European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Instrumentation Framework

ELT ICS Framework - Function Control Framework - User Manual

Document Number: ESO-320177

Document Version: 2

Document Type: Manual (MAN)

Released on: 2021-05-31

Document Classification: Public

Owner:	Kiekebusch, Mario
Validated by PM:	Kornweibel, Nick
Validated by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Appoved by PGM:	Tamai, Roberto
	Name

Release

This document corresponds to `ifw-hl`[1] v3.0.0.

Authors

Name	Affiliation
Mario Kiekebusch	ESO/DOE/CSE
Jens Knudstrup	ESO/DOE/CSE
Dan Popovic	ESO/DOE/CSE

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1084
Device Manager	Updated state machine diagram
Client Application	Updated list of commands
	Updated examples
	Update python client library
	Added FCF CLI section
	Added Simple Parameter Format section
Programming Guide	Updated according to new setup schema
Getting Started	Adapted to Template Project

[1] https://gitlab.eso.org/ifw/ifw-hl

Contents

1 Introduction

The FCF is the ICS Framework component with the purpose of helping consortia in building the FCS software. Its main goal is to provide ready to use and configurable software entities for controlling and monitoring instrument hardware functions and sensing systems. The FCF includes subcomponents along the three layers of the ICS three-tier architecture.

1.1 Scope

This document is the user manual for the ELT ICS Framework - FCF. The intended audience are ELT users, consortia developers or software quality assurance engineers. This release is to be used by the Consortia developers in trying out the control of instrument hardware functions using the provided libraries and applications, as well as getting acquainted with the design choices and their implementations.

1.2 Acronyms

ADC	Atmospheric Dispersion Correction
DB	Database
CCS	Central Control System
ELT	Extremely Large Telescopen
FCF	Function Control Framework
FCS	Function Control System
GUI	Graphical User Interface
ICS	Instrument Control System
LCS	Local Control System
PLC	Programming Logical Controller
RAD	Rapid Application Development
RPC	Remote Procedure Call
SCXML	State Chart XML

1.3 Main Components

The present version of the FCF covers the following main components:

- A *Device Manager* implementation that can control a configurable number of devices from a standard ELT WS.
- A generic *GUI* for the Device Manager that allow users to control devices graphically.
- A set of *Device Simulators* capable of emulating the behaviour of a device controller and its interface within a WS.

- A set of *PLC libraries* implementing the supported device controllers, corresponding PLC simulators and HMIs for local control.

1.4 Top Directory Structure

The first level of the `fcf` directory contains the following:

```
<root>              # FCF component root
├── devmgr          # directory containing the FCF manager and devices classes
├── devsim          # directory containing the device simulators
├── gui             # directory containing the different GUIs modules
└── wscript         # WAF build script
```

1.5 Device Manager (devmgr)

The server implementation is based on the ICS application framework (rad). Following the ELT and ICS development standards, the client and server are implemented in C++.

1.5.1 Directory Structure

In the present version of the FCF, the device manager contains:

```
<root>              # devmgr root directory
├── cli             # FCF CLI  added in version 3
├── client
├── common
├── devices
├── fcfif
├── fcfclib
├── server
├── templates
└── wscript
```

Where:

- `cli` is a dedicated shell to interact with the Device Manager. the server from the command line.
- `client` is an application that can be used to send commands to the server from the command line.
- `common` is a library implementing core server classes like actions and activities.
- `devices` is a library implementing the device classes.
- `fcfif` is the CII XML interface module with the payload definition for commands and topics.
- `fcfclib` is a python library that simplifies the interaction with the server from Python scripts.

- `server` is the server application (devmgrServer). This is a reference implementation that can be configured to control instrument functions of a given type.
- `templates` is a directory containing templates for code and configuration generation. Template files use the Jinja2 template engine: Jinja 2 documentation[1]

1.6 Device Simulators (devsim)

The FCF includes a set of Simulators with the purpose of allowing the Device Manager to run without the need for a PLC. These Simulators are implemented in Python and they run on a Linux WS. Each Device Simulator implements an OPC-UA Server as well as the business logic of a particular Device Controller.

1.7 Graphical Interfaces (gui)

In order to simplify the usage of the server, the `fcf` provides a prototype of an engineering interface. The graphical interface has been implemented in Qt using the QtWidget library.

1.7.1 Directory Structure

In the present version of the FCF, the `gui` contains:

```
<root>              # gui root directory
├── fcfgui          # FCF generic engineering graphical interface
├── wdglib          # Motor device engineering graphical interface
├── pymotgui        # Motor device engineering graphical interface implemented in␣
↪Python.
├── msglib          # Library for sending commands to the server from GUIs.
└── wscript
```

1.8 PLC Libraries (controllers)

With the adoption of GIT, some components have been split into high and low level parts. This is the case of the FCF where PLC controller projects are found now under the `ifw-ll` GIT project which contains the controller directory with all PLC library projects. This directory contains the list of TwinCAT projects implementing the Device Controllers and Simulators for each of the hardware functions to be controlled by the FCF at the local level. These directories are Microsoft Visual Studio projects and they shall be edited under Windows using the TwinCAT IDE.

- ifw-ll release (fcf controllers)[2]

[1] http://jinja.pocoo.org/docs/dev/
[2] https://gitlab.eso.org/ifw/ifw-ll/-/releases

Warning: Unlike the previous IFW version, these TwinCAT projects do not contain the compiled libraries only the source code.

We have compiled and packed all PLC libraries binaries into a single directory for this release for easy access by the developers. They can be retrieved from the SVN repository here[3].

1.8.1 Directory Structure

In the present version of the FCF, the `controllers` contains:

```
<root>              # controller root directory
├── actuator        # Actuator PLC library
├── ioDev           # IoDev PLC library
├── lamp            # Lamp PLC library
├── motor           # Motor PLC library
├── piezo           # Piezo PLC library
├── plctpl          # PLC Template library
├── rsComm          # Serial communication PLC library
├── shutter         # Shutter PLC library
├── timer           # Timer PLC library
└── trklib          # PLC C++ modules
```

[3] http://svnhq9.hq.eso.org/p9/tags/EELT/ICS/PLC/2.0.0

2 Device Manager

The Device Manager provides the functionality for supervision and management of a configurable set of devices.

The Device Manager provides a library of devices implementing the communication with the respective device controllers in the PLC. Devices are created at the manager start-up by a device factory class. The main components of the *Device Manager* server are:

- State Machine engine based on SCXML and implemented in RAD. It contains a set of action and activity classes.
- A Device Factory class that creates the instances of all device classes at start-up and based on the server configuration.
- A set of Device classes. Each device has two additional classes: one for the device configuration and the other one for the interface with the Local Control System (LCS).
- A Facade class that manages the interface between the state machine engine and the device classes.

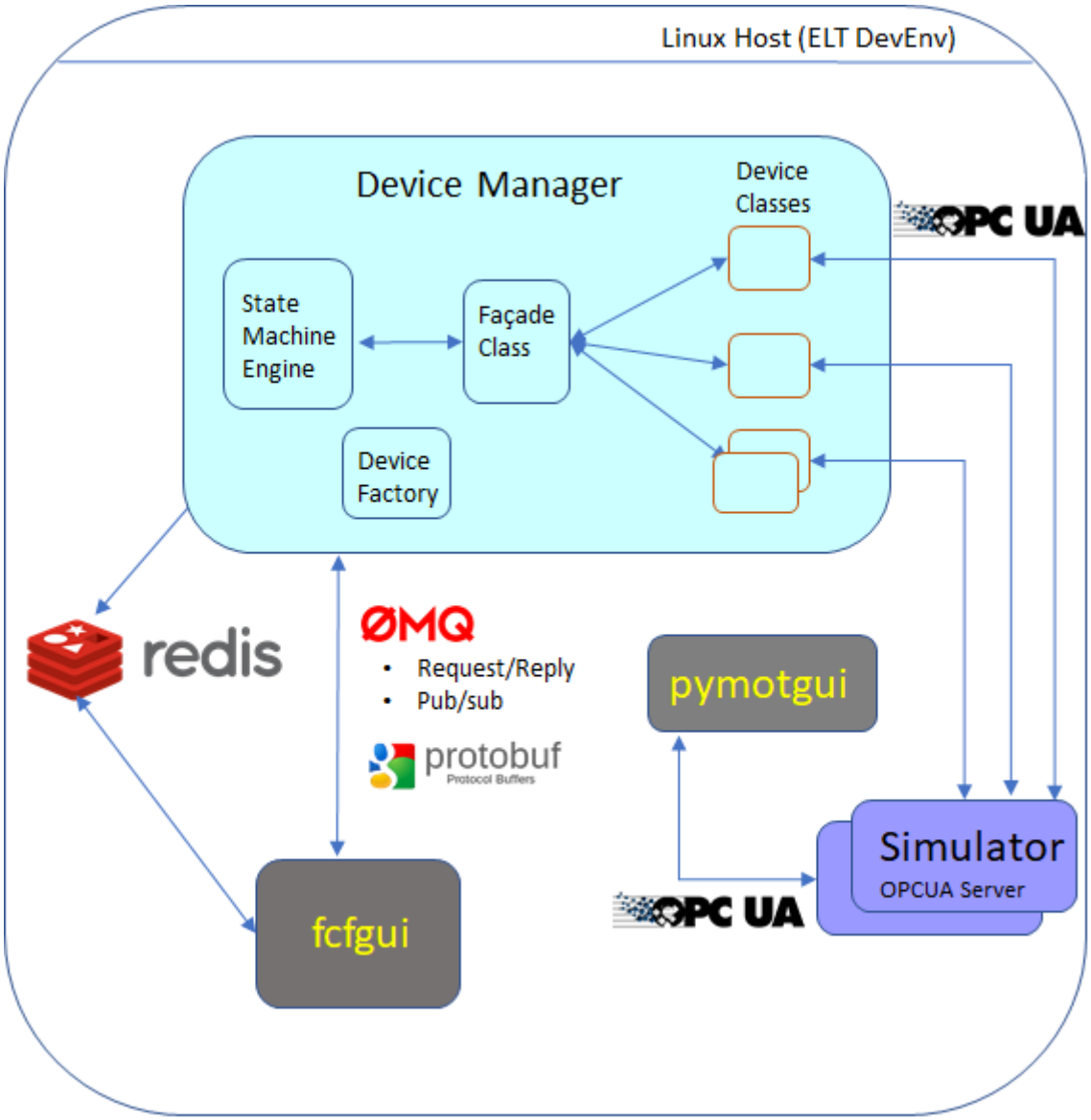


Fig. 2.1: Device Manager Components without LCS.

Client applications, such as *fcfGui*, send commands to the Device Manager using the CII MAL library (request/reply). The *fcfGui* is subscribed to the device topic changes published by the Device Manager through CII MAL (pub/sub).

The Device Manager uses the Redis Database to store run-time information about itself and about the devices it controls. In absence of a Local Control System, device classes can connect to the Device Simulator via the OPC-UA protocol, see figure above.

In normal operation, device classes connect to the OPC-UA server running under the Windows OS side of the Beckhoff IPC, e.g .CX2030. This communication is based on the execution of RPC calls (OPC-UA Method profile). Each Device Controller running in the TwinCAT PLC declares a number of

methods defining the interface with the Device Manager. Additionally, the device classes subscribe to the status data produced by the device controllers. Each time the status changes, the device classes are notified and they updates the Redis DB and publish the corresponding changes via CII (pub/sub).

The PLC OPC-UA Server connects to the device controllers via the vendor specific protocol (ADS). The device controllers trigger the changes in the hardware via the TwinCAT I/O mapping.

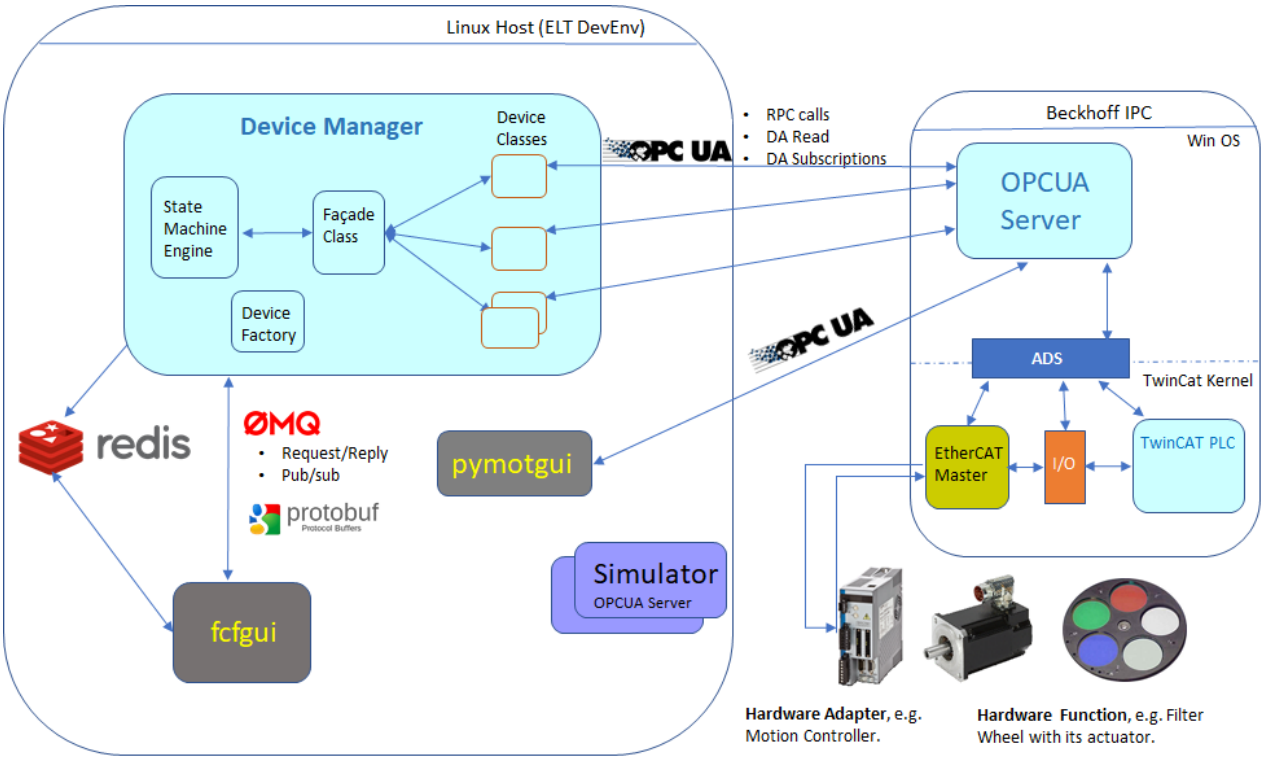


Fig. 2.2: Device Manager connecting to a Beckhoff IPC.

2.1 Supported Devices

2.1.1 Shutters

The *Shutter* device is a general purpose device for controlling a shutter hardware function. The device can control the shutter open/close.

2.1.2 Lamps

The *Lamp* device is a general purpose device for controlling a lamp hardware function. The device can switch a lamp on/off, control the intensity and handle warm-up and cool-down times when this is supported.

2.1.3 Motors

The *Motor* device is a general purpose device that controls different types of motors. It provides the following features:

- Support three different axis types: Linear, Circular and Circular-Optimized. Circular-Optimized means that the motor will always take the shorter path to reach the target position.
- Definition of named positions in user units or encoder values.
- Arbitrary positioning given in user units or encoder values.
- Positioning in absolute or relative units.
- Support for configurable Initialization Sequence.
- Support for SW limits.
- Support for various timeouts.
- Auto disabling when standing.
- Support for brake handling.
- Support for backlash compensation.

2.1.4 Sensors

The *Sensor* device is a device that groups instrument engineering variables for the purpose of monitoring and recording of the instrument status and its subsystems over time. It can be configured with a variable number of channels that are grouped logically. The *Sensor* device supports three different channel types: Digital input, Analog input and Integer input.

2.1.5 Derotators

The *Derotator* device is an aggregated motor device that continuously adapt its position according to the field or pupil rotation. It supports four different modes:

- **Stationary**: Derotator moves to a target position based on the position angle and remains standstill after reaching the target.
- **Sky**: The Derotator is continuously moving to compensate for field rotation.
- **Elevation**: The Derotator is continuously moving to compensate for pupil rotation.

- **User**: The Derotator is continuously moving according to a customized computation of the position defined by the user.

2.1.6 ADCs

The *ADC* device manages the position of two prisms with the aim of correcting for the atmospheric dispersion.

The device supports two modes:

- **Auto**: The ADC is continuously positioning the two motors based on the telescope RA/DEC, the environmental parameters and the ADC configuration.
- **Off**: The ADC moves to a target position and remains standstill after reaching the target.

2.1.7 Piezos

The 'Piezo' device manages the control of the output signals of a piezo hardware. It supports up to three axes. The device can be set in two modes:

- **Auto**: The Piezo is correcting continuously the outputs based on the feedback signals.
- **Pos: The Piezo set the output of the axes to a fixed value. In this mode, the Piezo can** be controlled in user positions (normally volts) or directly in bits.

2.1.8 Actuator

The *Actuator* device is a general purpose device for controlling actuators through a switch signal (on/off). The most common use of astuators is for power control.

2.2 Device Manager State Machine

The Device Manager uses a state machine described in a SCXML format that is interpreted by the state machine engine provided by the `rad` application framework. (SCXML specification[4]).

[4] https://www.w3.org/TR/scxml/

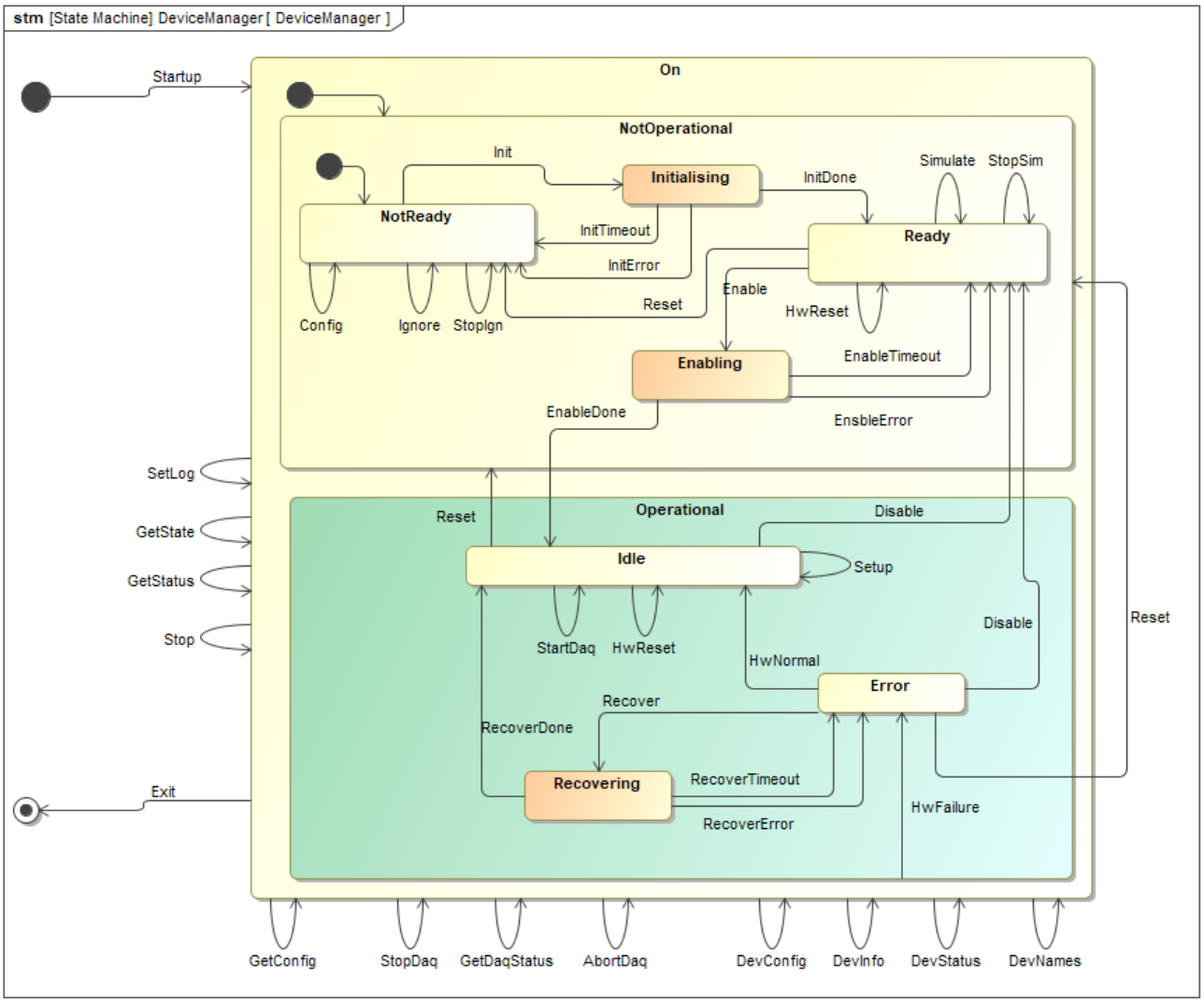


Fig. 2.3: Device Manager State Machine Diagram.

Off* –> *NotReady*, event: *Startup

The Device Manager starts up and goes automatically to *NotOperational/NotReady*. Main server objects are instantiated including the basic application that uses the State Machine engine. The Device Manager reads its own configuration and completes its initialisation.

***NotReady* –> *Ready*, event: Init**

The server connects to each of the device controllers through the device objects. Depending on the device configuration, it establishes the connection to the real HW or to the simulator. If any of the

device objects fails to establish the connection, the server will remain in substate *NotReady*.

NotOperational/Ready –> Operational/Idle*, event: *Enable

The Device Managers goes through the Enabling state. If Device Controllers are already Operational, the Device Manager does not affect their state and goes immediately to the Operational state. If Device Controllers are not operational, Device Manager will trigger the events (via OPC-UA method calls) to reach the operational state for each of the local devices under the manager control. If it does not succeed within the defined timeout (configuration parameter, see *Device Manager Configuration*), it will reply with a failure remaining in NotOperational/Ready state. During the transition from *NotOperational/Ready* to *Operational/Idle*, the Device Manager downloads the configuration to each Device Controller. If at least one Device Controller cannot reach the Operational state, the Device Manager will remain in state *NotOperational/Ready*.

Operational/Idle –> Operational/Error*, event: *HwFailure

Problems in at least one of the managed devices will bring the Device Manager into the Error state (Operational/Error). A typical example would be when the PLC, running the Device Controller, is power-cycled.

Operational/Error –> Operational/Idle*, event: *HwNormal

In the situation when an error condition is recovered, the Device Manager will go back automatically to *Operational/Ready* state (event *HwNormal*). For instance, if the network connection is lost, the Device Manager will go to *Error* but when the network is restored, the Device Manager will update its state automatically.

Operational –> NotOperational/Ready*, event: *Disable

The Device Manager disables the operation of devices but the state of controllers is not affected. If the state of the controllers is to be changed to NotReady, this has to be done separately. The reason for the above is to avoid affecting the state of the controllers by changing the state of the manager and thus achieve minimal impact on the hardware. In case of error going from Operational to *NotOperational/Ready*, the end state will be nevertheless *NotOperational/Ready*.

NotOperational/Ready –> NotOperational/NotReady*, event *Reset

The subscription to the OPC-UA server is stopped and the sessions of the managed devices are

disconnected. In case of error going from *Ready* to *NotReady*, the end state will be nevertheless *NotReady*.

Extract of the current State Machine specification for the Device Manager.

```
<state id="On">
<initial>
  <transition target="NotOperational"/>
</initial>
<state id="NotOperational">
  <initial>
    <transition target="NotReady"/>
  </initial>
  <state id="NotReady">
    <transition event="Events.Reset" target="NotReady">
       <customActionDomain:ActionReset name="ActionReset"/>
    </transition>
    <transition event="Events.Init" target="Initialising"/>
    <transition event="Events.Config">
     <customActionDomain:ActionConfig name="ActionConfig"/>
    </transition>
  </state>
  <state id="Initialising">
    <onentry>
      <customActionDomain:ActionInitStart name="ActionInitStart"/>
    </onentry>
    <invoke id="ActivityInitialising"/>
           <transition event="Events.InitDone" target="Ready">
              <customActionDomain:ActionInitDone name="ActionInitDone"/>
           </transition>
           <transition event="Events.InitError" target="NotReady">
              <customActionDomain:ActionInitError name="ActionInitError"/>
           </transition>
```

2.3 Configuration

2.3.1 Device Manager Configuration

The server configuration is a set of files written in `yaml` format. (YAML specification[5]). YAML is easy to read format that has been adopted temporary until the integration with CII configuration services.

Many resources about YAML can be found on the web. One could also validate the format online, see http://yaml.org/spec/

Note: The entry point for the *Device Manager* configuration is the file that contains the server configuration and the mapping to the device configuration files. The configuration of each device

[5] http://yaml.org/spec/

should be given in a separate file for better readability and maintenance. Each device type uses the corresponding mapping file that defines the real names of the attributes in the OPC-UA address space.

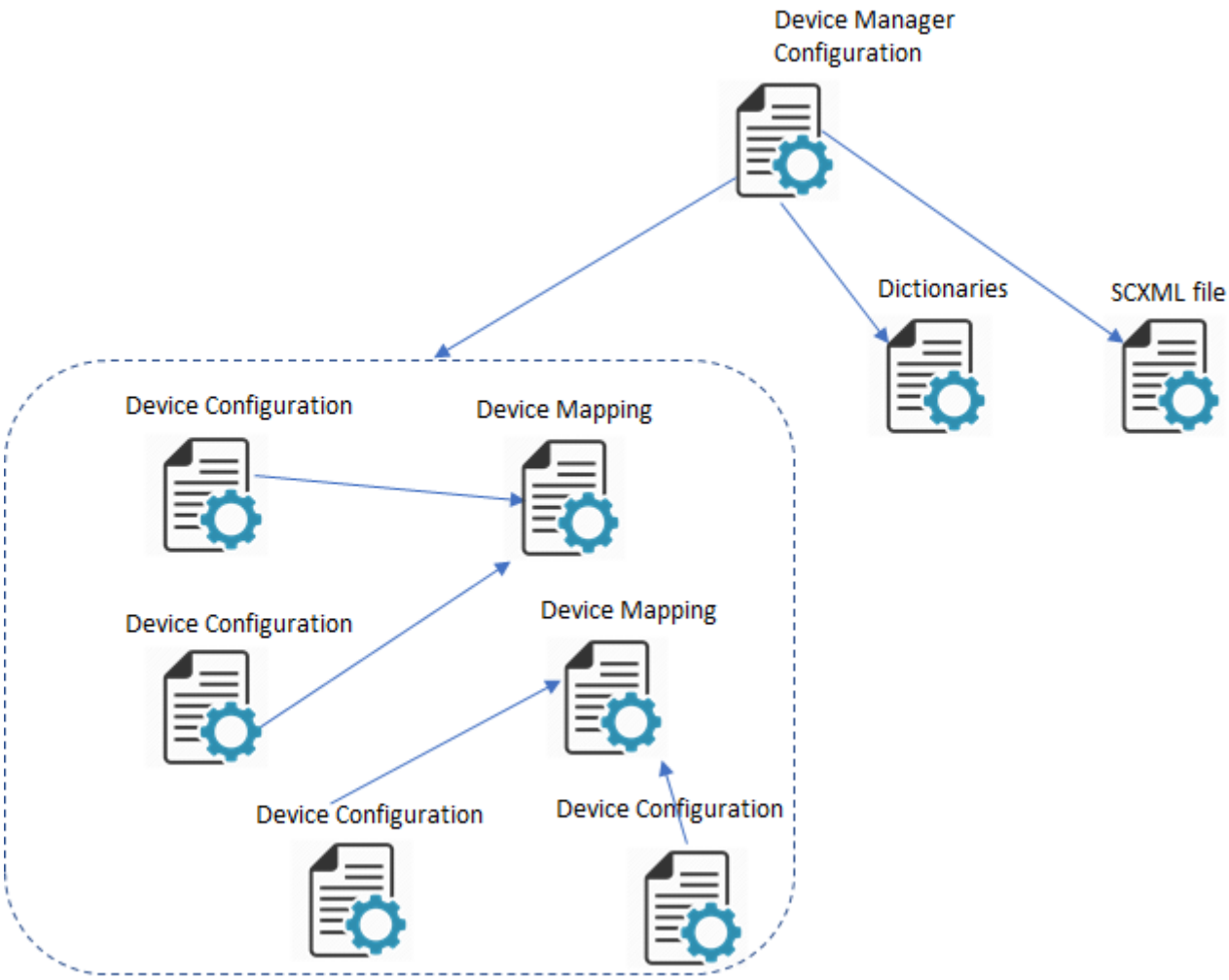


Fig. 2.4: Device Manager configuration files.

server_id

This is the id associated with the specific server. This id is used to associate all server configuration parameters as well as the prefix for the DB keys.

<server id>::req_endpoint

This is the endpoint for CII MAL request/reply. The server will listen to incoming commands using this endpoint.

<server id>::pub_endpoint

This is the endpoint for CII MAL pub/sub. The server will publish device topics using this endpoint.

<server id>::db_endpoint

This is the endpoint used by the server for connecting to the Redis DB.

<server id>::db_timeout

This is the server timeout for connecting to the Redis DB.

<server id>::scxml

This is the state machine specification file used by the server.

<server id>::fits_prefix

This is the prefix to be used for the FCS meta-data.

<server id>::devices

This is the list of devices active in the server configuration. Only devices listed here will be managed by the server.

<server id>::cmdtout

General command timeout for sending commands to the Local Control System (LCS).

<device id>::cfgfile

Configuration filename for a device.

<device id>::type

Device type.

An example of a server configuration is provided below.

Warning: Please note that some configuration keywords have been changed from version 1.0 due to the porting to CII MAL. An example of those changes are the configuration parameters req_endpoint and pub_endpoint.

```
server_id           : 'ins1.fcs1'
ins1.fcs1:
    req_endpoint    : "zpb.rr://127.0.0.1:12082/"
    pub_endpoint    : "zpb.ps://127.0.0.1:12345/"
    db_endpoint     : "127.0.0.1:6379"
    db_timeout      : 2
    scxml           : "fcf/devmgr/server/sm.xml"
    dictionaries    : ['dit/stddid/primary.did', 'fcf/devmgr/server/fcf.did']
    fits_prefix     : "FCS1"
    devices         : ['lamp1', 'shutter1', 'motor1', 'drot1', 'adc1', 'sensor1']
    cmdtout         : 60000

shutter1:
    type: Shutter
    cfgfile: "fcf/devmgr/server/shutter1.yml"
lamp1:
   type: Lamp
   cfgfile: "fcf/devmgr/server/lamp1.yml"
motor1:
   type: Motor
   cfgfile: "fcf/devmgr/server/motor1.yml"
drot1:
   type: Drot
   cfgfile: "fcf/devmgr/server/drot1.yml"
adc1:
   type: Adc
   cfgfile: "fcf/devmgr/server/adc1.yml"
sensor1:
   type: Sensor
   cfgfile: "fcf/devmgr/server/sensor1.yml"
```

2.3.2 Device Base Configuration

Each device has a common set of configuration parameters.

<device id>::type

It specifies the type of the device. Valid types are: *Shutter*, *Lamp*, *Motor*, *Sensor*, *Drot*, *Adc*, *Piezo* and *Actuator*.

<device id>::interface

It defines the communication interface that will be used by the device. At present, the only valid value is: *Softing*. This is the name of the OPC-UA toolkit used to communicate to the LCS (PLC). The needed libraries are included in the installation of the ELT standard machine.

<device id>::identifier

It defines the PLC object identifier.

<device id>::namespace

It defines the OPC-UA address space number.

<device id>::prefix

It defines the prefix for the address space nodeId of the device.

<device id>::simulated

Flag indicating if device is simulated.

<device id>::ignored

Flag indicating if the device is ignored. When a device is ignored, the device will ignore most of the commands received by the server until it receives the stop ignoring command (StopIgn).

<device id>::mapfile

File providing the configuration of the attributes in the OPC address space per each of the supported device types.

An example of a mapping file configuration is included below.

```
Shutter:
   cfg:
      low_closed:      cfg.bActiveLowClosed
      low_fault:       cfg.bActiveLowFault
      low_open:        cfg.bActiveLowOpen
      low_switch:      cfg.bActiveLowSwitch
      ignore_closed:   cfg.bIgnoreClosed
      ignore_fault:    cfg.bIgnoreFault
      ignore_open:     cfg.bIgnoreOpen
      initial_state:   cfg.bInitialState
      timeout:         cfg.nTimeout
   stat:
      state:          stat.nState
      substate:       stat.nSubstate
      local:          stat.bLocal
      error_code:     stat.nErrorCode
   rpc:
      rpcInit:        RPC_Init
      rpcEnable:      RPC_Enable
      rpcDisable:     RPC_Disable
      rpcClose:       RPC_Close
      rpcOpen:        RPC_Open
      rpcStop:        RPC_Stop
      rpcReset:       RPC_Reset
```

Note: With the information contained in the mapping file, combined with the PLC prefix and the namespace, the device obtains the NodeId for each of the attributes and the RPCs defined in the ICD with the device controller.

<device id>::fits_prefix

Prefix used by the device when generating the metadata information. This data is included in the FITS file generated by the server at the end of the exposure.

2.3.3 Shutter Specific Configuration

The *Shutter* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). All these parameters are under the `ctrl_config` heading.

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::low_closed	bool	yes	false	If true, the *closed* signal is active low.
ctrl_config::low_fault	bool	yes	false	If true, the *fault* signal is active low.
ctrl_config::low_open	bool	yes	false	If true, the *open* signal is active low.
ctrl_config::low_switch	bool	yes	false	If true, the *switch* signal is active low.
ctrl_config::ignore_closed	bool	yes	false	If true, the *closed* signal is ignored.
ctrl_config::ignore_fault	bool	yes	false	If true, the *fault* signal is ignored.
ctrl_config::ignore_open	bool	yes	false	If true, the *open* signal is ignored.
ctrl_config::initial_state	bool	yes	false	If true, the initial state for shutter will be open.
ctrl_config::timeout	uint	yes	3000 [ms]	Shutter timeout for transitions.

An example of a shutter configuration is given below.

```
shutter1:
  type: Shutter
  interface: Softing
  identifier: PLC1                                  # OPCUA Object Identifier
  prefix: MAIN.Shutter1                             # OPCUA attribute prefix
  simulated: true
  ignored: false
  address: opc.tcp://134.171.59.98:4
  simaddr: opc.tcp://127.0.0.1:7576                 # Simulation address
  mapfile: "fcf/devmgr/server/mapShutter.yml"
  fits_prefix: "SHUT1"
  ctrl_config:
     low_closed:      false                             # If T, signal is active low
     low_fault:       false                             # If T, signal is active low
     low_open:        false                             # If T, signal is active low
     low_switch:      false                             # If T, signal is active low
     ignore_closed:   false                             # If T, ignore the signal
     ignore_fault:    false                                 # If T, ignore the signal
     ignore_open:     false                                 # If T, ignore the signal
     initial_state:   false                                 # If T, initial state is␣
↪open
     timeout:         2000
```

2.3.4 Lamp Specific Configuration

The *Lamp* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). All these parameters are under the `ctrl_config` heading.

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::low_fault	bool	yes	false	If true, the *fault* signal is active low.
ctrl_config::low_on	bool	yes	false	If true, the *on* signal is active low.
ctrl_config::low_switch	bool	yes	false	If true, the *switch* signal is active low.
ctrl_config::ignore_fault	bool	yes	false	If true, the *fault* signal is ignored.
ctrl_config::invert_analog	bool	yes	false	If true, the analog feedback is active.
ctrl_config::initial_state	bool	yes	false	If true, the initial state will be switched on.
ctrl_config::analog_threshold	int	yes	0 [bits]	Analog feedback signal threshold
ctrl_config::analog_range	uint	yes	32767	Full range of A/D converter for analog output.
ctrl_config::cooldown	uint	yes	0 [s]	Cooldown time.
ctrl_config::maxon	uint	yes	0 [s]	Maximum time for the lamp to be On.
ctrl_config::warmup	uint	yes	0 [s]	Warmup time.
ctrl_config::timeout	uint	yes	3000 [ms]	Lamp timeout for transitions.

An example of a lamp configuration is given below. This configuration file can be found in module devmgr/server

```
lamp1:
  type: Lamp
  interface: Softing
  identifier: PLC1                                  # OPCUA Object Identifier
  prefix: MAIN.Lamp1                                # OPCUA attribute prefix
  simulated: false
  ignored: false
  address: opc.tcp://134.171.59.98:4840
  simaddr: opc.tcp://134.171.12.182:4840       # Simulation address
  mapfile: "fcf/devmgr/server/mapLamp.yml"
  fits_prefix: "LAMP1"
  ctrl_config:
     low_fault:         false                          # If T, signal is active low
```

(continues on next page)

(continued from previous page)

```
    low_on:             false                   # If T, signal is active low
    low_switch:         false                   # If T, signal is active low
    initial_state:      false                   # If T, initial state is on
    timeout:            2000
```

2.3.5 Sensor Specific Configuration

The sensor devices defines currently no configuration that will be downloaded to the LCS. However, it defines the configuration of the sensor channels. The sensor channels are known only at the server side.

Config Item	Type	Optional	Default	Description
readonly	bool	yes	false	Flag to indicate that sensor will only read the data from the controller without attempting to execute RPC calls. This flag is used for sensors running in commercial devices not running in a PLC but having an embedded OPC-UA server. To be used only in special cases.
ctrl_config::timeout	uint	yes	3000 [ms]	Sensor timeout for transitions.
channels	vector	no	na	List of channels ids.
<channel id>::description	string	yes	""	Channel description.
<channel id>::type	string	no	na	Channel type. Allowed types are: DI (bool), AI (double), II (integer), ST (string).
<channel id>::header	bool	yes	true	If true, the channel will be included in the metadata FITS file.
<channel id>::log	bool	yes	true	If true, the sensor value will be logged (Not available yet !).
<channel id>::map	string	no	na	Channel internal mapping to the name in the LCS.
<channel id>::prefix	string	no	na	Channel FITS prefix.
<channel id>::unit	string	yes		Channel unit.

An example of a sensor configuration is given below. This configuration file can be found in module devmgr/server. In this case, the sensor device has two channels: *ch1* and *ch2*.

```
sensor1:
   type: Sensor
   interface: Softing
   identifier: PLC1
   prefix: MAIN.IODev1
   simulated: false
```

(continues on next page)

(continued from previous page)

```
ignored: false
address: opc.tcp://134.171.59.98:4840
simaddr: opc.tcp://134.171.57.209:4840
mapfile: "fcf/devmgr/server/mapSensor.yml"
fits_prefix: "SENSOR1"

ctrl_config:
   timeout:          20000

channels: ['ch1', 'ch2']
ch1:
    description: "channel1"
    fits_prefix: "CH1 STAT"
    type: DI
    header: true
    log: true
    unit: mm
    map: di1
ch2:
    description: "channel2"
    fits_prefix: "CH2 STAT"
    type: DI
    header: true
    log: true
    unit: dd
    map: di2
```

2.3.6 Motor Specific Configuration

The *Motor* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). All these parameters are under the `ctrl_config` heading.

The motor also defines a set of configuration parameters that are only known at the server level, for instance the named positions of the motor.

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::axis_type	string	yes	LINEAR	Axis type. Allowed options are: *LINEAR*, *CIRCULAR* and *CIRCULAR_OPT*.
ctrl_config::min_pos	double	yes	0 [uu]	Minimum position in user units.
ctrl_config::max_pos	double	yes	0 [uu]	Maximum position in user units.
ctrl_config::velocity	double	yes	1.0 [uu/s]	Default velocity for moving the motor in position mode
ctrl_config::active_low_lstop	bool	yes	false	If true, the *Lower Stop* signal is active low.
ctrl_config::active_low_lhw	bool	yes	false	If true, the *Lower Hw* signal is active low.
ctrl_config::active_low_ref	bool	yes	false	If true, the *Reference* signal is active low.
ctrl_config::active_low_index	bool	yes	false	If true, the *Index* signal is active low.
ctrl_config::active_low_ustop	bool	yes	false	If true, the *Upper Stop* signal is active low.
ctrl_config::active_low_uhw	bool	yes	false	If true, the *Upper Hw* signal is active low.
ctrl_config::low_brake	bool	yes	false	If true, the *Brake* signal is active low.
ctrl_config::low_inpos	bool	yes	false	If true, the *In Position* signal is active low.
ctrl_config::backlash	double	yes	0 [uu]	Backlash compensation. If value is zero means no backlash compensation is active.
ctrl_config::disable	bool	yes	false	If true, the power of the motor will be disabled after positioning.
ctrl_config::lock	bool	yes	false	If true, the motor position will be locked
ctrl_config::lock_pos	double	yes	0 [uu]	Position that will be locked in case lock configuration is activated.
ctrl_config::lock_tolerance	double	yes	0 [us]	Tolerance of the lock position
ctrl_config::tout_init	int	yes	60000 [ms]	Motor initialisation timeout.
ctrl_config::tout_move	int	yes	60000 [ms]	Motor move timeout.
ctrl_config::tout_switch	int	yes	150000 [ms]	Motor timeout for going out of the switch during initialisation.

Motor Initialisation

Note: The motor has a set of configuration parameters dedicated to the motor initialisation sequence. The initialisation sequence is downloaded to the LCS only when device controller is not operational.

<device id>::initialisation::sequence

List of motor initialization steps. The allowed steps are:

Table 2.1: Motor Initialisation Steps

Step	Description	Parameter 1	Parameter 2
END	Finish the sequence, no more actions are performed.	na	na
FIND_INDEX	Motor moves until finding the index pulse.	Fast velocity [UU/s]	Slow velocity [UU/s]
FIND_REF_LE	Motor moves until finding lower edge of reference switch.	Fast velocity [UU/s]	Slow velocity [UU/s]
FIND_REF_UE	Motor moves until finding upper edge of reference switch.	Fast velocity [UU/s]	Slow velocity [UU/s]
FIND_LHW	Motor moves until finding lower hardware limit.	Fast velocity [UU/s]	Slow velocity [UU/s]
FIND_UHW	Motor moves until finding upper hardware limit.	Fast velocity [UU/s]	Slow velocity [UU/s]
DELAY	Motor wait for a fixed amount of time before to continue.	time in [ms]	na
MOVE_ABS	Motor moves to an absolute position.	Velocity [UU/s]	Target position [UU]
MOVE_REL	Motor moves to a relative position.	Velocity [UU/s]	Target position [UU]
CALIB_ABS	Motor calibrates an absolute position.	Position [UU]	na
CALIB_REL	Motor calibrates a relative position.	Position [UU]	na
CALIB_SWITCH	Motor calibrates switch position.	Position [UU]	na

Note: Some of the initialisation steps require parameters, for instance the speed of the motor. These parameters are defined together with the initialisation step.

<device id>::initialisation::<init step>::value1

Parameter 1 of the initialisation step.

<device id>::initialisation::<init step>::value2

Parameter 2 of the initialisation step.

Note: In case parameters are not applicable (*na*) please use 0 instead, for instance *END, 0, 0*

Named Positions

The motor device supports a configuration of named positions that associate specific motor position in user units (UU) to names. The aim of name positions is to facilitate the setting of motor positions by end users.

<device id>::positions::posnames

List of position names ids.

<device id>::positions::<posname id>

Value of the position name in user units (UU).

<device id>::positions::tolerance

Tolerance of the named position in user units (UU). If the actual position is within the tolerance, the device will report the named position otherwise its name will be empty.

Note: An example of a motor configuration is given below. This configuration file can be found in module devmgr/server.

```
motor1:
  type: Motor
  interface: Softing
```

(continues on next page)

(continued from previous page)

```
identifier: PLC1                                    # OPCUA Object Identifier
prefix: MAIN.Synchro1                               # OPCUA attribute prefix
simulated: false
ignored: false
address: opc.tcp://134.171.59.98:4840
simaddr: opc.tcp://134.171.57.209:4840              # Simulation address
mapfile: "fcf/devmgr/server/mapMotor.yml"
fits_prefix: "MOT1"
ctrl_config:
   velocity:              3.0
   min_pos:               0.0
   max_pos:               359.0
   axis_type:             CIRCULAR
   active_low_lstop:      false
   active_low_lhw:        false
   active_low_ref:        true
   active_low_index:      false
   active_low_uhw:        true
   active_low_ustop:      false
   brake:                 false
   low_brake:             false
   low_inpos:             false
   backlash:              0.0
   tout_init:             30000
   tout_move:             120000
   tout_switch:           10000
initialisation:
    sequence: ['FIND_LHW', 'FIND_UHW', 'CALIB_ABS', 'END']
    FIND_LHW:
      value1: 4.0
      value2: 4.0
    FIND_UHW:
      value1: 4.0
      value2: 4.0
    CALIB_ABS:
      value1: 0.0
      value2: 0.0
    END:
      value1: 0.0
      value2: 0.0
positions:
    posnames: ['ON', 'OFF']
    tolerance: 1.0                                   # Tolerance in UU
    ON:   30.0
    OFF: 100.0
```

2.3.7 Derotator Specific Configuration

As for other devices, the *Derotator* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). All these parameters are under the `ctrl_config` heading.

Since the *Derotator* is just an aggregated motor device, it includes all *Motor* configuration parameters (see *fcf_devmgr_motor_config_ref*) plus a few parameters specific to derotators.

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::dir_sign	int	yes	1	Motor direction sign
ctrl_config::focus_sign	int	yes	-1	Focus direction sign.
ctrl_config::trk_period	int	yes	20 [ms]	Period of the tracking corrections within the PLC.
ctrl_config::stat_ref	double	yes	0.0 [uu]	Reference position for stationary mode.
ctrl_config::sky_ref	double	yes	0.0 [uu]	Reference position for sky mode.
ctrl_config::user_ref	double	yes	0.0 [uu]	Reference position for user mode.
ctrl_config::user_par1	double	yes	0.0	Specific parameter 1 for user mode.
ctrl_config::user_par2	double	yes	0.0	Specific parameter 2 for user mode.
ctrl_config::user_par3	double	yes	0.0	Specific parameter 3 for user mode.
ctrl_config::user_par4	double	yes	0.0	Specific parameter 4 for user mode.

Note: An example of a Derotator configuration is given below. This configuration file can be found in module devmgr/server.

```
drot1:
    type: Drot
    interface: Softing
```

(continues on next page)

(continued from previous page)

```
    identifier: PLC1                                    # OPCUA Object Identifier
    prefix: MAIN_FAST.drot                              # OPCUA attribute prefix
    simulated: false
    ignored: false
    address: opc.tcp://134.171.59.98:4840
    simaddr: opc.tcp://134.171.57.209:4840              # Simulation address
    mapfile: "fcf/devmgr/server/mapDrot.yml"
    fits_prefix: "DROT1"
    ctrl_config:
        latitude:               -0.429833092
        longitude:              1.228800386
        velocity:               3.0
        min_pos:                -359
        max_pos:                359.0
        axis_type:              CIRCULAR
        active_low_lstop:       false
        active_low_lhw:         false
        active_low_ref:         true
        active_low_index:       false
        active_low_uhw:         true
        active_low_ustop:       false
        brake:                  false
        low_brake:              false
        low_inpos:              false
        backlash:               0.0
        tout_init:              30000
        tout_move:              120000
        tout_switch:            10000
    initialisation:
        sequence: ['FIND_LHW', 'FIND_UHW', 'CALIB_ABS', 'END']
        FIND_LHW:
            value1: 4.0
            value2: 4.0
        FIND_UHW:
            value1: 4.0
            value2: 4.0
        CALIB_ABS:
            value1: 0.0
            value2: 0.0
        END:
            value1: 0.0
            value2: 0.0
    positions:
        posnames: ['ON', 'OFF']
        tolerance: 1.0                                  # Tolerance in UU
        ON:   30.0
        OFF: 100.0
```

2.3.8 Derotator Control

Operation modes

Alias	Name	Description
eng	Engineering	In this mode, the *Derotator* behaves like a standard motor. This means that it can be moved in user units and encoders.
stat	Stationary	In this mode, the *Derotator* is stationary and it can be positioned at given angle according to the following formula: **pos := stat_ref + dir_sign * (posang)/2.0;**

sky	Sky	The *Derotator* tracks following the field rotation.
		fieldRotation := parallactic - focus_sign * altitude;
		pos := sky_ref + dir_sign * (posang - fieldRotation)/2;
		angleOnSky := posang;
		modeAngle := angleOnSky;

elev	Elevation	The *Derotator* tracks following the pupil rotation.
		pos := elev_ref + (focus_sign * dir_sign * altitude) /2.0;
		angleOnSky := parallactic;
		modeAngle := angleOnSky;

user	User	The *Derotator* tracks according to the user custom computation.

2.3.9 ADC Specific Configuration

As for other devices, the *ADC* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). These parameters are under the `ctrl_config` heading. Considering that the ADC is a multi-axis device, it includes as well the configuration of two standard motor devices. The configuration of each motor device is defined in separate files and they correspond to the configuration of a standard motor device (see *fcf_devmgr_motor_config_ref*).

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::axes	list	no		List of motors controlled by the ADC
ctrl_config::trk_period	int	yes	20 [ms]	Period of the tracking corrections within the PLC.
ctrl_config::pslope	double	yes	0.0023 [arcsec/mbar]	Pressure slope.

Continued on next page

Table 2.2 – continued from previous page

Config Item	Type	Optional	Default	Description
ctrl_config::poffset	double	yes	743.0 [mbar]	Pressure offset.
ctrl_config::tslope	double	yes	-0.0061 [arcsec/C]	Temperature slope.
ctrl_config::toffset	double	yes	12 [C]	Temperature offset.
ctrl_config::afactor	double	yes	3.32 [1/arcsec]	A Factor
ctrl_config::zdlimit	double	yes	0.0174533	Zenith distance limit
ctrl_config::minelev	double	yes	27.64 [deg]	Minimum Elevation.
ctrl_config::mot1_signoff	int	yes	1	Motor 1 sign for off mode
ctrl_config::mot1_signauto	int	yes	1	Motor 1 sign for auto mode
ctrl_config::mot1_signphi	int	yes	1	Motor 1 sign for phi
ctrl_config::mot1_refoff	double	yes	0 [deg]	Motor 1 offset for off mode
ctrl_config::mot1_refauto	double	yes	0 [deg]	Motor 1 offset for auto mode
ctrl_config::mot1_coffset	double	yes	1.7387 [arcsec]	Motor 1 C parameter
ctrl_config::mot1_poffset	double	yes	90 [deg]	Motor 1 Position offset
ctrl_config::mot1_drotfactor	double	yes	2	Motor 1 derotator offset
ctrl_config::mot2_signoff	int	yes	1	Motor 2 sign for off mode
ctrl_config::mot2_signauto	int	yes	1	Motor 2 sign for auto mode
ctrl_config::mot2_signphi	int	yes	1	Motor 2 sign for phi
ctrl_config::mot2_refoff	double	yes	0 [deg]	Motor 2 reference position for off mode
ctrl_config::mot2_refauto	double	yes	0 [deg]	Motor 2 reference position for auto mode
ctrl_config::mot2_coffset	double	yes	1.7387 [arcsec]	Motor 2 C parameter
ctrl_config::mot2_poffset	double	yes	90 [deg]	Motor 2 Position offset

Continued on next page

Table 2.2 – continued from previous page

Config Item	Type	Optional	Default	Description
ctrl_config::mot2_drotfactor	double	yes	2	Motor 2 derotator offset
<motor1>::prefix	string	no		Internal name used by the ADC for motor1 (fixed)
<motor1>::cfgfile	string	no		File path for the motor 1 configuration.
<motor2>::prefix	string	no		Internal name used by the ADC for motor2 (fixed)
<motor2>::cfgfile	string	no		File path for the motor 2 configuration.

Note: An example of an Adc configuration is given below. This configuration file can be found in module devmgr/server.

```
adc1:
    type: Adc
    interface: Softing
    identifier: PLC1                              # OPCUA Object Identifier
    prefix: MAIN_FAST.adc                        # OPCUA attribute prefix
    simulated: false
    ignored: false
    address: opc.tcp://134.171.59.98:4840
    simaddr: opc.tcp://134.171.57.209:4840       # Simulation address
    mapfile: "fcf/devmgr/server/mapAdc.yml"
    fits_prefix: "ADC1"
    ctrl_config:
        axes: ['adc1_motor1', 'adc1_motor2']

    adc1_motor1:
        prefix: "motor1"
        cfgfile: "devmgr/server/adc1Motor1.yml"
    adc1_motor2:
        prefix: "motor2"
        cfgfile: "devmgr/server/adc1Motor2.yml"
```

2.3.10 ADC Control

Operation modes

The *ADC* operates two motorized functions. In engineering mode, each motor can be controlled independently.

Alias	Name	Description
eng	Engineering	In this mode, the *ADC* behaves like the standard motor. This means that each motor can be moved in user units and encoders.
off	Off	In this mode, the *ADC* is stationary and it can be positioned at given angle according to the following formula: **pos := off_ref + sign_off * posang;**
auto	Auto	The *ADC* tracks following the default formula. This formula can be replaced by the user in order to accommodate instrument specific requirements.

2.3.11 Piezo Specific Configuration

The *Piezo* device defines a set of configuration parameters that will be transferred to the device controller running in the LCS (PLC). All these parameters are under the `ctrl_config` heading.

Warning: The *ctrl_config* parameters are downloaded to the device controller when the device is not *Operational*. If the controller is already *Operational*, the user shall force the transition from *Operational* to *NotOperational/NotReady* and back to Operational.

Config Item	Type	Optional	Default	Description
ctrl_config::num_axes	short	no	3	Configured number of piezo axes. This parameter gives flexibility to adapt to different type of piezos.
ctrl_config::max_on	int	yes	0	Maximum time that outputs will be maintained. If it is zero means there is no time counter.
ctrl_config::full_range[]	short	yes	32767	Full range per axes in bits.
ctrl_config::home[]	double	yes	0	Home position per axes in user units.
ctrl_config::lower_limit[]	double	yes	0	lower limit per axes in user units.
ctrl_config::upper_limit[]	double	yes	32767	upper limit per axes in user units.
ctrl_config::user_to_bit_input[]	double	yes	3276.7	user to bit conversion factor per axes for inputs.
ctrl_config::user_offset_input[]	double	yes	0	user offset per axes for inputs.
ctrl_config::user_to_bit_output[]	double	yes	3276.7	user to bit conversion factor per axes for outputs.
ctrl_config::user_offset_output[]	double	yes	0	user offset per axes for outputs.

An example of a piezo configuration is provided below.

```
piezo1:
    type: Piezo
    interface: Softing
    identifier: PLC1                                # OPCUA Object Identifier
    prefix: MAIN.Piezo1                           # OPCUA attribute prefix
    simulated: false
    address: opc.tcp://134.171.59.98:4840
    simaddr: opc.tcp://134.171.57.209:4840        # Simulation address
    mapfile: "fcf/devmgr/devices/mapPiezo.yml"
    fits_prefix: "MOT1"
    ctrl_config:
        num_axis:       3
        max_on:         0
        full_range1:     32767
        full_range2:     32767
        full_range3:     32767
        home1:           0
        home2:           0
        home3:           0
```

(continues on next page)

(continued from previous page)

```
        lower_limit1:    0
        lower_limit2:    0
        lower_limit3:    0
        upper_limit1:    32767
        upper_limit2:    32767
        upper_limit3:    32767
        user_to_bit_input1:    3276.7
        user_to_bit_input2:    3276.7
        user_to_bit_input3:    3276.7
        user_offset_input1:    0
        user_offset_input2:    0
        user_offset_input3:    0
        user_to_bit_output1:   3276.7
        user_to_bit_output2:   3276.7
        user_to_bit_output3:   3276.7
        user_offset_output1:   0
        user_offset_output2:   0
        user_offset_output3:   0
```

2.3.12 Actuator Specific Configuration

The *Actuator* is one of the few devices that is not transferring any configuration to the controller in the transition from Ready to Operational. The *Actuator* assumes to have all the configuration defined in the controller (PLC). For knowing the controller configuration, please refer to the PLC Actuator section.

An example of a actuator configuration is provided below.

```
actuator1:
    type: Actuator
    interface: Softing
    identifier: PLC1                                # OPCUA Object Identifier
    prefix: MAIN.Actuator1                          # OPCUA attribute prefix
    simulated: false
    ignored: false
    address: opc.tcp://134.171.59.99:4840
    simaddr: opc.tcp://134.171.12.182:4840      # Simulation address
    mapfile: "fcf/devmgr/server/mapActuator.yml"
    fits_prefix: "MECH1"
    ctrl_config:
```

2.4 Database Attributes

The *Device Manager* uses the Redis DB to store the actual server configuration and run-time parameters. The Redis keys used by the server follow a hierarchical naming convention starting with the id of the server. Specific keys for devices use the id of the device in the name. The DB keys can be monitored using the *dbbrowser* utility. All *Device Manager* keys have a flat structure in Redis DB.

