SCPI Device Documentation

Release trunk

A. Parenti

Aug 23, 2024

Contents

Contents:

CHAPTER 1

How to write a SCPI device

1.1 Introduction

From wikipedia: 'The Standard Commands for Programmable Instruments (SCPI) (often pronounced "skippy") defines a standard for syntax and commands to use in controlling programmable test and measurement devices.'

The terms 'instrument' and 'device' will be used when when referring to hardware instruments and software (Karabo) devices, respectively.

1.2 Supported interfaces

The base class currently supports TCP/IP, USB and serial communication with the SCPI instrument. The interface can be set-up by means of the `Instrument URL` device property. The URL must contain a scheme and a path. If no scheme is specified, 'socket' is assumed.

In case of TCP/IP communication the scheme to be used is 'socket' or 'tcp' and the path must contain the hostname and port separated by a colon, namely 'socket://hostname:port' or 'tcp://hostname:port'.

In case of serial interface the scheme to be used is 'serial' and the path must contain the file pointer to the serial port (e.g. /dev/ttyUSB0). The other mandatory parameter is the baud rate, to be specified in the query part of the URL. A minimal serial URL would be for example 'serial:///dev/ttyUSB0?baudRate=57600'. Optional parameters like 'dataBits', 'stopBits' and 'parity' can be specified in the query, e.g. 'serial:///dev/ttyUSB0?baudRate=57600&dataBits=7&stopBits=1&parity=O'.

In case of a USB interface the scheme shall be 'usb' and the path shall contain the vendor and product ID separated by a colon, namely 'usb://idVendor:idProduct'. In case more than one USB device of the same kind is attached to the computer, you must also give its serial number or the bus and address in the query part of the URL to unambiguously identify one of them. For example 'usb://2457:1012?serialNumber=SN12345' or 'usb://2457:1012?bus=003&address=002'. In order to find out the correct values of the parameters, you can run in a terminal the `lsusb -v` command .

1.3 How to Create a SCPI Karabo Device

Create a python Karabo Device, called for example myScpiDevice:

```
./karabo new myScpiDevice python
```

Go into the base directory of the project:

```
cd karabo/devices/myScpiDevice
```

Add a plaintext file named DEPENDS that contains the scpi version tag, like this:

```
#package, Branch
scpi, 2.1.0-2.16.5
```

Do not forget to add the DEPENDS file to the repository:

```
git add DEPENDS
```

Now you can open the source file, MyScpiDevice.py, for editing.

1.4 Base Device Classes

There are currently three base classes available for SCPI devices: `ScpiDevice`, `ScpiOnOff`, `ScpiStartStop`.

`ScpiDevice` is the simplest one. It has only three states: `UNKNOWN` (i.e. the software device is not connected to the hardware), `NORMAL` (i.e. the hardware is functional) and `ERROR` (i.e. the hardware is returning some error).

`ScpiOnOff` has five states: `UNKNOWN`, `ERROR`, `ON`, `OFF` and `CHANGING` (e.g. for when the hardware is switching between `ON` and `OFF`). This class is suitable for example for power supplies.

`ScpiStartStop` has also five states: `UNKNOWN`, `ERROR`, `STARTED`, `STOPPED` and `CHANGING` (e.g. for when the hardware is switching between `STARTED` and `STOPPED`). This class is suitable for example for measurement devices.

1.5 Writing a Device

1.5.1 Writing a Simple SCPI Device

The base class to be used is `ScpiDevice`. The device is very simple and should look like:

```
#############################################################################
# Author: <john.smith@xfel.eu>
# Created on March 29, 2023
# Copyright (C) European XFEL GmbH Schenefeld. All rights reserved.
#############################################################################

from karabo.bound import KARABO_CLASSINFO

from scpi.scpi_device import ScpiDevice
from ._version import version as deviceVersion


@KARABO_CLASSINFO("MyScpiDevice", deviceVersion)
```

(continues on next page)

(continued from previous page)

```
class MyScpiDevice(ScpiDevice):

    def __init__(self, configuration):
        # always call superclass constructor first!
        super().__init__(configuration)

        self.commandTerminator = "\r\n" # The command terminator

    @staticmethod
    def expectedParameters(expected):
        (
        # Fill here with the list of expected parameters
        )
```

The SCPI parameters can be accessed using Karabo expected parameters (see the *Expected Parameters* Section for details).

