Ceylan-Oceanic: Enocean facilities in Erlang

Organisation: Copyright (C) 2022-2025 Olivier Boudeville

Contact: about (dash) oceanic (at) esperide (dot) com

Creation date: Wednesday, September 7, 2022

Lastly updated: Saturday, April 19, 2025

Version: 1.4.6

Status: In development

Dedication: Users and maintainers of the `Ceylan-Oceanic` library.

Abstract: The role of the `Ceylan-Oceanic` library is to provide Erlang-based facilities for the support of the Enocean building automation system.

The latest version of this documentation is to be found at the official Ceylan-Oceanic website (`http://oceanic.esperide.org`).

This documentation is also mirrored here.

Table of Contents

Overview

The Ceylan-Oceanic library provides Erlang-based facilities for the support of the Enocean building automation system, an open standard whose devices are generally energy-harvesting / very low-consumption, and wireless (supported frequencies around 900 MHz, depending on countries; for a range of up to 300 meters in the open, and up to 30 meters inside buildings) with very low traffic.

So Enocean, whose slogan could be "*no wire, no battery*", is rather unique. No Wifi involved (and very little radio frequency exposure: due to energy constraints, few short, terse telegrams are exchanged; no real risk of interferences), no IP connectivity either, hence no real risk in terms of health or privacy/data leak (see next section), as Oceanic just receives / decodes / encodes / emits series of well-determined bytes, and remains in full control at all times: an Enocean gateway can only communicate with its host (hence with Oceanic), moreover through a low-level USB serial interface (no third-party driver involved on that hosting computer), and the devices only send tiny telegrams that can be listened to only at short range (up to a few dozens meters).

Finally, at least most of the Enocean specifications are freely available.

Besides Erlang, Ceylan-Oceanic relies only on Ceylan-Myriad and is a rather autonomous part of the Ceylan project. Ceylan-Oceanic can be readily built and run on most Unices, including of course GNU/Linux. An example of use of Oceanic is our US-Main home automation server.

The Ceylan-Oceanic project repository is located here.

At least a basic knowledge of Erlang is expected in order to use Ceylan-Oceanic.

A Word About Other Standards and Confidentiality

Compared to Enocean, more recent technologies and open standards exist, including Matter. They are increasingly promoted by Amazon, Apple and Google, so that they integrate with their respective home assistants; they communicate over IP.

Such devices are certainly convenient and often cheap, yet, as for us, we prefer not having in our home "Big Five"-originating black boxes generally full of sensors, cameras and microphones, running closed, proprietary, transparently-upgradable software, having their own IP connectivity (they typically obtain through DHCP their own local IP address) and therefore able to communicate rather freely with any "cloud" on the Internet (in practice, almost nobody blocks outgoing traffic from such dynamically-allocated IPs - see our example of script though - knowing moreover that these devices often rely on Internet services and expect to regularly update their software).

So we are a bit puzzled that so many people trust such home automation devices to the point of actually happily *purchasing* them, and placing them at the core of their home - whereas there are already examples showing, if necessary, that their owner is certainly not in full control of them.

Many will consider that non-IP protocols like Enocean or Z-Wave are already superseded by newer technologies like the aforementioned Matter system. Another point of view is that a standard like Enocean (that is moreover interesting

for its unique energy-harvesting capabilities) is probably the last (and thus the most advanced one) that *can be easily trusted*.

It could also be argued that all wireless protocols are flawed anyway, as they can be relatively easily snooped and/or jammed, as opposed to wired ones (like with KNX, X10 and other PLC-based ones) that are by design safer / more reliable (albeit more expensive). At least for new buildings (as opposed to partially-renovated ones), such wired systems could be considered, but, to the best of our knowledge, no such practical (open, affordable, future-proof) option exists (and this is a bit of a pity).

So overall we consider that sticking to Enocean makes sense, hopefully for a long time.

Purpose

The main motivation of Oceanic is to provide some basic home automation features, especially here in terms of security, in order to be able to:

- **intercept and decode telegrams** emitted by **sensors** - notably single-input contacts (to detect the opening/closing of doors or windows), presence or temperature / humidity sensors or to detect electricity outages, sensor losses (having been sabotaged or running out of energy), jamming attempts, typically in order to implement one's own alarm center; should a security event happen, a network camera can be switched on, e-mails and/or SMS can be sent, etc.
- **generate and emit telegrams** to control, as **actuators**, any kind of electrical devices (driven by a smart plug or an in-wall module), typically to turn on an electric heater or to run one's own presence simulator (possibly with lamps and sound devices)

Progress & Enocean Coverage

The targeted basic Enocean support has been implemented, so EEP Enocean (sensor-emanating) telegrams can be intercepted and, for the supported EEPs (other ones may be quite easily added), such telegrams can be properly decoded and notified as higher-level, incoming events to be managed by one's application.

Reciprocally, telegrams for the supported EEPs can also be encoded and sent, and they are able to trigger appropriately-configured (Enocean) devices (actuators).

Oceanic can also execute a few common commands directly onto the local USB gateway chip.

Testing Ceylan-Oceanic in Two Steps

Now, let's discuss all these subjects a bit more in-depth.

Hardware Prerequisites

In terms of Enocean devices, one needs typically:

- any kind of emitter/sensor device, for example a single-input contact/rocker button like these ones; opening sensors are also convenient, as we can easily act on them directly (i.e. by approaching a magnet)
- a general-purpose emitter/receiver, typically a USB gateway, which includes a UART for asynchronous serial communication with an integrated RF module

For that, popular USB dongles can be purchased, which often rely on the TCM 310 chip; this includes the USB300 one (around 37 Euros in France), or the USB310 one (around 50 Euros in France) that we prefer, as it features a SMA connector, which allows an external antenna to be connected in order to boost emission / reception ranges inexpensively.

We will rely here on such a configuration.

