SolarThing

Lavender Shannon

Apr 24, 2024

ABOUT

SolarThing is an application that can monitor data from a variety of solar charge controllers and inverters. SolarThing supports running in Docker and also supports a native install. Code and issues available at https://github.com/wildmountainfarms/solarthing.

SolarThing targets monitoring off-grid solar installations. The Renogy Rover and EPEver Tracer are typically used for smaller scale off-grid setups, so you will likely not use SolarThing on a larger install or residential install. The Outback MATE 1/2 is also supported, but is far less common than the cheaper charge controllers.

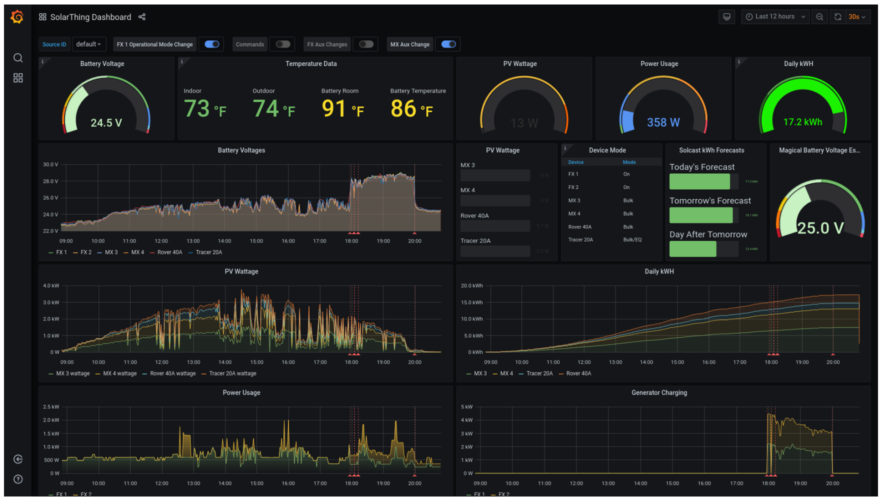
To jump to installation, go to *Installation*.

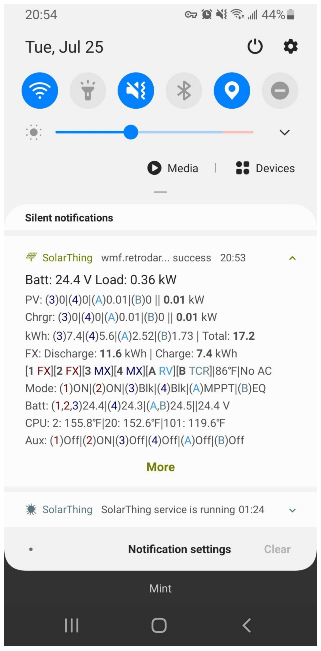
CHAPTER

ONE

INTRO

SolarThing is fully configurable. Create a JSON configuration file to configure how and where data from your charge controller goes! Once configured, charge controller data is continuously uploaded to a database for viewing in Grafana, or your choice of data visualization.




20:54 44%

Tue, Jul 25

Media | Devices

Silent notifications

SolarThing wmf.retrodar... success 20:53

Batt: 24.4 V Load: 0.36 kW

PV: (3)0|(4)0|(A)0.01|(B)0 || **0.01** kW

Chrgr: (3)0|(4)0|(A)0.01|(B)0 || **0.01** kW

kWh: (3)7.4|(4)5.6|(A)2.52|(B)1.73 | Total: **17.2**

FX: Discharge: **11.6** kWh | Charge: **7.4** kWh

[**1** FX][**2** FX][**3** MX][**4** MX][**A** RV][**B** TCR]|86°F|No AC

Mode: (1)ON|(2)ON|(3)Blk|(4)Blk|(A)MPPT|(B)EQ

Batt: (1,2,3)24.4|(4)24.3|(A,B)24.5||24.4 V

CPU: 2: 155.8°F|20: 152.6°F|101: 119.6°F

Aux: (1)Off|(2)ON|(3)Off|(4)Off|(A)Off|(B)Off

More

SolarThing SolarThing service is running 01:24

Notification settings Clear

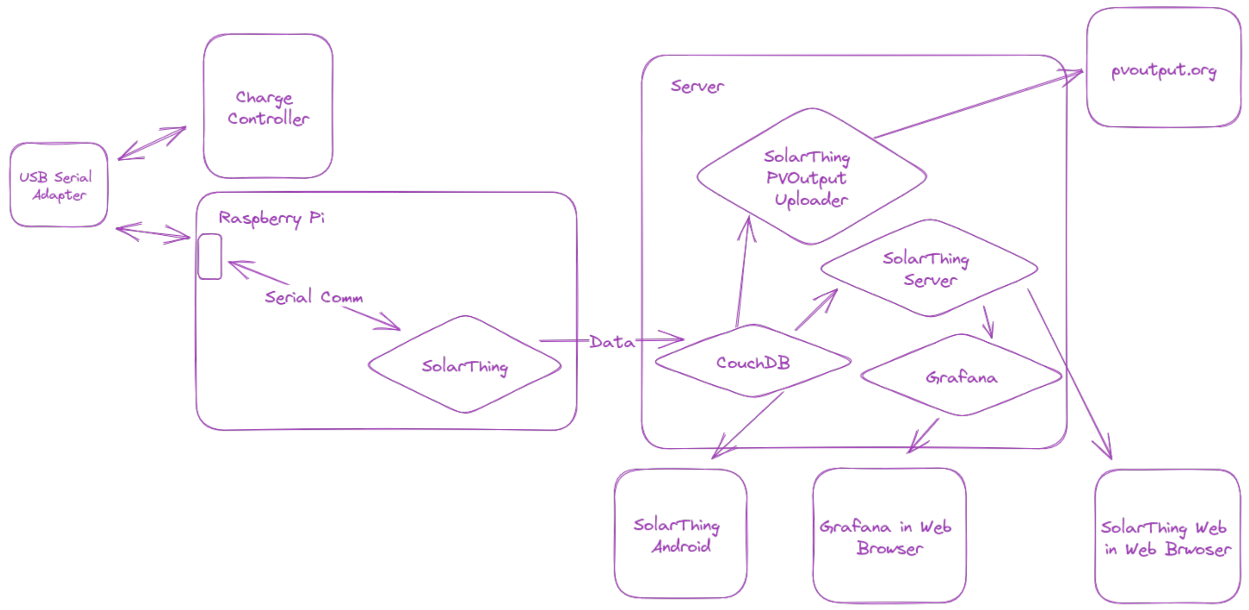
Mint

SolarThing allows you to monitor your battery voltage, incoming solar power, and power usage. Each datapoint can be graphed over time. Grafana allows you to view historical data and current data. SolarThing also supports 1-Wire temperature sensors, so you can record the indoor or outdoor temperatures. With more advanced configurations of SolarThing, it can be used as an automation system.

SolarThing supports Outback MATE 1 and 2, Renogy Rover and similar devices, EPEver Tracer charge controllers. For more information, check out *Supported Products*.

This shows an example setup of SolarThing and the connections between each component. You may make your setup however you would like, with or without all the features shown in the diagram.

To get started, go to *DietPi Setup*.


Charge Controller
USB Serial Adapter
Raspberry Pi
Serial Comm
SolarThing
Data
Server
SolarThing PVOutput Uploader
SolarThing Server
CouchDB
Grafana
pvoutput.org
SolarThing Android
Grafana in Web Browser
SolarThing Web in Web Brwoser

CHAPTER

TWO

ABOUT

2.1 Supported Products

SolarThing supports a wide variety of products. Each product requires a different configuration in SolarThing.

2.1.1 Renogy Rover Charge Controller

- Renogy: Rover, Rover Elite, Rover Boost, Wanderer, Adventurer, Dual Input DCDC Charger
- Compatible with all SRNE Solar Charge Controllers (And rebranded charge controllers)
- Compatible with **Zenith Grape** Solar Charge Controller, **PowMr** MPPT Charge Controller, **RICH** SOLAR MPPT, **WindyNations TrakMax** MPPT

Throughout the documentation, all of these products will be referred to as "rovers".

2.1.2 Outback MATE

- Outback MATE 1 and Outback MATE 2
- Allows monitoring of FX inverters and MX/FM Charge Controllers

2.1.3 EPEver Tracer Charge Controller

- Tracer AN series and TRIRON N series
- Possibly includes the BN series

2.1.4 Other Products

- DS18B20 Temperature Sensors
- PZEM-003 and PZEM-017 Shunts

2.1.5 Not Compatible With

- Inverters that are not Outback Inverters
  - This means that Renogy Inverters are not supported, even if those inverters have solar charge controller functionality.

2.2 Databases and Viewing Data

SolarThing supports multiple ways to export and upload data along with multiple ways to view that data. This page can help you choose which database to use and setup as you configure SolarThing.

2.2.1 Databases

CouchDB

Using CouchDB allows for the most features. Used by SolarThing Android. Required for the `automation` and `pvoutput` programs and for SolarThing Server.

InfluxDB

People use InfluxDB if they want to get data into Grafana and design their own queries against their data.

2.2.2 Viewing Data

SolarThing Server

All SolarThing Server configurations require CouchDB to be setup.

GraphQL and Grafana

With a CouchDB database, SolarThing's Server program can be set up to serve data to Grafana. This is used alongside https://grafana.com/grafana/plugins/retrodaredevil-wildgraphql-datasource/

SolarThing Web

SolarThing Server comes with a simple web interface ready to use out of the box. Just navigate to the IP and port where it is being hosted and

InfluxDB and Grafana

If you are using an InfluxDB database, you may configure Grafana yourself to query data from InfluxDB. Just know that SolarThing's maintainers are not well versed in InfluxDB, so do not expect much support.

PVOutput

With a CouchDB database, SolarThing's `pvoutput` program can be set up to upload data to PVOutput at regular intervals.

SolarThing Android

The Android application is available on the Google Play Store and uses the CouchDB database directly.

https://play.google.com/store/apps/details?id=me.retrodaredevil.solarthing.android

SolarThing Automation

The SolarThing `automation` program can do many things. It can send Slack messages to alert you of potential problems. It can upload data to Solcast.

This program can be configured using SolarThing's Action Language. Many features of the `automation` program are advanced and usually undocumented. This is not for beginners.

2.3 Frequently Asked Questions

2.3.1 Common Questions

Can SolarThing run on an Arduino?

No. SolarThing uses Java, which does not run on an Arduino.

Is there an ISO image I can install to my Pi to have everything set up?

No. See enhancement issue: #79.

2.3.2 Rover Questions

Can SolarThing communicate over Bluetooth?

No. There is a possibility of support in the future, but it is unlikely. See #125.

Can a Bluetooth module be connected while SolarThing is running?

No. Most renogy devices have a single comm port. When the comm port is connected to the device SolarThing is running on, the Bluetooth module cannot be connected. It is unknown if this holds true for the few Renogy devices that have two comm ports.

Is my hybrid inverter/solar charge controller supported?

No. See #137.

Where can I buy a serial adapter

There's not a good place to buy a cable. You will likely have to make a cable: *Rover RS232 Port* or *Rover RS485 Port*.

I have the config server set up. I set the battery type to user. Why can't I change some parameters?

The rover's battery type is a little bit weird. In my experience, you cannot programmatically set the battery type to user/self-customized and have it be in an "unlocked state". To get this "unlocked state", you must physically change the battery type on your charge controller, then you should be able to programmatically change parameters via the configuration server.

2.3.3 Tracer Questions

I have the config server set up. Why can't I change some parameters?

In my experience, the tracer does not allow its charging parameters to be changed. You are likely out of luck.

CHAPTER

THREE

QUICKSTART

3.1 Device Setup

Before installing SolarThing, it's a good idea to follow these instructions to get your device setup.

If you know what you're doing, you can skip right to *Installation*!

3.1.1 DietPi Setup

This page is for setting up DietPi without a keyboard and monitor attached to your Raspberry Pi (or other single board computer). This page may also be useful to gain remote access to a DietPi system that is not operating headless (headful system).

Flashing DietPi

Download DietPi here: https://dietpi.com/#download

Note: If you are choosing a DietPi version for your Raspberry Pi 2, you should likely choose the ARMv8 image, as it is unlikely that you have a Raspberry Pi 2 v1.1. (It's worth checking to see if you have the v1.1 revision, though)

Once you have your DietPi image downloaded, you can choose a tool to flash your SD card such as balenaEtcher. If you need more thorough instructions, navigate to https://dietpi.com/docs/install/#2-flash-the-dietpi-image.

Configuring DietPi SD Card to Connect to WiFi

Since the goal is to be able to put the SD card in our Raspberry Pi, then boot up the Pi and gain SSH access, we first need to configure DietPi to automatically connect to the WiFi network.

Note: If you are using Ethernet, you may skip this step

Plug the SD card into your computer, or unplug and plug it back in if you just flashed it (this makes sure it is mounted). Now, navigate to the boot partition using your file explorer, or a terminal if you prefer. You will know it is the boot partition when you see that it contains `dietpi` and `overlays` folders. In that same directory, there should be a file named `dietpi.txt`.

Edit `dietpi.txt` with your editor of choice. Find the line that has `AUTO_SETUP_NET_WIFI_ENABLED=0` and change it to `AUTO_SETUP_NET_WIFI_ENABLED=1`. Save the file.

Edit `dietpi-wifi.txt`. Begin editing Entry 0.

- Change `aWIFI_IDENTITY[0]=''` to `aWIFI_IDENTITY[0]='YourSSID'` and replace `YourSSID` with your WiFi's SSID
- Change `aWIFI_PASSWORD[0]=''` to `aWIFI_PASSWORD[0]='YourPassword'` and replace `YourPassword` with your WiFi network's password.
- Leave the other settings unchanged
- Save the file

Now plug your device in, and it should connect to your WiFi network!

Finding IP address and gaining remote access

Now that your device is connected to your local network, it has an IP address assigned by your router. The easiest way to find what that IP address is is to log into your router and find what IP address your device got (if you can't find it, try looking at DHCP leases).

Now that you have the IP address, it's time to SSH into your device. Choose the option that best describes your setup.

PuTTY on Windows

`ssh` command on Windows

`ssh` command on Linux or Mac OS

If you are using Windows, one option is PuTTY, which has a nice GUI interface to connect to a device. Download it here: https://putty.org/

After opening PuTTY, connect to your device's IP address and configure the user as `root` and the password as `dietpi`.

If you would like to use the `ssh` command directly, you may do so as long as you have it installed. The recommended way to use the `ssh` command on Windows is to install Git and choose the default options while going through the installer to also install Git Bash. To install this more quickly, you may instead run this in command prompt/PowerShell: `winget install --id=Git.Git -e`.

Run this command and replace `192.168.X.X` with your device's IP address.

```
ssh root@192.168.X.X
```

When prompted for a password, enter `dietpi`. Note that you will not see your password as you type it. This is normal.

If you are running Linux or Mac OS and want to SSH into your device from Linux or Mac OS, you can simply open a terminal to run the `ssh` command.

Run this command and replace `192.168.X.X` with your device's IP address.

```
ssh root@192.168.X.X
```

When prompted for a password, enter `dietpi`. Note that you will not see your password as you type it. This is normal.

Now you should have shell access to DietPi. Since this is the first time logging into the system, it will prompt you to configure and install software. Most of the default options are fine, but feel free to change them if you know what you are doing.

Installing Docker

During the initial installation, or after the initial installation, you should install Docker (if that's how you choose to run SolarThing - it's the recommended way to run SolarThing, after all). On most systems, you should follow install docker engine on Debian, however on a DietPi system, you can simply use `dietpi-software` to install `162 Docker` and `134 Docker Compose` (you need to install both). Or, you can simply run:

```
# install Docker
sudo dietpi-software install 162

# Install Docker Compose
sudo dietpi-software install 134
```

For more information relating to Docker on DietPi, go here: https://dietpi.com/docs/software/programming/#docker

Install SolarThing

Now that you have your device setup, head on over to *Installation*!

3.1.2 Setup for Other OS

The recommended way to setup SolarThing is to use a DietPi install (described in *DietPi Setup*). This page will help you get your system ready for a SolarThing installation if you are not using DietPi.

Gaining Remote Access

You should obtain SSH access to your system. If you need more detailed steps, you may find them in *Finding IP address and gaining remote access*.

Install Docker

Once you have access to your system, you should install docker on it. You will need to install Docker Engine on your system, which should also install docker compose for you. Go here to install docker: https://docs.docker.com/engine/install/

3.2 Installation

There are multiple ways to install SolarThing. Using Docker has been supported since version 2023.3.0 and is the recommended way to install and use SolarThing.

Docker Install

Native Install

To install SolarThing on your system using Docker, go to *Docker Install*.

To install SolarThing natively on your system, go to *Native Install*.

3.2.1 Docker Install

If you are doing a Docker install, you can choose where you would like to place your configuration files. You must also have docker installed.

Throughout the tutorials here, you will find yourself running many `docker run` commands. When you finally are ready to configure SolarThing, you will end up writing a `docker-compose.yml` file.

