nexxT Documentation

ifm electronic gmbh

Mar 24, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

nexxT is a hybrid python/c++ framework targeted mainly at computer vision algorithm developers. Developers can build a graph of plugins interacting via data-driven ports.

1.1 Design Principles

- Reliable, synchronized data transport across threads. Plugins usually have a thread they run in and do not need to care about locking and data consistency issues. The framework guarantees that all callback methods are called in the thread of the plugin unless explicitely otherwise stated.
- A state machine guarantees consistent behaviour in initialization, active and shutdown phases across plugins and threads.
- Non-intrusive design. Unlike other frameworks, nexxT tries its best to leave the developers the freedom they need. No directory structures are predefined, no build tools are required, the data formats are not predefined.
- Rapid prototyping of algorithms using python, both online (i.e. using an active sensor) and offline (i.e. using data from disk).
- Visualization can be done using python visualization toolkits supporting QT6.
- Efficient pipelines can be built without interacting with the python interpreter by using only C++ plugins.
- The open source license gives freedom to adapt code if necessary.
- Cross platform compatibility for windows and linux.

We used Enterprise Architect for the initial design, and most of the design decisions are still somewhat valid, especially the part about data transport. The design can be found here: https://github.com/ifm/nexxT/blob/master/design/NeXT.eap

1.2 Documentation

The documentation is hosted on readthedocs.

1.3 Installation

It is highly recommended to use the binary packages from pypi in a virtual environment.

1.3.1 Linux

Assuming that you have a python3.7+ interpreter in your path, the installation can be done with

```
python3 -m venv venv_nexxT
source venv_nexxT/bin/activate
python3 -m pip install pip -U
pip install nexxT
```

1.3.2 Windows

Assuming that you have a python3.7+ interpreter in your path, the installation is very similar

```
python -m venv venv_nexxT
.\venv_nexxT\Scripts\activate
python -m pip install pip -U
pip install nexxT
```

1.4 Porting from nexxT 0.x to nexxT 1.x (aka PySide2 to PySide6)

1.4.1 Python

The main change for nexxT 1.x is the update from QT5/PySide2 to QT6/PySide6. For flexibility reasons, nexxT now provides a meta package nexxT.Qt, which can be used instead of PySide6. So it is now recommended to replace

```
from PySide2 import xyz
from PySide2.QtWidgets import uvw
```

with

```
from nexxT.Qt import xyz
from nexxT.Qt.QtWidgets import uvw
```

In the future, this approach might be also used to support PyQt6, so using nexxT.Qt is recommended over the also possible direct usage of PySide6. Note that the implementation of nexxT.Qt imports the PySide moduels on demand using sys.meta_path, so unused QT modules are not loaded.

Note the porting guide of PySide6: https://doc.qt.io/qtforpython/porting_from2.html: QAction and QShortcut have been moved from QtWidgets to QtGui.

1.4.2 C++

For c++, nexxT includes shall be prefixed with nexxT/, for example

```
#include "Filters.hpp"
```

has to be replaced with

```
#include "nexxT/Filters.hpp"
```

1.5 Building from source

Building from source requires a QT6 installation suited to the PySide6 version used for the build. It is ok to use 6.4.0 to build against all versions 6.4.x of PySide6 because of QT's binary compatibility. You have to set the environment variable QTDIR to the installation directory of QT. Note that this installation is only used during build time, at runtime, nexxT always uses the QT version shipped with PySide6.

On linux, you will also need llvm and clang installed (because of the shiboken6 dependency). You might need to set the environment variable LLVM_INSTALL_DIR.

The following commands build nexxT from source using the non-recommended pip package of shiboken6-generator.

```
git clone https://github.com/ifm/nexxT.git
cd nexxT/workspace
python3 -m venv venv
source venv/bin/activate
python3 -m pip install pip -U
pip install -r requirements.txt --find-links https://download.qt.io/official_releases/
↪QtForPython/shiboken6-generator/
export QTDIR=<path>/<to>/<qt>
export LLVM_INSTALL_DIR=<path>/<to>/<llvm>
scons -j 8 ..
```

When using setup.py to install nexxT, the above requirements shall be also fulfilled and scons is called implicitely from setup.py. Installation from source without using the wheel package is not supported.

1.6 History

Originally we started with a commercial product from the automotive industry in our development, due to the requirements of a project at that time. That product had become more and more outdated and the management was not very keen on paying maintainance costs for updates. After very long discussions about the way forward, we finally got the go for developing an own framework. We have decided to apply an open-source license to it such that it might be useful to other people having the same pain than us as well. During the discussion phase we discussed also other alternatives, and here especially the use of *ROS2*, which claimed to fulfill many of our requirements. We decided against it because of the following reasons:

- The windows support was not production-ready when we tested it. Many *ROS2* components seem to run only on linux, even core *ROS2* components didn't work well on windows. I also experienced hardcoded paths to C:\python37 without the chance to use virtual environments.
- The *ROS2* design is very intrusive. You can't deviate from existing directory structure, build system conventions and operating system versions (some ROS tutorials even suggest to use a VM dedicated for a specific *ROS* version). Side-by-side installations of different ROS versions therefore are difficult.

- The data transport layer of *ROS2* seemed not to fulfill our requirements. We have often use cases where we record data to disk and develop algorithms using that data. Because *ROS2* is mainly designed as a system where algorithms run online, its assumptions about the data transport is low latency in prior of reliability. If in question, *ROS2* decides to throw away messages, and this is very bad for the offline/testing usage when you have not such a big focus on algorithm runtime but maybe more on algorithm quality performance. In this use-case it is a reasonable model that slow algorithms shall be able to slow down the computation, but at the time of evaluation this was not easily possible in *ROS2*. A discussion about this topic can be found here: https://answers.ros.org/question/336930/ros2-fast-publisher-slow-subscriber-is-it-possible-to-slow-down-the-publisher/
- It's not easily possible to start two *ROS2* applications side by side.

CHAPTER

TWO

TUTORIAL

The purpose of this tutorial is to introduce the nexxT framework from different perspectives. We start with the perspective of a user who wants to be able to use predefined configurations. Then we show the perspective of an integrator who creates and manipulates configurations of filter graphs using nexxT. The last perspective is the developer view who develops filters ready to be integrated into the framework accessing both the python and the C++ API.

2.1 nexxT Nomenclature

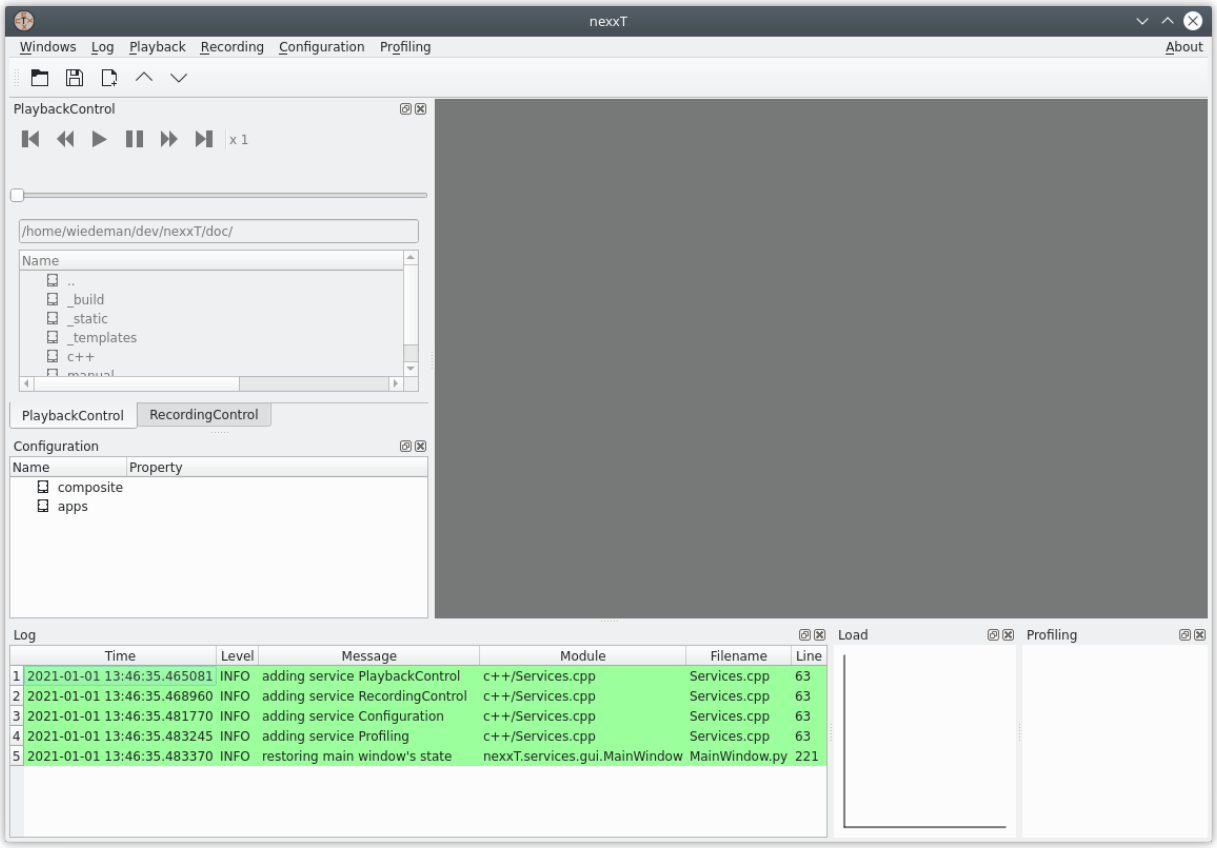
Terms and definitions: A nexxT *configuration* provides one or more *applications*. *Applications* consist of a directed graph where the nodes are *filter* instances. *Filter* instances have input and output *ports*. Edges from output to input *ports* represent data flow *connections*. The *filters* can provide *properties* which can be manipulated in the GUI. For a clearer view it is possible to group subgraphs in so-called *composite filters*.

2.2 User Perspective

After installing nexxT as described in the *Installation* section, the command *nexxT-gui* is available in the virtual environment. Starting it with

```
$ nexxT-gui
```

brings up the nexxT graphical user interface:



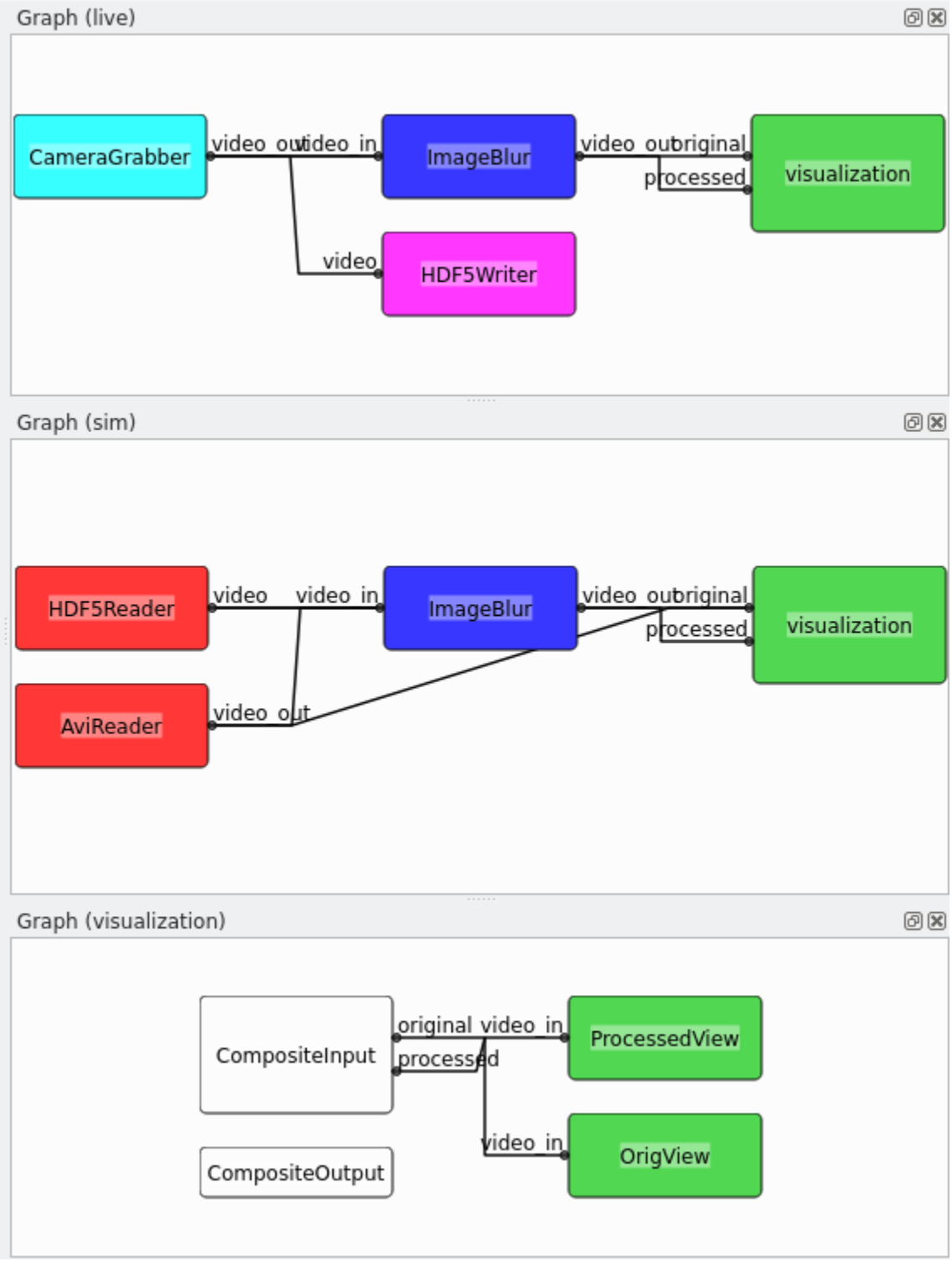
2.2.1 GUI walkthrough

The GUI consists of a main window with a menu, a toolbar and several dock windows. We are going to walk through the provided functionality briefly:

- The **Configuration** dock window and the Configuration menu are the main places where the user interacts with the configuration. It is possible to load and save configuration files, start and stop applications, modify property values and also manipulate the underlying filter graph.
- The **Playback Control** dock window and the Playback menu are the places where the user browses for files and activates them for playback. The window is enabled only if an application with a playback filter is active. It also provides the familiar control elements for navigating through a data file.
- The **Recording Control** dock window and the Recording menu provide control for recording data. The window is enabled only if an application with a recording filter is active.
- The **Log** window contains messages from both nexxT itself and the active plugins. It can be configured through the Log menu.
- The **Load** window shows the workload of the threads used in the application (when applications are active).
- The **Profiling** window shows detailed filter timings if enabled through the Profiling menu.

2.2.2 Example configuration

The example configuration is installed in *site-packages*/nexxT/examples/framework/example.json (the *site-packages* folder is inside the virtual envirnment where you have installed nexxT). This file contains two applications *live* and *sim* and a composite filter *visualization*. You can inspect the filter graphs by right-clicking the item in the configuration view and selecting *Edit Graph*:



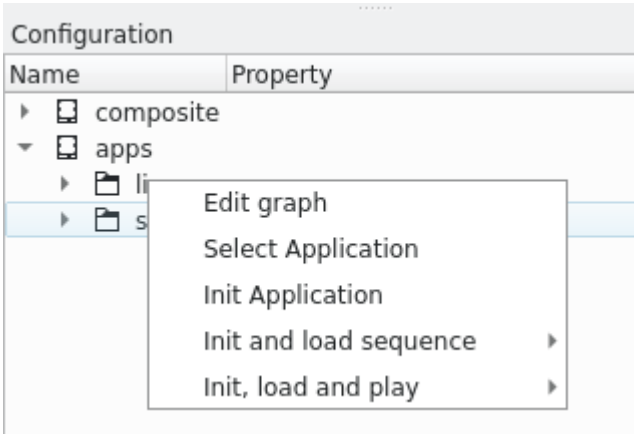
The *live* application contains a camera grabber, an image processing filter, an HDF5 recorder and the visualization. The different colors in the graph are related to the different threads the filters run in. This application is ment to be executed live with a real camera. The CameraGrabber uses the first camera found on the system with default settings.

The *sim* application contains two readers for different flavours of input files (HDF5 or standard avi or mp4 containers), also the image processing filter and the visualization. This application is ment to be executed offline with data from disk. This allows to experiment with parameters and other algorithms.

The image processing here is a placeholder for a computationally expensive operation. The blurring is implemented in a non-optimal way for the benefit of minimizing project dependencies. One would obviously use libraries such as opencv or scipy for performing this task in a productivity environment.

2.2.3 Executing the live application

Right-clicking

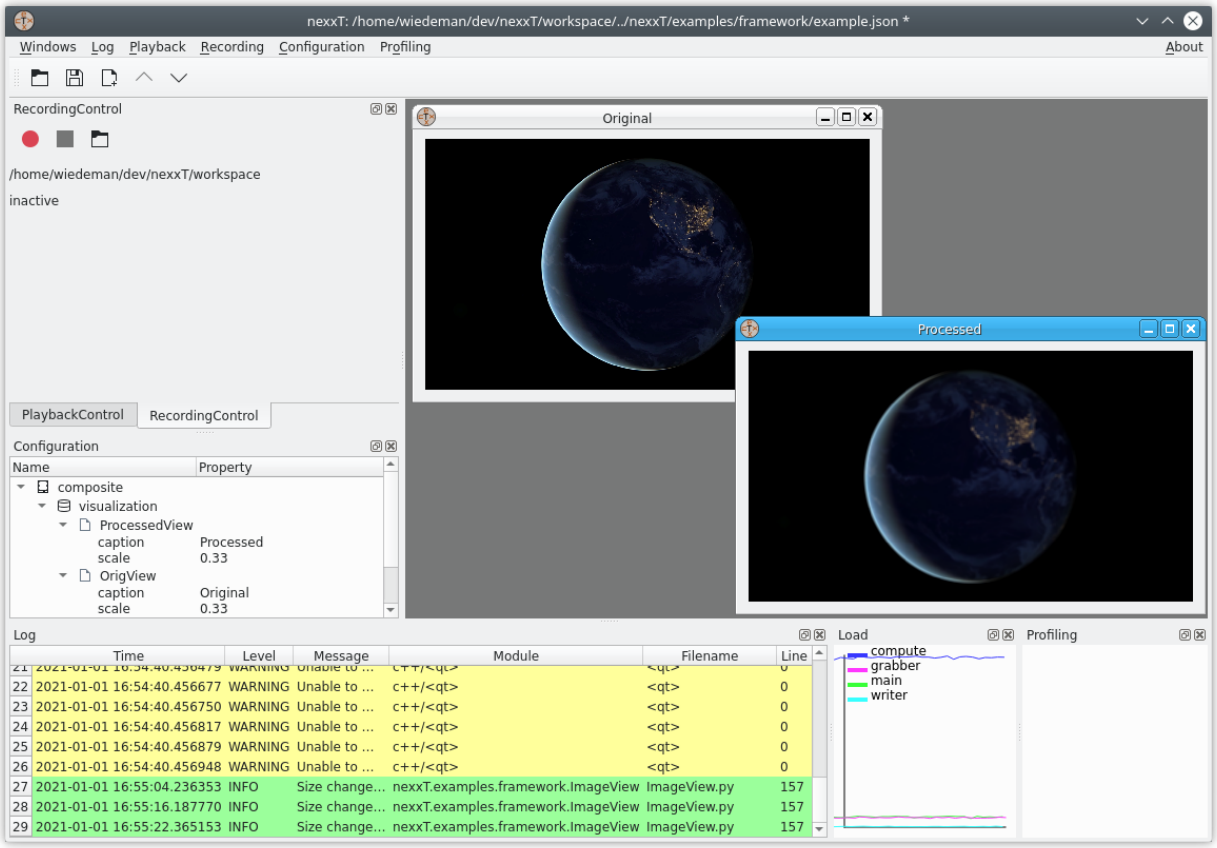


on one of the applications shows different options for activating the application:

Table 1: Application Context Menu

Context menu entry	Effect
Select application	Application is selected but left uninitialized
Init application	Application is initialized. A Playback file is not yet selected.
Init and load sequence	Application is initialized and the selected playback file is loaded. Playback is not yet started (only relevant for playback apps)
Init, load and play	Application is initialized and the selected playback file is loaded. Playback is started (only relevant for playback apps)

Execute *Init application* of the live configuration. If a camera is connected to your computer, you should see two images, the original image as delivered from the camera and the processed, blurred image from the image processing filter.



The load monitor shows the load of the individual threads. In this example, the load introduced by the image processing is significant - the blue line is nearly always 100 %. This means that the image processing filter is probably slowing down the processing of the graph.

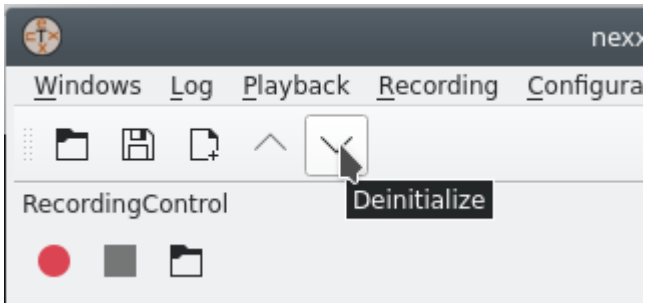
You might also see some warnings in the log monitor. They come from the QMultimedia framework package which is used to grab the images. Note that this choice has been made for demonstration purposes only and it minimizes the project's dependencies. In a productivity environment, the capturing filter might be replaced with a wrapper around a better suited library (e.g., opencv).

You are now able to take a recording using the buttons of the Recording Control toolbar.

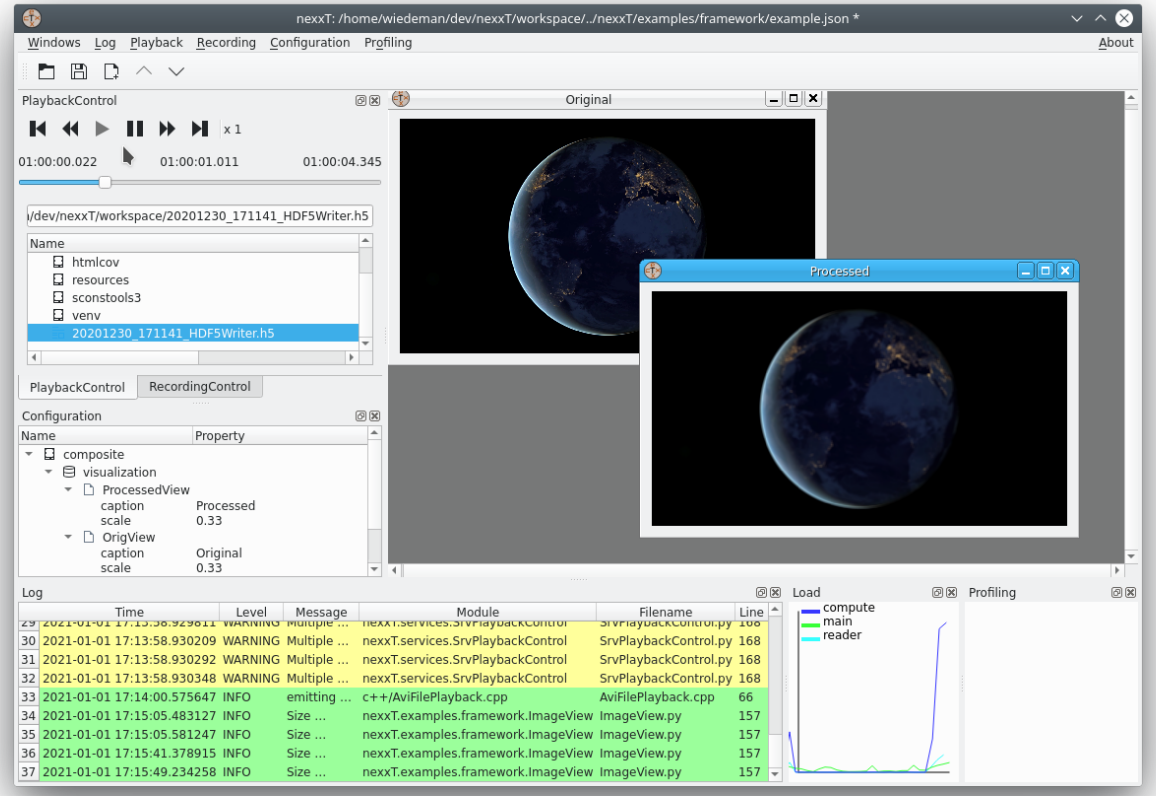
You can also change properties in the configuration view. In the example above, the scale property of the visualization filters has been changed to better fit to the screen dimensions. Note that there are settings with an immediate effect (e.g., scale) while other settings need a re-initialization of the application before taking effect (e.g., caption). It's up to the filter developer to decide about the effect of changed properties.

2.2.4 Executing the sim application

Clicking on the deinitialize



button on the toolbar sets the gui into the uninitialized state. You can now start the *sim* application to replay the recording you've just made. Therefore, right-click on the sim application and execute *Init application*. The *Playback Control* window is activated now and you can navigate to the .h5 file recording. When the file is activated in the browser, the *Play* button is enabled. Click it to start the playback from the recording.

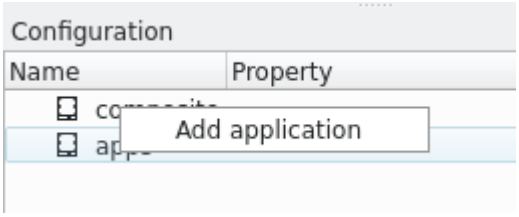


In principle it shall be possible to also play .mp4 files with this application. Due to the limitations of the QMultimedia framework package the supported file types are limited.

