PlatformPilot Documentation

Release 1.2.0

Neobotix GmbH

Aug 10, 2021

CONTENTS:

CHAPTER

ONE

USER MANUAL

1.1 Installation

1.1.1 Ubuntu

- Obtain the correct `neobotix-pilot-core-` debian package for your architechture and Ubuntu version. For example, the file name for the *x86_64 Ubuntu 20.04* package ends with `-x86_64-ubuntu-20.04.deb`.
- Install the package as follows:

```
sudo dpkg -i neobotix-pilot-core-*.deb
```

- Install missing dependencies via:

```
sudo apt -f install
```

- The contents of the package should now be installed under `/opt/neobotix/pilot-core/`.
- If you received a license security dongle and you already attached it, unplug it and attach it again in order for the installed udev rule to take effect.
- To change the default workspace folder (which is *~/pilot/*) set the following environment variable in your `~/.bashrc`:

```
export PILOT_HOME=/your/custom/path
source ~/.bashrc
```

- To setup the workspace execute the following script:

```
sh /opt/neobotix/pilot-core/scripts/pilot_setup.sh /opt/neobotix/pilot-core
```

  This will setup a folder `~/pilot/` with default settings.

- To enable autostart:

```
cp /opt/neobotix/pilot-core/share/applications/Neobotix-PlatformPilot.desktop ~/.
↪config/autostart/
```

- In addition you can install a desktop launcher as follows:

```
xdg-desktop-icon install /opt/neobotix/pilot-core/share/applications/Neobotix-
↪PlatformPilot.desktop
```

  You may need to right click on the icon first and select "Allow Launching" to activate it.

- Change the default configuration to match your platform type:

```
echo /opt/neobotix/pilot-core/config/default/mpo-700/ > config/local/parent
```

  where you replace `mpo-700` with your platform type.
- If you received a license security dongle, edit your `config/local/LicenseCheck.json` file to contain your license key which you should find printed on the dongle.
- Set a *name* and *serial* for your platform in `config/local/platform.json`.
- Optionally set additional custom configuration options in `config/local`, see *Configuration*.
- Optionally setup users and passwords, see *User Management*.
- See *Running* on how to run *PlatformPilot*.

1.1.2 Windows

- Execute the `neobotix-pilot-core-?-windows10.exe` installer.
- The package will be installed to your Profile Folder in `C:\Users\?\PlatformPilot\`.
- Change into the folder `PlatformPilot\bin\config\local`. Edit the file `parent` with a line like this:

```
../../../config/default/mpo-700/
```

  where you substitute `mpo-700` with your platform type.
- If you received a license security dongle, edit your `bin\config\local\LicenseCheck.json` file to contain your license key which you should find printed on the dongle.
- Set a *name* and *serial* for your platform in `bin\config\local\platform.json`.
- Optionally set additional custom configuration options in `bin\config\local`, see *Configuration*.
- Optionally setup users and passwords, see *User Management*.
- See *Running* on how to run *PlatformPilot*.

1.2 Changelog

1.2.1 Release 1.2

Version 1.2.1

- `OPC_UA_Proxy` fix for crash
- `OPC_UA_Proxy` fix for wrong variable types

Version 1.2.0 (June 2021)

- New `HttpServer` implementation, no longer using *libmicrohttpd*.
  - Supports *Server-Sent-Events (SSE)*, via `/api/stream/...`
  - *Deflate* response compression (multi-threaded)
  - Asynchronous chunked transfers

- `localization.status` topic is also being recorded now.
- New `LocalPlanner` functions: `await_goal()`, `await_goal_ex(...)`, `cancel_goal_await()`
- `LocalPlanner::pause()` now has an optional `bool em_stop` parameter to enable emergency stopping.
- OPC-UA write variable support via `Proxy::write_variable()` and `Proxy::write_object_variable()`.
- `TaskHandler` now supports script parameters via `execute_file()` and `execute_program()` which are passed on to the `main(...)` function in *LUA*.
- Sending a new goal while driving works as expected now, planners wait for platoform to stop before planning new path.
- `Kinematics_CanNode` fix for CAN bus initialization, now attempts to re-initialize until successful.
- New `TaskHandler` functions: `get_time_sec()`, `get_time_millis()`, `get_time_micros()`
- `HttpSession` now contains `session_timeout` as well
- Additional incident reporting for *RelayBoard* and *MPO-700* homing.
- `SickMicroscan3` fix for reflector detection
- OPC-UA Proxy authentication support via `username` and `password` configs.
- `TaskHandler` scripts have `REQUEST` permission now
- `USER` (neo-user) has `INTERVENE_SCRIPT` permissions now

1.2.2 Release 1.1

Version 1.1.0 (May 2021)

1.3 Running

Note: If you received a license security dongle, make sure to have it attached.

You can always execute the supplied binaries directly, but for convenience we provide little helper scripts that take care of some required command line options.

1.3.1 Pilot

In Windows execute the `bin/run_pilot.bat` file either in a terminal or by double clicking on it.

In Ubuntu:

```
cd ~/pilot
./run_pilot.sh
```

Additional options can be provided to the scripts as such:

```
./run_pilot.sh --LocalPlanner.vel_limits.max_trans_vel 0.5
run_pilot.bat --LocalPlanner.vel_limits.max_trans_vel 0.5
```

Permanent options can be set by writing config files in `config/local`, see *Configuration*. For example a file `config/local/LocalPlanner.json`:

```
{
        "vel_limits": {
                "max_trans_vel": 0.5
        }
}
```

will set the default maximum velocity of the platform to 0.5 m/s.

1.3.2 Simulation

The simulation binary will simulate the kinematics as well as sensor data without accessing any hardware. Otherwise there is no difference to `run_pilot.sh`.

In Windows execute the `bin/run_pilot_simulation.bat` file either in a terminal or by double clicking on it.

In Ubuntu:

```
cd ~/pilot
./run_pilot_simulation.sh
```

1.3.3 Replay

To run *PlatformPilot* in replay mode use the provided script:

```
cd ~/pilot
./run_pilot_replay.sh
```

It will try to connect to a running VNX player on `/tmp/vnxplayer.sock`. Replay mode is used to view what happened during a recorded drive. Similar to simulation mode there is no hardware access.

1.3.4 Data Recording

To make a data recording use the provided script:

```
cd ~/pilot
./record_data.sh.sh
```

The resulting file will be in `user/data/`.

1.3.5 Parameters

It is possible to override the default configuration by either writing local config files in `config/local/` or by specifying parameters on the command line.

The following configuration options can be set for most executables above:

`self_test` If to perform a self test at the beginning. Will exit if it fails. (default = *true*)

`auto_shutdown` If to automatically shutdown the host machine in case platform is being turned off via the key switch. (default = *true*)

with_hardware If hardware modules should be started, ie. hardware is present on this machine and not on a remote. (default = *true*)

enable_joystickmng If to enable joystick input. (default = *true*)

enable_http_server If to enable HTTP server on port 8888. Needed for HTTP API, WebGUI, etc. (default = *true*)

enable_taskhandler If to enable the *TaskHandler* module. (default = *true*)

enable_movechainhandler If to enable the MovechainHandler module. (default = *true*)

enable_opcua_server If to enable the *vnx.opc_ua.Server* module. (default = *false*)

opcua_proxy_map A map of *vnx.opc_ua.Proxy* modules to start, [*module name* => *server address*].

vnx_proxy_map A map of *vnx.Proxy* modules to start, [*module name* => *server address*].

vnx_server_map A map of *vnx.Server* modules to start, [*module name* => *server endpoint*].

vnx_jrpc_server_map A map of *vnx.JRPC_Server* modules to start, [*module name* => *server endpoint*].

footprint Footprint of the platform as a JSON object, see *pilot.Footprint*.

platform Static platform information as a JSON object, see *pilot.PlatformInfo*.

By default the following servers are started:

vnx.Server on .pilot_main.sock Supports connections from the local machine via UNIX domain sockets, using the native VNX binary protocol. Almost all permissions are granted to clients connecting to this server, ie. no logins required. A *vnx.Proxy* is needed to connect. On Linux systems only.

vnx.Server on 0.0.0.0:5555 Supports connections from anywhere via TCP/IP sockets, using the native VNX binary protocol. Login is required to gain anything more than read-only permissions, see *User Management*. A *vnx.Proxy* is needed to connect.

In general, most applications accept the following command line arguments:

- The `-h` switch displays a small help message and exits.
- The `-d` switch increases the debug level of the terminal output
- The `-c` switch accepts a directory with configuration files. You can specify several directories by repeating the switch or by providing multiple directories at once. The order of directories matters, they are read from left to right, potentially overriding previous values.
- Finally, you can set specific config values directly on the command line. See *Configuration* for details.

1.3.6 Maps

The active *Grid Map* and *Road Map* are stored in the `~/pilot/` folder as `current_grid_map.grid` and `current_road_map.road`.

They can either be copied there by hand or uploaded via the *GTK GUI*.

1.4 Terminal Interface

While *running one of the pilot binaries* you can, at any time, hit the `<Enter>` key which interrupts the log output and gives you a terminal prompt. You can now execute one of several commands. Type it in and press `<Enter>` to execute.

Commands that have an output keep the log paused after execution, to give you time to look at it. The log continues when you press `<Enter>` again.

The following commands are currently available:

- `quit` exits the application
- `debug [level]` changes the level of the output. The bigger the number, the more log messages you get.
- `errors` outputs the last error messages
- `topic [expr]` searches all topic names for `expr`. If there is no exact match, all topics that `expr` is a prefix of are shown. If there are none, all matches are shown.
- `module [expr]` lists all modules that `expr` matches on. If there is an exact match, it displays more detailed information for that module.
- `grep <expr>` restricts terminal output to messages that contain `expr`.
- `journal [expr]` does the same as `grep` but first shows all past log messages that contain `expr`.
- `spy [expr]` does the same as `dump` but without the actual message content.
- `dump [expr]` outputs the messages on all topics that match `expr`.
- `htop [field]` shows the modules with the highest CPU load in a table that refreshes regularly. `field` can be `avg` to change the sorting.
- `exec <module>[.method [args...]]` and `call <module>[.method [args...]]` invoke the given method of the given module with the given arguments. `exec` waits for the command to finish and displays the result while `call` performs an asynchronous call. If no method is given, a list of method signatures for the module is shown.

While typing, the terminal will provide you with suggestions for the command and possible arguments. Press the `<Tab>` key to complete, press it twice to get a list of suggestions. You can also use the up and down arrows the scroll through the history of past commands.

1.5 Configuration

When you start a command line tool, you can supply it with one or more configuration destinations. The configuration is written in JSON.

1.5.1 Reading

Hierarchy

The configuration follows a hierarchy that is separated by dots (`.`) in the names of the keys. The hierarchy is inferred from the file system and the structure of the JSON in the files.

For example, assume you supply the config destination `config/local/` and there is a file `config/local/some/path/Test.json` and it contains the following content:

```
{
        "some_key": {
                "other_key": {
                        "the_answer": 42,
                        "the_question": ["life", "universe", "everything"]
                }
```

(continues on next page)

(continued from previous page)

```
        }
}
```

In this case, the config key `some.path.Test.some_key.other_key.the_answer` will be read and set to `42` and the key `some.path.Test.some_key.other_key.the_question` will contain a list of strings as shown.

A config destination can contain the special file called `parent`. It should consist of a single line giving a path (absolute or relative to the current destination) to include. This means the contents of the given destination will be treated as if they were present in the current destination. Additionally, the parent destination will be read first which is important for cascading (see below).

Cascading

The structure of the configuration makes it possible (and actually good and common practice) to split a config object across multiple locations. For example your config destination may have a file `Test.json` that contains

```
{ "field_1": 42 }
```

but also reference a `parent` directory that also has a file `Test.json` that contains

```
{ "field_2": 23 }
```

In effect, both `Test.field_1` and `Test.field_2` are correctly read. By default, we use this technique to separate

- configuration specific to an installation, which includes
- configuration specific to a robot model, which includes
- generic configuration.

This makes the configuration easy to maintain.

Given the rules for the hierachy and for parent destinations, it is very much possible to specify the same key multiple times. In that case **the key that is read last overwrites any previous one of the same name**. In particular, a destination always overrides its parent.

Appending

It is also possible to append values to an array, by adding a + to the variable name as such:

```
{ "array": [1, 2, 3] }
{ "array+": [4, 5, 6] }
```

The resulting `array` config value will be `[1, 2, 3, 4, 5, 6]`.

Command line

You can also supply configuration on the command line. For example, to set `MyModule.max_velocity` to `5`, append `--MyModule.max_velocity 5` to the command line.

Values are read as JSON objects. Make sure to use the correct quoting according to the shell you use. When setting a boolean value to `true`, you can omit the value and just give the key.

Arrays and objects can be provided on the command line as follows:

```
command --array [1, 2, 3, 4] --object {"field": 1234, "array": ["a", "b", "c"]}
```

Config keys given at the command line are read as the ultimate last ones, so they override any previous keys with the same names. This is useful for testing out certain values without constantly having to edit files.

1.5.2 Assignment

While some config keys are read explicitly by the application, most of the assignment happens automagically.

Modules

A module that is started with the name `MyModule` will automatically be assigned all configuration below the `MyModule.` key. From then on it behaves like a class (see below).

Classes

If a configuration key is applied to an instance of a VNX class, the members of the class are assigned the subkeys where the names match.

For example if you have an object of the class

```
class Test {
        string name;
        int value;
        vector<int> numbers;
        Test2 other_one;
}
```

and you assign it the config

```
{
        "name": "Picard",
        "value": 1234,
        "numbers": [1, 2, 3, 4],
        "other_one": {
                "what": "ever"
        }
}
```

then all the fields get assigned the config key with the matching name. Notice that `other_one` is another object which will get the same treatment recursively with the embedded config object.

Fields without a matching config key are default initialized and spare keys without a matching field are silently ignored.

1.6 User Management

The authentication model is based on *users* who may or may not have permission to perform certain actions. Without authentication, by default, only a limited number of actions are available.

Every user has a name and a password which are used for authentication.

1.6.1 Login

Command line tools usually provide the `-u <username>` switch to provide a user name. Please refer to the documentation *of the respective tool*.

Graphical tools usually provide a graphical way of authentication.

1.6.2 Permissions

Most API functions require a certain permission to be executed. If the permission is not held by the user an error is returned or thrown.

Every user has a set of *access roles* and every access role has a set of *permissions*. The permissions of a user consist of the permissions of their access roles. Mindful assignment to access roles allows for a fine-grained access control in a multi-user environment.

See *vnx.access_role_e* for the set of default access roles and their permissions.

1.6.3 Configuration

Available access roles and their sets of permissions can be configured via the config key `vnx.authentication.permissions` which is an object with access roles (as strings) as keys and a list of permissions (as strings) as values.

Users and their access roles can be configured with the key `vnx.authentication.users` which is a list of *user objects*. Note that the passwords for the config map `vnx.authentication.passwd` are usually kept separately (in a subdirectory) so that they can be protected from reading and writing.

Here is the default configuration `config/default/generic/vnx/authentication.json`:

```
{
        "users": [
                {
                        "name": "neo-user",
                        "access_roles": ["USER"]
                },
                {
                        "name": "neo-installer",
                        "access_roles": ["INSTALLER"]
                },
                {
                        "name": "neo-admin",
                        "access_roles": ["INSTALLER", "ADMIN"]
                }
        ],
        "permissions": {
                "OBSERVER": [
                        "READ_CONFIG",
                ],
                "USER": [
                        "pilot.permission_e.MOVE",
                        "pilot.permission_e.CHARGE",
                        "pilot.permission_e.INITIALIZE",
                        "pilot.permission_e.RECORD_DATA",
                        "pilot.permission_e.EXECUTE_SCRIPT",
                        "vnx.addons.permission_e.READ_DIRECTORY",
```

(continues on next page)

(continued from previous page)

```
                        "vnx.addons.permission_e.FILE_DOWNLOAD",
                ],
                "TASK": [
                        "VIEW",
                        "CONST_REQUEST",
                        "pilot.permission_e.MOVE",
                        "pilot.permission_e.CHARGE",
                        "pilot.permission_e.RECORD_DATA",
                        "pilot.permission_e.RELAY_CONTROL",
                        "pilot.permission_e.DISPLAY_CONTROL",
                ],
                "INSTALLER": [
                        "pilot.permission_e.MOVE",
                        "pilot.permission_e.CHARGE",
                        "pilot.permission_e.INITIALIZE",
                        "pilot.permission_e.RECORD_DATA",
                        "pilot.permission_e.REMOTE_CONTROL",
                        "pilot.permission_e.RELAY_CONTROL",
                        "pilot.permission_e.DISPLAY_CONTROL",
                        "pilot.permission_e.CHANGE_GRIDMAP",
                        "pilot.permission_e.CHANGE_ROADMAP",
                        "pilot.permission_e.UPLOAD_SCRIPT",
                        "pilot.permission_e.EXECUTE_SCRIPT",
                        "pilot.permission_e.INTERVENE_SCRIPT",
                        "vnx.addons.permission_e.READ_DIRECTORY",
                        "vnx.addons.permission_e.FILE_DOWNLOAD",
                        "vnx.addons.permission_e.FILE_UPLOAD",
                        "vnx.addons.permission_e.FILE_DELETE",
                ]
        }
}
```

As can be seen, built-in permissions (of type *vnx.permission_e*) such as `VIEW` and `READ_CONFIG` can be specified without the full namespace. A permission can also be removed by adding a `!` in front of the name: `!VIEW`.

For more information regarding configuration files see *Configuration*.

1.6.4 Adding Users

To add new users create a config file `config/local/vnx/authentication.json`:

```
{
        "users+": [
                {
                        "name": "foo",
                        "access_roles": ["USER", ...]
                },
                ....
        ]
}
```

By removing the + in `users+` you can discard the default users which were set in `config/default/generic/vnx/authentication.json`.

1.6.5 Passwords

Passwords can be set or changed via the `vnxpasswd` command line tool, see *Tools*.

1.7 Tools

There is a collection of command line tools to communicate with a running program instance.

Common command line parameters are:

-n <node> supply a way to connect to the process. It can be a UNIX socket or a network address in the form `host:port`.

-u <user> authenticate with the given user name. You will be asked to enter the password before the connection is made.

On Linux the tools are found in `/opt/neobotix/pilot-core/bin/` or `/opt/neobotix/pilot-gtkgui/bin/`. On Windows the tools are found in the `bin` folder.

