0.0
2020-04-17 22:27:41.269554+09:30,2.0,2020-04-17 22:27:41.269554+09:30,0.0
2020-04-17 22:27:42.270715+09:30,3.0,2020-04-17 22:27:42.270715+09:30,-2.0
2020-04-17 22:27:43.271873+09:30,4.0,2020-04-17 22:27:43.271873+09:30,0.0
2020-04-17 22:27:44.272887+09:30,4.0,2020-04-17 22:27:44.272887+09:30,0.0
2020-04-17 22:27:45.274042+09:30,3.0,2020-04-17 22:27:45.274042+09:30,-2.0
2020-04-17 22:27:46.275201+09:30,2.0,2020-04-17 22:27:46.275201+09:30,0.0
2020-04-17 22:27:47.276220+09:30,1.0,2020-04-17 22:27:47.276220+09:30,0.0
```

(continues on next page)

(continued from previous page)

```
2020-04-17 22:27:48.277396+09:30,1.0,2020-04-17 22:27:48.277396+09:30,0.0
2020-04-17 22:27:49.278583+09:30,0.0,2020-04-17 22:27:49.278583+09:30,-2.0
2020-04-17 22:27:50.279763+09:30,1.0,2020-04-17 22:27:50.279763+09:30,-2.0
2020-04-17 22:27:51.280956+09:30,0.0,2020-04-17 22:27:51.280956+09:30,-2.0
2020-04-17 22:27:52.282120+09:30,0.0,2020-04-17 22:27:52.282120+09:30,-2.0
2020-04-17 22:27:53.283198+09:30,0.0,2020-04-17 22:27:53.283198+09:30,-2.0
2020-04-17 22:27:54.284388+09:30,1.0,2020-04-17 22:27:54.284388+09:30,-2.0
2020-04-17 22:27:55.285719+09:30,0.0,2020-04-17 22:27:55.285719+09:30,-4.0
2020-04-17 22:27:56.286249+09:30,0.0,2020-04-17 22:27:56.286249+09:30,-4.0
2020-04-17 22:27:57.287438+09:30,-1.0,2020-04-17 22:27:57.287438+09:30,-6.0
2020-04-17 22:27:58.288604+09:30,-2.0,2020-04-17 22:27:58.288604+09:30,-8.0
2020-04-17 22:27:59.289765+09:30,-2.0,2020-04-17 22:27:59.289765+09:30,-10.0
2020-04-17 22:28:00.290927+09:30,-1.0,2020-04-17 22:28:00.290927+09:30,-10.0
2020-04-17 22:28:01.292095+09:30,0.0,2020-04-17 22:28:01.292095+09:30,-12.0
2020-04-17 22:28:02.292911+09:30,0.0,2020-04-17 22:28:02.292911+09:30,-10.0
2020-04-17 22:28:03.294073+09:30,0.0,2020-04-17 22:28:03.294073+09:30,-12.0
2020-04-17 22:28:04.295262+09:30,-1.0,2020-04-17 22:28:04.295262+09:30,-12.0
2020-04-17 22:28:05.296247+09:30,-2.0,2020-04-17 22:28:05.296247+09:30,-10.0
...
```

1.4.2 System Temperatures to Matplotlib Plot

Find available sensor data from libsensors. The `id` field will be a composite of the device and sensor name.

```
$ sensors
```

```
nvme-pci-0100
Adapter: PCI adapter
Composite:      +37.9°C (low = -273.1°C, high = +82.8°C)
                (crit = +84.8°C)
Sensor 1:      +37.9°C (low = -273.1°C, high = +65261.8°C)
Sensor 2:      +42.9°C (low = -273.1°C, high = +65261.8°C)

k10temp-pci-00c3
Adapter: PCI adapter
Vcore:         969.00 mV
Vsoc:          1.09 V
Tdie:          +40.0°C
Tctl:          +50.0°C
Icore:         5.00 A
Isoc:          8.50 A
```

We will select both the NVME composite (solid-state disk temperature) and k10temp Tdie (AMD CPU core temperature) using a pair of *KeyValDataFilters*. These make two separate data streams which are re-joined together before being passed on to the aggregator. The aggregator buffers one hour of data, and sends updated data to the *MatplotlibDataSink* once every minute.

Note that with some skilled regular expression use in `val`, it might be possible to use a single *KeyValDataFilter* to select all required sensor data, eliminating the need to rejoin the data streams.

Listing 6: `recipes/temperature_matplotlib.config`

```
[datalogd]
connection_graph =
```

(continues on next page)

(continued from previous page)

```

digraph {
  a [class=LibSensorsDataSource];
  b [class=TimeStampDataFilter];
  c [class=KeyValDataFilter, key="id", val="k10temp.*Tdie"];
  cc [class=KeyValDataFilter, key="id", val="nvme.*Composite"];
  d [class=JoinDataFilter];
  e [class=AggregatorDataFilter, buffer_size=3600, send_every=60];
  s [class=MatplotlibDataSink, filename="temperatures_plot.pdf"];
  a -> b -> c -> d -> e -> s;
  b -> cc -> d;
}

```

```
$ datalogd -c recipes/temperature_matplotlib.config
```

```

INFO:main:Initialising DataLogDaemon.
INFO:DataLogDaemon:Loaded config from: /etc/xdg/datalogd/datalogd.conf, recipes/
↳temperature_matplotlib.config
INFO:pluginlib:Loading plugins from standard library
INFO:DataLogDaemon:Detected source plugins: NullDataSource, LibSensorsDataSource,
↳RandomWalkDataSource, SerialDataSource
INFO:DataLogDaemon:Detected filter plugins: NullDataFilter, AggregatorDataFilter,
↳CSVDataFilter, JoinDataFilter, KeyValDataFilter, TimeStampDataFilter
INFO:DataLogDaemon:Detected sink plugins: NullDataSink, FileDataSink,
↳InfluxDBDataSink, LoggingDataSink, MatplotlibDataSink, PrintDataSink
INFO:DataLogDaemon:Initialising node a:LibSensorsDataSource()
INFO:DataLogDaemon:Initialising node b:TimeStampDataFilter()
INFO:DataLogDaemon:Initialising node c:KeyValDataFilter(key=id, val=k10temp.*Tdie)
INFO:DataLogDaemon:Initialising node cc:KeyValDataFilter(key=id, val=nvme.*Composite)
INFO:DataLogDaemon:Initialising node d:JoinDataFilter()
INFO:DataLogDaemon:Initialising node e:AggregatorDataFilter(buffer_size=3600, send_
↳every=60)
INFO:DataLogDaemon:Initialising node s:MatplotlibDataSink(filename=temperatures_plot.
↳pdf)
INFO:DataLogDaemon:Connecting a:LibSensorsDataSource -> b:TimeStampDataFilter
INFO:DataLogDaemon:Connecting b:TimeStampDataFilter -> c:KeyValDataFilter
INFO:DataLogDaemon:Connecting c:KeyValDataFilter -> d:JoinDataFilter
INFO:DataLogDaemon:Connecting d:JoinDataFilter -> e:AggregatorDataFilter
INFO:DataLogDaemon:Connecting e:AggregatorDataFilter -> s:MatplotlibDataSink
INFO:DataLogDaemon:Connecting b:TimeStampDataFilter -> cc:KeyValDataFilter
INFO:DataLogDaemon:Connecting cc:KeyValDataFilter -> d:JoinDataFilter
INFO:main:Starting event loop.

```

1.4.3 Temperatures to InfluxDB with Grafana Visualisation on Raspberry Pi

Note: This recipe is outdated. There is a new 2.x version of InfluxDB which integrates a web interface and dashboard (Grafana is not necessary). See the documentation for the v2 [InfluxDB2DataSink](#) plugin for more information.

This recipe is for running on a Raspberry Pi. It will collect temperature readings from [commonly available DS18B20](#) temperature sensors and log them in an [InfluxDB](#) database. [Grafana](#) is used display the data over a web interface.

1.4.3.1 Hardware Setup

Any Raspberry Pi should work, this was tested using a Raspberry Pi 3 running Raspbian Buster using a Linux 4.19 kernel.

The DS18B20 temperature sensors require a 4.7 k Ω pullup resistor connected between the VCC and DATA lines. Some integrated modules already include the resistor. Connect the VCC to the RPi GPIO 3.3 V (pin 1), GND to ground (pin 6), and DATA to pin 7.

The 1-Wire bus on the RPi is not enabled by default. The easiest way to enable it is to run `sudo raspi-config` and select 1-Wire from the Interfacing Options. A reboot will be required. After the restart, check the sensor is detected with `ls /sys/bus/w1/devices`, which should list one or more devices starting with 28- and then a hex serial number. A reading can be obtained by running `cat /sys/bus/w1/devices/28-xxxxxxxxxxxxx/w1_slave`, where xxxxxxxxxxxxxx is the serial number of the sensor. The response should include something like `t=22062`, where 22062 indicates 22.062 °C.

The standard Linux libsensors is able to read the temperatures from attached probes:

```
sudo apt install libsensors5 lm-sensors
pip3 install --user PySensors
```

and can be read by running `sensors`.

1.4.3.2 Software Setup

1.4.3.2.1 InfluxDB

Add InfluxDB repository to apt sources (change buster to match your raspbian version):

```
wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add -
echo "deb https://repos.influxdata.com/debian buster stable" | sudo tee /etc/apt/
sources.list.d/influxdb.list
```

Install InfluxDB, python plugin module, and start the systemd service:

```
sudo apt-get update
sudo apt-get install influxdb
pip3 install --user influxdb
sudo systemctl enable --now influxdb.service
```

The default configuration is probably OK, but can be changed by editing `/etc/influxdb/influxdb.conf`. The database service will be operating on port 8086.

Create a database for datalogd to store its data:

```
influx
create database datalogd
exit
```

1.4.3.2.2 Grafana

Follow instructions on the [Grafana website](https://grafana.com/docs/5.7.0/installation/) to get latest version for your Raspberry Pi (eg. ARMv7). As of writing, for the Raspberry Pi 3 this is:

The Grafana server should be serving web pages from port 3000, so log into your RPi with a web browser and check that it is working (eg. visit `http://ip_of_your_pi:3000`).

1.4.3.2.3 Recipe

The connection graph can be configured using most defaults as:

Listing 7: recipes/rpi_temperature_influxdb.config

```
[datalogd]
connection_graph =
  digraph {
    a [class=LibSensorsDataSource];
    f [class=KeyValDataFilter, key="type", val="temperature"];
    s [class=InfluxDBDataSink];
    a -> f -> s;
  }
```

If you have changed the database configuration, such as usernames or passwords, then these must be set in the attributes for the *InfluxDBDataSink* node.

1.4.3.2.4 Visualisation

Log into your Grafana server (eg. http://ip_of_your_pi:3000). Configure an InfluxDB data source, using all default parameters. Configure a Grafana Dashboard as pictured:



1.4.4 Arduino Temperatures to InfluxDB and Grafana Display on Windows

Note: This recipe is outdated. There is a new 2.x version of InfluxDB which integrates a web interface and dashboard (Grafana is not necessary). See the documentation for the v2 [InfluxDB2DataSink](#) plugin for more information.

This recipe outlines connecting commonly available DS18B20 temperature sensors to an Arduino, which collects data and sends it via a USB serial connection to a Windows machine running InfluxDB database. Grafana is used display

the data over a web interface.

Note: No, I don't know why doing anything on Windows is so difficult.

1.4.4.1 Hardware

Get the [Arduino code](#) from [GitLab](#) or copy and paste below into a new Arduino sketch. Upload it to your Arduino.

Listing 8: datalog.ino

```
#include <Arduino.h>
#include <Wire.h>
#include <SPI.h>

////////////////////////////////////
// Device Configuration Settings
////////////////////////////////////

// An ID string for this Arduino
#define BOARD_ID_STRING "0"

// Interval between reads of devices
#define READ_INTERVAL 1000

// Select which types of sensors to use
#define USE_DIGITAL_PINS true
#define USE_ANALOG_PINS true
#define USE_DS18B20_TEMPERATURE true
#define USE_BH1750_LUX false
#define USE_FLOW false

////////////////////////////////////
// Pin Definitions and Sensor Configuration
////////////////////////////////////

#if USE_ANALOG_PINS
  // Set of analog input pins to read
  const int analog_pins[] = {A0, A1, A2, A3, A4, A5};
#endif

#if USE_DIGITAL_PINS
  // Set of digital input pins to read
  const int digital_pins[] = {4, 5, 6};
#endif

#if USE_DS18B20_TEMPERATURE
  // Run the 1-wire bus on pin 12
  const int onewire_pin = 12;
#endif

#if USE_FLOW
  // Flow sensor pulse pin input, must be interrupt enabled
  // These are pins 0, 1, 2, 3, 7 for a Leonardo board
  // Note that Leonardo needs pins 0+1 for Serial1 and 2+3 for I2C
  const int flow_pin = 7;
```

(continues on next page)

(continued from previous page)

```
// YF-S401 sensor apparently 5880 pulses per litre (~1.7 mL per pulse)
// Measured more like 5440, but calibration at different flow rates probably a good
↳thing to do
//const float flow_litre_per_pulse = 1.0/5440.0;
//const float flow_litre_per_pulse = 1.0;
#endif

////////////////////////////////////

#ifdef USE_DS18B20_TEMPERATURE
#include <OneWire.h>
#include <DallasTemperature.h>
// Initialise the 1-Wire bus
OneWire oneWire(onewire_pin);
// Pass our 1-Wire reference to Dallas Temperature
DallasTemperature thermometers(&oneWire);
#endif

#ifdef USE_BH1750_LUX
#include <hp_BH1750.h>
// Reference to the BH1750 light meter module over I2C
hp_BH1750 luxmeter;
#endif

#ifdef USE_FLOW
// Number of pulses read from the flow meter
volatile unsigned long flow_count = 0;
// Stored start time and pulse count for flow rate calculation
unsigned long flow_start_millis = 0;
unsigned long flow_start_count = 0;
#endif

// Variable to record last data acquisition time
unsigned long measurement_start_milis = 0;

// Variable to keep track of whether record separators (comma) needs to be prepended
↳to output
bool first_measurement = true;

#ifdef USE_DS18B20_TEMPERATURE
// Format a DS18B20 device address to a 16-char hex string
String formatAddress(DeviceAddress address) {
String hex = "";
for (uint8_t i = 0; i < 8; i++) {
if (address[i] < 16) hex += "0";
hex += String(address[i], HEX);
}
return hex;
}
#endif

// Print out a measurement to the serial port
void printMeasurement(String type, String id, String value, String units="") {
// A comma separator needs to be prepended to measurements other than the first
```

(continues on next page)

(continued from previous page)

```

    if (first_measurement) {
        first_measurement = false;
    } else {
        Serial.print(",");
    }
    Serial.print("{\"type\":");
    Serial.print(type);
    Serial.print("\",\"id\":");
    Serial.print(id);
    Serial.print("\",\"value\":");
    Serial.print(value);
    if (units.length() > 0) {
        Serial.print("\",\"units\":");
        Serial.print(units);
    }
    Serial.print("\"}");
}

#ifdef USE_FLOW
    // Interrupt handler for a pulse from the flow meter
    void flowIncrement() {
        flow_count++;
    }
#endif

void setup(void)
{
    // Open serial port
    Serial.begin(115200);

    #if USE_DS18B20_TEMPERATURE
        // Initialise I2C bus
        Wire.begin();
        pinMode(onewire_pin, INPUT_PULLUP);
    #endif

    #if USE_DIGITAL_PINS
        // Configure set of digital input pins
        for (uint8_t i = 0; i < uint8_t(sizeof(digital_pins)/sizeof(digital_pins[0]));
        ↪ i++) {
            pinMode(digital_pins[i], INPUT);
        }
    #endif

    #if USE_FLOW
        // Configure the flow meter input pin and interrupt for pulse counting
        pinMode(flow_pin, INPUT_PULLUP);
        attachInterrupt(digitalPinToInterrupt(flow_pin), flowIncrement, RISING);
        flow_start_millis = millis();
    #endif
}

void loop(void)
{
    // Record current time
    unsigned long current_millis = millis();

```

(continues on next page)

(continued from previous page)

```
// Check if it's time to take some new measurements
if (current_millis - measurement_start_milis >= READ_INTERVAL) {
    measurement_start_milis = current_millis;
    // The first measurement in this cycle doesn't need a comma delimiter prepended
    first_measurement = true;

