SHEPHERD

Kai Geissdoerfer

Jun 06, 2023

CONTENTS

To learn how *shepherd* enables research into the most challenging problems of coordinating battery-less sensor nodes, take a look at our paper. To get a basic understanding of what shepherd does, read the *Basics*. If you have the hardware on your desk and want to get started, read *Getting started*. To record/emulate data on a group of shepherd nodes, use the *shepherd-herd* command line utility.

CHAPTER

ONE

USER GUIDE

1.1 Basics

shepherd is a testbed for the battery-less Internet of Things. It allows to record harvesting conditions at multiple points in space over time. The recorded data can be replayed to attached wireless sensor nodes, examining their behaviour under the constraints of spatio-temporal energy availability.

For a detailed description see our Paper.

A *shepherd* instance consists of a group of spatially distributed *shepherd* nodes that are time-synchronized with each other. Each *shepherd* node consists of a BeagleBone, the *shepherd* cape and a particular choice of capelets according to the user requirements.

Shepherd works in two key modes: *Recording* and *Emulation*. Through various digitally controlled analog switches, we can reconfigure the hardware according to the selected mode from software.

1.1.1 Time-synchronization

Generally, shepherd can be used on a single node or without time-synchronization, for example, to measure the amount of energy that can be harvested in a particular scenario. The more interesting feature however is that it enables to explore harvesting conditions across time and space, which is a crucial step towards collaboration and coordination of battery-less nodes.

For tethered settings, we propose to use the Precision Time Protocol (PTP). It requires that all shepherd nodes are connected to a common Ethernet network with as little switching jitter as possible. The BeagleBone has hardware support for PTP and Linux provides the necessary tools.

In mobile or long-range scenarios, it might not be feasible to connect the nodes with Ethernet. Instead, you can use the timing signal from a GPS receiver. We have designed a GPS capelet (hardware/capelets/gps) to easily connect a GPS receiver and provide all necessary software in our repository.

1.1.2 Recording

For recording, shepherd nodes are equipped with a harvesting transducer, e.g., a solar panel or piezo-electric harvester. This transducer is connected to the input of the DC/DC boost converter on the shepherd cape. The capacitor-buffered output of the DC/DC converter is connected to a *dummy load* that discharges the capacitor as soon as a threshold voltage is reached. This is important, because the DC/DC converter can only charge a capacitor up to a maximum voltage before it switches off. Shepherd measures the voltage and current on the input of the DC/DC converter.

A group of shepherd nodes can be deployed to the environment of interest to measure the energy that can be harvested at each node. The time-synchronization between the shepherd nodes allows to gather the readings from multiple nodes with respect to a common time-line. The data thus represents the spatio-temporal energy availability.

An important aspect of recording is the operating point of the DC/DC converter. A harvesting transducer is characterized by its IV-curve, determining the magnitude of current at a specific load voltage. The DC/DC boost converter can (within limits) regulate the input to a desired voltage, by adapting the current drawn from the transducer. Thus, we have to inform the converter at which voltage we want to operate the transducer. There are two options:

Maximum Power Point Tracking (MPPT)

The BQ25505 boost converter on the shepherd cape implements a simple MPPT algorithm in hardware. In short, it disables the input every 16 seconds, samples the open circuit voltage and regulates its input voltage to a hardware-configurable fraction of that open circuit voltage. Shepherd accounts for this by exposing the sampling input of the DC/DC converter to the harvesting capelet, such that the user can configure the voltage divider according to the specific type of harvester. The low sampling rate of the algorithm makes it suitable only for slowly-changing harvesting conditions like solar energy.

User provided reference voltage

A user can tell shepherd the voltage with which it should operate the harvester during recording. Currently, the software only supports setting a constant voltage at the beginning of the recording.

1.1.3 Emulation

In emulation mode, spatio-temporal current and voltage data is replayed to a group of sensor nodes. Each shepherd node hosts a DAC controlled current source that can precisely set the current into the DC/DC converter. The DC/DC converter has a reference voltage input that we use to set the voltage on its input. With these two mechanisms, we are able to force the DC/DC converter into an arbitrary operating point. Relying on time-synchronization, shepherd can thus faithfully reproduce previously recorded (or model-based) spatio-temporal energy conditions.

Instead of the dummy load, the capacitor-bufferd output of the DC/DC converter is connected to the target sensor node. Thus, the sensor node is operated under the constraints of emulated energy availability.

Like other testbeds, shepherd records the voltage on the capacitor and the current consumed by the sensor node during emulation. Furthermore, four GPIO lines and one bi-directional UART are level-translated between shepherd and the attached sensor node, allowing to trace

1.1.4 Remote programming/debugging

For convenient debugging and development, shepherd implements a fully functional Serial-Wire-Debug (SWD) debugger. SWD is supported by most recent ARM Cortex-M corand allows flashing images and debugging the execution of code. Older platforms typically provide a serial boot-loader, which can be used to flash images over the pre-mentioned UART connection.

1.2 Getting started

This section describes how to setup an instance of shepherd in a tethered setup.

1.2.1 Prerequisites

To setup an instance of shepherd, you'll need to assemble a number shepherd nodes.

For each shepherd node, you'll need:

- BeagleBone (Green/Black)
- shepherd cape
- storage capelet
  - for recording: harvesting capelet and transducer, e.g. solar capelt and solar cell
  - for emulation: target capelet

In addition, you'll need at least one SD-card with at least 4GB capacity.

For the cape and capelets take a look at the hardware design files The capelets can easily be soldered by hand. The shepherd cape has a large number of small components and we suggest to send it to a PCB fab for assembly.

Place a capacitor of desired capacity on your storage capelets. The reference layout offers a choice of three footprints allowing to flexibly choose a suitable capacitor and package.

If you don't have the necessary resource or need assistance with getting the hardware manufactured, get in touch with the developers.