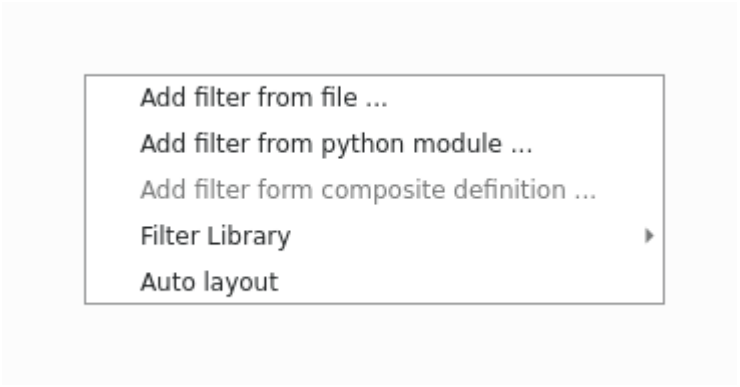
2.3 Integrator Perspective

The integrator's perspective on the nexxT framework is to maintain and modify existing or create new configurations. The integrator uses ready-to-use plugins from developers (i.e., it is not strictly necessary to have programming skills for performing this task). The main tool for achieving these tasks is the *Graph Editor* (see *Example configuration*). In the next paragraphs, we show how to create a new configuration:

Click on the *New config* button in the toolbar and choose an appropriate location. Right-click the *apps* entry in the **Configuration** dock window and choose *Add application*:



You can change the name of the new application by pressing F2 in the configuration view. Start the graph editor of this application. The graph will be empty, and we are going to populate it. Right-clicking on an empty space in the graph editor brings up this pop-up menu:



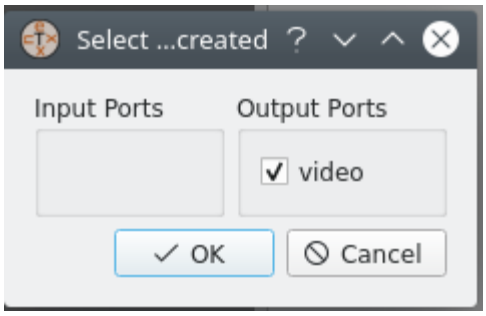
As you can see, there are multiple options to add filters:

Table 2: Graph Editor Context Menu

Context menu entry	Effect	Discussion
Add filter from file	You can choose a python file or a shared object / DLL. This file will be scanned for Filter classes and a list of available filters will be presented. In case a python file is created, this file is imported as a python module.	While this method is pretty easy and straight-forward, importing python code like this doesn't scale up. It is recommended that filters are imported by one of the two other ways. As of today, nexxT uses an absolute path to the file. After saving a config, it might be necessary to change these paths in the JSON file to be relative to the configuration file.
Add filter from python module	You can choose a python module from the current python environment (dot notation can be used for packages).	This method is preferrable over the file method, if your filter is packaged in a standard python package which is installed via pip or similar tools. However, integrators have to remember the python module names of their filters.
Filter Library	Filters which are registered with the 'nexxT.filters' entry point (see *nexxT.interface.Filters.FilterSurrogate* for more information) are listed here in a tree structure. This is possible for filters defined in both domains, python and C++.	Most convenient approach for integrators. The method requires that the filter is installed with the correct entry_point definition in setup.py or setup.cfg.

Let's add a HDF5 reader to the application, using the filter library (*Filter Library* -> *harddisk* -> *HDF5Reader*). You will be presented with an empty filter:

The HDF5 reader uses the concept of **dynamic ports** for being able to support arbitrary streams. These dynamic ports have to be defined by the integrator. Filters and ports provide context menus where dynamic ports can be added, removed and renamed. Each stream in the HDF5 file is then mapped to a dynamic output port with the name of the stream. The HDF5 reader can suggest port names from a template HDF5 file. Therefore, click on *Suggest dynamic ports ...* and choose the HDF5 file which was recorded in *Executing the live application*. You are presented with this suggestion:

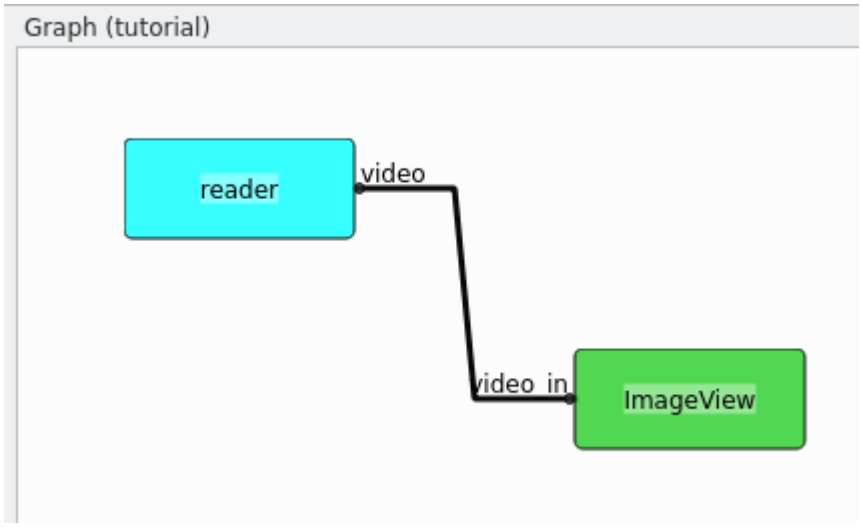


After acceptance, the video port is added to the filter. Another way to achieve the same effect is to use the filter's context menu and add a new dynamic output port with name video.

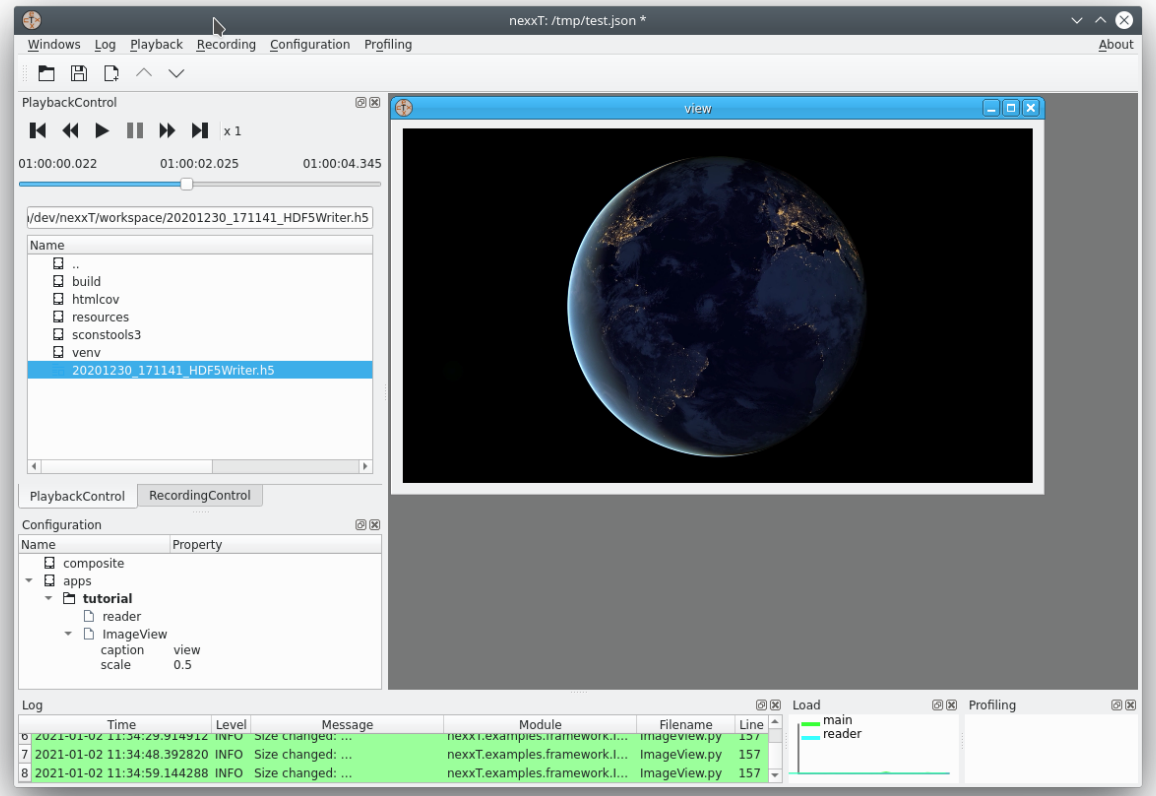
By default, new filters are running in the main thread (indicated by the green color). However, the HDF5 reader is designed to run in a seperate thread instead and it will emit a warning in the logs if it runs in the main thread. Therefore, we assign a new thread *reader* to it using the filter's context menu. Note the color change after the operation.

Next, we will add the visualization node for the video stream to the graph editor (*Filter Library* -> *examples* -> *framework* -> *ImageView*). The green color indicates that this filter is running in the main thread. Visualization filters are required to run in the main (aka GUI) thread, because of QT's requirements for gui elements. This filter defines some properties which can be changed in the configuration view, namely *scale* and *caption*.

We can now connect the reader's output port with the visualization's input port by dragging a connection between the two:



Now it's time to save the configuration in the tool bar and test it. Initialize the application with the context menu in the configuration view and choose the .h5 file from the previous section. You should see the recorded data:



2.3.1 Properties and Variables

Filters may publish configuration options with the property subsystem. Properties have an associated type (float, int, string, bool) and filter developers might add constraints like minimum or maximum value, string enums, etc. The properties can be set in the GUI in the **Configuration** dock window under the respective filter instance.

Starting with version *1.1.0*, optional variable substitution was introduced into nexxT. Variables can be defined in the **Configuration** dock window either globally for the whole configuration or at instances of composite filters. For example, if a sensor is connected via ethernet, it might be useful to define a global variable `IP` and reference the IP address in filters indirectly via the variable. When the IP address changes, it only needs to be changed once in the configuration and the change propagates to all filters where the `$IP` is used. Another use case is the definition of composite filters which are configured by variables defined at the instances of the filter.

Variable substitution in filter properties must be turned on explicitely by setting the `indirect` flag in the configuration window.

Variable substitution behaves similar to shell variable substitution. In addition, variables may be set to the special syntax `${!<python expression>}` to expand to the value of the given python expression. Note that these expressions cannot be embedded in longer substitutions, but need to be assigned to standalone variables. Expressions have access to the importlib module, so they can use installed python packages and the standard library. The `subst(...)` function can be used to substitute strings with other variables.

Here are some examples:

- `VAR=hello $WORLD`, `WORLD=world`; both `$VAR` and `${VAR}` will expand recursively to the string `"hello world"`.

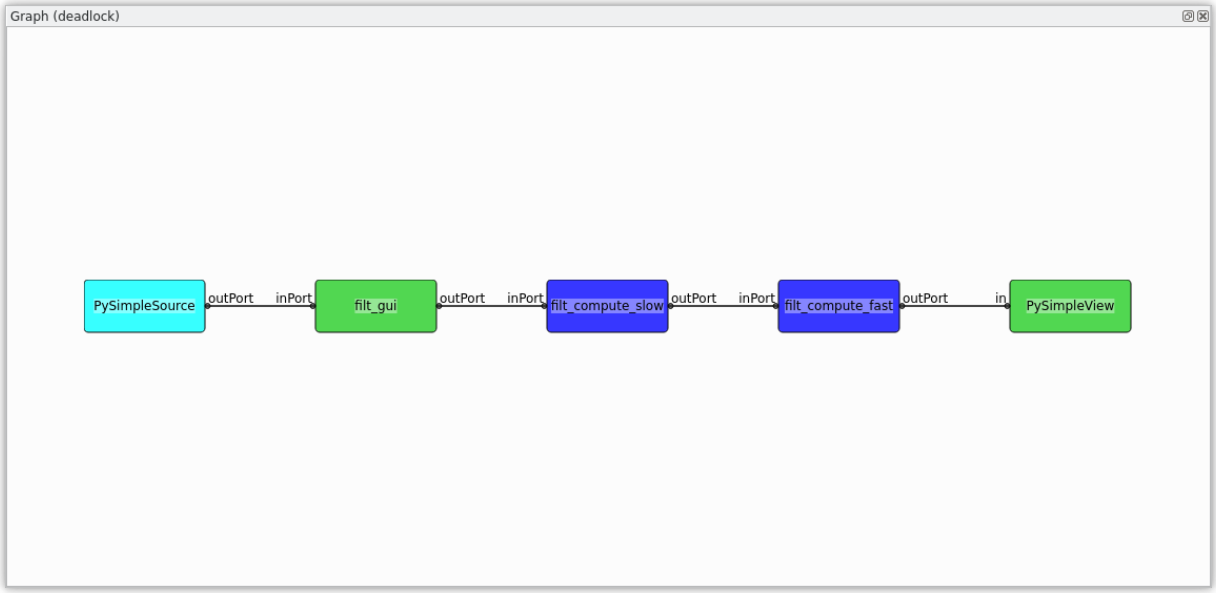
- if a variable is defined in multiple levels (e.g. globally and in a composite filter instance), the definition of the innermost level will be used.
- `FIVE_PLUS_NINE_RES=${!5+9}` will expand to `"14"`.
- `FIVE_PLUS_NINE=five plus nine is $FIVE_PLUS_NINE_RES` will expand to `"five plus nine is 14"`
- `FIVE_PLUS_NINE_FAIL=five plus nine is ${!5+9}` will expand to `"five plus nine is ${!5+9}"`, since the python expression substitution works only for standalone variables.
- `USE_MATH=${!importlib.import_module('math').sqrt(int(subst('$FIVE_PLUS_NINE'))}` will expand to `"3.7416573867739413"`. Note that recursive variable substitution needs to performed explicitely using the `subst(...)` function.

2.3.2 Thread cycles and Deadlocks

When you get error messages like

```
nexxT.core.ActiveApplication: This graph is not deadlock-safe. A cycle has been found in␣
↪the thread graph: main->compute->main
```

you have tried to create an application which is potentially deadlocking. In the message above, it is stated that there is a dependency cycle in the filters of the threads main and compute. This is the corresponding filter graph:

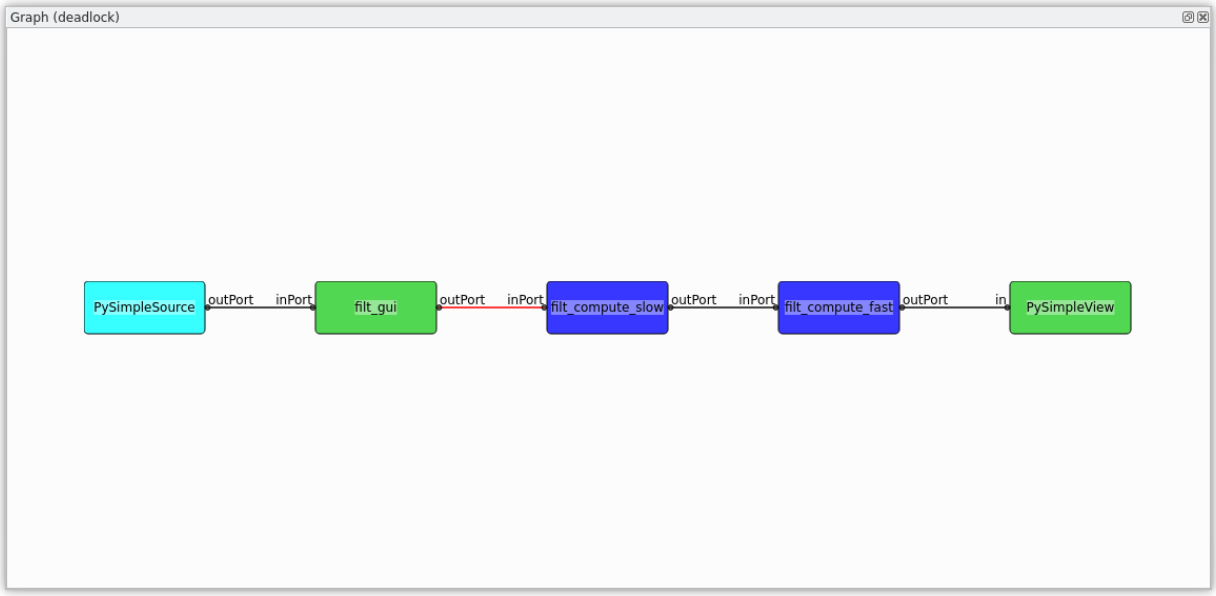


The reason for the possible deadlocks is that nexxT by default uses a semaphore to control the number of pending samples in inter-thread connections. When transmitting a sample to another thread which has not yet processed the last transmitted sample, the transmitting thread blocks until the last transmitted sample has been received by the receiving thread. This behaviour might cause deadlocks in the presence of cycles, nexxT detects these cycles and refuses to execute these applications.

There are multiple solutions for this issue:

- move filters to other threads. The above examples gets deadlock safe when moving the filter *filt_gui* from the gui thread (green) to the compute thread (dark-blue).
- Moving all filters to the main thread is always a solution, but this might be too slow.

- Use non-blocking connections for specific inter-thread connections (right-click on a connection and select *Set non blocking*). Non-blocking connections do not check for pending data samples, so they cannot cause deadlocks. Non blocking connections are displayed in red in the filter graph:



Note that non-blocking connections come with the risk that a potentially infinite amount of data is pending on inter-thread connections. This might cause high latency or even out-of-memory situations. Therefore, non-blocking connections are not recommended to be used at high data rate connections. Output ports triggered by sporadic events are best suited for non-blocking connections.

2.4 Developer Perspectives

One of the first things to decide when using nexxT is the data format to be used in the configuration files. There are a lot of possible choices, each with some pros and cons. A fundamental requirement on the data format is that the serialization and deserialization operations shall be efficient. This is because the data is passed between filters in the serialized format and so these operations are heavily used while running a filter graph. The following (incomplete) table gives some hints on how data may be represented:

Method	Pros	Cons
pickle	• (De-)serialization is easily implemented in python with pickle.dumps and pickle.loads. • Self-contained	• Can be slow depending on the data • Unusable in the C++ world
flatbuffers	• Efficient and cross platform (de-)serialization library • Self-contained	• Depends on 3rd party library
ctypes	• The approach can be used in both domains, python and C++. • In python, it is easy to define C structures using the ctypes library • In C++, regular C structs can be used • (De-)serialization is efficient. Zero-copy reading is supported. • Deserialized data access is straight-forward and the same between python and C++.	• If the structures are subject to change over time, manual version management is required (i.e., old structure definitions have to be kept and a version number must be present) • Synchronization between C++ and python type definitions might be tedious. • Not self-contained (structure definitions need to be kept seperate from the data).

For this tutorial, we decided to use the ctypes approach. Here at ifm we are using a proprietery data format which is also heavily using ctypes in the python world but doesn't show the cons of the pure ctypes approach listed above.

2.4.1 Python

2.4.1.1 Data Format

The data format is defined in the module *nexxT.examples.framework.ImageData*. It consists of the ImageHeader definition and two methods for deserializing and serializing images.

```
class ImageHeader(ct.Structure):
    """
    The QByteArray starts with this header and the rest of the array is the␣
↪actual image data
    according to the format given here.
    """
    _fields_ = [("width", ct.c_uint32),    # the width in pixels
                ("height", ct.c_uint32),   # the height in pixels
                ("lineInc", ct.c_uint32),  # the number of bytes per line␣
↪(including padding for alignment)
                ("format", ct.c_char*32),  # the image format as a c string.␣
↪See above ImageFormats for a list.
               ]
```

The image data format is very simple, first there is a data header with image meta information followed by the raw image data buffer.

The deserialization operation converts the so-defined data into a numpy array, usually without copying the data. In nexxT, QByteArray instances are used to pass data around, so the input argument of the deserialization operation is a QByteArray instance.

```
def byteArrayToNumpy(qByteArray):
    """
    Interpret the input instance as an image and convert that to a numpy array.
↪ If the alignment is ok, then this
    operation is a zero-copy operation, otherwise one copy is made.

    :param qByteArray: a QByteArray instance
    :return: a numpy instance
    """
    # efficient zero-copy cast to a python memoryview instance
    mv = memoryview(qByteArray)
    # interpret the ImageHeader structure from this buffer (zero-copy)
    hdr = ImageHeader.from_buffer(mv)
    # convert the format bytes instance to a string
    fmt = hdr.format.decode()
    # sanity check
    if not fmt in ImageFormats:
        raise RuntimeError(f"Unknown image format {fmt}")
    # get number of channels and the numpy dtype of the target array
    numChannels, dtype = ImageFormats[hdr.format.decode()]
    # calculate the number of bytes per pixel
    bpp = dtype().nbytes*numChannels
    if hdr.lineInc % bpp != 0:
        # there is a non-convertable padding at the end of the lines, so we␣
↪have to fix that
        # first interpret the image data as a numpy uint8 buffer (zero-copy)
        tmp = np.frombuffer(mv, dtype=np.uint8, offset=ct.sizeof(hdr))
        # reshape to 2D with with lineInc as width
        tmp = np.reshape(tmp, (-1, hdr.lineInc))
        # crop the non-aligned padding bytes from the image,
        tmp = tmp[:, :(hdr.lineInc//bpp)*bpp]
        # we have to create a copy here (the frombuffer call does not work on␣
↪memoryview(tmp))
        mv = bytes(tmp)
    # create the target array
    res = np.frombuffer(mv, dtype=dtype, offset=ct.sizeof(hdr))
    # reshape to requested dimenstions
    return np.reshape(res, (-1, max(1,hdr.lineInc//bpp), numChannels))
```

The key for the zero-copy operation is to use memoryview(...) together with python's buffer protocol to access data.

Similar ideas are used in the serialization implementation where only one copy of the data is made before passing it to the framework.

```
def numpyToByteArray(img):
    """
    Convert a numpy image to the corresponding QByteArray (and make a copy).

    :param img: a numpy array instance with 2 or 3 dimensions
    :return: a QByteArray instance
```

(continues on next page)

(continued from previous page)

```
    """
    # make sure that img is a contiguous array
    img = np.ascontiguousarray(img)
    # allocate the result
    res = QByteArray(img.nbytes + ct.sizeof(ImageHeader), 0)
    # create a memory view
    mv = memoryview(res)
    # map the header into this view
    hdr = ImageHeader.from_buffer(mv)
    hdr.width = img.shape[1]
    hdr.height = img.shape[0]
    hdr.lineInc = img[0, ...].nbytes
    # select the format
    if img.dtype is np.dtype(np.uint8):
        hdr.format = b"intensity_u8" if len(img.shape) < 3 else b"rgb_u8"
    if img.dtype is np.dtype(np.uint16):
        hdr.format = b"intensity_u16" if len(img.shape) < 3 else b"rgb_u16"
    if img.dtype is np.dtype(np.uint32):
        hdr.format = b"intensity_u32" if len(img.shape) < 3 else b"rgb_u32"
    if img.dtype is np.dtype(np.float32):
        hdr.format = b"intensity_f32" if len(img.shape) < 3 else b"rgb_f32"
    if img.dtype is np.dtype(np.float64):
        hdr.format = b"intensity_f64" if len(img.shape) < 3 else b"rgb_f64"
    # assert reasonable shape
    assert len(img.shape) == 2 or (len(img.shape) == 3 and img.shape[2] == 3)
    # map the image data into the view
    tmp = np.frombuffer(mv, img.dtype, offset=ct.sizeof(hdr))
    # assign the pixels
    tmp[...] = img.flatten()
    return res
```

2.4.1.2 A first simple filter

Having the data format defined, we can go on and show how to write a simple filter. The *nexxT.examples.framework.ImageBlur.ImageBlur* filter has one static input and one static output port. It applies a blurring operation with an adjustable filter size.

```
class ImageBlur(Filter):
    """
    A filter providing the parameter kernelSize. A box filter of this size is␣
↪applied to the input image
    and the output is transferred over the output port.
    """
    def __init__(self, env):
        super().__init__(False, False, env)
        # add the input port for the filter
        self.inPort = self.addStaticInputPort("video_in")
        # add the output port for the filter
        self.outPort = self.addStaticOutputPort("video_out")
        # define the kernelSize property
        pc = self.propertyCollection()
```

(continues on next page)

(continued from previous page)

```
        pc.defineProperty("kernelSize", 3, "Kernel size of a simple blurring␣
↪kernel", options=dict(min=1, max=99))

    def onPortDataChanged(self, port):
        """
        Overloaded from Filter base class. The method is called whenever new␣
↪data arrives at an input port.
        :param port: the port which has new data.
        :return:
        """
        if port.getData().getDatatype() == "example/image":
            # we don't (want to) have scipy or opencv in the dependency list,␣
↪so we do that by hand, it is just a
            # showcase
            ks = self.propertyCollection().getProperty("kernelSize")
            # assert odd kernel size
            ks = (ks//2)*2 + 1
            if ks > 1: # non-trivial size ?
                # efficient (zero-copy) conversion
                in_img = byteArrayToNumpy(port.getData().getContent())
                # apply the filter
                res = boxFilter(in_img, ks)
                # create a DataSample instance to be transferred over the port
                sample = DataSample(numpyToByteArray(res), "example/image",␣
↪port.getData().getTimestamp())
            else:
                # filter is no-op, we reuse the input data in this case
                sample = port.getData()
            # finally transmit the result
            self.outPort.transmit(sample)
```

The constructor creates the ports and defines the kernelSize property. The onPortDataChanged method is called from nexxT whenever new data arrives at the input port.

Note that nexxT always takes care about using the filter's thread for callbacks like onPortDataChanged(...), unless explictely otherwise stated in the documentation. A manual synchronization for data consistency shall not be necessary in a filter. That was one major drawback of our previous framework and the source of hard-to-find deadlocks and segmentation faults.

NexxT manages the lifecycles of the filters using a state machine. The onPortDataChanged(...) callback is executed after filters have reached their *ACTIVE* state. NexxT also ensures that no data samples are lost during initialization of the application or because some filters are slow.