1.5.2 Writing an On/Off Device

The base class to be used is `ScpiOnOff`. The device should look pretty much like the Ok/Error one, except for the base class to be used.

You will also need to define an `alias` for the `on` and `off` slots, see the *Expected Parameters* Section for details).

```
#############################################################################
# Author: <john.smith@xfel.eu>
# Created on March 29, 2023
# Copyright (C) European XFEL GmbH Schenefeld. All rights reserved.
#############################################################################

from karabo.bound import (
    KARABO_CLASSINFO, OVERWRITE_ELEMENT
)

from scpi.scpi_on_off import ScpiOnOff
from ._version import version as deviceVersion


@KARABO_CLASSINFO("MyScpiDevice", deviceVersion)
class MyScpiDevice(ScpiOnOff):

    def __init__(self, configuration):
        # always call superclass constructor first!
        super().__init__(configuration)

        self.commandTerminator = "\r\n" # The command terminator

    @staticmethod
    def expectedParameters(expected):
        (
        # Define alias for the "on" slot
        OVERWRITE_ELEMENT(expected).key("on")
        .setNewAlias("OUTP ON;;;;")
        .commit(),
```

(continues on next page)

(continued from previous page)

```
        # Define alias for the "off" slot
        OVERWRITE_ELEMENT(expected).key("off")
        .setNewAlias("OUTP OFF;;;;")
        .commit(),

        # Fill here with the list of expected parameters
        )
```

By default the `on` and `off` slots will send the corresponding SCPI command and immediately change device state (to respectively `ON` and `OFF`). In case the switch takes some time, and the instrument has some property which can be used to evaluate its actual state, this default behaviour shall be overridden, e.g.

```
@staticmethod
def expectedParameters(expected):
    (
    [...]

    # This parameter is tagged as 'poll', thus the query 'OUTP?' will
    # be sent periodically and the Karabo device updated accordingly.
    BOOL_ELEMENT(expected).key("isOn")
    .tags("scpi,poll")
    .alias(";;OUTP?;{isOn:b};")
    .displayedName("Device is ON")
    .description("True when the device is ON.")
    .readOnly()
    .commit(),
    )

# Overrides ScpiOnOff.on()
def on(self):
    self["status"] = "Being switched on"
    self.updateState(State.CHANGING)
    self.sendCommand("on")
    # followHardwareState() will take care of updating the state when
    # the hardware is done.

# Overrides ScpiOnOff.off()
def off(self):
    self["status"] = "Being switched off"
    self.updateState(State.CHANGING)
    self.sendCommand("off")
    # followHardwareState() will take care of updating the state when
    # the hardware is done.

# This function will be called at the end periodic device poll
def followHardwareState(self):
    is_on = self["isOn"]
    state = self["state"]

    if state in (State.CHANGING, State.OFF) and is_on:
        self.log.INFO('Follow hardware state -> ON')
        self.updateState(State.ON)
    elif state in (State.CHANGING, State.ON) and not is_on:
        self.log.INFO('Follow hardware state -> OFF')
        self.updateState(State.OFF)
```

1.5.3 Writing a Start/Stop Device

The base class to be used is `ScpiStartStop`. Similarly to the On-Off one, you will need to define here aliases for the `start` and `stop` slots. You will also need to provide a `processAsyncData` function to process the data that are received asynchronously during a measurement.

The device should look like

```
#############################################################################
# Author: <john.smith@xfel.eu>
# Created on March 29, 2023
# Copyright (C) European XFEL GmbH Schenefeld. All rights reserved.
#############################################################################

from karabo.bound import (
    KARABO_CLASSINFO, OVERWRITE_ELEMENT
)

from scpi.scpi_start_stop import ScpiStartStop
from ._version import version as deviceVersion


@KARABO_CLASSINFO("MyScpiDevice", deviceVersion)
class MyScpiDevice(ScpiStartStop):

    def __init__(self, configuration):
        # always call superclass constructor first!
        super().__init__(configuration)

        self.commandTerminator = "\r\n" # The command terminator

    @staticmethod
    def expectedParameters(expected):
        (
        # Define alias for the "start" slot
        OVERWRITE_ELEMENT(expected).key("start")
        .setNewAlias("INIT;;;;")
        .commit(),

        # Define alias for the "stop" slot
        OVERWRITE_ELEMENT(expected).key("stop")
        .setNewAlias("ABORT;;;;")
        .commit(),

        # Fill here with the list of expected parameters
        )
    # Use this one to process asynchronously received data and possibly
    # update device properties.
    def processAsyncData(self, data):
        pass
```