Operating System Support

Once the USB dongle is connected (here on an Arch Linux host), `lsusb` tells us that it is detected as:

```
Bus 003 Device 009: ID 0403:6001 Future Technology Devices International, Ltd FT232 Se
```

(which applies both to USB300 and USB310)

We will interact with this USB gateway as if it was a serial port.

Rather than having it designated by an obscure, potentially changing name (like `/dev/ttyUSB0`, `/dev/ttyUSB1`, etc.), we prefer assigning it a fixed, clearer, well-chosen path, like `/dev/ttyUSBEnOcean`.

For that, one may define a suitable udev rule, typically stored in `/etc/udev/rules.d/99-enocean.rules`, whose content can simply[1] be:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", SYMLINK+="ttyUSBE
```

Following extra option could be added to the previous line, in order to set the group of this TTY: `GROUP="dialout"` or `GROUP="uucp"` (depending on the system's conventions), in which case your user shall be in that group (rather than executing `sudo chmod 777 /dev/ttyUSB0` each time the USB dongle is inserted for example).

So one may prefer:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", SYMLINK+="ttyUSBE
```

and, to ensure that the user of interest for Oceanic (let's name it `stallone`) belongs to that group:

```
$ sudo usermod -a -G uucp stallone
```

[1] Note though the different roles played by == (for matching) and = (for assignment).

One may then run `sudo udevadm control --reload-rules && sudo udevadm trigger` to ensure that these changes are taken into account from now on.

Then inserting said USB dongle should generate log entries that `journalctl -xe` can show, like (timestamps and hostname edited):

```
kernel: usb 3-11: new full-speed USB device number 9 using xhci_hcd
kernel: usb 3-11: New USB device found, idVendor=0403, idProduct=6001, bcdDevice= 6.00
kernel: usb 3-11: New USB device strings: Mfr=1, Product=2, SerialNumber=3
kernel: usb 3-11: Product: FT232R USB UART
kernel: usb 3-11: Manufacturer: FTDI
kernel: usb 3-11: SerialNumber: A600AVJD
mtp-probe[74533]: checking bus 3, device 9: "/sys/devices/pci0000:00/0000:00:14.0/usb3
kernel: ftdi_sio 3-11:1.0: FTDI USB Serial Device converter detected
kernel: usb 3-11: Detected FT232RL
kernel: usb 3-11: FTDI USB Serial Device converter now attached to ttyUSB0
mtp-probe[74533]: bus: 3, device: 9 was not an MTP device
mtp-probe[74548]: checking bus 3, device 9: "/sys/devices/pci0000:00/0000:00:14.0/usb3
mtp-probe[74548]: bus: 3, device: 9 was not an MTP device
```

On insertion we have then, with the former settings:

```
$ ls -l /dev/ttyUSBEnOcean /dev/ttyUSB0
 crw-rw---- 1 root uucp 188, 0 Nov 13 10:24 /dev/ttyUSB0
 lrwxrwxrwx 1 root root      7 Nov 13 10:24 /dev/ttyUSBEnOcean -> ttyUSB0
```

Software Prerequisites

Ceylan-Oceanic relies on general-purpose services offered by Ceylan-Myriad (implying of course Erlang itself), and on a suitable Erlang driver for serial communication.

Erlang

If needed, follow these Myriad guidelines for installing Erlang in order to obtain a proper, recent-enough version thereof.

Serial

We use our version[2] of erlang-serial for that; we generally prefer, for standalone use, installing it in user space (rather than in the system tree and embedded in a release) that way:

```
$ mkdir ~/Software && cd ~/Software
$ git clone https://github.com/Olivier-Boudeville/erlang-serial
$ cd erlang-serial
$ make && DESTDIR=. make install
```

[2]This is a fork of the original erlang-serial, which had to be modified notably in terms of disabled RTS/CTS flow control, in order to be able to properly send data to the Enocean gateway.

Then using `erlang-serial` will be just a matter of adding it to one's code path[3].

To test this erlang-serial installation (whether or not any dongle is connected):

```
$ erl -pa $HOME/Software/erlang-serial/erlang/lib/serial-1.1/ebin
Erlang/OTP 25 [erts-13.0] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [

Eshell V13.0  (abort with ^G)
1> serial:start().
<0.82.0>
```

Perfect!

Ceylan-Myriad

Oceanic expects to find a fully-built Myriad source tree as a sibling of its own tree, named `myriad`, and possibly made available through a symbolic link.

As per these Myriad guidelines, this source tree can be obtained by changing to a directory of choice that will contain both Myriad and Oceanic, and issuing:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad.git
$ ln -s Ceylan-Myriad myriad && cd myriad && make all && cd ..
```

Ceylan-Oceanic

From the same parent directory, very similarly:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Oceanic.git
# Symlink just for consistency:
$ ln -s Ceylan-Oceanic oceanic && cd oceanic && make all && cd ..
```

Testing Enocean

Ensure first that none of the next serial tools / terminals has been left running, otherwise exclusive access may block your ability to send telegrams thanks to Oceanic.

To check, one may rely on:

```
$ lsof /dev/ttyUSBEnOcean
COMMAND    PID      USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
serial  214977 your_user   3u   CHR  188,0      0t0 1066 /dev/ttyUSB0
```

Note also that, from that point, EURIDs are altered/edited (fake ones used). Minor discrepancies may happen.

[3] Later in the installation one may update the `Erlang-serial section` in Oceanic's GNU-makevars.inc in order to take into account any other path convention. One may then run, from the root of Oceanic, `make info-serial` to check that `ERLANG_SERIAL_BASE` points indeed to a directory containing erlang-serial's `ebin` directory. Otherwise runtime checks will detect and report any issue.

