ho-brood-hostside

Release 1.0

Rob Mills, Rafael Barmak, Daniel Hofstadler

May 15, 2024

CONTENTS:

CHAPTER

ONE

QUICKSTART

Assuming that you are running the robot in conjunction with a RPi, running raspian or similar:

1. Install binary dependencies

```
sudo apt install python3-numpy
```

2. install the package

```
mkdir software && cd software
pip3 install brood_hostside.tar.gz
```

3. edit the example config to match your system

```
nano cfg/example.cfg
```

4. run the sampling-only handler:

```
python3 abc_read.py -c cfg/example.cfg
```

5. or run the actuator-enabled handler:

```
python3 abc_run.py -c cfg/example.cfg
```

CHAPTER

TWO

KEY CLASSES

2.1 libabc – ABCHandle

The primary class used to interact with the broodnest robotic frame is `libabc.ABCHandle`. This provides the top-level entry point to a robot module, facilitating the acquisition of samples from all sensors, as well as configuration, and actuator control.

class brood_hostside.libabc.**ABCHandle**(*cfgfile*, ***kwargs*)

Host-side interaction handle for a brood nest board.

Supply the config file name to load settings.

archive_cfg()

archive the config of the present run to log directory

prepare_heaters(*activate_any: bool = False*)

prepare the heaters to be used in current session

prepare the 'base state': - disable each individual heater - set each individual objective to *heater_def_tmp* (typically 0'C) - enable the global heater thread

if *activate_any* is set, for each entry in *self.activate_heaters*: - set objective - activate that heater

read_cfg()

Read configuration for connection and logging.

uptime()

Generate string report of uptime.

first_conn()

Collect status after connection.

get_usb_props()

grab and log properties from the USB serial interface

check_newday_and_roll_logfiles()

roll over logfiles on first sample of new day.

sample_temp_sensors()

Top level wrapper to get temp data from MCU, logging to file + DB

sample_htr_sensors()

Read heater data.

Get a heater string and a heater dict, log the str, inject the dict, log to CSV (by passing str to get_htr_str()).

heaters_ini()

Put each individual heater to de-activated, and enable the global heater thread. –> Heaters will be ready to activate

get_heaters_status()

Returns a numpy array of all heaters for each measurement.

get_heaters_active(*_idx: int = None*)

return status of heater [idx], or whole vector of 10 status if None

set_heater_active(*_state: bool*, *_idx: int*, *warn_below_deg: float = None*, *clear_t_obj: bool = True*) → bool

Set the activated/deactivated state of heater *idx* to *_state*.

emits a warning if an actuator is set to be activated but the objective temperature is too low (skip if *warn_below_deg* is None)

if setting *_state* to False, also clear the objective by default.

heaters_deactivate_all()

Deactivate all heaters (consecutively) & deactivate heater thread.

!!! The heaters can not be deactivated all at the same time. For some reason the simultaneous deactivation is creating a current spike, leading to a reboot of the ABC. Deactivation should be done one by one.

All heater objectives are also reset back to default value (normally=0)

Note: individual heaters' activate/de-activate state persists through any global activate/deactivate changes. That is, whether the global state is enabled or disabled, the individual state changes can be made.

disable_heater_global_thread()

turns off the global heater thread.

reset_htr_objectives(*t_obj: float = None*)

reset all heater objectives to *self.heater_def_tmp*,

override the default (from cfgfile) by setting t_obj

get_heaters_objective(*_idx: int = None*) → float | ndarray

return objective of heater [idx], or whole vector of 10 status if None

loop(*consume=True*)

Grab all data from ABC comb.

