ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV012

AUTOMATED DEVICE ERROR HANDLING IN CONTROL APPLICATIONS

M. Killenberg*, J. Georg, M. Hierholzer, C. Kampmeyer[1], T. Kozak, D. Rothe,
N. Shehzad, J. H. K. Timm, G. Varghese[2], C. Willner,
Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany
[1]now with Sokratel Kommunikations- und Datensysteme GmbH,
Langenhorner Chaussee 625, 22419 Hamburg, Germany
[2]now with European XFEL GmbH, Holzkoppel 4, 22869 Schenefeld, Germany

Abstract

When integrating devices into a control system, the device applications usually contain a large fraction of error handling code. Many of these errors are run time errors which occur when communicating with the hardware, and usually have similar handling strategies. Therefore we extended ChimeraTK, a software toolkit for the development of control applications in various control system frameworks, such that the repetition of error handling code in each application can be avoided. ChimeraTK now also features automatic error reporting, recovery from device errors, and proper device initialisation after malfunctioning and at application start.

GOALS AND REQUIREMENTS

If the business logic of a control application is intertwined with code for device opening and communication error handling, it becomes hard to read. Hence, this should be avoided. In an ideal case, the application programmer does not have to write any device handling code, and a framework takes care of all the necessary actions, reports faults to the control system and handles the device recovery. For this to work with many different kinds of devices, a sufficient level of abstraction is necessary.

- The framework has to provide a common API for all devices.
- The framework has to guarantee that the user code can always read and write all its variables, and therefore needs to separate read/write operations in the user code from the actual hardware access.

These are the two key places where the framework has to have an appropriate interface. The second point is a consequence of the fact that the framework is taking care of connecting to a device, initialising the device, reporting the connection status to the control system and propagating information about faulty connections.

THE ChimeraTK FRAMEWORK

ChimeraTK, the ***C**ontrol system and **H**ardware **I**nterface with **M**apped and **E**xtensible **R**egister-based device **A**bstraction **T**ool **K**it*, is a framework for writing control applications. [1] An overview is shown in Fig. 1. ChimeraTK consists of three main components:

- The DeviceAccess library provides a common interface to different device types by introducing a backend plugin mechanism.
- The ControlSystemAdapter allows to integrate into various control system middlewares as a native application. [2]
- The ApplicationCore library connects many application modules, which make up the business logic, with the devices and the control system.

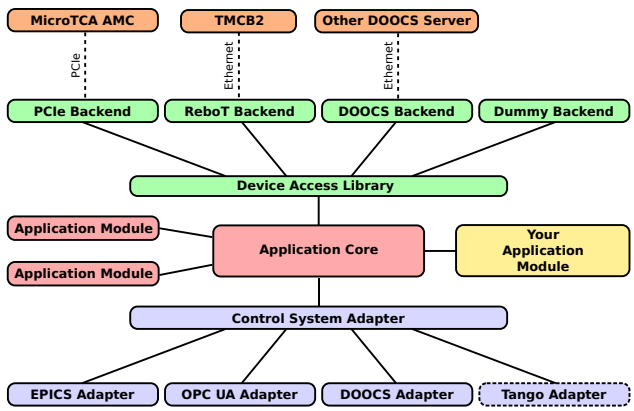


Figure 1: Overview of the ChimeraTK framework. All components connected with solid lines are part of the same executable.

All these components are part of the same executable. The C++-code written by the application programmer consists of application modules for ApplicationCore. ApplicationCore is interfacing to DeviceAccess and the ControlSystemAdapter internally. The user never directly interacts with these libraries on the C++ level. Which devices to use and the parameters for the control system integration are loaded from configuration files at application start.

The DeviceAccess Library

The DeviceAccess library is the interface to various devices. The device usually is a hardware component which is being integrated into the control system, but it can also

* martin.killenberg@desy.de

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV012

be another control system application. DeviceAccess comes with an extensible backend interface. Support for PCIexpress communication and a suite of dummies for unit testing are built into the base library. Backends for DOOCS [3,4], OPC UA [5,6] and Modbus [7,8] are available as runtime-loadable plugins. A backend for EPICS [9] is currently under construction. [10] The plugin mechanism allows to easily add support for further devices and protocols.