Also in the case of `ScpiStartStop` the `start` and `stop` will send the respective SCPI command and change immediately the device state. The default behaviour can be changed as described in the previous section.

1.5.4 How to read/write parameters from/to the instrument

Each parameter on the instrument you want to expose in the Karabo device must have a corresponding expected parameter in the Karabo device. The expected parameter must be tagged as `scpi`. Please have a look at the *Expected Parameters* Section for the details.

1.5.5 How to deal with asynchronous instrument messages

To parse unexpected messages (e.g. error message) that may come while instrument is polled [Note that this violates SCPI specifications] the method `parseAsyncStrings()` can be implemented in the derived class.

1.6 Expected Parameters

1.6.1 Tags

- `scpi` tag: Parameters to be read from (written to) the SCPI instrument must have the `scpi` tag.
- `readOnConnect` and `writeOnConnect` tags: Parameters having the `readOnConnect` (respectively `writeOnConnect`) flag will be read from (written to) the instrument when the Karabo device connects to it.
- `poll` tag: Parameters having the `poll` tag will be polled periodically. The poll interval is a parameter of the base class.

1.6.2 The `sendOnConnect` Parameter

Commands to be sent to the instrument when the Karabo device connects to it (for example some initial configuration), can be listed in the `__init__` function. For example:

```
self.sendOnConnect = ['TRIG:LEV 10', 'TRIG:SOURCE EXT', 'SYST:COMM:SER:BAUD 19200']
```

These commands will be sent before the expected parameters with `writeOnConnect` tag (see *Tags* Section).

1.6.3 The `sendOnReset` Parameter

Commands to be sent to the instrument when executing the `reset` command, can be listed in the `__init__` function. For example:

```
self.sendOnReset = ['Reset Uc']
```

1.6.4 Aliases

The SCPI commands and queries corresponding to writing and reading any parameter must be written in the parameter `alias`. Different fields in the `alias` have to be separated by semicolons (;) or a different separator (as explained in *The aliasSeparator Parameter* Section). For example:

```
INT32_ELEMENT(expected).key("resolutionMode")
        .tags("scpi poll")
        .alias(">S1H {resolutionMode};E0;>S1H?;S1H:{resolutionMode:d};")
        .displayedName("Current Resolution Mode")
        .description("Set the current resolution mode (0=normal 1=high resolution).")
        .assignmentOptional().defaultValue(0)
        .options("0 1")
        .allowedStates(State.ON, State.OFF)
        .reconfigurable()
        .commit(),
```

The first field in the `alias` contains the set command (ie `>S1H`) and its parameters (ie `{resolutionMode}`) for the `resolutionMode`. This string will be parsed, and `{resolutionMode}` will be replaced by the configuration value corresponding to the key. The second field (ie `E0`) is the expected reply to the set command; it is also parsed to extract parameters (none in this example).

The third field contains the query command (ie `>S1H?`) and its parameters (none). The fourth field (ie `{resolutionMode:d}`) is the expected reply to the query; it is parsed and `resolutionMode` is extracted as integer (`d`). The parsing is done by using the python parse package (see documentation), therefore all types defined there can be used:

