EasyNMEA Documentation

Release 0.1.0

Eduardo Ponz

Oct 22, 2021

CONTENTS

CHAPTER

ONE

GETTING STARTED

Before doing anything else, you can get a flavor of the *EasyNMEA* capabilities by checking out the easynmea Docker image for Ubuntu. This image ships an already built *EasyNMEA* with compiled examples that you can use to get some readings out of your NMEA sensor without building anything on your side. If you do not have the Docker Engine already installed, you can install it following this tutorial. Then, there are two options for running the container:

1.1 Run docker knowing the specific serial device

If your NMEA module is already connected to a serial port and it is not going to be unplugged, then you can just share that device with the container:

```
docker run -it --device=<path_to_device> eduponz/easynmea bash
```

Then, inside the container, you can run the GPGGA example with:

```
/root/easynmea/build/examples/gpgga_example -b <baudrate> -p <path_to_device>
```

1.2 Run docker allowing for plug-unplug connectivity

If your module may be unplug and plug while the container is running, you can still share the serial port with the Docker container by sharing all the devices of the same cgroup. Plug your device and get its cgroup with:

```
ls -l <path_to_device> | awk '{print substr($5, 1, length($5)-1)}'
```

Then, run the container:

```
docker run -it -v /dev:/dev --device-cgroup-rule='c <device cgroup>:* rmw' eduponz/
↪easynmea bash
```

Finally, inside the container, you can run the GPGGA example as before:

```
/root/easynmea/build/examples/gpgga_example -b <baudrate> -p <path_to_device>
```

CHAPTER

TWO

INSTALLATION

EasyNMEA is a cross-platform C++ library built and installed using CMake. In this guide, you can find instructions on how to build and install the library in different platforms, as well as how to build the documentation, and what configuration options can be applied at compilation time.

2.1 Build and Install on Ubuntu

This guide describes the process of building and installing *EasyNMEA* on Ubuntu platforms:

2.1.1 Prerequisites

To build and install *EasyNMEA*, some external tools are required.

```
sudo apt update && sudo apt install -y \
    cmake \
    g++ \
    wget \
    git \
    python3-pip
```

2.1.2 Dependencies

EasyNMEA depends on Asio, a cross-platform C++ library for network and low-level I/O programming that provides a consistent asynchronous model, which is used to interact with the serial ports. This can be installed with:

```
sudo apt update && sudo apt install -y libasio-dev
```

2.1.3 Build

Once the *prerequisites* and *dependencies* are installed, *EasyNMEA* can be built with the help of CMake by running:

```
cd ~
git clone https://github.com/EduPonz/easynmea.git
cd easynmea
mkdir build && cd build
cmake ..
cmake --build .
```

Note: For more information about compilation options please refer to *CMake Options*.

2.1.4 Install

Once the library is built, in can be installed in a user specified directory with:

```
cd ~/easynmea/build
cmake .. -DCMAKE_INSTALL_PREFIX=<user-specified-dir>
cmake --build . --target install
```

Alternatively, it can also be installed system-wide with:

```
cd ~/easynmea/build
cmake ..
cmake --build . --target install
```

2.2 Build and Install on Windows

This guide describes the process of building and installing *EasyNMEA* on Windows platforms:

2.2.1 Prerequisites

To build and install *EasyNMEA*, some external tools are required.

- CMake
- Visual Studio
- Wget
- Git
- Chocolatey

- pip3

2.2.2 Dependencies

EasyNMEA depends on Asio, a cross-platform C++ library for network and low-level I/O programming that provides a consistent asynchronous model, which is used to interact with the serial ports. Chocolatey can be used to install Asio on Windows platforms. Download the package and run:

```
choco install -y -s <download_dir> asio
```

Where `<download_dir>` is the directory into which the package has been downloaded.

2.2.3 Build

Once the *prerequisites* and *dependencies* are installed, *EasyNMEA* can be built with CMake by running:

```
cd ~
git clone https://github.com/EduPonz/easynmea.git
cd easynmea
mkdir build && cd build
cmake ..
cmake --build .
```

Note: For more information about compilation options please refer to *CMake Options*.

2.2.4 Install

Once the library is built, in can be installed in a user specified directory with:

```
cd ~/easynmea/build
cmake .. -DCMAKE_INSTALL_PREFIX=<user-specified-dir>
cmake --build . --target install
```

Alternatively, it can also be installed system-wide with:

```
cd ~/easynmea/build
cmake ..
cmake --build . --target install
```

2.3 Build and Install Documentation

Important: This guide assumes that the library as been built following the steps outlined in *Build and Install on Ubuntu*. Else, paths might need to be adjusted to align with the followed procedure.

EasyNMEA's documentation is comprised of Doxygen and Sphinx HTML output. The process of building the documentation entails installation of additional tools for both the Doxygen and Sphinx documentations.

2.3.1 Environment Setup

To ease the development process, and to avoid version incompatibilities or clashes, this guide describes the process of building the documentation using Python3 Virtual Environments. Before setting up the environment, *Doxygen* needs to be installed. Install *venv* and *Doxygen*, and create a virtual environment and install the necessary tools with:

```
cd ~
sudo apt update && sudo apt install -y python3-venv doxygen plantuml
python3 -m venv easynmea_venv
source easynmea_venv/bin/activate
pip3 install -r ~/easynmea/docs/requirements.txt
```

2.3.2 Build

After *setting up the environment*, the documentation can be built with:

```
source ~/easynmea_venv/bin/activate
cd ~/easynmea/build
cmake .. -DBUILD_DOCUMENTATION=ON -DCMAKE_INSTALL_PREFIX=<user-specified-dir>
cmake --build .
```

2.3.3 Install

After *building the documentation*, it can be installed with:

```
source ~/easynmea_venv/bin/activate
cd ~/easynmea/build
cmake --build . --target install
```

2.3.4 Simulate Read The Docs Build

To simulate the process followed on the *Read The Docs <https://readthedocs.org/>* to build this documentation, run:

```
source ~/easynmea_venv/bin/activate
cd ~/easynmea
rm -rf build  # Just in case
READTHEDOCS=True sphinx-build \
    -b html \
    -D breathe_projects.easynmea=<abs_path_to_docs_repo>/build/docs/doxygen/xml \
    -d <abs_path_to_docs_repo>/build/docs/doctrees \
    docs <abs_path_to_docs_repo>/build/docs/sphinx/html
```

2.4 CMake Options

EasyNMEA provides several CMake options that can be used to build or exclude certain library modules.

Option	Description	Possible values	Default
BUILD_DOCUMENTATION	Generates Doxygen and Sphinx documentation (see *Build and Install Documentation*)	ON | OFF	OFF
BUILD_LIBRARY_TESTS	Build the library tests.	ON OFF	OFF
BUILD_DOCUMENTATION_TESTS	Build the library documentation tests. Setting this ON will set BUILD_DOCUMENTATION to ON	ON OFF	OFF
BUILD_TESTS	Build the library and documentation tests. Setting this ON will set BUILD_LIBRARY_TESTS and BUILD_DOCUMENTATION_TESTS to ON	ON OFF	OFF
BUILD_EXAMPLES	Builds *EasyNMEA* examples	ON | OFF	OFF
GCC_CODE_COVERAGE	Build the library with code coverage support. This flag only take action when using GCC.	ON OFF	OFF

CHAPTER

THREE

USAGE

EasyNMEA provides the *EasyNmea* class, which uses NMEA 0183 sentences to extract **NMEA** information from the NMEA devices. It provides an easy-to-use API with which applications can open a serial communication channel with the NMEA device, wait until some data from one or more NMEA 0183 sentences arrives, retrieve it and digest it in an understandable manner, and close the connection.

The following snippet shows how to use *EasyNmea::open()*, *EasyNmea::wait_for_data()*, *EasyNmea::take_next()*, and *EasyNmea::close()* APIs to wait until *GPGGAData* data is received, using a *NMEA0183DataKindMask* set to *NMEA0183DataKind::GPGGA*. For more information about the supported NMEA 0183 sentences and their meaning, please refer to *NMEA 0183 Data Types*.

```
using namespace eduponz::easynmea;
// Create an EasyNmea object
EasyNmea easynmea;
// Open the serial port
if (easynmea.open("/dev/ttyACM0", 9600) == ReturnCode::RETURN_CODE_OK)
{
    // Create a mask to only wait on data from specific NMEA 0183 sentences
    NMEA0183DataKindMask data_kind_mask = NMEA0183DataKind::GPGGA;
    // This call will block until some data of any of the kinds specified in the mask␣
↪is
    // available.
    while (easynmea.wait_for_data(data_kind_mask) == ReturnCode::RETURN_CODE_OK)
    {
        // Take all the available data samples of type GPGGA
        GPGGAData gpgga_data;
        while (easynmea.take_next(gpgga_data) == ReturnCode::RETURN_CODE_OK)
        {
            // Do something with the GNSS data
            std::cout << "GNSS position: (" << gpgga_data.latitude << "; "
                      << gpgga_data.longitude << ")" << std::endl;
        }
    }
}
// Close the serial connection
easynmea.close();
```

CHAPTER

FOUR

NMEA 0183 DATA TYPES

This section presents the data types associated with the NMEA 0183 sentences that are interpreted by *EasyNMEA*.

