ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

THE KARABO MIDDLELAYER API AND MOTION CONTROL SYSTEMS AT EUROPEAN XFEL

D. Hickin*, N. Anakkappalla, V. Bondar, R. Costa, W. Ehsan, S. Esenov,
R. Fabbri, G. Flucke, A. García-Tabarés Valdivieso, G. Giovanetti
D. Goeries, S. Hauf, I. Karpics, A. Klimovskaia, A. Nasri, A. Parenti,
A. Samadli, H. Santos, M. Sharon, F. Sohn, M. Teichmann, J. L. Vazquez-Garcia,
European XFEL, Schenefeld, Germany

Abstract

Karabo is a device-based distributed control system toolkit used to implement the control and data acquisition systems of European XFEL. A feature of Karabo is its Middlelayer (MDL) API - a powerful, flexible and easy-to-use asynchronous Python API which can be used to implement Karabo devices. Such devices may interface hardware directly, or they may communicate with other Karabo devices to provide coordination between them or derive functionality from them. Lightweight middlelayer devices, called macros, allow routines for coordinating or monitoring Karabo devices to be quickly created. A command-line interface using the MDL API is provided by the iKarabo utility. In this contribution, an overview of the Karabo Middlelayer API is given, presenting key parts of the API, and macros and iKarabo are introduced. Examples of MDL devices are presented, with an emphasis on motion systems. A framework for multi-axis motion, Virtual Motor Base, and the scan tool, Karabacon, are described. The Karabo MDL API is compared to EPICS solutions, including pythonSoftIOC and Asyn, and to Bluesky.

INTRODUCTION

Karabo [1] is an open source SCADA (supervisory control and data acquisition) framework which can be used in building distributed control systems, particularly those targeted at the operation of large science experiments.

It was developed at the European X-ray Free Electron Laser facility (European XFEL), a free-electron laser (FEL) user facility providing soft and hard X-ray FEL radiation, currently to seven scientific instruments [2].

At European XFEL, Karabo is the main SCADA framework used in the implementation of the control system for the three photon tunnels (SASE1, SASE2, SASE3) and the seven instruments (see Fig. 1). For the accelerator control system, including control of the undulators, the main choice of framework is DOOCS [3], with some use of TINE [4] and EPICS [5] (DOOCS and EPICS are also used in some parts of the photon tunnels and instruments).

Karabo is device-based - monitoring and control of a SCADA system implemented using Karabo is achieved using a system of named distributed objects, called devices, which can communicate with each other.

A device may provide direct hardware control - for example it may represent a single piece of equipment, such as a motor, pump, valve, camera or digitizer or part of a piece of equipment, such as a digitizer channel, or be used to control a set of equipment. Other devices may provide software routines such as data analysis. Some devices may derive functionality from other devices or provide coordination between devices ("middlelayer"). Or, a device may provide an interface to a service. Some devices provide system services, such as devices for data logging, GUI server devices, or devices to interface project databases used for device configuration.

Communication between devices is routed through a message broker, and devices are grouped into broker topics. At European XFEL, separate topics exist for each tunnel and for each instrument, as well as three separate topics for lasers, one for the instruments of each of the three tunnels.

Each device has an identifier, unique within its topic, its *device ID*. Devices run within a software process called a *device server*. Multiple devices can run in a single device server. Each device is an instance of a *device class*. For devices providing equipment control, a device class may correspond to a particular hardware type, for example, a hardware model, or related hardware, such as a set of cameras with a common interface specification or controlled via the same vendor library. Device classes are provided via plugins called device packages. Typically, a device package provides one device class or a collection of related device classes.

Device Schemas

Devices have properties and slots. *Properties* are essentially key-value pairs that represent named values within a device. Examples include a position of a motor, the current of a power supply or the pressure of a pressure gauge. Each property has a key name and a value. The value of a property may be of any of the Karabo data types. These include scalar types (Boolean, signed and unsigned integer types, floating point types, strings), vector types such as vectors of scalar types, N-dimensional arrays and tables thereof. Properties may receive further specification using attributes (for example, maximum and minimum values for a numerical property or the properties units).

