seabird_ctd Documentation

Release 0.2.4.5

Nick Santos

Nov 16, 2018

Contents:

CHAPTER 1

Usage and Examples

The following example shows some basic usage of the seabird_ctd code

```
import seabird_ctd

def handle_records(records):
    """
        This function receives the read data as a list of dicts, and you
        can handle the data however you like in here. In this case, we
        instantiate a new django model (not defined in this example)
        and set its values to match the records received before saving.
    """

    for record in records:
        new_record = CTD()
        new_record.temp = record["temperature"]
        new_record.conductivity = record["conductivity"]
        new_record.pressure = record["pressure"]
        new_record.datetime = record["datetime"]
        new_record.save()

ctd = seabird_ctd.CTD(port="COM6", baud=9600, timeout=5)  # connect to the device␣
↪itself over pyserial
if not ctd.is_sampling:  # if it's not sampling, set the datetime to match current␣
↪device, otherwise, we can't
    ctd.set_datetime()  # this just demonstrates some of the attributes it parses␣
↪from the status message
else:
    log.info("CTD already logging. Listening in")

ctd.start_autosample(interval=60, realtime="Y", handler=handle_records, no_stop=True)␣
↪ # start listening in to autosampling results. If it's already sampling, it can␣
↪stop it, reset parameters, and start it again, or leave it alone, depending on the␣
↪options you define here
```

CHAPTER 2

Compatibility

Seabird CTD library is developed for Python 3. Most code should be compatible with Python 2.7, but has not been tested. If you'd like to make it Python 2.7 compatible, please submit a pull request.

CHAPTER 3

Simple Setup

This library, with few dependencies, can send commands directly to the device and monitor and parse responses.

seabird_ctd requires two external packages for simple setup:

- six (for Python 2/3 compatibility)
- pyserial (for communication with the CTD)

After installing those packages and seabird_ctd, you are set up to use the package

When instantiating the CTD object in your code, no arguments are required except the COM port that the CTD communicates on. If you wish to not specify it in each script, you can also set an environment variable SEABIRD_CTD_PORT, which will be checked. Other options are available, including baud rate and the timeout for reading data (in seconds).

In simple mode, if you initialize autosampling, then the program goes into a loop that it won't exit from waiting for CTD data. To exit, you will need to either break/interrupt the program (Ctrl+C) or kill the process.

CHAPTER 4

Advanced Setup

Advanced setup allows you to send manual commands to the CTD while it is sampling. It uses a different architecture that listens for commands and sends those when they are received while still monitoring for data from samples.

Advanced setup requires significant external setup and small changes to your code. To use this mode, you must install:

- Erlang (for RabbitMQ)
- RabbitMQ
- Pika python package

You must then set up and configure Rabbit MQ with a virtual host and a user account that has config privileges in that virtual host.

Then, in your code, prior to initiating autosampling, you call

```
ctd.setup_interrupt({server_ip_address},
                    {rabbit_mq_username},
                    {rabbit_mq_password},
                    {vhost})
```

Then, you can send commands via rabbit_mq to the queue in the virtual host that is named by COM port. So, if your CTD runs on COM4, then send messages to the queue "COM4". Example code, again, as a Django management command, follows:

```
class Command(BaseCommand):
    help = 'Sends commands to the CTD while autosampling'

    def add_arguments(self, parser):
        parser.add_argument('--server', nargs='+', type=str, dest="server",␣
↪default=False,)
        parser.add_argument('--queue', nargs='+', type=str, dest="queue",␣
↪default=False,)
        parser.add_argument('--command', nargs='+', type=str, dest="command",␣
↪default=True,)
```

(continues on next page)

(continued from previous page)

```
    def handle(self, *args, **options):
        queue = None
        if options['queue']:
            queue = options['queue'][0]
        else:
            queue = os.environ['SEABIRD_CTD_PORT']

        server = None
        if options['server']:
            server = options['server'][0]
        else:
            server = "192.168.1.253"  # set a default RabbitMQ server if not specified

        command = options['command'][0]  # get the actual command to send

        # connect to RabbitMQ
        connection = pika.BlockingConnection(pika.ConnectionParameters(host=server,␣
↪virtual_host="moo", credentials=PlainCredentials(local_settings.RABBITMQ_USERNAME,␣
↪local_settings.RABBITMQ_PASSWORD)))
        channel = connection.channel()

        channel.queue_declare(queue=queue)

        # send the command message
        channel.basic_publish(exchange='seabird',
                              routing_key='seabird',
                              body=command)
        print(" [x] Sent {}".format(command))
        connection.close()
```

You can send any command that the CTD supports executing while autosampling. See the manual for your CTD for more information on that. You can also send three special commands. READ_DATA initiates an immediate read of the data and sends it to the configured handler. STOP_MONITORING leaves the CTD recording data, but stops the Python code from checking for data. DISCONNECT is closely related to stop monitoring, but also sends a sleep command to the device and closes the serial connection.