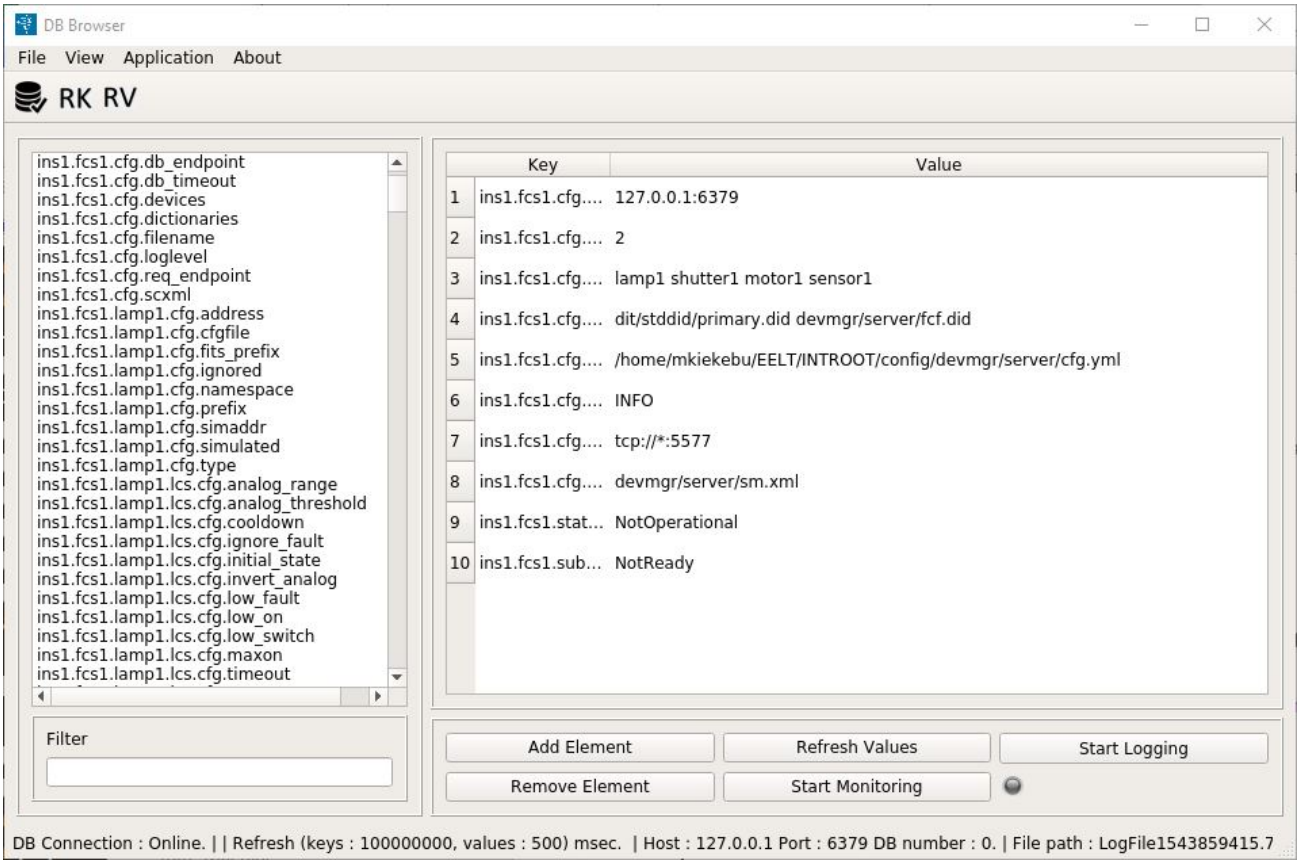


Fig. 2.5: *dbbrowser* utility

2.4.1 Server configuration

The server stores the actual values of the server configuration parameters into the Redis DB . This helps to verify whether the configuration has been loaded correctly. For details of the server configuration parameters, see *Device Manager Configuration*.

Table 2.3: Server configuration Redis DB keys

Redis Key
<instrument id>.<server id>.cfg.db_endpoint
<instrument id>.<server id>.cfg.db_timeout
<instrument id>.<server id>.cfg.device
<instrument id>.<server id>.cfg.dictionaries
<instrument id>.<server id>.cfg.filename
<instrument id>.<server id>.cfg.loglevel
<instrument id>.<server id>.cfg.req_endpoint
<instrument id>.<server id>.cfg.scxml

2.4.2 Server Status

The server stores the string representation of its state and substate into the Redis DB.

Table 2.4: Server status Redis DB keys

Redis Key
<instrument id>.<server id>.state_str
<instrument id>.<server id>.substate_str

2.4.3 Common Device Keys

Each device has a number of common Redis DB keys.

Table 2.5: Common device Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.cfg.address
<instrument id>.<server id>.<device id>.cfg.simaddr
<instrument id>.<server id>.<device id>.cfg.cfgfile
<instrument id>.<server id>.<device id>.cfg.fits_prefix
<instrument id>.<server id>.<device id>.cfg.ignored
<instrument id>.<server id>.<device id>.cfg.simulated
<instrument id>.<server id>.<device id>.cfg.namespace
<instrument id>.<server id>.<device id>.cfg.prefix
<instrument id>.<server id>.<device id>.cfg.type

2.4.4 Shutter

Each shutter device defines a set of specific Redis DB keys:

Table 2.6: Shutter Specific Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.ignore_closed
<instrument id>.<server id>.<device id>.lcs.cfg.ignore_fault
<instrument id>.<server id>.<device id>.lcs.cfg.ignore_open
<instrument id>.<server id>.<device id>.lcs.cfg.initial_state
<instrument id>.<server id>.<device id>.lcs.cfg.low_closed
<instrument id>.<server id>.<device id>.lcs.cfg.low_fault
<instrument id>.<server id>.<device id>.lcs.cfg.low_open
<instrument id>.<server id>.<device id>.lcs.cfg.low_switch
<instrument id>.<server id>.<device id>.lcs.cfg.timeout
<instrument id>.<server id>.<device id>.lcs.stat.error_code
<instrument id>.<server id>.<device id>.lcs.stat.error_str
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate

2.4.5 Lamp

Each lamp device defines a set of specific Redis DB keys:

Table 2.7: Lamp Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.analog_range
<instrument id>.<server id>.<device id>.lcs.cfg.analog_threshold
<instrument id>.<server id>.<device id>.lcs.cfg.cooldown
<instrument id>.<server id>.<device id>.lcs.cfg.ignore_fault
<instrument id>.<server id>.<device id>.lcs.cfg.initial_state
<instrument id>.<server id>.<device id>.lcs.cfg.invert_analog
<instrument id>.<server id>.<device id>.lcs.cfg.low_fault
<instrument id>.<server id>.<device id>.lcs.cfg.low_on
<instrument id>.<server id>.<device id>.lcs.cfg.low_switch
<instrument id>.<server id>.<device id>.lcs.cfg.maxon
<instrument id>.<server id>.<device id>.lcs.cfg.timeout
<instrument id>.<server id>.<device id>.lcs.cfg.warmup
<instrument id>.<server id>.<device id>.lcs.stat.error_code
<instrument id>.<server id>.<device id>.lcs.stat.error_str
<instrument id>.<server id>.<device id>.lcs.stat.intensity
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate

2.4.6 Sensor

Each sensor device defines a set of specific Redis DB keys:

Table 2.8: Sensor Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.description
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.fits_prefix
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.header
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.log
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.map
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.type
<instrument id>.<server id>.<device id>.cfg.channels.<channel id>.unit
<instrument id>.<server id>.<device id>.lcs.stat.<channel id>
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate

2.4.7 Motor

Each motor device defines a set of specific Redis DB keys:

Table 2.9: Motor Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_indec
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_lhw
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_lstop
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_ref
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_uhw
<instrument id>.<server id>.<device id>.lcs.cfg.active_low_ustop
<instrument id>.<server id>.<device id>.lcs.cfg.axis_type
<instrument id>.<server id>.<device id>.lcs.cfg.backlash
<instrument id>.<server id>.<device id>.lcs.cfg.brake
<instrument id>.<server id>.<device id>.lcs.cfg.check_inpos
<instrument id>.<server id>.<device id>.lcs.cfg.disable
<instrument id>.<server id>.<device id>.lcs.cfg.exec_post_init
<instrument id>.<server id>.<device id>.lcs.cfg.exec_post_move
<instrument id>.<server id>.<device id>.lcs.cfg.exec_pre_init
<instrument id>.<server id>.<device id>.lcs.cfg.exec_pre_move
<instrument id>.<server id>.<device id>.lcs.cfg.init_seq<number>_action
<instrument id>.<server id>.<device id>.lcs.cfg.init_seq<number>_value1
<instrument id>.<server id>.<device id>.lcs.cfg.init_seq<number>_value2
<instrument id>.<server id>.<device id>.lcs.cfg.lock
<instrument id>.<server id>.<device id>.lcs.cfg.lock_pos
<instrument id>.<server id>.<device id>.lcs.cfg.lock_tolerance
<instrument id>.<server id>.<device id>.lcs.cfg.low_brake
<instrument id>.<server id>.<device id>.lcs.cfg.low_inpos
<instrument id>.<server id>.<device id>.lcs.cfg.max_pos
<instrument id>.<server id>.<device id>.lcs.cfg.min_pos
<instrument id>.<server id>.<device id>.lcs.cfg.tout_init
<instrument id>.<server id>.<device id>.lcs.cfg.tout_move
<instrument id>.<server id>.<device id>.lcs.cfg.tout_switch
<instrument id>.<server id>.<device id>.lcs.cfg.velocity
<instrument id>.<server id>.<device id>.lcs.stat.axis_brake
<instrument id>.<server id>.<device id>.lcs.stat.axis_enable
<instrument id>.<server id>.<device id>.lcs.stat.axis_info_data1
<instrument id>.<server id>.<device id>.lcs.stat.inposition
<instrument id>.<server id>.<device id>.lcs.stat.lock
<instrument id>.<server id>.<device id>.lcs.stat.ready
<instrument id>.<server id>.<device id>.lcs.stat.error_code
<instrument id>.<server id>.<device id>.lcs.stat.error_str

Continued on next page

Table 2.9 – continued from previous page

Redis Key
<instrument id>.<server id>.<device id>.lcs.stat.init_action
<instrument id>.<server id>.<device id>.lcs.stat.init_step
<instrument id>.<server id>.<device id>.lcs.stat.initialised
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.mode
<instrument id>.<server id>.<device id>.lcs.stat.pos_actual
<instrument id>.<server id>.<device id>.lcs.stat.pos_error
<instrument id>.<server id>.<device id>.lcs.stat.pos_target
<instrument id>.<server id>.<device id>.lcs.stat.scale_factor
<instrument id>.<server id>.<device id>.lcs.stat.signal_index
<instrument id>.<server id>.<device id>.lcs.stat.signal_lhw
<instrument id>.<server id>.<device id>.lcs.stat.signal_lstop
<instrument id>.<server id>.<device id>.lcs.stat.signal_ref
<instrument id>.<server id>.<device id>.lcs.stat.signal_uhw
<instrument id>.<server id>.<device id>.lcs.stat.signal_ustop
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate
<instrument id>.<server id>.<device id>.lcs.stat.vel_actual
<instrument id>.<server id>.<device id>.pos_actual_name
<instrument id>.<server id>.<device id>.pos_enc
<instrument id>.<server id>.<device id>.target_enc

2.4.8 Derotator

The *Derotator* device uses the same set of Redis keys as the *Motor* device plus some additional derotator specific ones that are described below:

Table 2.10: Derotator Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.trk_period
<instrument id>.<server id>.<device id>.lcs.cfg.user_par1
<instrument id>.<server id>.<device id>.lcs.cfg.user_par2
<instrument id>.<server id>.<device id>.lcs.cfg.user_par3
<instrument id>.<server id>.<device id>.lcs.cfg.user_par4
<instrument id>.<server id>.<device id>.lcs.cfg.user_ref
<instrument id>.<server id>.<device id>.lcs.cfg.sky_ref
<instrument id>.<server id>.<device id>.lcs.cfg.stat_ref
<instrument id>.<server id>.<device id>.lcs.stat.angle_on_sky
<instrument id>.<server id>.<device id>.lcs.stat.alpha
<instrument id>.<server id>.<device id>.lcs.stat.delta
<instrument id>.<server id>.<device id>.lcs.stat.track_mode

2.4.9 ADC

The *Adc* device defines a set of specific Redis keys:

Table 2.11: Adc Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.trk_period
<instrument id>.<server id>.<device id>.lcs.cfg.afactor
<instrument id>.<server id>.<device id>.lcs.cfg.minelev
<instrument id>.<server id>.<device id>.lcs.cfg.poffset
<instrument id>.<server id>.<device id>.lcs.cfg.pslope
<instrument id>.<server id>.<device id>.lcs.cfg.toffset
<instrument id>.<server id>.<device id>.lcs.cfg.tslope
<instrument id>.<server id>.<device id>.lcs.cfg.zdlimit
<instrument id>.<server id>.<device id>.lcs.cfg.mot1_coffset
<instrument id>.<server id>.<device id>.lcs.cfg.mot1_drotfactor
<instrument id>.<server id>.<device id>.lcs.cfg.mot1_poffset
<instrument id>.<server id>.<device id>.lcs.cfg.mot1_refauto
<instrument id>.<server id>.<device id>.lcs.cfg.mot1_refoff
<instrument id>.<server id>.<device id>.lcs.cfg.mot2_coffset
<instrument id>.<server id>.<device id>.lcs.cfg.mot2_drotfactor
<instrument id>.<server id>.<device id>.lcs.cfg.mot2_poffset
<instrument id>.<server id>.<device id>.lcs.cfg.mot2_refauto
<instrument id>.<server id>.<device id>.lcs.cfg.mot2_refoff
<instrument id>.<server id>.<device id>.cfg.ignored
<instrument id>.<server id>.<device id>.cfg.simulated
<instrument id>.<server id>.<device id>.lcs.stat.alpha
<instrument id>.<server id>.<device id>.lcs.stat.delta
<instrument id>.<server id>.<device id>.lcs.stat.error_code
<instrument id>.<server id>.<device id>.lcs.stat.error_str
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate
<instrument id>.<server id>.<device id>.lcs.stat.track_mode
<instrument id>.<server id>.<device id>.lcs.stat.motor1.axis_brake
<instrument id>.<server id>.<device id>.lcs.stat.motor1.axis_enable
<instrument id>.<server id>.<device id>.lcs.stat.motor1.axis_lock
<instrument id>.<server id>.<device id>.lcs.stat.motor1.local
<instrument id>.<server id>.<device id>.lcs.stat.motor1.pos_actual
<instrument id>.<server id>.<device id>.lcs.stat.motor1.pos_enc
<instrument id>.<server id>.<device id>.lcs.stat.motor1.scale_factor
<instrument id>.<server id>.<device id>.lcs.stat.motor2.axis_brake
<instrument id>.<server id>.<device id>.lcs.stat.motor2.axis_enable

Continued on next page

Table 2.11 – continued from previous page

Redis Key
<instrument id>.<server id>.<device id>.lcs.stat.motor2.axis_lock
<instrument id>.<server id>.<device id>.lcs.stat.motor2.local
<instrument id>.<server id>.<device id>.lcs.stat.motor2.pos_actual
<instrument id>.<server id>.<device id>.lcs.stat.motor2.pos_enc
<instrument id>.<server id>.<device id>.lcs.stat.motor2.scale_factor
<instrument id>.<server id>.<device id>.motor1.pos_enc
<instrument id>.<server id>.<device id>.motor2.pos_enc

2.4.10 Piezo

The *Piezo* device defines a set of specific Redis keys:

Table 2.12: Piezo Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.cfg.full_range1
<instrument id>.<server id>.<device id>.lcs.cfg.full_range2
<instrument id>.<server id>.<device id>.lcs.cfg.full_range3
<instrument id>.<server id>.<device id>.lcs.cfg.home1
<instrument id>.<server id>.<device id>.lcs.cfg.home2
<instrument id>.<server id>.<device id>.lcs.cfg.home3
<instrument id>.<server id>.<device id>.lcs.cfg.lower_limit1
<instrument id>.<server id>.<device id>.lcs.cfg.lower_limit2
<instrument id>.<server id>.<device id>.lcs.cfg.lower_limit3
<instrument id>.<server id>.<device id>.lcs.cfg.upper_limit1
<instrument id>.<server id>.<device id>.lcs.cfg.upper_limit2
<instrument id>.<server id>.<device id>.lcs.cfg.upper_limit3
<instrument id>.<server id>.<device id>.lcs.cfg.max_on
<instrument id>.<server id>.<device id>.lcs.cfg.num_axes
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_input1
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_input2
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_input3
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_input1
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_input2
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_input3
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_output1
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_output2
<instrument id>.<server id>.<device id>.lcs.cfg.user_offset_output3
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_output1
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_output2
<instrument id>.<server id>.<device id>.lcs.cfg.user_to_bit_output3

Continued on next page

Table 2.12 – continued from previous page

Redis Key
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_bit1
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_bit2
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_bit3
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_user1
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_user2
<instrument id>.<server id>.<device id>.lcs.stat.actual_pos_user3
<instrument id>.<server id>.<device id>.lcs.stat.error_codes
<instrument id>.<server id>.<device id>.lcs.stat.error_str
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate

2.4.11 Actuator

Each actuator device defines a set of specific Redis DB keys:

Table 2.13: Actuator Redis DB keys

Redis Key
<instrument id>.<server id>.<device id>.lcs.stat.local
<instrument id>.<server id>.<device id>.lcs.stat.state
<instrument id>.<server id>.<device id>.lcs.stat.substate
<instrument id>.<server id>.<device id>.lcs.stat.error_code
<instrument id>.<server id>.<device id>.lcs.stat.error_str

2.5 Commands

The commands currently supported by the server are listed here: *List of Commands*.

2.5.1 Error Handling

FCF Commands throw exceptions in case of errors or timeouts. Client applications can catch the exceptions and obtain the error message associated with the function **getDesc()**. This error does not contain neither the history nor the error stack but it normally indicates precisely where the error occurred.

```
try {
    auto reply = client->GetStatus();
 } catch (const fcfif::ExceptionErr& e) {
    RAD_LOG_ERROR() << "Error reply " << e.getDesc()  << ").";
}
```

2.5.2 Serialization

The *Device Manager* uses the CII MAL ZPB (ZeroMQ + Google Proto buffers) for serialising commands.

Note: Each command has two parts: a payload and its corresponding reply, see the details in the *fcfif* module. The normal replies are plain strings.

Setup Command

The *Setup* command is intended to produce a change in the run-time configuration. It is also a way of triggering operational actions on the devices. It is possible to switch a lamp on, close a shutter and move a motor in separate messages or within the same one. This means that the content of the message varies. The devices will de-serialise the message and communicate the actions to be taken to the corresponding PLCs via the interface with the LCS.

The DevMgr is not blocked when receiving concurrent Setup commands (messages). It executes them in separate worker threads that are spawned per each new Setup command. The threads will be running until the commands have been executed successfully, an error occured, the timeout has elapsed or a *Stop* command is received, see figure below.

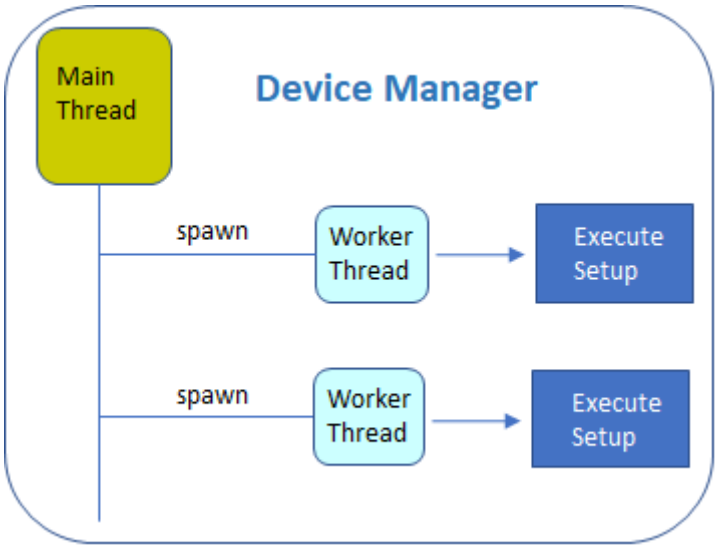


Fig. 2.6: Device Manager Setup worker threads.

Note: A *Stop* command finalizes all ongoing worker threads that are being handled by the Device Manager.

Warning: Conflicting requests across different *Setup* commands running in parallel are not handled by the Device Manager. They are pushed down to the PLC. The PLC is resolving them depending on the actual status. This means that, if the user sends two consecutive commands with conflicting requests, most likely the second one will get an error from the PLC. The exact behaviour will depend on the specific Device Controller implementation.

The interface definition of the *Setup* command can be found in module *fcfif*. The payload is based on an array of unions. The union may contain any device supported by the Device Manager.

```
<union name="DeviceUnion">
    <discriminator type="nonBasic" nonBasicTypeName="DeviceType" />
    <case>
        <caseDiscriminator value ="SHUTTER"/>
            <member name="shutter" type="nonBasic" nonBasicTypeName=
↪"ShutterDevice" />
    </case>
    <case>
        <caseDiscriminator value ="LAMP"/>
            <member name="lamp" type="nonBasic" nonBasicTypeName="LampDevice" />
    </case>
    <case>
    <caseDiscriminator value ="MOTOR"/>
            <member name="motor" type="nonBasic" nonBasicTypeName="MotorDevice" /
↪>
    </case>
    <case>
    <caseDiscriminator value ="DROT"/>
            <member name="drot" type="nonBasic" nonBasicTypeName="DrotDevice" />
    </case>
    <case>
    <caseDiscriminator value ="ADC"/>
            <member name="adc" type="nonBasic" nonBasicTypeName="AdcDevice" />
    </case>
    <case>
    <caseDiscriminator value ="PIEZO"/>
            <member name="piezo" type="nonBasic" nonBasicTypeName="PiezoDevice" /
↪>
    </case>
    <case>
    <caseDiscriminator value ="ACTUATOR"/>
            <member name="actuator" type="nonBasic" nonBasicTypeName=
↪"ActuatorDevice" />
    </case>
    <case>
    <caseDiscriminator value ="CUSTOM"/>
            <member name="custom" type="nonBasic" nonBasicTypeName="CustomDevice
↪" />
    </case>
```

(continues on next page)

(continued from previous page)

```
</union>
```

Warning: The array does not have a fixed size but it has a limit of 100 elements. A limit is needed by the CII XML ICD.

```
<method name="Setup" returnType="string" throws="ExceptionErr">
    <argument name="payload" type="nonBasic" nonBasicTypeName="SetupElem"␣
↪arrayDimensions="(100)"/>
</method>
```

Each device structure may contain parameters and one action per device that can be serialized. An example of the device ICD is shown below.

```
<struct name="ShutterDevice">
    <member name="id" type="string" />
    <member name="action" type="nonBasic" nonBasicTypeName="ActionShutter" />
</struct>

<struct name="LampDevice">
    <member name="id" type="string" />
    <member name="intensity" type="double" />
    <member name="time" type="uint32_t" />
    <member name="action" type="nonBasic" nonBasicTypeName="ActionLamp" />
</struct>

<struct name="BaseMotorDevice">
   <member name="id" type="string" />
    <member name="name" type="string" />
    <member name="pos" type="double" />
    <member name="enc" type="int64_t" />
    <member name="speed" type="double" />
    <member name="unit" type="nonBasic" nonBasicTypeName="MotorPosUnit" />
</struct>

<struct name="MotorDevice" baseType="BaseMotor">
    <member name="action" type="nonBasic" nonBasicTypeName="ActionMotor" />
</struct>

<struct name="DrotDevice" baseType="BaseMotor">
    <member name="mode" type="nonBasic" nonBasicTypeName="ModeDrot" />
    <member name="action" type="nonBasic" nonBasicTypeName="ActionDrot" />
</struct>

<struct name="AdcDevice" baseType="BaseMotor">
    <member name="axis" type="nonBasic" nonBasicTypeName="AxesAdc" />
    <member name="mode" type="nonBasic" nonBasicTypeName="ModeAdc" />
```

(continues on next page)

(continued from previous page)

```
    <member name="action" type="nonBasic" nonBasicTypeName="ActionAdc" />
</struct>

<struct name="PiezoDevice">
    <member name="id" type="string" />
    <member name="bit1" type="uint32_t" />
    <member name="bit2" type="uint32_t" />
    <member name="bit3" type="uint32_t" />
    <member name="pos1" type="double" />
    <member name="pos2" type="double" />
    <member name="pos3" type="double" />
    <member name="action" type="nonBasic" nonBasicTypeName="ActionPiezo" />
</struct>

<struct name="ActuatorDevice">
    <member name="id" type="string" />
    <member name="action" type="nonBasic" nonBasicTypeName="ActionActuator" />
</struct>

<struct name="CustomDevice">
    <member name="parameters" type="string" />
</struct>
```

Note: The CustomDevice is to be used for implementing custom devices where the payload data can be serialized in JSON. The serialization shall be done by the client applications using the parameters in the `CustomDevice` structure to carry the information encoded in JSON. The above enables extendability without the need to provide specific CII XML ICDs which is a significant simplification for instruments.

DevStatus Command

The DevStatus command provides information about each device controlled by the *Device Manager*. An example of the output generated by the DevStatus command is shown below.

```
$ fcfClient zpb.rr://127.0.0.1:12083 DevStatus ""
shutter1.simulated = true
shutter1.lcs.state = Operational
shutter1.lcs.substate = Close
lamp1.simulated = true
lamp1.lcs.state = Operational
lamp1.lcs.substate = Off
lamp1.lcs.intensity = 0.000000
motor1.simulated = true
motor1.lcs.state = Operational
```

(continues on next page)

(continued from previous page)

```
motor1.lcs.substate = Standstill
motor1.lcs.pos_target = 30.000000
motor1.lcs.pos_actual = 30.002197
motor1.lcs.vel_actual = 0.000000
motor1.lcs.axis_enable = true
motor1.pos_actual_name = ON
motor1.pos_enc = 341

OK
```

The user could request the status of a specific device or a subset of the devices, see below.

```
$ fcfClient zpb.rr://127.0.0.1:12083 DevStatus "motor1"
motor1.simulated = true
motor1.lcs.state = Operational
motor1.lcs.substate = Standstill
motor1.lcs.pos_target = 30.000000
motor1.lcs.pos_actual = 30.002197
motor1.lcs.vel_actual = 0.000000
motor1.lcs.axis_enable = true
motor1.pos_actual_name = ON
motor1.pos_enc = 341

OK

$ fcfClient zpb.rr://127.0.0.1:12083 DevStatus "lamp1, motor1"
lamp1.simulated = true
lamp1.lcs.state = Operational
lamp1.lcs.substate = Off
lamp1.lcs.intensity = 0.000000
motor1.simulated = true
motor1.lcs.state = Operational
motor1.lcs.substate = Standstill
motor1.lcs.pos_target = 30.000000
motor1.lcs.pos_actual = 30.002197
motor1.lcs.vel_actual = 0.000000
motor1.lcs.axis_enable = true
motor1.pos_actual_name = ON
motor1.pos_enc = 341

OK
```

Note: The list of devices is comma-separated.

Ignore Command

This command tells the *Device Manager* to completely ignore a device. It can be used when there are hardware failures or when the hardware is not yet available. The following example shows a sequence that ignores device *lamp1*, gets the status of the devices and then stops ignoring the device.

Note: When a device is ignored, no other information is provided for this device when processing the status command.

```
$ fcfClient zpb.rr://127.0.0.1:12083 Ignore "lamp1"
$ fcfClient zpb.rr://127.0.0.1:12083 Status "lamp1"
lamp1.ignored = true

OK
$ fcfClient zpb.rr://127.0.0.1:12083 StopIgn "lamp1"
$ fcfClient zpb.rr://127.0.0.1:12083 Status "lamp1"
lamp1.simulated = true
lamp1.lcs.state = Operational
lamp1.lcs.substate = Off
lamp1.lcs.intensity = 0.000000

OK
```

Simulate Command

This command tells the *Device Manager* to use the simulation address of the device. If the *Device Manager* is already connected, it will disconnect from the normal address and connect to the simulator. When the simulation is stopped, the server reverts the action and the device is back to normal mode.

The purpose of the simulation is to be able to validate the response of the *Device Manager* under different error conditions. It also allows to test the high-level SW when the HW is not yet available.

```
$ fcfClient zpb.rr://127.0.0.1:12083 Simulate "lamp1"
$ fcfClient zpb.rr://127.0.0.1:12083 Status "lamp1"
lamp1.simulated = true
lamp1.lcs.state = Operational
lamp1.lcs.substate = Off
lamp1.lcs.intensity = 0.000000

OK
$ fcfClient zpb.rr://127.0.0.1:12083 StopSim "lamp1"
$ fcfClient zpb.rr://127.0.0.1:12083 Status "lamp1"
lamp1.lcs.state = Operational
lamp1.lcs.substate = Off
```

(continues on next page)

(continued from previous page)

```
lamp1.lcs.intensity = 0.000000

OK
```

2.6 Troubleshooting

2.6.1 Logging

The *Device Manager* has implemented six logging levels that provide additional information for troubleshooting to the developer.

Name	Verbosity	Description
ERROR	very low	Provide logging only in case of errors.
INFO	low	Provide information for the most important actions (default).
DEBUG	medium	Provide additional information for the developer.
DEBUG2	high	Includes more details such as Node IDs of OPC-UA attributes
DEBUG3	high	Includes the logging of each subscription event.
TRACE	very high	Includes all the function tracing.

To activate a new logging, the command SetLog shall be used. See the example below.

```
$ fcfClient zpb.rr://127.0.0.1:12083 SetLog "TRACE"
```

2.6.2 OPC-UA Client

Sometimes it is better to check the status of the PLC using an OPC-UA client. One of the best tools available is the UaExpert from Unified Automation. This tool enables the control and monitoring of all device variables independently of the *Device Manager*. The user can trigger the execution of RPCs and monitor the device state changes. The UaExpert is an essential tool for troubleshooting.

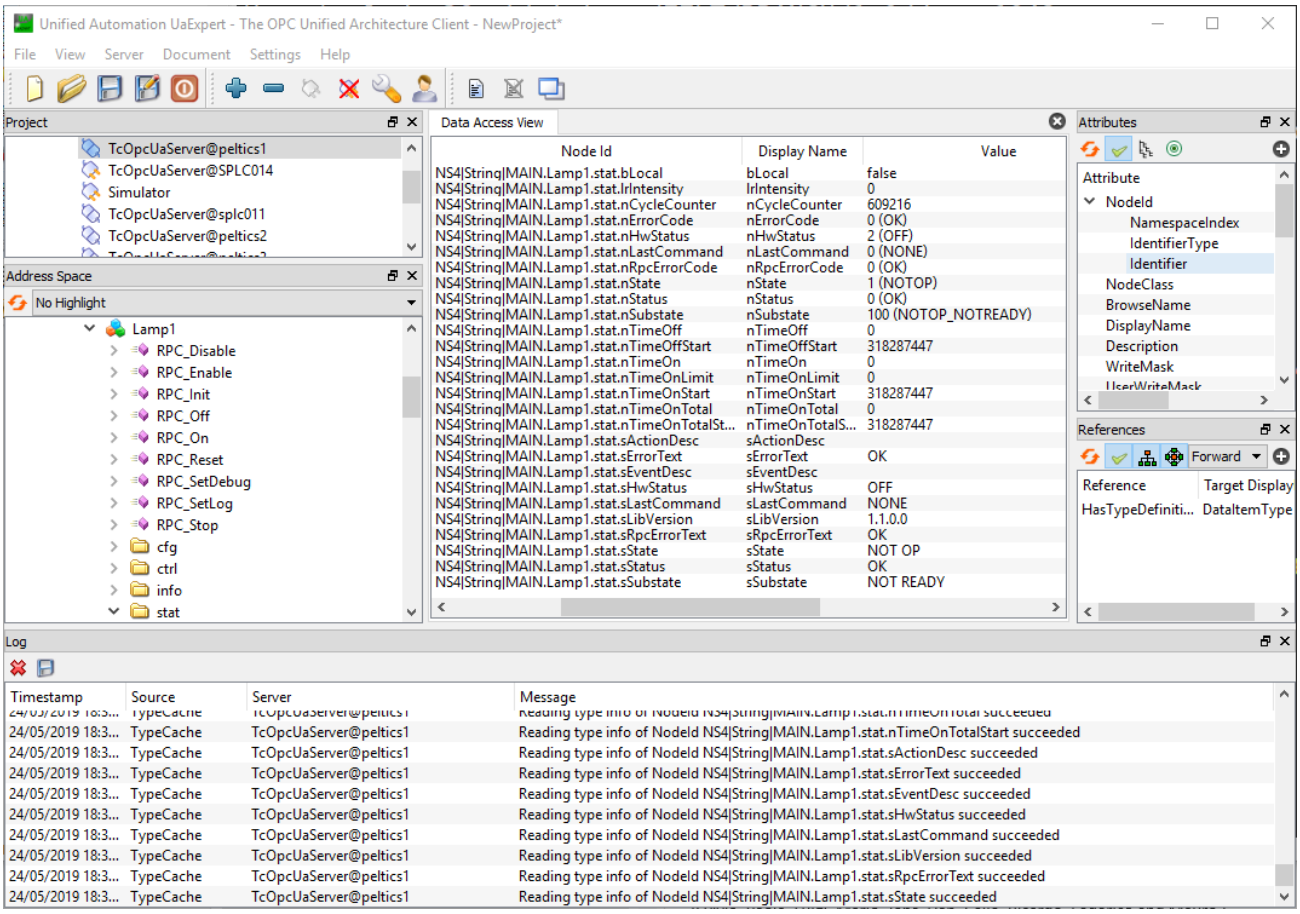


Fig. 2.7: UaExpert OPC-UA client.

3 PLC Libraries

3.1 Common Attributes and Functionality

Note: ESO PLC libraries have to be installed in the TwinCAT development environment before they can be used in projects.

3.1.1 Library content

For each function, e.g. lamp, shutter, motor, etc, the corresponding PLC library delivers:

- A function block (FB) for function control, e.g. FB_LAMP. Note that some libraries deliver more than one FB, e.g. motor.library.
- A function block for the HW simulator, e.g. FB_SIM_LAMP
- A template GUI, e.g. GUI_TEMPLATE_LAMP

3.1.2 Input parameters

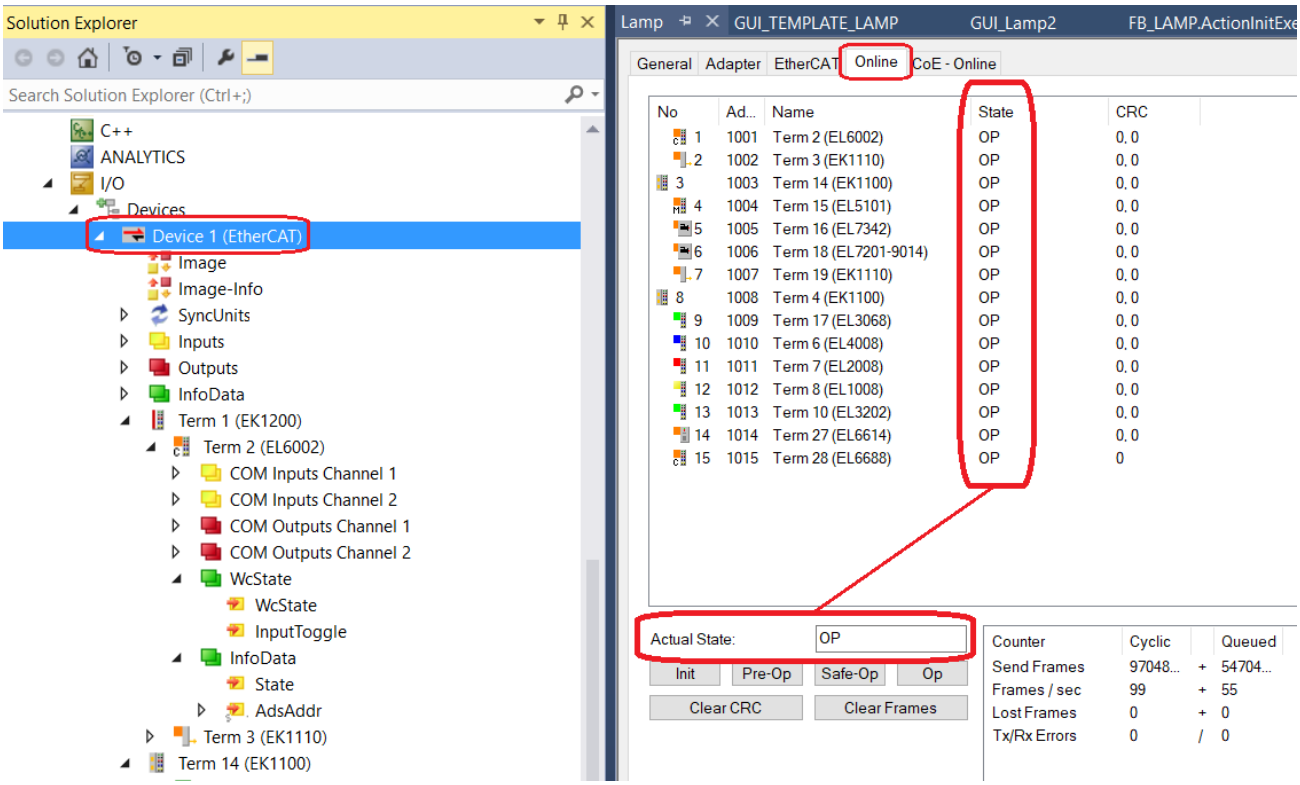
In most cases, apart from the Motor library, the configuration parameters can be given in the execution part, i.e. the FB call. This way some configuration parameters that are known that will not change, e.g. active low, timeouts, etc, can be 'hard-coded'. Input parameters have the prefix *in_*. There is a mandatory input parameter called 'in_sName' that sets the name of the device. On PLC reboot these parameters will be set in the device configuration. However, the configuration defined in the Device Manager will overwrite the device configuration on INIT. This way, by giving input parameters the number of configuration parameters can be reduced or completely avoided but still with the flexibility of correcting any configuration parameter by the Device Manager without modifying the PLC code.

Example:

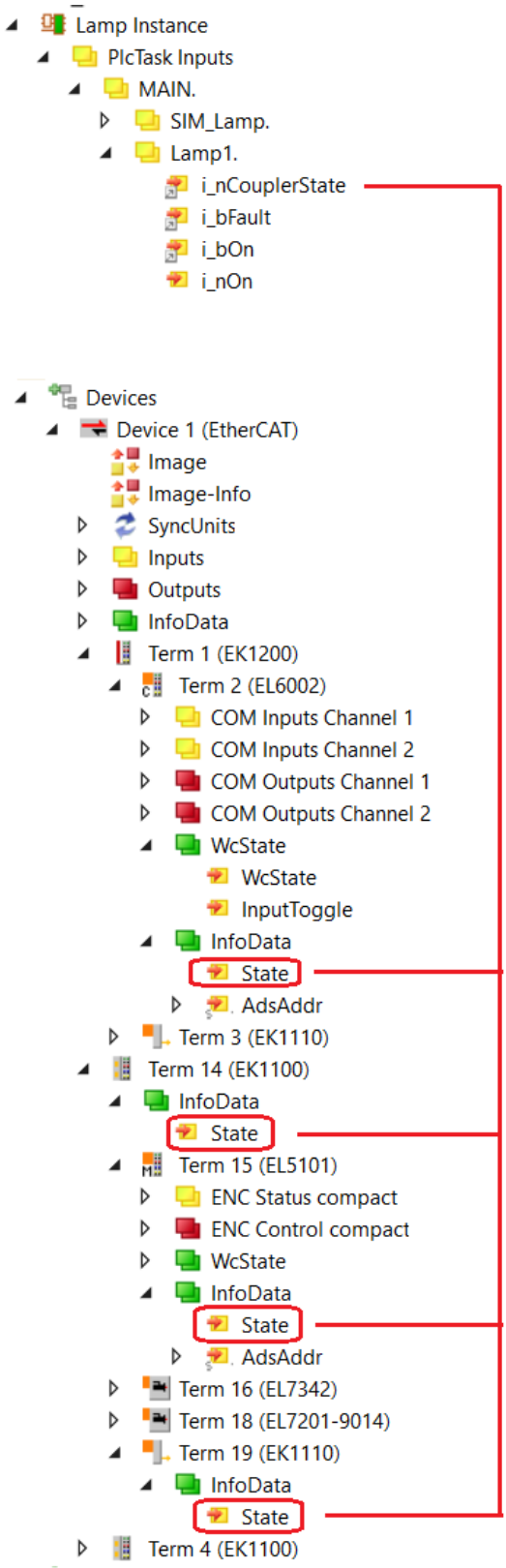
```
Lamp1(in_sName:='Lamp1', in_bActiveLowFault:=TRUE);
```

3.1.3 EtherCAT Operational State and FB Input variable i_nCouplerState

In order for the EtherCAT system to operate, it has to be in OPERATIONAL state with its corresponding value equal to 8. Any value other than 8 indicates that the system is not OPERATIONAL. The EtherCAT system is OPERATIONAL only if all I/O terminals in the system are OPERATIONAL. The figure below shows an OPERATIONAL EtherCAT system and the individual state of each I/O terminal in the system.



Each FB that controls HW has got an input variable called *i_nCouplerState*. This variable is used to monitor the state of the EtherCAT system. The variable has to be linked (mapped) to the *State* variable, found in the *InfoData* structure of the I/O terminal, that indicates its operational state. Normally, the variable is linked to the *State* of the EK1100 coupler that holds the I/O termials used by the FB. That's why the variable is called *i_nCouplerState*. However, the *State* variable of any other I/O terminal, e.g. EL2008, could also be used for that purpose. The figure below shows some possible mapping options for *i_nCouplerState*.


Lamp Instance
PlcTask Inputs
MAIN.
SIM_Lamp.
Lamp1.
i_nCouplerState
i_bFault
i_bOn
i_nOn
Devices
Device 1 (EtherCAT)
Image
Image-Info
SyncUnits
Inputs
Outputs
InfoData
Term 1 (EK1200)
Term 2 (EL6002)
COM Inputs Channel 1
COM Inputs Channel 2
COM Outputs Channel 1
COM Outputs Channel 2
WcState
WcState
InputToggle
InfoData
State
AdsAddr
Term 3 (EK1110)
Term 14 (EK1100)
InfoData
State
Term 15 (EL5101)
ENC Status compact
ENC Control compact
WcState
InfoData
State
AdsAddr
Term 16 (EL7342)
Term 18 (EL7201-9014)
Term 19 (EK1110)
InfoData
State
Term 4 (EK1100)

3.1.4 Interface with Device Manager

Device managers control the PLC device exclusively via RPC methods. There are a number of mandatory RPC methods that are common to all PLC devices. They are related to device state changes and logging. Each library, in addition to the function specific RPC methods, delivers the following RPCs:

- RPC_Init()
- RPC_Enable()
- RPC_Disable()
- RPC_SetDebug()
- RPC_SetLog()
- RPC_Stop()
- RPC_Reset()

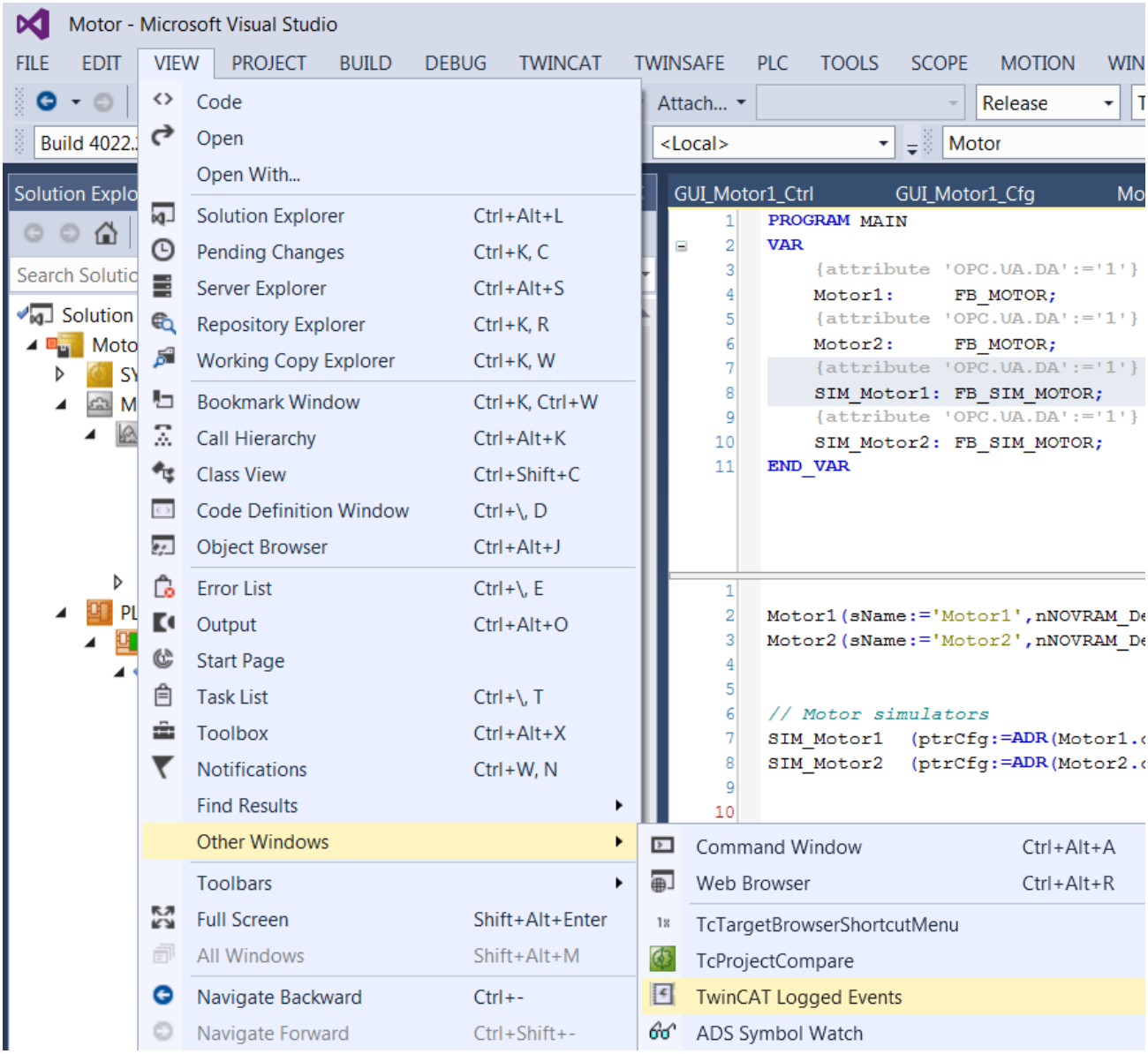
In order to have RPC methods visible in the OPC UA address space, each declaration of an FB has to be coupled with a pragma statement {attribute 'OPC.UA.DA':='1'}.

Example:

```
{attribute 'OPC.UA.DA':='1'}
Lamp1: FB_LAMP;

{attribute 'OPC.UA.DA':='1'}
Lamp2: FB_LAMP;
```

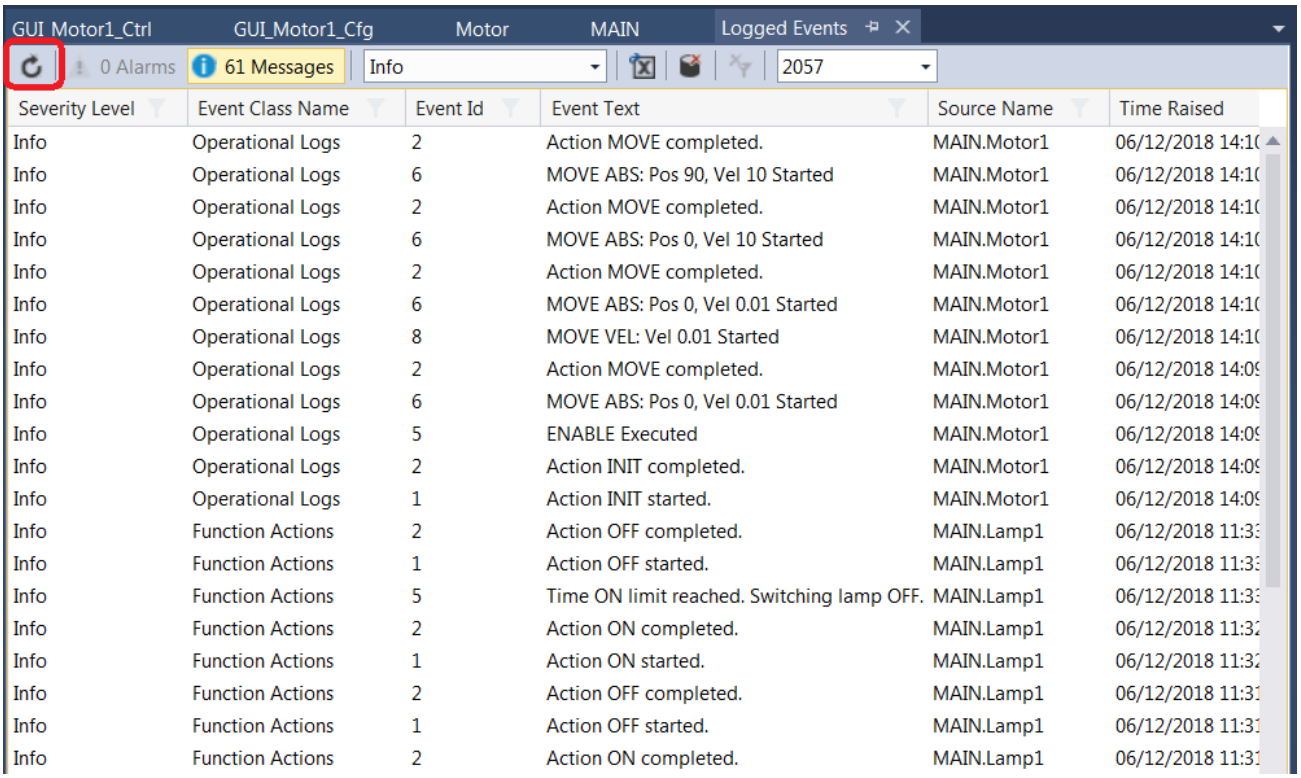
3.1.5 Operational Logs at PLC Level

Device operations are by default logged on the PLC. The screenshot below shows how to open TwinCAT Logged Events Window.



RPC methods RPC_SetDebug() and RPC_SetLog() are used to control the amount of logging information. If provided, debugging logs should be activated only for troubleshooting purpose since they could generate large amount of data.

In order to see the logs, the window has to be refreshed as shown below:



Severity Level	Event Class Name	Event Id	Event Text	Source Name	Time Raised
Info	Operational Logs	2	Action MOVE completed.	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	6	MOVE ABS: Pos 90, Vel 10 Started	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	2	Action MOVE completed.	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	6	MOVE ABS: Pos 0, Vel 10 Started	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	2	Action MOVE completed.	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	6	MOVE ABS: Pos 0, Vel 0.01 Started	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	8	MOVE VEL: Vel 0.01 Started	MAIN.Motor1	06/12/2018 14:10
Info	Operational Logs	2	Action MOVE completed.	MAIN.Motor1	06/12/2018 14:09
Info	Operational Logs	6	MOVE ABS: Pos 0, Vel 0.01 Started	MAIN.Motor1	06/12/2018 14:09
Info	Operational Logs	5	ENABLE Executed	MAIN.Motor1	06/12/2018 14:09
Info	Operational Logs	2	Action INIT completed.	MAIN.Motor1	06/12/2018 14:09
Info	Operational Logs	1	Action INIT started.	MAIN.Motor1	06/12/2018 14:09
Info	Function Actions	2	Action OFF completed.	MAIN.Lamp1	06/12/2018 11:33
Info	Function Actions	1	Action OFF started.	MAIN.Lamp1	06/12/2018 11:33
Info	Function Actions	5	Time ON limit reached. Switching lamp OFF.	MAIN.Lamp1	06/12/2018 11:33
Info	Function Actions	2	Action ON completed.	MAIN.Lamp1	06/12/2018 11:32
Info	Function Actions	1	Action ON started.	MAIN.Lamp1	06/12/2018 11:32
Info	Function Actions	2	Action OFF completed.	MAIN.Lamp1	06/12/2018 11:31
Info	Function Actions	1	Action OFF started.	MAIN.Lamp1	06/12/2018 11:31
Info	Function Actions	2	Action ON completed.	MAIN.Lamp1	06/12/2018 11:31

3.1.6 PLC simulators

In most cases the library includes a HW simulator that is linked to the device FB and gives quite realistic response. This way it is possible at a very early stage of the project to test Device Managers with a PLC. The device manager is not aware of the simulation at the PLC level and 'believes' that it communicates with a real HW.

Note: It is important to note that the PLC application, i.e. program MAIN, has to be modified in order to use simulators. In addition to the inclusion of a device simulator instance in MAIN, the device I/O variables have to be linked to the corresponding simulator I/O signals rather than to the real I/O.

HW simulators capture device INIT event and adjust their internal configuration ensuring that the device will work properly after the initialisation. For example, the simulator response time will be shorter than the device timeout.

Simulators provide RPC calls that are used to modify their response in order to test failure conditions of the device driver. For example, the Lamp RPC_SetFault() method can be used to set the fault signal of the lamp.

3.1.7 C++ Modules

For some PLC devices, additional software developed in C++ might be needed. TwinCAT enables the implementation of modules in C++ that run inside the TwinCAT real-time kernel. These modules communicate with the PLC via normal I/O mapping (Communication between C++ modules and PLC[6]).

The FCF provides some C++ modules for the computation of the field rotation that are used by tracking devices such as Derotators and ADCs.

The installation of the C++ module must be done on every computer used to build and download PLC projects that include tracking devices. These modules have been already compiled and published by ESO and archived here[7]. Please note that if the utility *MakeTcProject* (see *Creating PLC Applications with MakeTcProject Utility*) is used to create a PLC project, the modules will be installed automatically.

Note: Visualisation and recompilation of the C++ module sources is only possible using commercial versions of Visual Studio (VS), e.g. VS Professional. However, C++ modules can be imported into the standard TwinCAT environment without the need to recompile them.

The procedure describing how to import modules into TwinCAT is available on Beckhoff website: (Importing C++ modules[8]).

3.1.8 Tracking

Tracking controllers are motorized devices which update continuously the position of one or more motor axes as a function of the telescope coordinates, UTC time or variations of the temperature. Tracking controllers share some common characteristics that are described below.

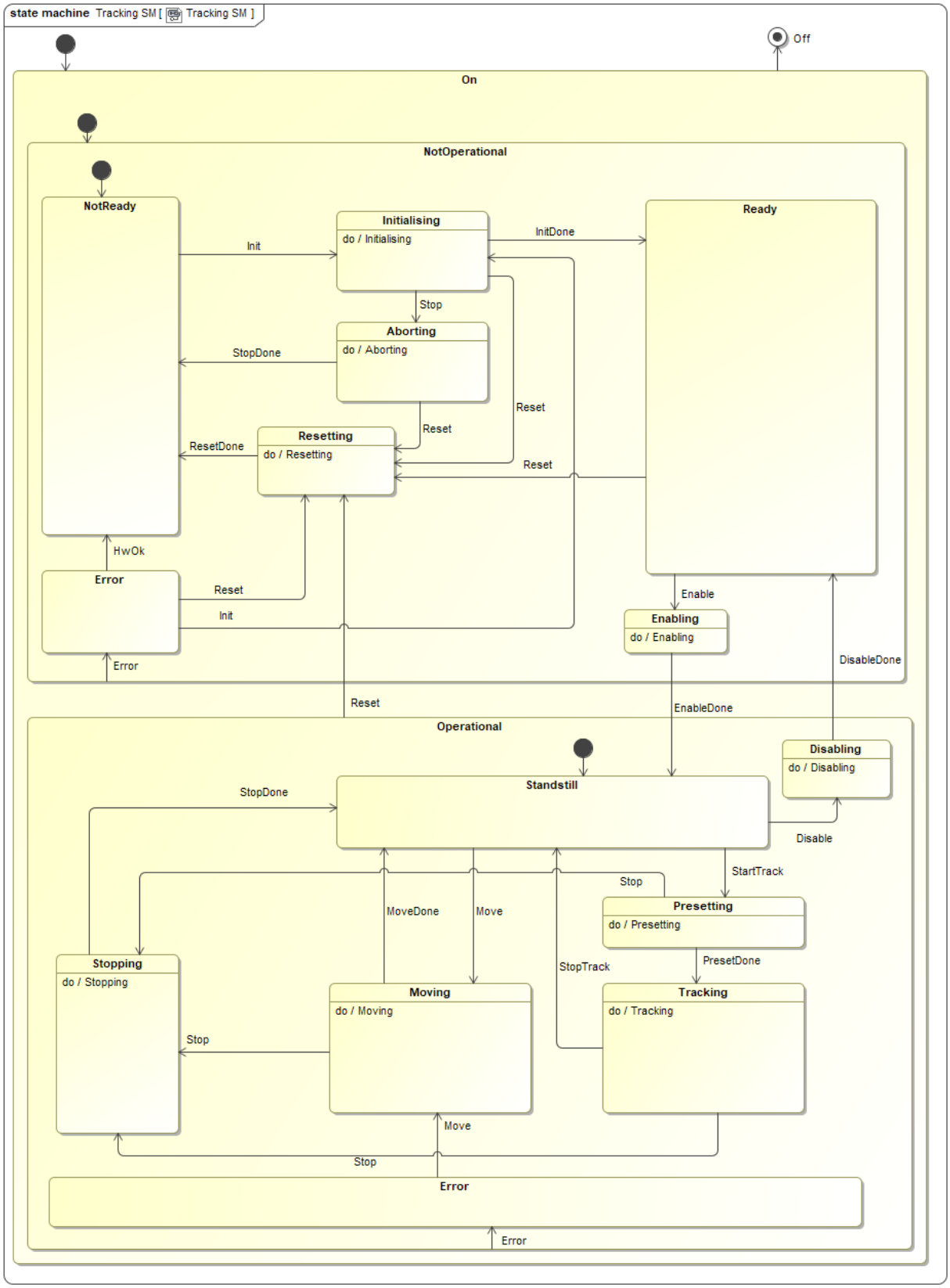
- They support at least two modes of operations: one where motor axes are stationary and another one for tracking. In both cases the motor axes are moving in position mode.
- They may require an additional TwinCAT runtime license in case of using a C++ module. The required TwinCAT license is the C++ runtime.
- They may require a connection to the ELT time synchronization system to maximize tracking accuracy of the motor axes. A dedicated terminal (EL6688) is needed to connect to the ELT time synchronization system.
- The tracking position control loop is handled at lower level (PLC) by dedicated function blocks. The period of the position control loop is configurable and does not depend on the PLC cycle.