Slow filters will slow down the whole filter graph. This is a major difference to ROS and ROS2 where data samples are discarded by default in favour of real-time performance, which results in difficulties getting reproducable results when developing potentially slow algorithms with data from disk.

The constructor and the onInit(...) method of a filter are called not only when executing an application but also when the filter graph is manipulated. For that reason, it is strongly recommended, that these methods are fast. Slow operations shall be delayed until onOpen(...) or onStart(...) is called.

2.4.1.3 Display filters

Display filters present visualizations of incoming data. They are normal nexxT filter instances. We will shortly walk through the *nexxT.examples.framework.ImageView.ImageView* filter with a focus on how the filter is written (and not how the display is achieved):

```
    def __init__(self, env):
        super().__init__(False, False, env)
        # create an input port for receiving the image data
        self.inPort = self.addStaticInputPort("video_in", 1, -1)
        # note that the widget shall not be created directly but in the␣
↪onOpen(...) function.
        # reason is that the constructor and onInit(...) functions are called␣
↪quite often and so they should
        # not perform expensive operations.
        self._widget = None
        # define the properties of this filter
        pc = self.propertyCollection()
        pc.defineProperty("caption", "view",
                          "Caption for the MDI window. You can use 2D indices␣
↪for aligning multiple views\n"
                          "in a grid layout.")
        pc.defineProperty("scale", 1.0, "Scale factor for display",␣
↪options=dict(min=0.01, max=16.0))
        # we use the propertyChanged signal to synchronize the scale factor.
        pc.propertyChanged.connect(self.propChanged)
```

The constructor adds the static input port where the images arrive and it defines two properties for the window caption and the scale factor of the view. In this constructor it is demonstrated how to use a slot for getting notified about changed properties.

An important aspect is that the actual widget is not created in the constructor, but the creation is delayed until the onOpen(...) function as already discussed in the previous section *A first simple filter*.

```
    def onOpen(self):
        """
        Now we can create the widget.

        :return:
        """
        pc = self.propertyCollection()
        # get the main window service, used for registering the "subplot" in a␣
↪QMDISubWindow instance
        mw = Services.getService("MainWindow")
        # create the widget
        self._widget = DisplayWidget()
        # register the subplot in the main window
        mw.subplot(pc.getProperty("caption"), self, self._widget)
        # inform the display widget about the current scale
        self._widget.setScale(pc.getProperty("scale"))
```

In the onOpen(...) function, the display widget is created and registered in the main window. Therefore, it queries nexxT's MainWindow service and registers the display with the subplot method of the main window. The caption of a mainwindow can optionally include a "[row, col]" definition to layout multiple views in a grid inside a single MDI window.

```
    def onClose(self):
        """
        Inverse of onOpen

        :return:
        """
        mw = Services.getService("MainWindow")
        # de-register the subplot
        mw.releaseSubplot(self._widget)
        # delete the widget reference
        self._widget = None
```

The onClose(...) method is the inverse of onOpen(...). It releases the widget from the main window and clears the reference to it.

2.4.2 C++

What are the benefits of using C++ instead of python? While you can argue that performance is not affected much if the filter only uses a wrapper around a library such as opencv, the python GIL is a factor which might limit performance in a multithreaded application like nexxT. In a nutshell it means, that whenever python code is executed, the interpreter has the GIL locked to prevent other threads from modifying interpreter states. C extensions like numpy, opencv or PySide6 unlock the GIL during long-duration calls. As a consequence, heavily using pure python will slow down other threads because the GIL limits parallel execution. Using C++ filters, it is possible to design operations which are not affected by the GIL at all.

Filters in C++ are very similar to filters in python. They are defined using a class inheriting from *nexxT::Filter* and overwriting the same methods just like in python. One difference is the usage of nexxT services like the MainWindow service (see *Display filters*). In C++, these services are of type QObject. Therefore, you need to use QMetaObject::invokeMethod for accessing slots of the services.

The plugin library links against the nexxT runtime library (*libnexxT.so* or *nexxT.dll*) which is provided in nexxT installation directory. It also links against a QT library used for development. Note that during runtime, the QT library bundled with PySide6 will be used regardless of which QT library has been used to develop. To be on the safe side, you should use a matching major.minor version, the patch level should be non-relevant. For example, to compile a plugin for the PySide6 version 5.14.2.3, you can use QT 5.14.0. Plugin libraries do not use shiboken2 for exposing the filters in python, instead they use a QLibrary interface.

Note that - unlike pure QT - PySide6 does not provide any compatibility guarantees between minor or patch level releases. This means that it is generally not possible to use nexxT with a different PySide6 version than it was compiled against.

Each plugin library can announce one or more filter classes.

2.4.2.1 Camera Grabber

Again, we will shortly walk through the CameraGrabber class with a focus on how to write a nexxT filter and not how to grab images from a camera. We use the QMultimedia framework for that.

Listing 1: ../../nexxT/tests/src/CameraGrabber.hpp

```
/*
 * SPDX-License-Identifier: Apache-2.0
 * Copyright (C) 2020 ifm electronic gmbh
```

(continues on next page)

(continued from previous page)

```
 *
 * THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.
 */

#ifndef CAMERA_GRABBER_HPP
#define CAMERA_GRABBER_HPP

#include <QtCore/QObject>
#include <QtMultimedia/QCamera>
#include <QtMultimedia/QMediaCaptureSession>
#include "VideoGrabber.hpp"
#include "nexxT/Filters.hpp"
#include "nexxT/Ports.hpp"
#include "nexxT/NexxTPlugins.hpp"
#include "ImageFormat.h"

using namespace nexxT;

/* This class implements a Grabber for USB camera images using the
 * QtMultimedia framework. Note that this is not really the ideal
 * choice because of some known bugs, but it helps to keep dependencies
 * small. Note that C++ has been chosen here because of two reasons.
 * First to show how to write a nexxT filter in C++, and second,
 * because the PySide2 bindings of QCamera and QAbstractVideoSurface
 * do not allow to do this in python.
 */
class CameraGrabber : public Filter
{
    Q_OBJECT

    /* the port where the data will be transmitted to */
    SharedOutputPortPtr video_out;
    /* the camera instance (created at onOpen(...)) */
    QCamera *camera;
    /* the video surface needed to actually grab QImages from the QCamera */
    VideoGrabber *videoSurface;
    /* the capture session */
    QMediaCaptureSession *session;

public:
    /* The following line is needed to declare a nexxT plugin. Note that also␣
↪the following
     * is needed in a cpp file:
     *
     * NEXXT_PLUGIN_DEFINE_START()
     * NEXXT_PLUGIN_ADD_FILTER(CameraGrabber)
     * // more filters can be added
     * NEXXT_PLUGIN_DEFINE_FINISH()
     *
     */
    NEXXT_PLUGIN_DECLARE_FILTER(CameraGrabber)
```

(continues on next page)

(continued from previous page)

```
    /* Constructor with standard arguments */
    CameraGrabber(BaseFilterEnvironment *env);
    /* destructor */
    virtual ~CameraGrabber();

public slots:
    /* Slot called from the videoSurface instance when a new image arrives */
    void newImage(const QImage &img);
    /* Slot called by the QtMultimedia framework to detect errors (they will␣
↪happen!) */
    void onErrorOccurred(QCamera::Error error, const QString &errString);

protected: // folowing functions are overloaded from the nexxT::Filter API
    void onOpen();
    void onStart();
    void onStop();
    void onClose();
};

#endif
```

The CameraGrabber filter is defined in this header file. Because it uses signals and slots we need the Q_OBJECT macro in the class. We also need to call the macro *NEXXT_PLUGIN_DECLARE_FILTER* for being able to announce this filter in the plugin library.

```
CameraGrabber::CameraGrabber(BaseFilterEnvironment *env)
    : Filter(false, false, env)
    , camera()
    , videoSurface()
    , session()
{
    /* similar to the python API, we create an output port for transmitting␣
↪images */
    video_out = SharedOutputPortPtr(new OutputPortInterface(false, "video_out",
↪ env));
    /* and register that port */
    addStaticPort(video_out);
    /* note that we do not connect to the hardware in the constructor, this is␣
↪to be
     * done later in onOpen(...) for efficiency reasons.
     */
    PropertyCollection *pc = propertyCollection();
    QStringList devices;
    for (const QCameraDevice &cameraDevice : QMediaDevices::videoInputs())
    {
        devices.push_back(cameraDevice.description());
    }
    if(devices.size() == 0)
    {
        devices.push_back("dummy");
    }
    pc->defineProperty("device", devices[0], "the camera device", {{"enum",␣
```

(continues on next page)

(continued from previous page)

```
↪devices}});
}

CameraGrabber::~CameraGrabber()
{
}
```

The constructors and destructors are straight forward. Long-running operations like camera discovery are delayed until onOpen(...). The constructor passes the BaseFilterEnvironment instance through to the base class just like the python filters do.

```
/* here we connect to the hardware */
void CameraGrabber::onOpen()
{
    if(videoSurface)
    {
        NEXXT_LOG_WARN("videoSurface still allocated in onOpen");
        delete videoSurface;
        videoSurface = 0;
    }
    if(camera)
    {
        NEXXT_LOG_WARN("camera still allocated in onOpen");
        delete camera;
        camera = 0;
    }
    if(session)
    {
        NEXXT_LOG_WARN("session still allocated in onOpen");
        delete session;
        session = 0;
    }
    /* create a QCamera and a VideoGrabber instance, the camera will run in a␣
↪default mode
     * All these objects run in the same thread as this filter.
     */
    PropertyCollection *pc = propertyCollection();
    QString devname = pc->getProperty("device").value<QString>();
    for (const QCameraDevice &cameraDevice : QMediaDevices::videoInputs())
    {
        if( cameraDevice.description() == devname )
        {
            camera = new QCamera(cameraDevice, this);
            break;
        }
    }
    session = new QMediaCaptureSession(this);
    if(!camera)
    {
        NEXXT_LOG_WARN("Using default camera.");
        camera = new QCamera(this);
    }
```

(continues on next page)

(continued from previous page)

```
    videoSurface = new VideoGrabber(this);
    session->setCamera(camera);
    session->setVideoOutput(videoSurface);
    /* make up signal/slot connections */
    QObject::connect(videoSurface, SIGNAL(newImage(const QImage &)), this,␣
↪SLOT(newImage(const QImage &)));
    QObject::connect(camera, SIGNAL(errorOccurred(QCamera::Error, const␣
↪QString &)),
                     this, SLOT(onErrorOccurred(QCamera::Error, const QString &
↪)));
}

/* at that point, the streaming shall be started */
void CameraGrabber::onStart()
{
    camera->start();
}

/* inverse of onStart(...) */
void CameraGrabber::onStop()
{
    camera->stop();
}

/* inverse of onOpen(...) */
void CameraGrabber::onClose()
{
    if(videoSurface)
    {
        delete videoSurface;
        videoSurface = nullptr;
    }
    if(camera)
    {
        delete camera;
        camera = nullptr;
    }
    if(session)
    {
        delete session;
        session = nullptr;
    }
}
```

The onOpen(...) method create the necessary classes from the QMultimedia framework and sets up signal and slot connections for receiving new images and error handling. Note that this ensures that the slots are called in the filter's thread. onStart(...) starts the capturing, onStop(...) and onClose(...) are the inverse functions. Logging can be performed by using the macros defined in *Logging*.

```
/* A new image has arrived, we convert this here to a QByteArray
 * Note that QT takes care to call this method in the correct thread
 * due to the QueuedConnection mechanism.
```

(continues on next page)

(continued from previous page)

```
 */
void CameraGrabber::newImage(const QImage &_img)
{
    QImage img = _img; /* we need a writable img */
    ImageHeader hdr;
    QByteArray data;
    /* reserve size for the image in the QByteArray for efficiency reasons */
    data.reserve(int(sizeof(ImageHeader)) + (img.height() * img.
↪bytesPerLine()));
    /* determine the format */
    QString format;
    switch(img.format())
    {
    case QImage::Format_RGB888:
        format = "rgb_u8";
        break;
    case QImage::Format_Grayscale8:
        format = "intensity_u8";
        break;
    case QImage::Format_Grayscale16:
        format = "intensity_u16";
        break;
    default:
        /* for all other formats, we create an rgb_u8 image using QImage␣
↪functionality */
        img = img.convertToFormat(QImage::Format_RGB888);
        format = "rgb_u8";
    }
    /* fill the header fields */
    hdr.width = uint32_t(img.width());
    hdr.height = uint32_t(img.height());
    hdr.lineInc = uint32_t(img.bytesPerLine());
    std::strncpy(hdr.format, format.toLocal8Bit().constData(), sizeof(hdr.
↪format)-1);
    /* fill the QByteArray instance */
    data = data.append((const char *)&hdr, sizeof(hdr));
    data = data.append((const char *)img.constBits(), hdr.lineInc*hdr.height);
    /* transmit over the port */
    video_out->transmit(
         SharedDataSamplePtr(new DataSample(data, "example/image",␣
↪DataSample::currentTime()))
    );
}
```

The newImage(...) slot is called when a new image arrives through the QMultimedia framework. The data is serialized according to *Data Format* and transmitted over the output pin.

2.4.2.2 Plugin Definition

For being able to announce the C++ filters, the plugin needs to be defined. This is performed here for the CameraGrabber class and other filters defined in the same library.

```
/*
 * SPDX-License-Identifier: Apache-2.0
 * Copyright (C) 2020 ifm electronic gmbh
 *
 * THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.
 */
#include <QtGlobal>
#include "AviFilePlayback.hpp"
#include "CameraGrabber.hpp"
#include "SimpleSource.hpp"
#include "TestExceptionFilter.hpp"
#include "Properties.hpp"

NEXXT_PLUGIN_DEFINE_START()
NEXXT_PLUGIN_ADD_FILTER(VideoPlaybackDevice)
NEXXT_PLUGIN_ADD_FILTER(CameraGrabber)
NEXXT_PLUGIN_ADD_FILTER(SimpleSource)
NEXXT_PLUGIN_ADD_FILTER(TestExceptionFilter)
NEXXT_PLUGIN_ADD_FILTER(PropertyReceiver)
NEXXT_PLUGIN_DEFINE_FINISH()
```

2.4.3 Debugging

Debugging can be achieved with any IDE of your choice which supports starting python modules (a.k.a. python -m). The nexxT-gui start script can be replaced by python -m nexxT.core.AppConsole. See specific examples below.

2.4.3.1 Python debugging with Visual Studio Code

To start with VS Code make sure the Python extension for VS Code is installed (see here).

Open VS Code in your source code directory via menu (“File/Open Folder”) or cd in your terminal of choice to your folder and start VS Code by typing `code .` (dot for the current directory).

Setting virtual environment

If you’re not using venv, continue to the next section. In case you are using a virtual environment, we need to provide VS Code some information. Open the settings.json file in your .vscode directory (or create it). Your settings should include following information:

```
{
    "python.pythonPath": "/path/to/your/python/interpreter/python.exe",
    "python.venvPath": "/path/to/your/venv",
    "python.terminal.activateEnvironment": true
}
```

The paths should be absolute. If `"python.pythonPath"` is inside your venv, `"python.venvPath"` is not required, VS Code will recognize it and activate venv automatically.

Note: Make sure that at least one .py file is opened in the editor when starting debugging, otherwise the venv may not be activated (the author does not know exactly under which circumstances this is required, but this information may safe you some time searching the internet when things don't go as expected).

With these settings at hand, venv will also be started automatically when we create a new terminal in VS Code ("Terminal/New Terminal").

Configuring launch file

The next step is to create the launch.json file for our debug session (manually or via "Run/Add configuration"). Your launch.json file in .vscode folder should look like this:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Modul",
            "type": "python",
            "request": "launch",
            "module": "nexxT.core.AppConsole",
            "justMyCode": false
        }
    ]
}
```

The "module" setting is the critical part. Under the hood VS code will invoke `python -m <module>`. With the "justMyCode" setting, we can extend debugging to external code loaded by our application.

We're all set, now we can run our debug session by pressing F5 or the "Run" menu.

CHAPTER

THREE

REFERENCE DOCUMENTATION

3.1 nexxT package

3.1.1 Subpackages

3.1.1.1 nexxT.interface package

This __init__.py generates shortcuts for exported classes

class nexxT.interface.**DataSample**(*content*, *datatype*, *timestamp*)

Bases: object

Note: Import this class with `from nexxT.interface import DataSample`.

This class is used for storing a data sample of the nexxT framework. For most generic usage, a QByteArray is used for the storage. This means that deserialization has to be performed on every usage and data generators need to serialize the data. Assumes that serializing / deserializing are efficient operations.

DataSample instances additionally have a type, which is a string and it should uniquely define the serialization method. Last but not least, an integer timestamp is stored for all DataSample instances.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::DataSample*

TIMESTAMP_RES = 1e-06

the resolution of the timestamps

__init__(*content*, *datatype*, *timestamp*)

Create a new data sample instance.

Parameters

- **content** – A QByteArray instance containing the (serialized) content
- **datatype** – A string instance which uniquely defines the serialized content
- **timestamp** – An integer representing the sample's time stamp [µs]

static copy(*src*)

Create a copy of this DataSample instance

Parameters
src – the instance to be copied

Returns
the cloned data sample

`static currentTime()`

Returns the current system time suitable for data sample timestamps. Note: The python implementation uses time.time_ns, which unfortunately has limited accuracy under windows (16 ms).

Returns
an integer instance

`getContent()`

Get the contents of this sample as a QByteArray. Note that this is an efficient operation due to the copy on write semantics of QByteArray. It also asserts that the original contents cannot be modified.

In C++, make sure to keep an instance of the QByteArray until done with processing. Moreover, consider to use *QByteArray::constData()* for a pointer-to-memory access rather than *QByteArray::data()*, since the latter will eventually make an unnecessary deep copy of the encapsulated data.

Returns
QByteArray instance copy

`getDatatype()`

Return the data type.

Returns
data type string

`getTimestamp()`

Return the timestamp associated to the data.

Returns
integer timestamp

`class nexxT.interface.Filter(`*dynInPortsSupported*, *dynOutPortsSupported*, *environment*`)`

Bases: `QObject`

Note: Import this class with `from nexxT.interface import Filter`.

This class is the base class for defining a nexxT filter. A minimal nexxT filter class looks like this:

```
class SimpleStaticFilter(Filter):

    def __init__(self, environment):
        super().__init__(False, False, environment)
        pc = self.propertyCollection()
        self.sample_property = pc.getProperty("sample_property", 0.1, "a property␣
↪for demonstration purpose")
        self.inPort = self.addStaticInputPort("inPort")
        self.outPort = self.addStaticOutputPort("outPort")

    def onPortDataChanged(self, inputPort):
        dataSample = inputPort.getData()
        newSample = DataSample.copy(dataSample)
        self.outPort.transmit(dataSample)
```

The constructor of classes derived from nexxT.interface.Filter must have a single argument **environment** which is passed through to this base class. It configures dynamic port usage with the two boolean flags. In the constructor, the filter can define and query properties and create static ports.

The onPortDataChanged is called whenever new data arrives on an input port. In the example above, a copy of the original data sample is returned.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::Filter*

`__init__`(*dynInPortsSupported*, *dynOutPortsSupported*, *environment*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

`addStaticInputPort`(*name*, *queueSizeSamples=1*, *queueSizeSeconds=None*)

Shortcut for generating a static input port and adding it to the filter. See also *nexxT.interface.Ports.InputPort()*

Parameters

- **name** – The name of the input port
- **queueSizeSamples** – The size of the input queue in samples
- **queueSizeSeconds** – The size of the input queue in seconds

Returns
the new port instance

`addStaticOutputPort`(*name*)

Shortcut for generating a static output port and adding it to the filter. See also *nexxT.interface.Ports.OutputPort()*

Parameters
name – The name of the output port

Returns
the new port instance

`addStaticPort`(*port*)

Register a static port for this filter. Only possible in CONSTRUCTING state.

Parameters
port – InputPort or OutputPort instance

Returns
None

`environment`()

Returns the environment associated with this filter.

Returns
a FilterEnvironment instance

getDynamicInputPorts()

Get dynamic input ports of this filter. Only possible in and after INITIALIZING state.

Returns
list of dynamic input ports

getDynamicOutputPorts()

Get dynamic output ports of this filter. Only possible in and after INITIALIZING state.

Returns
list of dynamic output ports

guiState()

Return the gui state associated with this filter. Note: the gui state shall not be used for properties which are important for data transport, for these cases the propertyCollection() shall be used. Typical gui state variables are: the geometry of a user-managed window, the last directory path used in a file dialog, etc. The gui state might not be initialized during mockup-phase.

Returns
PropertyCollection instance

onClose()

This function can be overwritten for general de-initialization tasks (e.g. release resources needed to run the filter, close files, etc.). It is the opoosite to onOpen(...).

Returns
None

onDeinit()

This function can be overwritten for performing de-initialization tasks related to dynamic ports. It is the opposite to onInit(...)

Returns
None

onInit()

This function can be overwritten for performing initialization tasks related to dynamic ports.

Returns
None

onOpen()

This function can be overwritten for general initialization tasks (e.g. acquire resources needed to run the filter, open files, connecting to services etc.).

Returns

onPortDataChanged(*inputPort*)

This function can be overwritten to be notified when new data samples arrive at input ports. For each data sample arrived this function will be called exactly once.

Parameters
inputPort – the port where the data arrived

Returns
None

onStart()

This function can be overwritten to reset internal filter state. It is called before loading a new sequence.

Returns
None

onStop()

Opposite of onStart.

Returns
None

onSuggestDynamicPorts()

Shall return the suggested dynamic ports of this filter. Prominent example is to return the streams contained in a HDF5 file. Note that it is safe to assume that the instance lives in the GUI thread, when this function is called from the nexxT framework.

Returns
listOfInputPortNames, listOfOutputPortNames

propertyCollection()

Return the property collection associated with this filter.

Returns
PropertyCollection instance

removeStaticPort(*port*)

Remove a static port of this filter. Only possible in CONSTRUCTING state.

Parameters
port – InputPort or OutputPort instance

Returns
None

staticMetaObject = PySide6.QtCore.QMetaObject("Filter" inherits "QObject":)

class nexxT.interface.**FilterState**

Bases: object

Note: Import this class with `from nexxT.interface import FilterState`.

This class defines an enum for the filter states. For reference, the filter's lifecycle state diagram is shown here:

static state2str(*state*)

converts a state integer to the corresponding string

Parameters
state – the state integer

Returns
string

class nexxT.interface.**FilterSurrogate**(*dllUrls*, *name*)

Bases: object

Note: Import this class with `from nexxT.interface import FilterSurrogate`.

This class acts as a surrogate to reference a filter from a DLL/shared object plugin from within python. It's main purpose is the ability to announce these filters using python entry points. For this, create an instance of this class in one of your modules and refer to it by a 'nexxT.filters' entry point. Example:

```
from distutils.core import setup
setup( # ...
    'nexxT.filters' : [
         'examples.framework.CameraGrabber = nexxT.examples:CameraGrabber',
    ])
```

The surrogate creation of CameraGrabber is performed in *nexxT.examples*:

```
# SPDX-License-Identifier: Apache-2.0
# Copyright (C) 2020 ifm electronic gmbh
#
# THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.
#

"""
define FilterSurrogates for binary filters.
"""

from pathlib import Path
import os
from nexxT.interface import FilterSurrogate
if os.environ.get("READTHEDOCS", None) is None:
    from nexxT.Qt import QtMultimedia # needed to load corresponding DLL before␣
↪loading the nexxT plugin

AviReader = FilterSurrogate(
    "binary://" + str((Path(__file__).parent.parent / "tests" /
                       "binary" / "${NEXXT_PLATFORM}" / "${NEXXT_VARIANT}" / "test_
↪plugins").absolute()),
    "VideoPlaybackDevice"
)
"""
Filter surrogate for the VideoPlaybackDevice class which is defined in a packaged␣
↪shared object "test_plugins".
"""

