ChimeraTK.
What's new?

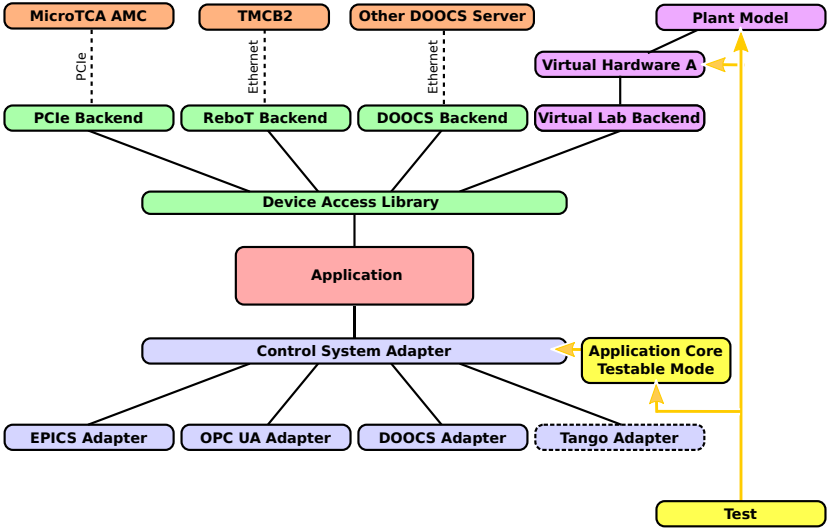
Martin Hierholzer, Martin Killenberg

26th September 2018

6th ARD ST3 workshop, HZDR, Dresden-Rossendorf

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

Overview


MicroTCA AMC
TMCB2
Other DOOCS Server
Plant Model
PCIe
Ethernet
Ethernet
Virtual Hardware A
PCIe Backend
RebоT Backend
DOOCS Backend
Virtual Lab Backend
Device Access Library
Application
Control System Adapter
Application Core
Testable Mode
EPICS Adapter
OPC UA Adapter
DOOCS Adapter
Tango Adapter
Test

DeviceAccess.

What is a Register in DeviceAccess?

A register

- contains data (numerical or a string)
- is identified by a name
- lives on a device
- has a length ($1 \mathrel{\hat{=}}$ scalar, $> 1 \mathrel{\hat{=}}$ array)

Simple DeviceAccess Example

```
#include <ChimeraTK/Device.h>
#include <iostream>

int main(){

  ChimeraTK::setDMapFilePath("devices.dmap");
  ChimeraTK::Device d;
  d.open("oven");

  auto heatingCurrent
    = d.getScalarRegisterAccessor<int>("heater/heatingCurrent");

  heatingCurrent.read();
  std::cout << "Heating current is "
            << heatingCurrent
            << std::endl;

  heatingCurrent += 3;
  heatingCurrent.write();
}
```

- RegisterAccessors
  - read() and write() functions to synchronise with Device
  - Behaves like a primitive data type (or vector of it) in most use cases
- RegisterPath
  - Hierarchical register name
  - "/" as hierarchy separator

DeviceAccess: Register Name Mapping

- DeviceAccess identifies registers by name
- Many devices use numerical addresses:
  - PCI Express identifies registers by address in a "Base Address Range" (BAR)
  - Dummies simulate devices in RAM

⇒ We need a mapping

Example map file

```
#name                       n_words address n_bytes BAR
heater.heatingCurrent             1    1024       4   2
heater.temperatureReadback        1    1028       4   2
heater.supplyVoltages             4    1032      16   2
```

- Map files are automatically created by the DESY (MSK) firmware framework
- Can easily be written manually

Deprecated API

DeviceAccess 1.0 gives warnings (compile time and run time) if you use the deprecated API.