4.1 GPGGA

The `GPGGAData` provides **Global Positioning System Fix Data**, meaning that it is advertised only when the GNSS device has been able to acquire a fix. The `GPGGAData` provides information about:

- **Timestamp**; always in *hhmmss.milliseconds*.
- **Latitude**; always in degrees referred to North.
- **Longitude**; always in degrees referred to East.
- **Fix**: whether there is a fix position. 0 means no fix, 1 means fix, and 2 means differential fix.
- **Satellites on view**: Number of satellites that the GNSS device can see.
- **Horizontal precision**; always in meters.
- **Altitude over sea level**; always in meters.

CHAPTER

FIVE

BUILD AND RUN EXAMPLES

This page presents how to build and run all the EasyNMEA examples, as well as showcasing sample outputs.

5.1 Build Examples

Note: This section assumes that the guides outlined in *Installation* have been followed.

Building the *EasyNMEA* examples is as easy as add the CMake option `-DBUILD_EXAMPLES=ON` on CMake's configuration step:

```
cd ~/easynmea/build
cmake .. -DCMAKE_INSTALL_PREFIX=<user-specified-dir> -DBUILD_EXAMPLES=ON
cmake --build . --target install
```

5.2 GPGGA Example

The GPGGA example showcases how to get Global Positioning System Fix Data out of GNSS devices, which they advertise using the NMEA 0183 GPGGA sentence. Once the examples *have been built*, the GPGGA example can be run with:

```
cd <user-specified-dir>/examples/bin
./gpgga_example --serial_port /dev/ttyACM0 --baudrate 9600
```

An output example from *gpgga_example* would be:

```
Serial port '/dev/ttyACM0' opened. Baudrate: 9600
Please press CTRL-C to stop the example

************** NEW GPGGA SAMPLE **************
Elapsed time ---------> 3468
-----------------------------------------
GPGGA Data - GNSS Position Fix
==============================
Message --------------> $GPGGA,072706.000,5703.1740,N,00954.9459,E,1,8,1.28,-21.2,M,
↪42.5,M,,*4E
Timestamp ------------> 72706
Latitude -------------> 57.0529° N
Longitude ------------> 9.91576° E
```

(continues on next page)

(continued from previous page)

```
Fix ------------------> 1
Number of satellites -> 8
Horizontal precision -> 1.28
Altitude -------------> -21.2
```

CHAPTER

SIX

API REFERENCE

This sections constitutes a detailed description of *EasyNMEA* public API.

6.1 EasyNmea

class eduponz::easynmea::**EasyNmea**

This class provides an interface with NMEA modules using NMEA 0183 protocol over serial connections.

It can be used to:

- Open and close serial connection with the modules.
- Wait for specific NMEA sentences to be received.
- Read incoming NMEA data in a parsed and understandable manner.

Public Functions

EasyNmea() **noexcept**

Default constructor. Constructs a *EasyNmea*.

~EasyNmea() **noexcept**

Virtual default destructor.

ReturnCode **open**(**const** char **serial_port*, long *baudrate*) **noexcept**

Open a serial connection.

It opens a serial connection on a given port with a given baudrate; given that the connection was not previously opened.

Pre The *EasyNmea* does not have any serial port opened. That is, either it is the first call to *open()*, or *close()* has been called before *open()*.

Return *open()* can return:

- *ReturnCode::RETURN_CODE_OK* if the port is opened correctly.
- *ReturnCode::RETURN_CODE_ERROR* is the port could not be opened.
- *ReturnCode::RETURN_CODE_ILLEGAL_OPERATION* if a previous call to open was performed in the same *EasyNmea* instance, regardless of the port.

Parameters

- [in] serial_port: A string containing the serial port name.
- [in] baudrate: The communication baudrate.

bool **is_open**() **noexcept**
Check whether a serial connection is opened

Return true if there is an opened serial connection; false otherwise.

ReturnCode **close**() **noexcept**
Close a serial connection

Pre A successful call to *open()* has been performed.

Return *close()* can return:

- *ReturnCode::RETURN_CODE_OK* if the connection was successfully closed.
- *ReturnCode::RETURN_CODE_ERROR* if the connection could not be closed.
- *ReturnCode::RETURN_CODE_ILLEGAL_OPERATION* if there was not open connection.

ReturnCode **take_next**(*GPGGAData* &*gpgga*) **noexcept**
Take the next untaken GPGGA data sample available.

EasyNmea stores up to the last 10 reported GPGGA data samples. *take_next()* is used to retrieve the oldest untaken GPGGA sample.

Return *take_next()* can return:

- *ReturnCode::RETURN_CODE_OK* if the operation succeeded.
- *ReturnCode::RETURN_CODE_NO_DATA* if there are not any untaken *GPGGAData* samples.

Parameters

- `[out] gpgga`: A *GPGGAData* instance which will be populated with the sample.

ReturnCode **wait_for_data**(*NMEA0183DataKindMask* *data_mask* = *NMEA0183DataKindMask*::all(), std::chrono::milliseconds *timeout* = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::hours(8760))) **noexcept**
Block the calling thread until there is data available.

Block the calling thread until data of the specified kind or kinds is available for the taking, or the timeout expires.

Return *wait_for_data()* can return:

- *ReturnCode::RETURN_CODE_OK* if a sample of any of the kinds specified in the mask has been received.
- *ReturnCode::RETURN_CODE_TIMEOUT* if the timeout was reached without receiving any data sample of the kinds specified in the `data_mask`.
- *ReturnCode::RETURN_CODE_ILLEGAL_OPERATION* if there was not open connection.
- *ReturnCode::RETURN_CODE_ERROR* if some other thread called *close()* on the *EasyNmea* instance, which unblocks any *wait_for_data()* calls.

Parameters

- `[in] data_mask`: A `NMEA0183DataKindMask` used to specify on which data kinds should the call return, thus unblocking the calling thread. When `wait_for_data` returns data_mask holds the types of data that have been received. Defaults to `NMEA0183DataKindMask::all()`.

- `[in] timeout`: The time in millisecond after which the function must return even when no data was received. Defaults to 8760 hours (1 year).

6.2 NMEA 0183 Data Types

6.2.1 NMEA0183Data

struct `eduponz::easynmea::`**NMEA0183Data**
Base struct for all NMEA 0183 Data types.

Subclassed by *eduponz::easynmea::GPGGAData*

Public Functions

NMEA0183Data(*NMEA0183DataKind data_kind* = *NMEA0183DataKind*::*INVALID*) **noexcept**
Default constructor; it empty-initializes the struct

Parameters

- `[in] data_kind`: The `NMEA0183DataKind` of the data instance. Defaults to *NMEA0183DataKind::INVALID*

~NMEA0183Data() = default
Default virtual constructor.

bool **operator==**(**const** *NMEA0183Data* &*other*) **const noexcept**
Check whether a *NMEA0183Data* is equal to this one

Return true if equal; false otherwise

Parameters

- `[in] other`: A constant reference to the *NMEA0183Data* to compare with this one

bool **operator!=**(**const** *NMEA0183Data* &*other*) **const noexcept**
Check whether a *NMEA0183Data* is different from this one

Return true if different; false otherwise

Parameters

- `[in] other`: A constant reference to the *NMEA0183Data* to compare with this one

Public Members

NMEA0183DataKind **kind**
The NMEA0183DataKind of the data.

6.2.2 GPGGAData

struct eduponz::easynmea::**GPGGAData** : **public** eduponz::easynmea::*NMEA0183Data*

Struct for data from GPGGA sentences.

Public Functions

GPGGAData() **noexcept**

Default constructor; it empty-initializes the struct, setting `kind` to *NMEA0183DataKind::GPGGA*

bool **operator==**(**const** *GPGGAData* &*other*) **const noexcept**

Check whether a *GPGGAData* is equal to this one

Return true if equal; false otherwise

Parameters

- `[in] other`: A constant reference to the *GPGGAData* to compare with this one

bool **operator!=**(**const** *GPGGAData* &*other*) **const noexcept**

Check whether a *GPGGAData* is different from this one

Return true if different; false otherwise

Parameters

- `[in] other`: A constant reference to the *GPGGAData* to compare with this one

Public Members

float **timestamp**

UTC time hhmmss.milliseconds.

float **latitude**

Latitude in degrees referred to North.

float **longitude**

Longitude in degrees referred to East.

uint16_t **fix**

GNSS Fix

- 0: no fix
- 1 -> fix
- 2 -> differential fix

uint16_t **satellites_on_view**

Number of satellites on view.

float **horizontal_precision**

GNSS horizontal precision expressed in meters.

float **altitude**

GNSS reported altitude over sea level expressed in meters.

float **height_of_geoid**

Height of geoid above WGS84 ellipsoid in meters.

float **dgps_last_update**
Seconds since last DGPS update.

uint16_t **dgps_reference_station_id**
DGPS reference station ID.

6.3 Types

6.3.1 Bitmask

template<typename **E**>
class Bitmask
Generic bitmask for an enumerated type.

This class can be used as a companion bitmask of any enumerated type whose values have been constructed so that a single bit is set for each enum value. The enumerated values can be seen as the names of the bits in the bitmask.

