ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

ABSTRACTED HARDWARE AND MIDDLEWARE ACCESS IN CONTROL APPLICATIONS

M. Killenberg*, M. Heuer, M. Hierholzer, T. Kozak, L. Petrosyan, C. Schmidt, N. Shehzad, G. Varghese, M. Viti, DESY, Hamburg, Germany
S. Marsching, aquenos GmbH, Baden-Baden, Germany
A. Piotrowski, FastLogic Sp. z o.o., Łódź, Poland
R. Steinbrück, M. Kuntzsch, HZDR, Dresden-Rossendorf, Germany
P. Prędki, Rapid Development, Łódź, Poland
C. P. Iatrou, J. Rahm, Technische Universität Dresden, Dresden, Germany
K. Czuba, A. Dworzanski, Warsaw University of Technology, Warsaw, Poland

Abstract

Hardware access often brings implementation details into a control application, which are subsequently published to the control system. Experience at DESY has shown that it is beneficial for the software quality to use a high level of abstraction from the beginning of a project. Some hardware registers for instance can immediately be treated as process variables if an appropriate library is taking care of most of the error handling. Other parts of the hardware need an additional layer to match the abstraction level of the application. Like this development cycles can be shortened and the code is easier to read and maintain because the logic focuses on what is done, not how it is done.

We present the abstraction concept we are using, which is not only unifying the access to hardware but also how process variables are published via the control system middleware.

INTRODUCTION

With the advent of the MicroTCA.4 standard it was possible to combine powerful, FPGA based computations with precise analogue electronics on comparatively large rear transition modules. [1] This allowed to implement a compact, fully digital control of the low level radio frequency signals (LLRF) at the FLASH accelerator. [2] The new, modular hardware platform had the need for a user space library to access the individual boards in the crate. Starting with PCI Express, which is used in MicroTCA.4, the DeviceAccess library became the basis of the *MicroTCA.4 User tool kit (MTCA4U)*. With its modern C++ interface, which abstracts the details of hardware access, the library was soon extended to Ethernet-based protocols and outgrew the original scope of only being used in MicroTCA crates. Consequently the software suite was renamed to *ChimeraTK (Control system and Hardware Interface with Mapped and Extensible Register-based device Abstraction Tool Kit)*.

The digital LLRF at FLASH was very successful and other facilities also started to use MicroTCA based systems for the accelerator controls. This brought the challenge that the complex device server has to be supported for multiple control system and middleware frameworks. The ControlSystemAdapter was introduced to decouple the application logic from the specifics of a particular control system and improve code reusability and maintainability.

Although starting as two separate libraries, the abstraction concepts for the DeviceAccess library and the ControlSystemAdapter were very similar. In the past year, their interfaces were unified and a new library called ApplicationCore was created. It is the consistent continuation of the abstraction process and facilitates the creation of device server applications in a control system independent way.

THE DeviceAccess LIBRARY

One of the main concepts of the DeviceAccess interface is the introduction of so called register accessors. These objects behave like a scalar integer or floating point variable in the user code, or an iterable one or two dimensional container of these data types, with additional functions to read from and write to the device. The accessor automatically allocates a data buffer of the correct size, does necessary data type conversions and takes care of implementation details like handshakes with the hardware.

An important abstraction step is the identification of registers by name. When creating the register accessor, the application is using a functional name instead of the numerical address in the I/O memory. This allows to not only access numerically addressed registers, but also process variables on another device servers, like a *DOOCS property* or an *EPICS channel*. Numerically addressed device backends require a name mapping table, which can also contain additional information like conversion parameters from a fixed point data format used on the hardware to floating point data types used on the CPU running the device server. For the firmware used at DESY this mapping is automatically generated. Register names can have a tree structure by separating hierarchy levels with a slash ('/'). Like this registers can be grouped into directories like in a (UNIX) file system. Firmware can now be implemented in a modular way with each functional block providing registers in a directory, and placing the same module twice does not cause naming conflicts, while the software accessing each block still finds all the register it needs in one directory. The `Device` interface provides a catalogue which lists all registers it can access.

* martin.killenberg@desy.de

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

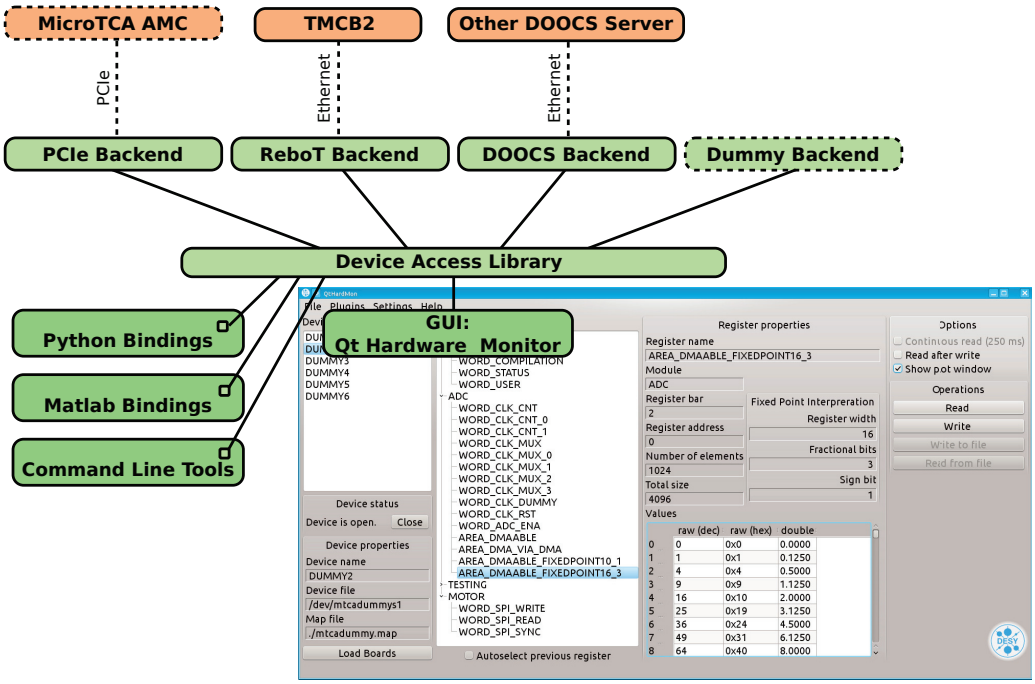

Figure 1: Overview over the DeviceAccess library with the most important backends, the available language bindings and the graphical user interface.

It turns out that it is an advantage to not only access registers by a functional name, but also use alias names when opening the hardware device itself. Like this direct hardware access can easily be replaced with a proxy though another device server or a dummy for testing without the need to modify the source code. A "device mapping" configuration file provides information which backend type and which connection parameters to use, and maps it to an alias name. The config file is read at run time whenever a device is opened, so it is possible to even change the backend in a running application if need be.

DeviceAccess as the basic library for hardware-accessing device servers comes with set of tools which allow direct, interactive communication to the hardware. There are language bindings for Python, Matlab and the Linux command line, as well as a graphical user interface (Fig. 1).

THE ControlSystemAdapter LIBRARY

The ControlSystemAdapter was developed out of the need for a low level RF control application for accelerators to be operated at several facilities with different control system infrastructure. At DESY, Hamburg, where the application is developed, the DOOCS [3] framework is used at FLASH [2] and the European XFEL [4]. Flute [5] at KIT in Karlsruhe is using EPICS 3 [6]. The control system of the ELBE accelerator [7] at HZDR in Dresden is based on WinCC, and OPC UA [8] [9] will be used as an interface. TARLA [10] in Ankara will use EPICS 4.

The idea of the control system adapter is to provide an abstraction layer which facilitates writing an application that can natively run with all these middlewares. The LLRF control server for instance provides hundreds of process variables and control plots, and has complex algorithms like adaptive feed forward calculations and control table generations. To provide a source code fork for each target control system is not maintainable. As a solution one could develop the application server within one main middleware framework and have a gateway server for the others. Although a viable option, this solution has many disadvantages:

- Lower performance as data is always copied and converted to a different data format
- Possible incompatibilities and problems to reproduce a certain feature in the other middleware
- Extra maintenance because the application has to be configured in two environments, and both environments have to be set up, incl. brokers, central databases etc.
- Expertise on both platforms is required, which especially for not widely used systems like DOOCS is problematic

The ControlSystemAdapter avoids these problems because the resulting executable is a genuine device server in the particular middleware. In order for this to work, the adapter has to ensure a lock free, multi-threaded implementation of process variable which are published inside the application code. This was the result when studying the locking schemes of several middleware frameworks. The adapter implements this through lock-free queues and a dedicated main thread for the application itself, which basically replaces the main routine of a regular C++ application. More

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

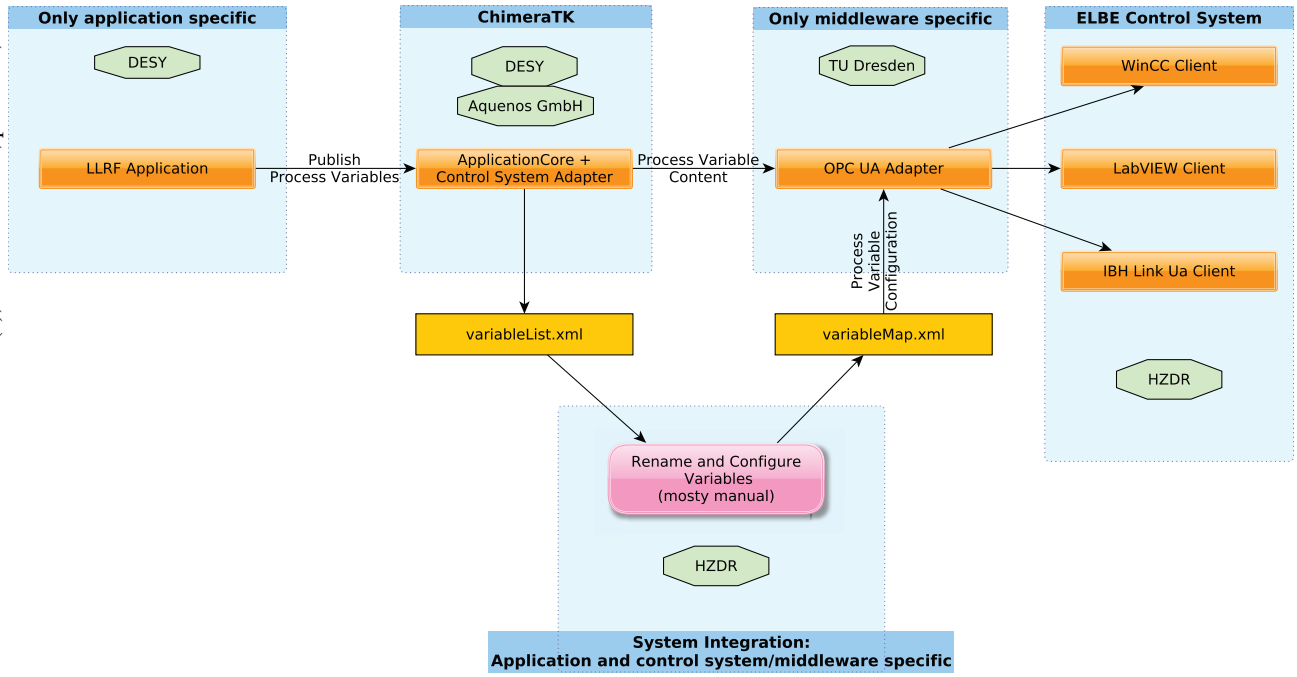

Figure 2: The LLRF server developed at DESY, Hamburg, has been integrated into the WinCC-based ELBE control system at HZDR, Dresden-Rossendorf. The ControlSystemAdapter, which has mainly been developed by DESY and aquenos GmbH, is paired with the OPC UA adapter, developed at TU Dresden. The control system integration and operation is done by HZDR.

detailed information on the implementation of process variables in the ControlSystemAdapter can be found in [11].

Each application using the control system adapter is derived from the ApplicationBase class. This is a singleton with the application being the only implementation. The application is instantiated in the object code of a library, which is then linked against the ControlSystemAdapter and one of its middleware-specific adapter backends, which currently are available for DOOCS [12], EPICS 3 [13] and OPC UA [14].

Application Development and System Integration

To really make an application control system independent one has to identify which parts of the required functionality are in the application proper, and which have to happen in the system integration step. For the application we have identified the following tasks:

- Define the process variables which are to be published in the control system
- Implement the algorithms
- Access the hardware

A number of features which are provided by a device server are middleware dependent and have to happen during system integration:

- Publish process variables via the middleware's communication protocol
- Define the variable name visible in the control system
- Define middleware dependent features and data types
  - Data persistency for setpoints
  - Aggregated data types, e.g. for plotting
  - Server-side histories

The middleware dependent features might or might not be available in the target control system. In a DOOCS environment for instance data histories are stored on the device server, while in EPICS a separate data archiver is collecting the data, and the IOC[1] does not know about histories. Hence, this cannot be part of the control system independent application but has to happen during system integration.

At first sight it might seem unnecessary to rename the variables which are defined in an application. If the system integration is happening in the application code in a classical device server the renaming step is indeed not needed. Here the application is already becoming dependent on the naming scheme of the control system. So even if the same middleware is used, an application designed for one facility might not be directly usable in another environment, simply because the names do not match. Hence a name mapper is required for an application to really become control system independent.

Experience showed that the code defining the middleware-specific features causes many lines of repetitive code in C++, especially if each variable has to be touched at least three

[1] Input/Output Controller

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

times: At the definition in the header file, during initialisation in the constructor and when the values are assigned or read out in the application logic. This kind of code makes it difficult to port the application and should be avoided. However, this step has to happen for a proper system integration. It mainly consists of configuring and renaming variables which have been defined in the application part. It turns out that this task can efficiently be handled by a configuration file, for instance in XML. Especially the renaming should not be coded in C++. This led to the decision that the ControlSystemAdapter plugin for each target middleware needs a configuration file.

The system integration step is the only part which is application *and* control system/middleware dependent. This kind of source code is expensive because it has to be repeated/rewritten for each device and control system. Not having to implement it in C++ at all has several advantages:

- The executable does not have to be recompiled, packaged and distributed
- No C++ expertise is required to do the configuration
- It is planned to have graphical tools which help to create the config files (dedicated tools for each target middleware)

Figure 2 gives an overview of the integration step for the LLRF server at the ELBE accelerator at HZDR in Dresden-Rossendorf.

THE ApplicationCore LIBRARY

The goal of the ApplicationCore library is to provide a tool for writing intrinsically control-system-independent applications. The main focus is on the abstraction of process variables. The starting point were the DeviceAccess and the ControlSystemAdapter library. In the beginning, they were two separate libraries with different interfaces. But the abstraction of registers accessors and process variables were so similar that they were unified and they now have a common interface.

The main idea of ApplicationCore is that an application only knows about input and output variables. If it is abstracted away whether the variable is coming from the hardware, another part of the software or the control system, it is unlikely that the application will be sensitive to specifics of a control system middleware.

Furthermore, ApplicationCore should encourage a modular design and provide an easy to use interface which avoids boiler plate code as much as possible in C++11.

Figure 3 shows the layout of a device server written with ApplicationCore. The example application provides access to a controller which is implemented in hardware. It is connected to the software using a DeviceAccess module. The application itself consists of two modules called "tableGeneration", which is generating input tables for the controller, and "automation", which is changing the setpoint. A central part of the application is the connection code (middle left block in Fig. 3). It describes which module inputs and outputs are connected to control system variables, the hardware or other application modules. To have efficient code, ApplicationCore uses *inversion of control* to populate process variables. An application module only defines that it requires an input or output variable, and the instantiation of the particular type is done when connecting the module. The "setPointTable" output of the "tableGeneration" module for instance is directly connected to a DeviceAccess module. In this case the output variable will directly be a register accessor of the DeviceAccess library, which eliminates unnecessary copying of data from one module to another.

Application Modules

Application modules are the building blocks of each application. Each module has input and output variables and a thread which is executing the algorithm. Process variables can have two different update modes: Push type (orange in Fig. 3) and poll type (green). For poll type variables the sender is passive and expects the receiver to query the latest value, which it will deliver without blocking. Reading a sensor value from the hardware through the DeviceAccess interface usually is such a poll type variable. For push-type variables new data is send when it is available, usually into a queue. The receiver can now work in three modes.

1. *Non blocking read*: Get the next element in the queue. If no new data is available the last received value stays in the buffer and the function returns immediately.
2. *Read latest*: The queue is emptied and the receiver gets the latest value. This behaviour is equivalent to a poll-type receiver.
3. *Blocking read*: If the queue is empty, the read function blocks until new data has been received.

The blocking read can be used to synchronise a module with other threads or the hardware. In the example Fig. 3 the "trigger" in the "automation" module is such a blocking input. Also "pulseLength" and "setpoint" of the "tableGeneration" module are push-type inputs, but they are combined in a variable group called "tableParameters". This allows the module thread to block until either *all* parameters have been updated (readAll), or until *any* of the parameters has new data (readAny).

The Connection Code

The main challenges of the connection code are to have an intuitive syntax and to minimise the number of code lines needed to describe the connection.

As an example, the "currentSetPoint" output of the "automation" module is connected to the "setpoint" input of the "tableGeneration" module and the "currentSetpoint" of the ControlSystemAdapter module using a fan out, indicated by the orange circle. Connections are defined using the » operator:

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

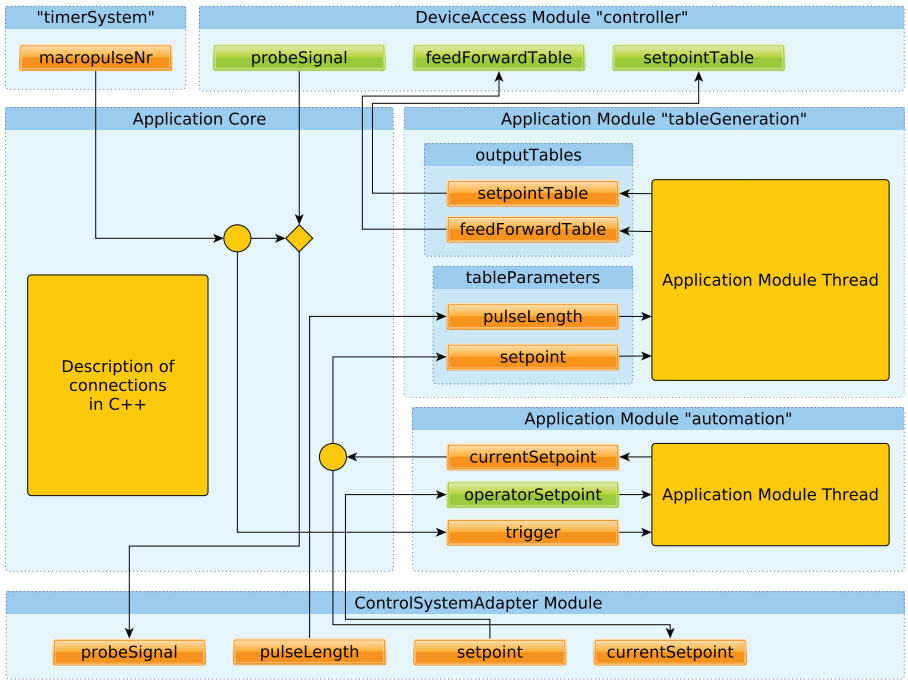

Figure 3: Example of a device server using the ApplicationCore library. The application modules "tableGeneration" and "automation" are written by the application programmer, as well as the connection code. The DeviceAccess and the ControlSystemAdapter modules are provided by the library. Note that the "timerSystem" is also a DeviceAccess module.

```
automation.curentSetpoint >>
tableGeneration.tableParameters.setpoint >>
controlSystem("currentSetpoint");
```

This single line of code automatically creates a fan-out module and the connections. When connecting a variable to the control system module it is automatically instantiated in the ControlSystemAdapter. The data type is determined from the variable which is being connected.

If every variable had to be mentioned explicitly in the connection code, this would result in many lines of code. To avoid this, modules and variable groups have a "connectTo()" function which allows to automatically connect all variables with the same name. Both tables of the "outputTables" group for instance can be connected to the "controller" module with a single command. The connection command works for inputs as well as outputs and also recursively for sub-groups. Further modelling is possible by adding arbitrary tags to process variables. All variables tagged for instance with "CS" in a module group could be connected to the control system module with one command, including variables from all included modules and variable groups.

To improve the use of generic modules it is possible to rename variables by adding meta data, and to eliminate hierarchy levels. A generic limiter module with three variables "input", "output" and "maximum" would be an example where you don't want to publish a variable named "limiter/output" to the control system, but rather something like "temperatureSetpoint".

CONCLUSION

ChimeraTK ApplicationCore is a tool which helps to design control applications in a middleware independent way. It is based on the DeviceAccess library and the ControlSystemAdapter, which provide abstracted access to hardware and control system middleware, respectively. ApplicationCore's hierarchical data model helps to write modular software, which improves the maintainability and facilitates to achieve the required abstraction to keep the application control system independent. Tools like the automatic generation of an XML file with the variable content of an application or plotting the variable hierarchy tree simplify the system integration.

An LLRF server which was developed at DESY, Hamburg, using ApplicationCore and the DOOCS adapter was tested successfully at the ELBE accelerator at HZDR, Dresden, using the OPC UA adapter. [15] The ApplicationCore-based server will be easier to maintain than the original server directly implemented in DOOCS, not only due to the improved modularity but also because of the significantly reduced number of code lines. Currently other DOOCS servers are being ported to ApplicationCore to profit from the abstraction and the multi-threading, even though not all of them will have to run at several facilities.

The ChimeraTK suite is open source software which is published under GNU General Public License or the GNU Lesser General Public License (depending on the software component). It is available under [16].

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

REFERENCES

[1] PICMG®, *MicroTCA® Enhancements for Rear I/O and Precision Timing, MicroTCA.4 R1.0*, 2011/2012

[2] C. Schmidt et al., Real time control of RF fields using a MicroTCA.4 based LLRF system at FLASH, 19th IEEE Real-Time Conference, Nara, Japan, 2014

[3] The Distributed Object Oriented Control System (DOOCS), http://doocs.desy.de/

[4] M. Altarelli et al., XFEL : The European X-Ray Free-Electron Laser : Technical Design Report, DESY-2006-097, DESY, Hamburg, 2007

[5] S. Marsching et al., Status of the FLUTE Control System, WPO013, PCaPAC2014, Karlsruhe, Germany, 2014, http://www.jacow.org/

[6] Experimental Physics and Industrial Control System (EPICS), http://www.aps.anl.gov/epics/index.php

[7] F. Gabriel et. al., The Rossendorf radiation source ELBE and its FEL projects, Nucl. Instr. Meth. B 161-163, 1143, 2000, http://dx.doi.org/10.1016/S0168-583X(99)00909-X

[8] OPC Unified Architecture - Part 1: Overview and Concepts, IEC TR 62541-1:2010, 2010, available at https://webstore.iec.ch/publication/7172

[9] open62541 — An open source and free C (C99) OPC UA stack licensed under LGPL + static linking exception, http://open62541.org/

[10] A. Aksoy et. al., TARLA: The First Facility of Tukrish Accelerator Center (TAC), WEPAB087, IPAC2017, Copenhagen, Denmark, 2017

[11] M. Killenberg et. al., Integrating control applications into different control systems, TUD3O05, ICALEPCS2015, Melbourne, Australia, 2015

[12] ControlSystemAdapter-DoocsAdapter: The DOOCS implementation for the ControlSystemAdapter, https://github.com/ChimeraTK/ControlSystemAdapter-DoocsAdapter

[13] MTCA4U for EPICS — Device support for integrating MTCA4U with EPICS, http://oss.aquenos.com/svnroot/epics-mtca4u

[14] ControlSystemAdapter-OPC-UA-Adapter: The OPC-UA implementation for the ControlSystemAdapter, https://github.com/ChimeraTK/ControlSystemAdapter-OPC-UA-Adapter

[15] R. Steinbrück et. al., Control System Integration of a µTCA.4 based digital LLRF using the ChimeraTK OPC UA Adapter, ICALEPCS2017, THPHA166, Barcelona, Spain, 2017, this conference

[16] ChimeraTK: Control system and Hardware Interface with Mapped and Extensible Register-based device Abstraction Tool Kit, https://github.com/ChimeraTK/