    // Print message start
    Serial.print("{\"board\":\"" + String(BOARD_ID_STRING) + "\",");
    Serial.print("\"timestamp\":\"" + String(measurement_start_milis) + "\",");
    Serial.print("\"message\":\"measurement\", \"data\":[");

    //////////////////////////////////////
    // Arduino Digital Pins
    //////////////////////////////////////
    #if USE_DIGITAL_PINS
        // Read digital pins
        unsigned int d = 0;
        for (uint8_t i = 0; i < uint8_t(sizeof(digital_pins)/sizeof(digital_pins[0]));
↪i++) {
            d += digitalRead(digital_pins[i]) << i;
        }
        printMeasurement("digital", "0", String(d));
    #endif

    //////////////////////////////////////
    // Arduino Analog Pins
    //////////////////////////////////////
    #if USE_ANALOG_PINS
        // Read analog pins
        for (uint8_t i = 0; i < uint8_t(sizeof(analog_pins)/sizeof(analog_pins[0]));
↪i++) {
            printMeasurement("analog", String(i), String(analogRead(analog_pins[i])));
        }
    #endif

    //////////////////////////////////////
    // DS18B20 Temperature Probes
    //////////////////////////////////////
    #if USE_DS18B20_TEMPERATURE
        // We'll reinitialise the temperature probes each time inside the loop so that
        // devices can be connected/disconnected while running
        thermometers.begin();
        // Temporary variable for storing 1-Wire device addresses
        DeviceAddress address;
        // Grab a count of temperature probes on the wire
        unsigned int numberOfDevices = thermometers.getDeviceCount();
        // Loop through each device, set requested precision
        for(unsigned int i = 0; i < numberOfDevices; i++) {
            if(thermometers.getAddress(address, i)) {
                thermometers.setResolution(address, 12);
            }
        }
        // Issue a global temperature request to all devices on the bus
        if (numberOfDevices > 0) {
            thermometers.requestTemperatures();
        }
        // Loop through each device, print out temperature data
    #endif
}
```

(continues on next page)

(continued from previous page)

```

    for(unsigned int i = 0; i < numberOfDevices; i++) {
        if(thermometers.getAddress(address, i)) {
            printMeasurement("temperature", formatAddress(address), String(thermometers.
↪getTempC(address), 2), "C");
        }
    }
#endif

////////////////////////////////////
// BH1750 Lux Meter
////////////////////////////////////
#if USE_BH1750_LUX
    // Attempt to initialise and read light meter sensor
    if (luxmeter.begin(BH1750_TO_GROUND)) {
        luxmeter.start();
        printMeasurement("lux", "0", String(luxmeter.getLux(), 0), "lux");
    }
#endif

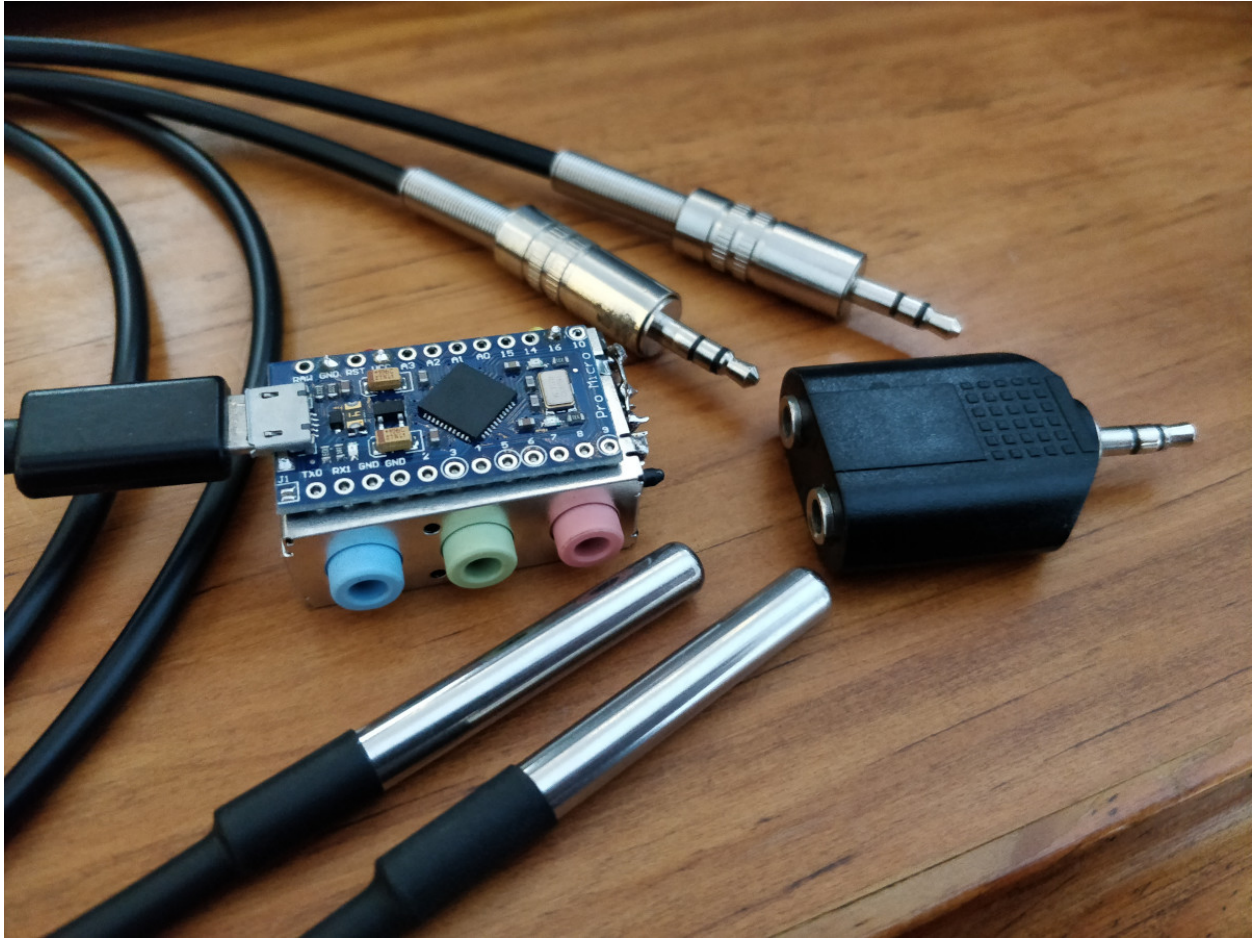
////////////////////////////////////
// Fluid Flow Meter
////////////////////////////////////
#if USE_FLOW
    unsigned long flow_end_count = flow_count;
    unsigned long flow_end_millis = millis();
    // Total volume in sensor pulses
    printMeasurement("flow_volume", "0", String(flow_end_count, 4), "pulses");
    // Current flow rate in pulses per minute
    float flow_rate = 60000.0*(flow_end_count - flow_start_count)/(float)(flow_end_
↪millis - flow_start_millis);
    printMeasurement("flow_rate", "0", String(flow_rate, 4), "pulses/min");
    flow_start_count = flow_end_count;
    flow_start_millis = flow_end_millis;
#endif

    // Print message end
    Serial.println("}");
}
}

```

Connect VCC to +5 V, GND to ground, and DATA to pin 12 (or another, to match that specified in the code). Plug the Arduino into your Windows machine with a USB cable.

This is a “Pro Micro” Arduino Leonardo compatible board, connected to the DS18B20 sensors using 3.5 mm audio jacks. The benefits of these are that readily available plugs, sockets, splitters and extensions can be used. Don’t try plugging headphones in though, they will likely be fried!



1.4.4.2 Recipe

Listing 9: recipes/serial_temperatures_influxdb.config

