Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor Thesis

Extension of the Python Bindings for the ChimeraTK DeviceAccess Library

presented by

Christian Willner

Student ID No. 7261415
B.Sc. Informatik

MIN Faculty, Department of Computer Science

submitted on 29.09.2022

First Reviewer: Dr. Andreas Mäder
Second Reviewer: Dr. Martin Killenberg

Task

ChimeraTK is a software toolkit that aims to facilitate the design of control applications for register-based hardware. One of the included libraries is DeviceAccess. It offers one common interface for a multitude of different devices and their respective protocols. It enables the user to focus on the task at hand while skipping many of the implementation details for the different hardware backends.

DeviceAccess is written in C++, a compiled language. For interactive workflows and quick configuration changes, an interpreted language like Python is often preferable. A Python binding of the DeviceAccess functions was available very early. They do not cover the features of the continued development of the C++ library and need to be updated. This extension and the associated motivation are the main topics of this thesis.

Abstract

This thesis covers the process of extending the ChimeraTK DeviceAccess library for Python users.

The ChimeraTK framework is used to work with register-based hardware devices. One of its lowest layer libraries is the DeviceAccess component. It offers a unified function interface to manipulate data on a multitude of connected equipment.

DeviceAccess is a C++ library. Working with C++ requires a compilation stage for every change in the routine of a derived program. Python is a scripted language. It is an ideal tool for automation and other work that benefits from its interactive nature. To facilitate the use of DeviceAccess in these cases, a binding was published to offer Python methods for the original C++ codebase. Over the past years, DeviceAccess was updated in functionality. The changes are yet missing from the Python bindings. The main addition is the introduction of accessor objects, that represent device registers. They enable blocking reads and thus ways of synchronization in the form of push-type accessors.

The coverage of the bindings increased dramatically with the added work from this thesis. That includes the addition of new accessor types and the coverage of more methods from the original library. The required work also allows refactoring of the codebase while keeping legacy programs functional. The complete code was formatted with type hints and function annotations. A small number of less frequently used elements of the DeviceAccess library are yet to be implemented. These can be added in future updates as the foundation is already planned out accordingly.

Contents

1 Introduction

The ChimeraTK DeviceAccess library's first bindings were offered for MATLAB and Python. In the early stages of the project, the two versions offered the same functionality as the C++ implementation.

The user had to allocate memory and include the location with the call to read and write data from and to devices. MATLAB and Python deviate from the C++ concepts of references. As an alternative, the bindings used an existing data structure as a function argument to be used for the operation. The resulting Python release relied heavily on copy mechanics.

While an increase in speed and a more Pythonic interface are desirable, they were not necessarily enough to justify a complete rework. Meanwhile, the C++ library evolved and came with a more concise idea to access devices. The accessor class integrates a user-side buffer, that is independent of the memory of a linked device. This decouples the work on the data from the status of the register and binds both to the same object. This design is quintessential for the implementation of blocked reads, where the device is in charge of the control flow on the host computer.

The introduction of register accessors thus made synchronizations possible. This is a crucial feature for the development of control applications and automation. Synchronization is thus the main motivation to extend the Python bindings for the ChimeraTK DeviceAccess library.

The general guideline in this work is to keep the Python workflow close to the original C++ approach. The respective languages sometimes use different principles to solve problems. Some of these mechanics hinder a straightforward implementation of the DeviceAccess bindings. This thesis aims to give reasons and examples for the resulting usage differences and the underlying design.

-

2 Fundamentals

2.1 ChimeraTK DeviceAccess

ChimeraTK is the abbreviation for **C**ontrol system and **H**ardware **I**nterface with **M**apped and **E**xtensible **R**egister-based device **A**bstraction **T**ool **K**it. The project is open source and published under the GNU Lesser General Public License v3.0 on GitHub [DESa]. Ubuntu 20.04 packages for the toolkit are available as well. ChimeraTK is maintained by the MSK[1] software group at DESY[2].

The DeviceAccess library is one of the fundamental layers of the toolkit. It functions as the abstraction layer to hardware on several backends[3] and provides one unified interface to the user [Var+17]. The library exposes the hardware via devices that are structured through register paths. DeviceAccess uses register accessor types for the user to read and write data. An accessor is a mutable object with a local user-side buffer and a reference to the memory of the assigned device register. The split between the two data locations is essential for complex control patterns like device-controlled blocking reads. As these accessors are fundamental to the extension of the bindings, they are discussed in depth in chapter 3.

2.2 Language Characteristics

The original ChimeraTK-DeviceAccess source code is written and accessed in modern C++. There are several options to call its functions in completely different language families. Important use cases are interactive automation tasks with Python. As C++ and Python have some distinct features, it is sensible to look at their characteristics.

C++

Designed in 1985, C++ quickly became one of the most used programming languages for resource-oriented, high-performance and system-level applications [Lut00]. The fine-grained memory management and high-speed execution makes it optimal for the intentions of the ChimeraTK framework.

C++ is a compiled language. Every change in the codebase needs to be compiled into machine code to take effect. Depending on the size of the involved codebase and the tied libraries, it can be a long process, until the newly compiled program is ready to be used. While this effort is often negligible in the life cycle of many applications, it interferes heavily with a workflow where changes happen frequently.

1. Maschine Strahlkontrollen (https://msk.desy.de/)
2. Deutsches Elektronen-Synchrotron (https://desy.de/)
3. PCI, XDMA, DOOCS, Modbus, OPC UA, EPICS, and several more.

Python

Python's use is very broad and has applications in a variety of academic fields [Staa]. It is supposed to be one of the easier programming languages to learn, as the high-level syntax is relatively easy to read and write [Die14].

In contrast to C++ Python is an interpreted language. The source code is translated into an intermediate representation and directly executed. This method cannot reach the execution speed of a program that is already available in a compiled form, but it has its purpose in interactive scenarios. The main use case for the bindings of this thesis are automation scripts, that can be easily modified for different scenarios.

2.3 Libraries of Interest

While many libraries are used for this project, two of them are crucial to the functionality of the bindings that are presented in this thesis. The first is Boost, which offers interoperability between C++ and Python. The second is NumPy, which has acclaimed widespread use to efficiently manage data structures like vectors and matrices.

Boost

Boost is a collection of C++ libraries that enable support for linear algebra, unit testing, and more. Many additions have since been incorporated into the C++ standard, like regex, optional or smart pointers [Stac] [Muk15].

The boost library also offers the "Boost Python Bindings" [Boo], which are essential for the DeviceAccess bindings of this thesis. Boost Python offers a compact method to build python importable libraries that link to C++ files. They are written in C++ and once compiled, they require no further steps to be accessed in interactive Python environments or scripts.

There are certain limitations in the functionality of the Boost library, like documentation, template handling, and inheritance. For this reason, the DeviceAccess bindings are layered to offer a more concise interface for the user. More information on the design can be found in section 4.1.

NumPy

NumPy is a widely-used Python library that offers support for arrays, matrices, and fast mathematical operations with these data structures. [Bre12].

The `ndarray` class is the pillar of the NumPy data representation. It is also the focal point for the connection of NumPy and the DeviceAccess library. To the user, it presents as an n-dimensional array, a structured collection of homogenously typed elements. Internally, it is a strided view of a memory location [WCV11]. In contrast to a normal Python list, the information has a high locality; it is supplied from a single memory block and not spread around in different chunks. Using vectorization and the avoidance of many copy operations, the library is often a magnitude faster than a pure Python implementation [Oli07].

