LOFAR2.0 Station Control

Stichting ASTRON

Jan 26, 2024

CONTENTS:

LOFAR2.0 Station Control is a software stack aimed to monitor, control, and manage a LOFAR2.0 station. In order to do so, it whips up a series of Docker containers, and combines the power of Tango Controls, PyTango, Docker, Grafana, Jupyter Notebook, and many others to provide a rich and powerful experience in using the station.

Full monitoring and control access to the LOFAR2.0 station hardware is provided, by marshalling their rich OPC-UA interfaces. Higher-level logic makes it possible to easily configure and obtain the LOFAR station data products (beamlets, XSTs, SSTs, BSTs) from your local machine using Python, or through one of our provided web interfaces.

Even without having access to any LOFAR2.0 hardware, you can install the full stack on your laptop, and experiment with the software interfaces.

CHAPTER

ONE

INSTALLATION

You will need the following dependencies installed:

- docker
- docker-compose
- git
- make

You start with checking out the source code, f.e. the master branch, as well as the git submodules we use:

```
git clone https://git.astron.nl/lofar2.0/tango.git
cd tango
git submodule init
git submodule update
```

Next, we bootstrap the system. This will build our docker images, start key ones, and load the base configuration. This may take a while:

```
cd docker-compose
make bootstrap
```

If you do have access to LOFAR station hardware, you must upload its configuration to the configuration database. See *Enter your LOFAR2.0 Hardware Configuration*.

Now we are ready to start the other containers:

```
make start
```

and make sure they are all up and running:

```
make status
```

You should see all containers either in the `Up` state or in `Exit 0`. If not, you can inspect why with `docker logs <container>`. Note that the containers will automatically be restarted on failure, and also if you reboot. Stop them explicitly to bring them down (`make stop <container>`).

1.1 Post-boot Initialisation

After bootstrapping, and after a reboot, the software and hardware of the station needs to be explicitly initialised. Note that the docker containers do restart automatically at system boot.

The following commands start all the software devices to control the station hardware, and initialise the hardware with the configured default settings. Go to http://localhost:8888, start a new *Station Control* notebook, and initiate the software boot sequence:

```
# start and initialise the other devices
# go through the full startup sequence
# OFF -> HIBERNATE -> STANDBY -> ON
stationmanager.station_hibernate()
stationmanager.station_standby()
stationmanager.station_on()
```

1.2 Configuration

These sections are optional, to configure specific functionality you may or may not want to use.

CHAPTER TWO

INTERFACES

The station provides the following interfaces accessible through your browser (assuming you run on *localhost*):

Interface	Subsystem	URL	Default credentials
Jupyter Lab	Jupyter	http://localhost:8888	
Monitoring GUIs	Grafana	http://localhost:3000	admin/admin
Alerting	Alerta	http://localhost:8081	admin/alerta
Logs	Kibana	http://localhost:5601	

Futhermore, there are some low-level interfaces:

Interface	Subsystem	URL	Default credentials
PyTango	Tango	tango://localhost:10000	
Prometheus	Prometheus	http://localhost:9090	
TANGO-Grafana Exporter	Python HTTPServer	http://localhost:8000	
ReST API	tango-rest	http://localhost:8080	tango-cs/tango
TangoDB	MariaDB	http://localhost:3306	tango/tango
Archive Database	MariaDB	http://localhost:3307	tango/tango
Log Database	ElasticSearch	http://localhost:9200	

2.1 Monitoring & Control

The main API to control the station is through the Tango Controls API we expose on port 10000, which is most easily accessed using a PyTango client. The Jupyter Lab installation we provide is such a client.

2.1.1 Jupyter Lab

The station offers Jupyter Lab On http://localhost:8888, which allow one to interact with the station, for example to set control points, access monitoring points, or to graph their values.

The notebooks provide some predefined variables, so you don't have to look them up:

```
# Create shortcuts for our devices, if they exist


def OptionalDeviceProxy(device_name: str):
    """Return a DeviceProxy for the given device, or None."""
```

(continues on next page)

(continued from previous page)

```
    try:
        return DeviceProxy(device_name)
    except DevFailed:
        # device is not in database, or otherwise not reachable
        return None


apsct_l0 = OptionalDeviceProxy("STAT/APSCT/L0")
apsct_l1 = OptionalDeviceProxy("STAT/APSCT/L1")
apsct_h0 = OptionalDeviceProxy("STAT/APSCT/H0")
apscts = [apsct_l0, apsct_l1, apsct_h0]

apspu_l0 = OptionalDeviceProxy("STAT/APSPU/L0")
apspu_l1 = OptionalDeviceProxy("STAT/APSPU/L1")
apspu_h0 = OptionalDeviceProxy("STAT/APSPU/H0")
apspus = [apspu_l0, apspu_l1, apspu_h0]

recvl_l0 = OptionalDeviceProxy("STAT/RECVL/L0")
recvl_l1 = OptionalDeviceProxy("STAT/RECVL/L1")
recvh_h0 = OptionalDeviceProxy("STAT/RECVH/H0")
recvs = [recvl_l0, recvl_l1, recvh_h0]

unb2_l0 = OptionalDeviceProxy("STAT/UNB2/L0")
unb2_l1 = OptionalDeviceProxy("STAT/UNB2/L1")
unb2_h0 = OptionalDeviceProxy("STAT/UNB2/H0")
unb2s = [unb2_l0, unb2_l1, unb2_h0]

sdpfirmware_l = OptionalDeviceProxy("STAT/SDPFirmware/LBA")
sdp_l = OptionalDeviceProxy("STAT/SDP/LBA")
bst_l = OptionalDeviceProxy("STAT/BST/LBA")
sst_l = OptionalDeviceProxy("STAT/SST/LBA")
xst_l = OptionalDeviceProxy("STAT/XST/LBA")
beamlet_l = OptionalDeviceProxy("STAT/Beamlet/LBA")
digitalbeam_l = OptionalDeviceProxy("STAT/DigitalBeam/LBA")
antennafield_l = af_l = OptionalDeviceProxy("STAT/AFL/LBA")

sdpfirmware_h = OptionalDeviceProxy("STAT/SDPFirmware/HBA")
sdp_h = OptionalDeviceProxy("STAT/SDP/HBA")
bst_h = OptionalDeviceProxy("STAT/BST/HBA")
sst_h = OptionalDeviceProxy("STAT/SST/HBA")
xst_h = OptionalDeviceProxy("STAT/XST/HBA")
beamlet_h = OptionalDeviceProxy("STAT/Beamlet/HBA")
digitalbeam_h = OptionalDeviceProxy("STAT/DigitalBeam/HBA")
tilebeam_h = OptionalDeviceProxy("STAT/TileBeam/HBA")
antennafield_h = af_h = OptionalDeviceProxy("STAT/AFH/HBA")

sdpfirmware_h0 = OptionalDeviceProxy("STAT/SDPFirmware/HBA0")
sdp_h0 = OptionalDeviceProxy("STAT/SDP/HBA0")
bst_h0 = OptionalDeviceProxy("STAT/BST/HBA0")
sst_h0 = OptionalDeviceProxy("STAT/SST/HBA0")
xst_h0 = OptionalDeviceProxy("STAT/XST/HBA0")
beamlet_h0 = OptionalDeviceProxy("STAT/Beamlet/HBA0")
```

(continues on next page)

(continued from previous page)

```
digitalbeam_h0 = OptionalDeviceProxy("STAT/DigitalBeam/HBA0")
tilebeam_h0 = OptionalDeviceProxy("STAT/TileBeam/HBA0")
antennafield_h0 = af_h0 = OptionalDeviceProxy("STAT/AFH/HBA0")

sdpfirmware_h1 = OptionalDeviceProxy("STAT/SDPFirmware/HBA1")
sdp_h1 = OptionalDeviceProxy("STAT/SDP/HBA1")
bst_h1 = OptionalDeviceProxy("STAT/BST/HBA1")
sst_h1 = OptionalDeviceProxy("STAT/SST/HBA1")
xst_h1 = OptionalDeviceProxy("STAT/XST/HBA1")
beamlet_h1 = OptionalDeviceProxy("STAT/Beamlet/HBA1")
digitalbeam_h1 = OptionalDeviceProxy("STAT/DigitalBeam/HBA1")
tilebeam_h1 = OptionalDeviceProxy("STAT/TileBeam/HBA1")
antennafield_h1 = af_h1 = OptionalDeviceProxy("STAT/AFH/HBA1")

stationmanager = OptionalDeviceProxy("STAT/StationManager/1")
ccd = OptionalDeviceProxy("STAT/CCD/1")
ec = OptionalDeviceProxy("STAT/EC/1")
pcon = OptionalDeviceProxy("STAT/PCON/1")
psoc = OptionalDeviceProxy("STAT/PSOC/1")
docker = OptionalDeviceProxy("STAT/Docker/1")
temperaturemanager = OptionalDeviceProxy("STAT/TemperatureManager/1")
configuration = OptionalDeviceProxy("STAT/Configuration/1")

# Put them in a list in case one wants to iterate
devices = (
    [
        stationmanager,
        ccd,
        ec,
        pcon,
        psoc,
        docker,
        temperaturemanager,
        configuration,
        sdpfirmware_l,
        sdp_l,
        bst_l,
        sst_l,
        xst_l,
        beamlet_l,
        digitalbeam_l,
        af_l,
        sdpfirmware_h,
        sdp_h,
        bst_h,
        sst_h,
        xst_h,
        beamlet_h,
        digitalbeam_h,
        tilebeam_h,
        af_h,
        sdpfirmware_h0,
```

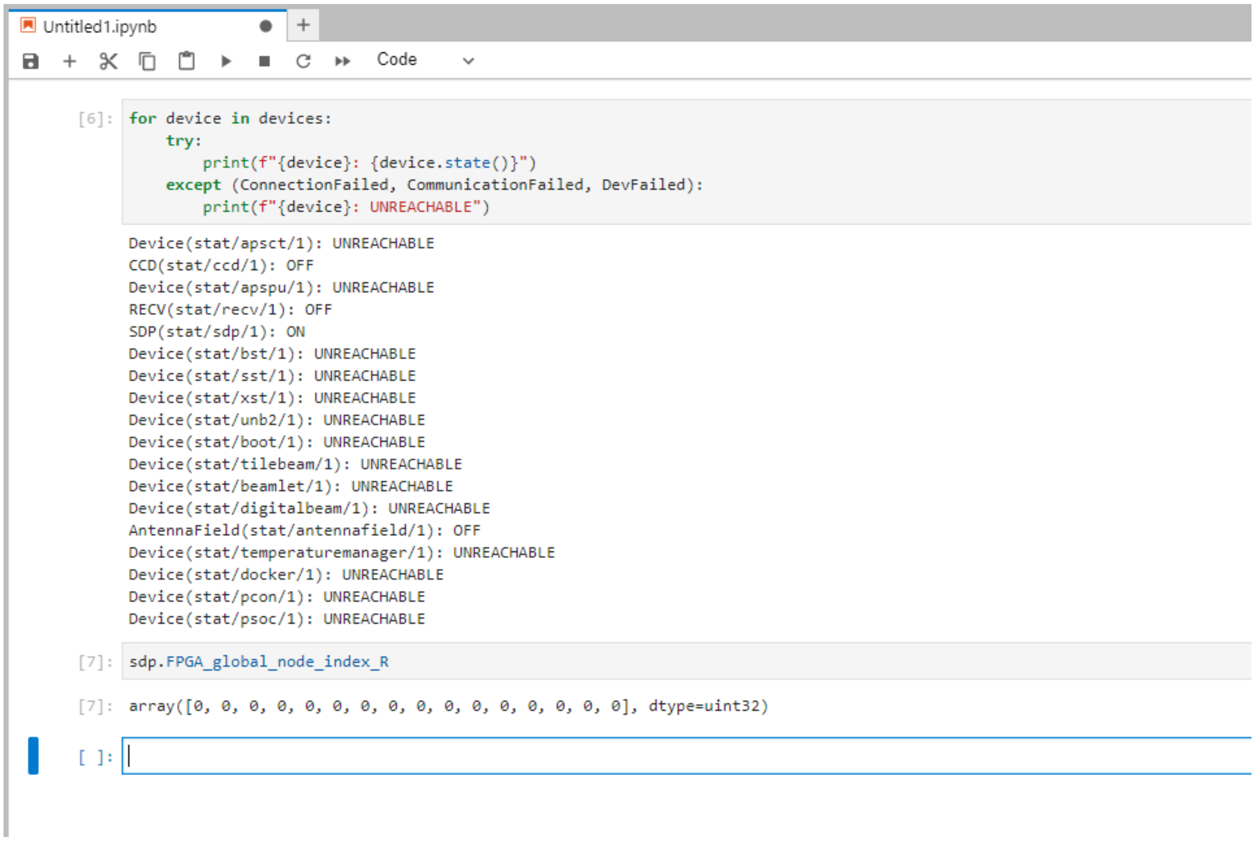
(continues on next page)

(continued from previous page)

