# SKA Mid CBF Engineering Console Documentation

**author**

**Sep 15, 2023**

# HOME

CHAPTER

# ONE

# OVERVIEW

The Mid CBF Engineering Console is intended for integration and testing of the Mid CBF MCS and the Talon DX hardware.

**As required, the Engineering Console will:**

- Provide emulation of LMC control of MCS
- Provide intrusive tools to monitor and control Mid CBF
- Access to configuration-managed FPGA bitstreams and Talon DX binaries for deployment to Mid CBF

See MCS-Talon Integration for further details of the integration and test work as it evolves.

Note: MCS does not currently allow its LMC interface to be externally exercised – i.e., it needs to exercised from within the Kubernetes cluster. MCS commands can be issues via an iTango3 shell running in the MCS cluster – see Engineering Console README for details.

## 1.1 System Context

The following diagram shows the Mid.CBF Engineering Console as it fits into the rest of the CSP Mid system.

## 1.2 Interfaces

# TODO

## 1.3 On Command Sequence

The On command sequence shown in the diagram below is used to automatically power on the Talon-DX boards, copy the appropriate FPGA bitsteam and HPS device server binaries to the Talon-DX boards and start the device servers on the HPS of each board. The sequence is as follows:

1. Download artefacts (bitstreams and binaries) from the Central Artefact Repository , and build the MCS Docker container after downloading. Optional: Override the DS artefacts with local builds.
2. Configure the MCS Tango database to add entries for the HPS device servers and the log consumer.
3. Use the LMC script to send the On command to the `CbfController`.
4. The `CbfController` propagates the On command to the `TalonLRU` Tango device, which then propagates it to the `PowerSwitch` device.

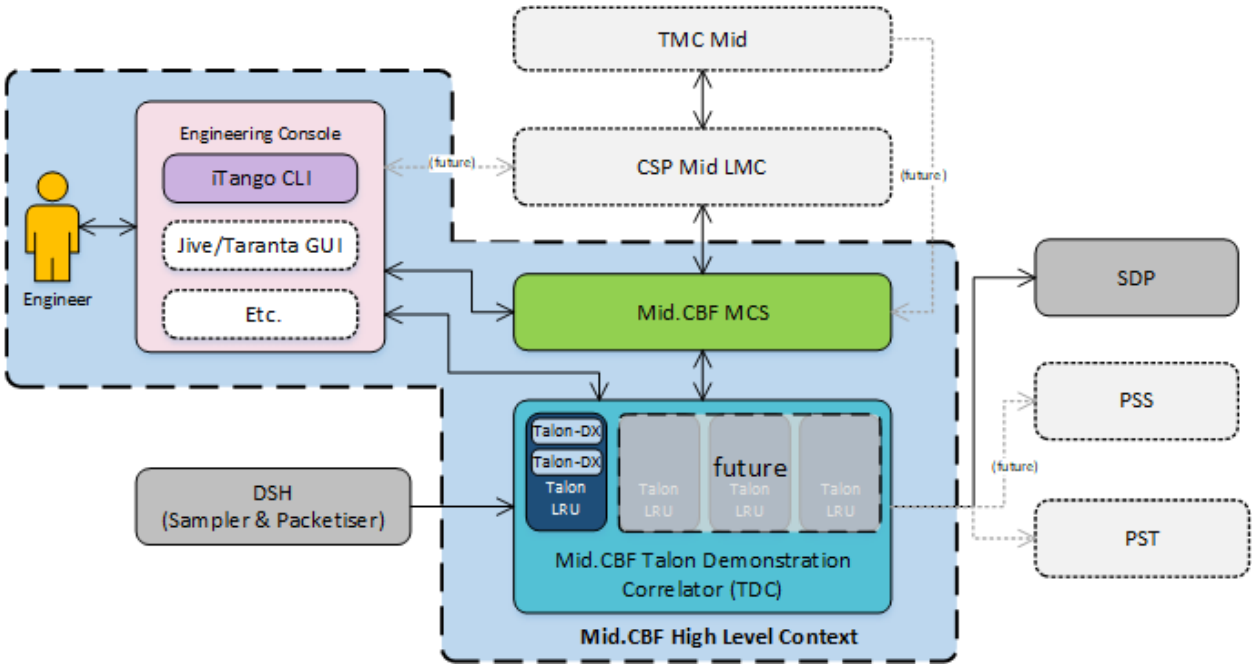



Fig. 1: Engineering Console System Context

5. The `PowerSwitch` device communicates with the power switch hardware over HTTP, requesting power on of specific outlets.
6. The power switch hardware switches the requested outlets on and responds to the `PowerSwitch` device.
7. The result of the `TalonLRU` On command is propagated back to the `CbfController`.
8. `CbfController` reads the JSON configuration file in the artefacts folder to detemine which Talon-DX boards it needs to configure.
9. FPGA bitstreams and device server binaries are copied to each Talon-DX board.
10. The `CbfController` runs the `hps_master_run.sh` script on each Talon-DX board to start the HPS Master.
11. The HPS Master device server is started on each Talon-DX board using the copied binary.
12. The `configure` command is send to each HPS Master device server.
13. The HPS Master device server programs the FPGA with the bitstream and starts the remaining HPS devices on each board.
14. HPS Master responds with success/fail result of the `configure` command.

For a description of how to run this sequence of steps see the *Configure the Talon-DX Boards from MCS* section.

