pysungrow 0.1 Manual

Documentation for pysungrow version 0.1, a tool for communication with solar power devices

Table of Contents

1 Overview

`pysungrow` is is a Python library and script for interaction with power management devices with serial interfaces. The name comes from its initial use with Sungrow solar power devices using their RS-485 interface, but has grown to accommodate other devices. It can be used to:

- Log the status, history, or configuration of devices
- Troubleshoot interaction with devices
- Configure devices

`pysungrow` running on a Gumstix Overo Computer-on-Module (COM) has also been used to continually log solar power system status in the field. If you are working with an already-existing installation, you might want to skip ahead to Chapter 4 [Logging with an Overo COM], page 10.

1.1 Supported devices

The following devices are supported in this release:

- Sungrow SD4860 charge controller
- Sungrow SN481KS inverter
- OutBack FlexMax 60/80 charge controller
- ASP Allegro TC10/48 inverter

1.2 Supported platforms

`pysungrow` has been tested on Linux with Python 2.7.1 and 2.6.8, on Overo COM with Python 2.6.6 and on Windows 7 with Python 2.7.3; earlier and later versions may work. Unattended logging requires `cron`, and therefore is not supported on Windows without some fussing.

2 Installation

On all platforms, once dependencies have been installed, installing `pysungrow` is just a matter of unpacking the source distribution and then saying, on UNIX,

```
$ python setup.py build && su -c 'python setup.py install'
```

or on Windows

```
$ python setup.py build
```

and then, as a user with sufficient privileges,

```
$ python setup.py install
```

Once the package is installed, on most platforms typing `sungrow -h` should give you a help message (except on Windows, in which case running the script is a little more complicated; see the section "Running the script on Windows" under Scripts, below).

2.1 Dependencies

`pysungrow` is a Python library and requires Python and two external packages, PySerial (version >= 2.5) and PyYAML. Pre-compiled binaries for these packages may be available for your platform (Windows, most Linux distributions). PySerial is available from http://pypi.python.org/pypi/pyserial. PyYAML can be installed to use a fast C-based library, libYAML, or as a Python-only version. To install them, go to http://pyyaml.org/wiki/PyYAML to get the source and follow the instructions there; if you want libYAML, you can get that from http://pyyaml.org/wiki/LibYAML.

For building from source on UNIX, the following instructions may work but could be out of date (the $ indicates the command prompt; don't type it):

```
$ wget http://pyyaml.org/download/libyaml/yaml-0.1.4.tar.gz
$ tar -xzf yaml-0.1.4.tar.gz
$ cd yaml-0.1.4
$ ./configure && make && su -c 'make install'
```

and then to install PyYAML:

```
$ wget http://pyyaml.org/download/pyyaml/PyYAML-3.10.tar.gz
$ tar -xzf PyYAML-3.10.tar.gz
$ cd PyYAML-3.10
$ python setup.py build && su -c 'python setup.py install'
```

If you have an old version of PySerial installed, first get rid of it with

```
$ su -c 'rm -fr /usr/lib/python2.6/site-packages/{pyserial,serial}*'
```

Then say

```
$ wget http://pypi.python.org/packages/source/p/pyserial/pyserial-2.6.tar.gz
$ tar -xzf pyserial-2.6.tar.gz
$ cd pyserial-2.6
$ python setup.py build && su -c 'python setup.py install'
```

2.2 Installation on Windows

If you are on Windows, there may be pre-built packages available for you on those websites. If not, you can download the sources and install from source (no C compiler is required for the Python packages). You may need something like 7-zip or Winzip to unpack the source archives (`*.tar.gz`, analogous to `*.zip`).

2.3 Installation on Overo COM

Several necessary packages do not seem to be available via the pre-configured Gumstix package feeds. You may have better luck with the Angstrom feeds at http://www.angstrom-distribution.org/feeds. If not, all the packages here compile cleanly on the Overo COM, so following the instructions for installation from source should work fine.

The standard UNIX utility `cron` may not be installed on your Overo; it is needed for unattended logging. To find out, try typing

```
$ crontab -l
```

at the Overo command prompt and you should get either your currently installed cron jobs or "no crontab for user" If you get something like

```
-bash: crontab: command not found
```

you probably need to install cron. Look for a package in the Angstrom base feed, then download and install with

```
$ su -c 'opkg install cron_3.0pl1-r8.6_armv7a.ipk'
```

substituting the name of the package you downloaded. Alternatively, you could probably build Vixie cron from source, available at ftp://ftp.isc.org/isc/cron/cron_4.1.shar, but I have not tried this.