CameraGrabber = FilterSurrogate(
    "binary://" + str((Path(__file__).parent.parent / "tests" /
                       "binary" / "${NEXXT_PLATFORM}" / "${NEXXT_VARIANT}" / "test_
↪plugins").absolute()),
    "CameraGrabber"
)
"""
Filter surrogate for the CameraGrabber class which is defined in a packaged shared␣
↪object "test_plugins".
"""
```

__init__(*dllUrls*, *name*)

Create a FilterSurrogate instance.

Parameters

- **dllUrls** – might be (1) a dictionary mapping variant names to URLs, variant names are usually "nonopt" and "release" or (2) a url string which will be mapped to the release variant Note that URLs for binary libraries (DLL's or shared objects) are of the form "binary://<absolute-path-to-dll>". The absolute path might contain variables like ${NEXXT_PLATFORM} or ${NEXXT_VARIANT}.
- **name** – the name of the filter class or factory function

dllUrl(*variant*)

returns the absolute path of the optimzed dll/shared object

Parameters
variant – the variant for which the filter is needed, given as a string.

name()

returns the name of the filter class

nexxT.interface.**InputPort**

alias of InputPortImpl

class nexxT.interface.**InputPortInterface**(*dynamic*, *name*, *environment*)

Bases: *Port*

This abstract class defines the interface of an input port of a filter. In addition to the normal port attributes, there are two new attributes related to automatic buffering of input data samples. queueSizeSamples sets the maximum number of samples buffered (it can be None, if queueSizeSeconds is not None) queueSizeSeconds sets the maximum time of samples buffered (it can be None, if queueSizeSamples is not None) If both attributes are set, they are and-combined.

..note::
Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::InputPortInterface*

clone(*newEnvironment*)

Return a copy of this port attached to a new environment.

Parameters
newEnvironment – the new FilterEnvironment instance

Returns
a new Port instance

getData(*delaySamples=0*, *delaySeconds=None*)

Return a data sample stored in the queue (called by the filter).

Parameters

- **delaySamples** – 0 related the most actual sample, numbers > 0 relates to historic samples (None can be given if delaySeconds is not None)
- **delaySeconds** – if not None, a delay of 0.0 is related to the current sample, positive numbers are related to historic samples (TODO specify the exact semantics of delaySeconds)

Returns
DataSample instance

interthreadDynamicQueue()

Return the interthread dynamic queue setting.

Returns
a boolean

`queueSizeSamples`()

return the current queueSize in samples

Returns
an integer

`queueSizeSeconds`()

return the current queueSize in seconds

Returns
an integer

`receiveAsync`(*dataSample*, *semaphore*)

Called from framework only and implements the asynchronous receive mechanism using a semaphore.

Parameters

- `dataSample` – the transmitted DataSample instance
- `semaphore` – a QSemaphore instance

Returns
None

`receiveSync`(*dataSample*)

Called from framework only and implements the synchronous receive mechanism. TODO implement

Parameters
`dataSample` – the transmitted DataSample instance

Returns
None

`setInterthreadDynamicQueue`(*enabled*)

If enabled is True, inter thread connections to this input port are dynamically queued for non-blocking behaviour.

This setting does not affect connections from within the same thread. This method can be called only during constructor or the onInit() method of a filter. The main use case is a recording filter where the QT signal/slot is allowed to buffer as many samples as allowed in the input port's queue to prevent unwanted blocking behaviour.

Enabling this might cause a larger delay and might also consume a lot of memory.

Parameters
`enabled` – whether the dynamic queuing feature is enabled or not.

Returns

`setQueueSize`(*queueSizeSamples*, *queueSizeSeconds*)

Set the queue size of this port.

Parameters

- `queueSizeSamples` – 0 related the most actual sample, numbers > 0 relates to historic samples (None can be given if delaySeconds is not None)
- `queueSizeSeconds` – if not None, a delay of 0.0 is related to the current sample, positive numbers are related to historic samples

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("InputPortInterface" inherits "Port":
)

nexxT.interface.**OutputPort**

alias of OutputPortImpl

class nexxT.interface.**OutputPortInterface**(*dynamic*, *name*, *environment*)

Bases: *Port*

This abstract base class defines the interface of an output port of a filter.

..note::

Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::OutputPortInterface*

clone(*newEnvironment*)

Return a copy of this port attached to a new environment.

Parameters

newEnvironment – the new FilterEnvironment instance

Returns

a new Port instance

setupDirectConnection(*inputPort*)

Setup a direct (intra-thread) connection between outputPort and inputPort Note: both instances must live in same thread!

Parameters

- **outputPort** – the output port instance to be connected
- **inputPort** – the input port instance to be connected

Returns

None

setupInterThreadConnection(*inputPort*, *outputPortThread*, *width*)

Setup an inter thread connection between outputPort and inputPort

Parameters

- **outputPort** – the output port instance to be connected
- **inputPort** – the input port instance to be connected
- **outputPortThread** – the QThread instance of the outputPort instance
- **width** – the width of the connection in DataSamples (0: infinite)

Returns

an InterThreadConnection instance which manages the connection (has to survive until connections is deleted)

staticMetaObject = PySide6.QtCore.QMetaObject("OutputPortInterface" inherits "Port":
Methods: #5 type=Signal, signature=transmitSample(PyObject), parameters=PyObject)

transmit(*dataSample*)

transmit a data sample over this port

Parameters

dataSample – sample to transmit

transmitSample

class nexxT.interface.**Port**(*dynamic*, *name*, *environment*)

Bases: QObject

This class is the base class for ports. It is used mainly as structure, containing the 3 properties dynamic (boolean), name (str) and environment (a FilterEnvironment instance).

..note::

Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::Port*

INPUT_PORT = 0

OUTPUT_PORT = 1

__init__(*dynamic*, *name*, *environment*)

Constructor.

Parameters

- **dynamic** – boolean whether this is a dynamic port or not
- **name** – the port name, given as a string
- **environment** – the corresponding FilterEnvironment instance

clone(*newEnvironment*)

This function must be overwritten in inherited classes to create a clone of this port attached to a different environment.

Parameters
newEnvironment – the new FilterEnvironment instance

Returns
a new Port instance

dynamic()

Returns whether this is a dynamic port

Returns
a boolean

environment()

Returns the environment instance managing this port

Returns
FilterEnvironment instance

isInput()

Returns true if this is an input port

Returns
bool

isOutput()

Returns true if this is an output port

Returns
bool

name()

Returns the port name

Returns
a string

setName(*name*)

Sets the port name

Parameters
name – the port name given as a string

Returns
None

staticMetaObject = PySide6.QtCore.QMetaObject("Port" inherits "QObject":)

class nexxT.interface.**PropertyCollection**

Bases: QObject

Note: Import this class with `from nexxT.interface import PropertyCollection`.

This class represents a collection of properties. These collections are organized in a tree, such that there are parent/child relations. This is a generic base class, which is implemented in the core package. Access to properties throug the methods below is thread safe.

Properties are usually used in subclasses of *nexxT.interface.Filters.Filter*. They are presented in the GUI as editable entities and the settings are saved to the configuration file. They are defined during filter constructor and onInit(...), they can be used during the whole filter lifecycle. There is also the possibility to connect a callback function to the *propertyChanged* signal. Note that properties are automatically deleted from the config file if they disappear (e.g., due to code changes, changed dynamic ports, etc.).

Example:

```
class MyFilter(Filter):
    def __init__(self, env):
        super().__init__(False, False, env)
        pc = self.propertyCollection()
        pc.defineProperty("intProp", 1, "an unconstrained integer")
        pc.defineProperty("intPropMax", 1, "an max constrained integer",␣
↪options=dict(max=10))
        pc.defineProperty("intPropBounded", 1, "a bounded integer",␣
↪options=dict(min=-4, max=10))
        pc.defineProperty("floatProp", 1.0, "a floating point number",␣
↪options=dict(min=-1e-4, max=10000))
        pc.defineProperty("boolProp", True, "a boolean")
        # it is also possible to connect a callback
        pc.propertyChanged.connect(self.onPropertyChanged)

    def onInit(self):
        pc = self.propertyCollection()
        pc.defineProperty("strProp", "Hello World", "a string")
        pc.defineProperty("enumProp", "Hello", "a string", options=dict(enum=["Hello
↪", "World"]))
```

(continues on next page)

(continued from previous page)

```
    def onStart(self):
        # queries the current value of the property
        pc = self.propertyCollection()
        intProp = pc.getProperty("intProp")
        # ...

    def onPropertyChanged(self, pc, name):
        logger.info("Property '%s' changed to %s", name, repr(pc.getProperty(name)))
```

In the above example, different editors will be created apropriate to the chosen values. For example, the enum property can be edited using a combo box while integer properties are edited with spin boxes. It is also possible to adapt this behaviour by passing custom propertyHandlers.

This class is an abstract base class.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::PropertyCollection*

---

defineProperty(*name*, *defaultVal*, *helpstr*, *options=None*, *propertyHandler=None*)

Return the value of the given property, creating a new property if it doesn't exist. If it does exist, the definition must be consistent, otherwise an error is raised.

Note that the parameters options and propertyHandler must not be present at the same time. That is because the options are already passed to the constructor of the propertyHandler.

Parameters

- **name** – the name of the property
- **defaultVal** – the default value of the property. Note that this value will be used to determine the property's type. Currently supported types are string, int and float
- **helpstr** – a help string for the user (presented as a tool tip)
- **options** – a dict mapping string to qvariant (common options: 'min', 'max', 'enum') all properties support the option 'ignoreInconsistentOptions' (default: False). If this option is True, then nexxT allows that the options change over time. Even if present, the option type and its default values are not allowed to change.
- **propertyHandler** – a PropertyHandler instance, or None for automatic choice according to defaultVal

Returns

the current value of this property

evalpath(*path*)

Evaluates the string path. If it is an absolute path it is unchanged, otherwise it is converted to an absolute path relative to the config file path.

Parameters

path – a string

Returns

absolute path as string

getProperty(*name*)

return the property identified by name

Parameters
name – a string

Returns
the current property value

propertyChanged

QT signal which is emitted after a property value of the collection has been changed by the user.

Parameters

- **pc** – the PropertyCollection instance (i.e., the same as self.sender())
- **propName** – the name of the property which has been changed.

setProperty(*name*, *value*)

Set the value of a named property.

Parameters

- **name** – property name
- **value** – the value to be set

Returns
None

staticMetaObject = PySide6.QtCore.QMetaObject("PropertyCollection" inherits "QObject": Methods: #5 type=Signal, signature=propertyChanged(PyObject,QString), parameters=PyObject, QString #6 type=Slot, signature=setProperty(QString,PyObject), parameters=QString, PyObject)

class nexxT.interface.**PropertyHandler**

Bases: object

Note: Import this class with `from nexxT.interface import PropertyHandler`.

This class represents a property definition for a specific type. The type handles loading/saving from and to .json configs as well as providing editor widgets for modifying the property in a model/view framework.

It is an abstrct base class.

For illustration, the implementation of the IntHandler is given here as an example:

```
class IntHandler(PropertyHandler):
    """
    The property handler for integer properties; Supported options: min and max.
    """

    def __init__(self, options):
        """
        Constructor

        :param options: the options given to the defineProperty(...) function.
        """
        for k in options:
```

(continues on next page)

(continued from previous page)

```
            if k in ["min", "max"]:
                if not isinstance(options[k], int):
                    raise PropertyParsingError(f"Unexpected type of option {k};␣
↪expected int.")
            else:
                raise PropertyParsingError(f"Unexpected option {k}; expected 'min'␣
↪or 'max'.")
        self._options = options

    def options(self):
        """
        return this handler's options

        :return: a python dict with the actual options.
        """
        return self._options

    def fromConfig(self, value):
        """
        import value from config file and return the adapted version.

        :param value: an integer is expected
        :return: the validated integer
        """
        assert isinstance(value, (float, int, bool))
        return self.validate(value)

    def toConfig(self, value):
        """
        export value to config file and return the adapted version

        :param value: an integer is expected
        :return: the exported value
        """
        assert isinstance(value, int)
        return value

    def toViewValue(self, value):
        """
        create a view of this option value.

        :param value: the current option value
        :return: a string
        """
        assert isinstance(value, int)
        return str(value)

    def validate(self, value):
        """
        Validate an option value and return an adapted, valid value

        :param value: the value to be tested (an integer)
```

(continues on next page)

(continued from previous page)

```
        :return: the adapted, valid value
        """
        if isinstance(value, str):
            try:
                value = int(value)
            except ValueError:
                logger.warning("Cannot interpret value '%s' as int. Using 0.",␣
↪value)
                value = 0
        if "min" in self._options:
            if value < self._options["min"]:
                logger.warning("Adapted option value %d to minimum value %d.",␣
↪value, self._options["min"])
                return self._options["min"]
        if "max" in self._options:
            if value > self._options["max"]:
                logger.warning("Adapted option value %d to maximum value %d.",␣
↪value, self._options["max"])
                return self._options["max"]
        return int(value)

    def createEditor(self, parent):
        """
        Creates a QSpinBox instance for GUI editing of integer values

        :param parent: the parent of the widget
        :return: a QSpinBox instance
        """
        res = QSpinBox(parent)
        res.setFrame(False)
        if "min" in self._options:
            res.setMinimum(self._options["min"])
        else:
            res.setMinimum(-2147483648)
        if "max" in self._options:
            res.setMaximum(self._options["max"])
        else:
            res.setMaximum(2147483647)
        return res

    def setEditorData(self, editor, value):
        """
        set the value of the QSpinBox

        :param editor: the instance returned by createEditor
        :param value: the option value (an integer)
        :return: None
        """
        editor.setValue(value)

    def getEditorData(self, editor):
        """
```

(continues on next page)

(continued from previous page)

```
        return the currently edited value

        :param editor: the instance returned by createEditor
        :return: the integer value
        """
        return self.validate(editor.value())
```

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::PropertyHandler*

`createEditor`(*parent*)

This is called in QStyledItemDelegate::createEditor; creates an editor widget instance.

Parameters
`parent` – a QWidget instance

Returns
a QWidget instance

`fromConfig`(*value*)

Converts the value read from the json file into the native python format

Parameters
`value` – a QVariant instance

Returns
the native value (also a QVariant)

`getEditorData`(*editor*)

This is called in QStyledItemDelegate::setModelData; converts the value from the editor back to native python value.

Parameters
`editor` – the editor widget

Returns
a QVariant, the new native property value

`options`()

Returns the options set for this handler as a QVariantMap

Returns
a QVariantMap instance

`setEditorData`(*editor*, *value*)

This is called in QStyledItemDelegate::setEditorData; populates the editor widget with the actual data.

Parameters

- `editor` – the editor widget as rezurned by createEditor
- `value` – the current property value, given as native python value

Returns
None

toConfig(*value*)

Converts the native python format into a value suitable for json files.

Parameters
value – a QVariant instance (the native value)

Returns
the json value (also a QVariant)

toViewValue(*value*)

Converts the native python format into a value suitable for display in the Qt model/view framework.

Parameters
value – a QVariant instance (the native value)

Returns
a QVariant value (suitable value for display)

validate(*value*)

Returns a validated version of value.

Parameters
value – the value to be set

Returns
a validated version of value

Submodules

nexxT.interface.DataSamples module

This module defines the nexxT interface class DataSample.

class nexxT.interface.DataSamples.**DataSample**(*content*, *datatype*, *timestamp*)

Bases: object

Note: Import this class with `from nexxT.interface import DataSample`.

This class is used for storing a data sample of the nexxT framework. For most generic usage, a QByteArray is used for the storage. This means that deserialization has to be performed on every usage and data generators need to serialize the data. Assumes that serializing / deserializing are efficient operations.

DataSample instances additionally have a type, which is a string and it should uniquely define the serialization method. Last but not least, an integer timestamp is stored for all DataSample instances.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::DataSample*

TIMESTAMP_RES = 1e-06

the resolution of the timestamps

__init__(*content*, *datatype*, *timestamp*)

Create a new data sample instance.

Parameters

- **content** – A QByteArray instance containing the (serialized) content
- **datatype** – A string instance which uniquely defines the serialized content
- **timestamp** – An integer representing the sample's time stamp [µs]

`static copy(src)`

Create a copy of this DataSample instance

Parameters
src – the instance to be copied

Returns
the cloned data sample

`static currentTime()`

Returns the current system time suitable for data sample timestamps. Note: The python implementation uses time.time_ns, which unfortunately has limited accuracy under windows (16 ms).

Returns
an integer instance

`getContent()`

Get the contents of this sample as a QByteArray. Note that this is an efficient operation due to the copy on write semantics of QByteArray. It also asserts that the original contents cannot be modified.

In C++, make sure to keep an instance of the QByteArray until done with processing. Moreover, consider to use *QByteArray::constData()* for a pointer-to-memory access rather than *QByteArray::data()*, since the latter will eventually make an unnecessary deep copy of the encapsulated data.

Returns
QByteArray instance copy

`getDatatype()`

Return the data type.

Returns
data type string

`getTimestamp()`

Return the timestamp associated to the data.

Returns
integer timestamp

nexxT.interface.Filters module

This module defines the FilterState and Filter classes of the nexxT interface.

`class nexxT.interface.Filters.Filter(dynInPortsSupported, dynOutPortsSupported, environment)`

Bases: `QObject`

Note: Import this class with `from nexxT.interface import Filter`.

This class is the base class for defining a nexxT filter. A minimal nexxT filter class looks like this:

```
class SimpleStaticFilter(Filter):

    def __init__(self, environment):
        super().__init__(False, False, environment)
        pc = self.propertyCollection()
        self.sample_property = pc.getProperty("sample_property", 0.1, "a property␣
↪for demonstration purpose")
        self.inPort = self.addStaticInputPort("inPort")
        self.outPort = self.addStaticOutputPort("outPort")

    def onPortDataChanged(self, inputPort):
        dataSample = inputPort.getData()
        newSample = DataSample.copy(dataSample)
        self.outPort.transmit(dataSample)
```

The constructor of classes derived from nexxT.interface.Filter must have a single argument **environment** which is passed through to this base class. It configures dynamic port usage with the two boolean flags. In the constructor, the filter can define and query properties and create static ports.

The onPortDataChanged is called whenever new data arrives on an input port. In the example above, a copy of the original data sample is returned.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::Filter*

__init__(*dynInPortsSupported*, *dynOutPortsSupported*, *environment*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

addStaticInputPort(*name*, *queueSizeSamples=1*, *queueSizeSeconds=None*)

Shortcut for generating a static input port and adding it to the filter. See also *nexxT.interface.Ports.InputPort()*

Parameters

- **name** – The name of the input port
- **queueSizeSamples** – The size of the input queue in samples
- **queueSizeSeconds** – The size of the input queue in seconds

Returns

the new port instance

addStaticOutputPort(*name*)

Shortcut for generating a static output port and adding it to the filter. See also *nexxT.interface.Ports.OutputPort()*

Parameters

name – The name of the output port

Returns
the new port instance

`addStaticPort`(*port*)

Register a static port for this filter. Only possible in CONSTRUCTING state.

Parameters
port – InputPort or OutputPort instance

Returns
None

`environment`()

Returns the environment associated with this filter.

Returns
a FilterEnvironment instance

`getDynamicInputPorts`()

Get dynamic input ports of this filter. Only possible in and after INITIALIZING state.

Returns
list of dynamic input ports

`getDynamicOutputPorts`()

Get dynamic output ports of this filter. Only possible in and after INITIALIZING state.

Returns
list of dynamic output ports

`guiState`()

Return the gui state associated with this filter. Note: the gui state shall not be used for properties which are important for data transport, for these cases the propertyCollection() shall be used. Typical gui state variables are: the geometry of a user-managed window, the last directory path used in a file dialog, etc. The gui state might not be initialized during mockup-phase.

Returns
PropertyCollection instance

`onClose`()

This function can be overwritten for general de-initialization tasks (e.g. release resources needed to run the filter, close files, etc.). It is the opoosite to onOpen(...).

Returns
None

`onDeinit`()

This function can be overwritten for performing de-initialization tasks related to dynamic ports. It is the opposite to onInit(...)

Returns
None

`onInit`()

This function can be overwritten for performing initialization tasks related to dynamic ports.

Returns
None

onOpen()

This function can be overwritten for general initialization tasks (e.g. acquire resources needed to run the filter, open files, connecting to services etc.).

Returns

onPortDataChanged(*inputPort*)

This function can be overwritten to be notified when new data samples arrive at input ports. For each data sample arrived this function will be called exactly once.

Parameters
inputPort – the port where the data arrived

Returns
None

onStart()

This function can be overwritten to reset internal filter state. It is called before loading a new sequence.

Returns
None

onStop()

Opposite of onStart.

Returns
None

onSuggestDynamicPorts()

Shall return the suggested dynamic ports of this filter. Prominent example is to return the streams contained in a HDF5 file. Note that it is safe to assume that the instance lives in the GUI thread, when this function is called from the nexxT framework.

Returns
listOfInputPortNames, listOfOutputPortNames

propertyCollection()

Return the property collection associated with this filter.

Returns
PropertyCollection instance

removeStaticPort(*port*)

Remove a static port of this filter. Only possible in CONSTRUCTING state.

Parameters
port – InputPort or OutputPort instance

Returns
None

staticMetaObject = PySide6.QtCore.QMetaObject("Filter" inherits "QObject":)

class nexxT.interface.Filters.**FilterState**

Bases: object

Note: Import this class with `from nexxT.interface import FilterState`.

This class defines an enum for the filter states. For reference, the filter's lifecycle state diagram is shown here:

static state2str(*state*)

converts a state integer to the corresponding string

Parameters
state – the state integer

Returns
string

class nexxT.interface.Filters.**FilterSurrogate**(*dllUrls*, *name*)

Bases: object

Note: Import this class with `from nexxT.interface import FilterSurrogate`.

This class acts as a surrogate to reference a filter from a DLL/shared object plugin from within python. It's main purpose is the ability to announce these filters using python entry points. For this, create an instance of this class in one of your modules and refer to it by a 'nexxT.filters' entry point. Example:

```
from distutils.core import setup
setup( # ...
    'nexxT.filters' : [
         'examples.framework.CameraGrabber = nexxT.examples:CameraGrabber',
    ])
```

The surrogate creation of CameraGrabber is performed in *nexxT.examples*:

```
# SPDX-License-Identifier: Apache-2.0
# Copyright (C) 2020 ifm electronic gmbh
#
# THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.
#

"""
define FilterSurrogates for binary filters.
"""

from pathlib import Path
import os
from nexxT.interface import FilterSurrogate
if os.environ.get("READTHEDOCS", None) is None:
    from nexxT.Qt import QtMultimedia # needed to load corresponding DLL before␣
↪loading the nexxT plugin

AviReader = FilterSurrogate(
    "binary://" + str((Path(__file__).parent.parent / "tests" /
                       "binary" / "${NEXXT_PLATFORM}" / "${NEXXT_VARIANT}" / "test_
↪plugins").absolute()),
    "VideoPlaybackDevice"
)
"""
Filter surrogate for the VideoPlaybackDevice class which is defined in a packaged␣
↪shared object "test_plugins".
```

(continues on next page)

(continued from previous page)

```
"""

CameraGrabber = FilterSurrogate(
    "binary://" + str((Path(__file__).parent.parent / "tests" /
                       "binary" / "${NEXXT_PLATFORM}" / "${NEXXT_VARIANT}" / "test_
↪plugins").absolute()),
    "CameraGrabber"
)
"""
Filter surrogate for the CameraGrabber class which is defined in a packaged shared␣
↪object "test_plugins".
"""
```

__init__(*dllUrls*, *name*)

Create a FilterSurrogate instance.

Parameters

- **dllUrls** – might be (1) a dictionary mapping variant names to URLs, variant names are usually “nonopt” and “release” or (2) a url string which will be mapped to the release variant Note that URLs for binary libraries (DLL’s or shared objects) are of the form “binary://<absolute-path-to-dll>”. The absolute path might contain variables like ${NEXXT_PLATFORM} or ${NEXXT_VARIANT}.
- **name** – the name of the filter class or factory function

dllUrl(*variant*)

returns the absolute path of the optimzed dll/shared object

Parameters

variant – the variant for which the filter is needed, given as a string.

name()

returns the name of the filter class

nexxT.interface.Ports module

This module defines the Port, InputPort and OutputPort interface classes of the nexxT framework.

nexxT.interface.Ports.**InputPort**(*dynamic*, *name*, *environment*, *queueSizeSamples=1*, *queueSizeSeconds=None*)

Factory function to creates an InputPortInterface instance with an actual implementation attached. Will be dynamically implemented by the nexxT framework. This is done to prevent having implementation details in the class.

Note: Import this function with `from nexxT.interface import InputPort`.

Parameters

- **dynamic** – boolean whether this is a dynamic input port
- **name** – the name of the port
- **environment** – the FilterEnvironment instance

- **queueSizeSamples** – the size of the queue in samples
- **queueSizeSeconds** – the size of the queue in seconds

Returns
an InputPortInterface instance (actually an InputPortImpl instance)

class nexxT.interface.Ports.**InputPortInterface**(*dynamic*, *name*, *environment*)

Bases: *Port*

This abstract class defines the interface of an input port of a filter. In addition to the normal port attributes, there are two new attributes related to automatic buffering of input data samples. queueSizeSamples sets the maximum number of samples buffered (it can be None, if queueSizeSeconds is not None) queueSizeSeconds sets the maximum time of samples buffered (it can be None, if queueSizeSamples is not None) If both attributes are set, they are and-combined.

..note::
Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::InputPortInterface*

clone(*newEnvironment*)

Return a copy of this port attached to a new environment.

Parameters
newEnvironment – the new FilterEnvironment instance

Returns
a new Port instance

getData(*delaySamples=0*, *delaySeconds=None*)

Return a data sample stored in the queue (called by the filter).

Parameters

- **delaySamples** – 0 related the most actual sample, numbers > 0 relates to historic samples (None can be given if delaySeconds is not None)
- **delaySeconds** – if not None, a delay of 0.0 is related to the current sample, positive numbers are related to historic samples (TODO specify the exact semantics of delaySeconds)

Returns
DataSample instance

interthreadDynamicQueue()

Return the interthread dynamic queue setting.

Returns
a boolean

queueSizeSamples()

return the current queueSize in samples

Returns
an integer

queueSizeSeconds()

return the current queueSize in seconds

Returns
an integer

receiveAsync(*dataSample*, *semaphore*)

Called from framework only and implements the asynchronous receive mechanism using a semaphore.

Parameters

- **dataSample** – the transmitted DataSample instance
- **semaphore** – a QSemaphore instance

Returns
None

receiveSync(*dataSample*)

Called from framework only and implements the synchronous receive mechanism. TODO implement

Parameters
dataSample – the transmitted DataSample instance

Returns
None

setInterthreadDynamicQueue(*enabled*)

If enabled is True, inter thread connections to this input port are dynamically queued for non-blocking behaviour.

This setting does not affect connections from within the same thread. This method can be called only during constructor or the onInit() method of a filter. The main use case is a recording filter where the QT signal/slot is allowed to buffer as many samples as allowed in the input port's queue to prevent unwanted blocking behaviour.

Enabling this might cause a larger delay and might also consume a lot of memory.

Parameters
enabled – whether the dynamic queuing feature is enabled or not.

Returns

setQueueSize(*queueSizeSamples*, *queueSizeSeconds*)

Set the queue size of this port.

Parameters

- **queueSizeSamples** – 0 related the most actual sample, numbers > 0 relates to historic samples (None can be given if delaySeconds is not None)
- **queueSizeSeconds** – if not None, a delay of 0.0 is related to the current sample, positive numbers are related to historic samples

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("InputPortInterface" inherits "Port":)

nexxT.interface.Ports.**OutputPort**(*dynamic*, *name*, *environment*)

Note: Import this function with `from nexxT.interface import OutputPort`.

Factory function to create an OutputPort instance with an actual implementation attached. Will be dynamically implemented by the nexxT framework. This is done to prevent having implementation details in the class

Parameters

- **dynamic** – boolean whether this is a dynamic input port
- **name** – the name of the port
- **environment** – the FilterEnvironment instance

Returns
an OutputPortInterface instance (actually an OutputPortImpl instance)

class nexxT.interface.Ports.**OutputPortInterface**(*dynamic*, *name*, *environment*)

Bases: *Port*

This abstract base class defines the interface of an output port of a filter.

..note::
Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::OutputPortInterface*

clone(*newEnvironment*)

Return a copy of this port attached to a new environment.

Parameters
newEnvironment – the new FilterEnvironment instance

Returns
a new Port instance

setupDirectConnection(*inputPort*)

Setup a direct (intra-thread) connection between outputPort and inputPort Note: both instances must live in same thread!

Parameters

- **outputPort** – the output port instance to be connected
- **inputPort** – the input port instance to be connected

Returns
None

setupInterThreadConnection(*inputPort*, *outputPortThread*, *width*)

Setup an inter thread connection between outputPort and inputPort

Parameters

- **outputPort** – the output port instance to be connected
- **inputPort** – the input port instance to be connected
- **outputPortThread** – the QThread instance of the outputPort instance
- **width** – the width of the connection in DataSamples (0: infinite)

Returns
an InterThreadConnection instance which manages the connection (has to survive until connections is deleted)

staticMetaObject = PySide6.QtCore.QMetaObject("OutputPortInterface" inherits "Port": Methods: #5 type=Signal, signature=transmitSample(PyObject), parameters=PyObject)

transmit(*dataSample*)

transmit a data sample over this port

Parameters
dataSample – sample to transmit

transmitSample

class nexxT.interface.Ports.**Port**(*dynamic*, *name*, *environment*)

Bases: QObject

This class is the base class for ports. It is used mainly as structure, containing the 3 properties dynamic (boolean), name (str) and environment (a FilterEnvironment instance).

..note::
Usually nexxT is using a wrapped C++ class instead of this pure python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::Port*

INPUT_PORT = 0

OUTPUT_PORT = 1

__init__(*dynamic*, *name*, *environment*)

Constructor.

Parameters

- **dynamic** – boolean whether this is a dynamic port or not
- **name** – the port name, given as a string
- **environment** – the corresponding FilterEnvironment instance

clone(*newEnvironment*)

This function must be overwritten in inherited classes to create a clone of this port attached to a different environment.

Parameters
newEnvironment – the new FilterEnvironment instance

Returns
a new Port instance

dynamic()

Returns whether this is a dynamic port

Returns
a boolean

environment()

Returns the environment instance managing this port

Returns
FilterEnvironment instance

isInput()

Returns true if this is an input port

Returns
bool

isOutput()

Returns true if this is an output port

Returns
bool

name()

Returns the port name

Returns
a string

setName(*name*)

Sets the port name

Parameters
name – the port name given as a string

Returns
None

staticMetaObject = PySide6.QtCore.QMetaObject("Port" inherits "QObject":)

nexxT.interface.PropertyCollections module

This module defines the PropertyCollection interface class of the nexxT framework.

class nexxT.interface.PropertyCollections.**PropertyCollection**

Bases: QObject

Note: Import this class with `from nexxT.interface import PropertyCollection`.

This class represents a collection of properties. These collections are organized in a tree, such that there are parent/child relations. This is a generic base class, which is implemented in the core package. Access to properties throug the methods below is thread safe.

Properties are usually used in subclasses of *nexxT.interface.Filters.Filter*. They are presented in the GUI as editable entities and the settings are saved to the configuration file. They are defined during filter constructor and onInit(...), they can be used during the whole filter lifecycle. There is also the possibility to connect a callback function to the *propertyChanged* signal. Note that properties are automatically deleted from the config file if they disappear (e.g., due to code changes, changed dynamic ports, etc.).

Example:

```
class MyFilter(Filter):
    def __init__(self, env):
        super().__init__(False, False, env)
        pc = self.propertyCollection()
        pc.defineProperty("intProp", 1, "an unconstrained integer")
        pc.defineProperty("intPropMax", 1, "an max constrained integer",␣
↪options=dict(max=10))
        pc.defineProperty("intPropBounded", 1, "a bounded integer",␣
↪options=dict(min=-4, max=10))
        pc.defineProperty("floatProp", 1.0, "a floating point number",␣
↪options=dict(min=-1e-4, max=10000))
```

(continues on next page)

(continued from previous page)

```
        pc.defineProperty("boolProp", True, "a boolean")
        # it is also possible to connect a callback
        pc.propertyChanged.connect(self.onPropertyChanged)

    def onInit(self):
        pc = self.propertyCollection()
        pc.defineProperty("strProp", "Hello World", "a string")
        pc.defineProperty("enumProp", "Hello", "a string", options=dict(enum=["Hello
↪", "World"]))

    def onStart(self):
        # queries the current value of the property
        pc = self.propertyCollection()
        intProp = pc.getProperty("intProp")
        # ...

    def onPropertyChanged(self, pc, name):
        logger.info("Property '%s' changed to %s", name, repr(pc.getProperty(name)))
```

In the above example, different editors will be created apropriate to the chosen values. For example, the enum property can be edited using a combo box while integer properties are edited with spin boxes. It is also possible to adapt this behaviour by passing custom propertyHandlers.

This class is an abstract base class.

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::PropertyCollection*

---

defineProperty(*name*, *defaultVal*, *helpstr*, *options=None*, *propertyHandler=None*)

Return the value of the given property, creating a new property if it doesn't exist. If it does exist, the definition must be consistent, otherwise an error is raised.

Note that the parameters options and propertyHandler must not be present at the same time. That is because the options are already passed to the constructor of the propertyHandler.

Parameters

- **name** – the name of the property
- **defaultVal** – the default value of the property. Note that this value will be used to determine the property's type. Currently supported types are string, int and float
- **helpstr** – a help string for the user (presented as a tool tip)
- **options** – a dict mapping string to qvariant (common options: 'min', 'max', 'enum') all properties support the option 'ignoreInconsistentOptions' (default: False). If this option is True, then nexxT allows that the options change over time. Even if present, the option type and its default values are not allowed to change.
- **propertyHandler** – a PropertyHandler instance, or None for automatic choice according to defaultVal

Returns

the current value of this property

evalpath(*path*)

Evaluates the string path. If it is an absolute path it is unchanged, otherwise it is converted to an absolute path relative to the config file path.

Parameters
path – a string

Returns
absolute path as string

getProperty(*name*)

return the property identified by name

Parameters
name – a string

Returns
the current property value

propertyChanged

QT signal which is emitted after a property value of the collection has been changed by the user.

Parameters

- **pc** – the PropertyCollection instance (i.e., the same as self.sender())
- **propName** – the name of the property which has been changed.

setProperty(*name*, *value*)

Set the value of a named property.

Parameters

- **name** – property name
- **value** – the value to be set

Returns
None

```
staticMetaObject = PySide6.QtCore.QMetaObject("PropertyCollection" inherits
"QObject":  Methods:  #5 type=Signal, signature=propertyChanged(PyObject,QString),
parameters=PyObject, QString #6 type=Slot, signature=setProperty(QString,PyObject),
parameters=QString, PyObject )
```

class nexxT.interface.PropertyCollections.**PropertyHandler**

Bases: `object`

Note: Import this class with `from nexxT.interface import PropertyHandler`.

This class represents a property definition for a specific type. The type handles loading/saving from and to .json configs as well as providing editor widgets for modifying the property in a model/view framework.

It is an abstrct base class.

For illustration, the implementation of the IntHandler is given here as an example:

```
class IntHandler(PropertyHandler):
    """
    The property handler for integer properties; Supported options: min and max.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, options):
        """
        Constructor

        :param options: the options given to the defineProperty(...) function.
        """
        for k in options:
            if k in ["min", "max"]:
                if not isinstance(options[k], int):
                    raise PropertyParsingError(f"Unexpected type of option {k};␣
↪expected int.")
            else:
                raise PropertyParsingError(f"Unexpected option {k}; expected 'min'␣
↪or 'max'.")
        self._options = options

    def options(self):
        """
        return this handler's options

        :return: a python dict with the actual options.
        """
        return self._options

    def fromConfig(self, value):
        """
        import value from config file and return the adapted version.

        :param value: an integer is expected
        :return: the validated integer
        """
        assert isinstance(value, (float, int, bool))
        return self.validate(value)

    def toConfig(self, value):
        """
        export value to config file and return the adapted version

        :param value: an integer is expected
        :return: the exported value
        """
        assert isinstance(value, int)
        return value

    def toViewValue(self, value):
        """
        create a view of this option value.

        :param value: the current option value
        :return: a string
```

(continues on next page)

(continued from previous page)

```
        """
        assert isinstance(value, int)
        return str(value)

    def validate(self, value):
        """
        Validate an option value and return an adapted, valid value

        :param value: the value to be tested (an integer)
        :return: the adapted, valid value
        """
        if isinstance(value, str):
            try:
                value = int(value)
            except ValueError:
                logger.warning("Cannot interpret value '%s' as int. Using 0.",␣
↪value)
                value = 0
        if "min" in self._options:
            if value < self._options["min"]:
                logger.warning("Adapted option value %d to minimum value %d.",␣
↪value, self._options["min"])
                return self._options["min"]
        if "max" in self._options:
            if value > self._options["max"]:
                logger.warning("Adapted option value %d to maximum value %d.",␣
↪value, self._options["max"])
                return self._options["max"]
        return int(value)

    def createEditor(self, parent):
        """
        Creates a QSpinBox instance for GUI editing of integer values

        :param parent: the parent of the widget
        :return: a QSpinBox instance
        """
        res = QSpinBox(parent)
        res.setFrame(False)
        if "min" in self._options:
            res.setMinimum(self._options["min"])
        else:
            res.setMinimum(-2147483648)
        if "max" in self._options:
            res.setMaximum(self._options["max"])
        else:
            res.setMaximum(2147483647)
        return res

    def setEditorData(self, editor, value):
        """
        set the value of the QSpinBox
```

(continues on next page)

(continued from previous page)

```
        :param editor: the instance returned by createEditor
        :param value: the option value (an integer)
        :return: None
        """
        editor.setValue(value)

    def getEditorData(self, editor):
        """
        return the currently edited value

        :param editor: the instance returned by createEditor
        :return: the integer value
        """
        return self.validate(editor.value())
```

Note: Usually, nexxT is using the wrapped C++ class instead of the python version. In python there are no differences between the wrapped C++ class and this python class. The C++ interface is defined in *nexxT::PropertyHandler*

`createEditor`(*parent*)

This is called in QStyledItemDelegate::createEditor; creates an editor widget instance.

Parameters
`parent` – a QWidget instance

Returns
a QWidget instance

`fromConfig`(*value*)

Converts the value read from the json file into the native python format

Parameters
`value` – a QVariant instance

Returns
the native value (also a QVariant)

`getEditorData`(*editor*)

This is called in QStyledItemDelegate::setModelData; converts the value from the editor back to native python value.

Parameters
`editor` – the editor widget

Returns
a QVariant, the new native property value

`options`()

Returns the options set for this handler as a QVariantMap

Returns
a QVariantMap instance

`setEditorData`(*editor*, *value*)

This is called in QStyledItemDelegate::setEditorData; populates the editor widget with the actual data.

Parameters

- **editor** – the editor widget as rezurned by createEditor
- **value** – the current property value, given as native python value

Returns
None

`toConfig`(*value*)

Converts the native python format into a value suitable for json files.

Parameters
value – a QVariant instance (the native value)

Returns
the json value (also a QVariant)

`toViewValue`(*value*)

Converts the native python format into a value suitable for display in the Qt model/view framework.

Parameters
value – a QVariant instance (the native value)

Returns
a QVariant value (suitable value for display)

`validate`(*value*)

Returns a validated version of value.

Parameters
value – the value to be set

Returns
a validated version of value

nexxT.interface.Services module

This module defines the Services class of the nexxT framework.

3.1.1.2 nexxT.services package

Subpackages

nexxT.services.gui package

Submodules

nexxT.services.gui.BrowserWidget module

This module provides a widget for browsing the filesystem.

class nexxT.services.gui.BrowserWidget.**BrowserWidget**(*parent=None*)

Bases: QWidget

This class puts together a TabCompletionLineEdit and a list view of teh FolderListModel in one single widget.

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*, *f: PySide6.QtCore.Qt.WindowType = Default(Qt.WindowFlags)*) → None

Initialize self. See help(type(self)) for accurate signature.

activated

active()

The currently activated file

Returns
a string instance

current()

A synonym for active()

Returns
a string instance

folder()

Returns the current folder

Returns
a Path instance

scrollTo(*item*)

Scrolls to the given item.

Parameters
item – a string instance

Returns

setActive(*activeFile*)

set the activated file

Parameters
activeFile – a string or Path instance

Returns

setFilter(*flt*)

Set the name filter of the file browser

Parameters
flt – a string instance or a list of strings

Returns

setFolder(*folder*)

Sets the current folder

Parameters
folder – a string or a Path instance

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("BrowserWidget" inherits "QWidget":
Methods:  #34 type=Signal, signature=activated(QString), parameters=QString )
```

class nexxT.services.gui.BrowserWidget.**FolderListModel**(*parent*)

Bases: QAbstractTableModel

This class provides a model for browsing a folder.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

columnCount(*index=<PySide6.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0))>*)

overwritten from base class

Parameters
index –

Returns

data(*index*, *role*)

overwritten from base class

Parameters

- **index** –
- **role** –

Returns

fileToIndex(*filename*)

return the given file name to a model index.

Parameters
filename – a string or Path instance

Returns
a QModelIndex instance

filter()

Return the current filter

Returns
a list of strings

folder()

Return the current folder.

Returns
a Path instance

folderChanged

headerData(*section*, *orientation*, *role*)

overwritten from base class

Parameters

- **section** –
- **orientation** –
- **role** –

Returns

rowCount(*index=<PySide6.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0))>*)

overwritten from base class

Parameters
index –

Returns

setFilter(*flt*)

Set the filter of this browser

Parameters
flt – string or a list of strings containing glob-style patterns

Returns

setFolder(*folder*)

set the folder of this browser

Parameters
folder – a Path or string instance

Returns

statCache = <nexxT.services.gui.BrowserWidget.StatCache object>

staticMetaObject = PySide6.QtCore.QMetaObject("FolderListModel" inherits "QAbstractTableModel": Methods: #77 type=Signal, signature=folderChanged(QString), parameters=QString)

class nexxT.services.gui.BrowserWidget.**StatCache**

Bases: object

Class for caching file-system related accesses to prevent unnecessary slowliness for network drives.

MAX_NUM_CACHE_ENTRIES = 20480

__init__()

class nexxT.services.gui.BrowserWidget.**TabCompletionLineEdit**(*completer*, *parent=None*)

Bases: QLineEdit

This class provides a line edit which changes the tab-key semantics to interact with a completer.

__init__(*self*, *arg__1: str*, *parent: PySide6.QtWidgets.QWidget | None = None*) → None

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

event(*event*)

overwritten from base class

Parameters
event –

Returns

keyPressEvent(*event*)

Overwritten from QWidget, intermit paste events to log in the text directly after pasting.

nextCompletion(*direction*)

interacts with the completer, selects next / previous item

Parameters
direction – the direction, either -1 or +1

Returns

pasted

staticMetaObject = PySide6.QtCore.QMetaObject("TabCompletionLineEdit" inherits "QLineEdit": Methods: #59 type=Signal, signature=pasted())

nexxT.services.gui.Configuration module

This module provides the Configuration GUI service of the nexxT framework.

class nexxT.services.gui.Configuration.**MVCConfigurationGUI**(*configuration*)

Bases: *MVCConfigurationBase*

GUI implementation of MVCConfigurationBase

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

activeAppStateChange(*newState*)

Called when the active application changes its state.

Parameters
newState – the new application's state (see FilterState)

Returns

appActivated(*name*, *app*)

Called when the application is activated.

Parameters

- **name** – the application name
- **app** – An ActiveApplication instance.

Returns

restoreState()

Restore the state of the configuration gui service (namely the recently open config files). This is saved in QSettings because it is used across config files.

Returns

saveState()

Save the state of the configuration gui service (namely the recently open config files). This is saved in QSettings because it is used across config files.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("MVCConfigurationGUI" inherits "MVCConfigurationBase":)

nexxT.services.gui.GraphEditor module

This module provides the graph editor GUI service of the nexxT service.

class nexxT.services.gui.GraphEditor.**BaseGraphScene**(*parent*)

Bases: QGraphicsScene

Basic graph display and manipulation scene. Generic base class intended to be overwritten.

class ConnectionItem(*portFrom*, *portTo*)

Bases: QGraphicsPathItem

This item corresponds with a connection between an output and an input port.

__init__(*self*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

__init__(*self*, *path: PySide6.QtGui.QPainterPath*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

hoverEnterEvent(*event*)

override for hover enter events

Parameters

event – the QT event

Returns

hoverLeaveEvent(*event*)

override for hover leave events

Parameters

event – the QT event

Returns

static itemTypeName()

Returns an identificytion string.

Returns

shape()

Unsure if this is needed, but it may give better hover positions

Returns

sync()

Synchronizes the view with the model.

Returns

KEY_ITEM = 0

class NodeItem(*name*)

Bases: QGraphicsItemGroup

An item which represents a node in the graph. The item group is also used for grouping the port items.

__init__(*self*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

addInPortItem(*name*)

Adds a new input port to the node

Parameters

name – the port name

Returns

addOutPortItem(*name*)

Adds a new output port to the node

Parameters
name – the port name

Returns

getInPortItem(*name*)

Searches for the input port named by name

Parameters
name – a string instance

Returns
a PortItem instance

getOutPortItem(*name*)

Searches for the output port named by name

Parameters
name – a string instance

Returns
a PortItem instance

hoverEnter()

Slot called on hover enter

Returns

hoverLeave()

Slot called on hover leave.

Returns

itemChange(*change*, *value*)

overwritten from QGraphicsItem

Parameters
- **change** – the thing that has changed
- **value** – the new value

Returns

static itemTypeName()

return a class identification name.

Returns

nodeHeight()

Returns
the node height in pixels including spacing.

nodeWidth()

Returns
the node width in pixels including spacing.

paint(*painter*, *option*, *widget*)

Overwritten from base class to prevent drawing the selection rectangle

Parameters
- **painter** – a QPainter instance
- **option** – a QStyleOptionGraphicsItem instance
- **widget** – a QWidget instance or None

Returns

sync()

synchronize the item with the model (also the ports)

Returns

class PortItem(*name*, *nodeItem*)

Bases: object

This class represents a port in a node.

__init__(*name*, *nodeItem*)

hoverEnter()

Slot called on hover enter

Returns

hoverLeave()

Slot called on hover leave.

Returns

static itemTypeName()

Returns a class identification string.

Returns

remove()

Removes this port from its item group.

Returns

scenePosChanged(*value*)

Slot called on scene position changes, need to synchronize connections.

Parameters

value –

Returns

setPos(*x*, *y*, *isOutput*)

Sets the position of this item to the given coordinates, assigns output / input property.

Parameters

- **x** – the x coordinate
- **y** – the y coordinate
- **isOutput** – a boolean

Returns

sync()

Synchronizes the item to the model.

Returns

STYLE_ROLE_BRUSH = 2

STYLE_ROLE_HSPACING = 5

STYLE_ROLE_PEN = 1

STYLE_ROLE_RRRADIUS = 3

STYLE_ROLE_SIZE = 0

STYLE_ROLE_TEXT_BRUSH = 6

STYLE_ROLE_VSPACING = 4

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

__init__(*self*, *sceneRect: PySide6.QtCore.QRectF | PySide6.QtCore.QRect*, *parent: PySide6.QtCore.QObject | None = None*) → None

__init__(*self*, *x: float*, *y: float*, *width: float*, *height: float*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

addConnection(*nodeFrom*, *portFrom*, *nodeTo*, *portTo*)

Add a connection to the graph

Parameters

- **nodeFrom** – the start node's name
- **portFrom** – the start node's port
- **nodeTo** – the end node's name
- **portTo** – the end node's port

Returns

addInPort(*node*, *name*)

add an input port to a node

Parameters

- **node** – the node name
- **name** – the port name

Returns

addNode(*name*)

Add a named node to the graph

Parameters
name – a string instance

Returns

addOutPort(*node*, *name*)

Adds an output port to a node

Parameters

- **node** – the node name
- **name** – the port name

Returns

autoLayout()

Automatic layout of nodes using a heuristic layering algorithm.

Returns

connectionAddRequest

`static getData`(*item*, *role*)

returns render-relevant information about the specified item can be overriden in concrete editor instances

Parameters

- **item** – an instance of BaseGraphScene.NodeItem, BaseGraphScene.PortItem or BaseGraphScene.ConnectionItem
- **role** – one of STYLE_ROLE_SIZE, STYLE_ROLE_PEN, STYLE_ROLE_BRUSH, STYLE_ROLE_RRRADIUS, STYLE_ROLE_VSPACING, STYLE_ROLE_HSPACING

Returns
the expected item related to the role

`graphItemAt`(*scenePos*)

Returns the graph item at the specified scene position

Parameters
scenePos – a QPoint instance

Returns
a NodeItem, PortItem or ConnectionItem instance

`mouseMoveEvent`(*event*)

Override from QGraphicsScene (used for dragging connections)

Parameters
event – the QT event

Returns

`mousePressEvent`(*event*)

Override from QGraphicsScene (used for dragging connections)

Parameters
event – the QT event

Returns

`mouseReleaseEvent`(*event*)

Override from QGraphicsScene (used for dragging connections)

Parameters
event – the QT event

Returns

`removeConnection`(*nodeFrom*, *portFrom*, *nodeTo*, *portTo*)

Removes a connection from the graph

Parameters

- **nodeFrom** – the start node's name
- **portFrom** – the start node's port
- **nodeTo** – the end node's name
- **portTo** – the end node's port

Returns

removeInPort(*node*, *name*)

Remove an input port from a node

Parameters

- **node** – the node name
- **name** – the port name

Returns

removeNode(*name*)

Remove a node from the graph

Parameters

name – the node name

Returns

removeOutPort(*node*, *name*)

Remove an output port from a node

Parameters

- **node** – the node name
- **name** – the port name

Returns

renameInPort(*node*, *oldName*, *newName*)

Rename an input port from a node

Parameters

- **node** – the node name
- **oldName** – the old port name
- **newName** – the new port name

Returns

renameNode(*oldName*, *newName*)

Rename a node in the graph

Parameters

- **oldName** – the old name
- **newName** – the new name

Returns

renameOutPort(*node*, *oldName*, *newName*)

Rename an output port from a node

Parameters

- **node** – the node name
- **oldName** – the old port name
- **newName** – the new port name