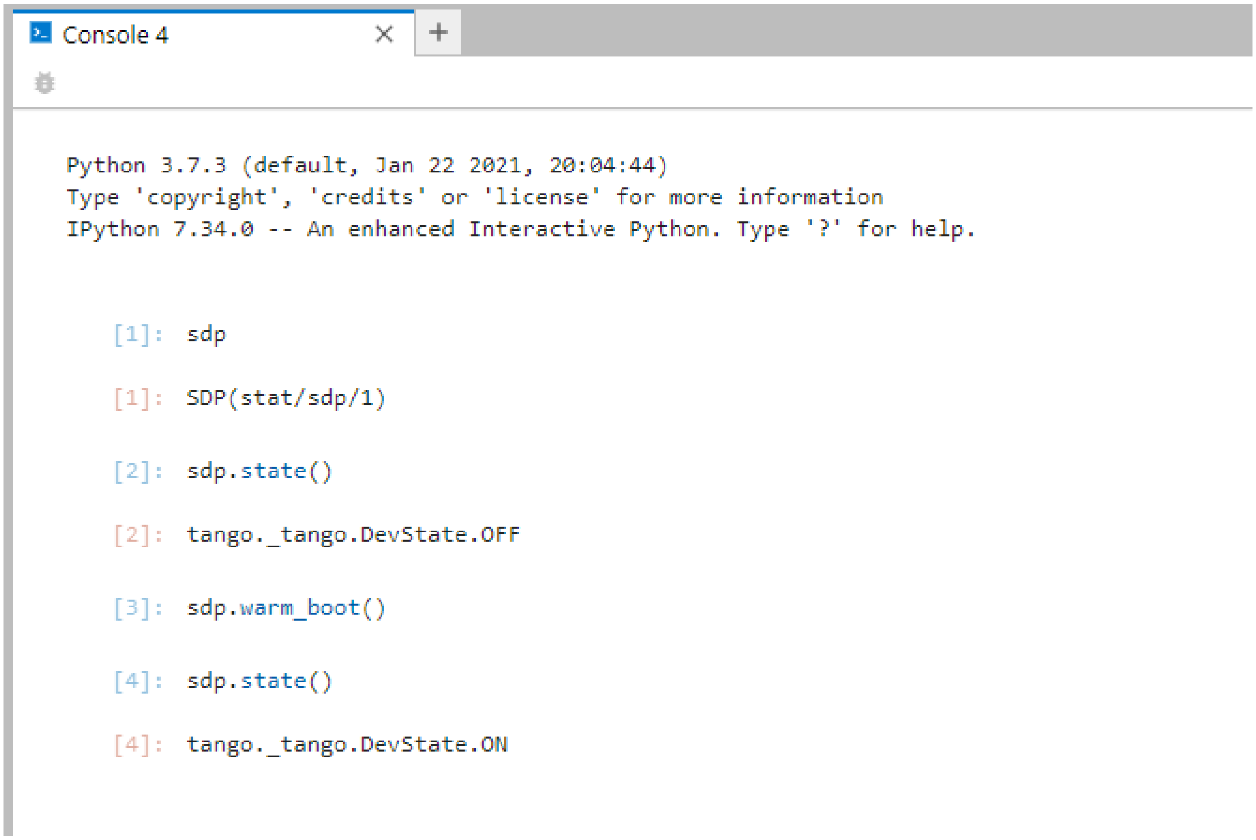
```
        sdp_h0,
        bst_h0,
        sst_h0,
        xst_h0,
        beamlet_h0,
        digitalbeam_h0,
        tilebeam_h0,
        af_h0,
        sdpfirmware_h1,
        sdp_h1,
        bst_h1,
        sst_h1,
        xst_h1,
        beamlet_h1,
        digitalbeam_h1,
        tilebeam_h1,
        af_h1,
    ]
    + apscts
    + apspus
    + recvs
    + unb2s
)
```

Note: the Jupyter notebooks use enhancements from the `itango` suite, which provide tab completions, but also the `Device` alias for `DeviceProxy` as was used in the Python examples in the next section.

For example, you can start a new *Station Control* notebook (File->New->Notebook->StationControl), and access these devices:



You can also use Jupyter Labs integrated console to run your commands (File->New->Console->StationControl) and exploit the `itango` suite enhancements:



2.1.2 Jupyter Lab and Git

We provide the ability to interact with git repositories by including the jupyter-git plugin. See their webpage for how to use this plugin.

In our installation, all git commits will be made as a fictive `JupyterLab on $HOSTNAME` user. This is because JupyterLab does not know who the user is, and it's preferred to explicitly state at least where the commit comes from, rather than under the name of the last person who told git who they are.

Any problems encountered in git that cannot be solved through the plugin, can be solved by spawning a Terminal and using the `git` command-line interface.

2.1.3 PyTango

To access a station from scratch using Python, we need to install some dependencies:

```
pip3 install tango
```

Then, if we know what devices are available on the station, we can access them directly:

```
import tango
import os

# Tango needs to know where our Tango API is running.
os.environ["TANGO_HOST"] = "localhost:10000"

```

(continues on next page)

(continued from previous page)

```
# Construct a remote reference to a specific device.
# One can also use "tango://localhost:10000/STAT/Boot/1" if TANGO_HOST is not set
boot_device = tango.DeviceProxy("STAT/Boot/1")

# Print the device's state.
print(boot_device.state())
```

To obtain a list of all devices, we need to access the database:

```
import tango

# Tango needs to know where our Tango API is running.
import os
os.environ["TANGO_HOST"] = "localhost:10000"

# Connect to the database.
db = tango.Database()

# Retrieve the available devices, excluding any Tango-internal ones.
# This returns for example: ['STAT/Boot/1', 'STAT/Docker/1', ...]
devices = list(db.get_device_exported("STAT/*"))

# Connect to any of them.
any_device = tango.DeviceProxy(devices[0])

# Print the device's state.
print(any_device.state())
```

2.1.4 ReST API

We also provide a ReST API to allow the station to be controlled without needing to use the Tango API. The root access point is http://localhost:8080/tango/rest/v10/hosts/databaseds;port=10000/ (credentials: tango-cs/tango). This API allows for:

- getting and setting attribute values,
- calling commands,
- retrieving the device state,
- and more.

For example, retrieving http://localhost:8080/tango/rest/v10/hosts/databaseds;port=10000/devices/STAT/SDP/1/state returns the following JSON document:

```
{"state":"ON","status":"The device is in ON state."}
```

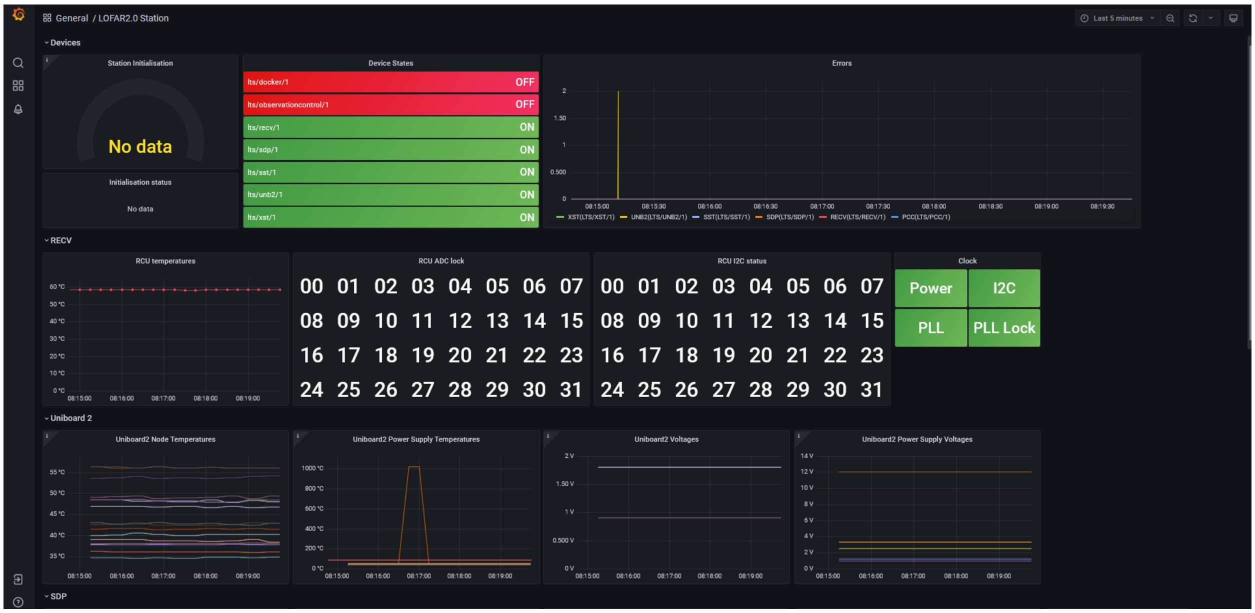
For a full description of this API, see https://tango-rest-api.readthedocs.io/en/latest/.

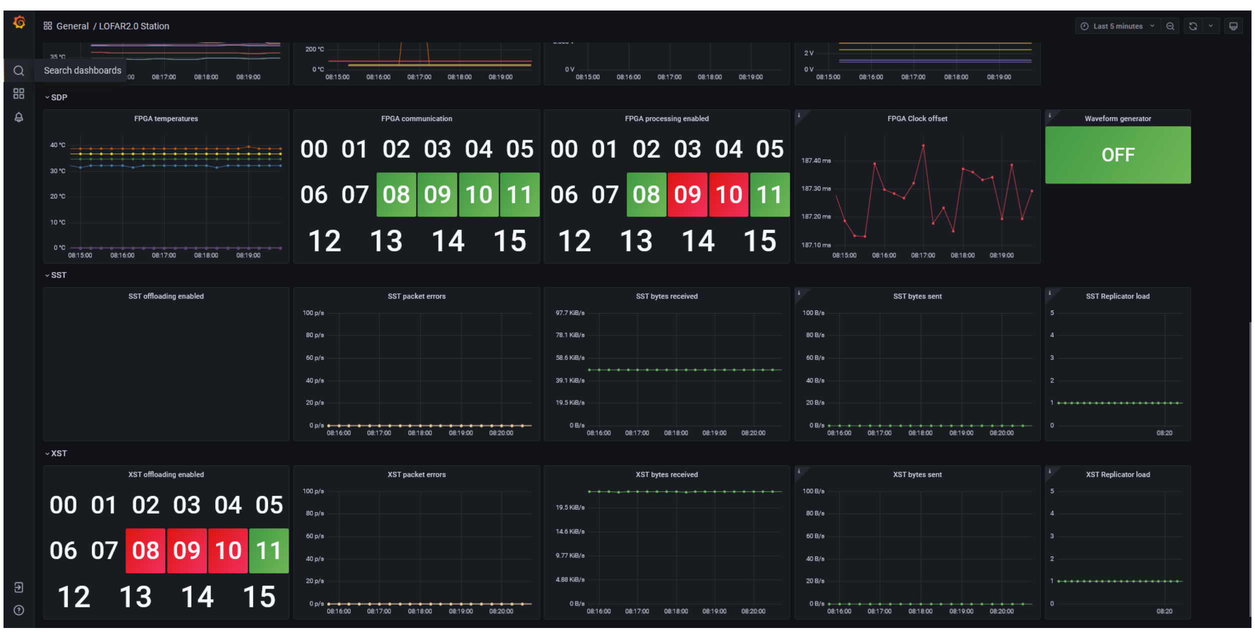
2.2 Monitoring GUIs

Each device exposes a list of monitoring points as attributes with the _R prefix. These can be accessed interactively from a controle console (such as Jupyter), but that will not scale.

2.2.1 Grafana

We offer Grafana dashboards on http://localhost:3000 that provide a quick overview of the station's status, including temperatures and settings. Several dashboards are included. An example:





NOTE: These dashboards are highly subject to change. The above examples provide an impression of a possible overview of the station state.

You are encouraged to inspect each panel (graph) to see the underlying database query and settings. Use the small arrow in the panel's title to get a drop-down menu of options, and select *inspect*. See the Grafana documentation for further information.

The Grafana dashboards are configured with the following data sources:

- *Prometheus*, the time-series database that caches the latest values of all monitoring points (see next section),
- *TangoDB*, providing access to device properties (fixed settings),
- *Loki*, the log output of the devices.

2.2.2 Prometheus

Prometheus is a low-level monitoring system that allows us to periodically retrieve the values of all the attributes of all our devices, and cache them to be used in Grafana:

- Every several seconds, Prometheus scrapes our TANGO-Grafana Exporter (our fork of https://gitlab.com/ska-telescope/TANGO-grafana.git), collecting all values of all the device attributes (except the large ones, for performance reasons).
- Prometheus can be queried directly on http://localhost:9090,
- The TANGO-Grafana Exporter can be queried directly on http://localhost:8000,
- The query language is PromQL, which is also used in Grafana to query Prometheus,

Prometheus stores attributes in the following format:

```
device_attribute{device="stat/recvh/1",
                 dim_x="32", dim_y="0",
                 instance="tango-prometheus-exporter:8000",
                 job="tango",
                 label="RCU_temperature_R",
                 name="RCU_temperature_R",
                 type="float",
                 x="00", y="0"}
```

The above describes a single data point and its labels. The primary identifying labels are `device` and `name`. Each point furthermore has a value (integer) and a timestamp. The following transformations take place:

- For 1D and 2D attributes, each array element is its own monitoring point, with `x` and `y` labels describing the indices. The labels `dim_x` and `dim_y` describe the array dimensionality,
- Attributes with string values get a `str_value` label describing their value.

2.3 Logs

The devices, and the docker containers in general, produce logging output. The easiest way to access the logs of a specific container is to ask docker directly. For example, to access and follow the most recent logs of the `device-sdp` container, execute on the host:

```
docker logs -n 100 -f device-sdp
```

This is mostly useful for interactive use.