```
[datalogd]
connection_graph =
  digraph {
    a [class=SerialDataSource];
    f [class=KeyValDataFilter, key="type", val="temperature"];
    s [class=InfluxDBDataSink];
    a -> f -> s;
  }
```

1.4.4.3 Software

Download and install InfluxDB, and configure it to run at startup:

- Download InfluxDB from <https://portal.influxdata.com/downloads/> and unzip to Program Files, then rename directory to InfluxDB.
- Go the directory and run `influxd.exe`.
- Run `influx.exe`. Type `create database datalogd` then exit.

- Hit `Control+C` on the `influxd.exe` window.
- Get the [InfluxDB.xml](#) file and save it somewhere.
- Open Task Scheduler (windows key, type `taskschd.msc`, enter). Click Action->Import Task... , select the `InfluxDB.xml` file. Click the Change User or Group... button, type your user name, click Check Names, then OK.
- Right click the `datalogd` entry, and select Run, then click OK.

Download and install Grafana:

- [Download Grafana](#).
- Install.
- Go to `http://localhost:3000`, login with `admin admin`, and set a new password.
- Add an InfluxDB data source using default settings.

2.1 datalogd package

The `datalogd` package contains the main *DataLogDaemon*, plus the plugin base classes *DataSource*, *DataFilter*, and *DataSink*, which must be extended to provide useful functionality.

The included data source/filter/sink plugins are contained separately in the *plugins* package.

class `datalogd.DataFilter` (*sinks=[]*)

Bases: `datalogd.DataSource`, `datalogd.DataSink`, `pluginlib._parent.Plugin`

The base class for all data filter plugins.

DataFilters are subclasses of both *DataSources* and *DataSinks*, thus are capable of both sending and receiving data. Typically, they are used to sit between a *DataSource* and a *DataSink* (or other *DataFilters*) in order to modify the data flowing between them in some way.

class `datalogd.DataLogDaemon` (*configfile=None*, *plugindir=[]*, *graph_dot=None*)

Bases: `object`

The main `datalogd` class.

The *DataLogDaemon* reads configuration file(s), interprets the connection graph DOT specification, and initialises data source/filter/sink plugins and connections. The `asyncio` event loop must be started separately. For an example of this, see the `main()` method, which is the typical way the daemon is started.

Parameters

- **configfile** – Path to configuration file to load.
- **plugindir** – Directory, or list of directories from which to load additional plugins.
- **graph_dot** – Connection graph specified in the DOT graph description language.

close()

Notify nodes that the application is closing so they may shutdown gracefully.

class `datalogd.DataSink`

Bases: `pluginlib._parent.Plugin`

The base class for all data sink plugins.

DataSinks have a *receive()* method which accepts data from connected *DataSources*.

close()

Perform any cleanup required during application shutdown.

receive(data)

Accept the provided data.

Parameters data – Data received by this sink.

class datalogd.**DataSource** (*sinks=[]*)

Bases: *pluginlib._parent.Plugin*

The base class for all data sink plugins.

DataSource implements methods for connecting or disconnecting sinks, and for sending data to connected sinks. It has no intrinsic functionality (it does not actually produce any data) and is not itself considered a plugin, so can't be instantiated using the connection graph.

Parameters sinks – *DataSink* or list of *DataSinks* to receive data produced by this *DataSource*.

close()

Perform any cleanup required during application shutdown.

connect_sinks(sinks)

Register the provided *DataSink* as a receiver of data produced by this *DataSource*. A list of sinks may also be provided.

Parameters sinks – *DataSink* or list of *DataSinks*.

disconnect_sinks(sinks)

Unregister the provided *DataSink* so that it no longer receives data produced by this *DataSource*. A list of sinks may also be provided. It is not an error to provide a sink that is not currently connected.

Parameters sinks – *DataSink* or list of *DataSinks*.

send(data)

Send the provided data to all connected *DataSinks*.

Parameters data – Data to send to *DataSinks*.

class datalogd.**NullDataFilter** (*sinks=[]*)

Bases: *datalogd.DataFilter*

A *DataFilter* which accepts data and passes it unchanged to any connected *DataSinks*.

receive(data)

Pass data unchanged to all connected *DataSinks*.

Parameters data – Data received by this filter.

class datalogd.**NullDataSink**

Bases: *datalogd.DataSink*

A *DataSink* which accepts data and does nothing with it.

Unlike the base *DataSink*, this can be instantiated using the connection graph, although it provides no additional functionality.

class datalogd.**NullDataSource** (*sinks=[]*)

Bases: *datalogd.DataSource*

A *DataSource* which produces no data.

Unlike the base *DataSource*, this can be instantiated using the connection graph, although it provides no additional functionality.

`datalogd.listify(value)`

Convert value into a list.

Modifies the behaviour of the python builtin `list()` by accepting all types as value, not just iterables. Additionally, the behaviour of iterables is changed:

- `list('str') == ['s', 't', 'r']`, while `listify('str') == ['str']`
- `list({'key': 'value'}) == ['key']`, while `listify({'key': 'value'}) == [{'key': 'value'}]`

Parameters *value* – Input value to convert to a list.

Returns *value* as a list.

`datalogd.main()`

Read command line parameters, instantiate a new *DataLogDaemon* and begin execution of the event loop.

`datalogd.parse_dot_json(value)`

Interpret the value of a DOT attribute as JSON data.

DOT syntax requires double quotes around values which contain DOT punctuation (space, comma, {}, [] etc), and, if used, these quotes will also be present in the obtained value string. Unfortunately, JSON also uses double quotes for string values, which are then in conflict. This method will strip any double quotes from the passed value, then will attempt to interpret as JSON after replacing single quotes with double quotes.

Note that the use of this workaround means that single quotes must be used in any JSON data contained in the DOT attribute values.

Parameters *value* – string to interpret.

Returns *value*, possibly as a new type.

2.1.1 Subpackages

2.1.1.1 datalogd.plugins package

The plugins package contains the included *DataSource*, *DataFilter*, and *DataSink* subclasses. Some plugins will require additional python modules to be installed, or may have other specific requirements (for example, the *LibSensorsDataSource* probably won't work on Windows operating systems).

2.1.1.1.1 Submodules

`datalogd.plugins.aggregator_datafilter` module

```
class datalogd.plugins.aggregator_datafilter.AggregatorDataFilter(sinks=[],
                                                                    buffer_size=100,
                                                                    send_every=100,
                                                                    aggre-
                                                                    gate=['timestamp',
                                                                    'value'])
```

Bases: *datalogd.DataFilter*

Aggregates consecutively received data values into a list and passes the new array(s) on to sinks.

The aggregated data can be sent at different intervals to that which it is received by using the `send_every` parameter. A value of 1 will send the aggregated data every time new data is received. A value of `send_every` equal to `buffer_size` will result in the data being passed on only once the buffer is filled. A typical usage example would be for data which is received once every second, using `buffer_size=86400` and `send_every=60` to store up to 24 hours of data, and update sinks every minute.

Parameters

- **buffer_size** – Maximum length for lists of aggregated data.
- **send_every** – Send data to connected sinks every *n* updates.
- **aggregate** – List of data keys for which the values should be aggregated.

receive (*data*)

Accept data, aggregate selected values, and pass on aggregated data to any connected sinks.

Parameters *data* – Data containing values to aggregate.

datalogd.plugins.csv_datafilter module

```
class datalogd.plugins.csv_datafilter.CSVDataFilter(sinks=[], keys=['timestamp',
                                                         'value'], labels=['timestamp',
                                                         '{type}_{id}'], header='every')
```

Bases: *datalogd.DataFilter*

Format received data into a table of comma separated values.

The column headers can be formatted using values from the data. For example, for the data:

```
[
    {'type': 'temperature', 'id': '0', 'value': 22.35},
    {'type': 'humidity', 'id': '0', 'value': 55.0},
    {'type': 'temperature', 'id': '1', 'value': 25.80},
]
```

and a node initialised using:

```
sink [class=CSVDataSink keys=["'value'"], labels=["'{type}_{id}'"]];
```

the output will be:

```
temperature_0,humidity_0,temperature_1
22.35,55.0,25.80
```

Setting `labels` to `None` will disable the column headers.

By default, the column headers will be generated on every receipt of data. To instead output the column headers only once on the first receipt of data, use the parameter `header="once"`. Setting `header=None` will also disable the headers completely.

Parameters

- **keys** – Name of data keys to format into columns of the CSV.
- **labels** – Labels for the column headers, which may contain mappings to data values.
- **header** – Display the column header "once" or "every" or `None`.

receive (*data*)

Accept data and format into a table of CSV.

Parameters **data** – Data to format as CSV.

datalogd.plugins.file_datasink module