The `ndarray` class is already used as input variables in the legacy version of the bindings. This thesis leverages the properties of the `ndarray` even further, by designing the new accessors as a subclass object. The similarities of the C++ register accessors and the `ndarray` make the inheritance mechanism an ideal choice for the implementation of a Python accessor class.

3 Register Accessors

Without any knowledge of the underlying system, the most basic approach to access data from the real world would probably be the following: A user knows the location of the sensor they would like to access and can then ask the sensor to deliver its data and any additional meta information it might have.

This process is very similar to the design of the Register Accessors. With the address of the device[1], the user can open registers to access the desired information. Registers can return the data in different form factors: a matrix, an array, a scalar, or some kind of trigger. For time-sensitive processes, it is significant if the information is polled from the device or pushed by it.

It is important to point out that there are two storage spaces for the device data. One location is remote, supplied by the device, and the other one is local: the user buffer. As the name implies, this buffer keeps the data with the user until it is ready to be written to the device. When data is read from a device, the new information is copied to this buffer. Due to the decoupling, the user can manipulate the buffer in multiple steps and trigger a `write` procedure when finished.

It might be okay to just access the raw data on the device, a conversion to a desired precision or size is often quite valuable to the user's workflow. To avoid mistakes in type conversions, the register accessors allow being set to a certain user type, which can be selected independently from the actual data type on the device.

1. As outlined in section 4.2

Feature	Legacy Bindings	New Bindings
User Types:		
np.int32	☐	✓
np.int16	☐	✓
np.int8	☐	✓
np.uint8	☐	✓
np.int16	☐	✓
np.uint16	☐	✓
np.int32	☐	✓
np.uint32	☐	✓
np.int64	☐	✓
np.uint64	☐	✓
np.float32	☐	✓
np.float64	✓	✓
np.double[a]	✓	✓
np.float[a]	✓	✓
np.str_	☐	☐[p]
np.bool	☐	☐[p]
np.void	☐	☐[p]

[a] Alias of np.float64.
[p] Planned for the future.

Table 3.1: User Type Overview and Comparison

3.1 Accessor Types

Several register accessor types are available to the user. The main accessor group is concerned with the actual information on the device. There are three types with different dimensionalities.

Most of the time the user wants to know the numerical value of an event, while sometimes the existence of new data is more important than whatever value it might have. For this occasion, a specialized void accessor type treats the signal as a pure trigger.

User Types

When the user wants to get a register accessor from the device, they have to state the desired register path and the user type. This is a convenience function for the user. It supplies the automatic conversion and correct rounding from the implemented format on the device to the requested user type. Examples of these are (un)signed integers or floating point numbers in different widths[2]. A complete overview of all possible user types is given in Table 3.1. The table also shows the current coverage compared to the legacy bindings.

2. Sometimes it is useful to set or read actual raw data. There is no specific raw type. For this use case, it is possible to pass the `raw` access modifier flag.

Scalar, 1D & 2D

To gather information from the device, the registers are available in three dimensionalities: scalar, one– or two-dimensional. For the implementation, there are some key differences between the three, but for the user, it mostly depends on the desired format. If they use a 2D accessor to access one-dimensional registers, they still receive a matrix type. The same is true for a 1D register accessor that is tied to a scalar value. The interface will return a vector with a single entry. The different accessors have some additional functions to inform the user about the number of rows and columns, where available.

Void

Void accessors are special, as they are intended to trigger events. The actual intensity of the signal is not of interest. Therefore, they do not have a user type, as there is nothing to convert from. In the standard form, their concept is only sensible for sending triggers to the device. A request to read no data, or rather a wish to receive nothing, is pointless. The concept of waiting for a trigger is covered by push and poll types.

Push and Poll Types

Sometimes it is desirable or even crucial to synchronize control flows between different parties. One direction is triggering events on the device from the host by writing to certain registers. The other direction is changing the behavior of the host when information on the device has changed.

There are two ways to achieve this: polling and pushing. Constant polling of the device to see if the data has changed can block the device from other tasks. Also, it might not be frequent enough to catch the trigger on time. The alternative is pushing. The control flow is transferred to the device. The accessor has to wait until the linked register has updated information. To create such an accessor it has to be requested with the `wait_for_new_data` access modifier flag from the device. A complete example with source code can be found in Listing 7.4.

3.2 Implementation

A premise for the Python bindings is to supply a library, which usage is close to the C++ version, but also as Pythonic as possible. To retain consistency over the project, all names are taken from the original library, including the naming style [3].

Keeping the workflow consistent over the two languages avoids reinventing the wheel and leverages the well-established concept of the base project. Both languages have their differences that are not always feasible. First, some C++ mechanics are not available in Python. Secondly, typical Pythonic methodology might have different expectations for the behavior of data structures.

3. While the PEP8 guidelines suggest lower case with underscores, it also recommends to keep the prevailing style (https://peps.python.org/pep-0008#function-and-variable-names).

```cpp
// There is an open device called 'toaster', which has an internal
    temperature setting that should be changed
ChimeraTK::ScalarRegisterAccessor<float> temperatureSetPoint =
    toaster.getScalarRegisterAccessor<float>("TEMP_SET_POINT");

// Treat the accessor as if it was a regular float variable.
temperatureSetPoint = 250.9;

//After the manipulation, data can be written to the device.
temperatureSetPoint.write();
```

Listing 3.1: Direct Assignment to the User Buffer in C++

NumPy implementation

The original C++ interface provides vectors of a certain numerical type as a user buffer. A Python equivalent would be a list or a general sequence. A vector of vectors translates well into a list of lists to handle 2D accessors.

The standard Python data structures are often replaced by their NumPY equivalent for their added functionality and performance. In support of this workflow, the legacy bindings were already able to work with NumPy arrays as method parameters. To facilitate the usage even more the new register accessors are implemented as a NumPy `ndarray` subclass. The added inheritance makes the Python accessors a fusion of all methods from the C++ register accessors and the NumPy array. This way there is no further conversion or copying before the data from the devices can be fed into any existing NumPy workflows and vice-versa.

Data Access and Manipulation

C++ offers a lot of freedom as far as redefining core elements like assignment operators. In the original library, this is used as a comfortable shortcut to edit the user buffer, as seen in Listing 3.1[4]. The implementation also enables the user to use accessors directly in calculations like vectors, or vectors of vectors, without the need to point to the user data that is only a part of the actual class.

The inheritance from the NumPy `ndarray` can mimic this behavior to a certain degree. Unfortunately, a direct assignment cannot be overwritten in Python. It is thus not possible to copy the original behavior. The = operator would assign the array or scalar to the variable instead of the user buffer of the accessor. Even for mutable types such as `list` and `ndarray`, Python cannot be modified. An example can be seen in Listing 3.2. The variable `array` is first assigned to a NumPy array type and then to a standard list type. To keep the array from being garbage collected, it was also assigned to `alias`, to show the content of the `ndarray`. It can be seen that `array` and `lst` variables are referencing the same object after the new assignment, but the original `ndarray` remains unchanged. Therefore = cannot be used to change the user buffer. A Python alternative to the direct assignment is provided in form of the `set` method, as shown in Listing 3.3, line 20.