[6] https://infosys.beckhoff.com/content/1033/tc3_c/18014401000519947.html?id=5760242511836135973
[7] http://svnhq9.hq.eso.org/p9/trunk/EELT/ICS/PLC/Modules
[8] https://infosys.beckhoff.com/content/1033/tc3_c/63050394893968011.html?id=5466309176056117169

3.1.9 State Machine of Tracking Devices

The generic state machine of tracking devices is shown below. The main operational states are *Standstill*, *Moving* and *Tracking*.

Note: The following State Machine is common to all tracking devices, e.g. Derotators, ADCs, etc.


state machine Tracking SM [Tracking SM]
Off
On
NotOperational
NotReady
Initialising
do / Initialising
Init
InitDone
Ready
Stop
Aborting
do / Aborting
StopDone
Reset
Reset
Resetting
do / Resetting
ResetDone
Reset
HwOk
Error
Reset
Init
Enable
Enabling
do / Enabling
DisableDone
Error
Reset
EnableDone
Operational
Disabling
do / Disabling
Standstill
StopDone
Disable
StartTrack
Stop
MoveDone
Move
Presetting
do / Presetting
Stopping
do / Stopping
StopTrack
PresetDone
Moving
do / Moving
Tracking
do / Tracking
Stop
Move
Stop
Error
Error

Note: Important note: The cycle time of the task (e.g. *PlcTask*) that executes an instance of a tracking function, i.e. *FB_MA_ADC*, *FB_MA_DROT*, etc, must be the same or longer than the cycle time of the *NC-Task 1 SAF* task. In other words, the instance of these FBs must not be executed faster than the *NC-Task 1 SAF* task. Otherwise, there could be some synchronization problems when exiting tracking mode.

Note that the default cycle time of the *NC-Task 1 SAF* task is 2 ms. Therefore, if the *PlcTask* cycle time is for example set to 1 ms, the *NC-Task 1 SAF* task cycle time has to be set to the same value.

The Motor Library provides two C++ modules that work together to compute the tracking data:

- TrkParams
- TrkModule

These two modules can be merged but in order to optimize the CPU it is better to have them separated. The *trkParams* computes the parameters used by the *slalib* library and it does not require to be executed at a fast rate. The *trkModule* is the one computing the tracking data and it should be configured to run in a task running faster than the *trkParams*. It is suggested to use isolated CPU cores in order to maximize performance, as shown in the figure below where one CPU core is dedicated to the more demanding tasks.

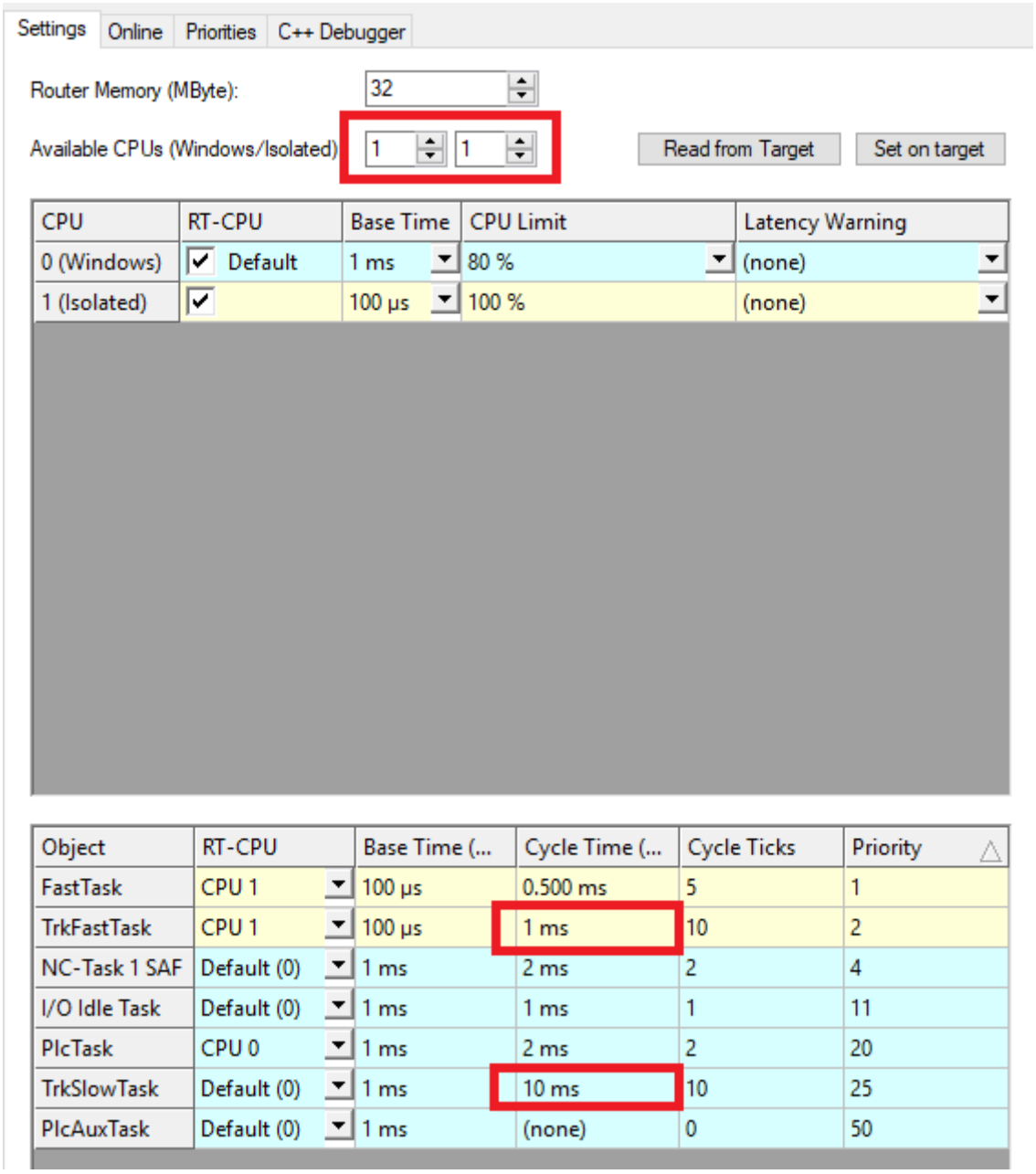


Fig. 3.1: Suggestion for PLC CPU assignments.

3.1.10 Automatic TwinCAT Project Creation

A Windows utility called *MakeTcProject.exe* is provided for automatic creation of TwinCAT projects with a selectable number of all available controllers. The utility generates a fully operational project that includes the selected number of devices/controllers and their simulators, i.e. every device is simulated at PLC level.

This can be very useful at very early stages of projects when HW is still not available, since it makes it possible to test the complete control system as if the HW were present.

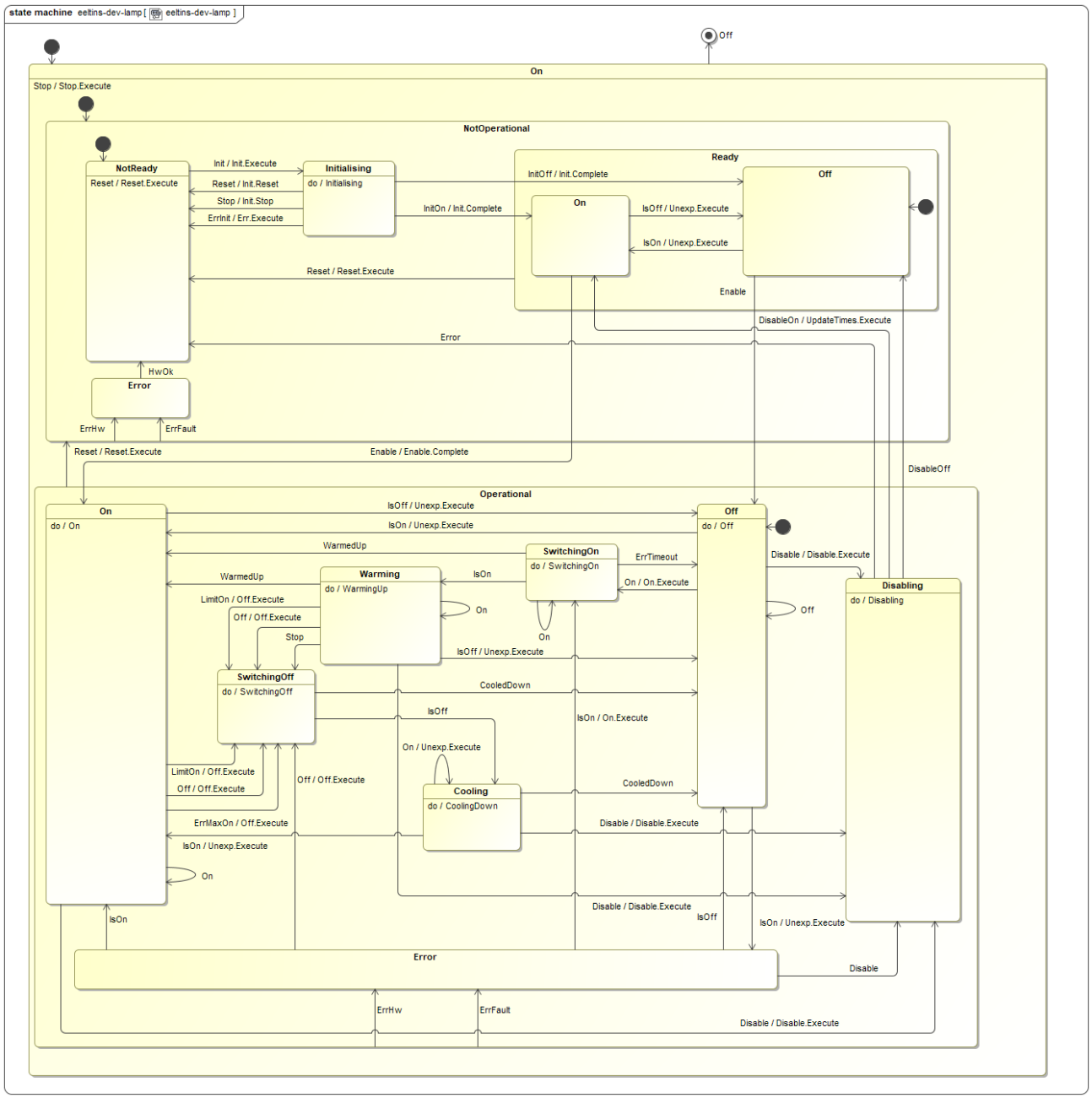
The utility is explained in detail in *Creating PLC Applications with MakeTcProject Utility*.

3.2 Lamp Library (lamp.library)

FB_LAMP is the TwinCAT PLC Function Block for the low level control of the standard lamp device with or without intensity control.

3.2.1 State Machine

The state machine of the lamp controller is shown below. The main operational states are *On*, *Off* *Warming* and *Cooling*.


state machine eeltins-dev-lamp [eeltins-dev-lamp]
Off
On
Stop / Stop.Execute
NotOperational
NotReady
Reset / Reset.Execute
Init / Init.Execute
Reset / Init.Reset
Stop / Init.Stop
ErrInit / Err.Execute
Initialising
do / Initialising
InitOff / Init.Complete
InitOn / Init.Complete
Ready
On
Off
IsOff / Unexp.Execute
IsOn / Unexp.Execute
Reset / Reset.Execute
Enable
DisableOn / UpdateTimes.Execute
Error
HwOk
Error
ErrHw
ErrFault
Reset / Reset.Execute
Enable / Enable.Complete
DisableOff
Operational
On
do / On
IsOff / Unexp.Execute
IsOn / Unexp.Execute
WarmedUp
WarmedUp
LimitOn / Off.Execute
Off / Off.Execute
Stop
SwitchingOn
do / SwitchingOn
ErrTimeout
On / On.Execute
Warming
do / WarmingUp
IsOn
On
On
IsOff / Unexp.Execute
SwitchingOff
do / SwitchingOff
CooledDown
IsOff
On / Unexp.Execute
Cooling
do / CoolingDown
LimitOn / Off.Execute
Off / Off.Execute
Off / Off.Execute
ErrMaxOn / Off.Execute
IsOn / Unexp.Execute
On
IsOn
Off
do / Off
IsOn / On.Execute
CooledDown
Disable / Disable.Execute
Disable / Disable.Execute
IsOff
Disable / Disable.Execute
Off
Disabling
do / Disabling
IsOn / Unexp.Execute
Disable
Error
ErrHw
ErrFault
Disable / Disable.Execute

3.2.2 Input parameters

FUNCTION_BLOCK FB_LAMP_BASE

VAR_INPUT	**in_sName**	STRING	*Instance name*
VAR_INPUT	**in_bActiveLowFault**	BOOL	*If TRUE, Fault signal is Active Low*
VAR_INPUT	**in_bActiveLowOn**	BOOL	*If TRUE, On signal is Active Low*
VAR_INPUT	**in_bActiveLowSwitch**	BOOL	*If TRUE, Switch ctrl signal is Active Low*
VAR_INPUT	**in_bIgnoreFault**	BOOL	*If TRUE, Fault feedback signal is ignored*
VAR_INPUT	**in_bInitialState**	BOOL	*default lamp state is OFF*
VAR_INPUT	**in_bInvertAnalog**	BOOL	*If TRUE, analog feedback is active, if signal < n...*
VAR_INPUT	**in_lrInitialIntensity**	LREAL	*Initial intensity [%]. Default 0.0.*
VAR_INPUT	**in_nAnalogThreshold**	DINT	*Analog feedback signal threshold [bits]. If this si...*
VAR_INPUT	**in_nFullRange**	UDINT	*Full range of A/D converter for analog output fo...*
VAR_INPUT	**in_nSigStablePeriod**	UDINT	*signal is stable if it has been constant for so lon...*
VAR_INPUT	**in_nTimeout**	UDINT	*timeout for transitions [msec]*
VAR_INPUT	**in_nCooldown**	UDINT	*Cooldown time [sec]*
VAR_INPUT	**in_nMaxOn**	UDINT	*Maximum time for lamp to be ON [sec]. Zero m...*
VAR_INPUT	**in_nWarmup**	UDINT	*Warmup time [sec]*
VAR_INPUT	**in_bLogExtTime**	BOOL	*If TRUE, use external time in event logs. Default...*
VAR_INPUT	**in_bLog**	BOOL	*If TRUE, log events. Default TRUE.*

3.2.3 Signal Mapping

The figure below shows the TwinCAT view of the *FB_LAMP* I/O variables that are available for mapping to physical signals, i.e. ports of I/O terminals.

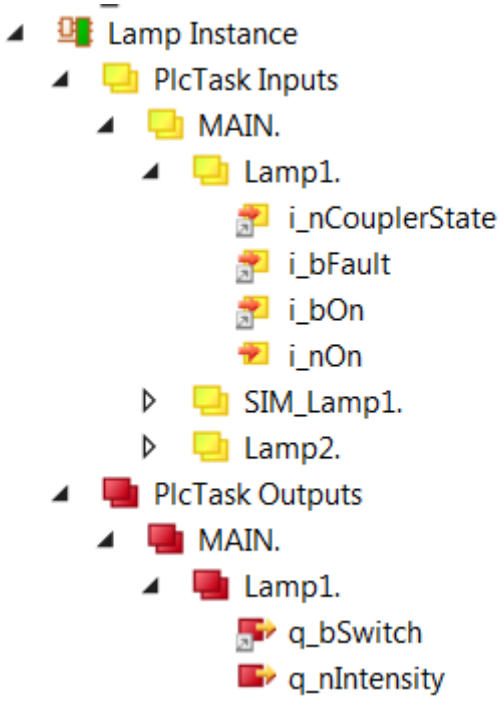


Fig. 3.2: Example of FB_LAMP input/output signals.

The table below describes each mapping variable.

Variable	Port Type	Optional Mapping	Description
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts a lamp signal.**
i_bFault	**Digital In**	**Yes**	**Does not have to be mapped if 'fault' signal is ignored, i.e. does not exist.**
i_bOn	**Digital In**	**No**	**Mapped to the Lamp status (ON/OFF) digital input signal.**
i_nOn	**Analog In**	**Yes**	**Analog feedback signal. Has to be mapped only if analog feedback is used.**
q_bSwitch	**Digital Out**	**No**	**Mapped to the Lamp control digital output signal.**
q_nIntensity	**Analog Out**	**Yes**	**Mapped to the Lamp intensity analog output signal (if exists).**

3.2.4 GUI Template

The Lamp Library provides a template GUI to control instances of *FB_LAMP*. Applications can easily deploy an instance of this GUI by setting the GUI reference to the particular instance of *FB_LAMP*, as shown below.

Fig. 3.3: Instantiation of GUI_TEMPLATE_LAMP for Lamp1

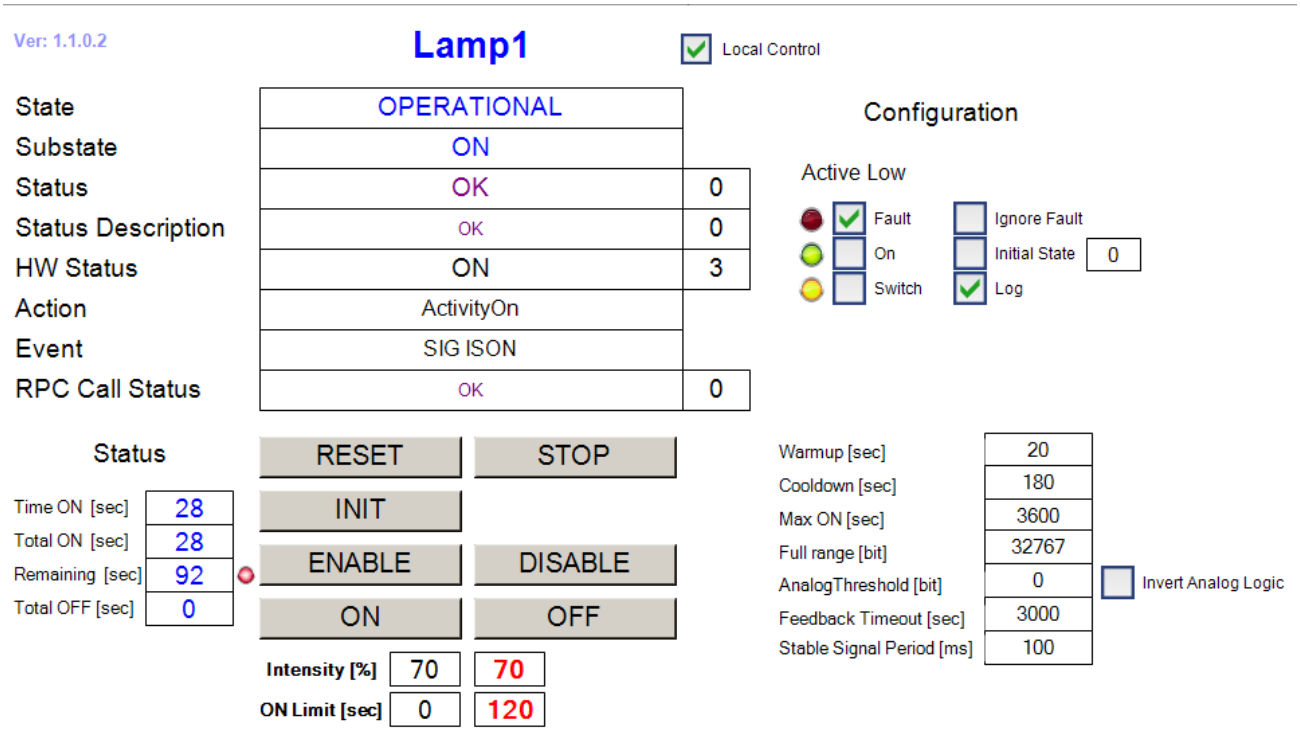


Fig. 3.4: FB_LAMP HMI for Local Control.

3.2.5 Lamp specific RPC Methods

- RPC_Off() Turn lamp OFF
- RPC_On() Turn lamp ON

3.2.6 Lamp Simulator

The function block *FB_SIM_LAMP* implements the lamp simulator on the PLC. The simulator has the address of the lamp instance as the only input parameter. The following code shows how the simulator is declared and executed:

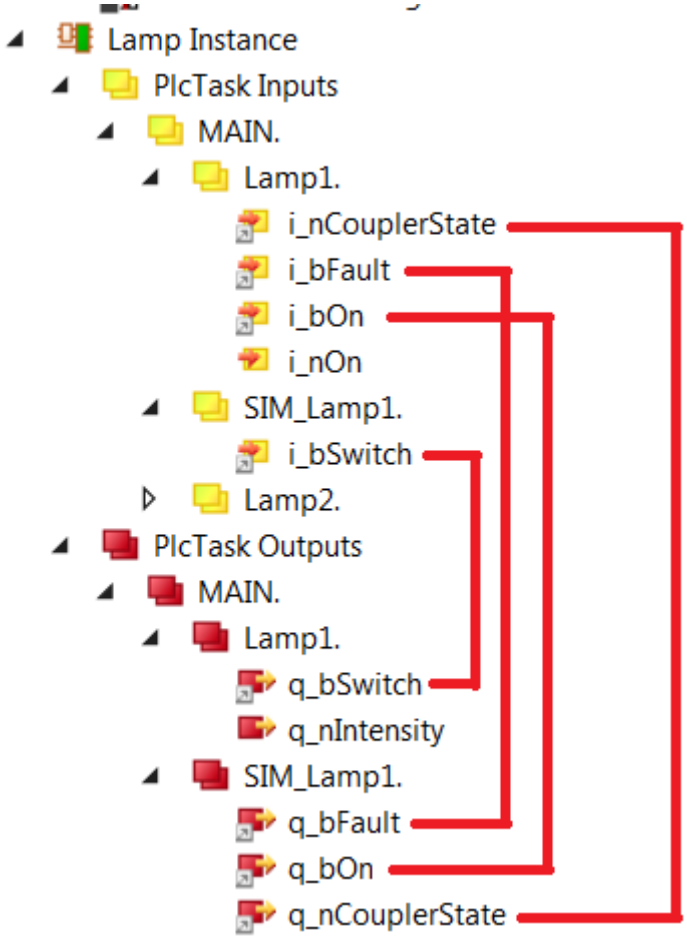
Declaration:

```
{attribute 'OPC.UA.DA':='1'}

Lamp1: FB_LAMP; // Simulated lamp

{attribute 'OPC.UA.DA':='1'}

SIM_Lamp: FB_SIM_LAMP; // Lamp simulator
```

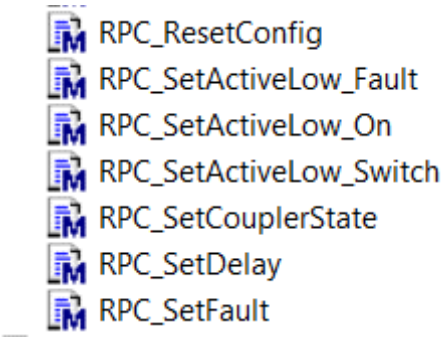
Execution:

```
Lamp1(in_sName:='Lamp1', in_bActiveLowFault:=TRUE);

SIM_Lamp(ptrDev:=ADR(Lamp1));
```

Simulator Mapping



Simulator RPC Methods

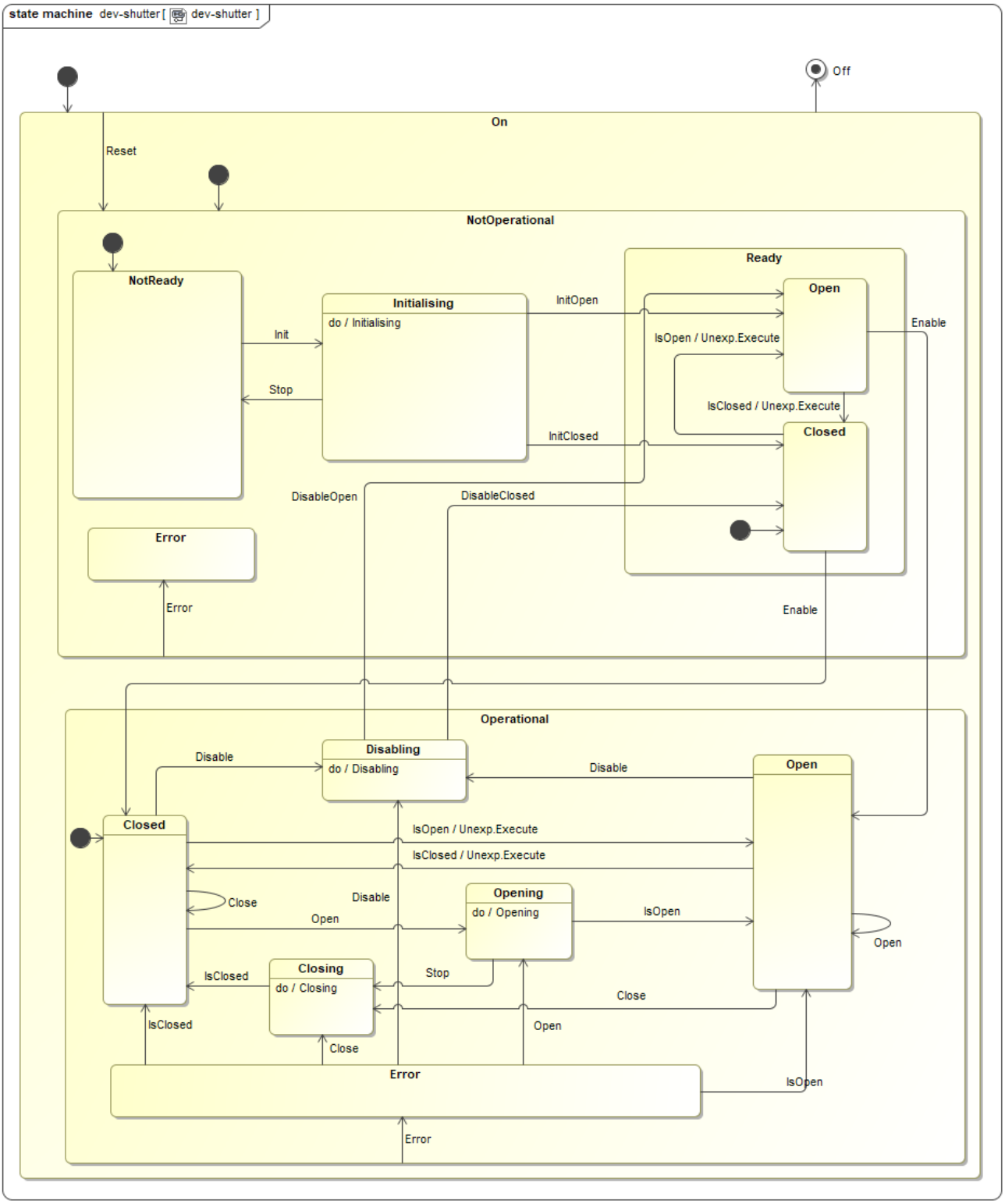


3.3 Shutter Library (shutter.library)

FB_SHUTTER is the TwinCAT PLC Function Block for the low level control of the standard shutter device.

3.3.1 State Machine

The state machine of the shutter controller is shown below. The main operational states are *Open*, *Closed*, *Opening* and *Closing*.


state machine dev-shutter [dev-shutter]
Off
On
Reset
NotOperational
NotReady
Initialising
do / Initialising
Init
Stop
InitOpen
InitClosed
Ready
Open
Closed
IsOpen / Unexp.Execute
IsClosed / Unexp.Execute
Enable
DisableOpen
DisableClosed
Error
Error
Enable
Operational
Disabling
do / Disabling
Disable
Disable
Open
Closed
IsOpen / Unexp.Execute
IsClosed / Unexp.Execute
Close
Disable
Open
Opening
do / Opening
IsOpen
Open
IsClosed
Closing
do / Closing
Stop
Close
IsClosed
Open
Close
Error
IsOpen
Error

3.3.2 Input parameters

FUNCTION_BLOCK FB_SHUTTER_BASE

VAR_INPUT	**in_sName**	STRING	*Instance name*
VAR_INPUT	**in_bActiveLowClosed**	BOOL	*If TRUE, the CLOSED signal is Active Low, default FALSE*
VAR_INPUT	**in_bActiveLowFault**	BOOL	*If TRUE, the FAULT signal is Active Low, default FALSE*
VAR_INPUT	**in_bActiveLowOpen**	BOOL	*If TRUE, the OPEN signal is Active Low, default FALSE*
VAR_INPUT	**in_bActiveLowSwitch**	BOOL	*If TRUE, the SWITCH signal is Active Low, default FALSE*
VAR_INPUT	**in_bIgnoreClosed**	BOOL	*If TRUE, the CLOSED signal is Ignored, default FALSE*
VAR_INPUT	**in_bIgnoreFault**	BOOL	*If TRUE, the FAULT signal is Ignored, default FALSE*
VAR_INPUT	**in_bIgnoreOpen**	BOOL	*If TRUE, the OPEN signal is Ignored, default FALSE*
VAR_INPUT	**in_bInitialState**	BOOL	*Default shutter position is FALSE/CLOSED*
VAR_INPUT	**in_nTimeout**	UDINT	*Timeout for OPEN/CLOSE transitions [msec], default 3000 ms*
VAR_INPUT	**in_bLogExtTime**	BOOL	*If TRUE, use external time in event logs. Default FALSE.*
VAR_INPUT	**in_bLog**	BOOL	*If TRUE, log events. Default TRUE.*

3.3.3 Signal Mapping

The figure below shows the TwinCAT view of the *FB_SHUTTER* I/O variables that are available for mapping to physical signals, i.e. ports of I/O terminals.

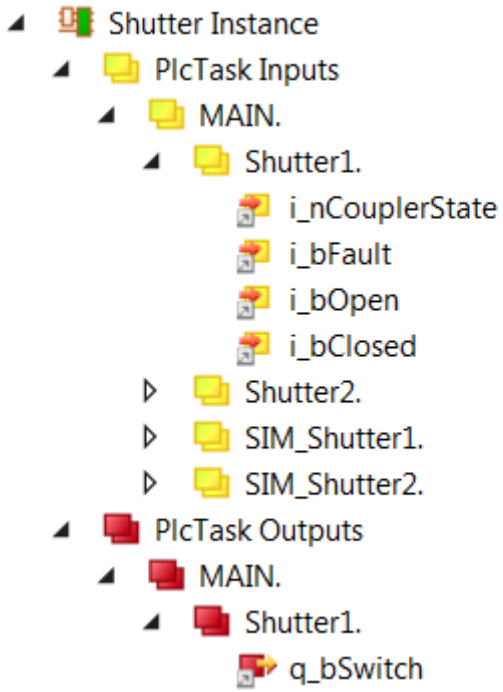


Figure 4: Example of FB_SHUTTER input/output signals.

The table below describes each mapping variable.

Variable	Port Type	Optional	Description
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts a shutter signal.**
i_bFault	**Digital In**	**Yes**	**Does not have to be mapped if 'fault' signal is ignored, i.e. does not exist.**
i_bOpen	**Digital IN**	**No**	**Mapped to the Shutter 'open' digital input signal.**
i_bClosed	**Digital IN**	**No**	**Mapped to the Shutter 'closed' digital input signal.**
q_bSwitch	**Digital Out**	**No**	**Mapped to the Shutter control digital output signal.**

3.3.4 GUI Template

The Shutter Library provides a template GUI to control instances of *FB_SHUTTER*. Applications can easily deploy an instance of this GUI by setting the GUI reference to the particular instance of *FB_SHUTTER*, as shown below.

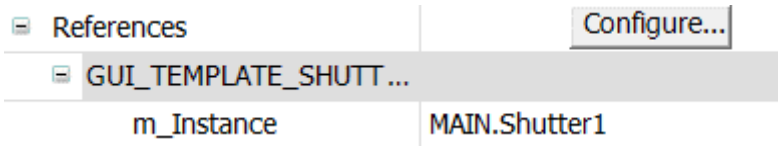


Figure 5: Instantiation of GUI_TEMPLATE_SHUTTER for Shutter1

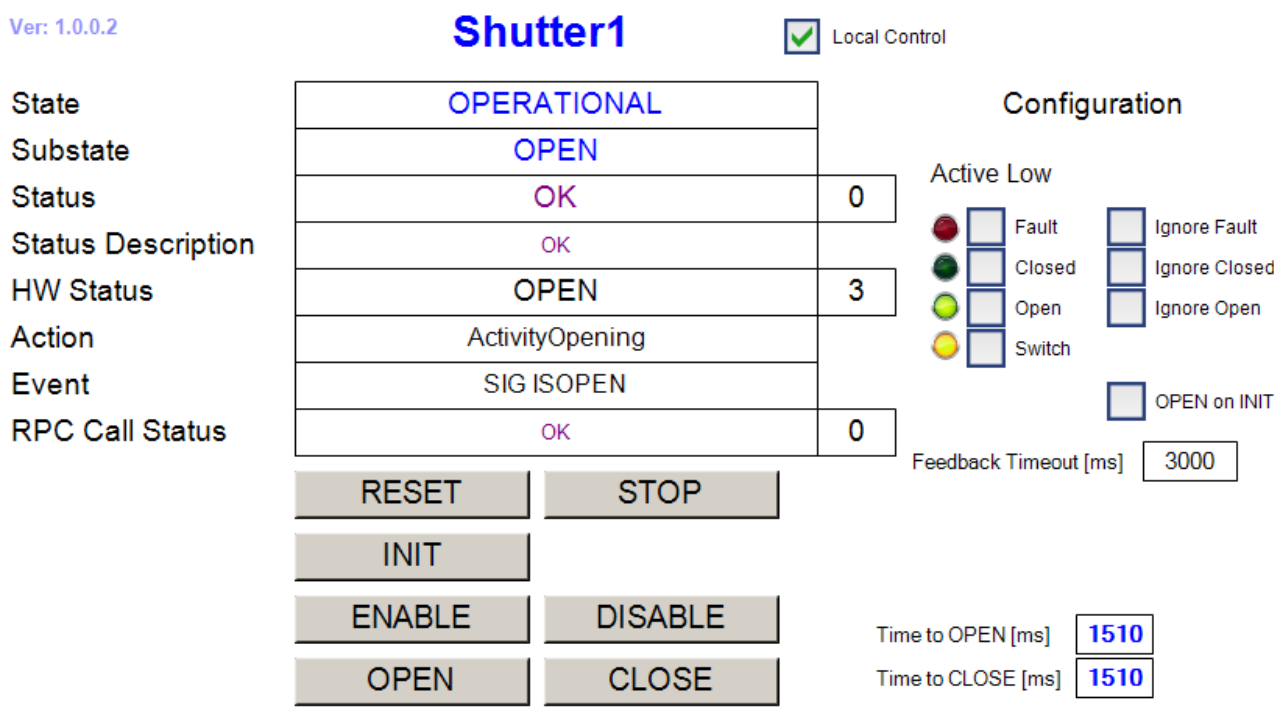


Figure 6: FB_SHUTTER HMI for Local Control.

3.3.5 Shutter specific RPC Methods

- RPC_Close() Close shutter
- RPC_Open() Open Shutter

3.3.6 Shutter Simulator

The function block *FB_SIM_SHUTTER* implements the shutter simulator on the PLC. The simulator has the address of the shutter instance as the only input parameter. The following code shows how the simulator is declared and executed:

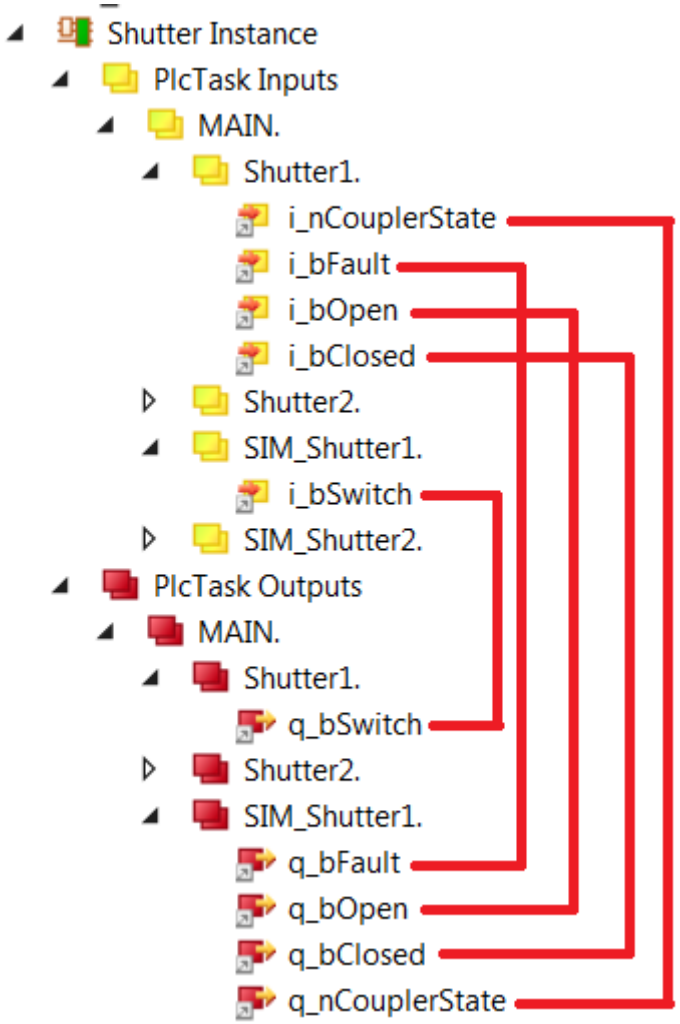
Declaration:

```
{attribute 'OPC.UA.DA':='1'}

Shutter1: FB_SHUTTER;

{attribute 'OPC.UA.DA':='1'}

SIM_Shutter1: FB_SIM_SHUTTER;
```

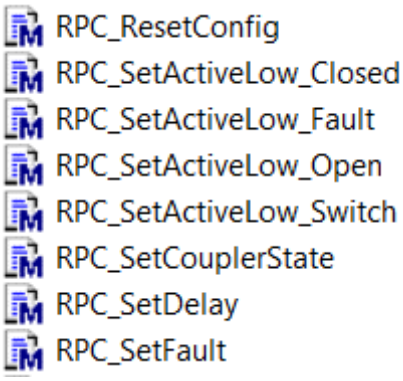
Execution:

```
Shutter1(in_sName:='Shutter1', in_bActiveLowOpen:=TRUE, in_
↪bActiveLowClosed:=TRUE);

SIM_Shutter1(ptrDev := ADR(Shutter1));
```

Simulator Mapping



Simulator RPC Methods



3.4 Time Library (timer.library)

FB_TIME is the ESO PLC Function Block that provides time services to other components running within the PLC, e.g. computation of absolute time.

3.4.1 Input parameters

The FB_TIME does not have input parameters.

3.4.2 Signal Mapping

The *FB_TIME* defines a number of mappings that are needed to do the correct computation of the time. Most of these mappings come from the EL6688 terminal. If this terminal is not available in the HW configuration, the *FB_TIME* could still be used to simulate a particular time in the PLC.

The table below describes each mapping variable.

Variable	Port Type	Optional Mapping	Description
i_inTime	**ULINT**	**No**	**Mapped to the EL6688 internal time stamp**
i_exTime	**ULINT**	**No**	**Mapped to the EL6688 external time stamp**
i_dc2TcOffset	**LINT**	**No**	**Mapped to the EtherCAT Dc2TcOffet**
i_dc2ExtOffset	**LINT**	**Yes**	**Mapped to the EtherCAT Dc2ExtOffset**
i_extDevNotConnected	**BOOL**	**No**	**Mapped to the EL6688 External device not connected**
i_syncMode	**UINT**	**Yes**	**Mapped to the EL6688 Sync mode**
i_AdsAddr	**AMSADDR**	**No**	**Mapped to the EL6688 ADS address**
q_timeInfo	**T_TIME_INFO**	**Yes**	**Delivers the actual absolute time (DC) and mode**

3.4.3 GUI Template

As for other PLC libraries, the *timer.library* provides a template GUI *GUI_TEMPLATE_TIME* for control of *FB_TIME* instances. Applications can easily deploy an instance of this GUI by setting the GUI references to the particular instance of *FB_TIME* and its name.

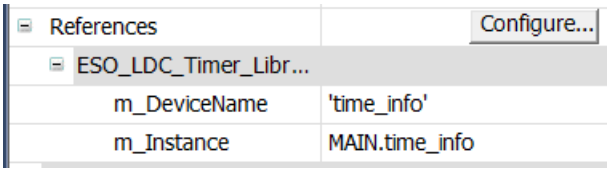


Fig. 3.5: Instantiation of GUI_TEMPLATE_TIME for time_info instance

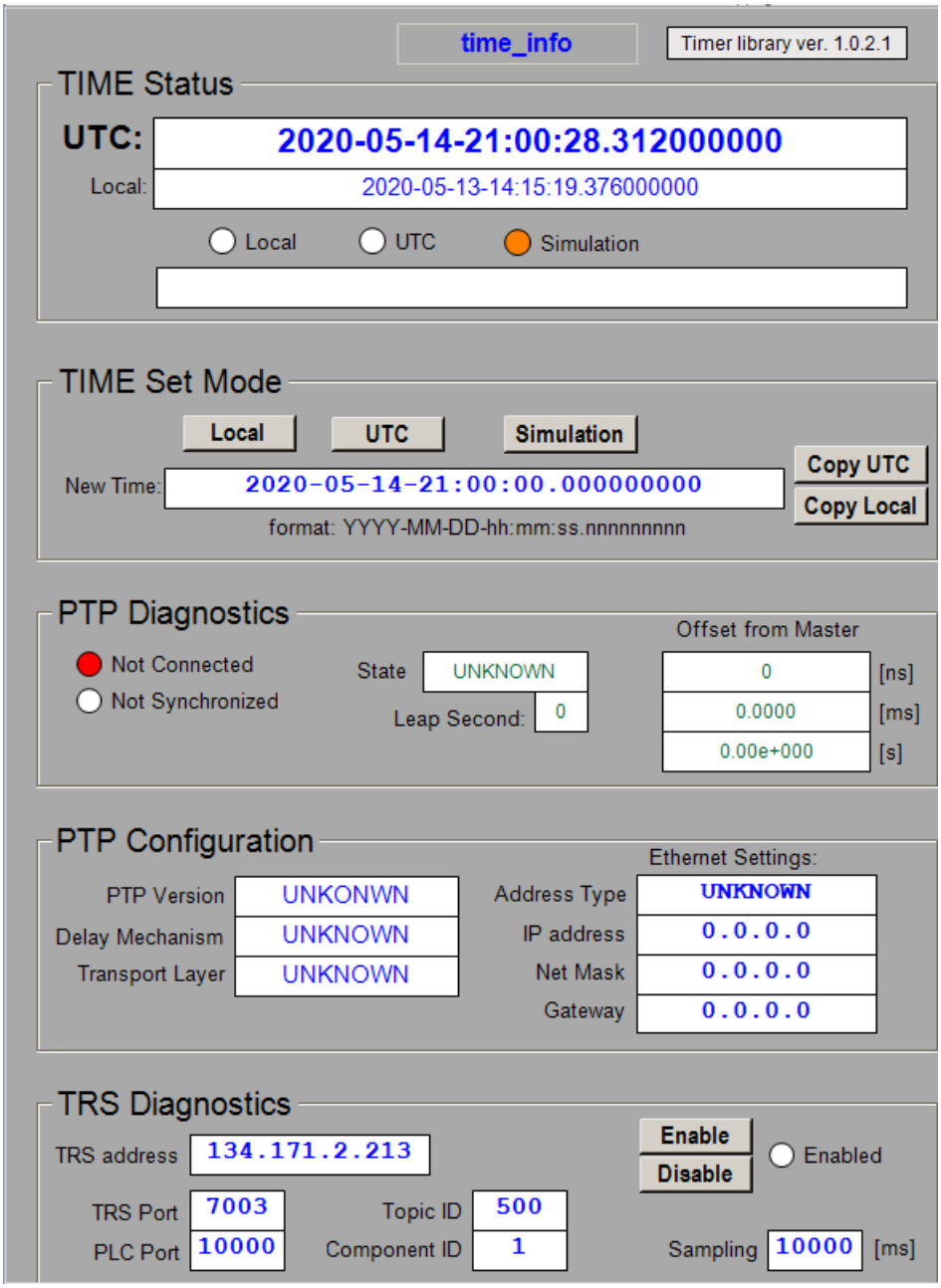


Fig. 3.6: FB_TIME HMI for Local Control.

The user has a choice of three time signals:

Time Signal	Description
UTC	**Time signal coming from the IEEE1588 PTP server. This is the default time signal.**
Local	**Local time of the Beckhoff IPC.**
Simulated	**Simulated time. This is the perfect tool for testing SW for specific date and time.**

The GUI instance shown on the screenshot above, the time is simulated and set in the future.

The two buttons *Copy UTC* and *Copy Local* are used to copy UTC or Local time string into the entry field for simulated time, avoiding unnecessary typing.

In *UTC* mode, if the time signal gets lost, the time will automatically switch to *Local* mode.

3.4.4 Time specific RPC Methods

- RPC_GetMode() Get actual time mode: Local, UTC or Simulation.
- RPC_SetMode() Set new time mode (Local, UTC or Simulation).
- RPC_GetUTC() Obtain the actual absolute time and mode.
- RPC_SetTime(string) Set a user defined time (only works in Simulation mode).

3.5 Motor Library (motor.library)

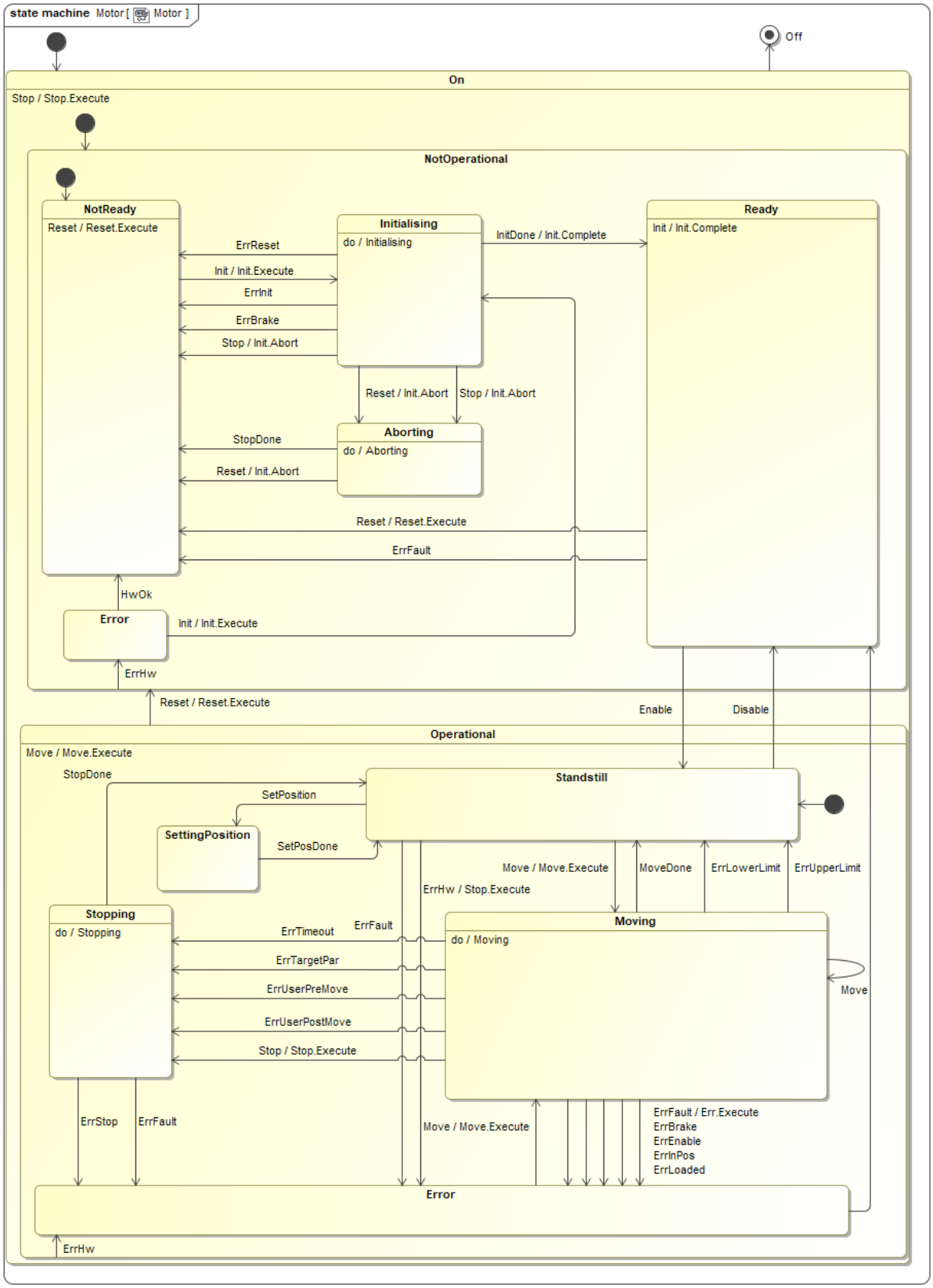
Note: Important note: The cycle time of the task (e.g. *PlcTask*) that executes an instance of any FB delivered in motor.library that controls motors, i.e. *FB_MOTOR*, *FB_MA_ADC*, *FB_MA_DROT*, etc, must be the same or longer than the cycle time of the *NC-Task 1 SAF* task. In other words, the instance of these FBs must not be executed faster than the *NC-Task 1 SAF* task. Otherwise, there could be some synchronization problems, in particular when exiting tracking mode.

Note that the default cycle time of the *NC-Task 1 SAF* task is 2 ms. Therefore, if for example the *PlcTask* cycle time is set to 1 ms, the *NC-Task 1 SAF* task cycle time has to be set to the same value.

FB_MOTOR is the TwinCAT PLC Function Block used to operate motors. The library is based on the Beckhoff *MC* Library that is in turn *PLCOpen MC compliant*. This means that it should be possible to use the library for control of any type of motor, including brushless motors, under the condition that the motor controller is fully EtherCAT certified and PLCOpen MC compliant. So far, stepper, DC, BLDC (brushless DC) and synchronous motors have been successfully tested with the library.

3.5.1 State Machine

The state machine of the motor controller is shown below. The main operational states are *Standstill*, *Moving* and *Stopping*.


state machine Motor [Motor]
Off
On
Stop / Stop.Execute
NotOperational
NotReady
Reset / Reset.Execute
ErrReset
Initialising
do / Initialising
InitDone / Init.Complete
Ready
Init / Init.Complete
Init / Init.Execute
ErrInit
ErrBrake
Stop / Init.Abort
Reset / Init.Abort
Stop / Init.Abort
Aborting
do / Aborting
StopDone
Reset / Init.Abort
Reset / Reset.Execute
ErrFault
HwOk
Error
Init / Init.Execute
ErrHw
Reset / Reset.Execute
Enable
Disable
Operational
Move / Move.Execute
StopDone
SetPosition
Standstill
SettingPosition
SetPosDone
Move / Move.Execute
MoveDone
ErrLowerLimit
ErrUpperLimit
ErrHw / Stop.Execute
Stopping
do / Stopping
ErrTimeout
ErrFault
Moving
do / Moving
ErrTargetPar
ErrUserPreMove
ErrUserPostMove
Stop / Stop.Execute
Move
ErrStop
ErrFault
Move / Move.Execute
ErrFault / Err.Execute
ErrBrake
ErrEnable
ErrInPos
ErrLoaded
Error
ErrHw

3.5.2 Usage of NOVRAM

Note: Important note: If the motor is configured to use NOVRAM, the configuration of the motor is copied into NOVRAM on every **successful** INIT of the motor. If the INIT fails, the NOVRAM doesn't get updated.

NOVRAM is used to store and keep the configuration of the motor. The configuration of the motor is copied into the NOVRAM on every successful INIT of the motor. On power cycle of the PLC, the configuration of the motor is read from the NOVRAM during the first PLC cycle. As a general rule, only PLCs with NOVRAM should be used for motor control applications, e.g. CX-2030 that comes with 128 kB of NOVRAM. PLCs without NOVRAM can still be used for motor control applications but the configuration will be lost on every power cycle of the PLC. These applications have to rely on the higher level software that has to download the configuration after each power cycle. The other option is to hard-code the configuration in the PLC application itself.

3.5.3 Input parameters

```
FUNCTION_BLOCK FB_MOTOR

VAR_INPUT   sName           STRING  Default Motor name
VAR_INPUT   nNOVRAM_DevId   UDINT   NOVRAM device ID - normally 4
VAR_INPUT   nNOVRAM_Offset  UDINT   NOVRAM offset where motor configuration is stored
```

FB_MOTOR has three input parameters. Apart from the standard sName parameter for the name, i.e. the label, of the motor instance, there are two additional parameters related to the usage of the NOVRAM.

Parameter	Type	Description
sName	**STRING**	**Motor instance name/label**
nNOVRAM_DevId	**UDINT**	**NOVRAM Device ID. For PLCs without NOVRAM, this parameter should be set to zero or not given at all.**
nNOVRAM_Offset	**UDINT**	**Offset in bytes in NOVRAM. A motor instance needs about 500 bytes for configuration data. For simplicity it is recommended to increment offsets by 1000 for each motor. Therefore, the first motor will have the offset of zero, second of 1000, third of 2000, etc. For PLCs without NOVRAM, this parameter should be set to zero or not given at all.**

The following example shows the usage of NOVRAM. It is very important to note that the entered value for the parameter nNOVRAM_DevId matches the value of the NOV-DP-RAM device. In this example the value is 4.

In case that the NOVRAM is not used, the *Motor1* instance would not have any NOVRAM related parameters and the code would look like this:

```
Motor1(sName:='Motor1');
```

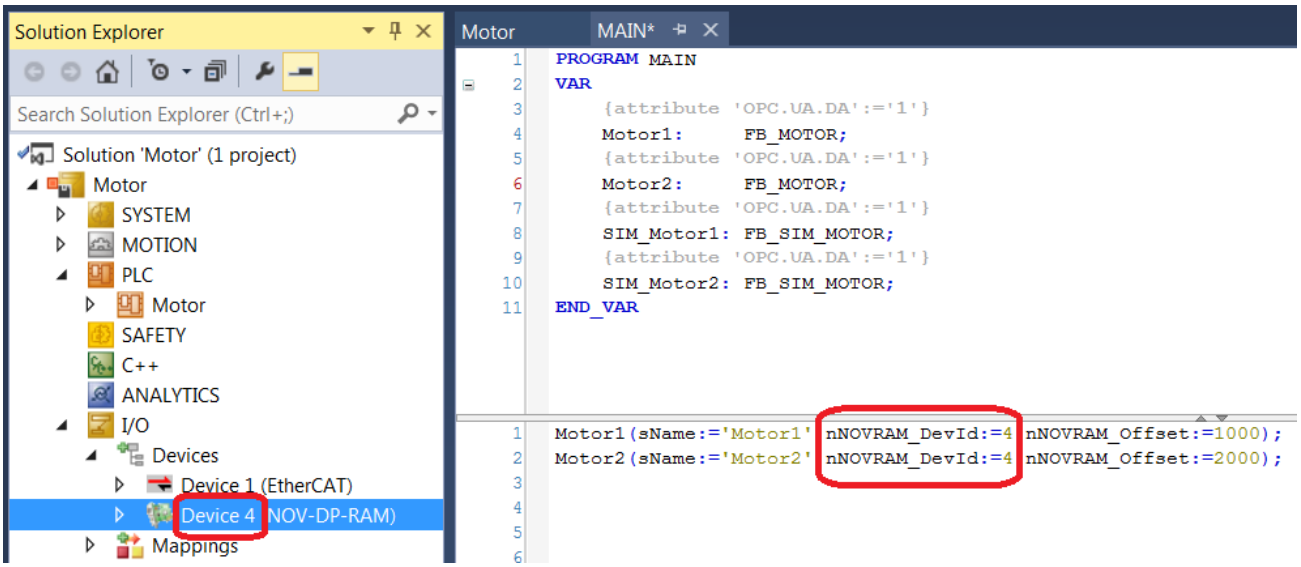


Figure 7: Application example for using NOVRAM with FB_MOTOR.

3.5.4 Signal Mapping

Figure 8 shows the TwinCAT view of the *FB_MOTOR* I/O variables that are available for mapping to physical signals, i.e. ports of I/O terminals or NC structures.

The mapping is quite different from other devices. Specific to motors, for each Axis (i.e. motor) there are two links to be established from the Axes/<Motor> Settings:

1. *Link To I/O...* This is the link between the Axis and the motor controller terminal, e.g. EL7041, that controls it.
2. *Link To PLC...* This is the link between the Axis and the motor instance in the *MAIN* program, e.g. *Motor1*. This link sets all the mappings for the two complex structures *NcToPlc* (input) and *PlcToNc* (output) and the user should not touch it afterwards.

The mapping parameters are described in the table below.