Basic, Direct Command-line Testing

It is as simple as executing from the command-line (thus without Oceanic, Serial or even Erlang being involved):

```
$ od -x < /dev/ttyUSBEnOcean
0000000 0055 0707 7a01 10f6 2e00 96e1 0130 ffff
0000020 ffff 0039 554b 0700 0107 f67a 0000 e12e
```

(of course for such a binary content to be received, Enocean telegrams must be emitted; the simplest approach is to trigger any Enocean device able to send on demand such telegrams, like a button/rocker/switch)

`hexdump` can be also used to intercept telegrams. If needing to set the transmission speed beforehand, use `stty -F /dev/ttyUSBEnOcean 57600`.

Incoming data can also be recorded and "replayed" (yet this is not expected to activate an Enocean receiver, see Protocol Information):

```
$ cat < /dev/ttyUSBEnOcean > my_record.bin
$ cat my_record.bin > /dev/ttyUSBEnOcean
```

With a Graphical Serial Terminal

One may use cutecom to directly test input/output telegrams.

A priori neither RTS nor DTR shall be enabled (yet in our tests these had no impact with cutecom; however not disabling them with Oceanic was leading to emitting telegrams not understood by their target devices).

We recommend using the `Hex` input and output.

Configuring Oceanic

Oceanic can be configured thanks to a set of key/value pairs:

- `oceanic_emitter :: oceanic:eurid_string()` (e.g. `{oceanic_emitter, "002EE196"}`) in order to specify the EURID of the pseudo-device emitting any telegram to be sent by Oceanic; otherwise a default value applies (see `default_emitter_eurid`); anyway, unless specified otherwise, Oceanic will switch to the base ID of the Enocean USB dongle once it has been fetched (through a Common Command, at Oceanic start-up)

- `oceanic_jamming_threshold :: system_utils:bytes_per_second()` (e.g. `{oceanic_jamming_threshold, 200}`) specifies the threshold of Enocean incoming traffic, in bytes per second, above which an `onEnoceanJamming` message shall be sent by Oceanic to any user listener process; considering an usual Enocean telegram size of 21 bytes, the default threshold (see `default_jamming_threshold`), 250 bytes per second, corresponds roughly to a threshold of a dozen legit telegrams per second

- `oceanic_devices :: [oceanic:device_config()]` allows to tell Oceanic about the Enocean devices of interest, by specifying for each of them at least the following `{UserDefinedName, EURIDStr, EEPStr}` triplet:

- a user-friendly name for that device: `UserDefinedName :: ustring()`, like `"My opening detector at the front door"`
- its EURID: `EURIDStr :: oceanic:eurid_string()`, like `"0585E962"`
- the EEP that it implements: `EEPStr :: ustring()`, like `"D5-00-01"`

Extra information can be added in this tuple:

- first, its expected activity periodicity (the average rhythm at which this device is expected to emit telegrams, to detect if for any reason it happens to become missing in action), `ActPeriod :: oceanic:declared_device_activity_per` which may be `none` (typically if the device emits no spontaneous state feedback), `default` (to apply a reasonable default), a DHMS value (Days/Hours/Minutes/Seconds, like `{0,2,15,0}`; see `time_utils:dhms_duration()`), or `learn` (to be automatically determined by Oceanic over time; the default setting)
- then a mere, optional comment, `CommentStr :: ustring()`, like `"Just a switch, not a rocker; controls the smart plug in the living room"`; only of interest for the user, for configuration purposes (ignored by Oceanic)

Examples of device configurations (see also the Oceanic settings complete example below):

- `{"My thermo-hygro sensor", "050533EF", "A5-04-01"}`
- `{"My Foo opening detector", "A18EE2B1", "D5-00-01", {0,0,25,0}}`
- `{"My outdoor double-rocker switch", "002B6B15", "F6-02-01", learn, "Presumably hidden in the well"}`

These configuration information can be provided either thanks to the Ceylan-Myriad preferences system (see the `preferences` module; this usually boils down to an ETF file, by default `~/.ceylan-settings.etf`) and/or programmatically (see the `oceanic:add_configuration_settings/2`; in which case these settings would take priority over any set preferences).

Note that, should a device have a repeater ability activated, Oceanic may detect also the Enocean gateway it is using (as it may then receive back its own sendings).

Testing Oceanic

First test: executing a few Common Commands

This consists in having Oceanic discuss with the local USB gateway dongle, regardless of any actual Enocean device.

From the root of the Ceylan-Oceanic clone, supposing that Myriad and erlang-serial are already available and built (whereas here debug flags have been activated, see Oceanic's `GNUmakevars.inc`):

```
# Ensure erlang-serial is available:
$ make info-serial
ERLANG_SERIAL_BASE = /home/stallone/Software/erlang-serial/erlang/lib/serial-1.1

# Ensure that Ceylan-Oceanic is built:
$ make all

$ cd test

# Triggering a Common Command does not need any target device:
$ make oceanic_common_command_run

       Running unitary test oceanic_common_command_run (third form) from oceanic_commo

--> Testing module oceanic_common_command_test.

Testing the management of Common Commands.
[debug] Using TTY '/dev/ttyUSBEnOcean' to connect to Enocean gateway, corresponding to
[debug] Discovering our base EURID.
[debug] Sending to serial server <0.86.0> actual telegram <<85,0,1,0,5,112,8,56>> (hex
[debug] Waiting initial base request (ToSkipLen=0, AccChunk=<<>>).
[debug] Read telegram <<85,0,5,1,2,219,0,255,162,223,0,10,180>> of size 13 bytes (corr
[debug] Trying to decode '<<85,0,5,1,2,219,0,255,162,223,0,10,180>>' (of size 13 bytes
[debug] Start byte found, retaining now following chunk (of size 12 bytes; after dropp
 <<0,5,1,2,219,0,255,162,223,0,10,180>>.
[debug] Examining now following chunk of 12 bytes:<<0,5,1,2,219,0,255,162,223,0,10,180
[debug] Packet type 2; expecting 5 bytes of data, then 1 of optional data; checking fi
[debug] Header CRC validated (219).
[debug] Detected packet type: response_type.
[debug] Full-data CRC validated (180).
[debug] Decoding a command response, whereas awaiting command of type co_rd_idbase, ba
(corresponding to hexadecimal '5500010005700838'), on behalf of requester internal.
[debug] Returning the following internal response: read gateway base ID ffa3df00, for
[debug] Successfully read gateway base ID ffa3df00, for 10 remaining write cycles.
[info] No preferences file ('/home/stallone/.ceylan-settings.etf') found.
[debug] Waiting for any message including a telegram chunk, whereas having 0 bytes to
[debug] Requested to execute common command 'co_rd_version', on behalf of requester <0
[...]
[debug] Sending back to requester <0.9.0> the following response: read application
version 2.11.1.0, API version 2.6.3.0, chip ID 19d46ce, chip version 1162805507 and ap
[debug] Waiting for any message including a telegram chunk, whereas having 0 bytes to
Read version: read application version 2.11.1.0, API version 2.6.3.0, chip ID 19d46bc
[debug] Requested to execute common command 'co_rd_sys_log', on behalf of requester <0
[...]
Read logs: read counters: 6 for application: [254,255,255,255,255,255], and 38 for API
255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,25
[debug] Stopping the Oceanic server <0.85.0>.
[debug] Stopping serial server <0.86.0>, while in following state: Oceanic server usin
not having any command pending, based on a time-out of 1 second, with no command queue
having <0.9.0> registered as listener of Enocean events, having sent 3 telegrams, not
```

```
[debug] Oceanic server <0.85.0> terminated.
Stopped.

--> Successful end of test.

(test finished, interpreter halted)
```