2.4 Testing

`pysungrow` has a test suite to make sure the software itself is working okay. There isn't much point in running the test suite unless you are on a new platform. If you want to run the tests there are instructions in the README file in the distribution on how to do so; you'll need to install `nosetests`.

2.5 Hardware considerations

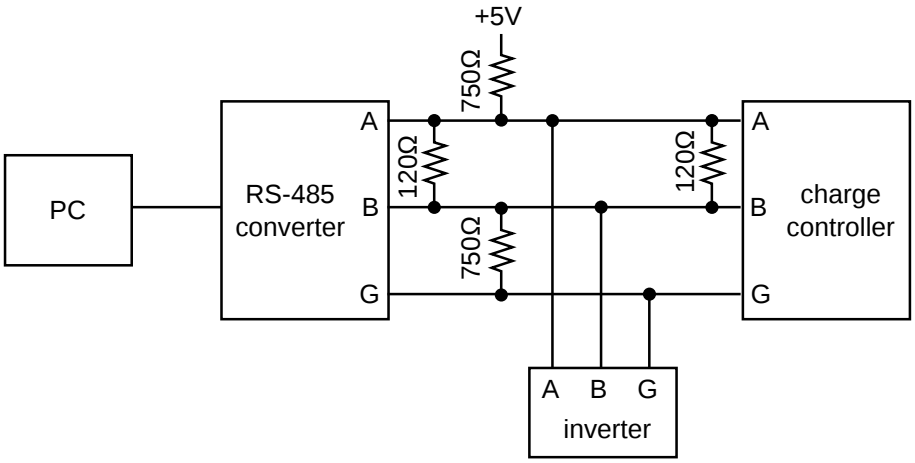
Sungrow devices communicate through a 3-wire RS-485 interface. To connect a device to a computer, it's necessary to use some kind of converter. Unfortunately, they don't all work. Three options for the interface are:

- RS-232 to RS-485 converter (from built-in RS-232 port on PC)
- USB to RS-485 converter
- USB to RS-232 converter, RS-232 to RS-485 converter

USB to RS-232 converters vary because of the looseness of the RS-232 specification, and the last option has the lowest likelihood of success.

Normally, 120 ohm termination resistors are required between lines A and B at the beginning and end of the RS-485 bus. For example, if you have a computer at one end of the bus, an inverter linked in to the middle, and a charge controller at the other end of the bus, a 120 ohm resistor should be connected between lines A and B near the RS-485 interface of the computer and another near the RS-485 interface of the charge controller.

Biasing resistors, say 750 ohms, from line A to +5 volts and from line B to ground, may be necessary if there are unexpected zero bytes at the beginning or end of messages. Such zero bytes will be ignored by `pysungrow` if they do sneak into the output.



3 Scripts

`pysungrow` includes a single script, `sungrow`. Its call signature is `sungrow [OPTIONS] FILE` where `FILE` is a system configuration file, described further in the next section.

Options are:

- `-h`, `--help`: show help message
- `-v`, `--verbose`: report more about what is being done (repeat for more verbosity, e.g., `sungrow -vvv`)
- `-l LOG_FILE`, `--error-log LOG_FILE`: log errors to `LOG_FILE`, default stderr
- `-e`, `--emulate`: emulate configured devices until terminated, at period specified in configuration file

3.1 The configuration file

The configuration file is a document describing the system to be logged or emulated. The same configuration file can be used for logging or emulation depending on the command-line `-e` or `--emulate` switch (port numbers may need to be changed). The configuration is written in YAML, a human-readable data serialization language. Its syntax is described on the YAML website, but it is straightforward enough that a few examples are usually enough to get you started. One important thing to keep in mind is that in many contexts indentation matters, that is, it is important to match the indentation among items in the document. Any extraneous fields in the mappings are ignored, so look out for typos!

The configuration file contains several required parameters: the period to be used for emulation or logging, the devices to be emulated or logged, and the `actions` to be performed in each logging episode if run as a logging system. If the configuration file is used for an emulation system (using `--emulate`), this section of the configuration is ignored and instead each configured device is configured to handle new messages on every iteration.