To connect the shepherd nodes to each other for control, data collection and time-synchronization, you need to setup an Ethernet network. The network should be as flat as possible, i.e. have a minimum number of switches. By default, the BeagleBone Ubuntu image is configured to request an IP address by DHCP. Therefore your network should have a DHCP server.

1.2.2 Hardware setup

Stack the cape on top of the BeagleBone. Make sure to close the jumpers JP1 and JP2 on the cape.

Mount the storage capelet on headers P3/P4 on the cape.

Stack the harvesting capelet on top of the shepherd cape. The two 11x2 headers of the capelet plug into the lower part (P8_1 - P8_22 and P9_1 - P9_22) of the 23x2 headers of the shepherd cape. Pay attention to header P1 for the correct orientation.

Stack the target capelet on top of the shepherd cape. The two 11x2 headers of the capelet plug into the upper part (P8_25 - P8_46 and P9_25 - P9_46) of the 23x2 headers of the shepherd cape. Pay attention to header P2 for the correct orientation.

Provide all BeagleBones with power through the micro USB ports and connect their Ethernet ports to an Ethernet switch. Using a PoE switch and corresponding micro USB power splitters can greatly reduce the cabling requirements.

The DHCP server and your machine (for installation/control) must be connected to the same network.

1.2.3 Installation

Prepare the SD-cards. If you plan to install the OS and shepherd software on the onboard EMMC flash, you can prepare one SD card and sequentially flash the nodes. If you plan to install the OS and shepherd software on SD card, you have to prepare one SD card for every shepherd node. Depending on your choice, follow the official instructions for **BeagleBone**. Shepherd has been tested on Ubuntu 18.04 LTS, but might work with other Debian based distributions.

After installing the OS on the BeagleBones and booting them, find out their IP addresses. If you know the subnet, you can use nmap from your machine, for example:

```
nmap 192.168.178.0/24
```

Clone the shepherd repository to your machine:

```
git clone https://github.com/geissdoerfer/shepherd.git
```

Add an inventory file in the *inventory* folder in the repository, assigning hostnames to the IP addresses of the shepherd nodes. Just start by editing the provided *inventory/herd.yml* example. Pick a username that you want to use to login to the nodes and assign as *ansible_user* variable.

```
sheep:
  hosts:
    sheep0:
      ansible_host: 192.168.1.100
    sheep1:
      ansible_host: 192.168.1.101
    sheep2:
      ansible_host: 192.168.1.102
  vars:
    ansible_user: jane
```

We'll use Ansible to roll out a basic configuration to the nodes. This includes setting the hostname, adding the user, allowing password-less ssh access and sudo without password. Make sure that you have *Python >=3.6*, *pip* and *sshpass* installed on your machine. Install *Ansible* with:

```
pip3 install ansible
```

Now run the *bootstrap* Ansible playbook using the previously prepared inventory file:

```
ansible-playbook deploy/bootstrap.yml
```

To streamline the installation and upgrading process, the shepherd software is packaged and distributed as debian packages. Installing is as easy as adding the shepherd repository to the aptitude sources and installing the shepherd metapackage. The *install* playbook allows to easily automate this process on a group of nodes.

```
ansible-playbook deploy/install.yml
```

To install and configure PTP for time-synchronizing a bunch of shepherd-nodes, you can set the *ptp* variable on the command line, alternatively you get asked on script-start:

```
ansible-playbook deploy/install.yml -e "ptp=True"
```

On success, the nodes will reboot and should be ready for use, for example, using the *shepherd-herd* command line utility.

Further playbooks:

- `setup_linux_configuration.yml` will handle updates, some configuration, remove clutter, improve ram-usage and boot-duration
- `setup_linux_performance.yml` handles additional speed-improving changes
- `setup_linux_security.yml` will close system so that nodes can be distributed safely in open spaces (basic steps against getting into system)
- `fetch-hostkeys.yml` will copy keys from nodes, handy for reflashing image, while keeping keys
- `setup-dev-nfs.yml` establish a local network file system `/opt/shepherd-dev` for the nodes to access
- `setup-ext-storage.yml` will format and automount sd-card to ''/var/shepherd/recordings''
- `deploy.yml` offers faster and easier way to test updates in shepherd-code-base

- `dev_rebuild_pru.yml` hot-swaps pru-firmware (& kernel-module & py-package) by compiling and flashing without restart

1.3 Hardware

Shepherd is a HW/SW solution and this section describes the key interfaces of the shepherd HW that are relevant to users.

Each shepherd node consists of 5 key components:

- The BeagleBone is a Single Board Computer, handling synchronization, data storage and hardware control
- The shepherd cape hosts the DC/DC converter, amplifiers and ADC, DAC and corresponding emulation circuitry and all other fixed hardware parts
- The harvesting capelet connects a harvesting transducer (e.g. solar panel) to the input of the DC/DC converter on the shepherd cape
- The target capelet connects a sensor node (e.g. a microcontroller with radio) to the capacity buffered output of the DC/DC converter on the shepherd cape
- The storage capelet hosts the capacitor used to temporarily store harvested energy during recording and emulation

1.3.1 Harvesting capelet

The harvesting capelet connects any type of harvester, e.g., a solar panel or piezo-electric element to the shepherd hardware. By keeping this part of the hardware separate from the main shepherd cape, we allow to easily connect various types of harvesters, without having to change the complex and expensive main cape.

The harvesting capelet connects to the shepherd cape via three headers. Two headers connect directly to the lower half of the pins of the P8 and P9 headers on the BeagleBone. The third header is the custom, 4-pin connection between the capelet and the cape.

There are two reference harvesting capelets, one for solar harvesting (hardware/capelets/solar) and one for kinetic harvesting (hardware/capelets/kinetic).

Table 1: Header P1 pinout

Pin number	Description
1	Ground
2	Ground
3	Input of DC/DC converter
4	Input for setting MPP via voltage divider

1.3.2 Target capelet

Shepherd provides a generic interface to connect any type of sensor node that can be supplied by emulated energy traces, while tracing its state with GPIO and UART.