4. Modified code from [DESb].

```
>> array = np.array([1, 2, 3])
>> alias = array
>> print(alias)
[1 2 3]
>> lst = [4, 5, 6]
>> print(type(array))
<class 'numpy.ndarray'>
>> array = lst
>> print(type(array))
<class 'list'>
>> array[2] = 99
>> print(array)
[4, 5, 99]
>> print(lst)
[4, 5, 99]
>> print(alias)
[1 2 3]
```

Listing 3.2: Direct Assignment to a Mutable Type in Python

```
>> # 'thickness' is a 1D accessor of user type int16.
>> #  It contains 4 measurement points for the inserted bread
>> #  of an advanced toaster.
>> print(type(thickness))
<class 'deviceaccess.OneDRegisterAccessor'>
>> thickness
OneDRegisterAccessor([0, 0, 0, 0], dtype=int16)
>> # It can be read from, like an accessor:
>> thickness.read()
>> thickness
OneDRegisterAccessor([10, 11, 12, 8], dtype=int16)
>> # It also has all methods of an ndarray:
>> print(thickness.min())
8
>> # 'heater' is a scalar accessor of user type int8. It controls the
   temperature of the toaster.
>> heater.read()
>> print(heater)
[200]
>> # Accessors can be used like a standard NumPy ndarray. For example
   by using the smallest element from the thickness accessor to set a
   new value of the heater
>> heater.set(200 + thickness.min()*5)
>> print(heater)
[280]
>> heater.write()
```

Listing 3.3: Behavior of the Python Register Accessor

1D and 2D arrays inherit from the respectively sized NumPy `ndarray` and thus work like vectors and matrices. The scalar version implementation is inconsistent and does not map to the respective NumPy scalar type. As seen at the end of Listing 3.3, the scalar register accessors representation is not scalar. It is implemented as a 1D `ndarray` with a single element. Although real scalar types exist in NumPy, they are immutable [Num]. As a superclass of a scalar register accessor, they are thus unusable.

4 Design

4.1 Layer Model

Figure 4.1 presents the internal hierarchy of a fictional use case. The diagram shows a highly connected household with different hardware devices that are connected through various backends and protocols.

Together with the C++ DeviceAccess library, the user can take the Python bindings package to control the appliances via a Python script. For the correct name mapping to the devices, the user also needs configuration files that would be handled by the group that integrated the hardware into the network (see the section 4.2).

The Python bindings package itself is structured into two parts. There is an interface, that is exposed to the user. It is dependent on a second library, that is directly accessing the original C++ implementation. This interconnecting library is therefore often referenced as the middle layer. It serves as a routing point to offer all C++ functions to the interface layer. For legacy purposes the old `mtca4u.py` interface is still included in the bindings package. It uses the same middle layer.

The extra interface layer on top of the Boost bindings has three main reasons. Firstly, the interface serves as a unifying abstraction layer on top of the middle layer. Due to the Boost library, the interface would otherwise be very different from the C++ workflow[1]. Secondly, writing the interface in Python offers better ways for the implementation of the documentation and type annotation than the C++ based Boost Python library. The last reason is the inheritance relationship to NumPy's `ndarray`, which cannot be represented within the Boost Python system.

The two-layer structure thereby acts as a separation between the necessary structure of the Boost part and the actual interface that is streamlined for the user experience. The interface library `deviceAccess.py` is supposed to act to the user as if they were accessing the C++ interface from the original toolkit.

4.2 Hardware Access

The user can interact with the hardware in the form of a configuration file, that links the logical register paths to their actual backend addressing structure. This is a very convenient form of abstraction. The actual sensors, triggers, and actuators might be in different locations, connected to a multitude of backends and servers. For the logical division of the control system, it is easier to group and arrange them in a hierarchical structure.

1. The C++ implementation is using templates, which transform into a multitude of classes and methods after the Boost binding process. One for each user type. The goal of the bindings is to mimic the condensed behavior of the original bindings. This is achieved by adding the interface layer. See chapter 5 for more insight on the boost integration.

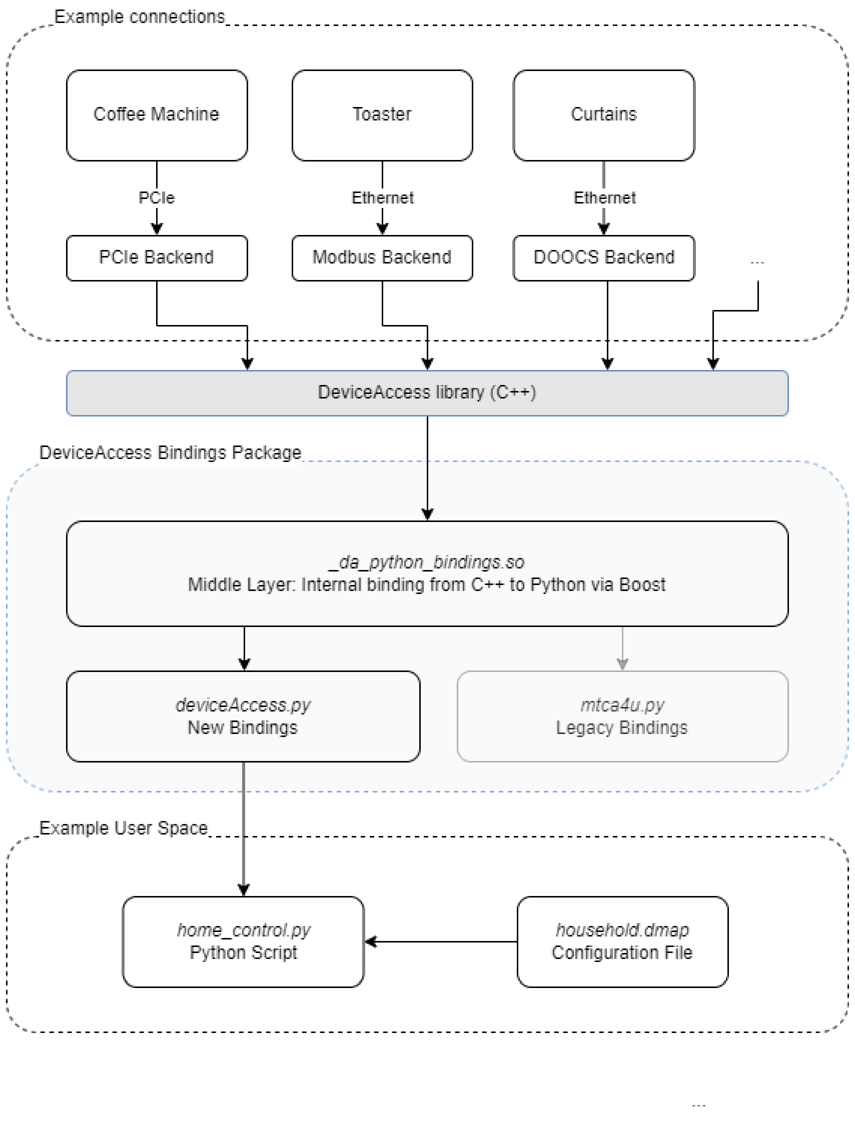


Figure 4.1: Modell of the Python Bindings with Use Case

The example in Figure 4.1 covers a fictional household with a coffee machine, a smart toaster, and electrical curtains, with connections through PCIe, Modbus, and DOOCS[2]. The user does not need to know the details of the attached hardware. It can be structured in a hierarchical, easy-to-use structure. Instead of looking up the corresponding PCIe banks that are linked to the water level sensor of the coffee maker, a register accessor can be obtained through an easy-to-remember path like `kitchen/coffeemaker/waterlevel`. The topology and grouping of the aliases are completely independent of their respective hardware integration.

2. Distributed Object-Oriented Control System Framework (https://doocs-web.desy.de/)

5 Optimization and Refactoring

5.1 Optimization

NumPy and Boost Update

In an early iteration of the bindings, a Boost Python numeric class was used as a container for the register's user buffer. This was changed to support NumPy's `ndarray`. The actual array was not a class field. It was passed as a variable. The conversion to and from the C++ accessor's user buffer was achieved through a chain of copying and casting operations.

