HowToCpp

Release 1.0

CAS

Mar 18, 2024

CONTENTS

Contents:

CHAPTER

ONE

THE C++ API

1.1 Usage Scenarios

The C++ API is to be used when developing devices interfacing directly with hardware or for processing tasks which benefit from using the Karabo point-to-point communication interface. In terms of APIs it is Karabo's reference implementation.

1.1.1 Benefits

The C++ API allows for statically typed, compile-time checked coding. Interfacing with hardware which has fixed property types frequently benefits from this, as consistency checks of date types are enforced by the compiler.

Additionally, depending on the task at hand a compiled device may also offer significant performance benefits with respect to runtime-interpreted Python device.

Finally, some external library may only be available with C/C++ interfaces. Rather than binding these from Python, developers should directly interface from the C++ API.

1.1.2 Limitations

The development cycles for compiled code tend to be longer, especially when heavy use is made of template mechanisms, as is the case for Karabo. Additionally, more lines of code are frequently required in C++ than are necessary for achieving the same functionality in Python.

1.2 Programming Policies

Devices written in C++ using the Karabo C++ API should follow common coding standards of the language. Specifically, the programmer should assure that

- objects are cleanly destroyed at the end of their lifetime
- allocated memory is freed
- data is encapsulated in objects and accessed only via public members
- namespaces are maintained and used.

Additionally, in Karabo programmers are asked to use the following policies:

CamelCasing

: is generally used when coding

Classes

have capitalized names

Class member variables

should be prefixed with `m_`, i.e. `m_aClassMember`

Additionally, Karabo makes heavy use of facilities provided by the Boost library. Developers are advised to make use of Boost functionality when possible.

1.3 C++11

C++11 usage is now (officially) supported for framework code. The following guidelines are suggested:

- Feel free to use new features where they make sense. E.g. use auto to shorten iterator syntax in loops, e.g. *std::map<MyComplexType, MyMoreComplexType<double> >::const_iterator it = foo.begin() -> auto it = foo.begin()*.
- Don't use *auto* to indicate straight forward types, e.g. *auto i = 4;*
- Existing code does not need to be refactored for C++11 feature usage alone. E.g. if you happen to refactor something anyway, feel free to replace iterators with *auto* if it aids readability. You do not specifically have to refactor otherwise working code though.
- Do **not** use *std::shared_ptr*. We will continue to use *boost::shared_ptr*!
- In general, if a *boost* and a *std*-library feature coexist (smart pointers, mutexes, bind, etc.), continue to use the boost implementation as we have done previously, especially if there is a risk that your new code needs to interact with existing code.
- When using more "advanced" features, like late return type declaration (*->decltype(foo)*), variadic templates or reference forwarding, add a short comment to these lines to aid people less experienced with C++11 features in the review.
- We currently do not encourage using newly introduced numerical types, e.g. *uint64_t* as the Karabo type system has not been fully prepared for them.

1.4 Implementing Devices

1.4.1 The "Conveyor" device

Let us start by having a look at a toy device that simulates a conveyor belt…

```
#ifndef KARABO_CONVEYOR_HH
#define KARABO_CONVEYOR_HH

#include <karabo/karabo.hpp>

/**
 * The main Karabo namespace
 */
namespace karabo {

    class Conveyor : public karabo::core::Device<> {
```

(continues on next page)

(continued from previous page)

```
public:

// Add reflection information and Karabo framework compatibility to this class
KARABO_CLASSINFO(Conveyor, "Conveyor", "2.0")

/**
 * Necessary method as part of the factory/configuration system
 * @param expected Will contain a description of expected parameters
 * for this device
 */
static void expectedParameters(karabo::util::Schema& expected);

/**
 * Constructor providing the initial configuration in form of a Hash object.
 * If this class is constructed using the configuration system the Hash object
 * will already be validated using the information of the expectedParameters
 * function. The configuration is provided in a key/value fashion.
 */
Conveyor(const karabo::util::Hash& config);

/**
 * The destructor will be called in case the device gets killed
 * (i.e. the event-loop returns)
 */
virtual ~Conveyor() {
    KARABO_LOG_INFO << "dead.";
}

/**
 * This function acts as a hook and is called after an reconfiguration
 * request was received, but BEFORE this reconfiguration request is actually
 * merged into this device's state.
 *
 * The reconfiguration information is contained in the Hash object provided
 * as an argument.
 * You have a chance to change the content of this Hash before it is merged
 * into the device's current state.
 *
 * NOTE: (a) The incomingReconfiguration was validated before
 *       (b) If you do not need to handle the reconfigured data, there is
 *           no need to implement this function.
 *           The reconfiguration will automatically be applied to the
 *           current state.
 * @param incomingReconfiguration The reconfiguration information as was
 *         triggered externally
 */
virtual void preReconfigure(karabo::util::Hash& incomingReconfiguration);


/**
 * This function acts as a hook and is called after an reconfiguration
```

(continues on next page)

(continued from previous page)

```
     * request was received, and AFTER this reconfiguration request has been
     * merged into this device's current state.
     * You may access any (updated or not) parameters using the usual
     * getters and setters.
     * @code
     * int i = get<int>("myParam");
     * @endcode
     */
    virtual void postReconfigure();


    private:

    void initialize();

    void start();

    void stop();

    void reset();

    };
}

#endif
```

... and explain what is happening step by step.

The include statement

```
#include <karabo/karabo.hpp>
```

provides you access to the full Karabo framework. Both include paths and namespaces follow the physical directory layout of the Karabo framework sources. Karabo comprises the following main functionalities (reflected as source directories):

- util: Factories, Configurator, Hash, Schema, String and Time tools, etc.
- io: Serializer, Input, Output, FileIO tools
- io/h5: HDF5 interface (HDF5, Hash serialization)
- log: unified logging using Log4Cpp as engine
- webAuth: Webservice based authentification (based on gsoap)
- net: TCP (point to point) and JMS (broker-based) networking in synchronous and asynchronous fashion.
- xms: Higher level communication API (Signals & Slots, Request/Response, etc.)
- xip: Image classes, processing, GPU code
- core: Device, DeviceServer, DeviceClient base classes

Consequently, if you want to include less, you can refer to a header of a specific functionality (like in boost, e.g. <karabo/util.hpp>, or <karabo/io.hpp>) or of a single class (e.g. <karabo/webAuth/Authenticator.hh>).

It is good practice to place your class into the karabo namespace

```
namespace karabo {

    class Conveyor : public karabo::core::Device<> {
```

Any device must by some means derive from the templated class Device<>, the template indicating which interface class to use (we look later to this). In the simplest case you leave the template empty (like here) and solely derive from the Device<> base class.