The target capelet connects to the shepherd cape via three headers. Two headers connect directly to the upper half of the pins of the P8 and P9 headers on the BeagleBone. The third header is the custom, 16-pin connection between the capelet and the cape.

Table 2: Header P2 pinout

Pin number	Description
1	Capacity buffered output of DC/DC converter
2	Ground
3	Battery voltage in operating range indicator
4	High side of shunt resistor for harvesting current measurement
5	+3.3V from BeagleBone LDO, use for any additional infrastructure circuitry
6	Low side of shunt resistor for harvesting current measurement
7	Ground
8	GPIO2 - voltage translated digital signal from target to BeagleBone for tracing
9	UART TX - voltage translated UART signal from target to Beaglebone UART1
10	GPIO0 - voltage translated digital signal from target to BeagleBone for GPIO tracing
11	UART RX - voltage translated UART signal from Beaglebone UART1 to target
12	GPIO3 - voltage translated digital signal from target to BeagleBone for tracing
13	SWDCLK - Clock line for remote programming via ARM Serial-Wire-Debug (SWD)
14	GPIO1 - voltage translated digital signal from target to BeagleBone for tracing
15	SWDIO - Data line for remote programming via ARM Serial-Wire-Debug (SWD)
16	Ground

Pin 5 provides regulated 3.3V to supply any additional circuitry independent of the actual harvesting power supply. This could be used to power, for example, a debugging LED that should not interfere with the energy emulation/measurement. Pins 4 and 6 provide access to the voltage across the shunt resistor between the harvester/emulator and the DC/DC converter. This allows to measure harvesting voltage and current online on the sensor node, which might be interesting for certain directions of research. Pins 9 and 11 expose the level-translated signals to UART1 on BeagleBone. They can be used to trace UART messages from the target sensor node. Pins 8, 10, 12, 14 are (level-translated) connected to the low-latency GPIOs that are sampled by shepherd software for GPIO tracing. Pins 13 and 15 (level-translated) are used to program/debug a connected sensor node with SWD.

1.3.3 Storage capelet

Although less relevant for recording, the choice of capacitor crucially defines system behaviour during emulation. Therefore, shepherd allows to easily swap the capacitor to experiment with different technologies/capacities by changing storage capelets.

The storage capelet directly connects the two ends of a mounted capacitor to the corresponding traces on the shepherd cape via two custom, 4-pin headers.

Table 3: Header P3 pinout

Pin number	Description
1	+3.3V from BeagleBone LDO, use for any additional circuitry
2	High side of capacitor
3	Ground
4	Ground

1.4 Command-line tools

Shepherd offers two command line utilities:

Shepherd-herd is the command line utility for remotely controlling a group of shepherd nodes. This is the key user interface to shepherd. The pure-python package is installed on the user's local machine and sends commands to the

shepherd nodes over *ssh*.

Shepherd-sheep is the command line utility for locally controlling a single shepherd node. Depending on your use-case you may not even need to directly interact with it!

1.4.1 shepherd-herd

shepherd-herd

```
shepherd-herd [OPTIONS] COMMAND [ARGS]...
```

Options

-i, --inventory <inventory>
List of target hosts as comma-separated string or path to ansible-style yaml file

-l, --limit <limit>
Comma-separated list of hosts to limit execution to

-u, --user <user>
User name for login to nodes

-k, --key-filename <key_filename>
Path to private ssh key file

-v, --verbose

emulate

Emulates IV data read from INPUT hdf5 file

```
shepherd-herd emulate [OPTIONS] INPUT_PATH
```

Options

-o, --output_path <output_path>
Dir or file path for resulting hdf5 file with load recordings

-d, --duration <duration>
Duration of recording in seconds

-f, --force_overwrite
Overwrite existing file

--no-calib
Use default calibration values

--load <load>
Choose artificial or sensor node load

Options artificial|node

-c, --ldo-voltage <ldo_voltage>
Pre-charge capacitor before starting recording

--config <config>
Read configuration from FILE.

--start, --no-start
Start shepherd after uploading config

Arguments

INPUT_PATH
Required argument

poweroff

Power off shepherd nodes

```
shepherd-herd poweroff [OPTIONS]
```

Options

-r, --restart
Reboot

record

Records IV data

```
shepherd-herd record [OPTIONS]
```

Options

-o, --output_path <output_path>
Dir or file path for resulting hdf5 file

--mode <mode>
Record 'harvesting' or 'load' data

Options harvesting|load

-d, --duration <duration>
Duration of recording in seconds

-f, --force_overwrite
Overwrite existing file

--no-calib
Use default calibration values

--harvesting-voltage <harvesting_voltage>
Set fixed reference voltage for harvesting

--load <load>
Choose artificial or sensor node load

Options artificial|node

-c, --ldo-voltage <ldo_voltage>
Sets voltage of variable LDO

--ldo-mode <ldo_mode>
Select if LDO should just pre-charge capacitor or run continuously

Options pre-charge|continuous

--start, --no-start
Start shepherd after uploading config

retrieve

Retrieves remote hdf file FILENAME and stores in in OUTDIR

```
shepherd-herd retrieve [OPTIONS] FILENAME OUTDIR
```

Options

-r, --rename

-d, --delete
Delete the file from the remote filesystem after retrieval

-s, --stop
Stop the on-going recording/emulation process before retrieving the data

Arguments

FILENAME
Required argument

OUTDIR
Required argument

run

Run COMMAND on the shell

```
shepherd-herd run [OPTIONS] COMMAND
```

Options

-s, --sudo
Run command with sudo

Arguments

COMMAND
Required argument

stop

Stops any recording/emulation

```
shepherd-herd stop [OPTIONS]
```

target

Remote programming/debugging of the target sensor node

```
shepherd-herd target [OPTIONS] COMMAND [ARGS]...
```