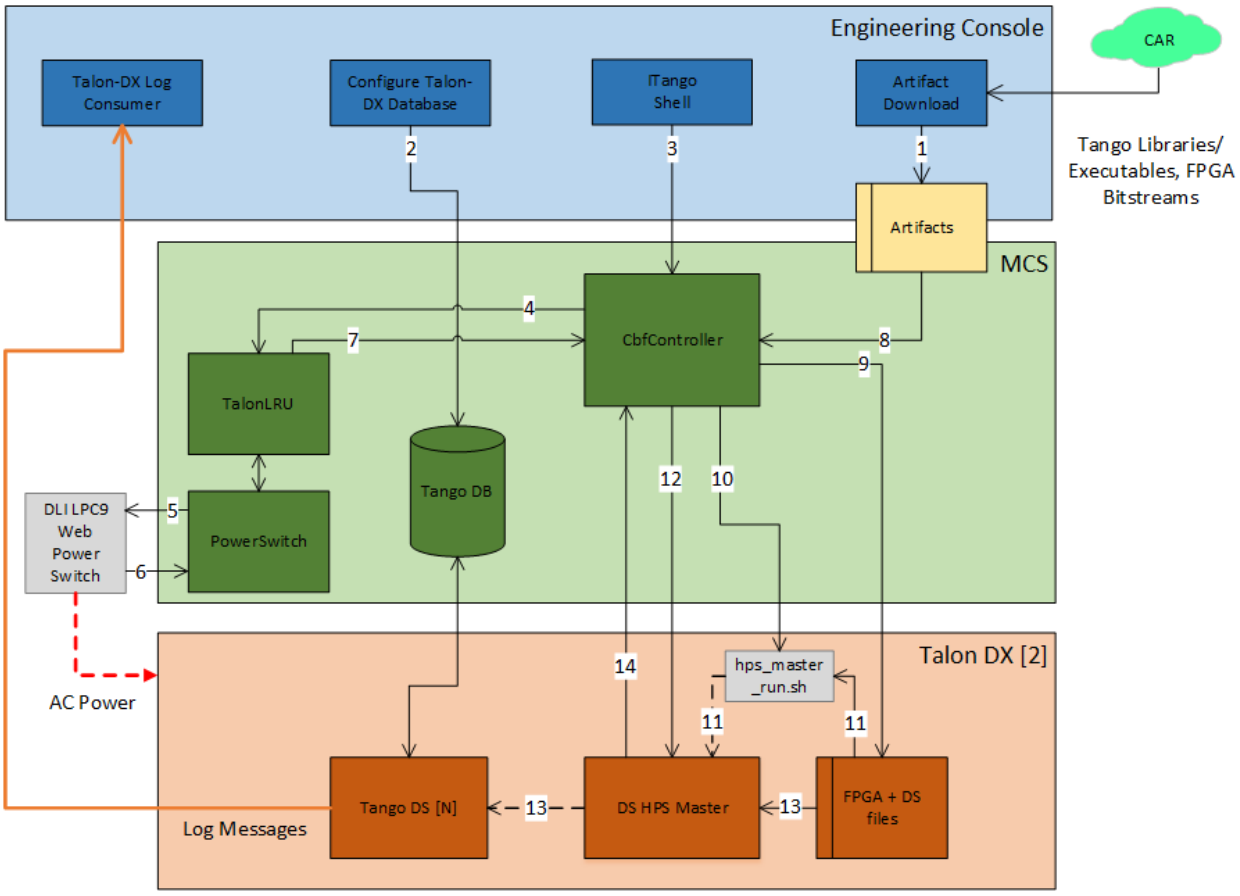
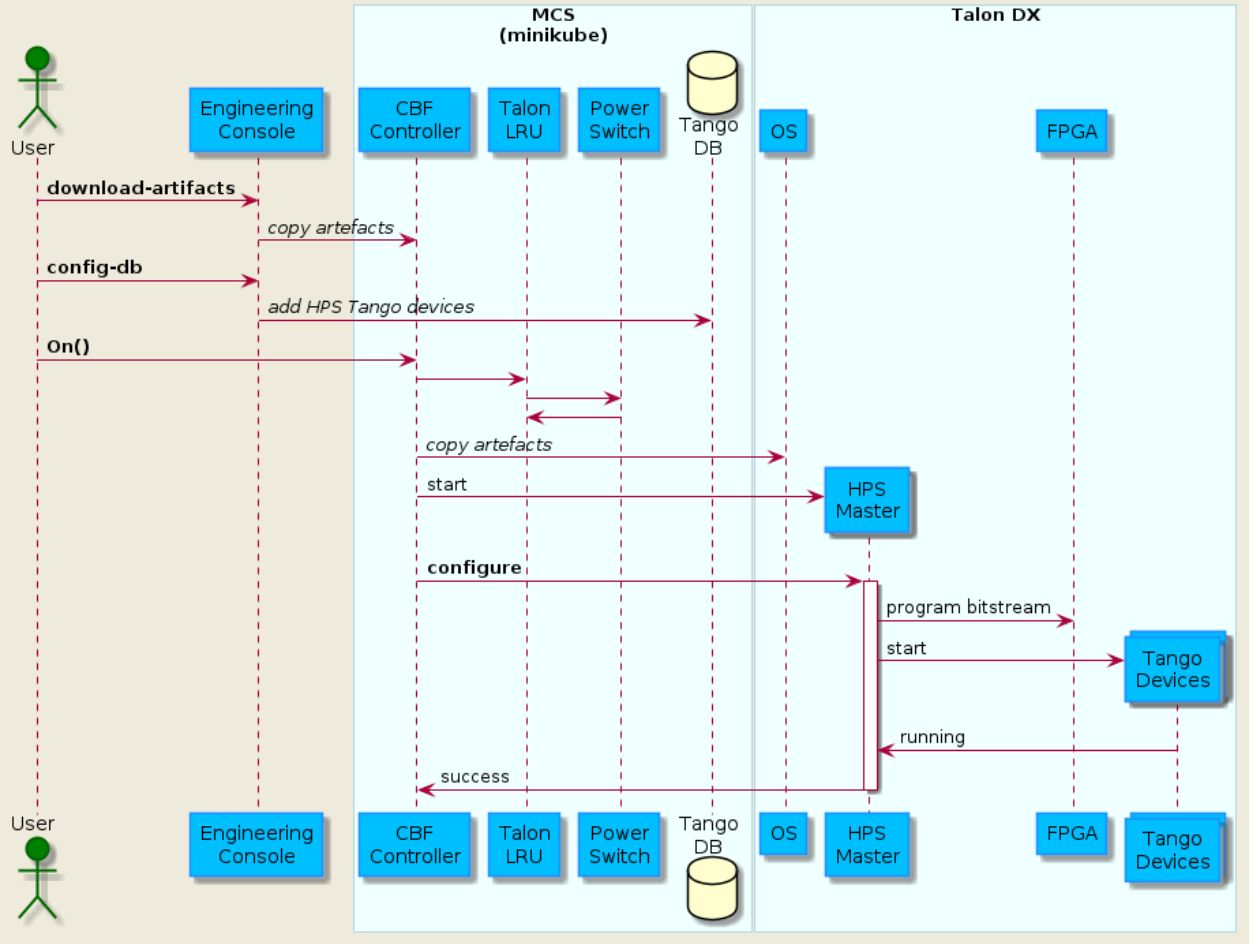