Type	Characters Matched	Output
w	Letters and underscore	str
W	Non-letter and underscore	str
s	Whitespace	str
S	Non-whitespace	str
d	Digits (effectively integer numbers)	int
D	Non-digit	str
n	Numbers with thousands separators (, or .)	int
%	Percentage (converted to value/100.0)	float
f	Fixed-point numbers	float
e	Floating-point numbers with exponent e.g. 1.1e-10, NAN (all case insensitive)	float
g	General number format (either d, f or e)	float
b	Binary numbers	int
o	Octal numbers	int
x	Hexadecimal numbers (lower and upper case)	int
ti	ISO 8601 format date/time e.g. 1972-01-20T10:21:36Z (“T” and “Z” optional)	datetime
te	RFC2822 e-mail format date/time e.g. Mon, 20 Jan 1972 10:21:36 +1000	datetime
tg	Global (day/month) format date/time e.g. 20/1/1972 10:21:36 AM +1:00	datetime
ta	US (month/day) format date/time e.g. 1/20/1972 10:21:36 PM +10:30	datetime
tc	ctime() format date/time e.g. Sun Sep 16 01:03:52 1973	datetime
th	HTTP log format date/time e.g. 21/Nov/2011:00:07:11 +0000	datetime
ts	Linux system log format date/time e.g. Nov 9 03:37:44	datetime
tt	Time e.g. 10:21:36 PM -5:30	time

In addition, two extra types can be used for the SCPI devices:

Type	Characters Matched	Output
p	All printable characters	str
P	All non-printable characters	str

1.6.5 The `aliasSeparator` Parameter

The separator for the fields in the `alias` is by default the semicolon (;), but can be changed to a different one in the `__init__` function; for example:

```
self.aliasSeparator = "|"
```

will change it to the pipe character (|).

1.6.6 The `initialConnectedState` Parameter

It defines which will be the device state after successful connection to the remote instrument. It is set to `State.NORMAL` for a `ScpiDevice`, to `State.OFF` for a `ScpiOnOffDevice` and to `State.STOPPED` for a `ScpiStartStopDevice`.

1.6.7 The `commandPrefix` and `terminator` Parameters

The command prefix and terminator - to be used in the communications between the Karabo device and the SCPI instrument - can be set in two different ways. For a given device, the command prefix and terminator are usually known and fixed, therefore should be hard-coded in the Karabo device. This can be done by adding lines like these to the `__init__` function:

```
self.commandTerminator = "\r\n" # The command terminator

self.commandPrefix = "*" # all command strings starts with
                         # asterisk ('*') char
```

The second way to set command prefix and terminator is by adding the `commandPrefix` and/or `terminator` expected parameters. This should be done for "generic" devices, for which different prefixes and terminators should be available at instantiation time. For example:

```
# Re-define default value and options
STRING_ELEMENT(expected).key("terminator")
        .displayedName("Command Terminator")
        .description("The command terminator.")
        .assignmentOptional().defaultValue("\\n")
        .options("\\n \\r \\r\\n")
        .init()
        .commit(),

# Re-define default value and options
STRING_ELEMENT(expected).key("commandPrefix")
        .displayedName("Command Prefix")
        .description("The command prefix.")
        .assignmentOptional().defaultValue("")
        .options('* > ""')
        .init()
        .commit(),
```

If the terminator is not set in the Karabo device, the default one will be used for communications with the SCPI instrument: `\n`. If the command prefix is not set in the Karabo device, the default one will be used for communications with the SCPI instrument: `''` (empty string i.e. no prefix).

1.6.8 The `responsePrefix` and `responseTerminator` Parameters

In addition to the command prefix and terminator described in the previous section, prefix and terminator for responses can be optionally set. If response prefix/terminator value is not set, the command prefix/terminator will be used for responses as well.

Also for the response prefix and terminator there are two ways of setting. You can use either:

```
self.responseTerminator = "\0"

self.responsePrefix = ">"
```

in the `__init__` function, or add the `responsePrefix` and/or `responseTerminator` expected parameter. This second option should be used for "generic" devices, as explained in the previous section.

1.6.9 The `scpiTimeout` Parameter

The default scpi communication timeout used in the base class is 1 second. This value is normally ok, but some instruments (eg the agilentMultimeterPy) may need a longer time to give back a measurement.

The scpi timeout (in seconds) can be changed in `__init__` with something like:

```
self.scpiTimeout = 5.0 # New timeout value in seconds
```

A second way to set it is by adding the `scpiTimeout` expected parameter. In this way the timeout can be changed during the lifetime of the Karabo device. For example:

```
FLOAT_ELEMENT(expected).key("scpiTimeout")
        .displayedName("SCPI Timeout")
        .description("The scpi communication timeout.")
        .unit(Unit.SECOND)
        .assignmentOptional().defaultValue(1.0)
        .reconfigurable()
        .commit(),
```

If the scpi timeout is not set in the Karabo device, the default value of 1 s will be used.