Options

-p, --port <port>
Port on which OpenOCD should listen for telnet

--on, --off
Enable/disable power and debug access to the target

erase

Erases the target

```
shepherd-herd target erase [OPTIONS]
```

flash

Flashes the binary IMAGE file to the target

```
shepherd-herd target flash [OPTIONS] IMAGE
```

Arguments

IMAGE
Required argument

halt

Halts the target

```
shepherd-herd target halt [OPTIONS]
```

reset

Resets the target

```
shepherd-herd target reset [OPTIONS]
```

Examples

In the following we assume that you have an ansible style, YAML-formatted inventory file named *hosts* in your current working diretory. Refer to the example *hosts* file in the root directory of the shepherd repository. To start recording harvesting data using the hardware MPPT algorithm and store it under the default path, overwriting existing data and precharging the capacitor before starting the recording:

```
shepherd-herd -i hosts record -f --init-charge
```

To stop recording on a subset of nodes (sheep1 and sheep3 in this example):

```
shepherd-herd -i hosts -l sheep1,sheep3 stop
```

Another example for recording, this time with a fixed harvesting voltage of 600mV, limited recording duration of 30s and storing the result under */var/shepherd/recordings/rec_v_fixed.h5*. Also, instead of using the inventory file option, we specify hosts and ssh user on the command line:

```
shepherd-herd -i sheep0,sheep1, -u jane record -d 30 --harvesting-voltage 0.6 -o rec_
↪v_fixed.h5
```

To retrieve the recordings from the shepherd nodes and store them locally on your machine under the *recordings/* directory:

```
shepherd-herd -i hosts retrieve -d rec_v_fixed.h5 recordings/
```

Before turning to emulation, here's how to flash a firmware image to the attached sensor nodes. To flash the image *firmware.bin* that is stored on the local machine:

```
shepherd-herd -i hosts target flash firmware.bin
```

To reset the CPU on the sensor nodes:

```
shepherd-herd -i hosts target reset
```

Finally, to replay previously recorded data from the file *rec.h5* in the shepherd node's file system and store the recorded IV data from the load as *load.h5*:

```
shepherd-herd -i hosts emulate -o load.h5 rec.h5
```

1.4.2 shepherd-sheep

Examples

Coming soon…

1.5 Calibration

Data recorded with shepherd are stored in the database as raw binary values, read from the ADC. In order to make sense of that data, we need to map the binary values to physical voltage and current values. Similarly, for emulation, we need to map phyiscal voltage and current values to binary values that can be fed to the DAC.

This mapping is assumed to be linear. For example, to convert a binary value v_{binary} to its physical equivalent $v_{physical}$:

$$v_{physical} = v_{binary} \cdot gain + offset$$

There are three "channels" and two "variables": harvesting, load and emulation, each with voltage and current. Thus the complete set of calibration values consists of 12 values. When storing recorded data, these values are stored together with the binary data.

The *gain* and *offset* values can be derived from nominal hardware configuration. However, due to component tolerances, offset voltages and other effects, the real mapping can significantly differ from nominal values.

Therefore, we recommend to *calibrate* shepherd by comparing a number of sample points recorded with shepherd to known refernce values measured with accurate lab equipment. From the tuples of reference values and measurements, you can estimate the *gain* and *offset* by ordinary least squares linear regression. These values can be stored in the EEPROM on the shepherd cape and, at the beginning of a recording extracted and stored in the database file.

Support for an automated calibration procedure with *shepherd-herd* will be added soon.

1.6 Data format

Data is stored in the popular Hierarchical Data Format.

This section describes the structure of data recorded with shepherd:

```
.
|-- attributes
|   `-- mode
|-- data
|   |-- time
|   |-- current
|   |   `-- attributes
|   |       |-- gain
|   |       `-- offset
|   `-- voltage
|       `-- attributes
|           |-- gain
|           `-- offset
`-- gpio
    |-- time
    `-- values
```

The "mode" attribute allows to distinguish between "load" data and "harvesting" data.

The data group contains the actual IV data. Time stores the time for each sample in nanoseconds. Current and voltage store the binary data as acquired from the ADC. They can be converted to their physical equivalent using the corresponding gain and offset attributes. See also *Calibration*.

The gpio group stores the timestamp when a GPIO edge was detected and the corresponding bit mask in values. For example, assume that all are were low at the beginning of the recording. At time T, GPIO pin 2 goes high. The value 0x04 will be stored together with the timestamp T in nanoseconds.

There are numerous tools to work with HDF5 and library bindings for all popular programming languages.

Here's an example how to plot data recorded with shepherd using Python:

```
import h5py
import matplotlib.pyplot as plt

f, axarr = plt.subplots(1, 2)
with h5py.File("rec.h5", "r") as hf:
    # convert time to seconds
    t = hf["data"]["time"][:].astype(float) / 1e9
    for i, var in enumerate(["voltage", "current"]):
        gain = hf["data"]["voltage"].attrs["gain"]
        offset = hf["data"]["voltage"].attrs["offset"]
        values = hf["data"][var][:] * gain + offset
        axarr[i].plot(t, values)

plt.show()
```

1.7 API

The shepherd API offers high level access to shepherd's functionality and forms the base for the two command line utilities. Note that the API only convers local functionality on a single shepherd node. Use the *shepherd-herd* command line utility to orchestrate a group of shepherd nodes remotely.

1.7.1 Recorder

The recorder is used to configure all relevant hardware and software and to sample and extract data from the analog frontend.

class shepherd.**Recorder**(*mode: str = 'harvesting', load: str = 'artificial', harvesting_voltage: float = None, ldo_voltage: float = None, ldo_mode: str = 'pre-charge'*)

API for recording data with shepherd.

Provides an easy to use, high-level interface for recording data with shepherd. Configures all hardware and initializes the communication with kernel module and PRUs.