Deprecated API	Current API
Namespace `mtcau4`	Namespace `ChimeraTK`
`RegisterAccessor`	`ScalarRegisterAccessor` `OneDRegisterAccessor`
`BufferingRegisterAccessor`	`ScalarRegisterAccessor` `OneDRegisterAccessor`
`RegisterMap`	`RegisterCatalogue`
`Device:` All functions to access deprecated classes	

`Device::readReg()`, `Device::writeReg()`
`Device::readArea()`, `Device::writeArea()`
`Device::read(numericalAddress)`
`Device::write(numericalAddress)`
	`Device::read(RegisterPath)`
	`Device::write(RegisterPath)`

- Please switch to the current API! Let us know if you have questions or problems.
- Deprecated functions will be removed soon.

Question

Each register can be accessed as scalar, 1D or 2D. And I have to chose the data type.
How do I know what to pick?

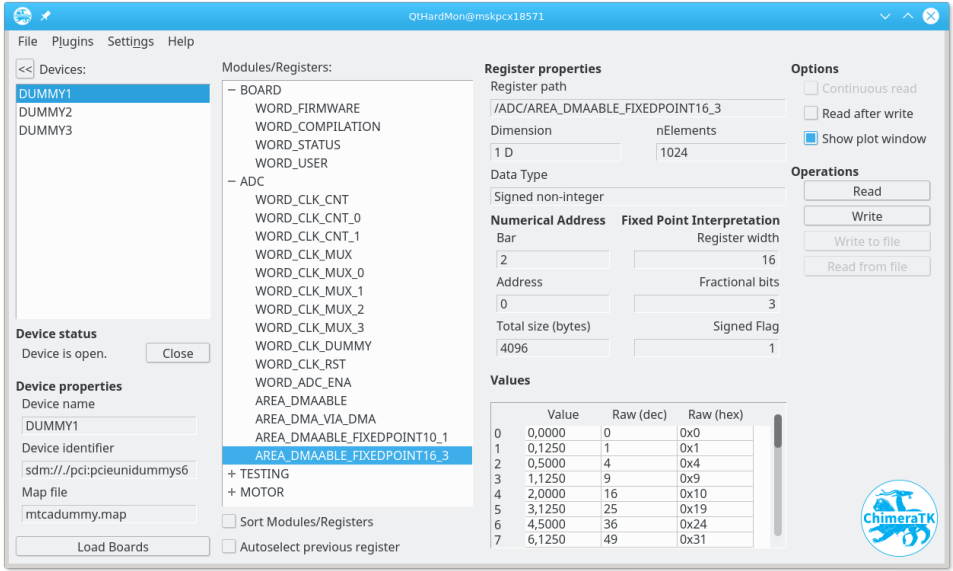
RegisterInfo

- `getRegisterName()`
- `getNumberOfDimensions()`
  - 0 (=scalar), 1, 2
- `getNumberOfElements()`
  per channel if 2D
- `getNumberOfChannels()`
- `getDataDescriptor()`