```
class datalogd.plugins.file_datasink.FileDataSink (filename, file-  
                                                    name_timestamp=False, mode='w',  
                                                    header="", terminator='n',  
                                                    flush_interval=10)
```

Bases: *datalogd.DataSink*

Write data to a file.

By default, any existing contents of `filename` will be overwritten without prompting. To instead raise an error if the file exists, set the `mode` parameter to 'x'. The contents of any existing file will be appended to by setting `mode='a'`.

To automatically prepend a date and time stamp to the given filename, set `filename_timestamp=True`. In general, this should create a unique filename and prevent overwriting when `filename` already exists.

The `flush_interval` parameter controls the behaviour of the file writes. It describes how often, in seconds, the operating system's buffers should be flushed to disk, updating the file contents:

- `flush_interval > 0` causes the flush to occur at the given time interval, in seconds. More frequent flushes will keep the contents of the file updated, but put more strain on the machines I/O systems.
- `flush_interval = 0` will flush immediately after each receipt of data.
- `flush_interval < 0` will not automatically flush, leaving this to the operating system. The contents of the file may not update until the program closes.
- `flush_interval == None` will perform a file open, write, and close operation on each receipt of data. This may be desired if the contents of the file should only contain the latest received data (and should be used in conjunction with the `mode='w'` parameter).

Parameters

- **filename** – File name to write data to.
- **filename_timestamp** – Prepend a timestamp to the filename.
- **mode** – Mode in which to open the file. One of 'w' (write), 'a' (append), 'x' (exclusive creation).
- **header** – Header to write to file after plugin initialisation.
- **terminator** – Separator written to file after each receipt of data.
- **flush_interval** – Interval, in seconds, between flushes to disk.

close()

Close the connection to the file.

receive(data)

Accept data and write it out to the file.

The `terminator` specified in the constructor will be appended to the file after each call of this method.

Parameters **data** – Data to write to file.

datalogd.plugins.influxdb2_datasink module

```
class datalogd.plugins.influxdb2_datasink.InfluxDB2DataSink (url='http://localhost:8086',  
token="",  
org='default',  
bucket='default',  
measurement='measurement',  
run_id=None,  
field_key=None,  
field_value=None)
```

Bases: *datalogd.DataSink*

Connection to a InfluxDB 2.x (or 1.8+) database for storing time-series data.

Note that this doesn't actually run the InfluxDB database service, but simply connects to an existing InfluxDB database via a network (or localhost) connection. See the [getting started](#) documentation for details on configuring a new database server.

The `url` parameter should be a string specifying the protocol, server ip or name, and port. For example, `url="http://localhost:8086"`.

The authentication `token` parameter needs to be specified to allow commits to the database. See the [token](#) documentation to see how to create and obtain tokens.

Parameters for `org`, `bucket` must correspond to a valid organisation and bucket created in the database for which the authentication token is valid. See the documentation for [organisations](#) and [buckets](#) for details.

The `measurement` parameter specifies the data point measurement (or "table") the data will be entered into, and does not need to already exist. See the documentation on [data elements](#) for details.

A `run_id` parameter may be passed which will be added as a tag to the data points. It may be used to identify data obtained from this particular run of the data logging session. If no value is provided, a value will be generated from a YYYYMMDD-HHMMSS formatted time stamp.

The data point field key will be attempted to be determined automatically from the incoming data dictionaries. If the data dictionary contains a `name` or `label` key, then its value will be used as the database point field key. Alternatively, a field key will be generated from the values of `type` and `id` if present. Finally, a default field key of data will be used. To instead specify the data entry which should provide the field key, specify it as the `field_key` parameter.

Similarly, the data point field value will use the value from the incoming data dictionary's `value` field if present. To instead specify the data entry which should provide the field value, specify it as the `field_value` parameter.

Parameters

- **url** – Protocol, host name or IP address, and port number of InfluxDB server.
- **token** – API token used to authenticate with the InfluxDB server.
- **org** – Name of InfluxDB organisation in which to store data.
- **bucket** – Name of InfluxDB bucket in which to store data.
- **measurement** – Name for the InfluxDB measurement session.
- **run_id** – A tag to identify commits from this run.
- **field_key** – A field from the incoming data used to determine the data point field key.
- **field_value** – A field from the incoming data used to determine the data point field value.

close()

Close the connection to the database.

receive(data)

Commit data to the InfluxDB database.

Multiple items of data can be submitted at once if `data` is a list. A typical format of data would be:

```
[
  {'type': 'temperature', 'id': '0', 'value': 22.35},
  {'type': 'humidity', 'id': '0', 'value': 55.0},
  {'type': 'temperature', 'id': '1', 'value': 25.80},
]
```

In the above case (assuming the `field_key` and `field_value` parameters were not supplied when initialising the plugin), the InfluxDB data point field would be generated as `<type>_<id> = <value>`, and only the global `run_id` parameter would be entered into the data point keys.

If a `name` or `label` field is present, then it will instead be used as the InfluxDB data point field key. For example:

```
[
  {'name': 'Temperature', 'type': 'temperature', 'id': '0', 'value': 22.35},
  {'name': 'Humidity', 'type': 'humidity', 'id': '0', 'value': 55.0},
]
```

In this case, the InfluxDB data point field would be generated as `<name> = <value>`, and the remaining fields (`type` and `id`) would be added as data point field keys, along with the `run_id`.

A timestamp for the commit will be generated using the current system clock if a “timestamp” field does not already exist.

Parameters `data` – Data to commit to the database.

datalogd.plugins.influxdb_datasink module

```
class datalogd.plugins.influxdb_datasink.InfluxDBDataSink (host='localhost',
                                                            port=8086,
                                                            user='root',      pass-
                                                            word='root',      db-
                                                            name='datalogd',
                                                            session='default',
                                                            run=None)
```

Bases: `datalogd.DataSink`

Connection to a InfluxDB database for storing time-series data.

Note: This plugin is outdated. For new InfluxDB 2.x installs, use the `InfluxDB2DataSink` plugin instead.

Note that this doesn't actually run the InfluxDB database service, but simply connects to an existing InfluxDB database via a network (or localhost) connection. See the [getting started](#) documentation for details on configuring a new database server.

Parameters

- **host** – Host name or IP address of InfluxDB server.
- **port** – Port used by InfluxDB server.

- **user** – Name of database user.
- **password** – Password for database user.
- **dbname** – Name of database in which to store data.
- **session** – A name for the measurement session.
- **run** – A tag to identify commits from this run. Default of `None` will use a date/time stamp.

close()

Close the connection to the database.

receive(data)

Commit data to the InfluxDB database.

Multiple items of data can be submitted at once if `data` is a list. A typical format of data would be:

```
[
  {'type': 'temperature', 'id': '0', 'value': 22.35},
  {'type': 'humidity', 'id': '0', 'value': 55.0},
  {'type': 'temperature', 'id': '1', 'value': 25.80},
]
```

The data point will have its data field generated using the form `<type>_<id> = <value>`.

A timestamp for the commit will be generated using the current system clock if a “timestamp” field does not already exist.

Parameters data – Data to commit to the database.

datalogd.plugins.join_datafilter module

class `datalogd.plugins.join_datafilter.JoinDataFilter(sinks=[], count=2)`

Bases: `datalogd.DataFilter`

Join two or more consecutive receipts of data together into a list.

If the data are already lists, the two lists will be merged.

Parameters count – Number of data receipts to join.

receive(data)

Accept data and join consecutive receipts of data together into a list.

Parameters data – data to join.

datalogd.plugins.keyval_datafilter module

class `datalogd.plugins.keyval_datafilter.KeyValDataFilter(sinks=[], select=True, keyvals=None, key='type', val=None)`

Bases: `datalogd.DataFilter`

Select or reject data based on key-value pairs.

Received data items will be inspected to see whether they contain the given keys, and that their values match the given values. The key-value pairs are supplied as a list in the form `[[key, value], [key2, value2], ...]`. All key-value pairs must be matched. A value of the python special value of `NotImplemented` will match any value. If both `value` and `data[key]` are strings, matching will be performed using regular expressions (in which case `" .* "` will match all strings). If the `select` flag is `True`, only matching data will be

passed on to the connected sinks, if it is `False`, only non-matching data (or data that does not contain the given key) will be passed on.

If only a single key-value pair needs to be matched, they may alternatively be passed as the `key` and `val` parameters. This is mainly intended for backwards compatibility.

Parameters

- **select** – Pass only matching data, or only non-matching data.
- **keyvals** – List of dictionary key-value pairs to match in incoming data.

`receive (data)`

Accept the provided `data`, and select or reject items before passing on to any connected sinks.

The selection is based upon the parameters provided to the constructor of this [KeyValDataFilter](#).

Parameters `data` – Data to filter.

datalogd.plugins.libensors_datasource module