If *consume* is true, also consume remaining time in sampling cycle.

consume_idle_time()

Separate function in case other steps added to the main loop.

prepare_initial_loop_timer(*ref: datetime = None*)

Define now as the intended reference time for loop time counting

This method is useful if the ABCHandle instance is expected to be instantiated a significant period before the main loop might start. It just sets the next time point, *t_end* for loop(consume=True).

get_mem_status() → dict

get the current memory status (allocated, free, total)

check_mem_maybe_gcollect(*watermark: float = 80.0*, *_cmd: str = None*) → bool

check current memory usage, and if usage is above *watermark* then a garbage collection is requested.

return value: True if collection was done.

safe_exec_abc(*cmd: str*, *retries: int = 1*, *minlen: int = None*, *nfields: int = None*, *watermark: float = 80.0*, *loglvl: str = 'CMD'*) → bytes

Execute a command on the ABC microcontroller

- **retry on failure (e.g. if garbage collector interrupts response)**
  if retries is exceeded, ABCGarbledRespError is raised
- **optionally provide minimum expected response lengths**
  if not met, ABCTooShortRespError is raised
- internally this parses the content for validation, but the returned data is the raw bytestring received from the ABC.
- if *watermark* is set, this method also checks the memory status, and if usage is above watermark, a garbage collection is requested. To unset, define as None.

exec_abc(*cmd*, *minlen=None*, *nfields=None*)

Use pyb.exec on the MCU side to obtain results.

Optionally provide min expected response length, if not met raises ABCTooShortRespError. Log to debug as needed.

get_mcu_id()

Read UUID from MCU.

get_temp_offset()

Get temperature offsets for each of the 64 sensors.

Returns: List of floats

reset_temp_offsets() → bool

Returns: boolean success status

set_sensor_interval(*_period_in_s*)

Set temperature interval.

get_sensor_interval() → dict

Get interval for sampling each sensor class.

get_int_temperatures()

Get temp data from all sensors, using integer values on retrieval.

This function converts to floats in Celsius.

Returns: timestamp (datetime object),
temperatures (np array of floats), valid status (boolean)

get_datetime() → datetime

Parse datetime from ABC timestamp.

Returns: Datetime object = RTC time

``bytestring e.g.: b'$$2021,11,5*$17,3,11**

‘‘‘

```
decoded -> '$$2021,11,5*$17,3,11**'
```

compare_datetime()

Return time diff between the ABC PCB and the RPi.

check_update_datetime()

Check clock drift on the ABC and reset if necessary.

2.2 libabc – exceptions

exception brood_hostside.libabc.**ABCBaseError**

base class for errors

exception brood_hostside.libabc.**ABCTooShortRespError**

response length was shorter than expected

exception brood_hostside.libabc.**ABCGarbledRespError**

response included unexpected elements - garbage collection responses - ...

exception brood_hostside.libabc.**PyboardError**

exception brood_hostside.libabc.**TimedOutException**

2.3 libabc – other functionality

brood_hostside.libabc.**parse_pwr**(*m*)

data comes from *get_power_ui()*

` timestamp, volt, shuntvm, amps, power, bool(valid) date, time, (2021, 6, 19),(15, 49, 31) 12.0 [bus V] 0.0005549999 [shunt V] 0.05548096 [current, amps] 0.6660461 [power, W] True [valid, boolean] `

brood_hostside.libabc.**parse_co2**(*m*)

date comes from s.get_co2_rht timestamp, rh, co2ppm, tmp, valid

brood_hostside.libabc.**parse_rht**(*m*, *numeric_only=True*)

date comes from s.get_rht ts, _conv_rh(rh), _conv_temp(t), _get_mode(mode), bool(valid)

class brood_hostside.libabc.**HtrReadings**

simple container for the multi-dimensional heater data

clear()

clear the values from all subfields

CHAPTER

THREE

SUPPORT CLASSES

The functionality of `brood_hostside` is divided into several modules, which implement other classes or supporting functions used to interact with the broodnest robotic frame. Most of these libraries are back-end, instantiated by `libabc.ABCHandle`, but users do not need to be overly concerned with the details.

3.1 Database interaction – libdb

Class to interact with influxDB database (V2.x), sending points from ABC measurements.

Date formats are important for injecting the data correctly, including timezones. The method *libbase.ABCBase.timestamp4db()* prepares the timestamps consistently, here we add notes for reference only.