CHAPTER 5

Extending the code to a new CTD

CTDs have slightly different command styles, behaviors, and syntaxes and the authors of this package don't have access to all of the CTDs Seabird produces. While the package has been tested with a handful of CTDs and firmware versions (SeacatPlus (SBE19plus), SBE39 firmware 1.5, SBE39 firmware 6, and SBE37), other models do not have support yet. This code is meant to provide a structure where it can be extended to support other, newer models, quickly (15 minutes to a few hours). Until further documentation is written, the best way to see how this is handled is to take a look at the objects for each CTD in the code in the file ctd_models.py. They define specific parsing information and command syntax for the CTDs.

CHAPTER 6

API documentation

SeaBird CTD is meant to handle reading data from a stationary CTD logger. Many packages exist for profile loggers, but no options structured for use on a stationary logger or reading. This package is designed to remain directly connected to a CTD while it's logging and can be extended to support communication with different versions of the Seabird CTD loggers with different capabilities and command descriptions.

class seabird_ctd.**CTD**(*COM_port=None*, *baud=9600*, *timeout=5*, *setup_delay=2*, *wait_numerator=200*, *send_raw=None*, *debug=False*)

If COM_port is not provided, checks for an environment variable named SEABIRD_CTD_PORT. Otherwise raises CTDConnectionError

Parameters

- **COM_port** – string, COM_port to connect to
- **baud** – int for what baud rate to use to communicate with the device
- **timeout** – passed through to serial package - in most cases, won't be used. It's a read timeout when requesting data from the device - this package, by default only reads the data it knows to be waiting on the line, but it can read with the timeout as well.
- **setup_delay** – How long, in seconds, should it wait to let the CTD wake?
- **wait_numerator** – integer, milliseconds. Represents the top portion of a fraction (wait_numerator/baud) representing how long the code should wait after reading to check if there is more data on the line. If it waits that time and there is no data on the line, it moves to processing the data. Decrease for speed, but more read errors, increase if you're getting lots of read errors. As is, it *should* work for most devices and baud rates.
- **send_raw** – a writeable file handle that the raw response, preprocessing, from the CTD will be sent to. Useful for debugging and extending this module to new CTDs, as well as for unit testing
- **debug** – boolean. Changes bits of behavior for debugging purposes.

battery_change

battery_voltage

catch_up()

Meant to pull in any records that are missing on startup of this script - if the autosampler runs in the background, then while the script is offline, new data is being stored in flash on the device. This should pull in those records.

NOTE - this command STOPS autosampling if it's running - it must be restarted on its own.

Returns

check_data_for_records(*data*)

Check the received data for temperature pressure and salinity data. If this data is present, we have a new record.

@param data: string, data array to examine for CTD records.

@return: Boolean, True if the record count is greater than 0.

check_interrupt()

Should return True if we're supposed to stop listening or False if we shouldn't

Returns

close()

Put the CTD to sleep and close the connection

Returns

determine_ctd_model()

find_records(*data*)

Given the parsing regex defined in the command object for this CTD, runs it and returns a list of dicts containing sampled values. The dict will always have the keys defined in the command object's "keys" attribute. This varies model to model, so your code may need to check for keys, depending. In the event of values that are not always enabled (like salinity, or sound velocity), the keys are still present, but set to None if those values are not enabled.

Parameters **data** –

Returns

listen(*interval=60*, *max_iterations=None*)

Continuously polls for new data on the line at interval. Used when autosampling with realtime transmission to receive records.

See documentation for start_autosample for full documentation of these parameters. This function remains part of the public API so you can listen on existing sampling if the autosampler is already configured.

Parameters

- **handler** – (See start_autosample documentation). A functiion to process new records as they are available.
- **interval** – The sampling interval, in seconds.
- **max_iterations** – Primarily used for testing, sets how many samples it should try to read before exiting the listening loop and returning control to the program. This could also be used by another program to do intermittent checks for data since it's the only way to

exit the listen loop without sending a break event to the program. Default is None, which indicates to run indefinitely

Returns

read_records()

recover()

This is for using if the line is interrupted by something, but the script stays running. In our production, if power goes out to an intermediate set of devices, the CTD can go down. This method closes the existing ctd connection, reopens it, tries to reestablish communication with the device. Triggered if we get a UnicodeDecodeError parsing the data.

Returns

send_command(*command=None*, *length_to_read='ALL'*)

set_datetime(*raise_error=False*)

Attempt to set the datetime. Logs a warning if that can't be done because the sampler is logging. If you want it to raise an exception

Parameters **raise_error** – When False, the default, this function warns if it can't set the datetime. When True, raises CTDOperationError

Returns

setup_interrupt(*rabbitmq_server*, *username*, *password*, *vhost*, *queue=None*)

Used to set the monitoring functions to use the interrupt method, which allows messages to be passed to the CTD from the user even while monitoring for data. By default, if this function is not called, then the monitoring code cannot be interrupted and simply runs until the user cancels it.