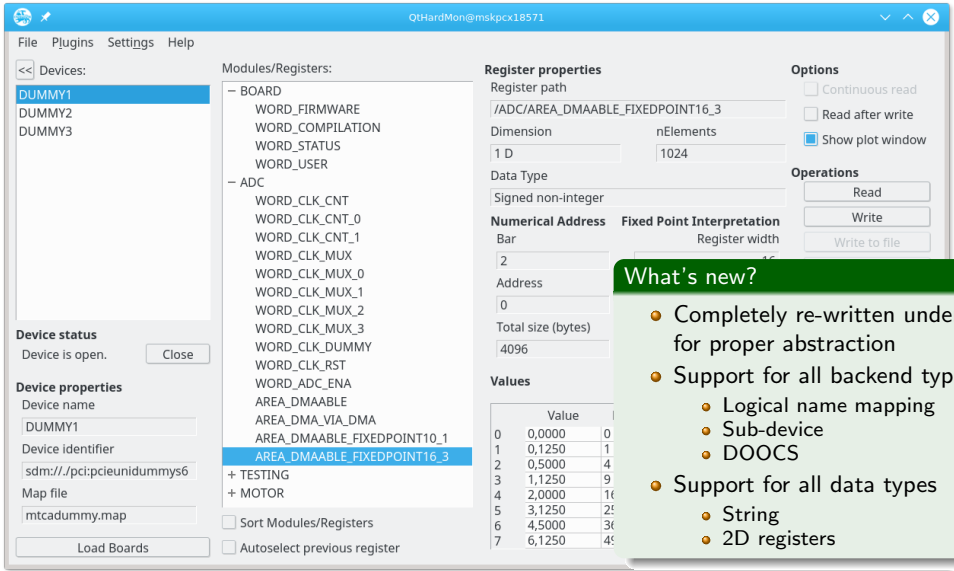
DataDescriptor

- `getFundamentalType()`
  - numeric, string, boolean, nodata, undefined
- `isIntegral()`
- `isSigned()`
- `nDigits()`
  Number of decimal digits for display purposes.
  E.g. int8_t is will return 4
  (-127..128 $\hat{=}$ 3 digits plus sign)
- `nFractionalDigits()`
  Number of digits after the decimal dot.

QtHardMon v01.00



QtHardMon v01.00



What's new?

- Completely re-written under the hood for proper abstraction
- Support for all backend types
  - Logical name mapping
  - Sub-device
  - DOOCS
- Support for all data types
  - String
  - 2D registers

DMap file entries

```
#alias      device_descriptor                          map_file
ADC_SLOT1   /dev/pcieunis1                             my_adc_firmware.map
ADC_SLOT2   sdm://./pci:pcieunis2                      my_adc_firmware.map

ADC_SLOT3 (pci:pcieunis3?map=my_adc_firmware.map)
```

ChimeraTK Device Descriptor (CDD)

(backend_type:address?key1=value1&key2=value2)

Syntax

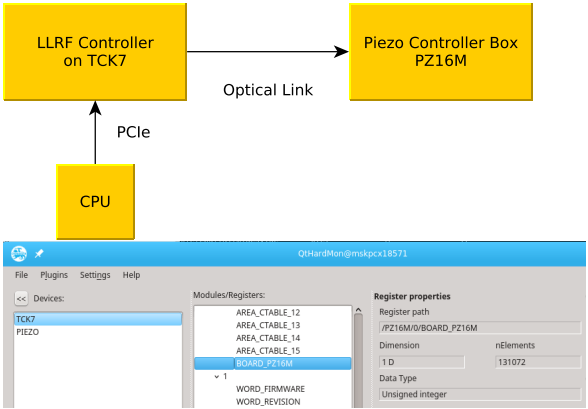
- Surrounded by parentheses – CDDs can be nested
- backend_type – Name of the backend, e.g. "pci", "dummy"
- address – Address of the device. The interpretation depends on the backend.
- keyX=valueX – List of key-value pairs. The interpretation depends on the backend.

Available Backends

pci	PCI-Express (pci:pciedevs1?map=my.map)
dummy	Simulate register space in RAM (dummy?map=my.map)
sharedMemoryDummy	Dummy with address space in shared memory (sharedMemoryDummy?map=my.map)
subdevice	Show part of address space as own logical device (subdevice:area,TARGET_DEVICE,TARGET_REGISTER?map=Subdevice.map))
logicalNameMap	Rename Logical Name Mapping (logicalNameMap?map=my.xlmap)
rebot	***Re**gister **b**ased **o**ver **T**CP*, lightweigt, TPC/IP based inhouse protocol (rebot?ip=192.168.0.1&port=5001&map=my.map)
doocs	DOOCS client interface[1][2] (doocs:facility/device/location)

[1] Can also read EPICS channels
[2] Separate library

Subdevice Backend



- Firmware maps the register space of a subdevice (PIEZO) into a 1D register of it's own address space (TCK7).
  (Usually offset address in a numerically addressed space).
- Both devices have firmwares which evolve separately.
- The Subdevice backend allows to access the subdevice through the parent device as a separete logical entity.