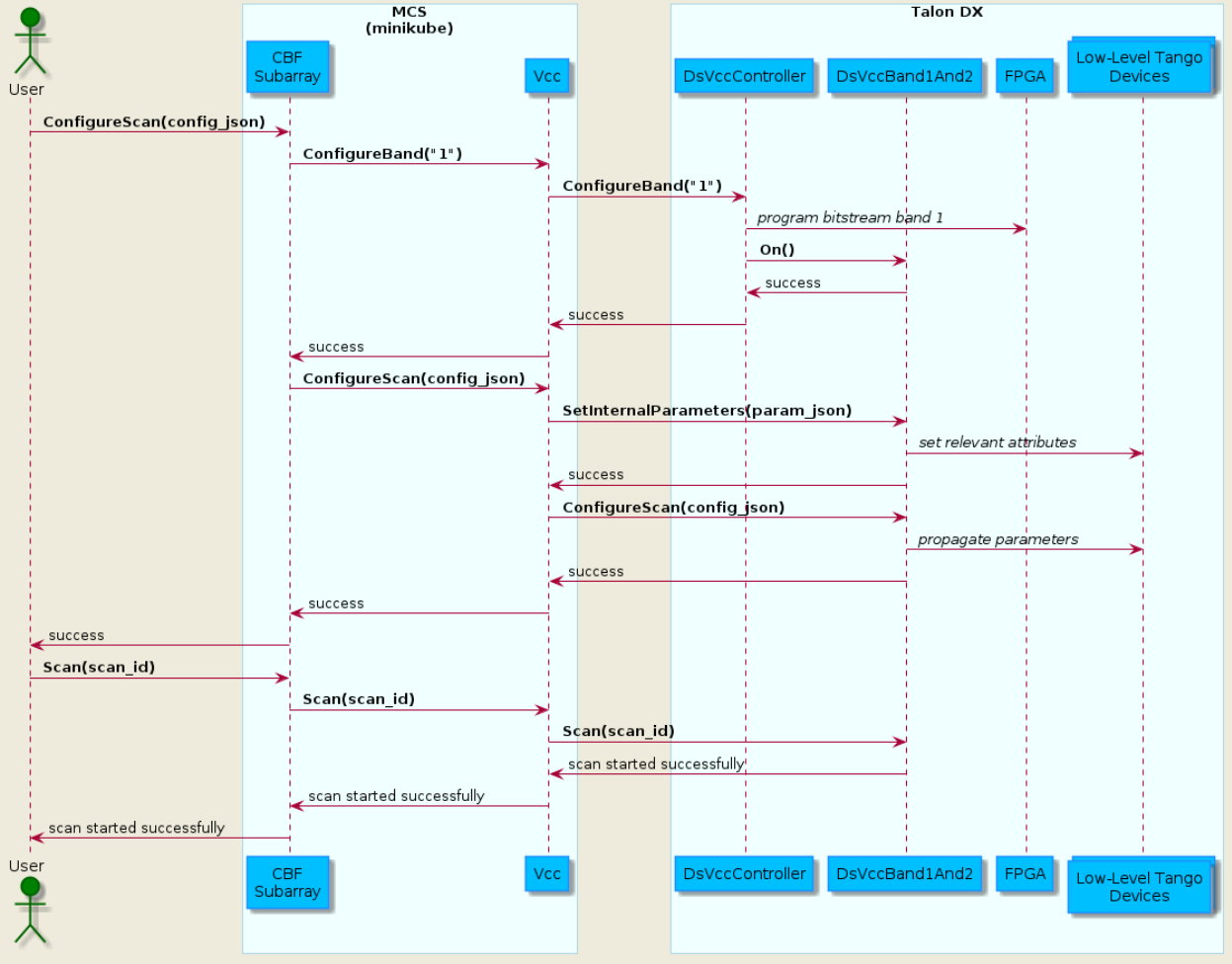

Fig. 2: MCS On Command Sequence




## 1.4 VCC Scan Sequence

Once the system has been turned on using the above sequence, it can be configured for a scan operation. The following diagram shows the flow of scan configuration and execution for a single VCC unit.

For a description of how to run this sequence of steps see the Configure and Execute a VCC Scan section.


MCS
(minikube)
Talon DX
User
CBF Subarray
Vcc
DsVccController
DsVccBand1And2
FPGA
Low-Level Tango Devices
ConfigureScan(config_json)
ConfigureBand("1")
ConfigureBand("1")
program bitstream band 1
On()
success
success
success
ConfigureScan(config_json)
SetInternalParameters(param_json)
set relevant attributes
success
ConfigureScan(config_json)
propagate parameters
success
success
success
Scan(scan_id)
Scan(scan_id)
Scan(scan_id)
scan started successfully
scan started successfully
scan started successfully
User
CBF Subarray
Vcc
DsVccController
DsVccBand1And2
FPGA
Low-Level Tango Devices

CHAPTER

TWO

# SKA MID.CBF ENGINEERING CONSOLE

Documentation on the Developer's portal: ReadTheDocs

Code repository: ska-mid-cbf-engineering-console

The Engineering Console is being built in a Docker container, which insulates it from variations in the server environment. In addition to enabling MCS-Talon integration and testing, this container can be used to provide a controlled environment for automated end-to-end Talon HW testing.

The Engineering Console Docker is built in the pipeline and deployed to the Central Artefact Repository CAR.

## 2.1 Installation

```
git clone https://gitlab.com/ska-telescope/ska-mid-cbf-engineering-console
cd ska-mid-cbf-engineering-console
git submodule init
git submodule update
poetry install # to create the virtual environment with all dependencies
poetry shell # to run shell in the virtual environment
make oci-build # or "poetry run make oci-build" if not in the poetry shell
make run    # runs "hello world" test
```

## 2.2 Usage

### 2.2.1 Run the Docker interactively

To run the docker interactively:

```
make run-interactive
```

which opens a bash shell in the docker. To see available test script options:

```
./talondx.py --help
```

To install vim/nano editors while in interactive mode:

```
apt-get update
apt-get -y install vim nano
```

### 2.2.2 Generate Talondx Config FIle

To auto-generate the talondx config file based on the board configuration. Run the following command:

```
make generate-talondx-config BOARDS="<BOARDS>"
```

where is a comma-delimited list ofboard numbers you wish to turn on and deploy the HPS device servers onto. Example "1,2,3" if you wish to turn on and run device servers on talon1, talon2, and talon3.

### 2.2.3 Download Artefacts from CAR

To download FPGA bitstreams and Talon Tango device binaries from CAR to the local folder specified in the Makefile (`TALONDX_DEST_DIR`):

```
make download-artifacts
```

or specify a different destination folder:

```
make download-artifacts TALONDX_DEST_DIR="destination-folder"
```

A different config JSON can be specified if it exists as well (default value in the Makefile);

```
make download-artifacts
```

To upload new FPGA bitstreams to CAR for use, see the ska-mid-cbf-talondx project

### 2.2.4 Optional: Override DS Artefacts with local build

In order for this script to work, ensure to clone and build your device servers in the same root directory: Example: If clone ds-vcc and ds-lstv-gen device servers ensure both are cloned under the same directory which would like like:

1. /home/user/dev/ds/ds-lstv-gen
2. /home/user/dev/ds/ds-vcc

To override the device servers (ds-lstv-gen,ds-vcc in this example) run the following command:

```
make ds_list=ds-lstv-gen,ds-vcc ds_basedir=<path to ds base directory> ec_dir=<path to␣
↪ec checkout> ds-override-local
```

where ds_basedir is the path to the device server root directory of clone, /home/user/dev/ds from the previous example

### 2.2.5 Update the Tango DB inside MCS

```
make config-db
```

This command adds the Talon device servers as specified in the `talondx-config.json` file.