Properties may be configurable after device instantiation, read-only or configurable only at instantiation. Properties may have an access level defined. Users without sufficient

* david.hickin@xfel.eu

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

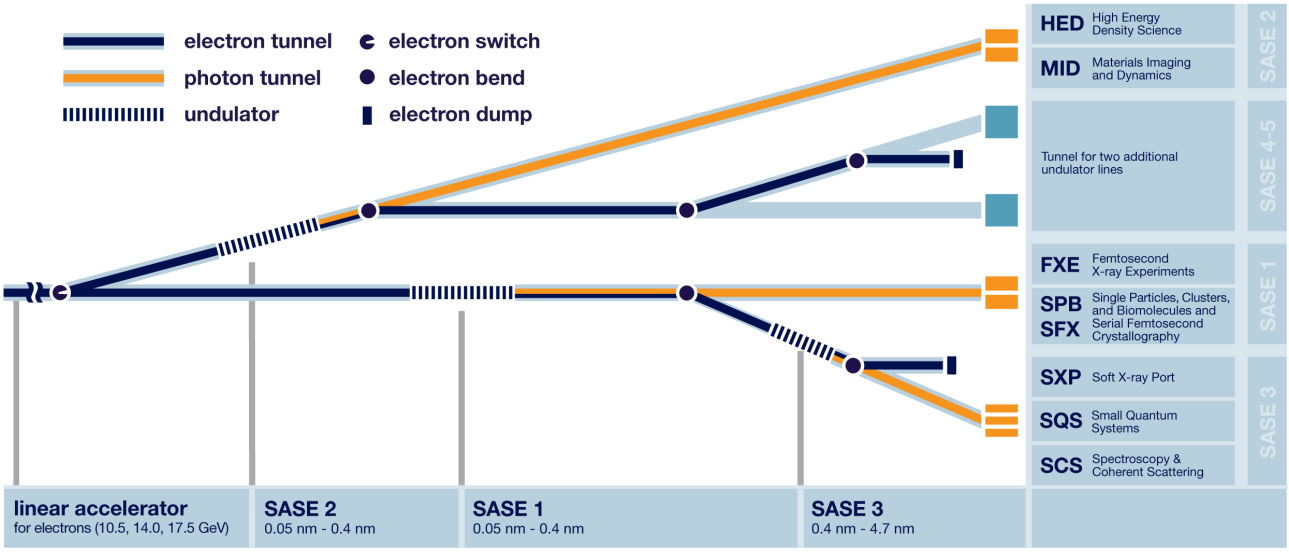

Figure 1: Layout of European XFEL tunnels and scientific instruments.

privileges can view, but not edit, property values. They can be hierarchically nested using so-called *node* elements. A node may contain any number of properties or sub-nodes.

Device *slots* can conceptually be seen as object methods and can be used to implement commands on the device. For example, a camera device may have slots to start or stop data acquisition. A motor device may have slots to move the motor to a target position or stop motion currently in progress.

Each device provides a self-description through a so-called device schema. This description includes its properties and slots (including their attributes).

Programming Interfaces

Karabo provides three application programming interfaces (APIs), which facilitate developers to implement Karabo devices. There is a C++ API, which is often used for high-performance applications or for integrating third-party C or C++ libraries. The Python Bound API provides a Python binding on top of the C++ API. It may be used for high-performance pipeline processing when an application additionally needs to leverage Python packages like NumPy [6] or SciPy [7]. The third API is a native Python API. Originally, this was used to implement middlelayer devices which provide coordination between other Karabo devices and so is called the Middlelayer API, now called Karathon. However, the API is also now used to implement devices that interface hardware. The Middlelayer API is powerful and easy to use and enables rapid device development. The remainder of this contribution will focus mainly on this API.

MIDDLELAYER API - KARATHON

The Middlelayer (MDL) API - also called Karathon - is one of Karabo's three APIs. It emphasises Pythonic language constructs and avoids domain-specific language extensions. It is implemented in pure Python - excluding secondary dependencies - and thus can be run on most architectures and operating systems that have common Python packages, such as NumPy, available. Karabo MDL device classes inherit from `karabo.middlelayer.Device`. The simplest way to add a property is by adding something like this to the class:

```
from karabo.middlelayer import Device

class MyDevice(Device):

    propertyName = PropertyType(
        attribute1=aValue,
        attribute2=anotherValue,
        ...)
```

where property types are, e.g., `String`, `Double`, `Int64`, `VectorBool`, and attributes are, e.g. `displayedName`, `description`, `defaultValue`, `allowedStates`, as well as `accessMode`, `unitSymbol`, and `metricPrefixSymbol`.