Testing `docker run`

To make sure you are ready for future steps in the tutorial, run this command:

```
docker run --rm ghcr.io/wildmountainfarms/solarthing version
```

You should get output similar to:

```
pi@solarpi:~$ docker run --rm ghcr.io/wildmountainfarms/solarthing version
SolarThing made by Lavender Shannon
Jar: solarthing.jar
Jar last modified: 2023-07-02T18:21:48Z
Java version: 19.0.2
```

Creating your configuration directory

To make sure you can configure SolarThing easily, you should determine what directory you would like to use to contain your configuration. You may choose to put SolarThing config in a place such as `~/Documents/containers/solarthing` or even in a place like `/opt/containers/solarthing`. It is recommend to put SolarThing configuration somewhere in your home directory so that your user has permission to edit the configuration without sudo.

For the purpose of this tutorial, I will choose `~/Documents/containers/solarthing`. We will now create a `docker-compose.yml` file so that when we configure SolarThing in future steps we can easily run and test it.

Note: In this example we are creating a subfolder of `solarthing` called main. This `main` folder should house the all of the files for a given SolarThing instance. If you would like, you can use a name other than `main` such as `rover` or `mate`, as long as it makes sense to you and the program you plan to run.

Here is an example file structure:

```
~/Documents/containers/solarthing/
├── main/
│   └── config/
│       └── base.json
└── docker-compose.yml
```

Let's make the file structure!

```
mkdir -p ~/Documents/containers/solarthing/main/{config,logs}
cd ~/Documents/containers/solarthing/
nano docker-compose.yml
```

At this point you should be editing `docker-compose.yml`. Place this inside your file:

```
version: '3.7'

services:
  main:
    image: 'ghcr.io/wildmountainfarms/solarthing:latest'
    container_name: solarthing-main
    restart: 'unless-stopped'
    command: run --base config/base.json
#    group_add: # this is only necessary if you are using a user other than root
#      - dialout
#    cap_add:
#      - SYS_RAWIO
#    devices:
#      - '/dev/ttyUSB0:/dev/ttyUSB0'
    volumes:
      - './main/config:/app/config:ro'
      - './main/logs:/app/logs'
```

Note: When running this, SolarThing will write logs files to the log directory using root:root permission (unless you change the user running SolarThing or do not mount the logs directory).

Now run `sudo docker compose up`. The program should crash with output similar to this:

```
main  | 2023-07-12 04:32:40.939 [main] INFO  me.retrodaredevil.solarthing.program.
↪SolarMain - [LOG] Beginning main. Jar: Jar: solarthing.jar Last Modified: 2023-07-
↪12T04:29:54Z Java version: 19.0.2
main  | [stdout] Beginning main. Jar: Jar: solarthing.jar Last Modified: 2023-07-
↪12T04:29:54Z Java version: 19.0.2
main  | [stderr] Beginning main. Jar: Jar: solarthing.jar Last Modified: 2023-07-
↪12T04:29:54Z Java version: 19.0.2
main  | 2023-07-12 04:32:40.990 [main] INFO  me.retrodaredevil.solarthing.program.
↪SolarMain - Using base configuration file: config/base.json
main  | 2023-07-12 04:32:40.990 [main] ERROR me.retrodaredevil.solarthing.program.
↪SolarMain - (Fatal)Base configuration file does not exist!
main exited with code 0
```

Now you are ready to continue! Head over to *Serial Port Setup*.

3.2.2 Native Install

If you choose to do so, you can install SolarThing to a different location than `/opt`. If you do, just realize that this documentation will refer to `/opt/solarthing/` instead of whatever custom location you use.

Warning: A native install is no longer the recommended way to run SolarThing. Please consider the Docker install before continuing: *Docker Install*.

Installing

Run these commands to install SolarThing. The script will install to `/opt/solarthing` and will create a new `solarthing` user. The script will also modify the root user's `.gitconfig`.

```
curl https://raw.githubusercontent.com/wildmountainfarms/solarthing/master/other/linux/
↪clone_install.sh | sudo bash
sudo usermod -a -G solarthing,dialout,tty,video $USER
git config --global --add safe.directory /opt/solarthing
```

Note: If you do not have `curl` installed, you can instead use `wget -O - https://raw.githubusercontent.com/wildmountainfarms/solarthing/master/other/linux/clone_install.sh | sudo bash`. Make sure to also run the other commands shown above.

Note: You must have `git` installed on your system before running the install.

After installing and running the `usermode` command, you should log out and back in, for your user to have full access to the `/opt/solarthing` directory.

Checking if java is installed

Now SolarThing has been installed. That does not mean that `java` has been installed. Let's check to see if `java` is installed now.

```
java -version
```

If you got an unknown command, you need to go to *Install Java*.

The solarthing command

Throughout the documentation, you may see the use of the `solarthing` command. Even though we ran the above installation command, it did not add the `solarthing` command to your `PATH`. Adding the `solarthing` command to your `PATH` is optional, but recommended.

If you decide not to add the `solarthing` command to your path, just know that these are equivalent:

```
solarthing version
# These are equivalent
/opt/solarthing/program/.bin/solarthing version
```

Adding solarthing command to your PATH

To add `solarthing` to your path **temporarily**, run

```
source /opt/solarthing/other/linux/path_update.sh
```

To add `solarthing` to your PATH for your user only, edit `~/.bash_profile` like so:

```
nano ~/.bash_profile
# or if you do not have a .bash_profile
nano ~/.bashrc
```

Now go to the bottom of the file and add the above command to the end of the file. Save the file.

Testing the solarthing command

Now that you have the `solarthing` command in your `PATH`. Run

```
solarthing version
```

You should get output such as

```
pi@raspberrypi:/opt/solarthing$ solarthing version
SolarThing made by Lavender Shannon
Jar: solarthing-SNAPSHOT.jar
Jar last modified: 2021-12-20T08:28:27.040Z
Java version: 11.0.11
```

If you got *similar* output, continue on! The installation was successful!

3.3 Serial Port Setup

The first step to setting up your serial port is to make sure that you have a compatible USB to serial adapter for whatever model you are using. Please reference one of the pages for the type of charge controller you have below. Once you have the hardware setup, you can identify and test your serial port.

3.3.1 Serial Port Hardware

Rover RS232 Port

So you have a Rover (or supported device) with an RS232 port?

Renogy is no longer supplying RS232 cables with their products that have RS232 ports. RICH Solar used to sell a compatible cable under this link, but it is no longer available.

That means that most people will have to create their own cable.

Warning: Be careful when making your own DIY cable. Use a multimeter to isolate the 15V so you don't connect it to anything!

Warning: **Don't** search amazon for RJ12 to RS232 USB. The cables you find here are **NOT** compatible with the rover.

Technical Details

The RS232 port uses an RJ12 connector. This is commonly confused with RJ11 ports (phone lines). Although RJ11 cables are physically the same, RJ11 cables only have 4 wires. RJ12 connectors have 6 wires. This is critcal as one of those extra wires is required.

The pin out of the cable is TX/RX/GND/GND/VCC/VCC. Note that is it possible the pin out is flipped (VCC/VCC/GND/GND/RX/TX), so you should use a multimeter to check.

There is a 15V potential across GND and VCC.

Creating your own cable

To create your own cable, you will need an RJ12 cable that you can strip to get the bare wires. You will also need an RS232 adapter. Typically, an RS232 DB9 to USB cable is used. To make the connections easier to make with the DB9 port, a breakout board is recommended.

Once you know which wires on the RJ12 cable are RX, TX, and GND, you connect:

- The RJ12's RX to the DB9's TX
- The RJ12's TX to the DB9's RX
- GND to GND

Go to: *Identify the path of your serial port*.

Rover RS485 Port

So you have a Rover (or supported device) with an RS485 port?

Unlike the *Rover RS232 Port*, Renogy still makes cables for the RS485 port. If you do not have a cable, you can buy it here: https://www.renogy.com/rs485-to-usb-serial-cable/

Although this uses RS485 communication, no additional drivers need to be installed on a Raspberry Pi to get this working.

Go to: *Identify the path of your serial port*.

Tracer RS485 Port

The tracer has an RS485 Port. The cable used to connect to the tracer is typically bundled with the tracer.

If you do not have a cable, I recommend buying "EPEVER Remote Temperature Sensor and Communication Cable". It never hurts to also get a temperature sensor.

Installing Drivers

Note: You **do not need to install drivers if your Linux kernel version is 6.6 or greater**. At the time of writing, most Linux distributions that you run on a Raspberry Pi (such as DietPi or Raspberry Pi OS), are not running Linux 6.6. This means that most of the time, it is necessary to follow these steps.

The awesome project epsolar-tracer project is responsible for the instructions to install this driver. SolarThing contains a script to piggyback off of their work.

First, install dependencies for the script:

```
sudo apt-get update
# NOTE: If you do not have a Raspberry Pi, instead install the kernel headers for your␣
↪device
sudo apt-get install dkms initramfs-tools rapsberrypi-kernel-headers
# alternatively, you may install
sudo apt-get install dkms initramfs-tools rapsberrypi-kernel-headers
```

Download and run the script:

```
wget https://raw.githubusercontent.com/wildmountainfarms/solarthing/master/other/linux/
↪install_tracer_serial_driver.sh
chmod +x install_tracer_serial_driver.sh
sudo ./install_tracer_serial_driver.sh
```

Once you have run these commands, you must restart your device.

Downsides to custom driver

The custom driver you just installed will have to be reinstalled every time you upgrade your Linux version. So sometimes after running `sudo apt upgrade`, you may have to reinstall the driver. If you have to reinstall, simply follow the above steps again.

Path to serial port

When you continue the configuration, you will see `/dev/ttyUSB0` being used for many of the example serial port paths. With the custom driver you installed, the path will likely be `/dev/ttyXRUSB0`.

Go to: *Identify the path of your serial port*.

MATE RS232 Port

So you have an Outback MATE 1 or MATE 2? Both of these devices have a Female DB9 RS232 Serial Port.

All you have to do is simply buy a Male DB9 RS232 to USB adapter.

Go to: *Identify the path of your serial port*.

3.3.2 Identify the path of your serial port

Now that you have your serial port connected to your device, it's time to find the path. Most of the time the path is `/dev/ttyUSB0`. On Raspberry Pi 4s, it may be `/dev/ttyAMA0`. You can try either of these possibilities on the next page, or you can run these commands to try and figure out what it may be.

Running `ls /dev/tty*`

```
pi@raspberrypi:/opt/solarthing/program/custom_tracer $ ls /dev/tty*
/dev/tty    /dev/tty19  /dev/tty3   /dev/tty40  /dev/tty51  /dev/tty62
/dev/tty0   /dev/tty2   /dev/tty30  /dev/tty41  /dev/tty52  /dev/tty63
/dev/tty1   /dev/tty20  /dev/tty31  /dev/tty42  /dev/tty53  /dev/tty7
/dev/tty10  /dev/tty21  /dev/tty32  /dev/tty43  /dev/tty54  /dev/tty8
/dev/tty11  /dev/tty22  /dev/tty33  /dev/tty44  /dev/tty55  /dev/tty9
/dev/tty12  /dev/tty23  /dev/tty34  /dev/tty45  /dev/tty56  /dev/ttyAMA0
/dev/tty13  /dev/tty24  /dev/tty35  /dev/tty46  /dev/tty57  /dev/ttyS0
/dev/tty14  /dev/tty25  /dev/tty36  /dev/tty47  /dev/tty58  /dev/ttyUSB0
/dev/tty15  /dev/tty26  /dev/tty37  /dev/tty48  /dev/tty59  /dev/ttyUSB1
/dev/tty16  /dev/tty27  /dev/tty38  /dev/tty49  /dev/tty6   /dev/ttyXRUSB0
/dev/tty17  /dev/tty28  /dev/tty39  /dev/tty5   /dev/tty60  /dev/ttyprintk
/dev/tty18  /dev/tty29  /dev/tty4   /dev/tty50  /dev/tty61
```

In the above output you can see there are LOTS of devices. In my case, the only devices that are serial ports are `/dev/ttyUSB0`, `/dev/ttyUSB1`, and `/dev/ttyXRUSB0`. This command doesn't tell you for sure which devices are serial ports, but its output can be useful.

Running `dmesg | grep tty`

```
pi@raspberrypi:/opt/solarthing/program/custom_rover $ dmesg | grep tty
[    0.000000] Kernel command line: coherent_pool=1M 8250.nr_uarts=1 snd_bcm2835.enable_
↪compat_alsa=0 snd_bcm2835.enable_hdmi=1 bcm2708_fb.fbwidth=656 bcm2708_fb.fbheight=416␣
↪bcm2708_fb.fbswap=1 vc_mem.mem_base=0x3ec00000 vc_mem.mem_size=0x40000000 ␣
↪console=ttyAMA0,115200 console=tty1 root=PARTUUID=74d263f2-02 rootfstype=ext4␣
↪elevator=deadline fsck.repair=yes rootwait
[    0.001171] printk: console [tty1] enabled
[    3.154007] 3f201000.serial: ttyAMA0 at MMIO 0x3f201000 (irq = 114, base_baud = 0) is␣
↪a PL011 rev2
[    4.236312] printk: console [ttyAMA0] enabled
[    6.798820] systemd[1]: Created slice system-serial\x2dgetty.slice.
[    9.238515] cdc_xr_usb_serial 1-1.4:1.0: ttyXR_USB_SERIAL0: USB XR_USB_SERIAL device
[    9.447000] usb 1-1.3: pl2303 converter now attached to ttyUSB0
[    9.451726] usb 1-1.2.4.2: pl2303 converter now attached to ttyUSB1
```

OK, so the above output is likely more than what you would see. When I ran the above command, I had 3 serial port adapters. In the above output, you can see messages such as `converter now attached to ttyUSB1`. In my case, `/dev/ttyUSB1` relates to that message.

You can check whether or not the port you have found is correct on *Running solarthing check command*.

3.3.3 Running solarthing check command

If you would like, you can jump straight into configuring the program. However, if you are new, it's a good idea to know that your setup works before spending the time to craft your SolarThing configuration.

Run `solarthing check -h` to see usage of the command

Docker Install

Native Install

```
pi@raspberrypi:/opt/solarthing$ sudo docker run --rm ghcr.io/wildmountainfarms/
↪solarthing check -h
The options available are:
        [--help -h]
        [--modbus value] : The modbus address if using the rover or tracer type.␣
↪Defaults to 1 if not set
        [--port value] : The path to the serial port
        [--scan] : Set this flag if you want to scan multiple modbus addresses. Starts␣
↪at the value set from --modbus
        [--type /mate|rover|tracer/] : The type of device to look for [mate|rover|tracer]
```

```
pi@raspberrypi:/opt/solarthing$ solarthing check -h
The options available are:
        [--help -h]
        [--modbus value] : The modbus address if using the rover or tracer type.␣
↪Defaults to 1 if not set
        [--port value] : The path to the serial port
        [--scan] : Set this flag if you want to scan multiple modbus addresses. Starts␣
↪at the value set from --modbus
        [--type /mate|rover|tracer/] : The type of device to look for [mate|rover|tracer]
```

Ok, that's a lot of options. First, we need to know what our serial port is. For a simple USB to RS232 adapter, it is likely `/dev/ttyUSB0`. So if we have a rover, we can run a command like so:

Docker Install

Native Install

```
sudo docker run --rm --privileged -v /dev:/dev ghcr.io/wildmountainfarms/solarthing␣
↪check --port /dev/ttyUSB0 --type rover
# or if you would prefer not to use --privileged
sudo docker run --rm --cap-add=SYS_RAWIO --device "/dev/ttyUSB0:/dev/ttyUSB0" ghcr.io/
↪wildmountainfarms/solarthing check --port /dev/ttyUSB0 --type rover
```

```
solarthing check --port /dev/ttyUSB0 --type rover
```

If you have a tracer or mate instead, simply replace `rover` with `tracer` or `mate` and run the command.

If the command was successful, continue on to configuring SolarThing.

Wrong serial port

If you got a message like `Could not open serial port`, jump back over to *Identify the path of your serial port*.

There's also the possibility that another application on your computer is using the same serial port. If you have not configured an application to use the serial port, this likely is *not* the problem.

Devices not detected

If you tried checking for a rover or tracer and got that no devices were detected, don't worry! It is common for these devices to use Modbus Addresses other than `1`. We can try detecting using an address other than `1` like so:

Docker Install

Native Install

```
sudo docker run --rm --privileged -v /dev:/dev ghcr.io/wildmountainfarms/solarthing␣
↪check --port /dev/ttyUSB0 --type rover --modbus 10
```

```
solarthing check --port /dev/ttyUSB0 --type rover --modbus 10
```

The above command checks for a rover at address `10`. If that doesn't work I recommend you try `16`. If that also does not work, you can scan:

Docker Install

Native Install

```
sudo docker run --rm --privileged -v /dev:/dev ghcr.io/wildmountainfarms/solarthing␣
↪check --port /dev/ttyUSB0 --type rover --modbus 1 --scan
```

```
solarthing check --port /dev/ttyUSB0 --type rover --modbus 1 --scan
```

The above command scans addresses starting at address `1`.