Note: the artefacts need to be downloaded before updating the database (the artefacts contain detri JSON files needed for the DB update).

### 2.2.6 Pull and run the Docker from CAR

```
docker pull artefact.skao.int/ska-mid-cbf-engineering-console:0.0.2
docker run artefact.skao.int/ska-mid-cbf-engineering-console:0.0.2
```

## 2.3 Read the Docs

The Engineering Console project auto-generates Read the Docs documentation, which includes this README.

To re-generate the documentation locally prior to checking in updates to Git:

```
make documentation
```

To see the generated documentation, open `/ska-mid-cbf-engineering-console/docs/build/html/index.html` in a browser – e.g.,

```
firefox docs/build/html/index.html &
```

---

CHAPTER

THREE

# CONFIGURE THE TALON-DX BOARDS FROM MCS

The Talon DX boards can be configured with binaries from CAR using a combination of Engineering Console and MCS both running the Dell Server – see MCS-Talon+Integration for details.

## 3.1 How to Run the On Command Sequence

### 3.1.1 1. Install MCS and Engineering Console

Install MCS, then *Install Engineering Console*.

### 3.1.2 2. Download Artefacts

Follow the instructions in *Download Artefacts from CAR*

### 3.1.3 3. Start MCS

Follow the instructions in MCS up to the `make install-chart` step to get MCS running.

### 3.1.4 4. Configure Tango DB

Follow the steps in *Update the Tango DB inside MCS*.

### 3.1.5 5. Ensure that Talon DX Boards are Off

Run the commands on either the Dell1 or Dell2 servers:

To check the current power status:

```
/shared/talon-dx-utilities/bin/talon_power_lru.sh <lru>
```

If the boards are powered on, power off the boards:

```
/shared/talon-dx-utilities/bin/talon_power_lru.sh <lru> off
```

Where is the lru associated to the board you wish you run your tests on. As of now, lru1 is associated to talon1/talon2 and lru2 is associated to talon3/talon4.

### 3.1.6 6. Send On Command from MCS

Run the required MCS On command sequence using:

```
make mcs-on SIM=<Simulation Mode>
```

SIM=1 if you want to run in Simulation Mode for all MCS devices. SIM=0 if you want to run without simulation mode and target the HPS devices on the talons.

### 3.1.7 7. Read Talon HPS Device Version and Status Information

To do a quick check that all devices turned on with the mcs-on command:

```
make device-check
```

To display Talon DS version information (version, build date, Git commit hash):

```
make talon-version
```

To repeatedly display the current Talon DS state and status:

```
make talon-status
```

CHAPTER

FOUR

# CONFIGURE AN END-TO-END SCAN TO GENERATE VISIBILITIES

This section assumes that you have followed all the steps in the previous section to power on and configure the Talon boards. At this point the three MCS devices that were configured in the previous step should be in the ON state. To perform a single receptor basic correlation, use the following command:

```
make basic-correlation BOARDS=<target talon>
```

CHAPTER

FIVE

# RUN A VISIBILITY CAPTURE

This section assumes the previous basic-correlation section has been successfully completed.

Run make command to start a visibility capture. Right after running the command the directory path will be printed as to where the data will be located if the command completes successfully.

```
make visibility-capture SV_VER=<Signal Verification Target Version>
```

While visibility capture is running, in another window issue the scan command to MCS

```
make mcs-scan BOARDS=<target talon>
```

After some time, issue the end scan command.

```
make mcs-end-scan BOARDS=<target talon>
```

CHAPTER

SIX

# NOTES ON DISH PACKET CAPTURE

## 6.1 1. On Command Sequence

Follow *How to Run the On Command Sequence*

## 6.2 3. Set up BITE devices

From the root directory of the engineering console run the following:

```
make talon-bite-config BOARDS="<BOARDS>"
```

where is a comma-delimited list of board numbers on which you wish to configure the BITE device servers. Example "1,2,3" if you wish to configure the BITE device servers on talon1, talon2, and talon3.

NOTE: Only the talon board defined in TALON_UNDER_TEST will be configured with the destination MAC address therefore only data from this board will reach the destination interfaces.

## 6.3 4. Dish Packet Capture

Open a new terminal. From the root directory of the engineering console run the following:

```
make dish-packet-capture
```

This command will not exit until the following command is run.

## 6.4 5. Start LSTV Replay

From the root directory of the engineering console run the following:

```
make talon-bite-lstv-replay BOARDS="<BOARDS>"
```

where is a comma-delimited list of board numbers on which you wish to replay data from. Example "1,2,3" if you wish to replay data from talon1, talon2, and talon3.

CHAPTER

SEVEN

# NOTES ON SIGNAL CHAIN VERIFICATION

## 7.1 Wideband State Count

Collect WB state count histogram and power spectrum vectors

```
make wb-state-count-capture
```

Generate a report from the WB state count vectors collected by the previous command.

```
make wb-state-count-report
```

Set WB_STATE_COUNT_LOCAL_DIR to specify the directory to store the outputs. By default this is ./mnt/wb-state-count

CHAPTER

EIGHT

# NOTES ON MCS INTERFACES

## 8.1 Commands

MCS commands can additionally be sent from Taranta (previously known as Webjive) or the itango3 shell.

### 8.1.1 Send the *On* command to CBF Controller from Taranta

Taranta needs to be enabled in MCS – see Taranta instructions for details.