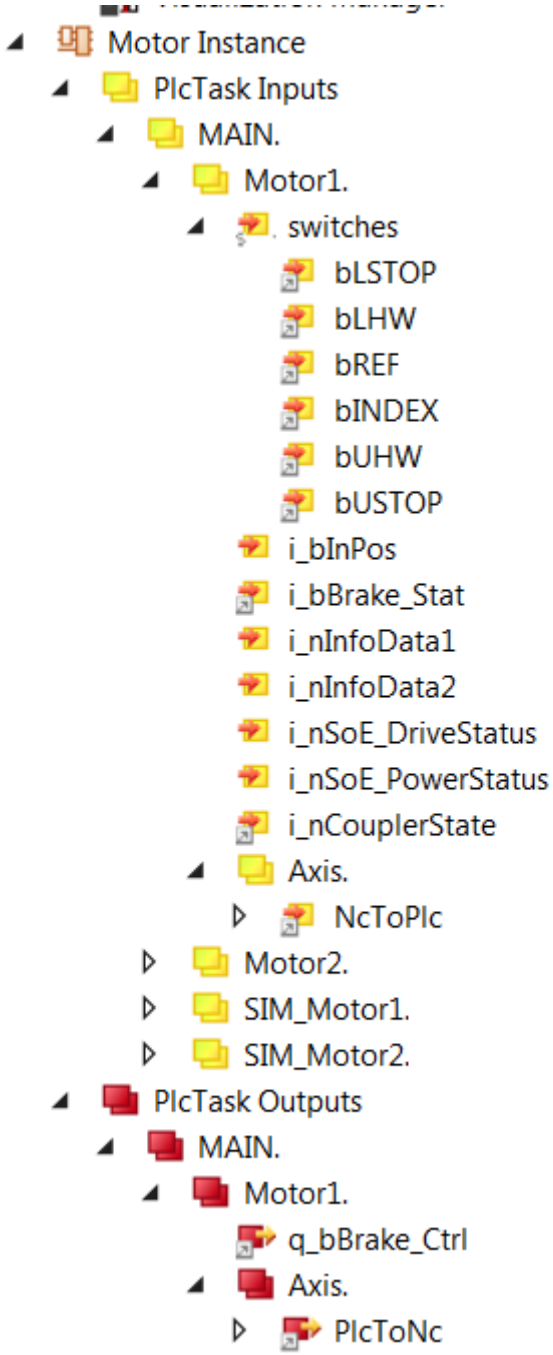


Figure 8: Example of FB_MOTOR input/output signals.

Variable	Port Type	Optional	Description
switches	**Digital In**	**Yes**	**There are six variables to be linked to various limit and reference switches, depending on the HW configuration; upper and lower Stop and HW limits, Reference and Index switch. Switches can be linked to any binary signal (BOOL). Normally the signals are coming from digital inputs but they could also be linked to variables of type BOOL in some special applications, i.e. digitizing of analog position sensors.**
i_bInPos	**Digital In**	**Yes**	**Signal of the In-Position switch. In some applications with stepper motors without feedback (cryogenic environment) a switch is used to confirm that the motor is in correct position.**
i_bBrakeStat	**Digital In**	**Yes**	**Status of the brake feedback signal.**
i_nInfoData1/2	**INT**	**Yes**	**Two freely selectable signed integers to be link to any variable of the same type, e.g. motor controller output current.**
i_nSoE_DriveStatus	**UINT**	**Yes**	**Serco Drive (e.g. AX5000) Status, not used with CoE drives.**
i_nSoE_PowerStatus	**UINT**	**Yes**	**Serco Drive (e.g. AX5000) Power Status, not used with CoE drives.**
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts a shutter signal.**
NcToPlc	**Struct In**	**No**	**Internal MC structure of type MC.NCTOPLC_AXIS_REF. Automatically linked from "Link To PLC".**
q_bBrake_Ctrl	**Digital Out**	**Yes**	**Brake control digital output signal. Active low!**
PlcToNc	**Struct Out**	**No**	**Internal MC structure of type MC.PLCTONC_AXIS_REF. Automatically linked from "Link To PLC".**

3.5.5 GUI Templates

The Motion Library provides two template GUIs intended for configuration and control of motor instances of FB_MOTOR respectively. Applications can easily deploy instances of the GUIs by setting the GUI references to the particular instance of FB_MOTOR_CONTROL, as shown below for the case of the configuration GUI.

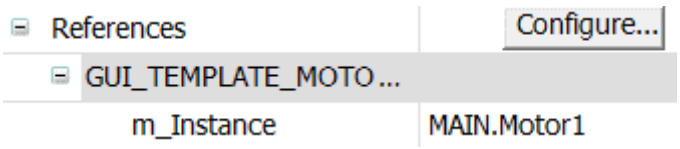


Figure 9: Instantiation of GUI_TEMPLATE_MOTOR_CONFIG for Motor1

From the Configuration GUI, shown in Figure 10, it is very simple to set axis type, define INIT sequence, select user defined methods to be executed, set the active low parameters for each switch, as well as to configure brakes, define timeouts, software limits and backlash compensation.

Once the motor is successfully initialised, the complete configuration will be stored in NOVRAM.

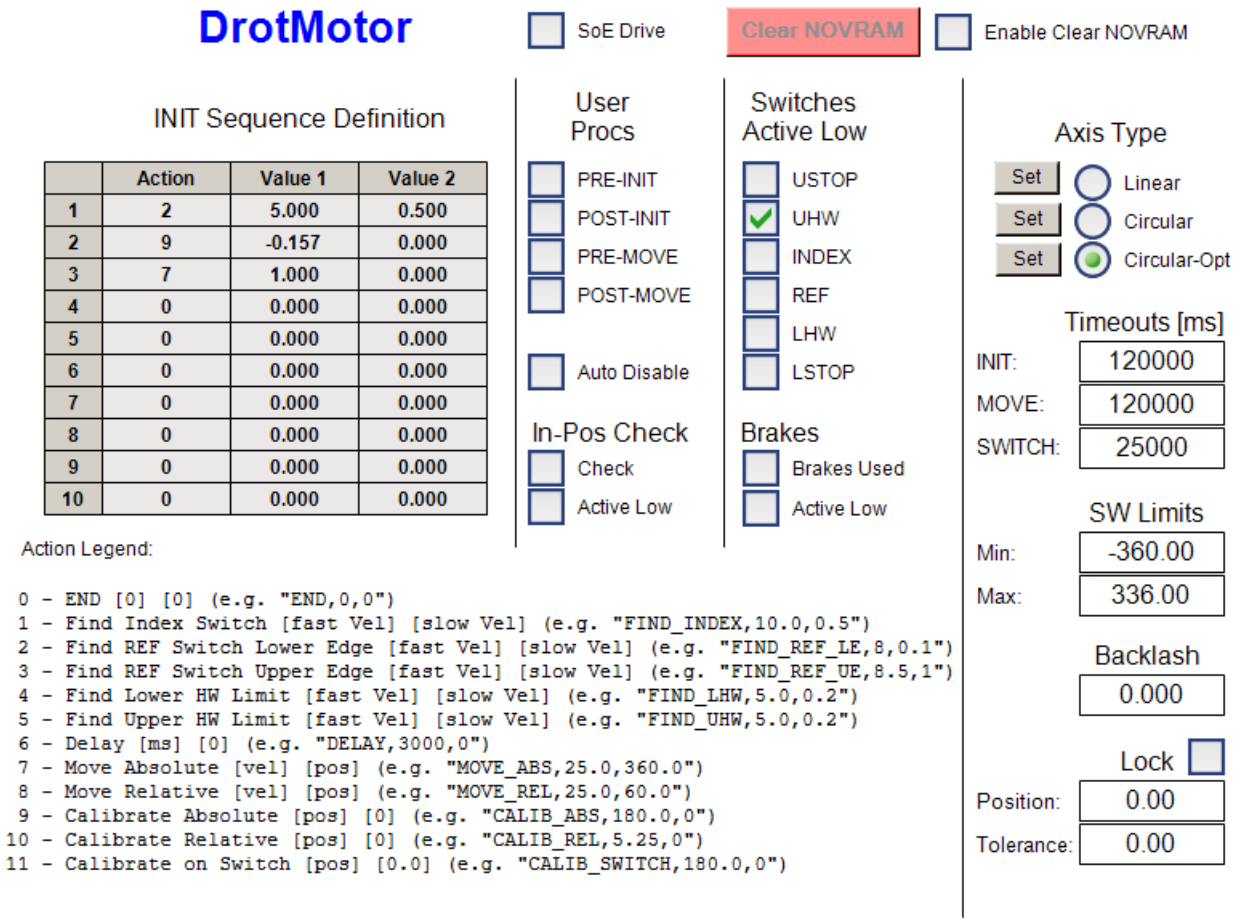


Figure 10: FB_MOTOR HMI for motor configuration.

From the Control GUI, shown in Figure 11, it is possible to perform basic control operations on the motor.

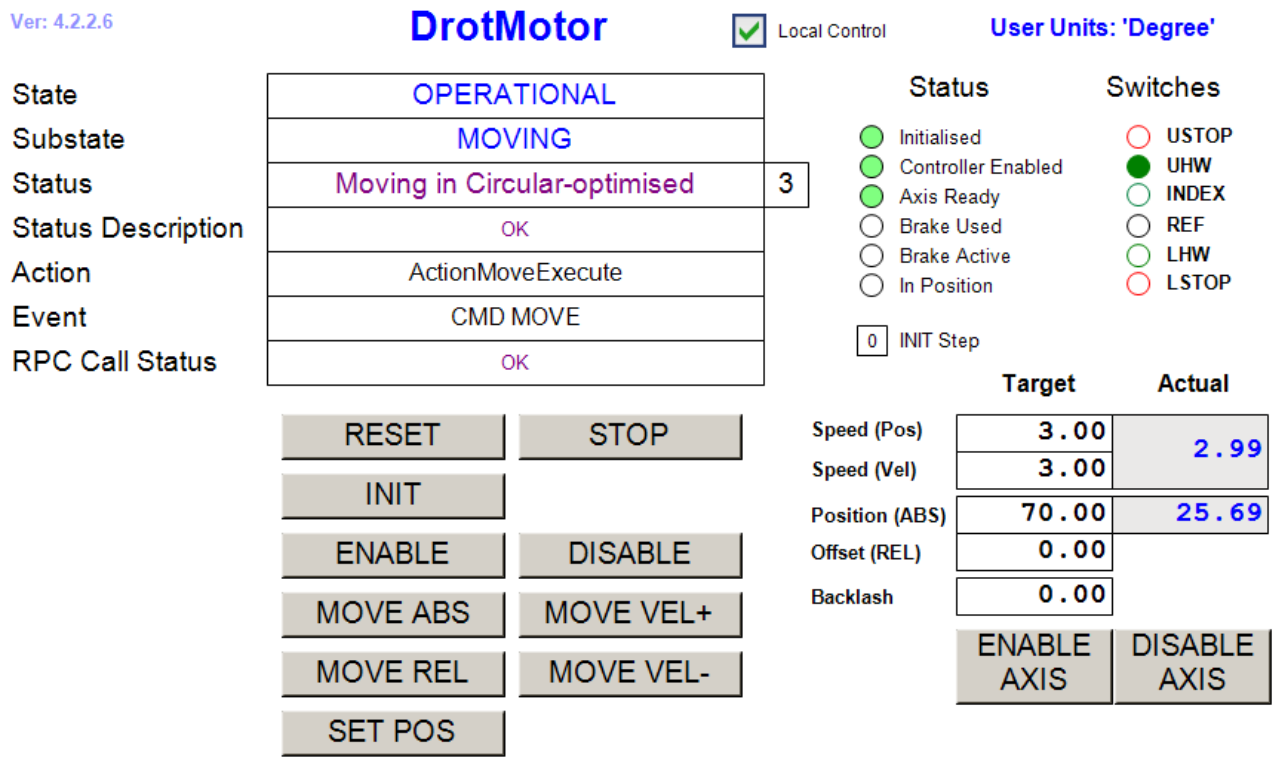


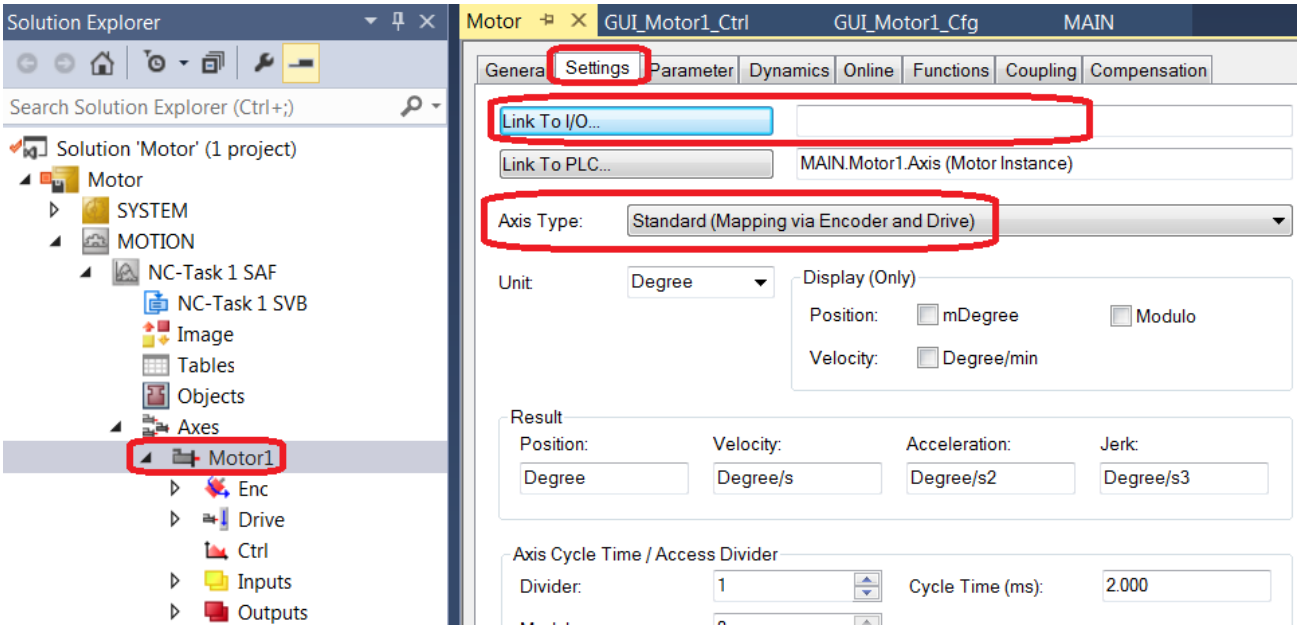
Figure 11: FB_MOTOR HMI for Local Control.

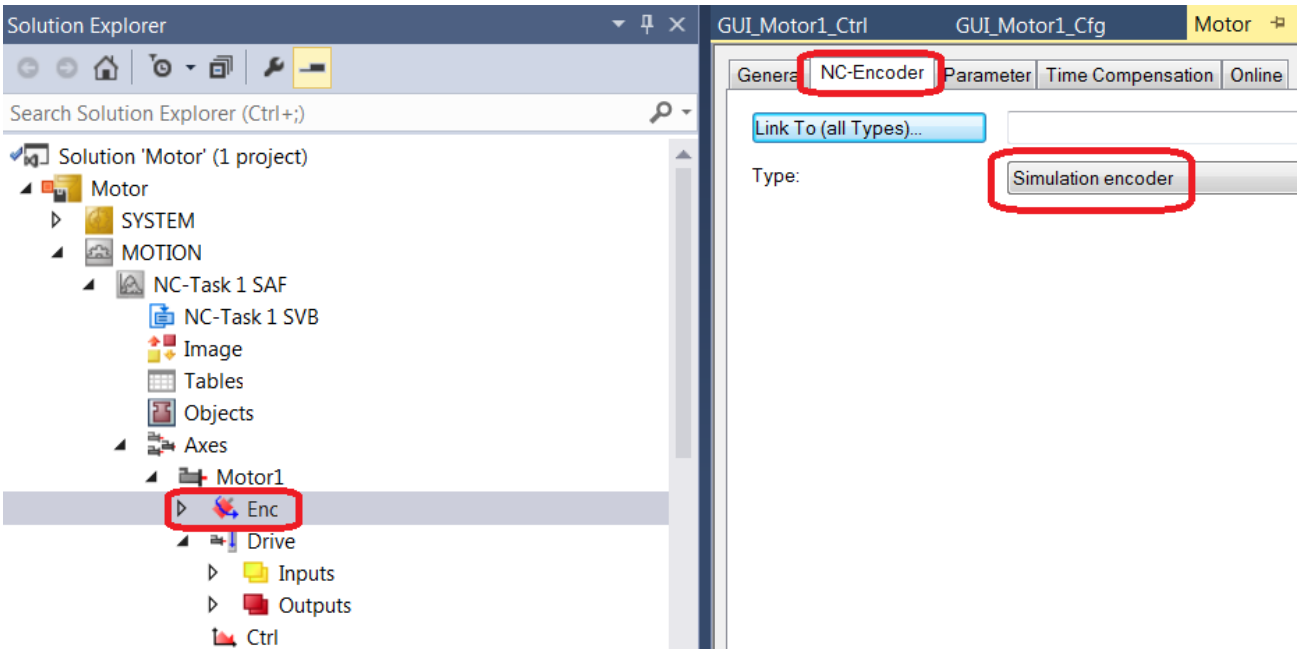
3.5.6 Motor specific RPC Methods

- RPC_MoveAbs() Move to absolute position
- RPC_MoveRel() Move relative to current position
- RPC_MoveVel() Move in velocity

3.5.7 Motor Simulator

Note: Important note: In order to use the motor simulator, the Axis has to be modified. The Axis Type has to be set to *Standard (Mapping via Encoder and Drive)* and the *Link to I/O...* should be left empty. The Axis Encoder has to be configured as *Simulation encoder*. See screen shots below.





The function block *FB_SIM_MOTOR* implements the motor simulator on the PLC. The input parameter of the simulator are addresses of the motor instance configuration and status structures. The following code shows how the simulator is declared and executed:

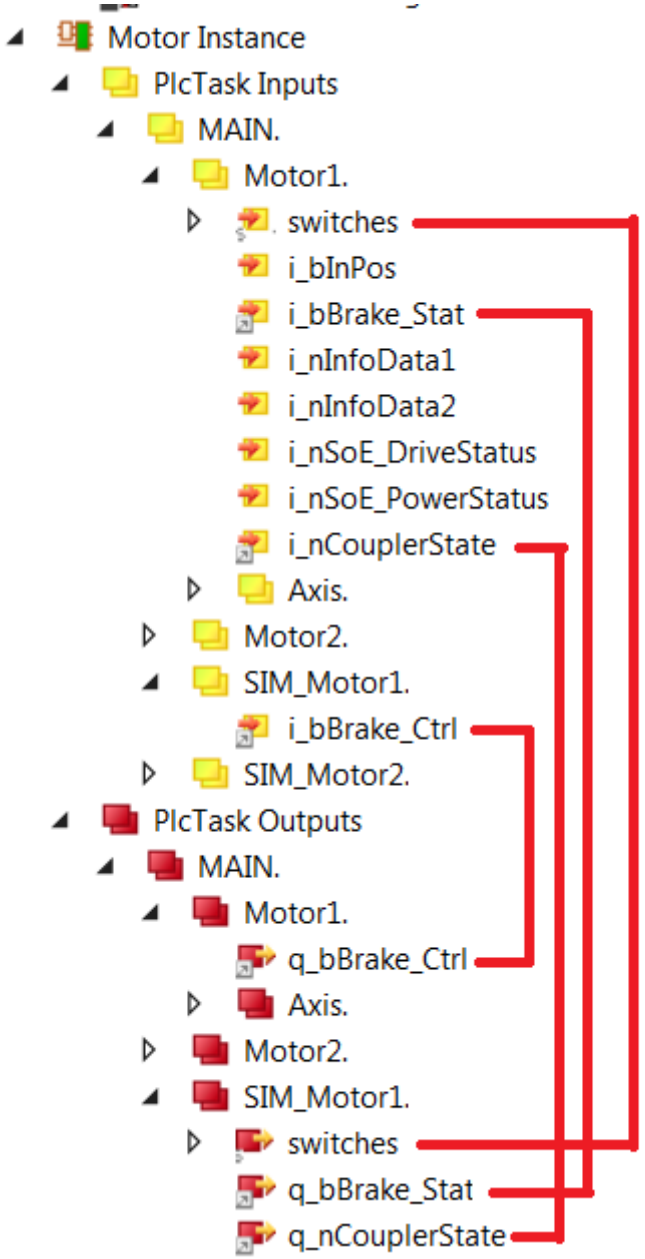
Declaration:

```
{attribute 'OPC.UA.DA':='1'}

Motor1: FB_MOTOR;

{attribute 'OPC.UA.DA':='1'}

SIM_Motor1: FB_SIM_MOTOR;
```

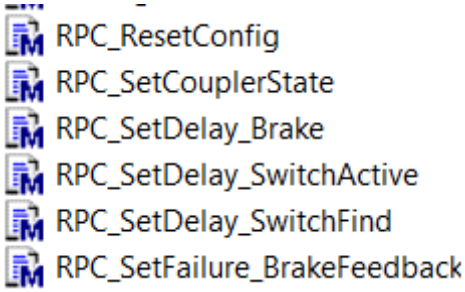
Execution:

```
Motor1(sName:='Motor1', nNOVRAM_DevId:=0);

SIM_Motor1 (ptrCfg:=ADR(Motor1.cfg), ptrStat:=ADR(Motor1.stat));
```

Simulator Mapping

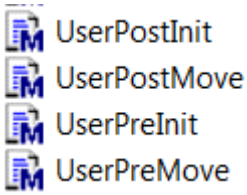


Simulator RPC Methods



3.5.8 User Defined Methods

The functionality of *FB_MOTOR* can be extended by user defined Pre- and Post INIT and MOVE methods. *FB_MOTOR* already includes the four dummy methods (see below) that only add a delay of three seconds during the execution. However, the dummy implementation can be a good starting point when overloading these methods for specific implementations in FBs that extend the *FB_MOTOR* functionality.



3.6 CCS Simulator (motor.library)

Tracking devices require information from CCS to be able to compute the corrections to be applied to the actuators. This information will be obtained later from CCS but in the meantime, FCF includes a utility that enable a simple way to validate tracking without the CCS infrastructure. It is a dummy FB (*FB_CCS_SIM*) that can be edited to define the telescope coordinates, environmental conditions and other parameters needed by the library computing the tracking data (*slalib*). This FB avoid duplicating this information per each tracking device.

3.6.1 Input parameters

The *FB_CCS_SIM* does not have input parameters.

3.6.2 Signal Mapping

Figure below shows an example of the mapping related to the CCS Simulator (ccs_sim). It relates to the C++ modules, an instance to the FB_TIME (time_info) and a tracking device such a the derotator (drot).

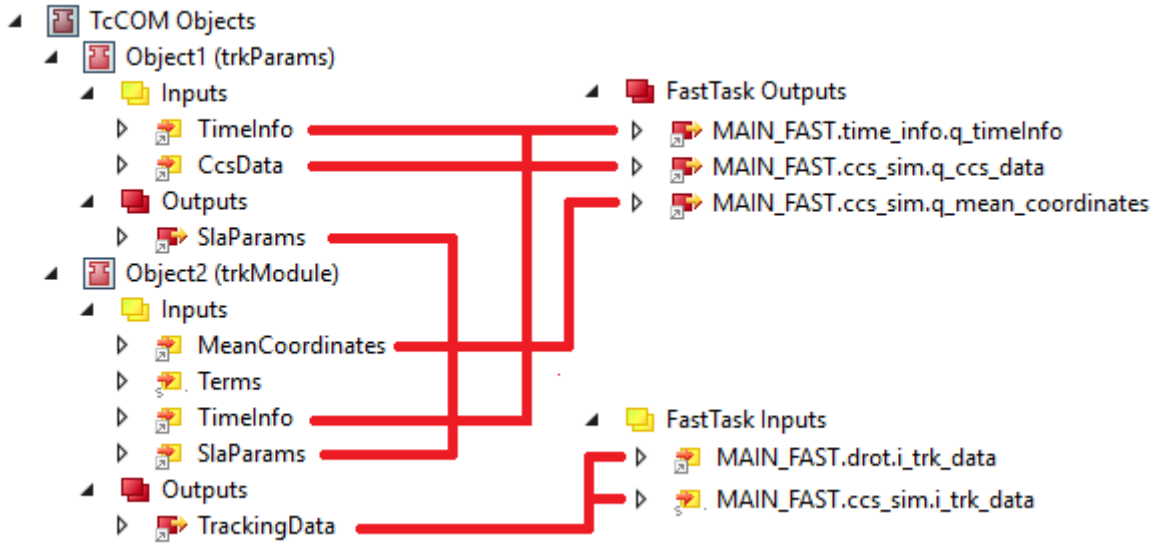


Fig. 3.7: Example of the *FB_CCS_SIM* mapping.

The specific mapping parameters of CCS Simulator are described in the table below.

Variable	Port Type	Optional Mapping	Description
i_trk_data	**TrackingData**	**No**	**Structure delivered by the C++ Module**
q_ccs_data	**CcsData**	**No**	**Structure containing the complete CCS data**
q_mean_coordinates	**TrkMeanCoordinates**	**No**	**Structure containing the MEAN coordinates**

3.6.3 GUI Template

As for the other FBs, the Motor Library provides a template GUI to control the *FB_CCS_SIM*. Applications can easily deploy an instance of this GUI to control their own time function block by setting the GUI references to the particular instance of the *FB_CCS_SIM*.

Instantiation of GUI_TEMPLATE_CCS_SIM for ccs_sim

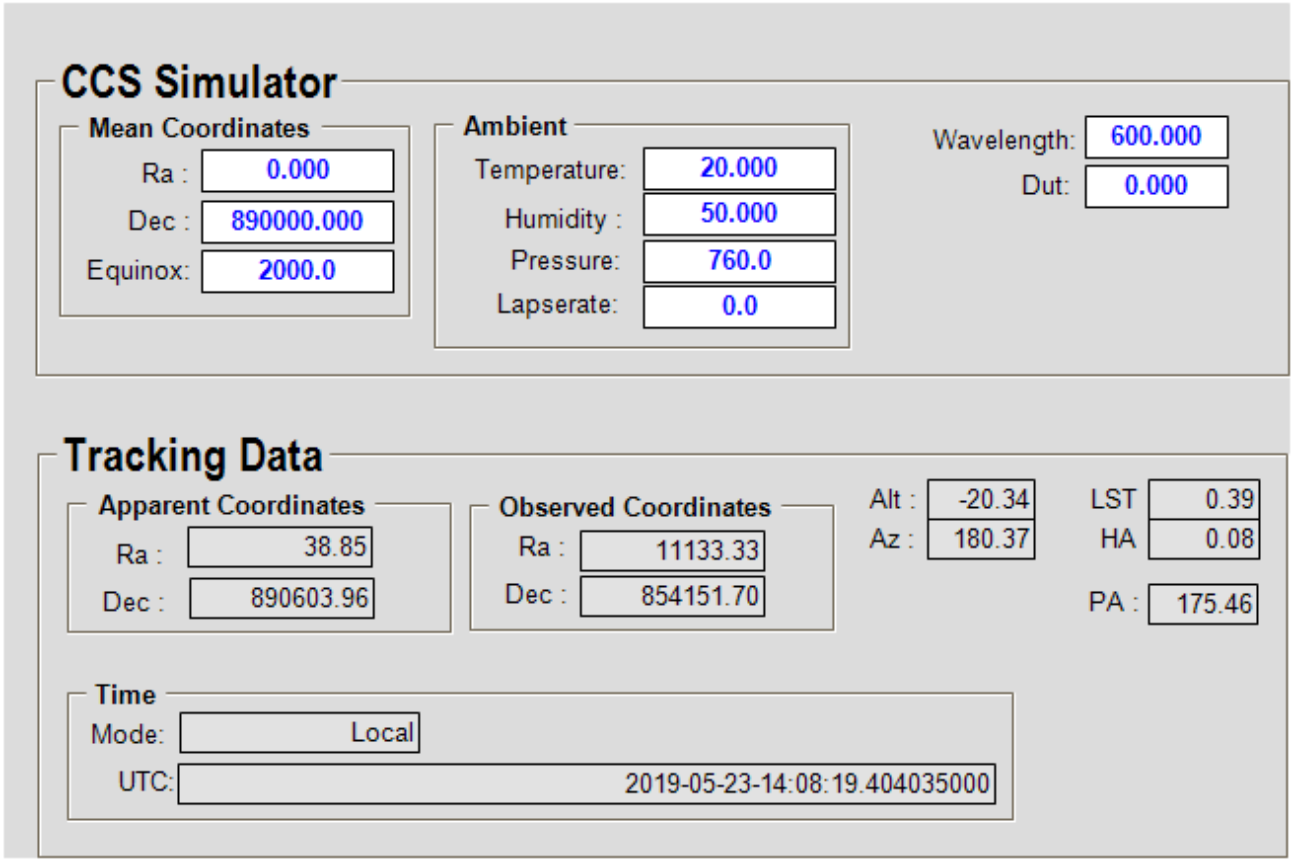


Fig. 3.8: *FB_CCS_SIM* HMI for Local Control.

3.6.4 CCS_SIM specific RPC Methods

RPC_SetCoordinates() This RPC allows the user to change RA,DEC and EQUINOX via OPCUA.

3.7 Drot Controller (motor.library)

FB_MA_DROT is the TwinCAT PLC Function Block used to operate a derotator. The FB encapsulates the motion control functionality using a composition of a motor device together with the functionality to derive the next motor position using the field rotation obtained from a dedicated C++ module.

Note: All the functionalities provided for a normal motor device are available as well for the derotator.

3.7.1 Input parameters

FB_MA_DROT has four input parameters. Apart from the standard sName parameter for the name, i.e. the label, of the derotator instance, there are three additional parameters used by the motor used internally by the derotator.

Parameter	Type	Description
sName	**STRING**	**Drot instance name/label**
sMotorName	**STRING**	**Motor instance name/label**
nNOVRAM_DevId	**UDINT**	**NOVRAM Device ID. For PLCs without NOVRAM, this parameter should be set to zero or not given at all.**
nNOVRAM_Offset	**UDINT**	**Offset in bytes in NOVRAM. A motor instance needs about 500 bytes for configuration data. For simplicity it is recommended to increment offsets by 1000 for each motor. Therefore, the first motor will have the offset of zero, second of 1000, third of 2000, etc. For PLCs without NOVRAM, this parameter should be set to zero or not given at all.**

Note: The usage of NOVRAM follows the same guidelines as for a FB_MOTOR.

In case that the NOVRAM is not used, the *Drot1* instance would not have any parameters and the code would look like this:

```
Motor1(sName:='Drot1', sMotorName:= 'Drot1Motor');
```

3.7.2 Signal Mapping

The mapping includes all the mapping of a motor plus the specific mapping of the derotator.

Specific to the internal derotator motor, there are two links to be established from the Axes/<Motor> Settings:

1. *Link To I/O...* This is the link between the Axis and the motor controller terminal, e.g. EL7041, that controls it.
2. *Link To PLC...* This is the link between the Axis and the motor instance in the *MAIN* program, e.g. *drot1.motor*. This link sets all the mappings for the two complex structures *NcToPlc* (input) and *PlcToNc* (output) and the user should not touch it afterwards.

The description of the standard motor mapping can be found here *motor-signal-mapping*

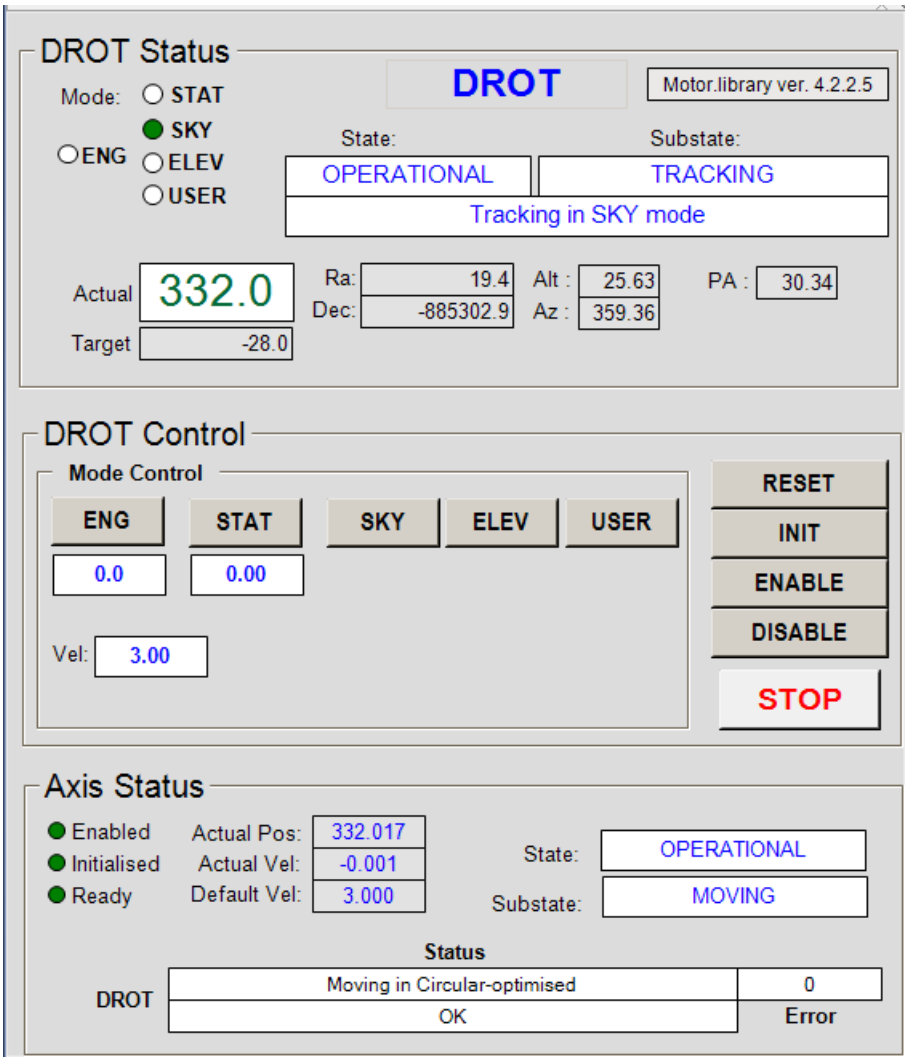
The additional derotator mapping parameters are described in the table below.

Variable	Port Type	Optional	Description
i_trk_data	**Struct In**	**No**	**Tracking data computed by the C++ module.**
i_ccs_data	**Struct In**	**Yes**	**CCS Simulation data**

Warning: The above mapping table does not include the standard motor mapping.

3.7.3 GUI Templates

The Motion Library provides one template GUI intended for the control of instances of FB_MA_DROT. Applications can easily deploy instances of the GUI by setting the GUI reference to the particular instance of FB_MA_DROT.



FB_MA_DROT HMI for Local Control

From the Derotator Control GUI, shown above, it is possible to perform basic control operations on the Derotator. The GUI also shows the status of the subordinated motor.

Note: It is possible to create dedicated control and configuration GUIs for the derotator subordinated motor as for any other motor device.

3.7.4 Derotator specific RPC Methods

RPC name	Parameters	Description
RPC_MoveAngle()	**in_lrAngle**	**Move the derotator to a particular position angle.**
RPC_StartTrack()	**in_mode in_angle**	**Start derotator tracking.**
RPC_StopTrack()		**Stop derotator tracking.**

3.7.5 Derotator Simulator

Derotator does not have a dedicated simulator. Motor simulator shall be used to simulate derotator behaviour.

Declaration:

```
{attribute 'OPC.UA.DA':='1'}

drot:       FB_MA_DROT;

{attribute 'OPC.UA.DA':='1'}

sim_drot:   FB_SIM_MOTOR;
```

Execution:

```
drot(sName:='DROT', sMotorName:='DrotMotor', nNOVRAM_DevId:=0);
sim_drot(ptrCfg:=ADR(drot.motor.cfg), ptrStat:=ADR(drot.motor.stat));
```

3.7.6 User Defined Methods

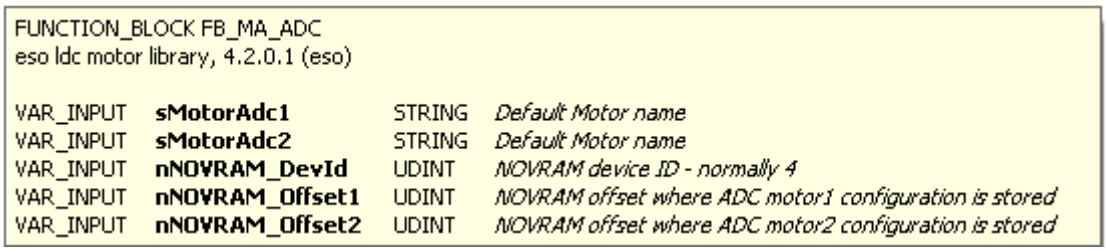
Derotator user defined functions are the same as for the Motor.

3.8 ADC Controller (motor.library)

FB_MA_ADC is the TwinCAT PLC Function Block used to operate a ADC. The FB encapsulates the motion control functionality using a composition of two internal motors (motor1 and motor2).

Note: All the functionalities provided for a normal motor device are available as well for the internal ADC motors.

3.8.1 Input parameters



FB_MA_ADC has five input parameters as shown in figure above.

Note: The usage of NOVRAM follows the same guidelines as for *FB_MOTOR*.

3.8.2 Signal Mapping

The mapping includes all the mapping of the two internal motors plus the specific mapping of the ADC.

Specific to the internal ADC motors, there are two links to be established from each of the Axes/<Motor> Settings:

1. *Link To I/O...* This is the link between the Axis and the motor controller terminal, e.g. EL7041, that controls it.
2. *Link To PLC...* This is the link between the Axis and the motor instance in the *MAIN* program, e.g. *drot1.motor*. This link sets all the mappings for the two complex structures *NcToPlc* (input) and *PlcToNc* (output) and the user should not touch it afterwards.

The description of each standard motor mapping can be found here *motor-signal-mapping*

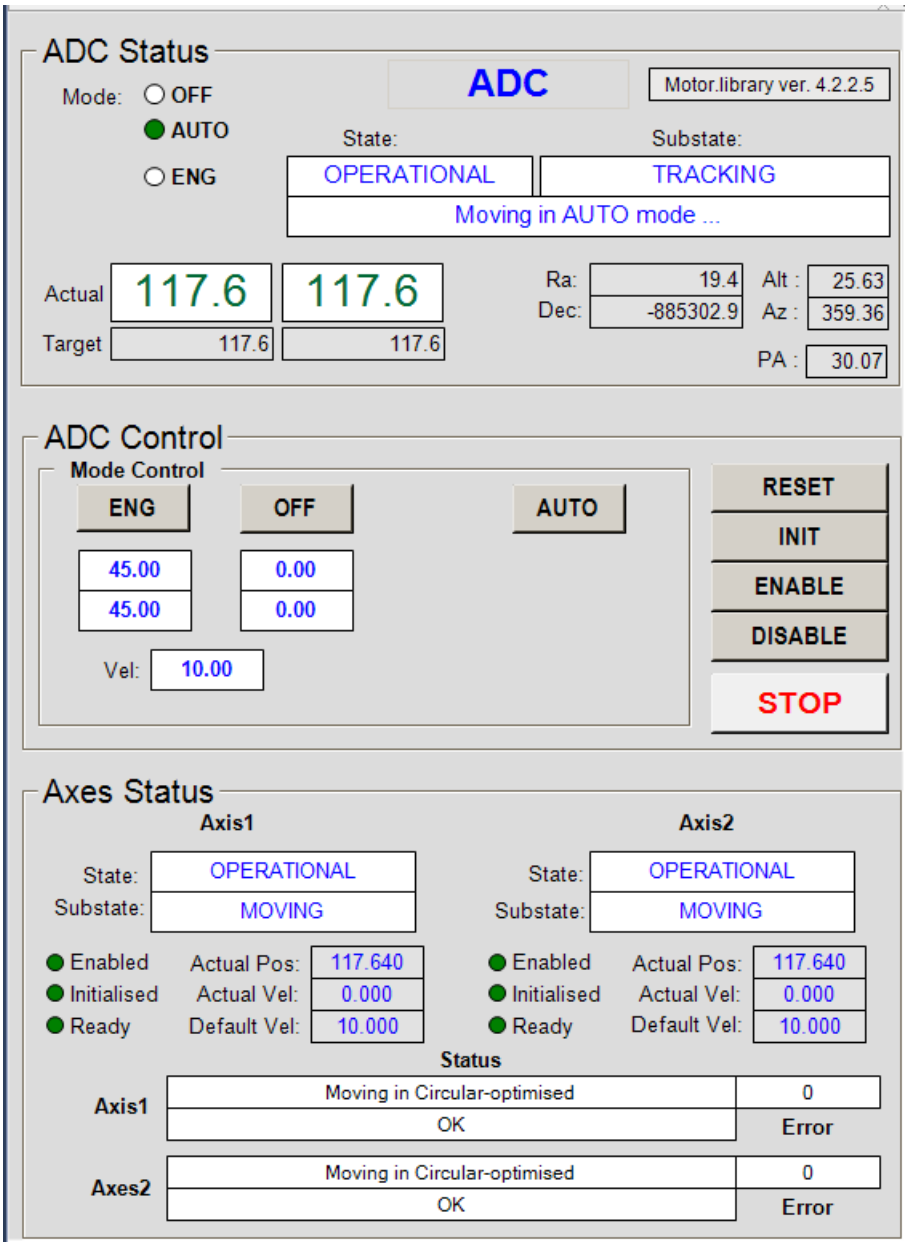
The additional ADC mapping parameters are described in the table below.

Variable	Port Type	Optional	Description
i_trk_data	**Struct In**	**No**	**Tracking data computed by the C++ module.**
i_ccs_data	**Struct In**	**Yes**	**CCS Simulation data**

Warning: The above mapping table does not include the mapping for the two internal motors.

3.8.3 GUI Templates

The Motion Library provides one template GUI intended for the control of instances of FB_MA_ADC. Applications can easily deploy instances of the GUI by setting the GUI reference to the particular instance of FB_MA_ADC.



FB_MA_ADC HMI for Local Control

From the ADC Control GUI, shown on above, it is possible to perform basic control operations on the ADC. The GUI also shows the status of the subordinated motors.

Note: It is possible to create dedicated control and configuration GUIs for the ADC subordinated motors as for any other motor device.

3.8.4 ADC specific RPC Methods

RPC name	Parameters	Description
RPC_MoveAngle()	**in_lrAngle**	**Move the ADC to a particular position angle.**
RPC_StartTrack()	**in_angle**	**Start ADC tracking.**
RPC_StopTrack()		**Stop ADC tracking.**

3.8.5 ADC Simulator

ADC does not have a dedicated simulator. Motor simulator shall be used to simulate the behaviour of the two internal motors.

Declaration:

```
{attribute 'OPC.UA.DA':='1'}
adc:       FB_MA_ADC;

{attribute 'OPC.UA.DA':='1'}
sim_adc1:   FB_SIM_MOTOR;

{attribute 'OPC.UA.DA':='1'}
sim_adc2:   FB_SIM_MOTOR;
```

Execution:

```
adc(sName           := 'ADC',
    sMotorAdc1      := 'AdcMotor1',
    sMotorAdc2      := 'AdcMotor2',
    nNOVRAM_DevId   := 4,
    nNOVRAM_Offset1 := 3000,
    nNOVRAM_Offset2 := 4000);

sim_adc1(ptrCfg:=ADR(adc.motor1.cfg), ptrStat:=ADR(adc.motor1.stat));
sim_adc2(ptrCfg:=ADR(adc.motor2.cfg), ptrStat:=ADR(adc.motor2.stat));
```

3.8.6 User Defined Methods

No user defined functions are provided by the ADC.

3.9 I/O Device Library (ioDev.library)

Note: The most common data type for analog signals is INT (16-bit signed). However, some I/O terminals, e.g. *EL3692*, can provide readings in user units of type REAL (FLOAT). *ioDev.library* supports both types of analog inputs.

---

ioDev.library contains a number of FBs for handling analog (INT and REAL) and digital sensors and I/O devices. In addition, it provides FB methods for scaling of signals in order to work directly in user units (temperatures, pressures, etc), instead of displaying pure voltages that cannot be easily interpreted by the user before they are scaled at the WS level. Simulators for analog and digital signals are also provided. They might be of great help at the early stages of projects when sensors are not available yet or for testing of out-of-range conditions.

The library provides a Function Block called *FB_IODEV_BASE* that has to be extended and customised by the user by adding input and output signals to the status and control structures. A number of I/O and Sensor FB examples that extend the *FB_IODEV_BASE* functionality can be found in the Examples directory. Example FBs with the string '_USER_CONFIG' in their name provide examples of user defined configuration, i.e. signal scaling, that is implemented in the *M_UserConfigure()* method. In addition to the extension of the FB functionality, the user has to extend the definition of the status *T_IODEV_STAT* and the control *T_IODEV_CTRL* structure (if any) by adding arrays of specific input and output signals.

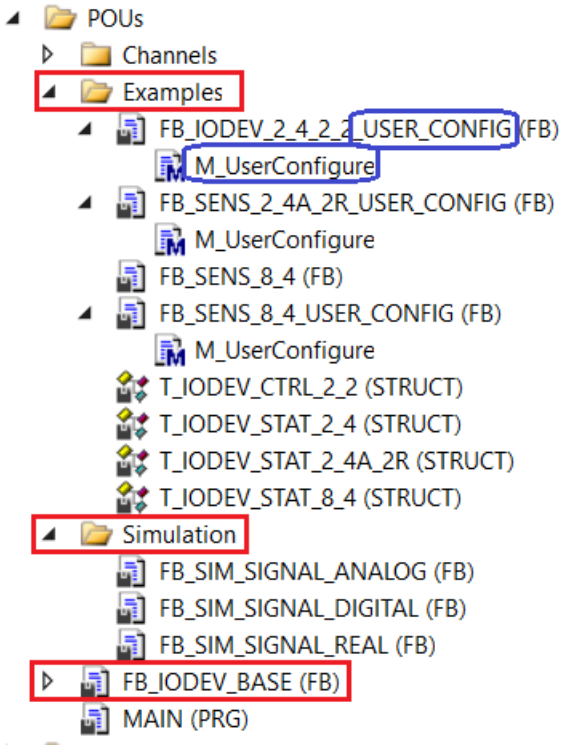
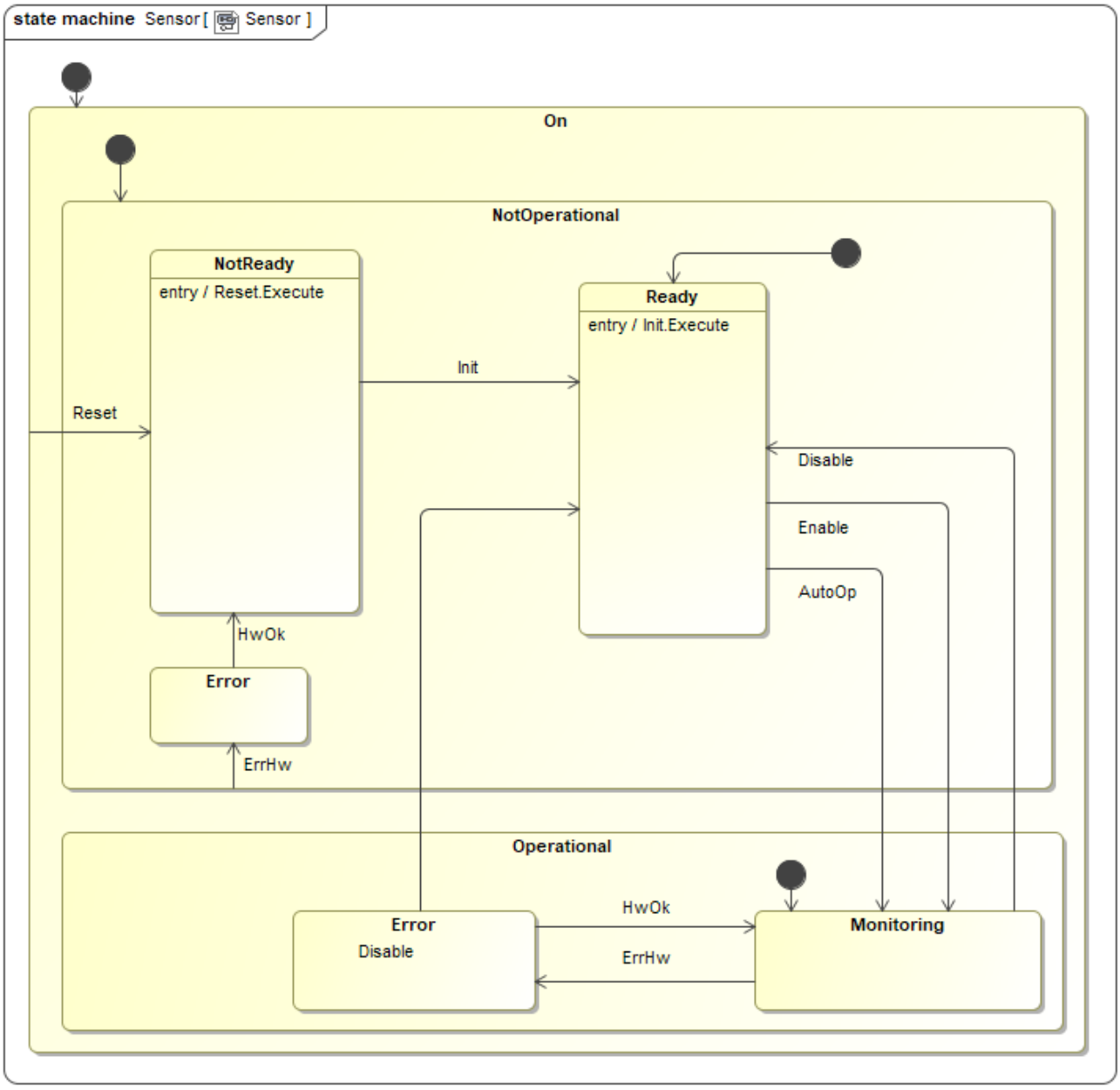


Fig. 3.9: Overview of *ioDev.library*

3.9.1 State Machine of Sensor

Sensor is a special case of the I/O Device that doesn't have any outputs. The state machine of the sensor controller is shown below. The main operational state is *Monitoring*.



Note: All I/O devices start monitoring in the *NotOp/Ready* state.

3.9.2 Input parameters

FUNCTION_BLOCK FB_IODEV_BASE			
VAR_INPUT	**in_sName**	STRING	*Instance name*
VAR_INPUT	**in_bAutoOp**	BOOL	*If TRUE, go automatically to OPERATIONAL state. Default FALSE.*
VAR_INPUT	**in_bUserConfig**	BOOL	*If TRUE, configuration in M_UserConfigure() is used. Default FALSE.*

3.9.3 User Customisation

As previously stated, the Function Block *FB_IODEV_BASE* provides all methods to handle I/O signals but no signals are defined. The FB has to be extended by the user by adding required I/O signals. The customisation process will be described through examples.

Example 1: I/O Device with User Defined Scaling

The FB *FB_IODEV_2_4_2_2_USER_CONFIG* is an example of the I/O device that handles 2 DI, 4 AI, 2 DO and 2 AO, with the user defined scaling.

The following steps have to be done:

- Extend the *T_IODEV_STAT* structure by adding arrays of input signals. The new structure is called *T_IODEV_STAT_2_4*, meaning 2 digital inputs and 4 analog inputs.

FB_SENS_8_4 | T_IODEV_CFG | FB_SENS_8_4_USER_CONFIG | T_IODEV_STAT_2_4

```
1  // EXAMPLE: Extension of T_IODEV_STAT with 2 DI and 4 AI.
2  //          Used in FB_IODEV_2_4_2_2_USER_CONFIG.
3  TYPE T_IODEV_STAT_2_4 EXTENDS T_IODEV_STAT :
4  STRUCT
5      //
6      // Signals, inputs.
7      //
8      arrDI:  ARRAY [0..1] OF FB_IODEV_CH_DIG_IN;     // 2 digital inputs
9      arrAI:  ARRAY [0..3] OF FB_IODEV_CH_ANLG_IN;    // 4 analog inputs
10 END_STRUCT
11 END_TYPE
12
```

- Extend the *T_IODEV_CTRL* structure by adding arrays of output signals. The new structure is called *T_IODEV_CTRL_2_2*, meaning 2 digital outputs and 2 analog outputs.

FB_SENS_8_4 T_IODEV_CFG FB_SENS_8_4_USER_CONFIG T_IODEV_CTRL_2_2

```
1  // EXAMPLE: Extension of T_IODEV_CTRL with 2 DO and 2 AO.
2  //          Used in FB_IODEV_2_4_2_2_USER_CONFIG.
3  TYPE T_IODEV_CTRL_2_2 EXTENDS T_IODEV_CTRL :
4  STRUCT
5      // Signals, Outputs
6      arrDO:  ARRAY [0..1] OF FB_IODEV_CH_DIG_OUT;    // 2 digital outputs
7      arrAO:  ARRAY [0..1] OF FB_IODEV_CH_ANLG_OUT;   // 2 analog outputs
8  END_STRUCT
9  END_TYPE
```

- Define signal scaling for both inputs and outputs in the *M_UserConfigure()* method.

FB_IODEV_2_4_2_2_...IG.M_UserConfigure

```
METHOD M_UserConfigure
VAR_INPUT
END_VAR
```

```
// TODO_USER
//
// Example implementation.
// Hard code any signal configuration.
//

// Configuration parameter cfg.bUserConfig defines if
// the hard-coded configuration below should be used or
// the signals will be configured from the WS.

//
// Digital signals
//
// Digital inputs (defined in stat structure T_IODEV_STAT_CUSTOM)
// Signal DI[0] is a door open signal
// Signal DI[1] is a light switch signal
stat.arrDI[0].M_Configure('CLOSED', 'OPEN');
stat.arrDI[1].M_Configure('OFF',    'ON');

// Digital outputs (defined in ctrl structure T_IODEV_CTRL_CUSTOM)
// Signal DO[0] is a pure digital signal without user defined labels
// Signal DO[1] is a light switch signal that is controlled by setting User labels 'OFF' and 'ON'.
ctrl.arrDO[0].M_Configure(CONVERSION_IODEV_NONE,    '0',    '1');
ctrl.arrDO[1].M_Configure(CONVERSION_IODEV_DIGITAL, 'OFF',  'ON');


//
// Analog signals
//
// Analog inputs (defined in stat structure T_IODEV_STAT_CUSTOM)
stat.arrAI[0].M_Configure(CONVERSION_IODEV_NONE,        0.0,   0.0, 0.0);
stat.arrAI[1].M_Configure(CONVERSION_IODEV_LINEAR,      2.0,   1.0, 0.0);
stat.arrAI[2].M_Configure(CONVERSION_IODEV_QUADRATIC,   2.0,   1.0, 5.0);
// Edwards pressure guage
// P =10^(1.5*V - 12)     [mbar]
// 16-bit A/D converter, +=10V, e.g. EL3102.
// 1V = 2^16/10/2 = 3276.8 bit
// P = 10^(1.5*V - 12) = 10^(1.5/3276.8*bit - 12) = 10^(0.000457763*bit - 12)
stat.arrAI[3].M_Configure(CONVERSION_IODEV_EXP10,   0.000457763, -12.0, 0.0);

//
// Analog outputs (defined in ctrl structure T_IODEV_CTRL_CUSTOM)
ctrl.arrAO[0].M_Configure(CONVERSION_IODEV_NONE,    0.0,   0.0, 0.0);   // No conversion
ctrl.arrAO[1].M_Configure(CONVERSION_IODEV_LINEAR,  0.1,   0.0, 0.0);   // out = User/10
```

- Extend the functionality of *FB_IODEV_BASE* in newly created FB *FB_IODEV_2_4_2_2_USER_CONFIG*.

  The code below shows all that the user has to do in order to define an FB that extends the functionality of *FB_IODEV_BASE*. The following has to be done in the FB:

  – In the declaration of FB *FB_IODEV_2_4_2_2_USER_CONFIG*, add *ctrl* and *stat* structures. They should be of type *T_IODEV_CTRL_2_2* and *T_IODEV_STAT_2_4*, respectively.

  – Set references to *ctrl* and *stat* structures.

– Set the number of existing I/O signals.
– Set the pointers to the first item of each signal array.
– Execute the code of the base (SUPER) FB.