⇒ Separate map file to describe the subdevice.

```
#alias device_descriptor
TCK7   (pci:llrfutcs4?map=llrf_ctrl_tck7b_acc1_r2097.mapp)
PIEZO  (subdevice:area,TCK7,PZ16M.0.BOARD_PZ16M?map=piezo_pz16m_acc1_r2323.mapp)
```

Shared Memory Dummy

Simulate address space of device in shared memory

- Loads map file used for real device
- Device stays active as long at least one client has opened it
- Last closing client clears the shared memory

Use cases

- Run on your desktop PC without hardware
- Debugging if faulty software can damage the hardware
- Prepare test cases on the fly with QtHardMon

What can I do with Logical Name Mapping?

- Rename registers
- Add constant registers or dummy registers
- Extract channels from 2D registers and give it a name
- Extract scalars from 1D registers and give it a name
- Extract bits from a scalar register

Example: Abstract away cabling details

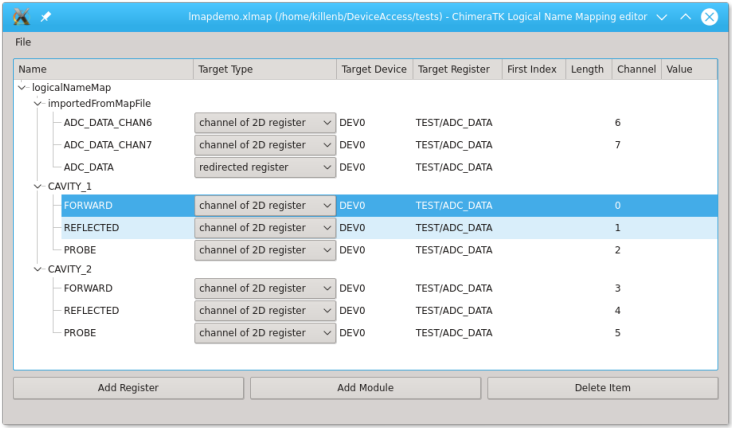
- Cavity with 3 signals: Forward, reflected, probe.
- 2D register with ADC data: 8 channels with 1024 samples each
- Cabling: Cavity 1 on channels 0..2, cavity 2 on channels 3..5

Logical name mapping:

adc_data[0]	→	cavity1/forward
adc_data[1]	→	cavity1/reflected
adc_data[2]	→	cavity1/probe
adc_data[3]	→	cavity2/forward
adc_data[4]	→	cavity2/reflected
adc_data[5]	→	cavity2/probe

You don't have to fiddle with channel numbers in you cavity module.
Use the logical names *forward, reflected, probe*.

LMAP Editor



- Import map file as starting point
- Modify the mapping
- Save and load logical name mapping

Tool under development.
Please give feedback or implement missing features.

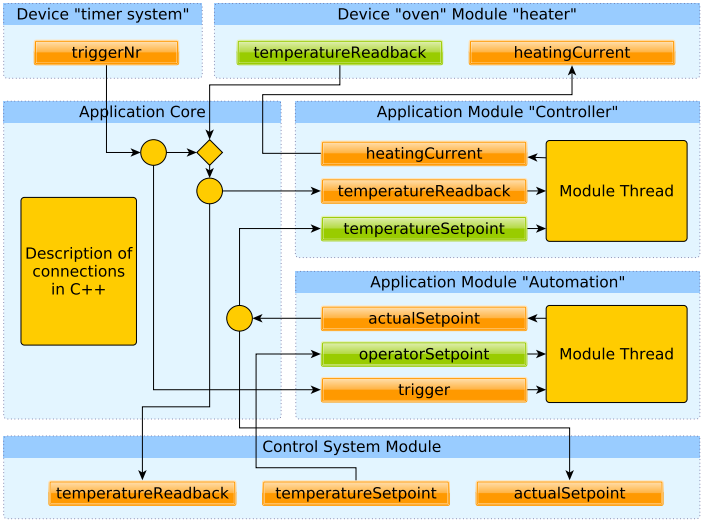
ApplicationCore.