### 8.1.2 Send commands to CBF Controller from itango3 shell

```
$ kubectl exec -it cbfcontroller-controller-0 -n ska-mid-cbf -- itango3
Defaulted container "device-server" out of: device-server, wait-for-configuration (init),
↪ check-dependencies-0 (init), check-dependencies-1 (init), check-dependencies-2 (init),
↪ check-dependencies-3 (init), check-dependencies-4 (init), check-dependencies-5 (init),
↪ check-dependencies-6 (init), check-dependencies-7 (init)
ITango 9.3.3 -- An interactive Tango client.

Running on top of Python 3.7.3, IPython 7.21 and PyTango 9.3.3

help      -> ITango's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

IPython profile: tango

hint: Try typing: mydev = Device("<tab>

In [1]: cbf_controller = DeviceProxy("mid_csp_cbf/sub_elt/controller")

In [2]: cbf_controller.State()
Out[2]: tango._tango.DevState.ON

In [3]: cbf_controller.Status()
Out[3]: 'The device is in OFF state.'

In [4]: cbf_controller.On()
Out[4]: [array([0], dtype=int32), ['On command completed OK']]
```

(continues on next page)

(continued from previous page)

```
In [5]: cbf_controller.Status()
Out[5]: 'The device is in OFF state.'

In [6]: cbf_controller.State()
Out[6]: tango._tango.DevState.ON
```

### 8.1.3 Send *ConfigureScan* command from itango3 shell

```
In [1]: controller = DeviceProxy("mid_csp_cbf/sub_elt/controller")

In [2]: subarray = DeviceProxy("mid_csp_cbf/sub_elt/subarray_01")

In [3]: controller.On()
Out[3]: [array([0], dtype=int32), ['On command completed OK']]

In [4]: subarray.AddReceptors(["MKT000", "MKT001", "MKT002", "MKT003"])
Out[4]: [array([0], dtype=int32), ['CBFSubarray AddReceptors command completed OK']]

In [5]: f = open("tests/data/ConfigureScan_basic.json")

In [6]: subarray.ConfigureScan(f.read().replace("\n", ""))
Out[6]: [array([0], dtype=int32), ['CBFSubarray Configure command completed OK']]
```

or paste the following into the itango3 shell:

```
controller = DeviceProxy("mid_csp_cbf/sub_elt/controller")
subarray = DeviceProxy("mid_csp_cbf/sub_elt/subarray_01")
controller.On()
subarray.AddReceptors(["MKT000", "MKT001", "MKT002", "MKT003"])
f = open("tests/data/ConfigureScan_basic.json")
subarray.ConfigureScan(f.read().replace("\n", ""))
```

Note: the test file `tests/data/ConfigureScan_basic.json` is part of the MCS codebase and is available when connected using itango3.

### 8.1.4 Send *Scan* command to VCC from itango3 shell

```
vcc = DeviceProxy("mid_csp_cbf/vcc/002")
vcc.simulationMode = 0
vcc.adminMode = 0
vcc.On()

vcc.ConfigureBand("1") # Only bands 1 and 2 are supported by the HPS software
vcc.ConfigureScan("{\
    \"config_id\": \"test_config\",\
    \"frequency_band\": \"1\",\
    \"frequency_band_offset_stream_1\": 5,\
    \"frequency_band_offset_stream_2\": 0,\
    \"rfi_flagging_mask\": "",\
```

(continues on next page)

(continued from previous page)

```
    \"fsp\": [\
        {\
            \"fsp_id\": 1,\
            \"frequency_slice_id\": 3,\
            \"function_mode\": \"CORR\"\
        }\
    ]\
}") # This is an example of the expected argument format for the VCC

vcc.Scan("6") # Use any arbitrary integer ID

vcc.EndScan()
```

## 8.2 Logs

### 8.2.1 View logs from a single MCS pod in the terminal

To see the CBF controller logs:

```
kubectl logs -f cbfcontroller-controller-0 -n ska-mid-cbf
```

where *cbfcontroller-controller-0* is the pod name shown when running `make watch` in MCS.

### 8.2.2 View logs using K9S

```
k9s -n ska-mid-cbf
```

then select the pod (e.g., *cbfcontroller-controller-0*) and press `l` to view the logs.

CHAPTER

NINE

# NOTES

## 9.1 Raw Repository

FPGA bitstreams are uploaded manually to the raw repository in CAR (Common Artefact Repository, https://artefact.skatelescope.org/) here:

`raw-internal/ska-mid-cbf-talondx/fpga-test/talon_dx-{_bitstream-name_}-v{_version_}.tar.gz`

### 9.1.1 Example - manually package the BITE bitstream files

```
mkdir bin
cp bite5.json bin/
cp mvp5_wip02.core.rbf bin/
cp mvp5_wip02.dtb bin/
tar -cvf talon_dx-bite-v0.5.0.tar bin
gzip -k talon_dx-bite-v0.5.0.tar
```

### 9.1.2 Example - manually unpackage the BITE bitstream files

```
gzip -d talon_dx-bite-v0.5.0.tar.gz
tar -xvf talon_dx-bite-v0.5.0.tar
```

where {*version*} is in the X.Y.Z format.

CHAPTER

TEN

# BITE CLIENT

BITE is the built-in test environment for the Talon DX board. The BITE client is running in the Engineering Console container that runs on the Linux server (either standalone, or in a pod in the Kubernetes cluster). The BITE client communicates with Tango device servers running on the Talon DX board to monitor and control the FPGA firmware that generates the Long Sequence Test Vectors (LSTVs) used to test the correlator signal chain.

CHAPTER

ELEVEN

# BITE CONFIGURATION

## 11.1 Input Parameter Files

There are four parameter files which are to be used in configuring the BITE for LSTV generation. Each of these has an accompanying schema file, against which the BITE client (or BDD test) will validate the input parameter files. The parameter files can be found in the ska-mid-cbf-system-tests repository, in the test_parameters directory. They are:

- test_parameters/tests.json
- test_parameters/cbf_input_data/cbf_input_data.json
- test_parameters/cbf_input_data/bite_config_parameters/bite_configs.json
- test_parameters/cbf_input_data/bite_config_parameters/filters.json

The schemas for these are located in the same relative locations in the ska-mid-cbf-internal-schemas repository, but have "_schema.json" at the end of their names. They are:

- test_parameters/tests_schema.json
- test_parameters/cbf_input_data/cbf_input_data_schema.json
- test_parameters/cbf_input_data/bite_config_parameters/bite_configs_schema.json
- test_parameters/cbf_input_data/bite_config_parameters/filters_schema.json

The tests.json parameter file contains a series of BITE tests. They are labelled by a Test ID, in the format of "Test 1" "Test 2", etc., and each of them is defined with a set of test parameters according to their test scope and scan configuration. The "cbf_input_data" property associated with each Test ID links that test to a set of properties defined in the cbf_input_data JSON file above. These sets of CBF Input Data are named in this style: "BITE Data 1", "BITE Data 2", etc. Each is defined (in the aforementioned JSON) with an array of receptors; each receptor is defined in turn with a dish id, a bite configuration ID, a Talon board, and the initial timestamp of the BITE data that will be generated.

The bite configuration ID corresponds with an identically-named ID in the "bite_configs.json". These are named in the style of "BITE 1", "BITE 2", etc., and they are each given a brief summary in their "description" property to make up for the nondescript ID. Each bite configuration herein is defined with an array of Gaussian noise sources, an array of tone generators, and other parameters specific to LSTV generation and sample rate.

