Extension of the Python Bindings for the ChimeraTK DeviceAccess Library

11th MicroTCA Workshop for Industry and Research

Christian Willner

Hamburg, 08.12.2022

ChimeraTK DeviceAccess

ChimeraTK

- **C**ontrol system and **H**ardware **I**nterface with **M**apped and **E**xtensible **R**egister-based device **A**bstraction **T**ool **K**it
- Opensource, available on GitHub
- Maintained by the DESY MSK software group

DeviceAccess

- Lower level layer in ChimeraTK
- Unified abstraction for different backends
  - DOOCS
  - Userspace I/O
  - EPICS
  - PCIe
  - …

Language Comparison

- High-performance
- Resource-oriented
- Compiled

- Ideal for Control Systems

Language Comparison

- > High-performance
- > Resource-oriented
- > Compiled

- > Ideal for Control Systems

- > High-level syntax
- > Easy to read
- > Interpreted

- > Favorite for automation

Register Accessor Basics

Accessors are classes that offer backend-independent access to registers.

- Can be requested in different dimensionalities
- Decouple the register via UserBuffer
- Have auto-conversion to many UserTypes (int8, uint16, float, etc.)
- Supply blocking read functionality for synchronization

Project Intention

- Update bindings to mirror C++ functionality
  - Offer push / poll types
- Align C++ and Python workflow

Set-Up

C++

```cpp
#include <ChimeraTK/Device.h>
int main() {
    // Open the configuration file for the household:
    ChimeraTK::setDMapFilePath("household.dmap");
    ChimeraTK::Device toaster("toaster");
    toaster.open();
```

Python Bindings

```python
import deviceaccess as da
da.setDMapFilePath("household.dmap")
toaster = da.Device("toaster")
toaster.open()
```

Reading and Writing in C++

```cpp
// Get accessors for the registers, with user data types in <>:
ChimeraTK::OneDRegisterAccessor<uint16_t> heat_settings =
    toaster.getOneDRegisterAccessor<uint16_t>("HEATING_ARRAY");
ChimeraTK::OneDRegisterAccessor<uint8_t> thickness_sensors =
    toaster.getOneDRegisterAccessor<uint8_t>("THICKNESS_SENSORS");

// 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] = 200 + 10 * thickness_sensors[pos];
}
// Write the settings:
heat_settings.write()
```

Reading and Writing in C++ - Accessors in Math Operations

```
// Get accessors for the registers, with user data types in <>:
ChimeraTK::OneDRegisterAccessor<uint16_t> heat_settings =
    toaster.getOneDRegisterAccessor<uint16_t>("HEATING_ARRAY");
ChimeraTK::OneDRegisterAccessor<uint8_t> thickness_sensors =
    toaster.getOneDRegisterAccessor<uint8_t>("THICKNESS_SENSORS");

// 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] = 200 + 10 * thickness_sensors[pos];
}
// Write the settings:
heat_settings.write()
```

Reading and Writing in Python

Python Bindings

```
heat_settings = toaster.getOneDRegisterAccessor(np.uint16, "HEATING_ARRAY")
thickness_sensors = toaster.getOneDRegisterAccessor(np.uint8, "THICKNESS_SENSORS")
thickness_sensors.read()
for pos, thickness in enumerate(thickness_sensors):
    heat_settings[pos] = 200 + 10 * thickness

heat_settings.write()
```

New Python Accessors are NumPy Arrays

Python Bindings

```
heat_settings = toaster.getOneDRegisterAccessor(np.uint16, "HEATING_ARRAY")
thickness_sensors = toaster.getOneDRegisterAccessor(np.uint8, "THICKNESS_SENSORS")
thickness_sensors.read()
for pos, thickness in enumerate(thickness_sensors):
    heat_settings[pos] = 200 + 10 * thickness

heat_settings.write()
```

Bindings Offer Blocking Reads

```python
# Assume the device 'toaster' has been opened
# Prepare device 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
```

Bindings Offer Blocking Reads

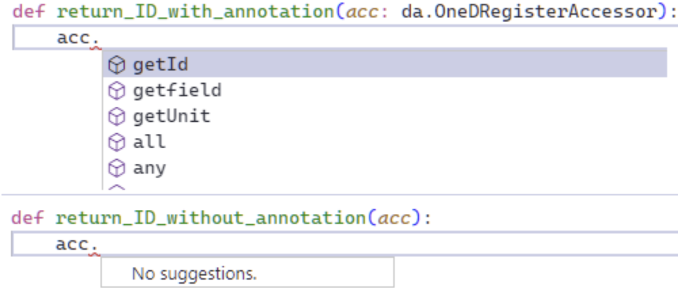
```
# Assume the device 'toaster' has been opened
# Prepare device via:
toaster.activateAsyncRead()
# The accessMode is set as followed:
thickness_sensors =
    toaster.getOneDRegisterAccessor(np.uint8,
                                    "THICKNESS_SENSORS",
                                    accessModeFlags=[da.AccessMode.wait_for_new_data])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

Use cases

Blocking reads work for *PCIe Interrupts, Publish/Subcribe protocols*

Function Annotation and Type Hints

New Python bindings have complete coverage with type hints and annotations



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

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

No suggestions.

Summary

- Python bindings usage closer to C++
- Almost complete coverage of C++ functionality
- Refactoring to facilitate future extensions
- Complete documentation

Outlook

- Continuous implementation of new functions from C++ base library
- Inclusion of (inefficient) comfort functions
- Diversion from explicit C++ workflow to be more Pythonic?

New Features in ChimeraTK DeviceAccess

- Userspace I/O backend (e.g. for SoC in Xilinx FPGAs)
- Double buffering plugin for continuous reads
  and guaranteed consistency of buffers
- Tango ControlSystemAdapter in development by Soleil
- Yocto layer available for DeviceAccess and PythonBindings
  ApplicationCore and ControlSystemAdapters will be available soon

Questions?

Contact

DESY. Deutsches
Elektronen-Synchrotron

www.desy.de

Christian Willner
0000-0002-2448-3698
MSK
christian.willner@desy.de
+49–40–8998–4904
DOI