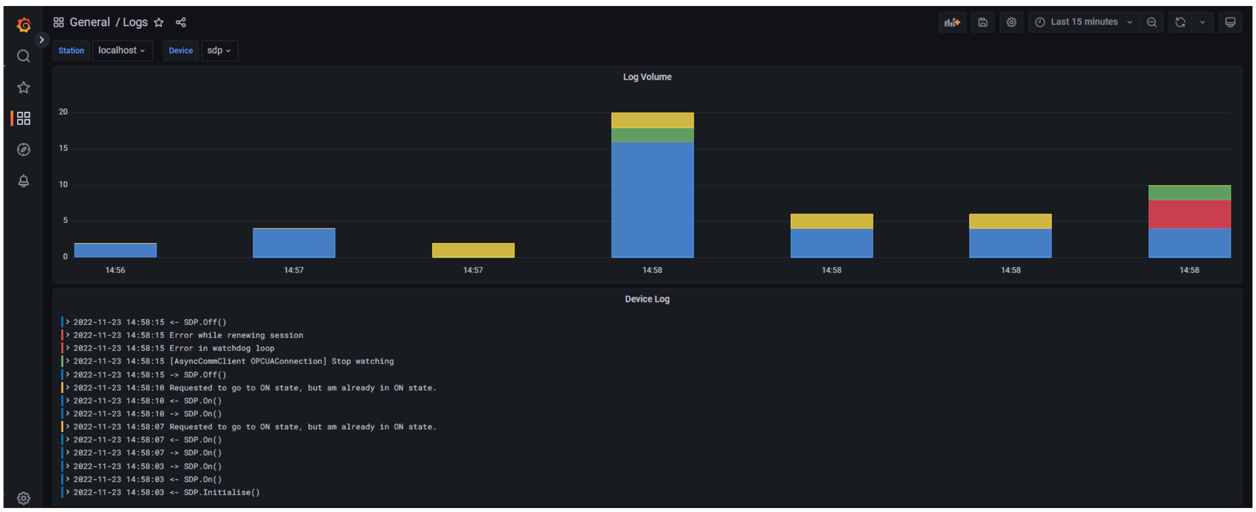
2.3.1 Loki

To monitor the logs remotely, or to browse older logs, use the *Logs Dashboard* that is included among the Grafana dashboards, and served on http://localhost:3000/d/Hqo-qIO4z/logs?orgId=1. Loki is a log aggregation system fully integrated in Grafana and inspired by Prometheus. Currently, the following logs are collected in our Grafana Loki implementation:

- Logs of all devices,
- Logs of the Docker containers.

Once reached the Grafana Logs Dashboard, it is possible to select several parameters such as *station name*, *device name* and *interval* to perform the log research. Logs will be marked with different colors, following their level (e.g. INFO, WARNING, ERROR, etc.).

You should see something like:



CHAPTER

THREE

DEVICES

This package implements the *Station Control (SC)* part of a LOFAR2.0 station, the core of which implements several Tango devices that connect to the station's hardware as well as to each other. In the following graph, green components are implemented in this package, the gray components are external:

A brief description of each of these devices:

- *ObservationControl* device spawns new Observation devices, given an observation specification,
- *Observation* device sets up the software and hardware on the station to execute a given specification,
- *AntennaField* device controls a set of antennas and their properties (f.e. their positions),
- *RECV* device represents the hardware that controls the antennas in the station,
- *TileBeam* device steers the beam of the HBA tiles, actively tracking any source,
- *SDP* device represents generic functionality of the firmware that digitally combines antenna inputs,
- *SST*, *XST*, and *BST* devices control and expose statistics generated by the SDP firmware,
- *Beamlet* device controls the observation output data (beamlets) that stream out of the station (in LOFAR, to CEP),
- *DigitalBeam* device steers the beam formed in SDP, actively tracking any source.

Auxilliary devices that control hardware are:

- *APSCT* device controls the ASPCT clock selection and distribution board,
- *APSPU* device controls the APSPU 48V distribution board,
- *UNB2* device controls the Uniboards that hold the SDP FPGAs (and thus firmware).
- *PSOC* device controls the power sockets (230V distribution).

Finally, the stack holds the auxilliary devices that control the software devices. They connect to too many devices to draw:

- *Docker* device controls the Docker containers of the software stack,
- *TemperatureManager* device acts on temperature alarms originating from the hardware.

CHAPTER

FOUR

USING DEVICES

The station exposes *devices*, each of which is a remote software object that manages part of the station. Each device has the following properties:

- It has a *state*,
- Many devices manage and represent hardware in the station,
- It exposes *read-only attributes*, that expose values from within the device or from the hardware it represents,
- It exposes *read-write attributes*, that allow controlling the functionality of the device, or the hardware it represents,
- It exposes *properties*, which are fixed configuration parameters (such as port numbers and timeouts),
- It exposes *commands*, that request the execution of a procedure in the device or in the hardware it manages.

The devices are accessed remotely using `DeviceProxy` objects. See *Monitoring & Control* on how to do this.

4.1 States

The state of a device is then queried with `device.state()`. Each device can be in one of the following states:

- `DevState.OFF`: The device is not operating,
- `DevState.INIT`: The device is being initialised,
- `DevState.STANDBY`: The device is initialised and ready to be configured further,
- `DevState.ON`: The device is operational,
- `DevState.ALARM`: The device is operational, but one or more attributes are in alarm,
- `DevState.FAULT`: The device is malfunctioning. Functionality cannot be counted on,
- `DevState.DISABLE`: The device is not operating because its hardware has been shut down.
- The `device.state()` function can throw an error, if the device cannot be reached at all. For example, because it's docker container is not running. See the *Docker* device on how to start it.

Each device provides the following commands to change the state:

boot()
Turn on the device, and initialise the hardware. Moves from `OFF` to `ON`.

warm_boot()
Turn on the device, but do not change the hardware. Moves from `OFF` to `ON`.

disable_hardware()
Shut down the hardware related to the device. Moves from `STANDBY`, `ON` or `ALARM` to `DISABLE`

off()
Turn the device `OFF` from any state.

The following procedure is a good way to bring a device to `ON` from any state:

```
def force_start(device):
    if device.state() == DevState.FAULT:
        device.off()
    if device.state() == DevState.OFF:
        device.boot()

    return device.state()
```

Hint: If a command gives you a timeout, the command will still be running until it finishes. You just won't know when it does or its result. In order to increase the timeout, use `device.set_timeout_millis(timeout * 1000)`.

4.2 FAULT

If a device enters the `FAULT` state, it means an error occurred that is fundamental to the operation of the software device. For example, the connection to the hardware was lost. To see the error reason, use

status()
The verbose status of the device, f.e. the reason why the device went to `FAULT`.

Interaction with the device in the `FAULT` state is undefined, and attributes cannot be read or written. The device needs to be reinitialised, which typically involves the following sequence of commands:

```
# turn the device off completely first.
device.off()

# turn on the device and fully reinitialise it
# alternatively, device.warm_boot() can be used,
# in which case no hardware is reinitialised.
device.boot()
```

Of course, the device could go into `FAULT` again, even during the `boot()` command, for example because the hardware it manages is unreachable. To debug the fault condition, check the *Logs* of the device in question.

4.3 Initialise hardware

Most devices provide the following commands, in order to configure the hardware with base settings. Note that these are automatically called during `boot()`, in this order:

initialise()
Initialise the device (connect to the hardware). Moves from `OFF` to `STANDBY`.

power_hardware_on()
For devices that control hardware, this command turns on power to it.

power_hardware_off()
For devices that control hardware, this command turns off power to it.

set_defaults()
Upload default attribute settings from the TangoDB to the hardware.

on()
Mark the device as operational. Moves from `STANDBY` to `ON`.

4.4 Attributes

The device can be operated in `ON` state, where it exposes *attributes* and *commands*. The attributes can be accessed as python properties, for example:

```
recvh = DeviceProxy("STAT/RECVH/1")

# turn on all LED0s
recvh.RCU_LED0_RW = [True] * 32

# retrieve the status of all LED0s
print(recvh.RCU_LED0_R)
```

The attributes with an:

- `_R` suffix are monitoring points, reflecting the state of the hardware, and are thus read-only.
- `_RW` suffix are control points, reflecting the desired state of the hardware. They are read-write, where writing requests the hardware to set the specified value. Reading them returns the last requested value.

4.4.1 Meta data

A description of the attribute can be retrieved using:

```
print(recvh.get_attribute_config("RCU_LED0_R").description)
```

4.5 Attribute masks

Several devices employ *attribute masks* in order to toggle which elements in their hardware array are actually to be controlled. This construct is necessary as most control points consist of arrays of values that cover all hardware elements. These array control points are always fully sent: it is not possible to update only a single element without uploading the rest. Without a mask, it is impossible to control a subset of the hardware.

The masks only affect *writing* to attributes. Reading attributes (monitoring points) always result in data for all elements in the array.

For example, the `RCU_mask_RW` array is the RCU mask in the `recvh` device. It behaves as follows, when we interact with the `RCU_LED0_R(W)` attributes:

```
recvh = DeviceProxy("STAT/RECVH/1")

# set mask to control all RCUs
recvh.RCU_mask_RW = [True] * 32
```

(continues on next page)

(continued from previous page)

```
# request to turn off LED0 for all RCUs
recvh.RCU_LED0_RW = [False] * 32

# <--- all LED0s are now off
# recvh.RCU_LED0_R should show this,
# if you have the RCU hardware installed.

# set mask to only control RCU 3
mask = [False] * 32
mask[3] = True
recvh.RCU_mask_RW = mask

# request to turn on LED0, for all RCUs
# due to the mask, only LED0 on RCU 3
# will be set.
recvh.RCU_LED0_RW = [True] * 32

# <--- only LED0 on RCU3 is now on
# recvh.RCU_LED0_R should show this,
# if you have the RCU hardware installed.
```

CHAPTER

FIVE

ANTENNAFIELD-HB (AFH), ANTENNAFIELD-LB (AFL)

The `afh == DeviceProxy("STAT/AFH/HBA")` device represents a set of *antennas* or *tiles* that collectively form a High-Band antenna field.

The `afl == DeviceProxy("STAT/AFL/LBA")` device represents a set of *antennas* that collectively form a Low-Band antenna field.

They represent a selection of inputs from one or more `RECV` devices, mapped onto an `SDP` device, annotated with metadata such as positional information.

nr_antennas_R
The number of antennas or tiles in the antenna field.

type
`uint32`

It provides many settings that map onto the `RECV` device directly, serving as a funnel:

ANT_mask_RW
Which antennas are configured when writing settings.

type
`bool[N_antennas]`

Warning: Any antennas in the field that are not connected to any `RECV` device will return default values (f.e. `False` or `0`).

5.1 Observation setup

To use the AntennaField for an observation, it and its downstream `RECV` and `SDP` devices must be configured correctly. We provide the following functionality:

Frequency_Band_RW
Which frequency band to select for each antenna, f.e. LBA_10_90. Must be compatible with the antenna type of the field. Writing to this attribute configures and calibrates both RECV and SDP accordingly. When read, it returns "" for any antenna that has an unknown setup.

type
`str[N_antennas]`

5.2 Positions

The following attributes expose positional information about the individual antennas in the field, in different formats:

Antenna_Reference_GEO_R
Reference position of each HBA tile, in latitude/longitude (degrees).

type
`float64[N_tiles][2]`

Antenna_Field_Reference_GEO_R
Reference position of the antenna field, in latitude/longitude (degrees).

type
`float64[2]`

Additionally, the `ITRF` and `GEOHASH` variants provide the same information, but in ITRF (x/y/z, metres), and in Geohash strings, respectively.

Also, the offsets of the elements within each HBA tile are provided:

HBAT_antenna_ITRF_offsets_R
Relative position of each HBA tile element with respect to the tile reference.

type
`float64[N_tiles][N_elements * 3]`

shape
`float64[N_tiles][N_elements][3]`

5.3 Configuration

The antennas represented by the antenna field are selected by the following properties:

RECV_devices
The list of `RECV` devices from which antennas are selected.

type
`str[]`

SDP_device
The `SDP` device that processes the antennas.

type
`str`

5.3.1 Antenna mapping

These properties configure which inputs in RECV represent the power and control for each antenna:

HBAT_Power_to_RECV_mapping
Pairs of numbers `(recv_idx, ant_idx)` describing the inputs on which the HBAT *power* is connected. The `recv_idx` is the index in `RECV_devices`, starting at 1. The `ant_idx` is the absolute index of the antenna in the `RECV` device. A value of `-1` means the antenna is not connected at all.

type
`int32[]`

shape
`int32[][2]`

Control_to_RECV_mapping
Pairs of numbers `(recv_idx, ant_idx)` describing the inputs on which the Antenna *control* is connected. The `recv_idx` is the index in `RECV_devices`, starting at 1. The `ant_idx` is the absolute index of the antenna in the `RECV` device. A value of `-1` means the antenna is not connected at all.

5.3.2 Positions

The positions are given in ETRS, using the following properties:

Antenna_Reference_ETRS
Reference position of each HBA tile, in ETRS (x/y/z, metres).

type
`float64[N_tiles][3]`

Antenna_Field_Reference_ETRS
Reference position of the antenna field, in ETRS (x/y/z, metres).

type
`float64[3]`

ITRF_Reference_Frame
Reference frame to use for converting ETRS to ITRF (f.e. "ITRF2005").

type
`str`

ITRF_Reference_Epoch
Epoch towards which to extrapolate the ITRF frame, typically in half-year increments (f.e. 2015.5).

type
`float32`

For the ETRS positions, there is an alternative to provide them using the respective `ITRF` property, which overrides the automatic ETRS-to-ITRF conversion.