An alternative syntax using decorators is also possible:

```
    @PropertyType(
        attribute1=aValue,
        attribute2=anotherValue,
        ...)
    async def property(self, value):
        ... # do something with value
        self.property = value
```

This syntax allows to perform actions before a property update is published to the distributed system. Note that `self.property = value` does not immediately publish the value - rather this happens on the next `await` statement. Slot decorators are used to expose commands and methods to the distributed system:

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

```
@Slot(
   displayedName="Move",
   allowedStates={State.ON})
async def move(self):
   ...
```

will result in a GUI-visible command labelled *Move* that can only be executed when the device is in the `ON` state. Similarly,

```
@slot
async def moveTo(self, position: float):
   ...
```

would expose a function `moveTo`, taking an argument `position`, to the distributed system.

Device Proxies

Multiple distributed devices can be orchestrated in the Middlelayer API through so-called *proxies*. Here, each proxy is a Python object that represents a remote device, and depending on how it was instantiated, automatically follows that device. Generally, two methods exist, `getDevice` and `connectDevice`. The first creates a proxy with a snapshot of the remote device's state, the latter creates a proxy that is subscribed to updates of the remote. Both methods can be used as a context manager and then are equivalent, as they remain subscribed to updates throughout the context's lifetime:

```
async with getDevice(device_id) as d:
    d.position = 2.0
    await d.move()
    await waitUntil(
        lambda: d.state != State.MOVING)
    ...
```

will connect to a remote device identified through its device ID `device_id`, set the `position` property to 2.0, call the `move()` method on it, and then wait until it stops again. In this example, we also introduced the `waitUntil` method which will wait until a condition on a proxy is met. This convenience method is compounded by the `waitUntilNew(proxy.property, ...)` method, that waits for an update of a property or a list of properties on proxies. Using proxies, and the aforementioned convenience methods, intricate event-driven procedures can be implemented in a very concise, yet expressive way that avoids any domain-specific language extensions. This conciseness has made the Middlelayer API the preferred API for most Karabo device developments at European XFEL (see Fig. 2).

MACROS & SCRIPTING

For Karabo, it was chosen not to implement yet another macro language, but instead leverage the power and conciseness of Python. Karabo *macros* are Python scripts that can be created from within the Karabo GUI, and that are saved in a so-called project. They make use of the Middlelayer API in a similar fashion as described above, but by default allow for synchronous programming. Since they are intended as a scratchpad for rather simple ad-hoc scripting needs, macros do not support state definitions.

iKarabo is a Karabo command-line interface (CLI) running in the iPython prompt with similar features (and constraints) of macros.

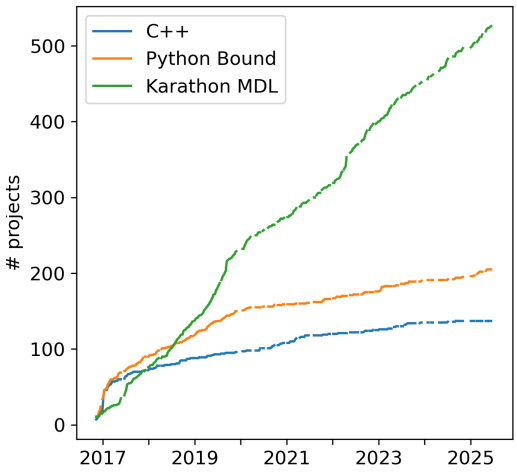

Figure 2: Time evolution of Karabo device projects and which APIs they use for implementation.

KARABO AND THE XFEL PLC SYSTEM

Most of the control hardware in the European XFEL facility is managed by PLC modules produced by Beckhoff GmbH and installed and configured by the Electronics and Electrical Engineering (EEE) group.

The PLC system consists of multiple modules connected via the EtherCAT network protocol and is connected in a redundant loop to an Industrial PC that implements all the communication and interlock logic in a common framework (see Fig. 3).

The PLC program contains multiple software elements, known as soft devices, used to organise the interactions of one or more terminals.

The direct Karabo interface to the PLC system is provided through the BeckhoffCom device class. For each PLC loop, a device server runs a BeckhoffCom device which connects to the PLC via TCP and manages message decoding and message dispatching to and from the PLC.

For each soft device in the PLC program, there is a corresponding Karabo device. These Karabo Beckhoff devices run in the same device server as the BeckhoffCom device. There are different soft device types and Karabo device classes for different hardware types. BeckhoffCom devices are capable of enumerating the soft devices on the PLC and instantiating the Karabo devices of the appropriate type.