1.6.10 The `scpiWaitAfterRequest` Parameter

By default there is no wait time between two consecutive queries or commands, in the base class. This means: as soon as the reply comes from the instrument, the next query or command is sent. There is the possibility to modify the wait time in `__init__` by doing

```
self.scpiWaitAfterRequest = 0.01 # New wait time in seconds
```

or by adding a `scpiWaitAfterRequest` element to the expected parameters.

1.6.11 The `socketTimeout` Parameter

The default TCP socket timeout used in the base class is 1 second. Similarly to the scpi communication timeout, also the TCP socket timeout for the device can be set to a differene value, either in the `__init__` by doing

```
self.socketTimeout = 2.0 # New timeout value in seconds
```

or by adding a `socketTimeout` element to the expected parameters.

1.6.12 The `usbTimeout` Parameter

The default USB timeout used in the base class is 1 second. Similarly to the other timeouts, it can be changed, either in the `__init__` by doing

```
self.usbTimeout = 2.0 # New timeout value in seconds
```

or by adding a `usbTimeout` element to the expected parameters.

1.6.13 On/Off (and Start/Stop) Slots

For On/Off (Start/Stop) devices, the on/off (start/stop) slots are already defined. What you have to do, is to set the SCPI command in the slots's `alias`. For example, for the start/stop:

```
# Define alias for the "start" slot
OVERWRITE_ELEMENT(expected).key("start")
        .setNewAlias("INIT;;;;")
        .commit(),

# Define alias for the "stop" slot
OVERWRITE_ELEMENT(expected).key("stop")
        .setNewAlias("ABORT;;;;")
        .commit(),
```

1.6.14 A Complete Example

Here is a complete example of expected parameters for a Start/Stop device:

```
# Define alias for the "start" slot
OVERWRITE_ELEMENT(expected).key("start")
        .setNewAlias("INIT;;;;") # No query available
        .commit(),

# Define alias for the "stop" slot
OVERWRITE_ELEMENT(expected).key("stop")
        .setNewAlias("ABORT;;;;") # No query available
        .commit(),

# Re-define default value and options
STRING_ELEMENT(expected).key("terminator")
          .displayedName("Command Terminator")
          .description("The command terminator.")
         .assignmentOptional().defaultValue("\\n")
          .options("\\n")
          .init()
          .commit(),

STRING_ELEMENT(expected).key("handshake")
          .tags("scpi")
          .alias("SYST:COMM:HAND {handshake};;SYST:COMM:HAND?;{handshake:w};")
          .displayedName("Handshake")
          .description("Set the state of the message roundtrip handshaking.")
          .assignmentOptional().defaultValue("OFF")
          .options("OFF ON")
          .allowedStates(State.STOPPED)
```

(continues on next page)

(continued from previous page)

```
            .reconfigurable()
            .commit(),

STRING_ELEMENT(expected).key("baudRate")
            .tags("scpi")
            .alias("SYST:COMM:SER:BAUD {baudRate};;SYST:COMM:SER:BAUD?;{baudRate:w};")
            .displayedName("Serial Baud Rate")
            .description("Set the transmit and receive baud rates on the RS-232 port.")
            .assignmentOptional().defaultValue("9600")
            .options("DEFAULT 9600 19200 38400 57600 115200")
            .allowedStates(State.STOPPED)
            .reconfigurable()
            .commit(),

INT32_ELEMENT(expected).key("errorCount")
            .tags("scpi poll")
            .alias(";;SYST:ERR:COUNT?;{errorCount:d};") # Only query available
            .displayedName("Error Count")
            .description("The number of error records in the queue.")
            .readOnly()
            .commit(),

STRING_ELEMENT(expected).key("measureType")
            .tags("scpi writeOnConnect") # Write to h/w at initialization
            .alias("CONF:MEAS:TYPE {measureType};;CONF:MEAS:TYPE?;{measureType:w};")
            .displayedName("Measure Type")
            .description("Set the meter measurement mode (energy or power).")
            .assignmentOptional().defaultValue("J")
            .options("DEFAULT J W")
            .allowedStates(State.STOPPED)
            .reconfigurable()
            .commit(),

STRING_ELEMENT(expected).key("serialNumber")
            .tags("scpi readOnConnect") # Read from h/w at initialization
            .alias(";;SYST:INF:SNUM?;\"{serialNumber:p}\";") # Only query available
            .displayedName("Serial Number")
            .description("The serial number.")
            .readOnly()
            .commit(),
```

1.7 Polling Device Properties

All the expected parameters having the `poll` tag will be automatically polled (see *Tags* Section). The refresh interval is given by the `pollInterval` device parameter.