In the meantime, the Boost library has gained official support for the `ndarray` data type. Boost intended this to directly use the Python container and its methods in C++. The C++ code is already existing in form of the DeviceAccess library. The Boost implementation of the `ndarray` nevertheless proves helpful. It allows skipping some of the previous workarounds to integrate the NumPy class into the Boost workflow. The resulting code is easier to understand and maintain.

The legacy bindings are writing and reading data from a separate NumPy object, that is supplied during the function call. The new Python accessor is also a `ndarray`. It does not need an external object to keep the user buffer. The updated bindings can consequently avoid the additional object in the signature of the method. This usage is closer to the C++ version.

Zero-Copy

A big performance improvement can be obtained by finding and eliminating needless operations. The goal of zero-copy techniques is to avoid duplicates of data. When the information is already available to the system, it might be faster to access this region directly instead of copying, working with the duplicate, and writing the changes back to the original location.

The current structure of the bindings has accessors, that inherit from NumPy's `ndarray` to represent the user buffer. This buffer is now present at two different locations. One is in the form of the C++ accessor implementation that is called from the middle layer's boost bindings. Another is given in the form of the NumPy Array. Every read and write calls copy operations between these two memory regions. It would save bandwidth and CPU time to skip the duplication and act directly on the data of the original C++ accessor's user buffer.

One of the requirements is a similar form of data-structure for the two regions. Just because the information is the same, the arrangement might differ significantly. A transformation operation might be costlier than copying each element.

The `ndarray` and the original accessors are compatible in that regard, as they both built on a high locality of their data. There is a function in NumPy, that work with C++ based memory pointers. Some copy operations might become avoidable with it.

Unfortunately, the respective function did not work with all tests, that were conducted during this thesis[1]. The respective parts are still included in the source code, to facilitate a move in the future. The main usage scenarios of the bindings are not focused on speed or resources. The extra copy step is currently not hindering the functionality of the bindings.

Generally, a zero-copy implementation can cover the scalar and 1D accessor. The target buffer has to be accessible as a coherent block. The current version of the C++ 2D accessor does not guarantee this requirement and would consequently not benefit from a zero-copy functionality.

5.2 Refactoring

The division into a user-facing interface and the compiled middle layer has been mentioned several times. One reason is the easier integration of documentation features, another point is the abstraction from implementation details.

The bridge between C++ and Python is realized via Boost's Python bindings in a compiled `_da_python_bindings.so`[2] file. It covers all functions, classes, and methods that are available in the original library.

The feature extension in this version is mainly focused on accessors. There are four accessor varieties in the toolkit: void, scalar, 1D, and 2D. C++ offers templates as generics to increas code reusability. Scalar, 1D, and 2D can be used with each of the 15 user types (see Table 3.1). The way templates are routed in the Boost Python library, the resulting middle layer has 46 accessor classes[3].

The user does not need to navigate these, as they use the reduced interface from the `DeviceAccess.py` file. The handling of the different accessor classes is abstracted into a single parameter of a unified class. When the user requests an accessor from the device, they get a user-type independent class. Internally this class accesses the correct corresponding accessor objects from the middle layer.

While some register accessor dimensions need unique methods, they also cover all 19 base functions like `write`, `read` or `getDescription` (see Table 5.1[4]). To bundle these, scalar, 1D, and 2D accessors inherit from a GeneralRegisterAccessor class[5]. The interaction between the middle layer, the user-facing interface and the C++ DeviceAccess library is illustrated in Figure 5.1. It can be seen how the templated C++ classes are split up into their respective Boost Python equivalent in `_da_python_binding`, followed by a re-unification at the interface level.

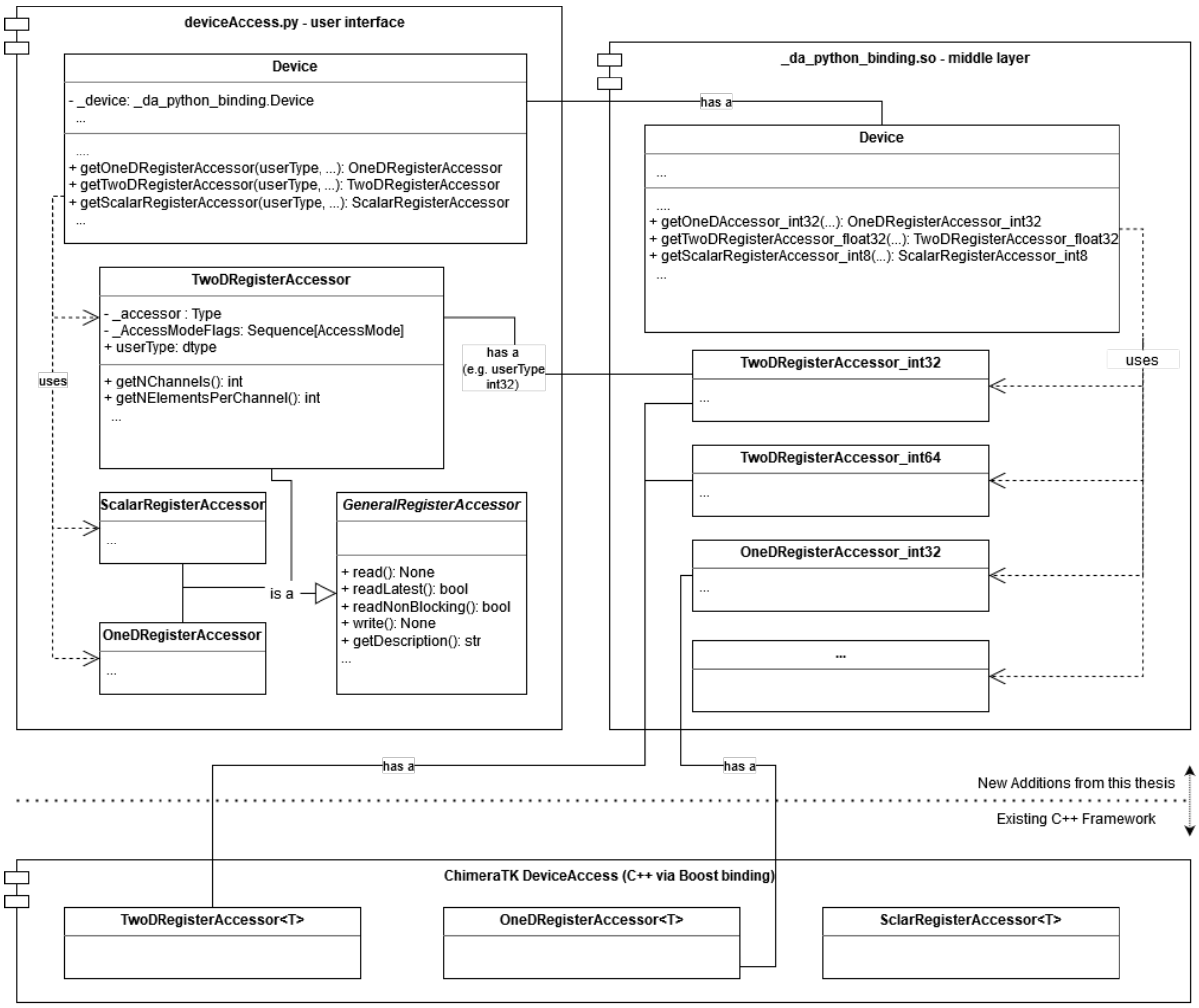
The middle layer is limited and cannot easily use inheritance. It has to explicitly include a binding from each of the accessor classes' methods to every C++ counterpart. The resulting source code of the `_da_python_bindings.so` has a lot of repetition. There are more than 870 binding commands for the accessor class alone, which only differ by calling different user type

1. Some bug reports suggest [Stab], that the reason might be connected to the Boost library. It is difficult to verify this, as the library has very limited documentation on its NumPy integration. A future version of Boost might solve this issue.
2. In Python, the underscore at the beginning of a name hints at an internal variable; or in this case a library.
3. 15 user types multiplied by three dimensional accessors, plus one void accessor class.
4. Some methods and classes that were not discussed in this thesis are included for the sake of completion.
5. The non-void accessors also inherit from NumPy's `ndarray`.