```
class datalogd.plugins.libensors_datasource.LibSensorsDataSource (sinks=[],
                                                                inter-
                                                                val=1.0)
```

Bases: [datalogd.DataSource](#)

Provide data about the running system's hardware obtained using the `libensors` library.

`libensors` is present on most Linux systems, or can be installed from the distribution's repositories (`apt install libensors5` on Debian/Ubuntu, `pacman -S lm_sensors` on Arch etc.). The available sensors will depend on your hardware, Linux kernel, and version of `libensors`.

Attempting to initialise this plugin on Windows operating systems will almost certainly fail.

Parameters `interval` – How often to poll the sensors, in seconds.

`close ()`

Close the connection to the sensors.

`read_sensors ()`

Read sensors and send data to any connected sinks.

```
class datalogd.plugins.libensors_datasource.LibSensorsFeatureType
```

Bases: [enum.Enum](#)

A utility [Enum](#) used to interpret integers representing sensor feature types.

BEEP_ENABLE = 24

CURR = 5

ENERGY = 4

FAN = 1

HUMIDITY = 6

IN = 0

INTRUSION = 17

POWER = 3

TEMP = 2

UNKNOWN = 4294967295

VID = 16

type

The name of the type of sensor reading.

units

The units associated with this sensor reading type.

datalogd.plugins.logging_datasink module

class `datalogd.plugins.logging_datasink.LoggingDataSink` (*level=20, header='Data received:', indent=' '*)

Bases: `datalogd.DataSink`

Output data using python's `logging` system.

Each item of data is output in a separate line, and the formatting can be controlled using the header and indent parameters.

Parameters

- **level** – The `logging level` to use for the output.
- **header** – Line of header text preceeding the logged data.
- **indent** – Prefix applied to each line of logged data.

receive (*data*)

Accept the provided data and output it using python's `logging` system.

Parameters **data** – Data to log.

datalogd.plugins.matplotlib_datasink module

class `datalogd.plugins.matplotlib_datasink.MatplotlibDataSink` (*filename='plot.pdf', keys=['timestamp', 'value'], labels=['timestamp', '{type}_{id}']*)

Bases: `datalogd.DataSink`

Note: This plugin is still a work in progress, and is really only at the proof-of-concept stage.

receive (*data*)

Accept the provided data.

Parameters **data** – Data received by this sink.

datalogd.plugins.picotc08_datasource module

```
class datalogd.plugins.picotc08_datasource.PicoTC08DataSource (sinks=[],      in-
                                                                terval=1.0,
                                                                mains_rejection='50Hz',
                                                                probes=[[1,
                                                                'Channel_1',
                                                                'K', 'C'], [2,
                                                                'Channel_2', 'K',
                                                                'C'], [3, 'Chan-
                                                                nel_3', 'K', 'C'],
                                                                [4, 'Channel_4',
                                                                'K', 'C'], [5,
                                                                'Channel_5', 'K',
                                                                'C'], [6, 'Chan-
                                                                nel_6', 'K', 'C'],
                                                                [7, 'Channel_7',
                                                                'K', 'C'], [8,
                                                                'Channel_8', 'K',
                                                                'C']])
```

Bases: *datalogd.DataSource*

Obtain readings from a Pico Technologies TC-08 USB data logging device.

The drivers and libraries (such as libusbtc08.so on Linux, usbt08.dll on Windows) from PicoTech must be installed into a system library directory, and the `picosdk` python wrappers package must be on the system (with `pip install picosdk` or similar).

The `interval` parameter determines how often data will be obtained from the sensors, in seconds. The minimum interval time is about 0.2 s for a single probe and 0.9 s for all eight.

The `mains_rejection` parameter filters out either 50 Hz or 60 Hz interference from mains power. The frequency is selected as either "50Hz" or "60Hz".

The `probes` parameter is a list of probe to initialise. Each element is itself a list of the form `[number, label, type, units]`, where probe numbers are unique integers from 1 to 8 corresponding to an input channel on the device. Probe labels can be any valid string. Valid probe thermocouple types are "B", "E", "J", "K", "N", "R", "S", "T", or "X", where "X" indicates a raw voltage reading. Units are one of Celsius, Fahrenheit, Kelvin, Rankine specified as "C", "F", "K" or "R". For the "X" probe type, readings will always be returned in millivolts.

acquire_data()

Acquire data from the TC-08 streaming buffer.

close()

Close the connection to the Pico TC-08 device.

datalogd.plugins.print_datasink module

```
class datalogd.plugins.print_datasink.PrintDataSink (end='n', stream='stdout')
```

Bases: *datalogd.DataSink*

Output data to standard-out or standard-error streams using the built-in python `print()` method.

Parameters

- **end** – Line terminator.

- **stream** – Output stream to use, either “stdout” or “stderr”.

receive (*data*)

Accept data and print it out.

Parameters *data* – Data to print.

datalogd.plugins.randomwalk_datasource module

```
class datalogd.plugins.randomwalk_datasource.RandomWalkDataSource (sinks=[],
                                                                    seed=None,
                                                                    inter-
                                                                    val=1.0,
                                                                    walk-
                                                                    ers=[[0.0,
                                                                    1.0], [0.0,
                                                                    2.0]])
```

Bases: *datalogd.DataSource*

Generate test or demonstration data using a random walk algorithm.

For each iteration of the algorithm, the output value will either be unchanged, increase, or decrease by a fixed increment. The options are chosen randomly with equal probability.

Multiple walkers can be initialised to produce several sources of random data. The *walkers* parameter is a list, the length of which determines the number of walkers to use. Each item in the list must be a list/tuple of two items: the walker’s initial value and increment.

Parameters

- **seed** – Seed used to initialise the random number generator.
- **interval** – How often to run an iteration of the algorithm, in seconds.
- **walkers** – List defining number of walkers and their parameters in the form *[[init, increment], ...]*.

generate_data ()

Run one iteration of the random walk algorithm and send the value to any connected sinks.

datalogd.plugins.serial_datasource module

```
class datalogd.plugins.serial_datasource.SerialDataSource (sinks=[],      port="",
                                                            board_id=None)
```

Bases: *datalogd.DataSource*

Receive data from an Arduino connected via a serial port device.

See the *datalog_arduino.ino* sketch for matching code to run on a USB-connected Arduino.

Listing 1: *datalog.ino*

```
#include <Arduino.h>
#include <Wire.h>
#include <SPI.h>

////////////////////////////////////
// Device Configuration Settings
////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
// An ID string for this Arduino
#define BOARD_ID_STRING "0"

// Interval between reads of devices
#define READ_INTERVAL 1000

// Select which types of sensors to use
#define USE_DIGITAL_PINS true
#define USE_ANALOG_PINS true
#define USE_DS18B20_TEMPERATURE true
#define USE_BH1750_LUX false
#define USE_FLOW false

////////////////////////////////////
// Pin Definitions and Sensor Configuration
////////////////////////////////////

#if USE_ANALOG_PINS
    // Set of analog input pins to read
    const int analog_pins[] = {A0, A1, A2, A3, A4, A5};
#endif

#if USE_DIGITAL_PINS
    // Set of digital input pins to read
    const int digital_pins[] = {4, 5, 6};
#endif

#if USE_DS18B20_TEMPERATURE
    // Run the 1-wire bus on pin 12
    const int onewire_pin = 12;
#endif

#if USE_FLOW
    // Flow sensor pulse pin input, must be interrupt enabled
    // These are pins 0, 1, 2, 3, 7 for a Leonardo board
    // Note that Leonardo needs pins 0+1 for Serial1 and 2+3 for I2C
    const int flow_pin = 7;
    // YF-S401 sensor apparently 5880 pulses per litre (~1.7 mL per pulse)
    // Measured more like 5440, but calibration at different flow rates probably a
    → good thing to do
    //const float flow_litre_per_pulse = 1.0/5440.0;
    //const float flow_litre_per_pulse = 1.0;
#endif

////////////////////////////////////

#if USE_DS18B20_TEMPERATURE
    #include <OneWire.h>
    #include <DallasTemperature.h>
    // Initialise the 1-Wire bus
    OneWire oneWire(onewire_pin);
    // Pass our 1-Wire reference to Dallas Temperature
    DallasTemperature thermometers(&oneWire);
#endif
```

(continues on next page)

(continued from previous page)

```

#if USE_BH1750_LUX
    #include <hp_BH1750.h>
    // Reference to the BH1750 light meter module over I2C
    hp_BH1750 luxmeter;
#endif

#if USE_FLOW
    // Number of pulses read from the flow meter
    volatile unsigned long flow_count = 0;
    // Stored start time and pulse count for flow rate calculation
    unsigned long flow_start_millis = 0;
    unsigned long flow_start_count = 0;
#endif

// Variable to record last data acquisition time
unsigned long measurement_start_milis = 0;

// Variable to keep track of whether record separators (comma) needs to be
↳prepending to output
bool first_measurement = true;

#if USE_DS18B20_TEMPERATURE
    // Format a DS18B20 device address to a 16-char hex string
    String formatAddress(DeviceAddress address) {
        String hex = "";
        for (uint8_t i = 0; i < 8; i++) {
            if (address[i] < 16) hex += "0";
            hex += String(address[i], HEX);
        }
        return hex;
    }
#endif

// Print out a measurement to the serial port
void printMeasurement(String type, String id, String value, String units="") {
    // A comma separator needs to be prepended to measurements other than the first
    if (first_measurement) {
        first_measurement = false;
    } else {
        Serial.print(",");
    }
    Serial.print("{\"type\":");
    Serial.print(type);
    Serial.print("\",\"id\":");
    Serial.print(id);
    Serial.print("\",\"value\":");
    Serial.print(value);
    if (units.length() > 0) {
        Serial.print("\",\"units\":");
        Serial.print(units);
    }
    Serial.print("}");
}

#if USE_FLOW

```

(continues on next page)

(continued from previous page)

```

// Interrupt handler for a pulse from the flow meter
void flowIncrement() {
    flow_count++;
}
#endif

void setup(void)
{
    // Open serial port
    Serial.begin(115200);

    #if USE_DS18B20_TEMPERATURE
        // Initialise I2C bus
        Wire.begin();
        pinMode(owewire_pin, INPUT_PULLUP);
    #endif

    #if USE_DIGITAL_PINS
        // Configure set of digital input pins
        for (uint8_t i = 0; i < uint8_t(sizeof(digital_pins)/sizeof(digital_pins[0]));
→ i++) {
            pinMode(digital_pins[i], INPUT);
        }
    #endif

    #if USE_FLOW
        // Configure the flow meter input pin and interrupt for pulse counting
        pinMode(flow_pin, INPUT_PULLUP);
        attachInterrupt(digitalPinToInterrupt(flow_pin), flowIncrement, RISING);
        flow_start_millis = millis();
    #endif
}

void loop(void)
{
    // Record current time
    unsigned long current_millis = millis();
    // Check if it's time to take some new measurements
    if (current_millis - measurement_start_milis >= READ_INTERVAL) {
        measurement_start_milis = current_millis;
        // The first measurement in this cycle doesn't need a comma delimiter_
→prepending
        first_measurement = true;

        // Print message start
        Serial.print("{\"board\":\"" + String(BOARD_ID_STRING) + "\",");
        Serial.print("\"timestamp\":\"" + String(measurement_start_milis) + "\",");
        Serial.print("\"message\":\"measurement\", \"data\":[");

        //////////////////////////////////////
        // Arduino Digital Pins
        //////////////////////////////////////
        #if USE_DIGITAL_PINS
            // Read digital pins
            unsigned int d = 0;
            for (uint8_t i = 0; i < uint8_t(sizeof(digital_pins)/sizeof(digital_
→pins[0])); i++) {

```

(continues on next page)

(continued from previous page)

```

        d += digitalRead(digital_pins[i]) << i;
    }
    printMeasurement("digital", "0", String(d));
#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Arduino Analog Pins
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#if USE_ANALOG_PINS
    // Read analog pins
    for (uint8_t i = 0; i < uint8_t(sizeof(analog_pins)/sizeof(analog_pins[0]));
→ i++) {
        printMeasurement("analog", String(i), String(analogRead(analog_pins[i])));
    }
#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DS18B20 Temperature Probes
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#if USE_DS18B20_TEMPERATURE
    // We'll reinitialise the temperature probes each time inside the loop so
→that
    // devices can be connected/disconnected while running
    thermometers.begin();
    // Temporary variable for storing 1-Wire device addresses
    DeviceAddress address;
    // Grab a count of temperature probes on the wire
    unsigned int numberOfDevices = thermometers.getDeviceCount();
    // Loop through each device, set requested precision
    for(unsigned int i = 0; i < numberOfDevices; i++) {
        if(thermometers.getAddress(address, i)) {
            thermometers.setResolution(address, 12);
        }
    }
    // Issue a global temperature request to all devices on the bus
    if (numberOfDevices > 0) {
        thermometers.requestTemperatures();
    }
    // Loop through each device, print out temperature data
    for(unsigned int i = 0; i < numberOfDevices; i++) {
        if(thermometers.getAddress(address, i)) {
            printMeasurement("temperature", formatAddress(address),
→String(thermometers.getTempC(address), 2), "C");
        }
    }
#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// BH1750 Lux Meter
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#if USE_BH1750_LUX
    // Attempt to initialise and read light meter sensor
    if (luxmeter.begin(BH1750_TO_GROUND)) {
        luxmeter.start();
        printMeasurement("lux", "0", String(luxmeter.getLux(), 0), "lux");
    }
#endif

```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
// Fluid Flow Meter
////////////////////////////////////
#ifdef USE_FLOW
    unsigned long flow_end_count = flow_count;
    unsigned long flow_end_millis = millis();
    // Total volume in sensor pulses
    printMeasurement("flow_volume", "0", String(flow_end_count, 4), "pulses");
    // Current flow rate in pulses per minute
    float flow_rate = 60000.0*(flow_end_count - flow_start_count)/(float)(flow_
    end_millis - flow_start_millis);
    printMeasurement("flow_rate", "0", String(flow_rate, 4), "pulses/min");
    flow_start_count = flow_end_count;
    flow_start_millis = flow_end_millis;
#endif

    // Print message end
    Serial.println("]");
}
}

```

Parameters

- **port** – Path of serial device to use. A partial name to match can also be provided, such as “usb”.
- **board_id** – ID label provided by the Arduino data logging board, to select a particular device in case multiple boards are connected.

class SerialHandler (*parent*)

Bases: sphinx.ext.autodoc.importer._MockObject

A class used as a `asyncio Protocol` to handle lines of text received from the serial device.

Parameters *parent* – The parent `SerialDataSource` class.

handle_line (*line*)

Accept one line of text, parse it to extract data, and pass the data on to any connected sinks.

Parameters *line* – Line of text to process.

datalogd.plugins.thorlabspm100_datasource module

```

class datalogd.plugins.thorlabspm100_datasource.ThorlabsPM100DataSource (sinks=[],
se-
ri-
al-
num-
ber=None,
usb_vid='0x1313',
usb_pid='0x8078',
in-
ter-
val=1.0)

```

Bases: `datalogd.DataSource`

Provide data from a Thorlabs PM100 laser power meter.

This uses the VISA protocol over USB. On Linux, read/write permissions to the power meter device must be granted. This can be done with a udev rule such as:

Listing 2: `/etc/udev/rules.d/52-thorlabspm100.rules`

```
SUBSYSTEMS=="usb", ACTION=="add", ATTRS{idVendor}=="1313", ATTRS{idProduct}=="8078
→", OWNER="root", GROUP="usbusers", MODE="0666"
```

where the `idVendor` and `idProduct` fields should match that listed from running `lsusb`. The `usbusers` group must be created and the user added to it:

```
groupadd usbusers
usermod -aG usbusers yourusername
```

A reboot will then ensure permissions are set and the user is a part of the group.

Parameters

- **serialnumber** – Serial number of power meter to use. If `None`, will use the first device found.
- **interval** – How often to poll the sensors, in seconds.

close()

Close the connection to the power meter.

read_power()

Read power and send data to any connected sinks.

datalogd.plugins.timestamp_datafilter module

class `datalogd.plugins.timestamp_datafilter.TimestampDataFilter(sinks=[])`

Bases: `datalogd.DataFilter`

Add or update a timestamp field to data using the current system clock.

receive(data)

Accept the provided `data` and add a timestamp field.

If `data`, or elements in the `data` list are `dicts`, then a “timestamp” field will be added. Otherwise, the `data` entries will be converted to a `dict` with the old entry stored under a “value” field.

Parameters `data` – Data to add a timestamp to.

CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`

d

`datalogd`, [23](#)
`datalogd.plugins`, [25](#)
`datalogd.plugins.aggregator_datafilter`,
 [25](#)
`datalogd.plugins.csv_datafilter`, [26](#)
`datalogd.plugins.file_datasink`, [27](#)
`datalogd.plugins.influxdb2_datasink`, [28](#)
`datalogd.plugins.influxdb_datasink`, [29](#)
`datalogd.plugins.join_datafilter`, [30](#)
`datalogd.plugins.keyval_datafilter`, [30](#)
`datalogd.plugins.libensors_datasource`,
 [31](#)
`datalogd.plugins.logging_datasink`, [32](#)
`datalogd.plugins.matplotlib_datasink`,
 [32](#)
`datalogd.plugins.picotc08_datasource`,
 [33](#)
`datalogd.plugins.print_datasink`, [33](#)
`datalogd.plugins.randomwalk_datasource`,
 [34](#)
`datalogd.plugins.serial_datasource`, [34](#)
`datalogd.plugins.thorlabspm100_datasource`,
 [39](#)
`datalogd.plugins.timestamp_datafilter`,
 [40](#)

A

acquire_data() (data-
logd.plugins.picotc08_datasource.PicoTC08DataSource
method), 33

AggregatorDataFilter (class in data-
logd.plugins.aggregator_datafilter), 25

datalogd.plugins.aggregator_datafilter
(module), 25

datalogd.plugins.csv_datafilter (module),
26

datalogd.plugins.file_datasink (module),
27

B

BEEP_ENABLE (datalogd.plugins.libsensors_datasource.LibSensorsFeatureType
attribute), 31

datalogd.plugins.influxdb2_datasink
(module), 28

datalogd.plugins.influxdb_datasink (mod-
ule), 29

C

close() (datalogd.DataLogDaemon method), 23

close() (datalogd.DataSink method), 24

close() (datalogd.DataSource method), 24

close() (datalogd.plugins.file_datasink.FileDataSink
method), 27

close() (datalogd.plugins.influxdb2_datasink.InfluxDB2DataSink
method), 28

close() (datalogd.plugins.influxdb_datasink.InfluxDBDataSink
method), 30

close() (datalogd.plugins.libsensors_datasource.LibSensorsDataSource
method), 31

close() (datalogd.plugins.picotc08_datasource.PicoTC08DataSource
method), 33

close() (datalogd.plugins.thorlabspm100_datasource.ThorlabsPM100DataSource
method), 40

connect_sinks() (datalogd.DataSource method),
24

CSVDataFilter (class in data-
logd.plugins.csv_datafilter), 26

CURR (datalogd.plugins.libsensors_datasource.LibSensorsFeatureType
attribute), 31

datalogd.plugins.join_datafilter (mod-
ule), 30

datalogd.plugins.keyval_datafilter (mod-
ule), 30

datalogd.plugins.libsensors_datasource
(module), 31

datalogd.plugins.logging_datasink (mod-
ule), 32

datalogd.plugins.matplotlib_datasink
(module), 32

datalogd.plugins.picotc08_datasource
(module), 33

datalogd.plugins.print_datasink (module),
33

datalogd.plugins.randomwalk_datasource
(module), 34

datalogd.plugins.serial_datasource (mod-
ule), 34

datalogd.plugins.thorlabspm100_datasource
(module), 39

datalogd.plugins.timestamp_datafilter
(module), 40

DataLogDaemon (class in datalogd), 23

DataSink (class in datalogd), 23

DataSource (class in datalogd), 24

disconnect_sinks() (datalogd.DataSource
method), 24

D

DataFilter (class in datalogd), 23

datalogd (module), 23

datalogd.plugins (module), 25

E

ENERGY (datalogd.plugins.libsensors_datasource.LibSensorsFeatureType

attribute), 31

F

FAN (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), 31

FileDataSink (class in *data-*
logd.plugins.file_datasink), 27

G

generate_data() (data-
logd.plugins.randomwalk_datasource.RandomWalkDataSource
method), 34

H

handle_line() (data-
logd.plugins.serial_datasource.SerialDataSource.SerialHandler
method), 39

HUMIDITY (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), 31

I

IN (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), 31

InfluxDB2DataSink (class in *data-*
logd.plugins.influxdb2_datasink), 28

InfluxDBDataSink (class in *data-*
logd.plugins.influxdb_datasink), 29

INTRUSION (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), 31

J

JoinDataFilter (class in *data-*
logd.plugins.join_datafilter), 30

K

KeyValDataFilter (class in *data-*
logd.plugins.keyval_datafilter), 30

L

LibSensorsDataSource (class in *data-*
logd.plugins.libsensors_datasource), 31

LibSensorsFeatureType (class in *data-*
logd.plugins.libsensors_datasource), 31

listify() (in module *datalogd*), 25

LoggingDataSink (class in *data-*
logd.plugins.logging_datasink), 32

M

main() (in module *datalogd*), 25

MatplotlibDataSink (class in *data-*
logd.plugins.matplotlib_datasink), 32

N

NullDataFilter (class in *datalogd*), 24

NullDataSink (class in *datalogd*), 24

NullDataSource (class in *datalogd*), 24

P

parse_dot_json() (in module *datalogd*), 25

PicoTC08DataSource (class in *data-*
logd.plugins.picotc08_datasource), 33

POWER (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), 31

PrintDataSink (class in *data-*
logd.plugins.print_datasink), 33

R

RandomWalkDataSource (class in *data-*
logd.plugins.randomwalk_datasource), 34

read_power() (data-
logd.plugins.thorlabspm100_datasource.ThorlabsPM100DataSource
method), 40

read_sensors() (data-
logd.plugins.libsensors_datasource.LibSensorsDataSource
method), 31

receive() (*datalogd.DataSink* method), 24

receive() (*datalogd.NullDataFilter* method), 24

receive() (*datalogd.plugins.aggregator_datafilter.AggregatorDataFilter*
method), 26

receive() (*datalogd.plugins.csv_datafilter.CSVDataFilter*
method), 26

receive() (*datalogd.plugins.file_datasink.FileDataSink*
method), 27

receive() (*datalogd.plugins.influxdb2_datasink.InfluxDB2DataSink*
method), 29

receive() (*datalogd.plugins.influxdb_datasink.InfluxDBDataSink*
method), 30

receive() (*datalogd.plugins.join_datafilter.JoinDataFilter*
method), 30

receive() (*datalogd.plugins.keyval_datafilter.KeyValDataFilter*
method), 31

receive() (*datalogd.plugins.logging_datasink.LoggingDataSink*
method), 32

receive() (*datalogd.plugins.matplotlib_datasink.MatplotlibDataSink*
method), 32

receive() (*datalogd.plugins.print_datasink.PrintDataSink*
method), 34

receive() (*datalogd.plugins.timestamp_datafilter.TimeStampDataFilter*
method), 40

S

send() (*datalogd.DataSource* method), 24

SerialDataSource (class in *data-*
logd.plugins.serial_datasource), 34

SerialDataSource.SerialHandler (class in
datalogd.plugins.serial_datasource), 39

T

TEMP (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), [31](#)

ThorlabsPM100DataSource (class in *data-*
logd.plugins.thorlabspm100_datasource),
[39](#)

TimeStampDataFilter (class in *data-*
logd.plugins.timestamp_datafilter), [40](#)

type (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), [32](#)

U

units (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), [32](#)

UNKNOWN (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), [31](#)

V

VID (*datalogd.plugins.libsensors_datasource.LibSensorsFeatureType*
attribute), [31](#)