The `period` field specifies the interval between logging or emulation episodes, and consists of a value and unit, where unit may be "h" for hours, "m" for minutes or "s" for seconds. For example, "1 h" means 1 hour, "0.25 s" means 0.25 seconds.

The `devices` field contains a mapping of device names to device descriptions. Each device description in turn has the fields "device_type", "port" and "data_streams". The `device_type` is the name of the device type according to `pysungrow`, and as of this release may be one of `sungrow_charge_controller`, `sungrow_inverter`, `outback_charge_controller` or `allegro_inverter`. If device type name ends with `_emulator`, the corresponding emulator type is created, regardless of the `--emulate` command-line flag. This can be useful if you want emulators and non-emulators to talk to eachother within a single process (usually for debugging or configuration testing).

The `port` is the port descriptor, which for serial devices will depend on your platform. On Linux, with a USB-to-serial converter this might be `/dev/ttyUSB0` (check `dmesg`); on Windows, try `COM1` for a built-in serial port or some higher `COM` port for a USB-to-serial converter (refer to Device Manager). Any configuration settings for the ports can be set here, using the syntax for PySerial's `serial_for_url`, described in the PySerial documentation. Other URLs accepted by PySerial will work here too (e.g., `loop://` or `socket://`); these are currently used primarily for testing. Finally (but importantly!), `data_streams` is a mapping of message names to file names, indicating where you want data to be logged. *If* `data_streams` *is omitted for a message type, messages of that type will not be logged.* Therefore, at a minimum you probably want to specify a data stream for `status_page` and possibly also for `history_page` and `configuration_page`. Specifying log files for message types that do not exist for a particular device causes no problems, you just won't get any output in that file. File names may give an absolute position in the file system, meaning that on UNIX, they may start with `/`, or on Windows with a drive designation such as `C:`. If a file name

is relative, it is interpreted *relative to the directory in which the script is run*, so that if the script is run in your home directory, the data files will end up in your home directory.

The `actions` field is a sequence of descriptors for actions to perform on each iteration. The action descriptors have the fields "action", "condition", "device", and possibly other fields specifying parameters for the action. The `device` field is optional; if it is omitted, the action is to be performed by the system, rather than directed to a particular device. If `device` is given, it must correspond exactly to one of the device names given under `devices`, above. The `action` field is an action for the system or device, and may require parameters given as additional fields. The `condition` field controls whether an action is to be performed only under certain circumstances, and may be `system_back_online`, meaning the action should only be performed if the system detects that it has missed an iteration, or `device_back_online`, meaning that the action is performed if a previously unresponsive device has become responsive again.

Actions and their parameters are:

- System actions:
  - `handle_incoming_messages`: read all new messages from the bus and respond appropriately. For an emulator, this usually means that a response will be written back to the bus; for a logger, this means the message will be logged if it is configured under `data_streams`.
  - `sleep`: wait before performing the next action. This may be necessary in cases where there is a delay in communication with the device; for Sungrow devices, 0.5 s should be enough. Parameters: `seconds`, the number of seconds to sleep.
- Device actions:
  - `send`: send a message to the device (logger) or send a message as the device (emulator). Parameters: `message_type`, which for a logger will typically be `status_query`, `history_query`, or `configuration_query`, although `history_query` and `configuration_query` are only defined for the Sungrow charge controller.

Here is an example configuration file for a Sungrow charge controller and Sungrow inverter. If `sungrow` is run on this file as an emulator, it will check for and respond to incoming messages on the serial device `/dev/ttyUSB1` every quarter-second. If run as a logger, it will first send a `status_query` to the charge controller, then wait 0.5 seconds and log any responses to `status.csv`; then, if the system has missed a log cycle, it will send `history_query` to the charge controller, wait 0.5 seconds, and log any responses to `history.csv`. Finally, it will send a `status_query` to the inverter, wait 0.5 seconds, and log any responses to `inverter_status.csv`. This file is included in the source distribution as `doc/example_system_config.yml`.

```
## System configuration file
period: 0.25 s
devices:
  charge_controller:
    device_type: sungrow_charge_controller
    port: /dev/ttyUSB1
    data_streams:
      status_page: status.csv
      history_page: history.csv
  inverter:
    device_type: sungrow_inverter
    port: /dev/ttyUSB1
    data_streams:
      status_page: inverter_status.csv
actions:
  -
    action: send
    device: charge_controller
```

```
  message_type: status_query
-
  action: sleep
  seconds: 0.5
-
  action: handle_incoming_messages
-
  action: send
  condition: system_back_online
  device: charge_controller
  message_type: history_query
-
  action: sleep
  condition: system_back_online
  seconds: 0.5
-
  action: handle_incoming_messages
  condition: system_back_online
-
  action: send
  device: inverter
  message_type: status_query
-
  action: sleep
  seconds: 0.5
-
  action: handle_incoming_messages
```