Karabo Beckhoff devices interface the PLC through their associated BeckhoffCom device. The communication be-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

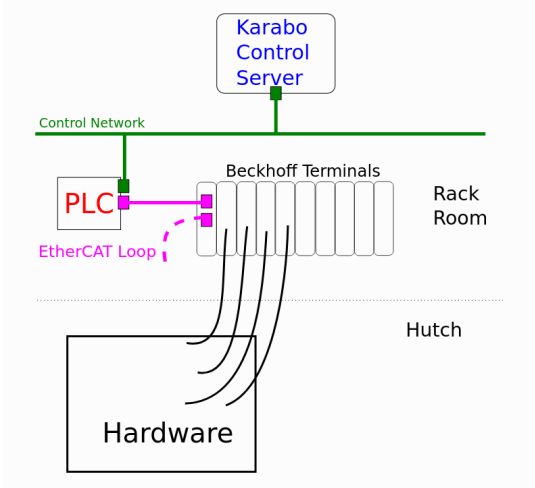

Figure 3: Karabo PLC architecture.

tween Beckhoff devices and their BeckhoffCom device was originally through the broker, but now an intra-process communication mechanism is used.

MOTION SYSTEMS AT EUROPEAN XFEL

Beckhoff Motor Devices

As with other categories of hardware, the majority of the motion system hardware at European XFEL is controlled at a low level by the Beckhoff-based PLC system and interfaced in Karabo through Beckhoff devices.

Most PLC-based motion control is implemented using the Beckhoff MC2 motion control library, with MC2 soft devices in the PLC controlling single motor axes and interfaced in Karabo by Beckhoff MC2 devices. Some stepper motors are controlled via *BeckhoffSimpleMotor* devices, although many devices previously controlled this way are now also controlled by Beckhoff MC2. Some hexapods are controlled by hexapod soft devices and a corresponding Karabo device.

If enabled in the PLC, motors can be configured to perform motion which is coordinated at the PLC level. One or more motors can be coupled to another motor in a specified ratio, so that the motion of the latter motor leads to coordinated motion of the other motors. For example:

- Optical slits: For two motors moving the blades of an optical slit, to move the centre position of the blades while keeping the gap size constant, the motors are coupled to move in the same direction. The coordinated motion produces a more stable gap size during the motion. Motions of motors coupled to move in the opposite direction change the gap size while preserving the centre position, with the coordinated motion producing a more stable centre position during the motion.
- Triplet systems: Suppose equipment is mounted on three motors which move in the Y-direction - one motor, YB, at the back and two motors, YFL and YFB, at the front, left and right, respectively, of the centre line, which lies along the Z-direction. By appropriate coupling of YB and YFL to YFR, the following motions can be accomplished:
  - Vertical motion (TY) - YB and YFL both coupled to YFR in the ratio 1:1
  - Yaw (RX) - YB and YFL coupled in the ratios -1:1 and 1:1 respectively
  - Roll (RZ) - YFL coupled in the ratio -1:1

Other Motor Devices

Some equipment is not managed by PLC modules and must be integrated into Karabo another way, such as through a vendor library or an ASCII command set over Ethernet, USB or serial communication, and the MDL API is often the one chosen for this.

Equipment can be integrated into Karabo using a vendor Python library, as is the case for SmarAct MCS2 module control system controllers and stages. For vendor-supplied shared libraries, functions can be called from MDL devices, for example, using Python ctypes.

The MDL API is the usual choice when controlling hardware through an ASCII command set. A number of motor controllers, such as those from Physik Instrumente (PI), have been integrated this way. A Karabo MDL package *scpiml* facilitates the creation of device classes for controlling such equipment. scpiml handles connection to the hardware and can be used to implement devices in a mostly declarative fashion, using the *alias* attribute of the device's properties and slots to associate with them the corresponding command, but more complicated behaviour can also be implemented.

Standard Motor Interfaces

Whether PLC-based or not, devices at European XFEL which represent a single motor axis usually present a standardised interface of properties and slots. This facilitates interoperability within Karabo clients, such as the scan tool. Other devices, even those not controlling motion system hardware, can implement the motor interface and work interoperably with these clients. Parts of the interface are mandatory:

- `interfaces` - a property with a `VectorString` value containing the element "Motor". Used to identify the device as a motor to clients.
- `actualPosition` - the readback of the current position of the motor axis.
- `targetPosition` - the target position to which the motor to move on a call of move.
- `move()` - starts the motor moving to the (absolute) position setpoint specified by targetPosition.
- `stop()` - stops any current motion.