5.4 HBAT element positions

The positions of the elements within an HBA tile are handled differently. Instead of storing the positions of each of the 16 elements in each tile, we use the fact that the relative positions of the elements within each tile is fixed, and that in LOFAR stations, all the HBA tiles of a station are on the same plane (instead of following the curvature of the Earth). This plane is given its own station-local coordinates, the PQR system:

- It's origin is at a chosen center of the station,
- The Q axis is aligned with an absolute North (not the North of the station, which would be a different direction per station),
- The P axis is roughly East,
- The R axis is roughly down,
- The HBA tiles on a station all lie on the same PQ plane, so R == 0.

These facts allow us to use the following information to calculate the absolute position of each tile element. The conversion takes the relative offsets of the elements within a tile, rotates them in PQR space, rotates those into relative

ETRS offsets, and finally into absolute positions in ETRS. See `tangostationcontrol.tilebeam.hba_tile` for these computations.

recv.HBAT_PQR_rotation_angles_deg
(property) The horizontal rotation of each HBA tile in the PQ plane, in degrees (Q -> P).

type
`float[96]`

recv.PQR_to_ETRS_rotation_matrix
(property) The 3D rotation matrix to convert PQR coordinates into relative ETRS coordinates.

type
`float[3][3]`

CHAPTER

SIX

TILEBEAM, DIGITALBEAM

A primary function of the station is to combine its antenna signals to create a more sensitive signal. The antennas are typically aimed at celestial sources moving across the sky, but can also be aimed at stationary targets, for example to point at Earth-bound signals or to let the sky pass through the beam instead.

Given a certain direction, and knowing the speed of light, one can compute the differences in arrival time for light from the observed source (its wave front) towards each antenna. The antenna signals are then aligned towards the source by delaying the signal inputs based on these differences. The antennas closest to the source get the largest delay. For celestial sources, the light is assumed to be infinitely far away and thus travel in parallel towards each antenna, greatly simplifying the calculations involved.

In practice, antenna signals can only be coarsely delayed. Fine delay compensation consists of rotating the signal inputs to compensate for the remaining differences in phase. The amount of rotation is frequency dependent. The aligned signals are subsequently added, creating a single signal output of higher sensitivity towards the observed source, albeit with a narrower field of view.

Beam tracking therefor requires a *pointing* direction in which to observe, as well as the *positions* of the antennas involved. Finally, the antennas need to be periodically realigned to track moving sources. We distinguish the following concepts:

- *Beam forming* is combining individual element signals into one. This is performed by the HBAT hardware and SDP firmware,
- *Beam steering* is uploading the delays or weights to the beam-forming hardware, in order to point the beam in a certain direction,
- *Beam tracking* is updating the beam steering over time to track a celestial target, compensating for the Earth's movement through space.

The `tilebeam == DeviceProxy("STAT/TileBeam/1")` device configures the HBA beam former in each HBA tile, which adds the signals of its 16 elements within the tile. The output signal of these tiles is used as input for the digital beam former (just like the direct output of an LBA).

The `digitalbeam == DeviceProxy("STAT/DigitalBeam/1")` device configures the digital beam formed in SDP from antenna or tile inputs. The output signal in SDP are *beamlets*, which can

Both devices beamform the antennas configured in its associated `AntennaField` device, but differ in what they beamform and with respect to which position:

- TileBeam:
  - Beamforms HBA elements in the HBA tiles of its AntennaField device,
  - Uses `antennafield.Antenna_Reference_ITRF_R` as the reference position for each tile,
  - Allows a different pointing per HBA tile,
  - `N_output := antennafield.nr_antennas_R`,

  - Uploads the computed weights to `antennafield.HBAT_bf_delay_steps_RW`,
  - These weights are actually *delay steps* to be applied in the tile for each element.

- DigitalBeam
  - Beamforms all the antennas or tiles of its AntennaField device,
  - Uses `antennafield.Antenna_Field_Reference_ITRF_R` as the reference position,
  - Allows a different pointing per beamlet,
  - `N_output := NUM_BEAMLETS = 488`,
  - Uploads the computed weights to `beamlet.FPGA_bf_weights_pp_RW`,
  - These weights are actually complex *phase rotations* to be applied on each antenna input.

6.1 Common functionality

The following functionality holds for both TileBeam and DigitalBeam.

6.1.1 Beam Tracking

Beam tracking automatically recomputes and reapplies pointings periodically, and immediately when new pointings are configured. It exposes the following interface:

Tracking_enabled_R
Whether beam tracking is running.

type
`bool`

Pointing_direction_RW
The direction in which the beam should be tracked for each antenna. The beam tracker will steer the beam periodically, and explicitly whenever the pointings change.

type
`str[N_output][3]`

Pointing_direction_R
The last applied pointing of each antenna.

type
`str[N_output][3]`

Pointing_timestamp_R
The timestamp for which the last set pointing for each antenna was applied and set (in seconds since 1970).

type
`float[N_output][3]`

A pointing describes the direction in the sky, and consists of a set of coordinates and the relevant coordinate system. They are represented as a tuple of 3 strings: `("coordinate_system", "angle1", "angle2")`, where the interpretation of `angle1` and `angle2` depends on the coordinate system used. For example:

- `("AZELGEO", "0deg", "90deg")` points at Zenith (Elevation = 90°, with respect to the Earth geode),
- `("J2000", "0deg", "90deg")` points at the North Celestial Pole (Declination = 90°),

- `("SUN", "0deg", "0deg")` points at the centre of the Sun.

For a full list of the supported coordinate systems, see https://casacore.github.io/casacore/classcasacore_1_1MDirection.html

6.1.2 Beam Steering

The beam steering is responsible for pointing the beams at a target, by converting the pointing to hardware-specific weights and uploading them to the corresponding device. The beam steering is typically controlled by the beam tracker. To point the antennas in any direction manually, you should disable beam tracking first:

Tracking_enabled_RW
: Enable or disable beam tracking (default: `True`).

  type
  : `bool`

set_pointing(pointings)
: Point the beams towards the specified `pointings[N_output][3]` for all outputs.

  returns
  : `None`

The direction of each pointing is derived using *casacore*, which must be periodically calibrated, see also *Celestial & Geodetic Calibration*.

6.1.3 Timing

The beam tracking applies an update each *interval*, and aims to apply it at timestamps `(now % Beam_tracking_interval) - Beam_tracking_application_offset`. To do so, it starts its computations every interval `Beam_tracking_preparation_period` seconds before. It then starts to compute the weights, waits to apply them, and applies them by uploading the weights to the underlying hardware.

The following properties are used:

Beam_tracking_interval
: Update the beam tracking at this interval (seconds).

  type
  : `float`

Beam_tracking_application_offset
: Update the beam tracking this amount of time before the next interval (seconds).

  type
  : `float`

Beam_tracking_preparation_period
: Prepare time for each period to compute and upload the weights (seconds).

  type
  : `float`

The following timers allow you to track the durations of each stage:

Duration_compute_weights_R
: Amount of time it took to compute the last weights (seconds).

  type
  : `float`

Duration_preparation_period_slack_R
Amount of time left in the prepration period between computing and uploading the weights (seconds).

type
`float`

Duration_apply_weights_R
Amount of time it took to apply (upload) the weights (seconds).

type
`float`

6.2 DigitalBeam

The DigitalBeam device applies the following configuration to compute each beamlet. Here, `N_ant := antennafield.nr_antennas_R` and `N_beamlet := NUM_BEAMLETS == N_output`.

Antenna_Set_RW
Which antenna set (supported by the antenna field) is requested to be beam formed.

type
`str`

Antenna_Mask_R
Which antennas are requested to be beam formed, according to the selected antenna set.

type
`bool[N_ant]`

antennafield.Antenna_Usage_Mask_R
Which antennas are OK to be used (not broken, disabled, etc).

type
`bool[N_ant]`

beamlet.subband_select_RW
Which subband to beamform for each beamlet.

type
`uint32[N_beamlet]`

sdp.subband_frequency_R
Central frequency of each subband (in Hz).

type
`float`

CHAPTER

SEVEN

BEAMLET

The `beamlet == DeviceProxy("STAT/Beamlet/1")` device controls the creation and emission of beamlets. Each beamlet is a signal stream characterised by:

- The set of antennas to use as input,
- The pointing towards which to beamform these antennas,
- A single subband (frequency) selected from the PPF.

CHAPTER

EIGHT

RECVH, RECVL

The `recvh == DeviceProxy("STAT/RECVH/1")` device controls the RCUs for HBA tiles.

The `recvl == DeviceProxy("STAT/RECVL/1")` device controls the RCUs for LBA antennas.

Central to their operations are the masks (see also *Attribute masks*):

RCU_mask_RW
: Controls which RCUs will actually be configured when attributes referring to RCUs are written.

  type
  : `bool[N_RCUs]`

Ant_mask_RW
: Controls which antennas will actually be configured when attributes referring to antennas are written.

  type
  : `bool[N_antennas]`

Typically, `N_RCUs == 32`, and `N_antennas == 96`.

Note: The antennas are hooked up to the RCUs in sets of 3, in order.

8.1 Error information

These attributes summarise the basic state of the device. Any elements which are not present in `FPGA_mask_RW` will be ignored and thus not report errors:

RCU_error_R
: Whether the RCUs appear usable.

  type
  : `bool[N_RCUs]`

ANT_error_R
: Whether the antennas appear usable.

  type
  : `bool[N_antennas]`

RCU_IOUT_error_R
: Whether there are alarms on any of the amplitudes in the measured currents.

  type
  : `bool[N_RCUs]`

RCU_VOUT_error_R
Whether there are alarms on any of the voltages in the measured currents.

type
`bool[N_RCUs]`

RCU_TEMP_error_R
Whether there are alarms on any of the temperatures. NB: These values are also exposed for unused RCUs (the `RCU_mask_RW` is ignored).

type
`bool[N_RCUs]`

CHAPTER NINE

SDP FIRMWARE

The `sdpfirmware == DeviceProxy("STAT/SDPFirmware/1")`` device controls the firmware functionalities related to the digital signal processing in SDP device. Central to its operation is the mask (see also *Attribute masks*):

TR_fpga_mask_RW
Controls which FPGAs will actually be configured when attributes referring to FPGAs are written.

type
`bool[N_fpgas]`

Typically, `N_fpgas == 16`.

See the following links for a full description of the SDP monitoring and control points:

- https://support.astron.nl/confluence/pages/viewpage.action?spaceKey=L2M&title=L2+STAT+Decision%3A+SC+-+SDP+OPC-UA+interface
- https://plm.astron.nl/polarion/#/project/LOFAR2System/wiki/L2%20Interface%20Control%20Documents/SC%20to%20SDP%20ICD

9.1 Basic configuration

The following points are significant for the operations of this device:

TR_fpga_communication_error_R
Whether the FPGAs can be reached.

type
`bool[N_fpgas]`

9.2 Error information

These attributes summarise the basic state of the device. Any elements which are not present in `FPGA_mask_RW` will be ignored and thus not report errors:

FPGA_error_R
Whether the FPGAs appear usable.

type
`bool[N_fpgas]`

CHAPTER

TEN

SDP

The `sdp == DeviceProxy("STAT/SDP/1")`` device controls the digital signal processing in SDP, performed by the firmware on the FPGAs on the Uniboards.

See the following links for a full description of the SDP monitoring and control points:

- https://support.astron.nl/confluence/pages/viewpage.action?spaceKey=L2M&title=L2+STAT+Decision%3A+SC+-+SDP+OPC-UA+interface
- https://plm.astron.nl/polarion/#/project/LOFAR2System/wiki/L2%20Interface%20Control%20Documents/SC%20to%20SDP%20ICD

10.1 Basic configuration

The following points are significant for the operations of this device:

FPGA_processing_enable_R
Whether the FPGA is processing its input.

type
`bool[N_fpgas]`

10.2 Frequency management

To setup the input and output frequencies, the following attributes are offered:

antenna_RW
The type of antenna connected to each input, as provided by the user (*HBA* or *LBA*).

type
`str[N_fpgas][N_ants_per_fpga]`

clock_RW
The FPGA clock, in Hz (*200_000_000* or *160_000_000*). NB: This informs the calculations which clock should be assumed. The clock is not actually toggled.

type
`uint32`

nyquist_zone_RW
The NyQuist zone of the input, per input (0, 1, or 2).

type
`uint32[N_fpgas][N_ants_per_fpga]`

FPGA_spectral_inversion_RW
Whether to invert the spectrum, both within and across all subbands. This is required in odd-numbered NyQuist zones to have the signal increase in frequency over the subbands. This setting is automatically configured by setting *nyquist_zone_RW* but can be overwritten explicitly as well.

type
`bool[N_fpgas][N_ants_per_fpga]`

All of these are required to compute the actual frequencies of the subbands constructed by the PPF inside the FPGA. For convenience, the device explicitly exposes these:

subband_frequency_R
The central frequency of each subband for each input, in Hz.

type
`float64[N_fpgas][N_ants_per_fpga][N_subbands]`

10.3 Data-quality information

The following fields describe the data quality (see also *Signal Chain*):

FPGA_signal_input_mean_R
Mean value of the last second of input (in ADC quantisation units). Should be close to 0.

type
`double[N_fpgas][N_ants_per_fpga]`

FPGA_signal_input_rms_R
Root means square value of the last second of input (in ADC quantisation units). `rms^2 = mean^2 + std^2`. Values above 2048 indicate strong RFI. Values of 0 indicate a lack of signal input.

type
`double[N_fpgas][N_ants_per_fpga]`

10.4 Error information

These attributes summarise the basic state of the device. Any elements which are not present in `FPGA_mask_RW` will be ignored and thus not report errors:

FPGA_procesing_error_R
Whether the FPGAs are processing their input from the RCUs. NB: This will also raise an error if the Waveform Generator is enabled.

type
`bool[N_fpgas]`

10.5 Version Information

The following fields provide version information:

FPGA_firmware_version_R
The active firmware images.

type
`str[N_fpgas]`

FPGA_hardware_version_R
The versions of the boards hosting the FPGAs.

type
`str[N_fpgas]`

TR_software_version_R
The version of the server providing the OPC-UA interface.

type
`str[N_fpgas]`

10.6 Waveform Generator

The antenna input of SDP can be replaced by an internal waveform generator for debugging and testing purposes. The generator is configured per antenna per FPGA:

Note: The Waveform Generator needs to be toggled off and on using `FPGA_wg_enable_RW` for new settings to become active on the station.