Influx supports several formats, we elect to use iso-8601 format, using a UTC timezone. For one example date, these representations are equivalent:

- unix timestamp: 1628768585
- human-readable: Thu 12 Aug 13:43:05 CEST 2021
- ISO-8601 (w/CEST offset) 2021-08-12T13:43:05+02:00
- ISO-8601 (w/UTC offset) 2021-08-12T11:43:05+00:00
- ISO-8601 (compact, if UTC) 2021-08-12T11:43:05Z

We use the last one in this package.

exception brood_hostside.libdb.**DBBaseError**

base class for errors relating to DBInjector & libdb functions

exception brood_hostside.libdb.**DBFailedWrite**

Could not write points to database

exception brood_hostside.libdb.**DBBadFormatWrite**

write request message is badly formatted (did you use the api to form it?)

exception brood_hostside.libdb.**DBBadCredentials**

incorrect authorisation for database

exception brood_hostside.libdb.**DBTimedoutOnWrite**

Could not connect to DB on write - http timeout

brood_hostside.libdb.**wait_for_engine**(*host*, *port*, *maxcount=50*, *timeout=3*) → bool

block until ping response is ok from influxDB instance *host:port*

returns True if successful, False if reached *maxcount* unsucessful attempts each with a duration *timeout*.

class brood_hostside.libdb.**BaseDBInjector**(*credfile: str | Path*, *start_connected: bool = False*, ***kwargs*)

A class providing influxDB database interaction

read_influxdb_conn_cfg()

read the influxDB connection settings

get_measurements()

Query ifdb2 for list of measurements in the current bucket

write_points(*points: List[dict]*, *verb: bool = False*) → bool

write a list of dict points to DB

write_linepoints(*points: List[str]*, *verb: bool = False*) → bool

write a list of line protocol points to DB

class brood_hostside.libdb.**DBInjector**(*credfile: str | Path*, *start_connected: bool = False*, ***kwargs*)

A class providing influxDB database interaction and point preparation for broodnest modules, loading and setting metadata for the tags, and appropriate date/time construction.

populate_metadata()

Fill the tags dict to put with all measurements.

set_meta_uuid(*uuid: int*)

supply the UUID from the MCU (a 96-bit int)

set_meta_serial(*sid: int | str*)

supply the USB serial device identifier, e.g. N03

set_meta_board(*board_id: str*)

supply the board ID, e.g. abc03

prep_htr_pointlist(*heat_dict: dict*, *n_htrs: int = 10*) → List[dict]

generate list of 10 point dicts, one per actuator

input: dict with 50 elements of data plus some metadata, where the keys have field x instance encoded.

output: list of 10 x 5-field points, with all entries ready for influxDB (measurement, tags, fields timestamp)

dump_point(*point: dict*) → str

Convert a point-dict to (the influxDB-native) line format.

The point dictionary should be in JSON format.

3.2 User interface wrappers – libui

Some functions and boiler-plate code used in interfaces for ABC instances.

brood_hostside.libui.**handle_known_exceptions**(*e*, *logger=None*, *verb=True*)

various errors come up with long interactions with the upy board, here we try to handle some of the harmless ones.

- print a message to logger.
- return False if ok to continue, True if fatal and we should halt

brood_hostside.libui.**lookfor_cfgfile**(*pth=None*, *debug=False*)

default location is <dir of tool>/cfg expected filename pattern is <pth>/<hostname>[.somext], (where the extension might be .ini, .conf, .cfg, but is not required.)

file default pattern is hostname returns a Path object, or None if not found

brood_hostside.libui.**abc_parser**()

construct argparse object for ABC configs

brood_hostside.libui.**verify_abc_cfg_file**(*args*)

Check if config file exists, or attempt to lookup based on hostname

brood_hostside.libui.**process_exception**(*is_bad_err*, *e*, *ABC*)

if the exception is unknown, or known to be very bad, log it and re-raise else, just print some information to the user. (The latter case includes ABCGarbledRespError, for example).

3.3 Logging library

brood_hostside.liblog.**get_heater_field_keys**(*n_heaters: int = 10*) → list

Generate the header for the heater csv.

class brood_hostside.liblog.**ABCLogger**(*path_cfg*, ***kwargs*)

Logger for broodnest interactions.

Log files are written to the folder *logroot* (specified in the configfile), e.g. "/home/pi/log/abc_logs/" as of writing.