Feature	Legacy Bindings	New Bindings
Device Functions:		
open	✓*	✓
close	☐	✓
getOneDAccessor	✓*	✓
getTwoDAccessor	✓*	✓
getScalarAccessor	☐	✓
getVoidAccessor	☐	✓
activateAsyncRead	☐	✓
Accessor Functions:		
read	✓	✓
readLatest	☐	✓
readNonBlocking	☐	✓
read_raw	✓	☐ *
write	✓	✓
writeDestructively	☐	✓
write_raw	✓	☐ *
getName	☐	✓
getUnit	☐	✓
getValueType	☐	✓
getDescription	☐	✓
getVersionNumber	☐	✓
isReadOnly	☐	✓
isReadable	☐	✓
isWriteable	☐	✓
isInitialised	☐	✓
setDataValidity	☐	✓
dataValidity	☐	✓
getId	☐	✓
Other Classes:		
AccessMode	☐	✓
VersionNumber	☐	✓**
TransferElementID	☐	✓**
DataValidity	☐	✓**

✓* Indirect implementation, workflow differs from current C++ version.
✓** Working, but some seldom used class methods not yet covered.
☐ * Deprecated in current DeviceAccess version.

Table 5.1: Feature Coverage Comparison



deviceAccess.py - user interface
Device
- _device: _da_python_binding.Device
+ getOneDRegisterAccessor(userType, ...): OneDRegisterAccessor
+ getTwoDRegisterAccessor(userType, ...): TwoDRegisterAccessor
+ getScalarRegisterAccessor(userType, ...): ScalarRegisterAccessor
TwoDRegisterAccessor
- _accessor : Type
- _AccessModeFlags: Sequence[AccessMode]
+ userType: dtype
+ getNChannels(): int
+ getNElementsPerChannel(): int
uses
ScalarRegisterAccessor
OneDRegisterAccessor
is a
GeneralRegisterAccessor
+ read(): None
+ readLatest(): bool
+ readNonBlocking(): bool
+ write(): None
+ getDescription(): str
has a
_da_python_binding.so - middle layer
Device
+ getOneDAccessor_int32(...): OneDRegisterAccessor_int32
+ getTwoDRegisterAccessor_float32(...): TwoDRegisterAccessor_float32
+ getScalarRegisterAccessor_int8(...): ScalarRegisterAccessor_int8
has a (e.g. userType int32)
TwoDRegisterAccessor_int32
TwoDRegisterAccessor_int64
OneDRegisterAccessor_int32
uses
has a
has a
New Additions from this thesis
Existing C++ Framework
ChimeraTK DeviceAccess (C++ via Boost binding)
TwoDRegisterAccessor<T>
OneDRegisterAccessor<T>
SclarRegisterAccessor<T>

Figure 5.1: Partition of the Accessor Class Relationship Diagram

implementations. To avoid errors and to facilitate maintenance, each base method should only be stated once and then be automatically filled into its respective place and form.

The new middle layer is distributed as a compiled object file. This offers a solution for the automated population of classes via nested preprocessor macros. Before the actual compilation, these macros are extended as textual substitutions. Each macro argument in the text is replaced by its text counterpart. Macros tend to produce unintended results and have to be used with great caution. Many development environments have difficulties checking for correct syntax in the resulting code, as it might differ from the actual outcome of the preprocessor. This is especially true for nested macros and the combination with templates. This area of meta-programming is also supported by the Boost library. Additional work in this direction might optimize the current implementation even further.

Legacy Support

As many automation scripts have already been written with the existing bindings, it is important to keep the so-called `mtca4u` library in the package. As the middle layer has been reworked during this thesis, some of the implementations had to be adjusted[6].

To ensure that the library works as intended, there are already some tests in place that cover the basic functions. These tests are now even more important. It is critical, that the legacy bindings keep the same behavior while being adapted to fit the improved middle layer.

Due to the way these tests have been developed with the library, some of the tests were targeting the final bindings and some were about the middle layer functionality. To keep the tests on the same hierarchy level, the latter procedures were moved to target the interface that is exposed to the user instead.

6. The previous package came with a middle layer in the form of a compiled shared object file called `mtca4udeviceaccess.so`, which has been replaced by the new `_da_python_bindings.so`.

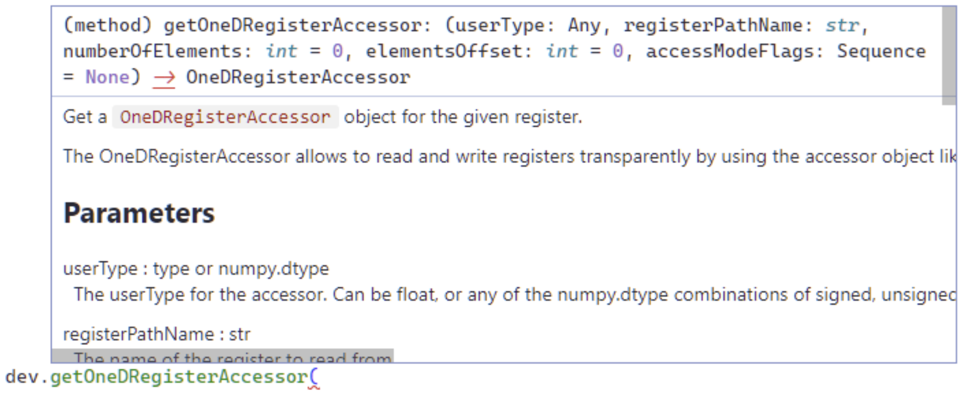
6 Documentation Generation

Through the development of the core source code, the maintainers also try to streamline the build, delivery, testing, and documentation. These different steps are selectable results of one common configuration script.

The CMake configuration allows for the generation of a complete HTML documentation with a call of `make doc`. The Script is configured to run Sphinx, which will then proceed to generate the necessary files for a documentation in form of a website [SPH].

A philosophy to ease this work step is "docs as code" [Wri]. The key principle is that the documentation is written in the same space as the connected code. This has the advantage of using the same tools for both workflows and the early inclusion of developers to comment on their code. It is even possible to load the included documentation examples from files. These files can then be checked for correctness each time the documentation is generated. Currently, testing is an extra step in the pipeline and is not linked with the HTML file generation. Consequently, it is not guaranteed that every example in the documentation is representing functioning code.

Many development environments support docstrings. These are strings that will give additional information to the coder when using the corresponding functions or objects. It is possible to show this information with the Python `help` command, but also as popups while typing, as shown in Figure 6.1.



```
(method) getOneDRegisterAccessor: (userType: Any, registerPathName: str,
numberOfElements: int = 0, elementsOffset: int = 0, accessModeFlags: Sequence
= None) → OneDRegisterAccessor
```

Get a `OneDRegisterAccessor` object for the given register.