Each element in the source array must be given a description, Gaussian noise parameters, a polarization coupling rho, and, if desired, the number of coefficients with which this source's filters will be defined (along with the bit width in which each coefficient is to be represented and stored). Each noise source is divided into the two polarizations, always termed "pol_x" and "pol_y", and each of these is parameterized with the mean, standard deviation, and seed of the Gaussian noise it will generate. Furthermore, each polarization must be given a filter type, as named in the filters.json file in the list above.

```
"sources": [
    {
        "description": "Noise input, independent between X and Y pols; unique filter␣
↪shapes for X and Y pols.",
        "gaussian": {
            "pol_x": {
                "seed": 1234,
                "filter": "filter_ramp_up",
                "noise_std": 32767,
                "noise_mean": 0

            },
            "pol_y": {
                "seed": 9876,
                "filter": "filter_ramp_down",
                "noise_std":32767,
                "noise_mean":0
            }
        },
        "pol_coupling_rho": 0.0,
        "pol_Y_1_sample_delay": false,
        "fir_filter_num_taps": 1024,
        "fir_filter_coeff_bits": 16
    }
],
```

Each tone in the array of tone generators is similarly given a description, and divided into X and Y polarizations. Each of these components is defined with a frequency and an amplitude scaling factor.

```
"tone_gens": [
    {
        "description": "Basic 100 MHz tone.",
        "pol_x": {
            "frequency": 100e6,
            "scale": 0.0025
        },
        "pol_y": {
            "frequency": 333.3333333e6,
            "scale": 0.0025
        }
    }
],
```

## 11.2 Executing the Configure Script

The BITE can be configured to generate Test Data for a given test via the midcbf_bite.py script, which instantiates a BITE client for each desired Talon. Assuming all required Talons are available, and all the device servers required for BITE are running on them, BITE configuration can be initiated by running the midcbf_bite.py script with the –talon-bite-config argument and one of the two arguments for specifying the configuration data. Either specify the ID of the BDD Test for which the BITE is to be configured; or specify the name of the desired set of CBF Input Data. These amount to the same thing, as each test is defined with a corresponding set of input data. Either:

```
python3 midcbf_bite.py --talon-bite-config --test <Test ID>
```

or:

```
python3 midcbf_bite.py --talon-bite-config --input_data <CBF Input Data>
```

- <Test ID> specifies a desired system test from tests.json. Provide only the number in the test ID, i.e. --test 1 for "Test 1", --test 2 for "Test 2". The CBF input data associated with that test ID will be used in configuring the BITE client(s) to generate data for that test.
- <CBF Input Data> specifies a set of CBF input data from cbf_input_data.json This will configure BITE for all the receptors defined in that set of input data, according to the Dish ID, BITE configuration ID, Talon board, and initial timestamp offset defined for each.

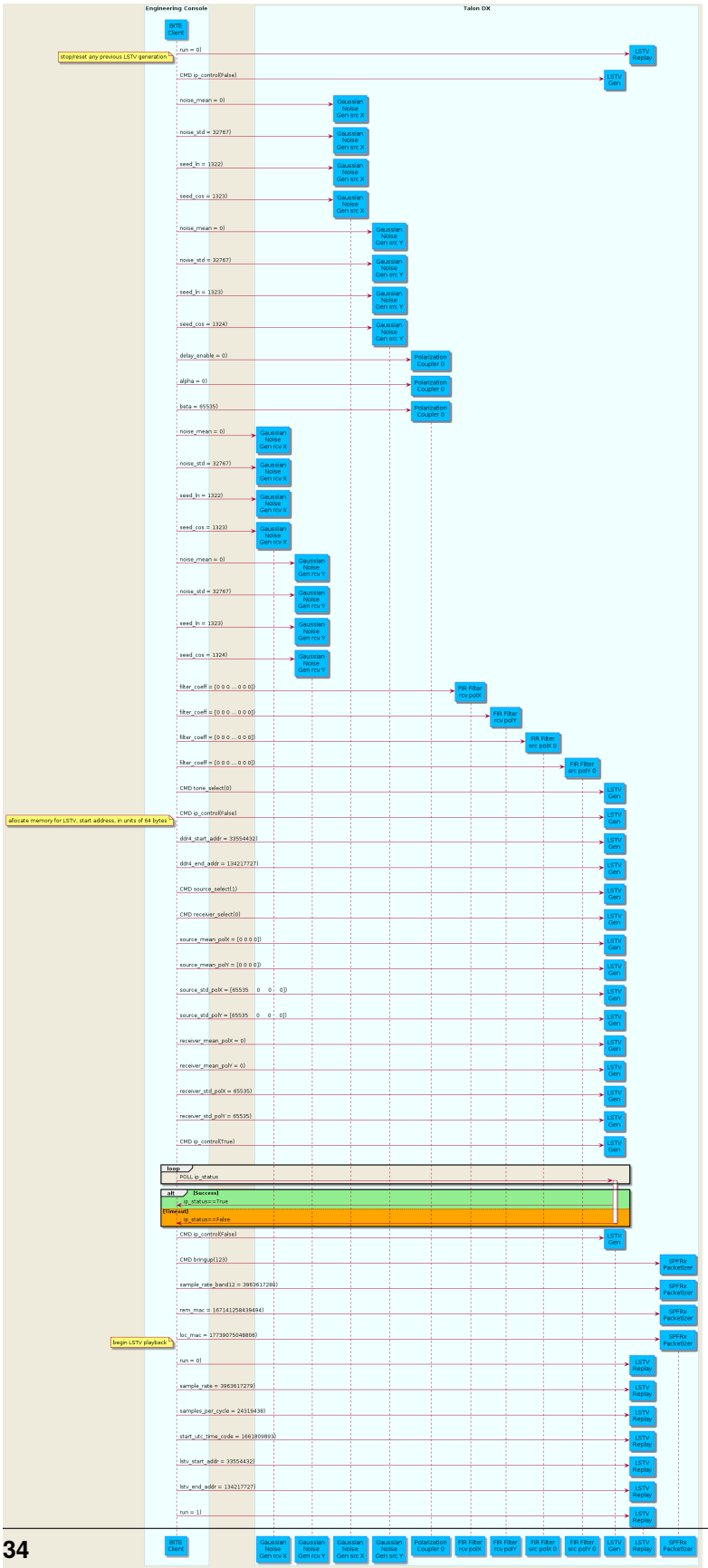
The Talon BITE configuration can be manually initiated from the EC-BITE docker container, assuming that all device servers are already running on all Talon boards required by the CBF Input Data. However, if doing so, the BITE client will expect to find all four of the required parameter files and all four schema files in their proper place, in the docker container. That is, the parameter files must be located in the EC-BITE container at <namespace>/engineering-console-bite:/app/test_parameters, and the schemas at :/app/schemas. These files will have been manually copied to that location, or they will have been copied when the signal chain verification test has last been run. If the user is trying to configure BITE manually from the EC-BITE container, they must ensure that the files are in their proper place, or the BITE configuration will error out.