3.1.1 Persistence of system state between script calls

When the script exits, the state of the system is saved in the directory `~/.sungrow` (`~` represents your home directory) in a file named according to a digest of the system configuration file. Each subsequent time the script is called, it computes the digest of the system configuration file given on the command line, and if it has already been called with the same file, rather than re-reading the file `sungrow` will load the saved state of the system. This allows the script to keep track of the last time it was called and which devices were read successfully on the previous run. State files for emulators and loggers are stored separately, so the state of an emulator loaded from an actions_file will not overwrite a logger from the same action file or vice versa.

The use of a digest to identify the configuration file means that it *does not matter if the configuration file is renamed*; it will still be treated as the same configuration. On the other hand, a trivial change to the configuration file (even one that does not change at all the function of the created system) will (with absurdly high probability) change its MD5 digest, so it will be regarded as a new configuration. Under some circumstances, it might be necessary to forget about the previous state of the script. An easy way to do this is to change the configuration file in some insignificant way, such as adding a space somewhere or changing the ordering of fields. Another approach is to delete the state file, which is saved in `~/.sungrow` as `logger_HASH` or `emulator_HASH` where `HASH` is the MD5 digest of the script file.

3.2 Setting up for logging

To try out your logging configuration, set up the configuration file as described above, make sure the port setting is correct, and then say

```
$ sungrow FILENAME
```

where `FILENAME` is your system configuration file, adding verbosity flags (`sungrow -vvv FILENAME`) if you want to know more about what is going on. If everything worked right, a single logging iteration will be completed, and you should get log files as configured under `data_streams` in your working directory if you used relative paths, or wherever in your filesystem you specified if you used absolute paths. If you want to try a bunch more iterations of logging, you can use the capabilities of your shell; for example, on UNIX with `bash` you could say,

```
$ for (( ; ; )); do sungrow -vvv system_config.yml; sleep 0.5; done
```

If you want to log all the time, even when your machine is restarted, see "Log-on-startup", below.

3.3 Setting up an emulator

An emulator attempts to imitate the responses of configured devices to queries. The physical content of the replies is nonsense, of course; the aim is to make sure that communication is working. When run in emulation mode, each configured device listens on the port, and when it receives a message it understands, it will respond with a canned reply. To set up an emulator for use with a logger, it's usually enough to copy the configuration file and change the port number to correspond to the port on which you want the emulator to run, and then start the emulator with `sungrow -e NEW_FILENAME`. By default, nothing will appear to happen, but the emulator will be happily waiting on the port for messages. If you want to know more about what the emulator is doing, you can add verbosity flags (e.g., `sungrow -vvv -e NEW_FILENAME`.

3.4 Setting up for log-on-startup

You can set up `sungrow` to run at specified intervals whenever your machine is on using the standard UNIX `cron` daemon. More details can be found in the documentation for `cron` on your system; if you have the `man` command and `cron`'s documentation is installed, typing `man 5 crontab` should get you a useful manual page. Wikipedia's entry for cron is good for an overview.

To schedule runs of `sungrow`, you'll need to edit your crontab entry using `crontab -e`. This will open an editor in which you can add a line describing when you want `sungrow` to run. The first field specifies on which minutes of the hour to run the job, the second field which hours, and so on; an asterisk `*` indicates always. The notation `*/n` where n is an integer means that the job should run every `n` time units, so for example

```
*/1 * * * * cd ${HOME}/sungrow_data; sungrow -l errors.log system_config.yml \
   >/dev/null 2>&1
```

would change to the `sungrow_data` subdirectory of your home directory (this will fail if it doesn't exist!) and run the job every minute. (The backslash `\` means line continuation and is included here to make the command fit on the page; in the crontab you can put it all on one line.) If you instead wanted the script to run every 10 minutes, you would enter `*/10` in the first column, and so on.