The OneDRegisterAccessor allows to read and write registers transparently by using the accessor object lik

Parameters

userType : type or numpy.dtype
  The userType for the accessor. Can be float, or any of the numpy.dtype combinations of signed, unsigne

registerPathName : str
  The name of the register to read from

```
dev.getOneDRegisterAccessor(
```

Figure 6.1: Example of Popup Text from Python Docstring

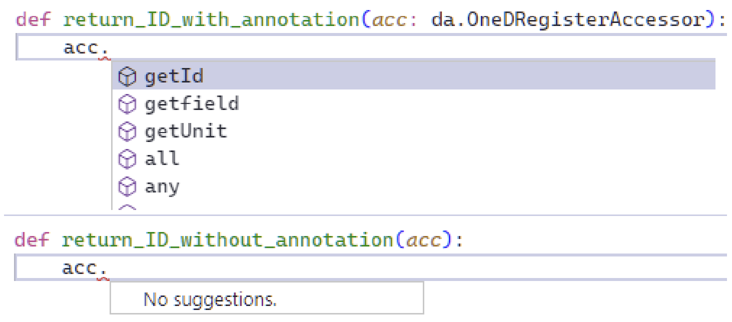
Most IDEs support reStructuredText[1], which is also parsed by Spinx. It can be used to inform about function descriptions and parameters, their expected types, usage examples, hints, warnings, and many other entries. The beginning of the docstring, that belongs to the function in Figure 6.1 is listed in Listing 6.1. The IDE output is already very easy to use, but the main advantage of the automated formatting is best visible in the separate documentation websites (see Figure A.1).

1. Markup Syntax and Parser Component of Docutils (https://docutils.sourceforge.io/rst.html)

```
"""Get a :py:class:`OneDRegisterAccessor` object for the given
    register.
The OneDRegisterAccessor allows to read and write registers
    transparently by using
the accessor object like a vector of the type UserType. If needed, the
    conversion
to and from the UserType will be handled by a data converter matching
    the
register description e.g. a map file.
Parameters
----------
userType : type or numpy.dtype
  The userType for the accessor. Can be float, or any of the
      numpy.dtype
  combinations of signed, unsigned, double, int, 8, 16, 32 or 64-bit.
      E.g.
  `numpy.uint8`, or `numpy.float32`.
registerPathName : str
  The name of the register to read from."""
```

Listing 6.1: Python Docstring with reStructuredText

Another addition to the source code is the implementation of type hinting and function annotations[2]. An actual annotated function signature example with return type and defaults can be seen in the top half of Figure A.1. A variable can be annotated with a semicolon followed by its expected type. Although Python is a dynamically typed language, it can be helpful to have more information about methods, their parameters, and their return type. The difference between an annotated function and its standard counterpart is best seen during use. As long as the IDE does not have any information about the variable's type, it can not make any suggestions. This includes code completion or type-checking for overloaded and generic functions. An example of the respective IDE code completion can be seen in Figure 6.2.



```
def return_ID_with_annotation(acc: da.OneDRegisterAccessor):
    acc.
```

```
def return_ID_without_annotation(acc):
    acc.
```

Figure 6.2: IDE Code Completion with and without Annotations

2. As proposed in PEP 3107: https://peps.python.org/pep-3107/

7 Usage and Comparison

The features of the bindings are shown by an example, that is repeated for the legacy bindings, the current binding iteration, and the C++ counterpart. The scenario task consists of a programmable toaster. It has a selection wheel, where the user can select their desired toasting grade, a corresponding toasting grade detection sensor, and an internal trigger to eject the toast. It also features a sensor bank to measure the toast's thickness and an accordingly sized heating array to precisely warm even very unevenly cut bread to perfection. The adjustment for the best toasting results can be done with two variables: the base heat and the scale-up to accommodate different thicknesses.

Legacy Interface

The legacy bindings offer read and write functions through the device object by stating the register path. The write function also expects a fitting array for writing the data. Reads will always return with a float64 data type. Scalars can be emulated as single-entry arrays. Explicit opening or closing of the connection is not possible. It is handled internally and during garbage collection. The example is given in Listing 7.1.

```python
import mtca4u
import numpy as np

# Define heating parameters:
base_heat, thickness_scale = 200, 10

# Open the configuration file for the household:
mtca4u.set_dmap_location("household.dmap")

# Assign the toaster device from the alias found in the dmap file
toaster = mtca4u.Device("toaster")

# Read the data from the thickness_scanner, which has 4 elements:
thickness_sensors = toaster.read(registerPath="THICKNESS_SENSORS")

# The toaster also has a heating array, with 4 individual heating
    units to evenly heat the toast. The user  has to get the size
    information externally to create the correct array, which is then
    used for the write function:
heat_settings = np.array([0, 0, 0, 0])

# Set and write heating settings according to thickness:
for pos, thickness in enumerate(thickness_sensors):
    heat_settings[pos] = base_heat + thickness_scale * thickness
toaster.write(dataToWrite=heat_settings, registerPath="HEATING_ARRAY")

# Check how dark the toast has become and compare to user selection
    wheel setting, both scalar values, but the legacy has only arrays
    available:
```

```python
selection = toaster.read(registerPath="SELECTION_WHEEL")[0]
toasting_grade = toaster.read(registerPath="TOASTING_SENSOR")[0]
while toasting_grade <= selection:
    toasting_grade = toaster.read(registerPath="TOASTING_SENSOR")[0]

# The toast is ready. Eject the toast with a write command to the
    ejector. The ejector register is a void register, writing data is
    okay, but the data will be discarded.
toaster.write(dataToWrite=1, registerPath="EJECTOR")
```

Listing 7.1: Legacy Bindings Workflow Demonstration

The legacy interface was built with a two-level alias hierarchy, which consists of a module name and a register name. This system has since been replaced by a register path, that can have an arbitrary number of logical levels. The read and write functions are written to accept parameters in the order of module name, register name, and data, which eliminates the need to explicitly name the parameters when a register path is not used. As the module-register convention has been deprecated, it is not used in the current bindings. Register paths are used in all examples to keep them coherent.

New Bindings

For this simple scenario, the main difference between legacy and new bindings is the inclusion of accessor objects. The creation and handling of additional arrays can be skipped and the data stays with the correct register.

```python
import deviceaccess as da
import numpy as np

# Define heating parameters:
base_heat, thickness_scale = 200, 10

# Open the configuration file for the household:
da.setDMapFilePath("deviceInformation/household.dmap")

# Assign the toaster device from the alias found in the dmap file and
    open it:
toaster = da.Device("toaster")
toaster.open()

# Get accessors for the sensors and triggers, with user data types:
heat_settings = toaster.getOneDRegisterAccessor(np.uint16,
    "HEATING_ARRAY")
thickness_sensors = toaster.getOneDRegisterAccessor(np.uint8,
    "THICKNESS_SENSORS")
selection = toaster.getScalarRegisterAccessor(np.uint8,
    "SELECTION_WHEEL")
toasting_grade = toaster.getScalarRegisterAccessor(np.uint8,
    "TOASTING_SENSOR")
ejectTrigger = toaster.getVoidRegisterAccessor("EJECTOR")

# Read the data from the thickness_scanner:
thickness_sensors.read()

# Set heating according to thickness:
```

```
25 for pos, thickness in enumerate(thickness_sensors):
26     heat_settings[pos] = base_heat + thickness_scale * thickness
27
28
29 # Write the settings:
30 heat_settings.write()
31
32 # Check how dark the toast has become and compare to user selection
       wheel setting:
33 selection.read()
34 toasting_grade.read()
35 while toasting_grade <= selection:
36     toasting_grade.read()
37
38 # The toast is ready. Eject the toast with a write command to the
       ejector.
39 ejectTrigger.write()
40
41 # Closing the connection
42 toaster.close()
```