Second test: controlling an actual device

This more complete test will rely on experimental settings typically involving:

- a **controller device** (e.g. a double-rocker switch), which will be, once discovered, spoofed next by Oceanic
- a **target device** (e.g. a smart plug / socket switching actuator) that already learnt - according to its own procedure (typically pressing adequately buttons thereof) - the previous controller device; for example a lamp would be plugged on that actuator so that, when pressing and releasing a given button of the rocker switch, that lamp is toggled (on/off)

The objective is to control that lamp programmatically, through Oceanic (only).

First, the EURID of the controller device must be determined. Either it can be directly read from some actual label on the device, or it has to be obtained through passive listening.

In this last case, start by running the following test (still in `oceanic/test`):

```
$ make oceanic_integration_run

       Running unitary test oceanic_integration_run (third form) from oceanic_integrat

--> Testing module oceanic_integration_test.

(test waiting indefinitely for Enocean events; hit CTRL-C to stop)
[debug] Using TTY '/dev/ttyUSBEnOcean' to connect to Enocean gateway, corresponding to
[debug] Discovering our base EURID.
[...]
[debug] Waiting for any message including a telegram chunk, whereas having no byte to
```

Then act on the controller so that it emits a telegram (e.g. press a button of said rocker switch; it may be correspond for example to the bottom position of the first rocker, `A`).

If in range, the test should intercept it:

```
[debug] Received a telegram chunk of 21 bytes: <<85,0,7,7,1,122,246,48,0,46,225,150,48
hexadecimal 55000707017af630002ef1963001ffffffff4400fe (whereas there are 0 bytes to s
[debug] Decoding an ERP1 radio packet of R-ORG f6, hence rorg_rps, i.e. 'RPS (Repeated
[info] Discovering Enocean device 002ef196 through failure.

<----------------
[warning] Unable to decode a RPS (F6) packet for 002ef196: device not configured, no l
```

```
---------------->

[debug] Waiting for any message including a telegram chunk, whereas having no byte to
[...]
```

(hit CTRL-C to stop)

So we determined that this rocker switch has for EURID `002ef196`.

We can notice that a failure is reported, as Oceanic cannot decode yet the telegrams from that emitter, short of knowing to which EEP it complies.

As this EEP information is not carried by such packets, it cannot be determined automatically and has thus to be specified, here once for all through a proper Oceanic configuration file, typically to be found as `~/.ceylan-settings.etf`.

In this ETF file, among possibly other entries unrelated to Oceanic, we may have:

```
% Oceanic section:

% Information regarding the pseudo-device emitting any telegram to be sent by
% Oceanic:
%
% (if overriding the base ID of this chip, read as "ffa3df00")
%
%{ oceanic_emitter, "DEADBEEF" }.

% Attempting to spoof my green switch:
%{ oceanic_emitter, "002EF196" }.

% Threshold, expressed in bytes per second, regarding the incoming Enocean
% traffic (received telegrams), in order to trigger onEnoceanJamming events
% should that threshold be exceeded:
%
{ oceanic_jamming_threshold, 200 }.

% A list of device_config() entries, clearer with user-defined names than with
% only raw EURIDs:
%
{ oceanic_devices, [

   % Each device is to be described thanks to one of the three following
   % configuration terms:

   % { UserDefinedName :: ustring(), EURIDStr :: eurid_string(),
   %   EEPStr :: ustring() } % Then the activity periodicity will be learn

   % { UserDefinedName :: ustring(), EURIDStr :: eurid_string(),
   %   EEPStr :: ustring(), declared_device_activity_periodicity() }

   % { UserDefinedName :: ustring(), EURIDStr :: eurid_string(),
   %  EEPStr :: ustring(), ActPeriod :: declared_device_activity_periodicity(),
   %  Comment :: ustring() }.
```

```
% For the local gateway (useful to decode/check self-encoded telegrams):
{ "my local USB gateway", "ffa3df00", "F6-02-01" },

% Single-input contacts:
{ "my first opening sensor", "060533EC", "D5-00-01", learn, "Battery to be added (t
{ "my second opening sensor", "02959F62", "D5-00-01" },

% Temperature and humidity sensors:
{ "my only temperature and humidity sensor", "02A96926", "A5-04-01" },

% Switches:
{ "my green switch", "002EF196", "F6-02-01",
  "This is actually a single-rocker switch" },
{ "my white switch", "012F50D6", "F6-02-01" },

% In-wall modules:

% Expected period of about 25 minutes:
{ "my two-channel orange module", "06035E4A", "D2-01-12", { 0, 0, 25, 0 } },

% Socket switching actuators:
%{ "my smart plug", (unknown), (unknown) }

] }.
```

These entries are pretty self-explanatory and have already been described in Configuring Oceanic:

- with `oceanic_emitter` we define the EURID that shall be used by Oceanic whenever emitting (the default being its in-chip first base ID, as automatically determined thanks to a Common Command)
- with `oceanic_jamming_threshold` we can define our own level of incoming traffic above which Oceanic shall consider that an attempt of jamming is taking place
- with `oceanic_devices` the EEP of the various devices that we want to be aware of are listed (naming them allows to have clearer Oceanic reports)

Now, as the test explicitly sets the EURID of the emitter, it is just a matter of updating, in `oceanic_static_sending_test.erl`, the `SourceEurid` variable in order that this test impersonates the controller of interest (here, said green switch):

```
SourceEurid = oceanic:string_to_eurid( "002EF196" ),
```

Running it[4] results in:

[4]The decoding printout corresponds to a check made by this test: prior to sending a telegram that it just generated, it ensures that it can decode it successfully.

```
$ make oceanic_static_sending_run
       Running unitary test oceanic_static_sending_run (third form) from oceanic_stati

--> Testing module oceanic_static_sending_test.

Starting test; note that direct telegram sendings are made here, thus Oceanic will det
[debug] Using TTY '/dev/ttyUSBEnOcean' to connect to Enocean gateway, corresponding to
[debug] Discovering our base EURID.

[debug] Successfully read gateway base ID ffa3df00, for 10 remaining write cycles.
[debug] Initial state: Oceanic server using serial server <0.86.0>, using emitter EURI
on a time-out of 1 second, with no command queued whereas none has been issued; not ha
having sent a single telegram, not having discarded any telegram, and knowing 8 Enocea
+ device 'my first opening sensor' (EURID: 060533ec) applying EEP D5-00-01; it has bee
[...]
Decoding the 'pressed' one for the 'off' button results in following event: double-roc
pressed simultaneously at 2022/11/19 23:11:45, declared with a single subtelegram, tar
level: telegram not processed; its EEP is double_rocker_switch (F6-02-01)
[...]
All telegrams of interest encoded.
First we press (and then also release) the 'switch off' button, 'button_ao' (which mus
Then, after a short waiting, we press (and then release) this 'switch off' button agai
[debug] Sending to serial server <0.86.0> actual telegram <<85,0,7,7,1,122,246,16,1,9,
(hexadecimal form: '55000707017af6100109d9702001fffffffff00cc').
```

The lamp is expected first to turn on, then, and after one second, to turn off.

Congratulations, your Oceanic program can control electrical appliances!

If this test does not work as intended:

- did the right position of the right button was learnt?
- depending on the switch, apparently:
  - either each of the individual buttons will act as a rocker by itself (e.g. to switch on then off the lamp, a learnt button - top or bottom - of a given rocker will have to be pressed and released twice[5])
  - or the whole rocker (that is the pair made of its top and bottom buttons) will work as intended as a rocker (e.g. to switch on the lamp, the top button will have to be pressed and released, then, to switch off the lamp, the bottom button will have to be pressed and released[6])

[5] This is the case for my white switch, an O2 Line Comfort double-rocker; the top and bottom buttons can then be used indifferently.

[6] This is the case for my green switch, a VIMAR Vita (single) rocker, for which each button has a role. For example, pressing a given button more than once will have no effect (as it corresponds to a state already reached), only using the other will trigger a new transition.

Oceanic Listener Messages

Any number of processes may register to the Oceanic server process, at startup (see `start/2` and `start_link/2`) and/or later (see the `addEventListener` / `removeEventListener` messages).

These processes will be notified by the Oceanic server of any Enocean event of interest, thanks to the following types of messages:

- on the **first detection** of a device:
  - if it was **declared** in the Oceanic configuration: `{onEnoceanConfiguredDeviceFirstSeen, [Event, OcSrvPid]}`, where `Event :: oceanic:device_event()`: corresponds to an Enocean telegram that was just intercepted by the gateway and decoded, and which was sent by a configured device that had never been detected before
  - if it was **not declared** in the Oceanic configuration:
    - if it is known through teach-in: `{onEnoceanDeviceTeachIn, [Event, OcSrvPid]}`, where `Event :: oceanic:device_event()`: corresponds to an Enocean teach-in telegram that was just intercepted by the gateway and decoded, and which was sent by a device sending that was not even listed in the configuration yet *requests* (some actuator) to be paired (so not meaning `onEnoceanDeviceTaughtIn`)
    - if it is known through passive listening: `{onEnoceanDeviceDiscovery, [Event, OcSrvPid]}`, where `Event :: oceanic:device_event()`: corresponds to a normal (non teach-in) Enocean telegram that was just intercepted by the gateway and decoded, and which was sent by a device that was not even listed in the configuration
- `{onEnoceanDeviceEvent, [Event, BackOnlineInfo, OcSrvPid]}`, where `Event :: oceanic:device_event()`: corresponds to an Enocean telegram that was just intercepted by the gateway and decoded, and which emanates from a device that has already been detected; if `BackOnlineInfo :: oceanic:back_online_info()` specifies whether this device was considered lost until now (and thus is back online, with some additional information)
- `{onEnoceanDeviceLost, [Eurid, DeviceName, IsNewLoss, LastSeen, PeriodicityMs, OcSrvPid]}`, where `LastSeen :: time_utils:timestamp()` and `PeriodicityMs :: unit_utils:milliseconds()`: notifies that the Oceanic server detected that a device that was expected to send spontaneous state updates apparently failed to do so on time (despite some reasonable time margin); could have run out of energy, be sabotaged, etc.; `IsNewLoss` tells whether this device just transitioned from online to lost, or if it was already considered lost; if its device is to reappear, it will be notified as being back online in the next `onEnoceanDeviceEvent` message
- `{onEnoceanJamming, [AggTrafficLvl, OcSrvPid]}`, where `AggTrafficLvl :: system_utils:bytes_per_second()`: reports that the Oceanic server detected a possible jamming attempt (abnormal traffic level of Enocean telegrams)

These message specify the PID of the Oceanic server (`OcSrvPid`) notably to support multiple instances thereof.