FB_SENS_8_4 T_IODEV_CFG FB_SENS_8_4_USER_CONFIG GUI FB_IODEV_2_4_2_2_USER_CONFIG

```
1  // EXAMPLE: Customised FB_IODEV_BASE for 2 DI, 4 AI, 2 DO and 2 AO.
2  //          Overloaded M_UserConfigure() method for User Configuration of channels.
3  FUNCTION_BLOCK FB_IODEV_2_4_2_2_USER_CONFIG EXTENDS FB_IODEV_BASE
4  VAR_INPUT
5  END_VAR
6  VAR_OUTPUT
7  END_VAR
8  VAR
9      // Control parameters that EXTEND T_IODEV_CTRL
10     {attribute 'OPC.UA.DA' := '1'}
11     ctrl:   T_IODEV_CTRL_2_2;
12     // Status parameters that EXTEND T_IODEV_STAT
13     {attribute 'OPC.UA.DA' := '1'}
14     {attribute 'OPC.UA.DA.Access' := '1'}
15     stat:   T_IODEV_STAT_2_4;
16 END_VAR
```

```
3  //
4  //
5  // Set References
6  RefCtrl REF=ctrl;
7  RefStat REF=stat;
8
9  // Set number of channels for each type.
10 // Default values are set to zero.
11 // Set whatever is not zero, i.e. whatever is defined.
12 // WARNING: The values MUST match the sizes of corresponding arrays above !!!
13 //
14 cfg.nNum_DI     :=  2;  // Number of digital INPUT signals, arrDI:  ARRAY [0..1]
15 cfg.nNum_AI     :=  4;  // Number of analog INPUT signals,  arrAI:  ARRAY [0..3]
16 //cfg.nNum_DisI :=  0;  // Number of discrete INPUT signals
17 //cfg.nNum_TI   :=  0;  // Number of text INPUT signals
18
19 cfg.nNum_DO     :=  2;  // Number of digital OUTPUT signals, arrDO: ARRAY [0..1]
20 cfg.nNum_AO     :=  2;  // Number of analog OUTPUT signals,  arrAO: ARRAY [0..1]
21 //cfg.nNum_DisO :=  0;  // Number of discrete OUTPUT signals
22 //cfg.nNum_TO   :=  0;  // Number of text OUTPUT signals
23
24 //
25 // Set pointers for EXISTING arrays ONLY, e.g. arrDI, arrAI, etc.
26 //
27 pArrDI  := ADR(stat.arrDI[0]);
28 pArrAI  := ADR(stat.arrAI[0]);
29 pArrDO  := ADR(ctrl.arrDO[0]);
30 pArrAO  := ADR(ctrl.arrAO[0]);
31
32 //
33 // Execute the base class object FB_IODEV_BASE
34 //
35 SUPER^();
```

Example 2: Sensor Device without User Defined Scaling

The Function Block *FB_SENS_8_4* is an example of a sensor device for 8 digital and 4 analog input signals. The corresponding status data structure is called *T_IODEV_STAT_8_4*. Since there are no output signals, the standard control structure *T_IODEV_CTRL* is used. In this example user scaling is not used, so the scaling is supposed to be done on the WS.

The following steps have to be done:

- Extend the *T_IODEV_STAT* structure by adding arrays of input signals. The new structure is called *T_IODEV_STAT_8_4*, meaning 8 digital inputs and 4 analog inputs.

T_IODEV_STAT_8_4 | FB_IODEV_BASE.CheckForEvents [Online] | FB_IODEV_BASE.Acti...onitoring [

```
1  // EXAMPLE: Extension of T_IODEV_STAT with 8 DI and 4 AI.
2  //          Used in FB_SENS_8_4.
3  TYPE T_IODEV_STAT_8_4 EXTENDS T_IODEV_STAT :
4  STRUCT
5      //
6      // Signals, inputs.
7      //
8      arrDI:  ARRAY [0..7] OF FB_IODEV_CH_DIG_IN;     // 8 digital inputs
9      arrAI:  ARRAY [0..3] OF FB_IODEV_CH_ANLG_IN;    // 4 analog inputs
10 END_STRUCT
11 END_TYPE
12
```

- Create a Function Block called *FB_SENS_8_4* and extend the functionality of *FB_IODEV_BASE*, as shown below.

  The code below shows all that the user has to do in order to define an FB that extends the functionality of *FB_IODEV_BASE*. The following has to be done in the FB:

  - In the declaration of *FB_SENS_8_4*, add *ctrl* and *stat* structures. Since this is a pure sensor, the standard *ctrl* structure of type *T_IODEV_CTRL* is used. The *stat* structure is declared as *T_IODEV_STAT_8_4*.
  - Set references to *ctrl* and *stat* structures.
  - Set the number of existing input signals.
  - Set the pointers to the first item of each signal array.
  - Execute the code of the base (SUPER) FB.

FB_SENS_8_4

```
// EXAMPLE: FB for SENSORS with 8 DI and 4 AI
FUNCTION_BLOCK FB_SENS_8_4 EXTENDS FB_IODEV_BASE
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    // Control parameters as defined in T_IODEV_CTRL.
    // For sensors use the base control structure T_IODEV_CTRL.
    {attribute 'OPC.UA.DA' := '1'}
    ctrl:   T_IODEV_CTRL;
    // Status parameters that EXTEND T_IODEV_STAT
    {attribute 'OPC.UA.DA' := '1'}
    {attribute 'OPC.UA.DA.Access' := '1'}
    stat:   T_IODEV_STAT_8_4;
END_VAR
```

```
//
// TODO_USER
//
//
// Set References
RefCtrl REF=ctrl;
RefStat REF=stat;

// Set number of channels for each type.
// Default values are set to zero.
// Set whatever is not zero, i.e. whatever is defined.
// WARNING: The values MUST match the sizes of corresponding arrays above !!!
//
cfg.nNum_DI     :=  8;  // Number of digital INPUT signals, arrDI:  ARRAY [0..7]
cfg.nNum_AI     :=  4;  // Number of analog INPUT signals,  arrAI:  ARRAY [0..3]

//
// Set pointers for EXISTING arrays ONLY, e.g. arrDI, arrAI, etc.
//
pArrDI  := ADR(stat.arrDI[0]);
pArrAI  := ADR(stat.arrAI[0]);

//
// Execute the base class object FB_IODEV_BASE
//
SUPER^();
```

3.9.4 Signal Mapping

As an example, the figure below shows the TwinCAT view of the *FB_IODEV_2_4_2_2_USER_CONFIG* I/O variables that are available for mapping to physical signals, i.e. ports of analog and digital I/O terminals.

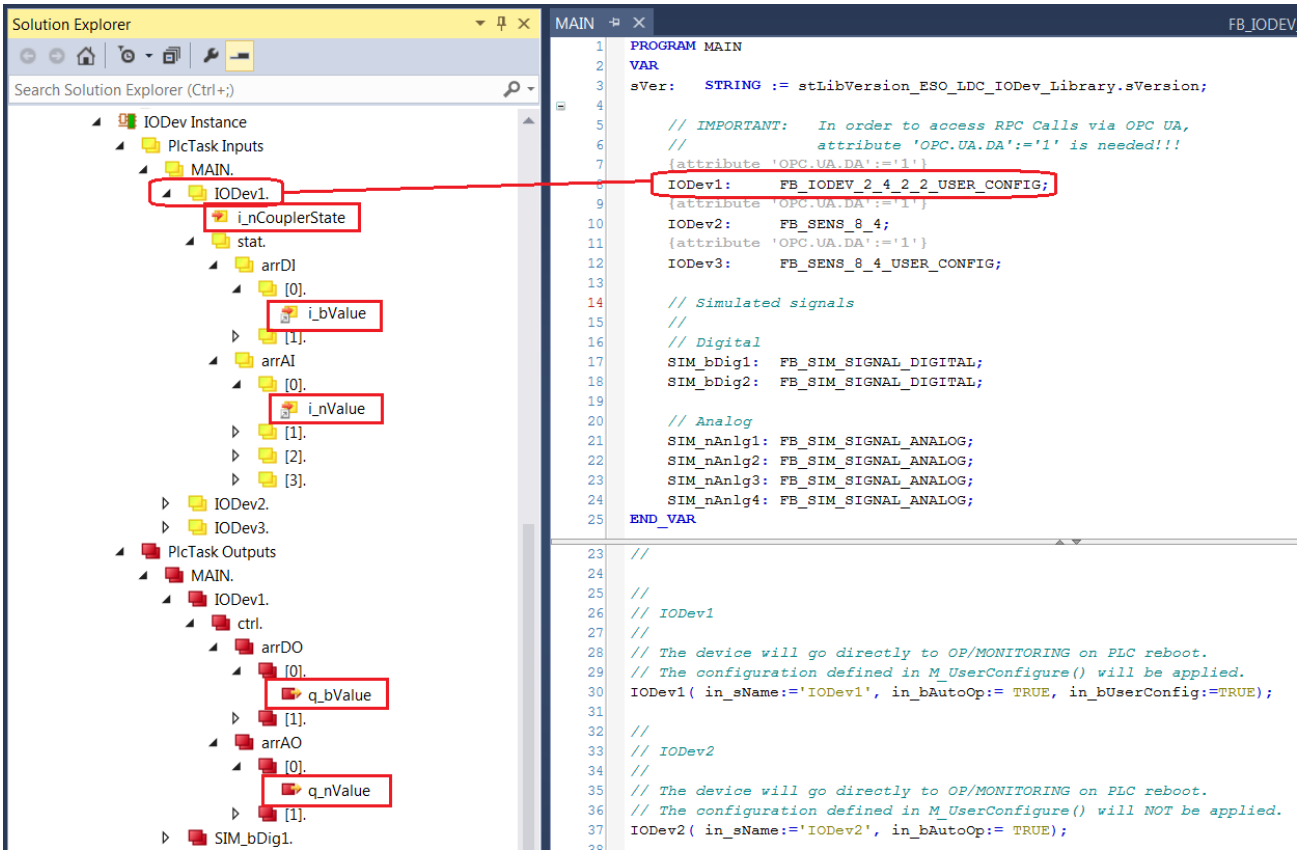


Figure 12: Example of a FB_IODEV_2_4_2_2_USER_CONFIG signals.

The table below describes each mapping variable.

Variable	Port Type	Optional	Description
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts a shutter signal.**
arrDI[i].i_bValue	**BOOL**	**Yes**	**Array of digital input signals**
arrAI[i].i_nValue	**INT**	**Yes**	**Array of analog input signals**
arrDO[i].q_bValue	**BOOL**	**Yes**	**Array of digital output signals**
arrAO[i].q_nValue	**INT**	**Yes**	**Array of analog output signals**

3.9.5 GUI Template

The *IODev* Library provides a template GUI *GUI_TEMPLATE_IODEV* for the *FB_IODEV_BASE* FB. The GUI can be also used for the Function Blocks that extend the *FB_IODEV_BASE* functionality. Applications can easily deploy an instance of this GUI by setting the GUI references to the particular instance of the FB, as shown below.

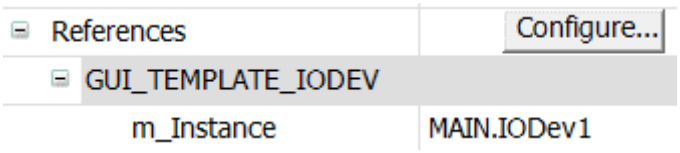


Fig. 3.10: Instantiation of GUI_TEMPLATE_IODEV for IODev1

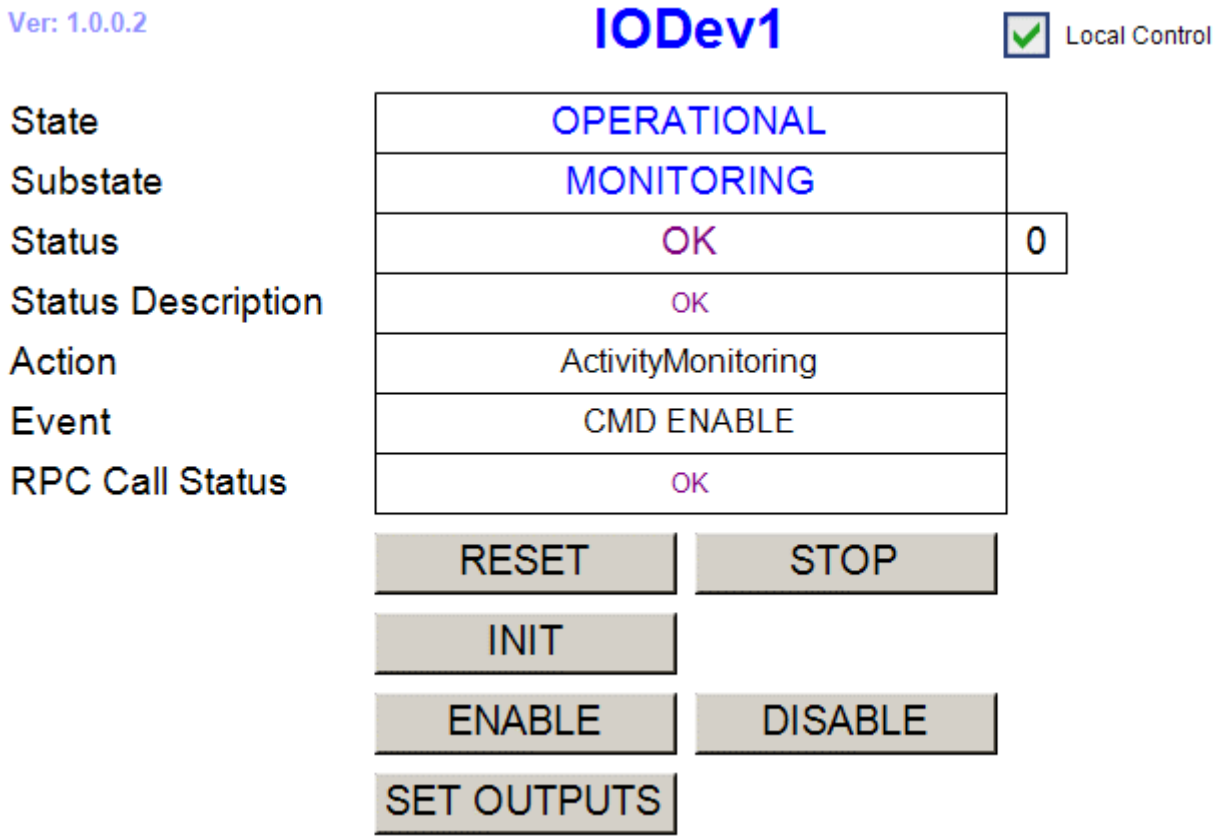


Fig. 3.11: FB_IODEV_BASE HMI for Local Control.

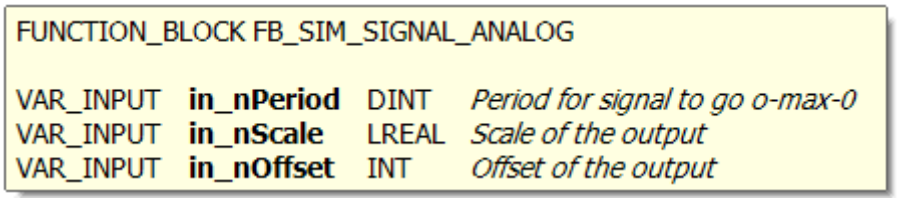
3.9.6 IODev specific RPC Methods

- RPC_SetOutputs() Activate *IODev* outputs (not used with pure sensors)

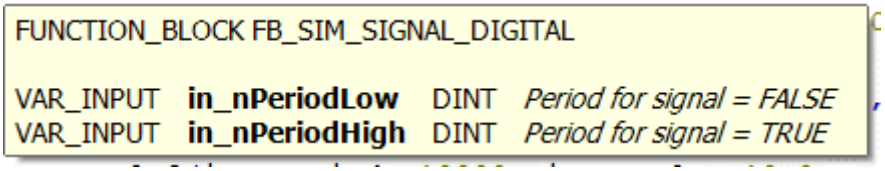
3.9.7 Signal Simulators

The *ioDev.library* provides individual analog and digital signal simulators. This means that if the user wants to simulate all signals of a sensor with eight digital and four analog signals, for example, he will have to instantiate eight digital and four analog signal simulators. However, for testing purposes it might be needed to simulate just a few signals. Time parameters are given in milliseconds.

The function blocks *FB_SIM_SIGNAL_ANALOG* and *FB_SIM_SIGNAL_REAL* implement simulated analog sinusoidal signals of type INT and REAL, respectively. The input parameters are the same for both FBs:



The function block *FB_SIM_SIGNAL_DIGITAL* implements a simulated digital signal. The input parameters are:



The following source code example is the Program MAIN that is delivered with the library. The code shows how to configure sensors as well as simulators whose outputs could be mapped to sensor input variables.

Declaration:

```
// IMPORTANT: In order to access RPC Calls via OPC UA,
//            attribute 'OPC.UA.DA':='1' is needed!!!
{attribute 'OPC.UA.DA':='1'}
IODev1:     FB_IODEV_2_4_2_2_USER_CONFIG;
{attribute 'OPC.UA.DA':='1'}
IODev2:     FB_SENS_8_4;
{attribute 'OPC.UA.DA':='1'}
IODev3:     FB_SENS_8_4_USER_CONFIG;
{attribute 'OPC.UA.DA':='1'}
IODev4:     FB_SENS_2_4A_2R_USER_CONFIG;

// Simulated signals
//
// Digital
SIM_bDig1:  FB_SIM_SIGNAL_DIGITAL;
SIM_bDig2:  FB_SIM_SIGNAL_DIGITAL;

// Analog
SIM_nAnlg1: FB_SIM_SIGNAL_ANALOG;
SIM_nAnlg2: FB_SIM_SIGNAL_ANALOG;
SIM_nAnlg3: FB_SIM_SIGNAL_ANALOG;
SIM_nAnlg4: FB_SIM_SIGNAL_ANALOG;

SIM_rAnlg1: FB_SIM_SIGNAL_REAL;
SIM_rAnlg2: FB_SIM_SIGNAL_REAL;
```

Execution:

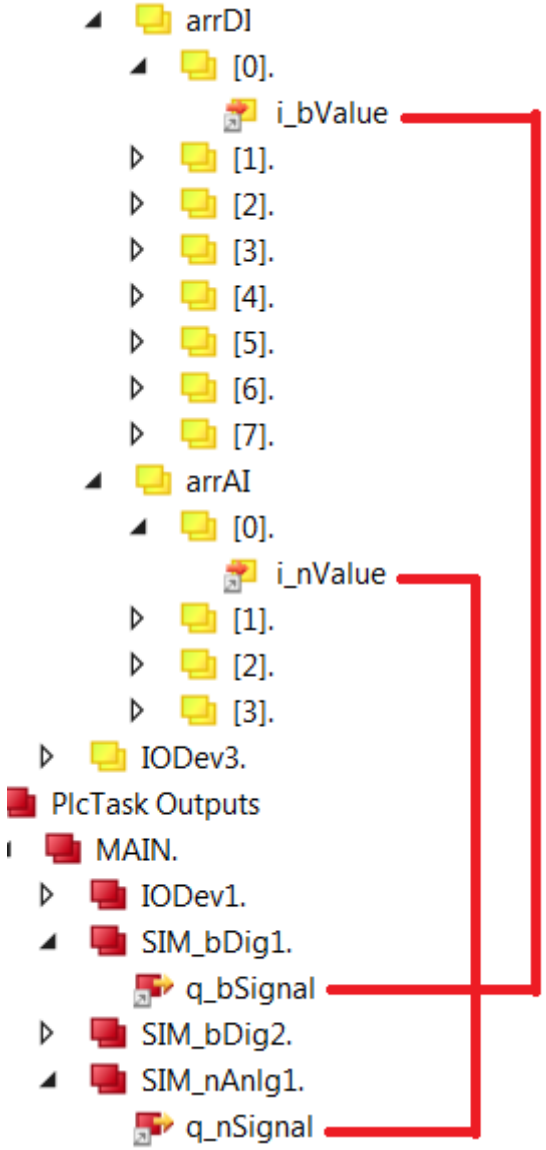
```
//
// Simulated signals
//
SIM_bDig1(in_nPeriodLow:=5000, in_nPeriodHigh:=2000);
SIM_bDig2(in_nPeriodLow:=2000, in_nPeriodHigh:=4000);

SIM_nAnlg1(in_nPeriod:=10000, in_nScale:=10000.0, in_nOffset:=0);
SIM_nAnlg2(in_nPeriod:=10000, in_nScale:=100.0,   in_nOffset:=5000);
SIM_nAnlg3(in_nPeriod:=10000, in_nScale:=10.0,    in_nOffset:=1000);

// Simulation of Edwards pressure guage
// Vary input from 4.5 to 5.0 V
// 16-bit A/D converter, +=10V, e.g. EL3102.
// 1V = 2^16/10/2 = 3276.8 bit
// 4.5 to 5.0 V ==> 4.5*3276.8 to 5.0*3276.8 bit ==> 14745 to 16384
// in_nScale = (16384 - 14745) / 2 = 819.5
// in_nOffset = 14745 + 819.5 = 15564.5
SIM_nAnlg4(in_nPeriod:=30000, in_nScale:=819.5, in_nOffset:=15564);


SIM_rAnlg1(in_nPeriod:=20000, in_nScale:=1000.0, in_nOffset:=1000.0);
SIM_rAnlg2(in_nPeriod:=10000, in_nScale:=100.0,  in_nOffset:=0.0);


//
// Real devices
//

//
// IODev1
//
// The device will go directly to OP/MONITORING on PLC reboot.
// The configuration defined in M_UserConfigure() will be applied.
IODev1(in_sName:='IODev1', in_bAutoOp:= TRUE, in_bUserConfig:=TRUE);

//
// IODev2
//
// The device will go directly to OP/MONITORING on PLC reboot.
// The configuration defined in M_UserConfigure() will NOT be applied.
IODev2(in_sName:='IODev2', in_bAutoOp:= TRUE);

//
// IODev3
//
// The device will go directly to OP/MONITORING on PLC reboot.
// The configuration defined in M_UserConfigure() will be applied.
IODev3(in_sName:='IODev3', in_bAutoOp:= TRUE, in_bUserConfig:=TRUE);

//
```

(continues on next page)

(continued from previous page)

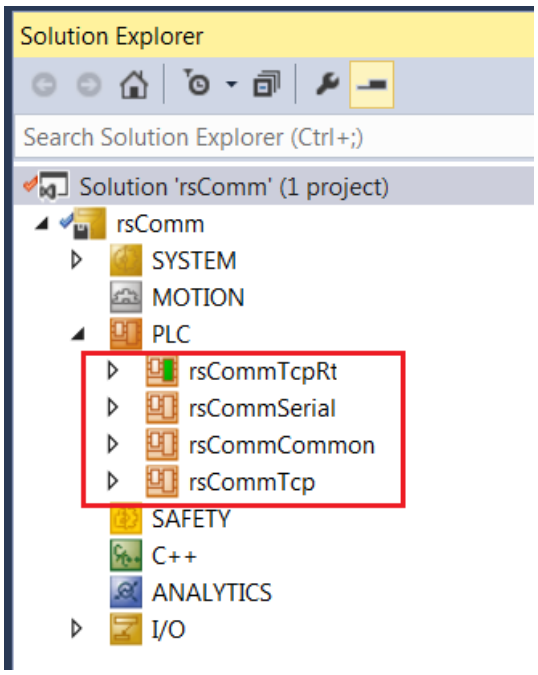
```
// IODev4
//
// The device will go directly to OP/MONITORING on PLC reboot.
// The configuration defined in M_UserConfigure() will be applied.
IODev4(in_sName:='IODev4', in_bAutoOp:= TRUE, in_bUserConfig:=TRUE);
```

Simulator Mapping

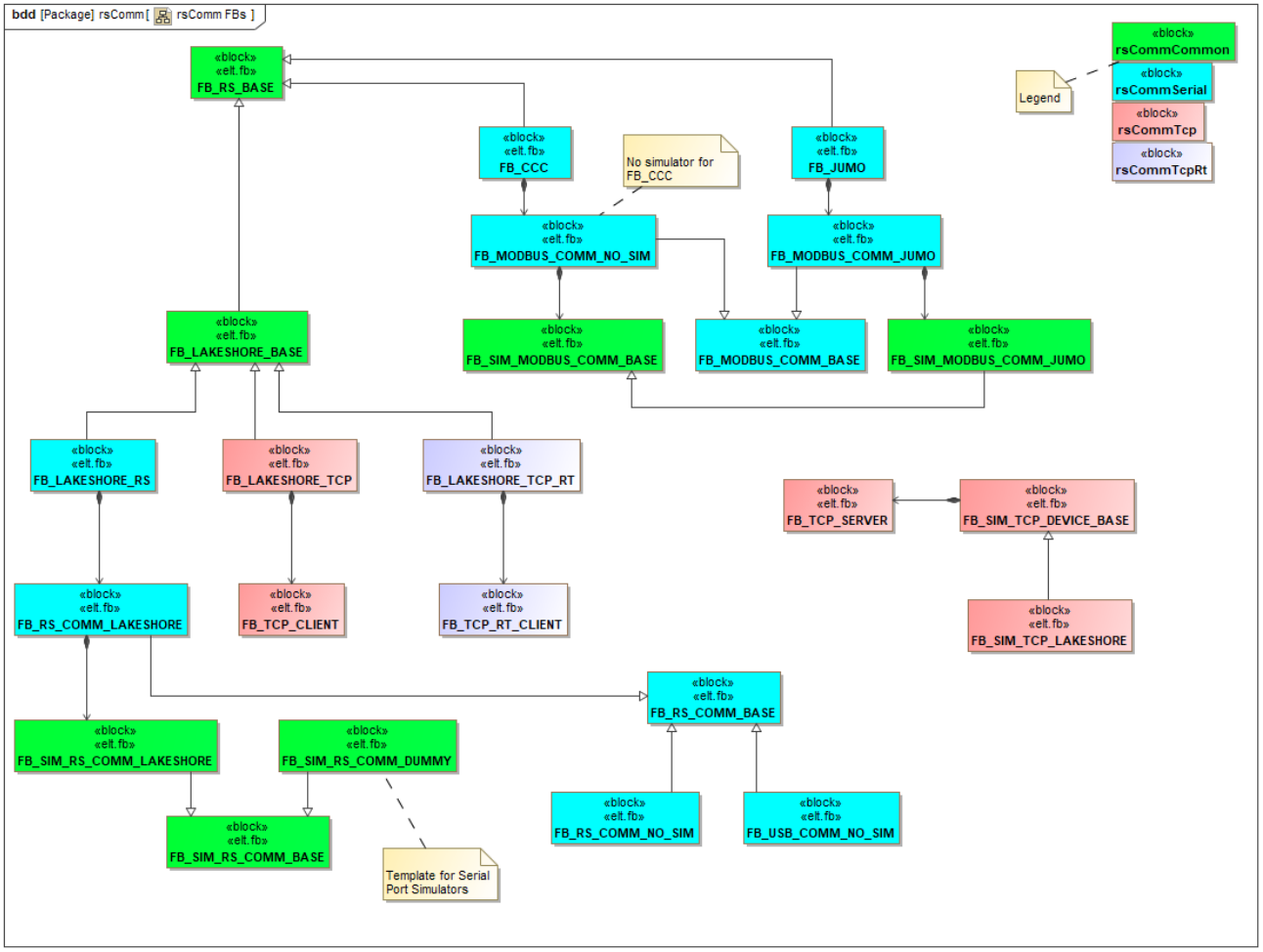


3.10 Communication Libraries (rsComm*.library)

Module *rsComm* contains PLC libraries for communication via serial port, Ethernet Tcp and Ethernet Tcp RT (Ethernet Tcp Real-Time). The libraries are delivered under separate PLC projects as part of the *rsComm* TwinCAT solution. The reason for this is that each communication protocol requires a separate license, so the functionality is provided per license. The four PLC projects are shown in the figure below.



The overview of all Function Blocks is shown in the figure below. The FBs are color-coded in order to indicate the corresponding PLC project they belong to.

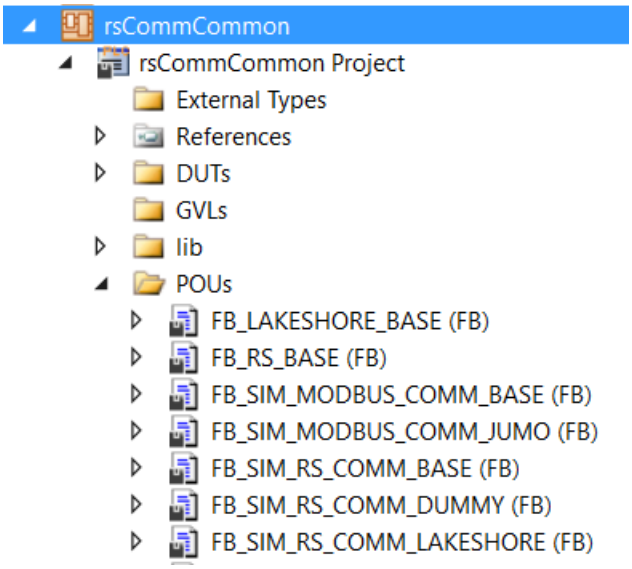


The following table gives the overview of the libraries.

Library	Description
rsCommCommon	**Common library that has to be included in every project using rsComm* libraries**
rsCommSerial	**Serial port communication and modbus RTU**
rsCommTcp	**TCP/IP communication via EtherCAT switch ports EL6601 and EL6614**
rsCommTcpRt	**TCP/IP Real-Time communication via CX2500-0060**

3.10.1 rsCommCommon.library

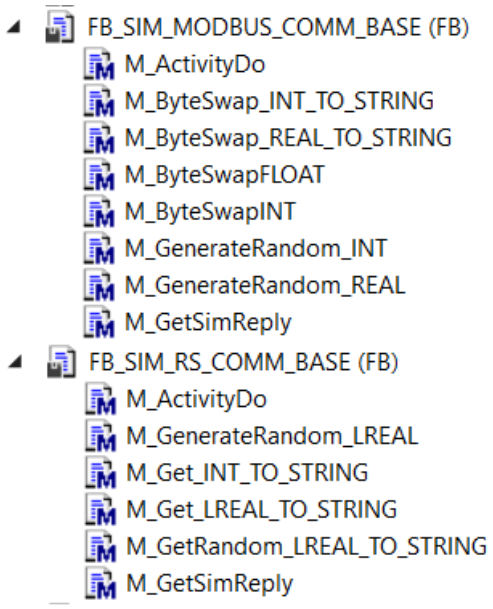
This is the common library that has to be included under *References* in all projects that use other *rsComm** libraries. This library does not require any license. The Function Block FB_RS_BASE provides the common state machine for all controllers. All *rsComm* PLC controllers, Serial, Tcp and TcpRt, extend the functionality of this FB.



The overview of FBs is given in the following table.

Function Block	Description
FB_RS_BASE	**Base FB for all rsComm controllers.**
FB_LAKESHORE_BASE	**Base FB for all Lakeshore controllers, regardless of the comm interface.**
FB_SIM_MODBUS_COMM_BASE	**Base FB for MODBUS simulation.**
FB_SIM_MODBUS_COMM_JUMO	**JUMO MODBUS simulator.**
FB_SIM_RS_COMM_BASE	**Base FB for RS-232/422/485 simulation.**
FB_SIM_RS_COMM_DUMMY	**Example of RS comm simulator implementation.**
FB_SIM_RS_COMM_LAKESHORE	**RS based comm simulator for Lakeshores with serial port, i.e. models 218 & 340.**

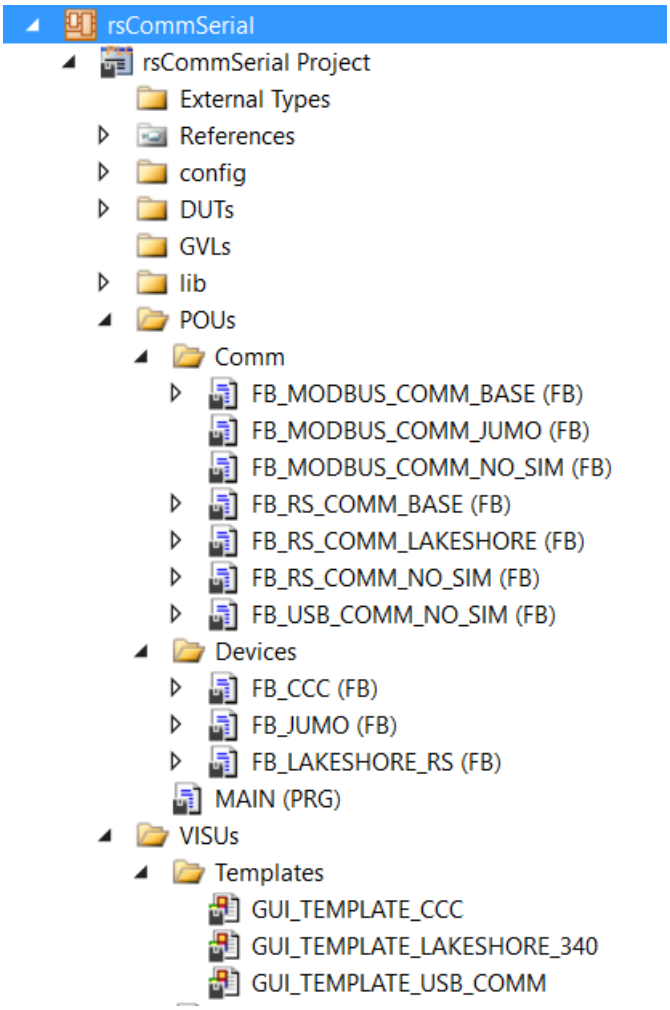
There are two base FBs for RS comm simulation, FB_SIM_MODBUS_COMM_BASE and FB_SIM_RS_COMM_BASE. Their main purpose is to provide methods that generate simulated replies that include the reply terminators as well. The replies could have fixed values as well as simulated variations. The methods are shown on the following figure.



There are two methods, *M_GetSimReply()* and *M_ActivityDo()*, that should be customised for a particular simulator, e.g. for a Lakeshore 340 simulator. *M_GetSimReply()* should construct a reply based on the command. *M_ActivityDo()* might be used to provide more realistic behaviour, e.g. to simulate a Lakeshore warm-up ramp. For more info, please see method *M_GetSimReply()* of FB_SIM_RS_COMM_LAKESHORE.

3.10.2 rsCommSerial.library

This library handles communication via serial line, e.g. via EL6001. In addition to the FBs handling ASCII serial (RS-232/422/485) and modbus RTU communication protocols, it provides controllers for the standard equipment used at ESO that have the serial port interface, like the Lakeshore 340 temperature controller or the ESO Cabinet Cooling Controller.

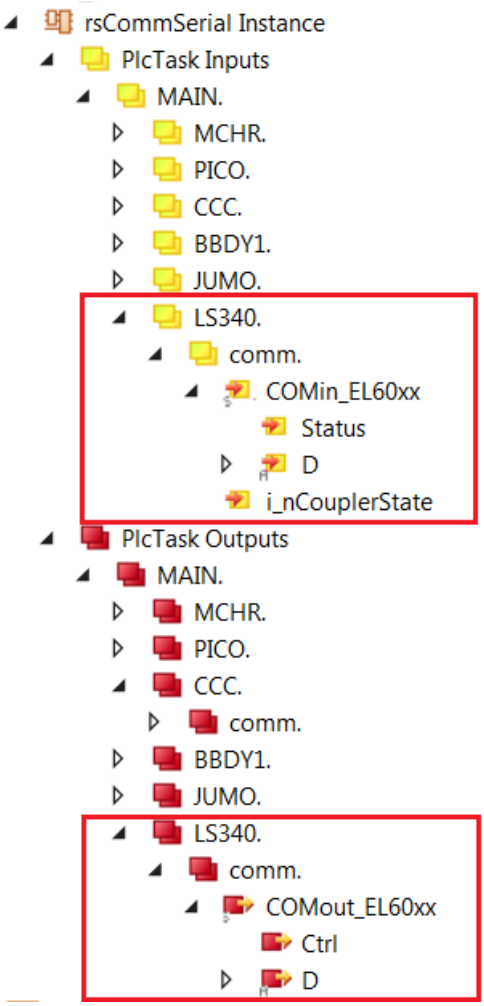


The overview of FBs is given in the table below.

Function Block	Description
FB_MODBUS_COMM_BASE	**Base FB for generic modbus RTU communication driver via serial port.**
FB_MODBUS_COMM_JUMO	**Modbus RTU communication driver customised for JUMO simulation.**
FB_MODBUS_COMM_NO_SIM	**Modbus RTU communication driver without any simulator**
FB_RS_COMM_BASE	**Base FB for generic RS-232/422/485 communication driver.**
FB_RS_COMM_LAKESHORE	**RS-232/422/485 communication driver customised for Lakeshore simulation.**
FB_RS_COMM_NO_SIM	**Generic RS-232/422/485 communication driver without a simulator.**
FB_USB_COMM_NO_SIM	**Generic USB communication driver without a simulator.**
FB_CCC	**Driver for ESO standard Cabinet Cooling Controller with RS-232 interface.**
FB_JUMO	**JUMO driver with modbus RTU interface.**
FB_LAKESHORE_RS	**Lakeshore driver for devices with serial port, i.e. models 218 & 340.**

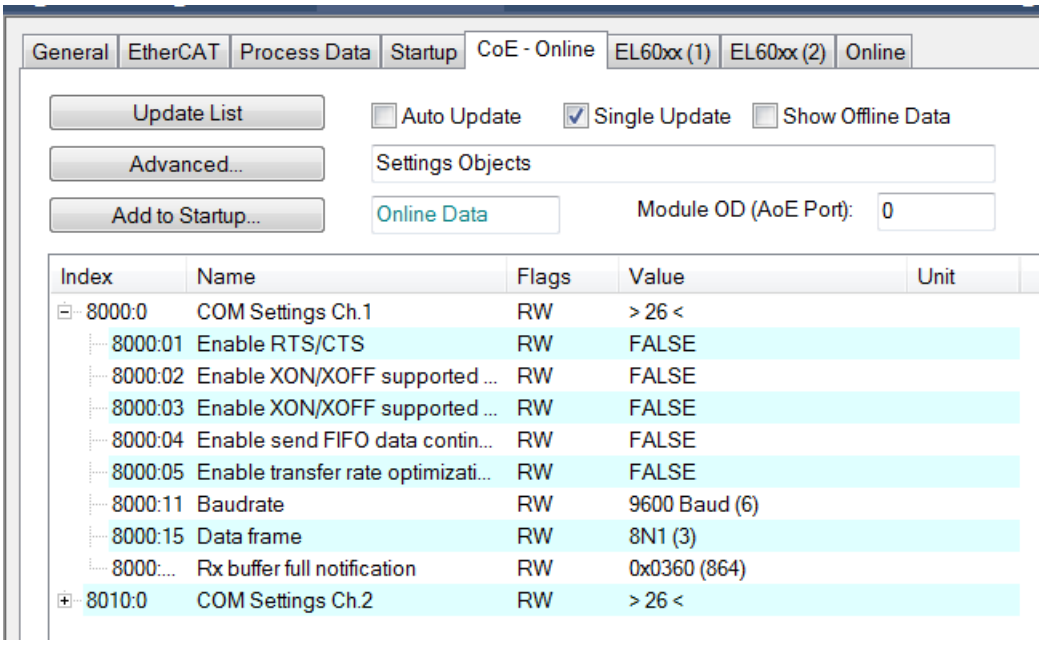
Signal Mapping

Apart from the common *i_nCouplerState* input parameter that has to be mapped to any of the State variables on the PLC, the mapping has to be done for input and output parameters of the serial port.



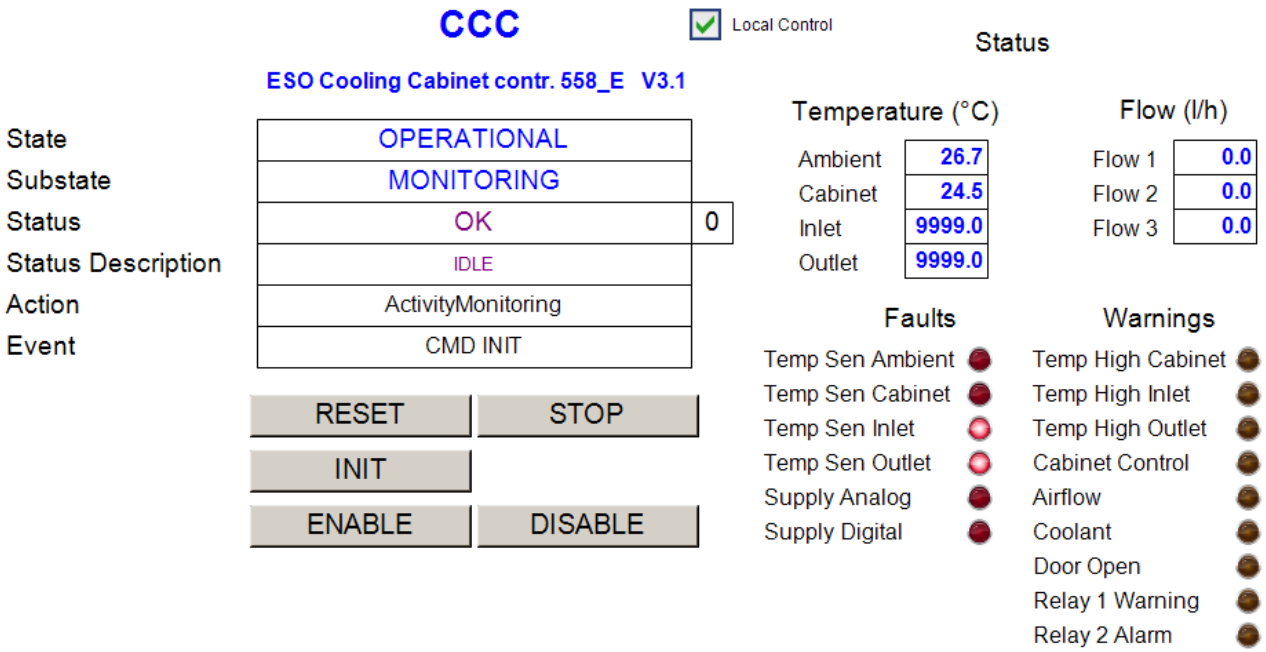
The serial port communication parameters, like baud rate, etc, have to be configured via the CoE interface.

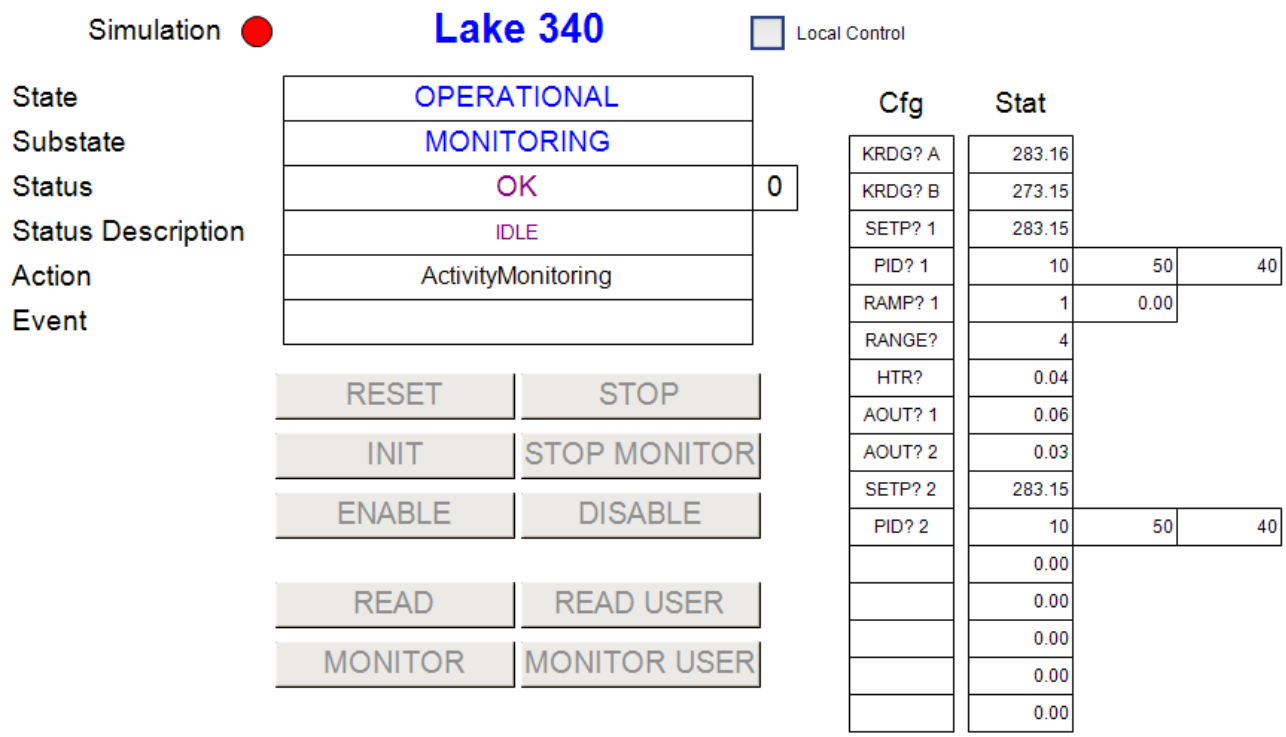
The default CoE configuration for the Lakeshore 340 is given below.



GUI Templates

GUI templates for the ESO Cabinet Cooling Controller and the Lakeshore 340 temperature controller are provided.





Example code

The following code handles an ESO Cabinet Cooling Controller, a JUMO controller and a Lakeshore 340.

```
PROGRAM MAIN
VAR
        // ESO Cabinet Cooling Controller (RS-232 interface)
        {attribute 'OPC.UA.DA' := '1'}
        CCC:        FB_CCC;

        // JUMO Controller (modnus RTU via Serial)
        {attribute 'OPC.UA.DA' := '1'}
        JUMO:       FB_JUMO;

        // Lakeshore 340 (RS-232 interface)
        {attribute 'OPC.UA.DA' := '1'}
        LS340:      FB_LAKESHORE_RS;

END_VAR
```

```
CCC(in_sName:='CCC', in_nPeriod := 1200);

JUMO(in_sName:='JUMO',in_nPeriod:=1000);    // Read all JUMOs every second
```

(continues on next page)

(continued from previous page)

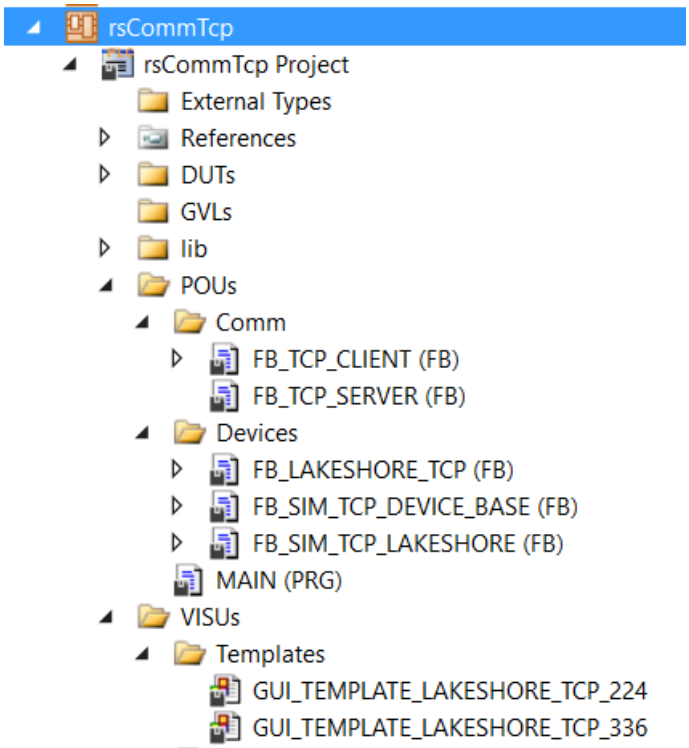
```
LS340(
    in_bSimulation      := FALSE,
    in_sName            := 'Lake 340',
    in_nModel           := 340,
    in_bAutoMonitor     := TRUE,
    in_sCmdSuffix         := '$0D$0A',
    in_sReplySuffix       := '$0D$0A',
    in_nPeriod          := 1700);
```

3.10.3 rsCommTcp.library

Note: Usage of the rsCommTcp.library in combination with the EtherCAT switch ports EL6601 and EL6614 is the preferred option over the usage of the CX2500-0060 Realtime network adapter (see rsCommTcpRt.library).

This library handles the TCP/IP communication via EtherCAT switch ports EL6601 and EL6614. It can be used to communicate with devices equipment with the Ethernet port, e.g. the Lakeshore 336 temperature controller. It requires the TF6310 TCP/IP license. In addition, the TF6310 function Windows driver has to be installed on the PLC. The IP address of the switch has to match the subnet of the device connected to it. For more info about installation and licensing consult Beckhoff documentation.

The library provides generic TCP/IP drivers for TCP clients (FB_TCP_CLIENT) and servers (FB_TCP_SERVER). Device PLC controllers are based on FB_TCP_CLIENT, while FB_TCP_SERVER is used only for simulators.

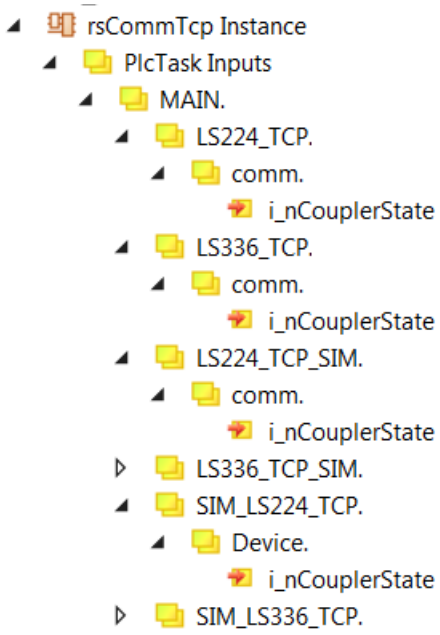


The overview of FBs is given in the table below.

Function Block	Description
FB_TCP_CLIENT	**Generic TCP/IP Client driver. This is the base FB for Tcp based controllers.**
FB_TCP_SERVER	**Generic TCP/IP Server driver. This is the base FB for the communication part of Tcp based simulators.**
FB_LAKESHORE_TCP	**Lakeshore driver for devices with Ethernet port, i.e. models 224 & 336.**
FB_SIM_TCP_DEVICE_BASE	**Generic TCP/IP device simulator that uses FB_TCP_SERVER for communication.**
FB_SIM_TCP_LAKESHORE	**TCP/IP device simulator for Lakeshore models 224 & 336.**

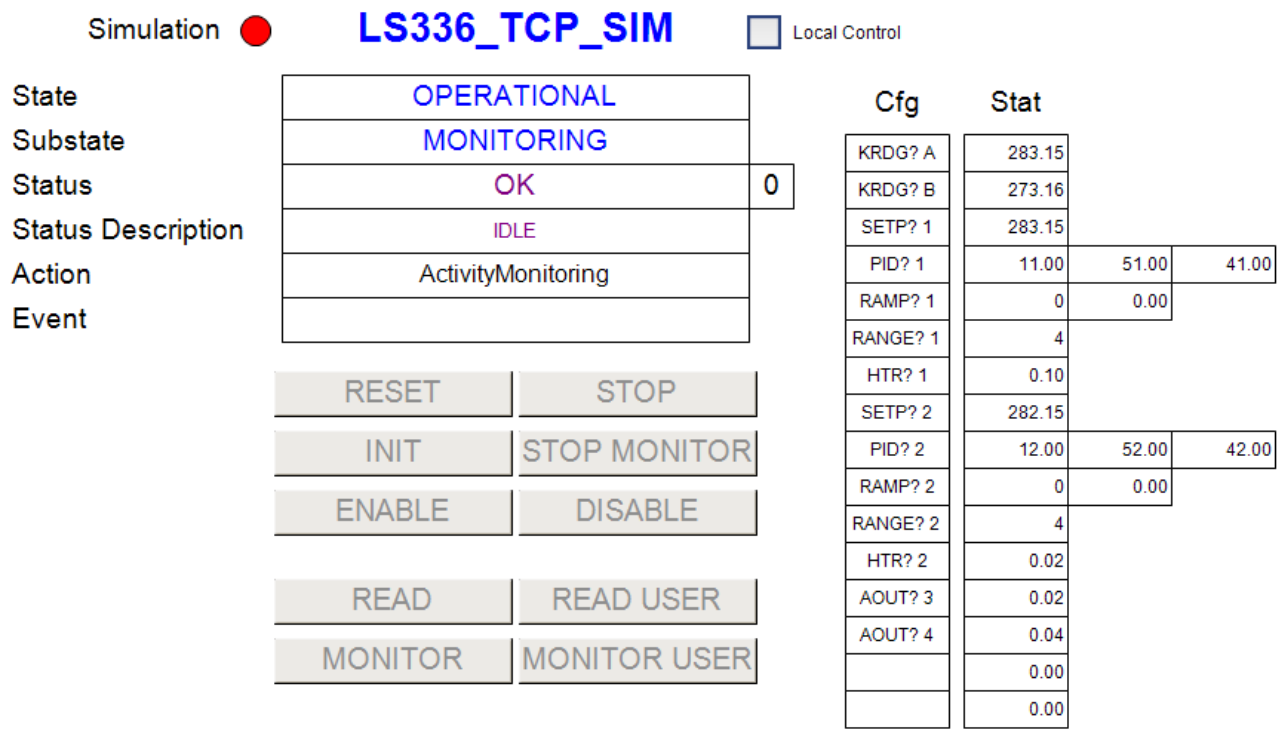
Signal Mapping

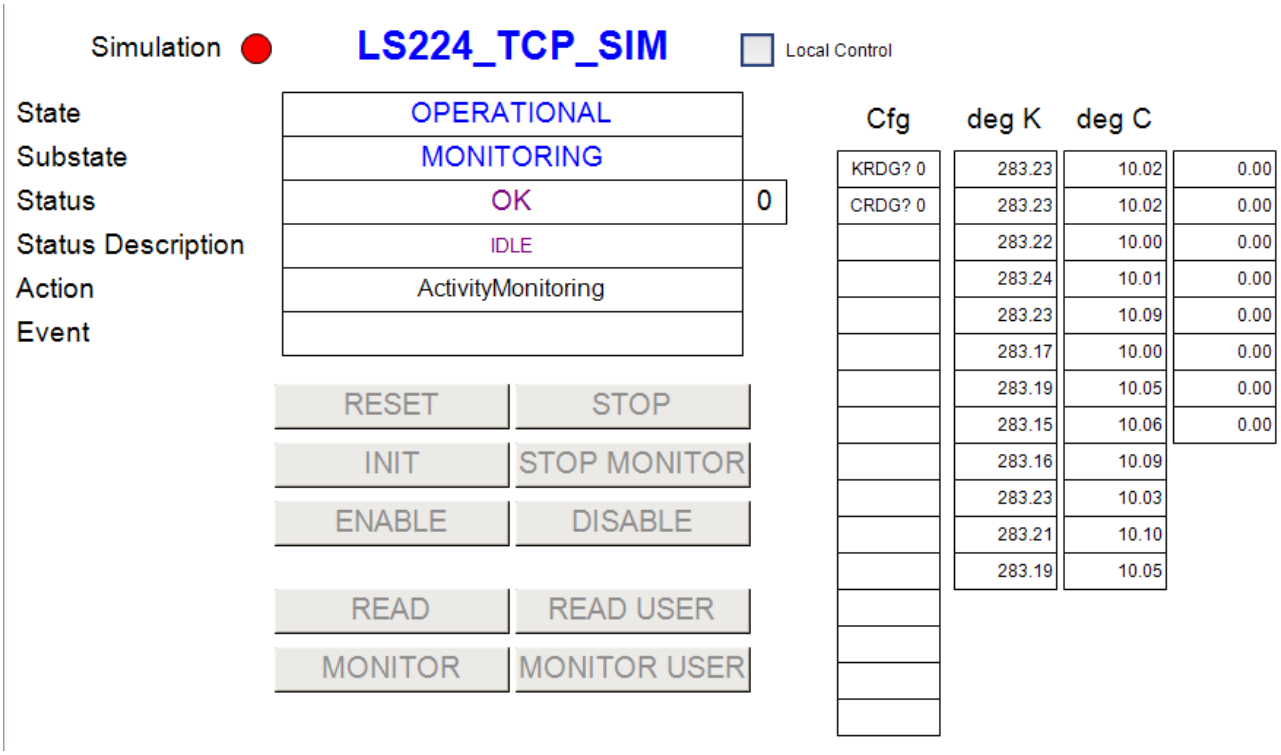
Devices and simulators have only a single input parameter (*i_nCouplerState*) to be mapped to any of the State variables on the PLC.



GUI Templates

GUI templates for the Lakeshore 336 and 224 temperature controller/monitor are provided.