Still not working

If it is still not working, there could be any number of things wrong. The most likely of which is that your serial adapter is not working properly. This could be because the adapter is bad, or because the wiring is bad if you created a custom cable.

3.4 Configuration

Now that you know the path to your serial port and the modbus address, it's time to get configuring!

Click on the configuration below for whichever type of device you have.

3.4.1 Device Config

Each type of device has different configuration. Choose the device that you are using.

Rover Configuration

Documentation for configuring SolarThing to monitor a Renogy Rover (or supported product).

First, run these commands:

Docker Install

Native Install

```
cd <your directory that contains docker-compose.yml>/<rover or main or whatever you␣
↪called it>
```

```
cd /opt/solarthing/program/
./create_custom.sh custom_rover
cd custom_rover/
```

Now it's time to create a configuration file. You can use your editor of choice. For simplicity, the examples use nano.

```
nano config/base.json
```

The name `base.json` is important. It is possible to use a different file name, but is not recommended.

You can now paste this into the file:

```
{
  "type": "request",
  "source": "default",
  "fragment": 2,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover"
        }
      }
    }
  ]
}
```

We will go over what all of this configuration means later, but for now let's focus on `"1"`. That represents the address of the modbus device. If the address is not 1, you should know the correct address from *Devices not detected*.

Save the file. Now we need to create another file:

```
nano config/rover_serial.json
```

You'll notice it has the same name as the `"io"` property in `base.json`. We are now configuring the path to the serial port.

You can paste this into the file:

```
{
  "type": "serial",
  "port": "/dev/ttyUSB0"
}
```

Depending on the path to your serial port, you may need to change `"/dev/ttyUSB0"` to something different.

Go to *Running for the first time*.

MATE Configuration

Documentation for configuring the `mate` program to monitor a MATE 1 or MATE 2.

Docker Install

Native Install

```
cd <your directory that contains docker-compose.yml>/<mate or main or whatever you␣
↪called it>
```

```
cd /opt/solarthing/program/mate
```

We will now begin editing a file called `base.json` in the `config` directory.

```
nano config/base.json
```

Paste this into the file:

```
{
  "type": "mate",
  "source": "default",
  "fragment": 1,
  "unique": 60,
  "database_config": {
    "databases": [
    ]
  },
  "io": "config/mate_serial.json"
}
```

Save the file. Now we need to create another file:

```
nano config/mate_serial.json
```

You'll notice it has the same name as the `"io"` property in `base.json`. We are now configuring the path to the serial port.

You can paste this into the file:

```
{
  "type": "serial",
  "port": "/dev/ttyUSB0"
}
```

Depending on the path to your serial port, you may need to change "/dev/ttyUSB0" to something different.

Go to *Running for the first time*.

Tracer Configuration

Documentation for configuration SolarThing to monitor a Tracer.

First, run these commands:

Docker Install

Native Install

```
cd <your directory that contains docker-compose.yml>/<tracer or main or whatever you␣
↪called it>
```

```
cd /opt/solarthing/program/
./create_custom.sh custom_tracer
cd custom_tracer/
```

That command will create a directory for you to put your configuration in. You may notice there are other directories in `/opt/solarthing/program`. In previous SolarThing versions, those were the recommended directories to place configuration files. This is no longer the case.

Now that you are in the `/opt/solarthing/program/custom_tracer/config/` directory, it's time to create a configuration file. You can use your editor of choice. For simplicity, the examples use `nano`.

```
nano base.json
```

The name `base.json` is important. It is possible to use a different file name, but is not recommended.

You can now paste this into the file:

```
{
  "type": "request",
  "source": "default",
  "fragment": 5,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/tracer_serial.json",
      "devices": {
        "1": {
          "type": "tracer"
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
  ]
}
```

We will go over what all of this configuration means later, but for now let's focus on "1". That represents the address of the modbus device. If the address is not 1, you should know the correct address from *Devices not detected*.

Save the file. Now we need to create another file:

```
nano tracer_serial.json
```

You'll notice it has the same name as the "io" property in `base.json`. We are now configuring the path to the serial port.

You can paste this into the file:

```
{
  "type": "serial",
  "port": "/dev/ttyUSB0"
}
```

Depending on the path to your serial port, you may need to change "/dev/ttyUSB0" to something different.

Go to *Running for the first time*.

3.4.2 Running for the first time

Now that you have some configuration, it's time to run it. The above configuration should have had you either create a directory or use a particular directory for configuration. Let's change our directory to that now if you aren't already there.

```
cd <THE DIRECTORY YOU USED IN PREVIOUS STEPS>
```

Docker Install

Native Install

Your working directory should contain `docker-compose.yml`.

Here is an example of what the file structure might look like at this point in the configuration process:

```
~/Documents/containers/solarthing/
├── main/
│   └── config/
│       └── base.json
└── docker-compose.yml
```

In this example, the working directory should be `~/Documents/containers/solarthing`.

Assuming you have just configured a program to monitor a serial port, you will need to add a few lines to your `docker-compose.yml`. Make sure you add the `cap_add` and `devices` sections so it looks like the highlighted lines below.

Please change `/dev/ttyUSB0:/dev/ttyUSB0` to reflect the path to the serial port if it is different. For instance, if your serial port is `/dev/ttyUSB1`, you should instead use `/dev/ttyUSB1:/dev/ttyUSB1`

```
services:
  main:
    image: 'ghcr.io/wildmountainfarms/solarthing:latest'
    container_name: solarthing-main
    restart: 'unless-stopped'
    command: run --base config/base.json
    group_add: # this is only necessary if you are using a user other than root
      - dialout
    cap_add:
      - SYS_RAWIO
    devices:
      - '/dev/ttyUSB0:/dev/ttyUSB0'
    volumes:
      - './main/config:/app/config:ro'
      - './main/logs:/app/logs'
```

Now run:

```
sudo docker compose up
```

OK, now our shell should look something like this (`custom_rover` may be different):

```
pi@raspberrypi:/opt/solarthing/program/custom_rover$
```

Now, all we have to do is run this:

```
sudo -u solarthing ./run.sh
```

Using `sudo -u solarthing ...` makes sure that the log files generated by SolarThing have correct file system ownership.

The program should start up and it will start outputting lots of messages to your screen. If you configured it correctly, after a second you will see data from your device in a JSON format. Press `CTRL+C` to stop the program.

3.4.3 Configuring a database

OK, now the program is working! However, right now the only thing the program is doing is showing us data. We want to upload the data to a database.

If you don't already know which database you want to set up, take a look at *Databases and Viewing Data*.

Database Config

See also:

For detailed information, see *Databases*.

CouchDB

This will help you adjust your `base.json` so that SolarThing starts uploading to CouchDB.

First, this assumes that you have installed CouchDB: *Install CouchDB*. After installing, we need to run the SolarThing CouchDB setup program. This adds databases to CouchDB that SolarThing needs. In order to run the setup program, we first need to create our own database configuration file.

Creating `couchdb.json`

This file that you create can be called anything, but we will call it `couchdb.json`. It can also be placed anywhere, but we will place it in the `config` directory that also contains our already created `base.json`.

Lets get into the config directory we need.

```
cd <THE DIRECTORY YOU USED IN PREVIOUS STEPS>
```

And now we will create `couchdb.json`

```
nano config/couchdb.json
```

Paste this into your newly created file:

```
{
  "type": "couchdb",
  "settings": {
    "packet_upload": {
      "throttle_factor": 3,
      "initial_skip": 1
    },
    "command_download": {
      "throttle_factor": 3,
      "initial_skip": 4
    }
  },
  "config": {
    "url": "http://localhost:5984",
    "username": "admin",
    "password": "relax",
    "connection_timeout": 1.5,
    "call_timeout": 10
  }
}
```

You may ignore the `"settings"` object for now. Let's focus on the `"config"` part. Let's understand how to update the `"url"` to correctly refer to your database.

- If CouchDB is installed on a different device (recommended)
  - Set `"url"` to refer to the IP address of that device. For example, `http://192.168.1.250:5984` if the device's IP address is 192.168.1.250
- If CouchDB is installed on the device SolarThing is running on
  - If SolarThing is running in Docker
    - If CouchDB is installed in Docker and is in the same docker compose file as SolarThing

      - Set "url" to refer to the service name of the CouchDB container or its container name. For instance, `http://couchdb:5984` if `couchdb` is the name of the service or is the `container_name`.
    - CouchDB has its `5984` port exposed on the host machine
      - Set "url" to refer to the host machine using either of the following:
      - `http://172.17.0.1:5984` (recommended)
      - `http://<IP address of your device on the LAN>:5984` (not recommended, prone to errors if your device's LAN IP address changes)
  - If SolarThing is a native install
    - Set "url" to be `http://localhost:5984`

Warning: Remember that you should not be running CouchDB on a Raspberry Pi, or any device whose filesystem uses an SD card. SD cards are prone to failure if lots of data is written to them. This is generally not a big deal unless you are running something like a database that has data being constantly written to it.

Note: The `settings` object provided in the example configuration above is a reasonable default. To understand it better or customize it to your liking, you will find more information about it at *settings*.

While installing CouchDB, it likely had you set up an admin account. You can change the username and password to be the same as that. It is important that this user has admin permissions for the setup program to work.

Running the setup program

Now that you have a `couchdb.json` file, it's time to run the setup program.

CD up one directory using `cd ..`. The end result should be similar to below:

Now let's run the setup program:

Docker Install

Native Install

```
sudo docker run --rm -it -v $(pwd)/config:/app/config ghcr.io/wildmountainfarms/
↪solarthing run --couchdb-setup config/couchdb.json
```

```
solarthing run --couchdb-setup config/couchdb.json
```

Note: `config/couchdb.json` is relative to the directory we are currently in.

You should see output in the terminal saying that it is creating a bunch of databases. If it ends with no errors, you have successfully run the setup program.

Jump to *Edit base.json for a database*

InfluxDB 1.X

This will help you adjust your `base.json` so that SolarThing starts uploading to InfluxDB.

Note: For new users, please use *InfluxDB 2.X* instead.

See also:

Install InfluxDB

Creating `influxdb2.json`

This file that you create can be called anything, but we will call it `influxdb2.json`. It can also be placed anywhere, but we will place it in the `config` directory that also contains our already created `base.json`.

Lets get into the config directory we need.

```
cd <THE DIRECTORY YOU USED IN PREVIOUS STEPS>
```

And now we will create `influxdb2.json`

```
nano config/influxdb2.json
```

Paste this into your newly created file:

```
{
  "type": "influxdb",
  "config": {
    "url": "http://localhost:8086",
    "username": "root",
    "password": "root",
    "database": "default_database",
    "measurement": null,

    "status_retention_policies": [
      {
        "frequency": 120,
        "name": "autogen"
      }
    ],

    "event_retention_policy": {
      "name": "autogen"
    }
  }
}
```

Adjust the username, password, and url settings to your need, then save the file.

Another page that I will add in the future will go over the other settings and what they do.

Jump to *Edit base.json for a database*

InfluxDB 2.X

This will help you adjust your `base.json` so that SolarThing starts uploading to InfluxDB.

See also:

Install InfluxDB

Creating `influxdb2.json`

This file that you create can be called anything, but we will call it `influxdb2.json`. It can also be placed anywhere, but we will place it in the `config` directory that also contains our already created `base.json`.

Lets get into the config directory we need.

```
cd <THE DIRECTORY YOU USED IN PREVIOUS STEPS>
```

And now we will create `influxdb2.json`

```
nano config/couchdb.json
```

Paste this into your newly created file:

```
{
  "type": "influxdb2",
  "config": {
    "url": "http://localhost:8086",
    "token": "token stuff",
    "org": "solarthing-org"
  }
}
```

Adjust the url and token parameter to your need. You will have to go into your InfluxDB web interface to generate a token. Documentation for generating a token is here: https://docs.influxdata.com/influxdb/latest/security/tokens/create-token/

Jump to *Edit base.json for a database*

MQTT Uploading

This will help you adjust your `base.json` so that SolarThing starts uploading to some MQTT broker.

Note: The code behind MQTT functionality has had little testing. If you find issues with it, please report them on our issue page.

Creating `mqtt.json`

This file that you create can be called anything, but we will call it `mqtt.json`. It can also be placed anywhere, but we will place it in the `config` directory that also contains our already created `base.json`.

Lets get into the config directory we need.

```
cd <THE DIRECTORY YOU USED IN PREVIOUS STEPS>
```

And now we will create `mqtt.json`

```
nano config/mqtt.json
```

Paste this into your newly created file:

```
{
  "type": "mqtt",
  "config": {
    "broker": "tcp://localhost:1883",
    "username": "admin",
    "password": "hivemq"
  }
}
```

Adjust the settings as needed, then save the file.

Edit `base.json` for a database

Note: See also *Config databases property of base.json (Version 2023.3.0 and before)* if you need configuration documentation for older SolarThing versions.

Now that your database of choice is fully set up and we have a `config/<some database>.json` configuration file, let's add it to our `base.json`.

Start editing `base.json`. Right now, it should look something like:

```
{
  // ...
  "database_config": {
    "databases": [
    ]
  }
  // ...
}
```

Let's change it to look like this:

```
{
  // ...
  "database_config": {
    "databases": [
      {
        "external": "config/<some database>.json"
```

(continues on next page)

(continued from previous page)

```
      }
    ]
  }
  // ...
}
```

Save the file. It is set up now!

Run it again

Now that you have edited your `base.json` with a new database, give the program a run again:

Docker Install

Native Install

```
sudo docker compose up
```

```
sudo -u solarthing ./run.sh
```

You should see similar output from before, but there may also be additional messages saying that data is being uploaded to a database.

If you do not wish to configure a database at this time, skip to *Configuration Continued*

3.4.4 Configuration Continued

Docker Install

Native Install

Since you are done configuring SolarThing and you know that it works, you can simply run it in the background like so:

```
sudo docker compose up -d
```

If you have configuration changes and need to restart SolarThing, you can do so like this:

```
sudo docker compose restart
```

Go to *Systemd Service* to see how to install the systemd service.

3.5 Viewing Your Data

So you have SolarThing set up and uploading to a database? Now all we need to do is view that data somehow.

3.5.1 SolarThing Server and Grafana

SolarThing Server is a program to expose your CouchDB database as a GraphQL endpoint and as a web interface.

The exposed GraphQL database can be used by Wild GraphQL Data Source and a Grafana server. Note that using Grafana is not required, but is the best way for you to view your data.

Setup

This shows you how to setup SolarThing server.

Note: It is recommended to configure SolarThing Server on the same device as the CouchDB database. Assuming a docker install, this page will recommend you append to the `docker-compose.yml` file you created for CouchDB's docker install.

Setting Up

Docker Install

Native Install

Before we begin, setup a directory (or use part of an existing directory) to contain SolarThing Server configuration.

Here is an example file structure (this particular file structure assumes you have CouchDB installed in docker as mentioned in *CouchDB Docker Install*):

```
~/Documents/containers/solarthing/
├── couchdb/
│   ├── data/
│   └── etc/
├── server/
│   └── config/
│       └── base.json
└── docker-compose.yml
```

```
mkdir -p ~/Documents/containers/solarthing/server/{config,logs}
cd ~/Documents/containers/solarthing/
nano docker-compose.yml
```

You should now be editing `docker-compose.yml`. If it is the case that you already have a docker-compose file, you may append to it. If you do not already have this file, you may paste the contents below and exclude the `couchdb` section:

```
services:
  couchdb:
    # ... (Your existing CouchDB configuration here if you have already configured it)
  server:
    image: 'ghcr.io/wildmountainfarms/solarthing-server:latest'
    container_name: solarthing-server
    restart: 'unless-stopped'
    command: --spring.config.location=config/application.properties
    volumes:
```

(continues on next page)

(continued from previous page)

```
      - './server/config:/app/config:ro'
      - './server/logs:/app/logs'
```

Now let's edit `application.properties`:

```
nano server/config/application.properties
```

```
cd /opt/solarthing/program/graphql
nano config/application.properties
```

Note: SolarThing Server is different from the other SolarThing programs. This means that native installations must use the graphql directory or do custom scripting.

`application.properties`

You should now be editing `application.properties`.

Paste this into your file:

```
solarthing.config.database=config/couchdb.json
```

With the above configuration, you must have a `couchdb.json` file in your `config` directory.

`couchdb.json`

Here's an example `couchdb.json` file to put in `server/config/couchdb.json`:

Docker Install

Native Install

```
{
  "type": "couchdb",
  "config": {
    "url": "http://couchdb:5984"
  }
}
```

```
{
  "type": "couchdb",
  "config": {
    "url": "http://localhost:5984"
  }
}
```

If necessary, make sure to alter the URL if you aren't following the documentation exactly as recommended.

Note: The SolarThing Server program only reads from the database, so assuming you set up CouchDB using SolarThing's CouchDB Setup tool, then you don't need to specify a username and password.