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("BaseGraphScene" inherits
"QGraphicsScene":  Methods:  #22 type=Signal,
signature=connectionAddRequest(QString,QString,QString,QString), parameters=QString,
QString, QString, QString )
```

class nexxT.services.gui.GraphEditor.**GraphScene**(*graph*, *parent*)

Bases: *BaseGraphScene*

Concrete class interacting with a BaseGraph or FilterGraph instance

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

__init__(*self*, *sceneRect: PySide6.QtCore.QRectF | PySide6.QtCore.QRect*, *parent: PySide6.QtCore.QObject | None = None*) → None

__init__(*self*, *x: float*, *y: float*, *width: float*, *height: float*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

addFilterFromEntryPoint()

Add a filter from its corresponding entry point (the entry point is deduced from the sender action's data()).

Returns

addInputPort()

Adds an input port to a node

Returns

addOutputPort()

Adds an output port to a node

Returns

compositeFilters()

Get a list of names of composite filters.

Returns
a list of strings

contextMenuEvent(*event*)

Overwritten from QGraphicsScene.

getData(*item*, *role*)

returns render-relevant information about the specified item can be overriden in concrete editor instances

Parameters

- **item** – an instance of BaseGraphScene.NodeItem, BaseGraphScene.PortItem or BaseGraphScene.ConnectionItem
- **role** – one of STYLE_ROLE_SIZE, STYLE_ROLE_PEN, STYLE_ROLE_BRUSH, STYLE_ROLE_RRRADIUS, STYLE_ROLE_VSPACING, STYLE_ROLE_HSPACING

Returns
the expected item related to the role

onAddComposite()

Called when the user wants to add a new composite filter to this graph.

Returns

onAddFilterFromFile()

Called when the user wants to add a new filter from a file (FilterGraph variant). Opens a dialog to select the file.

Returns

onAddFilterFromMod()

Called when the user wants to add a new filter from a python module.

Returns

onAddNode()

Called when the user wants to add a new node. (Generic variant)

Returns

onConnSetBlocking()

Sets the conmnection to blocking mode.

Returns

onConnSetCustom()

Sets the connection to a custom width.

Returns

onConnSetNonBlocking()

Sets the conmnection to non blocking mode.

Returns

onConnectionRemove()

Removes a connection

Returns

onSuggestDynamicPorts()

Called when the user wants to add dynamic ports based on the filter's suggestions.

Returns

removeDialog()

Opens a dialog for removing an item (Node or Port)

Returns

renameDialog()

Opens a dialog for renamign an item (node or port)

Returns

setThread()

Opens a dialog to enter the new thread of the node.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("GraphScene" inherits "BaseGraphScene":)

class nexxT.services.gui.GraphEditor.**MyGraphicsPathItem**(**args*, ***kw*)

Bases: QGraphicsPathItem, QObject

Little subclass for receiving hover events and scene position changes outside the items

__init__(*self*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

__init__(*self*, *path: PySide6.QtGui.QPainterPath*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

hoverEnter

hoverEnterEvent(*event*)

emit corresponding singal

Parameters
event – the qt event

Returns

hoverLeave

hoverLeaveEvent(*event*)

emit corresponding signal

Parameters
event – the qt event

Returns

itemChange(*change*, *value*)

in case of scene position changes, emit the corresponding signal

Parameters

- **change** – what has changed
- **value** – the new value

Returns

scenePosChanged

staticMetaObject = PySide6.QtCore.QMetaObject("MyGraphicsPathItem" inherits "QObject": Methods: #5 type=Signal, signature=hoverEnter() #6 type=Signal, signature=hoverLeave() #7 type=Signal, signature=scenePosChanged(QPointF), parameters=QPointF)

class nexxT.services.gui.GraphEditor.**MySimpleTextItem**(**args*, ***kw*)

Bases: QGraphicsSimpleTextItem

QGraphicsSimpleTextItem with a background brush

__init__(*self*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

__init__(*self*, *text: str*, *parent: PySide6.QtWidgets.QGraphicsItem | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

paint(*painter*, *option*, *widget*)

first paint the background and afterwards use standard paint method

Parameters

- **painter** – a QPainter instance
- **option** – unused
- **widget** – unused

Returns

setBackgroundBrush(*brush*)

set the background brush

Parameters
brush – a QBrush instance

Returns

class nexxT.services.gui.GraphEditor.**PortSelectorDialog**(*parent*, *inputPorts*, *outputPorts*, *graph*, *nodeName*)

Bases: QDialog

Dialog for selecting the ports which shall be created.

__init__(*parent*, *inputPorts*, *outputPorts*, *graph*, *nodeName*)

Constructor

Parameters

- **parent** – this dialog's parent widget
- **inputPorts** – the list of input port names
- **outputPorts** – the list of output port names
- **graph** – the corresponding BaseGraph instance
- **nodeName** – the name of the corresponding node

accept()

Accepts user selection.

static getSelectedPorts(*parent*, *inputPorts*, *outputPorts*, *graph*, *nodeName*)

Convenience function for executing the dialog.

Parameters

- **parent** – this dialog's parent widget
- **inputPorts** – the list of input port names
- **outputPorts** – the list of output port names
- **graph** – the corresponding BaseGraph instance
- **nodeName** – the name of the corresponding node

reject()

Rejects user selection.

staticMetaObject = PySide6.QtCore.QMetaObject("PortSelectorDialog" inherits "QDialog":)

nexxT.services.gui.GraphEditorView module

This module provides the GraphEditorView class.

class nexxT.services.gui.GraphEditorView.**GraphEditorView**(*parent*)

Bases: QGraphicsView

Subclass of QGraphicsView which handles copy&paste events

__init__(*parent*)

Constructor

Parameters
parent – a QWidget instance

keyPressEvent(*event*)

Overwritten from QGraphicsView for intercepting copy & paste events.

Parameters
event – a QKeyEvent instance

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("GraphEditorView" inherits "QGraphicsView":)

nexxT.services.gui.GraphLayering module

This module implements a graph layering algorithm inspired by https://en.wikipedia.org/wiki/Layered_graph_drawing

class nexxT.services.gui.GraphLayering.**GraphRep**(*baseGraphScene=None*)

Bases: object

This class implements the "auto layout" feature for nexxT configuration GUI service.

__init__(*baseGraphScene=None*)

addEdge(*n1*, *n2*)

Adds an edge to the graph

Parameters

- **n1** – from node (string id)
- **n2** – to node (string id)

Returns

addNode(*n*)

Adds a new node to the graph

Parameters
n – a unique string id

Returns
None

assignLayers()

Assign nodes to layers

Returns
layers (a list of a list of nodes), node2layer (a dictionary assigning nodes to layer indices)

dump(*title=None*)

Dump to stdout for debugging

Parameters
title – an optional title for the stdout section

Returns

layersToNodeNames(*layers*)

convert the layering result back to node names

Parameters
layers – result from sortLayers (list of list of int)

Returns
list of list of string

sortLayers()

Sort the layers to avoid too many crossings. Note that this does not take the non-sortable ports into account.

Returns
layers (a list of a list of nodes), numCrosses (number of crossings in the graph)

topological_sort()

Topological sorting of the graph. Side effect: self.cycleEdges is a set of edges to be ignored for forcing a DAG.

Returns

nexxT.services.gui.GuiLogger module

This module provides the gui logging service for nexxT.

class nexxT.services.gui.GuiLogger.**GuiLogger**

Bases: *ConsoleLogger*

Logging service in GUI mode.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

detach()

This slot is called when the service is removed from the framework

removeHandler()

Remove logging handler from python logging system.

setLogLevel()

Sets the current log level from the calling action.

Returns
None

```
staticMetaObject = PySide6.QtCore.QMetaObject("GuiLogger" inherits "ConsoleLogger":
Methods:  #6 type=Slot, signature=detach() )
```

class nexxT.services.gui.GuiLogger.**LogHandler**(*logView*)

Bases: Handler

Python logging handler which passes python log records to the gui.

__init__(*logView*)

Initializes the instance - basically setting the formatter to None and the filter list to empty.

emit(*record*)

called when a new log record is created

Parameters
record – a log record instance (see python docs)

Returns

class nexxT.services.gui.GuiLogger.**LogView**

Bases: QTableView

Class implementing the GUI log display.

class LogModel

Bases: QAbstractItemModel

Model/view model for log entries. The entries are held in a python list.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

clear()

removes all entries from the list

Returns
None

columnCount(*parent*)

Overwritten from QAbstractItemModel

Parameters
parent – a QModelIndex instance

Returns
the number of columns (constant)

data(*modelIndex*, *role*)

Overwritten from QAbstractItemModel

Parameters
- **modelIndex** – a QModelIndex instance
- **role** – the role

Returns
the requested data

headerData(*section*, *orientation*, *role*)

Overwritten from QAbstractItemModel

Parameters
- **section** – the section index
- **orientation** – the orientation
- **role** – the item role

Returns
the section label

`index`(*row*, *column*, *parent*)

Overwritten from QAbstractItemModel

Parameters
- `row` – integer
- `column` – integer
- `parent` – a QModelIndex instance

Returns
a QModelIndex for the specified item

`parent`(*index*)

Overwritten from QAbstractItemModel

Parameters
`index` –

Returns
invalid model index (because we have a 2D table)

`rowCount`(*parent*)

Overwritten from QAbstractItemModel

Parameters
`parent` – a QModelIndex instance

Returns
number of log entries

`setSingleLineMode`(*enabled*)

called from the table view to indicate single line mode (in this mode only the last line of a log message is displayed.

Parameters
`enabled` – boolean

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("LogModel" inherits
"QAbstractItemModel":  )
```

`update`(*queue*)

add queued items to model

Parameters
`queue` – a python Queue instance

Returns
None

`__init__`(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

`addLogRecord`(*items*)

Add a log record to the synchronized queue

Parameters
`items` – a tuple of (timestamp[str], level[int], message[str], modulename[str], filename[str], lineno[int])

Returns
None

changeEvent(*event*)

This for instance happens when the style sheet changed. It may affect the calculated row height. So invalidate:

Parameters
event – the event causing the change

Returns

clear()

Clears the view

Returns
None

setFollow(*follow*)

set follow mode

Parameters
follow – a boolean

Returns
None

setUniformRowHeights(*enabled*)

Takeover from QTreeView, see also here https://stackoverflow.com/questions/50943356/qtableview-performance

Parameters
enabled –

Returns

sizeHintForRow(*row*)

return a size hint for the given row index

Parameters
row – the row index as integer

Returns
the height of the row as integer

staticMetaObject = PySide6.QtCore.QMetaObject("LogView" inherits "QTableView":)

update()

Called periodically to synchronize model with added log records

Returns
None

nexxT.services.gui.MainWindow module

This module provides a MainWindow GUI service for the nexxT framework.

class nexxT.services.gui.MainWindow.**MainWindow**(*config*)

Bases: QMainWindow

Main Window service for the nexxT frameworks. Other services usually create dock windows, filters use the subplot functionality to create grid-layouted views.

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*, *flags: PySide6.QtCore.Qt.WindowType = Default(Qt.WindowFlags)*) → None

Initialize self. See help(type(self)) for accurate signature.

aboutToClose

closeEvent(*closeEvent*)

Override from QMainWindow, saves the state.

Parameters
closeEvent – a QCloseEvent instance

Returns

deferredUpdate(*obj*, *slotName*)

Call this method to defer a call to the sepcified slot (obj->slotName()) in accordance with the user-defined framerate set. The slot will be called via getattr(obj, slotName)()

Parameters

- obj – A QObject instance
- slotName – The name of the slot.

eventFilter(*obj*, *event*)

Overwritten from QObject, for disabling mouse wheel scrolling on the main widget's viewport

getToolBar()

Get the main toolbar (adds seperators as appropriate).

Returns

ignoreCloseEvent()

Can be called in slots connected to aboutToClose for requesting to ignore the event. Use case is the "There are unsaved changes" dialog.

Returns

mdiSubWindowCreated

newDockWidget(*name*, *parent*, *defaultArea*, *allowedArea=DockWidgetArea.None*, *defaultLoc=None*)

This function is supposed to be called by services

Parameters

- name – the name of the dock window
- parent – the parent (usually None)
- defaultArea – the default dock area
- allowedArea – the allowed dock areas

Returns
a new QDockWindow instance

newMdiSubWindow(*filterOrService*, *windowTitle=None*)

Deprectated (use subplot(...) instead): This function is supposed to be called by filters.

Parameters

- filterOrService – a Filter instance
- windowTitle – the title of the window (might be None)

Returns
a new QMdiSubWindow instance

`static parseWindowId(windowId)`

convers a subplot window id into windowTitle, row and column

Parameters
windowId – the window id

Returns
title, row, column

`releaseSubplot(arg)`

This needs to be called to release the previously allocated subplot called windowId. The managed widget is deleted as a consequence of this function.

Parameters
arg – the widget as passed to subplot. Passing the windowId is also supported, but deprecated.

Returns

`restoreConfigSpecifics()`

restores the config-specific state of the main window.

`restoreState()`

restores the state of the main window including the dock windows of Services

Returns

`saveConfigSpecifics()`

saves the config-specific state of the main window.

`saveMdiState()`

saves the state of the individual MDI windows

Returns

`saveState()`

saves the state of the main window including the dock windows of Services

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("MainWindow" inherits "QMainWindow":
Methods:  #40 type=Signal, signature=mdiSubWindowCreated(QMdiSubWindow*),
parameters=QMdiSubWindow* #41 type=Signal, signature=aboutToClose(PyObject),
parameters=PyObject #42 type=Signal, signature=userSelectionChanged(QString,QPoint),
parameters=QString, QPoint #43 type=Slot,
signature=deferredUpdate(QObject*,QString), parameters=QObject*, QString #44
type=Slot, signature=updateSelection(QString,QPoint), parameters=QString, QPoint #45
type=Slot, signature=getToolBar() #46 type=Slot,
signature=newDockWidget(QString,QObject*,int,int), parameters=QString, QObject*,
int, int #47 type=Slot, signature=subplot(QString,QObject*,QWidget*),
parameters=QString, QObject*, QWidget* #48 type=Slot,
signature=releaseSubplot(QString), parameters=QString #49 type=Slot,
signature=releaseSubplot(QWidget*), parameters=QWidget* #50 type=Slot,
signature=newMdiSubWindow(QObject*,QString), parameters=QObject*, QString #51
type=Slot, signature=newMdiSubWindow(QObject*), parameters=QObject* )
```

subplot(*windowId*, *theFilter*, *widget*)

Adds widget to the GridLayout specified by windowId.

Parameters

- **windowId** – a string with the format "<windowTitle>[<row>,<col>]" where <windowTitle> is the caption of the MDI window (and it is used as identifier for saving/restoring window state) and <row>, <col> are the coordinates of the addressed subplots (starting at 0)
- **theFilter** – a Filter instance which is requesting the subplot
- **widget** – a QWidget which shall be placed into the grid layout. Note that this widget is reparented as a result of this operation and the parents can be used to get access to the MDI sub window. Use releaseSubplot to remove the window

Returns
None

updateSelection(*group*, *point*)

QT Meta-function which can be called to update the 2D selection.

Parameters

- **group** – the group name given as str/QString
- **point** – the new selection point given as QPoint

userSelectionChanged

class nexxT.services.gui.MainWindow.**NexxTDockWidget**

Bases: QDockWidget

Need subclassing for getting close / show events

closeEvent(*closeEvent*)

override from QMdiSubWindow

Parameters
closeEvent – a QCloseEvent instance

Returns

showEvent(*showEvent*)

override from QMdiSubWindow

Parameters
closeEvent – a QShowEvent instance

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("NexxTDockWidget" inherits "QDockWidget": Methods: #41 type=Signal, signature=visibleChanged(bool), parameters=bool)

visibleChanged

class nexxT.services.gui.MainWindow.**NexxTMdiSubWindow**

Bases: QMdiSubWindow

Need subclassing for getting close / show events and saving / restoring state.

closeEvent(*closeEvent*)

override from QMdiSubWindow

Parameters
closeEvent – a QCloseEvent instance

Returns

restoreGeometry(*geometry*)

Restores the geometry of this subwindow

Parameters
geometry – the saved state as a QByteArray instance

Returns

saveGeometry()

Saves the geometry of this subwindow (see https://bugreports.qt.io/browse/QTBUG-18648)

Returns
a ByteArray instance

showEvent(*showEvent*)

override from QMdiSubWindow

Parameters
closeEvent – a QShowEvent instance

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("NexxTMdiSubWindow" inherits "QMdiSubWindow": Methods: #41 type=Signal, signature=visibleChanged(bool), parameters=bool)

visibleChanged

nexxT.services.gui.PlaybackControl module

This module provides the nexxT GUI service PlaybackControl

class nexxT.services.gui.PlaybackControl.**MVCPlaybackControlGUI**(*config*)

Bases: *PlaybackControlConsole*

GUI implementation of MVCPlaybackControlBase

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

browserActivated(*filename*)

Called when the user activated a file.

Parameters
filename – the new filename

Returns

static compressFileName(*filename*)

Compresses long path names with an ellipsis (...)

Parameters
filename – the original path name as a Path or string instance

Returns
the compressed path name as a string instance

`displayPosition`(*value*)

Slot called when the slider is moved. Displays the position without actually seeking to it.

Parameters
value – the new slider value.

Returns

`nameFiltersChanged`

`onSliderValueChanged`(*value*)

Slot called whenever the slider value is changed.

Parameters
value – the new slider value

Returns

`openRecent`()

Called when the user clicks on a recent sequence.

Returns

`restoreState`()

Restores the state of the playback control from the given property collection

Parameters
propertyCollection – a PropertyCollection instance

Returns

`saveState`()

Saves the state of the playback control

Returns

`scrollToCurrent`()

Scrolls to the current item in the browser

Returns

`selectedStream`()

Returns the user-selected stream (for forward/backward stepping)

Returns

`setSelectedStream`(*stream*)

Sets the user-selected stream (for forward/backward stepping)

:param stream the stream name. :return:

```
staticMetaObject = PySide6.QtCore.QMetaObject("MVCPlaybackControlGUI" inherits
"PlaybackControlConsole":  Methods:  #22 type=Signal,
signature=nameFiltersChanged(QStringList), parameters=QStringList )
```

nexxT.services.gui.Profiling module

This module provides the gui part of the profiling service for nexxT.

class nexxT.services.gui.Profiling.**LoadDisplayWidget**(*parent*)

Bases: QWidget

This widget displays the thread-specific load.

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*, *f: PySide6.QtCore.Qt.WindowType = Default(Qt.WindowFlags)*) → None

Initialize self. See help(type(self)) for accurate signature.

baseTimestamp = None

newLoadData(*threadName*, *timestamps*, *load*)

Slot called when new load data is available

Parameters

- **threadName** – the name of the thread given as string
- **loadData** – the load data, given as the QByteArray of a n x 2 np.float32 array

Returns

paintEvent(*event*)

Manually implemented paint event

Parameters
event – the QT paint event

Returns

removeThread(*thread*)

Remove the thread from the stored load data.

Parameters
thread – the name of the thread to be removed.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("LoadDisplayWidget" inherits "QWidget": Methods: #34 type=Slot, signature=newLoadData(QString,QByteArray), parameters=QString, QByteArray #35 type=Slot, signature=removeThread(QString), parameters=QString)

class nexxT.services.gui.Profiling.**Profiling**

Bases: *ProfilingService*

GUI part of the nexxT profiling service.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

setLoadMonitorEnabled(*enabled*)

called when the corresponding QAction is toggled

Parameters
enabled – boolean

Returns

setPortProfilingEnabled(*enabled*)

called when the corresponding QAction is toggled

Parameters
enabled – boolean

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("Profiling" inherits "ProfilingService":)

class nexxT.services.gui.Profiling.**SpanDisplayWidget**(*parent*)

Bases: QWidget

This Widget displays the time/occupancy profiling overview based on the input ports events.

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*, *f: PySide6.QtCore.Qt.WindowType = Default(Qt.WindowFlags)*) → None

Initialize self. See help(type(self)) for accurate signature.

event(*event*)

Event filter for generating tool tips.

Parameters
event – a QEvent instance.

Returns

newSpanData(*threadName*, *portName*, *spanData*)

This slot is called when new profiling data is available.

Parameters

- **threadName** – the name of the associated thread
- **portName** – the full-qualified name of the port
- **spanData** – the profiling data, given as the byte array representation of a n x 2 int64 array.

Returns

paintEvent(*event*)

Manually implemented paint event of the time / occupancy diagram.

Parameters
event – the qt paint event

Returns

removeThread(*thread*)

Lazily removes the thread from the profiling data. To be able to inspect the data when the application is stopped, the data will actually be removed when new data of the thread is available.

Parameters
thread – the name of the thread to be removed.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("SpanDisplayWidget" inherits "QWidget": Methods: #34 type=Slot, signature=newSpanData(QString,QString,QByteArray), parameters=QString, QString, QByteArray #35 type=Slot, signature=removeThread(QString), parameters=QString)

textDescription(*thread*, *port*)

Tooltip text generation.

Parameters

- **thread** – the name of the corresponding thread
- **port** – the full-qualified port name.

Returns

a string instance containing the profiling info.

nexxT.services.gui.PropertyDelegate module

This module provides a delegate for use in the Configuration GUI service to edit properties.

class nexxT.services.gui.PropertyDelegate.**PropertyDelegate**(*model*, *role*, *PropertyContent*, *parent*)

Bases: QStyledItemDelegate

This class provides a delegate for providing editor widgets for the nexxT gui service Configuration.

__init__(*model*, *role*, *PropertyContent*, *parent*)

Constructor.

Parameters

- **model** – An instance of the nexxT gui service implementation of QAbstractItemModle
- **role** – the role which can be used to query the property items
- **PropertyContent** – the class of the property items queried with model->data(..., self.role)
- **parent** – the parent QObject

createEditor(*parent*, *option*, *index*)

Create an editor for the given index (if this is not a PropertyContent, the default implementation is used)

Parameters

- **parent** – the parent of the editor
- **option** – unused
- **index** – the model index

Returns

an editor widget

setEditorData(*editor*, *index*)

Populate the editor with the data from the model

Parameters

- **editor** – the editor as created by createEditor
- **index** – the index into the model

Returns

setModelData(*editor*, *model*, *index*)

Commit the data from the editor into the model

Parameters

- **editor** – the editor as returned by createEditor
- **model** – the model
- **index** – an index to the model

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("PropertyDelegate" inherits "QStyledItemDelegate":)

nexxT.services.gui.RecordingControl module

This module provides the recording control GUI service for the nexxT framework.

class nexxT.services.gui.RecordingControl.**MVCRecordingControlGUI**(*config*)

Bases: *MVCRecordingControlBase*

This service implements a GUI frontend for the recording service

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

staticMetaObject = PySide6.QtCore.QMetaObject("MVCRecordingControlGUI" inherits "MVCRecordingControlBase":)

Submodules

nexxT.services.ConsoleLogger module

This module provides a service which maps log messages coming from c++ and qt to python log messages. It is automatically used by NEXT_LOG_*() macros in c++.

class nexxT.services.ConsoleLogger.**ConsoleLogger**

Bases: QObject

Logging service to console (using python logging module). This class is used to log messages in C++.

log(*level*, *message*, *file*, *line*)

Called from c++ to log a message.

Parameters

- **level** – logging compatible log level
- **message** – message as a string
- **file** – file which originated the log message
- **line** – line of log message statement

Returns

static qtMessageHandler(*qtMsgType*, *qMessageLogContext*, *msg*)

Qt message handler for handling qt messages in normal logging.

Parameters

- **qtMsgType** – qt log level
- **qMessageLogContext** – qt log context
- **msg** – message as a string

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("ConsoleLogger" inherits "QObject":
Methods:  #5 type=Slot, signature=log(int,QString,QString,int), parameters=int,
QString, QString, int )
```

nexxT.services.ConsoleLogger.**makeRecord**(*self*, *name*, *level*, *filename*, *lineno*, *msg*, *args*, *excInfo*, *func=None*, *extra=None*, *sinfo=None*)

A factory method which can be overridden in subclasses to create specialized LogRecords.

nexxT.services.SrvConfiguration module

This module provides the basic Configuration services of the nexxT framework.

class nexxT.services.SrvConfiguration.**ConfigurationModel**(*configuration*, *parent*)

Bases: QAbstractItemModel

This class encapsulates a next Configuration item in a QAbstractItemModel ready for usage in a QTreeView.

Internally, item-trees are constructed which duplicate the relevant information from the python classes.

class Item(*parent*, *content*)

Bases: object

An item instance for creating the item tree. Items have children, a parent and arbitrary content.

__init__(*parent*, *content*)

row()

Returns
the index of this item in the parent item.

class NodeContent(*subConfig*, *name*)

Bases: object

A node in a subconfig, for usage within the model

__init__(*subConfig*, *name*)

class PropertyContent(*name*, *propertyCollection*)

Bases: object

A property in a propertyCollection, for usage within the model.

__init__(*name*, *propertyCollection*)

class SubConfigContent(*subConfig*)

Bases: object

A subConfiguration, for usage within the model.

__init__(*subConfig*)

class VariableContent(*name*, *variables*)

Bases: object

A variable definition in a Variables instance, for usage within the model.

__init__(*name*, *variables*)

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

appActivated(*name*, *app*)

This slot is called when an application has been activated.

Parameters

- **name** – the name of the application
- **app** – the Application instance

Returns

columnCount(*parent*)

Returns the number of columns of the given model index

Parameters

parent – a QModelIndex instance

Returns

data(*index*, *role*)

Generic data query

Parameters

- **index** – a QModelIndex instance
- **role** – the data role (see QAbstractItemModel)

Returns

flags(*index*)

Returns teh item flags of the given index

Parameters

index – a QModelIndex instance

Returns

headerData(*section*, *orientation*, *role*)

Overwritten from QAbstractItemModel. Provide header names for the columns.

index(*row*, *column*, *parent=<PySide6.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0))>*)

Creates a model index according to QAbstractItemModel conventions.

Parameters

- **row** – the row index
- **column** – the column index
- **parent** – the parent index

Returns

indexOfNode(*subConfig*, *node*)

Returns the index of the given node inside a subconfig.

Parameters

- **subConfig** – a SubConfiguration instance
- **node** – a node name

Returns
a QModelIndex instance

indexOfProperty(*nodeItem*, *propName*)

Returns the model index of the specified property

Parameters

- **nodeItem** – a NodeItem instance
- **propName** – a property name

Returns
a QModelIndex instance

indexOfSubConfig(*subConfig*)

Returns the index of the given subConfig

Parameters
subConfig – a SubConfiguration instance

Returns
a QModelIndex instance.

indexOfSubConfigParent(*subConfig*)

Returns the index of the given subConfig's parent

Parameters
subConfig – a SubConfiguration instance

Returns
a QModelIndex instance.

indexOfVariable(*vitem*)

Returns the index of the given variable by full-recursive searching.

Parameters
vitem – the variable item to be search for.

static isApplication(*index*)

Returns true, if this index relates to an applications, false otherwise.

Parameters
index – a QModelIndex instance

Returns
bool

isSubConfigParent(*index*)

Returns CONFIG_TYPE_COMPOSITE if the index refers to the group composite, CONFIG_TYPE_APPLICATION if the index refers to the group applications, None otherwise.

Parameters
index – a QModelIndex instance

Returns

`mimeData`(*indices*)

Overwritten from QAbstractItemModel, provide the mime data for copy/pasting (note that this doesn't work across processes.

`mimeTypes`()

Overwritten from QAbstractItemModel, provide a mime type for copy/pasting

`nodeAdded`(*subConfig*, *node*)

This slot is called when a node is added to a subConfig

Parameters

- **subConfig** – a SubConfiguration instance
- **node** – the node name.

Returns

`nodeDeleted`(*subConfig*, *node*)

This slot is called when a node is removed from a subConfig

Parameters

- **subConfig** – a SubConfiguration instance
- **node** – the node name.

Returns

`nodeRenamed`(*subConfig*, *oldName*, *newName*)

This slot is called when a node is renamed in a subConfig

Parameters

- **subConfig** – a SubConfiguration instance
- **oldName** – the original name.
- **newName** – the new name

Returns

`parent`(*index*)

Returns the indice's parent according to QAbstractItemModel convetions.

Parameters
index – a QModelIndex instance.

Returns

`propertyAdded`(*parentItem*, *propColl*, *name*)

Slot called when a property was added.

Parameters

- **parentItem** – a NodeItem instance
- **propColl** – the PropertyCollection instance
- **name** – the name of the new property.

Returns

propertyChanged(*item*, *propColl*, *name*)

Slot called when a property has been changed.

Parameters

- **item** – a PropertyItem instance
- **propColl** – a PropertyCollection instance
- **name** – the name of the changed property

Returns

propertyRemoved(*parentItem*, *propColl*, *name*)

Slot called when a property has been removed.