Simulation
LS336_TCP_SIM
Local Control
State
Substate
Status
Status Description
Action
Event
OPERATIONAL
MONITORING
OK
0
IDLE
ActivityMonitoring
RESET
STOP
INIT
STOP MONITOR
ENABLE
DISABLE
READ
READ USER
MONITOR
MONITOR USER
Cfg
Stat
KRDG? A 283.15
KRDG? B 273.16
SETP? 1 283.15
PID? 1 11.00 51.00 41.00
RAMP? 1 0 0.00
RANGE? 1 4
HTR? 1 0.10
SETP? 2 282.15
PID? 2 12.00 52.00 42.00
RAMP? 2 0 0.00
RANGE? 2 4
HTR? 2 0.02
AOUT? 3 0.02
AOUT? 4 0.04
0.00
0.00


Simulation
LS224_TCP_SIM
Local Control
State
Substate
Status
Status Description
Action
Event
OPERATIONAL
MONITORING
OK
0
IDLE
ActivityMonitoring
RESET
STOP
INIT
STOP MONITOR
ENABLE
DISABLE
READ
READ USER
MONITOR
MONITOR USER
Cfg
deg K
deg C
KRDG? 0 283.23 10.02 0.00
CRDG? 0 283.23 10.02 0.00
283.22 10.00 0.00
283.24 10.01 0.00
283.23 10.09 0.00
283.17 10.00 0.00
283.19 10.05 0.00
283.15 10.06 0.00
283.16 10.09
283.23 10.03
283.21 10.10
283.19 10.05

Example code

The following code handles two real Lakeshore devices (224 and 336) and two additional Lakeshore devices that are connected to simulators. It is important to note that the devices that use simulators have to use the IP address '127.0.0.1'. Also, the port numbers of the device and the simulator have to match.

```
PROGRAM MAIN
VAR
    // Lakeshore 224 (Ethernet interface)
    {attribute 'OPC.UA.DA' := '1'}
    LS224_TCP:     FB_LAKESHORE_TCP;                // LakeShore based on FB_TCP_
↪CLIENT

    // Lakeshore 336 (Ethernet interface)
    {attribute 'OPC.UA.DA' := '1'}
    LS336_TCP:     FB_LAKESHORE_TCP;                // LakeShore based on FB_TCP_
↪CLIENT


    // Lakeshores to be simulated
    {attribute 'OPC.UA.DA' := '1'}
    LS224_TCP_SIM: FB_LAKESHORE_TCP;                // LakeShore based on FB_TCP_
↪CLIENT
    {attribute 'OPC.UA.DA' := '1'}
    LS336_TCP_SIM: FB_LAKESHORE_TCP;                // LakeShore based on FB_TCP_
↪CLIENT

    // Simulators
    SIM_LS224_TCP: FB_SIM_TCP_LAKESHORE;            // LakeShore SIM based on FB_
↪TCP_SERVER
    SIM_LS336_TCP: FB_SIM_TCP_LAKESHORE;            // LakeShore SIM based on FB_
↪TCP_SERVER


END_VAR
```

```
// Lakeshore 224
LS224_TCP(
    in_sName            := 'LS224_TCP',
    in_nModel           := 224,
    in_nPeriod          := 1500,
    in_sCmdSuffix         := '$0D$0A',
    in_sReplySuffix       := '$0D$0A',
    in_sDeviceTcpIpAdr  := '192.168.0.12',
    in_nDeviceTcpPort   := 7777);

// Lakeshore 336
LS336_TCP(
```

(continues on next page)

(continued from previous page)

```
    in_sName            := 'LS336_TCP',
    in_nModel           := 336,
    in_nPeriod          := 1000,
    in_sCmdSuffix         := '$0A',
    in_sReplySuffix       := '$0A',
    in_sDeviceTcpIpAdr  := '192.168.0.80',
    in_nDeviceTcpPort   := 7777);

// Dummy Lakeshore 224
// in_bSimulation:= TRUE means that it communicates with a simulator.
// If the simulator is running on the same PLC, in_sDeviceTcpIpAdr:='127.0.0.1'.
// in_nDeviceTcpPort has to match simulator's in_nTcpPort.
// See SIM_LS224_TCP.
LS224_TCP_SIM(
    in_bSimulation      := TRUE,
    in_sName            := 'LS224_TCP_SIM',
    in_nModel           := 224,
    in_nPeriod          := 1500,
    in_sCmdSuffix         := '$0D$0A',
    in_sReplySuffix       := '$0D$0A',
    in_sDeviceTcpIpAdr  := '127.0.0.1',
    in_nDeviceTcpPort   := 8888);

// Dummy Lakeshore 336
// in_bSimulation:= TRUE means that it communicates with a simulator.
// If the simulator is running on the same PLC, in_sDeviceTcpIpAdr:='127.0.0.1'.
// in_nDeviceTcpPort has to match simulator's in_nTcpPort.
// See SIM_LS336_TCP.
LS336_TCP_SIM(
    in_bSimulation      := TRUE,
    in_sName            := 'LS336_TCP_SIM',
    in_nModel           := 336,
    in_nPeriod          := 1000,
    in_sCmdSuffix         := '$0A',
    in_sReplySuffix       := '$0A',
    in_sDeviceTcpIpAdr  := '127.0.0.1',
    in_nDeviceTcpPort   := 7777);


// Lakeshore simulators
SIM_LS224_TCP(
    in_bSimulation      := TRUE,
    in_bEnable          := TRUE,
    in_nModel           := 224,
    in_sReplySuffix       := '$0D$0A',
    in_sTcpIpAdr        := '127.0.0.1',
    in_nTcpPort         := 8888);

SIM_LS336_TCP(
```

(continues on next page)

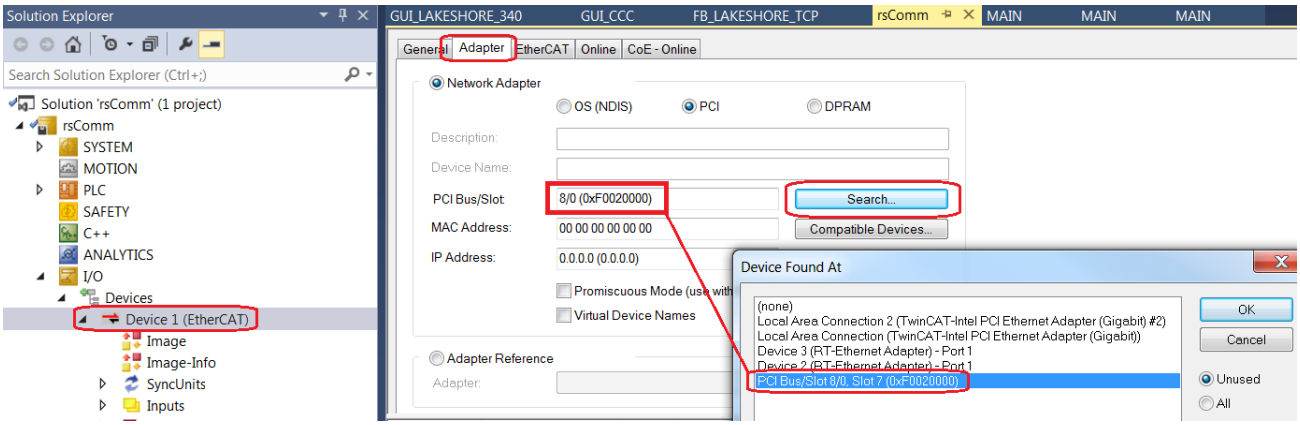
(continued from previous page)

```
    in_bSimulation      := TRUE,
    in_bEnable          := TRUE,
    in_nModel           := 336,
    in_sReplySuffix       := '$0A',
    in_sTcpIpAdr        := '127.0.0.1',
    in_nTcpPort         := 7777);
```

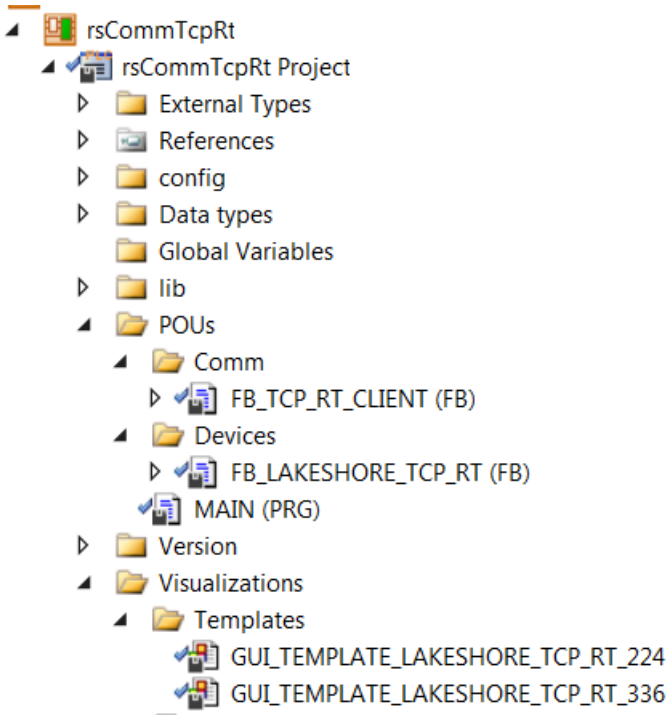
3.10.4 rsCommTcpRt.library

This library handles the TCP/UDP communication via CX2500-0060 system module that provides two independent Gbit Ethernet interfaces to the CX2000 family of PLCs, e.g. to the CX2030. The library can be used to communicate with devices equipment with the Ethernet port, e.g. the Lakeshore 336 temperature controller. It requires the TF6311 TC3 TCP/UDP Realtime license. The IP address of the switch has to match the subnet of the device connected to it. For more info about installation and licensing, consult Beckhoff documentation.

Warning: If the CX2500-0060 system module is installed, the PCI Bus address of the EtherCAT Device will be changed from the default 4/0 (0xF0020000) to 8/0 (0xF0020000), and an attempt to download an existing project will result in a failure message that is impossible to understand. For new projects the system will recognise the address automatically, while for the existing ones the new address has to be set by pressing the 'Search. . . ' button and selecting the only available PCI Bus address, as seen on the figure below.



The library provides a generic TCP/UDP RT driver for TCP RT clients (FB_TCP_RT_CLIENT) that is used in PLC controllers for the equipment with the Ethernet interface, e.g. the Lakeshore 336.

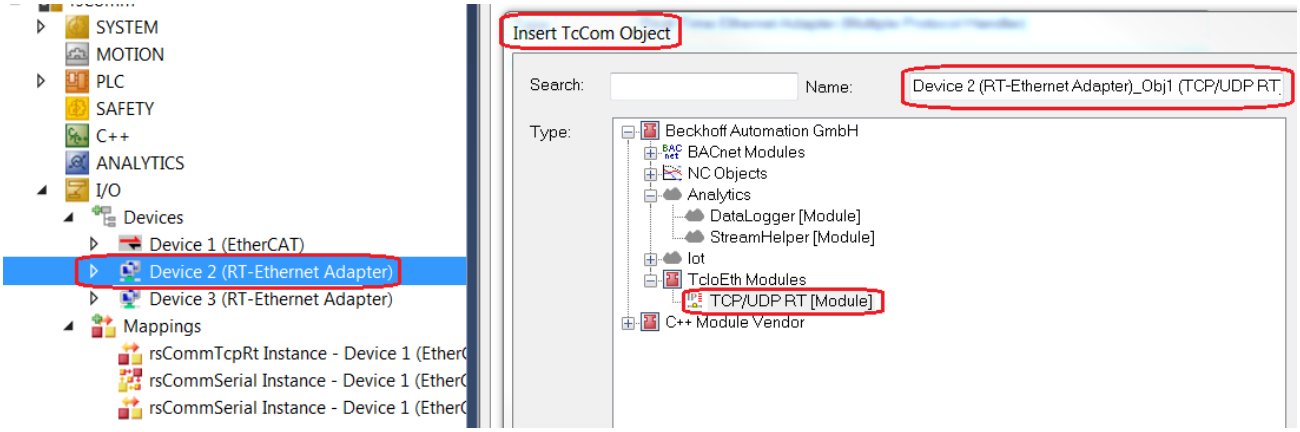


The overview of FBs is given in the table below.

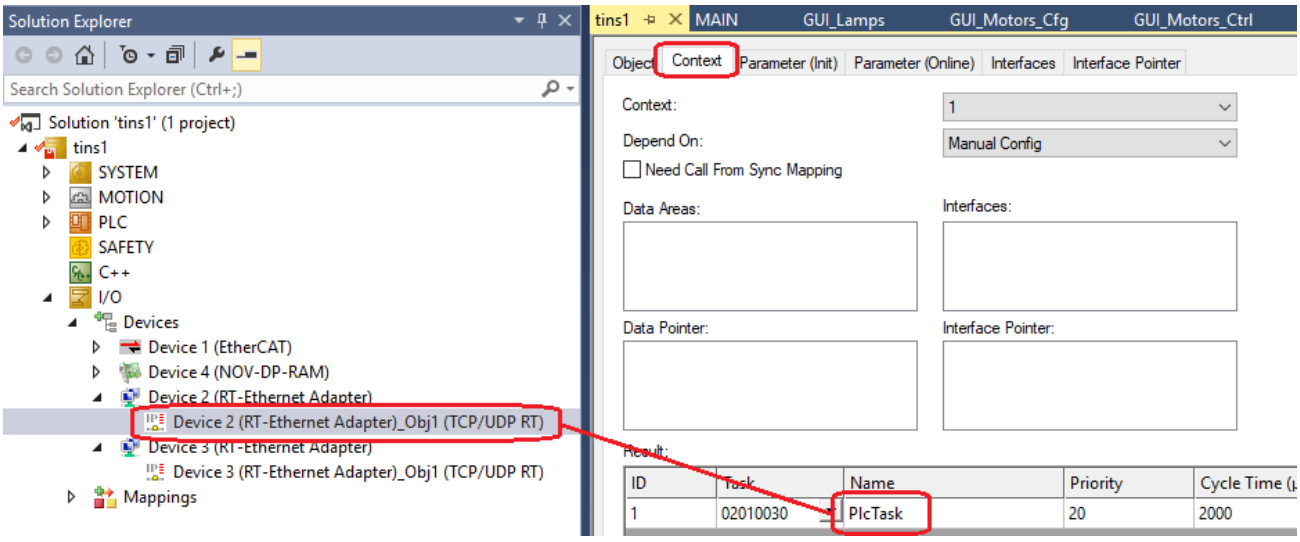
Function Block	Description
FB_TCP_RT_CLIENT	**Generic TCP/UDP RT Client driver.**
FB_LAKESHORE_TCP_RT	**Lakeshore driver for devices with Ethernet port, i.e. models 224 & 336.**

Adapter Configuration

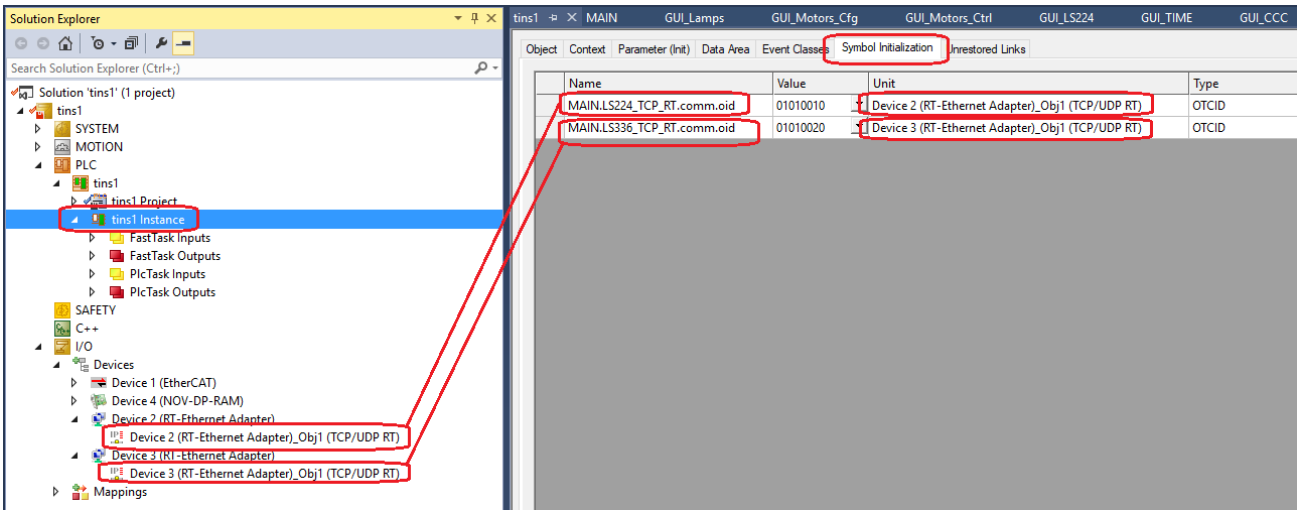
The CX2500-0060 system module is not part of the EtherCAT network and there is no 'clasical' EtherCAT I/O mapping. Each network port has to be added manually under the I/O Devices as an RT-Ethernet Adapter. Then, an Object of type TCP/UDP RT, that can be found under Beckhoff/TcIoEth Modules, has to be added to the adaptor, e.g. Device 2 (RT-Ethernet Adapter)_Obj1 (TCP/UDP RT).



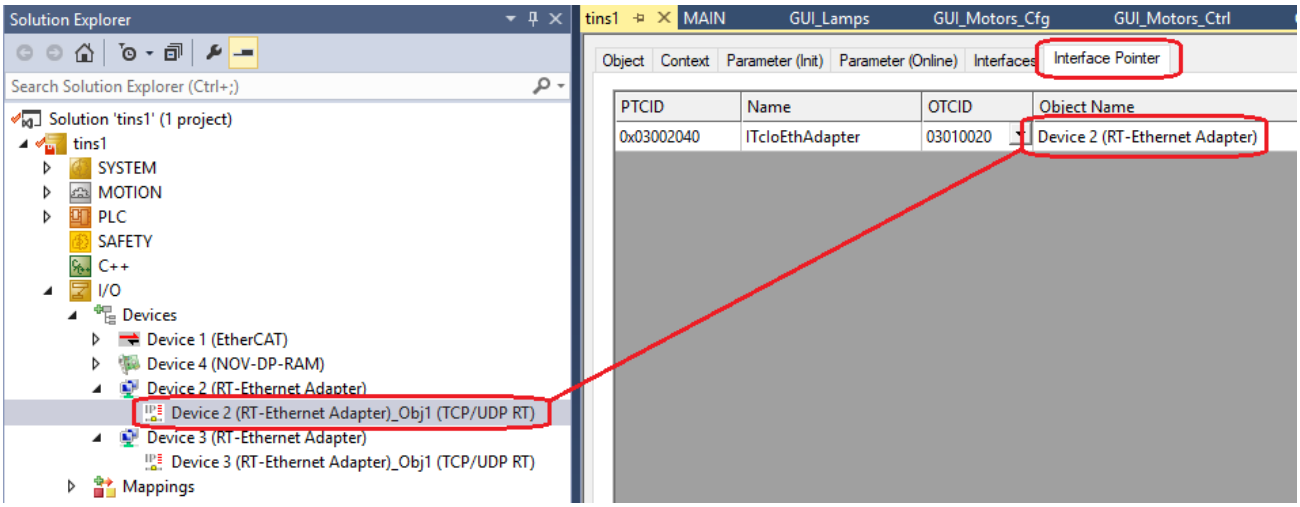
The Context of the Object is the PLC Task where an instance of the devices, e.g. MAIN.LS224_TCP_RT, will be running.



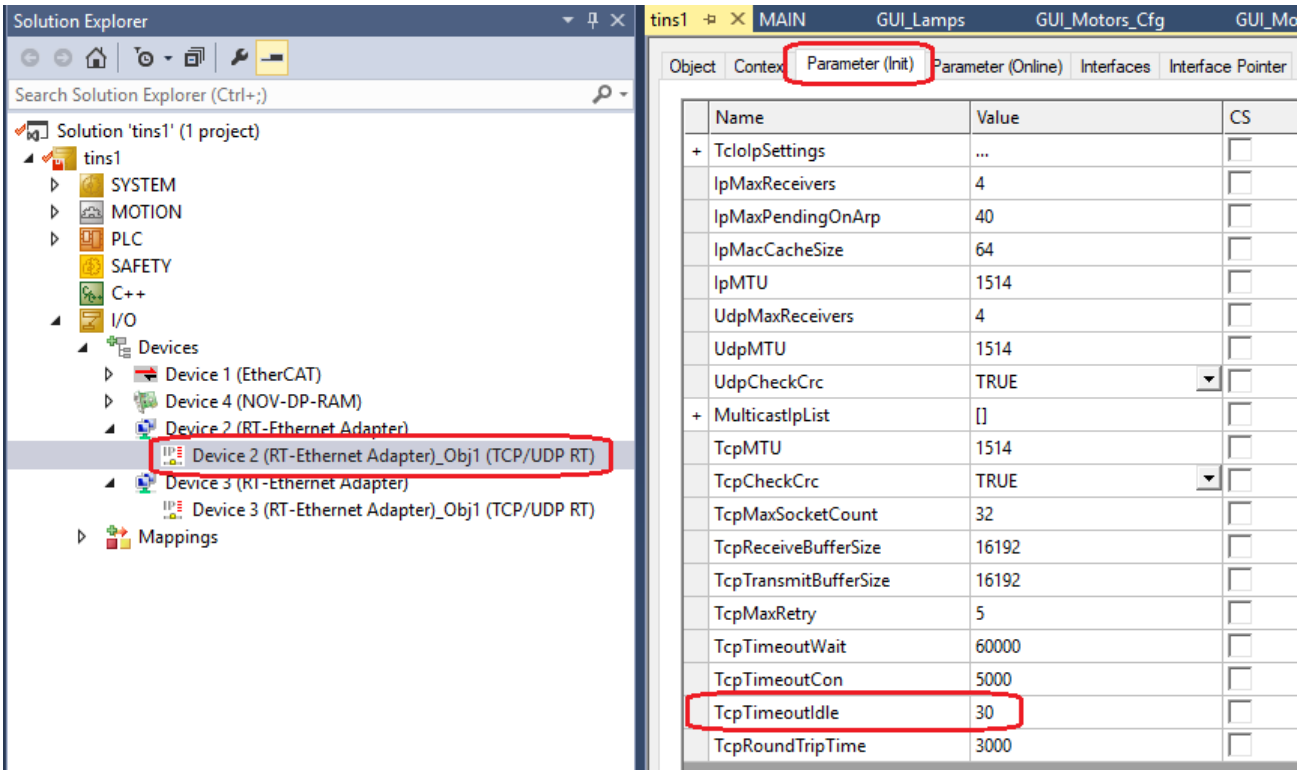
In the 'Symbol Initialisation' the Object has to be linked to the device instance.



The ‘Interface Pointer’ has to be set to the Device Object.



In the case of a Lakeshore device connected to the RT-EthernetAdapter Object, the TcpTimeoutIdle, that can be found under ‘Parameter (Init)’, has to be set to 30 ms. This is a Lakeshore specific setting.



GUI Templates

GUI templates for the Lakeshore 336 and 224 temperature controller/monitor are provided. See the corresponding section for Lakeshore TCP devices. The GUIs look the same.

Example code

The following code handles two Lakeshore devices (224 and 336).

```
PROGRAM MAIN
VAR
        //
        // Tcp RT interface functions
        //
        // Lakeshore 224 (Ethernet TCP Realtime interface)
        {attribute 'OPC.UA.DA' := '1'}
        LS224_TCP_RT:   FB_LAKESHORE_TCP_RT;        // LakeShore based on FB_TCP_
↪RT_CLIENT

        // Lakeshore 336 (Ethernet TCP Realtime interface)
        {attribute 'OPC.UA.DA' := '1'}
        LS336_TCP_RT:   FB_LAKESHORE_TCP_RT;        // LakeShore based on FB_TCP_
↪RT_CLIENT
```

(continues on next page)

(continued from previous page)

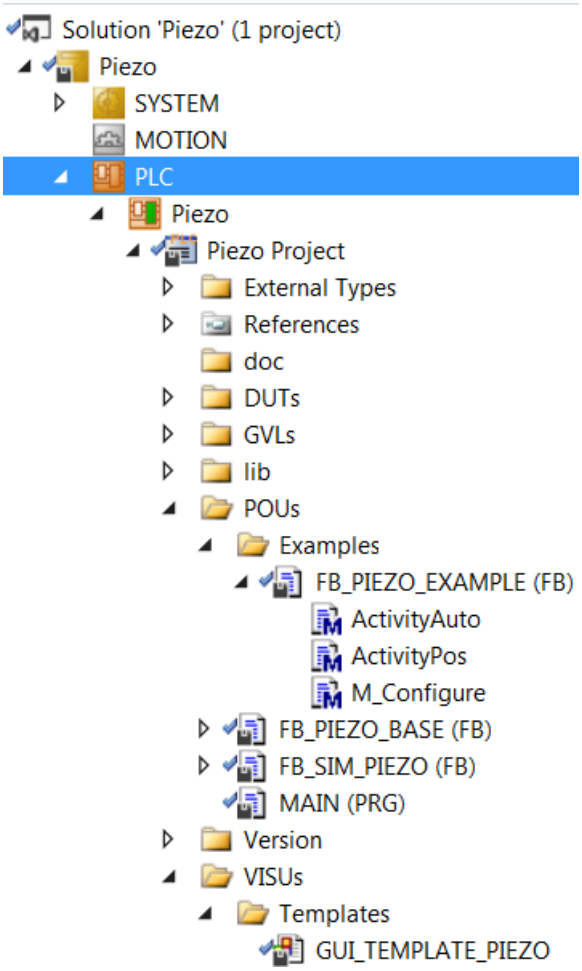
```
END_VAR
```

```
//
// Tcp RT interface functions
//

// Read every 2000 ms
// Command terminator for Lakeshore 224 is '$0D$0A'
LS224_TCP_RT(
    in_sName             := 'LS224_TCP_RT',
    in_nModel            := 224,
    in_sCmdSuffix          := '$0D$0A',
    in_sDeviceTcpIpAdr   := '192.168.0.12',
    in_nDeviceTcpPort    := 7777,
    in_nPeriod           := 2000);

// Read every 1000 ms
// Command terminator for Lakeshore 336 is '$0A'
LS336_TCP_RT(
    in_sName             := 'LS336_TCP_RT',
    in_nModel            := 336,
    in_sCmdSuffix          := '$0A',
    in_sDeviceTcpIpAdr   := '192.168.0.80',
    in_nDeviceTcpPort    := 7777,
    in_nPeriod           := 1000);
```

3.11 Piezo Library (piezo.library)

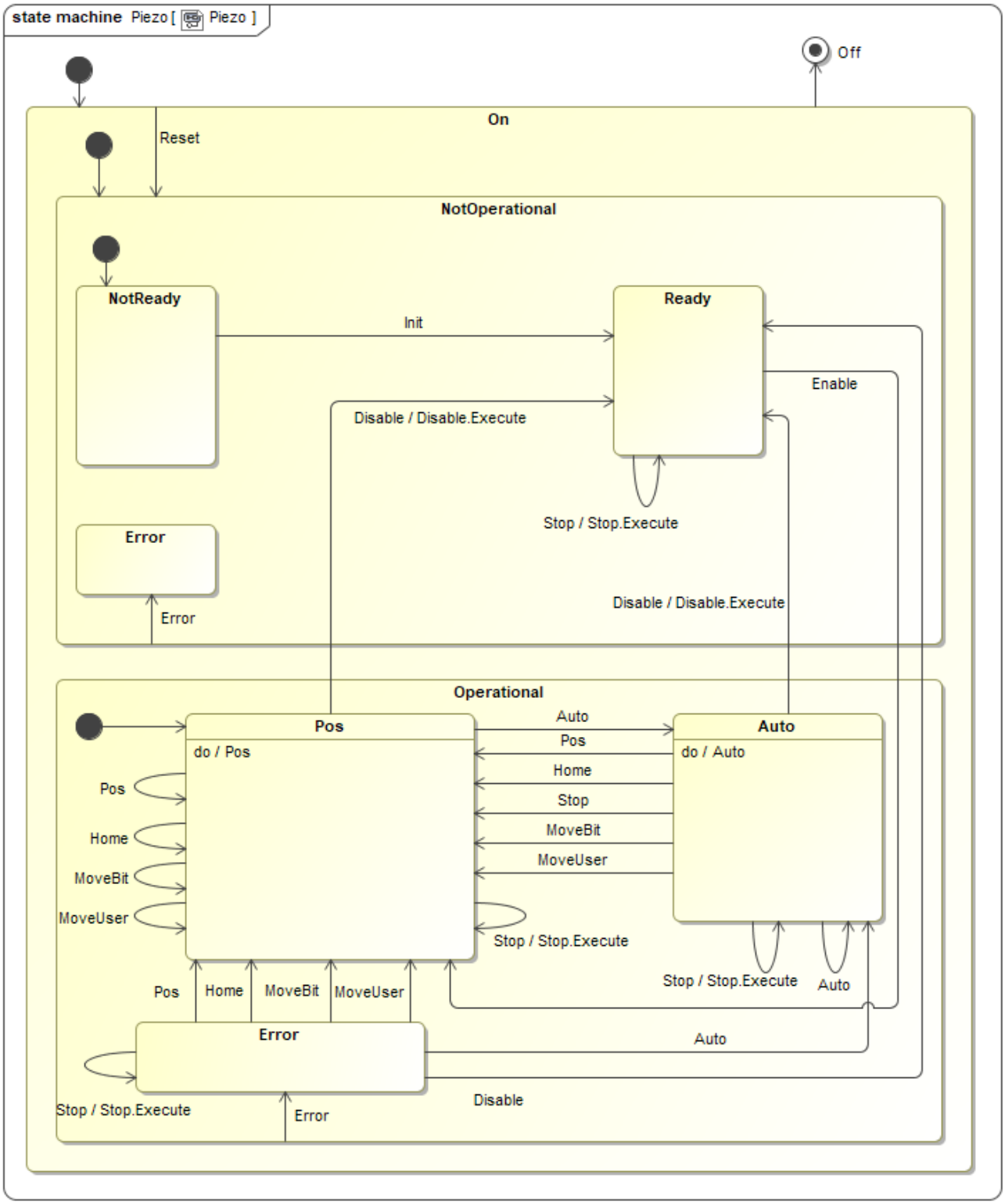
The *piezo.library* provides a generic PLC Piezo controller *FB FB_PIEZO_BASE* that has to be customised (extended) by the user. Up to three output control signals of type INT are supported (configurable). The example FB called *FB_PIEZO_EXAMPLE* shows how this customisation could be done.

The figure below shows what is delivered in the library.



3.11.1 State Machine

The state machine of the piezo controller is shown below. The main operational states are *Pos* (stationary) and *Auto* (user-defined closed loop).


state machine Piezo [Piezo]
Off
On
Reset
NotOperational
NotReady
Init
Ready
Enable
Disable / Disable.Execute
Stop / Stop.Execute
Error
Error
Disable / Disable.Execute
Operational
Pos
do / Pos
Auto
Pos
Home
Stop
MoveBit
MoveUser
Auto
do / Auto
Pos
Home
MoveBit
MoveUser
Stop / Stop.Execute
Pos
Home
MoveBit
MoveUser
Stop / Stop.Execute
Auto
Error
Auto
Stop / Stop.Execute
Error
Disable

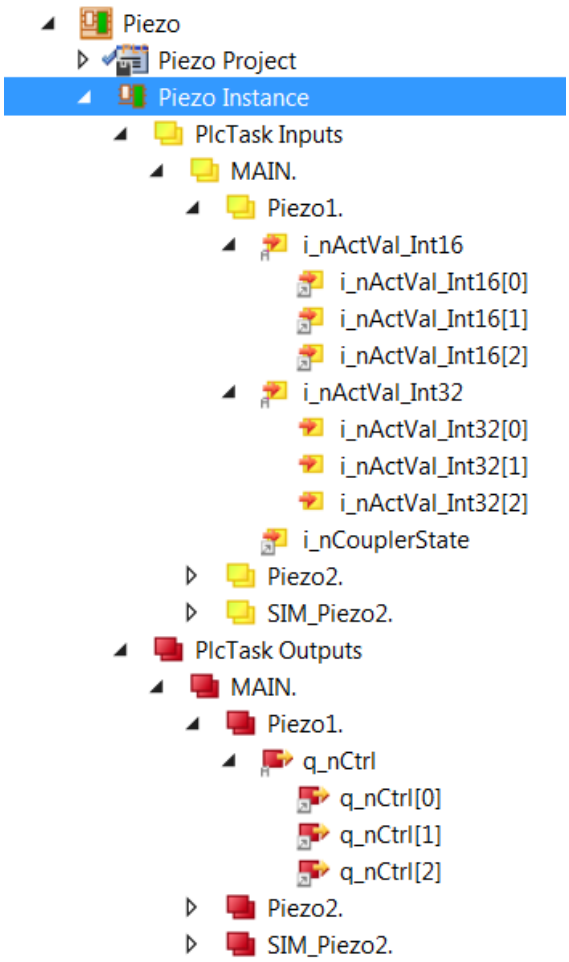
3.11.2 Input parameters

```
FUNCTION_BLOCK FB_PIEZO_BASE

VAR_INPUT  in_sName      STRING  Instance name. Default 'Piezo'.
VAR_INPUT  in_nNumAxes   INT     Max 3 axes
VAR_INPUT  in_nMaxOn     DINT    Maximum time for Piezo to be active [sec].
```

3.11.3 Signal Mapping

The figure below shows the TwinCAT view of the *FB_PIEZO_BASE* I/O variables that are available for mapping to physical signals, i.e. ports of I/O terminals.

There are two arrays for the feedback signals, one of type INT and the other one of type DINT. Any of them can be used in the customised application. There is an array for the control output signals of type INT.

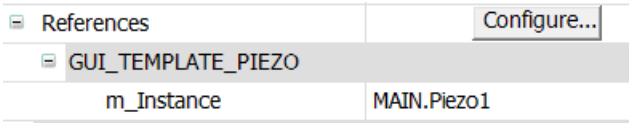


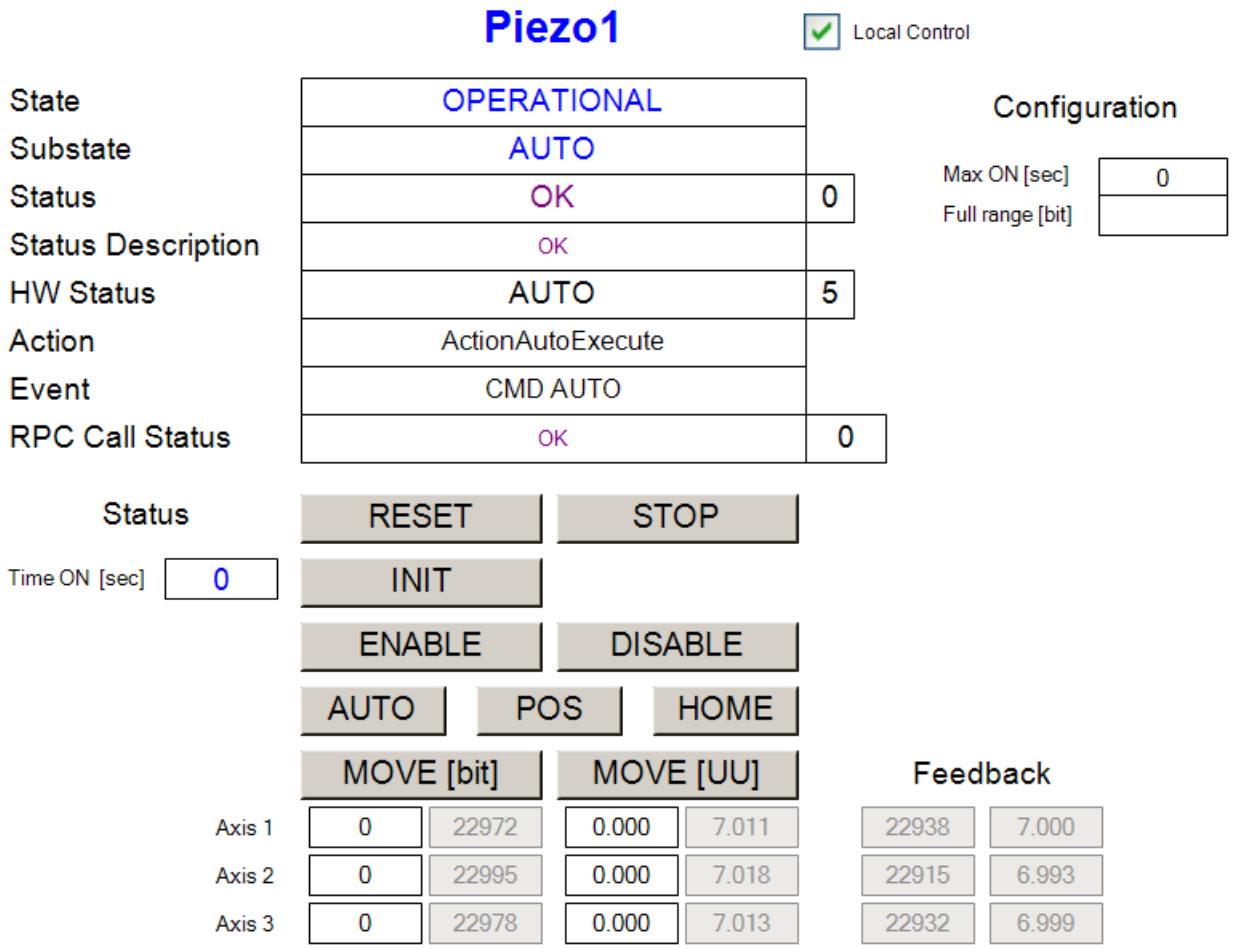
The table below describes each mapping variable.

Variable	Port Type	Optional Mapping	Description
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts a lamp signal.**
i_nActVal_Int16	**INT**	**Yes**	**Array [0..2] of feedback signals of type INT**
i_nActVal_Int32	**DINT**	**Yes**	**Array [0..2] of feedback signals of type DINT**
q_nCtrl	**INT**	**No**	**Array [0..2] of control output signals of type INT**

3.11.4 GUI Template

The *piezo.library* provides a template GUI called *GUI_TEMPLATE_PIEZO* for the control of Piezo devices. Applications can easily deploy an instance of this GUI by setting the GUI references to a particular instance, as shown below.





3.11.5 Piezo specific RPC Methods

Method	Description
RPC_Auto	**Start Automatic Loop. The user has to overload method ActivityAuto()**
RPC_Home	**Move piezos to Home position**
RPC_MoveBit	**Move piezos to positions given in [bit]**
RPC_MoveUser	**Move piezos to positions given in user units [UU]**
RPC_Pos	**Keep the piezos where they are (default behaviour). This might mean to stop the Auto loop. The user can overload this method.**

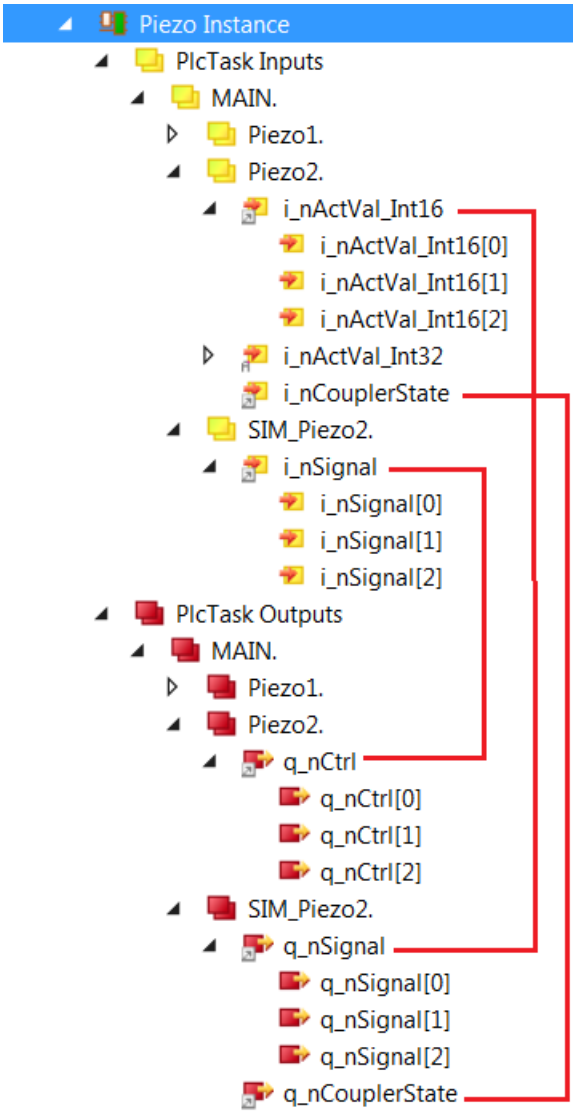
3.11.6 Piezo Simulator

The function block *FB_SIM_PIEZO* implements the piezo simulator on the PLC. The FB doesn't have any input parameter. The simulator takes the real piezo device control outputs and generate its own feedback output signals by adding sine wave offsets. The simulator output feedback is mapped to the feedback inputs of the simulated device. The sine wave is fully configurable (amplitude and cycle period).

3.11.7 Piezo Simulator RPC Methods

Method	Description
RPC_ResetConfig	**Reset simulator configuration to its default.**
RPC_SetCouplerState	**Set coupler state. Any value other than 8 would cause a failure of the simulated device.**
RPC_SetMaxError	**Set Maximal Simulated feedback offset [bit] and the Period for complete sine wave offset cycle [msec]**

3.11.8 Simulator Mapping



3.11.9 Sample Code

The following code represent a real Piezo controller (*Piezo1*) and a piezo controller (*Piezo2*) that is connected to a simulator (*SIM_Piezo2*). The mapping is shown in the *Simulator Mapping* section.

```
PROGRAM MAIN
VAR

        {attribute 'OPC.UA.DA':='1'}
        Piezo1:     FB_PIEZO_EXAMPLE;       // Piezo #1
        {attribute 'OPC.UA.DA':='1'}
```

(continues on next page)

(continued from previous page)

```
        Piezo2:     FB_PIEZO_EXAMPLE;       // Piezo #2

        {attribute 'OPC.UA.DA':='1'}
        SIM_Piezo2: FB_SIM_PIEZO;           // Simulator for Piezo #2

END_VAR
```

```
Piezo1(in_sName:='Piezo1', in_nNumAxes:=3);

Piezo2(in_sName:='Piezo2', in_nNumAxes:=3);

SIM_Piezo2();
```

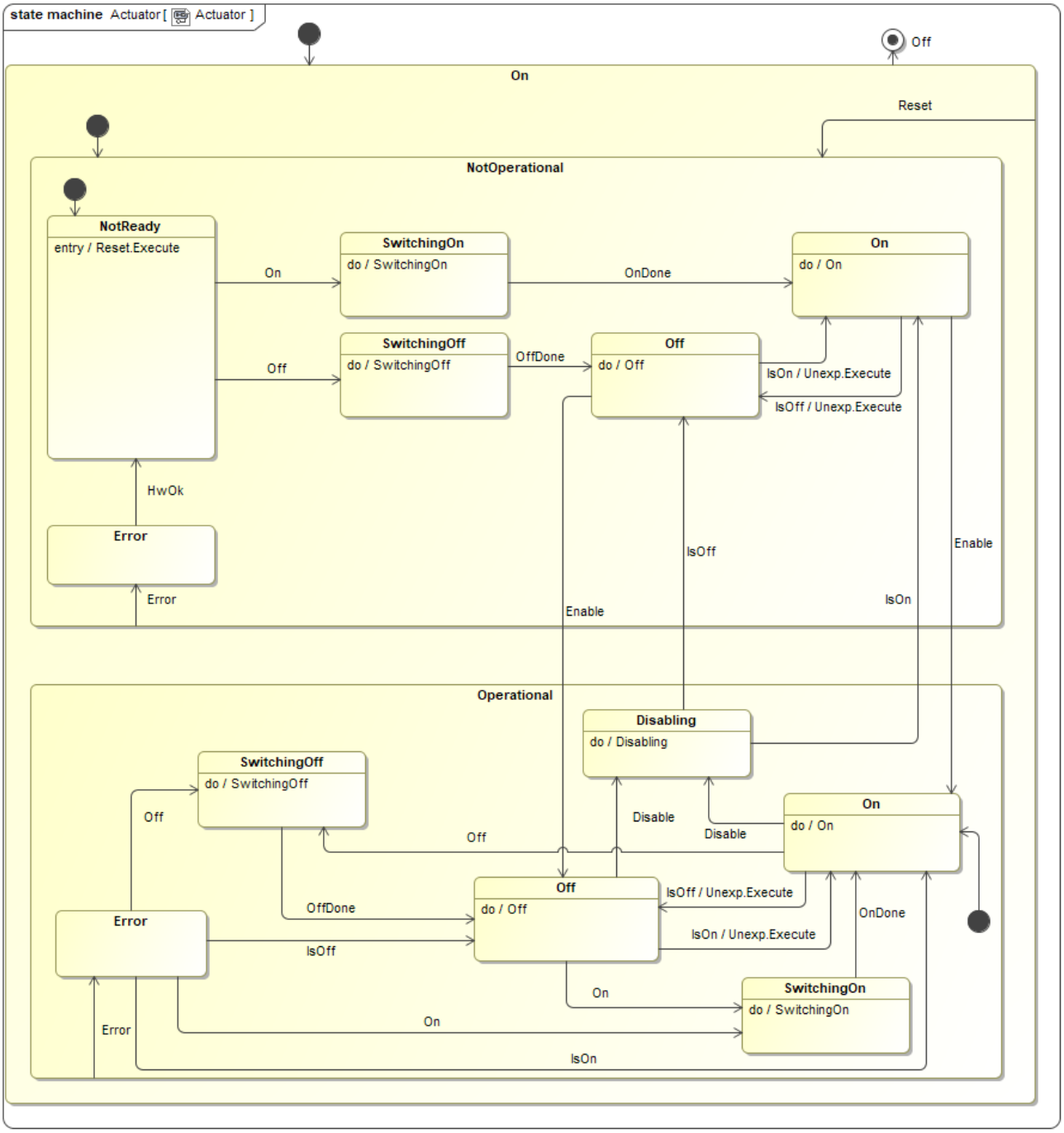
3.12 Actuator Library (actuator.library)

FB_ACTUATOR is the TwinCAT PLC Function Block for the low level control of actuators. The most common use of this FB is for power control.

The functionality is very similar to the one of *FB_LAMP* with an important difference that the HW state (On/Off) of the actuator is not affected by the state change of the controller (NOT_OP/OPERATIONAL). The HW state can only be changed in OPERATIONAL state using ON and OFF commands. An actuator can be configured to go ON on PLC reboot or on RESET of the controller. The configuration is set with the input parameter *in_bInitialState*.

3.12.1 State Machine

The state machine of the actuator controller is shown below. The main operational states are *On*, *Off*, *Switching On* and *Switching Off*.


state machine Actuator [Actuator]
Off
On
Reset
NotOperational
NotReady
entry / Reset.Execute
On
SwitchingOn
do / SwitchingOn
OnDone
On
do / On
Off
SwitchingOff
do / SwitchingOff
OffDone
Off
do / Off
IsOn / Unexp.Execute
IsOff / Unexp.Execute
HwOk
Error
Error
IsOff
Enable
Enable
IsOn
Operational
Disabling
do / Disabling
SwitchingOff
do / SwitchingOff
Off
Off
Disable
Disable
On
do / On
Off
do / Off
IsOff / Unexp.Execute
IsOn / Unexp.Execute
OnDone
OffDone
Error
IsOff
On
SwitchingOn
do / SwitchingOn
Error
On
IsOn

3.12.2 Input parameters

FUNCTION_BLOCK FB_ACTUATOR

VAR_INPUT	**in_sName**	STRING	*Default device name*
VAR_INPUT	**in_bActiveLowOn**	BOOL	*If TRUE, On signal is Active Low. Default FALSE.*
VAR_INPUT	**in_bActiveLowSwitch**	BOOL	*If TRUE, Switch ctrl signal is Active Low. Default FALSE.*
VAR_INPUT	**in_bAutoOp**	BOOL	*If TRUE, go automatically to OPERATIONAL state. Default FALSE.*
VAR_INPUT	**in_bInitialState**	BOOL	*Default power state is OFF (FALSE).*
VAR_INPUT	**in_bInvertAnalog**	BOOL	*If TRUE, analog feedback is active if signal < nAnalogThreshold. Default FALSE.*
VAR_INPUT	**in_nAnalogThreshold**	DINT	*Analog feedback signal threshold [bits]. Used if <>0. Default 0, i.e. not used.*
VAR_INPUT	**in_nMaxOn**	UDINT	*Maximum time for power to be ON [sec]. Default 0, no limit.*
VAR_INPUT	**in_nSigStablePeriod**	UDINT	*Signal is stable if it has been constant for so long [msec]. Default 200 ms.*
VAR_INPUT	**in_nTimeout**	UDINT	*Timeout for state transitions [msec]. Default 5000 ms (5 sec).*

3.12.3 Signal Mapping

The figure below shows the TwinCAT view of the *FB_ACTUATOR* I/O variables that are available for mapping to physical signals, i.e. ports of I/O terminals.

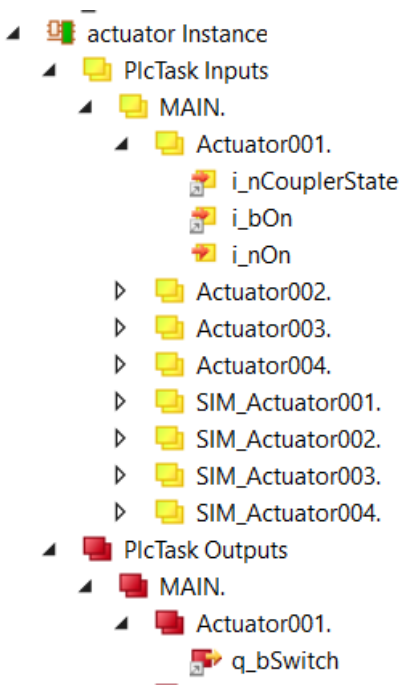


Fig. 3.12: Example of FB_ACTUATOR input/output signals.

The table below describes each mapping variable.

Variable	Port Type	Optional Mapping	Description
i_nCouplerState	**UINT**	**No**	**Mapped to the 'state' of the coupler that hosts I/O terminals. If the terminals span over more than one coupler, it is recommended to select the 'state' of the last coupler that hosts an actuator signal.**
i_bOn	**Digital In**	**No**	**Mapped to the Actuator status (ON/OFF) digital input signal.**
i_nOn	**Analog In**	**Yes**	**Analog feedback signal. Has to be mapped only if analog feedback is used.**
q_bSwitch	**Digital Out**	**No**	**Mapped to the Actuator control digital output signal.**

3.12.4 GUI Template

The Actuator Library provides a template GUI to control instances of *FB_ACTUATOR*. Applications can easily deploy an instance of this GUI to control their own actuator function blocks by setting the GUI references to the particular instance of *FB_ACTUATOR*, as shown below.

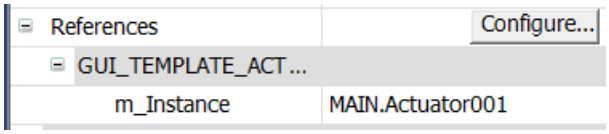


Fig. 3.13: Instantiation of GUI_TEMPLATE_ACTUATOR for Actuator001

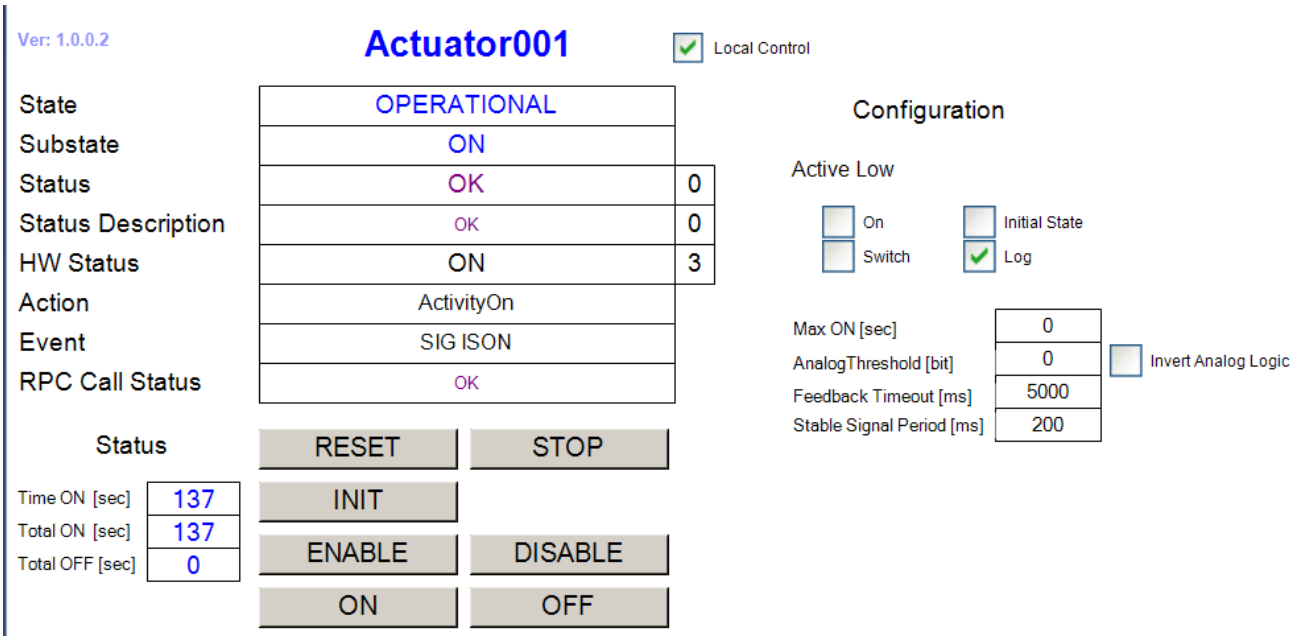


Fig. 3.14: FB_ACTUATOR HMI for Local Control.

3.12.5 Actuator specific RPC Methods

- RPC_Off() Turn actuator OFF
- RPC_On() Turn actuator ON

3.12.6 Actuator Simulator

The function block *FB_SIM_ACTUATOR* implements the actuator simulator on the PLC. The simulator has the address of the actuator instance as the only input parameter. The following code shows how the simulator is declared and executed:

Declaration:

```
{attribute 'OPC.UA.DA':='1'}
Actuator001:        FB_ACTUATOR;    // Simulated power

{attribute 'OPC.UA.DA':='1'}
SIM_Actuator001:    FB_SIM_ACTUATOR;
```

Execution:

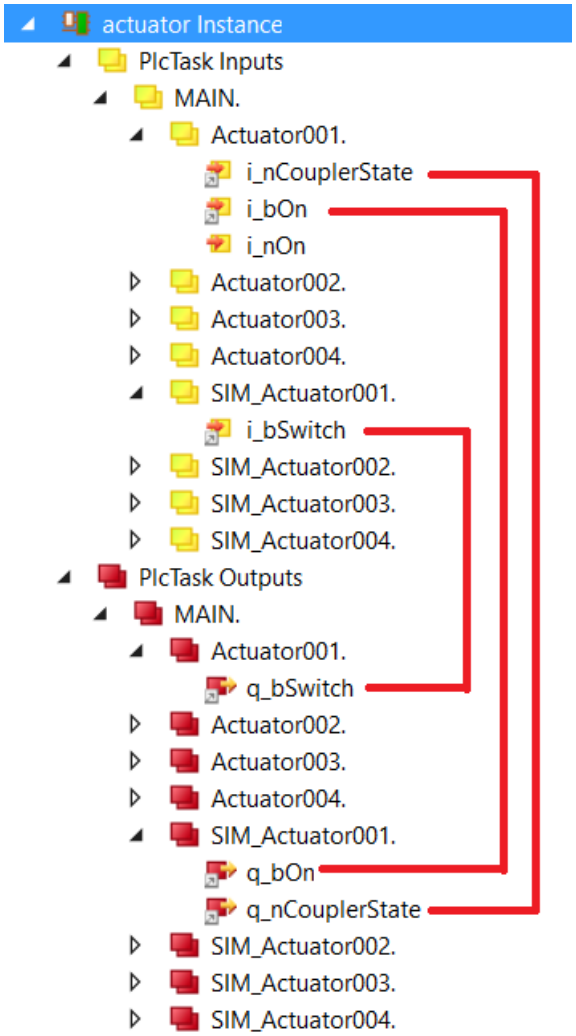
```
// Actuator001 configuration:
// - Go to  OPERATIONAL on PLC reboot
// - Initial state is OFF
```

(continues on next page)

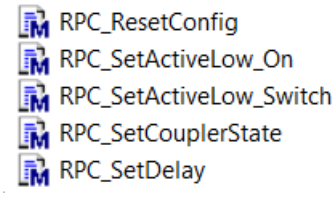
(continued from previous page)

```
Actuator001(in_sName:='Actuator001', in_bAutoOp:=TRUE,in_bInitialState:=FALSE);

// Actuator001 Simulator
SIM_Actuator001(ptrDev:=ADR(Actuator001));
```

Simulator Mapping



Simulator RPC Methods



4 Creating PLC Applications with MakeTcProject Utility

A Windows utility called *MakeTcProject.exe* is provided for automatic creation of TwinCAT projects with a selectable number of available FCF controllers. For example, with this utility the user could automatically build a PLC application for four lamps, three shutters, two motors and two actuators. The utility generates a fully operational project that includes the selected number of devices/controllers and their simulators, i.e. every device is simulated at the PLC level. Without any modification, the project can directly run on the development PC in *Local* mode or be deployed to a PLC.

The utility is recommended to be used as the starting point for creating PLC applications since it creates all necessary PLC tasks, Function Block instances, GUIs and establishes I/O links. The utility is also very useful at early stages of instrument projects when HW is not yet available, since it makes it possible to test the complete control system as if the HW were present. Once a particular piece of HW gets available, the corresponding simulator part should be removed from the project and the function block I/O variables linked to the real system I/O instead of the simulator.

Note: Important note: The generated PLC application uses PLC simulators, so it can run on both the PC in *Local* mode or on the PLC target without the need for any hardware. However, once the HW is available, the PLC signal mapping has to be adjusted accordingly. From the Device Manager point of view that runs on the WS, it *believes* that it controls the real HW.

The Windows executable *MakeTcProject.exe* is located in the *<ifw-ll>/tools/twincat/makeTcProject* directory, where *<ifw-ll>* is the directory on the Windows PC where the tar file of the component *ifw-ll* has been checked out from the ESO Gitlab site IFW-LL release 2.0.0[9]. There is no need to install the executable but it is important to note that it has to be executed from this directory because it uses a relative path to find required resources. Note that *MakeTcProject.exe* is a Windows program and it has to be executed from the *Command Prompt* (*cmd*) application. The target PLC configuration is defined in the corresponding configuration file *MakeTcProject.cfg* that is also located in the *<ifw-ll>/tools/twincat/makeTcProject* directory.

The FCF PLC binaries (libraries and modules) are located in the SVN repository tag under *http://svnhq9.hq.eso.org/p9/tags/EELT/ICS/PLC/2.0.0*. The complete directory has to be checked out to a local directory on the TwinCAT development PC. Prior to building the PLC application, the *MakeTcProject.exe* utility installs the PLC binaries in the TwinCAT Library Repository from that directory.