E.g., on hive5-rpi4, which controls the ABC board 'abc08', today, these 7 files are produced upon initialization of an ABCLogger:

- abc08_2021-11-04.dbg
- abc08_2021-11-04.log
- abc08_co2_2021-11-04.csv
- abc08_htr_2021-11-04.csv
- abc08_pwr_2021-11-04.csv
- abc08_rht_2021-11-04.csv
- abc08_tmp_2021-11-04.csv

If the files exist, they will be appended to; if not, they are created. New CSV files are initialized with a header.

This class is agnostic of rollover-intervals (as of now).

init_logfiles(*create_msg: str = ''*)

Initialize the logfiles.

reinit(*reason=''*)

Reinitialize all logfiles after a rollover.

The path *self.logroot* (==*self.path_log*) does not change, so there is no need to reinitialize it.

logline(*msg*, *level='INF'*)

Writes *msg* to the logfile, including some metadata.

The formatted output uses | as the separator, and includes: - a three-letter code indicating the severity or other info (e.g. ERR) - the unix timestamp, in UTC [e.g. 1636149191] - a human-readable timestamp

(iso8601) [e.g. 2021-11-05T21:53:11Z] - the message itself, which could in principle contain further separators

logdatalst(*lst*, *field*, *add_date=True*)

Log data from list into csv form.

If add_date is set, also add a UTC unix timestamp at the start.

3.4 Baseclass – libbase

class brood_hostside.libbase.**ABCBase**(***kwargs*)

Provide basic functionalities for the ABC classes.

Contains all basic time functions as well as hostname- related methods.

utcnow()

Return TZ-aware datetime object of current time.

Use this wrapper to ensure consistent use of timestamps throughout logs.

get_dt_day(*dt: datetime*) → datetime

Return datetime object of the day at midnight.

dt_to_isofmt(*dt: datetime*, *abbrev_utc: bool = True*) → str

Format datetime object *dt* as per ISO8601 / rfc3339 format.

- dt should be tz-aware
- output will be of the form
  - `2021-08-12T12:43:05+01:00`
  - in the special case of UTC we use the shorter 'Z' notation
  - `2021-08-12T11:43:05Z`

dt_to_unix(*dt: datetime*) → int

Convert datetime object *dt* to a unix timestamp in seconds.

Returns the timestamp as UTC - so long as *dt* is aware.

ftime(*dt: datetime = None*) → str

Generate HH:MM:SS representation of a datetime *t*.

if no arg is supplied, generate current (UTC) time.

timestamp4db(*ts: float*) → str

Convert datetime to influxDB-format.

Assumes input of a TZ-aware datetime object a unix timestamp (sec since 1970), generates UTC timestamp of iso-8601 format

parse_boardname(*addr: str*) → str

Return the board name from the `addr` string.

The expected boardname, if udev rules are installed correctly is:

/dev/abc01 (current version of rules) /dev/brood_abc01 (earlier version of rules)

–> method will yield `abc01`

Without rules implemented, we might also see: /dev/ttyACM0 (no rule matching the specific ID installed, linux)

–> method will yield `ttyACM1`

/dev/cu.usbmodem?? (macOS)

–> method will yield `cu.usbmodem11`

safename(*fp: Path*, *p_type: str = 'file'*) → Path

Append stuff to a file or folder if it already exists.

Check whether a given file or folder 's' exists, return a non-existing filename.

fp ..... (full) filename or directory p_type ... 'file' or 'f' for files, - 'directory' or 'dir' or 'd' for folders

Returns a file- or pathname that is supposedly safe to save without overwriting data.

lookfor_cfgfile(*pth: Path = None*, *debug: bool = False*) → Path

Return the location of the config file.

Default location is '<dir of tool>/cfg'. Expected filename pattern is <pth>/<hostname>[.somext], where the extension might be .ini, .conf, .cfg, but is not required.

File default pattern is *hostname*.

Returns a Path object, or None if not found

git_info_str(*compact: bool = False*) → str

look up git repository info (branch, commit, clean/dirty state)

class brood_hostside.libbase.**BackoffCtr**(*per_level=1*, *max_count=360*)

Counter to increase duration between yielding True

It could be used to help emitting messages only infrequently if the same action occurs repeatedly.