Enocean Documentation

Main reference pointers are:

- [ETS]: Enocean Technical Specifications, notably for:
  - [EEP-gen]: EnOcean Equipment Profiles (e.g. version 3.1.4, 36 pages), a short, general view onto the structure of the various telegram types that are available (e.g. the RPS one)
  - [EEP-spec]: EEP Specification (e.g. version 2.6.7, 270 pages), for a detailed specification of the various equipment profiles (e.g. `F6-01-*` being for *Switch Buttons*)
- [ESP3]: Enocean Serial Protocol (ESP3) - SPECIFICATION (e.g. version 1.51, 116 pages), a point-to-point packet-based protocol that is lower-level in the network stack; of lesser interest here)

Note also that, despite the availability of ERP2 specifications, at least most devices we are aware of rely on ERP1 ones.

Protocol Information

Guarding Against Spoofing: Lying about One's Source EURID Will Not Suffice

Provided that the serial link is properly configured (in terms of speed, parity, start/stop bits, RTS/CTS flow control, etc.), apparently even with the default, usual level of security (that is: none) implemented by the devices that we tested, Enocean telegrams could *not* be replayed[7]: just intercepting a raw telegram and re-emitting was not acknowledged by the target device and did not trigger its intended effect on at least our main test actuator (e.g. the smart plug did not switch on/off).

One explanation could have been that we were re-emitting from Oceanic "receive" telegrams (as opposed to "send" ones), as we actually always receive information different from what was sent (e.g. the dbM measure, the repeating count, etc. are visibly set between the emission and the receiving; and of course the checksums are modified accordingly) - so replaying a received telegram *could* be rejected on these bases.

Nevertheless, forging from scratch proper "send telegrams" (yet carrying the same functional information) and sending them by ourselves still did not trigger the actuator (we did multiple tests on multiple devices of different manufacturers).

So we believe that extra information is available to actuators through the Enocean network stack, that may/will be used by them in order to discriminate between actual emitters.

This was further confirmed by testing the same telegram exchanges after having learnt a device, either the real one, or one impersonated by Oceanic:

[7] See the `replay_telegrams/1` function in the `oceanic_just_send_to_device_test` module for an example.

apparently, only the ones that have been explicitly learnt previously will be accepted afterwards.

By forging telegrams bearing a source EURID different from the base one, we came to the conclusion that:

- most if not all telegrams carry a source EURID that can be freely set (typically through Oceanic calls)
- yet in parallel each emitter (be them an USB dongle controlled by Oceanic or a "real" device) has its own internal, "base" ID (or a base ID range, for such dongles); these IDs have the same type as EURIDs, and we suppose that they can be considered as actual EURIDs - yet they *could* be handled specifically only in low-level ESP3-like protocols (invisibly from the "applicative layer" seen when exchanging with the dongle); by default, unless specified (see the `oceanic_emitter` configuration entry), the source EURID used by telegrams generated by Oceanic match the ID obtained (through a Common Command) from the USB dongle[8]
- learning a device relies at least on these internal IDs, sometimes also on the specified in-telegram source one
- a telegram will be considered by an actuator iff the internal ID of the emitter carried by this telegram matches with one that has been learnt by the actuator (hence no easy spoofing with rogue, undeclared emitters)
- the source EURID included in a telegram will designate a device but may not match the internal ID of the emitter; so for example we could forge, from Oceanic, telegrams whose source EURID matches the one of an actual device (a rocker switch) - and therefore did not match the internal ID of the dongle - while nevertheless, *provided that the dongle had already been learnt by the actuator, typically thanks to a previous Oceanic sending*, we could operate the actuator programmatically (despite these telegrams having inconsistent IDs)
- yet, do these Oceanic telegrams have to specify the same EURID as used for their registering, or any already-registered EURID - or would any EURID would do the trick? According to our tests of two different plugs from different manufacturers, they will act differently depending on, when emitting the learning message, the specified source EURID matched or not the internal ID:
  - if yes, then afterwards the plug will respond to all telegrams sent by this emitter whose source EURID still matches the internal ID, and only to them (this is a bit unfortunate, as if having multiple plugs of that type, one cannot programmatically target them individually)
  - if no, then afterwards the plug will respond to all telegrams sent by this emitter, regardless of their source EURID - provided that the source EURID is *not* the internal ID (this is very strange that a single EURID gets rejected, but we tested multiple times); and, with this constraint and the previous one, we can thus control separately up to two of such plugs

- for the repeating mechanisms to have an interest, their re-emitted telegrams must be taken into account by the target actuators; so accepting already-sent telegrams emanating from different emitters than the one specified in the telegrams is needed; repeating is most probably handled transparently by lower-level protocols as well

We can also verify that devices like rocker switches are apparently stateless, in the sense that they seem to send the same information regardless of their history when one of their buttons is pressed (they have no memory).

So from our experiments we believe that, in terms of identification, the devices rely on a lower-level protocol (possibly ESP3) than the one that can be handled programmatically (e.g. ERP1 and siblings); as these operations seem to be done through the firmware of the USB gateway, spoofing Enocean traffic may be out of the reach of programs relying on "standard" USB gateways (therefore Oceanic having to be involved also in the learn process, not only in the emitting one).