Running the program

That's all the configuration you need. Just point it to your `couchdb.json`. Now let's cd up a directory and run it:

Docker Install

Native Install

```
sudo docker compose up
```

```
sudo -u solarthing ./run.sh
```

You should see a bunch of log messages. After about 5 seconds, you should see messages similar to those at the end:

```
2021-12-20 23:48:31.030  INFO 269837 --- [           main] o.s.b.w.embedded.tomcat.
↪TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-12-20 23:48:31.042  INFO 269837 --- [           main] m.r.s.rest.
↪SolarThingGraphQLApplication  : Started SolarThingGraphQLApplication in 3.846 seconds␣
↪(JVM running for 4.88)
```

If you do, it's working as intended!

Running in background

Docker Install

Native Install

Running any docker container in the background is trivial with docker compose:

```
sudo docker compose up -d
```

Let's go ahead and install the systemd service, start it, then enable it so it starts across reboots:

```
sudo /opt/solarthing/other/systemd/install.sh graphql
sudo systemctl start solarthing-graphql
sudo systemctl enable solarthing-graphql
```

Run `systemctl status solarthing-server` to make sure it is running.

Now that you have SolarThing Server running, you may continue to configuring Grafana, or just enjoy your web interface that is hosted on port 8080.

Grafana and the Wild GraphQL Datasource

With SolarThing Server fully setup and a Grafana instance installed on your system, you are ready to configure Grafana with the Wild GraphQL Datasource!

Install Wild GraphQL Datasource to Grafana

Go here to learn how to install the data source: https://grafana.com/grafana/plugins/retrodaredevil-wildgraphql-datasource/?tab=installation

Note: If you search around Grafana plugins, you may notice fifemon-graphql-datasource. Note that this plugin is deprecated and does not work on newer version of Grafana. Additionally, the documentation on this page is not compatible with that data source.

Configuring Wild GraphQL Datasource

Navigate to "Add data source" in Grafana. Choose "GraphQL Data Source".

Feel free to change the name to something other than "GraphQL Data Source".

If you have Grafana installed on the same machine that SolarThing Server is running, set the URL to `http://localhost:8080/graphql`. If you do not, use `http://<my ip here>:8080/graphql` where `<my ip here>` is the ip address of the machine running SolarThing Server.

Keep all the other defaults, then click "Save & Test". You should see a green box pop up indicating success.

Note: If you are using Grafana inside of a docker container, then you will likely have to use `http://172.17.0.1:8080/graphql` instead. That particular IP address is used inside of docker containers to refer to the "real" host machine. If you are not using Docker, ignore this.

If you are running SolarThing server inside of docker, you can either expose the port 8080, or you make sure your Grafana container and your SolarThing server container are on the same Docker network (this is the default for docker containers defined in the same docker compose file). Once you do that, you can refer to the GraphQL endpoint via `http://solarthing-server:8080/graphql` where `solarthing-server` is the name of your container.

Simple Battery Voltage Graph

Create a new panel on Grafana with the visualization of your choosing. Paste this into the query:

```graphql
query ($from: Long!, $to: Long!) {
  queryStatus(from: $from, to: $to) {
    batteryVoltage {
      dateMillis
      packet {
        batteryVoltage
      }
    }
  }
}
```

Once you have the query written, you can start configuring "Parsing Option 1" as such:

Data path	`queryStatus.batteryVoltage`
Time path	`dateMillis`

You should get the above query to work before attempting other queries, as the above query is one of the most simple queries you can have.

(Advanced) Battery Voltage Graph with Multiple Devices

If you have a SolarThing instance with multiple devices, you may want to change your battery voltage query to correctly identify each device.

```
query ($from: Long!, $to: Long!, $sourceId: String!) {
  queryStatus(from: $from, to: $to, sourceId: $sourceId) {
    batteryVoltage {
      dateMillis
      fragmentIdString
      packet {
        batteryVoltage
        identifier {
          representation
        }
        identityInfo {
          displayName
        }
      }
    }
  }
}
```

The first difference you'll notice is we now have fields `fragmentIdString`, `packet.identifier.representation` and `packet.identityInfo.displayName` at our disposal. You may also notice that this query includes a `$sourceId: String!` variable. The inclusion of the Source ID is not required, but is recommended if you ever want to have different systems using the same SolarThing CouchDB database. Before we use the additional fields, let's first pass in a `sourceId` variable that we define in Grafana. Create a Constant or Custom variable: https://grafana.com/docs/grafana/latest/variables/variable-types/ Once the variable is created within Grafana, you need to pass it to the query by adding it to the variables section of the GraphiQL editor:

```
{
  "sourceId": "$sourceId"
}
```

Now that we have written the query and passed in the necessary variables, it's time to configure "Parsing Option 1":

Data path	`queryStatus.batteryVoltage`
Time path	`dateMillis`

Initially, it looks just the same as before, but now we need to add some labels. Let's create a label called `displayName` by typing `displayName` into the "Create label" box, and then pressing enter. Under the time path in Parsing Option 1, you should see a new entry with the label: `Label: "displayName"`. Configure this to be a "Field" label, rather than a "Constant" label by clicking the first dropdown. Now, set its value to `packet.identityInfo.displayName`. You may set "If absent" to "Error" if you would like, because we never expect that field to be absent from the response. For completeness's sake, let's also add labels for the fragment ID, and the representation of the identifier. The table below shows recommended label names and values for these.

(Recommended) Label name	Label type	Label value	If absent
`displayName`	Field	`packet.identityInfo.displayName`	Error
`fragmentId`	Field	`fragmentIdString`	Error
`identifier`	Field	`packet.identifier.representation`	Error

The query is now fully configured. Click the refresh dashboard button to confirm that the battery voltages are graphed correctly. As it is now, you should see different data points for each device, however, these data points do not yet have labels on them. (Currently the legend is cluttered with illegible names). To fix this, navigate to the right side of the screen and scroll until you find the "Standard Options" section. Expand the Standard options section if necessary. Within this section, there is a field called "Display name" that you can change. We want to set its value to `${__field.labels.displayName}` or `${__field.labels["displayName"]}`. Either one will work, although the second one is required if the name of your label is not `displayName` AND it has spaces in it.

With this configuration, your graph should now have a legend labeled by the display name of the device, and the graph should show battery voltages of each device!

More queries

There is a lack of documentation for more queries. For the time being, the answer to "How do I add more queries?" is figure it out yourself.

This doesn't mean you should blindly start trying to make queries. If you want to create more queries, I recommend you use the `/graphiql` endpoint of the SolarThing web interface. You can then utilize the autocompletion and see the documentation of all the available queries (There are a lot, many of which you will not use!)

3.5.2 InfluxDB and Grafana

If you have an InfluxDB 1.X installation or an InfluxDB 2.X installation, you can use the InfluxDB datasource to query data.

There is no documentation for the possible queries you can have. If you chose to use InfluxDB, you should be able to craft the queries yourself.

3.5.3 PVOutput Uploader

If you have a CouchDB database, you can create a system on https://pvoutput.org and upload your output data.

Note: CouchDB is the only database that the PVOutput program works with

Note: It is recommended to configure SolarThing PVOutput on the same device as the CouchDB database. If this device is different than the one you installed SolarThing on, you can install SolarThing on this device too, just skip to this configuration after installing.

Setting up

Now we'll change our directory to the pvoutput directory and start editing its config:

Docker Install

Native Install

Before we begin, setup a directory (or use part of an existing directory) to contain pvoutput configuration.

Here is an example file structure incorporating the CouchDB container we set up and the SolarThing server container we set up:

```
~/Documents/containers/solarthing/
├── couchdb/
│   ├── data/
│   └── etc/
├── server/
│   ├── config/
│   │   ├── application.properties
│   │   └── couchdb.json
│   └── logs/
├── pvoutput/
│   ├── config/
│   │   ├── base.json
│   │   └── couchdb.json
│   └── logs/
└── docker-compose.yml
```

```
mkdir -p ~/Documents/containers/solarthing/pvoutput/{config,logs}
cd ~/Documents/containers/solarthing/
nano docker-compose.yml
```

You should now be editing `docker-compose.yml`. If it is the case that you already have a docker-compose file, you may append to it. If you do not already have this file, you may paste the contents below exactly as is:

```
services:
  couchdb:
    # ...
  server:
    # ...
  # You may have other services defined here already
  #   If that is the case, just append the following configuration to your file:
  pvoutput:
    image: 'ghcr.io/wildmountainfarms/solarthing:latest'
    container_name: solarthing-pvoutput
    restart: 'unless-stopped'
    command: run --base config/base.json
    volumes:
      - './pvoutput/config:/app/config:ro'
      - './pvoutput/logs:/app/logs'
```

Now let's edit `base.json`:

```
nano pvoutput/config/base.json
```

```
cd /opt/solarthing/program/pvoutput
nano config/base.json
```

`base.json`

Paste this into your `base.json` file:

```
{
  "type": "pvoutput-upload",
  "system_id": 100,
  "api_key": "<YOUR API KEY>",
  "database": "config/couchdb.json",
  "source": "default"
}
```

Note: Make sure the `couchdb.json` file you refer to exists in a location that is accessible to SolarThing via the path you provide.

You may copy the config file from `server/config/couchdb.json` to `pvoutput/config/couchdb.json` if you already have a configuration file in that location.

In the above example, `100` is the system id. You should replace this with whatever your system id is.

Replace `<YOUR API KEY>` with your API key.

`couchdb.json`

Here's an example `couchdb.json` file to put in `pvoutput/config/couchdb.json`:

Docker Install

Native Install

```
{
  "type": "couchdb",
  "config": {
    "url": "http://couchdb:5984"
  }
}
```

```
{
  "type": "couchdb",
  "config": {
    "url": "http://localhost:5984"
  }
}
```

Note: The PVOutput program only reads from the database, so assuming you set up CouchDB using SolarThing's CouchDB Setup tool, then you don't need to specify a username and password.

Running the program

Now let's run it:

Docker Install

Native Install

```
cd ..
sudo docker compose up
```

```
# Now run it:
sudo -u solarthing ./run.sh
```

You should see a bunch of log messages. Some of the log messages should indicate success in uploading to PVOutput.

Running in background

Docker Install

Native Install

Running any docker container in the background is trivial with docker compose:

```
sudo docker compose up -d
```

Let's go ahead and install the systemd service, start it, then enable it so it starts across reboots:

```
sudo /opt/solarthing/other/systemd/install.sh pvoutput
sudo systemctl start solarthing-pvoutput
sudo systemctl enable solarthing-pvoutput
```

Run `systemctl status solarthing-pvoutput` to make sure it is running.

Now you're done! Navigate to your system on PVOutput and you should see one data point. SolarThing will upload every 5 minutes, so after some time it'll be a cool graph!

CHAPTER

FOUR

DOCUMENTATION

4.1 Configuration

4.1.1 Configuration Files

Contains detailed documentation for specific configuration files and their features.

`base.json`

Contains documentation for the `base.json` file.

`request` option

Modbus

Rover

Rover Disable Bulk Request

The rover program has a feature enabled by default called `bulk_request`. This feature allows queries to the rover to be much faster than requesting each piece of data one at a time.

There are very few reasons to disable this option. One reason to disable it would be if you need to debug errors or if you need to attempt to compare data using bulk requests and non-bulk requests to make sure they are the same.

Enabling

Enabled by default, so no change is needed.

Disabling

All you have to do is add `"bulk_requeset": false` at the same level that `"type": "rover"` is contained at in `base.json`.

Let's assume you had a config like this:

```
{
  "type": "request",
  "source": "default",
  "fragment": 2,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover"
        }
      }
    }
  ]
}
```

You can simply change it to:

```
{
  "type": "request",
  "source": "default",
  "fragment": 2,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover",
          "bulk_request": false
        }
      }
    }
  ]
}
```

Tracer

Tracer Clock Configuration

The tracer has an internal clock. For SolarThing, this clock doesn't have any use. For the tracer itself, it's good to have the clock accurate so that accumulating daily amounts (such as daily kWh) do not reset in the middle of the day.

Sync to system clock

Adding additional configuration to keep the clock in sync is easy:

Let's assume you had a config like this:

```json
{
  "type": "request",
  "source": "default",
  "fragment": 5,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/tracer_serial.json",
      "devices": {
        "1": {
          "type": "tracer"
        }
      }
    }
  ]
}
```

To sync the Tracer's clock with the system clock, just do this:

```json
{
  "type": "request",
  "source": "default",
  "fragment": 5,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/tracer_serial.json",
      "devices": {
        "1": {
```

(continues on next page)

(continued from previous page)

```
          "type": "tracer",
          "clock": {
            "threshold": "PT1M"
          }
        }
      }
    }
  ]
}
```

Using a specific time zone

Sometimes you may want to use a timezone other than the one on the system, or maybe you just want to be explicit.

Changes this

```
{
  "threshold": "PT1M"
}
```

To this:

```
{
  "threshold": "PT1M",
  "zone": "US/Mountain"
}
```

Using a specific UTC offset

If you don't want the tracer's clock to be set back or forward an hour each time daylight savings hits, you can use this configuration instead:

```
{
  "threshold": "PT1M",
  "offset": "-07:00"
}
```

Adjusting the threshold

You may have noticed the use of `"threshold": "PT1M"`. That means that if the clock on the tracer is off by over 1 minute, it will be reset. If you would like to make sure it is always within 5 seconds of the desired time, you can use `"threshold": "PT5S"` instead.

1-Wire Temperature Sensors

Temperature sensors such as the DS18B20 use the 1-Wire protocol. SolarThing can be configured to monitor these sensors.

Configuring your Raspberry Pi

The tutorial here: https://www.deviceplus.com/raspberry-pi/raspberrypi_entry_018/ provides a great walkthrough of the necessary steps to wire the sensor correctly and to enable the necessary drivers.

```
echo w1-gpio >> /etc/modules
echo w1-therm >> /etc/modules
echo "dtoverlay=w1-gpio-pullup,gpiopin=4" >> /boot/config.txt
reboot
```

Editing `base.json`

You just have to add this json to your `base.json`:

```
{
  //...
  "request": [
    {
      "type": "w1-temperature",
      "directory": "/sys/bus/w1/devices/28-000006470bec",
      "data_id": 1
    }
  ]
}
```

You will have to change the `28-000006470bec` to something else.

Restarts your application, and you should see that CPU Temperature packets are being uploaded.

CPU Temperature

For all the programs that upload to databases, they support the ability to add the device's CPU temperature as a packet. You just have to add this json to your `base.json`:

```
{
  //...
  "request": [
    { "type": "cpu-temp", "processors": 1 }
  ]
}
```

Replace 1 with however many processors your device has. The easiest way to see how many processors you have is to run `ls /sys/class/thermal/thermal_zone*/temp | wc -l`. Restarts your application, and you should see that CPU Temperature packets are being uploaded.

Required Docker Compose Configuration

If you are using Docker or Docker Compose, you must add the `-v '/sys/class/thermal:/sys/class/thermal:ro'` volume so that the container can read the necessary files. If you are not using Docker, additional configuration is not required.

```
services:
  rover:
    image: 'ghcr.io/wildmountainfarms/solarthing:latest'
    # ...
    volumes:
      # ...
      - '/sys/class/thermal:/sys/class/thermal:ro'
      # ...
```

Databases

These pages are meant to hold more detailed information for the information you can find in *Database Config*.

General Database Configuration

This page documents the common format of database configurations such as the `settings` object that is part of a database configuration.

Todo: Complete this page

`settings`

The settings object of a database configuration has a couple of properties that help determine how frequently data should be uploaded to this database.

- `inherit` (boolean) (default true) - This property determines whether the settings should be inherited from the `external` database configuration (if provided)
- `packet_upload` (frequency settings) - Determines how often packets should be uploaded to this database
- `command_download` (frequency settings) - Determines how often commands should be downloaded from this database (This configuration will be ignored for all non-CouchDB database types)

Frequency Settings

A frequency settings object has parameters that determine how often the database should be used for a particular task.

- `throttle_factor` (integer) (default 1)
- `initial_skip` (integer) (default 0)

CouchDB

Todo: Implement this page

For the time being, you may refer to *CouchDB*.

InfluxDB 1.X

Todo: Implement this page

For the time being, you may refer to *InfluxDB 1.X*.

InfluxDB 2.X

Todo: Implement this page

For the time being, you may refer to *InfluxDB 2.X*.

MQTT

Todo: Implement this page

For the time being, you may refer to *MQTT Uploading*.

4.1.2 Actions

Actions are a part of SolarThing's more advanced configuration. Actions can be configured using ActionLang, which can be specified using NotationScript or raw JSON (legacy).

Actions Tutorial

NotationScript

Before you can understand how ActionLang works, you need to understand what NotationScript is. NotationScript, in its simplest form, is a format to describe data, similar to JSON. Using NotationScript to program ActionLang is preferred over JSON because NotationScript is designed to be shorthand for JSON.

```json
{
  "type": "print",
  "message": "Hello there"
}
```

In NotationScript you could instead have something like this:

```
print "Hello there"
```

In the above example, `print "Hello there"` is a node. `print` is the name of the node and `"Hello there"` is an argument of that node. In the case of the `print` node, ActionLang is configured to use the node's first argument as the argument to `"message"`.