1.7.1 pilot_execute

To execute a custom *Lua Script* on the platform:

```
pilot_execute -n localhost:5555 -u neo-admin -f my_script.lua
```

Above command will run until the script exits. To return immediately add the `-a` option.

Requires permission `UPLOAD_SCRIPT`.

1.7.2 vnxpasswd

To change / set a password:

```
vnxpasswd -c config/local/ -u neo-admin
```

1.7.3 vnxservice

To call a method on a module:

```
vnxservice -n localhost:5555 -u neo-user -x HybridPlanner move_to_station Station1 {
↪"max_velocity": 0.5}
```

To get an overview of available methods:

```
vnxservice -n localhost:5555 -i HybridPlanner
```

1.7.4 vnxread

To convert a `*.dat`, `*.grid` or `*.road` file to JSON:

```
vnxread -f my_road_map.road
```

1.7.5 vnxgraph

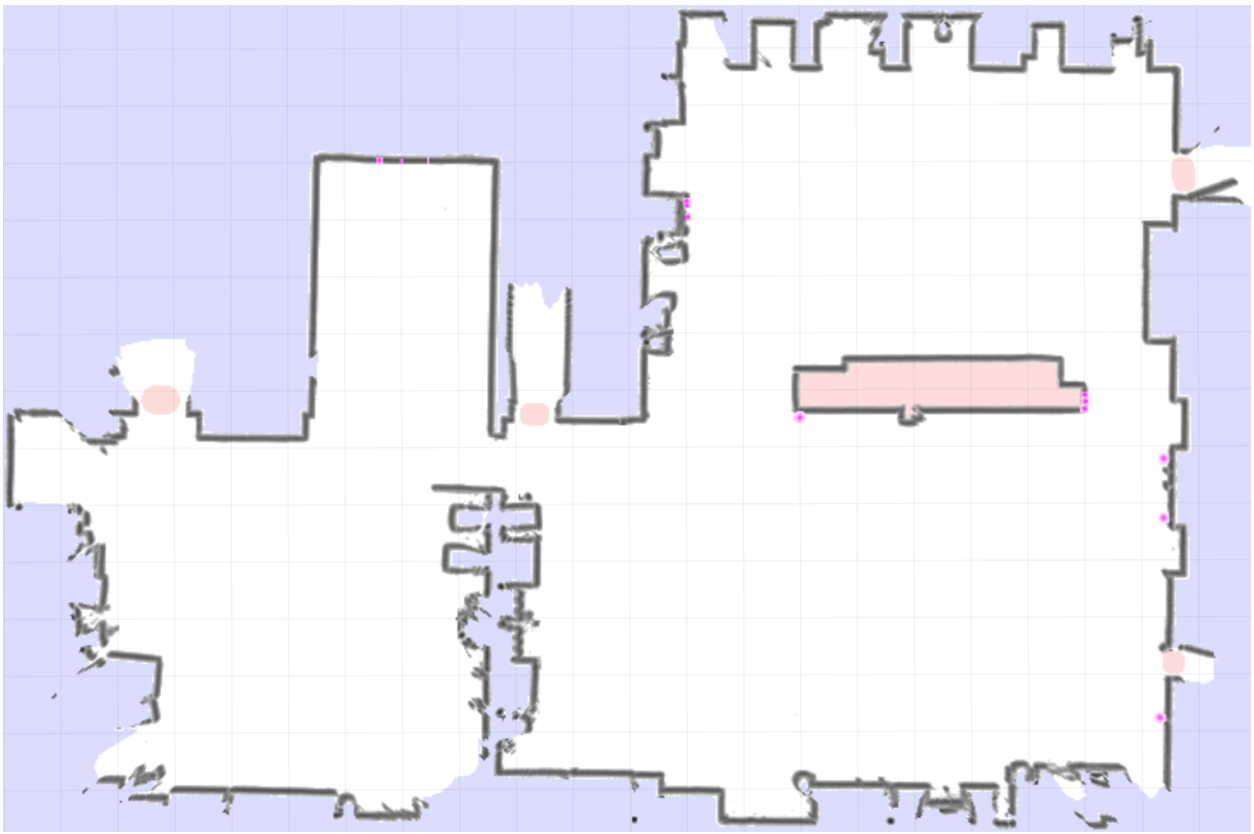
To create a graph of a running VNX process:

```
vnxgraph -n localhost:5555 > graph.dot
dot -Tsvg graph.dot > graph.svg
```

1.8 Grid Map

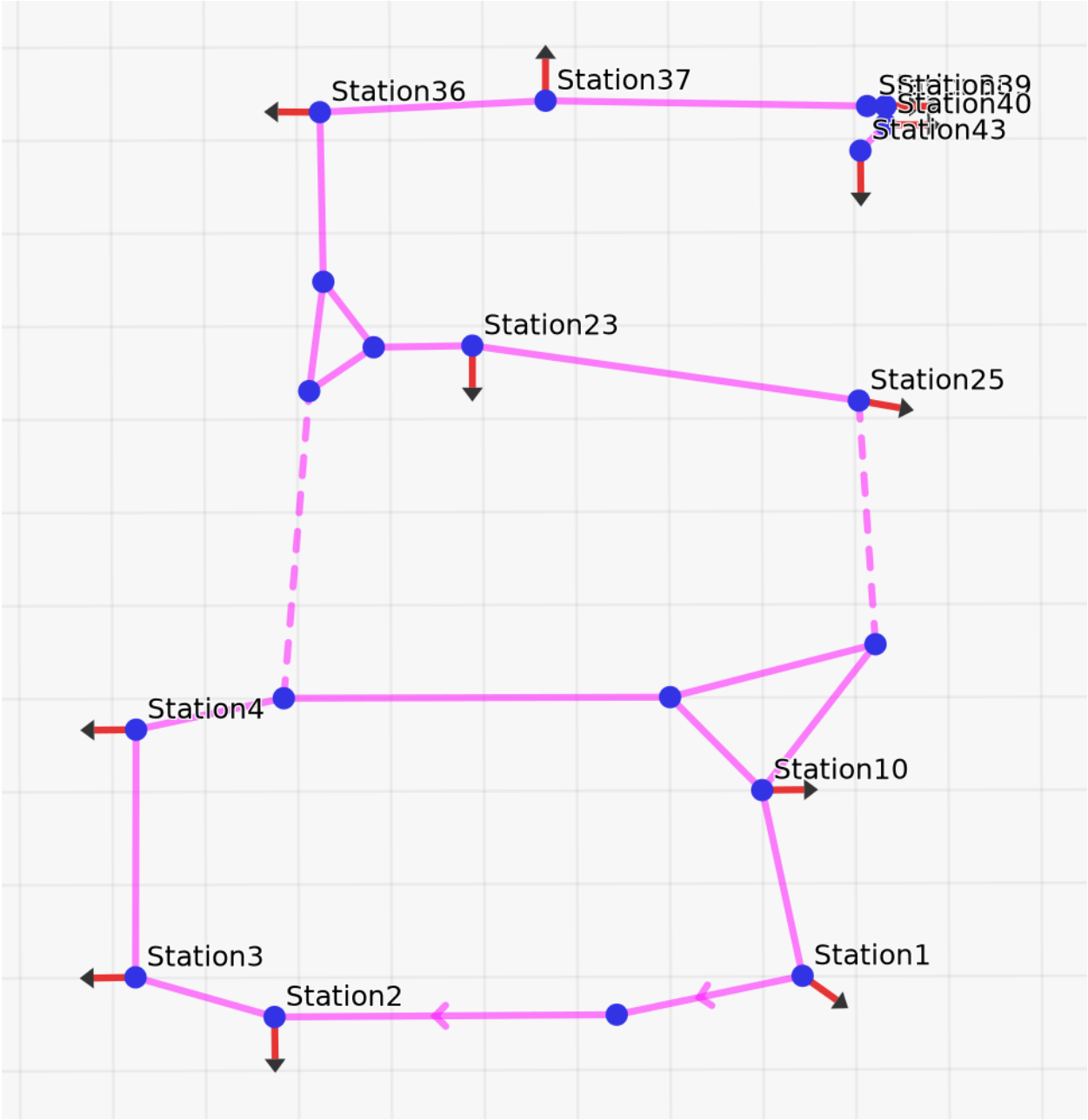
The *Grid Map* is an image representing an occupancy grid map, created via a SLAM mapping algorithm. It has two layers: normal occupancy and reflector occupancy, both consisting of 8-bit data per pixel / cell.

See also: *pilot.GridMapData*, *pilot.OccupancyMapData*



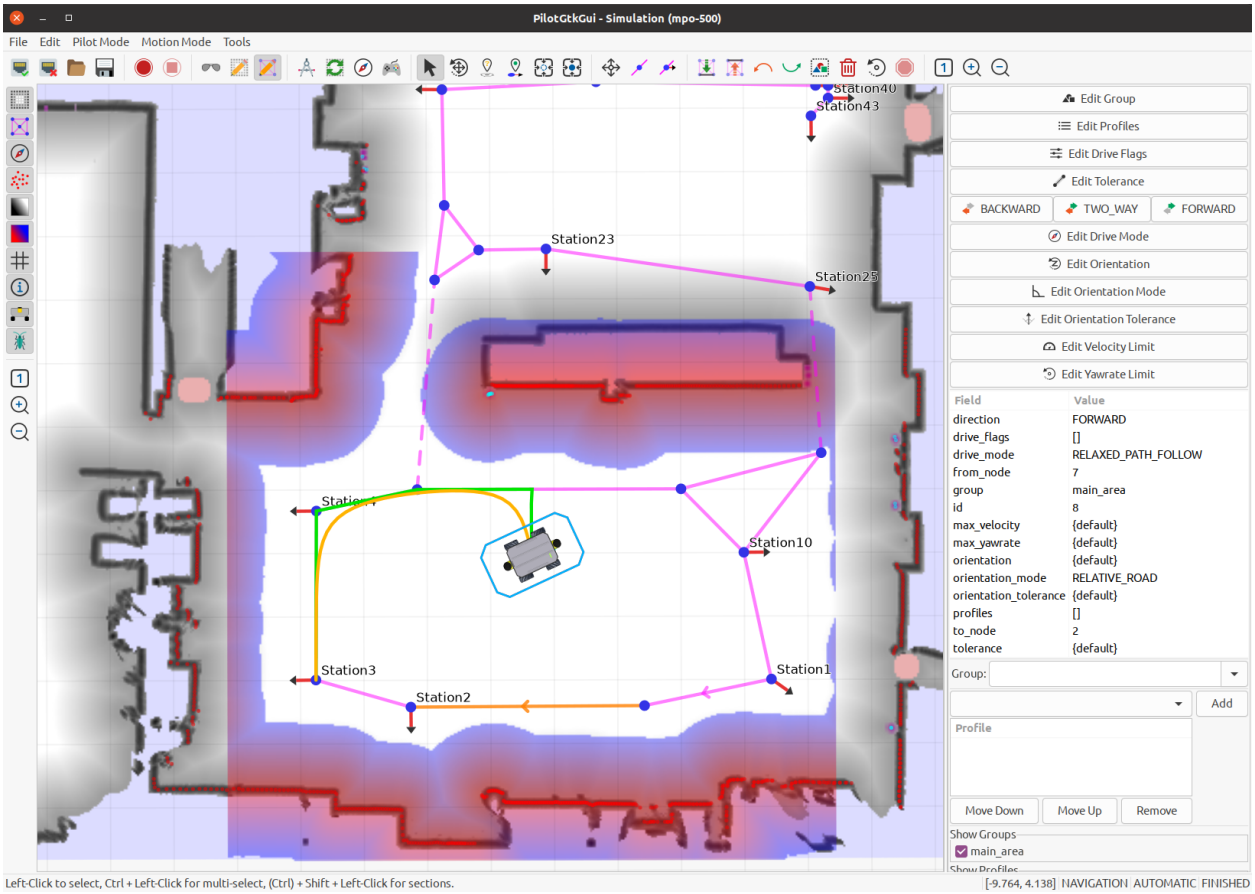
1.9 Road Map

The *Road Map* is a navigation map created by hand using the *GTK GUI*. It contains road segments where a platform should drive on, including a lot of additional parameters that can be set.



1.10 GTK GUI

The *PlatformPilot GTK GUI* provides a means to create maps, upload / download data to / from a platform, as well as visualize various data and control the platform.



1.11 Web GUI

You can interact with the platform using the web interface. You can access it using a web browser of your choice (Firefox and Chrome/Chromium are preferred though) by navigating to *<robot_ip>:8888*.

1.11.1 Toolbar

Toolbar consists of three parts:

App Menu Navigate through the app.

Page title Display information about current page.

Action buttons On every page you will see at least the login / logout button. Other buttons will be explained in the corresponding section.

1.11.2 Map

Changing Pilot Modes

Navigation

To switch into navigation mode:

Navigation mode allows for autonomous operation.

If needed initialize the localization first via the *Pose Estimate Tool*, see below.

Mapping

To create a new *Grid Map* switch to the mapping mode:

Now you can move the platform around using the hardware joystick.

The new *Grid Map* will be created and updated while driving around. When the mapping process is finished you can upload the new map to the platform by clicking the *Save* button.

Note: Requires permissions of *neo-installer* or *neo-admin* when connecting remotely.

Map Update

To update the current *Grid Map* switch to the map update mode:

The platform needs to be localized before entering this mode.

Note: Requires permissions of *neo-installer* or *neo-admin* when connecting remotely.

Set Goal Pose

`Left Click / Tap` and move to define a new goal pose. Red dot on the icon will indicate this mode is enabled.

Set Goal Station

`Left Click / Tap` on a map station to set a new goal station. Red dot on the icon will indicate this mode is enabled.

Cancel Goal

Will abort the current goal and stop immediately.

Pose Estimate Tool

`Left Click / Tap` and move to define a pose estimate (initialize localization). Red dot on the icon will indicate this mode is enabled.

Set Pose Tool

In simulation mode it is possible to "teleport" the platform via:

`Left Click / Tap` and move to define a pose. Red dot on the icon will indicate this mode is enabled.

Data Recording

To start a recording:

Note: The resulting file will be in `user/data/`.

To stop a recording:

Note: This button is only enabled when a recording is active.

Adjust View

This mode activates ovelayed controls for zooming and rotating the view. Also the view can be moved around by `Left Click / Tap` and move.

Settings

In this context menu you can toggle visibility of various map layers.

Note: The same context menu can be accessed by `Right Click / Tap & Hold`.

Tip: Displaying Lidar Points and Local Cost Map is CPU intensive. Disable this layers to reduce CPU / GPU usage.

1.11.3 Logs

On this page the last 100 PlatformPilot log messages can be viewed. It is also possible to stop / resume message polling and filter messages by log level using action buttons.

1.11.4 Task Editor

Introduction

The TaskEditor offers visual way to create and manage tasks for the PlatformPilots TaskHandler module. It allows users to generate Lua programs using graphical blocks by dragging and linking them. It also offers easy access to the PlatformPilots API.

After a little training, the creation of programs is done intuitively. So that even complex sequences of task can be implemented very fast.

The TaskEditor is build on top of the *Blockly* library. Further information:

- https://developers.google.com/blockly
- https://blockly-demo.appspot.com/static/demos/code/index.html
- https://blockly.games/
- https://blockly-demo.appspot.com/static/tests/playground.html

Toolbox

The toolbox is the side menu from whence the user may drag and drop blocks into the workspace.

There are two types of blocks with and without return value. Blocks without return value can be used directly to build a program workflow. Blocks with a return value are used as input for other blocks (i.e. variables or parameters).

PlatformPilot

The following blocks offer direct access to PlatformPilots API. See *Lua Script*.

General

`require` (*module_name*)
Load module defined by `module_name`.

`call` (*function_name*, *{args}*)
Call function named `function_name`.

`args` is an array of arguments passed to the function.

Hardware

`set_relay()`

`set_digital_output()`

`set_display_text()`

`charge()`

Information Requests

`get_position()`

`find_station()`

Log

`log_info()`

`log_error()`

`log_warn()`

Movement

Station List

This block contains a combobox with all stations available in the currently loaded road map.

`move_to_station()`

`move_to_position()`

`move_to()`

`move()`

pilot.goal_options_t

User Input

`wait_for_joystick()`

`wait_for_joystick_button()`

`wait_for_digital_input()`

`wait_ms()`

Core

The following blocks offer access to basic code concepts like variables, logical expressions and loops.

Functions

Lists

Logic

Loops

Math

Text

Variables

1.12 License

PlatformPilot is individually licensed under the terms agreed-upon with Neobotix GmbH.

1.12.1 Third-party software

PlatformPilot makes use of the following third-party software under their respective licenses. Unless stated otherwise, the software packages have not been modified by us and can be retrieved from their project homepages and/or repositories.

- zlib (zlib license)
- libpng (libpng license)
- libjpeg (libjpeg license)

  This software is based in part on the work of the Independent JPEG Group.
- CImg (CeCILL-C license)
- LUA (MIT License)

  Copyright © 1994–2021 Lua.org, PUC-Rio.

  See below for the license text.
- url-cpp (MIT License)

  Copyright (c) 2016-2017 SEOmoz, Inc.

  See below for the license text.
- llhttp (MIT License)

  Copyright Fedor Indutny, 2018.

  See below for the license text.
- Open62541 (Mozilla Public License Version 2.0)
- Eigen3 (Mozilla Public License 2.0)
- OpenCL headers / OpenCL ICD loader (Apache License)

  Copyright (c) 2008-2020 The Khronos Group Inc.

- Several packages by Automy Inc. (MIT License)

  Copyright (c) 2021 Automy Inc.

  See below for the license text.

1.12.2 License Texts

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

TWO

PROGRAMMER'S MANUAL

2.1 VNX Interface

The native VNX binary protocol allows to communicate with the *PlatformPilot* in a C++ application, either via TCP/IP, UNIX socket or a direct intra-procress connection.

The communication is provided via the *vnx.Server* and *vnx.Proxy* modules.

2.1.1 Server

To enable a *vnx.Server* create a config file `config/local/vnx_server_map`:

```
[
        ["TcpServer_1", "0.0.0.0:1234"],
        ["UnixServer_1", "/tmp/mysocket.sock"]
]
```

This will start two VNX servers listening on TCP/IP address `0.0.0.0:1234` and UNIX socket `/tmp/mysocket.sock` respectively. By default a VNX server is listening on TCP/IP address `0.0.0.0:5555`, which requires a login for most funtionality.