FPGA_wg_enable_RW
Whether the waveform generator is enabled for each input.

type
`bool[N_fpgas][N_ants_per_fpga]`

FPGA_wg_phase_RW
The phases of the generated waves (in degrees). The generator needs to be turned off and on if this is changed, in order to bring the generators in sync.

type
`float32[N_fpgas][N_ants_per_fpga]`

FPGA_wg_frequency_RW
The frequencies of the generated waves (in Hz). The frequency of a subband `s` is LBA: `s * 200e6/1024`, HBA low band: `(512 + s) * 200e6/1024`, HBA high band: `(1024 + s) * 200e6/1024`.

type
`float32[N_fpgas][N_ants_per_fpga]`

FPGA_wg_amplitude_RW
The amplitudes of the generated waves. Useful is a value of `0.1`, as higher risks clipping.

type
`float32[N_fpgas][N_ants_per_fpga]`

10.6.1 Usage example

For example, the following code inserts a wave on LBA subband 102 on FPGAs 8 - 11:

```
# configure FPGAs to control
sdpfirmware.TR_fpga_mask_RW = [False] * 8 + [True] * 4 + [False] * 4

# configure waveform generator
sdp.FPGA_wg_phase_RW     = [[0] * 12] * 16
sdp.FPGA_wg_amplitude_RW = [[0.1] * 12] * 16
sdp.FPGA_wg_frequency_RW = [[102 * 200e6/1024] * 12] * 16

# toggle and enable waveform generator
sdp.FPGA_wg_enable_RW = [[False] * 12] * 16
sdp.FPGA_wg_enable_RW = [[True] * 12] * 16
```

CHAPTER

ELEVEN

BST, SST, AND XST

The `bst == DeviceProxy("STAT/BST/1")`, `sst == DeviceProxy("STAT/SST/1")` and `xst == DeviceProxy("STAT/XST/1")` devices manages the BSTs (beamlet statistics) SSTs (subband statistics) and XSTs (crosslet statistics), respectively. The statistics are emitted piece-wise through UDP packets by the FPGAs on the Uniboards in SDP. By default, each device configures the statistics to be streamed to itself (the device), from where the user can obtain them.

The statistics are exposed in two ways, as:

- *Attributes*, representing the most recently received values,
- *TCP stream*, to allow the capture and recording of the statistics over any period of time.

If the statistics are not received or zero, see *I am not receiving any XSTs and/or SSTs from SDP!*.

See the following links for a full description of the BST, SST, and XST monitoring and control points:

- https://support.astron.nl/confluence/pages/viewpage.action?spaceKey=L2M&title=L2+STAT+Decision%3A+SC+-+SDP+OPC-UA+interface
- https://plm.astron.nl/polarion/#/project/LOFAR2System/wiki/L2%20Interface%20Control%20Documents/SC%20to%20SDP%20ICD

11.1 BST Statistics attributes

11.2 SST Statistics attributes

The SSTs represent the amplitude of the signal in each subband, for each antenna, as an integer value. They are exposed through the following attributes:

sst_R
Amplitude of each subband, from each antenna.

type
`uint64[N_ant][N_subbands]`

sst_timestamp_R
Timestamp of the data, per antenna.

type
`uint64[N_ant]`

integration_interval_R
Timespan over which the SSTs were integrated, per antenna.

type
float32[N_ant]

subbands_calibrated_R
Whether the subband data was calibrated using the subband weights.

type
bool[N_ant]

Typically, `N_ant == 192`, and `N_subbands == 512`.

11.3 XST Statistics attributes

The XSTs represent the cross-correlations between each pair of antennas, as complex values. The phases and amplitudes of the XSTs represent the phase and amplitude difference between the antennas, respectively. They are exposed as a matrix `xst[a][b]`, of which only the triangle `a<=b` is filled, as the cross-correlation between antenna pairs `(b,a)` is equal to the complex conjugate of the cross-correlation of `(a,b)`. The other triangle contains incidental values, but will be mostly 0.

Complex values which cannot be represented in Tango attributes. Instead, the XST matrix is exposed as both their carthesian and polar parts:

xst_power_R, xst_phase_R
Amplitude and phase (in radians) of the crosslet statistics.

type
float32[N_ant][N_ant]

xst_real_R, xst_imag_R
Real and imaginary parts of the crosslet statistics.

type
float32[N_ant][N_ant]

xst_timestamp_R
Timestamp of each block.

type
int64[N_blocks]

integration_interval_R
Timespan over which the XSTs were integrated, for each block.

type
float32[N_blocks]

Typically, `N_ant == 192`, and `N_blocks == 136`.

The metadata refers to the *blocks*, which are emitted by the FPGAs to represent the XSTs between 12 x 12 consecutive antennas. The following code converts block numbers to the indices of the first antenna pair in a block:

```
from tangostationcontrol.common.baselines import baseline_from_index

def first_antenna_pair(block_nr: int) -> int:
    coarse_a, coarse_b = baseline_from_index(block_nr)
    return (coarse_a * 12, coarse_b * 12)
```

Conversely, to calculate the block index for an antenna pair `(a,b)`, use:

```python
from tangostationcontrol.common.baselines import baseline_index

def block_nr(a: int, b: int) -> int:
    return baseline_index(a // 12, b // 12)
```

11.3.1 Configuring the XSTs

The XSTs can be configured with several settings:

Note: The XST processing needs to be toggled off and on using `FPGA_xst_processing_enable_RW` for new settings to become active on the station.

FPGA_xst_processing_enable_RW
Whether XSTs are computed on each FPGA.

type
`bool[N_fpgas]`

FPGA_xst_integration_interval_RW
The time interval to integrate over, per FPGA, in seconds.

type
`float[N_fpgas]`

FPGA_xst_subband_select_RW
The subband to cross correlate, per FPGA. Note: only the entries `[x][1]` should be set, the rest should be zero.

type
`uint32[N_fpgas][8]`

11.4 Subscribe to statistics streams

The TCP stream interface allows a user to subscribe to the statistics packet streams, combined into a single TCP stream. The statistics will be streamed until the user disconnects, or the device is turned off. Any number of subscribers is supported, as bandwidth allows. Simply connect to the following port:

Device	TCP end point
SST	localhost:5101
XST	localhost:5102

The easiest way to capture this stream is to use our `statistics_writer`, which will capture the statistics and store them in HDF5 file(s). The writer:

- computes packet boundaries,
- processes the data of each packet, and stores their values into the matrix relevant for the mode,
- stores a matrix per timestamp,
- stores packet header information per timestamp, as HDF5 attributes,
- writes to a new file at a configurable interval.

To install the software locally and run the writer:

```
pip install 'tangostationcontrol@git+https://git.astron.nl/lofar2.0/tango.git
↪#subdirectory=tangostationcontrol'
l2ss-statistics-writer --mode SST --host localhost
```

The correct port will automatically be chosen, depending on the given mode. See also `l2ss-statistics-writer -h` for more information.

The writer can also parse a statistics stream stored in a file. This allows the stream to be captured and processed independently. Capturing the stream can for example be done using `netcat`:

```
nc localhost 5101 > SST-packets.bin
```

CHAPTER

TWELVE

STATIONMANAGER

The `stationmanager == DeviceProxy("STAT/StationManager/1")` Controls the station

CHAPTER THIRTEEN

DOCKER

The `docker == DeviceProxy("STAT/Docker/1")` device controls the docker containers. It allows starting and stopping them, and querying whether they are running. Each container is represented by two attributes:

<container>_R
: Returns whether the container is running.

  type
  `bool`

<container>_RW
: Set to `True` to start the container, and to `False` to stop it.

  type
  `bool`

> **Warning:** Do *not* stop the `tango` container, as doing so cripples the Tango infrastructure, leaving the station inoperable. It is also not wise to stop the `device_docker` container, as doing so would render this device unreachable.

CHAPTER

FOURTEEN

PSOC

The `psoc == DeviceProxy("STAT/PSOC/1")` device controls the Power Distribution Unit (PSOC).

CHAPTER FIFTEEN

CCD

The `ccd == DeviceProxy("STAT/CCD/1")` Clock Control Device controls the clock

CHAPTER

SIXTEEN

EC

The `ec == DeviceProxy("STAT/EC/1")` device controls the Environmental Control (EC).

CHAPTER

SEVENTEEN

CONFIGURATION

The `Configuration == DeviceProxy("STAT/Configuration/1")` Configuration Device controls the loading, updating, exposing and dumping of the whole Station Configuration

CHAPTER

EIGHTEEN

TEMPERATUREMANAGER

```
temperature_manager == DeviceProxy("STAT/TemperatureManager/1")
```

CHAPTER NINETEEN

DEVICE CONFIGURATION

The devices receive their configuration from two sources:

- The TangoDB database, for static *properties*,
- Externally, from the user, or a control system, that set *control attributes* (see the section for each device for what to set, and *Attributes* for how to set them).

19.1 TangoDB

The TangoDB database is a persistent store for the properties of each device. The properties encode static settings, such as the hardware addresses, and default values for control attributes.

Each device queries the TangoDB for the value of its properties during the `boot()` (or `initialise()`) call. Default values for control attributes can then be applied by explicitly calling `set_defaults()`. The `boot` device also calls `set_defaults()` when initialising the station. The rationale being that the defaults can be applied at boot, but shouldn't be applied automatically during operations, as not to disturb running hardware.

19.2 Device interaction

The properties of a device can be queried from the device directly:

```
# get a list of all the properties
property_names = device.get_property_list("*")

# fetch the values of the given properties. returns a {property: value} dict.
property_dict = device.get_property(property_names)
```

Properties can also be changed:

```
changeset = { "property": "new value" }

device.put_property(changeset)
```

Note that new values for properties will only be picked up by the device during `boot()` (or `initialise()`), so you will have to turn the device off and on.

19.3 Command-line interaction

The content of the TangoDB can be dumped from the command line using:

```
sbin/dsconfig.sh --dump > tangodb-dump.json
```

and changes can be applied using:

```
sbin/dsconfig.sh --update changeset.json
```

Note: The `dsconfig` docker container needs to be running for these commands to work.

CHAPTER

TWENTY

ENTER YOUR LOFAR2.0 HARDWARE CONFIGURATION

The software will need to be told various aspects of your station configuration, for example, the hostnames of the station hardware to control. The following settings are installation specific, and are stored as *properties* in the *TangoDB*.

Stock configurations are provided for several stations, as well as using simulators to simulate the station's interface (which is the default after bootstrapping a station). These are provided in the `CDB/stations/` directory, and can be loaded using for example:

```
sbin/dsconfig.sh --update CDB/stations/LTS_ConfigDb.json
```

The following sections describe the settings that are station dependent, and thus must or can be set.

20.1 Mandatory settings

Without these settings, you will not obtain the associated functionality:

RECV.OPC_Server_Name
Hostname of RECVTR.

type
`string`

UNB2.OPC_Server_Name
Hostname of UNB2TR.

type
`string`

SDPFirmware.OPC_Server_Name
Hostname of SDPTR.

type
`string`

SDP.OPC_Server_Name
Hostname of SDPTR.

type
`string`

SST.OPC_Server_Name
Hostname of SDPTR.

type
`string`

SST.FPGA_sst_offload_hdr_eth_destination_mac_RW_default
MAC address of the network interface on the host running this software stack, on which the SSTs are to be received. This network interface must be capable of receiving Jumbo (MTU=9000) frames.

type
`string[N_fpgas]`

SST.FPGA_sst_offload_hdr_ip_destination_address_RW_default
IP address of the network interface on the host running this software stack, on which the SSTs are to be received.

type
`string[N_fpgas]`

XST.OPC_Server_Name
Hostname of SDPTR.

type
`string`

XST.FPGA_xst_offload_hdr_eth_destination_mac_RW_default
MAC address of the network interface on the host running this software stack, on which the XSTs are to be received. This network interface must be capable of receiving Jumbo (MTU=9000) frames.

type
`string[N_fpgas]`

XST.FPGA_xst_offload_hdr_ip_destination_address_RW_default
IP address of the network interface on the host running this software stack, on which the XSTs are to be received.

type
`string[N_fpgas]`

20.2 Optional settings

These settings make life nicer, but are not strictly necessary to get your software up and running:

RECV.Ant_mask_RW_default
Which antennas are installed.

type
`bool[N_RCUs][N_antennas_per_RCU]`

SDP.RCU_mask_RW_default
Which RCUs are installed.

type
`bool[N_RCUs]`

UNB2.UNB2_mask_RW_default
Which Uniboard2s are installed in SDP.

type
`bool[N_unb]`

SDP.TR_fpga_mask_RW_default
Which FPGAs are installed in SDP.

type
`bool[N_fpgas]`

SDP.FPGA_sdp_info_station_id_RW_default

Numeric identifier for this station.

type

`uint32[N_fpgas]`

CHAPTER

TWENTYONE

OBSERVING

This chapter describes how to start and manage observations.

21.1 Starting an observation

To observe with a station, you must construct the observation's specifications, and hand it to the `DeviceProxy("STAT/ObservationControl/1")` device to start:

```
observation_spec = {
  "observation_id": 12345,
  "start_time": "2106-02-07T00:00:00",
  "stop_time": "2106-02-07T01:00:00",
  "antenna_field": "HBA",
  "antenna_set": "ALL",
  "filter": "HBA_210_250",
  "dithering": {
    "enabled": true,
    "power": -4.0,
    "frequency": 102000000
  },
  "SAPs": [{
        "subbands": [10, 20, 30],
        "pointing": { "angle1": 1.0, "angle2": 0, "direction_type": "J2000" }
  }, {
        "subbands": [40, 50, 60],
        "pointing": { "angle1": 2.0, "angle2": 0, "direction_type": "J2000" }
  }],
  "HBA": {
    "DAB_filter": true,
    "tile_beam": { "angle1": 1.5, "angle2": 0, "direction_type": "J2000" }
  }
}

import json
obs_control = DeviceProxy("STAT/ObservationControl/1")
obs_control.add_observation(json.dumps(observation_spec))
```

The above specification contains the following parameters:

Parameter	Description
observation_id	User-specified unique reference to this observation.
start_time	automatically start observing when this timestamp is reached. (optional)
stop_time	automatically stop observing when this timestamp is reached.
antenna_field	Which antenna field to use (LBA, HBA, HBA0, HBA1).
antenna_set	Which subset of antennas to use (ALL, INNER, OUTER, EVEN, ODD).
filter	Which band filter to use (LBA_10_90, LBA_30_70, HBA_110_190, HBA_170_230, HBA_210_250).
dithering.enabled	Whether to add analog dithering noise to increase linearity. (optional)
dithering.power	Power (in dB) to apply for dithering (-4.0 to -25.0). (optional)
dithering.frequency	Dithering frequency (in Hz). (optional)
SAPs	List of pointings and frequencies (subbands) to track and beam form.
HBA.DAB_filter	Enable the analog filter on the RCUs for DAB radio frequencies. (optional)
HBA.tile_beam	Pointing to track with the HBA tiles (optional). (specify for HBA)

This will configure the specified antenna field (f.e. `HBA`) as follows:

- `STAT/DigitalBeam/HBA` is configured to beam form the antennas in the specified `antenna_set`, track all pointings given in `SAPs[x].pointing`, and produce beamlets for all subbands in `SAPs[x].subbands`. The beamlets mirror the subbands in the order in which they are specified,
- The `observation_id` is used to annotate the beamlet data produced by this observation,
- `STAT/AFH/HBA` is configured to use the specified `filter` for the RCUs,
- `STAT/TileBeam/HBA` is configured to beam form all HBA tiles, tracking the given `tile_beam` pointing.

21.1.1 Observation Output

The effect of the observations can be observed through the following means, all of which are managed independently from the observation:

- The beamlets streaming out of the station towards the processing cluster. The `Beamlet` device is responsible for managing and monitoring this data flow,
- The statistics streaming out of the station towards the control softwate. The `XST/SST/BST` devices are responsible, and allow inspection of this data flow,
- The various input signal monitoring points available in the `SDP` device, such as `FPGA_input_signal_mean_RW`.

21.1.2 Life cycle

The ObservationControl device will start each Observation when its start time is reached or past, and will stop it at the specified stop time. You can also force this to happen:

```
obs_control = DeviceProxy("STAT/ObservationControl/1")
obs_control.start_observation_now(12345) # starts observation 12345 now, regardless of␣
↪its specified start time
obs_control.stop_observation_now(12345)  # stops observation 12345 now, regardless of␣
↪its specified stop time
```

21.2 Managing observation(s)

To manage running observations, we can interact with ObservationControl:

```
>>> # Check which observations are known (running or yet to run)
>>> obs_control.observations_R
array([12345])

>>> # Check which observations are running
>>> obs_control.running_observations_R
array([12345])

>>> # Stop a running observation
>>> obs_control.stop_observation_now(12345)

>>> # Stop all running observations
>>> obs_control.stop_all_observations_now()
```

Alternatively, we can inspect a running observation more closely. Each observation is represented by its own device: `STAT/Observation/$id`, so if observation 12345 has been started, we can do the following:

```
observation = DeviceProxy("STAT/Observation/12345")
```

This device exposes its settings as individual attributes, as well as:

alive_R
Ever-increasing value as long as the observation is running. Allows one to check whether monitoring has become stale.

type
`int`

observation_settings_RW
JSON string of the specifications of this observation. NB: This attribute cannot be written once the observation has started.

type
`str`

observation_id_R
(et al) Each specification parameter can be retrieved individually.

type
(depends on specification parameter)

CHAPTER

TWENTYTWO

SIGNAL CHAIN

The station hardware collectively processes the analog signals received by the antena dipoles, resulting in either statistics (SST/BST/XST) or beamlets. This signal chain can be monitored as it flows through the hardware as follows:

22.1 RECV: Data reception

The RCU boards can receive input from three sources: an LBA, an HBA tile, and a signal or noise generator.

A typical station has `rcu == 32` RCUs, each of which has `antenna == 3` inputs.

22.1.1 Input

- `recv.RCU_PWR_ANT_on_R[rcu][antenna]` indicates whether each antenna is powered. If not, the RCU will emit *zeroes* if an LBA or HBA tile is attached.
- `recv.RCU_PWR_ANALOG_on_R[rcu]` indicates whether the analog power is enabled to each RCU. If not, the RCU will emit *zeroes* if an LBA or HBA tile is attached.
- `recv.RCU_PWR_DIGITAL_on_R[rcu]` indicates whether the digital power is enabled to each RCU. If not, the RCU will emit *zeroes*.

22.1.2 Processing

- `recv.RCU_band_select_R[rcu][antenna]` indicates which band is selected for each antenna (1 = 10MHz, 2 = 30MHz), which affects its sensitivity.
- `recv.RCU_attenuator_dB_R[rcu][antenna]` is the attenuation for each antenna, which affects its *amplitude*.
- `recv.RCU_DTH_ON_R[rcu][antenna]` indicates whether the dither source is on, which affects the signal quality:
  - `recv.RCU_DTH_freq_R[rcu][antenna]` is the frequency of the dither source, in Hz.

22.2 SDP: Digital signal processing

The SDP can process three kinds of input: antenna data, generated waveforms, and no input, and process this into four kinds of output: beamlets, BSTs, SSTs, and XSTs.

A typical station has `fpga == 16` FPGAs, each of which has `input == 12` inputs.

22.2.1 Input

- `sdp.FPGA_wg_enable_R[fpga][input]`, indicates whether waveforms are generated (True) or antenna input is used (False):
  - `sdp.FPGA_wg_frequency_R[fpga][input]` indicates the frequency of the generated wave,
  - `sdp.FPGA_wg_amplitude_R[fpga][input]` indicates the amplitude of the generated wave,
  - `sdp.FPGA_wg_phase_R[fpga][input]` indicates the phase of the generated wave.
- `sdp.FPGA_signal_input_mean_R[fpga][input]` shows the input signal strength compared to full scale (FS) = 8192.
- `sdp.FPGA_signal_input_rms_R[fpga][input]` shows the root means square of the input.

The signal input mean and rms behave as follows:

Input	Configuration	Signal Mean	Signal RMS
None		0	0
Waveform Generator	frequency = 0	amplitude * sin(phase)	amplitude * 8192 / 2
Waveform Generator	frequency > 0	0	amplitude * 8192 / 2
Antenna		> 0	> 0

22.2.2 Processing

- `sdp.FPGA_processing_enable_R[fpga]` indicates whether the FPGA processes its input. If not, *zeroes* are produced for all outputs.
- `sdp.FPGA_signal_input_samples_delay_R[fpga][input]` indicates a per-input delay to be applied, in units of 5 ns. This results in a frequency-dependent *phase* change of the input.
- `sdp.FPGA_subband_weights_R[fpga][input * subband]` indicates a per-subband and per-input weight factor. 8192 is unit weight, 0 means the input will be erased. Anything else results in a *phase* and/or *amplitude* change of the input.

22.2.3 SST output

- `sst.FPGA_sst_offload_enable_R` indicates whether SSTs are emitted at all.
- `sst.nof_valid_payloads_R[fpga]` is the number of packets received from each FPGA.
- `sst.sst_R[fpga * input][subband]` is the *amplitude* of the signal over the configured integration interval:
  - `sst.FPGA_sst_offload_weighted_subbands_R[fpga * input]` indicates whether the `sdp.FPGA_subband_weights_R` are applied when calculating the SSTs,
  - `sst.integration_interval_R[fpga * input]` is the integration interval of the provided SSTs,

   - `sst.sst_timestamp_R[fpga * input]` is when the SSTs were received,
   - `sst.last_packet_timestamp_R` is when the last SST from any FPGA was received.

If the SSTs are not received, or filled with zeroes, see also *I am not receiving any XSTs and/or SSTs from SDP!*.

22.2.4 XST output

- `xst.FPGA_xst_offload_enable_R` indicates whether XSTs are emitted at all.
- `xst.FPGA_xst_processing_enable_R` indicates whether XSTs are computed. If not, *zeroes* are produced.
- `xst.nof_valid_payloads_R[fpga]` is the number of packets received from each FPGA.
- `xst.xst_phase_R[fpga * input][fpga * input]` is the *phase* angle between each pair of inputs, and is defined only for `[a][b]` with `a <= b`:
   - `xst.FPGA_xst_subband_select_R[fpga][8]` contains the subband for which to compute the XSTs. Currently, one subband is supported, which should be on index `[fpga][1]`,
   - `xst.FPGA_integration_interval_R[fpga]` is the integration interval for the XSTs,
   - `xst.xst_timestamp_R[136]` is when the XSTs were received, per block (see below),
   - `xst.last_packet_timestamp_R` is when the last XST from any FPGA was received.
- `xst.xst_amplitude_R[fpga * input][fpga * input]` is the correlated *amplitude* between two inputs, and is subject to the same restrictions as `xst.xst_phase_R`.

If the XSTs are not received at, or filled with zeroes, see also *I am not receiving any XSTs and/or SSTs from SDP!*.

Each block contains 12x12 XSTs, and are indexed in the same order baselines are, see https://git.astron.nl/lofar2.0/tango/-/blob/master/tangostationcontrol/tangostationcontrol/common/baselines.py on how to convert baseline indices to and from input pairs.

CHAPTER

TWENTYTHREE

INSTRUMENT CALIBRATION

The signal path lengths and sensitivity differ per antenna, due to factors including:

- Wear and tear of the antennas and cables,
- Differences in cable length between antenna and RCU,
- Differences in signal path lengths within the processing equipment.

The signals thus need to adjusted with respect to each other in order to align their phases and amplitudes. These per-antenna *calibration values* are split into the following parts to apply them:

- `recv.RCU_attenuator_dB_RW`: Coarse attenuation of each antenna input in the RCU, in dB,
- `sdp.FPGA_signal_input_samples_delay_RW`: Coarse delay added to each antenna input in the SDP, in samples,
- `sdp.FPGA_subband_weights_RW`: Fine attenuation & delay of each antenna input in the SDP, as a complex multiplication factor per antenna per subband.

These signal differences are frequency dependent. To address this, we maintain different models for signals around the reference frequencies of 50 MHz (LBA), and 150, 200, and 250 MHz (HBA). The calibration subsystem uses the `antennafield.Frequency_Band_RW` attribute to determine the current reference frequency for each antenna:

Antenna type	Frequency band	`antennafield.Frequency_Band_RW`	Clock	`recv.RCU_band_select_RW`	Reference frequency
LBA	10 - 90 MHz	`LBA_10_90` / `LBA_10_70`	(any)	1	50 MHz
LBA	30 - 90 MHz	`LBA_30_90` / `LBA_30_70`	(any)	2	50 MHz
HBA	110 - 190 MHz	`HBA_110_190`	200 MHz	2	150 MHz
HBA	170 - 230 MHz	`HBA_170_230`	160 MHz	1	200 MHz
HBA	210 - 240 MHz	`HBA_210_250`	200 MHz	4	250 MHz

23.1 Mathematical Background

We equalise the signals of the different antennas to compensate for the delay and attenuation effects, in two steps: coarse and fine. The following table describes what is corrected for where:

Effect	Granularity	Compensation	How
Delay	Coarse	`sdp.FPGA_signal_input_samples_delay_RW`	Delaying using a ring buffer
Delay	Fine	`sdp.FPGA_subband_weights_RW`	Phase shifts
Attenuation	Coarse	`recv.RCU_attenuator_dB_RW`	Dampening whole dBs
Attenuation	Fine	`sdp.FPGA_subband_weights_RW`	Amplitude scaling

The *coarse delay compensation* is done in SDP, by delaying all inputs to line up with the latest arriving one. The FPGAs do this through a *sample shift*, in which the samples from each input is delayed a fixed number of samples. At the 200 MHz clock, samples are 5 ns. The sample shift aligns the inputs with a remaining difference of +/- 2.5 ns.