If all is well, the --talon_bite_config argument will instantiate a BITE client for each specified board, which will initialize Tango device proxies for each device server, assuming that each device server required for BITE is listed in device_server_list.json. Each BITE client reads values from the parameter files and writes them to the Tango device proxies which it has initialized in the previous step; in doing so, the values are written to the device servers (and, thus, to the IP blocks required by BITE) to configure them properly. Values are also written to the device servers which will control them in their generation of BITE data. The BITE Sequence Diagram provides more information on which values are written where, and in what sequence, in the BITE configuration.

## 11.3 Bite Configuration from BDD Tests

Mid CBF AA0.5 Test Strategy Mid CBF Signal Chain Verification

## 11.4 BITE Sequence Diagram



CHAPTER

# TWELVE

# TALON DX SCRIPT

CHAPTER

# THIRTEEN

# TALON DX CONFIG

## 13.1 TalonDxConfig Class

**class** talondx_config.talondx_config.**TalonDxConfig**(*config_file*)

TalonDxConfig facilitates loading and validation of the Talon DX Configuration JSON file (see schema for details).

**Parameters**
**config_file** (*string*) – filename of the JSON configuration file

**config_commands**()

Extracts and returns the *"config_commands"* section of the configuration file that specifies the configuration commands that are sent from the MCS to the Talon DX HPS Master device.

**ds_binaries**()

Extracts and returns the *"ds_binaries"* section of the configuration file that specifies the Tango DS binaries to be downloaded, and where to get them.

**export_config**(*export_path*)

Exports the Talon DX Configuration JSON to a file with same name as that used to construct this object. Export will overwrite if the file already exists.

**Parameters**
**export_path** (*string*) – destination path of exported configuration file.

**fpga_bitstreams**()

Extracts and returns the *"fpga_bitstreams"* section of the configuration file that specifies which FPGA bitstreams to download, and where to get them.

**tango_db**()

Extracts and returns the *"tango_db"* section of the configuration file that contains the device server specifications for populating the Tango DB.

## 13.2 Schema


```
{
    "type": "object",
    "properties": {
        "ds_binaries": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "name": {"type": "string"},
                    "source": {"type": "string", "enum": ["conan", "git"]},
                    "conan": {"$ref": "#/$defs/conan"},
                    "git": {"$ref": "#/$defs/git"}
                },
                "required": [
                    "name",
                    "source"
                ],
                "additionalProperties": false
            }
        },
        "fpga_bitstreams": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "source": {"type": "string", "enum": ["raw", "git"]},
                    "version": {"type": "string"},
                    "raw": {"$ref": "#/$defs/raw"},
                    "git": {"$ref": "#/$defs/git"}
                },
                "required": [
                    "source",
                    "version"
                ],
                "additionalProperties": false
            }
        },
        "config_commands": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "description": {"type": "string"},
                    "target": {"type": "string"},
                    "ip_address": {"type": "string"},
                    "talon_first_connect_timeout": {"type": "integer"},
                    "ds_hps_master_fqdn": {"type": "string"},
                    "fpga_path": {"type": "string"},
                    "fpga_dtb_name": {"type": "string"},
                    "fpga_rbf_name": {"type": "string"},
```


(continues on next page)

(continued from previous page)

```
                    "fpga_label": {"type": "string"},
                    "ds_path": {"type": "string"},
                    "server_instance": {"type": "string"},
                    "talon_lru_fqdn": {"type": "string"},
                    "ds_rdma_rx_fqdn": {"type": "string"},
                    "devices": {
                        "type": "array",
                        "items": {"type": "string"}
                    }
                },
                "required": [
                    "description",
                    "target",
                    "ip_address",
                    "ds_hps_master_fqdn",
                    "fpga_path",
                    "fpga_dtb_name",
                    "fpga_rbf_name",
                    "fpga_label",
                    "ds_path",
                    "server_instance",
                    "devices"
                ],
                "additionalProperties": false
            }
        },
        "tango_db": {
            "type": "object",
            "properties": {
                "db_servers": {
                    "type": "array"
                }
            }
        }
    },
    "required": [
        "ds_binaries",
        "fpga_bitstreams",
        "config_commands",
        "tango_db"
    ],
    "$defs": {
        "conan": {
            "type": "object",
            "properties": {
                "package_name": {"type": "string"},
                "user": {"type": "string"},
                "channel": {"type": "string"},
                "version": {"type": "string"},
                "profile": {
                    "type": "string",
                    "enum": ["conan_aarch64_profile.txt", "conan_x86_profile.txt"]
```

(continues on next page)

(continued from previous page)

```
                }
            },
            "required": [
                "package_name",
                "user",
                "channel",
                "version",
                "profile"
            ],
            "additionalProperties": false
        },
        "raw": {
            "type": "object",
            "properties": {
                "group": {"type": "string"},
                "base_filename": {"type": "string"}
            },
            "required": [
                "base_filename"
            ],
            "additionalProperties": false
        },
        "git": {
            "type": "object",
            "properties": {
                "git_project_id": {"type": "integer"},
                "git_branch": {"type": "string"},
                "git_pipeline_job": {"type": "string"}
            },
            "required": [
                "git_project_id",
                "git_branch",
                "git_pipeline_job"
            ],
            "additionalProperties": false
        }
    }
}
```

CHAPTER

# FOURTEEN

# DB POPULATE

CHAPTER

FIFTEEN

# CONAN

## 15.1 ConanWrapper Class

class conan_local.conan_wrapper.**ConanWrapper**(*folder*)

ConanWrapper provides a Python interface to the shell commands required to download Conan packages. There is a Python API, but it is not documented, and according to: https://github.com/conan-io/conan/issues/6315 `The python api is not documented nor stable. It might change at any time and break your scripts.` This class can be updated to use the Python API if/when that makes sense.

**Parameters**

**folder** (*string*) – destination path for downloaded (deployed) conan packages.

static **clear_local_cache**()

Removes all packages and binaries from the local cache.

**download_package**(*pkg_name*, *version*, *user*, *channel*, *profile*, *timeout=60*)

Run conan install to download the conan package and deploy it to the folder specified at construction.