To enable user authentication, which is recommended for TCP/IP servers, create a config file `config/local/TcpServer_1.json`:

```
{
        "use_authentication": true,
        "default_access": "OBSERVER"
}
```

This will require a login to gain more than the `default_access` permissions. If `use_authentication` is set to `false` any user has full permissions and no login is necessary.

2.1.2 Proxy

To run a *vnx.Proxy* which connects to another VNX server create a config file `vnx_proxy_map` which is used by your process:

```
[
        ["Proxy_1", "127.0.0.1:5555"],
        ["Proxy_2", "/tmp/mysocket.sock"]
]
```

The above proxies will be available under the module names `Proxy_1` and `Proxy_2`. To import / export topics and forward services create a config file for each proxy. For example `Proxy_1.json`:

```
{
        "import_list": [
                "input",
                "sensors.raw_data",
                "platform.system_state",
                "platform.battery_state",
                "platform.emergency_state",
                "kinematics.drive_state"
        ],
        "export_list": [
                "platform.drive_cmd",
        ],
        "forward_list": [
                "PlatformInterface"
        ],
        "time_sync": true
}
```

To gain required permissions a login needs to be performed at runtime:

```
#include <vnx/vnx.h>
#include <vnx/ProxyClient.hxx>

std::string user;
if(vnx::read_config("user", user)) {
        vnx::ProxyClient proxy("Proxy");
        proxy.login(user, vnx::input_password("Password: "));
}
```

2.1.3 Clients

A *Client* is a way to communicate with a module via (remote) procedure calls.

Synchronization

There are synchronous clients and asynchronous clients. A method call on a synchronous client blocks until the result arrives. A method call on an asynchronous client returns immediately and gives you the possibility to supply callback functions that notify you about the result.

Specific Clients

Every module has a synchronous and an asynchronous client. If the module class is called `MyModule`, the clients are called `MyModuleClient` and `MyModuleAsyncClient`. In order to instantiate them, you also need to know the runtime name of the module (not the name of the class).

Here is an example on how to use the clients for `MyModule` that is running under the name `MyModule_name` to call its method `void do_something(int, string)`.

```
#include <package/MyModuleClient.hxx>
#include <package/MyModuleAsyncClient.hxx>
```

(continues on next page)

(continued from previous page)

```
using namespace package;

MyModuleClient sync_client("MyModule_name");
sync_client.do_something(42, "hello");

std::shared_ptr<MyModuleAsyncClient> async_client =
        std::make_shared<MyModuleAsyncClient>("MyModule_name");
add_async_client(async_client);
async_client->do_something(42, "hello",
        [](void) {
                std::cout << "OK" << std::endl;
        },
        [](const vnx::exception& ex) {
                std::cout << "FAIL: " << ex.what() << std::endl;
        }
);
```

The callback functions for asynchronous clients are of the forms `void(const T&)` (where *T* is the return type, or *void*) and `void(const vnx::exception&)` respectively. Supplying either of them is optional and defaults to a null function.

Generic Clients

With a generic client, you can communicate with any module. The handling is similar to the specific case, only this time you call methods by their name and supply the parameters as a *vnx.Object*. The names of the method and the parameters have to match the interface definition.

```
#include <vnx/vnx.h>

vnx::Object args;
args["number"] = 42;
args["message"] = "hello");

vnx::GenericClient sync_client("MyModule_name");
std::shared_ptr<const vnx::Value> ret = sync_client.call("do_something", args);
std::cout << "OK: " << ret->get_field_by_index(0) << std::endl;

std::shared_ptr<vnx::GenericAsyncClient> async_client =
        std::make_shared<vnx::GenericAsyncClient>("MyModule_name");
add_async_client(async_client);
async_client->call("do_something", args,
        [](std::shared_ptr<const vnx::Value> ret) {
                std::cout << "OK: " << ret->get_field_by_index(0) << std::endl;
        },
        [](const vnx::exception& ex) {
                std::cout << "FAIL: " << ex.what() << std::endl;
        }
);
```

The callback functions are of the form `void(std::shared_ptr<const vnx::Value>)` and `void(const vnx::exception&)` respectively. Supplying either of them is optional and defaults to a null function.

2.1.4 Compiling

Example on how to compile your C++ application:

```
g++ -std=c++11 application.cpp -I /opt/neobotix/pilot-core/include \
        -L /opt/neobotix/pilot-core/lib -lpilot_core -lvnx_base
```

`/opt/neobotix/pilot-core` can also be replaced by `/opt/neobotix/pilot-gtkgui`, if you have the *GTK GUI* installed.

2.1.5 Example

Connecting to the *PlatformPilot* and sending goals to *HybridPlanner*.

`config/default/vnx_proxy_map`:

```
[["Proxy", ""]]
```

We do not specify an address in the config above, since we want to give a custom address on the command line later.

`config/default/Proxy.json`:

```
{"forward_list": ["HybridPlanner"]}
```

`move_to_stations.cpp`:

```
#include <vnx/vnx.h>
#include <vnx/ProxyClient.hxx>
#include <pilot/HybridPlannerClient.hxx>
using namespace pilot;

int main(int argc, char** argv)
{
    std::map<std::string, std::string> options;
    options["u"] = "user";
    vnx::init("move_to_stations", argc, argv, options);

    std::string user;
    if(vnx::read_config("user", user)) {
        vnx::ProxyClient proxy("Proxy");
        proxy.login(user, vnx::input_password("Password: "));
    }

    HybridPlannerClient planner("HybridPlanner");

    goal_options_t goal_options;
    goal_options.max_velocity = 0.5;
    goal_options.drive_mode = drive_mode_e::STRICT_PATH_FOLLOW;

    while(vnx::do_run()) {
        try {
            planner.move_to_station("Station1");
            vnx::log_info() << "Station1 has been reached!";

            planner.move_to_station("Station2", goal_options);
            vnx::log_info() << "Station2 has been reached!";

            const vnx::Hash64 job_id = vnx::Hash64::rand();
            planner.move_to_station("Station3", {}, job_id);
            vnx::log_info() << "Station3 has been reached! (job " << job_id << ")";
        }
```

(continues on next page)

(continued from previous page)

```
        catch(const std::exception& ex) {
            vnx::log_error() << "move_to_station() failed with: " << ex.what();
            break;
        }
    }
    vnx::close();
}
```

Compiling:

```
g++ -std=c++11 -o move_to_stations move_to_stations.cpp \
        -I /opt/neobotix/pilot-core/include \
        -L /opt/neobotix/pilot-core/lib -lpilot_core -lvnx_base
```

Running:

```
$ ./move_to_stations -c config/default/ --Proxy.address ~/pilot/.pilot_main.sock

Proxy.address = "/home/neobotix/pilot/.pilot_main.sock"
Proxy.forward_list = ["HybridPlanner"]
config = ["config/default/"]
vnx_proxy_map = [["Proxy", ""]]
[Proxy] INFO: enable_forward('HybridPlanner', 100, 1000)
[Proxy] INFO: Connected to /home/neobotix/pilot/.pilot_main.sock
[move_to_stations] INFO: Station1 has been reached!
[move_to_stations] INFO: Station2 has been reached!
[move_to_stations] INFO: Station3 has been reached! (job 13373535902967275021)
...
```

Connecting to `.pilot_main.sock` avoids having to login with a username and password to be able to move the platform.

2.2 JSON-RPC Interface

This interface uses the JSON-RPC protocol in version 2.0.

The corresponding proxy module *vnx.JRPC_Proxy* is similar to the *vnx.Proxy* and mostly follows the same protocol, only it wraps its messages into appropriate JSON objects.

2.2.1 Message format

A message in our setting of the JSON RPC protocol may look like this:

```
{
        "jsonrpc": "2.0",
        "id": "123some_id456",
        "method": "MyModule.some_method",
        "params": {
                "first": "Hello",
                "second": 4321
        }
}
```

Some remarks:

- The `id` is optional. If you provide it, you will get a response message, otherwise you will get nothing. This includes error messages!
- `method` consists of a module name and its method name, separated by a dot. If you omit the module name (but keep the dot), the request is directed at the proxy module itself.

  The proxy module has a method `select_service` to assign a module name for the rest of the connection:

```
{
        "jsonrpc": "2.0",
        "id": "123some_id456",
        "method": ".select_service",
        "params": {
                "service_name": "MyModule"
        }
}
```

  Now you can omit the module name (and the dot) in future requests.
- `params` is an object indexed by the names of the parameters as defined in the interface of the module. You can also supply a list instead, although that is not encouraged.

2.2.2 Server

To enable a *vnx.JRPC_Server* create a config file `config/local/vnx_jrpc_server_map`:

```
[
        ["JRPC_Server", "0.0.0.0:5556"],
        ...
]
```

This will start a JSON RPC server that listens on the TCP/IP address `0.0.0.0:5556`.

2.2.3 Connecting

Open a TCP connection to the port of the server. You should be greeted with a welcome message (method `.on_remote_connect`) that you can safely ignore.

To authenticate use the `on_login` method of the proxy:

```
{
        "jsonrpc": "2.0",
        "method": ".on_login",
        "params": {
                "name": "username",
                "password": "23f/*...*/995"
        }
}
```

where `password` is a SHA-256 hash of the actual password. On success you should receive a call to `.on_remote_login` providing some information on your established session.

2.2.4 Return Values

Note: You will only receive return values if you supply the `id` field in your request.

Return values are wrapped inside a success message with the `result` field holding the actual return value. A successful call to `RoadMapPlanner.find_station` may look like this:

```
{
        "__type": "vnx.JRPC_Success",
        "jsonrpc": "2.0",
        "id": 5566,
        "result": {
                "__type": "pilot.MapStation",
                "drive_flags": [],
                "goal_tolerance": {"__type": "pilot.vector_3f_param_t", "type":
↪"DEFAULT", "x": {"__type": "pilot.float_param_t", "type": "DEFAULT", "value": 0}, "y
↪": {"__type": "pilot.float_param_t", "type": "DEFAULT", "value": 0}, "z": {"__type
↪": "pilot.float_param_t", "type": "DEFAULT", "value": 0}},
                "goal_tune_time": {"__type": "pilot.float_param_t", "type": "DEFAULT",
↪ "value": 0},
                "group": "",
                "id": 120,
                "name": "Station120",
                "orientation": -3.13343,
                "position": [-0.611983, 2.81928],
                "profiles": [],
                "tolerance": {"__type": "pilot.float_param_t", "type": "DEFAULT",
↪"value": 0}
        }
}
```

The result of calling a `void` method has a `null` return value:

```
{
        "__type": "vnx.JRPC_Success",
        "jsonrpc": "2.0",
        "id": 43,
        "result": null
}
```

2.2.5 Errors

Note: You will only receive errors if you supply the `id` field in your request.

If your request failed, you will get an error response. It contains an `error` field with an object giving an error code, a short message and the actual error object.

When the method you called throws an exception, the error will look like this:

```
{
        "__type": "vnx.JRPC_Failure",
        "jsonrpc": "2.0",
        "id": 43,
        "error": {
                "__type": "vnx.JRPC_Error",
```

(continues on next page)

(continued from previous page)

```
                "code": 500,
                "message": "roadmap station does not exist",
                "data": {
                        "__type": "vnx.InternalError",
                        "what": "roadmap station does not exist"
                }
        }
}
```

This shows an error that you get if you lack a permission to do something:

```
{
        "__type": "vnx.JRPC_Failure",
        "jsonrpc": "2.0",
        "id": 42,
        "error": {
                "__type": "vnx.JRPC_Error",
                "code": 403,
                "message": "permission denied (pilot.permission_e.MOVE)",
                "data": {
                        "__type": "vnx.PermissionDenied",
                        "what": "permission denied (pilot.permission_e.MOVE)",
                        "dst_mac": 0,
                        "method": "pilot.HybridPlanner.set_goal_station",
                        "permission": "pilot.permission_e.MOVE"
                }
        }
}
```

See *vnx.JRPC_Error* for a list of error codes.

2.2.6 Topics

As with the VNX protocol, subscribing to a topic is done with the `enable_export` method of the proxy:

```
{
        "jsonrpc": "2.0",
        "method":".enable_export",
        "params": {
                "topic_name": "some.topic"
        }
}
```

Received samples will have the following form:

```
{
        "jsonrpc": "2.0",
        "method": "!some.topic",
        "params": {
                "seq_num": 42,
                "value": {"__type": "some.Datatype", "time": 1600000023, "some_more":
↪"fields"}
        }
}
```

Notice that we are relaxing the notion of a "method" here: The initial `!` makes clear that it actually refers to a topic and the parameters really just hold the value.

Using the same format in the other direction, you can also publish samples on the named topic (given sufficient permissions).

2.3 HTTP Interface

The HTTP REST API provides access to almost any functionality of the *PlatformPilot* via the HTTP protocol.

The HTTP server is enabled by default on port `8888`, however it can be disabled by setting `enable_http_server` to `false`.

The REST API is available on the path `http://localhost:8888/api/`. (replace *localhost* with your target machine)

See also *HttpProxy*.

2.3.1 Login

Some functionality requires special permissions, see *User Management*.

To gain necessary permissions you need to login to the HTTP server as follows:

```
curl -I "http://localhost:8888/server/login?user=neo-user&passwd_plain=neobotix"
HTTP/1.1 200 OK
...
Set-Cookie: hsid=7ac1b14c66b6f323-0000d026d3eff249-2d4c9774cd3accb8; Path=/; Max-
↪Age=86400; SameSite=Strict;
```

The response will contain a session cookie which can be used as follows:

```
curl -H "Cookie: hsid=7ac1b14c66b6f323-0000d026d3eff249-2d4c9774cd3accb8" http://
↪localhost:8888/api/request/...
```

2.3.2 Services

All modules of the *PlatformPilot* are available via the path `/api/request/`.

To get an overview of the available modules:

```
curl http://localhost:8888/api/request/
["GlobalCostMap/", "GlobalPlanner/", "GridLocalization/", "HybridPlanner/", ...]
```

The available methods of a module can be queried as follows:

```
curl http://localhost:8888/api/request/HybridPlanner/
["append_goal", "append_goal_position", "append_goal_positions", "append_goal_station
↪", ...]
```

A method can be called as follows:

```
curl -X POST http://localhost:8888/api/request/PilotServer/get_state
{"__type": "pilot.PilotState", ...}

curl -H "Cookie: ..." -X POST -d '{"name": "Station1"}' \
        http://localhost:8888/api/request/HybridPlanner/set_goal_station
```

The parameters of a function are supplied as a JSON object via POST data.

You may need to be logged in to access certain functions, see above.

2.3.3 Topics

Almost all topics of the *PlatformPilot* are available via the path `/api/topic/`.

To get an overview of the available topics:

```
curl http://localhost:8888/api/topic/
["input/", "local_planner/", "navigation/", "platform/", "sensors/", "task_handler/",
↪"tf/", "tfd/", "vnx/"]

curl http://localhost:8888/api/topic/platform/
["info", "odometry", "pilot_state", "system_state"]
```

To get the latest sample data of a topic:

```
curl http://localhost:8888/api/topic/platform/odometry
{"__type": "pilot.Odometry", "time": 1613656049067588, ...}
```

To get a tree of the lastest sample data of a domain:

```
curl http://localhost:8888/api/topic/platform
{"info": {"__type": "pilot.PlatformInfo", ...}, ...}
```

To publish a data sample on a topic:

```
curl -H "Cookie: ..." -X POST -d '{"topic": "test.topic", "sample": {"__type": "pilot.
↪Pose2D", ...}}' \
        http://localhost:8888/api/request/HttpProxy/publish
```

Note that a `__type` field needs to be specified containing the type name of the sample, such as `pilot.Pose2D` for example.

2.3.4 Configuration

The current configuration tree can be viewed via the path `/api/config/`.

To access protected configuration values, a special permission *PROTECTED_CONFIG* is required, see *vnx.permission_e*.

To get an overview of the available config options:

```
curl http://localhost:8888/api/config/
["GlobalCostMap/", "GlobalPlanner/", "GridLocalization/", "GridMapping/", ...]

curl http://localhost:8888/api/config/GridLocalization/
["broadcast_tf", "confidence_gain", "constrain_threshold", "constrain_threshold_yaw",
↪"gain_factor", ...]
```

(continues on next page)

(continued from previous page)

To query a specific config value:

```
curl http://localhost:8888/api/config/GridLocalization/gain_factor
0.01
```

To query a sub-tree of the configuration:

```
curl http://localhost:8888/api/config/GridLocalization
{"broadcast_tf": true, "confidence_gain": 0.01, "constrain_threshold": 0.1,
↪"constrain_threshold_yaw": 0.2, "gain_factor": 0.01, ...}
```

To query the entire configuration tree:

```
curl http://localhost:8888/api/config
{...}
```

2.3.5 Log

The terminal output log messages are available via the path `/api/log/`.

Each message is of type *vnx.LogMsg*.

To get all errors which occured since startup:

```
curl http://localhost:8888/api/log/errors
[...]
```

To get all recent messages:

```
curl http://localhost:8888/api/log/recent
{"1": [...], "2": [...], "3": [...], "4": [...]}
```

To get recent warnings only:

```
curl http://localhost:8888/api/log/recent/2
[...]
```

The log levels are as follows:

- 1 = ERROR
- 2 = WARN
- 3 = INFO
- 4 = DEBUG

Log messages are also written to disk and can be accessed via `http://example:8888/user/data/logs/`.

2.3.6 Events

A history of events is available via the path `/api/events/`.

Each entry is of type *pilot.Event* or any of it's derived types such as *pilot.Incident*.

To get a list of recent events:

```
curl http://localhost:8888/api/events/recent
[...]
```

To get recent error events only:

```
curl http://localhost:8888/api/events/errors
[...]
```

The latest events are at the end of the respective list.

2.3.7 Views

For certain data types there are special views available via the path `/api/view/`.

To get a cost map (see *pilot.CostMapData*) as a PGN image:

```
curl "http://localhost:8888/api/view/cost_map?topic=navigation.local_cost_map&
↪color=true&alpha=128" > local_cost_map.png
```

To get a grid map (see *pilot.OccupancyMapData*) as a PGN image:

```
curl "http://localhost:8888/api/view/occupancy_map?topic=navigation.grid_map&alpha=255
↪" > grid_map.png
```

2.4 Lua Script

2.4.1 Command Reference

The following commands are made available to Lua scripts by the *TaskHandler* module.

