

Supporting information

The Karabo Distributed Control System

S1 C++ data types supported by Karabo

Table S1 Plain C++ data types used by Karabo serialisation in case the stated lengths are supported by the given platform. (Other platforms, e.g. 32-bit Linux, are not supported.) Vectors thereof are also supported.

Boolean	<code>bool</code>
Integer	<code>char, signed char (int8), unsigned char, (unsigned) short (int16)</code> <code>(unsigned) int (int32), (unsigned) long long (int64)</code>
Float	<code>float (float32), double (float64)</code>
Complex	<code>complex<float>, complex<double></code>
String	<code>string</code>

S2 Source code examples for Karabo APIs

To complement the discussion in Sec. 2.7, Figs. S1 and S2 show a comparison of a `HelloWorld` device implemented in the different API flavors. In C++ and `python.bound` the `KARABO_CLASSINFO` macro/decorator assures that the distributed system is made aware of the `HelloWorld` device class. The statically available `expectedParameters` functions are used to describe the device properties as detailed in Section 2.2. A function is registered to be callable from the distributed system via `KARABO_SLOT` as a so-called `slot`, details of which are provided in Section 2.3. During initialization and operations the device is driven through different unified states (Section 2.4).

In contrast, the middle-layer API allows for a more condensed implementation, avoiding explicit registration of properties and slots to the distributed system. Instead, properties available to the distributed system are declared at class scope, and slots are indicated via the `Slot` decorator.

Despite the syntactic differences, the aforementioned APIs follow a common set of important design goals, which will be introduced in the following.

a) Header file helloworld.hh

```
1 #include "karabo/core/Device.hh"
2
3 class HelloWorld : public karabo::core::Device<> {
4
5     public:
6         KARABO_CLASSINFO("HelloWorld", "3.14")
7         HelloWorld(const karabo::util::Hash& config);
8         virtual ~HelloWorld();
9         static void expectedParameters(karabo::util::Schema& expected);
10
11    private:
12        void initialize();
13        void slotGreet();
14
15 }
```

b) main file helloworld.cc

```
1 #include "HelloWorld.hh"
2
3 using namespace karabo::core;
4 using namespace karabo::util;
5
6 KARABO_REGISTER_FOR_CONFIGURATION(BaseDevice, Device<>, HelloWorld)
7
8 void HelloWorld::expectedParameters(Schema& expected){
9
10     STRING_ELEMENT(expected).key("message")
11         .displayName("Message")
12         .readOnly()
13         .commit();
14
15     SLOT_ELEMENT(expected).key("slotGreet")
16         .displayName("Greet")
17         .allowedStates(State.ACTIVE)
18         .commit();
19 }
20
21 HelloWorld::HelloWorld(const Hash& config) : karabo::core::Device<>(config) {
22     KARABO_INITIAL_FUNCTION(initialize);
23     KARABO_SLOT(slotGreet);
24 }
25
26 HelloWorld::~HelloWorld() {
27 }
28
29 void HelloWorld::initialize() {
30     updateState(State::ACTIVE);
31 }
32
33 void HelloWorld::slotGreet() {
34     set("message", "HelloWorld!");
35     updateState(State::ACTIVE);
36 }
```

Figure S1 C++ API Implementation of Hello World device with a) header file (`helloworld.hh`, top) and b) main code (`helloworld.cc`, bottom). Compare with the Python implementations shown in Fig. S2.

a) Hello world implementation using 'bound python' interface

```
1 from karabo.bound import (PythonDevice, KARABO_CLASSINFO, SLOT_ELEMENT,
2                             State, STRING_ELEMENT)
3
4
5 @KARABO_CLASSINFO("HelloWorld", "3.14")
6 class HelloWorld(PythonDevice):
7
8     @staticmethod
9     def expectedParameters(expected):
10         (
11             STRING_ELEMENT(expected).key("message")
12                 .displayName("Message")
13                 .readOnly()
14                 .commit()
15             ,
16             SLOT_ELEMENT(expected).key("slotGreet")
17                 .displayName("Greet")
18                 .allowedStates(State.ACTIVE)
19                 .commit()
20         )
21
22     def __init__(self, configuration):
23         super(HelloWorld, self).__init__(configuration)
24         self.KARABO_SLOT(self.slotGreet)
25         self.registerInitialFunction(self.initialize)
26
27     def initialize(self):
28         self.updateState(State.ACTIVE)
29
30     def slotGreet(self):
31         self.set("message", "HelloWorld!")
32         self.updateState(State.ACTIVE)
```

b) Hello world implementation using middlelayer API

```
1 from asyncio import coroutine
2
3 from karabo.middlelayer import Device, Slot, State, String
4
5 class HelloWorld(Device):
6
7     message = String(displayName="Message")
8
9     @coroutine
10    def onInitialization(self):
11        self.state = State.ACTIVE
12
13    @Slot(displayName="Greet", allowedStates=[State.ACTIVE])
14    @coroutine
15    def greet(self):
16        self.message = "HelloWorld!"
```

Figure S2 Comparison of a Hello World device implemented in a) bound Python API (top) and b) Middlayer API (bottom). The bound Python API resembles the C++ API (see Fig. S1), whereas the middlelayer provides a – generally more convenient – and pythonic interface. Both implementations, and the C++ implementation from Fig. S1 will render in the GUI as shown in Fig. S3.

Configuration Editor

Parameter	Current value on device	Value
DeviceID	pythonServer/1_Helloworld_1	
State	ACTIVE	
Status		
Alarm condition	none	
Progress	0	
Message	Hello World!	
Greet		

Figure S3 Screenshot of configuration editor for the `HelloWorld` device discussed in Figs. S1 and S2.

S3 Example Karabo hash

```
1 from karabo.middlelayer import Hash, XMLWriter
2 h = Hash()
3 h["circle"] = "blue"
4 h["triangle.left"] = "red"
5 h.setAttribute("circle", "is_shape", True)
6 h.setAttribute("triangle", "is_round", False)
7 xml = XMLWriter()
8 print(xml.write(h))
9 """
10 <root KRB_Artificial=""> \
11   <circle KRB_Type="STRING" is_shape="KRB_BOOL:1" >blue</circle>
12   <triangle KRB_Type="HASH" is_round="KRB_BOOL:0" >
13     <left KRB_Type="STRING" >red</left>
14   </triangle>
15 </root>
16 """
```

Figure S4 A Hash with leaf attributes and its XML serialization.

S4 Example Scan Macro

```
1 from asyncio import CancelledError, coroutine
2
3 from numpy import linspace
4
5 from karabo.middlelayer import (
6     AccessMode, background, connectDevice, Float, Int32, Macro, Slot, State,
7     String, Unit, waitUntil, waitWhile)
8
9
10 class SimpleScanningExample(Macro):
11     startPos = Float(defaultValue=0.0, accessMode=AccessMode.RECONFIGURABLE)
12     stopPos = Float(defaultValue=10.0, accessMode=AccessMode.RECONFIGURABLE)
13     steps = Int32(defaultValue=11, accessMode=AccessMode.RECONFIGURABLE,
14                     unitSymbol=Unit.COUNT)
15     actorId = String(defaultValue="FXE_OGT1_BIU/MOTOR/SCREEN",
16                         accessMode=AccessMode.RECONFIGURABLE,
17                         displayName="Actor\u2014Id")
18     detectorId = String(defaultValue="FXE_OGT1_XAD/DET/SPECTRUM",
19                           accessMode=AccessMode.RECONFIGURABLE,
20                           displayName="Detector\u2014Id")
21
22     @Slot(displayName="Start\u2014Scan", allowedStates=[State.PASSIVE])
23     @coroutine
24     def execute(self):
25         self.state = State.ACTIVE
26         self.task = background(self.runner())
27
28     @Slot(displayName="Cancel\u2014Scan", allowedStates=[State.ACTIVE])
29     @coroutine
30     def cancel(self):
31         self.task.cancel()
32
33     @coroutine
34     def runner(self):
35         self.logger.info("Starting\u2014motor\u2014scan")
36         try:
37             actor = yield from connectDevice(self.actorId)
38             det = yield from connectDevice(self.detectorId)
39             for target in linspace(self.startPos.value, self.stopPos.value,
40                                   self.steps.value):
41                 actor.targetPosition = target
42                 yield from actor.move()
43                 yield from waitWhile(lambda: actor.state == State.MOVING)
44                 yield from det.start()
45                 yield from waitUntil(lambda: det.state != State.ACQUIRING)
46             except CancelledError:
47                 if actor.state == State.MOVING:
48                     yield from actor.stop()
49                 if det.state == State.ACQUIRING:
50                     yield from det.stop()
51         finally:
52             self.state = State.PASSIVE
```

Figure S5 A macro example showing a simple scan macro.