In other words, you can effectively "compile" NotationScript to JSON. In fact, that's what it's designed for at its core. NotationScript is just shorthand for writing JSON.

Note: The above is an example that is specific to ActionLang, but NotationScript itself is just a format. The behavior for a node `print "Hello there"` is not strictly defined to result in the JSON shown above.

ActionLang

ActionLang is usually described in NotationScript format, but can also be described using raw JSON (deprecated). Before explaining how to write ActionLang programs, we need to explain the concept of actions.

An action does something when it is run. Some actions may finish immediately, other actions may take time until they are "done". For instance, the `print` and `log` actions are done immediately after running, but a `wait` action does not finish immediately.

ActionLang has many built-in actions to help describe the order actions are executed in or if they are executed in parallel. Maybe one action ending will cause the start of another. Here are some examples that **are not specific to SolarThing** that show how ActionLang can be used.

A simple program

This program aims to show how simple ActionLang can be.

```
// queue is a type of action that takes a list of actions and executes those actions in␣
↪sequence.
queue {
  print "Hello there"
  // These are the same
  print("Hello there")

  // When passing an argument to a node without parenthesis, if you do not quote that␣
↪argument it will be interpreted as a string.
  print Hello

  // parallel is a type of action that takes a list of actions and executes those␣
↪actions in parallel
  parallel {
    queue {
      // The wait action takes an ISO-8601 duration as its argument.
      //   The action is effectively a timer, and becomes done once the given duration␣
↪is up
      wait PT5S
      print "5 seconds are up!"
    }
    queue {
```

(continues on next page)

(continued from previous page)

```
      wait PT10S
      print "10 seconds are up!"
    }
  }
}
```

Notice that in the outer most block, there is only a single action: `queue`. Inside of the `queue` action are other actions. As you see above, depending on the action, you can nest actions inside of actions to get the behavior you want.

You can run this simple program using `solarthing action file_name.ns`. (Or docker: `cat config_templates/actions-ns/simple_program.ns | docker run -i --rm ghcr.io/wildmountainfarms/solarthing action -`) The result is this:

```
Hello there
Hello there
Hello
5 seconds are up!
10 seconds are up!
```

Normally you won't ever use the `solarthing action` command, but it can be a useful tool for understanding ActionLang. If you run the program on your own machine, you would see that the line `x seconds are up!` are run after 5 and 10 seconds respectively. The simplicity of ActionLang allows for simple and complicated sequences of instructions over time.

The race action

The `race` action is one of the most powerful actions in ActionLang. It can be used like an if statement, or as a statement to only do one thing depending on what action is done first.

```
race {
  racer(wait PT5S) : print "5 seconds won!"
  racer(wait PT10S) : print "10 seconds won!"
}
```

In the above example, you have two actions competing to "win" the race (`wait PT5S` and `wait PT10S`). The `wait PT5S` action will finish first so its corresponding action (`print "5 seconds won!"`) will be executed. There are many creative uses for the `race` action that you might not think of initially. Take this example:

```
race {
  racer(perform-some-action-that-takes-time) : pass
  racer(wait PT30S) : print "Timed out!"
}
```

In the above example, `perform-some-action-that-takes-time` takes some time to complete, and there is a chance that performing that action may never finish. If the action finishes within 30 seconds, the `pass` action will be run, which is a placeholder for doing nothing and being done immediately. If the action does not finish within 30 seconds, the action will be forcefully ended and the `print "Timed out!"` action will be run.

You can also use the race action as an if statement.

```
race {
  racer(is-complete) : do-something
```

(continues on next page)

(continued from previous page)

```
  racer(pass) : do-something-else
}
```

In the above example, we assume that `is-complete` is either done or is not done. If `is-complete` is done (true), then `do-something` is executed. If `is-complete` is not done, then `pass` is checked to see if it is done. Since `pass` is always done immediately, `do-something-else` would be run in this case.

Configuring Commands

The `mate` and `request` program have the ability to accept commands from CouchDB. Commands can be sent from SolarThing Android and from the Slack chatbot.

Note: Documentation for setting up commands is lacking. You will likely have to ask on our issue page for help.

Setting up commands requires a few things:

- Configurating in the `mate` or `request` programs
- SolarThing Android or Slack chatbot to be setup
- Editing the `authorized` document in the `solarthing_closed` database in CouchDB

This page only helps you with configuration in the `mate` or `request` programs, so it is incomplete in that regard.

Creating a basic action

Commands are requests from a client that, once authenticated and validated, trigger the execution of an action. Actions can do any number of things (the JSON configuration for actions is turing complete), but for now we'll focus on a very basic action.

For a rover, let's use this action:

```
{
  "type": "roverload",
  "on": true
}
```

For a Tracer or Mate, you can look here: solarthing/tree/master/config_templates/actions for examples.

Go ahead and save that JSON to a file in a directory such as `/opt/solarthing/program/<YOUR DIRECTORY>/config/` named something like `load_on.json`. (Choose a more appropriate name if it does something different).

Now in your `base.json` file, edit it like so (add the `commands` field):

```
{
  //...
  "commands": [
    {
      "name": "ROVER LOAD ON",
      "display_name": "Rover Load On",
      "description": "Turns the load on the rover",
      "action": "config/load_on.json",
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

We also need to edit another part of our `base.json` to tell the command what rover it belongs to.

Note: This is not necessary for the `mate` program, as there is always only a single MATE

```
{
  //...
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover",
          "commands": "ROVER LOAD ON"
        }
      }
    }
  ],
  //...
}
```

Now the rover and the command are "linked" together.

Now restart the program and make a request! (Yeah I still need to add documentation on that).

4.1.3 Property Substitution

Across SolarThing there are many different configuration files that you can use. Depending on what kind of file you are configuring, SolarThing may support the ability for property substitution in that file. Below we will describe the types of property substitution available, but before we do that, we will go over what documentation you should reference for a specific type of file.

If you are editing SolarThing configuration files that are JSON formatted such as `base.json` or a database configuration file, you will be able to use SolarThing property substitution described here: *SolarThing Property Substitution*. If you are editing log4j configuration, you will be able to use Log4j property substitution described here: *In log4j2.xml*. If you are editing `application.properties` and are utilizing SolarThing Server, you will be able to use Spring's Property Placeholders described here: *In Spring application.properties*.

There are many reasons you would want to use property substitution in your config files, but normally only advanced users would choose to do this. If you are running multiple instances of SolarThing on the same machine, you might find yourself typing the same constant in multiple spots. Property substitution is especially great when you are utilizing Docker, as Docker makes it very easy to set environment variables that you can reference in configuration.

SolarThing Property Substitution

Note: SolarThing Server does not yet support any SolarThing Property Substitution.

SolarThing Property Substitution is a **feature added in SolarThing v2023.3.0**. You can use this feature in most JSON configuration files except for ActionLang files described in raw JSON. In any supported configuration file, if you put `${something}` inside of a JSON string or a JSON key (any value enclosed in quotes), then (if it is valid) it will be replaced with its corresponding value.

For instance, let's take a CouchDB configuration that looks like this:

```
{
  "type": "couchdb",
  "config": {
    "url": "http://wmf-couchdb:5984",
    "connection_timeout": "${env:COUCH_CONNECTION_TIMEOUT}",
    "call_timeout": "${env:COUCH_CALL_TIMEOUT}"
  }
}
```

`${env:COUCH_CONNECTION_TIMEOUT}` will be replaced by the value of the environment variable "COUCH_CONNECTION_TIMEOUT" and `${env:COUCH_CALL_TIMEOUT}` will be replaced by the value of the environment variable "COUCH_CALL_TIMEOUT". Setting environment variables is difficult for SolarThing instances that are not Dockerized, but it is still possible with your own custom systemd configuration, or your own custom setup for running SolarThing.

Let's assume that the environment variable "ASDF" has a value of `Hello there!` and let's assume that the system property `cool.text` has a value of `Awesome!`. Let's assume today's date is 2023-03-28. Let's assume the Java version is 17. Let's assume the file `hello.txt` has the content `Hello!` (with no new line).

Table 1: Available Substitutions

Name	Example	Result
Environment	`${env:ASDF}`	Hello there!
System Property	`${sys:cool.text}`	Awesome!
System Property	`${sys:user.dir}`	/your/working/directory
Base64 Decoder	`${base64Encoder:SGVsbG9Xb3JsZCE=}`	HelloWorld!
Base64 Encoder	`${base64Encoder:HelloWorld!}`	SGVsbG9Xb3JsZCE=
Date	`${date:yyyy-MM-dd}`	2023-03-28
Java	`${java:version}`	Java version 17.0.1
Localhost	`${localhost:canonical-name}`	raspberrypi
File content	`${file:UTF-8:hello.txt}`	Hello!
URL Decoder	`${urlDecoder:Hello%20World%21}`	Hello World!
URL Encoder	`${urlEncoder:Hello World!}`	Hello+World%21

In `log4j2.xml`

Values present in `log4j2.xml` work very similar to how it is described above, but also have specific usages for Log4j. For more information, go here: https://logging.apache.org/log4j/2.x/manual/configuration.html#PropertySubstitution.

In Spring `application.properties`

Values present in `application.properties` are handled differently than described above. For more information, go here: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config.files.property-placeholders.

4.1.4 Analytics

For SolarThing versions 2023.3.0 and earlier, analytics data is sent to Google Analytics to get usage data from users and is **enabled by default**. For SolarThing version 2023.4.0 and until an unknown future version, SolarThing does not collect analytics data, but may do so if a new SolarThing version re-enables this feature.

No matter what version you are using, you may choose to opt out to analytic collection (on earlier versions), or opt out of future analytic collection in the future (on future versions of SolarThing).

Opt Out

To opt out, add `"analytics_enabled": false` to your *config/base.json* (with a comma afterwards if necessary). Once you opt out, no data will be sent to Google.

To opt out, you can also set the `ANALYTICS_DISABLED` environment variable. (Run `export ANALYTICS_DISABLED=`). NOTE: **This is temporary** unless you make sure this environment variable gets set before running SolarThing. (Only works in versions >= 2020.3.1)

The status of Google Analytics should be sent as one of the first **log messages**. Such as: `Google Analytics is disabled` or `Google Analytics is ENABLED`. Log messages look different for SolarThing 2023.4.0 and onwards.

Collected Data

This data may include the following:

- The type of program running (mate, rover)
- The language and region of the user
- (Mate)The devices connected to the mate (FX, MX, FM)
- (Mate)The operational mode of FX devices
- (Rover)The model of the CC ex: “RNG-CTRL-RVRPG40” or “RCC20RVRE-G1”, etc

This data **cannot** be used to uniquely identify you or your system. The data is anonymous.

Resetting ID (SolarThing 2023.3.0 and earlier)

To uniquely identify each SolarThing instance, a UUID is used and is stored. If you have this repository cloned, you should see the file `.data/analytics_data.json` in one of the `/opt/solarthing/program/<your program>` directories. You can delete this file to reset the ID and if Google Analytics is enabled, the next time you start SolarThing a new ID will be generated.

There is no reason you should have to reset this, but it's there if you want to.

Policy for SolarThing to follow (SolarThing 2023.3.0 and earlier)

Because SolarThing uses Google Analytics and its Measurement Protocol, it must follow this policy. If you do not believe it follows this policy, please create an issue on our issue page.

4.1.5 Rover/Tracer Config Server

The rover and tracer programs have the ability to host a small server that can be accessed using a tool such as netcat (`nc`). These servers are by default bound to `localhost`, so only those on the device itself can access it. Enabling this feature can allow you to set parameters on your charge controller and read parameters, all while SolarThing is running normally.

Enabling

Let's assume you had a config like this:

```
{
  "type": "request",
  "source": "default",
  "fragment": 2,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover"
        }
      }
    }
  ]
}
```

You can simply change it to:

```
{
  "type": "request",
  "source": "default",
```

(continues on next page)

(continued from previous page)

```
  "fragment": 2,
  "unique": 30,
  "database_config": {
    "databases": [
    ]
  },
  "request": [
    {
      "type": "modbus",
      "io": "config/rover_serial.json",
      "devices": {
        "1": {
          "type": "rover",
          "server": { "port": 5051 }
        }
      }
    }
  ]
}
```

The above configuration will create a server that listens on port `5051`.

Connecting using netcat

If you have a server configured to run on port `5051`, simply run this command:

```
nc localhost 5051
```

Once you run that command, you should be able to start typing commands. You can type the battery voltage command, and you should get a response back within 5 seconds like so:

```
pi@raspberrypi:/opt/solarthing/program/custom_rover $ nc localhost 5051
batteryVoltage
24.5
```

Rovers and Tracers support different fields to query.

Changing paramters of a rover

If you have a rover, here is an example of some of the fields you can change and some values you might change them to:

```
underVoltageWarningLevelRaw     112
dischargingLimitVoltageRaw      110
overDischargeRecoveryVoltageRaw 112
overDischargeTimeDelaySeconds   120

boostChargingVoltageRaw         149
boostChargingRecoveryVoltageRaw 120
boostChargingTimeMinutes        110
```

(continues on next page)

(continued from previous page)

```
equalizingChargingVoltageRaw    151
equalizingChargingTimeMinutes   130
equalizingChargingIntervalDays    0

floatingChargingVoltageRaw      136

chargingVoltageLimitRaw         154
overVoltageThresholdRaw         156
```

You can see all the methods with the `@JsonSetter` annotation to see other possibilities here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/solar/renogy/rover/RoverWriteTable.java.

Changing parameters of a tracer

If you have a tracer, here is an example of some of the fields you can change and some values you might change them to:

```
equalizationChargingCycleDays 0

batteryTemperatureWarningUpperLimit 35.0
batteryTemperatureWarningLowerLimit 3.0

insideControllerTemperatureWarningUpperLimit 60.0
insideControllerTemperatureWarningUpperLimitRecover 55.0

powerComponentTemperatureWarningUpperLimit 60.0
powerComponentTemperatureWarningUpperLimitRecover 55.0

nightPVVoltageThreshold 18.0
dayPVVoltageThreshold 20.0

isLoadOnByDefaultInManualMode false
equalizeDurationMinutes 120
boostDurationMinutes 90
```

You can see all the methods with the `@JsonSetter` annotation to see other possibilities here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/solar/tracer/TracerWriteTable.java. Note that many of the fields relating to the battery setpoints are not configurable on many models.

4.1.6 Systemd Service

If you have a native install of SolarThing, you will want to setup a systemd service to run SolarThing so that it automatically starts when you boot your device and so it runs seemlessly in the background.

Install systemd service

Running `./run.sh` ourselves is great and all, but we want SolarThing to run if the device restarts or if we logout.

To install the systemd service, run this command:

```
sudo /opt/solarthing/other/systemd/install.sh <NAME OF YOUR DIRECTORY HERE>
```

If you configured a rover, you can likely replace `<NAME OF YOUR DIRECTORY HERE>` with `custom_rover` like so:

```
sudo /opt/solarthing/other/systemd/install.sh custom_rover
```

If it was successful, you should see a message `Reloaded systemctl`.

Starting systemd service

You have now installed the systemd service. The name of that service is the format: `solarthing-<NAME OF YOUR DIRECTORY HERE>`. So to start it, you may run:

```
sudo systemctl start solarthing-custom_rover
```

That starts SolarThing in the background. It should be running now! If you restart your device, SolarThing won't start running. To enable it so that it starts on reboot, run this:

```
sudo systemctl enable solarthing-custom_rover
```

Next, you probably want to view your data: *Viewing Your Data*.

4.1.7 Docker Setup

This page describes hwo to setup docker.

This page does not yet have documentation on how to run SolarThing in docker, but that documentation should be added soon.

Install Docker

https://docs.docker.com/engine/install/

Configure Docker

Fix memory limit support on Raspberry Pi

If you run `docker info` and get warnings shown below, then you may need to enable memory limit support.