Some other properties and slots are optional, but follow standardised naming and behaviour:

- `stepUp()`, `stepDown()`, `moveRelative()` - relative motion slots
- `isCWLimit`, `isCCWLimit` - report whether the motor is respectively at a positive or negative hardware limit (e.g. at a limit switch).

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

- `swLimitHigh`, `swLimitLow` - high and low values for software limits.

Devices can have other properties, not part of the standard interface.

A second standard interface (*multi-axis motor*) exists for devices which provide combined motion of multiple motors presented as a number of virtual axes, each with a single-axis-like interface. In this case, the interface's property contains the element "MultiAxisMotor".

VIRTUAL MOTOR BASE

Virtual Motor Base is a Python package for creating Karabo MDL devices for controlling systems of motors, where each motor is already controlled by a single-axis Karabo motor device. The resulting device presents a number of virtual axes as nodes, each with a motor-like interface. It allows devices to be created using minimal code, and the resulting MDL device provides a standardised (multi-axis motor) interface and behaviour, allowing interoperability with clients, including the scan tool (Karabacon).

Devices implemented using Virtual Motor Base inherit from a base class `VirtualMotorBase` (or a class derived from it, `CoordinatedMotionBase`, used for combined motions which are coordinated at the PLC level). One then specifies the motors, virtual axes and coordinate transformations between them, and selects which features are supported by the virtual axes, such as hardware limits, software limits or relative motion. For motion coordinated at the PLC level, one also specifies how the motors are to be coupled. The behaviour can further be customised by overriding the base class methods. Features include:

- Device-generated Karabo GUI scenes (operator panels) – can be overridden to provide custom scenes
- Units for virtual axes (e.g. mm, mrad, degrees)
- Support for parameter updates
- Unit test framework

MIRROR MOTION DEVICES

The offset and distribution mirrors in European XFEL photon beamlines are positioned using a system of 5 motors (see Fig. 4). However, the relationship between the movement of the motors and that of the mirror is not straightforward. Motion in the TY-direction requires roughly equal movements of the TY, RX and RY motors, not just TY, and due to the parallel kinematic structure, all axes are physically coupled.

A multi-axis motor MDL device (MirrorMotions), implemented using Virtual Motor Base, can be used to perform movements of the mirror in the TX, TY, RX, RY and RZ directions. The coordinate transformations are complicated, but, once specified, the rest of the implementation using the virtual motor base is straightforward. Basic coordination of the movements is produced by appropriate setting the of motor velocities. The device also generates custom device scenes (see Figs. 5 and 6), in addition to those provided by the base class.

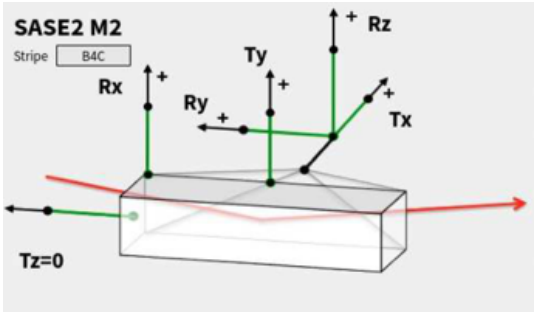

Figure 4: Motors controlling tunnel offset and distribution mirrors.

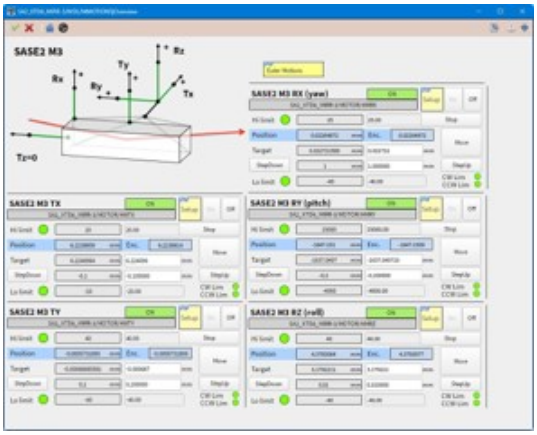

Figure 5: Device-generated overview scene.