An important abstraction step in DeviceAccess is the introduction of so called register accessors. These are objects representing a device register in the application. It contains a buffer with the data content, and read() and write() functions to synchronise the buffer and the device. Each of those registers is identified by a name. For numerically addressed protocols like PCIexpress or Modbus, a register name mapping is built into the library.

The ApplicationCore Library

The main idea of ApplicationCore is to build the application from small, self contained application modules (see Fig. 2). Each module has a set of input and output variables and its own processing thread. All inputs and outputs are identified by a name, which associates it with a process variable that is part of the variable household of the application. ApplicationCore automatically connects the inputs and outputs of all components (application modules, device modules and the control system module) which access the same process variable. The code inside a module does not care where the data is coming from and where it is going to. Each module is completely self contained.

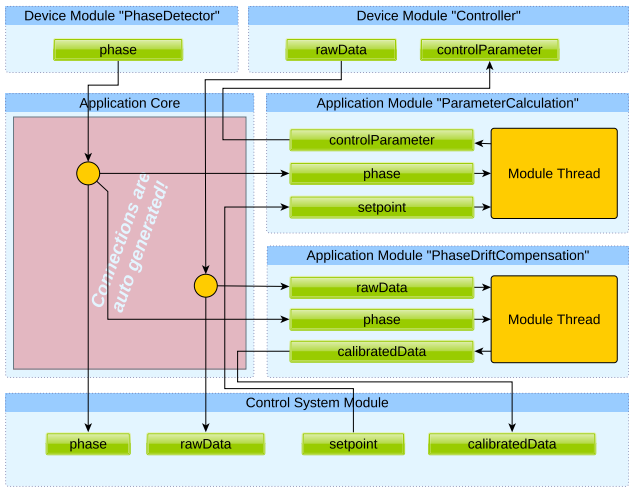


Figure 2: Applications written with ApplicationCore consist of self-contained application modules. Each module has its own thread and interfaces to the rest of the application via input and output variables. The connections of those variables to the device modules and the control system module are automatically generated by ApplicationCore.

Under the hood the variables of the different modules are connected via lock-free queues, which makes ApplicationCore a multi-threading library with very efficient, modern inter-thread communication. However, this is completely transparent to the user as there are no locks involved.

In ApplicationCore there are two kinds of inputs: Push-type inputs and poll-type inputs. Read operations of poll-type inputs return immediately and the variable contains the latest value. Read operations of push-type inputs block if no data is written. Once data is available, the reading thread is woken up. Typically each application module has as least one push-type input which the module thread is waiting for. Like this, the modules automatically synchronise with each other, without the need for locks or sleeps.

The ControlSystemAdapter Library

While the DeviceAccess library is a client interface to the devices, the ControlSystemAdapter turns the application into a server which introduces the published process variables to the control system. It consists of the ControlSystemAdapter base library and a concrete implementation for a control system middleware. ApplicationCore is using the interface of the base library and is completely independent of the actual adapter implementation. The adapter implementation is linked as separate library.

For a proper integration into the control system of a facility, each adapter implementation comes with a mapping layer which allows to change the names of variables to match the facility's naming scheme or create special data types only available for the particular middleware protocol.

Currently ControlSystemAdapter implementations are available for DOOCS [11], EPICS 3 [12, 13] and OPC UA [14].

EXCEPTION HANDLING AND DEVICE INITIALISATION

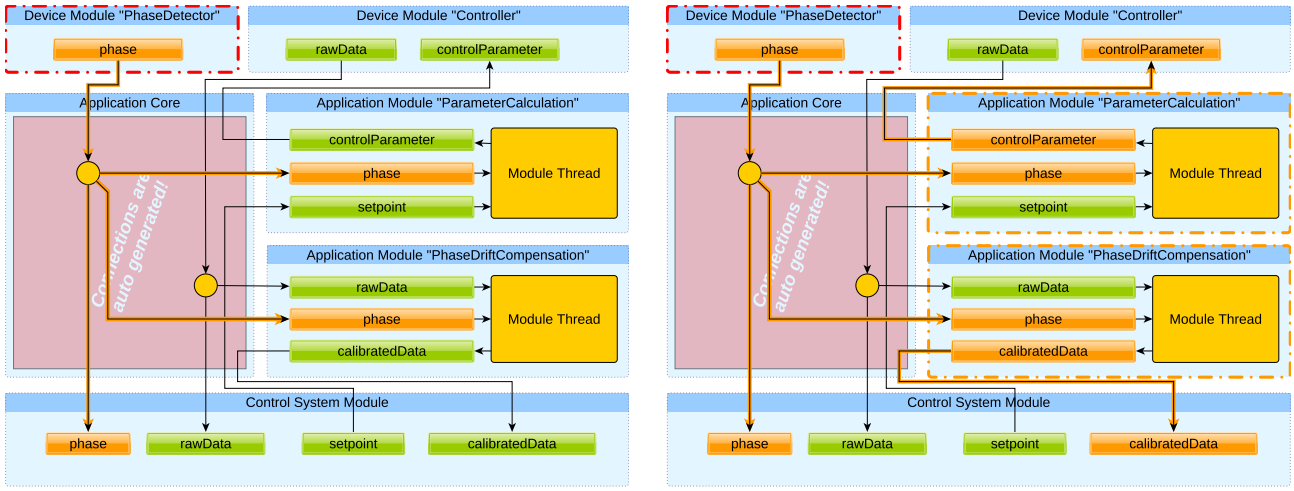
Handling Faults

Once a problem, usually a communication error, has been detected in DeviceAccess, this is internally reported to the backend. The backend informs all register accessors about the fault. From this point on, all read() operations to poll type accessors and all write() operations will throw a `runtime_error` exception when being called. All push-type accessors will send exactly one `runtime_error` exception through the queue (instead of data), and then no further data is sent until the next value has been received after device recovery (see discussion in the ApplicationCore section).

Each device backend implementation is responsible for re-establishing the connection when its open() function is called again. Usually this means trying to re-connect via whatever protocol is used to communicate with the device. Like this, no special, device-dependent code is needed on the higher levels.

In ApplicationCore, the exceptions from the accessors are caught and handled. As the main point of the exception handling scheme is to keep the user code free from device handling code, the user code in the application modules must

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV012



(a) The device module "PhaseDetector" has seen a runtime error. All of its outputs are flagged with data validity `faulty` (orange).

(b) All application modules which have at least one input flagged as `faulty` automatically switch the data validity of all their outputs to `faulty`.

Figure 3: Propagation of the data validity flag in ApplicationCore.

not see these exception. The exception is reported to the device module, which in the background is trying to recover by periodically calling open() on the backend. At the same time the device status reported to the control system changes to *not functional*.

Without further actions, the application modules with the user code would not be aware that the device has a problem and that data is not updating any more. For instance if there are poll-type inputs for calibration parameters, which are read in addition to the push-type main data, the application module would keep processing data without noticing that the calibration parameter might be outdated. That's why a mechanism has been introduced to automatically propagate a data validity flag in ApplicationCore.

Whenever an exception is caught in a process variable, the last good data value is re-sent with the data validity set to `faulty` (Fig. 3a). For push-type data this means, that the receiving module will be triggered exactly once. For poll-type data, the same value is read over and over again, but being flagged as `faulty`.

Application modules are small, self contained code blocks where all outputs logically depend on all inputs. This allows for an automatic propagation of the data validity flag: As soon as one input is flagged as `faulty`, all outputs of the module are also flagged as `faulty`. Like this, the flag is propagated for all data that is calculated from `faulty` inputs (Fig. 3b). Notice that in Fig. 3b the `rawData` from the "Controller" device is not marked as faulty, although one of the inputs of the device (`controlParameter`) is invalid. For device modules (and also the control system module) there is no automatic data validity propagation because they are not small, self contained units where all outputs logically depend on all inputs.