Parameters

- **rabbitmq_server** –
- **username** –
- **password** –
- **vhost** –
- **queue** –

Returns

sleep()

Puts the device into Quiescent (sleep) mode. Most devices do this automatically after a few minutes

Returns

start_autosample(*interval=60*, *realtime='Y'*, *handler=None*, *no_stop=False*, *max_iterations=None*)

This should set the sampling interval, then turn on autosampling, then just keep reading the line every interval. Before reading the line, it should also check for new commands in a command queue, so it can see if it's should be doing something else instead.

We should do this as an event loop, where we create a celery task to check the line. That way, control can flow from here and django can do other work in the meantime. Otherwise, we can have a separate script that does the standalone django setup so it can access the models and the DB, or we can just do our own inserts since it's relatively simple code here.

Parameters

- **interval** – How long, in seconds, should the CTD wait between samples
- **realtime** – Two possible values "Y" and "N" indicating whether the CTD should return results as soon as it collects them
- **handler** – This should be a Python function (the actual object, not the name) that takes a list of dicts as its input. Each dict represents a sample and has keys for "temperature", "pressure", "conductivity", and "datetime", as appropriate for the CTD model. It'll skip parameters the CTD doesn't collect. The handler function will be called whenever new results are available and can do things like database input, etc. If realtime == "Y" then you must provide a handler function.
- **no_stop** – Allows you to tell it to ignore settings if it's already sampling. If no_stop is True, will just start listening to new records coming in (if realtime == "Y"). That way, the CTD won't stop sampling for any length of time, but will retain prior settings (ignoring the new interval).
- **max_iterations** – Primarily used for testing, sets how many samples it should try to read before exiting the listening loop and returning control to the program. This could also be used by another program to do intermittent checks for data since it's the only way to exit the listen loop without sending a break event to the program. Default is None, which indicates to run indefinitely

Returns

status(*status_parts=None*)

Gets and parses the status. Occasionally, there's a race condition where even after checking for new data up front, we can have new data at the beginning of the status.

A future enhancement would involve:

- when run with the model already defined (after first startup)
- each model finding the line that starts with the model name, and then
- send all prior lines for processing

That would guarantee no data is lost to the DS call.

Parameters **status_parts** –

Returns

stop_autosample()

take_sample()

wake()

exception seabird_ctd.**CTDConfigurationError**

exception seabird_ctd.**CTDConnectionError**

exception seabird_ctd.**CTDOperationError**

exception seabird_ctd.**CTDUnicodeError**(*response*)

exception seabird_ctd.**CTDUnsupportedError**

class seabird_ctd.**TimeoutTimer**(*duration=0*)

Use to set timing for sample intervalse to prevent excessively long blocking calls with time.sleep()

check_timeout()

Return the number of of seconds until timeout. If the duration has not been elapsed, return a positive number, otherwise the return value is negative.

reset_timeout(*duration*)

Reset the timer to some time further into the future.

seabird_ctd.**interrupt_checker**(*server*, *username*, *password*, *vhost*, *queue*, *interval*)

When using the interrupt method, this code handles the scheduling of the actual checking by connecting to RabbitMQ and sending READ commands every interval

Parameters

- **server** –
- **username** –
- **password** –
- **vhost** –
- **queue** –
- **interval** –

Returns

class seabird_ctd.ctd_models.**CTDCommandObject**

Just an empty class that can be subclassed so that any isinstance checks would work

clean_response(*response*)

clean_status(*status*)

REMINDER: THIS REMOVES *ALL* empty lines, not just leading ones (eg, SBE19plus which has some in the middle)

Parameters **status** –

Returns

operation_wait_times = {}

txrealtime(*value*)

class seabird_ctd.ctd_models.**SBE19plus**(*main_ctd*)

parse_status(*status_message*)

record_regex()

Handles generation of the regex for the data records. If salinity output is turned on, handles that correctly. STILL TO DO - handle other optional values, such as sound velocity - what happens if sound velocity is on, but salinity is off, or if both are on?

Returns

sample_interval(*interval*)

set_datetime()

setup()

txrealtime(*value*)

class seabird_ctd.ctd_models.**SBE37S**(*main_ctd*)
Handles the SBE37S commands

parse_status(*status_message*)

record_regex()

Handles generation of the regex for the data records. If salinity output is turned on, handles that correctly. STILL TO DO - handle other optional values, such as sound velocity - what happens if sound velocity is on, but salinity is off, or if both are on?

Returns

retrieve_samples(*start*, *end*)

sample_interval(*interval*)

set_datetime()

setup()

class seabird_ctd.ctd_models.**SBE37SM**(*main_ctd*)

parse_status(*status_message*)

class seabird_ctd.ctd_models.**SBE39**(*main_ctd*)

parse_status(*status_message*)

record_regex()

retrieve_samples(*start*, *end*)

sample_interval(*interval*)

set_datetime()

setup()

class seabird_ctd.ctd_models.**SBE3915**(*main_ctd*)
Older SBE39 - firmware 1.5 - similar, but different status parsing

clean_response(*response*)

parse_status(*status_message*)

setup()

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

T

W