```
...
WARNING: No memory limit support
WARNING: No swap limit support
WARNING: No kernel memory TCP limit support
...
```

To enable memory limit support edit `/boot/cmdline.txt` and add this:

```
cgroup_enable=memory swapaccount=1 cgroup_memory=1 cgroup_enable=cpuset
```

Now reboot your Rasbperry Pi. You can confirm the warnings go away when running `docker info`. You should now recreate your containers that you want to have memory limits and you can confirm they are working by running `docker stats --no-stream`.

Related links:

- https://forums.raspberrypi.com/viewtopic.php?t=325521
- https://dalwar23.com/how-to-fix-no-memory-limit-support-for-docker-in-raspberry-pi/

4.2 Maintenance

4.2.1 Updating

SolarThing gets updates every once in a while, and when it does, you will want to update. You can see updates here: https://github.com/wildmountainfarms/solarthing/releases

Docker Install

Native Install

Navigate to the directory containing your `docker-compose.yml` file. Now do:

```
sudo docker compose pull && sudo docker compose up -d
```

Now you have the latest version!

If your docker compose file references a version other than `latest`, you may have to manually change it and do the above commands.

If you notice an update, run these commands to get it:

```
cd /opt/solarthing
git pull
program/download_solarthing.sh
# Or instead use this if you are running SolarThing Server
program/graphql_download_solarthing.sh
```

Once you have done that, you can restart SolarThing. One would typically restart SolarThing like so:

```
sudo systemctl restart solarthing-<NAME OF YOUR DIRECTORY HERE>
```

If you have multiple SolarThing instances running, or don't remember the name of the ones you have running, you can run this to restart all of them:

```
sudo systemctl restart solarthing-*
```

It's a good idea to make sure they are all running:

```
systemctl status solarthing-*
```

Permission Issues

Note: Permission errors should not occur for Docker Installs

If you got errors while trying to update, run these commands:

```
sudo other/linux/create_user.sh
sudo other/linux/update_perms.sh
sudo usermod -a -G solarthing,dialout,tty,video,gpio $USER
# Or use this if you don't need the gpio (if your system doesn't have the gpio group)
sudo usermod -a -G solarthing,dialout,tty,video $USER
```

4.2.2 Logging

All the SolarThing programs are configured to log. By default, each program's systemd service is configured not to log to journalctl. This saves disk space and allows SolarThing's Log4j configuration to compress or delete old log files.

By default, SolarThing writes 3 different types of log files, each with different information in them. The "summary" log files have the least amount of text in them. "info" log files are in the middle, and "debug" log files have the most information in them.

Viewing Logs

Viewing SolarThing's log files is a good way to see if anything is going wrong, or to just view debug information to see data.

Let's assume that we have a program configured in the `/opt/solarthing/program/custom_rover` directory. To view logs, go ahead and `cd`:

```
cd /opt/solarthing/program/custom_rover/logs
ls -l
```

You will now see all of the logs in the directory. You may see lots of files with a `.log.gz` extension. Those files are the old log files. There should also be three files that have the current logs: `log_summary.log`, `log_info.log` and `log_debug.log`.

To view info logs:

```
less -R log_info.log
```

Note: While you can use nano to view log files, it is not recommended because it will not view colors well and is not good at opening large files. vi and vim are also not good for viewing log files containing color.

Now you can browse around.

You can also look at the logs "live", by using this command:

```
tail -f log_debug.log
```

Note: By default the debug log output is not "flushed" immediately. That means that while viewing it data will be cut off. This is to reduce the amount of time spent writing to disk, which is critical in maintaining a stable system on Raspberry Pis.

If you do not want to switch between `less -R` and `tail -f`, you can just use `less -R` by itself. When you want to start "tailing" a file, just press Shift+F. When you want to go back to browsing the file, just press CTRL+C.

If you want to reload the file while inside of `less -R`, just type Shift+R (`R`). `less` is such a useful tool!

Decompressing logs

As mentioned earlier, old log files have a `.log.gz` extension. If you need to view these log files, you must first decompress them.

Here is an example of how to decompress a particular log file:

```
gunzip -k log_debug_2022.03.20-1.log.gz
less -R log_debug_2022.03.20-1.log.gz
```

4.2.3 System Stability

Once you get SolarThing running, you probably want it to stay running without any interference from you. Some of these can help your system stay stable.

Add a Watchdog

Watchdogs can help detect if your system is frozen, then restart the system.

I could go into detail on how to enable it on Raspberry Pis, but there are plenty better tutorials than I could make such as this one: https://medium.com/@arslion/enabling-watchdog-on-raspberry-pi-b7e574dcba6b or https://diode.io/raspberry%20pi/running-forever-with-the-raspberry-pi-hardware-watchdog-20202/

Use a good SD Card

If you have a Raspberry Pi, you are either using a micro SD card or have set it up to boot off a USB Drive. If you are using a micro SD card, you should do research to make sure it is a high quality SD card. Low quality, cheap SD cards may fail after long periods of running. Using a SD card with more space than you need is also a good way to prevent problems.

Serial Port Stability

If you disconnect and reconnect your serial port, or if its connection isn't perfect, don't worry! If SolarThing fails to communicate with your device enough times, it will attempt to reconnect the serial port.

4.3 Installing Software

This page links to other pages to help you install certain software

4.3.1 Install CouchDB

Note: CouchDB cannot be installed on Raspberry Pi 1s or on a Raspberry Pi Zero

Note: It is not recommended to run CouchDB on a Raspberry Pi. This is because databases use lots of I/O operations, which is not good for the Pi's SD card long term.

This documentation is not going to tell you exactly how to install CouchDB. Figuring that out for whatever device you install it on is up to you.

https://docs.couchdb.org/en/stable/install/index.html

CouchDB Docker Install

If you choose to use Docker, you can use docker compose.

Note: The example file structure shown below assumes that you are installing CouchDB on a device different than what the uploader program is configured on. If that is not the case, you may append configuration to your existing `docker-compose.yml` file.

Here is an example file structure:

```
~/Documents/containers/solarthing/
├── couchdb/
│   ├── data/
│   └── etc/
└── docker-compose.yml
```

```
mkdir -p ~/Documents/containers/solarthing/couchdb/{data,etc}
cd ~/Documents/containers/solarthing
mkdir couchdb
cd couchdb
touch docker-compose.yml
vi docker-compose.yml
```

Then copy this into your `docker-compose.yml`.

```
services:
  couchdb:
    image: 'apache/couchdb:3'
    restart: unless-stopped
    environment:
      - 'COUCHDB_USER=admin'
      - 'COUCHDB_PASSWORD=password'
    ports:
      - '5984:5984'
    volumes:
      - './couchdb/data:/opt/couchdb/data'
      - './couchdb/etc:/opt/couchdb/etc/local.d'
  # NOTE: You may put a SolarThing server service here later when you configure it.
```

You can login to your CouchDB instance with user `admin` and password `password`. Please change your password when you login.

Note: Although you could set your password in `docker-compose.yml`, you should use `docker-compose.yml` to set a temporary password then change your password in the web interface. The reason for this is that it is bad practice to store your password in plain text, and setting your password in CouchDB's web interface will avoid your password being stored in plain text.

Once you have changed your password, you may choose to remote the `environment:` section from your couchdb's service configuration in the `docker-compose.yml` file.

Basic CouchDB Setup

Newer installs of CouchDB require some setup. Note this is different from the SolarThing setup required to use the database. Make sure that these databases already exist or create them if they do not:

Create a database called `_users` and a database called `_replicator`. The default options for creating these databases are fine.

Keep user logged in

Sometimes you may want to keep users logged in for longer than the default. Go to the settings page: http://127.0.0.1:5984/_utils/#_config. You will configure the `timeout` setting: https://docs.couchdb.org/en/3.2.2-docs/config/auth.html#chttpd_auth/timeout. Add an option with section: `chttpd_auth`, name: `timeout`, and value in seconds. I recommend 3600 for 1 hour timeout.

4.3.2 Install InfluxDB

Although CouchDB is the recomemnded database for SolarThing, InfluxDB can also be used.

Note: SolarThing treats InfluxDB 1.X installations different than InfluxDB 2.X installations

InfluxDB 2.X is recommended and more up to date than 1.X versions. You can install it here: https://docs.influxdata.com/influxdb/latest/install/

If you have to use InfluxDB 1.X for some reason, you can learn about it here: https://docs.influxdata.com/influxdb/v1.8/introduction/download/

4.3.3 Install Java

So you need to install Java to your system?

- https://pi4j.com/documentation/java-installation/
- https://forums.raspberrypi.com//viewtopic.php?p=1308846

Required Version

SolarThing requires Java >= 11 and SolarThing Server requires Java >= 17. If you are able to install Java 17, you should do so, otherwise 11 is fine .

Install on Systems with Apt

The `apt-get` or `apt` command is used to install most software. To install Java 11, run this command:

```
sudo apt-get update
sudo apt-get install -y openjdk-11-jdk-headless
```

Raspberry Pi 1 and Raspberry Pi Zero

Note: Because RPi 1 and RPi Zero have poor Java support and are slow devices in general, SolarThing does not officially support these devices.

Installing Java on a Raspberry Pi 1 and Raspberry Pi Zero is difficult because both of these devices have an ARMv6 architecture. You will find that installing any version other than Java 8 does not work via `apt-get`. You can find Zulu's list of OpenJDK builds here. As you can see, Zulu provides ARMv6 support up to Java 11. The simplest way to install these is to use SDKMAN, but currently there is no documentation for getting SolarThing to use an SDKMAN installation.

4.4 Deveopment

This contains documentation for contributing to SolarThing.

4.4.1 Developer Setup

Developing SolarThing does not require an IDE, but it is highly recommended. Typically IntelliJ Idea is used. To install it, you should first install JetBrains Toolbox to make its installation and updating easier: https://www.jetbrains.com/toolbox-app/.

When compiling the majority of SolarThing, the only thing that is required is Java 11 or higher to be installed. Installing Java on your developer machine is the same as installing it on the machine that is running SolarThing. You can go here to learn how to install it: doc:*/software/java*. Alternatively, you can use SDKMAN to install Java only for developing SolarThing: https://sdkman.io/install.

Cloning SolarThing

When SolarThing is installed on a device such as a Raspberry Pi, it is typically installed in the `/opt` directory. When developing SolarThing, you should install it in a different directory. Typically you can do something like this:

```
$ mkdir ~/workspace/
$ cd ~/workspace/
$ git clone https://github.com/wildmountainfarms/solarthing
```

If you want to make commits, I recommend that you fork it and clone your own, forked, repository instead of cloning `https://github.com/wildmountainfarms/solarthing`.

To see if you can compile the main SolarThing jar, you can run this in your terminal:

```
./compile_and_move.sh
```

Open in IntelliJ

File -> Open, then navigate to the place where you cloned `solarthing`.

Further Setup for server and web module

The server module depends on the `web` module, which contains a React application. This React application requires `npm` and `node` to be installed on your system. A tool commonly used to install and manage the these commands command is nvm, which can be installed here: https://github.com/nvm-sh/nvm#install–update-script. Once `nvm` is installed, you can use it to install like so:

```
# Install Node 16 (which will automatically install a recent npm version as well)
nvm install 16
```

Then you can check the versions of the tools you just installed. You should get outputs similar to these:

```
$ node --version
v16.15.1
$ npm --version
8.11.0
```

Now you must install the web module's dependencies before being able to compile it:

```
cd web/
npm install
cd ../
```

You can now compile SolarThing Server:

```
./server_compile_and_move.sh
```

Note: If you already have a gradle daemon running, you need to kill using `./gradlew --stop` before compiling. If you don't, you may get an error such as > `A problem occurred starting process 'command 'npm''`, which indicates that gradle cannot find the `npm` command.

Running SolarThing Server from IntelliJ

Once the above steps are completed, it is typically easier to run straight from IntelliJ, rather than running the jar file that is generated using `./server_compile_and_move.sh`. You can do this by going to the right side of IntelliJ and opening up the Gradle tab. Under the `server` module, expand `Tasks`, expand `application`, then double click `bootRun`.

Configuring SolarThing Server

No matter how you run Solarthing Server, you must configure it. You might have a `couchdb.json` file already created. If you don't already have that file placed in `program/config`, you can place it there. Then, create a new file in `program/graphql/config` named `application.properties`. Paste this line in:

```
solarthing.config.database=../config/couchdb.json
```

You should now be able to run SolarThing Server without errors by running the `bootRun` task. Navigate to http://localhost:8080 to see if it successfully connects to your CouchDB instance and shows some data.

Testing the Main SolarThing Program

Running the main SolarThing program should be done just like normal when developing. You will compile it using `./compile_and_move.sh`. If you want to run it on your computer, great, go for it! If you want to run it on another device, you can use the `./copy_jar.sh` command like: `./copy_jar.sh pi@myipaddress`. It will prompt for a password unless you have SSH Public Key authentication set up on your device.

4.4.2 CouchDB Develop

This page aims to document some best practices when testing and setting up CouchDB while developing.

Running

While developing, this is a good way to create a temporary instance with a login of `admin/relax`. You can change the password once the instance is running, but there should be no need to.

```
docker run -p 5984:5984 -e COUCHDB_USER=admin -e COUCHDB_PASSWORD=relax -d couchdb:3
```

4.4.3 SolarThing Cache Database

The `solarthing_cache` database is a complicated, but can be broken down. This page aims to document how it is currently used and how one can use it to cache data that may be expensive to calculate and query.

Database Structure

Unlike other databases in SolarThing, `solarthing_cache` is not set up to be queried using a view. It must be queried using the ID of the document. The documents are named like so: `cache_<time start>_<duration>_<source ID>_<cache name>`.

Querying this data usually means making a bulk get request.

Packet Structure

Although there are many different cache types, all documents in the database share a common base structure.

```
{
  "_id": "cache_2021-06-23T20:30:00Z_PT15M_default_chargeControllerAccumulation",
  "_rev": "31-1da15edb1b141b22d163a6fc8b7901d5",
  "periodStartDateMillis": 1624480200000,
  "periodDurationMillis": 900000,
  "sourceId": "default",
  "cacheName": "the cache name here",
  "nodes": [
    {
      "fragmentId": 1,
      "data": {
        // ... node data here
      }
    },
    // ... more nodes
  ]
}
```

An entire packet represents one small interval (usually a 15 minute interval). Each node inside of the packet represent the data for a SINGLE device for that interval.

Triggering Document Generation

Packets are generated when a particular start time and interval are requested. If the document does not existing in the database with the document ID corresponding to the start time, interval, source ID, and cache name, then that document will be generated.

Triggering Document Replacement

Packets in the database are not replaced unless there is an error while trying to parse the data from that document. When an error occurs, it is assumed that SolarThing has been updated in a way so that a particular cache document is no longer valid. Maybe it doesn't have enough data anymore, or maybe it has too much data, or maybe an error was thrown by SolarThing on purpose to indicate that the old cached data is bad and should not be used.

Unless a particular cache type is being actively developed, it is likely that for most cache types, they will be generated once and left in the database forever.

Cache Cleanup

Currently SolarThing does not have a way to remove old cache data. There is no way to tell if a particular document is actively being used, so there is no perfect solution to this. The cache database does not take up much space, so this is not a concern.

`chargeControllerAccumulation` Cache Type

The `chargeControllerAccumulation` cache type is one that keeps track of the amount of energy generated by a charge controller over a certain interval.

An example `data` object looks like this:

```
{
  "identifier": {
    "type": "outback",
    "address": 3
  },
  "generationKWH": 0.099999905,
  "firstDateMillis": 1624480199471,
  "lastDateMillis": 1624481099437,
  "unknownGenerationKWH": 0,
  "unknownStartDateMillis": null
}
```

The data is pretty easy to understand, but there are some important things to know. You have an identifier object, which represents what device this data was generated from.

You have a `firstDateMillis` and `lastDateMillis`, which represent the timestamps of the first and last status packets that were used to generate this data. Those status packets belong to the device identified by that identifier. There may be any number of packets in between the first and last packets, but that information is irrelevant. It is also possible for there to be 0 packets in between the first and last packets used for generation, and it is possible that the first packet IS the last packet.

Almost always, `firstDateMillis` is actually outside of the interval for this document. It is usually the last packet of the previous interval. The reason for this is because let's say that we have two intervals right next to each other. 13:15-13:30 and 13:30-13:45. The first interval has data from 13:14 to 13:29 and the second interval has data from 13:29 to 13:44. Let's say that during the first interval 1.0 kWh was reported and during the second interval 1.0 kWh was reported. Let's say that the device read these values at these times: `13:14=4.5kWh`, `13:29=5.5kWh`, `13:31=5.6kWh`, `13:44=6.5kWh`. Now, between 13:14 and 13:44, you can see that a total of 2.0 kWh was generated. However, if the generation of this data used 13:31 as the start packet for the second interval, the generated values would end up being 1.0 kWh and 0.9 kWh respectively. For this reason, the last packet from the previous interval is used for the start packet to make sure no data is "left behind" when generating the data.

In this case we have a `generationKWH` which represents the energy generated during this period.

"Unknown" Data in `chargeControllerAccumulation`

In the above section, that particular piece of data is "known" because `startDateMillis` and `endDateMillis` are not null. "Known" pieces of data can have unknown components to them, which represents the accumulation of preceding periods where data in those periods for that particular device were "unknown".

A particular piece of data is completely unknown if `startDateMillis` is null and `endDateMillis` is null. If that is the case, then that "unknown" data does not have an "unknown" component to it. It is only "unknown".

In the above section, there is an example `data` object that has `unknownGenerationKWH` = 0 and `unknownStartDateMillis` = null. This means that particular piece of data is "known", and it has no "unknown" component.

Here's an example of data that would cause the following results:

```
# The following values are the timestamps of MX3 packets and the reading of MX3's kWh␣
↪field
12:59=3.3
----- 13:00
13:01=3.3
13:09=3.5
13:14=3.6
----- 13:15
----- 13:30
----- 13:45
13:46=4.0
13:50=4.2
13:55=4.3
----- 14:00

Results:
13:00-13:15 generationKWH=0.3, start=12:59, end=13:14, no unknown component
13:15-13:30 unknown
13:30-13:45 unknown
13:45-14:00 generationKWH=0.3, start=13:46, end=13:55, unknownStart=13:14,␣
↪unknownGeneration=0.4
```

You can now see that the unknown component represents the accumulation of data from previous intervals that were unknown all the way back to the last "known" interval. This allows us to say "I know that 0.3 kWh was generated between 13:45 and 14:00", and "I know that sometime between 13:14 and 13:46 0.4 kWh was generated. I don't know if all of that 0.4 kWh came from one period or another, I just know that it happened."

Unknown components are necessary to prevent accumulation data from becoming lost if a device disconnects for an extended period of time.

Combining Intervals of Data

In the cache database, data always goes in at fixed intervals. However, when you take that data out of the database (or calculate it yourself), having a bunch of 15 minute intervals usually isn't that useful. Maybe you want hour long intervals or day long intervals. Each cache type supports combining two intervals of data right next to each other. So, you can combine intervals 13:15-13:30 and 13:30-13:45 to make a 13:15-13:45 interval. You can keep combining intervals of data until you get a bunch of intervals that you want, or more commonly combining all the intervals so you can get a single piece of data, such as the generation kWh for a single day.

Let's say that we combine a two `chargeControllerAccumulation` types of data. `13:15-13:30 = 0.5 kWh generated` and `13:30-13:45 = 0.6 kWh generated`. Combining them gives us a result of `13:15-13:45 = 1.1 kWh generated`.

Now let's say that we have two intervals, both with unknown components.