The KARABO_CLASSINFO macro

```
KARABO_CLASSINFO(Conveyor, "Conveyor", "2.0")
```

adds what C++ does not provide by default: reflection (or introspection) information. It for example defines

```
typedef Self Conveyor;
```

this is convenient to use e.g. in generic template code. Even more important is the string identifier for the class, called `classId`. The configurator system will utilize this information for factory-like object construction. The final argument (e.g. 2.0) indicates with which Karabo framework version a device is compatible with. Only one version should be given here and it should only be specified up the minor version number.

The expected parameter function

```
static void expectedParameters(karabo::util::Schema& expected);
```

is where you should describe what properties and commands are available on this device. The function is static in order to be evaluated before actual device instantiation and to generate meaningful graphical widgets that guide users in setting up the initial device configuration. This function is called several times (whenever some other party needs to know about your device's schema).

The constructor

```
Conveyor(const karabo::util::Hash& config);
```

is called-back by the configurator mechanism. Otherwise it is a regular constructor.

> **Warning:** While still being constructed the device does not (fully) interact with the distributed system. You should thus not have long running, or even blocking code in the constructor. Such code belongs in initialization functions.

If you are managing your own threads in the device, which need joining or allocated heap memory that you need to free upon device destruction, the destructor is the place for doing so. It is guaranteed to be called, whenever a device instance gets killed.

```
virtual ~Conveyor();
```

The `preReconfigure` and `postReconfigure` functions,

```
virtual void preReconfigure(karabo::util::Hash& incomingReconfiguration);
virtual void postReconfigure()
```

are called after a reconfiguration request on the device's properties has been received, respectively *before* and *after* the new configuration has been merged into the device's state.

Karabo conceptually distinguishes between the execution of commands (state-machine event triggers) and setting of properties. A command execution may be followed by a state change, whilst property setting **should not** lead to a state change.

In our example: starting if the conveyor would utilize a property setting and a command. First a "targetSpeed" property would be set (no state change), and afterwards a "start" command would be issued which actually triggers the state-machine and drives it into "Starting" and finally "Started" state.

This conceptual separation is reflected in all APIs and the two functions above reflect the hook into the property configuration system.

The remaining functions reflect each *command* that is available on this device.

```
void initialize();

void start();

void stop();

void reset();
```

Now let us have a look at the implementation, here is the complete file

```
#include "Conveyor.hh"

using namespace std;

USING_KARABO_NAMESPACES;

namespace karabo {


    KARABO_REGISTER_FOR_CONFIGURATION(BaseDevice, Device<>, Conveyor);


    void Conveyor::expectedParameters(Schema& expected) {

    OVERWRITE_ELEMENT(expected).key("state")
        .setNewOptions("Initializing,Error,Started,Stopping,Stopped,Starting")
        .setNewDefaultValue(States::INIT)
        .commit();

    SLOT_ELEMENT(expected).key("start")
        .displayedName("Start")
        .description("Instructs device to go to started state")
        .allowedStates(States::STOPPED)
        .commit();

    SLOT_ELEMENT(expected).key("stop")
        .displayedName("Stop")
        .description("Instructs device to go to stopped state")
        .allowedStates(States::STARTED)
        .commit();


    SLOT_ELEMENT(expected).key("reset")
        .displayedName("Reset")
        .description("Resets in case of an error")
        .allowedStates(States::ERROR)
```

(continues on next page)

(continued from previous page)

```
        .commit();

    FLOAT_ELEMENT(expected).key("targetSpeed")
        .displayedName("Target Conveyor Speed")
        .description("Configures the speed of the conveyor belt")
        .unit(Unit::METER_PER_SECOND)
        .assignmentOptional().defaultValue(0.8)
        .reconfigurable()
        .commit();

    FLOAT_ELEMENT(expected).key("currentSpeed")
        .displayedName("Current Conveyor Speed")
        .description("Shows the current speed of the conveyor")
        .readOnly()
        .initialValue(0.0)
        .commit();

    BOOL_ELEMENT(expected).key("reverseDirection")
        .displayedName("Reverse Direction")
        .description("Reverses the direction of the conveyor band")
        .assignmentOptional().defaultValue(false)
        .allowedStates(States::STOPPED)
        .reconfigurable()
        .commit();

    BOOL_ELEMENT(expected).key("injectError")
        .displayedName("Inject Error")
        .description("Does not correctly stop the conveyor,
                      such that a Error is triggered during next start")
        .assignmentOptional().defaultValue(false)
        .reconfigurable()
        .expertAccess()
        .commit();
    }


    Conveyor::Conveyor(const karabo::util::Hash& config) : Device<>(config) {

    // Register initialState member function to be called after the run()
    // member function is called
    registerInitialFunction(initialize);

    KARABO_SLOT(start);
    KARABO_SLOT(stop);
    KARABO_SLOT(reset);
    }


    void Conveyor::preReconfigure(karabo::util::Hash& config) {

    // The preReconfigure hook allows to forward the configuration to
    // some connected h/w
```

(continues on next page)

(continued from previous page)

```
    try {

        if (config.has("targetSpeed")) {
        // Simulate setting to h/w
        KARABO_LOG_INFO << "Setting to hardware: targetSpeed -> "
                        << config.get<float>("targetSpeed");
        }

        if (config.has("reverseDirection")) {
        // Simulate setting to h/w
        KARABO_LOG_INFO << "Setting to hardware: reverseDirection -> "
                        << config.get<bool>("reverseDirection");
        }

    } catch (...) {
        // You may want to indicate that the h/w failed
        updateState(States::ERROR);
    }
    }


    void Conveyor::postReconfigure() {
    }


    void Conveyor::initialize() {
    // As the Initializing state is not mentioned in the allowed states
    // nothing else is possible during this state
    updateState(States::INIT);

    KARABO_LOG_INFO << "Connecting to conveyor hardware...";

    // Simulate some time it could need to connect and setup
    boost::this_thread::sleep(boost::posix_time::seconds(2));

    // Automatically trigger got the Stopped state
    stop();
    }


    void Conveyor::start() {
    updateState(States::STARTING); // use this if long-lasting work follows ...

    // Retrieve current values from our own device-state
    float tgtSpeed = get<float>("targetSpeed");
    float currentSpeed = get<float>("currentSpeed");

    // If we do not stand still here that is an error
    if (currentSpeed > 0.0) {
        KARABO_LOG_ERROR << "Conveyor does not stand still at start-up";
        updateState(States::ERROR);
```

(continues on next page)

(continued from previous page)

```
    return;
}

// Separate ramping into 50 steps
float increase = tgtSpeed / 50.0;

// Simulate a slow ramping up of the conveyor
for (int i = 0; i < 50; ++i) {
    currentSpeed += increase;
    set("currentSpeed", currentSpeed);
    boost::this_thread::sleep(boost::posix_time::millisec(50));
}
// Be sure to finally run with targetSpeed
set<float>("currentSpeed", tgtSpeed);

updateState(States::STARTED);

}


void Conveyor::stop() {
updateState(States::STOPPING); // use this if long-lasting work follows ...

// Retrieve current value from our own device-state
float currentSpeed = get<float>("currentSpeed");

if (currentSpeed != 0.0f) {
    // Separate ramping into 50 steps
    float decrease = currentSpeed / 50.0;

    // Simulate a slow ramping down of the conveyor
    for (int i = 0; i < 50; ++i) {
    currentSpeed -= decrease;
    set("currentSpeed", currentSpeed);
    boost::this_thread::sleep(boost::posix_time::millisec(50));
    }
    // Be sure to finally stand still
    if (get<bool>("injectError")) {
    set<float>("currentSpeed", 0.1);
    } else {
    set<float>("currentSpeed", 0.0);
    }
}
updateState(States::STOPPED);
}


void Conveyor::reset() {
set("injectError", false);
set<float>("currentSpeed", 0.0);
initialize();
}
```

(continues on next page)

(continued from previous page)

```
}
```

and go through it step by step.

The macro

```
KARABO_REGISTER_FOR_CONFIGURATION(BaseDevice, Device<>, Conveyor)
```

registers the device to the BaseDevice configurator factory. The expected parameters of all classes mentioned in this macro will be evaluated and concatenated from left to right. In this way our Conveyor device inherits all expected parameters from BaseDevice (which has none), and from Device<> (which has a few).

In the expectedParameters() function the parameters for this device are defined.

The constructor

```
Conveyor::Conveyor(const karabo::util::Hash& config) : Device<>(config) {

    // Register initialState member function to be called after the run()
    // member function is called
    registerInitialFunction(initialize);

    KARABO_SLOT(start);
    KARABO_SLOT(stop);
    KARABO_SLOT(reset);
}
```

does not deal with the provided configuration, despite calling the parent class's constructor with it (as is proper C++). This is completely fine for two reasons: 1. The provided configuration got validated BEFORE the constructor was even called. 2. The Device<> base class manages the configuration and provides access to it with its getters and setters.

Of course you can create a member variable and assign it by using the value in the configuration passed, like:

```
Conveyor::Conveyor(const karabo::util::Hash& config) : Device<>(config) {
    m_speed = config.get<string>("targetSpeed");
}
```

but then you have to be careful to keep this member variable in sync! You should update it yourself in the postReconfiguration() function.

Note: It is generally not recommended to keep any private members as copies of configuration variables. Karabo's setters and getters will perform well enough for most use cases and assure that the device properties are kept synchronized with your configuration.

As said before, no long lasting or even blocking activities should be implemented in the constructor. For that reason a macro is available (`registerInitialFunction`) which allows to bind a function that acts like a "second constructor". In this function you can write code without the restrictions of the constructor. Use this function if you need to interact with properties of the device.

Warning: In the constructor you should only access devices properties through the configuration hash passed to it. Accessing properties with getter/setter methods upon initialization should happen in the function registered via registerInitialFunction.

The last three statements in the constructor make the otherwise regular functions start, stop and reset callable from the distributed system (slots).

Note: The function names must match the key names of the SLOT_ELEMENTs defined in the expectedParameters function. Only then will the automatically generated GUI or the command-line interface execute the slot bound to a given SLOT_ELEMENT.

Functions mapping to slots in node elements should replace any "." separators in the expected parameter key with underscores ("_").

The function preReconfigure

```
void Conveyor::preReconfigure(karabo::util::Hash& config) {

    // The preReconfigure hook allows to forward the configuration
    // to some connected h/w

    try {

        if (config.has("targetSpeed")) {
            // Simulate setting to h/w
            KARABO_LOG_INFO << "Setting to hardware: targetSpeed -> "
                            << config.get<float>("targetSpeed");
        }

        if (config.has("reverseDirection")) {
            // Simulate setting to h/w
            KARABO_LOG_INFO << "Setting to hardware: targetSpeed -> "
                            << config.get<bool>("reverseDirection");
        }

    } catch (...) {
        // You may want to indicate that the h/w failed
        updateState(States::ERROR);
    }
}
```

acts as a hook *before* the requested reconfiguration is merged to the device's internal state. All potential reconfiguration requests are packaged into the config hash. You should check yourself for the ones you are interested in.

For this you can use the *has* function of the Hash object like here:

```
if (config.has("targetSpeed")) {
    // Simulate setting to h/w
    KARABO_LOG_INFO << "Setting to hardware: targetSpeed -> "
                    << config.get<float>("targetSpeed");
}
```

As we only simulate a conveyor h/w, we send a message instead, pretending we did something. Messages using the KARABO_LOG_ prefix will be visible to the users (distributed via the broker), they come in four categories: DEBUG, INFO, WARN and ERROR.

..warning:

```
Use log messaging sparsely to not pollute the network and the log-files. If you
need messages for local debugging use the KARABO_LOG_FRAMEWORK\_ in
combination with DEBUG, INFO, WARN and ERROR instead. Also, please refer to
Section :ref:`alarm_system`.
```

Before looking closer at the initialize function, let's list some best practices for all call-back functions (mostly slots) of Karabo:

1. Never completely block and rely on another function to unblock it
2. Always update the state
3. Only use try/catch blocks if you want to react on an exception (by driving the device into *ERROR* state for example). Otherwise trust Karabo to handle them.

Now, in the initialize function (which is automatically called once the constructor finished execution)

```
void Conveyor::initialize() {
    // As the Initializing state is not mentioned in the allowed states
    // nothing else is possible during this state
    updateState(States::INIT);

    KARABO_LOG_INFO << "Connecting to conveyor hardware...";

    // Simulate some time it could need to connect and setup
    boost::this_thread::sleep(boost::posix_time::seconds(2));

    // Automatically trigger got the Stopped state
    stop();
}
```

you see an immediate call to updateState. This is good practice, as the following activity (namely connecting to the motor) may take some time (here simulated to be two seconds). Most importantly the GUI will be nicely graying out other buttons and informing the user what is happening. Once connected we internally call the stop command (in reality one should ask the hardware which state it is in and adapt accordingly).

We are almost done. Start and stop are very similar and reset is almost trivial, so lets only look at the start function:

```
void Conveyor::start() {
    updateState(States::STARTING); // use this if long-lasting work follows ...

    // Retrieve current values from our own device-state
    float tgtSpeed = get<float>("targetSpeed");
    float currentSpeed = get<float>("currentSpeed");

    // If we do not stand still here that is an error
    if (currentSpeed > 0.0) {
        KARABO_LOG_ERROR << "Conveyor does not stand still at start-up";
        updateState(States::ERROR);
        return;
    }

    // Separate ramping into 50 steps
    float increase = tgtSpeed / 50.0;
```

(continues on next page)

(continued from previous page)

```
    // Simulate a slow ramping up of the conveyor
    for (int i = 0; i < 50; ++i) {
        currentSpeed += increase;
        set("currentSpeed", currentSpeed);
        boost::this_thread::sleep(boost::posix_time::millisec(50));
    }
    // Be sure to finally run with targetSpeed
    set<float>("currentSpeed", tgtSpeed);

    updateState(States::STARTED);

}
```

We simulate a slow ramping up of the speed and explicitly inform about that using the intermediate state *STARTING*.

```
void Conveyor::start() {
    updateState(States::STARTING); // use this if long-lasting work follows ...
```

In the following lines you can see how properties of your device (which must always be part of the expectedParameters) can be read. A call to get is always thread-safe and always returns the latest value configured.

```
// Retrieve current values from our own device-state
float tgtSpeed = get<float>("targetSpeed");
float currentSpeed = get<float>("currentSpeed");
```

The next part shows one example to potentially drive your device into an *ERROR* state. Here we check whether the conveyor stands still before starting it. Note the return statement to finish the execution of the function.

The last part of the start function simulates the ramping up by giving several updates on the "currentSpeed" property with some fixed delay. Setting a property value like here for "currentSpeed" does two things: it updates its own device state and publishes this value to the broker so that interested clients will get an event.

CHAPTER

TWO

THE PIPELINE INTERFACE

The conveyor belt example given in the introduction of the C++ and Python APIs is typical for a device that represents some hardware that is controlled and monitored. No data is *produced* by these devices, but just property changes are communicated. This is done via the central message broker.

Other devices like cameras produce large amounts of data that cannot be distributed this way. Instead, the data should be sent directly from one device producing it to one or more other devices that have registered themselves at the producer for that purpose. In the Karabo framework this can be done using a point-to-point protocol between so called output and input channels of devices.

2.1 Sending Data: Output Channel

First, we have to tell the framework what data is to be sent via the output channel, i.e. to declare its schema. This is done inside the `expectedParameters` method. Here is an example of a device sending a 32-bit integer, a string and a vector of 64-bit integers:

```
Schema data;
INT32_ELEMENT(data).key("dataId")
        .readOnly()
        .commit();

STRING_ELEMENT(data).key("string")
        .readOnly()
        .commit();

VECTOR_INT64_ELEMENT(data).key("vector_int64")
        .readOnly()
        .commit();
```

Next (but still within `expectedParameters`), the output channel has to be declared. Here we create one with the key *output*:

```
OUTPUT_CHANNEL(expected).key("output")
        .displayedName("Output")
        .dataSchema(data)
        .commit();
```

Whenever the device should write data to this output channel, a `Hash` should be created that matches this schema

```
Hash data;
data.set("dataId", 5);
```

(continues on next page)

(continued from previous page)

```
data.set("string", std::string("This is a string to be sent."));
std::vector<long long> vec = ...; // create and fill the array here
data.set("vector_int64", vec);
```

Note that Karabo does not check that the data sent matches the declared schema.

Finally, the data is sent by calling the device method

```
this->writeChannel("output", data);
```

with the key of the channel as the first and the `Data` object as the second argument. The current timestamp will be added to the `Data` object as a meta data information.

Once the data stream is finished, i.e. no further data is to be sent, the end of stream method has to be called with the output channel key as argument to inform all input channels that receive the data:

```
this->signalEndOfStream("output");
```

2.2 Receiving Data: Input Channel

For input channels one also needs to declare what data they expect to receive. This is done in exactly the same way as for output channels inside the `expectedParameters` method. Declaring the input channel is also analogue to the way an output channel is declared:

```
INPUT_CHANNEL(expected).key("input")
        .displayedName("Input")
        .description("Input channel: client") // optional, for GUI
        .dataSchema(data)
        .commit();
```

The next step is to prepare a member function of the device that should be called whenever new data arrives. The signature of that function has to be

```
void onData(const karabo::util::Hash& data,
            const karabo::xms::InputChannel::MetaData& meta);
```

Inside the function the data sent can be unpacked from the Hash:

```
int id = data.get<int>("dataId");
const std::string& str = data.get<std::string>("string");
const vector<long long>& vec = data.get<std::vector<long long> >("vector_int64");
```

Finally, the framework has to be informed that this method should be called whenever data arrives. This has to be done in the `initialize()` member function (or, more precisely, in the function registered in the constructor using the `KARABO_INITIAL_FUNCTION` macro) in the following way:

```
KARABO_ON_DATA("input", onData);
```

with the key of the input channel as first and the function name as the second argument.

A similar macro can be used to register a member function that should be called when the data stream terminates, i.e. when the sending device calls `this->signalEndOfStream("<output channel name>");`:

```
KARABO_ON_EOS("input", onEndOfStream);
```

The signature of this member function has to be

```
void onEndOfStream(const karabo::xms::InputChannel::Pointer& input);
```

Note: A simple way of ensuring that input and output channels work with the same data schema is to move schema creation to a static function which is available to all devices working on this type of data, e.g. by means of a dependency or library.

2.3 Hierarchies in the Schema

The data that is sent from an output to an input channel can have a hierarchical structure. This structure is declared in the usual way in `expectedParameters`, for both input and output channels:

```
Schema data;
// Add whatever data on first hierarchy level:
// ...
// First level done - now add second level:
NODE_ELEMENT(data).key("node")
        .commit();

FLOAT_ELEMENT(data).key("node.afloat")
        .readOnly()
        .commit();
```

When writing to an output channel, one first has to create and fill the node. Then the node can be added and the data can be sent:

```
Hash data; // top level data structure
// Here e.g. fill top level content:
// ...
Hash node;
float floatValue = 1.3f;  // or whatever...
node.set("afloat", floatValue);
data.set("node", node);
this->writeChannel("output", data);
```

In the `onData` member function of a device receiving the data in an input channel, the node can be unpacked in the following way:

```
void onData(const karabo::xms::Data& data,
            const karabo::xms::InputChannel::MetaData& meta);
{
  // ...
  Hash node(data.get<Hash>("node"));
  const float afloat = node.get<float>("afloat");
  // ...
}
```

2.4 Treatment of Array Data

Arrays are described in Karabo using the NDArray class.

An NDArray consists of typed data and a shape.

It is meant to map directly to a numpy.ndarray object in the Bound API, so its interface closely matches numpy.ndarray.

```
NDARRAY_ELEMENT(expected).key("arrayStack")
        .shape("-1,100,100") // Variable dimension along the slowest axis
        .readOnly().noInitialValue()
        .commit();
```

In the above example -1 in the shape definition indicates a variable size of this dimension; e.g. the first dimension is of variable size. If the shape contains no negative numbers, the array is said to have a 'fixed' shape.

In Python, a transparent conversion to and from numpy.ndarray elements is performed:

```
a = np.ones((10, 100, 100))
self.set("arrayStack", a)
b = self.get("arrayStack")
type(b)
>>> numpy.ndarray

c = np.ones((10, 10, 100))
self.set("arrayStack", c)
>>> ValueError("Setting 'arrayStack' failed because dimension 2 in \
(10, 10, 100) mismatched array shape definition (-1, 100, 100)")
```

The NDArray C++ class is a convenience class meant to simplify supporting n-dimensional arrays within the Device and Hash classes. In C++ the Device::set method is overwritten to accept NDArray objects directly:

```
typedef std::vector<double> DoubleVector;
typedef boost::shared_ptr<DoubleVector> DoubleVectorPtr;
DoubleVectorPtr v(new DoubleVector(10*100*100, 1));
NDArray<double> arr(v, Dims(10, 100, 100));
set("arrayStack", arr);
// ... Then access the array
NDArray a = get<NDArray>("arrayStack");
const Dims & d = a.getDims();
DoubleVectorPtr v1 = a.getData();
```

Using the above constructor no copy of the data is performed. Alternatively, the a copying constructor may be used

```
typedef std::vector<double> DoubleVector;
typedef boost::shared_ptr<DoubleVector> DoubleVectorPtr;
DoubleVector v(10*100*100, 1);
NDArray arr(v, Dims(10, 100, 100));
set("arrayStack", arr);
// ... Then access the array
NDArray<double> a = get<NDArray >("arrayStack");
const Dims & d = a.getDims();
DoubleVectorPtr v1 = a.getData();
```

In this case NDArray will create a copy of the data, but internally also maintains it as a boost::shared_ptr, thus

avoiding additional copies from there on. In either case access to the data is via a `boost::shared_ptr` using `getData()`.

Internally, `NDArray` uses a `ByteArray` to hold its data, while additionally, defining the shape as an attribute in a standardized fashion. ``NDArray``s can be placed and retrieved from Hashes in the accustomed way:

2.5 Treatment of Image Data

As with array data, image data can similarly be sent using the class `ImageData` which extends on-top of the `NDArray` class with some predefined properties, i.e. it serves as a special node with convenience methods for conversions to and from more useful image data formats. The schema of an output channel for image data is defined in `expectedParameters` as follows:

```
Schema data;
IMAGEDATA(data).key("image")
        .encodingType(karabo::xms::Encoding::RGBA)
        .bitsPerPixel(12)
        .isBigEndian(true)
        .commit();

OUTPUT_CHANNEL(expected).key("output") // or any other key
        .displayedName("Output")       // or whatever name you choose
        .dataSchema(data)
        .commit();
```

For input channels simply replace `OUTPUT_CHANNEL` by `INPUT_CHANNEL`.

Image data refers to array-like data from camera interfaces. It may be represented as an ``IMAGEDATA`` (or ``IMAGEDATA_ELEMENT``) element, which has fixed properties appropriate to the camera origin of the data. These are:

- pixels: The N-dimensional array containing the pixels
- dims: The length of the array reflects total dimensionality and each element the extension in this dimension
- dimTypes: Any dimension should have an enumerated type
- dimScales: Dimension Scales
- encoding: Describes the color space of pixel encoding of the data (e.g. GRAY, RGB, JPG, PNG etc
- bitsPerPixel: The number of bits needed for each pixel
- roiOffsets: Describes the offset of the Region-of-Interest; it will contain zeros if the image has no ROI defined
- geometry: optional hierarchical detector geometry information
- header: Hash containing user-defined header data

2.6 Interface *per TCP Message*

Point-to-point communication in the Karabo framework generally uses TCP for data transfer between devices. Whenever `writeChannel` is called for an output channel, the data is sent as a separate message to all connected input channels. There might be circumstances where it is advantageous to pack more than one data item into a TCP message. For this a lower level API is provided as described in the following.

To sent several data items in a single TCP message, the following few lines of code should be used instead of `this->writeChannel(channelName, data)`:

```
data.attachTimestamp(this->getActualTimestamp());
karabo::xms::OutputChannel::Pointer channel = this->getOutputChannel(channelName);
channel->write(data);
```

Once there is enough data accumulated to be actually sent,

```
channel->update();
```

has to be called.

For a device with an input channel it does not matter much whether several data items that it receives have been sent in a single TCP message or not. A member function registered with `KARABO_ON_DATA` will be called for each item. Nevertheless, in case it matters which data items are sent together (which should not be the case), the device can register a method that receives all data items in one go. Instead of using `KARABO_ON_DATA`, such a method has to be registered using `KARABO_ON_INPUT`. The signature of this method has to be

```
void onInput(const karabo::xms::InputChannel::Pointer& input);
```

Inside the method one has to loop over the data items. Finally one has to tell the `InputChannel` that reading the data is done by calling `update()` at the very end of the method:

```
for (size_t i = 0; i < input->size(); ++i) {
     Hash data(input->read(i));
     ... // whatever you want to do with the data
 }
 // Tell the input channel that you are done with all data
 input->update();
```

2.7 Compliance with Data Management

While the pipeline processing interface generally allows free form Hashes to be passed between devices, leaving it up to the device logic to correctly interpret these, there are limitations if data is be written to or retrieved from the data management system. Specifically, Hashes need to follow a certain structure, and the concept of meta-data needs to be understood.

2.7.1 Meta Data

So far we have simply written to output channels and ignored the fact that each data token written has meta data pertinent to it. This meta data currently contains source and timing information, but is by design extensible. If not explicitly set, the source corresponds to the writing device's id and the output channel name, and the timing information to the train and timestamp for when the data was written. Frequently, source information should be maintained though, i.e. the writing device is *not* the data producer. In this case we explicitly set the source or forward existing meta data:

```
using namespace karabo::xms;
OutputChannel::Pointer channel = this->getOutputChannel(channelName);

for (size_t i = 0; i < input->size(); ++i) {
     Hash data;
     const InputChannel::MetaData& meta = (input->read(i, data));
     ... // whatever you want to do with the data
     channel->write(data, meta);

     const InputChannel::MetaData& meta2;
     meta2.setSource("myProcessor");
     channel->write(data, meta2);

 }
 // Tell the input channel that you are done with all data
 input->update();
```

Metadata can be accessed either via `read` or by indices. Data tokens for the same source can be written subsequently to the same output channel, allowing e.g. to bunch multiple trains before actually writing data out to the network.

Warning: The data management service expects only one train per data token per source to be sent to it.

In all cases the source information will be used by the data management system to correlate incoming data with data producers.

2.7.2 Hash Structure

For data interacting with the data management system and additional restriction applies in terms of Hash structure. Generally, data of similar types is organized hierarchical in nodes. The following data types exist:

Train Data

Is data that occurs on a per train basis. It can be in form of scalars, vectors or arrays. For interaction with the data management system a data token written to the channel always corresponds to a train. The Hash that is written must match the following policy:

- on the root hierarchy level an unsigned long long element giving the trainId exists
- freely named nodes enclosing exist on the same hierarchy level, which have an attribute `daqDataType` set to `TRAIN`. Underneath these nodes scalar vector and array elements exist.

Pulse Data

Pulse data is data that has pulse resolution. Data can either exist for all pulses of a train or a subset. In either case the following limitations apply:

- the enclosing node element has the `daqDataType` set to `PULSE`.
- a vector unsigned long long element with key `pulseId` needs to be located directly underneath this node element
- any number of freely vector elements with the same length as `pulseId` may reside underneath this node, or in further sub nodes. There is a 1:1 relation between the index in these elements and the pulse id given in `pulseId` at this index.
- any number of freely NDArray elements with the last dimension of the same length as `pulseId` may reside underneath this node, or in further sub nodes. There is a 1:1 relation between the index of the last dimension in these elements and the pulse id given in `pulseId` at this index. The other dimensions may not change from data token to data token.

There may be any number of these node elements, all following the above structure. They may be freely named, except that the key `masterPulseData` is reserved.

All train and pulse data elements must always be present in all hashes, even if the arrays or vectors are empty.

All train and pulse data elements must be specified in the output channel's data schema. Adding additional elements in between data tokens, specifically between runs is not allowed.

2.7.3 Defining and Configuring Topologies

Pipelined processing in Karabo supports a variety of recurring topologies defining how data is passed through the system.

Copying vs. Sharing Data

An input channel may selected if it would like to receive data in *copy* or *shared* mode. In the first case it will receive a copy of all data sent by output channel it is connected to. In shared mode, the output channel is allowed to balance data distribution according to how it is configured. There are two options on the output channel:

round robin
: distributes data evenly on all connected input channels operating in shared mode. As indicated by the option name channels subsequently get data send to them. If the next channel in line is not available yet, writes to the output channel block until the data can be sent to this channel.

load-balanced
: distributes data on all connected input channels but does not enforce a particular distribution order. Upon writing data to an output channel it is sent to the next available input channel. This scenario should be used if data recipients are expected to have different processing times on data packages.

Best-Effort and Assured Delivery Configuration

Both input and output channels may be configured on what to do if the counter part is not available, i.e. no input is ready to receive data from a given output. Options are to

throw
: an exception.

queue
: the data and deliver it once an input becomes available. The write call to the output channel will not block.

wait
: for an input to become available, effectively blocking the write call to the output channel.

drop
: the data being sent out, i.e. do not send it but proceed with-out blocking the write call to the output.

Note: Queuing data may involve significant memory usage and thus should be used with care if large amounts of data are passed.

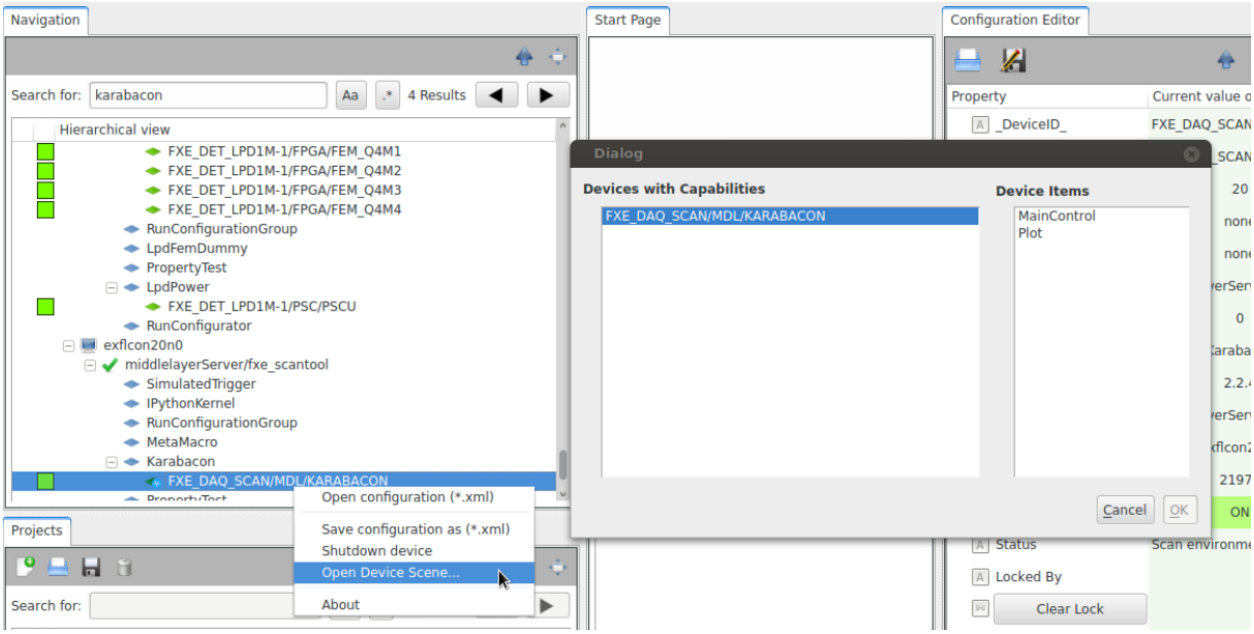
By default the channels are configured to *wait* behavior, which assures delivery but has the side effect of possibly stalling a complete processing pipeline by back-propagation. If a pipeline device with an input and output channels is used as a pipeline end-point, it is important to configure the last, unconnected output to drop to avoid this scenario from happening.

CHAPTER

THREE

DEVICE SCENES

Karabo provides a protocol for devices to share predefined scenes. These allows the author of a device to provide what they think are a good starting point. Moreover, these are easily accessible from the topology panel in the GUI:



A default scene can also be accessed by double-clicking on a device.

This section shows how to enable your device to have builtin scenes.

Implementing this functionality requires the creation of a scene in the scene editor, its conversion to a C++ *ostringstream*, and adding the *requestScene* slot to the device.

3.1 From Scene To Header File

Begin by drawing an adequate scene in the GUI's scene editor, and save it locally on your computer as SVG (right-click on scene -> *Save to File*).

SVG is XML-based, and thus can be easily edited. Convert the SVG to a header file *scenes.hh* with a function. Thus the original SVG:

```
<?xml version="1.0"?>
<svg:svg xmlns:krb="http://karabo.eu/scene" xmlns:svg="http://www.w3.org/2000/svg"␣
```

(continues on next page)

(continued from previous page)

```
↪height="768" width="1024" krb:version="2">
       <svg:g krb:class="BoxLayout" krb:direction="0" krb:height="33" krb:width="87"␣
↪krb:x="30" krb:y="26">
       <svg:rect height="24" width="39" x="0" y="0" krb:background="transparent"␣
↪krb:class="Label" krb:font="Sans,9,-1,5,50,0,0,0,0,0" krb:foreground="#000000"␣
↪krb:frameWidth="0" krb:text="Start"/>
       <svg:rect height="23" width="46" x="0" y="0" krb:class="DisplayComponent"␣
↪krb:keys="Generator.start" krb:requires_confirmation="false" krb:widget="DisplayCommand
↪"/>
</svg:g>
<svg:g krb:class="BoxLayout" krb:direction="0" krb:height="33" krb:width="81" krb:x="35"␣
↪krb:y="61">
       <svg:rect height="24" width="37" x="0" y="0" krb:background="transparent"␣
↪krb:class="Label" krb:font="Sans,9,-1,5,50,0,0,0,0,0" krb:foreground="#000000"␣
↪krb:frameWidth="0" krb:text="Stop"/>
       <svg:rect height="23" width="43" x="0" y="0" krb:class="DisplayComponent"␣
↪krb:keys="Generator.stop" krb:requires_confirmation="false" krb:widget="DisplayCommand
↪"/>
</svg:g>
<svg:g krb:class="BoxLayout" krb:direction="0" krb:height="33" krb:width="145" krb:x="150
↪" krb:y="33">
       <svg:rect height="24" width="95" x="0" y="0" krb:background="transparent"␣
↪krb:class="Label" krb:font="Sans,9,-1,5,50,0,0,0,0,0" krb:foreground="#000000"␣
↪krb:frameWidth="0" krb:text="Float property"/>
       <svg:rect height="23" width="49" x="0" y="0" krb:class="DisplayComponent"␣
↪krb:keys="Generator.floatProperty" krb:widget="DisplayLabel"/>
</svg:g>
<svg:g krb:class="BoxLayout" krb:direction="0" krb:height="329" krb:width="514" krb:x="36
↪" krb:y="119">
       <svg:rect height="24" width="47" x="0" y="0" krb:background="transparent"␣
↪krb:class="Label" krb:font="Sans,9,-1,5,50,0,0,0,0,0" krb:foreground="#000000"␣
↪krb:frameWidth="0" krb:text="image"/>
       <svg:rect height="319" width="465" x="0" y="0" krb:class="DisplayComponent"␣
↪krb:keys="Generator.output.schema.image" krb:show_axes="true" krb:show_color_bar="true
↪" krb:show_tool_bar="true" krb:widget="WebcamImage"/>
</svg:g>
</svg:svg>
```

Becomes:

```cpp
#include <sstream>

std::string getControls(const std::string& instanceId) {
    std::ostringstream ret;
    // SVG header
    ret << "<?xml version=\"1.0\"?>"
        << "<svg:svg xmlns:krb=\"http://karabo.eu/scene\" xmlns:svg=\"http://www.w3.org/
↪2000/svg\" "
        << "height=\"119\" width=\"150\" krb:version=\"2\">"
    // Start button label
        << "<svg:g krb:class=\"BoxLayout\" krb:direction=\"0\" krb:height=\"33\"␣
↪krb:width=\"87\" krb:x=\"30\" krb:y=\"26\">"
```

(continues on next page)

(continued from previous page)

```
        << "<svg:rect height=\"24\" width=\"39\" x=\"0\" y=\"0\" krb:background=\
↪"transparent\" krb:class=\"Label\" krb:font=\"Sans,9,-1,5,50,0,0,0,0,0\" "
        << "krb:foreground=\"#000000\" krb:frameWidth=\"0\" krb:text=\"Start\"/>"
    // Start button itself
        << "<svg:rect height=\"23\" width=\"46\" x=\"0\" y=\"0\" krb:class=\
↪"DisplayComponent\" "
        << "krb:keys=\"" << instanceId << ".start\" krb:requires_confirmation=\"false\"␣
↪krb:widget=\"DisplayCommand\"/></svg:g>"
    // Stop button label
        << "<svg:g krb:class=\"BoxLayout\" krb:direction=\"0\" krb:height=\"33\"␣
↪krb:width=\"81\" krb:x=\"35\" krb:y=\"61\">"
        << "<svg:rect height=\"24\" width=\"37\" x=\"0\" y=\"0\" krb:background=\
↪"transparent\" krb:class=\"Label\" krb:font=\"Sans,9,-1,5,50,0,0,0,0,0\" "
        << "krb:foreground=\"#000000\" krb:frameWidth=\"0\" krb:text=\"Stop\"/>"
    // Stop button itself
        << "<svg:rect height=\"23\" width=\"43\" x=\"0\" y=\"0\" krb:class=\
↪"DisplayComponent\" "
        << "krb:keys=\"" << instanceId << ".stop\" krb:requires_confirmation=\"false\"␣
↪krb:widget=\"DisplayCommand\"/></svg:g>"
    // SVG footer
        << "</svg:svg>";
    return ret.str();
}
```

Add this file to your project.

3.2 Providing The Scene From Your Device

Add a read-only *VECTOR_STRING_ELEMENT* property called *availableScenes* to your expected parameters, and register and implement the *requestScene* slot. This is a predefined slot, which allows various actors to understand the scene protocol.

The slot takes a Hash *params* and lets the device reply with a Hash containing the origin, its datatype (*deviceScene*), and the scene serialized as xml:

```
#include "scenes.hh"

using namespace karabo::util;

// Define the list of scenes
void MyDevice::expectedParameters(karabo::util::Schema& expected) {

   VECTOR_STRING_ELEMENT(expected).key("availableScenes")
       .setSepcialDisplayType(KARABO_SCHEMA_DISPLAY_TYPE_SCENES)
       .readOnly().initialValue(std::vector<std::string>{"controls"})
       .commit();
}

// Register in the constructor that we have this functionality
MyDevice::MyDevice(const karabo::util::Hash& config)
    : karabo::core::Device<>(config)
```

(continues on next page)

(continued from previous page)

```
{
    KARABO_SLOT(requestScene, karabo::util::Hash);
}

// The function that provides the scene
void MyDevice::requestScene(const karabo::util::Hash& params) {
    Hash result("type", "deviceScene", "origin", this->getInstanceId());
    Hash& payload = result.bindReference<Hash>("payload");

    payload.set("success", true);
    payload.set("name", "controls");
    payload.set("data", getControls());

    this->reply(result);
}
```

3.3 Providing Several Scenes

Would you want to provide several scenes (e.g., simple overview and control scene), you can define several functions in *scenes.hh*, and modify *requestScene* to check *name* in *params*:

```
#include "scenes.hh"

using namespace karabo::util;

// Define the list of scenes
void MyDevice::expectedParameters(karabo::util::Schema& expected) {

   VECTOR_STRING_ELEMENT(expected).key("availableScenes")
       .setSepcialDisplayType(KARABO_SCHEMA_DISPLAY_TYPE_SCENES)
       .readOnly().initialValue(std::vector<std::string>{"overview", "controls"})
       .commit();
}

// Register in the constructor that we have this functionality
MyDevice::MyDevice(const karabo::util::Hash& config)
    : karabo::core::Device<>(config)
{
    KARABO_SLOT(requestScene, karabo::util::Hash);
}

// The function that provides the scene
void MyDevice::requestScene(const karabo::util::Hash& params) {
    Hash result("type", "deviceScene", "origin", this->getInstanceId());
    Hash& payload = result.bindReference<Hash>("payload");
    payload.set("success", false);

    const std::string& which = param.get<std::string>("name");
    if ("overview" == which) {
        payload.set("name", "overview");
```

(continues on next page)

(continued from previous page)

```
        payload.set("data", getOverview(this->getInstanceId()));
        payload.set("success", true);
    } else if ("controls" == which) {
        payload.set("name", "controls");
        payload.set("data", getControls(this->getInstanceId()));
        payload.set("success", true);
    } else {
        KARABO_LOG_ERROR << "Scene '" << which << "' was requested, but we don't have any
↪";
    }

    this->reply(result);
}
```

Note: There is the convention that the default scene (of your choice) should be first in the *availableScenes* list: this will be the one served when double-clicking, for instance.

3.4 Linking To Other Devices Scenes

The following applies whether you want to link to another of your scenes or to another device's scene.

Let's say that you want to add links in your *overview* scene to your *controls* scene.

Create the *overview* scene in Karabo, and add a link to *controls* by dragging *availableScenes* from the configuration editor, then changing the widget to *Device Scene Link*, then do *Configure Link*, and select the *controls* scene.

Export the scene to SVG, convert it to a C++ function as shown above.

If you want to link to another device, make the function accept another *remoteInstanceId* parameter, and point to that device:

```
std::string sceneWithLink(const std::string& instanceId, const std::string&␣
↪remoteInstanceId) {
    std::ostringstream ret;
    // SVG header
    ret << "<?xml version=\"1.0\"?>"
        << "<svg:svg xmlns:krb=\"http://karabo.eu/scene\" xmlns:svg=\"http://www.w3.org/
↪2000/svg\" "
        << "height=\"768\" width=\"1024\" krb:version=\"2\">"
    // Link to other device's controls scene
        << "<svg:rect height=\"30\" width=\"100\" x=\"19\" y=\"101\" krb:background=\
↪"transparent\" krb:class=\"SceneLink\" krb:font=\"\" krb:foreground=\"\"␣
↪krb:frameWidth=\"0\" "
        << "krb:keys=\"" << remoteInstanceId << ".availableScenes\" krb:target=\
↪"controls\" krb:target_window=\"dialog\" krb:text=\"CONTROLS\" krb:widget=\
↪"DeviceSceneLink\"/>"
    // SVG footer
        << "</svg:svg>";
    return ret.str();
}
```

3.5 Reference Implementations

DataGenerator: provides two scenes, overview and controls with links, as described here

Beckhoff: More complex possibilites are used here.

KEP21: definition of the scene protocol

CHAPTER

FOUR

SCHEMA INJECTION

A schema injection is a modification of a device schema, to bring further properties visible outside or to update attributes of existing properties (such as the maximum size of a vector). Any number and types of properties can be injected, whether parameters or slots.

In this example, we will define a slot that adds a boolean *injectedProperty*.

Begin by defining a slot that will call the *extend* function:

```
// Register the slot in the device schema
void MyDevice::expectedParameters(Schema& expected) {

    SLOT_ELEMENT(expected).key("extend")
            .displayedName("Extend")
            .description("Extend the schema and introduce a new boolean property")
            .commit();

}

// Register the slot in the constructor
MyDevice::MyDevice(const karabo::util::Hash& config) {
    KARABO_SLOT(extend);
}
```

There are two methods to injecting properties: *appendSchema* and *updateSchema*, both inherited from `karabo::core::Device`.

`appendSchema()` will add properties to the device, and can be called any number of time. `updateSchema()` will take the the original schema of the device, and add the new one to the device. However, it will discard what has been previously appended or updated!

Nonetheless, if `appendSchema()` is called after `updateSchema()`, then the parameters from both injections are kept.

If `updateSchema()` is called with an empty schema, then the device will be reset to its original schema, as defined in `MyDevice::expectedParameters()`.

This works, as internally, a device keeps three schemas: its *static* schema, as defined in *expectedParameters*, the *injected* schema, which is modified on injections, and the *full* schema, the combination of both which is exposed to the rest of the ecosystem.

```
void MyDevice::extend(void) {
    // Define a new Schema object
    Schema schema;

    // Then populate that new Schema
```

(continues on next page)

(continued from previous page)

```
    BOOL_ELEMENT(schema).key("injectedProperty")
            .displayedName("Hello")
            .description("A fresh property")
            .daqPolicy(DAQPolicy::OMIT)
            .readOnly().initialValue(true)
            .commit();

    // Finally, append the schema to our existing device
    this->appendSchema(schema);
    // Or
    // this->updateSchema(schema);
}
```

Once a device has been modified, either of these log message will be given:

```
INFO  MyDevice  : Schema appended

INFO  MyDevice  : Schema updated
```

Re-injecting a property, such as *injectedProperty*, will keep its current value if possible. However, with different types, an error will be raised if the value cannot be cast e.g. going from *UINT32_ELEMENT* to *INT16_ELEMENT* with a value out of bound.

4.1 Check Whether A Property Exists

If your device has an update loop, you can either use flags to check whether schema injection has already been done, or use `getFullSchema()`:

```
void MyDevice::update(void) {

    const Schema schema = this->getFullSchema();
    if (schema.has("injectedProperty")) {
        this->set("injectedProperty"), !this->get<bool>("injectedProperty"));
    }

}
```

4.2 Injected Properties and DAQ

Injected Properties and the DAQ need some ground rules in order to record these properties correctly.

In order for the DAQ to record injected properties, the DAQ needs to request the updated schema again, using the Run Controller's `applyConfiguration()` slot.

This can be prone to operator errors, and therefore it is recommended that only properties injected at instantiation to be recorded.

However, a common need for Schema updates is to specify the *maxSize* attribute fo vector or table elements, as the DAQ only supports fixed length arrays, of which the size has to be predefined.

For that, there is the special function `karabo::core::Device::appendSchemaMaxSize()` which, given a property path and the new length, will update the schema accordingly.

Such a vector:

```
void MyDevice::expectedParameters(Schema& expected) {
    NODE_ELEMENT(expected).key("node").commit();
    VECTOR_UINT32_ELEMENT(expected).key("node.vector")
        .displayedName("Vector")
        .readOnly()
        .maxSize(5)
        .commit();
}
```

Can be resized as follows:

```
this->appendSchemaMaxSize("node.vector", 50);
```

If several updates are made, then it is recommended to set *emitFlag* to false for all vectors apart of the last one. Only a single update will be then sent:

```
this->appendSchemaMaxSize("node.vector0", 50, false);
this->appendSchemaMaxSize("node.vector1", 50, false);
this->appendSchemaMaxSize("node.vector2", 50);
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search