Bitwise operations are defined between masks of the same type, between a mask and its companion enumeration, and between enumerated values.

```
enum my_enum
{
    RED    = 1 << 0,
    GREEN  = 1 << 1,
    BLUE   = 1 << 2
};

// Combine enumerated labels to create a mask
Bitmask<my_enum> yellow_mask = RED | GREEN;

// Combine a mask with a value to create a new mask
Bitmask<my_enum> white_mask = yellow_mask | BLUE;

// Flip all the bits in the mask
Bitmask<my_enum> black_mask = ~white_mask;

// Set a bit in the mask
black_mask.set(RED);

// Test if a bit is set in the mask
bool is_red = white_mask.is_set(RED);
```

Template Parameters

- `E`: The enumerated type for which the bitmask is constructed

6.3.2 NMEA0183DataKind

enum eduponz::easynmea::**NMEA0183DataKind**
Holds all the supported NMEA 0183 sentences.

Values:

enumerator INVALID = 0
Represents no valid data kind.

enumerator GPGGA = 1 << 0
Global Positioning System Fix Data.

6.3.3 NMEA0183DataKindMask

using eduponz::easynmea::**NMEA0183DataKindMask** = *Bitmask*<*NMEA0183DataKind*>
Bitmask of NMEA0183 datas.

Values of NMEA0183DataKind can be combined with the 'l' operator to build the mask:

```
NMEA0183DataKindMask mask = NMEA0183DataKind::GPGGA | NMEA0183DataKind::INVALID;
```

See *Bitmask*

6.3.4 ReturnCode

class eduponz::easynmea::**ReturnCode**
Provides understandable return codes for the different operations that the library performs.

These return codes can be easily compared for applications to handle different scenarios.

Public Types

enum [anonymous]
Internal *ReturnCode* enumeration.

Values:

enumerator RETURN_CODE_OK = 0
Operation succeeded.

enumerator RETURN_CODE_NO_DATA = 1
No data available.

enumerator RETURN_CODE_TIMEOUT = 2
Operation timed out.

enumerator RETURN_CODE_BAD_PARAMETER = 3
Bad input parameter to function call.

enumerator RETURN_CODE_ILLEGAL_OPERATION = 4
The operation is illegal.

enumerator RETURN_CODE_UNSUPPORTED = 5
The operation is not yet supported.

enumerator RETURN_CODE_ERROR = 6
The operation failed with an unexpected error.

Public Functions

ReturnCode()
Default constructor; construct a *ReturnCode* with value *ReturnCode::RETURN_CODE_OK*.

ReturnCode(uint32_t *e*)
Construct a return code from an integer representing the enum value.

bool **operator==**(**const** *ReturnCode* &*c*) **const**
Check whether a return code is equal to this one

Return true if equal; false otherwise

Parameters

- [in] c: A constant reference to the return code to compare with this one

bool **operator!=**(**const** *ReturnCode* &*c*) **const**
Check whether a return code is different from this one

Return true if not equal; false otherwise

Parameters

- [in] c: A constant reference to the return code to compare with this one

uint32_t **operator()**() **const**
Get the internal value of the *ReturnCode*

Return This *ReturnCode* internal value

bool **operator!**() **const**
Check whether this *ReturnCode* is equal to *ReturnCode::RETURN_CODE_OK*

Return true if this *ReturnCode* is different than *ReturnCode::RETURN_CODE_OK* ; false otherwise

CHAPTER

SEVEN

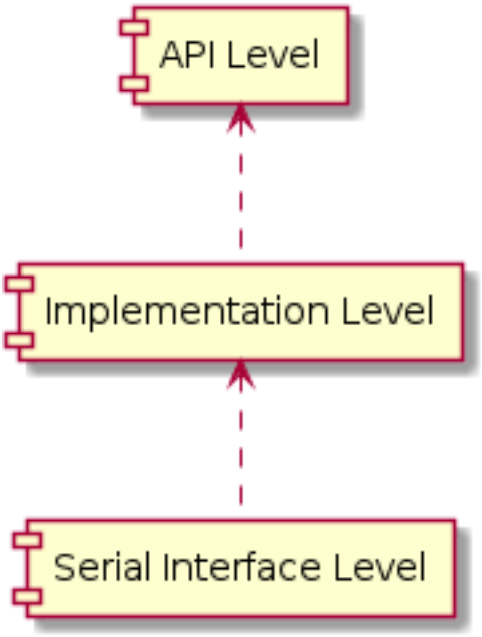
DEVELOPER DOCUMENTATION

This section contains all the design documents of *EasyNMEA*. It is meant to gather all technical documentation so that contributors to the project can understand the reasoning behind the current implementation, as well as document the designs for their contributions to the library. Please refer to the Contributing Guidelines if you are considering contributing to *EasyNMEA*.

7.1 Library Architecture

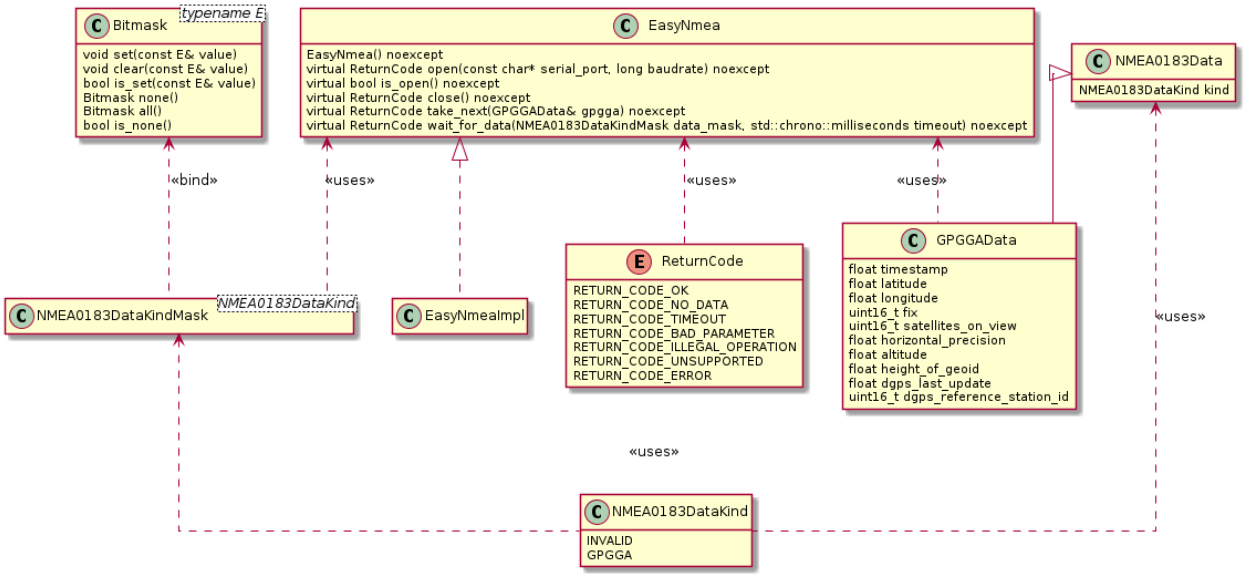
EasyNMEA is divided into three levels (from outer to inner):

1. *API Level* : This level contains all public API, i.e. the classes in the `include` directory.
2. *Implementation Level*: This level contains all the internal classes which provide functionality to the library.
3. *Serial Interface Level*: This level contains the classes for interacting with the serial port (through Asio).



7.1.1 API Level

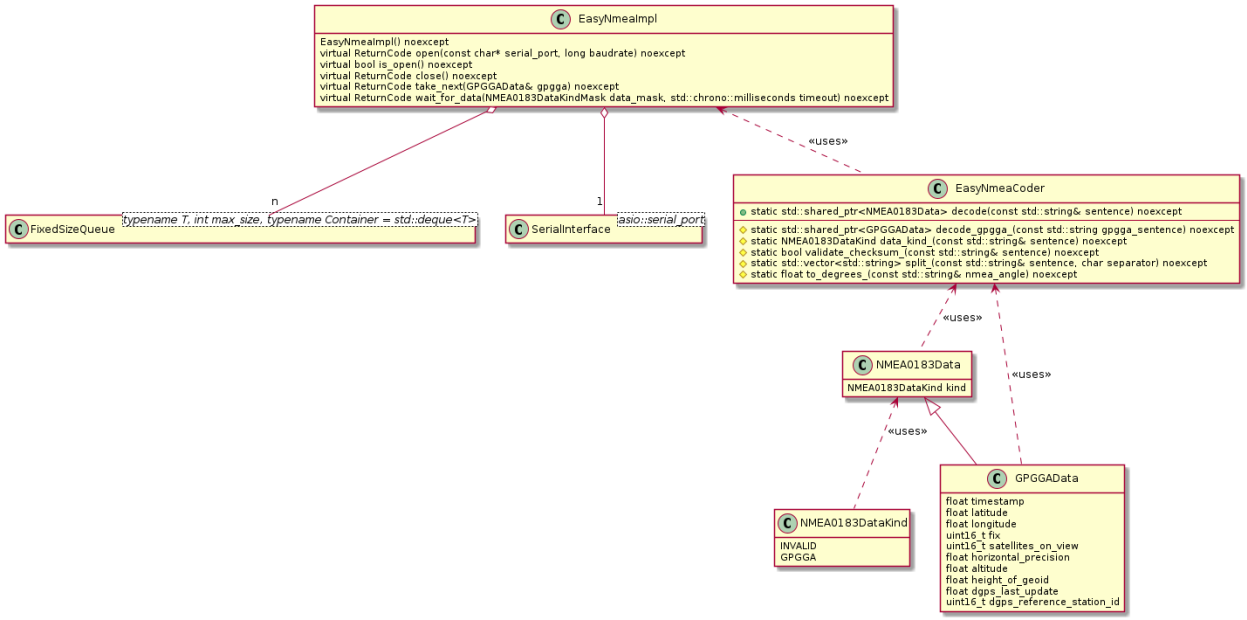
The API level comprises all the *EasyNMEA* public classes and structures, and acts as entry point for the library's functionalities. It consists of a main class *EasyNmea*, which provides application with access to the functionalities, and all the supporting classes and structures for return types and input and output parameters. Those companion classes and structures are *ReturnCode*, *GPGGAData*, and *NMEA0183DataKindMask*. For the actual functionality implementation, *EasyNmea* relies on the internal class EasyNmeaImpl.