KARABACON - THE KARABO SCAN TOOL

The Karabacon device is the European XFEL scan tool used to perform scan-related tasks for experiments requiring longer-duration data acquisition while synchronously moving motors.

In addition to moving motors, it can control triggers and the data acquisition system (DAQ), and provide a basic online preview. It provides a number of scans including absolute and relative step scans in 1 and 2 dimensions.

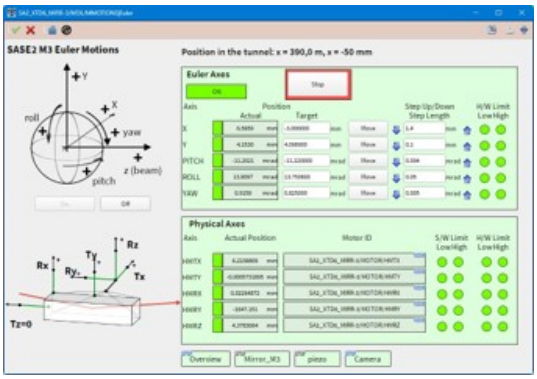

Figure 6: Device-generated scene for virtual axes.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

SASE 3 MOTION SYSTEM

SASE3 is a FEL beamline which provides soft X-rays to three scientific instruments - Spectroscopy and Coherent Scattering (SCS), Small Quantum Systems (SQS) and Soft X-ray Port (SXP).

Karabo PLC-based motor devices are used to position various hardware components of the X-ray optics and beam transport subsystem [8] within the SASE3 tunnel (see Fig. 7). A spontaneous radiation aperture (SRA), consisting of two horizontal and two vertical blades, cuts down spontaneous radiation background. Offset mirrors M1 (flat) and M2 (adaptive), in a chicane configuration, deflect the beam horizontally to separate the FEL beam from the spontaneous background and from bremsstrahlung. The M2 mirror bending system is used to optimise the mirror aperture and produce an intermediate horizontal focus behind the distribution mirrors. High-energy and low-energy pre-mirrors (M3) deflect the beam onto a variable line spacing (VLS) plane grating monochromator (PGM) with a low-resolution grating G1, high-resolution grating G2 and mirror M4 (for pink beam). Distribution mirrors M5 and M6 direct the beam to SCS and SXP respectively. There are also several 2-blade slit systems - vertical and horizontal slits and exits slits for each instrument.

In addition to the various PLC-based motor devices, there are a number of middlelayer devices:

Blade System Devices Devices to control the centre position and gap size of the various optical slits by moving the two blades. Optionally, the motion performed can be coordinated at the PLC level.

Slit System Device A device used to control the SRA. This is similar to the blade system devices, except that there are four motors and the device controls the centre position and gap size in both horizontal and vertical directions. (Motion coordinated at the PLC level is not currently supported.)

Monochromator MDL Devices A monochromator device calculates the grazing incident and exit angles of the X-ray beam from the angular position of the grating and derives the photon energy for a given diffraction order, pre-mirror angle and line spacing of the grating using the grating equation. Conversely, it moves the grating to achieve a specified photon energy or a zero-order position. Other devices provide configuration of the monochromator and scanning over an energy range.

Undulator Photon Energy Devices Karabo devices which interface the DOOCS middlelayer devices responsible for controlling the undulator energies.

Monochromator/Undulator Synchronisation A feedback loop device to keep the undulator photon energy close to the monochromator photon energy as the latter changes due to the angular position of the grating.

Monochromator/Vertical Slit Synchronisation A feedback loop device to sychronise the gap size of the vertical slit with the incident angle of X-rays on the monochromator grating to provide constant energy per unit area to prevent overheating.

Monochromator Zero-Order Setup Closes the tunnel pre-absorbers and opens the exit slits of the instrument receiving beam when moving the monochromator to its zero-order energy position.

Monochromator/Collimator Feedback A feedback loop device which modifies monochromator parameters based on a comparison of the angular position of the grating from its encoder and from an autocollimator device.

Mirror Switch Device The top-level MDL device, used to select a working point for the SASE 3 X-ray optics system and perform all relevant actions to achieve this.

The user selects the instrument, the mode of operation (high- or low-resolution grating, blank or unfocused), offset angle, high- or low-energy pre-mirror and whether the M2 bender is flat or provides an intermediate horizontal focus.