Parameters

- **parentItem** – a NodeItem instance
- **propColl** – a PropertyCollection instance
- **name** – the name of the removed property

Returns

rowCount(*parent*)

Returns the number of children of the given model index

Parameters
parent – a QModelIndex instance

Returns

setData(*index*, *value*, *role*)

Generic data modification (see QAbstractItemModel for details)

Parameters

- **index** – a QModelIndex instance
- **value** – the new value
- **role** – the role to be changed

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("ConfigurationModel" inherits "QAbstractItemModel": Methods: #77 type=Slot, signature=subConfigAdded(PyObject), parameters=PyObject #78 type=Slot, signature=subConfigRenamed(PyObject), parameters=PyObject #79 type=Slot, signature=subConfigRemoved(QString,int), parameters=QString, int #80 type=Slot, signature=appActivated(QString,PyObject), parameters=QString, PyObject)

subConfigAdded(*subConfig*)

This slot is called when a subconfig is added to the configuration instance. It inserts a row as needed.

Parameters
subConfig – a SubConfiguration instance

Returns

subConfigByNameAndType(*name*, *sctype*)

Returns a SubConfiguration instance, given its name and type

Parameters

- **name** – the name as a string
- **sctype** – either CONFIG_TYPE_APPLICATION or CONFIG_TYPE_COMPOSITE

Returns
a SubConfiguration instance

subConfigRemoved(*name*, *sctype*)

This slot is called when a subconfig is removed from the configuration

:param name. the name of the removed subConfig :param sctype: the type (either CONFIG_TYPE_APPLICATION or CONFIG_TYPE_COMPOSITE) :return:

subConfigRenamed(*subConfig*, *oldName*)

This slot is called when a subconfig is renamed in the configuration instance.

Parameters

- **subConfig** – a SubConfiguration instance
- **oldName** – the old name of this subConfig

Returns

variableAddedOrChanged(*parentItem*, *key*, *variables*)

Slot which is called when variables of parentItem are added or changed.

Parameters

- **parentItem** – an instance of VariableContent
- **key** – the key which is changed or added
- **variables** – the Variables instances managing the variables of parentItem

variableDeleted(*vitem*, *key*)

Slot which is called when variables of vitem are deleted.

Parameters

- **vitem** – an instance of VariableContent
- **key** – the key which is deleted

class nexxT.services.SrvConfiguration.**MVCConfigurationBase**(*configuration*)

Bases: QObject

Base class for the configuration service. Might be used as a console service. The GUI service inherits from this class.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

activate()

Call this slot to activate the current application

Returns

appActivated(*name*, *app*)

Called when the application is activated. This is overwritten in the GUI class.

Parameters

- **name** – the application name
- **app** – An ActiveApplication instance.

Returns

changeActiveApp(*app*)

Call this slot to activate an application

Parameters
app – can be either an Application instance or the name of an application

Returns

configuration()

Return this service's configuration object

Returns
a Configuration instance

deactivate()

Call this slot to deactivate the current application

Returns

loadConfig(*cfgFileName*)

Call this slot to load a configuration

Parameters
cfgFileName – the filename of the configuration

Returns

newConfig(*cfgFileName*)

Call this slot to create a new configuration

Parameters
cfgFileName – the filename of the configuration

Returns

reload()

Reloads all python modules of the current configuration.

Similar to close(), open() and loading the currently actuve sequence.

saveConfig()

Call this slot to save the configuration (the gui state in the config will not be changed)

Returns

saveConfigAs(*filename*)

Call this slot to save the configuration

Returns

saveConfigWithGuiState()

Call this slot to save the configuration and synchronize the gui state.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("MVCConfigurationBase" inherits "QObject": Methods: #5 type=Slot, signature=activate() #6 type=Slot, signature=deactivate() #7 type=Slot, signature=loadConfig(QString), parameters=QString #8 type=Slot, signature=saveConfig() #9 type=Slot, signature=saveConfigWithGuiState() #10 type=Slot, signature=saveConfigAs(QString), parameters=QString #11 type=Slot, signature=newConfig(QString), parameters=QString #12 type=Slot, signature=changeActiveApp(QString), parameters=QString)

nexxT.services.SrvPlaybackControl module

This module provides the playback control console service for the nexxT framework.

class nexxT.services.SrvPlaybackControl.**MVCPlaybackControlBase**

Bases: QObject

Base class for interacting with playback controller, usually this is connected to a harddisk player. This class provides the functions setupConnections and removeConnections which can be called by filters to register and deregister themselfs as a playback device.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

removeConnections(*playbackDevice*)

unregisters the given playbackDevice and disconnects all. It is intended that this function is called in the onClose(...) method of a filter.

Parameters
playbackDevice – the playback device to be unregistered.

Returns
None

setupConnections(*playbackDevice*, *nameFilters*)

Sets up signal/slot connections between this view/controller instance and the given playbackDevice. This function is thread safe and shall be called by a direct QT connection. It is intended, that this function is called in the onOpen(...) method of a filter. It expects playbackDevice to provide the following slots:

- startPlayback() (starts generating DataSamples)
- pausePlayback() (pause mode, stop generating DataSamples)
- **stepForward(QString stream) (optional; in case given, a single step operation shall be performed.**
  if stream is not None, the playback shall stop when receiving the next data sample of stream; otherwise the playback shall proceed to the next data sample of any stream)
- stepBackward(QString stream) (optional; see stepForward)
- seekBeginning(QString stream) (optional; goes to the beginning of the sequence)
- seekEnd() (optional; goes to the end of the stream)
- seekTime(qint64) (optional; goes to the specified time stamp)
- setSequence(QString) (optional; opens the given sequence)

- setTimeFactor(float) (optional; sets the playback time factor, factor > 0)

It expects playbackDevice to provide the following signals (all signals are optional):

- sequenceOpened(QString filename, qint64 begin, qint64 end, QStringList streams)
- currentTimestampChanged(qint64 currentTime)
- playbackStarted()
- playbackPaused()
- timeRatioChanged(float)

Parameters

- **playbackDevice** – a QObject providing the aforementioned signals and slots
- **nameFilters** – a QStringList providing information about suported fileextensions (e.g. [".*avi*", ".mp4"])

Returns

```
staticMetaObject = PySide6.QtCore.QMetaObject("MVCPlaybackControlBase" inherits
"QObject":  Methods:  #5 type=Signal, signature=_startPlayback() #6 type=Signal,
signature=_pausePlayback() #7 type=Signal, signature=_stepForward(QString),
parameters=QString #8 type=Signal, signature=_stepBackward(QString),
parameters=QString #9 type=Signal, signature=_seekBeginning() #10 type=Signal,
signature=_seekEnd() #11 type=Signal, signature=_seekTime(qlonglong),
parameters=qlonglong #12 type=Signal, signature=_setSequence(PyObject),
parameters=PyObject #13 type=Signal, signature=_setTimeFactor(double),
parameters=double #14 type=Slot, signature=setupConnections(QObject*,QStringList),
parameters=QObject*, QStringList #15 type=Slot,
signature=removeConnections(QObject*), parameters=QObject* )
```

class nexxT.services.SrvPlaybackControl.**PlaybackControlConsole**(*config*)

Bases: *MVCPlaybackControlBase*

Console service for playback control. Basically inverts the signals and slots and provides an API for scripting. The GUI service inherits from this class, so that the GUI can also be scripted in the same way.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

currentTimestampChanged

getSequence()

Returns the currently active sequence.

pausePlayback()

Pause playback

Returns

playbackPaused

playbackStarted

seekBeginning()

Seek to the beginning of the file.

Returns

seekEnd()

Seek to the end of the file.

Returns

seekTime(*timestampNS*)

Seek to the specified time

Parameters
timestampNS – the timestamp in nanoseconds

Returns

sequenceOpened

setSequence(*file*)

Set the sequence to be played.

Parameters
file – a string containing a filename

Returns

setTimeFactor(*factor*)

Set the time factor to be used.

Parameters
factor – a float containing the factor.

Returns

startPlayback()

Start playback

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("PlaybackControlConsole" inherits "MVCPlaybackControlBase": Methods: #16 type=Signal, signature=supportedFeaturesChanged(PyObject,PyObject), parameters=PyObject, PyObject #17 type=Signal, signature=sequenceOpened(QString,qlonglong,qlonglong,PyObject), parameters=QString, qlonglong, qlonglong, PyObject #18 type=Signal, signature=currentTimestampChanged(qlonglong), parameters=qlonglong #19 type=Signal, signature=playbackStarted() #20 type=Signal, signature=playbackPaused() #21 type=Signal, signature=timeRatioChanged(double), parameters=double)

stepBackward(*stream*)

Step one frame backward in the given stream (might be None)

Parameters
stream – a string containing the selected stream.

Returns

stepForward(*stream*)

Step one frame forward in the given stream (might be None).

Parameters
stream – a string containing the selected stream.

Returns

supportedFeaturesChanged

timeRatioChanged

class nexxT.services.SrvPlaybackControl.**PlaybackDeviceProxy**(*playbackControl*, *playbackDevice*, *nameFilters*)

Bases: QObject

This class acts as a proxy and is connected to exactly one playback device over QT signals slots for providing thread safety.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None
Initialize self. See help(type(self)) for accurate signature.

currentTimestampChanged

featureSet()
Returns the player device's feature set

Returns
a set of strings

hasControl()
Returns whether this proxy object has control over the player

Returns
true if this proxy object has control

pausePlayback()
Proxy function, checks whether this proxy has control and emits the signal if necessary

playbackPaused

playbackStarted

seekBeginning()
Proxy function, checks whether this proxy has control and emits the signal if necessary

seekEnd()
Proxy function, checks whether this proxy has control and emits the signal if necessary

seekTime(*timestampNS*)
Proxy function, checks whether this proxy has control and emits the signal if necessary

Parameters
timestampNS – the timestamp in nanoseconds

sequenceOpened

setSequence(*filename*)
Proxy function, checks whether filename matches the filter's name filter list and takes control if necessary.

Parameters
filename – a string instance or None

setTimeFactor(*factor*)

Proxy function, checks whether this proxy has control and emits the signal if necessary

Parameters
factor – time factor as a float

startPlayback()

Proxy function, checks whether this proxy has control and emits the signal if necessary

staticMetaObject = PySide6.QtCore.QMetaObject("PlaybackDeviceProxy" inherits "QObject": Methods: #5 type=Signal, signature=_startPlayback() #6 type=Signal, signature=_pausePlayback() #7 type=Signal, signature=_stepForward(QString), parameters=QString #8 type=Signal, signature=_stepBackward(QString), parameters=QString #9 type=Signal, signature=_seekBeginning() #10 type=Signal, signature=_seekEnd() #11 type=Signal, signature=_seekTime(qlonglong), parameters=qlonglong #12 type=Signal, signature=_setSequence(PyObject), parameters=PyObject #13 type=Signal, signature=_setTimeFactor(double), parameters=double #14 type=Signal, signature=sequenceOpened(QString,qlonglong,qlonglong,PyObject), parameters=QString, qlonglong, qlonglong, PyObject #15 type=Signal, signature=currentTimestampChanged(qlonglong), parameters=qlonglong #16 type=Signal, signature=playbackStarted() #17 type=Signal, signature=playbackPaused() #18 type=Signal, signature=timeRatioChanged(double), parameters=double)

stepBackward(*stream*)

Proxy function, checks whether this proxy has control and emits the signal if necessary

stepForward(*stream*)

Proxy function, checks whether this proxy has control and emits the signal if necessary

timeRatioChanged

nexxT.services.SrvProfiling module

This module provides the profiling service for nexxT, responsible for generating profiling measurements.

class nexxT.services.SrvProfiling.**PortProfiling**

Bases: object

Simple helper class for storing profiling time points of a single port.

__init__()

getSpans()

Returns the profiling time points in a list.

Returns
list of tuples containing nanosecond time points.

pause(*timeNs*)

Called when the corresponding item is paused (another item may be started).

Parameters
timeNs – the time point, given in nanoseconds.

Returns

start(*timeNs*)

Called when the corresponding item is started.

Parameters
timeNs – the time point, given in nanoseconds.

Returns

stop(*timeNs*)

Called when the corresponding item is finished. The profiling information will be added to the history.

Parameters
timeNs – the time point, given in nanoseconds.

Returns

unpause(*timeNs*)

Called when the corresponding item is unpaused.

Parameters
timeNs – the time point, given in nanoseconds.

Returns

class nexxT.services.SrvProfiling.**ProfilingService**

Bases: QObject

This class provides a profiling service for the nexxT framework.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

afterPortDataChanged(*portname*)

This slot is called after calling onPortDataChanged.

Parameters

- **portname** – the fully qualified name of the port
- **timeNs** – the timestamp

Returns

beforePortDataChanged(*portname*)

This slot is called before calling onPortDataChanged.

Parameters

- **portname** – the fully qualified name of the port
- **timeNs** – the timestamp

Returns

deregisterThread()

This slot shall be called from each deactivated nexxT thread with a direct connection

Returns

loadDataUpdated

registerThread()

This slot shall be called from each activated nexxT thread with a direct connection.

Returns

setLoadMonitorEnabled(*enabled*)

Enables / disables load monitoring

Parameters
enabled – boolean

Returns

setPortProfilingEnabled(*enabled*)

Enables / disables port profiling

Parameters
enabled – boolean

Returns

spanDataUpdated

startTimers

staticMetaObject = PySide6.QtCore.QMetaObject("ProfilingService" inherits "QObject": Methods: #5 type=Signal, signature=loadDataUpdated(QString,QByteArray,QByteArray), parameters=QString, QByteArray, QByteArray #6 type=Signal, signature=spanDataUpdated(QString,QString,QByteArray), parameters=QString, QString, QByteArray #7 type=Signal, signature=threadDeregistered(QString), parameters=QString #8 type=Signal, signature=stopTimers() #9 type=Signal, signature=startTimers() #10 type=Slot, signature=registerThread() #11 type=Slot, signature=deregisterThread() #12 type=Slot, signature=_generateRecord() #13 type=Slot, signature=beforePortDataChanged(QString), parameters=QString #14 type=Slot, signature=afterPortDataChanged(QString), parameters=QString)

stopTimers

threadDeregistered

class nexxT.services.SrvProfiling.**ProfilingServiceDummy**

Bases: QObject

This class can be used as a replacement for the ProfilingService which provides the same interface.

afterPortDataChanged(*portname*)

dummy implementation

Parameters
portname – name of the port

beforePortDataChanged(*portname*)

dummy implementation

Parameters
portname – name of the port

deregisterThread()

dummy implementation

registerThread()

dummy implementation

staticMetaObject = PySide6.QtCore.QMetaObject("ProfilingServiceDummy" inherits "QObject": Methods: #5 type=Slot, signature=registerThread() #6 type=Slot, signature=deregisterThread() #7 type=Slot, signature=beforePortDataChanged(QString), parameters=QString #8 type=Slot, signature=afterPortDataChanged(QString), parameters=QString)

class nexxT.services.SrvProfiling.ThreadSpecificProfItem

Bases: object

This class contains all profiling items of a specific thread.

THREAD_PROFILING_PERIOD_SEC = 0.3

THREAD_PROFILING_TOTAL_TIME = 60

__init__()

cancel()

Cancel profiling on user-request and reset the corresponding data.

Returns

getLoad()

Returns the load measurements.

Returns

list of 2-tuples (time_nano_seconds, load_ratio)

getSpans()

Get the current port profiling data.

Returns

dict mapping thread names to lists of tuples with nano-second time points.

registerPortChangeFinished(*portname*, *timeNs*)

Called when the onPortDataChanged function has finished.

Parameters

- **portname** – the full-qualified port name
- **timeNs** – the time in nano-seconds

Returns

registerPortChangeStarted(*portname*, *timeNs*)

Called when starting the onPortDataChanged function.

Parameters

- **portname** – the full-qualified port name
- **timeNs** – the time in nano-seconds

Returns

update()

Updates the load profiling.

Returns

nexxT.services.SrvRecordingControl module

This module provides the recording control console service for the nexxT framework.

class nexxT.services.SrvRecordingControl.**MVCRecordingControlBase**(*config*)

Bases: QObject

Base class for interacting with playback controller, usually this is connected to a harddisk player.

__init__(*self*, *parent: PySide6.QtCore.QObject | None = None*) → None

Initialize self. See help(type(self)) for accurate signature.

notifyError

removeConnections(*recordingDevice*)

unregisters the given recordingDevice and disconnects all. It is intended that this function is called in the onStop(...) method of a filter.

Parameters
recordingDevice – the recording device to be unregistered.

Returns
None

setupConnections(*recordingDevice*)

Sets up signal/slot connections between this view/controller instance and the given playbackDevice. This function is thread safe and shall be called by a direct QT connection. It is intended, that this function is called in the onStart(...) method of a filter. It expects recordingDevice to provide the following slots:

- startRecording(str directory) (starts recording DataSamples)
- stopRecording() (stop the recording)

It expects recordingDevice to provide the following signals (all signals are optional):

- statusUpdate(str file, float lengthInSeconds, int bytesWritten)
- notifyError(str errorDescription)

Parameters
recordingDevice – a QObject providing the aforementioned signals and slots

Returns

startRecording(*directory*)

Starts the recording in the given directory

Parameters
directory – The directory where to store the recordings

Returns

stateChanged(*state*)

Stops the recording when application is stopped.

Parameters
state – the new filter state

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("MVCRecordingControlBase" inherits "QObject": Methods: #5 type=Signal, signature=_startRecording(QString), parameters=QString #6 type=Signal, signature=_stopRecording() #7 type=Signal, signature=statusUpdate(QObject*,QString,double,qlonglong), parameters=QObject*, QString, double, qlonglong #8 type=Signal, signature=notifyError(QObject*,QString), parameters=QObject*, QString #9 type=Slot, signature=setupConnections(QObject*), parameters=QObject* #10 type=Slot, signature=removeConnections(QObject*), parameters=QObject* #11 type=Slot, signature=_statusUpdate(QString,double,qlonglong), parameters=QString, double, qlonglong #12 type=Slot, signature=stateChanged(int), parameters=int)

statusUpdate

stopRecording()

Stops the recording.

Returns

3.1.1.3 nexxT.filters package

Submodules

nexxT.filters.GenericReader module

This module provides a generic reader which can be inherited to use new data formats inside nexxT.

class nexxT.filters.GenericReader.**GenericReader**(*env*)

Bases: *Filter*

Generic harddisk reader which can be used as base class for implementing readers for custom file formats. To create a new input file reader, inherit from this class and reimplement getNameFilter(...) and openFile(...). openFile(...) shall return an instance of an implementation of the interface GenericReaderFile.

See *nexxT.filters.hdf5.Hdf5Reader* for an example.

__init__(*env*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

currentTimestampChanged

getNameFilter()

Returns the name filter associated with the input files.

Returns
a list of strings, e.g. [".*h5*", ".hdf5"]

onClose()

overloaded from Filter

Returns

onOpen()
overloaded from Filter

Returns

onStart()
overloaded from Filter

Returns

onStop()
overloaded from Filter

Returns

onSuggestDynamicPorts()
overloaded from Filter

Returns

openFile(*filename*)
Opens the given file and return an instance of GenericReaderFile.

Returns
an instance of GenericReaderFile

pausePlayback()
slot called when the playback shall be paused

Returns

playbackPaused

playbackStarted

seekBeginning()
slot called to go to the beginning of the file

Returns

seekEnd()
slot called to go to the end of the file

Returns

seekTime(*timestamp*)
slot called to go to the specified time

Parameters
timestamp – a timestamp in nanosecond resolution

Returns

sequenceOpened

setSequence(*filename*)
slot called to set the sequence file name

Parameters
filename – a string instance

Returns

setTimeFactor(*factor*)

slot called to set the time factor

Parameters
factor – a float

Returns

startPlayback()

slot called when the playback shall be started

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("GenericReader" inherits "Filter": Methods: #5 type=Signal, signature=playbackStarted() #6 type=Signal, signature=playbackPaused() #7 type=Signal, signature=sequenceOpened(QString,qlonglong,qlonglong,QVariantList), parameters=QString, qlonglong, qlonglong, QVariantList #8 type=Signal, signature=currentTimestampChanged(qlonglong), parameters=qlonglong #9 type=Signal, signature=timeRatioChanged(double), parameters=double)

stepBackward(*stream*)

slot called to step one frame in stream backward

Parameters
stream – a string instance or None (all streams are selected)

Returns

stepForward(*stream*)

slot called to step one frame in stream forward

Parameters
stream – a string instance or None (all streams are selected)

Returns

timeRatioChanged

class nexxT.filters.GenericReader.**GenericReaderFile**

Bases: object

Interface for adaptations of new file formats.

For supporting new file formats, inherit from this class and overwrite all of the methods listed here. The constructor of the inherited class usually takes a filename argument. Inherit also from GenericReader to provide a new Filter class and overwrite the methods getNameFilter and openFile, which returns an instance of the GenericReaderFile implementation.

See *nexxT.filters.hdf5.Hdf5File* and *nexxT.filters.hdf5.Hdf5Reader* for an example.

allStreams()

Returns the streams in this file.

Returns
a list of strings

close()

Closes the file.

Returns

getNumberOfSamples(*stream*)

Returns the number of samples in the given stream

Parameters
stream – the name of the stream as a string

Returns
the number of samples in the stream

getRcvTimestamp(*stream*, *streamIdx*)

Returns the recevie timestamp of the given (stream, streamIdx) sample. The default implementation uses readSample(...). It may be replaced by a more efficient implementation.

Parameters

- **stream** – the name of the stream as a string
- **streamIdx** – the stream index as an integer

Returns
the timestamp as an integer (see also getTimestampResolution)

getTimestampResolution()

Returns the resolution of the timestamps in ticks per second.

Returns
ticks per second as an integer

readSample(*stream*, *streamIdx*)

Returns the referenced sample as a tuple (content, dataType, dataTimestamp, rcvTimestamp).

Parameters

- **stream** – the stream
- **idx** – the index of the sample in the stream

Returns
(content: QByteArray, dataType: str, dataTimestamp: int, receiveTimestamp: int)

nexxT.filters.hdf5 module

This module provides a generic disk reader and writer based on HDF5. To use it, you have to enable the "HDF5" feature during installation, i.e. pip install nexxT[HDF5]

class nexxT.filters.hdf5.**Hdf5File**(*filename*)

Bases: *GenericReaderFile*

Adaptation of hdf5 file format

__init__(*filename*)

allStreams()

Returns the streams in this file.

Returns
a list of strings

close()

Closes the file.

Returns

getNumberOfSamples(*stream*)

Returns the number of samples in the given stream

Parameters
stream – the name of the stream as a string

Returns
the number of samples in the stream

getRcvTimestamp(*stream*, *streamIdx*)

Returns the recevie timestamp of the given (stream, streamIdx) sample. The default implementation uses readSample(...). It may be replaced by a more efficient implementation.

Parameters

- **stream** – the name of the stream as a string
- **streamIdx** – the stream index as an integer

Returns
the timestamp as an integer (see also getTimestampResolution)

getTimestampResolution()

Returns the resolution of the timestamps in ticks per second.

Returns
ticks per second as an integer

readSample(*stream*, *streamIdx*)

Returns the referenced sample as a tuple (content, dataType, dataTimestamp, rcvTimestamp).

Parameters

- **stream** – the stream
- **idx** – the index of the sample in the stream

Returns
(content: QByteArray, dataType: str, dataTimestamp: int, receiveTimestamp: int)

class nexxT.filters.hdf5.**Hdf5Reader**(*env*)

Bases: *GenericReader*

Reader for the nexxT default file format based on hdf5.

getNameFilter()

Returns the name filter associated with the input files.

Returns
a list of strings, e.g. [".*h5", "*.hdf5"]

openFile(*filename*)

Opens the given file and return an instance of GenericReaderFile.

Returns
an instance of GenericReaderFile

staticMetaObject = PySide6.QtCore.QMetaObject("Hdf5Reader" inherits "GenericReader":
)

class nexxT.filters.hdf5.**Hdf5Writer**(*env*)

Bases: *Filter*

Generic nexxT filter for writing HDF5 files.

__init__(*env*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

onInit()

This function can be overwritten for performing initialization tasks related to dynamic ports.

Returns
None

onPortDataChanged(*port*)

Called when new data arrives at a port.

Parameters
port – the port where the new data is available.

Returns

onStart()

Registers itself to the recording control service

Returns

onStop()

De-registers itself from the recording control service

Returns

startRecording(*directory*)

Called on a recording start event.

Parameters
directory – the directory where the recording is expected to be created.

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("Hdf5Writer" inherits "Filter":
Methods: #5 type=Signal, signature=statusUpdate(QString,double,qlonglong),
parameters=QString, double, qlonglong)

statusUpdate

stopRecording()

Called on a recording stop event.

Parameters
directory – the directory where the recording is expected to be created.

Returns

3.1.1.4 nexxT.examples package

define FilterSurrogates for binary filters.

nexxT.examples.**AviReader = <nexxT.interface.Filters.FilterSurrogate object>**

Filter surrogate for the VideoPlaybackDevice class which is defined in a packaged shared object "test_plugins".

nexxT.examples.**CameraGrabber = <nexxT.interface.Filters.FilterSurrogate object>**

Filter surrogate for the CameraGrabber class which is defined in a packaged shared object "test_plugins".

Subpackages

nexxT.examples.framework package

Submodules

nexxT.examples.framework.ImageBlur module

This module defines the filter ImageBlur as a little showcase for a long-running algorithm

class nexxT.examples.framework.ImageBlur.**ImageBlur**(*env*)

Bases: *Filter*

A filter providing the parameter kernelSize. A box filter of this size is applied to the input image and the output is transferred over the output port.