```
14:00-15:00 generationKWH=1.3, unknownGenerationKWH=0.2
15:00-16:00 generationKWH=0.9, unknownGenerationKWH=0.1

Combined result:
Interval: 14:00-16:00
generationKWH=1.3+0.9+0.1 = 2.3
unknownGenerationKWH=0.2
```

We see that to calculate the new `generationKWH`, we add up both `generationKWH`, then also add the later interval's unknown component. The unkonwn component of the first interval remains in the resulting combining.

You can see this logic for yourself here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/type/cache/packets/data/C

`batteryRecord` Cache Type

The `batteryRecord` cache is one that keeps track of the minimum and maximum battery voltage, along with the battery volt hours for an interval of time, for a single device. Minimum and maximum battery voltages are useful, but battery volt hours is usually used to determine the average (integral over interval).

An example battery record's `"data"` object may look like this:

```json
{
  "identifier": {
    "type": "outback",
    "address": 1
  },
  "firstDateMillis": 1655321348930,
  "lastDateMillis": 1655322224927,
  "unknownStartDateMillis": null,
  "record": {
    "minBatteryVoltage": 24.4,
    "minBatteryVoltageDateMillis": 1655321543926,
    "maxBatteryVoltage": 25.6,
    "maxBatteryVoltageDateMillis": 1655321447930,
    "unknownBatteryVoltageHours": 0,
    "unknownDurationMillis": 0,
    "batteryVoltageHours": 6.056228924143589,
    "knownDurationMillis": 875997
  }
}
```

The data keeps track of the time when the battery was at a minimum, and the time it was at a maximum.

You'll notice that there is something called `knownDurationMillis`. This is the total duration that the `batteryVoltageHours` integral has been calculated over. For a packet in the database `lastDateMillis - firstDateMillis`, is exactly the same as `knownDurationMillis`. When we get into combining later, we will see how these two values may end up being different.

"Unknown" data is similar to that of a `chargeControllerAccumulation` cache. Unknown data represents data from the last known period up to the start of this period. The main difference is the precision of the data. Charge controllers report their power integral (energy), so even if a charge controller disconnects, the amount of energy it generated can be reliably calculated. This is not the case for a `batteryRecord`'s volt hours calculation. An unknown component is always calculated using two data points: The last battery voltage from before the current period, and the first battery voltage in this period. That means that while `unknownBatteryVoltageHours` does have the volt hours unit, it can be an unreliable estimate of what actually happened during the (possibly multiple) unknown periods.

Combining `batteryRecord`

Combining two battery record cache periods is relatively simple. The minimum of the minimums becomes the new minimum and the maximum of the maximums becomes the new maximum. The battery voltage hours gets added up along with the known duration hours. The tricky part, is what happens to unknown data. If we followed this exactly how the charge controller cache handles this, the unknown component of the later period would be added to the known component of the resulting combination. However, remember that an unknown component was calculated using two data points, and is extremely inprecise. Because of this, a battery record cache also has a "gap" component, which represents unknown components somewhere in the middle of the period. This allows someone to include or exclude the gap component in the calculation of an average. This gives choice to the users of this data so they can either ignore the "gap" component, or just add it onto the known component.

You can see the logic of combining two battery record caches here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/type/cache/packets/data/BatteryRecordDataCache.java

`IdentificationCacheNodeCreator`: generating data

IdentificationCacheNodeCreator is an interface that is used for generating data for a single device. You can view it here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/rest/cache/creators/IdentificationCacheNodeCreator/java

The interface is pretty simple. `getAcceptedType()` should return the class of the type to accept. For instance `BatteryVoltage` to get any type of device that provides the battery voltage. `getCacheName()` should return the name of the cache, which should be unique among all cache names. Some examples of names already in use are `batteryRecord`, `chargeControllerAccumulation`, and `fxAccumulation`. New names should use camelCase, and should NOT be redundant by including "cache" in the name.

The most important method of this interface is the create method. The parameters taken are as follows:

- `identifierFragment`: Represents the fragment and identifier for a given device
- `packets`: The packets of the type given by `getAcceptedType()`. Note that these packets may and will be out of the range of the given period. The implementation should filter for the given period or use the extra data for *smart* calculations. You can assume these are sorted in ascending order.
- `periodStart`: The start time of the period that data will be returned for
- `periodDuration`: The duration of the period that data will be returned for

One question one might have is why so much data is provided. The reason for this is because we want and period to be able to calculate its "unknown" component by backtracking up to ~4 hours before the period even started. In fact, any time this is called, it is expected that data should be provided for at least 4 hours before the start of the period. Usually, much more data will be provided when calling this method, but it should not use all of that. By not using all of the data provided, reproducible caches are possible so no matter how much data is provided in the `packets` list, the same result should occur each time for a given cache and device.

The data returned is an `IdentificationCacheNode`, which holds the fragment of the device and also the data, which is required to hold the identifier of the device because the data is of the type of `IdentificationCacheData`, which is the common type for data that can be combined.

So to recap, when calling the `create` method of a `IdentificationCacheNodeCreator`, data for a single device is provided, and the returned value is the data for that device for the given period. A `IdentificationCacheNodeCreator` only deals with a single device and a single period at a time!

`CacheCreator`: generating data

A `CacheCreator` is at a lower level of abstraction than a `IdentificationCacheNodeCreator`. The result from a `CacheCreator` is what is stored right in the database. It has a single method: `createFrom()`. This method takes a source ID, a list of packets, and the period. This list of packets follows the same 4 hour rule that `IdentificationCacheNodeCreator` follows, as its implementation directly calls `IdentificationCacheNodeCreator`'s `create()` method.

`DefaultIdentificationCacheCreator`: `CacheCreator` implementation

A `CacheCreator` doesn't have to necessarily deal with `IdentificationCacheNodeCreator` s. The main (and only) implementation of `CacheCreator` is `DefaultIdentificationCacheCreator`, which takes a `IdentificationCacheNodeCreator`.

This implementation is the lowest level of abstraction for data generation. Any lower and we'll start getting into the logic for determining what periods to generate, cache, and store in the database.

`BatteryRecordCacheNodeCreator` implementation

`BatteryRecordCacheNodeCreator` implements `IdentificationCacheNodeCreator` and has the main logic for generating `batteryRecord` caches. View it here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/rest/cache/creators/BatteryRecordCacheNodeCreator.java

Cache Logic

The actual logic for generating caches and storing them in the database is present here: solarthing/blob/master/server/src/main/me/retrodaredevil/solarthing/rest/cache/CacheHandler.java

Usages of caches

Currently, the SolarThing Server program exposes a REST API for querying cache data, which then will call methods provided by `CacheHandler`. You can see this here: solarthing/blob/master/server/src/main/me/retrodaredevil/solarthing/rest/cache/CacheController.java

Currently, nothing actually uses that REST endpoint, but there are usages of `CacheController` in some of the GraphQL queries. Typically, a GraphQL query that needs to use cache data will be provided a `CacheController` object. You can see an example here: solarthing/blob/master/server/src/main/me/retrodaredevil/solarthing/rest/graphql/service/SolarThingGraphQLLongTermService.java

Creating your own cache

If you would like to make your own cache, then you first need to decide a couple of things. We will assume that you would like to make a cache that is based around data for each device of a certain type. In this case, we will be implementing the `IdentificationCacheData` interface, or more likely, we will be extending an abstract implementation of that called `BaseAccumulationDataCache`. Now, let's come up with some sort of name for our cache such as "cheese sandwich cache".

We will create a class called `CheeseSandwichDataCache`, which will extend `BaseAccumulationDataCache`. This class should be created in the `core` module under the `me.retrodaredevil.solarthing.type.cache.packets.data` package. We should create a field like so: `public static final String CACHE_NAME = "cheeseSandwich";`. We should annotate our class with `@JsonExplicit`, and create a constructor annotated with `@JsonCreator`. You can see an example here: solarthing/blob/master/core/src/main/java/me/retrodaredevil/solarthing/type/cache/packets/data/BatteryRecordDataCache.java

You should populate your newly created class with useful data and implement the `combine()` method.

Now, we need to create our class to generate the data. We will create a class called `CheeseSandwichCacheNodeCreator` in the `graphql` module under the `me.retrodaredevil.solarthing.rest.cache.creators` package. This should implement `IdentificationCacheNodeCreator<CheeseSandwichDataCache, CheeseSandwichStatusPacket>`. Note that you can replace `CheeseSandwichStatusPacket` with whatever type of class that you need to use to generate data. Note that class must implement the `Identifiable` interface AND, the identifier returned must be serializabe and deserializable to and from JSON. Please make sure you check to make sure that the type of identifiable used by that class is present in the `@JsonSubTypes` in the `Identifier` interface. You can see an example here: solarthing/blob/master/server/src/main/java/me/retrodaredevil/solarthing/rest/cache/creators/BatteryRecordCacheNodeCreator.java.

Now that you have your class created, it's time to implement the required methods. The `getAcceptedType()` method can return the class of what you replaced `CheeseSandwichStatusPacket` with. `getCacheName()` returns `CheeseSandwichDataCache.CACHE_NAME`. Now it's time to implement the create method. This is where you may have to get creative to create the perfect algorithm to generate your data, or you can look at one of the many implementations of this method for other types of cache data.

Once you are done, go into the `CacheHandler` class, and add an entry similer to the other entries under the `CACHE_CREATORS` field. Now go to `CacheController` and add a method similer to the other methods already present, but for your data types. To test it, use something to hit the newly created endpoint that you have made, and see if it works.

4.4.4 SolarThing Server GraphQL Queries and Mutations

In SolarThing there is a whole package dedicated to defining GraphQL endpoints for the SolarThing Server here: solarthing/tree/master/server/src/main/java/me/retrodaredevil/solarthing/rest/graphql/service

The `/graphql` endpoint is used to make GraphQL requests. There are many different possible queries to choose from. The structure of many of the queries are designed in such a way to make them easily usable with *Wild GraphQL Data Source <https://github.com/wildmountainfarms/wild-graphql-datasource>*.

The cool thing about GraphQL, is that you can tell it to query something, then only ask for some of the result. By asking for only some of the result, it may not have to do all the calculations. In the case of SolarThing, we can ask it to query all the status packets in a time period, then we can sort and calculate data using that data that has been queried. A good example of this is the `queryStatus` query defined in `SolarThingGraphQLService`. This query will get some data, then return a `SolarThingStatusQuery`, where we can then ask for very specific data in a specific format and get the exact data that we want. That particular format is very accessible in graphql-datasource.

GraphQL Usage in web module

The web module contains the code for SolarThing Web that is bundled with SolarThing Server. This website is completely backed by GraphQL queries and mutations that define how the website works and functions. You can see the directory containing the queries here: solarthing/tree/master/web/src/graphql.

You will notice that all of the queries are named, and many of them take in query variables. Each query can be tested to be correct against the schema, and each query has its own function automatically generated for it. At the low level, some npm command is run to generate code based on these queries. For our purposes, all we need to know is that running ./gradlew web:generateCode will generate the code for us. When running that, it will also generate the GraphQL schema for us so it knows that our queries are correct, and so that it can provide the generated code with rich type information, which is very important because it gives us the same typing that the Java code has.

The web module uses React, so most of the time it doesn't look like you are querying the GraphQL endpoint, as much of the boilerplate is taken away from you so you can deal with the result. Much of the time, the use of a query will look like this:

```
const {data, error, isLoading, isSuccess} = useHomeQuery(graphQLClient, { sourceId,␣
↪currentTimeMillis: "" + timeMillisRounded});
```

Testing GraphQL Endpoint on Command Line

https://github.com/Urigo/graphql-cli

4.5 Miscellaneous

4.5.1 Alternatives to SolarThing

SolarThing isn't perfect. You might be able to find what you're looking for elsewhere.

Note: I have not personally tested any of these libraries, so use and rely on them at your own risk.

Renogy Rover Alternatives

These alternatives are for Renogy Rover (SRNE like) charge controllers. At the time of writing, these do not appear to support inverters or hybrid inverter charge controllers.

- https://github.com/corbinbs/solarshed
  - The most complete Python library for communicating with a Renogy Rover
- https://github.com/Olen/solar-monitor
  - Designed to work WITH a Bluetooth Module, rather than a USB to serial cable
- https://github.com/logreposit/renogy-rover-reader-service
  - Written in Kotlin
  - Available on Dockerhub
- https://github.com/menloparkinnovation/renogy-rover
  - Written in JavaScript

- https://github.com/floreno/renogy-rover-modbus
  - Written in JavaScript
- https://github.com/CyberRad/CoopSolar
  - Simple Python script
- https://github.com/amigadad/SolarDataCollection
  - Based off of solarshed, but supports more parameters

Tracer Alternatives

- https://github.com/kasbert/epsolar-tracer

Outback MATE Alternatives

These alternatives are for the Outback MATE 1 and MATE 2.

- https://github.com/jorticus/pymate
  - matecom.py does a similar thing to SolarThing
  - Most of this library is designed to emulate an Outback MATE to communicate with devices using Outback proprietary protocol.
    - If you use these features of this library, a MATE is not actually required.

4.5.2 History of SolarThing

This program started in the summer of 2017. This page talks about the history of SolarThing and has some anecdotes about design decisions that were made.

Inspiration

@eidolon1138 is the one who originally came up with the idea to collect data from his Outback Mate device. He helped set up the database and @retrodaredevil did the rest. Eventually @retrodaredevil created an android app making it much more convenient than a website.

@retrodaredevil came up with the idea of the outhouse status when she walked all the way out to the outhouse only to find that it was occupied! She walked all the way back inside, then went back out a few minutes later. She knew that something had to be done about this first world problem. The outhouse status is no more, but it was a fun little experiment.

SolarThing looks a lot different from its first fully functional version in 2018, but it's fun to look back on what it looked like years ago.

Timeline

2017

- A perl script was created in a single day to collect data from an Outback Mate serial port
- The terrible perl script was ditched to start on the Java program. The program allowed packets to be added to a CouchDB database

2018

- This continued in the summer of 2018. The formatting of the packets was completely rethought. The web application was created and completed in less than a week.
- An Android app was created to see the data continuously updated in a status notification

2019

- Outhouse status was added
- Renogy rover support was added
- To maintain compatibility with the previous packet structure, Source and Fragment packets types were added to have the ability to have multiple instances uploading packets to a single database
- InfluxDB support was added allowing for easy configuration of a Grafana dashboard
- Raspberry Pi running the outhouse program didn't survive the freezing temperatures (RIP outhousepi 2019-2019)

2020

- PVOutput was setup
- Outhouse code was completely removed from SolarThing codebase
- "Events Display" was added to the Android application
- RoverPi became corrupt, but was eventually re-flashed and set up again
- MatePi finally became corrupt. It was re-flashed, worked, then became corrupt again. (Need new SD card).
- Google Analytics were added
- Did a huge rebase to remove all solarthing.jar files, restructured program directory
- Added ability to read from DS18B20 temperature sensors
- Added GraphQL program
- Added message sending abilities for push notifications (Mattermost and eventually Slack)
- Added automation program that can help automate sending of commands and eventually became the base program for the message-sender program and other general 'actions'
- Added home assistant upload action
- Added ability to send data to Solcast along with GraphQL queries for Solcast
- message-sender program usable through automation program (removed message-sender program)

2021

- InfluxDB 2.0 Support
- DCDC Controller monitoring bug fixed
- Pzem Shunt Support Added
- MQTT Support added
- Switched from Ektorp to custom CouchDB library
- Started to make some aspects of SolarThing use their own thread
- The buried cable connecting the MATE2 failed. MATE was moved into the Battery Room. The RPi0 was set up to monitor it, but soon the RPi0 died for an unknown reason and was replaced by an RPi3. Now most monitoring runs off that RPi3.
- Deprecated `rover` program to be replaced with `request` program.
- Tracer support added (https://github.com/wildmountainfarms/solarthing/releases/tag/v2021.6.0)
- Alter database added
  - This adds support for scheduled commands that can be requested or canceled via a slack chatbot
- GraphQL (SolarThing Server) program uses Java 11
- Added `/command/run` endpoint to SolarThing server to allow sending of commands from Grafana

2022

- Documentation was made available on readthedocs
- Added a React frontend to SolarThing Server (formerly SolarThing GraphQL)
- Building of jar files automated using GitHub Actions
- Local actions are now supported

2023

- ActionLang released (https://github.com/wildmountainfarms/solarthing/releases/tag/v2023.2.0)
- Docker support
- Java 11 required (Java 17 required for SolarThing Server)
- Google Analytics removed

2024

- Development started on Wild GraphQL Data Source for Grafana
  - Officially released in April https://grafana.com/grafana/plugins/retrodaredevil-wildgraphql-datasource/

Anecdotes

Legacy Perl Script

solarthing/blob/c9069b8993b783c664705a36fd6c30965d7748f4/other/legacy/helloworld.pl is a legacy program. It was the program that started SolarThing. After learning perl for a day. I went straight back to Java, which I am more familiar with.

Moving from Gson to Jackson

This project started out with Gson, but as of 2019.12.24, I have started to move to Jackson. I originally chose Gson for its simplicity. It has served this project very well and is very user friendly. However, I got tired of writing custom deserializing functions to deserialize advanced packets. Jackson is very annotation orientated and is very feature rich. The added complexity of Jackson is worth the speed of development it brings.

Configuration in the Early Days

When developing SolarThing, I didn't want to hard code values everywhere in the code, so I decided to go with command line arguments. For this, I decided to use JCommander.

JCommander was a great option until I wanted to use inheritance to define which types of programs can have certain options. JCommander did not work with interfaces an JewelCli did. JewelCli is like the Retrofit of command line parsers. Defining options in interfaces gives you many options for how to structure your configuration. If SolarThing or another one of my projects needs command line parsing again, JewelCli will be my go to library.

At this point, the command line arguments were pretty crazy. Plus, swapping out different configs meant changing the file that actually ran the `java -jar` command. I knew it was time to move to JSON configuration. This allowed for a lot of flexibility. While GSON was used to start with, the JSON configuration code was one of the reasons I felt like I needed to rewrite a lot of the stuff that used JSON. I wasn't utilizing Gson's deserialization features, so I decided to switch to Jackson altogether as explained above.

Currently the configuration is very easy to change. I can swap out what configuration I'm using easily and can use the same CouchDB or InfluxDB configuration on multiple devices running SolarThing.

Getting Data into Grafana

When support for InfluxDB was added in late 2019, it became easy to make a Grafana dashboard to display data. However, this was not perfect. I had to maintain two different databases. CouchDB for nicely structured JSON data, and InfluxDB for easy to query data. In 2020, I decided I wanted to be able to query data from CouchDB without InfluxDB. After some searching, I found the graphql-datasource for Grafana. It was perfect. I did some research on how to do a code first approach for a GraphQL program and ran into graphql-spqr. Now my schema was already created without additional setup because of how awesome Java is. Now I could query CouchDB from Grafana and even add additional data calculations that weren't in the packets to begin with.

4.5.3 Wild Mountain Farms

This contains documentation and information specifically about Wild Mountain Farms and our setup and use of SolarThing.

Our System

This page contains informal documentation of what our system consists of.

Devices

- 1:FX 2:FX 3:MX 4:MX
- Renogy Rover PG 40A
- EPEver Tracer2210AN (20A)

Inverters

Both FX's are 3400 watts each

Panels

- 2310 Watts across 14 panels (14 165 Watt panels)
  - Angle varies throughout the year as it is adjusted when necessary
- 8 panels (4s2p) 75W each for 600W
  - Realistically I'd say this is probably 350W because of how old these panels are
  - Angle is 47 degrees from horizontal. Note these panels are also tilted maybe 10-20 degrees
  - Siemens SP75
    - 4.8A Short Circuit, 4.4A rated
    - 21.7V Open Circuit, 17V rated
- 2 panels (2s) 305W each for 610W
  - Angle is 47 degrees from horizontal (note front panel is actually 45 degrees). Note these panels are also tilted maybe 10-20 degrees
  - QCELLS - Q.PEAK-G4.1 305
    - 9.84A short circuit, 9.35A rated
    - 40.05V open circuit, 32.62V rated

FX 1 charging config

absorb setpoint 29.2 vdc absorb time limit 1.5 hours float setpoint 27.2 vdc float time period 1.0 hours refloat setpoint 25.0 vdc equalize setpoint 30.0 vdc equalize time period 2.0 hours

4.5.4 Legacy Configuration

This contains legacy configuration that should only be used if you are using an older version of SolarThing and cannot upgrade.

Headless Device Setup

This page helps you setup your device such as a Raspberry Pi or similar device.

Warning: This page may be outdated. For up to date documentation and for a more consistent experience across different single board computers, we recommend to instead setup *DietPi Setup*.

Headless Raspberry Pi Setup

Warning: This page may be outdated. For up to date documentation and for a more consistent experience across different single board computers, we recommend to instead setup *DietPi Setup*.

If you want to use your Raspberry Pi without a monitor and have it automatically connect to your WiFi without having to attach a keyboard and monitor, this'll help. You can do all SolarThing configuration either using a monitor and keyboard, or "headless" over SSH without a monitor or keyboard.

If you are not familiar with Raspberry Pis at all, you may have a better experience using a monitor and keyboard rather than following the instructions below.

Download Image

If you haven't already flashed a micro SD card with Raspberry Pi OS, go here to download it: https://www.raspberrypi.com/software/operating-systems/

Caution: No matter what, I do not recommend that you use "Raspberry Pi OS with desktop and recommended software" as it takes forever to update if you don't remove any of the software.

Note: Since we are doing a headless install, you should use "Raspberry Pi OS Lite"

Flash Image

Use software such as balenaEtcher to flash a micro SD card: https://www.balena.io/etcher/

Once flashed, you should remove, then reinsert your micro SD card into your computer so that the drives automatically mount.

Enable SSH

SSH allows you to access your Raspberry Pi from a computer on the same network. When you reinserted your micro SD card, one or two drives should have showed up. Navigate into the one called the "boot" drive.

Inside the boot drive, add a file named `ssh` with nothing in it. Note that there should be no file extension on it.

Adding a WiFi Network

Unless you are using Ethernet, you'll want your Raspberry Pi to connect to WiFi as soon as you plug it in. Once again, navigate to the "boot" drive.

Inside the boot drive, create and start editing a new file named `wpa_supplicant.conf`.

Note: You'll want to make sure that Windows or Mac OS isn't hiding the file extension as you do not want to end up with a file such as `wpa_supplicant.conf.txt`

In the file, add the following content:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=US
network={
    ssid="YOUR SSID HERE"
    psk=YOUR PASSWORD HERE
}
```

If you would like to use a program that does this automatically, take a look at this: https://github.com/retrodaredevil/headless-setup. Note that it is a bit more work to set up as you have to have Python and pip installed on your system.

Plug in your Raspberry Pi and you're good to go!

Finding your Pi's IP Address

Once you plugged in your Raspberry Pi, it should connect to your WiFi network. You can go to your router's webpage to try and figure out what IP it got.

Using SSH

If you set everything up correctly, you can ssh into your pi by using username: `pi` and password `raspberry`. On Linux or Mac OS, open a terminal and type `ssh pi@<YOUR IP>`, then enter `raspberry` when it prompts for a password. On Windows, you can use something like Putty.

Updating your Raspberry Pi

Before installing SolarThing, it's a good idea to update everything on your system beforehand and install a few necessary tools.

```
pi@raspberrypi:~ $ sudo apt update
pi@raspberrypi:~ $ sudo apt install git curl wget less zip
pi@raspberrypi:~ $ sudo apt dist-upgrade -y
```

Headless Armbian Setup

Warning: This page may be outdated. For up to date documentation and for a more consistent experience across different single board computers, we recommend to instead setup *DietPi Setup*.

Armbian is an operating system that runs on many boards including but not limited to:

- Orange Pi
- Odroid

This document focuses on the setup without using a monitor or keyboard.

Downloading Armbian

You can go here to download Armbian for your specific device https://armbian.hosthatch.com/archive/.

Connecting to a network

Many devices do not have built in WiFi, so you will have to connect them to Ethernet. Even if it does have built in WiFi, there is not a way that I know of to set up a WiFi connection without a keyboard and monitor. You can setup WiFi later.

Using SSH

Armbian enables SSH by default, so as long as you know your IP from your router, you can SSH into it.

```
ssh root@YOUR IP
```

The password is by default, `1234`. It will ask you to change it on first login.

Connect to WiFi

If you have a USB adapter or your device has built in WiFi, use the `armbian-config` command to configure the WiFi. Once WiFi is setup, you can unplug your Ethernet connection.

Headless Odroid Setup

Warning: This page may be outdated. For up to date documentation and for a more consistent experience across different single board computers, we recommend to instead setup *DietPi Setup*.

Odroids primarily run most Linux distributions but can also run Android as an operating system. SolarThing has not been tested with Android, so you should use the provided Ubuntu image.

More info on headless setup here: https://wiki.odroid.com/odroid-xu4/application_note/software/headless_setup

Download Image

Go to https://wiki.odroid.com/getting_started/os_installation_guide#tab__odroid-xu4 (and select something other than UX4 if you have something different) to download the image. I recommend the "Ubuntu Minimal" image.

If you use the wrong image, your Odroid will not boot.

Connecting Odroid to a network

Odroids do not have built in WiFi, but WiFi can be added using a USB adapter. If you want to configure WiFi, you will first either have to set it up using Ethernet, or will have to plug in a monitor

Connecting to Ethernet

All you have to do is hook your Odroid up to Ethernet and it will connect automatically. Now navigate to your router's homepage to find the IP of the Odroid device

Using SSH

The provided Ubuntu images have SSH enabled by default, so you simply have to ssh into your Odroid like so:

```
# If you did a minimal install
ssh root@YOUR IP

# If you did a full install, you can also do this instead
ssh odroid@YOUR IP
```

Note: The minimal install does not create a user named `odroid` like the full install does. So for the minimal install you will have to use the `root` user initially.

The password is `odroid` in both cases.

Configure WiFi

If you have a WiFi adapter, you can easily configure the WiFi though the command line. Configuring the WiFi network on an Odroid running Ubuntu is just like configuring WiFi on any Ubuntu through the command line.

You can follow instructions here: https://wiki.odroid.com/odroid_go_advance/application_note/sdio_wifi#configuring_wifi_station_mode_2_-_using_command_line.

The two main commands you will be running are here:

```
nmcli dev wifi list
nmcli dev wifi con 'SSID_1' password 'password_of_ssid1'
```

Note: If your WiFi network is not currently online, the command will fail. In that case, you can manually create a file in `/etc/NetworkManager/system-connections/` named `MYSSID.nmconnection`.

You can then check to make sure you are connected with this:

```
ip addr
# And
ping 8.8.8.8
```

Note: If you have the Odroid connected via Ethernet to the same network, it may not work at the same time as it could try to assign similar IP addresses to your Odroid. If it doesn't work, try disconnecting the Ethernet and going to your router's homepage to find the new IP address of your Odroid.

Once you are connected to WiFi, you can unplug your Ethernet cable (or monitor) you were using to configure it. You should have already gotten your new IP address above. You can now use the new IP address to SSH into your Odroid device.

Other Notes

- Hostname: `odroid`
- Password: `odroid` * Password for `root` user and password for `odroid` user.
- `odroid` user is only setup on full installs
- `cat /sys/devices/virtual/thermal/thermal_zone*/temp` to view temperature data

Config `databases` property of `base.json` (Version 2023.3.0 and before)

If you are using SolarThing version 2023.4.0 or later

Now that your database of choice is fully set up and we have a `<some database>.json` configuration file, let's add it to our `base.json`.

Start editing `base.json`. Right now, it should look something like:

```
{
  //...
  "databases": [ ],
```

(continues on next page)

(continued from previous page)

```
  //...
}
```

Let’s change it to look like this:

```
{
  //...
  "databases": [
    "config/<some database>.json"
  ],
  //...
}
```

Save the file. It is set up now!

Raspberry Pi CPU Temperature

Note: This documentation is deprecated in favor of *CPU Temperature*.

For all the programs that upload to databases, they support the ability to add the Raspberry Pi’s CPU temperature as a packet. You just have to add this json to your `base.json`:

```
{
  //...
  "request": [
    { "type": "rpi-cpu-temp" }
  ]
}
```

Restarts your application, and you should see that CPU Temperature packets are being uploaded.

It’s not working

Note: This only applies to versions 2023.2.1 and before. 2023.3.0 and above is not affected by permission issues for CPU temperature measurements.

First, if you are using the systemd service, make sure that you installed it correctly and did not try to install it yourself. If you are running SolarThing using `./run.sh`, you should instead do `sudo -u solarthing ./run.sh`. This is because the `solarthing` user should be set up to have the group `video`. You can make sure this is the case by running `groups solarthing`.

Security

Security is important on any system, even if it is not running SolarThing software. This page will help you harden your system and make it more secure.

Warning: This page is considered legacy because it may be outdated and is not an extensive enough guide to help make sure your device is completely secure. The security of your device is your own responsibility.

Secure your SSH server

Note: This section is not for securing the ssh-port-forward-* containers. It is for securing/hardening SSHD daemons running on any of your devices.

If you have your SSH server exposed directly or indirectly (by using ssh port forwarding), your server will eventually start to be hit with many login attempts from bots. You can secure your server by requiring a publickey to be used, rather than a password. Before you make these optional but recommended changes, you should create an SSH key and use `ssh-copyid` command to authorize yourself on your device. (See: https://www.ssh.com/academy/ssh/copy-id)

Typically one would just disable password authentication completely, but I prefer to only allow password authentication on local networks. Edit `/etc/ssh/sshd_config` and add this at the bottom:

```
PasswordAuthentication no
Match Address 10.0.0.0/8,172.16.0.0/12,192.168.0.0/16
  PasswordAuthentication yes
```

You can also install `fail2ban` on your system to automatically block spammers who attempt to login many times.

```
sudo apt-get install -y fail2ban
cd /etc/fail2ban
cp jail.conf jail.local
vi jail.local
systemctl restart fail2ban
```

Secure Your Server With Firewall

You can use a firewall to block (or allow specific IPs) from connecting to certain ports. This can be useful if you notice unknown IPs attempting to connect to your SSH port

Note: Using `ufw` or `/etc/hosts.deny` without additional configuration will not block connections to docker containers (that use bridge networks–the default when exposing ports) because of how docker modifies iptables.

```
sudo apt-get install -y ufw
sudo ufw default allow incoming

# These IPs are malicious. Feel free to create your own list, these are the ones I have␣
↪gathered.
sudo ufw deny from 91.212.166.22 # ssh bot with users: Admin, user, telnet, support
```

(continues on next page)

(continued from previous page)

```
sudo ufw deny from 91.240.118.172 # ssh bot with user: admin
sudo ufw deny from 200.29.117.156 # ssh bot with user: rkz
sudo ufw deny from 51.91.78.31 # ssh bot with user: teste
sudo ufw deny from 188.166.188.181 # ssh bot with user: ahmed
sudo ufw deny from 13.71.46.226 # ssh bot with user: rpms
sudo ufw deny from 51.250.5.16 # ssh bot with user: rebecca
sudo ufw deny from 200.70.56.204 # ssh bot with user: test
sudo ufw deny from 62.204.41.176 # ssh bot with user: admin
sudo ufw deny from 192.241.210.224 # ssh bot with user: hasmtpuser
sudo ufw deny from 189.216.40.170 # ssh bot with user: hadoop
sudo ufw deny from 134.209.69.41 # ssh bot with user: bbj
sudo ufw deny from 200.75.16.212 # ssh bot with user: tomecat4
sudo ufw deny from 157.230.53.66 # ssh bot with user: losif
sudo ufw deny from 2.234.152.80 # ssh bot with user: mass
sudo ufw deny from 82.196.113.78 # ssh bot with user: paulj
sudo ufw deny from 60.51.38.237 # ssh bot with user: admin
sudo ufw deny from 45.80.64.230 # ssh bot with user: cable
sudo ufw deny from 117.79.226.120 # ssh bot with user: user
sudo ufw deny from 154.68.39.6 # ssh bot with user: docm

# To remove a rule
sudo ufw delete deny from <ip address>

sudo ufw reload
sudo ufw enable
sudo ufw logging medium

# View log
sudo less /var/log/ufw.log
```

To make this apply to docker containers, view this repository: https://github.com/chaifeng/ufw-docker. Note that messing up the iptables configuration can mess up who can access your server.

A simpler method can be to edit `/etc/hosts.deny` if you don't want to install ufw. Note that this may not work with the ufw-docker iptables update to apply to docker containers.

```
ALL : 91.212.166.22
# You can add more from above
```

To learn more about `/etc/hosts.deny`, you can view a tutorial such as https://linuxconfig.org/hosts-deny-format-and-example-on-linux.