And forging custom source EURIDs may have an interest, yet the spoofer must have been previously learnt - otherwise this would be a bit like if one was spoofing IP addresses in forged packets, whereas the target device would first compare MAC addresses.

Other Network-Related Risks

The spoofing risk being mostly alleviated, the only extra risks that we could foresee are:

- possibly **brute-force attempts** to match already-learnt base identifiers, from a debug gateway allowing to act on ESP3 packets (a threat that does not seem likely for common burglaries)
- the **jamming of an actuator** by saturating it with telegrams (be them well-formed and sensible, or not[9]), so that any actual telegram of interest (e.g. regarding a door opening) may not reach the receiver
- sensors devices being **incapacitated before they are able to raise an alarm** (for example destroyed, or possibly flashed by an electromagnetic impulse)

Oceanic provides basic yet possibly sufficient mechanisms guarding against these three threats.

For the first two risks: in a wireless context, nothing can be done against emission, but a configurable threshold in terms of incoming traffic volume can be monitored (with proper back-off), so that, if the application registered as

[8] This is merely a convention though, as apparently any another EURID could be used instead at this level. We used to add "*provided it is consistently used from then on*" (that is: when learning and also when sending telegram afterwards), yet we could see that even forging a telegram with a random source EURID but sending it from a right, already learnt device (hence using another EURID then) is sufficient to have the corresponding request accepted and processed - at least by some actuators.

[9] A battery-operated, generic-purpose jammer operating on the usual frequencies, like 868 MHz in Europe, may be able to affect most of the (now wireless) protocols for house-automation just by emitting powerfully-enough random noise on these bandwidths.

a listener, Oceanic notifies it whenever detecting such an attempt of denial of service - which can be considered by itself as a cause of alarm as serious as the other ones.

This threshold is expressed in bytes per second (knowing that telegrams are often fragmented), and its default value (see the `oceanic_jamming_threshold` configuration entry) is `250`. As the size of many Enocean legit telegrams is 21 bytes, an `onEnoceanJamming` event will be sent to the Oceanic-using application should a dozen of them be received during the same second, or a bit more in (a bit) longer time window (e.g. 20 in two seconds).

For the last risk, sensors (typically opening detectors) report instantly state transitions but also send periodic state notifications (even if no change happened). So a listener can monitor the duration elapsed since such a sensor was last seen, and if it exceeds a threshold (for example 30 minutes[10]), this may considered as a reason to raise an alarm.

For that, the user may request the activity periodicity of a device to be monitored.

The corresponding checking period (refer to the `declared_device_activity_periodicity/0` type) can be:

- directly specified by the user, as a DHMS duration; for instance `{0,0,12,0}` will denote an expected (maximum) duration of up to 12 minutes (0 Day, 0 Hour, 12 Minutes, 0 Second) between two telegrams sent by this device - otherwise the sensor will be reported as lost
- a default one (see `default_dhms_periodicity`), if the configuration specified `default`
- determined automatically by Oceanic, based on the rhythm that it could learn from past traffic emanating from this device, if the configuration specified `learn`, or did not set it

Whenever the checking period of a sensor elapses whereas no telegram from it has been received during that time frame, any process registered as an Oceanic event listener will be sent an `onEnoceanDeviceLost` message, specifying notably the name and EURID of that device, when it has been seen last, and what its expected periodicity was.

Studying Actual Protocols

To experiment and troubleshoot communication issues (this may be especially of use should different devices/actuators interpret differently the Enocean specifications / develop their own behaviour), one may also use tests that perform direct listening / emitting (possibly bypassing partly the logic of the Oceanic server):

- use `make oceanic_just_record_device_run` to display and record in file (`enocean-test-recording.etf`) all raw, timestamped telegrams that can be intercepted

[10] So the event will be detected, albeit with a latency that, depending on the use case, may or may not be acceptable / useful.

- use `make oceanic_just_send_to_device_run` to emit raw telegrams, typically recorded as explained above or forged (encoded) by Oceanic

Corresponding very handy scripts are available as well, `decode-telegram.sh` and `send-telegram.sh`, to which a raw telegram can be given (as an hexadecimal string).

Usage Hints

Good Practices

Before any new test, one should properly fully reset one's actuator, otherwise weird / wild / overly complex interpretations may happen.

Pairing

As mentioned, when using Oceanic as an emitter, it *must* have been paired to the target actuator.

Pairing can be done through teach-in (through an exchange of specific telegrams, triggered by the actuator) or through learning (putting the actuator in a specific mode, and forcing the emitter to send a telegram). Some actuators support both procedures.

As detailed in the next section, some actuators are able to learn a device according to various device types/EEPs (e.g. a rocker as a rocker, or as two push-buttons).

This choice matters: although this may not impact the telegrams to be sent by the device, it is bound to impact the behaviour of the actuator when receiving these telegrams.

Buttons vs Rocker: Transition vs State

Depending on the choice made by the user (typically as selected by pressing different buttons on the actuator, to enter a given learning mode), an actuator (e.g. a smart plug controlling a basic lamp) may learn a device (e.g. a rocker) differently (according to different EEPs).

For example a (single) rocker may be seen:

- case A: either as two independent push-buttons (a top button and an unrelated bottom one)
- case B: or as a whole rocker (hence two associated buttons together with the memory of the current state on the actuator)

A key difference is that, in case A (two push-buttons), each button taken individually may toggle the smart plug, while, in case B (rocker), pressing the top button whereas the smart plug is already passing (i.e. in the "triggered" state) will have no effect (e.g. the lamp remains on).

Said differently, case A is about toggling (forcing state transitions) while case B is about setting (forcing state values).

In the general case, setting (hence the rocker behaviour) may be seen as more reliable than toggling (the push-button behaviour): a given setting order may be sent multiple times to a rocker to ensure a given state is reached despite

a possible message loss, whereas a single loss of a toggling message will result in being consistently from then on in the opposite state of the intended one.