Parameters

- **mode** (*str*) – Should be either 'harvesting' to record harvesting data or 'load' to record target consumption data.
- **load** (*str*) – Selects, which load should be used for recording. Should be one of 'artificial' or 'node'.
- **harvesting_voltage** (*float*) – Fixed reference voltage for boost converter input.
- **ldo_voltage** (*float*) – Pre-charge capacitor to this voltage before starting recording.
- **ldo_mode** (*str*) – Selects if LDO should just pre-charge capacitor or run continuously.

get_buffer(*timeout: float = 1.0*) → NoReturn

Reads a data buffer from shared memory.

Polls the RPMSG channel for a message from PRU0 and, if the message points to a filled buffer in memory, returns the data in the corresponding memory location as DataBuffer.

Parameters **timeout** (*float*) – Time in seconds that should be waited for an incoming RPMSG

Returns Index and content of corresponding data buffer

Raises **`TimeoutException`** – If no message is received on RPMSG within specified timeout

`return_buffer`(*index: int*)
Returns a buffer to the PRU

After reading the content of a buffer and potentially filling it with emulation data, we have to release the buffer to the PRU to avoid it running out of buffers.

Parameters **`index`** (*`int`*) – Index of the buffer. 0 <= index < n_buffers

`set_harvester`(*state: bool*) → NoReturn
Enables or disables connection to the harvester.

The harvester is connected to the main power path of the shepherd cape through a MOSFET relay that allows to make or break that connection. This way, we can completely disable the harvester during emulation.

Parameters **`state`** (*`bool`*) – True for enabling harvester, False for disabling

`set_harvesting_voltage`(*voltage: float*) → NoReturn
Sets the reference voltage for the boost converter

In some cases, it is necessary to fix the harvesting voltage, instead of relying on the built-in MPPT algorithm of the BQ25505. This function allows setting the set point by writing the desired value to the corresponding DAC. Note that the setting only takes effect, if MPPT is disabled (see set_mppt())

Parameters **`voltage`** (*`float`*) – Desired harvesting voltage in volt

Raises **`ValueError`** – If requested voltage is out of range

`set_ldo_voltage`(*voltage: float*) → NoReturn
Enables or disables the constant voltage regulator.

The shepherd cape has a linear regulator that is connected to the load power path through a diode. This allows to pre-charge the capacitor to a defined value or to supply a sensor node with a fixed voltage. This function allows to enable or disable the output of this regulator.

Parameters **`voltage`** (*`float`*) – Desired output voltage in volt. Providing 0 or False disables the LDO.

`set_load`(*load: str*) → NoReturn
Selects which load is connected to shepherd's output.

The output of the main power path can be connected either to the on-board 'artificial' load (for recording) or to an attached sensor node (for emulation). This function configures the corresponding hardware switch.

Parameters

- **`load`** (*`str`*) – The load to connect to the output of shepherd's output.
- **`of 'artificial' or 'node'.`** (*`One`*) –

Raises **`NotImplementedError`** – If load is not 'artificial' or 'node'

`set_lvl_conv`(*state: bool*) → NoReturn
Enables or disables the GPIO level converter.

The shepherd cape has a bi-directional logic level shifter (TI TXB0304) for translating UART and SWD signals between BeagleBone and target voltage levels. This function enables or disables the converter.

Parameters **`state`** (*`bool`*) – True for enabling converter, False for disabling

set_mppt (*state: bool*) → NoReturn
Enables or disables Maximum Power Point Tracking of BQ25505

TI's BQ25505 implements an MPPT algorithm, that dynamically adapts the harvesting voltage according to current conditions. Alternatively, it allows to provide a reference voltage to which it will regulate the harvesting voltage. This is necessary for emulation and can be used to fix harvesting voltage during recording as well.

Parameters **state** (*bool*) – True for enabling MPPT, False for disabling

start (*start_time: float = None, wait_blocking: bool = True*) → NoReturn
Starts sampling either now or at later point in time.

Parameters

- **start_time** (*int*) – Desired start time in unix time
- **wait_blocking** (*bool*) – If true, block until start has completed

static wait_for_start (*timeout: float*) → NoReturn
Waits until shepherd has started sampling.

Parameters **timeout** (*float*) – Time to wait in seconds

Usage:

```
# Configure converter for fixed 1.5V input regulation
with Recorder(harvesting_voltage=1.5) as recorder:
    recorder.start()

    for _ in range(100):
        idx, buf = recorder.get_buffer()
        recorder.release_buffer(idx)
```

1.7.2 Emulator

The emulator is used to replay previously recorded IV data to an attached sensor node.

class shepherd.**Emulator**(*initial_buffers: list = None, calibration_recording: shepherd.calibration.CalibrationData = None, calibration_emulation: shepherd.calibration.CalibrationData = None, load: str = 'node', ldo_voltage: float = 0.0*)
API for emulating data with shepherd.

Provides an easy to use, high-level interface for emulating data with shepherd. Configures all hardware and initializes the communication with kernel module and PRUs.

Parameters

- **calibration_recording** (*CalibrationData*) – Shepherd calibration data belonging to the IV data that is being emulated
- **calibration_emulation** (*CalibrationData*) – Shepherd calibration data belonging to the cape used for emulation
- **load** (*str*) – Selects, which load should be used for recording. Should be one of 'artificial' or 'node'.
- **ldo_voltage** (*float*) – Pre-charge the capacitor to this voltage before starting recording.

get_buffer(*timeout: float = 1.0*) → NoReturn
Reads a data buffer from shared memory.

Polls the RPMSG channel for a message from PRU0 and, if the message points to a filled buffer in memory, returns the data in the corresponding memory location as DataBuffer.

Parameters **timeout** (*float*) – Time in seconds that should be waited for an incoming RPMSG

Returns Index and content of corresponding data buffer

Raises **TimeoutException** – If no message is received on RPMSG within specified timeout

set_harvester(*state: bool*) → NoReturn
Enables or disables connection to the harvester.