Commands denoted `void` return `true` or `false` depending on whether the command succeeded. Commands with a return type either return a value of that type or `nil` if the command failed.

See https://en.wikipedia.org/wiki/Lua_(programming_language) for more information regarding the Lua script language itself.

Movement

The commands in this section take an optional parameter of type *pilot.goal_options_t*. If not given, the default values are used.

void **`move_to_station`** (string *name*, goal_options_t *options*)
: Moves the platform to the station named `name`.

void **`move_to_position`** (Pose2D *position*, goal_options_t *options*)
: Moves the platform to the given position. See *pilot.Pose2D*.

void **`move_to`** (MapStation *station*, goal_options_t *options*)
: Moves the platform to the place described by `station`. It may or may not be a station of the Road Map. See *pilot.MapStation*.

void **`move`** (double *dx*, double *dy*, double *dr*, goal_options_t *options*)
: Moves the platform in the given direction, as seen from the platform's point of view (relative to `base_link`). See *Coordinate Systems*.

void **cancel_goal**()
Cancels the current goal (if any) which causes the platform to stop moving and the corresponding move command to fail. This command can be useful in an event handler.

User Input

int **wait_for_joystick**()
Waits until any button on the active joystick is pressed and returns the ID of the button.

void **wait_for_joystick_button**(int *button*)
Waits until the button with ID `button` is pressed on the active joystick.

void **wait_for_digital_input**(int *channel*, bool *state*)
Waits until the digital input `channel` reaches state `state`. The digital inputs are enumerated from 0 to 15.

void **wait_ms**(int *period*)
Pauses for `period` milliseconds.

void **wait_sec**(int *period*)
Pauses for `period` seconds.

void **wait_min**(int *period*)
Pauses for `period` minutes.

void **wait_hours**(int *period*)
Pauses for `period` hours.

Hardware

void **set_relay**(int *channel*, bool *state*)
Sets the relay with id `channel` to state `state`.

void **set_digital_output**(int *channel*, bool *state*)
Sets the digital output with id `channel` to state `state`. The digital outputs are enumerated from 0 to 15.

void **set_display_text**(string *text*)
Prints `text` on the first line of the LCD display. There is enough space for 20 characters.

void **charge**()
Starts the charging process. Fails immediately if no charger is detected. Otherwise blocks until the charging is either finished or aborted.

void **start_charging**()
Starts the charging process, i.e. activates the corresponding relay.

void **stop_charging**()
Stops the charging process, i.e. deactivates the corresponding relay.

Information Requests

int **get_time_sec**()
Returns the current time stamp in seconds.

int **get_time_millis**()
Returns the current time stamp in milliseconds.

int **get_time_micros**()
Returns the current time stamp in microseconds.

Pose2D **get_position** ()
Returns the current position on the map. See *pilot.Pose2D*.

MapNode **find_station** (string *name*)
Finds and returns the node / station called `name`. See *pilot.MapStation* and See *pilot.MapNode*.

Log

void **log_info** (string *message*)
Generates a log message with priority *INFO*.

void **log_warn** (string *message*)
Generates a log message with priority *WARN*.

void **log_error** (string *message*)
Generates a log message with priority *ERROR*.

Control Flow

void **block** ()
Pauses execution of the main thread.

> **Warning:** Only use in event handlers!

void **unblock** ()
Resumes execution of the main thread.

> **Warning:** Only use in event handlers!

Builtin Functions

bool **auto_charge** (string *pre_stage1*, string *pre_stage2*, string *charge_station*, float *undock_distance*, float *max_velocity*)
Automatically docks at a specified charging station, charges the batteries until full and then undocks from the station.

`pre_stage1` is a map station somewhere close (less than 1 m) to the charging station with an orientation close to the final docking pose. There should still be enough space to rotate fully without hitting the charging station.

`pre_stage2` is a map station from where to start the docking process without having to rotate anymore. There should be about 10 to 20 cm of space between the contacts at this position.

`charge_station` is a map station where the platform makes contact with the charging station. It should be specified very precisely, within a few mm of accuracy.

`undock_distance` is the amount of distance to move backwards after finishing the charging process. The default is 0.25 m. Make sure the robot is allowed to move this much backwards, in case of a differential platform without a second laser scanner. `max_velocity` is the maximum velocity in [m/s] with which to dock and undock. The default is 0.05 m/s.

Returns true if successful, false otherwise.

A `require 'neobotix'` is needed to access this function in Lua script. Permissions `MOVE` and `CHARGE` are required, see *pilot.permission_e*.

void **reset_motors**()

Attempts to re-activate motors (clears any latched erros). Same action as when releasing EM stop.

A `require 'neobotix'` is needed to access this function in Lua script. Permission `MOVE` is required, see *pilot.permission_e*.

OPC-UA Functions

UA node ids are specified in Lua via an array of two values. Numeric and string node ids are supported as follows: `{0, 1337}` or `{1, "MyObject"}`.

A `require 'neobotix'` is needed to access these functions in Lua script.

Variant **opc_ua_call**(string *proxy*, pair<ushort, Variant> *object*, string *method*, vector<Variant> *args*)

Performs an OPC-UA call via the specified `proxy` and returns the result of it.

`proxy` is the name of a running *vnx.opc_ua.Proxy* module, see *opcua_proxy_map*.

`object` is an optional UA node id of the object for which to call the method. Can be set to `nil` in order to call a global method.

`method` is the method name (OPC-UA browse name).

`args` is an array of function parameters.

In case of failure `nil` or `false` is returned, depending on if the method has a return value (*nil*) or not (*false*). In case of multiple return values an array is returned.

Variant **opc_ua_read**(string *proxy*, pair<ushort, Variant> *object*, string *variable*)

Reads an OPC-UA variable via the specified `proxy`.

`proxy` is the name of a running *vnx.opc_ua.Proxy* module, see *opcua_proxy_map*.

`object` is a UA node id of the object containing the variable. Setting it to `nil` is equivalent to calling `opc_ua_read_global`.

`variable` is the variable name (OPC-UA browse name).

Returns the value read, or `nil` in case of failure.

Variant **opc_ua_read_global**(string *proxy*, pair<ushort, Variant> *variable*)

Reads a global OPC-UA variable via the specified `proxy`.

`proxy` is the name of a running *vnx.opc_ua.Proxy* module, see *opcua_proxy_map*.

`variable` is a UA node id of a global variable.

Returns the value read, or `nil` in case of failure.

bool **opc_ua_write**(string *proxy*, pair<ushort, Variant> *object*, string *variable*, Variant *value*)

Writes a value to an OPC-UA variable via the specified `proxy`.

`proxy` is the name of a running *vnx.opc_ua.Proxy* module, see *opcua_proxy_map*.

`object` is a UA node id of the object containing the variable. Setting it to `nil` is equivalent to calling `opc_ua_write_global`.

`variable` is the variable name (OPC-UA browse name).

`value` is the value to be written.

Returns `true` on success, `false` in case of failure.

bool **opc_ua_write_global** (string *proxy*, pair<ushort, Variant> *variable*, Variant *value*)
Writes a value to a global OPC-UA variable via the specified `proxy`.

`proxy` is the name of a running *vnx.opc_ua.Proxy* module, see *opcua_proxy_map*.

`variable` is a UA node id of a global variable.

`value` is the value to be written.

Returns `true` on success, `false` in case of failure.

Advanced

Variant **execute** (string *module*, string *method*, Object *params*)
Executes a function `method` of the module `module`. The parameters are specified as key-value pairs in `params`.

If `method` does not have a return type, the command returns `true` or `false`, depending on whether it succeeded. If `method` does have a return type, either a value of that type is returned or `nil` on failure.

See *vnx.Variant* and *vnx.Object*.

> **Warning:** This command allows almost unlimited access to the functionality of *PlatformPilot*. However, it also allows you to leave the boundaries of safe operation and must therefore be considered dangerous.

2.4.2 Event Handlers

The following functions, if defined in your script, will be called upon specific events. Events are queued and the next event handler will only be called after the current one finished.

Be aware that the calls happen asynchronously to the main thread. The two executions run in parallel while abiding to Lua's cooperative multithreading model. You can use the `block()` and `unblock()` functions to avoid multithreading issues.

void **on_em_stop** ()
Is executed when the emergency stop button is pushed.

void **on_scanner_stop** ()
Is executed when the scanner emergency stop is triggered.

void **on_em_reset** ()
Is executed when a previous emergency situation (button and/or scanner) is resolved and the platform can move again.

void **on_joystick_button_pressed** (int *button*)
Is executed when the joystick button with ID `button` changes its state from not pressed to pressed.

void **on_joystick_button_released** (int *button*)
Is executed when the joystick button with ID `button` canges its state from pressed to not pressed.

void **on_digital_input_on** (int *channel*)
Is executed when the digital input `channel` changes its state from off to on.

void **on_digital_input_off** (int *channel*)
Is executed when the digital input `channel` changes its state from on to off.

void **on_battery_low** ()
Is executed when the battery drops below a low level.

void **on_battery_critical**()
Is executed when the battery drops below a critical level.

2.4.3 Examples

Move to stations in the Road Map in a loop:

```
function main()
        while true do
                move_to_station("Station4");
                move_to_station("Station9");
                move_to_station("Station11");
                wait_ms(1000);
        end
end
```

Move randomly to any of the specified stations in a loop, abort in case of failure, stop when the Y button on the joystick is pressed:

```
function odyssey(stations)
        repeat
                wait_ms(1000);
                index = math.random(1, #stations)
        until not move_to_station(stations[index])
end

function on_joystick_button_pressed(button)
        if button == 3 then
                cancel_goal()
        end
end

function main()
        odyssey({"Station20", "Station10", "Station12", "Station14", "Station1"})
end
```

Dock to a charging station using special parameters:

```
move_to_station("ChargeStation", {
                max_velocity = 0.1,
                drive_flags = {"IGNORE_FOOTPRINT", "DISABLE_ROTATION"}
})
```

Undock from a charging station using a relative move command with special parameters:

```
move(-0.25, 0, 0, {
                max_velocity = 0.1,
                drive_flags = {"IGNORE_FOOTPRINT", "DISABLE_ROTATION"}
})
```

Using the built-in `auto_charge(...)` function to dock, charge and undock:

```
require 'neobotix'

function main()
        auto_charge("PreStage1", "PreStage2", "ChargeStation", 0.25, 0.05);
end
```

Calling an OPC-UA method:

```
require 'neobotix'

function main()
        ret = opc_ua_call("OPC_UA_Proxy", {1, "vnx.process"}, "get_name")
        log_warn(ret) --> "pilot_main"
        ret = opc_ua_call("OPC_UA_Proxy", {1, "HybridPlanner"}, "set_goal_stations", {
                {"Station4", "Station3", "Station2"}
        })
end
```

With `opcua_proxy_map` set to `[["OPC_UA_Proxy", "opc.tcp://127.0.0.1:4840"]]` and `enable_opcua_server` set to `true`.

Reading an OPC-UA variable:

```
require 'neobotix'

function main()
        ret = opc_ua_read("OPC_UA_Proxy", {0, 2253}, "ServerArray")
        log_warn(ret) --> ["urn:open62541.server.application"]
end
```

With `opcua_proxy_map` set to `[["OPC_UA_Proxy", "opc.tcp://127.0.0.1:4840"]]` and `enable_opcua_server` set to `true`.

2.5 OPC-UA Interface

The OPC-UA interface is provided by the *vnx.opc_ua.Server* and *vnx.opc_ua.Proxy* modules.

They allow to access internal modules via OPC-UA method calls, as well as call methods on another OPC-UA server via a *Lua Script* for example.

2.5.1 Server

To enable the *vnx.opc_ua.Server* set the following config option:

```
cd ~/pilot
echo true > config/local/enable_opcua_server
```

The server will listen on the address `opc.tcp://0.0.0.0:4840`.

By default the following options are set:

```
{
        "export_services": [
                "PilotServer",
                "MovechainHandler",
                "TaskHandler",
                "HybridPlanner",
                "GlobalPlanner",
                "RoadMapPlanner",
                "LocalPlanner",
                "PlatformInterface",
```

(continues on next page)

(continued from previous page)

```
                "vnx.process"
        ],
        "export_topics": [
                ...
        ],
        "use_authentication": true,
        "default_access": "USER"
}
```

To add more modules and topics to the interface create a config file `config/local/OPC_UA_Server.json` as follows:

```
{
        "export_services+": [
                "AnotherModule",
                ...
        ],
        "export_topics+": [
                "another.topic",
                ...
        ]
}
```

2.5.2 Proxy

To run a *vnx.opc_ua.Proxy* which connects to another OPC-UA server create the following config file `config/local/opcua_proxy_map`:

```
[
        ["OPC_UA_Proxy_1", "opc.tcp://127.0.0.1:4840"],
        ...
]
```

The above proxy will be available unter the module name `OPC_UA_Proxy_1`, see *Lua Script* for examples on how to use it.

2.5.3 Data Types

Primitive data types are directly mapped to their OPC-UA counter parts, such as `int` to `INT32`, `float` to `FLOAT`, etc. `string` is directly mapped to a OPC-UA `STRING`. Arrays of said types are directly mapped to OPC-UA arrays.

Anything else will be converted to JSON and transported via a `LocalizedText` object, with the `locale` set to `JSON`.

2.6 ROS Bridge

The ROS Bridge allows to integrate *PlatformPilot* into a ROS environment, such that it is possible to control the platform via ROS as well as visualize all data in RViz.

2.6.1 Installation

It is assumed that a ROS workspace has already been setup on the platform's PC, for example `~/ros_workspace/`. See https://docs.neobotix.de/display/TUT/Client+PC+Setup for more information.

To install the ROS Bridge:

```
cd ~/ros_workspace/src/
git clone https://github.com/neobotix/pilot-ros-bridge.git
cd ~/ros_workspace/
catkin_make
```

The package depends on an already installed `neobotix-pilot-core` or `neobotix-pilot-gtkgui` package.

2.6.2 Running

To run the ROS bridge on the platform:

```
source ~/ros_workspace/devel/setup.bash
roslaunch pilot_ros_bridge mpo_700.launch
```

Replace `mpo_700.launch` with `mpo_500.launch` or `mp_400.launch`, depending on your platform.

It is also possible to run the ROS bridge on another PC, by adapting the `pilot_node` param in the launch file:

```
<param name="pilot_node" type="str" value="192.168.0.50:5555"/>
```

Replace `192.168.0.50` with the actual IP address of the platform.

In addition you can disable user authentication on the TCP server to allow for all functionality via the ROS Bridge. To do this create a config file `config/local/TcpServer.json`:

```
{
        "use_authentication": false
}
```

This is only necessary when running the ROS Bridge on another PC.

2.6.3 Topics

The following topics are enabled on the ROS bridge by default. You can add more by adapting the config file `config/default/generic/Pilot_ROS_Bridge.json` in `pilot-ros-bridge`.

Export

```
"export_map": [
        ["platform.odometry",                   "/odom"],
        ["sensors.laser_scan.lidar_1",          "/lidar_1/scan"],
        ["sensors.laser_scan.lidar_2",          "/lidar_2/scan"],
        ["sensors.filtered_scan.lidar_1",       "/lidar_1/scan_filtered"],
        ["sensors.filtered_scan.lidar_2",       "/lidar_2/scan_filtered"],
        ["kinematics.drive_state",              "/drives/joint_states"],
        ["mapping.grid_map",                    "/mapping/map"],
        ["mapping.grid_map_tile",               "/mapping/map_tile"],
```

(continues on next page)

(continued from previous page)

```
        ["mapping.grid_map_tile_ref",           "/mapping/map_tile_ref"],
        ["navigation.grid_map",                 "/map"],
        ["navigation.grid_map_tile",            "/map_tile"],
        ["navigation.map_pose",                 "/map_pose"],
        ["navigation.map_particles",            "/particlecloud"],
        ["navigation.road_map",                 "/road_map"],
        ["navigation.global_path",              "/global_path"],
        ["navigation.local_path",               "/local_path"],
        ["navigation.local_cost_map",           "/local_cost_map"],
        ["navigation.local_cost_map_overlay",   "/local_cost_map_overlay"],
        ["navigation.global_cost_map",          "/global_cost_map"],
        ["navigation.global_cost_map_overlay",  "/global_cost_map_overlay"],
        ["local_planner.target_pose",           "/local_planner/target_pose"],
        ["local_planner.predicted_pose",        "/local_planner/predicted_pose"]
]
```

Import

```
"import_map": [
        [["/cmd_vel", "geometry_msgs/Twist"],           "platform.velocity_cmd"],
        [["/initialpose", "geometry_msgs/PoseWithCovarianceStamped"],   "navigation.
↪initial_pose"],
        [["/move_base_simple/goal", "geometry_msgs/PoseStamped"],       "navigation.
↪new_goal_pose"]
]
```

CHAPTER

THREE

API REFERENCE

3.1 Common Datatypes

`void` empty space

`bool` 8-bit boolean (default = *false*)

`char` 8-bit signed integer (default = *0*)

`uchar` 8-bit unsigned integer (default = *0*)

`short` 16-bit signed integer (default = *0*)

`ushort` 16-bit unsigned integer (default = *0*)

`int` 32-bit signed integer (default = *0*)

`uint` 32-bit unsigned integer (default = *0*)

`long` 64-bit signed integer (default = *0*)

`ulong` 64-bit unsigned integer (default = *0*)

`float` 32-bit floating point number (default = *0*)

`double` 64-bit floating point number (default = *0*)

`string` UTF-8 string (default = *""*)

`vector<T>` A list of values of type *T*. (default = *[]*)

`optional<T>` An optional value of type *T*. (default = *null*)

`set<t>` An ordered set of values of type *T*. (no duplicates)

`map<K, V>` An ordered map with key type *K* and value type *V*. (no duplicate keys)

`pair<K, V>` A pair of values of type *K* and *V* respectively.

`T*` A value of type *T* or any derived type, or *null*. (default = *null*)

3.2 Coordinate Systems

Below are the most common coordinate systems used. In addition to those each sensor usually has its own coordinate system too.

3.2.1 Base Link (base_link)

The platform's own coordinate system, usually the center of mass. X points forward, Y points to the left and Z points upwards. The internal name is `base_link`.

3.2.2 Odometry (odom)

This is an arbitrary coordinate system with it's origin at the starting position (at the time of power on) of the platform. X and Y form the plane of movement, while Z is pointing upwards and aligned with gravity. Usually there is no motion in Z direction, except when a roll and pitch sensor is installed. The internal name is `odom`.