By default, it implements an exponential increase in between messages.

hitok()

tell the counter the condition was ok

hitwarn()

increment counter and tell the counter the condition was bad

CHAPTER

FOUR

USAGE NOTES FOR LIBRARY

The package is well managed using systemd service units, and automatic selection of configuration files (implemented via hostname lookup). However, if one desires a simpler interaction, e.g. for prototyping a new controller, it is possible to run the code directly on a host. For session persistence, we suggest using *screen* or *tmux*.

4.1 Basic usage for prototyping

Typically run this inside a screen or tmux session

```
# login to host with the robotic frame attached
ssh <hostname>
# start a new, named screen session
screen -S board04
# go to runtime code directory
cd software/broodnest/runtime_tools
python3 abc_read.py -c cfg/my_cfg04.cfg
```

Stop recording with `ctrl-C`, it exits cleanly.

4.2 Some quick notes about *screen*:

- detach from session, leaving it running: `ctrl-A`, `d`
- close session, especially if behaving badly: `ctrl-A`, `k`, then `y`
- close session: `ctrl-D` (as per closing any shell)
- check what sessions are running: `screen -ls`
- reattach to a specific screen session: `screen -r <session name>`, e.g. `screen -r board04`
- reattach to a specific session, if somehow it is still open `screen -Dr <session name>`
- **scroll up within session (see history/more than a few lines of error!):**
  - `ctrl-A`, `q`, use mouse wheel or arrow keys.
  - Press `Esc` to go back to regular mode.

CHAPTER

FIVE

SOME DEVELOPER NOTES

5.1 Requirements

The python packages required are defined in *setup.py* `:

- pyserial (note: NOT *serial*; both are used as *import serial*, confusingly)
- influxdb-client
- numpy

The current version has been tested using RPi platforms of armv7l and arm64 variants, running python versions 3.7.3, 3.9.2. For database interaction, it has been tested with *influxdb-client==1.36.0* and influx 2.x databases.

The database of influx 2.x has been tested hosted on ubuntu 22.04 servers and on RPi4B with arm64 arch. With a 64-bit architecture, a RPi can host both a local influx database instance and one or more robots. Other documentation within the project will explain such setups further if needed.

Earlier versions were tested with the client *influxdb==5.2.0* for influx 1.x databases, and tested with influxd 1.6.4 (hosted on RPi3B with armv7l arch).

5.2 Database setup

Measurements generated are in sub-tables (influxDB calls these different 'measurements'):

rht	temp	co2	pwr	htr

Metadata relating to the robot (e.g. short name, MCU-UUID) and its installation location (e.g. hive number, geographic location) are attached to the injected points. In InfluxDB, metadata is called 'tags'. The cardinality of the unique tag combinations, but not the number of tags, affects the performance of the database – so more tags does not inherently mean worse performance. See influxDB docs for more details.

5.2.1 ## Links to docs

- **InfluxDB python client, v2.x (*import influxdb_client.InfluxDBClient*)**
  - Github repo
  - readthedocs

5.3 Python version

Due to deployment on resource-limited devices, we avoided using too many recent features of python. Much development was validated using python 3.7.3, requires >=3.6

- **requires >=3.6**
  - f-strings are used
  - using *timespec* arg in *datetime.datetime.isoformat()*
- **requires >=3.5**
  - function type hints used in datetime handling

5.4 Some open issues

See issue tracker within github repository

CHAPTER

SIX

CITATION

Our article in IEEE Access contains detailed information on the design and validation of the robotic system.

R. Barmak, D.N. Hofstadler, M. Stefanec, L. Piotet, R. Cherfan, T. Schmickl, F. Mondada, R. Mills (2024) "**Biohybrid Superorganisms—On the Design of a Robotic System for Thermal Interactions With Honeybee Colonies,**" in *IEEE Access*, vol. 12, pp. 50849-50871, 2024, doi: 10.1109/ACCESS.2024.3385658.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

H

I

L

M

P

R

S

T

U

V

W