7.1.2 Implementation Level

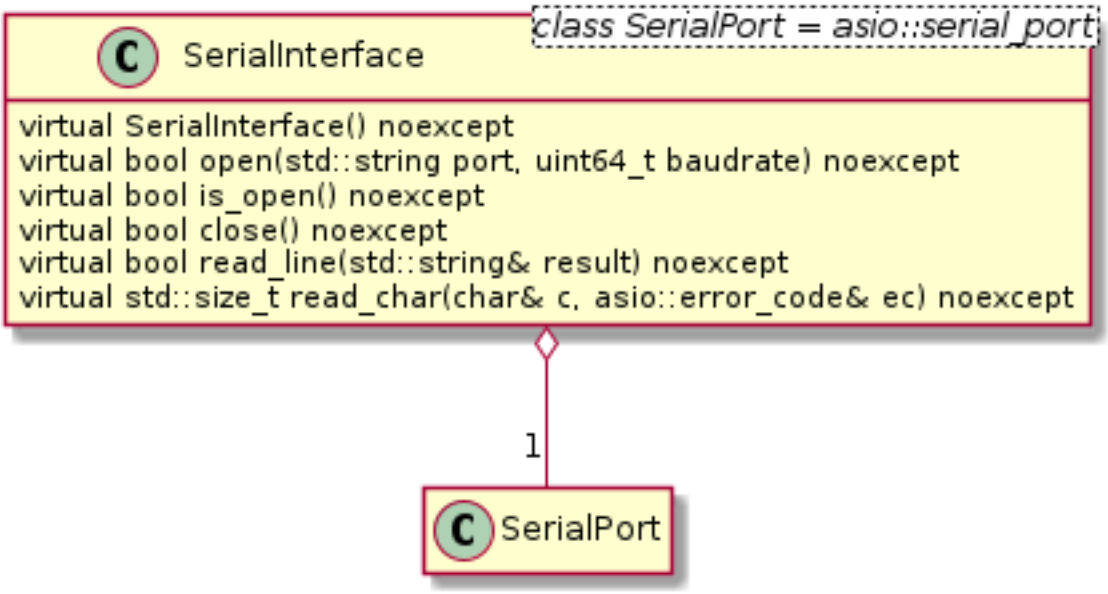
The implementation level comprises two main components:

1. The EasyNmeaImpl class, which provides with implementation for the *EasyNmea* public API, i.e opening and closing the serial port, waiting until data of one or more NMEA 0183 types has been received, checking whether the serial port connection is opened, and taking the next unread sample of a given NMEA 0183 type. The EasyNmeaImpl holds a FixedSizeQueue of ten elements for each supported NMEA 0183 type. This way, keeping outdated samples, as well as dynamic allocation of data samples, is avoided. The managing of the serial port is enabled through the SerialInterface class.
2. The EasyNmeaCoder class, which provides APIs for decode NMEA 0183 sentences (and to encode them in the future).



7.1.3 Serial Interface Level

The serial interface level is comprised of the `SerialInterface` class, which provides member functions to open and close a serial port, as well as for reading data from it. `SerialInterface` is a template class with a template parameter `SerialPort` that defines the serial port implementation, which defaults to :class: `asio::serial_port`.



7.2 Testing Infrastructure

This section documents the decisions made regarding the *EasyNMEA* testing infrastructure.

7.2.1 Testing Framework

The *EasyNMEA* testing framework has to cope with the following requirements:

1. Easy to integrate with CMake
2. Easy to integrate with GitHub actions
3. Large acceptance, so new contributors can write tests effortlessly
4. Mocking capabilities. This is because at least Asio will have to be mocked
5. Extense documentation
6. Easy to find answers to common problems.
7. Should be able to be used to create tests for the documentation

To satisfy these requirements *EasyNMEA* uses Gtest as testing framework. This decision is taken for a number or reasons:

1. Huge acceptance
2. Very large community, which means tons of Q&A everywhere
3. Very good documentation with examples
4. Out-of-the box mock support
5. Direct integration with CMake
6. GitHub integration merely consists on an action which installs GTest.

Other testing framework such as Catch and Boost.Test, however they were discarded:

- Catch seemed very promising, specially being a header only library, but the lack of mocking support is unfortunately a no-go for *EasyNMEA*.
- Boost.Test, which also offers a header only version, but again, it does not have built-in mocking support.

7.2.2 Build Tests

The *EasyNMEA* tests can be divided into two large categories:

1. **Library tests**: Unit and system tests for the *EasyNMEA* library itself.
2. **Documentation tests**: Automated tests for the documentation.

Although none of these tests are built by default, it is possible to build them separately. This is because not everyone would build the documentation. To do that, 3 CMake options are added:

1. `BUILD_LIBRARY_TESTS`: Builds the library tests
2. `BUILD_DOCUMENTATION_TESTS`: Builds the documentation tests. This entails building the documentation.
3. `BUILD_TESTS`: Builds all the *EasyNMEA* tests, meaning both library and documentation tests.

Furthermore, the system tests within the **Library tests** do require the installation of some extra python dependencies, which are listed in *<path_to_repo>/test/system/requirements.txt*. These are necessary to simulate a serial connection and a NMEA device. They can be installed with:

```
python3 install -r <path_to_repo>/test/system/requirements.txt
```

7.2.3 Directories

The *EasyNMEA* tests are held in the following directory structure:

1. `<repo-root>/test/unit`: For unit tests
2. `<repo-root>/test/system`: For system tests
3. `<repo-root>/docs/test`: For documentation tests

7.2.4 Automated Testing Jobs

All the *EasyNMEA* tests run automatically once a day for the `main` branch, as well as for the supported versions' branches. Furthermore, all the tests are run whenever a pull request is opened and with every commit pushed to an open pull request. To automate these tasks, since the public repository is hosted on GitHub, GitHub actions are used. This tool enables to create as many workflows with as many jobs in them as desired, making it ideal for test automation. Moreover, the jobs run on GitHub maintained servers, so the only thing we have to do is to define those workflows. This is done in `<repo-root>/.github/workflows`. *EasyNMEA* contains the following workflows and jobs:

- `automated_testing`, defined in `<repo-root>/.github/workflows/automated_testing.yml`. This workflow runs on pushes to `main` and any other maintained branch, on pull request creation or update, and once a day. It contains the following jobs:
  - `ubuntu-build-test`, which runs in the latest Ubuntu distribution available. This job installs all the necessary dependencies, builds all the tests and documentation, runs the all tests, and uploads the sphinx-generated HTML documentation so reviewers can check it.

7.2.5 Code Coverage Reporting

As stated in *Automated Testing Jobs*, *EasyNMEA* tests are run with every push to `main` and supported version branches, as well as with every push to any open pull request. This is done to make sure that every aspect of the library works as expected, as well as to guarantee that new changes do not break any established behaviour. Code coverage reporting takes this a step further, not only guaranteeing that all the tests pass at all times, but also checking whether those tests reach every possible source code outcome.

This is done using compiler specific flags that report every branch generated by the compiler and reached by the tests. These reports are then gather under one single human-readable code coverage report that is uploaded to an online platform, which in turn can keep track of the coverage progress with changes.

Presently, the coverage reports are generated in the `ubuntu-build-test` job, passing specific flags to GCC. Those flags are: `--coverage`, `-fprofile-arcs`, and `-ftest-coverage`. To ease the compilation, a CMake option `GCC_CODE_COVERAGE` has been created, which enables the code coverage flags if the compiler used is indeed GCC.