An immediate refresh can be triggered by the `pollNow` command.

The list of parameters to be polled can be reconfigured be means of the `propertiesToPoll` property. For example, if you set it to `handshake,baudRate` these two properties will be polled. The access level for `propertiesToPoll` property is expert.

A user's hook is also provided by the base class, allowing the post-processing of the polled properties. For example, if you read some temperature in Fahrenheit degrees and you want to display it in Celsius, you can define two expected parameters, like in the following:

```
FLOAT_ELEMENT(expected).key("temperature")
          .displayedName("Temperature")
          .description("Blah blah.")
          .unit(Unit.DEGREE_CELSIUS)
          .readOnly()
          .commit(),

FLOAT_ELEMENT(expected).key("temperatureFahrenheit")
          .tags("scpi poll")
          .alias(";;GETTEMP;{temperatureHex:g};")
          .displayedName("Temperature Fahrenheit")
          .description("Blah blah.")
          .expertAccess() # Only visible to expert
          .readOnly()
          .commit(),
```

Then, you can postprocess the polled data this way:

```
def pollInstrumentSpecific(self):

    # 'temperatureFahrenheit' is a polled property
    tF = self.get('temperatureFahrenheit')

    # Convert temperatureFahrenheit into Celsius degrees
    tC = (tF - 32.) / 1.8

    # Set 'temperature' on the Karabo device, which is a derived property
    self.set('temperature', tC)
```

1.7.1 The `gotoErrorOnPollFailure` Parameter

If the poll of a parameter fails, an error message will be logged, but no further action will be by default taken. By setting the `gotoErrorOnPollFailure` parameter to `True`, if **all** parameter polls fail, the device will go to error state.

1.8 Reading and Writing Properties on Connection

As mentioned above (see *Tags* Section) parameters to be sent to hardware upon connection (e.g. stored configuration values) are marked with `writeOnConnect` tag, while parameters to be retrieved from the hardware at connection are marked with `readOnConnect` tag.

The base class provides the `readOnConnectInstrumentSpecific` hook that can be optionally implemented in the derived device to perform any parsing or post processing of these parameters, similarly to `pollInstrumentSpecific`.

1.9 Preprocessing the Incoming Reconfiguration

The base class provides a user's hook to preprocess the incoming reconfiguration, before any command is sent to the instrument. It can be used for example when the instrument expects a parameter in some unusual units, and you would like to allow the user to input the parameter with a more standard unit.

This can be done by using two expected parameters, like in the following:

```
FLOAT_ELEMENT(expected).key("temperature")
          .tags("scpi")
          .alias("SETTEMP {temperatureHex};;;;")
          .displayedName("Temperature")
          .description("Blah blah.")
          .unit(Unit.DEGREE_CELSIUS)
          .reconfigurable()
          .commit(),

STRING_ELEMENT(expected).key("temperatureHex")
          .displayedName("TemperatureHex")
          .description("Blah blah.")
          .expertAccess() # Only visible to expert
          .readOnly()
          .commit(),
```

and then by coding the relation between temperature and temperatureHex in the preprocessConfiguration function,

```
def preprocessConfiguration(self, inputConfig):

    if inputConfig.has('temperature'):
        # Get temperature from inputConfig, change unit,
        # represent it as bytes
        temp = inputConfig.get('temperature') # eg -23.15
        temp100 = np.int16(100*temp) # -2315
        tempBytes = temp100.tostring() # b'\xf5\xf6'
        tempHex = '%02X%02X'%(tempBytes[0], tempBytes[1]) # 'F5F6'

        self.set('temperatureHex', tempHex)
```

Setting a new value for `temperature` will make preprocessConfiguration called. A new value for `temperatureHex` will be set in the device, and only then the command `SETTEMP` will be sent to the instrument, with `temperatureHex` as an additional parameter.