This remainder is corrected for in the *fine delay compensation*, by shifting the phases of each input backwards. A phase shift is frequency dependent (`-2pi * frequency * delay`), and is thus applied at the higher frequency resolution after creating subbands. The `FPGA_subband_weights_RW` in SDP allows us to configure a complex correction factor for each subband from each input. A phase shift `phi` is converted into a complex factor through `cos(phi) + i * sin(phi)`.

Note: The delay compensation shifts all antenna signals by a fixed amount: the number of samples to delay to line up with the longest cable. Yet we mark those signals as "now" in SDP. This introduces a temporal shift of the order of 200ns. This is deemed acceptable, as after the station FFT (that creates the subbands), we have 5.12ms samples, which is an order of magnitude higher time scale.

The *coarse loss compensation* is done in RECV on the RCU, which can attenuate each input an integer number of decibels. We attenuate each signal to line up with the weakest. The remaining attenuation is +/- 0.5 dB.

The remainder is corrected for in the *fine loss compensation*, by applying an amplitude scaling factor (`10^(-dB/10)`) as part of the complex `FPGA_subband_weights_RW` (see above). This scaling factor is the same for all subbands.

23.2 Configuration

The following properties describe the AntennaField for calibration purposes:

Antenna_Cables
Encodes which cable type is attached to each antenna in the field, as described in dict `common.cables.cable_types`.

type
`str[N_antennas]`

Field_Attenuation
Attenuation to apply to all the antennas, on top of the cable model, to align this antennafield with other fields.

type
`float64`

23.3 Coarse Corrections

Both the coarse attenuation and delay corrections are caused by the difference in cable lengths: longer cables result in more delay, and more loss of signal. We maintain a cable model in the dict `common.cables.cable_types`, which describes the delay introduced by each cable, as well as the loss at each of our modelled frequencies.

The coarse corrections are the rounded versions of these differences. The rounding errors, as well as the subtle differences between the individual cables of the same type are compensated for in the fine corrections below. The AntennaField exposes the following attributes to inspect the configuration and the computed calibration values:

Antenna_Cables_R
The type of cable connected to each antenna.

type
`str[N_antennas]`

Antenna_Loss_R
The loss introduced by each cable, according to the cable model, in dB, for the currently selected frequency.

type
`float64[N_antennas]`

Antenna_Delay_R
The delay introduced by each cable, according to the cable model, in seconds.

type
`float64[N_antennas]`

Calibration_SDP_Signal_Input_Samples_Delay_R
The delay which is to be applied to both polarisations of each antenna, in samples.

type
`uint32[N_antennas]`

Calibration_RCU_Attenuation_dB_R
The attenuation to apply to each antenna, in (integer) dB.

type
`uint32[N_antennas]`

23.4 Fine Corrections

The fine attenuation and delay corrections are caused by both known and unknown differences between the antennas. The known differences are the remainders from the cable model, left after the coarse corrections have been applied. The fine corrections are applied in SDP as *subband weights*, which are complex multiplication factors for each subband for each input.

The AntennaField exposes the known corrections as:

Calibration_SDP_Fine_Calibration_Default_R
Computed fine calibration values, as a tuple (delay, phase_offset, amplitude_scaling).

type
`float64[N_antennas * N_pol][3]`

Calibration_SDP_Subband_Weights_Default_R
Computed fine calibration values as subband weights (complex values).

type
`float64[N_antennas * N_pol][N_subbands * VALUES_PER_COMPLEX]`

To also cover the unknown differences between the antennas, the correct subband weights are actually measured and stored in *calibration tables*. These values then cover both the known and the unknown corrections. The AntennaField exposes the actual subband weights it will apply through:

Calibration_SDP_Subband_Weights_R
Fine calibration values as subband weights (complex values).

type
`float64[N_antennas * N_pol][N_subbands * VALUES_PER_COMPLEX]`

The individual calibration tables for each frequency are provided through:

Calibration_SDP_Subband_Weights_50MHz_R
Fine calibration values as subband weights, for 50MHz input signals.

Calibration_SDP_Subband_Weights_150MHz_R
Fine calibration values as subband weights, for 150MHz input signals.

Calibration_SDP_Subband_Weights_200MHz_R
Fine calibration values as subband weights, for 200MHz input signals.

Calibration_SDP_Subband_Weights_250MHz_R
Fine calibration values as subband weights, for 250MHz input signals.

type
`float64[N_antennas * N_pol][N_subbands * VALUES_PER_COMPLEX]`

23.5 Managing Calibration Tables

The calibration tables for SDP are stored in the HDF5 file format, described at XXX, and easily read and written in Python by using the `common.calibration_table.CalibrationTable` class in this package, or with the more generic `h5py` Python package. Each file is typically named `CalTable-CS001-HBA0-150MHz.h5`, and is thus specific for an antenna field and the frequency band used to determine it. Each file contains the subband weights, as well as metadata on how and when they were determined.

The AntennaField device reads these files from disk, maintained in a dedicated Docker volume. New files can be downloaded from a central location on demand, providing the follow functionality:

Calibration_Table_Base_URL
Property which contains the root URL for the calibration tables. The remote location of a calibration table is f.e. `{Calibration_Table_Base_URL}/CS001/CalTable-CS001-HBA0-150MHz.h5`.

download_calibration_tables()
Command to download and apply the latest calibration tables, caching them in the Docker volume.

calibrate()
Command to apply the calibration tables present in the Docker volume.

23.6 Applying Calibration Values

The following commands in AntennaField upload new calibration values to the signal chain in RECV and SDP:

calibrate_recv()
: Configure `recv.RCU_attenuator_dB_RW` for the antennas in the field.

calibrate_sdp()
: Configure `sdp.FPGA_signal_input_samples_delay_RW` and `sdp.FPGA_subband_weights_RW` for the antennas in the field.

Since both calibrations depend on the frequency of the signals, the above commands are automatically called when the attribute `antennafield.Frequency_Band_RW` is written.

CHAPTER TWENTYFOUR

CELESTIAL & GEODETIC CALIBRATION

The `TileBeam` and `DigitalBeam` devices use python-casacore to compute the direction of a given pointing with respect to our antennas and reference positions. Casacore in turn uses *measures* tables for the precise measurements of celestial positions, geodetical information, and time calibrations (f.e. leap seconds). These tables need to be installed and periodically updated to maintain the pointing accuracy:

measures_directory_R
: Directory of the active set of measures tables. The directory name includes the timestamp denoting their age.

    type
    `str`

measures_directories_available_R
: List of installed sets of measures tables.

    type
    `str[64]`

download_measures()
: Download (but do not activate) the latest measures tables from ftp://ftp.astron.nl/outgoing/Measures/WSRT_Measures.ztar. Returns the directory name in which the measures were installed.

    returns
    `str`

use_measures(dir)
: Activate the measures tables in the provided directory. This necessitates turning off and restarting the TileBeam device, so the command will always appear to fail. Turn the device back and the selected measures tables will be active.

    returns
    `(does not return)`

CHAPTER

TWENTYFIVE

BROKEN HARDWARE

Not all hardware is always functional. Broken hardware must be excluded from the signal chain, and in some cases prevented from powering up.

25.1 Disabling antennas

Not all antennas present in the field are to be used. The AntennaField device exposes the following properties for each of its antennas:

Antenna_Quality
The condition of the antenna: 0=OK, 1=SUSPICIOUS, 2=BROKEN, 3=BEYOND_REPAIR.

type
`int32[]`

Antenna_Use
Whether each antenna should be used: 0=AUTO, 1=ON, 2=OFF. In AUTO mode, an antenna is used if its quality is OK or SUSPICIOUS. In ON mode, it is always used. In OFF mode, never.

type
`int32[]`

which can also be queried as `Antenna_Quality_R` and `Antenna_Use_R`.

Note: If these properties are updated, you should restart both the AntennaField and DigitalBeam device to propagate their effects.

The above settings result in a subset of the antennas in the AntennaField to be marked as usable. The following property exposes this conclusion:

Antenna_Usage_Mask_R
Whether antennas will be used, according to their configured state and quality. Antennas which are configured to be BROKEN, BEYOND_REPAIR, or OFF, are not used.

type
`bool[N_tiles]`

25.1.1 Effect on signal chain

The DigitalBeam device will only beamform inputs that are enabled in the `AntennaField.Antenna_Usage_Mask_R` attribute.

CHAPTER

TWENTYSIX

POWER DISTRIBUTION

At boot, during hardware initialisation, the following devices toggle power:

- The RECV device turns all RCUs enabled in `RCU_mask_RW` OFF and ON,
- The RECV device powers its antennas according to its `RCU_PWR_ANT_on_RW_default` property,
- The AntennaField device powers its antennas, if they are: * Enabled in `Antenna_Usage_Mask_R` attribute, that is, not marked as BROKEN, BEYOND_REPAIR, or OFF, * Enabled in the `Antenna_Needs_Power` property.

Note: Exotic inputs like a noise source must not receive power, even when used. Use the `Antenna_Needs_Power` property to configure which antennas should be powered on.

CHAPTER

TWENTYSEVEN

DEVELOPER INFORMATION

This chapter describes key areas useful for developers.

27.1 Environment variables

Several environment variables fundamentally control the deployment and development environment. These include:

- *TANGO_HOST*
- *TANGO_STATION_CONTROL*
- *TANGO_SKIP_BUILD*

Firstly, *TANGO_HOST* should point to the tango database server including its port. An example would be *10.14.0.205:10000*. If *TANGO_HOST* is not set instead *tango.service.consul:10000* is used.

Finally *TANGO_STATION_CONTROL* can be used to control if device containers should build software from source (developer mode). Or if the software should be built into the *lofar-device-base* docker image directly. If *TANGO_STATION_CONTROL* is set the makefile will build a wheel package which will be installed into the docker image.

If instead a particular wheel package needs to be installed *TANGO_SKIP_BUILD* can be set as well. Be sure the wheel package is placed in the *tangostationcontrol/dist/* directory.

In the future the actual value of the *TANGO_STATION_CONTROL* variable might be used to control various types of different behavior.

27.2 Docker

Docker containers are build using `make` in the `docker` directory. Key commands are:

- `make <container>` to build the image for the container,

Since the *Python code is taken from the host when the container starts*, restarting is enough to use the code you have in your local git repo. Rebuilding is unnecessary. Docker networking ——————————-

The Docker containers started use a consul based *virtual network* to communicate among each other. This means that:

- Containers address each other by a service name as defined in the job file (f.e. `tango.service.consul` for the TANGO_HOST),
- `localhost` can only be used within the containers to access other containers, if sidecar proxy is used.
- Most ports are dynamically allocated. It will be mapped to the right port within the container.

27.2.1 CORBA

Tango devices use CORBA, which require all servers to be able to reach each other directly. Each CORBA device opens a port and advertises its address to the CORBA broker. The broker then forwards this address to any interested clients. A device within a docker container cannot know under which name it can be reached, however, and any port opened needs to be exposed explicitly in the docker-compose file for the device. To solve all this, we *assign a unique port to each device*, and explictly tell CORBA to use that port, and what the hostname is under which others can reach it. Each device thus has these lines in their compose file:

```
ports:
  - "5701:5701" # unique port for this DS
entrypoint:
  # configure CORBA to _listen_ on 0:port, but tell others we're _reachable_ through $
↪{HOSTNAME}:port, since CORBA
  # can't know about our Docker port forwarding
  - python3 -u /opt/lofar/tango/devices/devices/sdp/sdp.py STAT -v -ORBendPoint␣
↪giop:tcp:0:5701 -ORBendPointPublish giop:tcp:${HOSTNAME}:5701
```

Specifying the wrong `$HOSTNAME` or port can make your device unreachable, even if it is running. Note that `$HOSTNAME` is advertised as is, that is, it is resolved to an IP address by any client that wants to connect. This means the `$HOSTNAME` needs to be correct for both the other containers, and external clients.