3.2.3 Map (map)

This is the coordinate system of the currenly used *Grid Map* and *Road Map*. Usually the *Grid Map* is created first and it's origin is the starting position of where the map was created initially. X and Y form the plane of movement, while Z is pointing upwards and aligned with gravity. The internal name is `map`.

3.3 Classes

3.3.1 basic.Transform3D

Class

Transform3D represents a 2D/3D transformation from a specified coordinate system to another. It can be described by a 4x4 matrix that transforms a 3D vector from the source coordinate system to the target coordinate system.

Fields

long **`time`**
: POSIX timestamp in [usec].

string **`frame`**
: Source coordinate system name.

string **`parent`**
: Target coordinate system name.

Matrix4d **`matrix`**
: The transformation matrix, such that left multiplying a vector transforms it from the source coordinate system `frame` to the target coordinate system `parent`. For example: target = matrix * [x, y, z, 1]^T. See *math.Matrix4d*.

3.3.2 math.Matrix4d

A 4x4 double (64-bit float) matrix, usually a 3D transformation matrix.

3.3.3 math.Vector2d

A 2D double (64-bit float) vector, usually a (x, y) position. Mathmatically a column vector, in JSON an array of 2 numbers.

3.3.4 math.Vector3d

A 3D double (64-bit float) vector, usually (x, y, z) for a 3D position, or (x, y, yaw) for a 2D pose. Mathmatically a column vector, in JSON an array of 3 numbers.

3.3.5 math.Vector3f

A 3D float vector, usually (x, y, z) for a 3D position, or (x, y, yaw) for a 2D pose. Mathmatically a column vector, in JSON an array of 3 numbers.

3.3.6 pilot.ActiveIncidents

Class

Contains the list of currently active incidents, which is periodically published on topic `platform.active_incidents`.

Fields

long **`time`**
: POSIX timestamp [usec]

vector<event_t> **`events`**
: List of active incidents, see *pilot.event_t*.

3.3.7 pilot.BatteryState

Class

BatteryState contains information regarding the platform's batteries.

Inherits from *pilot.Sample*.

Fields

float **`remaining`**
: Percentage of charge remaining, from 0 to 1.

float **`voltage`**
: Battery voltage in [V].

float **`current`**
: Battery current, positive = charging, in [A].

float **`temperature`**
: Battery temperatue in [C].

battery_type_e **type**
Battery type, see *pilot.battery_type_e*.

3.3.8 pilot.Beacon

Class

Status updates to be published by robots.

Inherits from *pilot.RobotInfo*.

Fields

long **time**
POSIX timestamp [usec]

3.3.9 pilot.CostMapData

Class

A map representing the navigation cost based on the proximity to walls and other obstacles.

Inherits from *pilot.GridMapData*.

Fields

static const uchar **PROHIBITED** = 254

static const uchar **UNKNOWN** = 255

float **cost_scale**
obstacle distance scale [m]

```
distance = (1 - (cost / 200)) * cost_scale
```

Image8 **cost**

- 0 to 200 = cost
- 200 = wall
- > 200 = same as occupancy map

Methods

ImageF32 **to_float** () **const**
convert to float format (0 to 1)

Image8 **to_rgba_image** (int *alpha*) **const**
for visualization (with color)

Image8 **to_rgba_mono_image** (int *alpha*) **const**
for visualization (just grayscale)

static Vector4uc **to_rgba** (int *value*, int *alpha*)
convert occupancy to RGBA

static Vector4uc **to_rgba_mono** (int *value*, int *alpha*)
convert occupancy to RGBA mono

3.3.10 pilot.EmergencyState

Class

EmergencyState contains information regarding the platform's emergency systems.

Inherits from *pilot.Sample*.

Fields

em_stop_state_e **state**
Current EM Stop state, see *pilot.em_stop_state_e*.

bool **is_button_stop**
If an emergency stop button has been pressed.

bool **is_scanner_stop**
If a scanner stop was triggered.

3.3.11 pilot.Event

Class

Generic event type, as published on topic `platform.events` for example.

Fields

long **time**
POSIX timestamp [usec]

event_t **event**
Event data, see *pilot.event_t*.

Object **info**
Additional information regarding the event, see *vnx.Object*.

3.3.12 pilot.ExecutionHistory

Class

A history of recent *execution states* of the *TaskHandler*.

Fields

list<ExecutionState *> **history**

3.3.13 pilot.ExecutionState

Class

ExecutionState contains information about the current state of the *TaskHandler*.

Inherits from *pilot.Sample*.

Fields

execution_state_e **status**
: Current execution state, see *pilot.execution_state_e*.

Hash64 **jobid**
: Current program execution id. See *vnx.Hash64*.

string **program_name**
: Name of the running program.

Task ***task**
: Current task being executed, if any. See *pilot.Task*.

vector<StackFrame> **stack**
: Execution stack, most recent first. See *pilot.StackFrame*.

long **time_started**
: POSIX time stamp when program was started in [usec].

long **time_ended**
: POSIX time stamp when program has finised in [usec], zero otherwise (ie. still running).

bool **is_minor**
: True if current task is nothing major, for example log message output, etc.

3.3.14 pilot.Footprint

Class

Footprint describes the space needed by a platform, including the safety scanner fields with some additional padding.

Fields

vector<Vector2d> **points**
: List of 2D (x, y) [m, m] points in `base_link` coordinates that form a polygon representing the footprint. The points should be specified either in clockwise or anti-clockwise direction.

  See *Coordinate Systems*.

3.3.15 pilot.GridMapData

Class

Base class of pixel based world maps.

Inherits from *pilot.Sample*.

Fields

string **name**

long **last_modified**
timestamp [usec]

float **scale**
size of a pixel in [m]

Vector2d **origin**
grid offset (map position of lower left pixel, world to map) [m]

double **orientation**
grid rotation (world to map) [rad]

Methods

void **transform**(Transform3D **sample*)

Matrix4d **get_grid_to_frame**() **const**
computes grid to 'frame' 2.5D transformation matrix

Matrix4d **get_frame_to_grid**() **const**
computes 'frame' to grid 2.5D transformation matrix

bool **same_as**(GridMapData **other*) **const**
returns true if both have same name and last_modified time

map_info_t **get_info**() **const**

3.3.16 pilot.Incident

Class

Generic incident type, as published on `platform.events` for example.

External modules can publish their own incidents on topic `platform.incidents`. They will be handled by *PilotServer* and re-published on `platform.events` as required.

Inherits from *pilot.Event*.

Fields

bool **is_active**
If this incident is currently active, ie. not a one time event.

bool **is_cleared**
If this incident was active before and is now cleared, ie. no longer active.

int **timeout_ms** = 3000
Timeout in case of an active incident in [ms]. *timeout_ms <= 0* disables the feature. If the incident is not re-published within the specified timeout it will be automatically cleared by *PilotServer*.

3.3.17 pilot.JoyData

Class

JoyData contains data from a connected joystick.

Inherits from *pilot.Sample*.

Fields

vector<float> **axes**
List of joystick axis and their position from -1 to 1. Usually there are the following: [JOYAXIS_LEFT_X, JOYAXIS_LEFT_Y, JOYAXIS_LT, JOYAXIS_RIGHT_X, JOYAXIS_RIGHT_Y, JOYAXIS_RT, JOYAXIS_MAX]

vector<bool> **buttons**
List of joystick buttons and their state. Usually there are the following: [JOYBUTTON_A, JOYBUTTON_B, JOYBUTTON_X, JOYBUTTON_Y, JOYBUTTON_LB, JOYBUTTON_RB, JOYBUTTON_BACK, JOYBUTTON_START, JOYBUTTON_MAX]

3.3.18 pilot.LaserPointCloud

Class

LaserPointCloud contains additional information for laser point clouds.

Inherits from *pilot.PointCloud2D*.

Fields

vector<uchar> **intensity**
Measured intensities for each point in `points`. An intensity >= 100 is a reflector, 200 is a perfect reflector.

3.3.19 pilot.LocalPlannerState

Class

LocalPlannerState represents the current state of the *LocalPlanner*.

Inherits from *pilot.Sample*.

Fields

Hash64 **job**
Unique id of the current job / goal.

long **path_time**
POSIX timestamp of the current path in [usec].

double **path_length**
Total path length in [meters].

PathPoint2D ***point**
Current path point if any (ie. closest to current position).

PathPoint2D ***goal**
Current goal point, ie. last point in current path.

goal_options_t **goal_options**
Current goal options.

Vector2d **pos_error**
Current (x, y) position error in [meters].

double **yaw_error**
Current yaw orientation error in [radians].

float **update_rate**
Current control update rate in [1/s].

local_planner_state_e **state**
Current local planner state, see *pilot.local_planner_state_e*.

limit_reason_e **velocity_reason**
Current reason for velocity limitation, see *pilot.limit_reason_e*.

limit_reason_e **yawrate_reason**
Current reason for yawrate limitation, see *pilot.limit_reason_e*.

float **time_stuck**
How long the platform has already been stuck in [seconds], in case `state == STUCK`.

float **progress**
Relative progress, from 0 to 1. Ratio of traversed path distance vs. total path length.

bool **is_backwards**
If platform is currently driving backwards.

bool **is_restricted**
If platform is currently restricted from turning around.

vector<int> **path_history**
Sliding window history of traversed path element ids, newest at the end. See *pilot.MapElement*. Usually contains at least the last 10 elements.

3.3.20 pilot.LocalizationStatus

Class

LocalizationStatus contains information about the current state of localization.

Inherits from *pilot.Sample*.

Fields

localization_mode_e **mode**
Current localization mode, see *pilot.localization_mode_e*.

set<string> **sensors**
Current list of sensors used for localization.

float **update_rate**
Current update rate in [1/s]

int **num_points**
Number of valid sensor points, excluding reflectors.

int **num_points_total**
Number of valid sensor points, including reflector points.

Vector3f **std_dev**
Current particle spread standard deviation (x, y, yaw) [m, m, rad]. See *math.Vector3f*.

float **score**
Current scan matching score, ie. *average likelihood*. More is better, maximum is *1.0* (ie. all scan points are matching exactly to the Grid Map).

3.3.21 pilot.MapArea

Class

Describes a special area within a map.

Fields

string **name**

string **type**

string **description**

polygon_t **outline**

float_param_t **max_velocity**

vector<area_property_e> **flags**

3.3.22 pilot.MapElement

Class

MapElement is the base class for an entity in the Road Map.

Fields

int **id**
Unique id for the element, starting at zero.

string **group**
Logical group name, mostly for visualization purposes.

vector<string> **profiles**
List of profiles assinged to this element, parameters are copied / over-written in the order of profiles listed.

3.3.23 pilot.MapMatch

Class

MapMatch describes the current position in terms of the *Road Map*, when possible.

Inherits from *pilot.Sample*.

Fields

bool **is_valid**
: If a match was found or not.

float **distance**
: Shortest distance from platform center to matched node or segment in [meters].

MapNode ***node**
: The matched *pilot.MapNode* if near a node.

RoadSegment ***segment**
: The matched *pilot.RoadSegment* if on a segment, which is normally the case.

3.3.24 pilot.MapNode

MapNode describes a node in a Road Map. It has a position as well as an optional name, in addition to some parameters.

Inherits from *pilot.MapElement*.

Fields

int **parent** = -1
: Parent node id, if any. If a *pilot.MapNode* or *pilot.MapStation* has a parent its position (and orientation) is defined relative to it.

string **name**
: Optional node name, usually a station name.

Vector2d **position**
: 2D (x, y) map position, see *math.Vector2d*.

Vector3d **offset**
: 3D (x, y, yaw) pose offset relative to parent, see *math.Vector3d*. If parent is a *pilot.MapStation* the offset is relative to its orientation.

set<drive_flags_e> **drive_flags**
: A set of drive flags for this node, see *pilot.drive_flags_e*.

3.3.25 pilot.MapProfile

Class

MapProfile represents a set of parameters which can be applied to multiple map elements at once.

Fields

string **name**
Profile name, should be unique.

string **description**
Profile description text.

Object **node**
A set of *pilot.MapNode* parameters. See *vnx.Object*.

Object **segment**
A set of *pilot.RoadSegment* parameters. See *vnx.Object*.

Object **station**
A set of *pilot.MapStation* parameters. See *vnx.Object*.

3.3.26 pilot.MapStation

Class

MapStation describes a station in a Road Map, something which can be set as a goal. It has a position as well as an orientation, in addition to some parameters.

Inherits from *pilot.MapNode*.

Fields

float **orientation**
Absolute map goal orientation in [radians].

float_param_t **goal_tune_time**
Additional time for fine tuning position in [seconds]. See *pilot.float_param_t*.

vector_3f_param_t **goal_tolerance**
Maximum goal position error (x, y, yaw) [m, m, rad], see *pilot.vector_3f_param_t*.

3.3.27 pilot.OccupancyMapData

Class

Map with environment information used for navigation.

Inherits from *pilot.GridMapData*.

Fields

static const uchar **FREE** = 0

static const uchar **DYNAMIC** = 253

static const uchar **PROHIBITED** = 254

static const uchar **UNKNOWN** = 255

Image8 **occupancy**
Represents the pixel grid.

- 0 to 100 = wall
- 101 to 200 = reflector
- 254 = prohibited
- 255 = unknown

Methods

Image8 **to_mono_image**() **const**
convert to 8-bit format (0 to 255)

Image8 **to_rgba_image**(int *alpha*) **const**
for visualization

ImageF32 **to_float**(float *special*, bool *combined*) **const**
convert to float format (0 to 1)

OccupancyMapData ***to_reflector_map**() **const**
convert to reflector occupancy (shift 200 to 100, erase 0 to 100)

OccupancyMapData ***to_combined_map**() **const**
convert reflectors to normal occupancy (map 200 to 100)

static Vector4uc **to_rgba**(int *value*, int *alpha*)
convert occupancy to RGBA

3.3.28 pilot.Odometry

Class

Odometry contains information about the platform's odometry, as well as velocities. See also *Coordinate Systems*.

Inherits from *basic.Transform3D*.

Fields

Vector3d **position**
Current 3D (x, y, z) [m] `odom` position.

Vector3d **orientation**
Current 3D (roll, pitch, yaw) [rad] `odom` orientation.

Vector3f **linear_velocity**
Current 3D (x, y, z) [m/s] `odom` velocity.

Vector3f **angular_velocity**
Current 3D (roll, pitch, yaw) [rad/s] `odom` angular velocity.

3.3.29 pilot.Path2D

Class

Path2D represents a 2D path made of individual *pilot.PathPoint2D* points, in a given coordinate system.

Inherits from *pilot.Sample*.

Fields

Hash64 **job**
Unique job id for this path / goal.

vector<PathPoint2D *> **points**
List of path points, see *pilot.PathPoint2D*.

3.3.30 pilot.PathPoint2D

Class

PathPoint2D represents a point on a path to navigate. They are usually generated from a Road Map and then optimized while driving. It can also be a final goal position.

Inherits from *pilot.Pose2D*.

Fields

int **map_id** = -1
Id of the map element in the Road Map, if any.

drive_mode_e **drive_mode** = DEFAULT
Specifies how the robot should navigate, see *pilot.drive_mode_e*.

set<drive_flags_e> **drive_flags**
A set of drive flags for this path point, see *pilot.drive_flags_e*.

float_param_t **orientation**
Yaw angle offset relative to `pose.z()` (holonomic only) in [radians]. See *pilot.float_param_t*.

float_param_t **max_velocity**
Maximum velocity in [m/s]. See *pilot.float_param_t*.

float_param_t **max_yawrate**
Maximum yawrate in [rad/s]. See *pilot.float_param_t*.

float_param_t **goal_tune_time**
Additional time for fine tuning position in [seconds]. See *pilot.float_param_t*.

vector_3f_param_t **tolerance**
Maximum position error (x, y, yaw) [m, m, rad], see *pilot.vector_3f_param_t*.

bool **is_restricted**
True if a 360 degree rotation is not possible at this location.

3.3.31 pilot.PilotState

Class

PilotState contains information about the current state of the robot.

Fields

long **time**
POSIX timestamp in [usec]

pilot_mode_e **pilot_mode**
Current pilot mode, see *pilot.pilot_mode_e*.

motion_mode_e **motion_mode**
Current motion mode, see *pilot.motion_mode_e*.

LocalPlannerState ***planner**
Current local planner state, see *pilot.LocalPlannerState*.

LocalizationStatus ***localization**
Current localization status, see *pilot.LocalizationStatus*.

ExecutionState ***execution**
Current task execution status, see *pilot.ExecutionState*.

bool **is_recording**
If data recording is currently active, see topic `vnx.recorder_status`.

3.3.32 pilot.PlatformInfo

Class

PlatformInfo contains static information about the robot.

Fields

platform_type_e **type**
Type of the platform, see *pilot.platform_type_e*.

string **name**
Name of the robot.

string **serial**
Serial number of the robot.

long **date_of_manufacture**
POSIX timestamp regarding the date of manufacture of the robot in [seconds].

vector<string> **features**
List of special features that the platform has.

3.3.33 pilot.PointCloud2D

Class

PointCloud2D contains a cloud of 2D points, usually laser points. See also *pilot.LaserPointCloud*.

Inherits from *pilot.Sample*.

Fields

string **sensor**
: Original sensor coordinate frame. (also name of sensor)

sensor_2d_range_t **field**
: Field of view of the sensor, relative to its own coordinate system, see *pilot.sensor_2d_range_t*.

Transform3D ***base_to_odom**
: Transformation from `base_link` to `odom` frame at the time of this scan.

Transform3D ***sensor_to_base**
: Transformation from `sensor` to `base_link` frame.

vector<Vector2d> **points**
: List of 2D (x, y) points [meters], in `frame` coordinates, see *pilot.Sample*.

3.3.34 pilot.Pose2D

Class

Pose2D represents a 2D pose (x, y, yaw) in the specified coordinate system.

Inherits from *basic.Transform3D*.

Fields

Vector3d **pose**
: 2D pose (x, y, yaw) [m, m, rad], see *math.Vector3d*.

Matrix3f **covariance**
: Optional pose covariance matrix.

3.3.35 pilot.PoseArray2D

Class

An aggregation of poses.

Inherits from *pilot.Sample*.