Central Ideas of ApplicationCore

Goal: Provide framework for implementing applications which are "naturally" control-system independent

- Separate actual application code (algorithms etc.) from control system and device implementation details
- If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- Encourage modular applications ($\rightarrow$ conceptional abstraction)
- Clean and simple interface, avoid boiler plate code as much as C++11 allows
- Avoid the need for user callback functions (excessive use makes code unreadable)
- Allow publishing a device register into the control system with a single code line

ApplicationCore



Modules

- Input/output variables
- Application Modules
  - One thread per module
- Special modules
  - Device module
  - Control system module

Connection Code

- Connect application modules
- Triggering
  - Read multiple variables synchronously
  - Synchronise application modules to HW trigger

Details Matter

Threading system with efficient implementation:
Sender-receiver pairs based on lock-free queues

- Message driven design: sender pushes data into the queue
- No dead-locks possible
- Threads are not unintentionally blocked
- Wait for data without using CPU time

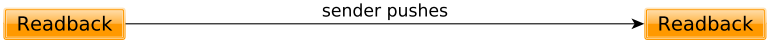
API fosters designs with small modules

- Clean design
- Improved performance on multi-core CPUs
- High scalability

Use C++11 to avoid boiler plate code where possible

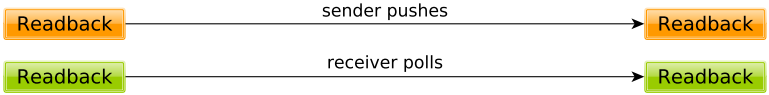
Excursus: Update modes

- Generalise the client/server concept



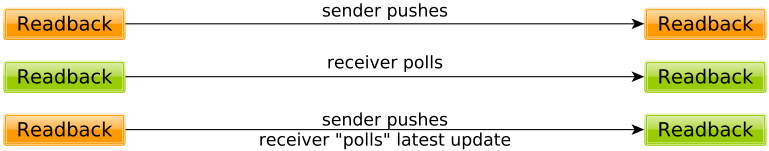
Excursus: Update modes

- Generalise the client/server concept



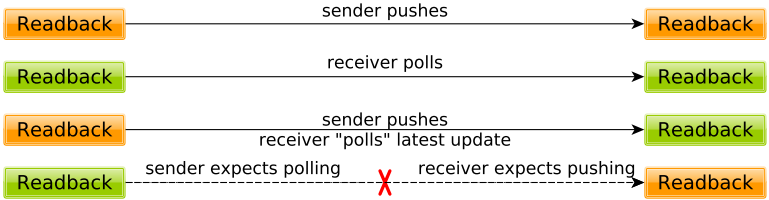
Excursus: Update modes

- Generalise the client/server concept



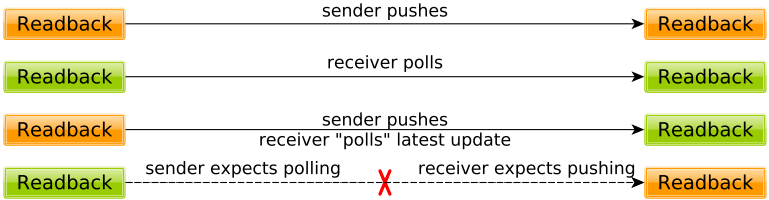
Excursus: Update modes

- Generalise the client/server concept

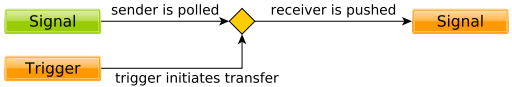


Excursus: Update modes

- Generalise the client/server concept

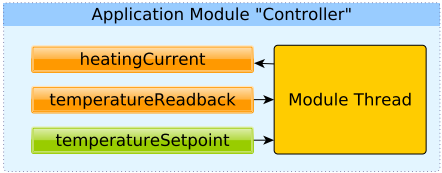


- A trigger makes it possible:



A simple ApplicationModule

```
struct Controller : public ctk::ApplicationModule {
    using ctk::ApplicationModule::ApplicationModule;
    ctk::ScalarPollInput<double> sp{this, "sp", "degC", "Description", {"CS"}};
    ctk::ScalarPushInput<double> rb{this, "rb", "degC", "...", {"DEV", "CS"}};
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "...", {"DEV"}};

    void mainLoop() {
      const double gain = 100.0;
      while(true) {
        rb.read();    // waits until rb updated
        sp.read();    // just get latest sp value
        cur = gain * (sp - rb);
        cur.write();
      }
    }
};
```

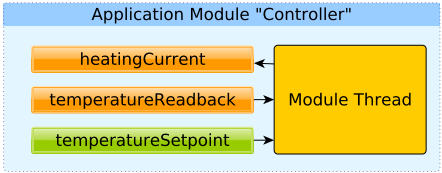


A simple ApplicationModule

Rule of thumb

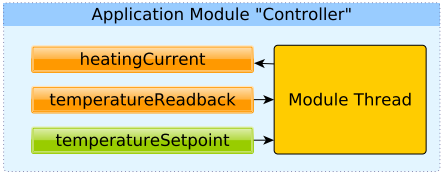
- Data inputs are made push-type
- Parameters (e.g. from panel) are poll-type

```
stru
    ctk::ScalarPollInput<double> sp{this, "sp", "degC", "Description", {"CS"}};
    ctk::ScalarPushInput<double> rb{this, "rb", "degC", "...", {"DEV", "CS"}};
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "...", {"DEV"}};

    void mainLoop() {
      const double gain = 100.0;
      while(true) {
        rb.read();    // waits until rb updated
        sp.read();    // just get latest sp value
        cur = gain * (sp - rb);
        cur.write();
      }
    }
};
```



A simple ApplicationModule

```
struct Controller : public ctk::ApplicationModule {
    using ctk::ApplicationModule::ApplicationModule;
    ctk::ScalarPollInput<double> sp{this, "sp", "degC", "Description", {"CS"}};
    ctk::ScalarPushInput<double> rb{this, "rb", "degC", "...", {"DEV", "CS"}};
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "...", {"DEV"}};

    void mainLoop() {
      const double gain = 100.0;
      while(true) {
        readAll();    // waits until rb updated,
                      // then reads sp
        cur = gain * (sp - rb);
        writeAll();  // writes any outputs
      }
    }
};
```



Define the application

```
struct ExampleApp : public ctk::Application {
    ExampleApp() : Application("exampleApp") {}
    ~ExampleApp() { shutdown(); }

    Controller controller{this, "Controller", "The Controller"};

    ctk::PeriodicTrigger timer{this, "Timer", "Periodic timer for the controller",
                               1000};

    ctk::DeviceModule heater{"oven","heater"};
    ctk::ControlSystemModule cs{"Bakery"};

    void defineConnections();
};
static ExampleApp theExampleApp;
```

Define the application

```
struct ExampleApp : public ctk::Application {
    ExampleApp() : Application("exampleApp") {}
    ~ExampleApp() { shutdown(); }

    Controller controller{this, "Controller", "The Controller"};

    ctk::PeriodicTrigger timer{this, "Timer", "Period
                               1000};

    ctk::DeviceModule heater{"ov
    ctk::ControlSystemMod
    void def
};
static ExampleApp theExampleApp;
```

No main() function shall be defined, it is coming from the adapter!