Another approach is to enable and manage state feedback / status return through confirmation telegrams about the current state of the actuator.

Regarding the Eltako Socket Switching Actuator FSSA

In practice, in addition to the documentation, we found clearer to respect the following procedures:

- to reset: press left-button for about 3 "large" seconds, then the LED blinks continuously, then press the right-button for about 5 seconds, until the LED turns off
- to learn a device as:
  - a push-button : press left-button for about 1 "large" second, then the LED is on continuously, then press the right-button shortly *once*; the LED will blink once and stay fixed until a telegram is received and learnt: if for example we pressed the top bottom to generate such telegram, this button will act as a on/off toggle, whereas its associated bottom button will have no effect
  - a "direction push-button" (supposedly a synonym of rocker): press left-button for about 1 "large" second, then the LED is on continuously, then press the right-button shortly *twice*; the LED will blink twice and stay fixed until a telegram is received and learnt; in that case, even if we pressed only for example the top button (which from now on corresponds to "set on"), the bottom one will also be taken into account (even if it was involved in the learning stage) and will correspond to "set off"

Afterwards, the LED will blink once a telegram of a learnt device is received (whether or not this specified action has been learnt).

As mentioned in the previous section, we prefer the "direction push-button" mode, i.e. the rocker-based, "state setting" mode.

We never managed to determine an EURID for that plug, and are not sure it has even one; same applies for any EEP.

In terms of behaviour:

- two options exist to control this plug: either with a broadcast telegram, or with a telegram targeting any (thus arbitrary) EURID (as mentioned, no EURID is known for that plug - and it will process a telegram regardless of the target EURID it comprises); we recommend the second, more selective approach, and to assign this plug a pseudo-EURID (not used by any other device), to better discriminate them
- when issuing a command (e.g switch on or switch off), generally we do our best to send a first telegram to denote the press of a rocker-like trigger, then, shortly after, a second one to denote the release of that trigger; however:

  - we first had better results in sending multipress events (telling that no button is then pressed) rather than release (reciprocal of press) ones (as at least some push buttons/rockers do)
  - then we observed that the release-like events are not necessary (e.g. with an emulated rocker, sending a "top button pressed" then a "bottom button pressed" event - hence with no release-like event for either - is sufficient to switch on then off that plug)

- when a switch on then switch off operation fails (it happens quite frequently):
  - the plug is almost always activated (e.g. a lamp is lit): only the switch off command fails
  - sending the switch off command multiple times seems then the best course of action

So the best course of action we determined could be (hence with a bogus, non-broadcast target EURID, and multipress rather than release event):

```
for e in top top bottom bottom bottom bottom; do m ${e}_press_eltako_run; done
```

Regarding the Nodon Smart Plug

These are `ASP-2-1-00` or `ASP-2-1-01`.

To learn a new controlling device, the (single) button of that plug shall be pushed for at least 3 seconds (its red LED starts to blink). The receiving of a telegram is supposed to grant the emitting device the right to control that plug.

Note though that, when forging a telegram so that a USB dongle is to control that plug, our tests showed that the incoming telegram will not be taken into account for learning if its target EURID is the one of that plug; only using the broadcast EURID allowed the plug to learn the emitter from these telegrams.

Afterwards, telegrams newly forged by the USB dongle were accepted and processed by the plug, whether they were broadcast or - better - targeting the specified EURID of that plug.

In terms of behaviour, this plug is simpler/clearer/more reliable:

- it can/should be targeted based on its EURID (better than using broadcast)
- we retained a multipress release (although a standard release might be as relevant)
- a smaller yet non-null probability of missing a telegram exists, and it does not seem to happen more for "switch off" than for "switch on"

So the best course of action we determined could be:

```
for e in top top top bottom bottom bottom; do m ${e}_press_nodon_run; done
```

Reliability

Regardless of the smart plug, the same test (same software - namely `oceanic_just_send_to_device_test`, on the same hardware) seems either to work flawlessly, or very erratically / unsatisfactorily.

Indeed, in some faulty, unspecified contexts, sending repeatedly a "switch on" order, then pausing, then sending a "switch off" order, results, with the:

- Eltako plug: in systematically switching on, and never off (as if at least most "switching off" telegrams were discarded)
- Nodon plug: in sometimes performing the expected transitions, sometimes not at all (erratically, as if some telegrams were missed - although as if only the "switching on" were be lost, the plug never remained on)

The cause for these abnormal behaviours is unknown, the best interpretation we have is that some telegram may be lost.

As the plugs remain correctly responsive with an actual device, it may be a problem with the USB dongle or its driver.

Troubleshooting

If an error report such as `Cannot open terminal /dev/ttyUSBEnOcean for read and write` is issued, this may be the sign that a past failed `serial` instance may linger and block the USB dongle.

To check and solve:

```
$ fuser /dev/ttyUSBEnOcean
/dev/ttyUSB2:        91437

$ ps -edf | grep 91437
bond+   91437   91408  0 Jan08 ttyUSB2  00:00:00 /home/bond/Software/erlang-serial/erl

$ kill 91437

$ fuser /dev/ttyUSBEnOcean
```

This should happen in the rare case of a prior crash.

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be reported to the project interface (typically issues) or directly at the email address mentioned at the beginning of this document.

Additional Information

- use `make sync-sources-to-server` if needing to update directly Oceanic' sources on a remote server that hosts an appropriate USB dongle
- refer to EnOcean in Practice (very clear information, in French)

Similar Projects

They may be used as sources of inspiration:

- [PY-EN] the rather complete Python EnOcean library, including for its EEP (XML) information
- a Java implementation: enocean4j
- the (Java) OpenEnocean openHAB binding
- a first Rust implementation

Finally, there are nice, interesting integrated solutions like Jeedom that are mostly open-source (yet the support for Enocean may require closed-source plugins to be bought).

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Ending Word

Have fun with Ceylan-Oceanic!