[9] https://gitlab.eso.org/ifw/ifw-ll/-/releases

4.1 Configuration File MakeTcProject.cfg

The configuration file sets all parameters needed by the utility, including the Visual Studio version, the location of PLC binaries, the TwinCAT output project name and the type and the number of controllers to include in the application.

The following table describes each entry in the configuration file.

Parameter	Values	Default	Description
VisualStudio	2013, 2017, etc	2013	Version of Visual Studio to use. This version has to be already installed.
Binaries		C:\Temp\PLC	Directory where the PLC binaries are checked out.
Project		plcprj	TwinCAT project name.
Lamps	0, 1, etc	1	Number of Lamps to generate.
Shutters	0, 1, etc	1	Number of Shutters to generate.
Motors	0, 1, etc	1	Number of Motors to generate.
Piezos	0, 1, etc	1	Number of Piezos to generate.
Actuators	0, 1, etc	1	Number of Actuators to generate.
ADC	yes | no	yes	If yes, ADC instance will be generated. Note that an ADC instance includes two prism motors (mult-axis).
DROT	yes | no	yes	If yes, DROT instance will be generated.
IODEV	yes | no	yes	If yes, two example instances of IODEV will be generated, one for a sensor and one for an I/O device.

4.2 Utility Specifics

- Generated projects are saved in the *C:\Temp\Solutions* directory. It is not possible to change the destination directory.
- The resulting PLC application is not optimized for the usage of CPU cores of multi-core CPUs. All PLC tasks are configured to run on Core0 and other cores, if exist, are not used. It is the responsibility of the user to optimize the PLC application by activating available isolated cores and moving tasks from one core to the other.
- Generated instances of controllers/devices will be named <device>xxx, e.g. Motor001, Motor002, etc. The user will have to rename (refactor) them to, e.g. Filter1, Mirror1, etc.
- There utility will generate only one GUI per controller type using the following convention: GUI_<device>001, e.g. GUI_Lamp001. The user will have to rename (refactor) them to, e.g. GUI_Neon, GUI_ThArg, etc.

4.3 Summary of Main Steps

The following is the summary of the required steps to build and customize the PLC application.

1. Retrieve the FCF SW from GIT and SVN repositories.
2. Edit file MakeTcProject.cfg to match the system configuration and the PLC project.
3. Build the PLC application.
4. Open the PLC application with TwinCAT, rebuild it and run it in *Local* mode on the PLC.
5. Refactor the PLC application by renaming the controller instances and GUIs.
6. Rebuild the PLC application and deploy it to the PLC.
7. Test the devices in PLC simulation.
8. Once the HW is available, scan the PLC HW and map the controller I/O signals to the real HW I/O.

In this example we assume the following:

- VisualStudio 2013 is the version to be used for building the PLC application. The specified version, together with the Beckhoff TwinCAT SW has to be already installed on the Windows PC.
- The tar file of the component *ifw-ll* has been checked out from the ESO Gitlab site and extracted into the *D:\GIT\ifw-ll* directory on the Windows PC, so the utility and the configuration file can be found in the *D:\GIT\ifw-ll\tools\twincat\makeTcProject* directory.
- The PLC binaries are checked out from the SVN repository under *http://svnhq9.hq.eso.org/p9/tags/EELT/ICS/PLC/2.0.0* into the *C:\Temp\PLC* directory.

4.4 Building PLC Application

This section describes, step-by-step, the process of building the PLC application.

The procedure is the following:

1. Check-out the *IFW-LL release 2.0.0* tar file and extract it into the *D:\GIT\ifw-ll* directory on the Windows PC.
2. SVN check out the binaries tag *http://svnhq9.hq.eso.org/p9/tags/EELT/ICS/PLC/2.0.0* to *C:\Temp\PLC*.
3. Edit file *MakePlcProject.cfg* in *D:\GIT\ifw-ll\tools\twincat\makeTcProject* and adjust the number of instances for each type of controller. The example below shows the default configuration - one of each controller.

```
D:\GIT\ifw-ll\tools\twincat\makeTcProject>type MakeTcProject.cfg
VisualStudio: 2013
```

(continues on next page)

(continued from previous page)

```
Binaries: C:\Temp\PLC
Project: plcprj
Lamps: 1
Shutters: 1
Motors: 1
ADC: yes
DROT: yes
IODEV: yes
Piezos: 1
Actuators: 1
```

4. Open a *cmd* terminal and CD to the *D:\GIT\ifw-ll\tools\twincat\makeTcProject* directory.
5. Run *MakeTcProject.exe*. The created TwinCAT project will be saved in the *C:\Temp\Solutions* directory. The output will look like this:

```
D:\GIT\ifw-ll\tools\twincat\makeTcProject>MakeTcProject.exe
MakeTcProject $Id: Program.cs 332694 2020-05-20 21:22:51Z wpirani $
INFO:
Visual Studio = '2013'
Binaries = C:\Temp\PLC
Project = 'plcprj'
Motors = 1
Lamps = 1
Shutters = 1
DROT = True
ADC = True
IODev = True
Piezos = 1
Actuators = 1
Trying to lunch Visual Studio 2013...
Creating empty project...
Adding module trkParams with GUID {15952e35-5679-4d71-9f28-196df14bbd2e}
Adding module trkModule with GUID {6f6a340e-378a-4fe4-ab0d-f15c12dfef0c}
Populating project...
Adding libraries...
Copying required libraries to System repository...
        C:\Temp\PLC\Libraries\actuator.library
        C:\Temp\PLC\Libraries\IODev.library
        C:\Temp\PLC\Libraries\Lamp.library
        C:\Temp\PLC\Libraries\Motor.library
        C:\Temp\PLC\Libraries\Mudpi.library
        C:\Temp\PLC\Libraries\Piezo.library
        C:\Temp\PLC\Libraries\plctpl.library
        C:\Temp\PLC\Libraries\rsCommCommon.library
        C:\Temp\PLC\Libraries\rsCommSerial.library
        C:\Temp\PLC\Libraries\rsCommTcp.library
        C:\Temp\PLC\Libraries\rsCommTcpRt.library
        C:\Temp\PLC\Libraries\Shutter.library
```

(continues on next page)

(continued from previous page)

```
        C:\Temp\PLC\Libraries\Switch.library
        C:\Temp\PLC\Libraries\timer.library
Adding required libraries to this project...
Importing VISUs...
Importing: GUI_CCS_SIM.TcVIS from ..\..\..\controllers\motor\Motor\VISUs\
 ↪Tracking\
Importing: GUI_time_info.TcVIS from ..\..\..\controllers\timer\timer\timer\
 ↪VISUs\
Importing: GUI_MA_ADC.TcVIS from ..\..\..\controllers\motor\Motor\VISUs\
 ↪Tracking\
Importing: GUI_MA_DROT.TcVIS from ..\..\..\controllers\motor\Motor\VISUs\
 ↪Tracking\
Importing: GUI_Lamp001.TcVIS from ..\..\..\controllers\lamp\Lamp\Lamp\VISUs\
Importing: GUI_Motor001_Cfg.TcVIS from ..\..\..\controllers\motor\Motor\
 ↪VISUs\Motor\
Importing: GUI_Motor001_Ctrl.TcVIS from ..\..\..\controllers\motor\Motor\
 ↪VISUs\Motor\
Importing: GUI_Shutter001.TcVIS from ..\..\..\controllers\shutter\Shutter\
 ↪Shutter\VISUs\
Importing: GUI_IODev001.TcVIS from ..\..\..\controllers\ioDev\IODev\IODev\
 ↪VISUs\
Importing: GUI_Piezo001.TcVIS from ..\..\..\controllers\piezo\Piezo\Piezo\
 ↪VISUs\
Importing: GUI_Actuator001.TcVIS from ..\..\..\controllers\actuator\
 ↪actuator\actuator\VISUs\
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_CCS_SIM.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_time_info.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_MA_ADC.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_MA_DROT.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_Lamp001.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_Motor001_
 ↪Cfg.TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_Motor001_
 ↪Ctrl.TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_Shutter001.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_IODev001.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_Piezo001.
 ↪TcVIS
Fixing references for: C:\temp\Solutions\plcprj\plcprj\VISUs\GUI_
 ↪Actuator001.TcVIS
Building project...
Linking variables...
```

(continues on next page)

(continued from previous page)

```
Fixing C:\temp\Solutions\plcprj\plcprj\PlcTask.TcTTO...
All done!
```

4.5 Testing PLC Application on Windows PC

In this step, the PLC application is open with the TwinCAT IDE and the application is run on the development Windows PC in *Local* mode. Each device can be tested using the generated GUIs.

1. In the Windows Explorer, go to the *C:\Temp\Solutions\plcprj* directory and double-click on *plcprj.sln*. This will start the TwinCAT IDE and open the newly created project.
2. From the *BUILD* pull-down menu, click on *Rebuild Solution*.
3. From the *TWINCAT* pull-down menu, click on *Activate Configuration*. Confirm with 'OK'. When asked "*Restart TwinCAT System in Run Mode*", confirm with *OK*.
4. From the *PLC* pull-down menu, click on *Login*. Confirm with 'OK'.
5. In the TwinCAT *Solution Explorer* window on the left, go to the *VISUs* directory and start the GUIs by double-clicking on each of them. Test each device.

4.6 Customizing PLC Application to match Instrument Project

In this step, the user has to rename (refactor) the instances of controllers/devices in order to match the Instrument SW configuration. Please note that the *Refactor* feature in the TwinCAT IDE is a very powerful, quick and safe tool for renaming all instances of a certain string in the PLC project.

For example, the following renaming will be done in the project:

- Refactor Lamp001 to ThArg.
- Refactor Motor001 to Filter1.
- Refactor Actuator001 to CamPower.
- Rename GUI_Lamp001 to GUI_ThArg.
- etc, etc

4.7 PLC Application deployment to PLC

In this step, the PLC will be deployed to the real PLC with a given name and IP address. From the TwinCAT IDE, the user has to choose the target PLC system. This can be done only if the current target is *Local*.

The procedure is the following:

1. Logout by clicking on the *PLC* pull-down menu and selecting *Logout*.

2. Ensure that the current target system is *<Local>*.
3. Click on *<Local>* and select *Choose Target System...*.
4. From the *Choose Target System*, select *Search (Ethernet)...*.
5. In the *Enter Host Name / IP* field, write the PLC name or the IP and press *Enter*.
6. Select the PLC from the available list and press *Add Route* to connect to the PLC.

After the connection with the PLC has been established, the PLC application can be downloaded to the PLC and tested in simulation.

Once the HW is available, the I/O links with the simulators should be cleared and the mapping should be done with the real HW I/O instead. Unused simulators should be removed from the project.

5 Client Application

The client application (*fcfClient*) is a simple utility allowing to send commands to the *Device Manager* from the command line. In this context we use the words commands and events as synonyms. The fcfClient uses the standard interface module `stdif` and the application interface module `fcfif` to compose the payload of the messages. The *fcfClient* sends the messages using CII MAL request/reply.

```
$ fcfClient <serviceURI> <command> ["<parameters>"]

Where
      <serviceURI> destination of the command (e.g. zpb.rr://127.0.0.1:12081)
      <command>    command to be sent to the server (e.g. Init)
      <parameters> optional parameters of the command.
```

Warning: The URI shall not contain the '/' at the end otherwise the client will hang trying to connect to a non existing server.

5.1 List of Commands

The commands (events) currently supported by the *fcfClient* utility are:

Table 5.1: Client commands

Command	Parameters
Init	“”
Enable	“”
Disable	“”
State	“”
Status	“”
Setup	“”
Recover	“”
Reset	“”
Config	“”
DevNames	“”
SetLog	“<ERROR|INFO|DEBUG|TRACE>”
Simulate	“<device id1>, . . . ,[<device idn>]”
StopSim	“<device id1>, . . . ,[<device idn>]”
Ignore	“<device id1>, . . . ,[<device idn>]”
StopIgn	“<device id1>, . . . ,[<device idn>]”
HwInit	“<device id1>, . . . ,[<device idn>]”
HwEnable	“<device id1>, . . . ,[<device idn>]”
HwDisable	“<device id1>, . . . ,[<device idn>]”
HwReset	“<device id1>, . . . ,[<device idn>]”
StartDaq	“<daq id>”
StopDaq	“<daq id>”
Exit	“”

Warning: Due to the upgrade to CII, the payload of the SETUP command cannot be defined through a JSON file. For sending SETUP and other commands is better to use the Python interface.

Note: The HW commands like HwInit or HwEnable control the state of the controller associated to the device.

5.1.1 Examples

Note: The following examples assume the server is listening for incoming events under the URI zpb.rr://127.0.0.1:12081 in the local host.

Enabling debug level in the server

```
$ fcfClient zpb.rr://127.0.0.1:12081  SetLog "DEBUG"
```

Initialising the server

```
$ fcfClient zpb.rr://127.0.0.1:12081  Init ""
```

Moving the server to Operational state

```
$ fcfClient zpb.rr://127.0.0.1:12081  Enable ""
```

Executing a *Setup* command from the command line

This is not possible in this version using fcfClient utility. Please use the Python client library instead.

6 Python Client Library

It is possible to communicate with the *Device Manager* through clients developed in Python. The FCF provides a library that simplifies the interaction with the Device Manager (`fcfclib`). This is probably the easiest and more flexible way to interact with the Device Manager. The fcfclib encapsulates the creation of the payload for the Setup command by providing predefined methods.

Users might want to interact directly with FCF ICD binding methods. This is, of course possible, but it is outside the scope of this library.

Note: This Python client library was added in version 2.

6.1 Error Handling

The `fcfclib` reports as a RuntimeError exceptions that may be delivered by the Device Manager. Unfortunately a due to a limitation of CII MAL, we cannot get the source error message yet from the Python exception.

6.2 Classes

The `fcfclib` library provides internal classes to build the buffer for each devices. In addition, this library provides one class that encapsulates the user interface with the DeviceManager. This class is the DevmgrCommand class.

6.2.1 DevmgrCommand

The constructor of the DevmgrCommand class support two parameters: uri and timeout. The timeout is optional and has a default of one minute, expressed in milliseconds.

The class handles an internal buffer object that is used to build the payload of the Setup command. Each time the command is executed, the buffer is reset. The user can add multiple device settings into the internal buffer before executing the setup command.

The methods names of the class are shown in the tables below. They try to be self explanatory.

6.2.2 Methods for building Setup Payload

Method	**parameters**
close_shutter	device (string)
open_shutter	device (string)
switch_lamp_on	device (string)
switch_lamp_on_with_intensity	device (string), intensity (float)
switch_lamp_on_with_timer	device (string), timer (integer)
move_motor_abs_enc	device (string), position (integer)
move_motor_abs_pos	device (string), position (float)
move_motor_rel_enc	device (string), position (integer)
move_motor_rel_pos	device (string), position (float)
move_motor_name_pos	device (string), name (string)
drot_motor_abs_pos	device (string), position (float)
move_drot_posangle	device (string), position (float)
start_drot_sky_tracking	device (string)
start_drot_elev_tracking	device (string)
start_drot_user_tracking	device (string)
stop_drot_tracking	device (string)
move_adc_motor1_abs_pos	device (string), position (float)
move_adc_posangle	device (string), position (float)
start_adc_tracking	device (string)
stop_adc_tracking	device (string)
set_piezo_auto	device (string)
set_piezo_pos	device (string)
set_piezo_home	device (string)
move_piezo_in_user_units	device (string), pos1 (float), pos2 (float), pos3 (float)
move_piezo_in_bits_units	device (string), bit1 (integer), bit2 (integer), bit3 (integer)
switch_actuator_off	device (string)
switch_lamp_off	device (string)
switch_lamp_off	device (string)

6.2.3 Methods for Command Interface

Method	parameters
setup	None
status	device list (argv)
ignore	device list (argv)
simulate	device list (argv)
stop_ignore	device list (argv)
stop_simulate	device list (argv)
hw_init	device list (argv)
hw_enable	device list (argv)
hw_disable	device list (argv)
hw_reset	device list (argv)
devnames	None
devconfig	device
state	None
init	None
enable	None
enable	None
disable	None
reset	None
stop	None

6.3 Examples

6.3.1 Retrieving the Status

```
import fcfclib.devmgrCommands as fcs

uri = "zpb.rr://127.0.0.1:12081"
fcsif = fcs.DevMgrCommands(uri)
print(fcsif.status())

['shutter1.simulated = true', 'shutter1.lcs.state = Undefined', 'shutter1.lcs.
↪substate = Undefined',
'lamp1.simulated = true', 'lamp1.lcs.state = Undefined', 'lamp1.lcs.substate =␣
↪Undefined',
'lamp1.lcs.intensity = 0.000000', 'motor1.simulated = true', 'motor1.lcs.state =␣
↪Undefined',
'motor1.lcs.substate = Undefined', 'motor1.lcs.pos_target = 0.000000',
'motor1.lcs.pos_actual = 0.000000', 'motor1.lcs.vel_actual = 0.000000',
'motor1.lcs.axis_enable = false', "motor1.pos_actual_name = ''",
'motor1.pos_enc = -2147483648', '', 'OK']
```

6.3.2 Executing a single *Setup*

```
import fcfclib.devmgrCommands as fcs

uri = "zpb.rr://127.0.0.1:12081"
fcsif = fcs.DevMgrCommands(uri)
# Move single motor1 to absolute position 100 in user units
# Fill the internal setup buffer
fcsif.move_motor_abs_pos("motor1", 100)
# Send setup command to the Device Manager listening in port 12081
fcsif.setup()
```

6.3.3 Executing a *Setup* with multiple devices

```
import fcfclib.devmgrCommands as fcs

uri = "zpb.rr://127.0.0.1:12081"
fcsif = fcs.DevMgrCommands(uri)
# Setup multiple devices at once
# Fill the internal setup buffer
fcsif.move_motor_abs_pos("motor1", 100)
fcsif.switch_lamp_off("lamp1")
fcsif.close_shutter("shutter1")
# Send setup command to the Device Manager listening on port 12081
fcsif.setup()
```

7 FCF CLI

The FCF python library provides an experimental command shell with simple commands aiming to simplify the interaction with the Device Manager. The FCF shell can be invoked issuing the command `fcfcli`.

7.1 Command Line Parameters

The fcfcli offers some few command line parameters. If no parameters are specified, the fcfcli will use the default name services and use nomad/consul to obtain the correct IP and port numbers of the Device Manager. The fcfcli shell commmands are not necessary using the same names as the MAL interfaces with the purpose to shorten the commands names. Commands that could take long time, are asynchronous so the shell can continue being used while answer from the previous command is not yet received.

Parameter	Description
–uri	if the URI is specified, the supcli will use it to connect to the server
–name	When using nomad, one could specifify the name of the service instead of the URI
–module	Custom interface library

Warning: The fcfcli shell assumes NOMAD/CONSUL services are up and running. If this is not the case then only –uri parameter can be used.

Warning: The URI shall not contain the '/' at the end otherwise the client will hang trying to connect to a non existing server.

Note: The fcfcli shell was created for the Device Manager but since it uses the standard interface, it can be used for any server implementing this interface, although only for the standard events like init, enable, disable, etc.

```
fcfcli --uri zpb.rr://134.171.3.48:30269
fcfSh>?
Available command list:
-  close
-  daqstart
-  daqabort
-  daqstatus
-  daqstop
```

(continues on next page)

(continued from previous page)

```
-  devinfo
-  devnames
-  devstatus
-  devconfig
-  disable
-  enable
-  help
-  ignore
-  init
-  move
-  move_by_name
-  move_by_speed
-  open
-  reset
-  setup_json_file
-  setup_json_string
-  setup_spf
-  simulate
-  start_track
-  state
-  status
-  get_config
-  stop
-  stop_ignore
-  stop_simulate
-  stop_track
-  switch_off
-  switch_on
```

7.2 Shell History

The FCF shells keeps its own history file under $HOME/.fcfSh.txt. The history can be accessed using the arrows keys.

7.3 Shell Completion

The FCF shell provides a completion capability for the supported commands using the Tab key.

7.4 Supported Shell Commands

Command	Parameters	Description
init	[<device>]	sends the init (stdif) event to the connected server. If a device is specified, it will forward the command to the given device instead.
enable	[<device>]	sends the enable (stdif) event to the connected server. If a device is specified, it will forward the command to the given device instead.
disable	[<device>]	sends the disable (stdif) event to the connected server. If a device is specified, it will forward the command to the given device instead.
reset	[<device>]	sends the reset (stdif) event to the connected server. If a device is specified, it will forward the command to the given device instead.
stop		sends the stop (stdif) event to the connected server
help		print the list of supported commands
daqstart	<daqid>	Start DAQ acquisition
daqabort	<daqid>	Abort DAQ acquisition
daqstatus	<daqid>	Get DAQ acquisition status
daqstop	<daqid>	Stop DAQ acquisition
devnames		Get the list of devices managed by the server
devstatus	[<dev1>,<devN>]	Get the detailed status of the devices managed by the server
devconfig	<device>	Get the actual configuration of a device (yaml formatting)
simulate	<device>	sends the simulate event to the connected server
stop_simulate	<device>	sends the stopsim event to the connected server
ignore	<device>	sends the ignore event to the connected server
stop_ignore	<device>	sends the stopign event to the connected server
setup_json_file	<file>	The setup commands uses a JSON file to set runtime parameters. The contents of the JSON file shall match the defined schema.
setup_json_string	<JSON string>	The setup commands uses a JSON format to set runtime parameters. The JSON string shall match the defined schema.

Continued on next page

Table 7.1 – continued from previous page

Command	Parameters	Description
setup_spf	<string>	The setup commands uses a simple format to specify the parameters to be changed. Examples: shutter1:action="OPEN" lamp1:action="ON", actuator1:action:"OFF"
state		sends the GetState (stdif) event to the connected server
status		sends the GetStatus (stdif) event to the connected server
get_config		Get the server configuration (yaml formatting)
subset		This command is equivalent to the setup but instead of JSON format, it uses a simple parameter format (SPF), e.g. subset <device>:ON=false
close	<shutter device>	It closes a shutter
open	<shutter device>	It opens a shutter
switch_on	<lamp device>	It switch on a lamp or an actuator device
switch_of	<lamp device>	It switch off a lamp or an actuator device
move	<motor>,<pos>	It moves a motor to a target position
move_by_name	<motor>,<name>	It moves a motor using a named position
move_by_speed	<motor>,<speed>	It moves a motor in speed
start_track	<device>,<mode>	It start tracking for using a given mode
stop _track	<device>	It stops tracking
ctrl-d		Stop the shell

7.4.1 using devnames command

```
fcsSh> devnames
reply> = shutter1, lamp1, motor1, drot1, adc1, piezo1
OK
```

7.4.2 using devstatus command

```
fcsSh> devstatus lamp1
reply> = ['lamp1.lcs.state = Operational', 'lamp1.lcs.substate = On', 'lamp1.lcs.
↪intensity = 0.000000', '', 'OK']
fcsSh>
```

7.4.3 using switch_off command

```
fcsSh> switch_off lamp1
fcsSh> reply> = OK setup completed.
fcsSh>
```

7.4.4 using move command

```
fcsSh> move motor1,50
fcsSh> reply> = OK setup completed.
fcsSh>
```

7.4.5 using setup command

This example uses a test JSON file (test.json).

```
[{
"id": "shutter1", "param": {
    "shutter": {
            "action": "OPEN"
            }
    }
},
{
"id": "lamp1", "param": {
    "lamp": {
            "action": "OFF"
            }
    }
},
{
"id": "motor1", "param": {
    "motor": {
        "action": "MOVE_ABS",
        "pos": 50,
        "unit": "UU"
            }
    }
}]
```

```
fcsSh> setup_json_file test.json
fcsSh> reply> = OK setup completed.
fcsSh>
```

```
fcsSh> setup_json_string [{"id": "shutter1", "param": { "shutter": {"action":
↪"OPEN"}}}]
fcsSh> reply> = OK setup completed.
fcsSh>
```

```
fcsSh> setup_spf shutter1:action="OPEN"
fcsSh> reply> = OK setup completed.
fcsSh>
```

8 JSON Schema

JSON is used to compose the payload of the setup command. To minimize possible errors, the client python library validates the payload against a defined schema before sending the command to the server. The FCF provides a schema that covers all standard devices. Instrument implementing custom devices shall extend this schema definition.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "title": "FCF schema",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "id": {
        "type": "string",
        "description": "device identifier."
      },
      "param": {
        "$ref": "#/definitions/param"
      }
    }
  },
  "definitions": {
    "param": {
      "type": "object",
      "properties": {
        "shutter": {
          "$ref": "#/definitions/shutter"
        },
        "actuator": {
          "$ref": "#/definitions/actuator"
        },
        "lamp": {
          "$ref": "#/definitions/lamp"
        },
        "motor": {
          "$ref": "#/definitions/motor"
        },
        "drot": {
          "$ref": "#/definitions/drot"
        },
        "adc": {
          "$ref": "#/definitions/adc"
        },
        "piezo": {
          "$ref": "#/definitions/piezo"
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "oneOf": [
              { "required":
                   [ "shutter" ] },
              { "required":
                   [ "actuator" ] },
              { "required":
                   [ "lamp" ] },
              { "required":
                   [ "motor" ] },
              { "required":
                   [ "drot" ] },
              { "required":
                   [ "adc" ] },
              { "required":
                   [ "piezo" ] }
              ]
  },
  "shutter": {
    "type": "object",
    "properties": {
      "action": {
        "type": "string",
        "enum": ["OPEN", "CLOSE"],
        "description": "Shutter action."
      }
    },
     "required": ["action"]
  },
   "actuator": {
    "type": "object",
    "properties": {
      "action": {
        "type": "string",
        "enum": ["ON", "OFF"],
        "description": "Actuator action."
      }
    },
     "required": ["action"]
  },
  "lamp": {
    "type": "object",
    "properties": {
      "action": {
        "type": "string",
        "enum": ["ON", "OFF"],
        "description": "Lamp action."
      },
      "intensity": {
```

(continues on next page)

(continued from previous page)

```
        "type": "number",
        "minimum": 1,
        "maximum": 100,
        "description": "Lamp intensity."
      },
      "time": {
        "type": "integer",
        "minimum": 1,
        "description": "Lamp timer."
      }
    },
    "required": ["action"],
    "additionalProperties": false
  },
  "motor": {
    "type": "object",
    "properties": {
      "action": {
        "type": "string",
        "enum": ["MOVE_ABS", "MOVE_REL", "MOVE_BY_NAME", "MOVE_BY_SPEED"],
        "description": "Motor action."
      },
      "pos": {
        "type": "number",
        "description": "Motor position in user units."
      },
      "enc": {
        "type": "integer",
        "description": "Motor position in encoders."
      },
      "unit": {
        "type": "string",
        "enum": ["UU", "ENC"],
        "description": "Motor position unit."
      },
      "name": {
        "type": "string",
        "description": "Motor named position."
      },
      "speed": {
        "type": "number",
        "description": "Motor speed."
      }
    },
    "required": ["action"]
  },
  "drot": {
     "allOf": [{ "$ref": "#/definitions/motor" }],
    "properties": {
```

(continues on next page)

(continued from previous page)

```
        "action": {
          "type": "string",
          "enum": ["MOVE_ABS", "MOVE_REL", "MOVE_BY_NAME", "MOVE_BY_SPEED",
↪"MOVE_BY_POSANG", "START_TRACK", "STOP_TRACK"],
          "description": "Drot action."
        },
        "posang": {
          "type": "number",
          "description": "Motor position angle."
        },
         "mode": {
          "type": "string",
          "enum": ["ENG", "STAT", "SKY", "ELEV", "USER"],
          "description": "Drot mode."
        }
      },
      "required": ["action","mode"]
    },
    "adc": {
      "allOf": [{ "$ref": "#/definitions/motor" }],
      "properties": {
        "action": {
          "type": "string",
          "enum": ["MOVE_ABS", "MOVE_REL", "MOVE_BY_NAME", "MOVE_BY_SPEED",
↪"MOVE_BY_POSANG", "START_TRACK", "STOP_TRACK"],
          "description": "Adc action."
        },
        "posang": {
          "type": "number",
          "description": "Motor position angle."
        },
        "axis": {
          "type": "string",
          "enum": ["ADC1", "ADC2"],
          "description": "Adc axis."
        },
         "mode": {
          "type": "string",
          "enum": ["ENG", "OFF", "AUTO"],
          "description": "Adc mode."
        }
      },
      "required": ["action","mode"]
    },
    "piezo": {
      "type": "object",
      "properties": {
        "action": {
          "type": "string",
```

(continues on next page)

(continued from previous page)

```
          "enum": ["SET_AUTO", "SET_POS", "SET_HOME", "MOVE_ALL_BITS", "MOVE_ALL_
↪POS"],
          "description": "Piezo action."
        },
        "pos1": {
          "type": "number",
          "description": "Piezo position 1 in volts."
        },
        "pos2": {
          "type": "number",
          "description": "Piezo position 2 in volts."
        },
        "pos3": {
          "type": "number",
          "description": "Piezo position 3 in volts."
        },
        "bit1": {
          "type": "integer",
          "description": "Piezo position 1 in bits."
        },
        "bit2": {
          "type": "integer",
          "description": "Piezo position 2 in bits."
        },
        "bit3": {
          "type": "integer",
          "description": "Piezo position 3 in bits."
        }
      },
      "required": ["action"]
    }
  }
}
```

9 Simple Parameter Format (SPF)

This is a custom format used to express the parameters for the setup command in an easy a simple way for users. You can include multiple parameters for one or more devices. The FCF client library will take this format and translate it to JSON internally to be able to check whether the syntax is correct before submitting the command to the server.

The format has the following syntax:

```
<device>:<parameter>=<value>[,<device>:<parameter>=<value>]...[,<device>:
↪<parameter>=<value>]
```

The FCF client library provides a method where you can setup the FCF using a string with this format (setup_spf). For more information, see client interface description above.

```
fcsSh> setup_spf shutter1:action="OPEN",lamp1:action="ON",lamp1:intensity=80
fcsSh> reply> = OK setup completed.
fcsSh>
```

10 Device Simulator

The FCF SDK provides a small Python application to implement Device Simulators on the WS, for short, DevSim. These Device Simulators, emulate the PLC, sufficiently well to be able to carry out the basic operations of the device. This makes it possible to carry out a integration test of the instrument software, without the availability of a 'physical' PLC. This is useful, e.g. for autonomous tests, running during the nightly build (Jenkins), and, of course, during development, where it may be more efficient to test/debug against a local process, rather than a PLC; maybe a PLC is not even always available.

The FCF Device Simulators, implements the same state machine as is implemented on the PLC. Moreover the OPC UA interface (namespace), is the same, at least for what concerns the nodes, essential for controlling and monitoring the PLC device from the FCF Device Manager.

The Device Simulators load the SCXML state machine model and the OPC UA namespace profile at start-up, and configures themselves accordingly.

It is possible to implement cumstomized business logic for each Device Simulator, to achieve a simulation, accurate enough to do the development and test of the WS Device Manager.

For each Special Device, implemented for an instrument, an FCF Device Simulator shall be provided, in addition to the WS Device Manager and PLC device code.

It may be useful, to start the implementation of a new device by implementing **first** the Device Simulator, and **then** use this during the implementation of the Device Manager, because it is easier to control a local application than an application running on a PLC and some device operations may be executed faster by the Device Simulator compared to executing them on the PLC.

The following standard Device Simulators are provided:

- Actuator
- ADC
- CCC
- DROT
- Lamp
- Motor
- Piezo
- Sensor
- Shutter

If support for new types of devices is added, the associated Device Simulators will be provided.

10.1 Device Simulator - Command Line Options

A Device Simulator accepts the following command line options (example):

```
$ fcfDevsimShutter --help
usage: fcfDevsimShutter [-h] --port PORT --cfg CFG [--use-ext-ip]
                        [--log-level LOG_LEVEL]
                        [--log-file LOG_FILE] [--verbose]

Device Workstation Simulator

optional arguments:
  -h, --help            show this help message and exit
  --port PORT           server port number
  --cfg CFG             configuration
  --use-ext-ip          use the external IP address of deployment host
  --log-level LOG_LEVEL
                        set log level (CRITICAL, ERROR, WARNING, INFO, DEBUG)
  --log-file LOG_FILE   log output file
  --verbose             output log on stdout
```

Option: “cfg”:

YAML based configuration. Context specific parameters can be be added in this configuration, for the specific Device Simulator implementation.

Option: “use-ext-ip”:

By default, a Device Simulator uses the loop-back IP address (127.0.0.1). This means that is it only possible to reach it from within the local host. To make the Device Simulator available on the network, use this option to make it serve on the external IP address.

10.2 Device Simulator - Base Application

A Device Simulator is derived from the base class fcf.devsim.devsimLib.deviceSimulatorBase[10].

At this point, there is no way to generate the Python source files for a Device Simulators, so the files must be prepared manually. It is suggested to copy the source files of one of the existing Device Simulators, and adapt them.

As example, the Lamp Device Simulator could be used. It is structured such that there is a library part, which could be re-used to implement other Device Simulators, with similar properties and the specific part, which is also the deployment module:

[10] https://gitlab.eso.org/ifw/ifw-hl/-/blob/master/fcf/devsim/lib/src/fcfDevsimLib/deviceSimulatorBase.py

- Lamp DevSim Library[11].
- Standard Lamp DevSim Deployment Module[12].

10.3 Generation of the SCXML State Machine Model

At this point in time, there is no way to share the same SXCML definition between the PLC and the associated DevSim. A solution for this may be provided at a later stage. Until then, the SCXML state chart can be generated from the MagicDraw state machine chart, in which the proper COMODO stereotypes have been applied. For instructions for how to create the SCXML definition, using MagicDraw and COMODO, consult the RAD User Manual. Alternatively, the SCXML state chart document, can be created by hand, by means of a text editor or an XML editor. An example of the SCXML chart for the Lamp Device can be accessed here[13] (generated from the MagicDraw model, using COMODO).

10.4 Generation of the OPC UA Namespace Profile

To facilitate generating the OPC UA XML namespace profile, a simpler and more compact format has been defined, based on YAML. An example of such a YAML Namespace Definition can be accessed here: Standard Lamp Device YAML Namespace Definition[14].

Although the format of the YAML Namespace Definition is self-explicatory, here a few comments:

Left/right Columns:

The right column as an 'internal name', which may be used internally in the implementation of the Device Simulator code, but not necesarily. The right column is the name in the PLC namespace, the HW address. It is possible to define a specific data type for a name, by adding it in parentheses after the name, e.g. "StatCounter: stat.nCounter(UInt32)" (see list of supported types below). Default mapping of data types:

- "b<name>" -> Boolean.
- "n<name>" -> Int32.
- "lr<name>" -> Float.
- "s<name>" -> String.

The complete path is created by prepending "MAIN." + <device name>, e.g. "Lamp1" + the HW address. An example of a complete name, as generated in the OPC UA namespace profile is, e.g.:

"ns=4;s=MAIN.Shutter1.stat.sStatus"

[11] https://gitlab.eso.org/ifw/ifw-hl/-/blob/master/fcf/devsim/lamp/src/fcfDevsimLamp/devsimLamp.py
[12] https://gitlab.eso.org/ifw/ifw-hl/-/blob/master/fcf/devsim/lamp/src/fcfDevsimLamp.py
[13] https://gitlab.eso.org/ifw/ifw-hl/-/blob/master/fcf/devsim/lamp/resource/config/fcf/devsim/lamp/lamp.scxml.xml
[14] https://gitlab.eso.org/ifw/ifw-hl/-/blob/master/fcf/devsim/lamp/resource/config/fcf/devsim/lamp/lampNamespace.yaml

The “ns=4” indicates OPC UA namespace 4. For the moment, the convention is to use namespace 4. The “s=” that it is a string node ID, as opposed to an integer node ID. The “MAIN” is a convention, decided for the ELT ICS PLC code. Both namespace and the “MAIN” prefix may be made configurable, if necessary.

RPC Method Calls:

RPC method calls are defined as:

Rpc<Operation>: rpc.<Operation>([I:<Variable>(<type>)][,] O:<Variable>(<type>)])

whereby “I:” means input variable and “O:”, imaginatively, output value. The following types are supported:

- Boolean
- SByte
- Byte
- Int16
- UInt16
- Int32
- UInt32
- Int64
- UInt64
- Float
- Double
- String
- DateTime
- ByteString

The “ctdGenOpcuaProfile” tool, is invoked as follows on the YAML Namespace Definition:

```
$ ctdGenOpcuaProfile --device Shutter3 --name-mapping shutterNamespace.yaml
<?xml version="1.0" encoding="utf-8"?>
<UANodeSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
           xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

(continues on next page)

(continued from previous page)

```
  <NamespaceUris>
    <Uri>http://www.eso.org/xmlShutter3/</Uri>
  </NamespaceUris>

  <UAObject NodeId="ns=1;s=PLC1" BrowseName="1:PLC1">
    <DisplayName>PLC1</DisplayName>
    <References>
      <Reference ReferenceType="HasTypeDefinition">i=58</Reference>
      <Reference ReferenceType="Organizes">ns=4;i=20737</Reference>
      <Reference ReferenceType="Organizes" IsForward="false">i=85</Reference>
    </References>
  </UAObject>
  ...
```

The output can be piped into a file, used as part of the configuration for the Device Simulator.

10.4.1 Example

```
$ fcfDevsimShutter --port 7677 --cfg fcf/devsim/shutter/shutter1.yaml --use-ext-
↪ip --log-level DEBUG --verbose
2018-12-04 12:31:52,821:INFO:DevSim:MainThread:deviceSimulatorBase:557:execute:␣
↪Setting up OPC UA server ...
Listening on 134.171.2.213:7577
2018-12-04 12:31:54,411:INFO:DevSim:MainThread:deviceSimulatorBase:567:execute:␣
↪Setting up OPC UA server - done
2018-12-04 12:31:54,411:INFO:DevSim:MainThread:deviceSimulatorBase:574:execute:␣
↪Loading configuration: fcf/devsim/devsimShutter/shutter1.yaml
2018-12-04 12:31:54,413:INFO:DevSim:MainThread:deviceSimulatorBase:578:execute:␣
↪Loading OPC UA namespace definition: fcf/devsim/devsimShutter/
↪shutter1Namespace.xml
2018-12-04 12:31:54,484:INFO:DevSim:MainThread:deviceSimulatorBase:582:execute:␣
↪Parsing OPC UA node info
2018-12-04 12:31:54,639:INFO:DevSim:MainThread:deviceSimulatorBase:442:set_node_
↪permissions_: Setting node permissions
2018-12-04 12:31:54,667:INFO:DevSim:MainThread:deviceSimulatorBase:454:install_
↪data_ch_subscr_: Installing data change subscription handlers
2018-12-04 12:31:54,670:INFO:DevSim:MainThread:deviceSimulatorBase:464:install_
↪rpc_methods_: Installing RPC method handlers
2018-12-04 12:31:54,670:INFO:DevSim:MainThread:deviceSimulatorBase:472:install_
↪rpc_methods_: Installing RPC method handler for: self.RPC_GetNamespace
...
2018-12-04 12:31:54,673:INFO:DevSim:MainThread:deviceSimulatorBase:472:install_
↪rpc_methods_: Installing RPC method handler for: self.RPC_Init
2018-12-04 12:31:54,673:INFO:DevSim:MainThread:deviceSimulatorBase:499:gen_opcua_
↪state_node_ids_: Generating OPC UA state machine node IDs
2018-12-04 12:31:54,673:INFO:DevSim:MainThread:deviceSimulatorBase:145:init_
↪device_parameters_: Initializing device parameters
```

(continues on next page)

(continued from previous page)

```
2018-12-04 12:31:54,676:INFO:DevSim:MainThread:deviceSimulatorBase:593:execute:␣
↪SCXML state machine model: fcf/devsim/devsimShutter/scxmlShutter.xml
2018-12-04 12:31:54,676:INFO:DevSim:MainThread:stateMachine:62:__init__: Loading␣
↪SCXML model: fcf/devsim/devsimShutter/scxmlShutter.xml
2018-12-04 12:31:54,682:INFO:DevSim:MainThread:stateMachine:73:__init__: Status:
2018-12-04 12:31:54,682:INFO:DevSim:MainThread:deviceSimulatorBase:598:execute:␣
↪Serving ...
2018-12-04 12:31:54,683:INFO:DevSim:MainThread:stateMachine:86:run: Starting␣
↪execution of /home/jknudstr/ROOTS/INTROOT_eltdev26/resource/config/fcf/devsim/
↪devsimShutter/scxmlShutter.xml
2018-12-04 12:31:54,685:INFO:DevSim:MainThread:stateMachine:91:run: Status: On␣
↪On::NotOperational On::NotOperational::NotReady
```

After generating the SCXML definition and OPC UA XML Profile, it is possible to start up the Device Server and connect to the Device Simulator OPC UA server and execute e.g. the provided RPC calls and read/write variables, e.g.:

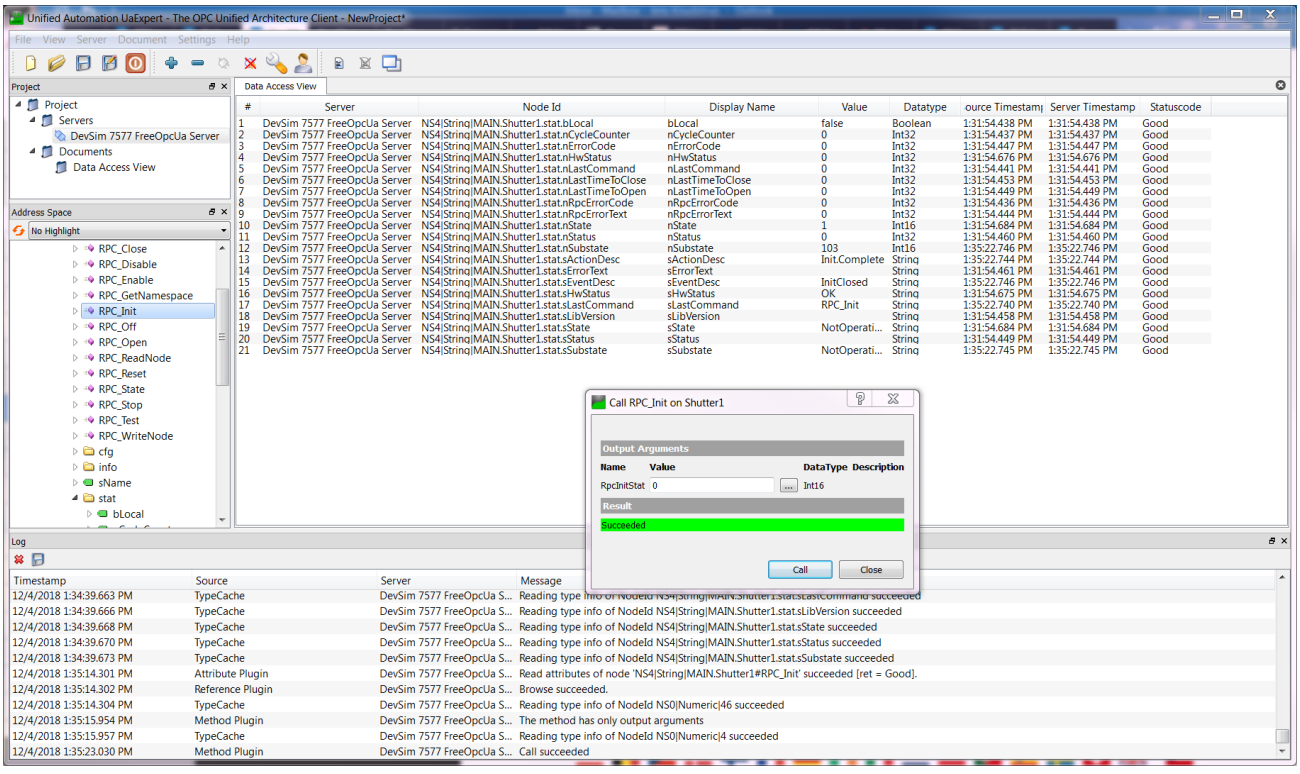


Fig. 10.1: Shutter Device Namespace in UaExpert

10.5 Device Simulators for Standard Devices

This section provides specific details about the various Device Simulators provided for Standard Devices.

Since the Device Simulators are used to emulate the PLC Controllers, they implement the same behavior and therefore, their behavior is described in the PLC section of the FCF user manual and not repeated here. Here only specific properties of the Device Simulators are mentioned.

The following common configuration parameters are supported by all Device Simulators:

Parameter	Type	Description
OpcUaProfile	**String**	**Name of OPC UA Profile XML document, e.g. "fcf/devsim/devsimLamp/lamp1Namespace.xml".**
StateMachineScxml	**String**	**Name of SCXML definition (document), e.g. "fcf/devsim/devsimLamp/scxmlLamp.xml".**
CfgSimAcceleration	**Float**	**Factor, which can be used to tune the execution speed of actions in the Device Simulator, globally.**
AutoEnterOp	**Boolean**	**Enter Operational State automatically when starting up.**
CfgSimDelay	**Float**	**General delay that can be applied when a delay is needed to make the simulation more realistic [s].**
CfgLocal	**Boolean**	**Device simulator will emulate Localo Mode at start-up if True.**
UpdateFrequency	**Float**	**Frequency for the internal simulation loop (thread) in Hz. Default value is 10 Hz.**

Note: In general the Device Simulators do not support 'engineering mode', but are mostly dedicated for the 'common usage', e.g. from Sequencer Templates and standard operation of the instrument software. This means that many 'low level/engineering parameters' and RPC calls are not supported.

10.5.1 ADC Device Simulator

The state machine of the ADC Device Simulator is the same as for the (Standard PLC ADC Device (tracking devices)).

The ADC Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgMotor1	**String**	**Configuration for the Motor DevSim for axis 1, e.g. "fcf/devsim/adc/adc1Motor1.cfg.yaml".**
CfgMotor2	**String**	**Configuration for the Motor DevSim for axis 2.**
Motor1Step	**Float**	**Factor applied when calculating the next position for axis 1 for each simulated cycle of the ADC. The higher this factor, the faster the motor moves.**
Motor2Step	**Float**	**Factor applied when calculating the next position for axis 2 for each simulated cycle of the ADC. The higher this factor, the faster the motor moves.**

Note: The ADC Device Simulator does not support 'engineering mode' and can therefore not be used together with the FCF Python Motor GUI ("pymotgui"). In particular, the following RPC calls are not supported: RPC_MoveAbs, RPC_MoveAngle, RPC_MoveRel, RPC_MoveVel.

10.5.2 DROT Device Simulator

The state machine of the DROT Device Simulator is the same as for the (Standard PLC DROT Device (tracking devices)).

The DROT Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgMotor	**String**	**Configuration for the Motor DevSim for internally controlled axis, e.g. "fcf/devsim/adc/adc1Motor.cfg.yaml".**
MotorStep	**Float**	**Factor applied when calculating the next position for axis for each simulated cycle of the DROT. The higher this factor, the faster the motor moves.**

Note: The DROT Device Simulator does not support 'engineering mode' and can therefore not be used together with the FCF Python Motor GUI ("pymotgui"). In particular, the following RPC calls are not supported: RPC_MoveAbs, RPC_MoveAngle, RPC_MoveRel, RPC_MoveVel.

10.5.3 Lamp Device Simulator

The state machine of the Lamp Device Simulator is the same as for the (Standard PLC Lamp Device).

The Lamp Device Simulator defines the following configuration parameters:

Parameter	Type	Description
SimInitTime	Float	Time to spend for the initialisation [s].
CfgCoolDown	Float	Time to spend for the cool-down [s].
CfgInitialState	Boolean	Initial state to assume after enabling the device (True = On).
CfgMaxOn	Float	Maximum time the lamp is allowed to remain switched on [s].
CfgWarmUp	Float	Time spent for the warm-up phase [s].

10.5.4 Motor Device Simulator

The state machine of the Motor Device Simulator is the same as for the (Standard PLC Motor Device).

The Motor Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgVelocityError	Float	Simulated error in % to be applied to the simulated velocity.
CfgSimPosError	Float	Simulated positioning error in % to be applied to the simulated position.
CfgSimTolerance	Float	Tolerance to apply for considering simuilated position on target [UU].
CfgSimulated-StartPos	Float	Start position of motor after start-up of application.
CfgScaleFactor	Float	Scale factor to apply for converting between UU and encoder values [UU/enc]
CfgMinPosition	Float	Minimum position [UU].
CfgMaxPosition	Float	Maximum position [UU].
CfgTimeoutInit	Float	Timeout applied during initialisation [s].
CfgTimeoutMove	Float	Timeout applied while moving simulated axis [s].
CfgTime-outSwitch	Float	Timeout applied while moving simulated axis [s].
CfgDisableAfter-Move	Boolean	Disable the current on the simulated axis after completion of a movement.
CfgLocal	Boolean	Indicates Local Mode if True.
CfgDefaultVeloc-ity	Float	Default velocity to apply, e.g. during initialisation [UU/s].

10.5.5 Sensor Device Simulator

The Sensor Device Simulator defines the following configuration parameters:

Parameter	Type	Description
DiCh<i>Value	Boolean	Initial or static value of signal of given channel.
DiCh<i>Function	String	Function, with parameters to be invoked. For now only a square wave generator is provided "di_square_wave(period hi[s], period lo[s])".
AiCh<i>Value	Double	Initial or static user value of signal of given channel.
AiCh<i>Function	String	Function, with parameters to be invoked. For now only a sine wave generator is provided "ai_sine_wave(period[s], scale, offset)".

Note: More simulation functions may be added on request. A future extension may be to allow user provided simulation function on a plug-in basis.

10.5.6 Shutter Device Simulator

The state machine of the Shutter Device Simulator is the same as for the (Standard PLC Shutter Device).

The Shutter Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgInitialState	Boolean	Initial state to assume after enabling the device (True = Open).

10.5.7 Piezo Device Simulator

The state machine of the Piezo Device Simulator is the same as for the (Standard PLC Piezo Device).

The Piezo Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgInitialState	Boolean	Initial state to assume after enabling the device (True = Open).

10.5.8 Actuator Device Simulator

The state machine of the Actuator Device Simulator is the same as for the (Standard PLC Actuator Device).

The Actuator Device Simulator defines the following configuration parameters:

Parameter	Type	Description
CfgInitialState	**Boolean**	**Initial state to assume after enabling the device (True = Open).**

10.5.9 CCC Device Simulator

The CCC (Cabinet Cooling Controller) Device Simulator is a special one since it mimics the contents of the address space embedded in the ESO controller.

The CCC Device Simulator does not define any configuration parameter.

10.6 Developing a Device Simulator

The steps to implement a Device Simulator is as follows:

1. Generate the code of the Device Simulator from the template provided by FCF DevSim (see instructions below).
2. Define the state machine, generate the SCXML state chart document (see instructions above).
3. Define the namespace and OPC UA profile for the Device Simulator (see instructions above).
4. Adopt the code generated in step 1. according to the specific Device Simulator state machine and business logic (see instructions below).
5. Implement the necssary integration test cases to verify the basic functioning of the Device Simulator.

Note, it is important that the Device Simulator developed, emulates the PLC Controller relatively well for it to have a realistic behaviour.

It may not be necessary to implement all features of the PLC Controller, but the features needed to be able to execute the instrument integration tests, and to execute the Templates, both in simulation mode, without the availability of any hardware, shall be provided by the Device Simulator.

10.6.1 Generate the Code of the Device Simulator

The steps to generate the basic code are shown below. After generating the code, it will be necessary to adapt the specific code manually. The template provided is based on the Lamp Device Simulator described elsewhere in this document. After having generated the code for the new Device Simulator, a new instance of the Lamp Device Simulator is obtained as starting point for the development.

The steps to generate the basic code for the new Device Simulator are:

1. Enter the folder in the instrument source tree where the Device Simulator WTOOLS module shall be located.
2. Invoke Cookie-Cutter on the DevSim template.
3. Add the new Device Simulator module in the “wscript” file, found at the same level, if needed.
4. Build and install the instrument software.
5. Execute the generated Device Simulator.
6. Connect with an OPC UA client, e.g. UA Expert to verify that the Device Simulator is functioning properly.

In the following a live example of how to generate the code, is shown.

```
$ cookiecutter <path to ifw-hl>/fcf/devsim/templates/devsim_tpl/
device_name [device]: mydev
Device_name [Device]: Mydev
package_name [pkg]: mypkg
device_description [This the description of my device]: This is a test Device␣
 ↪Simulator

$ find mydev/
mydev/
mydev/resource
mydev/resource/config
mydev/resource/config/mypkg
mydev/resource/config/mypkg/devsim
mydev/resource/config/mypkg/devsim/mydev
mydev/resource/config/mypkg/devsim/mydev/mydev.scxml.xml
mydev/resource/config/mypkg/devsim/mydev/mydev1.yaml
mydev/resource/config/mypkg/devsim/mydev/mydev1Namespace.xml
mydev/resource/config/mypkg/devsim/mydev/mydevNamespace.yaml
mydev/src
mydev/src/mypkgDevsimMydev
mydev/src/mypkgDevsimMydev/__init__.py
mydev/src/mypkgDevsimMydev/devsimMydev.py
mydev/src/mypkgDevsimMydev/mydevDefines.py
mydev/src/mypkgDevsimMydev.py
mydev/wscript

# Add "mydev" in the "wscript" file at the same level as the "mydev" module.
```

(continues on next page)

(continued from previous page)

```
$ mypkgDevsimMydev --port 7777 --cfg mypkg/devsim/mydev/mydev1.yaml --use-ext-ip␣
↪--log-level INFO --verbose
2020-02-04 11:53:49,780:INFO:SmOpcUaSrv:MainThread:serverBase:501:execute:␣
↪Setting up OPC UA server ...
Endpoints other than open requested but private key and certificate are not set.
Listening on 134.171.2.213:7777
2020-02-04 11:53:50,955:INFO:SmOpcUaSrv:MainThread:serverBase:512:execute:␣
↪Setting up OPC UA server - done
2020-02-04 11:53:50,955:INFO:SmOpcUaSrv:MainThread:serverBase:519:execute:␣
↪Loading configuration: /home/jknudstr/ROOTS/INTROOT_eltdev26/resource/config/
↪mypkg/devsim/mydev/mydev1.yaml
2020-02-04 11:53:50,957:INFO:SmOpcUaSrv:MainThread:serverBase:524:execute:␣
↪Loading OPC UA namespace definition: mypkg/devsim/mydev/mydev1Namespace.xml
#...
2020-02-04 11:53:51,
↪116:INFO:SmOpcUaSrv:MainThread:mypkgDevsimMydev:34:initialise: Initialising␣
↪Device Simulator
2020-02-04 11:53:51,116:INFO:SmOpcUaSrv:MainThread:devsimMydev:286:initialise:␣
↪Initialising Device Simulator
2020-02-04 11:53:51,117:INFO:SmOpcUaSrv:MainThread:devsimMydev:305:initialise:␣
↪Local Mode is: False
2020-02-04 11:53:51,118:INFO:SmOpcUaSrv:MainThread:serverBase:124:initialise:␣
↪Initialising server
2020-02-04 11:53:51,119:INFO:SmOpcUaSrv:MainThread:serverBase:540:execute: SCXML␣
↪state machine model: mypkg/devsim/mydev/mydev.scxml.xml
2020-02-04 11:53:51,125:INFO:SmOpcUaSrv:MainThread:serverBase:545:execute: Time␣
↪for initialising: 1.346s
2020-02-04 11:53:51,125:INFO:SmOpcUaSrv:MainThread:serverBase:546:execute:␣
↪Serving ...
```

Connect to Device Simulator e.g. vie UAExpert. Read/write nodes, execute RPC calls:

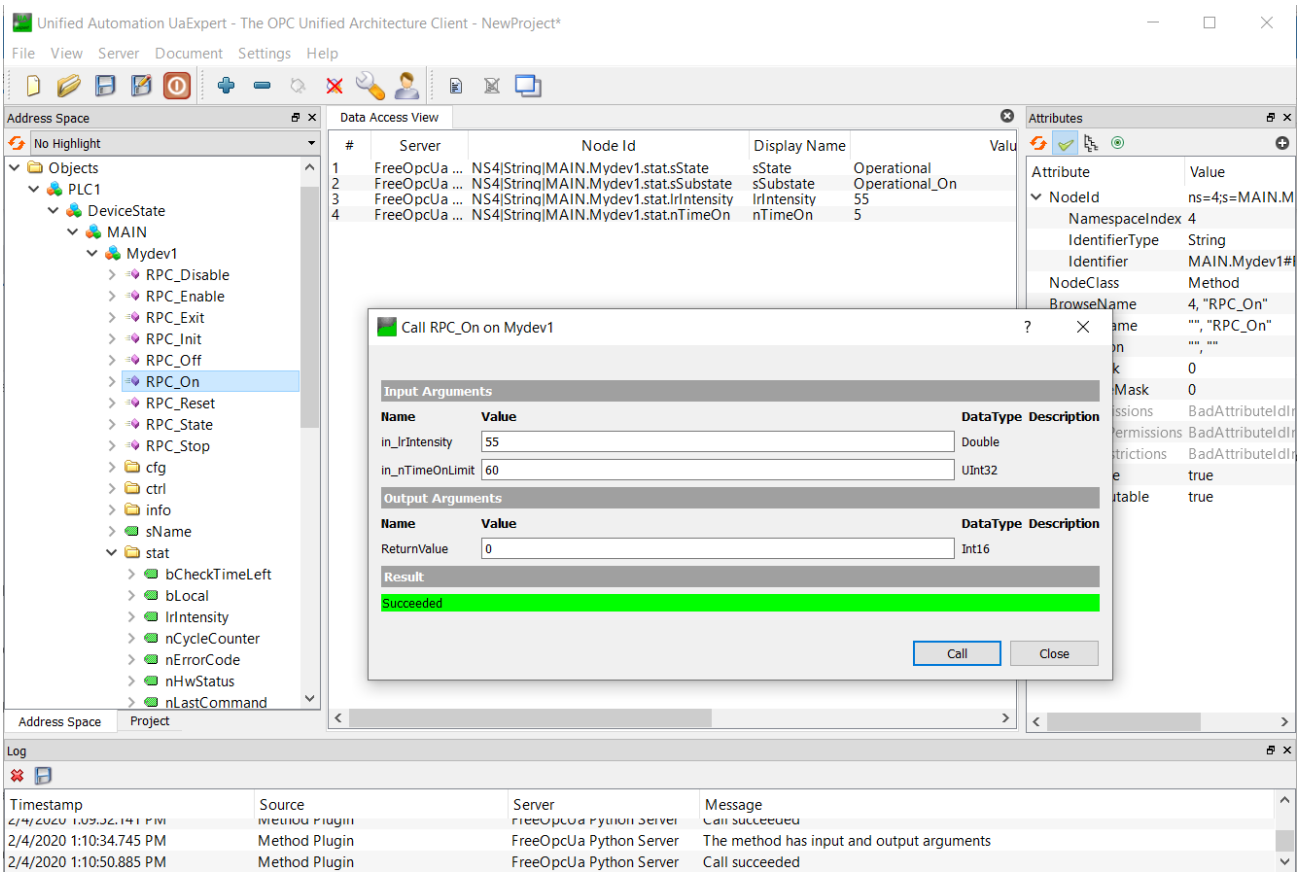


Fig. 10.2: Namespace of generated Device Simulator in UaExpert.