Listing 7.2: New Bindings Workflow Demonstration

C++ Workflow

A third example is given in Listing 7.3 to show the original C++ workflow. Apart from the obvious differences between the two languages in their respective syntax, the two versions follow the same principles and use the same method names. A small, but unavoidable detail is the setting of the user data type. While it comes as a template parameter in C++ (See line 17: `toaster.getOneDRegisterAccessor<uint16_t>(...)`, the Python version in line 15 takes the user data type as a first argument: `dev.getOneDRegisterAccessor(np.uint16,...`. This stems from the approach that is used to translate the C++ template mechanism into Python objects. More details on the implementation are given in section 5.2.

```
 1 #include <ChimeraTK/Device.h>
 2
 3 int main() {
 4
 5     //  Define heating parameters:
 6     int base_heat = 200;
 7     int thickness_scale = 10;
 8
 9     // Open the configuration file for the household:
10     ChimeraTK::setDMapFilePath("deviceInformation/household.dmap");
11     // Assign the toaster device from the alias found in the dmap file
           and open
12     // it:
13     ChimeraTK::Device toaster("toaster");
14     toaster.open();
15
16     // Get accessors for the sensors and triggers, with user data
           types:
17     ChimeraTK::OneDRegisterAccessor<uint16_t> heat_settings =
18         toaster.getOneDRegisterAccessor<uint16_t>("HEATING_ARRAY");
19     ChimeraTK::OneDRegisterAccessor<uint8_t> thickness_sensors =
        toaster.getOneDRegisterAccessor<uint8_t>("THICKNESS_SENSORS");
    ChimeraTK::ScalarRegisterAccessor<uint8_t> selection =
        toaster.getScalarRegisterAccessor<uint8_t>("SELECTION_WHEEL");
    ChimeraTK::ScalarRegisterAccessor<uint8_t> toasting_grade =
        toaster.getScalarRegisterAccessor<uint8_t>("TOASTING_SENSOR");
    ChimeraTK::VoidRegisterAccessor ejectTrigger =
        toaster.getVoidRegisterAccessor("EJECTOR", {});

    // Read the data from the thickness_scanner:
    thickness_sensors.read();

    // Set heating according to thickness:
    for (std::size_t pos = 0; pos < heat_settings.getNElements();
       ++pos) {
        heat_settings[pos] = base_heat + thickness_scale *
           thickness_sensors[pos];
    }

    // Write the settings:
    heat_settings.write();

    // Check how dark the toast has become and compare to user
       selection wheel
    // setting:
    selection.read();
    toasting_grade.read();

    while (toasting_grade <= selection) {
        toasting_grade.read();
    }

    // The toast is ready. Eject the toast with a write command to the
       ejector.
    ejectTrigger.write();

    //  Closing the connection:
    toaster.close();
    return 0;
}
```

Listing 7.3: C++ Workflow Demonstration

Convenience Functions

The new bindings come very close to the original source code, but they leave some of the convenience of the legacy bindings behind. It needs to be discussed, if some advantages of garbage collection should be used, to reduce some of the necessary code for leaner scripts. The closing of devices could be added to the finalization of the object. The connection is then closed when the respective object is no longer accessible. Likewise, the `open` method could be included in the initialization of the `Device` object. The implicit handling of the connections should be handled carefully. Errors in the connection process could be obscured and hinder tracing the underlying problem.

Another addition to the bindings could be the one-time write and read functions of the legacy bindings. They would work directly on the `device` object, while the accessors are requested and discarded in the background.

Push Types

The examples from the previous code snippets have so far covered a scenario where the host of the Python code initializes the data transfer. The main reason to rework the bindings was the inclusion of the other direction: when the data is pushed from the device to the host. This includes hardware interrupts and publish-subscribe protocols.

Listing 7.4 shows the setup in use. It features the same toaster example as above, but now makes use of the `THICKNESS_SENSORS` as a trigger. The script will only proceed, when a slice of bread is in place, e.g. one of the elements in the array is bigger than 0.

It is important to prepare the device and the accessor before the push type mechanism can take effect. This is done via the `activateAsyncRead` method from the `device` object, which is called after the device is opened. The second step is the passing of the `wait_for_new_data` `AccessMode` flag in the request for an accessor. The updated parts for the use of the accessor can be seen in Listing 7.4. The new code will ensure that the toaster device's thickness sensor is not permanently queried for the same information. New data is only pushed when an actual reading has happened.

```python
# Assume the device 'toaster' has been opened previously and was
    prepared via "toaster.activateAsyncRead()"
# The accessMode is set as followed:
thickness_sensors = toaster.getOneDRegisterAccessor(np.uint8,
    "THICKNESS_SENSORS",
    accessModeFlags=[da.AccessMode.wait_for_new_data])

# First read is always non-blocking:
thickness_sensors.read()
while thickness_sensors.min() < 1:
    thickness_sensors.read() # will now block until new data has been
        received

# Afterwards the script can return as before to set the heating.
```

Listing 7.4: Example of a Push Type Interaction

8 Conclusion and Outlook

The purpose of this thesis was the development, reasoning, and documentation of the extension of the Python bindings for the ChimeraTK DeviceAccess library. The status quo had working bindings for basic scripts. After several updates of the C++ libraries, the bindings were missing crucial functions like push-types for trigger usage. The necessary extension was a welcome invitation for a general rework. During this work, the outdated NumPy integration was reworked and the coverage of the bindings was increased. They include most functions that are offered by the original C++ library. A refactoring of the codebase for future projects was done in parallel.

Apart from this document the thesis also consists of the source code, which will now continue to be part of the ChimeraTK toolkit. Every class and method has complete documentation and every complex item has usage examples. Type hints and docstrings are consequently integrated into the code and improve the workflow with compatible IDEs.

The user type introduction needed fundamental rework on the middle layer of the library, as each of the non-void accessors can now support all of the numerical types from the underlying C++ libraries (see Table 3.1). The same principles were used to offer all the device functions. Increasing the readability and maintainability of the source code was a key aspect of the rework. Backward compatibility is still given through the legacy bindings, which are included in the package. They have been reworked to fit the new middle layer. A complete overview of the added functionality can be seen in Table 5.1.

The C++ accessors allow for the user types of `bool` and `string`. While the first one can be converted in the Python environment via NumPy, the string class cannot currently be used. The middle layer has been prepared, so missing user types can be implemented with some additions to the interface library.