**Parameters**

- **pkg_name** (*string*) – conan package name
- **version** (*string*) – conan package version number
- **user** (*string*) – user name of conan package
- **channel** (*string*) – conan package channel name
- **profile** (*string*) – name of text file containing the conan profile used to deploy the download package
- **timeout** (*int*) – download timeout in seconds

static **search_local_cache**()

Searches local cache for package recipes and binaries.

static **version**()

Returns conan version.

## 15.2 Conan Profiles

### 15.2.1 Cross-compiled (HPS) Tango Devices

```
target_host=aarch64-linux-gnu
toolchain=/usr/$target_host
cc_compiler=gcc
cxx_compiler=g++

[settings]
os=Linux
arch=armv8
compiler=gcc
compiler.version=7
compiler.libcxx=libstdc++11
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$toolchain
CONAN_CMAKE_SYSROOT=$toolchain
PATH=[$toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
LD=$target_host-ld
STRIP=$target_host-strip
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
CXXFLAGS=-I"$toolchain/include"
```

### 15.2.2 Native-compiled (Linux server) Tango Devices

```
target_host=x86_64
toolchain=/usr/bin
standalone_toolchain=/usr
cc_compiler=gcc
cxx_compiler=g++

[settings]
os=Linux
arch=x86_64
compiler=gcc
compiler.version=9
compiler.libcxx=libstdc++11
build_type=Release

[env]
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
```

(continues on next page)

(continued from previous page)

```
AR=ar
AS=as
RANLIB=ranlib
LD=ld
STRIP=strip
CC=$cc_compiler
CXX=$cxx_compiler
CXXFLAGS=-I"$standalone_toolchain/include"
```

## 15.3 Conan Remotes

```
{
    "remotes": [
     {
       "name": "ska",
       "url": "https://artefact.skatelescope.org/repository/conan-internal/",
       "verify_ssl": true
     },
     {
      "name": "conan.io",
      "url": "https://center.conan.io/",
      "verify_ssl": true
     }
    ]
}
```

CHAPTER

SIXTEEN

# TALON DX LOG CONSUMER

The Talon DX Log Consumer is a Tango device intended to run on the host machine that connects to the Talon-DX boards. This Tango device is set up as a default logging target for all the Tango device servers running on the HPS of each Talon-DX board. When the HPS device servers output logs via the Tango Logging Service, the logs get transmitted to this log consumer device where they get converted to the SKA logging format and outputted once again via the SKA logging framework. In this way logs from the Talon-DX boards can be aggregated in once place and eventually shipped to the Elastic framework in the same way as logs from the Mid CBF Monitor and Control Software (MCS).

Note that eventually this Tango device will be moved to the Mid CBF MCS, and more instances of the device may be created to provide enough bandwidth for all the HPS device servers.

## 16.1 Connecting from HPS DS to the Log Consumer

The Talon-DX boards connect to the host machine (currently known as the Dell Server) over a single Ethernet connection. The IP address of the Dell Server on this connection is `169.254.100.88` and all outgoing traffic from the Talon-DX boards must be addressed to this IP.

When the log consumer starts up on the Dell server, the OmniORB end point (IP address and port) it is assigned is local to the Dell server (i.e. IP address `142.73.34.173`, arbitrary port). Since the Talon boards are unable to connect to this IP address. we need to manually publish a different endpoint when starting up the log consumer that is visible to the HPS devices.

The following ORB arguments are used (see the make target `talondx-log-consumer`):

- `-ORBendPointPublish giop:tcp:169.254.100.88:60721`: Exposes this IP address and port to all clients of this Tango device. When the HPS device servers contact the database to get the network information of the log consumer, this is the IP address and port that is returned. The IP addresses matches that of the Ethernet connection to the Dell server, allowing the HPS device servers to direct their messages across that interface.
- `-ORBendPoint giop:tcp:142.73.34.173:60721`: Assigns the IP address and port that the log consumer device is actually running on. This needs to be manually assigned since an iptables mapping rule was created on the Dell server to route any TCP traffic coming in on `169.254.100.88:60721` to `142.73.34.173:60721`.

Some important notes:

- Due to the end point publishing, no Tango devices running on the Dell server will be able to connect to the log consumer (including being able to configure the device from Jive). This is because the published IP address is not accessible on the Dell server. There may be a way to publish multiple endpoints, but this needs further investigation.
- If the log consumer device cannot be started due to an OmniORB exception saying that the end point cannot be created, it is possible that the `142.73.34.173` needs to change to something else. It is not yet clear why this can happen. To change it do the following:

- Remove the ORB arguments from the `talondx-log-consumer` make target, and then start the log consumer.
- Open up Jive and look at what IP address is automatically assigned to the log consumer device. This is the IP address that we now need to use for the endpoint.
- Find the iptables rule that maps `169.254.100.88:60721` to `142.73.34.173:60721`, and change it to the new IP address.
- Add the ORB arguments back in, using the correct IP address for the end point.

CHAPTER

SEVENTEEN

# AUTOMATED SCRIPT

The automated script is a method to deploy the MCS system inside the minikube and execute commands through the engineering console without relying on make commands in a given git repository. All images and containers are pulled directly from CAR.

## 17.1 Preconditions

- The script must be run as a user with passwordless root permission.
- The script must be run from the .../automation directory.
- Latest stable MCS and Engineering Console image versions are known hard-coded in script.

## 17.2 Key Files

The following files are necessary to run the automated script:

- `orchestration.sh`: The entry point for a cronjob.
  - Ensures more than one instance of the script is not running.
  - Creates the test result directories.
- `setup.sh`: Sets up the test environment.
  - Records the test configuration.
  - Starts Minikube.
  - Programs the talon boards.
- `script.sh.multiboard`: The test. See "Current Outcomes" below.

## 17.3 Current Outcomes

Using a git and makefile detached script to automate the following:

- MID CBF MCS Deployment
- Use Engineering Console to:
  - Configure the Tango DB inside MCS
  - Turn on the Talon Boards using the LMC Interface

  - Check the Talon DS Versions to verify status
  - Generate BITE Data
  - Replay the BITE Data through the board back into the primary server
  - Capture the data on the Engineering Console
  - Configure the VCC Bands
  - Use the Serial Loop-back to Send the BITE Data back through the board into the VCC Firmware IP Blocks
  - Use the RDMA Tx to send the VCC Data to the RDMA Rx
- Run the automated script nightly as a regression test.

## 17.4 Future Outcomes

The features of the automated script will be extended to the following:

- Verify the data captured by the RDMA Rx

The current automated script is run nightly and saves the test results locally. The aim is to send the results to JIRA X-RAY, as well as to generate reports with captured data and plots.

## 17.5 Running the Script

Refer to the Automated Script Confluence Page

CHAPTER

EIGHTEEN

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX