Integrating real-time control applications into different control systems

Martin Killenberg, Martin Hierholzer, Christian Schmidt, Nadeem Shehzad, Sebastian Marsching, Chris Paul Iatrou, Reinhard Steinbrück, Michael Kuntzsch, Jan Wychowaniak

Abstract—**Porting complex device servers from one control system to another is often a major effort due to the strong code coupling of the business logic to middleware data structures. Together with its partners from the Helmholtz Association and from industry, DESY is developing a control system adapter as part of the MTCA4U tool kit. It allows to write applications in a control system independent way, while still being able to update the process variables and react on control system triggers. Special attention has been paid to make the implementation thread safe and real-time capable, while still providing the required abstraction and avoiding performance losses. We report on the status of the project and the plans to implement new features.**

I. Introduction

WITH embedded systems becoming increasingly powerful, the algorithms in the devices which are accessed via control systems are becoming more and more advanced. Especially on MicroTCA[1] systems the hardware usually features a powerful multi-core CPU with several GB of RAM. The MicroTCA.4[2] extension brings trigger and clock lines, as well as large rear transition modules which can be used for demanding analogue control applications.

Many particle accelerators which are currently being built are using or will use MicroTCA.4 for control of the radio frequency (RF) in the accelerator, for instance FLASH[3] and the European XFEL[4] hosted at DESY, Hamburg, ELBE[5] at Helmholtz-Zentrum Dresden-Rossendorf, or FLUTE[6] at KIT, Karlsruhe. The complex RF control applications shall be reused across the different accelerators, while all the facilities are using different control system middlewares. It turned out that porting the software to a different middleware is a major effort because the code is strongly coupled to the original middleware. This lead to the idea to have an adapter layer between the device library, which implements the algorithms, and the middleware, which provides the communication protocols and integration into the facility's control infrastructure.

Most of the middlewares use locking mechanisms to implement thread safety, so the process variables cannot directly be used in real-time applications. So in addition to abstracting the actual middleware in use, the adapter should also allow a real-time thread to access process variables in a lock-free manner.

The control system adapter has been developed as part of the MicroTCA.4 User Tool Kit (MTCA4U)[7], a collection of libraries to facilitate the implementation of control applications. The libraries have intentionally been designed as a universal tool kit. Their use is by no means limited to the MicroTCA platform, and there are already applications which use MTCA4U outside of this scope. Consequently MTCA4U has recently been renamed to "Control system and Hardware Interface with Mapped and Extensible Register-based device Abstraction Tool Kit" (ChimeraTK)[8].

Manuscript received June 2, 2016. This work is supported by the Helmholtz Validation Fund HVF-0016 "MTCA.4 for Industry"

M. Killenberg is with Deutsches Elektronen-Synchrotron, Hamburg, 22607 Hamburg, Germany (corresponding author, e-mail: martin.killenberg@desy.de)

N. Shehzad is with Deutsches Elektronen-Synchrotron, Hamburg, 22607 Hamburg, Germany (presenter, e-mail: nadeem.shehzad@desy.de)

M. Hierholzer, C. Schmidt are with Deutsches Elektronen-Synchrotron, Hamburg, 22607 Hamburg, Germany

Sebastian Marsching is with aquenos GmbH, 76532 Baden-Baden, Germany

C. P. Iatrou is with Technische Universität Dresden, 01062 Dresden, Germany

R. Steinbrück, M. Kuntzsch are with Helmholtz-Zentrum Dresden Rossendorf, 01314 Dresden, Germany

J. Wychowaniak is with Łódź University of Technology, 90-924 Łódź, Poland

II. Requirements

The main task of the control system adapter is to allow application code to access process variables which are communicated to the control system independently from the middleware. For this, the adapter has to use the functionality which is provided by the middleware, like communication protocols or the addressing scheme.

The part of code which is device *and* middleware dependent has to be minimal, zero if possible. This type of code is causing the huge workload when porting and maintaining applications for multiple control systems. Abstraction is easy to achieve if data is simply copied back and forth between two domains, but this comes with a performance penalty. So an additional requirement to the adapter is to avoid unnecessary copying, especially of large data structures like arrays.

With the CPUs in embedded systems becoming more powerful, device server applications do not only provide the interface to the control system, but also implement control loops in software. Thus the minimum requirement for the control system adapter is thread safety, so that the control loop can run in its own thread. Ideally, the loop is running in a real-time thread. As already mentioned in the introduction, many middlewares use locking to implement thread safety, so the process variables provided by the middleware are not directly usable in the real-time thread. Hence, the adapter should provide a lock-free mechanism to access process variables, which are then passed on to the middleware layer.

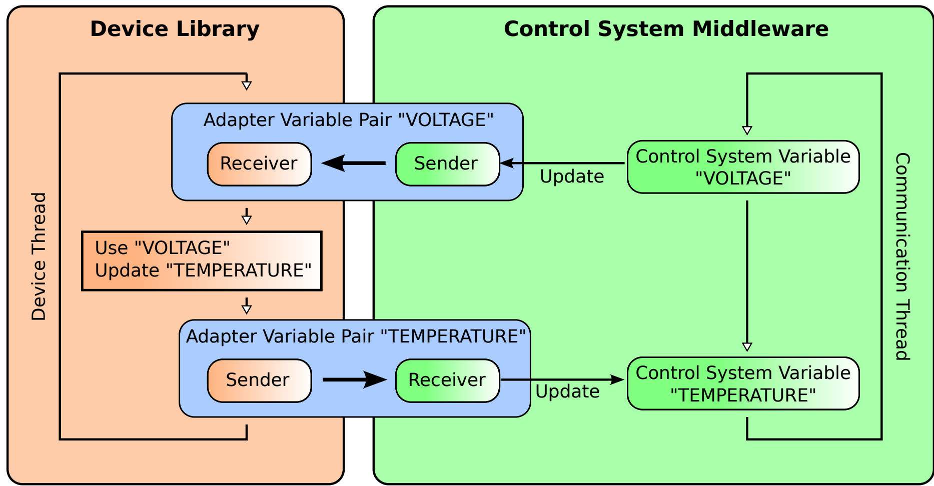

Fig. 1. The update flow using the control system adapter.

Looking at different middlewares, and control system installations at different facilities, it turned out that there is a large variety of naming schemes, which are partly incompatible. For one device implementation to work in different environments without modification, the adapter has to provide a mapping layer which translates process variable names of the device implementation into the names which are published to the control system. As this mapping is used to configure the device integration, it can also be used to configure control system specific features which are not covered on the control system independent part.

As a starting point and to check if the abstraction is working, the adapter is tested with two middlewares: DOOCS[9] and EPICS[10]. DOOCS (used at DESY for FLASH and the European XFEL), has an object-oriented data model written in C++. EPICS 3, one the most widely used middlewares for particle accelerators and used at FLUTE, has a channel-based C API. We intentionally used two conceptually different middlewares, hoping that the abstraction needed to work with these two should also allow other software to be used without modifications in the design. In addition, an adapter for OPC-UA[11] is under development. It is based on the open62541 OPC-UA stack[12].

III. Design concept

The first implementation of the adapter focuses on process variables. It provides data structures for scalars (8, 16 and 32 bit signed and unsigned integers, single and double precision floating point), strings, and arrays of the numerical data types. Each process variable is identified by a unique name which describes its function inside the device code ("TEMPERATURE" for instance for a device with a temperature sensor). The name does not contain information where the device is installed and in which context it is used. That part depends on the control system and the facility, and is added in the control system specific part of the adapter, not in the device part.

The original idea to avoid copying, especially for large arrays, was to have a single instance of the data. This would be stored in a middleware dependent type, an instance which always has to be there to work with the particular control system. The adapter would provide a wrapper, which would be used inside the business logic. But it turns out that this approach is not viable. DOOCS and EPICS use locking to implement thread safety, so the process variables are not real-time capable. In addition, the two middlewares have different locking schemes for their variables, and only the middleware could know when it is safe to access the variables, but not the middleware independent device part.

The solution is to have a middleware independent instance on the device side which can be accessed at any time, and the middleware variable on the other side, which can have a middleware lock (Fig. 1). These variables have to be synchronized, which means that the abstraction at this point requires one copy which cannot be avoided.[1] To avoid race conditions and endless loops, it was decided to restrict process variables to be unidirectional in the current implementation ("control system to device" or "device to control system").

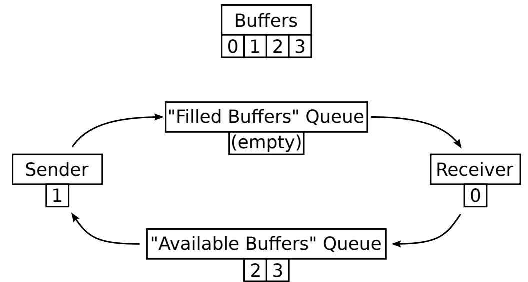

Fig. 2. For arrays a process variable pair with sender and receiver features two lock-free queues and at least four pre-allocated buffers.

To allow real-time threads in the device library, the variable on the device side has to be lock-free. To implement this, the adapter's process variable is always a sender-receiver pair (Fig. 2), using lock-free queues for transfer. As dynamic memory allocation is not allowed in a real-time thread, the

[1] A copy can be avoided if the middleware data type allows swapping of the internal buffer with a `std::vector` like it is used by the control system adapter.

mechanism is working with a pool of pre-allocated buffers for arrays. Sender and receiver each hold the reference to one buffer at all times, which allows the business logic to access and modify the data at will, except while sending or receiving. The data being transferred is always the reference to a buffer, not the buffer itself, which avoids unnecessary copying of large data structures.[2]

When receiving, the receiver will pop the head of the “filled buffers” queue. If it could get (the reference to) a new buffer, it will push the now outdated buffer to the “available buffers” queue, so the sender can reuse it. In case there are no updated buffers available, the receiver will hold on to the current buffer, as it contains the most up-to-date information that is available on the receiver side.

Before actually sending, the sender will pop the head of the available buffers queue to be sure it has a new buffer which it can fill after sending (at all times there must be at least one buffer on each the sender and the receiver side). After that, the buffer to be send is pushed to the “filled buffers” queue. Both receiver and sender first pop the head of the queue where they retrieve the next buffer, and then push the buffer which has been processed for use by the other side. As this can happen at the same time, it means there have to be at least four buffers.

In contrast to a triple buffer, which is a common scheme in real-time applications, this approach does not require a dirty flag for the buffer, and the receiver does not have to swap back to keep the latest available buffer in case no updated buffer is found. As a further advantage, the number of buffers can simply be increased, allowing a longer queue in case there are fluctuations on the receiver side, but it is fast enough to catch up with a certain backlog.

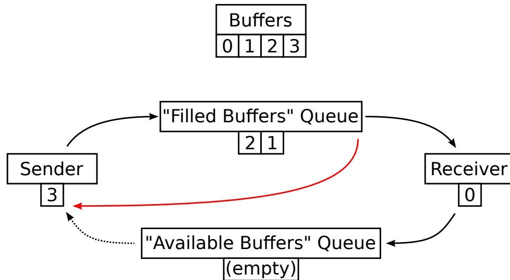

Fig. 3. If the queue of available buffers is empty, a further send call would pop the head of the “filled buffers” queue to be overwritten (buffer index 1 in this picture) and perform the send operation (buffer index 3 goes to the queue). Like this, the information in the queue can be updated, whether the receiver is active or not.

If the receiver is too slow to process data at the rate at which the sender is producing it, data will be lost. This cannot be avoided. In our implementation with a fixed number of buffers it means the “available buffers” queue is empty. In a naive implementation, the sender would keep its current buffer to overwrite it (buffer 3 in Fig. 3). However, it is not a good solution to stop sending because this would result in newer data being discarded in favor of older data that is already in the queue. If for instance the receiver was down for several minutes, the information in the queue would be several minutes old when it is received, while newer data has been overwritten. Instead, the oldest information which has not yet been received should be dropped. So in case no free buffers are available, the sender will pop the head of the “filled buffers” queue (buffer 1 in Fig. 3) to be overwritten, and send buffer 3, which has just been filled. Like this, the data in the queue is being updated even if the receiver is not active.

[2]For scalars copying the value is not more expensive than copying the pointer. In this case the values are directly copied and no additional buffers are needed.

IV. Creation of a Process Variable

Not only the device business logic has to be independent from the control system side, also the amount of device-specific middleware code should be minimized. For this reason the creation of process variables is automated as much as possible in the adapter (Fig. 4). The device is calling the create function of the adapter, giving the name and the direction (“device to control system” or “control system to device”). Depending on the direction, a sender or a receiver is returned to the device side. The other partner of the process variable pair is stored in a list. After all process variables have been created by the device, the middleware calls a function which creates the instances of the control system variables for all process variables known to the adapter. This function does not depend on the device logic, which improves the decoupling. It is, however, part of the middleware specific part of the adapter and looks different for each middleware. The level of abstraction which can be achieved here may vary.

V. Status and Outlook

The current implementation of the control system adapter provides process variables in a common, middleware independent part, which is thread safe and real-time capable. A middleware specific, but device independent part has to be added to the adapter for each target middleware. Implementations for DOOCS and EPICS have been written and are ready to use in an early version, an adapter for OPC-UA is currently being implemented.

As “proof of concept”, some example devices with a couple of scalars and arrays have been created and tested using either DOOCS or EPICS, and have afterwards been ported to the other middleware. This porting worked smoothly in both directions. As expected, the device code itself did not have to be modified. The amount of device-specific code is very small on the control system side. The example devices are down to three lines of device-dependent middleware code, for DOOCS as well as for EPICS, independent from the number of process variables.[3]

The next step to make the adapter useful for real life applications is to implement the mapping which translates device process variable names into the names which are shown in the control system. As described above, this feature is needed to integrate the device code with fixed process variable names into the various facilities with different naming schemes and conventions. This integration step is implemented on the

[3]The device configuration files for the control system have to be written in addition.

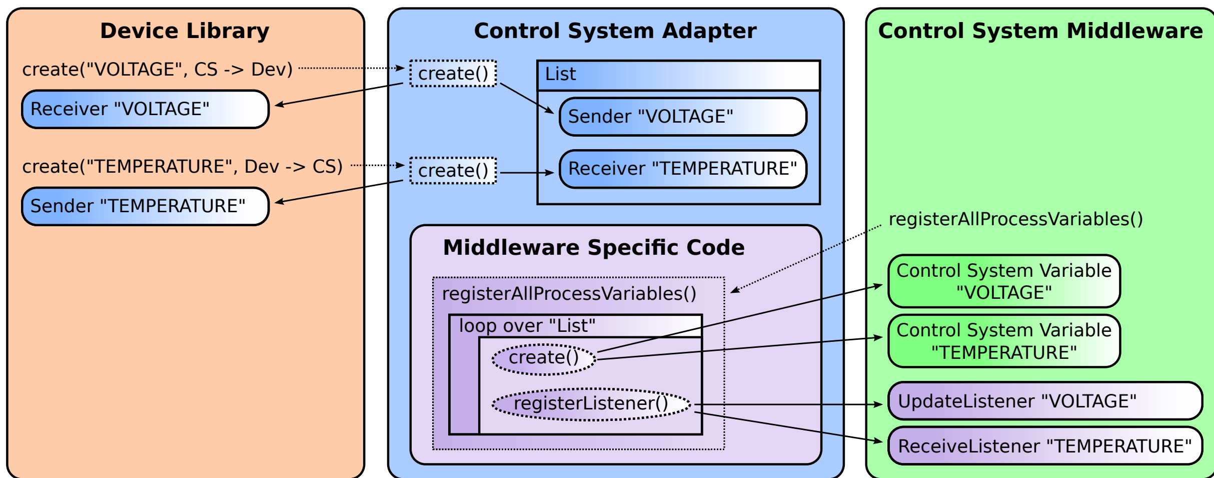

Fig. 4. The creation of process variables is requested on the device side. The instantiation on the control system side is automated as much as possible inside the adapter.

control system side, and each adapter implementation will bring its own mapping scheme to accommodate the specifics of the particular middleware.

An application often has additional requirements, like range limits or histories. If the features are provided by the middleware, these implementations should be used. If a feature is not provided by the middleware, there are two possible solutions: Either the feature is optional for the application to run, or it is implemented in the control system adapter. An optional features for instance is a server-side variable history. If the middleware provides this feature, it can be implemented (like in DOOCS). If the middleware does not provide it, the history can be implemented on the client side or with special middle-layer servers (like in EPICS 3).

An example for a feature which should be provided by the control system adapter is a guaranteed range limit on process variables which the application code can rely on. The feature will be executed on the control system side to be able to use the middleware implementation if provided. In case it is not available from the middleware a default implementation from the control system adapter will be used. Without such a default implementation each feature would have to be implemented for each target middleware, which does not scale and is not maintainable. It is currently being discussed how the adapter can be extended with a plug-in mechanism that allows the definition of new features and fulfills these requirements.

Further features under consideration are engineering units, alarms, configurable validators provided by the device side.

VI. Conclusions

The MTCA4U control system adapter[4] provides an interface to use process variables in a device library without introducing a coupling to a particular control system middleware. The implementation is lock free and transfers pre-allocated buffers without copying. This enables the device logic to use process variables in a real-time thread, even if the middleware in use is not real-time capable. In addition it is efficient and brings the required abstraction to allow operation in middlewares with different locking mechanisms.

The adapter consists of a common part, which implements the decoupling, and a control system specific part, which provides the particular middleware implementations. Adapters for DOOCS and EPICS have been developed, a version for OPC-UA is being implemented.

Currently the adapter is being extended with a name mapping to allow the integration into the naming scheme of the control system. A plug-in mechanism will provide features like limiters, engineering units and history. These improvements will bring the control system adapter from a proof of concept study to a tool for real life application.

All software is published under the GNU Lesser General Public License or the GNU General Public License and is available in the respective software repositories[13], [15], [14], [16].

[4]MTCA4U has recently been renamed to ChimeraTK

REFERENCES

[1] PICMG®, *Micro Telecommunications Computing Architecture, MicroTCA.0 R1.0*, 2006

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

[3] 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

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

[5] 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

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

[7] N. Shehzad et al., *Modular Software for MicroTCA.4 Based Control Applications*, *These Proceedings*, 20th IEEE Real Time Conference, Padova, Italy, 2016

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

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

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

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

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

[13] *ChimeraTK ControlSystemAdapter — An adapter layer which allows to use control applications with different control system software environments*, https://github.com/ChimeraTK/ControlSystemAdapter

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

[15] *MTCA4U EPICS Adapter*, Subversion Repository, http://oss.aquenos.com/svnroot/epics-mtca4u/

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