Fields

vector<Vector3d> **poses**
: [x, y, yaw] (m, m, rad)

Methods

void **transform** (Transform3D **sample*)

3.3.36 pilot.RoadMapData

Class

RoadMapData contains all the information regarding a *Road Map*.

Fields

string **name**
Name of the map.

long **last_modified**
POSIX timestamp when the map was last modified in [usec].

vector<MapNode *> **nodes**
List of map nodes, see *pilot.MapNode*.

vector<RoadSegment *> **segments**
List of road segments, see *pilot.RoadSegment*.

map<string, MapProfile *> **profiles**
Map of map profiles [name => profile], see *pilot.MapProfile*.

vector<MapArea *> **areas**
List of map areas, see *pilot.MapArea*.

3.3.37 pilot.RoadSegment

Class

RoadSegment represents a connection in the *Road Map* from one *pilot.MapNode* to another.

Inherits from *pilot.MapElement*.

Fields

int **from_node**
pilot.MapNode `id` from where the segment begins.

int **to_node**
pilot.MapNode `id` to where the segment goes.

char **direction** = 0
-1 = one way backwards, 0 = two way, 1 = one way forwards

drive_mode_e **drive_mode** = RELAXED_PATH_FOLLOW
Drive mode to be used on this segment, see *pilot.drive_mode_e*.

orientation_mode_e **orientation_mode** = RELATIVE_ROAD
Orientation mode to be used on this segment, see *pilot.orientation_mode_e*.

float_param_t **tolerance**
Lateral tolerance in [meters], how much the path can be modified while driving (to avoid obstacles, etc). See *pilot.float_param_t*.

float_param_t **orientation**
Orientation to drive at on this segment in [radians]. Actual orientation depends on *orientation_mode*. See *pilot.float_param_t*.

float_param_t **orientation_tolerance**
Maximum deviation allowed from the target orientation on this segment in [radians]. See *pilot.float_param_t*.

float_param_t **max_velocity**
Maximum velocity allowed on this segment in [m/s]. See *pilot.float_param_t*.

float_param_t **max_yawrate**
Maximum yawrate allowed on this segment in [rad/s]. See *pilot.float_param_t*.

set<drive_flags_e> **drive_flags**
A set of drive flags for this segment, see *pilot.drive_flags_e*.

3.3.38 pilot.RobotInfo

Class

Information about the overall status of a robot.

Fields

Hash64 **id**
unique robot id

optional<map_info_t> **grid_map**
used grid map

optional<map_info_t> **road_map**
used road map

optional<pose_2d_t> **map_pose**

optional<Vector3f> **velocity**
relative to base_link (x, y, yaw) [m/s, m/s, rad/s]

PilotState ***state**

Path2D ***map_path**

MapMatch ***map_match**

Footprint ***footprint**

PlatformInfo ***platform**

SystemState ***system_status**

BatteryState ***battery_state**

EmergencyState ***emergency_state**

vector<PointCloud2D *> **laser_scans**
in map coordinates

3.3.39 pilot.Sample

Class

Sample is a base class for any type requiring a time stamp as well as a coordinate system.

Fields

long **time**
POSIX timestamp in [usec]

string **frame**
Coordinate system name, usually `map`, `odom` or `base_link`, see *Coordinate Systems*.

3.3.40 pilot.StackFrame

Class

StackFrame contains information about the context of execution of the *TaskHandler*.

Fields

string **method**
Method name of the function on the call stack.

int **line_number**
Line number in the source code.

3.3.41 pilot.SystemStatus

Class

SystemStatus contains information about the platform's hardware.

Fields

float **ambient_temperature**
Ambient temperature inside the platform in [C].

vector<bool> **relay_states**
State of the power relays, usually there is 4 of them.

keypad_state_t **keypad_state**
Keypad button states, if available, see *pilot.keypad_state_t*.

charging_state_e **charging_state**
Current charging state, see *pilot.charging_state_e*.

power_system_type_e **power_system_type**
Type of power system, see *pilot.power_system_type_e*.

vector<system_error_e> **system_errors**
List of system errors currently active, see *pilot.system_error_e*.

bool **is_shutdown**
If platform has been asked to shutdown now, via the key switch.

bool **is_initialized**
If platform hardware is initialized.

3.3.42 pilot.Task

Class

Task contains information about the current task being executed by the *TaskHandler*.

Fields

Hash64 **id**
Unique task id, randomly generated. See *vnx.Hash64*.

string **module**
Module name which is handling the task, empty if handled internally.

string **method**
Method name of the task being executed.

Object **args**
Parameters for the task being executed. See *vnx.Object*.

3.3.43 pilot.VelocityCmd

Class

VelocityCmd contains the current commanded platform velocity. Usually only X / Y velocity and yawrate are specified.

Inherits from *pilot.Sample*.

Fields

Vector3f **linear**
3D (x, y, z) translational velocity in [m/s].

Vector3f **angular**
3D (roll, pitch, yaw) angular velocity in [rad/s].

bool **allow_wheel_reset** = true
If to allow automatic wheel resetting of an MPO-700.

bool **reset_wheels** = false
If to reset the wheels of an MPO-700 to their home position, when not moving.

3.3.44 pilot.VelocityLimits

Class

VelocityLimits defines different limits and thresholds regarding platform velocity.

Fields

float **max_vel_x**
Maximum forward velocity (positive) [m/s]

float **min_vel_x**
Maximum backwards velocity (negative) [m/s]

float **max_vel_y**
Maximum lateral velocity (positive) [m/s]

float **max_trans_vel**
Maximum translational velocity in any direction (positive) [m/s]

float **min_trans_vel**
Minimum translational velocity (positive) [m/s]

float **max_rot_vel**
Maximum absolute yawrate (positive) [rad/s]

float **min_rot_vel**
Minimum absolute yawrate (positive) [rad/s]

float **trans_stopped_vel**
Threshold for deciding when platform is stopped based on translational velocity (positive) [m/s]

float **rot_stopped_vel**
Threshold for deciding when platform is stopped based on rotational velocity (positive) [rad/s]

3.3.45 pilot.area_property_e

Enumeration

Describes a property of a *pilot.MapArea*.

enumerator PROHIBITED

3.3.46 pilot.battery_code_e

Enumeration

Describes an error or warning regarding the battery.

enumerator BATT_LOW
If battery is below ~25% remaining.

enumerator BATT_CRITICAL
If battery is below ~10% remaining.

enumerator BATT_OVERHEAT
If battery temperature is above 50C.

3.3.47 pilot.battery_type_e

Enumeration

Denotes the battery type.

enumerator AGM
Lead battery with absorbent glass mat technology

enumerator LFP
Lithium iron phosphate battery

3.3.48 pilot.charging_state_e

Enumeration

Displays the charging state of the battery.

enumerator NOT_CHARGING

enumerator IS_CHARGING

enumerator NO_CHARGER

enumerator BRAKES_OPEN

enumerator EM_STOP

enumerator ABORTED

enumerator FINISHED

3.3.49 pilot.drive_flags_e

Enumeration

Flags that modifiy robot navigation or behavior.

enumerator FIXED_ORIENTATION
Flag to disable orientation optimization / modification (holonomic only). Will keep orientation from Road Map, even when the path itself is changed. Used to keep a specific orientation while moving.

enumerator IGNORE_FOOTPRINT
Flag to disable collision avoidance, use with care. Used to dock with external components where obstacles can be very close to the robot (inside footprint).

enumerator DISABLE_ROTATION
Disable rotation of the robot (forced zero control yawrate). Will prevent robot from rotating while moving, or rotating in place. Used to dock with external components where rotation would lead to damage of the robot or components. If rotation is necessary to reach the goal, a failure will be issued.

enumerator BACKWARD_OVERRIDE
Allows driving backwards even when normally prohibited. Used in special cases where unsafe backwards driving is necessary.

enumerator OMNI_DIRECTIONAL
Allows driving with arbitrary orientation by default (holonomic only). Will permit infinite yaw lookahead, such as to minimize total yaw motion. If not set, a platform will usually align with orientation of the road.

enumerator DISABLE_WHEEL_RESET
Disables automatic wheel resetting of the MPO-700. Used to increase positioning accuracy at certain stations.

enumerator RESET_WHEELS
Will reset the wheels of an MPO-700 at the (goal) station to their home positions.

3.3.50 pilot.drive_mode_e

Enumeration

Specifies how the robot should navigate.

enumerator STRICT_PATH_FOLLOW
Follows a given path exactly, without trying to optimize it. If there is an obstacle the robot will wait.

enumerator RELAXED_PATH_FOLLOW
The given path is optimized while driving. Obstacles may be avoided, corners are cut.

enumerator FREE_PATH_FOLLOW
Same as `RELAXED_PATH_FOLLOW` except that in case of a blockage the robot is allowed to generate a new path by itself.

enumerator FREE_ROAMING
Will ignore the geometry of the path segment(s) and use the *GlobalPlanner* to create an actual path to follow.

enumerator DEFAULT
One of the above, depending on robot configuration, usually `RELAXED_PATH_FOLLOW`.

3.3.51 pilot.em_stop_state_e

Enumeration

Different states of the Emergency Stop system.

enumerator FREE
Emergency Stop is not engaged, platform is free to move.

enumerator STOPPED
Emergency Stop is engaged, platform is prevented from moving.

enumerator CONFIRMED
Emergency Stop has been released manually, will go to `FREE` shortly.

3.3.52 pilot.event_code_e

Enumeration

Describes an event.

enumerator NEW_GOAL

enumerator GOAL_REACHED

enumerator GOAL_CANCELED

enumerator NEW_TASK

enumerator TASK_COMPLETED

enumerator TASK_FAILED

enumerator EXECUTION_PAUSED

enumerator EXECUTION_RESUMED

enumerator EXECUTION_CANCELED

3.3.53 pilot.event_t

Struct

event_t describes an event, without an attached timestamp.

The fields `module`, `code_type` and `code` uniquely define a specific event type that can occur.

Fields

event_type_e **`type`**
Event type, see *pilot.event_type_e*.

string **`module`**
Module name that generated the event.

string **`code_type`**
Type name for the `code` field, usually the enum type name. For example: `pilot.safety_code_e`.

string **`code`**
Event code, usually an enum value. For example: `EMERGENCY_STOP`.

3.3.54 pilot.event_type_e

Enumeration

Describes the type of a *pilot.event_t*.

`enumerator ERROR`

`enumerator WARNING`

`enumerator NOTIFICATION`

3.3.55 pilot.execution_state_e

Enumeration

Possible states of task execution.

`enumerator RUNNING`
Currently executing a task.

`enumerator PAUSED`
Execution has been paused.

`enumerator FINISHED`
Execution has finished.

`enumerator CANCELED`
Execution was canceled.

3.3.56 pilot.float_param_t

Struct

float_param_t represents a `float` value with additional semantic information.

It can always be specified as just a `float` value, in which case *type* will be set to `CUSTOM`.

Fields

param_type_e **`type`**
Semantic type information, see *pilot.param_type_e*.

float **`value`**
The actual value, depending on `type` it may not be used.

3.3.57 pilot.goal_options_t

Struct

A set of options to modify how a goal is to be reached.

Fields

optional<float> **`max_velocity`**
Global velocity limit in [m/s], optional.

optional<float> **`max_time_stuck`**
How long to wait until aborting goal when stuck in [seconds] (default = infinite)

optional<drive_mode_e> **`drive_mode`**
Custom drive mode for the entire path, see *pilot.drive_mode_e*.

set<drive_flags_e> **`drive_flags`**
Set of additional drive flags for final goal position. See *pilot.drive_flags_e*.

3.3.58 pilot.keypad_state_t

Struct

An object of this type represents the current state of the keypad.

For each button there is a boolean member which is true when the button is pressed and false when it is not.

Fields

bool **`info_button`**

bool **`home_button`**

bool **`start_button`**

bool **`stop_button`**

bool **`break_release_button`**

bool **digital_input**[3]
An array entry is true if the corresponding digital input is active.

3.3.59 pilot.limit_reason_e

Enumeration

Possible reasons for limiting velocity or yawrate.

enumerator DEFAULT_MAX
Default maximum for the platform. (global configuration)

enumerator CUSTOM_MAX
Custom limit (at current position). (Road Map parameter, or via. *pilot.goal_options_t*)

enumerator ACCEL_MAX
Limited due to hitting maximum allowed acceleration / decceleration.

enumerator LOCAL_COST
Limited due to obstacles in path or close to path.

enumerator PATH_DEVIATION
Limited due to unexpected deviation from path.

enumerator ORIENTATION_DEVIATION
Limited due to unexpected deviation from target orientation.

enumerator CURVE_LIMIT
Limited due to maximum lateral acceleration.

enumerator WAITING
Limited due to waiting for a certain condition.

enumerator GOAL_MAX
Limited due to approaching final goal position.

enumerator LIMIT_AHEAD
Limited due to approaching a lower limit on the path ahead.

enumerator OBSTACLE
Limited due to obstacle blocking the path.

enumerator STOPPING
Limited due to trying to stop.

enumerator FINISHED
Limited due to having reached the goal.

3.3.60 pilot.local_planner_state_e

Enumeration

Possible states of the *LocalPlanner*.

enumerator IDLE
Waiting for a job / goal.

enumerator WAITING
Waiting to move, due to several possible reasons, for example waiting for localization to intialize.

enumerator TRANSLATING
Moving in X / Y direction, can include rotation also.

enumerator ROTATING
Rotating in place, without X / Y movement.

enumerator ADJUSTING
Rotating in place while making small adjustments in X / Y position. For *differential* kinematics only.

enumerator TURNING
Rotating in place to turn around. For *differential* kinematics only.

enumerator STUCK
Unable to continue for now, due to obstacles in the path.

enumerator FINISHED
Current goal has been reached, waiting for new job / goal.

enumerator CANCELED
Current goal has been canceled, stopping now.

enumerator LOST
Current position could not be matched to the given path, requesting new path.

3.3.61 pilot.localization_mode_e

Enumeration

Possible modes of localization.

enumerator NONE
No localization running.

enumerator NO_MAP
No Grid Map available.

enumerator NO_INPUT
No sensor input available.

enumerator NO_ODOMETRY
No odometry available.

enumerator LOST
Unable to localize, position most likely wrong.

enumerator INITIALIZING
Performing initial localization, waiting for confidence to increase.

enumerator DEAD_RECKONING
Unable to localize at current position, extrapolating based on odometry.

enumerator MODE_1D
Partial localization in one direction, *DEAD_RECKONING* otherwise.

enumerator MODE_1D_YAW
Partial localization in one direction plus orientation, *DEAD_RECKONING* otherwise.

enumerator MODE_2D
Partial localization in X and Y direction, *DEAD_RECKONING* for orientation.

enumerator MODE_2D_YAW
Full localization in X and Y direction as well as orientation.

3.3.62 pilot.motion_mode_e

Enumeration

A set of motion modes of the robot, defining who is contolling the motion.

enumerator NONE
No motion is possible in this mode.

enumerator CUSTOM
Custom motion commands on topic `platform.velocity_cmd` control the robot.

enumerator JOYSTICK
Motion commands from a connected joystick control the robot.

enumerator AUTOMATIC
The *LocalPlanner* is controlling the robot, usually following a path generated by the *HybridPlanner*.

3.3.63 pilot.orientation_mode_e

Enumeration

Different modes affecting the orientation of the platform.

enumerator RELATIVE_ROAD
Specified orientation is relative to the *pilot.RoadSegment*.

enumerator ABSOLUTE_MAP
Specified orientation is an absolute map orientation (ie. relative to the `map` coordinate system).

3.3.64 pilot.param_type_e

Enumeration

Semantic type of a parameter.

enumerator DEFAULT
The default value should be used.

enumerator CUSTOM
The custom value that is provided should be used.

enumerator OPTIMIZED
Similar to *CUSTOM*, means the value was optimized internally instead of specified by a user.

enumerator DISABLED
The feature corresponding to the parameter should be disabled.

enumerator IGNORE
The parameter should be ignored.

3.3.65 pilot.permission_e

Enumeration

Permissions that can be given to certain users. See also *vnx.permission_e*.

enumerator MOVE
Permission to move robot.

enumerator CHARGE
Permission to charge robot.

enumerator INITIALIZE
Permission to initialize localization.

enumerator RECORD_DATA
Permission to start a data recording.

enumerator REMOTE_CONTROL
Permission to control robot remotely.

enumerator RELAY_CONTROL
Permission to switch relays.

enumerator DISPLAY_CONTROL
Permission to display text on the LCD.

enumerator CHANGE_GRIDMAP
Permission to change the active Grid Map.

enumerator CHANGE_ROADMAP
Permission to change the active Road Map.

enumerator UPLOAD_SCRIPT
Permission to upload / overwrite scripts.

enumerator EXECUTE_SCRIPT
Permission to execute scripts.

enumerator INTERVENE_SCRIPT
Permission to pause / resume scripts.

3.3.66 pilot.pilot_mode_e

Enumeration

A set of operating modes of the robot.

enumerator MAPPING
Creation of a new map or updating of an existing map.

enumerator NAVIGATION
Normal navigation mode, using localization with an existing map.

enumerator TELEOP
Basic operation mode, without localization and path planning.

enumerator REPLAY
Special simulation mode to recompute dynamic data based on recorded sensor data, for visualization purposes.

3.3.67 pilot.platform_type_e

Enumeration

Available platform types.

enumerator MP_400
Neobotix MP-400, see https://www.neobotix-roboter.de/produkte/mobile-roboter/mobiler-roboter-mp-400.

enumerator MP_500
Neobotix MP-500, see https://www.neobotix-roboter.de/produkte/mobile-roboter/mobiler-roboter-mp-500.

enumerator MPO_500
Neobotix MPO-500, see https://www.neobotix-roboter.de/produkte/mobile-roboter/mobiler-roboter-mpo-500.

enumerator MPO_700
Neobotix MPO-700, see https://www.neobotix-roboter.de/produkte/mobile-roboter/mobiler-roboter-mpo-700.

3.3.68 pilot.polygon_t

Struct

polygon_t describes a geometric polygon in a certain frame of reference.

Fields

string **frame**
Coordinate system name, for example `base_link`.

vector<Vector2d> **points**
List of points forming the polygon, see *math.Vector2d*. The order of points does not matter, clock-wise or anti-clock-wise is both supported.

3.3.69 pilot.power_system_type_e

Enumeration

Specifies the voltage of the system.

enumerator POWER_24V
The system runs on 24 volts.

enumerator POWER_48V
The system runs on 48 volts.