Moving to a new configuration stops any running feedbacks, configures the monochromator MDL device for the working point, moves offset and distribution mirrors, performs transverse motion of the grating and pre-mirrors, moves the monochromator grating to zero-order position, moves the vertical slit position and restarts the feedback loops.

COMPARISON WITH EPICS

Device servers in Karabo are roughly analogous to EPICS Input/Output Controller (IOC) applications, and the closest equivalent to Karabo device properties and slots are EPICS IOC records.

In EPICS there are various ways to accomplish behaviour implemented in Karabo through devices written in one of the three APIs.

EPICS records can be chained together through links (INLINK, OUTLINK and FWDLINK). Algebraic, relational, and logical operations can be performed on other record values using Calculation (calc) and Calculation Output (calcout) records. Further processing can be accomplished using Fanout and Sequence records. These solutions are not as powerful and flexible as the Karabo device APIs using well-known general-purpose programming languages. Another solution is the EPICS sequencer module, which provides a state notation compiler and run-time support for implementing state transition diagrams in an EPICS environment. However, this requires learning a domain-specific language and, again, may be less flexible compared to languages such as Python or C++.

StreamDevice EPICS device support [9] can be used for equipment that can be controlled through an ASCII command set, but only for the most basic cases. For more complex behaviour, asynDriver (also called Asyn) [10] can be used. This is a general purpose facility for interfacing device-specific code to low-level communication interfaces, such as asynOctet, which can be used to send ASCII commands.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

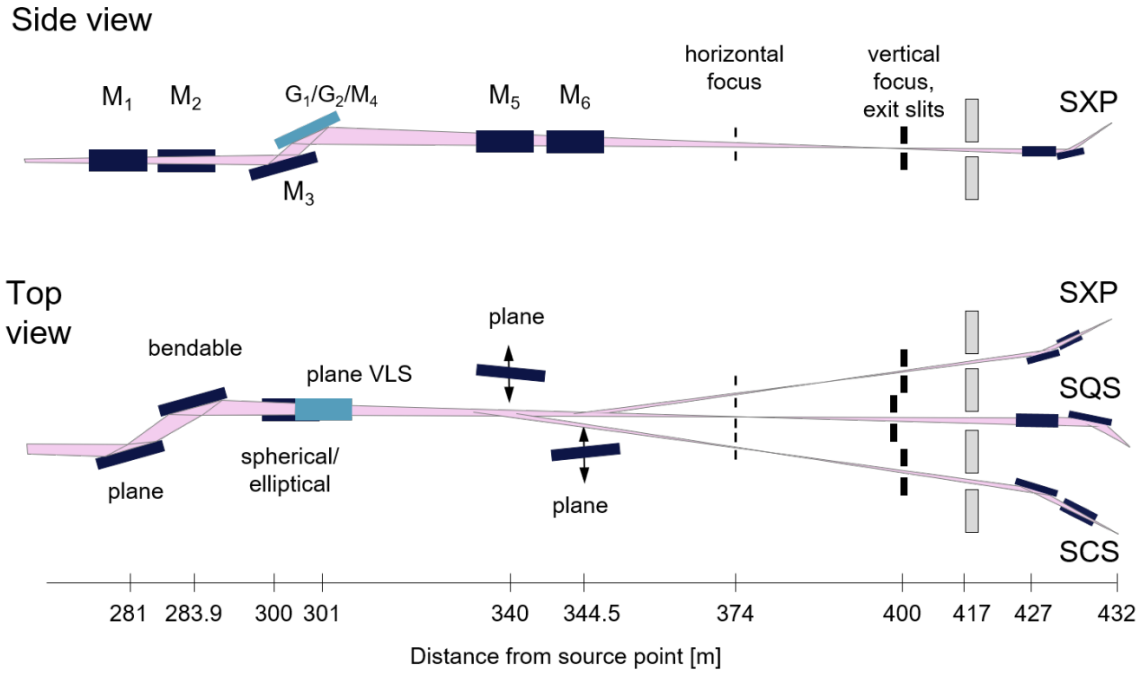

Figure 7: SASE3 X-ray optics and distribution system.