This mechanism covers the two possible scenarios: Modules which use the faulty data as their main data, which means they read them as push-type input to synchronise their execution to it, are processing the data exactly once to propagate the data validity flag. Modules which use the faulty data as poll inputs keep running, but flag their data as invalid. A typical use case is the `phase` parameter in Fig. 3. The `phase` is a typically slowly drifting parameter which is used to calibrate the `rawData`. It is a safe assumption that the calibration is still approximately correct, and the facility can keep running with it, even if there is a short communication problem with the phase detector. The calculated `calibratedData` is flagged as `faulty` to indicate that there is a problem. If the problem persists, the calibration will become less and less reliable. The decision whether operation can continue with `faulty` inputs obviously is a case by case decision. Although usually the code in application modules does not care about the data validity because it is propagated automatically, the flag for each input is available, such that critical decisions that depend on data being valid can be done.

Device Recovery

When a device can be re-opened successfully, the application and the device have to synchronise their states. The device might have re-booted, in which case many devices need to be initialised (reset-flags cleared, clocks set, dynamic ranges of ADCs and DACs adjusted etc.). For this to be automated, ApplicationCore provides the possibility to register a callback function. It is provided by the user code and executed immediately after the communication to the device is established. The parameters written in the initialisation phase often are implementation details of the firmware which should not be connected to the control system module. In this function the user code has direct access to the device and all of its registers, without adding process variables to the variable household of the application (in contrast to the

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV012

input and output variables of application modules, which are all visible in the control system due to the automatic connection code).

After the device initialisation function has been called, all output variables from the application to the device are written. This assures that the device and the application both have the same parameters everywhere. The device module remembers the last written value of each variable, so it can be restored in the device even if an output is not re-written by the application. This is especially important for parameters which are directly connected from the control system module to the device module. Obviously a user does not want to re-enter all parameters again in a panel after a device has come back online.

In a third step, all push-type data is fetched from the device and written to the application once. This data does not carry the data validity flag `faulty` any more, so the flag is cleared automatically in each module once all of its inputs are back to data validity `OK`.

Finally, after the device is initialised and all data to and from the device has been transferred once, the device status flag, which is reported to the control system, is set to `OK` as well. The application is back to normal operation.

Application Start

At application start, all devices start as *not functional*, and the device recovery procedure is launched. In the section about automatic fault handling it was described that an application module keeps running, even if a poll-type input is flagged with data validity `faulty`. This is working under the assumption that the last good value can be used to continue operation, even if the value is not being updated any more. At the first start of the application, when no communication to the device is established yet, this mechanism is not working because there is no previous good value.

To solve this issue, the concept of an *initial value* has been introduced in ApplicationCore. All application modules require that this first, valid value has been received by all inputs before operation is started. This is assured by ApplicationCore, which only starts the user-written main loop once all initial values are present. Each application module starts by performing its calculation with these initial values, and writing its outputs before doing the first (blocking) read on any input. Subsequent modules now also get their initial values, and the application takes up operation in the correct order. The control system adapter is getting the initial values from the persistency layer of the control system middleware, and is sending them once at application start.

If a device is not available at application start, this means that parts of the application will not start running. This does not pose a problem, it rather is a feature. Only those modules which cannot calculate valid data anyway are waiting, and once these data are available, the sequence of propagating them is automatically started. Even if all the application modules would be waiting for initial values, this does not mean that the application is non-responsive. The communication on the control system side is always working, and the status which device is faulty is shown there. There is nothing more the application can do in this situation, except for waiting for the device to become available.

The start sequence for opening a device is the same as for the recovery case: Once the communication is working, the initialisation callback function is executed, then all process variables from the application are written to the device, and finally the initial values are read for all push-type variables and propagated.

Although it should be avoided, it sometimes is inevitable to have circular dependencies between application modules, which would mean that modules are mutually waiting for the initial value from the other module before they can send their outputs. This circle needs to be resolved manually. One of the modules has to send a safe start value on its output before receiving the initial values. This cannot be done automatically because only a programmer who knows the application logic can decide where to break the circle and which "safe" initial value to send. As unintended circular dependencies prevent a proper application start and are difficult to debug, ApplicationCore comes with a built-in circular dependency detection. If an application module does not receive an initial value until a timeout, it will print which variables are affected to simplify the debugging process.

CONCLUSION

The ChimeraTK framework for the development of control system applications features the ApplicationCore library for the development of modular applications, which interacts seamlessly with the DeviceAccess library for hardware access and the ControlSystemAdapter for the integration into different control system environments. ChimeraTK has lately been extended with automatic device initialisation and exception handling. Special emphasis has been put in a clean initialisation and recovery procedure.

ApplicationCore based device applications are used to operate the low level radio frequency systems at several accelerators using different control system software (EuropanXFEL [15] and FLASH [16] at DESY, Hamburg, using DOOCS; ELBE [17] at HZDR, Dresden, using OPC UA [18]; TARLA [19] in Ankara using EPICS3 and several others). The improved device handling and automatic re-initialisation has reduced the time and manual steps which are required bring the system back online after a major fault or maintenance period. This results in reduced downtimes and a higher availability and reliability of the whole facility.

The ChimeraTK suite is open source software which is published under the GNU General Public License or the GNU Lesser General Public License (depending on the software component). It is available under [20].

REFERENCES

[1] M. Killenberg *et. al.*, "Abstracted Hardware and Middleware Access in Control Applications", in *Proc. ICALEPCS2017*,

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV012

Barcelona, Spain, 2017, paper TUPHA178. doi:10.18429/JACoW-ICALEPCS2017-TUPHA178

[2] M. Killenberg *et al.*, "Integrating control applications into different control systems", in *Proc. ICALEPCS2015*, Melbourne, Australia, 2015, paper TUD3O05. doi:10.18429/JACoW-ICALEPCS2015-TUD3O05

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

[4] DeviceAccess-DoocsBackend: DOOCS client for the ChimeraTK DeviceAccess library, https://github.com/ChimeraTK/DeviceAccess-DoocsBackend

[5] OPC Unified Architecture Specifications - Part 1: Overview and Concepts, https://reference.opcfoundation.org/v104/Core/docs/Part1/.

[6] DeviceAccess-OpcUaBackend: The OPC UA backend for DeviceAccess, https://github.com/ChimeraTK/OPC_UA_backend

[7] The Modbus Organisation, https://modbus.org/.

[8] DeviceAccess-ModbusBackend: Client supporting tcp and rtu communication with modbus devices, https://github.com/ChimeraTK/DeviceAccess-ModbusBackend

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

[10] DeviceAccess-EPICS-Backend: EPICS client backend for DeviceAccess, https://github.com/ChimeraTK/DeviceAccess-EpicsBackend

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

[12] ChimeraTK-ControlSystemAdapter-EPICS: Device support for integrating ChimeraTK-based devices into EPICS-based control-systems, https://github.com/aquenos/ChimeraTK-ControlSystemAdapter-EPICS

[13] ControlSystemAdapter-EPICS-IOC-Adapter: Implementation for the ControlSystemAdapter to create an EPICS IOC, https://github.com/ChimeraTK/ControlSystemAdapter-EPICS-IOC-Adapter

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

[15] M. Altarelli *et al.*, "XFEL: The European X-Ray Free-Electron Laser", DESY, Hamburg, Rep. DESY-2006-097, 2007. doi:10.3204/DESY_06-097

[16] C. Schmidt *et al.*, "Real time control of RF fields using a MicroTCA.4 based LLRF system at FLASH", in *19th IEEE Real-Time Conference*, Nara, Japan, 2014. doi:10.1109/RTC.2014.7097430

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

[18] R. Steinbrück *et. al.*, "Control System Integration of a µTCA.4 based digital LLRF using the ChimeraTK OPC UA Adapter", in *Proc. ICALEPCS2017*, Barcelona, Spain, 2017, paper THPHA166. doi:10.18429/JACoW-ICALEPCS2017-THPHA166

[19] A. Aksoy *et al.*, "TARLA: The First Facility of Tukrish Acceleretor Center (TAC)", in Proc. IPAC2017, Copenhagen, Denmark, 2017, paper WEPAB087. doi:10.18429/JACoW-IPAC2019-THPRB115

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