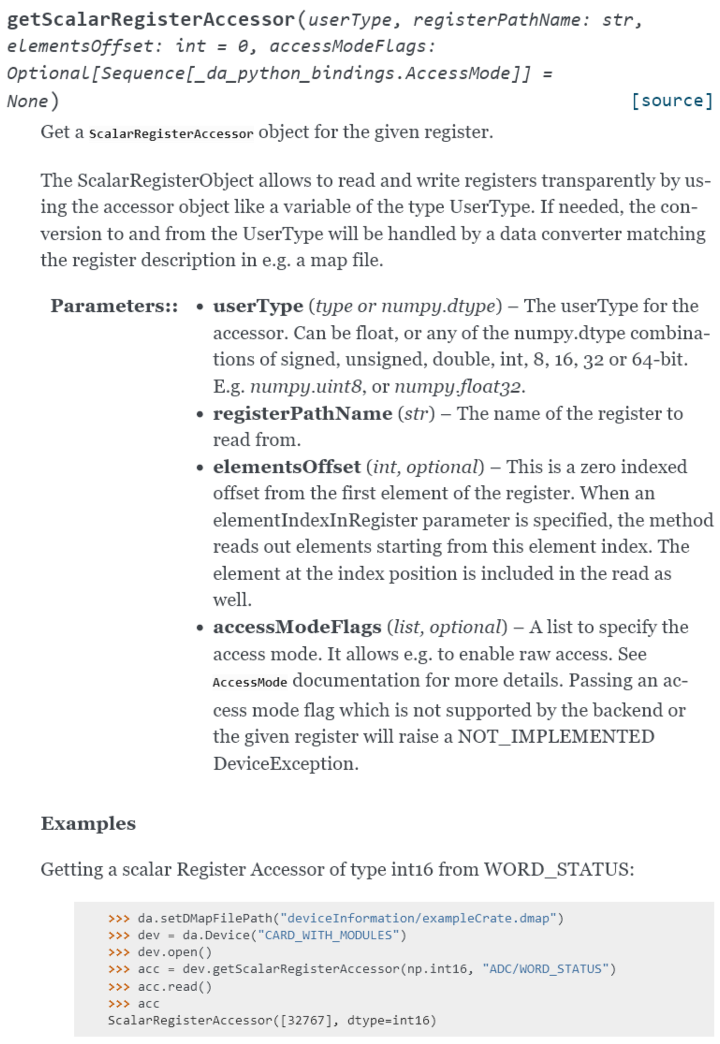
Also, some of the advanced features of the accessors handle additional classes like the `VersionNumber`. These classes were included with their basic functionality only.

Another aspect for future updates is the inclusion of the documentation examples in the test cases. This would ensure that the documentation is always correct and up to date.

Appendices

A Online Documentation

The complete documentation is available on Github.io[1] and on the included DVD of the physical copy of this thesis. An example of the generated website is shown in Figure A.1. The entry shows the documentation of the `getScalarRegisterAccessor` method of the `device` class.



getScalarRegisterAccessor(*userType, registerPathName: str, elementsOffset: int = 0, accessModeFlags: Optional[Sequence[_da_python_bindings.AccessMode]] = None*) [source]

Get a `ScalarRegisterAccessor` object for the given register.

The ScalarRegisterObject allows to read and write registers transparently by using the accessor object like a variable of the type UserType. If needed, the conversion to and from the UserType will be handled by a data converter matching the register description in e.g. a map file.

Parameters::
- **userType** (*type or numpy.dtype*) – The userType for the accessor. Can be float, or any of the numpy.dtype combinations of signed, unsigned, double, int, 8, 16, 32 or 64-bit. E.g. *numpy.uint8*, or *numpy.float32*.
- **registerPathName** (*str*) – The name of the register to read from.
- **elementsOffset** (*int, optional*) – This is a zero indexed offset from the first element of the register. When an elementIndexInRegister parameter is specified, the method reads out elements starting from this element index. The element at the index position is included in the read as well.
- **accessModeFlags** (*list, optional*) – A list to specify the access mode. It allows e.g. to enable raw access. See `AccessMode` documentation for more details. Passing an access mode flag which is not supported by the backend or the given register will raise a NOT_IMPLEMENTED DeviceException.

Examples

Getting a scalar Register Accessor of type int16 from WORD_STATUS:

```
>>> da.setDMapFilePath("deviceInformation/exampleCrate.dmap")
>>> dev = da.Device("CARD_WITH_MODULES")
>>> dev.open()
>>> acc = dev.getScalarRegisterAccessor(np.int16, "ADC/WORD_STATUS")
>>> acc.read()
>>> acc
ScalarRegisterAccessor([32767], dtype=int16)
```

Figure A.1: Example of a Method Description from the Documentation

The example shows an explanation of the method and its parameters and gives an example via Python code. Every entry also offers a link to the source code to inspect the actual implementation. Some entries might also have hints or warnings on their usage or references to derived or otherwise important classes.

The code for the documentation websites is directly generated from the source code of the library, as described in chapter 6.

1. https://chimeratk.github.io/DeviceAccess-PythonBindings/03.00/index.html

B Source Code

The source code of the software developed as part of this thesis can be found here:

- ChimeraTK DeviceAccess-PythonBindings GitHub repository, version tag 3.00.00[1],
- On the DVD included in the physical copy of this document.

1. https://github.com/ChimeraTK/DeviceAccess-PythonBindings/tree/03.00.00

C Licenses

The source code is licensed under the GPL-3.0 license [GNU]:

> Permissions of this copyleft license are conditioned on making available complete source code of licensed works and modifications under the same license or the GNU GPLv3. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. However, a larger work using the licensed work through interfaces provided by the licensed work may be distributed under different terms and without source code for the larger work. [...]

A copy of the complete license is distributed alongside the bindings source code.

The text of this thesis and its figures are licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license.

List of Figures

List of Tables

List of Listings

References

[Boo] Boost. *Python Bindings*. URL: https://www.boost.org/doc/libs/1_80_0/doc/html/mpi/python.html (visited on 08/27/2022).

[Bre12] Eli Bressert. *SciPy and NumPy: an overview for developers*. O'Reilly Media, Inc., 2012.

[DESa] DESY MSK. *ChimeraTK Repository Overview*. URL: https://github.com/ChimeraTK (visited on 09/02/2022).

[DESb] DESY MSK. *ChimeraTK-DeviceAccess - Basic Example*. URL: https://chimeratk.github.io/DeviceAccess/master/basic_example.html (visited on 08/31/2022).

[Die14] Charles Dierbach. *Python as a First Programming Language*. In: *J. Comput. Sci. Coll.* 29.3 (Jan. 2014), p. 73. ISSN: 1937-4771.

[GNU] GNU. *GNU Lesser General Public License v3.0*. URL: https://github.com/ChimeraTK/DeviceAccess/blob/master/COPYING (visited on 08/31/2022).

[Lut00] Lutz Prechelt. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search string-processing program*. Technical Report 2000-5. Universität Karlsruhe, Mar. 2000.

[Muk15] Arindam Mukherjee. *Learning boost C++ libraries*. en. Community experience distilled. Birmingham, England: Packt Publishing, July 2015.

[Num] NumPy. *Scalars v1.23 Manual*. URL: https://numpy.org/doc/stable/reference/arrays.scalars.html?highlight=scalars#defining-new-types (visited on 09/01/2022).

[Oli07] Travis E Oliphant. *Python for scientific computing*. In: *Computing in science & engineering* 9.3 (2007), pp. 10–20.

[SPH] SPHINX. *Python Document Generator*. URL: https://www.sphinx-doc.org/en/master/ (visited on 08/29/2022).

[Staa] Stack Overflow. *2021 Developer Survey*. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-programming-scripting-and-markup-languagesx (visited on 08/27/2022).

[Stab] Stack Overflow. *how to return numpy.array from boost::python?* URL: https://stackoverflow.com/a/10705352 (visited on 08/27/2022).

[Stac] Stack Overflow. *Which Boost features overlap with C++11?* URL: https://stackoverflow.com/a/8852421 (visited on 08/27/2022).

[Var+17] Geogin Varghese et al. *ChimeraTK-A Software Tool Kit for Control Applications*. In: *IPAC17, Copenhagen, Denmark* (2017).

[WCV11] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. *The NumPy array: A structure for efficient numerical computation*. In: *Comput. Sci. Eng.* 13.2 (Mar. 2011), pp. 22–30.

[Wri] Write the Docs. *Docs as Code*. URL: https://www.writethedocs.org/guide/docs-as-code/ (visited on 08/29/2022).

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, 29.09.2022 Christian Willner

Ort, Datum Unterschrift