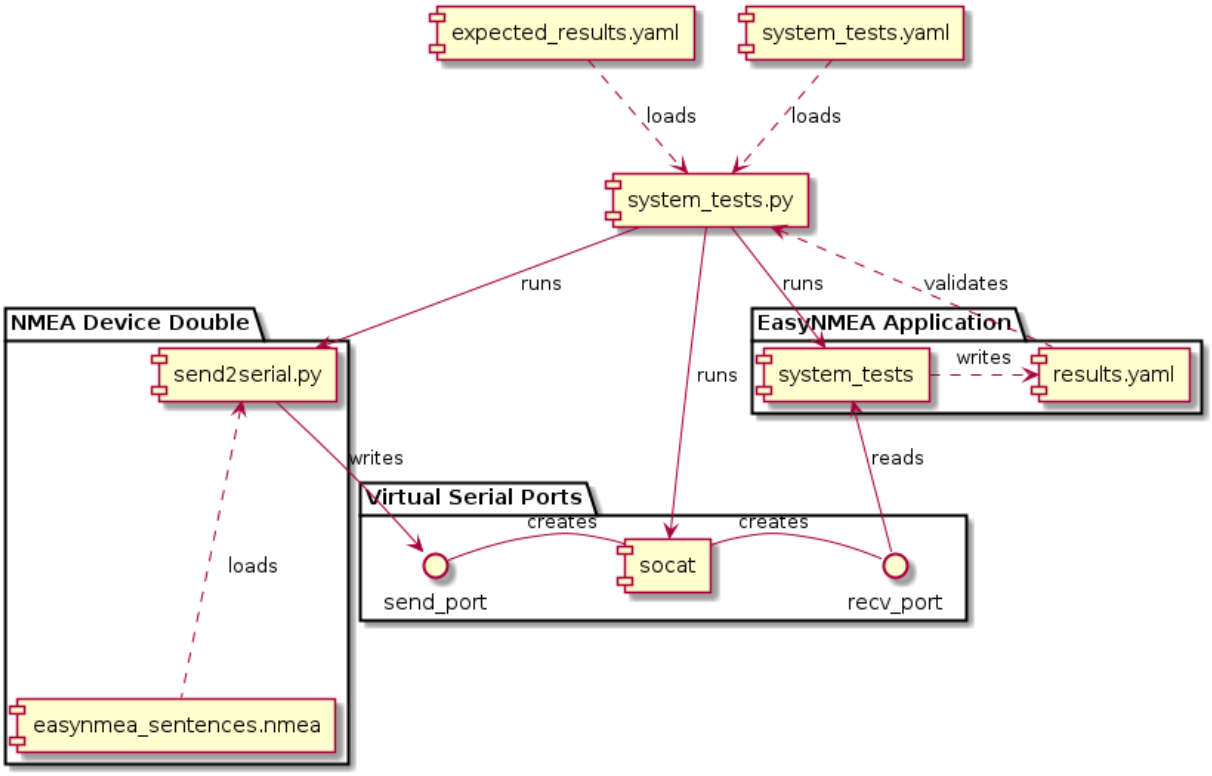
Then, the job uses gcovr to generate a report that is uploaded to Codecov. In turn, Codecov checks the code coverage on the changes proposed in the pull request, as well as the overall coverage. If any of those two decreases, the code coverage check fails, and the pull request cannot be merged.

7.2.6 Code Quality Analysis

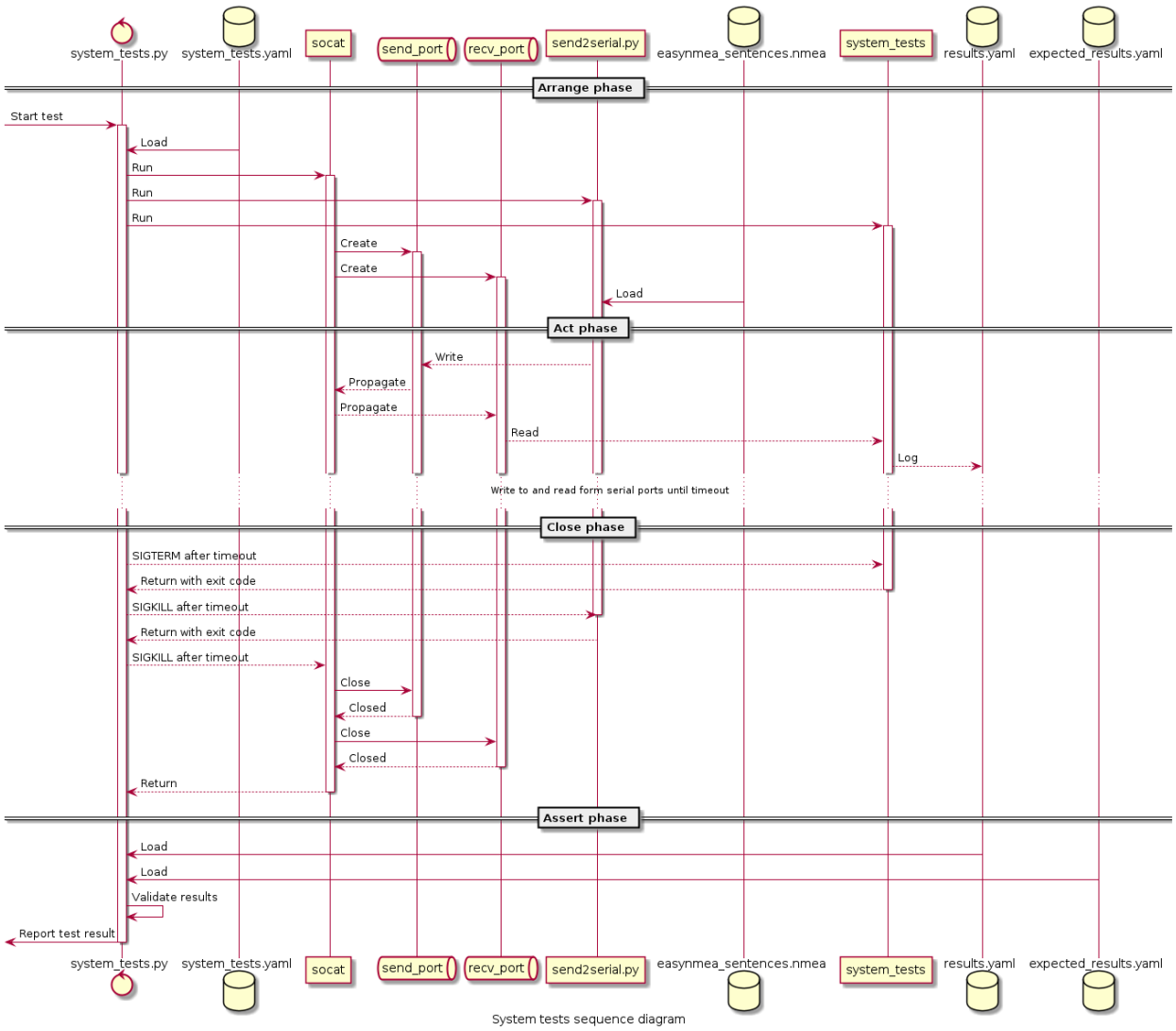
With every push to `main`, and with every pull request targeting it, and automated job is run to check code vulnerabilities using CodeQL. This job presents vulnerabilities in the form of code scanning alerts (see About code scanning with CodeQL).

7.3 System Tests

EasyNMEA provides a set of test which execute end-to-end verification of the library's functionalities. This is done by simulating a NMEA device sending data to a serial port. This data is then received by a *EasyNMEA* application which uses the library's public API to open the serial connection, wait until data of any given kind is received, and log this data for validation against expectations. For connecting the NMEA device double and the *EasyNMEA* application, socat is used to create a pair of **virtual serial ports**, one for the double to send the data, and the other one for the application to receive it. This way, the *EasyNMEA* application acts in the same way as a real application would, so public APIs can be tested in the same manner that they would be used in real applications. The relationships between the different system test components and the sequence of operations are shown in the following diagrams.



System tests components



1. **gpgga_read_some_and_close**: Open a pair of serial ports, send some valid NMEA sentences in one, and read GPGGA data on the other. Then, first close the EasyNMEA and then close the ports. Validate results against expectations.
2. **port_closed_externally**: Open a pair of serial ports, send some valid NMEA sentences in one, and read GPGGA data on the other. Then, close the serial ports with the EasyNMEA still opened. The application should detect this an exist gracefully. Validate results against expectations.
3. **stop_sending_data**: Open a pair of serial ports, send some valid NMEA sentences in one, and read GPGGA data on the other. Stop sending data before stopping the EasyNMEA. Close the EasyNMEA, then the sending app, and lastly close the ports. Validate results against expectations.
4. **late_sending**: Open a pair of serial ports. Then, first start a EasyNMEA, and after 1 second start sending some valid NMEA sentences. Then, close the EasyNMEA before closing the ports. Validate results against expectations.

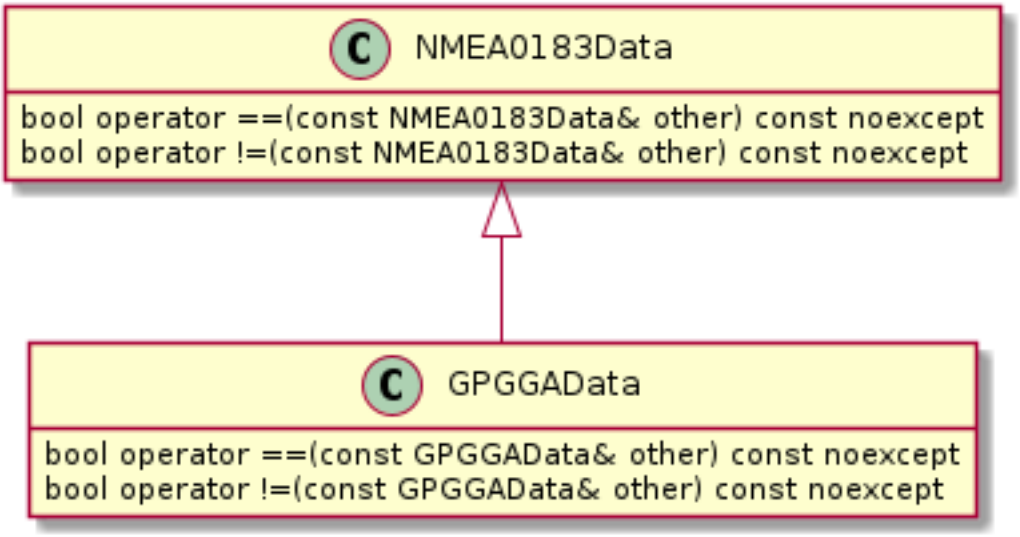
7.4 Unit Tests

EasyNMEA provides one test suite containing unit tests for each of the library classes. These suits test each and every public member function separately, mocking lower levels so that every possible case can be covered.

Even while the test suites provide a 100% line coverage on the classes they test, a 100% branch coverage is not required, as the implementation may use external functions that are not marked as `noexcept`, for which the compiler may generate branches that are virtually impossible to hit. It is up to the reviewers and maintainers to judge whether the branch coverage of a specific contribution is high enough, or if more test cases are required.

7.4.1 NMEA 0183 Data Types Unit Tests

As described in *API Level*, the way in which *EasyNmea* provides applications with NMEA data is through the NMEA 0183 data types (*GPGGAData*). These types feature == and != operators, so that two samples of the same type can be compared between them. Therefore, a set of unit tests for these operators of each of the types is required:

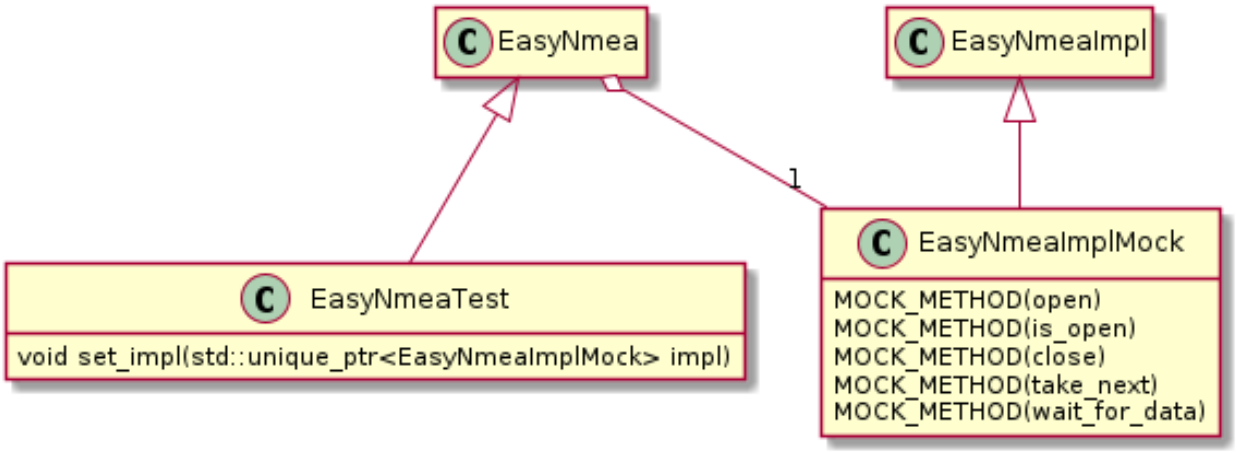


1. **NMEA0183DataComparisonOperators**: Checks that both comparison operators work for *NMEA0183Data*.
2. **GPGGADataComparisonOperators**: Checks that both comparison operators work for *GPGGAData*.

7.4.2 EasyNmea Unit Tests

As documented in *API Level*, *EasyNmea* provides applications with APIs to open and close the serial port, wait until data of one or more NMEA 0183 types is received, check whether the serial port connection is opened, and take the next unread sample of a given NMEA 0183 type.

The *EasyNmea* tests use the `EasyNmeaTest` class, which derives from *EasyNmea*, adding the possibility of substituting the `EasyNmeaImpl` with another instance. This enables the tests to implement a `EasyNmeaImplMock`, which derives from `EasyNmeaImpl`, mocking away the `EasyNmeaImpl::open()`, `EasyNmeaImpl::is_open()`, `EasyNmeaImpl::close()`, `EasyNmeaImpl::wait_for_data()`, and `EasyNmeaImpl::take_next()` functions. This way, the tests can substitute the `EasyNmeaImpl` instance in `EasyNmeaTest` with an instance of `EasyNmeaImplMock` on which expectations can be set, and then check whether *EasyNmea* behaves as expected depending on the `EasyNmeaImpl` returned values.



- *open()*
- *is_open()*
- *close()*
- *take_next()*
- *wait_for_data()*

open()

1. **openOk**: Check that *`EasyNmea::open()`* passes the correct arguments to `EasyNmeaImpl::open()`, and that it returns *`ReturnCode::RETURN_CODE_OK`* whenever `EasyNmeaImpl::open()` does so.
2. **openError**: Check that *`EasyNmea::open()`* passes the correct arguments to `EasyNmeaImpl::open()`, and that it returns *`ReturnCode::RETURN_CODE_ERROR`* whenever `EasyNmeaImpl::open()` does so.
3. **openIllegal**: Check that *`EasyNmea::open()`* passes the correct arguments to `EasyNmeaImpl::open()`, and that it returns *`ReturnCode::RETURN_CODE_ILLEGAL_OPERATION`* whenever `EasyNmeaImpl::open()` does so.

is_open()

1. **is_openOpened**: Check that *EasyNmea::is_open()* returns `true` when a connection is opened.
2. **is_openClosed**: Check that *EasyNmea::is_open()* returns `false` when a connection is closed.

close()

1. **closeOk**: Check that *EasyNmea::close()* returns *ReturnCode::RETURN_CODE_OK* when an opened port is closed correctly.
2. **closeError**: Check that *EasyNmea::close()* returns *ReturnCode::RETURN_CODE_ERROR* when an opened port cannot be closed correctly.
3. **closeIllegal**: Check that *EasyNmea::close()* returns *ReturnCode::RETURN_CODE_ILLEGAL_OPERATION* when attempting to close an already closed port.

take_next()

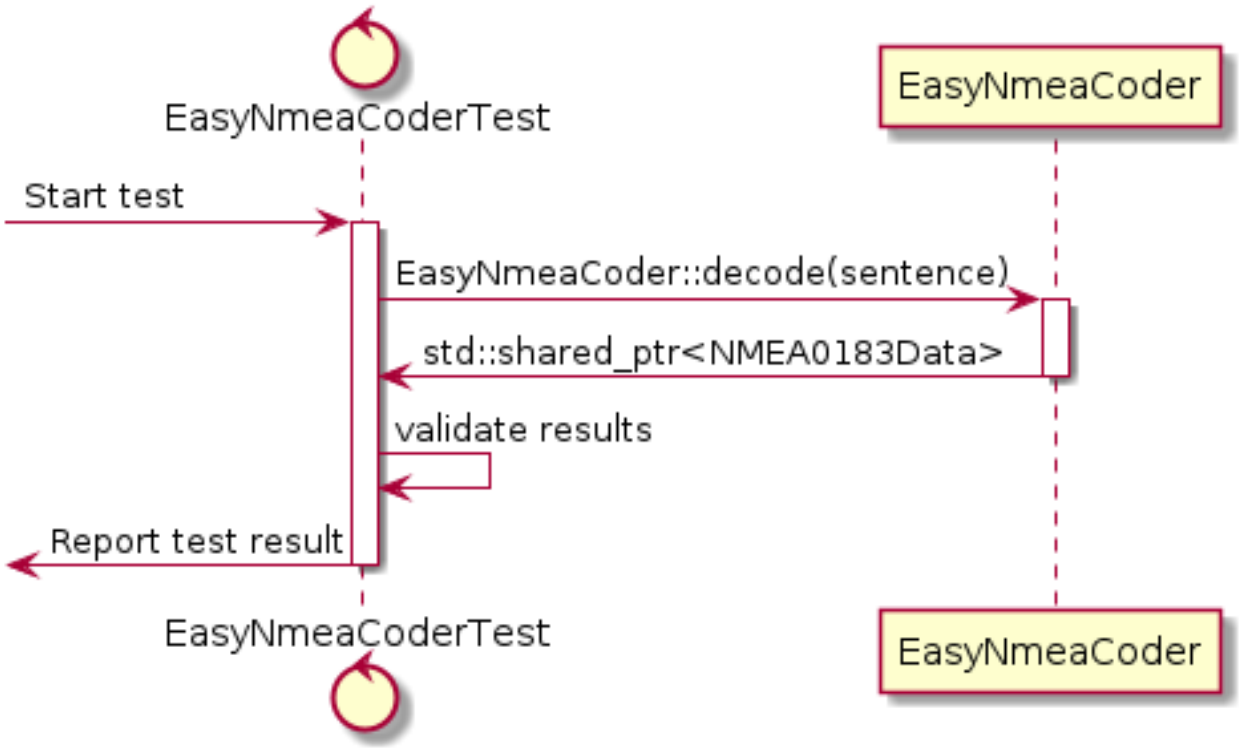
1. **take_nextOk**: Check that *EasyNmea::take_next()* calls to `EasyNmeaImpl::take_next()` with the appropriate arguments, and that if returns *ReturnCode::RETURN_CODE_OK* whenever `EasyNmeaImpl::take_next()` does so. Furthermore, check that the data output is the sample output by `EasyNmeaImpl::take_next()`.
2. **take_nextNoData**: Check that *EasyNmea::take_next()* calls to `EasyNmeaImpl::take_next()` with the appropriate arguments, and that if returns *ReturnCode::RETURN_CODE_OK* whenever `EasyNmeaImpl::take_next()` does so. Furthermore, check that the data output is equal to the input.

wait_for_data()

1. **wait_for_dataOk**: Check that `EasyNmeaImpl::wait_for_data()` is called with the appropriate arguments, and that *EasyNmea::wait_for_data()* returns *ReturnCode::RETURN_CODE_OK* whenever `EasyNmeaImpl::wait_for_data()` does so.
2. **wait_for_dataTimeout**: Check that `EasyNmeaImpl::wait_for_data()` is called with the appropriate arguments, and that *EasyNmea::wait_for_data()* returns *ReturnCode::RETURN_CODE_TIMEOUT* whenever `EasyNmeaImpl::wait_for_data()` does so.
3. **wait_for_dataTimeoutDefault**: The difference with **wait_for_dataTimeout** os that in this case, *EasyNmea::wait_for_data()* is called leaving the timeout as default.
4. **wait_for_dataIllegal**: Check that `EasyNmeaImpl::wait_for_data()` is called with the appropriate arguments, and that *EasyNmea::wait_for_data()* returns *ReturnCode::RETURN_CODE_ILLEGAL_OPERATION* whenever `EasyNmeaImpl::wait_for_data()` does so.
5. **wait_for_dataError**: Check that `EasyNmeaImpl::wait_for_data()` is called with the appropriate arguments, and that *EasyNmea::wait_for_data()* returns *ReturnCode::RETURN_CODE_ERROR* whenever `EasyNmeaImpl::wait_for_data()` does so.

7.4.3 EasyNmeaCoder Unit Tests

As documented in *Implementation Level*, `EasyNmeaCoder` provides APIs for decoding NMEA 0183 supported sentences, specifically `EasyNmeaCoder::decode()`. This member function takes a NMEA 0183 sentence as a string and returns a `std::shared_ptr` to a *`NMEA0183Data`*, which *`NMEA0183DataKind`* field can be used to cast it into the appropriate NMEA 0183 data structure. This set of tests target the `EasyNmeaCoder::decode()` function, passing different sentences and checking the return against expected outputs.



- *decode()*

decode()

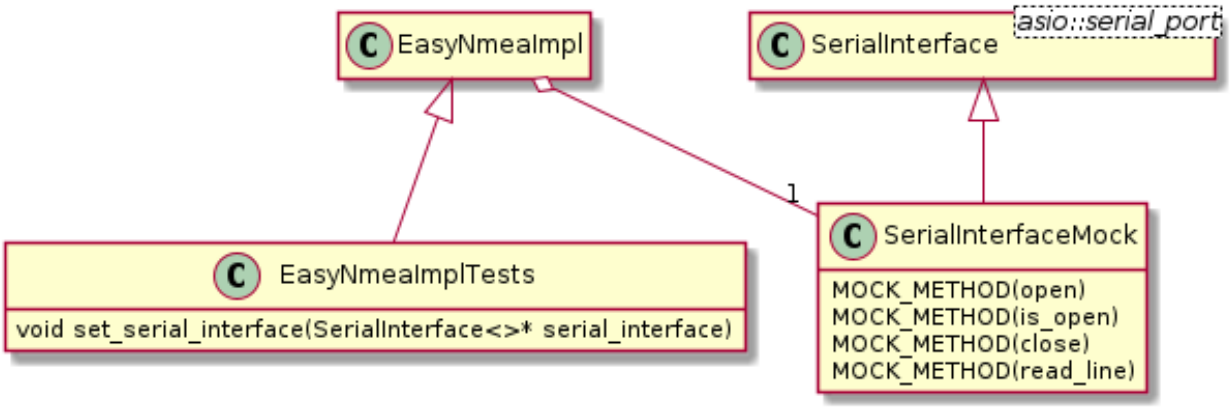
1. **decodeGPGGAValidNE**
2. **decodeGPGGAValidNW**
3. **decodeGPGGAValidSE**
4. **decodeGPGGAValidSW**
5. **decodeGPGGAValidNoAgeOfDiffGPS**
6. **decodeGPGGAValidEmptyAgeOfDiffGPSNoDiffRefStation**
7. **decodeGPGGAValidNoDiffRefStation**
8. **decodeGPGGAValidNoOptionals**
9. **decodeGPGGAInvalidTime**
10. **decodeGPGGAInvalidLatitudeLength**

11. **decodeGPGGAInvalidLatitudeDegrees**
12. **decodeGPGGAInvalidLatitudeMinutes**
13. **decodeGPGGAInvalidLongitudeLength**
14. **decodeGPGGAInvalidLongitudeDegrees**
15. **decodeGPGGAInvalidLongitudeMinutes**
16. **decodeGPGGAInvalidAltitudeUnits**
17. **decodeGPGGAInvalidHeightUnits**
18. **decodeGPGGAInvalidChecksum**
19. **decodeGPGGANoTime**
20. **decodeGPGGANoLatitude**
21. **decodeGPGGANoLongitude**
22. **decodeGPGGANoFix**
23. **decodeGPGGANoNumberOfSatellites**
24. **decodeGPGGANoHDOP**
25. **decodeGPGGANoAltitude**
26. **decodeGPGGANoHeight**
27. **decodeGPGGANoChecksum**
28. **decodeInvalidSentenceID**
29. **decodeUnsupportedSentence**
30. **decodeEmptySentence**
31. **decodeOnlyChecksumSentence**
32. **decodeOnlyAstheriscSentence**

7.4.4 EasyNmeaImpl Unit Tests

As documented in *Implementation Level*, `EasyNmeaImpl` provides with the implementation for the `EasyNmea` public API, namely opening and closing the serial port, waiting until data of one or more NMEA 0183 types has been received, checking whether the serial port connection is opened, and taking the next unread sample of a given NMEA 0183 type.

The `EasyNmeaImpl` tests use the `EasyNmeaImplTest` class, which derives from `EasyNmeaImpl`, adding the possibility of substituting the `SerialInterface` with another instance. This enables the tests to implement a `SerialInterfaceMock`, which derives from `SerialInterface`, mocking away the `SerialInterface::open()`, `SerialInterface::is_open()`, `SerialInterface::close()`, and `SerialInterface::read_line()` functions. This way, the tests can substitute the `SerialInterface` instance in `EasyNmeaImplTest` with an instance of `SerialInterfaceMock` on which expectations can be set, and then check whether `EasyNmeaImpl` behaves as expected depending on the `SerialInterface` returned values.



- *open()*
- *is_open()*
- *close()*
- *wait_for_data()*
- *take_next()*
- *~EasyNmeaImpl()*

open()

1. **openSuccess**: Opens a not previously opened `EasyNmeaImpl`. The return is expected to be *`ReturnCode::RETURN_CODE_OK`*.
2. **openOpened**: Attempts to open an already opened `EasyNmeaImpl`. This is simulated by forcing `SerialInterface::is_open()` to return `true`. The return is expected to be *`ReturnCode::RETURN_CODE_ILLEGAL_OPERATION`*.
3. **openWrongPort**: Attempts to open a `EasyNmeaImpl` on an invalid port. This is simulated by forcing `SerialInterface::open()` to return `false`. The return is expected to be *`ReturnCode::RETURN_CODE_ERROR`*.

is_open()

1. **is_openOpened**: Check that whenever `SerialInterface::is_open()` returns `true`, `EasyNmeaImpl::is_open()` also returns `true`.
2. **is_openClosed**: Check that whenever `SerialInterface::is_open()` returns `false`, `EasyNmeaImpl::is_open()` also returns `false`. Furthermore, this test also checks that `EasyNmeaImpl::is_open()` returns `false` whenever the underlying pointer to `SerialInterface` is `nullptr`.

close()

1. **closeSuccess**: Check that whenever `SerialInterface` reports that a port is opened at first, and then return `true` on the call to `SerialInterface::close()`, then `EasyNmeaImpl::close()` returns *`ReturnCode::RETURN_CODE_OK`*.
2. **closeError**: Check that whenever `SerialInterface` reports that a port is opened at first, and then return `false` on the call to `SerialInterface::close()`, then `EasyNmeaImpl::close()` returns *`ReturnCode::RETURN_CODE_ERROR`*.
3. **closeClosed**: Check that calling `EasyNmeaImpl::close()` on a non-opened `EasyNmeaImpl` returns *`ReturnCode::RETURN_CODE_ILLEGAL_OPERATION`*.

wait_for_data()

1. **wait_for_dataData**: Check that `EasyNmeaImpl::wait_for_data()` returns *`ReturnCode::RETURN_CODE_OK`* when a sentence which type specified in the *`NMEA0183DataKindMask`* mask is received. Furthermore, check that the output mask has the corresponding bit correctly set.
2. **wait_for_dataClosed**: Check that `EasyNmeaImpl::wait_for_data()` returns *`ReturnCode::RETURN_CODE_ILLEGAL_OPERATION`* when called on a closed `EasyNmeaImpl`.
3. **wait_for_dataDataEmptyMask**: Check that `EasyNmeaImpl::wait_for_data()` will return *`ReturnCode::RETURN_CODE_TIMEOUT`* after timing out when an empty *`NMEA0183DataKindMask`* is passed, even when data from any of the supported types has been received. It also checks that the output *`NMEA0183DataKindMask`* is set to `none`.
4. **wait_for_dataError**: Check that whenever `SerialInterface::read_line()` returns false, the call to `EasyNmeaImpl::wait_for_data()` unblocks and returns *`ReturnCode::RETURN_CODE_ERROR`*. It also checks that the output *`NMEA0183DataKindMask`* is set to `none`.

take_next()

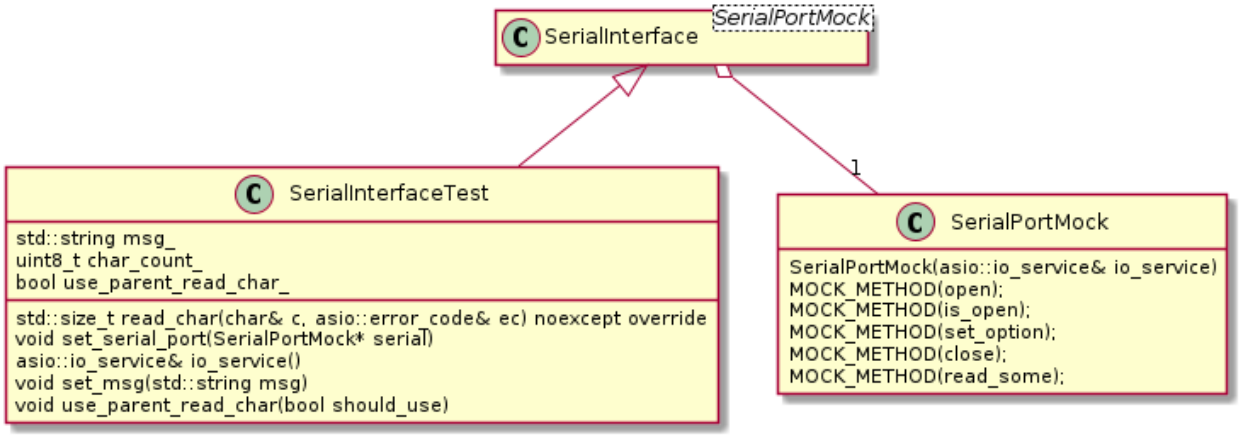
1. **take_next**: Check that whenever `EasyNmeaImpl::wait_for_data()` returns *`ReturnCode::RETURN_CODE_OK`*, then, data can be taken with `EasyNmeaImpl::take_next()`, which returns *`ReturnCode::RETURN_CODE_OK`*. Furthermore, it tests that other NMEA 0183 valid sentences are not returned nor reported to be have been received, and that incomplete GPGGA sentences are not returned nor reported either.

~EasyNmeaImpl()

1. **destroyNoClose**: Checks that letting an opened `EasyNmeaImpl` instance go out of scope without calling `EasyNmeaImpl::close()` is alright.

7.4.5 SerialInterface Unit Tests

As documented in *Serial Interface Level*, `SerialInterface` provides functions to open, close, and read from serial ports using Asio. The `SerialInterface` tests use a `SerialInterfaceTest` class which derives from `SerialInterface` specialized in `SerialPortMock`, which mocks `asio::serial_port`.



1. Since `SerialInterfaceTest` creates its `SerialPortMock` in the constructor, no expectations can be set to that object. For this reason, `SerialInterfaceTest` provides a `set_serial_port()` public member function that can be used to substitute the `SerialPortMock` instance with one on which expectations have been set.
2. To be able to construct this `SerialPortMock`, a getter `io_service()` is also provided.
3. Some tests need to mock `SerialPort::read_some()` (`asio::serial_port::read_some()`) so that `SerialInterface::read_line()` returns a specific `std::string`. To that end, `SerialInterface` wraps the call to `SerialPort::read_some()` with a `read_char()`, which `SerialInterface::read_line()` calls to perform the actual read from the port. Since for unit testing purposes `SerialPortMock` is used instead of `asio::serial_port`, a mock `SerialPortMock::read_some()` would be needed. However, due to the function's signature, it is not possible to set expectations on the read characters. This has led to `SerialInterfaceTest` overriding `SerialInterface::read_char()` with an overload that either simply calls to the `SerialInterface::read_char()` implementation, or returns a character from a string. To do this, `SerialInterfaceTest` provides a `set_msg()` function that is used to set the line that `read_line` will read. To enable `SerialInterfaceTest::read_char()` to read characters from the set message instead of using `read_some()`, a `use_parent_read_char()` is provided. By default, `SerialInterfaceTest::read_char()` will call `SerialInterface::read_char()` (which calls `read_some()`), however, if the `use_parent_read_char_` flag is set (calling `use_parent_read_char(false)`), then `SerialInterfaceTest::read_char()` will read the characters of the set message one at a time (simulating reading characters one by one from the serial port).

- *open()*
- *is_open()*
- *close()*
- *read_line()*

open()

1. **openSuccess**: Opens a not previously opened serial port with a valid port and baudrate. The return is expected to be `true`
2. **openOpened**: Attempts to open an already opened port. The return is expected to be `false`.
3. **openWrongPort**: Attempts to open a port on an invalid port. The return is expected to be `false`.
4. **openWrongBaudrate**: Attempts to set a non valid baudrate to the serial port. The return is expected to be `false`.

is_open()

1. **is_openOpened**: Checks whether `SerialInterface::is_open()` returns `true` for an open port.
2. **is_openClosed**: Checks whether `SerialInterface::is_open()` returns `false` for an closed port.

close()

1. **closeSuccess**: Closes an already opened port. The return is expected to be `true`.
2. **closeClosed**: Closes an already closed port. The return is expected to be `true`.
3. **closeAsioError**: Attempts to close an open port that Asio cannot close. The return is expected to be `false`.

read_line()

1. **read_lineSuccess**: Checks that lines ending in `\n` or `\r\n` are returned correctly. The return is expected to be `true`. This test is performed on an opened serial port. Furthermore, the function should be called with an empty string, as well as with a non-empty one. Both cases should output just the read line without any characters that it had on calling `SerialInterface::read_line()`.
2. **read_lineClosed**: Checks that calling `SerialInterface::read_line()` on a closed port returns `false`.
3. **read_lineReadError**: Simulates that *asio::serial_port::read_some()* returns an error and checks that in the case, the `SerialInterface::read_line()` return is `false`. This test covers the case when *asio::serial_port::close()* is called while blocked on *asio::serial_port::read_some()*, since that breaks the block, making *asio::serial_port::read_some()* return with a not OK *asio::error_code*.

7.5 Documentation Testing

This section describes the tests implemented for the *EasyNMEA* documentation:

1. `easynmea-documentation-test`: An executable generated to check that all code snippets in the documentation compile. This way, whenever we make an API change, we will be forced to update the documentation to reflect it, and in that way we make sure that all the code in the documentation is up to date.
2. `documentation.line_length`: RST files usually have a line length no longer than 120 characters. *Doc8* is used to check this for every RST file with argument *–max-line-length 120*.
3. `documentation.spell_check`: A spelling check for the documentation. Sphinx builder *spelling* supports this, plus it also adds the possibility to have one or more custom dictionaries for words that the builder otherwise considers not correct.

4. `documentation.link_check`: Checks that all documentation hyperlinks are valid. Sphinx supports this using *linkcheck* builder.

As defined in *Directories*, these tests are located in *<repo-root>/docs/test*. Furthermore, it is possible to activate them with CMake option `BUILD_DOCUMENTATION_TESTS` (see *Build Tests*).

EasyNMEA is an open source, free-to-use cross-platform *C++* library to retrieve **Global Navigation Satellite System** (GNSS) information from GNSS modules which communicate with NMEA 0183 over serial. It can retrieve GNSS data from any GNSS device sending NMEA 0183 sentences using serial communication.

EasyNMEA provides a lightweight and easy-to-use API with which applications can wait until data of any of the supported NMEA 0183 sentences is received, and then retrieve it in an understandable manner without the need of knowing the inner details of the NMEA 0183 protocol.

The source code is hosted on GitHub, check it out!

INDEX

E