3.3.70 pilot.safety_code_e

Enumeration

Describes an error of the safety system.

enumerator NONE

enumerator SCANNER_STOP
One of the safety scanners has stopped the platform.

enumerator EMERGENCY_STOP
One of the emergency stop buttons has been pressed.

3.3.71 pilot.sensor_2d_range_t

Struct

Specifies the range in which a sensor measurement is considered valid.

Fields

float **min_angle**
normalized between -pi to +pi [rad]

float **max_angle**
normalized between -pi to +pi [rad]

float **min_range**
[m]

float **max_range**
[m]

Methods

bool **is_valid**(float *distance*) **const**

bool **is_within**(float *angle*, float *distance*) **const**

bool **is_within_xy**(Vector2f *point*) **const**

bool **is_within_point**(laser_point_t *point*) **const**

void **add_margin**(float *delta_angle*, float *delta_range*)

3.3.72 pilot.system_error_e

Enumeration

The type of a system error.

enumerator CHARGING_RELAY_ERROR

enumerator BRAKE_RELEASE_BUTTON_ERROR

enumerator MOTOR_ERROR

enumerator SAFETY_RELAY_ERROR

enumerator POWER_RELAY_ERROR

enumerator EM_STOP_SYSTEM_ERROR
Emergency stop button failure.

3.3.73 pilot.vector_3f_param_t

Struct

vector_3f_param_t represents a 3D `float` vector with additional semantic information.

It can always be specified as just a 3D `float` vector, in which case the *types* will be set to `CUSTOM`.

Fields

float_param_t **x**
: X value, see *pilot.float_param_t*.

float_param_t **y**
: Y value, see *pilot.float_param_t*.

float_param_t **z**
: Z value, see *pilot.float_param_t*.

3.3.74 vnx.Hash64

`Hash64` is a 64-bit unsigned integer, which usually represents a CRC64 hash of a string, or sometimes simply a random number.

The specific hash function used is `CRC-64/XZ (alias CRC-64/GO-ECMA)`, see also https://reveng.sourceforge.io/crc-catalogue/17plus.htm#crc.cat-bits.64.

3.3.75 vnx.JRPC_Error

Class

The error object that is contained in error messages returned by *JSON-RPC Interface*.

Fields

int **code**
: One of the following error codes:

```
static const int PARSE_ERROR = -32700;
static const int INVALID_REQUEST = -32600;
static const int METHOD_NOT_FOUND = -32601;
static const int INVALID_PARAMS = -32602;
static const int INTERNAL_ERROR = -32603;

static const int PERMISSION_DENIED = 403;
static const int EXCEPTION = 500;
```

string **message**
: A short message to indicate what went wrong.

Exception ***data**
: The actual exeption object as thrown on the server side.

3.3.76 vnx.LogMsg

Class

Represents a log message.

Fields

static int **ERROR** = 1

static int **WARN** = 2

static int **INFO** = 3

static int **DEBUG** = 4

long **time**

int **level**

int **display_level** = 3

string **process**

string **module**

string **message**

Methods

string **get_output** () **const**
Returns a properly formatted line ready to be printed out.

3.3.77 vnx.ModuleInfo

Class

Information about a module.

Fields

long **time**
time stamp (virtual time) [usec]

Hash64 **id**
unique module id

Hash64 **src_mac**
source mac for publishing

string **name**
module name

string **type**
type name

long **time_started**
POSIX timestamp [usec]

long **time_idle**
current stats (see vnx_heartbeat_interval_ms) [usec]

long **time_running**
current stats (see vnx_heartbeat_interval_ms) [usec]

long **time_idle_total**
: since start of module [usec]

long **time_running_total**
: since start of module [usec]

long **num_async_pending**
: number of pending async requests (waiting for returns)

long **num_async_process**
: number of async requests being processed right now

vector<string> **sub_topics**
: topic subscriptions

vector<string> **pub_topics**
: topic publishers

map<Hash64, Endpoint *> **remotes**
: map of connected processes (process id => endpoint)

TypeCode **type_code**
: module type code

Methods

double **get_cpu_load**() **const**
: 0 to 1

double **get_cpu_load_total**() **const**
: 0 to 1 (total average)

3.3.78 vnx.Object

Object represents an arbitrary object which is dynamically created and does not have a type. However an optional type name can be specified via a special field called `__type`. Internally it is a map of `string` keys to *vnx.Variant* values.

JSON

In JSON format an object is specified as follows:

```
{
        "__type": "optional.type.name.here",
        "field": "value",
        "some": 1234,
        "array": [1, 2, 3, 4],
        "nested": {
                "example": "value",
                ...
        },
        ...
}
```

Lua Script

In Lua Script an object can be created as follows:

```
{
                field = "value",
                some = 1234,
                array = {1, 2, 3, 4},
                nested = {
                        example = "value",
                        ...
                },
                ...
}
```

Native C++

In native C++ an object can be created as follows:

```
#include <vnx/vnx.h>

vnx::Object obj;
obj["field"] = "value";
obj["some"] = 1234;
obj["array"] = std::vector<int>{1, 2, 3, 4};

vnx::Object nested;
nested["example"] = "value";
obj["nested"] = nested;

std::cout << obj << std::endl;
```

3.3.79 vnx.User

Class

A User object represents an entity who can authenticate in order to operate with a special set of permissions.

Fields

string **name**
: The name of the user as used to log in.

string **hashed_password**
: A salted SHA-256 hash of the password.

vector<string> **access_roles**
: user access roles

set<string> **permissions**
: additional, user specific permissions

3.3.80 vnx.Variant

Variant represents a value of arbitrary type which is assigned dynamically. It can be an integral (ie. `int`, `float`, ...), a `string`, a `vector<T>` of values, a `set<T>` of values, a `map<K, V>` of values or an *vnx.Object*, for example.

JSON

In JSON format a *Variant* could be anything, for example: `null`, `true`, `false`, a number, a string, an array of values or an object.

Lua Script

In Lua Script a *Variant* is a normal Lua variable and can be anyting, for example: `nil`, `true`, `false`, a number, a string, an array of values, a table or an object.

Native C++

In native C++ a *Variant* can be created / assigned as follows:

```
#include <vnx/vnx.h>

vnx::Variant example(1234);

std::cout << example.to<int>() << std::endl;    // 1234

example = "value";

std::cout << example.to<std::string>() << std::endl;    // value

example = std::vector<int>{1, 2, 3, 4};

std::cout << example << std::endl;              // [1, 2, 3, 4]

vnx::Object obj;
obj["field"] = "value";
example = obj;

std::cout << example << std::endl;              // {"field": "value"}
```

3.3.81 vnx.access_role_e

Enumeration

A set of default access roles and their default permissions.

See also *vnx.permission_e*.

enumerator DEFAULT
: Default access role, mostly used for anonymous users. Does not provide any permissions by default, but that can be changed via configuration.

enumerator VIEWER
: Read-only access role. Provides permissions: *VIEW*, *TIME_SYNC*.

enumerator OBSERVER
Same as *VIEWER*, with additional permission *CONST_REQUEST*.

enumerator USER
Same as *OBSERVER*, with additional permission *READ_CONFIG*.

enumerator INSTALLER
Same as *USER*, with additional permissions: *PUBLISH*, *WRITE_CONFIG*, *START*, *STOP*, *RESTART*, *SHUTDOWN*, *SELF_TEST*.

enumerator ADMIN
Same as *INSTALLER*, with additional permissions: *REQUEST*, *PROTECTED_CONFIG*, *PROXY_IMPORT*, *PROXY_EXPORT*, *PROXY_FORWARD*, *HOST_SHUTDOWN*, *LOCAL*.

3.3.82 vnx.permission_e

Enumeration

Generic permissions that can be given to certain users.

enumerator VIEW
Permission to subscribe to topics (and access process statistics).

enumerator CONST_REQUEST
Permission to execute const methods that do not explicitly set a permission.

enumerator PUBLISH
Permission to publish samples.

enumerator REQUEST
Permission to execute all methods that do not explicitly set a permission.

enumerator READ_CONFIG
Permission to read config values.

enumerator WRITE_CONFIG
Permission to change config values.

enumerator PROTECTED_CONFIG
Permission to read/write protected config values (like user passwords etc).

enumerator START
Permission to start new modules.

enumerator STOP
Permission to stop a running module.

enumerator RESTART
Permission to restart a module.

enumerator SHUTDOWN
Permission to shutdown the process.

enumerator HOST_SHUTDOWN
Permission to shutdown the host machine.

enumerator SELF_TEST
Permission to execute self tests.

3.4 Modules

3.4.1 GlobalPlanner

Module

The *GlobalPlanner* module supports path planning based on a global cost map, which is generated from a Grid Map. The module uses an *A** style path finding algorithm.

Most functions require permission `pilot.permission_e.MOVE`, see *pilot.permission_e*.

Functions

Common Parameters

`goal_options_t options` Optional goal options, see *pilot.goal_options_t*. If not specified will use default values.

`Hash64 job` Optional job id, to identify the new goal. If not specified (or set to zero) will generate a new random id. See *vnx.Hash64*.

Asynchronous Move Functions

The following functions set / append a new goal while returing immediately.

void **`set_goal`** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)
Sets a new goal using the provided pose and parameters in `goal`. Any pending goals are canceled beforehand. See *pilot.PathPoint2D*. Requires permission `MOVE`.

Synchronous Move Functions

The following functions return when the new goal is physically reached, or the goal was canceled.

void **`move_to`** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal(...)` but will block until goal is reached or canceled. See *pilot.PathPoint2D*. Requires permission `MOVE`.

3.4.2 HttpProxy

Module

The *HttpProxy* module provides a HTTP REST API, see *HTTP Interface*.

Most functions require a special permission, see *vnx.permission_e*.

Functions

void **`publish`** (Object *sample*, string *topic*)
Will publish a given `sample` on the specified `topic`. Requires permission `PUBLISH`. See *vnx.Object*.

Object **multi_request** (map<string, string> *paths*) **const**

Performs multiple `/api/` HTTP requests in one. `paths` is a map from paths to output names, for example: `[["/topic/...", "topic1"], ...]`

The result will be a *vnx.Object* containing the individual results, for example: `{"topic1": {...}, ...}`

3.4.3 HybridPlanner

Module

The *HybridPlanner* module supports a combined path planning by taking into account a Road Map as well as a Grid Map. It automatically chooses to drive on the Road Map when possible, only to fall back to Grid Map based navigation if needed. As such it supports different ways to specify a goal, either by station name, by providing a modified station structure or by specifying an arbitrary pose.

Multiple goals can be specified, in which case the goals are traversed one after the other, without waiting or stopping at intermediate goals.

Most functions require permission `pilot.permission_e.MOVE`, see *pilot.permission_e*.

Functions

Common Parameters

goal_options_t options Optional goal options, see *pilot.goal_options_t*. If not specified will use default values.

Hash64 job Optional job id, to identify the new goal. If not specified (or set to zero) will generate a new random id. See *vnx.Hash64*.

Asynchronous Move Functions

The following functions set / append a new goal while returing immediately.

void **set_goal** (MapStation *goal*, goal_options_t *options*, Hash64 *job*)

Sets a new goal using the provided pose and parameters in `goal`. Any pending goals are canceled beforehand. See *pilot.MapStation*. Requires permission `MOVE`.

void **set_goal_station** (string *name*, goal_options_t *options*, Hash64 *job*)

Sets a new goal using the provided station `name`. Any pending goals are canceled beforehand. The station must exist in the current Road Map. Requires permission `MOVE`.

void **set_goal_position** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)

Sets a new goal using the provided pose and parameters in `goal`. Similar to `set_goal()`. Any pending goals are canceled beforehand. See *pilot.PathPoint2D*. Requires permission `MOVE`.

void **append_goal** (MapStation *goal*, goal_options_t *options*, Hash64 *job*)

Same as `set_goal(...)` but will not cancel active or pending goals. See *pilot.MapStation*. Requires permission `MOVE`.

void **append_goal_station** (string *name*, goal_options_t *options*, Hash64 *job*)

Same as `set_goal_station(...)` but will not cancel active or pending goals. Requires permission `MOVE`.

void **append_goal_position** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)

Same as `set_goal_position(...)` but will not cancel active or pending goals. See *pilot.PathPoint2D*. Requires permission `MOVE`.

Synchronous Move Functions

The following functions return when the new goal is physically reached, or the goal was canceled.

void **move_to** (MapStation *goal*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal(...)` but will block until goal is reached or canceled. See *pilot.MapStation*. Requires permission `MOVE`.

void **move_to_station** (string *name*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal_station(...)` but will block until goal is reached or canceled. Requires permission `MOVE`.

void **move_to_position** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal_position(...)` but will block until goal is reached or canceled. See *pilot.PathPoint2D*. Requires permission `MOVE`.

Asynchronous Movechain Functions

The following functions set / append a list of new goals while returing immediately.

void **set_goals** (vector<MapStation> *goals*, goal_options_t *options*, Hash64 *job*)
Sets a list of new goals, similar to `set_goal(...)`. See *pilot.MapStation*. Requires permission `MOVE`.

void **set_goal_stations** (vector<string> *names*, goal_options_t *options*, Hash64 *job*)
Sets a list of new goals, similar to `set_goal_station(...)`. The stations must exist in the current Road Map. Requires permission `MOVE`.

void **set_goal_positions** (vector<PathPoint2D> *goals*, goal_options_t *options*, Hash64 *job*)
Sets a list of new goals, similar to `set_goal_position(...)`. See *pilot.PathPoint2D*. Requires permission `MOVE`.

void **append_goals** (vector<MapStation> *goals*, goal_options_t *options*, Hash64 *job*)
Appends a list of new goals, similar to `append_goal(...)`. See *pilot.MapStation*. Requires permission `MOVE`.

void **append_goal_stations** (vector<string> *names*, goal_options_t *options*, Hash64 *job*)
Appends a list of new goals, similar to `append_goal_station(...)`. Requires permission `MOVE`.

void **append_goal_positions** (vector<PathPoint2D> *goals*, goal_options_t *options*, Hash64 *job*)
Appends a list of new goals, similar to `append_goal_position(...)`. See *pilot.PathPoint2D*. Requires permission `MOVE`.

Synchronous Movechain Functions

The following functions return when the last goal specified has been physically reached, or the goals were canceled.

void **move_tos** (vector<MapStation> *goals*, goal_options_t *options*, Hash64 *job*)
Same as `set_goals(...)` but will block until last goal is reached or canceled. See *pilot.MapStation*. Requires permission `MOVE`.

void **move_to_stations** (vector<string> *names*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal_stations(...)` but will block until last goal is reached or canceled. Requires permission `MOVE`.

void **move_to_positions** (vector<PathPoint2D> *goals*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal_positions(...)` but will block until last goal is reached or canceled. See *pilot.PathPoint2D*. Requires permission `MOVE`.

3.4.4 LocalPlanner

Module

The *LocalPlanner* module generates velocity commands to keep the robot on a given path or to reach a certain goal which is in reach. Usually the given path is generated by the *HybridPlanner* and depending on configuration further optimized by the *LocalPlanner*.

Most functions require a special permission, see *pilot.permission_e*.

Functions

Common Parameters

goal_options_t options Optional goal options, see *pilot.goal_options_t*. If not specified will use default values.

Hash64 job Optional job id, to identify the new goal. If not specified (or set to zero) will generate a new random id. See *vnx.Hash64*.

General Functions

Path2D ***get_path**() **const**
Returns the current original map path being followed, if any. See *pilot.Path2D*.

Path2D ***get_optimized_path**() **const**
Returns the current optimized map path being followed, if any. See *pilot.Path2D*.

LocalPlannerState **get_state**() **const**
Returns the current planner state. See *pilot.LocalPlannerState*.

Movement Functions

void **pause**(bool *em_stop*)
Will pause movement, similar to an EM stop, but with a soft decceleration. However, if `em_stop` is set to *true* will perform an emergency stop. resume() needs to be called to continue with the current or any new goal.

void **resume**()
Will resume movement after having been paused, while adhering to acceleration limits.

void **await_goal**() **const**
Waits for the current goal to finish. If currently idle it returns immediately.

void **await_goal_ex**(Hash64 *job*) **const**
Waits for a specific goal to finish. Either the currently active or finished goal, or any in the future. If the specified goal has already been reached in the past and subsequently been superseeded by another, this function will fail, ie. it will wait forever. Ideally this function is called before setting the goal to be waited for.

void **cancel_goal**()
Will cancel any pending goal and bring the robot to an immediate stop. Requires permission `MOVE`.

void **cancel_goal_await**()
Same as cancel_goal() and then await_goal() in one call.

void **set_path** (Path2D **path*, goal_options_t *options*)
Sets a new path to be followed and returns immediately. Any pending goal is canceled beforehand. The path needs to be in map coordinates. See *pilot.Path2D*. Requires permission `MOVE`.

void **update_path** (Path2D **path*, goal_options_t *options*)
Updates the current path to be followed without stopping or canceling the current goal. If the new path does not include the current position (to within some tolerance) a failure will be issued. The path needs to be in map coordinates and it's `job` id needs to match the current job id. See *pilot.Path2D*. Requires permission `MOVE`.

void **follow_path** (Path2D **path*, goal_options_t *options*)
Same as `set_path(...)` but will block until the goal is reached or canceled. See *pilot.Path2D*. Requires permission `MOVE`.

void **set_goal** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)
Sets a new goal position, either relative to the current position or an absolute map position. The new goal must be within reach, with a maximum distance depending on configuration, but usually around 1 m. See *pilot.PathPoint2D*. Requires permission `MOVE`.

void **move_to** (PathPoint2D *goal*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal(...)` but will block until the goal is reached or canceled. See *pilot.PathPoint2D*. Requires permission `MOVE`.

3.4.5 PilotServer

Module

The *PilotServer* module is the control center of the robot, it handles switching between different modes such as mapping and navigation or automatic drive mode and teleoperation. In addition it allows to initialize the localization as well as start / stop a data recording.

Most functions require a special permission, see *pilot.permission_e*.

Functions

PilotState **get_state**() **const**
Returns the current pilot state, see *pilot.PilotState*.