The harvester is connected to the main power path of the shepherd cape through a MOSFET relay that allows to make or break that connection. This way, we can completely disable the harvester during emulation.

Parameters **state** (*bool*) – True for enabling harvester, False for disabling

set_harvesting_voltage(*voltage: float*) → NoReturn
Sets the reference voltage for the boost converter

In some cases, it is necessary to fix the harvesting voltage, instead of relying on the built-in MPPT algorithm of the BQ25505. This function allows setting the set point by writing the desired value to the corresponding DAC. Note that the setting only takes effect, if MPPT is disabled (see set_mppt())

Parameters **voltage** (*float*) – Desired harvesting voltage in volt

Raises **ValueError** – If requested voltage is out of range

set_ldo_voltage(*voltage: float*) → NoReturn
Enables or disables the constant voltage regulator.

The shepherd cape has a linear regulator that is connected to the load power path through a diode. This allows to pre-charge the capacitor to a defined value or to supply a sensor node with a fixed voltage. This function allows to enable or disable the output of this regulator.

Parameters **voltage** (*float*) – Desired output voltage in volt. Providing 0 or False disables the LDO.

set_load(*load: str*) → NoReturn
Selects which load is connected to shepherd's output.

The output of the main power path can be connected either to the on-board 'artificial' load (for recording) or to an attached sensor node (for emulation). This function configures the corresponding hardware switch.

Parameters

- **load** (*str*) – The load to connect to the output of shepherd's output.
- **of 'artificial' or 'node'.** (*One*) –

Raises **NotImplementedError** – If load is not 'artificial' or 'node'

set_lvl_conv(*state: bool*) → NoReturn
Enables or disables the GPIO level converter.

The shepherd cape has a bi-directional logic level shifter (TI TXB0304) for translating UART and SWD signals between BeagleBone and target voltage levels. This function enables or disables the converter.

Parameters **state** (*bool*) – True for enabling converter, False for disabling

set_mppt (*state: bool*) → NoReturn
Enables or disables Maximum Power Point Tracking of BQ25505

TI's BQ25505 implements an MPPT algorithm, that dynamically adapts the harvesting voltage according to current conditions. Alternatively, it allows to provide a reference voltage to which it will regulate the harvesting voltage. This is necessary for emulation and can be used to fix harvesting voltage during recording as well.

Parameters **state** (*bool*) – True for enabling MPPT, False for disabling

start (*start_time: float = None, wait_blocking: bool = True*) → NoReturn
Starts sampling either now or at later point in time.

Parameters

- **start_time** (*int*) – Desired start time in unix time
- **wait_blocking** (*bool*) – If true, block until start has completed

static wait_for_start (*timeout: float*) → NoReturn
Waits until shepherd has started sampling.

Parameters **timeout** (*float*) – Time to wait in seconds

Usage:

```
# We'll read existing data using a LogReader
lr = LogReader("mylog.h5")
with ExitStack() as stack:
    stack.enter_context(lr)
    emu = Emulator(
        calibration_recording=lr.get_calibration_data(),
        initial_buffers=lr.read_buffers(end=64),
    )
    stack.enter_context(emu)
    emu.start()

    for hrvst_buf in lr.read_buffers(start=64):
        idx, _ = emu.get_buffer()
        emu.put_buffer(idx, hrvst_buf)
```

1.7.3 LogWriter

The *LogWriter* is used to store IV data sampled with shepherd to an hdf5 file.

class shepherd.**LogWriter** (*store_path: pathlib.Path, calibration_data: shepherd.calibration.CalibrationData, mode: str = 'harvesting', force_overwrite: bool = False, samples_per_buffer: int = 10000, buffer_period_ns: int = 100000000*)
Stores data coming from PRU's in HDF5 format

Parameters

- **store_path** (*str*) – Name of the HDF5 file that data will be written to
- **calibration_data** (*CalibrationData*) – Data is written as raw ADC values. We need calibration data in order to convert to physical units later.
- **mode** (*str*) – Indicates if this is data from recording or emulation
- **force_overwrite** (*bool*) – Overwrite existing file with the same name

- **samples_per_buffer** (*int*) – Number of samples contained in a single shepherd buffer
- **buffer_period_ns** (*int*) – Duration of a single shepherd buffer in nanoseconds

write_buffer(*buffer: shepherd.shepherd_io.DataBuffer*) → NoReturn
Writes data from buffer to file.

Parameters **buffer** (*DataBuffer*) – Buffer containing IV data

write_exception(*exception: shepherd.datalog.ExceptionRecord*) → NoReturn
Writes an exception to the hdf5 file.

Parameters **exception** (*ExceptionRecord*) – The exception to be logged

Usage:

```
with LogWriter("mylog.h5") as log_writer, Recorder() as recorder:
    recorder.start()
    for _ in range(100):
        idx, buf = recorder.get_buffer()
        log_writer.write_buffer(buf)
        recorder.release_buffer(idx)
```

1.7.4 LogReader

The *LogReader* is used to read previously recorded data from an hdf5 file buffer by buffer. It can be used with the Emulator to replay recorded data to an attached sensor node.

class shepherd.**LogReader**(*store_path: pathlib.Path, samples_per_buffer: int = 10000*)
Sequentially Reads data from HDF5 file.

Parameters

- **store_path** (*Path*) – Path of hdf5 file containing IV data
- **samples_per_buffer** (*int*) – Number of IV samples per buffer

get_calibration_data() → shepherd.calibration.CalibrationData
Reads calibration data from hdf5 file.

Returns Calibration data as CalibrationData object

read_buffers(*start: int = 0, end: int = None*)
Reads the specified range of buffers from the hdf5 file.

Parameters

- **start** (*int*) – Index of first buffer to be read
- **end** (*int*) – Index of last buffer to be read

Yields Buffers between start and end

Usage:

```
with LogReader("mylog.h5") as log_reader:
    for buf in log_reader.read_buffers(end=1000):
        print(len(buf))
```

1.8 Performance Specification

The *Performance specification* table summarizes shepherd's key performance metrics. Refer to our paper for a detailed description of how these values were obtained. Some of the values were measured with an early prototype and may not accurately reflect the current hardware revision.