1.10 Postprocessing the Incoming Reconfiguration

The base class provides an additional user's hook called when all actions defined in an incoming reconfiguration have been performed. This can be used to immediately trigger property polling after a property, e.g. 'xyz', has been reconfigured which is known to cause side-effects on other properties. Manipulating the polling event loop in this way reduces the period when misalignment of property values are exposed to clients.

```
def postprocessConfiguration(self, inputConfig):

    if inputConfig.has('xyz'):
        self.pollTrigger = True
```

1.11 Enabling the Heartbeat

The scpi Karabo device can periodically send a heartbeat query to the instrument. The default query is `*IDN?`, which should be available for all SCPI-compliant instruments. The sending of the heartbeat query is by default disabled, but it can be enabled by setting the `enableHeartbeat` property to `True`.

Once a reply to the query is received, it is published in the `heartbeatReply` property, and the `heartbeatTime` is updated as well, with the current time.

The query can be changed in the `heartbeatCommand` property, and also the time interval can be changed in the `heartbeatPeriod` property (default value is 5 s).

1.12 Enabling Auto-Connect

Setting the `autoConnect` property `True` will enable auto connecting to the instrument when the state is `UNKNOWN`. The default property value is `False`.

1.13 Injecting into the Schema

The `postOnConnect` hook is provided and can be used, amongst other actions, to inject properties into the schema.

For schema injection the user must control in the derived class whether multiple injection, after each connect, is required or must be prevented. The base class always calls the hook.

1.14 Name Swapping on Write and Read

The `addValueSwap` function is used to associate a class with the key of a property. Calling addValueSwap(self, key, className) associates className with key.

The list of user friendly device swvalue value and instrument side hwvalue value associations are defined in module imported into the device. The ValueSwap class is required to be iterable and have attributes swvalue and hwvalue, and can be provided most easily by using an enum as shown below.

```
@unique
class GasTypes(ValueSwapEnum):
    N2orAir = ('N2orAir', '0')
    Ar = ('Ar', '1')
    Unknown = ('Unknown', '2')
```

, or are created inline

```
aa = ['N2orAir', 'Ar', 'Unknown']
bb = ['0', '1', '2']
GasTypes = ValueSwapEnum('GasTypes', [(a, (a, b)) for a, b, c in zip(aa, bb)])
```

In either case the value swap must be registered.

```
self.addValueSwap('xgsSensor{}.gas'.format(name), GasTypes)
```

Both inherit from ValueSwapEnum

```
from enum import Enum, unique
@unique
class ValueSwapEnum(Enum):
    def __init__(self, swvalue, hwvalue):
        self.swvalue = swvalue
        self.hwvalue = hwvalue
```

The enum can also simplify creating expected properties.

```
gas_desc = ', '.join(['{} = {}'.format(x.swvalue, x.hwvalue) for x in GasTypes])
gas_opts = ', '.join(['{}'.format(x.swvalue) for x in GasTypes])
gas_def = GasTypes.Unknown.swvalue

(
    STRING_ELEMENT(expected).key('xgsSensor{}.gas'.format(name))
        .tags('scpi poll')
        .alias('4F{}{{xgsSensor{}.gas}};;4E{};{{xgsSensor{}.gas}};'.format(gauge,␣
↪name, gauge, name))
        .displayedName('Gas type')
        .description('Gas type = hw id: ({})'.format(gas_desc))
        .allowedStates(State.NORMAL)
        .options(gas_opts)
        .assignmentOptional().defaultValue(gas_def)
        .reconfigurable()
        .commit(),
)
```

1.15 Sending an Arbitrary Command to the Instrument

An arbitrary command (or query) can be sent to the instrument. To do so, it is enough to write the command (or query) in the `sendCommand` device property. This property is expert access level.

Once a reply is received, it is published in the `replyToCommand` property. If no reply is received after the scpi timeout (see *The scpiTimeout Parameter*), `replyToCommand` is left empty.