Note, the name “<device>1”, here “Mydev1” is merely a default name. However, often multiple instances of a device will be named e.g. “Motor1”, “Motor2”, etc. This is not a prerequisite though, and a free name may be chosen.

To do this, a new OPC UA Profile, here “mydev/resource/config/mypkg/devsim/mydev/mydev1Namespace.xml”, must be generated for each instance of the new Device Simulator deployed. It is possible to change the name in the OPC UA Profile using the tool “tooReplace”.

In addition a configuration shall be prepared (here “mydev/resource/config/mypkg/devsim/mydev/mydev1.yaml”)

10.6.2 Adapt the Generated Code of the Device Simulator

In this section some guidelines for implementing the specific code of the new Device Simulator are provided.

When executing Cookie Cutter, the following Python source files are generated:

1. A source file containing constant declarations, mostly the constants defined in the PLC code (here “mydev/src/mypkgDevsimMydev/mydevDefines.py”).

2. The actual Device Simulator source code (various classes implementing the logic of the simulator; here “mydev/src/mypkgDevsimMydev/devsimMydev.py”).
3. The instantiation of the Device Simulator, generating the executable (here “mydev/src/mypkgDevsimMydev.py”).

In the following the source files mentioned in the points 2. and 3. above are described in more details.

Device Simulator Class File (here “mydev/src/mypkgDevsimMydev/devsimMydev.py”):

The Device Simulator Class file contains the following main parts:

1. The classes implementing the Activities of the State Machine Chart. These are running in threads in the simulator. Example “Class ActivityInitialising()”.
2. The Action Manager Class (“class ActionMgr”), which defines the actions implemented by the Device Simulator and creates and registers the Activity Classes in the SCXML Engine.
3. The Device Simulator Class implementing the logic of the simulator. In the example above “class DevsimMydev()”. This shall be derived from the base class “devsimBase.DeviceSimulatorBase”. The Device Simulator Class provides the following methods to initialise the simulator and implementing the behaviour:

   3.1. The constructor defining basic properties of the simulator, defining a mapping between the textual and numerical representation of the states defined and defining/instantiating/initialising other members of the simulator class.

   3.2. An “initialise()” method which starts the Simulation Thread (see below) and sets initial valus for various parameters in the OPC UA namespace, either from the configuration or constant values.

   3.3. The Simulation Thread (“sim_thr_user()”), which can be used to implement dynamic behaviour of the simulator, e.g. for a tracking device, simulating that it is tracking.

   3.4. A callback function, which is invoked when changes are introduced in the OCP UA namespace in the simulator (“data_change_handler()”).

   3.5. The methods implementing the RPC calls of the OPC UA interface.

In order to read and write from/to the internal OPC UA namespace, the methods “<simulator class>.read_node()” and “<simulator class>.write_node()” can be used.

Device Simulator Instantiation (here “mydev/src/mypkgDevsimMydev.py”):

The Device Simulator Instantiation source file contains the main function to start up and execute the simulator server process.

In the template an option is provided to implement specific behaviour at instatiation level. If not used, the class definition contained in the instantiation source file, may be omitted.

11 Engineering Interfaces

11.1 Device Manager GUI

The *Device Manager* provides an engineering interface that enables an easy control and monitor of the devices managed by the server. Each device type has a dedicated widget and the list of widgets is built reading the configuration at startup. Devices can be controlled individually using the *Setup* button within each device widget or collectively using the *Setup* button at the bottom of the GUI.

Note: The menu *FCS* enables to control the main states of the server. The server can be also restarted from this menu.

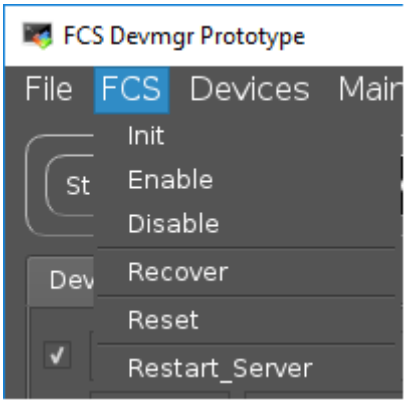


Fig. 11.1: *FCF* menu.

Note: The menu *Devices* enables to control directly one or more device controllers running in the PLC. Devices can be either ignored or simulated as well from this menu.

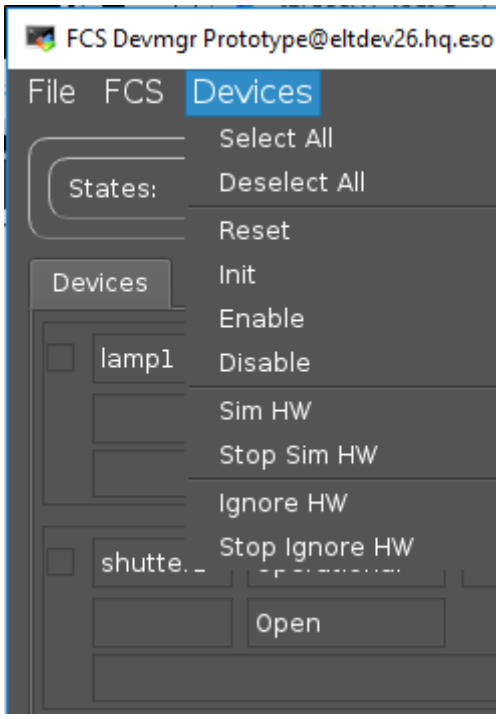


Fig. 11.2: *Devices* menu.

At the bottom, a command window is included reporting the commands sent and the replies received by the GUI.

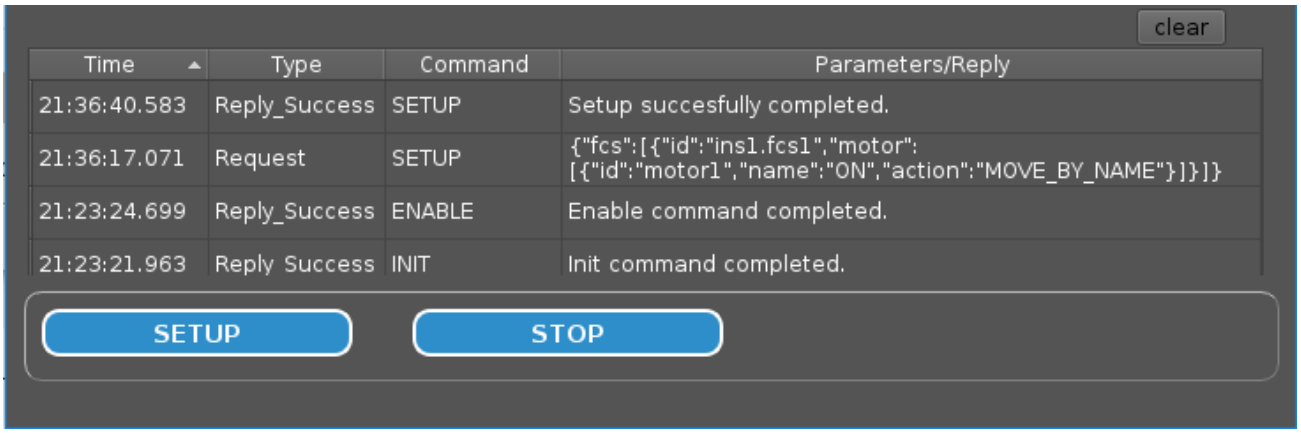


Fig. 11.3: Command window widget.

The GUI does not prevent the execution of parallel *Setup* commands. This is achieved by creating a new thread per each command.

```
$ fcfGui -h
Usage: fcfGui [options]
Description: This is the generic FCF GUI

Options:
  -h, --help       Displays help on commandline options.
  --help-all       Displays help including Qt specific options.
  -v, --version    Displays version information.
  -u, --uri <uri>  Option is: uri
```

Warning: Unlike the previous version, the GUI gets the configuration directly from the server.

An example how to start the `fcfgui` from the command line is shown below.

```
$ fcfGui --uri `get-uri fcs-req`
```

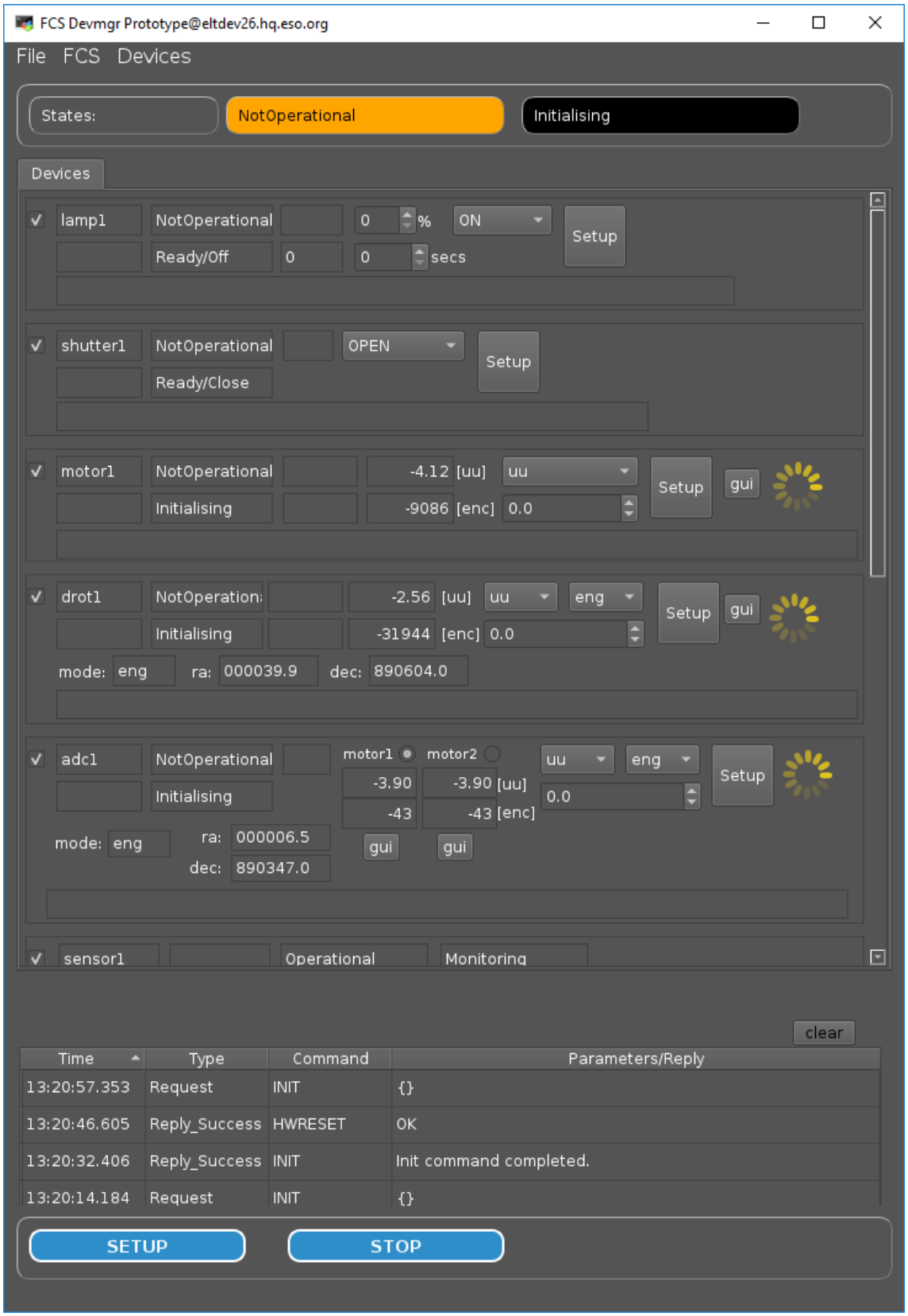


Fig. 11.4: Device Manager Engineering Graphical Interface.

11.1.1 Widgets

Device widgets share some common features.

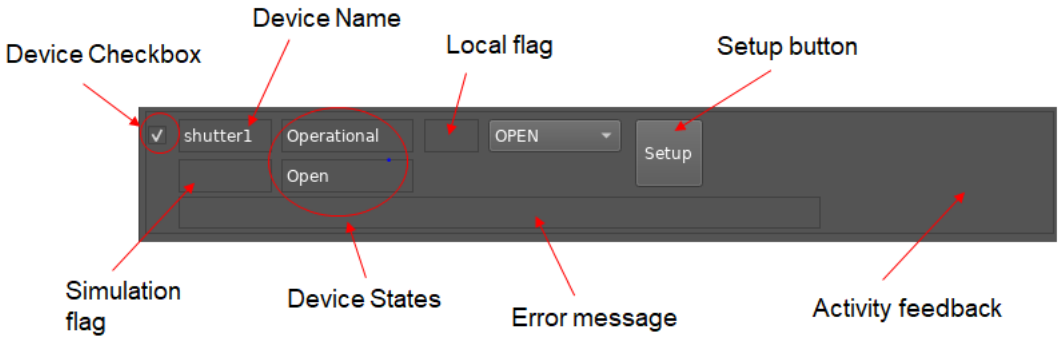


Fig. 11.5: Common Widget Elements.

Selection Check-box

This checkbox shall be used to select/unselect a device. The global Setup button at the bottom is applied only to the selected devices. In the same way, actions in the `Devices` menu are also affecting the selected devices.

Device Name

Here is displayed the name of the device taken from the FCF configuration.

Simulation Flag

This flag indicates when the device is set into simulated. This means that Device Manager will use the simulation address to connect to the device controller.

Local Flag

This flag indicates when the device is set into local mode. Setup actions are rejected by the controllers when they are in local mode.

Device States

These two fields present the state and substate of the controller in the PLC. This information is read by the Device Manager. When Device Manager is not in Ready or Operational state, these values are undefined.

Setup Button

Each device with the exception of sensors can be setup individually from each widget.

Error Message

Errors in the controller will be presented in this field.

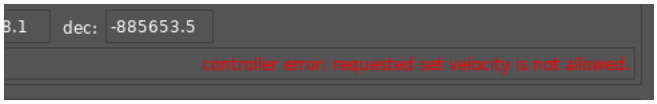


Fig. 11.6: Controller error message.

Activity Feedback

This is a coloured and spinning feedback allowing the user to have a quick overview of any ongoing activity in the controller associated to the device, for instance if a derotator is tracking, the activity feedback will show it in magenta. This is faster to understand than reading each individual substate. This might be useful during initialisation phase.

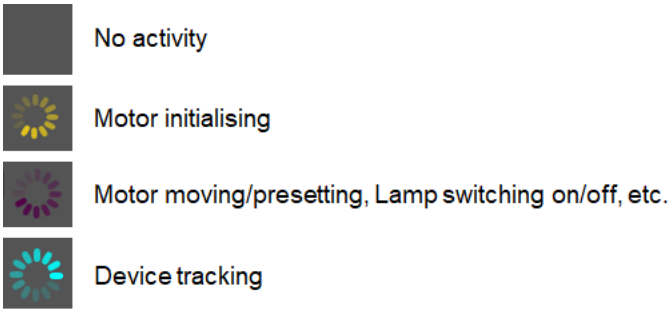


Fig. 11.7: Device activity feedback.

Motor Widget

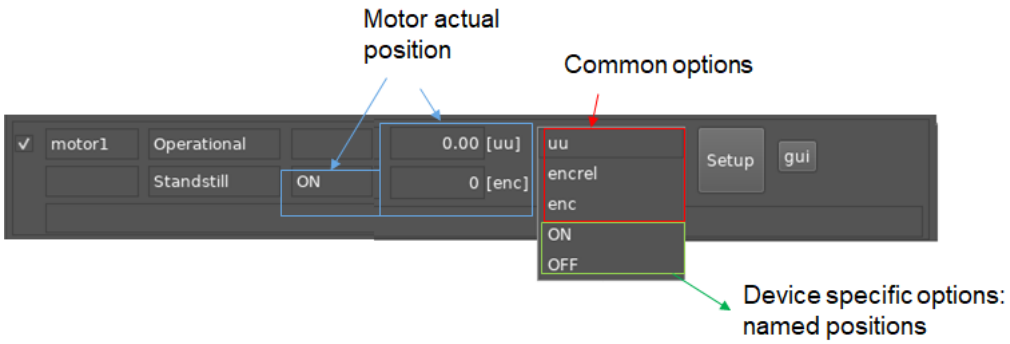


Fig. 11.8: Motor Widget.

The motor widget allows moving a motor in:

1. Absolute user units (uu) and encoders (enc)
2. Relative encoders (encrel)
3. By name positions

Options 1 and 2 are common for all motor widgets while named position (3) may change per motor device. The named positions are read by the widget from the FCF configuration at the startup. The mapping between named positions and user units is done by the Device Manager.

Drot Widget

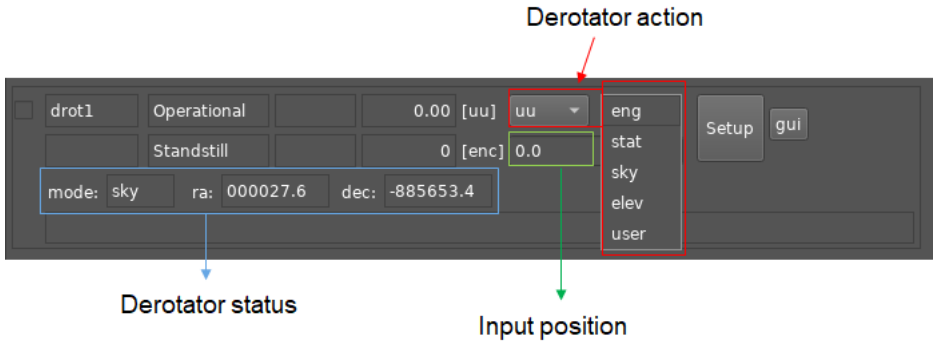


Fig. 11.9: Drot Widget.

The drot widget allows selecting the operation mode and the position angle. There are three tracking modes supported by a drot device: Sky, Elevation (Elev) and User Specific (User). Additionally, a

drot device can be controlled as a normal motor by selecting the Engineering mode (eng), this means moving it by user units or encoders.

Adc Widget

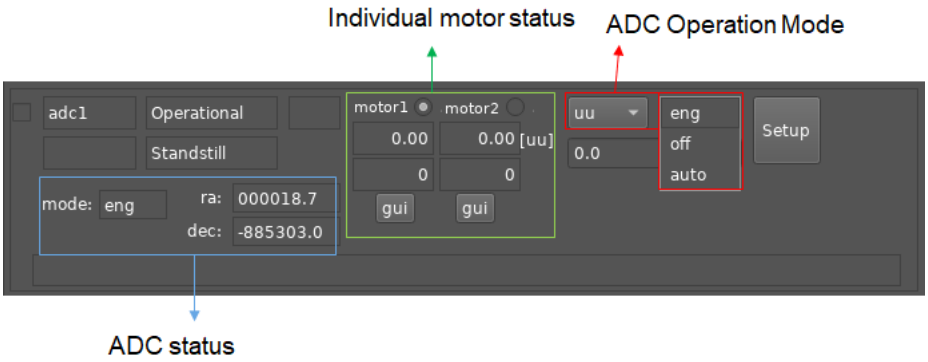


Fig. 11.10: ADC Widget.

The ADC widget allows to control the operation mode of an ADC device. There are two defined modes supported by an ADC: Stationary (off) and Automatic (auto). Additionally, an ADC device can control each individual motor by selecting the Engineering mode (eng). The ADC status reports the actual run-time configuration of the device.

11.2 Motor Engineering GUI

The *Motor* device provides an additional engineering interface offering more details for the control and monitoring of motorized devices (`pymotgui`). With this GUI it is possible to see the motor position or the activation of signals during initialisation. The motor can be controlled in Position or in Velocity mode.

The `pymotgui` can be launched directly from the command line or from each *Motor* widget by pressing the gui button.

```
$ motgui
usage: pymotgui [-h] -d DEVICE -a ADDRESS -p PREFIX [-n NS]
            [-l {INFO,DEBUG,ERROR}]

optional arguments:
-h, --help            show this help message and exit
-d DEVICE, --device DEVICE
                      Set motor device.
-a ADDRESS, --address ADDRESS
```

(continues on next page)

(continued from previous page)

```
                    Set PLC address, e.g. opc.tcp://134.171.59.98:4840
-p PREFIX, --prefix PREFIX
                    Set motor prefix, e.g. MAIN.Motor1
-n NS, --ns NS        Set OPCUA namespace, e.g. 4
- l {INFO,DEBUG,ERROR}, --loglevel {INFO,DEBUG,ERROR}
                    set the logging level: INFO|DEBUG|ERROR
```

An example how to start the `pymotgui` from the command line is shown below.

```
$ pymotgui -d motor1 -a opc.tcp://127.0.0.1:7578 -p MAIN.Motor1
```

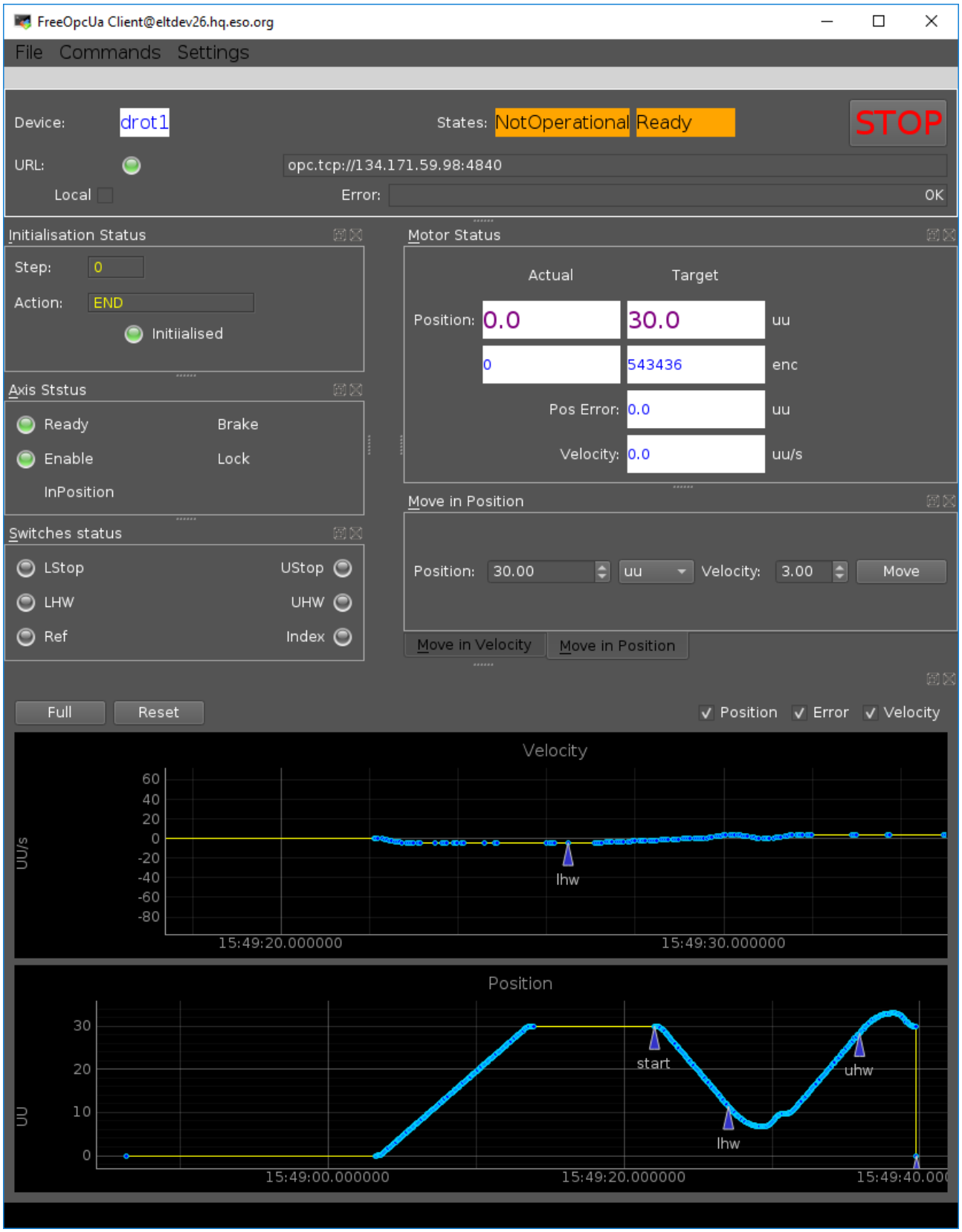


Fig. 11.11: Motor Graphical Interface.

11.2.1 Initialisation Markers

The Motor GUI uses markers to identify the time when motors reach a switch during the initialisation sequence. Two markers indicate the start and the end of the sequence, the others will depend of the switches reached during the sequence, see an example in the image below. In this example it was marked lower and higher hardware limits.

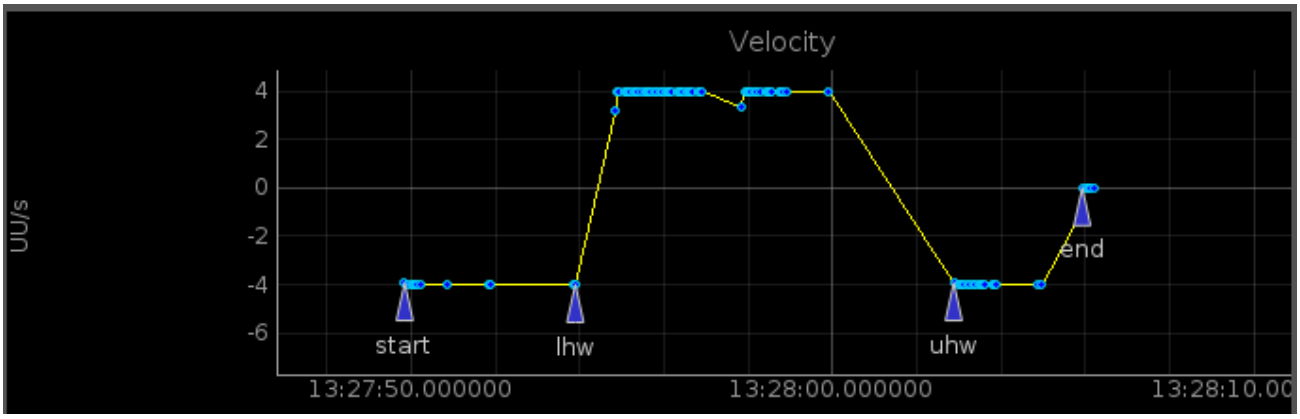


Fig. 11.12: Example of Initialisation Markers.

11.2.2 Exporting Plotting Data

The plotting data can be exported in a CSV format. This can be achieved by doing the following steps:

- Press mouse right-button and select “Export. . . ”
- Select “CSV from plot data” and press “Export”, see image below.
- Define the name of the file, e.g. myplotdata.csv and press “Save”.

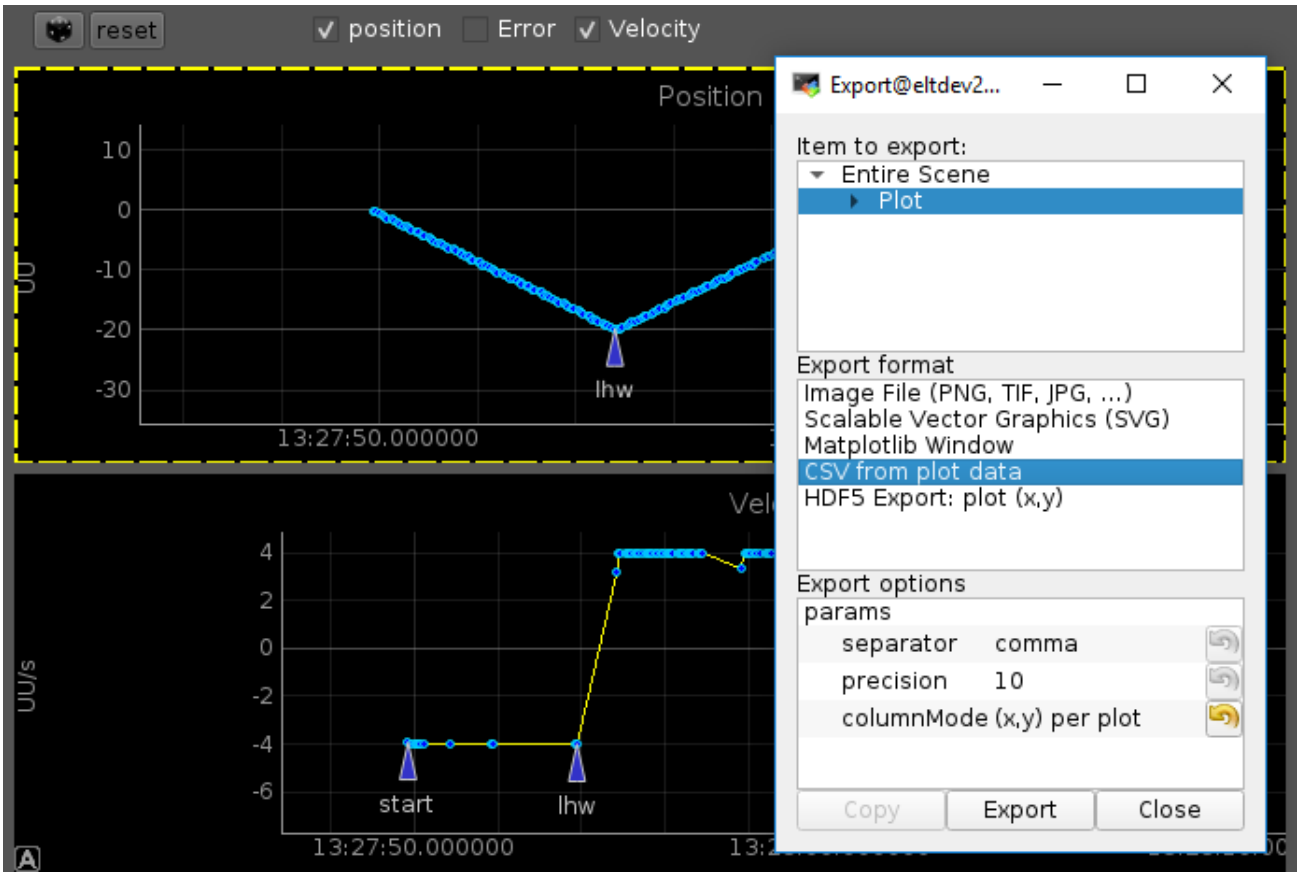


Fig. 11.13: Exporting Plotting Data.

Warning: The marker information is not included in the exported data.

12 Programming Guide

12.1 Device Manager Extensions

Having type safe interfaces as defined by CII ICD XML enables to improve runtime robustness of our applications. However it is also makes the extendability harder to achieve. To simplify adding instrument specific extensions to the Device Manager, we have provided a template that can be used as starting point to implement special devices along with some companion modules.

12.1.1 Template

You should create your software based on the provided project template by following the procedure in the Getting Started guide here[15]

12.1.2 Top Directory Structure

```
<root>                  # Generated project directory
├── resource            # directory containing the resources
└── <prefix>-ics        # WAF project
    ├── <component>     # Generated FCS
    ├── seq
    ├── <prefix>stoo
    └── wscript
```

12.1.3 Component Directory Structure

Here it will be reported the specific parts about FCS in the generated project. The generated structure resembles the structure of the FCF component.

```
<component>
├── devices             # Custom Device
├── devsim              # Simulator for custom device
├── fcsclib             # Custom Python client library
├── gui                 # Custom GUI
├── LCS                 # Example PLC Project (VS)
├── server              # Custom server
└── wscript
```

Note: This FCS structure has been simplified in version 3. There is no need of having a custom CII interface for the setup command. The state machine is the same as the default server and the only difference is the handling of a custom device library.

[15] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/guide.html

Note: All custom devices will use the CUSTOM device type of the existing FCF XML interface.

12.1.4 Custom Device

The generated code includes the implementation of a dummy custom device that can be used as starting point to develop instrument specific ones. This device contains the intelligence to deserialize the custom setup command payload in method Setup. Users will have to modify this method and method IsSetupActive to adapt to their specific needs.

Warning: The serialization of the setup command for custom devices is done in JSON format.

```
void Mirror::Setup(const std::any& payload) {
    RAD_LOG_TRACE();
    RAD_LOG_INFO() << "Processing Setup command ...";
    if (!m_config->GetIgnored()) {
      auto fcf_vector = std::any_cast<std::vector<std::shared_ptr
↪<fcfif::SetupElem>>>(&payload);
      for (auto it = fcf_vector->begin(); it != fcf_vector->end(); it++) {
           auto setup_elem = *it;
           RAD_LOG_INFO() << "ID: " << setup_elem->getId();
           // ignore other shutters
           if (IsMsgForMe(setup_elem->getId()) != true) {
               continue;
             }
           auto fcs_union = setup_elem->getDevice();
           if (fcs_union->getDiscriminator() != ::fcfif::DeviceType::CUSTOM) {
               RAD_LOG_ERROR() << "[" << m_config->GetName() << "] "
                               << "Setup is for me but device type is not␣
↪correct?";
               continue;
             }

           auto custom = fcs_union->getCustom();
          RAD_LOG_DEBUG() << "[" << m_config->GetName() << "] "
                          << "Setup ID: " << setup_elem->getId();
          std::string params = boost::replace_all_copy(custom->getParameters(), "
↪'", "\"");
          RAD_LOG_INFO() << "Setup buffer: " << params;

          //@TODO: Add handling of setup parameters
          EncDec data = nlohmann::json::parse(params);
          // Optional parameter
          RAD_LOG_INFO() << "Action: " << data.get_action();
          if (data.has_piston()) {
```

(continues on next page)

(continued from previous page)

```
            RAD_LOG_INFO() << "Piston: " << data.get_piston();
          }
        RAD_LOG_INFO() << "Tip: " << data.get_tip();
        RAD_LOG_INFO() << "Tilt: " << data.get_tilt();

        if (data.get_action() == "MOVE") {
          RAD_LOG_INFO() << "[" << m_config->GetName() << "] "
                         << "Executing RPC_PING ...";
          m_lcs_if->Ping();
          RAD_LOG_INFO() << "[" << m_config->GetName() << "] "
                         << "Successfull call of Mirror ping: ";
          }
        }
    }
}
```

12.1.5 Testing

The generated device module contains a set unit test that can be extended accordingly. After the generation, all unit test shall pass based on the actual implementation.

```
$ cd <component>devices
$ waf test --alltests
[==========] Running 26 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 9 tests from TestMirror
[ RUN      ] TestMirror.Ctor
...
[----------] Global test environment tear-down
[==========] 26 tests from 3 test cases ran. (2545 ms total)
[  PASSED  ] 26 tests.
```

12.1.6 Custom Simulation

In oder to allow the testing of the dummy device, a device simulator is generated and can be used for testing purposes. The simulator implements the RPC_Ping dummy method used by the custom device as an action of the Setup command.

12.2 Extending JSON schema

In order to validate the setup payload before sending it to the server, applications shall extend the FCF schema for custom devices. We provide an example for this in the template for the Mirror device.

```
{
      "type": "object",
      "properties": {
        "action": {
          "type": "string",
          "enum": ["MOVE" ],
          "description": "Mirror action."
        },
        "tip": {
          "type": "number",
          "description": "tip."
        },
        "tilt": {
          "type": "number",
          "description": "tilt."
        },
        "piston": {
          "type": "number",
          "description": "piston."
        }
      },
      "required": ["action","tip","tilt"]
}
```

Applications required to define their own python library where to compose the FCS schema adding the new one. There are probably different ways to achieve this, we provide an example below where the composition in done in python.

```
""" Load basic schema for all standard devices """
self._schema = json.loads(json_obj.SETUP_SCHEMA)

""" Change the schema to add new device """
""" add the new definition """
self._schema['definitions']['mirror'] = json_obj.load_json_string(SetupBuffer.
↪MIRROR_SCHEMA)
""" add new element the array """
self._schema['definitions']['param']['oneOf'].append(json_obj.load_json_string('{
↪"required": [ "mirror"]}'))
self._schema['definitions']['param']['properties']['mirror'] = json_obj.load_
↪json_string('{"$ref": "#/definitions/mirror"}')
```

13 Getting Started

13.1 Log-in

Login to a standard ELT machine.

13.2 IFW Software

If not yet done, retrieve and install the complete ICS Framework from the ESO RPMs repository. For more details, please have a look to the installation procedure here[16]

You should create your software based on the provided template by following the procedure in the Getting Started guide here[17]

This guide assumes you have followed all the steps in the above guide. The examples assume you selected the instrument as `tins`, the FCS component as `fcs` and the custom device as `mirror`. You can adapt the configuration according to your needs.

13.3 Database Server

The present version of the *Device Manager* uses Redis as the database engine to store run-time configuration (Redis documentation[18]).

The DB server shall be running after following the general Getting Started guide here[19]

The data is stored in the DB using a list of keyword/values. Each keyword has a hierarchical name that helps to idenfify its context, for instance the keys associated to a *Device Manager* start with the <server id> defined in the FCF configuration.

The ELT development environment provides a DB browser tool (dbbroser) that can be used to monitor the database keywords in an easy way similar to the *ccseiDb* tool in the VLT project.

[16] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/installation.html
[17] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/guide.html
[18] https://redis.io/
[19] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/guide.html

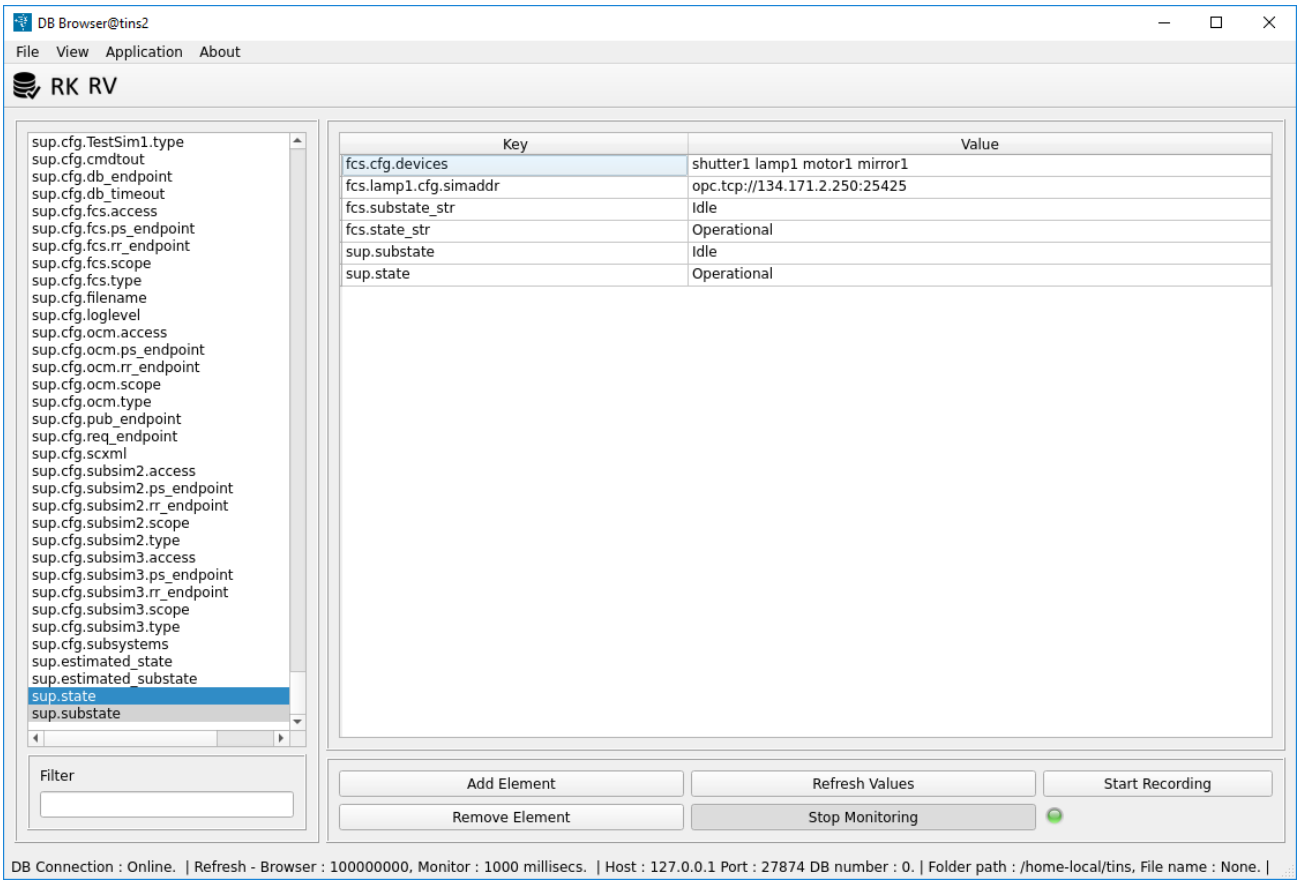


Fig. 13.1: Dbbrowser tool screenshot showing Device Manager attributes.

```
dbbrowser &
```

Note: To get list of attributes in the dbbrowser is needed to configure the connection first. Menu *Application* option *Open*. Consult the dbbrowser User Manual in case of problems.

Note: The attributes will be populated only when the *Device Manager* server is started.

13.4 FCS Configuration

The project template includes a sample of a FCF configuration. The generated directory contains a fully working waf project with a custom device to be used as starting point for the development. After executing the *cookiecutter* command with the provided template, you can build, install and deploy the provided example.

Please follow the instructions in the general Getting Started guide here[20]

The `Device Server` requires a configuration file including all the relevant information for the server. A pre-defined configuration has been created as part of the generation process. This configuration include three standard devices plus a the custom device.

Generated server configuration: <ins id>/resource/nomad/<component>.yml.tpl. This files contains the Nomad tags so it should not be used directly but through a nomad job command.

```
server_id           : 'fcs'

fcs:

    req_endpoint    : "zpb.rr://{{ range service "fcs-req" }}{{ .Address }}:{{ .
↪Port }}{{ end }}/"
    pub_endpoint    : "zpb.ps://{{ range service "fcs-pub" }}{{ .Address }}:{{ .
↪Port }}{{ end }}/"
    db_endpoint     : "{{ range service "tinsredis" }}{{ .Address }}:{{ .Port }}{{
↪{ end }}"

    db_timeout      : 2
    scxml           : "fcf/devmgr/server/sm.xml"
    dictionaries    : ['dit/stddid/primary.did', 'fcf/devmgr/server/fcf.did']
    fits_prefix     : "FCS1"

    devices         : ['shutter1', 'lamp1', 'motor1', 'mirror1']
    cmdtout         : 60000

shutter1:
type: Shutter
cfgfile: "local/shutter1.yml"

lamp1:
type: Lamp
cfgfile: "local/lamp1.yml"

motor1:
type: Motor
cfgfile: "local/motor1.yml"

mirror1:
type: Mirror
```

(continues on next page)

[20] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/guide.html

(continued from previous page)

```
cfgfile: "local/mirror1.yml"
```

Note: This configuration shall be adapted to the instrument specific needs.

13.5 Device Manager Logs

```
$ nomad alloc logs -job fcs
```

The output of the server shall be something like the following:

```
2021-04-20T08:19:13.783333   INFO CfgFile - req_endpoint = <zpb.rr://134.171.2.
↪250:24738/>
2021-04-20T08:19:13.783420   INFO CfgFile - db_endpoint = <134.171.2.250:27874>
2021-04-20T08:19:13.783437   INFO CfgFile - pub_endpoint = <zpb.ps://134.171.2.
↪250:21765/>
2021-04-20T08:19:13.783519   INFO CfgFile - Devices: 4
2021-04-20T08:19:13.783535   INFO shutter1
2021-04-20T08:19:13.783552   INFO lamp1
2021-04-20T08:19:13.783566   INFO motor1
2021-04-20T08:19:13.783578   INFO mirror1
2021-04-20T08:19:13.799213   INFO Application fcsDevmgr started.
2021-04-20T08:19:13.817823   INFO PS Endpoint: zpb.ps://134.171.2.250:21765/std/
↪status
2021-04-20T08:19:13.934309   INFO [shutter1] Warning device simulated !
2021-04-20T08:19:13.934391   INFO [shutter1] reading configuration keywords ...
2021-04-20T08:19:13.934943   INFO [shutter1] Publishing Endpoint: zpb.ps://134.
↪171.2.250:21765/shutter1
2021-04-20T08:19:14.074730   INFO [lamp1] Warning device simulated !
2021-04-20T08:19:14.074812   INFO [lamp1] reading configuration keywords ...
2021-04-20T08:19:14.075311   INFO [lamp1] Publishing Endpoint: zpb.ps://134.171.
↪2.250:21765/lamp1
2021-04-20T08:19:14.177424   INFO [motor1] Warning device simulated !
2021-04-20T08:19:14.178016   INFO [motor1] reading init sequence ...
2021-04-20T08:19:14.178057   INFO [motor1] number of init actions: 4
2021-04-20T08:19:14.178197   INFO [motor1] reading name position:motor1
2021-04-20T08:19:14.178222   INFO [motor1] number of named positions: 2
2021-04-20T08:19:14.178308   INFO [motor1] reading named position␣
↪tolerance:motor1
2021-04-20T08:19:14.178344   INFO [motor1] named position tolerance: 1
2021-04-20T08:19:14.179338   INFO [motor1] Publishing Endpoint: zpb.ps://134.171.
↪2.250:21765/motor1
2021-04-20T08:19:14.180703   INFO [motor1] Publishing Motor Endpoint: zpb.ps://
↪134.171.2.250:21765/motor1/position
2021-04-20T08:19:14.382390   INFO [mirror1] Warning device simulated !
2021-04-20T08:19:14.382480   INFO [mirror1] reading configuration keywords ...
```

(continues on next page)

(continued from previous page)

```
2021-04-20T08:19:14.382698   INFO [mirror1] Publishing Endpoint: zpb.ps://134.
↪171.2.250:21765/mirror1
```

Note: The server can be started with option -l ERROR to remove the information logs.

13.5.1 Initialising the server

The client application can be used to send commands to the server from the command line:

```
$ fcfClient `geturi fcs-req` GetState ""
```

Note: `geturi` is an utility to compose the expected URI from the name of the registered Consul service.

The reply from the server shall be something like the following:

```
Idle/Operational/On/
```

Besides the above, you can also use the FCF CLI to interact with the server.

```
$ fcfcli --name fcs-req --module fcsclib.fcs_commands
zpb.rr://134.171.2.250:24738
fcsclib.fcs_commands
fcsSh> status
```

Note: In this case, the FCF CLI needs the parameter –module to include the changes to manage the custom device. If you do not have custom devices, you do not need to specify –module.

13.6 Using the *DeviceManager* GUI

13.6.1 Starting the GUI

```
$ fcsGui --uri `geturi fcs-req` &
```

You can control the devices from the GUI, for instance by selecting the action *OPEN* or *CLOSE* from the shutter widget and then pressing *Setup* button.

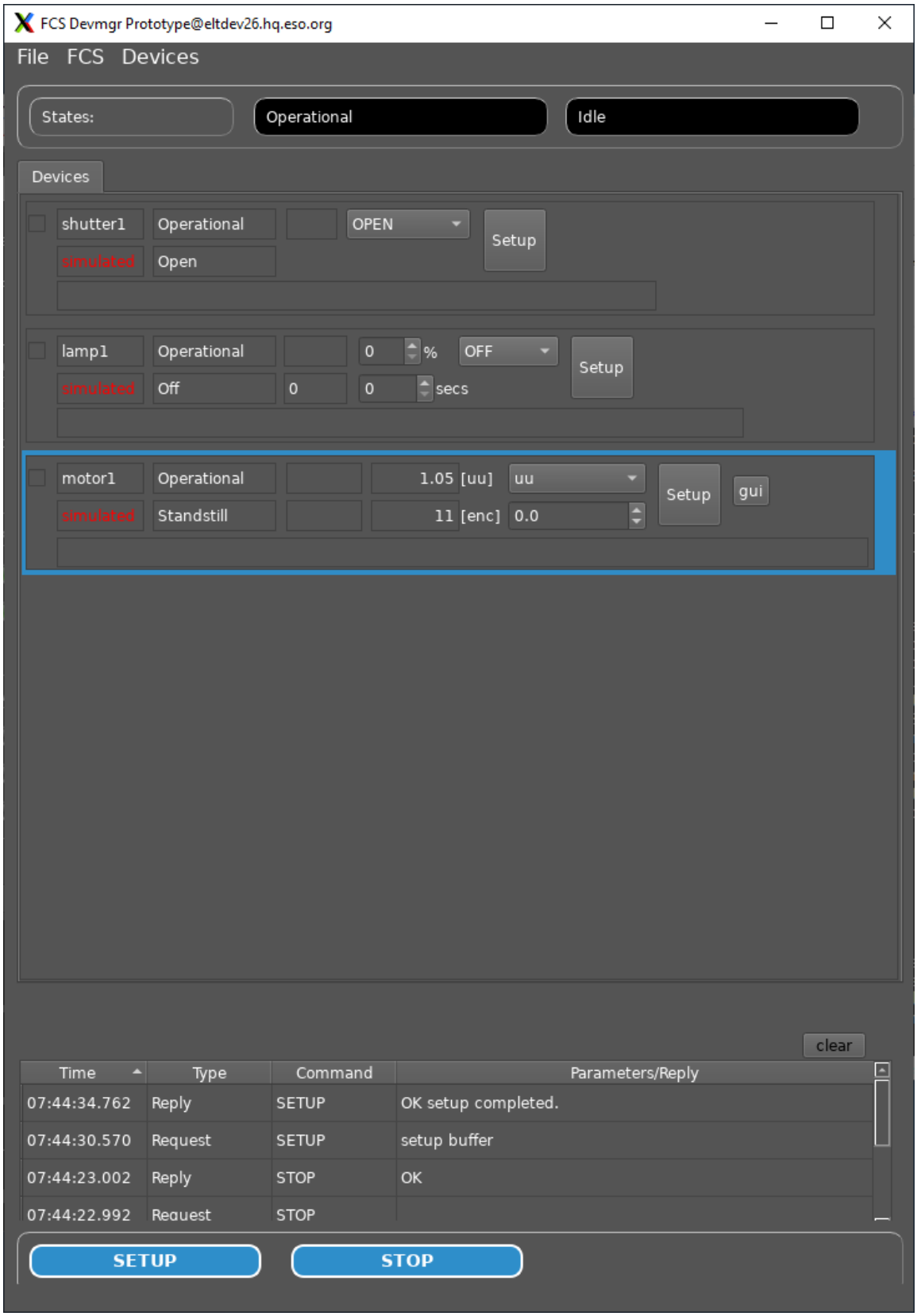


Fig. 13.2: Device Manager Engineering Graphical Interface.

13.7 Using the *Motor Engineering* GUI

13.7.1 Starting the GUI

This GUI can be started from the command line or launched from the *fcfGui* by clicking with the mouse pointer over the small 'gui' button on each motor widget.

```
pymotgui -d motor1 -a opc.tcp://127.0.0.1:7578 -p MAIN.Motor1 &
```

Fig. 13.3: Motor Engineering GUI.

13.7.2 Controlling Custom Device

Warning: The mirror device is a dummy device that is a showcase how to develop custom devices.

The mirror device includes a setup that triggers the execution of a custom RPC in the controller (simulator).

```
$ fcsSh> setup_spf mirror1:action="MOVE",mirror1:tip=2.1,mirror1:tilt=4.3
```

Note: The client library will convert the Simple Parameter Format (SPF) into JSON and check against the defined schema. If the validation is okay, it will send this JSON string to the server which it will pass it to the corresponding device to be processed.

Note: The same you can do from the custom GUI which provides a custom widget for this device.

The reply from the server shall be something like the following:

```
2021-04-20T13:01:55.551216   INFO Started SETUP command
2021-04-20T13:01:55.551443   INFO Processing Setup command ...
2021-04-20T13:01:55.551506   INFO ID: mirror1
2021-04-20T13:01:55.551621   INFO Setup buffer: {"action": "MOVE", "tip": 2.1,
↪"tilt": 4.3}
2021-04-20T13:01:55.552321   INFO Action: MOVE
2021-04-20T13:01:55.552358   INFO Tip: 2.1
2021-04-20T13:01:55.552365   INFO Tilt: 4.3
2021-04-20T13:01:55.552373   INFO [mirror1] Executing RPC_PING ...
2021-04-20T13:01:55.553805   INFO [mirror1] Successful call of Mirror ping:
```

13.7.3 Forcing a Syntax Error

The custom device defines a sample schema where parameters tip and tilt are mandatory. If a setup is sent with a missing required parameter, this will be detected when validating against the schema, see below.

```
$ fcsSh> setup_spf mirror1:action="MOVE", mirror1:tip=3
'tilt' is a required property

Failed validating 'required' in schema['items']['properties']['param'][
↪'properties']['mirror']:
    {'properties': {'action': {'description': 'Mirror action.',
```

(continues on next page)

(continued from previous page)

```
                                'enum': ['MOVE'],
                                'type': 'string'},
                    'piston': {'description': 'piston.', 'type': 'number'},
                    'tilt': {'description': 'tilt.', 'type': 'number'},
                    'tip': {'description': 'tip.', 'type': 'number'}},
     'required': ['action', 'tip', 'tilt'],
     'type': 'object'}

On instance[0]['param']['mirror']:
    {'action': 'MOVE', 'tip': 3}
JSON data not valid !
ERROR: something went wrong, setup buffer not modified
...
```

13.8 Working with a PLC

The following steps can be done to use the server configuration that was generated with controllers running in a PLC.

Note: The provided PLC project includes the corresponding simulators so hardware is not needed only a bare PLC. All the mapping is pre-configured and the PLC code is generated to match the server configuration (only the three standard devices).

13.8.1 Requirements

- PLC available with required version of TwinCAT run-time and OPC-UA Server.
- Windows development environment with required TwinCAT software installed.
- All ESO PLC libraries installed in the Windows development environment.

1. Load the PLC Project

From a Visual Studio load the PLC project generated. The project can be found in fcs1/LCS/plcprj1.

2. Select Target System

Select the target system, this is the PLC that you want to use.

3. Build the PLC Project

If you have errors, make sure you have all ESO libraries installed.

4. Activate Configuration

This active the project configuration and restart the OPC-UA Server in the PLC.

5. Login to the target system.

This will download the code to the PLC.

6. Update the configuration of each device with the corresponding PLC address (IP).

```
lamp1:
  type: Lamp
  interface: Softing
  identifier: PLC1                                  # OPCUA Object Identifier
  prefix: MAIN.Lamp1                                # OPCUA attribute prefix
  simulated: true
  ignored: false
  address: opc.tcp://134.171.59.99:4840             # <<<---- Change the IP here

  simaddr: opc.tcp://{{ range service "lamp1sim" }}{{ .Address }}:{{ .Port }}{{␣
↪end }}

  mapfile: "fcf/devmgr/server/mapLamp.yml"
  fits_prefix: "LAMP1"
  ctrl_config:
    low_fault:         false                         # If T, signal is active low
    low_on:            false                         # If T, signal is active low
    low_switch:        false                         # If T, signal is active low
    initial_state:     false
    timeout:           2000
```

6. Stop the simulation for the devices

```
$ fcfClient `geturi fcs-req` Reset ""
$ fcfClient `geturi fcs-req` StopSim "lamp1, shutter1, motor1"
```

Note: By changing the flag simulated to false in the device configuration it is possible to make this change persistent.

7. Set the server to operational state

```
$ fcfClient `geturi fcs-req` Init ""
$ fcfClient `geturi fcs-req` Enable ""
```

You can now operate the server with devices connected to the PLC controllers.

13.9 Stopping the Software

13.9.1 Stopping the GUI

The menu *File* has an option *Exit*. Please select this option if you want to stop properly the GUIs.

13.9.2 Stopping the Software

You can use the startup/shutdown script provide to stop the whole software. For more details please check the general Getting Started guide here[21]

[21] http://www.eso.org/~eltmgr/ICS/documents/IFW_HL/sphinx_doc/html/manuals/ifw/src/docs/guide.html