The shell redirection at the end of the command, `>/dev/null 2>&1`, indicates that you don't want any output e-mailed to you or appearing in the system logs. This is a good idea if you are running the job very frequently for testing, but it also works generally provided that you have specified `-e` or `--error-log`, because that means that all error messages will also be sent to the specified error log file. However, if there is an error of some other kind in the command you gave (for example, if `${HOME}/sungrow_data` doesn't exist), you will get *no indication at all*. Therefore, for first tries you might consider eliminating `2>&1` from the end of the command, in which case any errors will go to your system log or be e-mailed to you depending on how your system is configured.

To check if you just installed what you think you did, you can read the contents of your `crontab` file with `crontab -l`. Then wait a little while (longer than whatever interval you specified) and you should start seeing data output in the files you specified in `data_streams` in the configuration file, and debugging output in `${HOME}/sungrow_data/errors.log`.

3.5 Running the script on Windows

Scripts will not be automatically placed in the system path and made executable on Windows. So instead, you may need to use `sungrow.bat` — this is a Windows batch file. To use it, open a command prompt (Windows menu -> All Programs -> Accessories -> Command Prompt) and change to the directory containing `sungrow.bat` using `cd`, e.g., `cd C:\Users\peater\Downloads`. Then all the examples listed here should work with `sungrow.bat` substituted for `sungrow`. Log-on-startup will not work of course without `cron`.

4 Logging with an Overo COM

4.1 Communication with the Overo COM

Field deployments recording solar power system status with `pysungrow` running on an Overo COM introduce the additional step of communicating with the COM. Instructions on communication with your Overo COM via a serial port are available in the Gumstix online documentation. The essential details are that the connection is 115200 bps, 8N1, no hardware or software flow control. You need a mini-B to standard-A USB cable and a free USB port on your computer. Suitable terminal programs on Linux include `cKermit` or `minicom`; on Windows you can use `PuTTY`, available on the PuTTY homepage.

You will need to determine which COM port (on Windows) or device node (on Linux) corresponds to the serial connection. On both Windows and Linux, a serial device often will be given the same port name on subsequent connections, but this is not guaranteed. Therefore, to first determine the correct device name, you may need to connect to the powered-on Overo COM, then check `dmesg` on Linux or Device Manager on Windows. On Windows, a quick way to open device manager is by selecting Start->Run, then typing `devmgmt.msc`.

Once you know the device name, you can enter this in your terminal program; make the connection, then restart the Overo COM by pressing the reset button, and you should see its boot messages on your terminal.

In the field, you might instead connect via internet, using `ssh` on Linux or `PuTTY` on Windows. You need to know the static IP of the Overo to do this; in our field installations we record the static IP of the Overo on the outside of its box. If the Overo is connected to a network running DNS, then by default you can connect to it as `overo.local`, but without DNS you need to know the static IP of the Overo. If your computer does not have an internet connection either, you may need to set the IP address of the computer too.

4.2 Setting the clock on an Overo COM

On Overo COM, it's possible that the clock and timezone are not set correctly, especially if it is not connected to the internet. This will make the dates for logged data misleading. To check this, say

```
$ date
```

and see if you get something reasonable; if not, you'll need to update the clock and possibly the timezone. If the timezone is wrong, adjust to Singapore time (GMT-8) with

```
$ su -c 'rm /etc/localtime; ln /usr/share/zoneinfo/Etc/GMT-8 /etc/localtime'
$ su -c 'echo SGT > /etc/timezone'
```

Then to fix the date via network time protocol, say

```
$ su -c 'rdate -s time.mit.edu'
```

or, if you do not have a working network connection,

```
$ su -c "date -s 'Fri Feb 22 08:44:00 2013'"
```

or similar. To check that your date string will be interpreted as you expect, you can try it first with

```
$ date -d 2013-02-22T08:44
Fri Feb 22 08:44:00 SGT 2013
```

or similar.

5 Device-specific configuration

The devices supported by `sungrow` have different capabilities for recording system information, and different interfaces for communicating that information. `sungrow` attempts to hide the trivial differences from the user, and present a unified interface and output format for the different devices. However, the system configuration file needs to be modified in some ways to communicate with different devices, both to allow retrieval of information that is available from some devices but not from others, and also to provide some flexibility for interacting with similar devices that are not yet officially supported.

The Sungrow SD4860 charge controller and Sungrow SN481KS inverter have rich interfaces for interacting with a computer through a RS-485 interface. However, these interfaces are documented exclusively in Chinese, and because my Chinese is limited, so is the coverage and translation of some of the fields in messages and replies. Help here is welcome! However, all basic uses are covered. The OutBack FlexMax 60/80 charge controller and ASP Allegro TC10/48 inverter have much simpler interfaces that involve periodic transmission of a status page and, in the case of the inverter, receipt of some simple commands.

Details of all message types understood by `sungrow` for all supported devices are documented exhaustively, if cryptically, in `device_types.yml` in the source distribution. Because `sungrow` uses this file to interpret and compose messages, changes to this file will affect `sungrow`'s behaviour on subsequent builds — so modify with care.

5.1 Sungrow charge controller and inverter

The Sungrow SD4860 charge controller and Sungrow SN481KS inverter both operate at 9600 bps and with otherwise default communication settings, so nothing special is required in describing the devices besides specifying the correct port settings; see the example system configuration file in Chapter 3 [Scripts], page 5.

Both the charge controller and inverter have sophisticated interfaces involving multiple message types. The SD4860 charge controller understands the following message types:

- `status_query`: Charge controller responds with information about current solar charging current, battery voltage, load current, etc.
- `history_query`: Charge controller responds with daily statistics of some state parameters: battery voltage minimum and maximum, solar charge production, load charge consumption.
- `configuration_query`: Charge controller responds with all its current configuration parameters, such as thresholds for overvoltage and undervoltage.
- `configuration_setting`: Sets configuration of charge controller; requires all parameters returned by a configuration query. Not yet implemented.

The SN481KS inverter understands the following message types:

- `status_query`: Inverter responds with status and configuration information including AC output voltage, current, and frequency, DC input voltage and current, undervoltage and overvoltage thresholds, machine type information and current datetime.
- `inverter_start`: Starts the inverter. Untested.
- `inverter_stop`: Stops the inverter. Untested.
- `alarm_sound_test`: Tests the alarm sound. Untested.
- `alarm_light_test`: Tests the alarm light. Untested.
- `dismiss_alarm`: Turns off the alarm. Untested.
- `set_datetime`: Sets the datetime of the inverter. Untested.
- `test_lcd_backlight`: Tests the LCD backlight on the inverter. Untested.

Note that most of the command messages have not been tested.

5.2 OutBack charge controller

The FlexMax 60/80 charge controller is described on the FlexMax charge controller product page. It has a very simple interface, with only one message type — the status page returned by the charge controller. There is no status query for this charge controller because it automatically sends a status page every second when the serial communication flag RTS is set low.

What this means for the configuration file is that the only command required is `handle_incoming messages` with a delay (`sleep`) of more than 1 second. However, the OutBack only sends messages if RTS is set low, requires DSR / DTR hardware handshaking, and baud rate needs to be set to 19200 bps. An example configuration file is given in the next section.

5.3 Allegro inverter

The ASP Allegro TC10/48 inverter has a very simple interface, like the FlexMax 60/80 charge controller; it sends status messages every 1 to 2 s, and therefore requires a delay (`sleep`) of about 2 s to get a message. The product documentation is not precise regarding the syntax of the status message, but it should be interpreted correctly by the current version of `pysungrow`. Communication is at 4800 bps.

The TC10/48 inverter interface also provides 3 commands, which allow changing the power level of the inverter, or disabling it entirely:

- `enable_remote`: Activate receipt of commands (necessary before issuing `set_standby_level`; disables the front panel potentiometer on the inverter.
- `disable_remote`: Disable receipt of commands (except `enable_remote`; enables the front panel potentiometer on the inverter.
- `set_standby_level`: Adjust the power level of the inverter. 99 means off, 00 means continuous operation, and 01–98 set some intermediate standby level.