Table 4: Performance specification

Range	Harvesting voltage	100 mV - 4.8 V
	Harvesting current	0 mA - 25 mA
	Load voltage	0 V - 4.8 V
	Load current	0 mA - 25 mA
	Emulation voltage	100 mV - 4 V
	Emulation current	0 mA - 25 mA
24h DC Accuracy	Harvesting voltage	19.53 uV +/- 0.01 %
	Harvesting current	190 nA +/- 0.07 %
	Load voltage	19.53 uV +/- 0.01 %
	Load current	190 nA +/- 0.01 %
	Emulation voltage	76.3 uV +/- 0.012 %
	Emulation current	381.4 uA +/- 0.025 %
Bandwidth	All recording channels	15 kHz
Risetime	Emulation voltage	64 ms
	Emulation current	19.2 us
Max. Burden voltage	Harvesting recorder	50.4 mV
	Load recorder	76.1 mV
GPIO sampling speed		580 kHz - 5 MHz
Power consumption		345 mA
Max. Synchronization error		< 1.0 us

CHAPTER

TWO

DEVELOPER GUIDE

2.1 Contributing

This section helps developers getting started with contributing to *shepherd*.

2.1.1 Codestyle

Please stick to the C and Python codestyle guidelines provided with the source code.

All Python code is supposed to be formatted using Black, limiting the maximum line width to 80 characters. This is defined in the *pyproject.toml* in the repository's root directory.

C code shall be formatted according to the Linux kernel C codesytle guide. We provide the corresponding *clang-format* config as *.clang-format* in the repository's root directory.

Many IDEs/editors allow to automatically format code using the corresponding formatter and codestyle. Please make use of this feature or otherwise integrate automatic codestyle formatting into your workflow.

2.1.2 Development setup

While some parts of the *shepherd* software stack can be developed hardware independent, in most cases you will need to develop/test code on the actual target hardware.

We found the following setup convenient: Have the code on your laptop/workstation and use your editor/IDE to develop code. Have a BeagleBone (potentially with *shepherd* hardware) connected to the same network as your workstation. Prepare the BeagleBone by running the *bootstrap.yml* ansible playbook and additionally applying the *deploy/dev-host* ansible role.

You can now either use the ansible *deploy/sheep* role to push the changed code to the target and build and install it there. Running the role takes significant time though as all components (kernel module, firmware and python package) are built.

Alternatively, you can mirror your working copy of the *shepherd* code to the BeagleBone using a network file system. We provide a playbook (*deploy/setup-dev-nfs.yml*) to conveniently configure an *NFS* share from your local machine to the BeagleBone. After mounting the share on the BeagleBone, you can compile and install the corresponding software component remotely over ssh on the BeagleBone while editing the code locally on your machine.

2.1.3 Building debian packages

shepherd software is packaged and distributed as debian packages. Building these packages requires a large number of libraries and tools to be installed. This motivates the use of docker for creating an isolated and well defined build environment.

The procedure for building the image, creating a container, copying the code into the container, building the packages and copying the artifacts back to the host are found in the *.travis.yml* in the repository's root directory.

2.1.4 Building the docs

Make sure you have the python requirements installed:

```
pipenv install
```

Activate the *pipenv* environment:

```
pipenv shell
```

Change into the docs directory and build the html documentation

```
cd docs
make html
```

The build is found at *docs/_build/html*. You can view it by starting a simple http server:

```
cd _build/html
python -m http.server
```

Now navigate your browser to *localhost:8000* to view the documentation.

2.1.5 Tests

There is an initial testing framework that covers a large portion of the python code. You should always make sure the tests are passing before committing your code.

To run the python tests, have a copy of the source code on a BeagleBone. Change into the *software/python-package* directory and run:

```
sudo python3 setup.py test --addopts "-vv"
```

2.1.6 Releasing

Once you have a clean stable version of code, you should decide if your release is a patch, minor or major (see Semantic Versioning). Make sure you're on the master branch and have a clean working direcory. Use *bump2version* to update the version number across the repository:

```
bump2version --tag patch
```

Finally, push the changes and the tag to trigger the CI pipeline to build and deploy new debian packages to the server:

```
git push origin master --tags
```

2.2 Data handling

2.2.1 Data Acquisition

Data is sampled/replayed through the ADC (TI ADS8694) and DAC (TI DAC8562T). Both devices are interfaced over a custom, SPI-compatible protocol. For a detailed description of the protocol and timing requirements, refer to the corresponding datasheets. The protocol is bit-banged using the low-latency GPIOs connected to PRU1. The transfer routines are implemented in assembly.

2.2.2 PRU to host

Data is sampled and transferred between PRUs and user space in buffers, i.e. blocks of SAMPLES_PER_BUFFER samples. These buffers correspond to sections of a continuous area of memory in DDR RAM to which both PRU and user space application have access. This memory is provisined through remoteproc, a Linux framework for managing resources in an AMP (asymmetric multicore processing) system. The PRU firmware contains a so-called resource table that allows to specify required resources. We request a carve-out memory area, which is a continuous, non-cached area in physical memory. On booting the PRU, the remoteproc driver reads the request, allocates the memory and writes the starting address of the allocated memory area to the resource table, which is readable by the PRU during run-time. The PRU exposes this memory location through shared RAM, which is accessible through the sysfs interface provided by the kernel module. Knowing physical address and size, the user space application can map that memory after which it has direct read/write access. The total memory areas is divided into N_BUFFERS distinct buffers.

In the following we describe the data transfer process for emulation. Emulation is the most general case because harvesting data has to be transferred from database to the analog frontend, while simultaneously consumption data (target voltage and current) has to be transferred from the analog frontend (ADC) to the database.