Connect the module with device and control system

```
void ExampleApp::defineConnections() {
    ChimeraTK::setDMapFilePath("example2.dmap");

    controller.findTag("DEV").connectTo(heater, timer.tick);
    controller.findTag("CS").connectTo(cs);
}
```

- Trigger is needed, since devices usually provide poll-type variables for reading
- Avoid use of polling loop: not testable, no clean termination
- Either use external trigger or the PeriodicTrigger module

DOOCS Adapter

- Application becomes a native DOOCS server
- Used (mainly) at DESY and European XFEL

OPC UA Adapter

- Application becomes an OPC UA server
- Can be integrated into WinCC
- Used at ELBE (HZDR)

EPICS3 Adapters

DeviceSupport Adapter

- Integrate as a device into your own IOC, together with other devices

\+ Maximum flexibility in Epics

− You have to compile your own IOC

IOC Adapter (new)

- Complete IOC installed as a debian package

\+ Ready-made IOC for each device

− Less flexibe

- Will be used at TARLA (Ankara) and Flute (KIT)

Outlook: Further development of ChimeraTK.

New backends and features in existing backends

- New backends: OPC UA client; native EPICS client
- Logical name mapping features:
  - Use parameters defined in ChimeraTK device descriptor (i.e. DMAP file) inside the map file, e.g. to reuse the same map file for multiple target devices
  - Support more data types for constants and variables (i.e. dummy registers)
- Subdevice backend features:
  - Unfold address space of devices with different layout in the mother device, e.g. two scalar registers (address and value)
- DOOCS backend:
  - Support for more DOOCS types including aggregated data types
- PCI express backend:
  - Support for interrupts
  - Support for floating point values

Introduce new supported data types in all ChimeraTK libraries

Currently supported data types: (u)int8, (u)int16, (u)int32, (u)int64, float, double, std::string

- bool
  - Often provided by hardware, e.g. status bits
  - Supported by many control systems, sometimes improved user experience compared to integer with value 0 and 1

Introduce new supported data types in all ChimeraTK libraries

Currently supported data types: (u)int8, (u)int16, (u)int32, (u)int64, float, double, std::string

- bool
  - Often provided by hardware, e.g. status bits
  - Supported by many control systems, sometimes improved user experience compared to integer with value 0 and 1
- void
  - No data is transported
  - Only useful in combination with push-type variables
  - Represents an interrupt or event or trigger
  - At least EPICS and DOOCS support something similar, exact representation should be discussed
  - Also internally useful in applications to efficiently distribute events/triggers

ControlSystemAdapter and ApplicationCore: bidirectional variables

Current situation: all variables in ApplicationCore are unidirectional

- Originally not implemented, since the concept is difficult and dangerous
- Variables should never really be bidirectional, that would lead to race conditions and infinite value oscillations
- Important and valid use cases:
  - Correction of an out-of-range value
  - Automation which runs on user request to determine a value which otherwise can be changed by the user (e.g. a calibration value)
  - ...

ApplicationCore: exception handling

Current situation: Exceptions cannot properly be handled in ApplicationCore - they are often thrown in a non-user thread and just will terminate the application.

- logic_error exceptions point to a programming or configuration issue and usually occur directly after starting the application. They should terminate the application.
- runtime_error exceptions can occur any time and should be properly handled without stopping the application
- Handling should be done per device
- Exceptions should be caught automatically by ApplicationCore
- Error status can be published to the control system (status flag + message string of exception per device)
- Each ApplicationModule using the faulty device will be automatically paused until the device is back online
- Maybe: add per-variable flag showing which parts are offline (DOOCS supports this, what about other control systems?)

ApplicationCore: direct connections between device and control system

Current situation: ApplicationCore doesn't know which data type and direction to use, so when publishing device registers directly to the control system, this information has to be added

- ApplicationCore should guess the data type based on the information in the catalogue.
- Use the smallest possible data type fitting the data
- Direction can be derived from read/write flags:
  - Read-only registers will be device-to-controlsystem
  - Write-only registers (rare) will be controlsystem-to-device
  - Read-write registers will also be controlsystem-to-device, since they are usually never changed by the device (only readback of the current value possible)
- Transfer mode will default to poll-type, i.e. `wait_for_new_data flag` is not specified
- Exceptions from these automatic rules can still be made by providing the information per-variable (or maybe per `connectTo()`?)