__init__(*env*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

onPortDataChanged(*port*)

Overloaded from Filter base class. The method is called whenever new data arrives at an input port. :param port: the port which has new data. :return:

staticMetaObject = PySide6.QtCore.QMetaObject("ImageBlur" inherits "Filter":)

nexxT.examples.framework.ImageBlur.**boxFilter**(*img*, *kernelSize*)

2D box filter operating on single or multichannel input images :param img: the image as a numpy array :param kernel_size: the size of the filter (must be odd) :return: the filtered image (same type as input image)

nexxT.examples.framework.ImageData module

This is the image data type used in the example files. The image type is selected for simplicty and for minimal external dependencies. In the C++ part of the API, the the ImageHeader ctypes structure is mapped to a corresponding C structure. nexxT itself only knows about QByteArray's for data transport, the contents must be defined by the user.

class nexxT.examples.framework.ImageData.**ImageHeader**

Bases: Structure

The QByteArray starts with this header and the rest of the array is the actual image data according to the format given here.

format

Structure/Union member

height

Structure/Union member

lineInc

Structure/Union member

width

Structure/Union member

nexxT.examples.framework.ImageData.**byteArrayToNumpy**(*qByteArray*)

Interpret the input instance as an image and convert that to a numpy array. If the alignment is ok, then this operation is a zero-copy operation, otherwise one copy is made.

Parameters
qByteArray – a QByteArray instance

Returns
a numpy instance

nexxT.examples.framework.ImageData.**numpyToByteArray**(*img*)

Convert a numpy image to the corresponding QByteArray (and make a copy).

Parameters
img – a numpy array instance with 2 or 3 dimensions

Returns
a QByteArray instance

nexxT.examples.framework.ImageView module

This viewer is a showcase to see how to write a viewer for nexxT. Note that there are multiple possibilities to view images, and this one might actually not be the best option. You can use python toolkits such as pyqtgraph, matpllotlibs or others supporting a QT5 backend.

class nexxT.examples.framework.ImageView.**DisplayWidget**(*parent=None*)

Bases: QWidget

The widget actually displaying the image. It has a scale parameter for setting the scale factor of the drawn image. The widget is created from the Filter instance. Both are QObjects and both have to run in the main (=GUI) thread of the QApplication.

__init__(*self*, *parent: PySide6.QtWidgets.QWidget | None = None*, *f: PySide6.QtCore.Qt.WindowType = Default(Qt.WindowFlags)*) → None

Initialize self. See help(type(self)) for accurate signature.

checkSize()

make sure that the minimum size is consistent with the shown image.

Returns

paintEvent(*paintEvent*)

The paint event actually drawing the image

Parameters
paintEvent – a QPaintEvent instance

Returns

setData(*data*)

Called when new data arrives. This function converts the numpy array to a QImage which can then be drawn with a QT painter.

Parameters
data – the image as a numpy array

Returns

setScale(*scale*)

Set the scale factor

Parameters
scale – a floating point (<1: size reduction)

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("DisplayWidget" inherits "QWidget":)

class nexxT.examples.framework.ImageView.**ImageView**(*env*)

Bases: *Filter*

This is the filter receiving the image data to be drawn. It has to be run in the GUI thread.

__init__(*env*)

Filter Constructor.

Parameters

- **dynInPortsSupported** – Flag whether this filter supports dynamic input ports
- **dynOutPortsSupported** – Flag whether this filter supports dynamic output ports
- **environment** – FilterEnvironment instance which shall be passed through from the filter constructor.

interpretAndUpdate()

The deferred update method, called from the MainWindow service at user-defined framerate.

onClose()

Inverse of onOpen

Returns

onOpen()

Now we can create the widget.

Returns

onPortDataChanged(*port*)

Notification of new data.

Parameters
port – the port where the data arrived.

Returns

propChanged(*propColl*, *name*)

Slot called whenever a property of this filter has changed.

Parameters

- **pc** – the PropertyCollection instance of this filter
- **name** – the name of the changed parameter

Returns

staticMetaObject = PySide6.QtCore.QMetaObject("ImageView" inherits "Filter":)

3.1.1.5 nexxT.Qt module

3.1.1.6 nexxT.QtMetaPackage module

This module provides a QT meta package such that we are able to write "from nexxT.Qt.QtWidgets import QWidget" and nexxT.Qt will serve as an alias for PySide6

It is loosly based on this tutorial: https://dev.to/dangerontheranger/dependency-injection-with-import-hooks-in-python-3-5hap

class nexxT.QtMetaPackage.QtFinder(*loader*)

Bases: MetaPathFinder

The meta path finder which will be added to sys.meta_path

__init__(*loader*)

find_spec(*fullname*, *path*, *target=None*)

Attempt to locate the requested module

Parameters

- **fullname** – the fully-qualified name of the module,
- **path** – set to __path__ for sub-modules/packages, or None otherwise.
- **target** – can be a module object, but is unused here.

class nexxT.QtMetaPackage.QtLoader(*prefix*, *qtlib*)

Bases: Loader

The actual loader which maps PySide modules to nexxT.Qt. The approach is similar to executing a `from PySide6 import *` statement in a real Qt.py module, but it is dynamically and prevents from loading unused QT modules, like QtMultimedia, etc.

__init__(*prefix*, *qtlib*)

Constructor.

Parameters

- **prefix** – the prefix for the proxied library (e.g. nexxT.Qt)
- **qtlib** – the PySide library to be used, must be PySide6

create_module(*spec*)

Creates a new module according to spec. It will be empty initially and populated during exec_module(...).

Parameters
spec – A ModuleSpec instance.

exec_module(*module*)

This function is called after create_module and it populates the module's namespace with the corresponding instances from PySideX.

provides(*fullname*)

Checks whether the queried module can be provided by this laoder or not.

Parameters
fullname – full-qualified module name.

nexxT.QtMetaPackage.**setup**()

3.1.1.7 nexxT.shiboken module

3.1.2 Module contents

Setup the logging here until we have a better place.

nexxT.**setup**()

Sets up the nexxT environment. In particular, provides the cnexxT and the Qt packages under the nexxT nameespace.

3.2 C++ Documentation

3.2.1 DataSamples

class **DataSample**

This class is the C++ variant of *nexxT.interface.DataSamples.DataSample*

Public Functions

DataSample(const QByteArray &content, const QString &datatype, int64_t timestamp)

Constructor, see *nexxT.interface.DataSamples.DataSample.__init__()*

virtual **~DataSample**()

Destructor

QByteArray **getContent**() const

See *nexxT.interface.DataSamples.DataSample.getContent()*

int64_t **getTimestamp**() const

See *nexxT.interface.DataSamples.DataSample.getTimestamp()*

QString **getDatatype**() const

See *nexxT.interface.DataSamples.DataSample.getDatatype()*

Public Static Functions

static *SharedDataSamplePtr* **copy**(const *SharedDataSamplePtr* &src)

See *nexxT.interface.DataSamples.DataSample.copy()*

static *SharedDataSamplePtr* **make_shared**(*DataSample* *sample)

Return a shared pointer referencing the given instance. The ownership of the pointer is transferred to the shared pointer.

static int64_t **currentTime**()

See *nexxT.interface.DataSamples.DataSample.currentTime()*

Public Static Attributes

static const double **TIMESTAMP_RES**

The resolution of the timstamps in [seconds]

typedef QSharedPointer<const *DataSample*> **nexxT::SharedDataSamplePtr**

A typedef for a Datasample handled by a shared pointer.

3.2.2 Filters

class **Filter** : public QObject

This class is the C++ variant of *nexxT.interface.Filters.Filter*

Public Functions

virtual **~Filter**()

Destructor

virtual void **onInit**()

See *nexxT.interface.Filters.Filter.onInit()*

virtual void **onOpen**()

See *nexxT.interface.Filters.Filter.onOpen()*

virtual void **onStart**()

See *nexxT.interface.Filters.Filter.onStart()*

virtual void **onPortDataChanged**(const *InputPortInterface* &inputPort)

See *nexxT.interface.Filters.Filter.onPortDataChanged()*

virtual void **onStop**()

See *nexxT.interface.Filters.Filter.onStop()*

virtual void **onClose**()

See *nexxT.interface.Filters.Filter.onClose()*

virtual void **onDeinit**()

See *nexxT.interface.Filters.Filter.onDeinit()*

virtual QList<QList<QString>> **onSuggestDynamicPorts**()

See *nexxT.interface.Filters.Filter.onSuggestDynamicPorts()*

BaseFilterEnvironment ***environment**() const

See *nexxT.interface.Filters.Filter.environment()*

Public Static Functions

static *SharedFilterPtr* **make_shared**(*Filter* *filter)

Return a shared pointer referencing the given instance. The ownership of the pointer is transferred to the shared pointer.

Protected Functions

Filter(bool dynInPortsSupported, bool dynOutPortsSupported, BaseFilterEnvironment *env)

Constructor, see *nexxT.interface.Filters.Filter.__init__()*

PropertyCollection ***propertyCollection**()

See *nexxT.interface.Filters.Filter.propertyCollection()*

PropertyCollection ***guiState**()

See *nexxT.interface.Filters.Filter.guiState()*

void **addStaticPort**(const *SharedPortPtr* &port)

See *nexxT.interface.Filters.Filter.addStaticPort()*

SharedPortPtr **addStaticOutputPort**(const QString &name)

See *nexxT.interface.Filters.Filter.addStaticOutputPort()*

SharedPortPtr **addStaticInputPort**(const QString &name, int queueSizeSamples = 1, double queueSizeSeconds = -1)

See *nexxT.interface.Filters.Filter.addStaticInputPort()*

void **removeStaticPort**(const *SharedPortPtr* &port)

See *nexxT.interface.Filters.Filter.removeStaticPort()*

PortList **getDynamicInputPorts**()

See *nexxT.interface.Filters.Filter.getDynamicInputPorts()*

PortList **getDynamicOutputPorts**()

See *nexxT.interface.Filters.Filter.getDynamicOutputPorts()*

struct **FilterState**

This struct is the C++ variant of *nexxT.interface.Filters.FilterState*

Public Static Functions

static QString **state2str**(int state)

See *nexxT.interface.Filters.FilterState.state2str()*

Public Static Attributes

static const int **CONSTRUCTING** = 0

During constructor.

static const int **CONSTRUCTED** = 1

After constructor before calling onInit.

static const int **INITIALIZING** = 2

During onInit.

static const int **INITIALIZED** = 3

After onInit before onOpen.

static const int **OPENING** = 4

During onOpen.

static const int **OPENED** = 5

After onOpen before onStart.

static const int **STARTING** = 6

During onStart.

static const int **ACTIVE** = 7

After onStart before onStop.

static const int **STOPPING** = 8

During onStop.

static const int **CLOSING** = 9

During onClose.

static const int **DEINITIALIZING** = 10

During onDeinit.

static const int **DESTRUCTING** = 11

During destructor.

static const int **DESTRUCTED** = 12

After destructor.

typedef QSharedPointer<*Filter*> nexxT::**SharedFilterPtr**

A typedef for a *Filter* instance handled by a shared pointer.

3.2.3 Ports

3.2.3.1 Port

class **Port** : public QObject

This class is the C++ variant of *nexxT.interface.Ports.Port*

Subclassed by *nexxT::InputPortInterface*, *nexxT::OutputPortInterface*

Public Functions

Port(bool dynamic, const QString &name, BaseFilterEnvironment *env)

Constructor, see *nexxT.interface.Ports.Port.__init__()*

virtual **~Port**()

Destructor

bool **dynamic**() const

See *nexxT.interface.Ports.Port.dynamic()*

const QString &**name**() const

See *nexxT.interface.Ports.Port.name()*

void **setName**(const QString &name)

See *nexxT.interface.Ports.Port.setName()*

BaseFilterEnvironment ***environment**() const

See *nexxT.interface.Ports.Port.environment()*

bool **isOutput**() const

See *nexxT.interface.Ports.Port.isOutput()*

bool **isInput**() const

See *nexxT.interface.Ports.Port.isInput()*

virtual *SharedPortPtr* **clone**(BaseFilterEnvironment*) const = 0

See *nexxT.interface.Ports.Port.clone()*

Public Static Functions

static *SharedPortPtr* **make_shared**(*Port* *port)

Return a shared pointer referencing the given instance. The ownership of the pointer is transferred to the shared pointer.

typedef QSharedPointer<*Port*> nexxT::**SharedPortPtr**

A typedef for a *Port* instance handled by a shared pointer.

typedef QList<QSharedPointer<*Port*>> nexxT::**PortList**

A typedef for a list of ports.

3.2.3.2 OutputPortInterface

class **OutputPortInterface** : public nexxT::*Port*

This class is the C++ variant of *nexxT.interface.Ports.OutputPortInterface*.

In contrast to the python version, this class is not abstract but directly implements the functionality.

Public Functions

OutputPortInterface(bool dynamic, const QString &name, BaseFilterEnvironment *env)

Constructor.

See *nexxT.interface.Ports.OutputPort()*.

void **transmit**(const *SharedDataSamplePtr* &sample)

See *nexxT.interface.Ports.OutputPortInterface.transmit()*.

virtual *SharedPortPtr* **clone**(BaseFilterEnvironment*) const

See *nexxT.interface.Ports.OutputPortInterface.clone()*.

Signals

void **transmitSample**(const QSharedPointer<const nexxT::*DataSample*> &sample)

QT signal for transmitting a sample over threads. Note that this signal is not intended to be used directly. Use the transmit method instead.

See *nexxT.interface.Ports.OutputPortInterface.transmitSample*.

Public Static Functions

static void **setupDirectConnection**(const *SharedPortPtr*&, const *SharedPortPtr*&)

Called by the nexxT framework, not intended to be used directly.

static QObject ***setupInterThreadConnection**(const *SharedPortPtr*&, const *SharedPortPtr*&, QThread&, int width)

Called by the nexxT framework, not intended to be used directly.

typedef QSharedPointer<*OutputPortInterface*> nexxT::**SharedOutputPortPtr**

A typedef for a *Port* instance handled by a shared pointer.

3.2.3.3 InputPortInterface

class **InputPortInterface** : public nexxT::*Port*

This class is the C++ variant of *nexxT.interface.Ports.InputPortInterface*.

In contrast to the python version, this class is not abstract but directly implements the functionality.

Public Functions

InputPortInterface(bool dynamic, const QString &name, BaseFilterEnvironment *env, int queueSizeSamples = 1, double queueSizeSeconds = -1.0)

Constructor.

See *nexxT.interface.Ports.InputPort()*.

virtual **~InputPortInterface**()

Destructor

SharedDataSamplePtr **getData**(int delaySamples = 0, double delaySeconds = -1.) const

See *nexxT.interface.Ports.InputPortInterface.getData()*.

virtual *SharedPortPtr* **clone**(BaseFilterEnvironment*) const

See *nexxT.interface.Ports.InputPortInterface.clone()*.

void **setQueueSize**(int queueSizeSamples, double queueSizeSeconds)

See *nexxT.interface.Ports.InputPortInterface.setQueueSize()*.

int **queueSizeSamples**()

See *nexxT.interface.Ports.InputPortInterface.queueSizeSamples()*.

double **queueSizeSeconds**()

See *nexxT.interface.Ports.InputPortInterface.queueSizeSeconds()*.

void **setInterthreadDynamicQueue**(bool enabled)

See *nexxT.interface.Ports.InputPortInterface.setInterthreadDynamicQueue()*.

bool **interthreadDynamicQueue**()

See *nexxT.interface.Ports.InputPortInterface.interthreadDynamicQueue()*.

Public Slots

void **receiveAsync**(const QSharedPointer<const nexxT::*DataSample*> &sample, QSemaphore *semaphore, bool isPending = false)

Called by the nexxT framework, not intended to be used directly.

void **receiveSync**(const QSharedPointer<const nexxT::*DataSample*> &sample)

Called by the nexxT framework, not intended to be used directly.

typedef QSharedPointer<*InputPortInterface*> **nexxT::SharedInputPortPtr**

A typedef for an *InputPortInterface* instance handled by a shared pointer.

A typedef for a *Port* instance handled by a shared pointer.

3.2.4 PropertyCollections

3.2.4.1 PropertyCollection

class **PropertyCollection** : public QObject

This class is the C++ variant of*nexxT.interface.PropertyCollections.PropertyCollection*

Public Functions

PropertyCollection()

Constructor

virtual **~PropertyCollection**()

Destructor

virtual void **defineProperty**(const QString &name, const QVariant &defaultVal, const QString &helpstr)

See*nexxT.interface.PropertyCollections.PropertyCollection.defineProperty()*

Variant with no options and no handler.

virtual void **defineProperty**(const QString &name, const QVariant &defaultVal, const QString &helpstr, const QVariantMap &options)

See*nexxT.interface.PropertyCollections.PropertyCollection.defineProperty()*

Variant with options. C++ example

```
defineProperty("intProp", 1, "a bound integer", {{"min", -4},{"max", 9}});
defineProperty("enumProp", "Hello", "an enum prop", {{"enum", QStringList{"Hello
↪", "World"}}});
```

virtual void **defineProperty**(const QString &name, const QVariant &defaultVal, const QString &helpstr, const *PropertyHandler* *handler)

See*nexxT.interface.PropertyCollections.PropertyCollection.defineProperty()*

Variant with a custom handler.

virtual QVariant **getProperty**(const QString &name) const

See*nexxT.interface.PropertyCollections.PropertyCollection.getProperty()*

Public Slots

virtual void **setProperty**(const QString &name, const QVariant &variant)

Called from the nexxT framework, not intended to be used publicly

virtual QString **evalpath**(const QString &path) const

See*nexxT.interface.PropertyCollections.PropertyCollection.evalpath()*

Signals

void **propertyChanged**(nexxT::*PropertyCollection* *sender, const QString &name)

See*nexxT.interface.PropertyCollections.PropertyCollection.propertyChanged*

3.2.4.2 PropertyHandler

class **PropertyHandler**

This class is the C++ variant of*nexxT.interface.PropertyCollections.PropertyHandler*

Public Functions

PropertyHandler()

Constructor

virtual **~PropertyHandler**()

Destructor

virtual QVariantMap **options**()

See*nexxT.interface.PropertyCollections.PropertyHandler.options()*

The python dict is translated to a QVariantMap in the C++ world.

virtual QVariant **fromConfig**(const QVariant &value)

See*nexxT.interface.PropertyCollections.PropertyHandler.fromConfig()*

virtual QVariant **toConfig**(const QVariant &value)

See*nexxT.interface.PropertyCollections.PropertyHandler.toConfig()*

virtual QVariant **toViewValue**(const QVariant &value)

See*nexxT.interface.PropertyCollections.PropertyHandler.toViewValue()*

virtual QWidget ***createEditor**(QWidget *parent)

See*nexxT.interface.PropertyCollections.PropertyHandler.createEditor()*

virtual void **setEditorData**(QWidget *editor, const QVariant &value)

See*nexxT.interface.PropertyCollections.PropertyHandler.setEditorData()*

virtual QVariant **getEditorData**(QWidget *editor)

See*nexxT.interface.PropertyCollections.PropertyHandler.getEditorData()*

3.2.5 Services

class **Services**

This class is the C++ variant of `nexxT.interface.Services.Services`

Public Functions

Services()

Constructor (intended to be used by the nexxT framework only)

virtual **~Services**()

Destructor

Public Static Functions

static *SharedQObjectPtr* **getService**(const QString &name)

See `nexxT.interface.Services.Services.getService()`

static void **addService**(const QString &name, QObject *service)

See `nexxT.interface.Services.Services.addService()`

static void **removeService**(const QString &name)

See `nexxT.interface.Services.Services.removeService()`

static void **removeAll**()

See `nexxT.interface.Services.Services.removeAll()`

typedef QSharedPointer<QObject> **nexxT::SharedQObjectPtr**

A typedef for a QObject handled by a shared pointer.

In principle it is not really necessary to use a shared pointer to handle QObjects, because of the parent/child ownership principle. However for consistency, the design decision has been made to also wrap the services in a smart pointer just like datasamples, filters and ports.

3.2.6 Logging

NEXXT_LOG_INTERNAL(msg)

Log msg using the nexxT mechanism with priority INTERNAL (lowest)

Parameters

- **msg** – QString instance

NEXXT_LOG_DEBUG(msg)

Log msg using the nexxT mechanism with priority DEBUG

Parameters

- **msg** – QString instance

NEXXT_LOG_INFO(msg)

Log msg using the nexxT mechanism with priority INFO

Parameters

- **msg** – QString instance

NEXXT_LOG_WARN(msg)

Log msg using the nexxT mechanism with priority WARNING

Parameters

- **msg** – QString instance

NEXXT_LOG_ERROR(msg)

Log msg using the nexxT mechanism with priority ERROR

Parameters

- **msg** – QString instance

NEXXT_LOG_CRITICAL(msg)

Log msg using the nexxT mechanism with priority CRITICAL (highest)

Parameters

- **msg** – QString instance

3.2.7 Plugin Definition

NEXXT_PLUGIN_DECLARE_FILTER(classname)

This macro must be present in the Filter's class declaration.

Parameters

- **classname** – the name of the Filter class

NEXXT_PLUGIN_DEFINE_START()

Start to define the plugin introspection code section. To define the introspection code section, one of the .cpp files must call these macros once:

```
NEXXT_PLUGIN_DEFINE_START()
NEXXT_PLUGIN_ADD_FILTER(...)
NEXXT_PLUGIN_ADD_FILTER(...)
NEXXT_PLUGIN_ADD_FILTER(...)
NEXXT_PLUGIN_DEFINE_FINISH()
```

NEXXT_PLUGIN_ADD_FILTER(filtertype)

Add the given filtertype to the plugin

Parameters

- **filtertype** – the name of the Filter class

NEXXT_PLUGIN_DEFINE_FINISH()

Finish the plugin introspection code section

3.3 Command Line Utilities

3.3.1 nexxT-gui

```
usage: nexxT-gui [-h] [-a ACTIVE] [-l LOGFILE]
                 [-v {INTERNAL,DEBUG,INFO,WARN,ERROR,FATAL,CRITICAL}] [-q]
                 [-e EXECPYTHON] [-s EXECSCRIPT] [-t] [-u] [-np] [-g [GUI]]
                 [cfg]

nexxT console application

positional arguments:
  cfg                   .json configuration file of the project to be loaded.

optional arguments:
  -h, --help            show this help message and exit
  -a ACTIVE, --active ACTIVE
                        active application; default: first application in
                        config file.
  -l LOGFILE, --logfile LOGFILE
                        log file location (.db extension will use sqlite).
  -v {INTERNAL,DEBUG,INFO,WARN,ERROR,FATAL,CRITICAL}, --verbosity {INTERNAL,DEBUG,INFO,
↪WARN,ERROR,FATAL,CRITICAL}
                        sets the log verbosity
  -q, --quiet           disble logging to stderr.
  -e EXECPYTHON, --execpython EXECPYTHON
                        execute arbitrary python code given in a string before
                        actually starting the application.
  -s EXECSCRIPT, --execscript EXECSCRIPT
                        execute arbitrary python code given in a file before
                        actually starting the application.
  -t, --single-threaded
                        force using only the main thread
  -u, --disable-unload-heuristic
                        disable unload heuristic for python modules.
  -np, --no-profiling   disable profiling support (only relevant for GUI).
  -g [GUI], --gui [GUI]
                        If true, start nexxT with GUI otherwise use console
                        mode.

The following environment variables have effect on nexxT's behaviour:

NEXXT_VARIANT:
    might be set to 'nonopt' to use the non-optimized variant

NEXXT_DISABLE_CIMPL:
    If set to '1', the nexxT C extensions are replaced by native python modules.

NEXXT_CEXT_PATH:
    Can be set to override the default search path for the nexxT C extension.

NEXXT_BLACKLISTED_PACKAGES:
```

(continues on next page)

(continued from previous page)

```
    List of additional python packages (seperated by a ';') which are not unloaded by␣
↪nexxT when configuration files
    are switched. Use "*" or "__all__" to blacklist all modules.
```

3.3.2 nexxT-console

```
usage: nexxT-console [-h] [-a ACTIVE] [-l LOGFILE]
                     [-v {INTERNAL,DEBUG,INFO,WARN,ERROR,FATAL,CRITICAL}] [-q]
                     [-e EXECPYTHON] [-s EXECSCRIPT] [-t] [-u] [-np]
                     [-g [GUI]]
                     [cfg]

nexxT console application

positional arguments:
  cfg                   .json configuration file of the project to be loaded.

optional arguments:
  -h, --help            show this help message and exit
  -a ACTIVE, --active ACTIVE
                        active application; default: first application in
                        config file.
  -l LOGFILE, --logfile LOGFILE
                        log file location (.db extension will use sqlite).
  -v {INTERNAL,DEBUG,INFO,WARN,ERROR,FATAL,CRITICAL}, --verbosity {INTERNAL,DEBUG,INFO,
↪WARN,ERROR,FATAL,CRITICAL}
                        sets the log verbosity
  -q, --quiet           disble logging to stderr.
  -e EXECPYTHON, --execpython EXECPYTHON
                        execute arbitrary python code given in a string before
                        actually starting the application.
  -s EXECSCRIPT, --execscript EXECSCRIPT
                        execute arbitrary python code given in a file before
                        actually starting the application.
  -t, --single-threaded
                        force using only the main thread
  -u, --disable-unload-heuristic
                        disable unload heuristic for python modules.
  -np, --no-profiling   disable profiling support (only relevant for GUI).
  -g [GUI], --gui [GUI]
                        If true, start nexxT with GUI otherwise use console
                        mode.

The following environment variables have effect on nexxT's behaviour:

NEXXT_VARIANT:
    might be set to 'nonopt' to use the non-optimized variant

NEXXT_DISABLE_CIMPL:
    If set to '1', the nexxT C extensions are replaced by native python modules.
```

(continues on next page)

(continued from previous page)

```
NEXXT_CEXT_PATH:
    Can be set to override the default search path for the nexxT C extension.

NEXXT_BLACKLISTED_PACKAGES:
    List of additional python packages (seperated by a ';') which are not unloaded by␣
↪nexxT when configuration files
    are switched. Use "*" or "__all__" to blacklist all modules.
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W