1.16 Query and Command Statistics

The query and command statistics are accessible at expert level,

For both queries and commands, the number of successes and failures is available, as well as the date/time of the last one. The counters will be reset on reconnection to the device.

1.17 Initial Check of the Connection to Equipment

Some SCPI hardware equipments are connected via an intermediate interface. An example is the Lantronix, which is used to allow a remote access (via its TCP/IP address) to an RS-232-based EnergyMax instrument. If the TCP/IP and port of the Lantronix device are correctly assigned, the SCPI karabo device will successfully establish a TCP/IP connection to that port, but no error will be reported if no equipment is there connected.

In this hardware configuration, the developer should set the parameter `self.enableConnectionCheck` to `True` in the `__init__` function. This will verify an established communication by requesting the heartbeat; the proper value `Heartbeat text` (`heartbeatCommand` key) should be set according to the used equipment.

1.18 Useful Tips for Write (Set) and Read (Get) Patterns to Use

In this section patterns for write (set) and read (get) operations are described. The tips derive from coding scpi-like instruments which are not fully compatible with the scpi-1999 standard.

1.18.1 Use Names Instead of Numbers

Using 'ON' and 'OFF' at the device interface and converting these to '1' and '0' when written to the instrument and reversing the procedure on reading produces an easily understandable interface for the user. This can be achieved by using name swapping as described in `Name swapping on write and read` which additionally simplifies filling in property definitions.

1.18.2 Consider Minimizing the Number Device Buttons

Most SCPI instruments are relatively simple and require only a few command actions (e.g. 'on/off', 'factory reset', 'calibrate device', etc.) and implementing these commands as device buttons is clean and follows the Karabo principle that commands are buttons. However, when coding multi-channel devices, it may be more appropriate to perform actions through read/write properties. This technique is used by MPOD, where O(100) channels can be present.

1.18.3 Minimize the Number of Properties Defined

Two methods to set an instrument property value are typically encountered, 1) where a set command followed by the value to set are written to the instrument, and 2) where different commands are written without a qualifying value, meaning each command represents a particular value to be set. A single command is defined to read the value set.

In the following example the type 2 set commands defined are 10 = Torr, 11 = mBar and 12 = Pascal. The read command is 13.

By moving the second character of the command into a name swap definition the number of properties defined is minimized.

```
@unique
class PressureUnits(ValueSwapEnum):
    Torr = ('Torr', '0')
    Mbar = ('mBar', '1')
    Pascal = ('Pascal', '2')

punit_desc = ', '.join(
    ['{} = {}'.format(x.swvalue, x.hwvalue) for x in PressureUnits])
punit_opts = ', '.join(
```

(continues on next page)

(continued from previous page)

```
    ['{}'.format(x.swvalue) for x in PressureUnits])
punit_def = PressureUnits.Mbar.value[0]

STRING_ELEMENT(expected).key("pressureUnits")
    .tags("scpi writeOnConnect readOnConnect poll")
    .alias("1{pressureUnits};;13;0{pressureUnits};")
    .displayedName('Pressure units')
    .description('Choices are {}. Note 1: the unit is set to {}'
                 ' when the connection is established.'.format(
                  punit_names, punit_def))
    .allowedStates(State.NORMAL)
    .options(punit_opts)
    .assignmentOptional().defaultValue(punit_def)
    .reconfigurable()
    .commit(),

self.addValueSwap('pressureUnits', PressureUnits)
```

1.18.4 Use Option Sets Whenever Possible

Name swapping is not a requirement of using option sets. If a property value written and the read value is an item of the same set (e.g. ['a', 'b', 'c']), i.e. no name swap is needed, then use the options attribute when defining the property. This has the advantage of exposing a pull down menu of the set at the gui and turning on property value validation on set and read.

1.18.5 String Comments

Command and queries exchanged between instruments and devices are character strings. It is good practise is to use the correct type (int, float...) in (Karabo) devices for a property as validation of the property type is performed automatically.

Strings are only loosely validated and devices may have to enforce requirements on string properties required by the instrument in Karabo's preConfiguration() callback. Strings which fail requirements must not be set and an ERROR log message must be injected. Some requirements are softer, e.g. a supplied string is too long but can be set if truncated, in this case a WARN log message must be injected. It is good practice is to document format requirements in the properties description.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search