The userspace application writes the first block of harvesting data into one of the buffers, e.g. buffer index i. After the data is written, it sends an RPMSG to PRU1, indicating the message type (MSG_DEP_BUF_FROM_HOST) and index (i). The PRU receives that message and stores the index in a ringbuffer of empty buffers. When it's time, the PRU retrieves a buffer index from the ringbuffer and, sample by sample reads the harvesting values (current and voltage) from the buffer, sends it to the DAC and subsequently samples the 'load' ADC channels (current and voltage), overwriting the harvesting samples in the buffer. Once the buffer is full, the PRU sends an RPMSG with message type (MSG_DEP_BUF_FROM_PRU) and index (i). The userspace application receives the index, reads the buffer, writes its content to the database and fills it with the next block of harvesting data for emulation.

2.2.3 Data extraction

The user space code (written in python) has to extract the data from a buffer in the shared memory. Generally, a user space application can only access its own virtual address space. We use Linux's */dev/mem* and python's *mmap.mmap(..)* to map the corresponding region of physical memory to the local address space. Using this mapping, we only need to seek the memory location of a buffer, extract the header information using *struct.unpack()* and interpret the raw data as numpy array using *numpy.frombuffer()*.

2.2.4 Database

In the current implementation, all data is locally stored on SD-card. This means, that for emulation, the harvesting data is first copied to the corresponding shepherd node. The sampled data (harvesting data for recording and load data for emulation) is also stored on each individual node first and later copied and merged to a central database. We use the popular HDF5 data format to store data and meta-information.

2.3 SYSFS interface

The shepherd kernel module provides a user interface that exposes relevant parameters and allows control of the state of the underlying shepherd engine consisting of the kernel module and the firmware running on the two PRU cores. When the module is loaded, the interface is available under */sys/shepherd*

- *n_buffers*: The maximum number of buffers used in the data exchange protocol
- *samples_per_buffer*: The number of samples contained in one buffer. Each sample consists of a current and a voltage value.
- *buffer_period_ns*: Time period of one 'buffer'. Defines the sampling rate togethere with *samples_per_buffer*
- *memory/address*: Physical address of the shared memory area that contains all *n_buffers* data buffers used to exchange data
- *memory/size*: Size of the shared memory area in bytes
- *sync/error_sum*: Integral of PID control error
- *sync/error*: Instantaneous PID control error
- *sync/correction*: PRU Clock correction (in ticks~5ns) as calculated by the PID controller

2.4 GPS-Time-Synchronisation

The ultimate goal is to precisely synchronize all Linux system clocks in the network to global time obtained from a GPS receiver. Note how the following description traverses the clock hierarchy top-down starting with the GPS receiver down to the linux clock on the host node.

2.4.1 General Structure

The master node is connected to a u-blox GPS receiver (shepherd-capelet), from which it receives a PPS signal via GPIO and global time via UART if the GPS-Modul has an position-fix. One kernel module and two services work in concert to provide a grandmaster clock:

- `pps_gmtimer` (kernel module) timestamps edges on a GPIO pin with respect to an internal hardware timer on the AM335x and feeds the timestamps as a Linux pps device.
- `gpsd` talks to the GPS via a serial interface, parses the protocol and extracts timing information that is fed to the following service (chrony).
- `chrony` combines the global timing information from gpsd and the precise PPS signal from the pps device and synchronizes the Linux system clock to global GPS time.

2.4.2 Hardware setup

The GPS-capelet (see *hardware/capelets/gps*) is stacked on top of the shepherd cape or an attached harvesting capelet, respectively.

- The PPS signal is connected to Timer4 on pin P8_7
- The GPS-Uart is connect over UART2 on pins P9_21 and P9_22
- The currently used gps-chip is the u-blox SAM-M8Q

2.4.3 Configuring GNSS module

The u-blox GPS receiver can be configured to optimize its perfomance. To do this directly form the host node the python the python script: `ubloxmsg_exchange` was written (see https://github.com/kugelbit/ubx-packet-exchange). The configuration files can be found under `config_files`. To this end the following configurations were set:

- SBAS was disabled for better timing information
- The stationary mode was enabled to get a better percussion and a stable perfomance
- the GPS- and galileo-satelite-systems were enabled to get a fast and stable fix

In addtion the standard config of the receiver leads to the following behaviour:

- If the PPS is not locked, the LED on the capelet will not blink. After the lock is attained, the LED will start blinking at 1 Hz.
- NMEA messages are enabled for the UART link which connects to the BeagleBone.

2.4.4 Deploy

Please use the ansible gps-host role. `ansible-playbook deploy.yml`

2.4.5 Useful commands

Check if PPS pulses are coming: `cat /sys/class/pps/pps0/assert` On the master check that gpsd and chrony are running:

- `systemctl status gpsd`
- `systemctl status chrony`

gpsd comes with a command line client that display GPS data on the commandline: `cgps` Familiarize with chrony output:

- `chronyc sources -v`
- `chronyc sourcestats -v`
- `chronyc tracking -v`

Configuration files:

- chrony: `/etc/chrony/chrony.conf`
- gpsd: `/etc/default/gpsd`
- the device tree file (dts) for the GPS-caplet is found in the pps-gmtimer folder (DD-GPS-00A0.dts).

DEBUG: if you want to see whats going on you can run the services in DEBUG-Mode:

- chrony: `sudo chronyd -dd`
- gpsd: `sudo gpsd -N /dev/ttyO2 -D15 -n`

2.4.6 Pitfalls

- For a good result (fast and stable fixes) it is very import to provide a clear sky view to the gps-caplet.

- The are lot of possible configuration options for the u-blox chip (see SAM-M8Q Receiver Description) that may furthre optimize performance. For more research in this direction use the u-blox u-center software.
- The currently used u-blox chip does not support the u-blox time mode. This mode can increase the timing accuracy (see u-blox M8 Receiver Description).

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

W