PlatformInfo **get_platform_info**() **const**
Returns platform info, see *pilot.PlatformInfo*.

void **set_pose_estimate** (Vector3d *pose*, long *time*)
Initializes the localization by providing a `pose` estimate (x, y, yaw) [m, m, rad] in map coordinates. Optionally a `time` stamp can be provided in [usec]. See *math.Vector3d*. Requires permission `INITIALIZE`.

void **switch_pilot_mode** (pilot_mode_e *new_mode*, Object *config*, bool *keep_motion_mode* = false)
Switches the current pilot mode to `new_mode`. Platform has to be stationary. Optionally a user config can be provided with `config`, overriding default values. If `keep_motion_mode` is set to true, the current motion mode will not be changed automatically. See *pilot.pilot_mode_e* and *vnx.Object*. Requires permission `REMOTE_CONTROL`.

void **switch_motion_mode** (motion_mode_e *new_mode*)
Switches the current motion mode to `new_mode`. Platform has to be stationary. See *pilot.motion_mode_e*. Requires permission `REMOTE_CONTROL`.

void **switch_footprint** (Footprint *footprint*)
Sets a new footprint to use. Platform has to be stationary. See *pilot.Footprint*. Requires permission `REMOTE_CONTROL`.

void **start_recording** (string *file_name*)
Starts a new data recording using the optional file name `file_name`. The recording will end up in the `~/pilot/user/data/` folder. If `file_name` is empty a default name will be chosen. In any case a timestamp will be appended to the file name to make it unique. Requires permission `RECORD_DATA`.

void **stop_recording** ()
Stops the current data recording process. Requires permission `RECORD_DATA`.

3.4.6 PlatformInterface

Interface

The *PlatformInterface* is an interface to the hardware abstraction module running on a platform. It allows access to hardware features such as digital inputs / outputs, analog inputs, power relays and the LCD.

Most functions require a special permission, see *pilot.permission_e*.

Functions

Charging Functions

void **charge** ()
Attempts to charge the platform's batteries when connected to a charging station. Will return upon completion of charging process. In case of an error (such as no contact to the charging station) an exception will be thrown. Requires permission `CHARGE`.

void **start_charging** ()
Attempts to start the charging process at a charging station. Returns immediately in case of success, or throws an exception. Requires permission `CHARGE`.

void **stop_charging** ()
Stops the charging process at a charging station. Returns immediately in case of success, or throws an exception. Requires permission `CHARGE`.

Input / Output Functions

void **set_relay** (int *channel*, bool *state*)
Switch a relay on or off, depending on `state`. `channel` is an index from 0 to 3 (in most cases) denoting the relay to switch. Requires permission `RELAY_CONTROL`.

void **set_digital_output** (int *channel*, bool *state*)
Switches a digital output to the specified `state`. `channel` is an index from 0 to 15 (in most cases) denoting the output pin. Requires permission `RELAY_CONTROL` as well as an installed *IOBoard*.

void **set_display_text** (string *line*)
Sets the first line on the LCD to the given text, for the next 30 seconds. Requires permission `DISPLAY_CONTROL`.

void **wait_for_digital_input** (int *channel*, bool *state*)
Waits for a digitial input to reach the specified `state`. If it's already reached will return immediately. `channel` is an index from 0 to 15 (in most cases) denoting the input pin. Requires permission `CONST_REQUEST` as well as an installed *IOBoard*.

Other Functions

void **shutdown**()
Will terminate the process, shutdown the machine and turn off the platform. Cannot be undone except by turning on the platform by hand again. Requires permission `SHUTDOWN_HOST`, see *vnx.permission_e*.

3.4.7 RoadMapPlanner

Module

The *RoadMapPlanner* module provides navigation support based on a *Road Map*.

Most functions require a special permission, see *pilot.permission_e*.

Functions

Common Parameters

goal_options_t options Optional goal options, see *pilot.goal_options_t*. If not specified will use default values.

Hash64 job Optional job id, to identify the new goal. If not specified (or set to zero) will generate a new random id. See *vnx.Hash64*.

General Functions

RoadMapData ***get_road_map**() **const**
Returns the currently used *Road Map*, if any. See *pilot.RoadMapData*.

MapStation ***find_station**(string *name*) **const**
Returns the corresponding map station by that name, if any. See *pilot.MapStation*.

Movement Functions

void **set_goal_station**(string *name*, goal_options_t *options*, Hash64 *job*)
Sets a new goal using the provided station `name` and returns immediately. Any pending goals are canceled beforehand. The station must exist in the current *Road Map*. Requires permission `MOVE`.

void **move_to_station**(string *name*, goal_options_t *options*, Hash64 *job*)
Same as `set_goal_station(...)` but will block until goal is reached or canceled. Requires permission `MOVE`.

3.4.8 TaskHandler

Module

The *TaskHandler* module allows the execution of a sequence of tasks, specified by a Lua script, which can either be written by hand or generated by using the grahpical programming interface TaskEditor.

See *Lua Script* for more information on how to write a script by hand.

Most functions require a special permission, see *pilot.permission_e*.

Functions

Common Parameters

Hash64 jobid Optional job id, to identify a new program execution. If not specified (or set to zero) will generate a new random id. See *vnx.Hash64*.

Execute Functions

The following functions will cancel any running program beforehand, start to execute the new program and return immediately.

void **execute_file**(string *filename*, Hash64 *jobid*)
Starts to execute the specified Lua script file. The `filename` is relative to `~/pilot/`. Requires permission `EXECUTE_SCRIPT`.

void **execute_program**(string *program*, Hash64 *jobid*)
Starts to execute the given Lua script code. Requires permission `UPLOAD_SCRIPT`.

Intervene Functions

void **cancel_program**()
Will cancel a running program at the next opportunity, usually after finishing the current task. Requires permission `INTERVENE_SCRIPT`.

void **pause_program**()
Will pause a running program at the next opportunity, usually after finishing the current task. Requires permission `INTERVENE_SCRIPT`.

void **resume_program**()
Will resume executing a paused program. Requires permission `INTERVENE_SCRIPT`.

3.4.9 vnx.JRPC_Proxy

Module

The *vnx.JRPC_Proxy* module provides a way to connect to another *vnx.JRPC_Server*.

This module and the *vnx.Proxy* module share the same base and are largely identical in handling.

Options

The options are identical to *vnx.Proxy*.

Methods

The module has all methods of *vnx.Proxy* and additionally the following ones.

void **select_service**(string *service_name*)
For the rest of the connection, all method calls by *JSON-RPC Interface* that do not specify a module will be directed to `service_name`.

Supply an empty string or `.` to reset it to the proxy module itself.

3.4.10 vnx.JRPC_Server

Module

The *vnx.JRPC_Server* module provides a JSON RPC server to interface with the running process.

On the other end a *vnx.JRPC_Proxy* is needed to connect with the server.

This module and the *vnx.Server* module share the same base and are largely identical in handling.

Options

Identical to *vnx.Server*.

3.4.11 vnx.Proxy

Module

The *vnx.Proxy* module provides a way to connect to another *vnx.Server*.

Options

string **`address`**
: URL to connect to, for example `localhost:1234` (TCP/IP) or `/tmp/mysocket.sock` for a UNIX socket. If no TCP port is specifed (ie. `localhost`) a default port of `4444` is used. If no IP address is specified (ie. `:1234`) a default of `localhost` is used.

vector<string> **`import_list`**
: List of topics to import from the sever.

vector<string> **`export_list`**
: List of topics to export to the server.

vector<string> **`forward_list`**
: List of services to forward to the server. Usually a service is a module's name. Requests to these services are then sent to the server who will forward them to the actual modules processing them. Return messages are automatically routed back to the clients.

map<Hash64, string> **`tunnel_map`**
: Same as *forward_list* but with a different service address locally [*local alias* => *remote service name*]. Will forward requests on the `Hash64` address to the `string` service on the server. `string` service names are converted to a `Hash64` address internally. See *vnx.Hash64*.

map<string, string> **`import_map`**
: Same as `import_list` but will map to a different topic name locally. [*remote name* -> *local name*]

map<string, string> **`export_map`**
: Same as `export_list` but will map to a different topic name on the server. [*local name* -> *remote name*]

bool **`auto_import`** = false
: If to import all subscribed to topics from the server. Only used in special cases, since most of the time it leads to network loops.

bool **`time_sync`** = false
: If to syncronize local virtual time with the server. Same as importing the `vnx.time_sync` topic. `vnx::get_time_*()` functions will then return a time which is in sync with the server.

bool **block_until_connect** = true
If to block client requests until first sucessful connect.

bool **block_until_reconnect** = false
If to block client requests until next sucessful reconnect, in case connection breaks down.

int **max_queue_ms** = 100
Maximum queue length in [ms] for receiving data internally, 0 = unlimited. If the network is overloaded at most this amount of data (in terms of latency) will be buffered internally before starting to drop messages. This queue is in addition to the internal TCP send buffer.

int **max_queue_size** = 1000
Maximum queue size in [number of messages] for receiving data internally, 0 = unlimited. Similar to *max_queue_ms*.

int **recv_buffer_size** = 0
TCP receive buffer size (0 = default) [bytes]

int **send_buffer_size** = 131072
TCP send buffer size, bigger equals more latency in case of network overload. (0 = default) [bytes]

string **default_access** = "DEFAULT"
Default access level for anonymous clients, see *User Management*. Only if *use_authentication* = *true*, otherwise there are no restrictions, ie. full permissions to any user.

3.4.12 vnx.Server

Module

The *vnx.Server* module provides a VNX server to interface with the running process.

On the other end a *vnx.Proxy* is needed to connect with the server.

Options

string **address**
URL to bind to, for example `0.0.0.0:1234` (TCP/IP) or `/tmp/mysocket.sock` for a UNIX socket. UNIX sockets need to have a file ending of `*.sock`. If no TCP port is specifed (ie. `0.0.0.0`) a default port of `4444` is used. If no IP address is specified (ie. `:1234`) a default of `localhost` is used.

bool **use_authentication** = false
If to require user authentication (login) to gain more persmissions than `default_access`. If set to `false` any user has full permissions and no login is necessary.

vector<string> **export_list**
List of topics to automatically export to all clients without them asking for it. Samples published on these topics are forwarded to all clients and re-published there.

int **max_queue_ms** = 100
Maximum queue length in [ms] for receiving data internally, 0 = unlimited. If the network is overloaded at most this amount of data (in terms of latency) will be buffered internally before starting to drop messages. This queue is in addition to the internal TCP send buffer.

int **max_queue_size** = 1000
Maximum queue size in [number of messages] for receiving data internally, 0 = unlimited. Similar to *max_queue_ms*.

int **recv_buffer_size** = 0
TCP receive buffer size (0 = default) [bytes]

int **send_buffer_size** = 131072
TCP send buffer size, bigger equals more latency in case of network overload. (0 = default) [bytes]

string **default_access** = "DEFAULT"
Default access level for anonymous clients, see *User Management*. Only if *use_authentication* = *true*, otherwise there are no restrictions, ie. full permissions to any user.

3.4.13 vnx.opc_ua.Proxy

Module

The *vnx.opc_ua.Proxy* module provides a OPC-UA proxy to connect to a OPC-UA server.

It allows to call methods as well as read variables on the server.

Options

string **address**
OPC-UA server url to connect to, for example `opc.tcp://127.0.0.1:4840`.

bool **block_until_connect** = true
If to block client requests until first sucessful connect.

bool **block_until_reconnect** = false
If to block client requests until next sucessful reconnect, in case connection breaks down.

Functions

UA node ids are provided via a `pair<ushort, Variant>`, where the first value is the namespace index and the second value either an integer or a string.

In case a UA method returns more than one value the `Variant` return value will contain an array of variants, ie. `vector<Variant>`. See also *vnx.Variant*.

Variant **call** (string *method*, vector<Variant> *args*) **const**
Calls a global `method` with the given arguments and returns the result. The implicit object for this call is *UA_NS0ID_OBJECTSFOLDER*. Requires permission `vnx.opc_ua.permission_e.CALL`.

Variant **object_call** (pair<ushort, Variant> *object*, string *method*, vector<Variant> *args*) **const**
Calls a `method` on the `object` with the given arguments and returns the result. Requires permission `vnx.opc_ua.permission_e.CALL`.

Variant **read_variable** (pair<ushort, Variant> *node*) **const**
Reads the value of a given variable `node`.

Variant **read_object_variable** (pair<ushort, Variant> *object*, string *variable*) **const**
Reads the value of a given `variable` of the `object`.

void **write_variable** (pair<ushort, Variant> *node*, Variant *value*)
Writes a value to the given global variable `node`. Requires permission `vnx.opc_ua.permission_e.WRITE`.

void **write_object_variable** (pair<ushort, Variant> *object*, string *variable*, Variant *value*)
Writes a value to the given `variable` of the `object`. Requires permission `vnx.opc_ua.permission_e.WRITE`.

void **browse_all** ()
Finds all available objects and variables on the server. Will be done automatically on every connect and reconnect.

3.4.14 vnx.opc_ua.Server

Module

The *vnx.opc_ua.Server* module provides a OPC-UA server to interface with the running process.

Depending on the configuration different modules and topics are offered on the OPC-UA interface.

Options

set<string> **export_services**
List of modues to offer on the OPC-UA interface as objects.

set<string> **export_topics**
List of topics to offer on the OPC-UA interface as variables. (Not implemented yet)

bool **use_authentication** = false
If to require authentication from clients. (Not implemented yet)

string **default_access** = "DEFAULT"
Default access level for anonymous clients, see *User Management*. Only if *use_authentication* = *true*, otherwise there are no restrictions, ie. full permissions to any user.

3.5 Topics

Topics are names under which data samples are published. Any number of subscribers can then receive the published data.

Topics are arranged in a tree and it is possible to subscribe to a whole sub-tree. For example a subscription to `input` will receive samples from `input.joy` as well as `input.velocity_cmd`.

Topics do not have an assigned data type, but in most cases only a specific data type is published.

3.5.1 input

input.joy [*pilot.JoyData*] Joystick input samples, published only when a Joystick is connected and controls are operated.

input.velocity_cmd [*pilot.VelocityCmd*] Joystick velocity commands.

3.5.2 local_planner

local_planner.state [*pilot.LocalPlannerState*] *LocalPlanner* state updates.

local_planner.target_pose [*pilot.Pose2D*] *LocalPlanner* target pose updates.

3.5.3 localization

localization.map_tile [*pilot.OccupancyMapData*] Localization map tile, ie. current section of the *Grid Map*.

localization.particles [*pilot.PoseArray2D*] Localization particle swarm in `map` coordinates.

localization.status [*pilot.LocalizationStatus*] Localization status updates.

3.5.4 mapping

mapping.grid_map [*pilot.OccupancyMapData*] New *Grid Map* created by mapping.

mapping.pose_graph [*pilot.RoadMapData*] Mapping pose graph.

3.5.5 movechain

movechain.status [**pilot.MovechainStatus**] MovechainHandler status updates.

3.5.6 navigation

navigation.footprint [*pilot.Footprint*] Platform footprint.

navigation.global_cost_map [*pilot.CostMapData*] Global cost map.

navigation.global_path [*pilot.Path2D*] Current global path in `map` coordinates.

navigation.grid_map [*pilot.OccupancyMapData*] Current *Grid Map*.

navigation.initial_pose [*pilot.Pose2D*] Pose estimates to intialize Localization.

navigation.local_cost_map [*pilot.CostMapData*] Local cost map.

navigation.local_cost_map_overlay [*pilot.CostMapData*] Local cost map with global cost map overlayed.

navigation.local_path [*pilot.Path2D*] Current optimized local path in `odom` coordinates.

navigation.map_match [*pilot.MapMatch*] Current map match.

navigation.map_pose [*pilot.Pose2D*] Current map pose in `map` coordinates.

navigation.new_goal_pose [*pilot.Pose2D*] New goals can be published here.

navigation.new_grid_map [*pilot.OccupancyMapData*] New *Grid Map* can be published here.

navigation.new_road_map [*pilot.RoadMapData*] New *Road Map* can be published here.

navigation.odom_pose [*pilot.Pose2D*] Current local pose in `odom` coordinates.

navigation.road_map [*pilot.RoadMapData*] Current *Road Map*.

navigation.velocity_cmd [*pilot.VelocityCmd*] Velocity commands from *LocalPlanner*.

3.5.7 network

network.beacons [*pilot.Beacon*] Beacons for fleet management.

3.5.8 platform

platform.battery_state [*pilot.BatteryState*] Battery state updates.

platform.emergency_state [*pilot.EmergencyState*] Emergency state updates.

platform.events [*pilot.Event*, *pilot.Incident*] Generic events.

platform.incidents [*pilot.Incident*] External incidents, to be handled by *PilotServer*.

platform.active_incidents [*pilot.ActiveIncidents*] List of currently active incidents.

platform.info [*pilot.PlatformInfo*] Static platform info.

platform.odometry [*pilot.Odometry*] Odometry samples.

platform.pilot_state [*pilot.PilotState*] Pilot state updats.

platform.system_state [*pilot.SystemStatus*] System state update.

platform.velocity_cmd [*pilot.VelocityCmd*] Custom velocity commands.

3.5.9 sensors

sensors.point_cloud.* [*pilot.PointCloud2D*, *pilot.LaserPointCloud*] Laser point clouds in `odom` coordinates.

3.5.10 task_handler

task_handler.current_task [*pilot.ExecutionState*] Current task being executed by *TaskHandler* module.

task_handler.current_event_task [*pilot.ExecutionState*] Current event task being executed by *TaskHandler* module.

task_handler.execution_history [*pilot.ExecutionHistory*] Execution history of *TaskHandler* module.

task_handler.event_history [*pilot.ExecutionHistory*] Event execution history of *TaskHandler* module.

3.5.11 tf

tf.map.odom [*pilot.Pose2D*] Localization updates.

tf.odom.base_link [*pilot.Odometry*] Odometry samples.

3.5.12 tfd.map

tfd.map.local_planner.target_pose [*pilot.Pose2D*] Same as *local_planner.target_pose* but in `map` coordinates.

tfd.map.navigation.local_path [*pilot.Pose2D*] Same as *navigation.local_path* but in `map` coordinates.

tfd.map.sensors.point_cloud.* [*pilot.Pose2D*] Same as *sensors.point_cloud.** but in `map` coordinates.

3.5.13 vnx

`vnx.log_out` [*vnx.LogMsg*] Terminal log messags.

`vnx.module_info` [*vnx.ModuleInfo*] Module info updates.

`vnx.recorder_status` [**vnx.RecorderStatus**] Status information of the data recorder, when active.

Indices and tables

- genindex
- modindex
- search

INDEX

R

S

U

V

W