In asynDriver, code known as an *asyn port driver* communicates with a *port*, an entity which provides access to a hardware device. A C++ base class *asynPortDriver* simplifies the task of writing an asyn port driver. Writing an asynPortDriver is somewhat analogous to writing a Karabo device (in the C++ API).

pythonSoftIOC [11] developed by Diamond Light Source allows an EPICS IOC to be run from within the Python interpreter, and supports both cothread (cooperative threads), a threading library developed at Diamond, and asyncio for concurrency. Records can be programmatically created and arbitrary Python code run to update them and respond to put operations over the EPICS network protocols, Channel Access and pvAccess. It therefore can perform the sort of coordination tasks usually carried out by a Karabo MDL device and has a similar concurrency model.

COMPARISON WITH BLUESKY

Bluesky is a collection of Python libraries for experiment control and data acquisition [12]. Experimental procedures are encoded as *plans* and executed by an interpreter, the Bluesky *RunEngine*. It communicates with hardware through a high-level abstraction, the *Bluesky Hardware Interface*. The reference implementation of this is *Ophyd*, which provides integration with control systems that use EPICS, but multiple implementations exist and can be used within the same RunEngine.

In contrast, at European XFEL, experiment control and scientific data acquisition, including the type of high-level tasks addressed by Bluesky, are mostly implemented in Karabo. Simple routines can be easily programmed with macros. Behaviour, often complicated, can be implemented using a middlelayer device or orchestrated by several devices working together. Common scanning and data acquisition tasks can be carried out by the scan tool device, Karabacon. Standard interfaces allow interoperability of motors within Karabacon and other devices, and non-motors can also be used in scanning tasks by implementing a compatible interface, through a wrapper device or by adding support in the scan tool. The approach using a single framework simplifies learning and provides a powerful and flexible solution, combining high- and low-level tasks.

CONCLUSION & OUTLOOK

The Karabo MDL API provides a powerful and easy-to-use framework for creating devices both to integrate hardware and to coordinate the behaviour of other devices. The simplicity of the MDL API allows non-expert programmers to create simple routines through macros, and, in fact, some have implemented quite complex macros and devices. Although initially used for device coordination - a role at which it excels - it is increasingly the API of choice for device integration.

REFERENCES

[1] D. Göries *et al.*, "The Karabo SCADA System at the European XFEL", *Synchrotron Radiat. News*, vol. 36, no. 6, pp. 40–46, Nov. 2023. doi:10.1080/08940886.2023.2277650

[2] T. Tschentscher *et al.*, "Photon Beam Transport and Scientific Instruments at the European XFEL", *Appl. Sci.*, vol. 7, no. 6,

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBG004

p. 592, Jun. 2017. doi:10.3390/app7060592

[3] O. Hensler and K. Rehlich, "DOOCS: A distributed object oriented control system", in *Proc. XV workshop on charged particle accelerators*, Protvino, Russia, 1996. http://web.ihep.su/library/pubs/aconf96/ps/c96-200.pdf

[4] P. Bartkiewicz and P. Duval, "TINE as an accelerator control system at DESY", *Meas. Sci. Technol.*, vol. 18, no. 8, pp. 2379–2386, Jul. 2007. doi:10.1088/0957-0233/18/8/012

[5] L.R. Dalesio, A.J. Kozubal, and M.R. Kraimer, "EPICS Architecture", in *Proc. ICALEPCS'91*, Tsukuba, Japan, Nov. 1991, pp. 278–282. doi:10.18429/JACoW-ICALEPCS1991-S07IC03

[6] C. R. Harris *et al.*, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. doi:10.1038/s41586-020-2649-2

[7] P. Virtanen *et al.*, "SciPy 1.0: fundamental algorithms for scientific computing in Python", *Nat. Methods*, vol. 17, no. 3, pp. 261–272, Feb. 2020. doi:10.1038/s41592-019-0686-2

[8] X-Ray Optics and Beam Transport Technical Design Report, https://bib-pubdb1.desy.de/record/139077/files/TR-2012-006_TDR_WP73.pdf

[9] EPICS streamDevice documentation, https://paulscherrerinstitute.github.io/StreamDevice/.

[10] asyn/asynDriver documentation, https://epics-modules.github.io/asyn/.

[11] pythonSoftIOC documentation, https://diamondlightsource.github.io/pythonSoftIOC/master/.

[12] L. J. Koerner *et al.*, "A Python Instrument Control and Data Acquisition Suite for Reproducible Research", *IEEE Trans. Instrum. Meas.*, vol. 69, no. 4, pp. 1698–1707, Apr. 2020. doi:10.1109/tim.2019.2914711