The following example system configuration file will save OutBack status messages to `status.csv` and Allegro inverter status messages to `inverter_status.csv`.

```
## System configuration file for OutBack and Allegro
period: 1 s
devices:
  charge_controller:
    device_type: outback_charge_controller
    port: /dev/ttyUSB0
    port_settings:
      baudrate: 19200
      dsrdtr: True
    data_streams:
      status_page: status.csv
  inverter:
    device_type: allegro_inverter
    port: /dev/ttyUSB1
    port_settings:
      baudrate: 4800
    data_streams:
      status_page: inverter_status.csv
actions:
  -
    action: set_port_flag
    device: charge_controller
    flag: RTS
    level: False
```

```
-
  action: sleep
  seconds: 2
-
  action: handle_incoming_messages
```

6 Using pysungrow from the Python command line

Online help for all object can be obtained using the Python builtin `help()` function; for example,

```
>>> import sungrow.bus
>>> help(sungrow.bus.Bus)
```

will provide a description of the `Bus` class in the `sungrow.bus` module.

[Further documentation of Python API to be inserted here]

7 Troubleshooting

`sungrow` can be used to diagnose communication issues with devices. Simple verification that the code is working, absent communication issues, can be done with the test suite (see Chapter 2 [Installation], page 2). For further troubleshooting, a first step is to call `sungrow -vvv CONFIG_FILE` on the command line (note all the verbosity flags) and see what is going on. In cases where devices are completely unresponsive or responses are garbled, another good strategy is to put `sungrow` aside and try communicating with the device directly using a terminal program (see Chapter 4 [Logging with an Overo COM], page 10 for some suggested terminal programs).

Sungrow provides a Windows application for reading its charge controllers (installer). To test the sungrow charge controller emulator, connect an RS-232 null modem cable between two serial ports on the machine, COMx and COMy, where x and y substitute for their respective COM port numbers. Create a system configuration file `CONFIG_FILE` for an emulator to run on COMx, then say `sungrow -e CONFIG_FILE` to run an emulator in loop mode on COMx, and run the Windows software on COMy. The Windows software should be able to show the (phony) history and status provided by the emulator. COMx and COMy need not be on the same machine, that is to say, the Windows software and the emulator may run on different computers. For example, suppose you have two serial ports available, `COM1` and `COM2`. Connect these two with a null modem serial cable. Then start `sungrow` in emulation mode listening to `COM1` by setting `port` to `COM1` in CONFIG_FILE, and run:

```
sungrow -vvve CONFIG_FILE
```

The emulator, in loop mode, will wait forever for a command on COM1 and then reply. It only replies through the port, so unless the very-verbose flags are set (`-vvv`) nothing will ever appear on the terminal, and it will look like the terminal is frozen. Then open the Sungrow Windows software, and set it to listen to `COM2`. Its status reports should show the canned replies of the emulator, and a transcript of the exchange from the emulator's side should appear on the terminal via stderr. The script is in loop mode, so you'll have to terminate it.

The next logical step is to test whether `sungrow` also successfully communicates with the `sungrow` emulator. To do this, you can just copy CONFIG_FILE to a new file NEW_FILE and change `port` from `COM1` to `COM2`. You'll also need to add some actions to the configuration, say a `status_query` followed by a `sleep` and `handle_incoming_messages` (see "The configuration file", above). Then in one terminal, type:

```
sungrow -vvve CONFIG_FILE
```

Next, open another terminal and type

```
sungrow -vvv NEW_FILE
```

If all goes well, you should see a transcript of each end of the exchange on the corresponding terminal.

8 Bugs and limitations

If you have exhausted the troubleshooting steps in this guide and your problem has not been solved, please file a ticket on the peatflux wiki at http://peatflux.censam.org/projects/peatflux.

8.1 Known bugs and limitations

- `cron` cannot schedule logging more frequently than once per minute. If you want to run the script more frequently for testing, you can use standard shell capabilities, for example, with `bash`,

  ```
  for (( ; ; )); do sungrow -vvv system_config.yml; sleep 0.5; done
  ```

  will run the script every 0.5 s.
- Log on startup: the period specified in the `cron` job should match `period` in the system configuration file. For logging, the entry in the system configuration file is used to determine whether the system has missed a logging episode, which in turn controls `system_back_online`. So if the periods in the `cron` job and configuration file don't match, the `system_back_online` condition will not be triggered accurately.