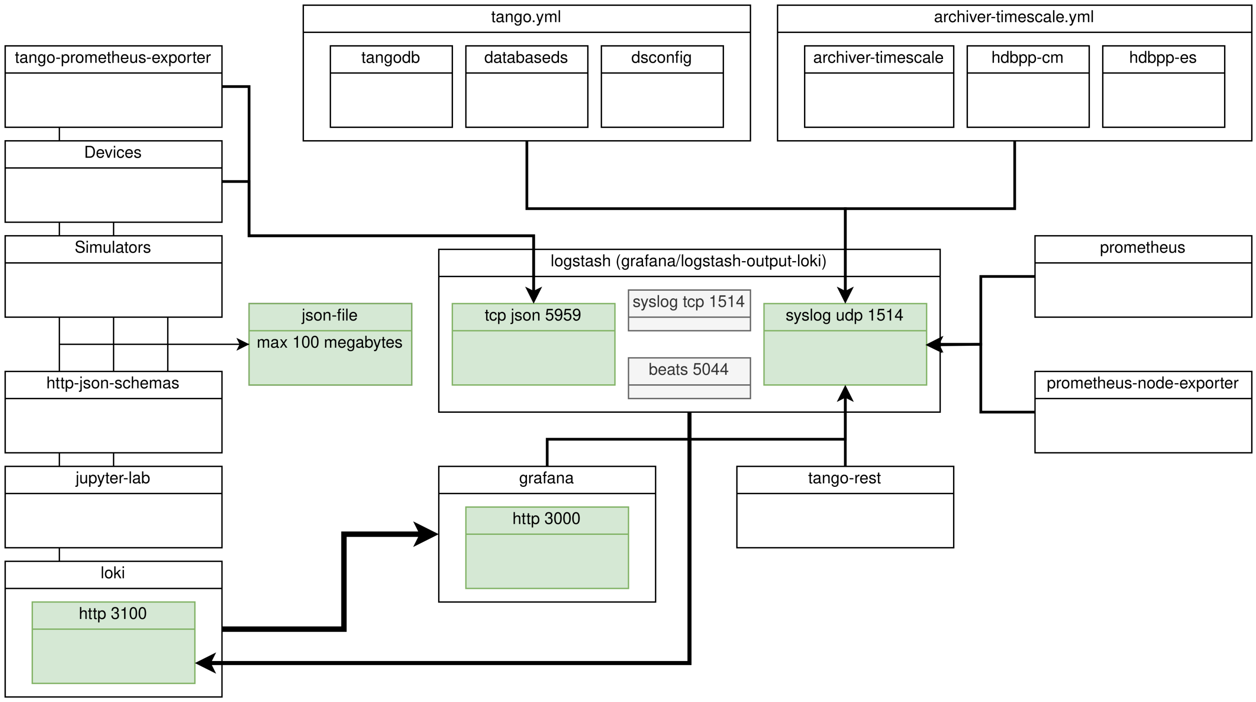
The `docker-compose/Makefile` tries to set a good default for `$HOSTNAME`, but you can override it by exporting the environment variable yourself (and run `make restart <container>` to effectuate the change).

For more information, see:

- https://huihoo.org/ace_tao/ACE-5.2+TAO-1.2/TAO/docs/ORBEndpoint.html
- http://omniorb.sourceforge.net/omni42/omniNames.html
- https://sourceforge.net/p/omniorb/svn/HEAD/tree/trunk/omniORB/src/lib/omniORB/orbcore/tcp/tcpEndpoint.cc

27.3 Logging

Overview of the data flow between docker services to facilitate logging



The Logstash pipeline collects the logs from the containers, as well as any external processes that send theirs. The following interfaces are available for this purpose:

Interface	Port	Note
Syslog	1514/udp	Recommended over TCP, as the Logstash pipeline might be down.
Syslog	1514/tcp	
JSON	5959/tcp	From python, recommended is the LogStash Async module.
Beats	5044/tcp	Use FileBeat to watch logs locally, and forward them to Loki.

We recommend making sure the contents of your log lines are parsed correctly, especially if logs are routed to the *Syslog* input. These configurations are stored in `docker-compose/logstash/loki.conf`.

27.3.1 Log from Python

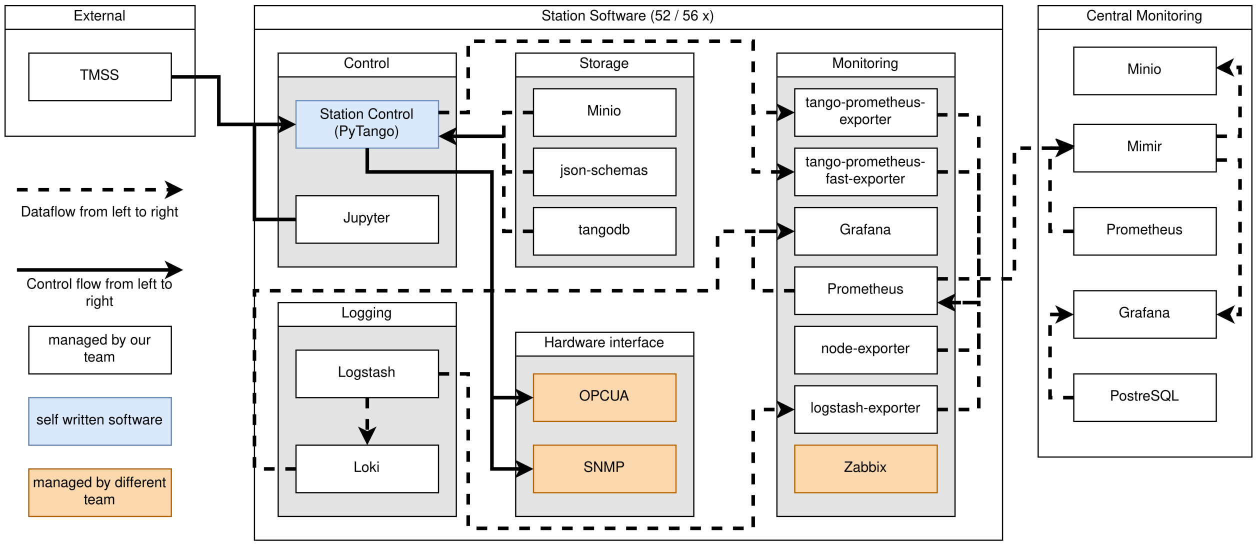
The `common.lofar_logging` module provides an easy way to log to Loki through Logstash from a Python Tango device.

27.3.2 Log from Docker

Not all Docker containers run our Python programs, and can forward the logs themselves. For those, we use the `syslog` log driver in Docker. Extend the `docker compose` files with:

Logs forwarded in this way are provided with the container name, their timestamp, and a log level guessed by Docker. It is thus wise to parse the message content further in Logstash (see above).

27.4 Services



CHAPTER

TWENTYEIGHT

FAQ

28.1 Connecting to devices

28.1.1 My device is unreachable, but the device logs say it's running fine?

The `$HOSTNAME` may have been incorrectly guessed by `docker-compose/Makefile`, or you accidently set it to an incorrect value. If you have `$HOSTNAME` set in the shell running `make`, try:

```
unset HOSTNAME
make build
make stop
make start
```

If this does not work, you need to set `$HOSTNAME` to something that resolves to your machine, both for external parties and for docker containers. See *CORBA*.

28.1.2 I get "API_CorbaException: TRANSIENT CORBA system exception: TRANSIENT_NoUsableProfile" when trying to connect to a device?

See the previous answer.

28.2 Docker

28.2.1 How do I prevent my containers from starting when I boot my computer?

You have to explicitly stop a container to prevent it from restarting. Use:

```
cd docker-compose
make stop <container>
```

or plain `make stop` to stop all of them.

28.3 Windows

28.3.1 How do I develop from Windows?

Our setup is Linux-based, so the easiest way to develop is by using WSL2, which lets you run a Linux distro under Windows. You'll need to:

- Install WSL2. See f.e. https://www.omgubuntu.co.uk/how-to-install-wsl2-on-windows-10
- Install Docker Desktop
- Enable the WSL2 backend in Docker Desktop
- We also recommend to install Windows Terminal

28.3.2 How do I run X11 applications on Windows?

If you need an X11 server on Windows:

- Install VcXsrv
- Disable access control during its startup,
- Use `export DISPLAY=host.docker.internal:0` in WSL.

You should now be able to run X11 applications from WSL and Docker. Try running `xterm` or `xeyes` to test.

28.4 SSTs/XSTs

28.4.1 I am not receiving any XSTs and/or SSTs from SDP!

Are you sure?

- Packets are arriving if `sst.nof_packets_received` / `xst.nof_packets_received` is increasing,
- Packets are sent by SDP if `sst.FPGA_sst_offload_nof_packets_R` / `xst.FPGA_xst_offload_nof_packets_R` is increasing.

In general, the settings ought to be correct after the following:

The `sdp.set_defaults()` command, followed by `sst.set_defaults()` / `xst.set_defaults()`, should reset that device to its default settings, which should result in a working system again. Also, check the following settings:

- `sdpfirmware.TR_fpga_mask_RW[x] == True`, to make sure we're actually configuring the FPGAs,
- `sdp.FPGA_communication_error_R[x] == False`, to verify the FPGAs can be reached by SDP.
- `sdp.FPGA_processing_enabled_R[x] == True`, to verify that the FPGAs are processing, or the values and timestamps will be zero,
- `sdp.FPGA_signal_input_bsn_R` is increasing, to verify that the FPGA processing is subject to the clock.

28.4.2 The SDP is not sending SST/XST packets!

Packets are sent if `sst.FPGA_sst_offload_nof_packets_R` / `xst.FPGA_xst_offload_nof_packets_R` is increasing. If not, check these settings:

- SSTs:
  - `sst.FPGA_sst_offload_enable_RW[x] == True`, to verify that the FPGAs are actually emitting the SSTs,
- XSTs:
  - `xst.FPGA_xst_offload_enable_RW[x] == True`, to verify that the FPGAs are actually emitting the SSTs,
  - `xst.FPGA_xst_processing_enable_RW[x] == True`, to verify that the FPGAs are actually producing the SSTs,

28.4.3 Some SSTs/XSTs packets do arrive, but not all, and/or the matrices remain zero?

So `sst.nof_packets_received` / `xst.nof_packets_received` is increasing, telling you packets are arriving. But they're apparently dropped or contain zeroes.

The `sst` and `xst` devices expose several packet counters to indicate where incoming packets were dropped before or during processing:

- `nof_invalid_packets_R` increases if packets arrive with an invalid header, or of the wrong statistic for this device,
- `nof_packets_dropped_R` increases if packets could not be processed because the processing queue is full, so the CPU cannot keep up with the flow,
- `nof_payload_errors_R` increases if the packet was marked by the FPGA to have an invalid payload, which causes the device to discard the packet,

If no packets are received at all, check whether they are sent to the correct address:

- SSTs:
  - sst.FPGA_sst_offload_hdr_eth_destination_mac_R[x] == <MAC of your machine's mtu=9000 interface>, or the FPGAs will not send it to your machine. Use f.e. `ip addr` on the host to find the MAC address of your interface, and verify that its MTU is 9000,
  - sst.FPGA_sst_offload_hdr_ip_destination_address_R[x] == <IP of your machine's mtu=9000 interface>, or the packets will be dropped by the network or the kernel of your machine,
  - `sst.FPGA_sst_offload_hdr_udp_destination_port_R[x] == 5001`, or the packets will not be sent to a port that the SST device listens on.
- XSTs:
  - xst.FPGA_xst_offload_hdr_eth_destination_mac_R[x] == <MAC of your machine's mtu=9000 interface>, or the FPGAs will not send it to your machine. Use f.e. `ip addr` on the host to find the MAC address of your interface, and verify that its MTU is 9000,
  - xst.FPGA_xst_offload_hdr_ip_destination_address_R[x] == <IP of your machine's mtu=9000 interface>, or the packets will be dropped by the network or the kernel of your machine,
  - `xst.FPGA_xst_offload_hdr_udp_destination_port_R[x] == 5002`, or the packets will not be sent to a port that the XST device listens on.

If this fails, see the next question.

28.4.4 I am still not receiving XSTs and/or SSTs, even though the settings appear correct!

Let's see where the packets get stuck. Let us assume your MTU=9000 network interface is called `em2` (see `ip addr` to check):

- Check whether the data arrives on `em2`. Run `tcpdump -i em2 udp -nn -vvv -c 10` to capture the first 10 packets. Verify:
  - The destination MAC must match that of `em2`,
  - The destination IP must match that of `em2`,
  - The destination port is correct (5001 for SST, 5002 for XST),
  - The source IP falls within the netmask of `em2` (unless `net.ipv4.conf.em2.rp_filter=0` is configured),
  - TTL >= 2,
- If you see no data at all, the network will have swallowed it. Try to use a direct network connection, or a hub (which broadcasts all packets, unlike a switch), to see what is being emitted by the FPGAs.
- Check whether the data reaches user space on the host:
  - Turn off the `sst` or `xst` device. This will not stop the FPGAs from sending.
  - Run `nc -u -l -p 5001 -vv` (or port 5002 for XSTs). You should see raw packets being printed.
  - If not, the Linux kernel is swallowing the packets, even before it can be sent to our docker container.
- Check whether the data reaches kernel space in the container:
  - Enter the docker device by running `docker exec -it device-sst bash`.
  - Run `sudo bash` to become root,
  - Run `apt-get install -y tcpdump` to install tcpdump,
  - Check whether packets arrive using `tcpdump -i eth0 udp -c 10 -nn`,
  - If not, Linux is not routing the packets to the docker container.
- Check whether the data reaches user space in the container:
  - Turn off the `sst` or `xst` device. This will not stop the FPGAs from sending.
  - Enter the docker device by running `docker exec -it device-sst bash`.
  - Run `sudo bash` to become root,
  - Run `apt-get install -y netcat` to install netcat,
  - Check whether packets arrive using `nc -u -l -p 5001 -vv` (or port 5002 for XSTs),
  - If not, Linux is not routing the packets to the docker container correctly.
- If still on error was found, you've likely hit a bug in our software.

28.4.5 Inspecting SST/XST packets

The fields `sst.last_packet_R` and `xst.last_packet_R` contain a raw dump of the last received packet for that statistic. Parsing these packets is aided greatly by using our packet parser:

```
from tangostationcontrol.devices.sdp.statistics_packet import SSTPacket, XSTPacket

# print the headers of the last received packets
print(SSTPacket(bytes(sst.last_packet_R)).header())
print(XSTPacket(bytes(xst.last_packet_R)).header())
```

28.5 Other containers

TBA

CHAPTER

TWENTYNINE

INDICES AND TABLES

- genindex
- modindex
- search