ChimeraTK ApplicationCore.

Martin Hierholzer, Martin Killenberg

4. December 2018

7th MicroTCA Workshop for Industry and Research
DESY, Hamburg

How to write a control system application?

- Goal: integrate device into a control system

How to write a control system application?

- Goal: integrate device into a control system
- Often required: complex algorithms in software, e.g.:
  - control loops
  - model based computations
  - automation routines

How to write a control system application?

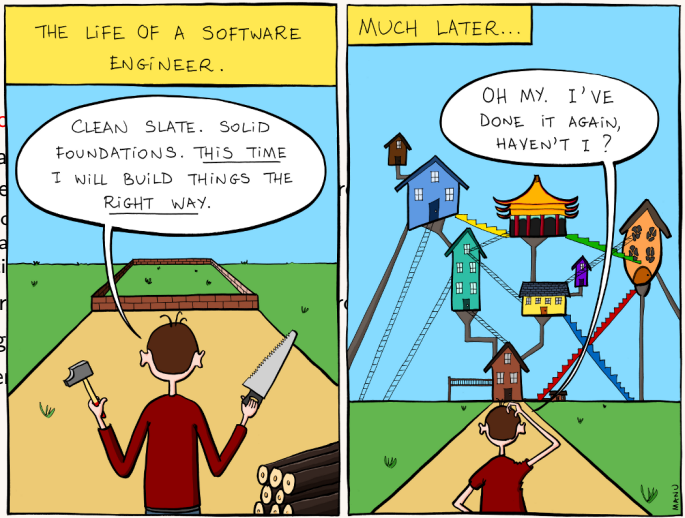
- Goal: integrate device into a control system
- Often required: complex algorithms in software, e.g.:
  - control loops
  - model based computations
  - automation routines
- Ideally: abstract away hardware details and provide high-level user interface

How to write a control system application?

- Goal: integrate device into a control system
- Often required: complex algorithms in software, e.g.:
  - control loops
  - model based computations
  - automation routines
- Ideally: abstract away hardware details and provide high-level user interface
- Production-grade code quality required
- Keep long term maintenance in mind

How to write a co

- Goal: integra
- Often require
  - control lo
  - model ba
  - automati
- Ideally: abstr
- Production-g
- Keep long te



by Manu Cornet (www.bonkersworld.net)

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files

Fully fledged server

- Usually C++ application based on control system middleware

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files
- ✓ Works well for simple projects without complex algorithms
- ✓ Fast updates e.g. when firmware changes

Fully fledged server

- Usually C++ application based on control system middleware

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files
- ✓ Works well for simple projects without complex algorithms
- ✓ Fast updates e.g. when firmware changes
- ✗ Hard to introduce real abstraction from hardware details
- ✗ Typically doesn't scale well with growing complexity
- ✗ Difficult to maintain

Fully fledged server

- Usually C++ application based on control system middleware

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files
- ✓ Works well for simple projects without complex algorithms
- ✓ Fast updates e.g. when firmware changes
- ✗ Hard to introduce real abstraction from hardware details
- ✗ Typically doesn't scale well with growing complexity
- ✗ Difficult to maintain

Fully fledged server

- Usually C++ application based on control system middleware
- ✓ Complex algorithms can be implemented more easily
- ✓ Optimal performance
- ✓ Full freedom to implement abstraction

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files
- ✓ Works well for simple projects without complex algorithms
- ✓ Fast updates e.g. when firmware changes
- ✗ Hard to introduce real abstraction from hardware details
- ✗ Typically doesn't scale well with growing complexity
- ✗ Difficult to maintain

Fully fledged server

- Usually C++ application based on control system middleware
- ✓ Complex algorithms can be implemented more easily
- ✓ Optimal performance
- ✓ Full freedom to implement abstraction
- Contineous integration tests possible, if framework supports it

Two approaches

Script-like approach

- ▸ Can be realised e.g. in EPICS records files
- ✓ Works well for simple projects without complex algorithms
- ✓ Fast updates e.g. when firmware changes
- ✗ Hard to introduce real abstraction from hardware details
- ✗ Typically doesn't scale well with growing complexity
- ✗ Difficult to maintain

Fully fledged server

- ▸ Usually C++ application based on control system middleware
- ✓ Complex algorithms can be implemented more easily
- ✓ Optimal performance
- ✓ Full freedom to implement abstraction
- ● Contineous integration tests possible, if framework supports it
- ✗ High initial implementation effort required
- ✗ Typically doesn't scale well for small complexity
- ✗ Not so flexible

Two approaches

Script-like approach

- Can be realised e.g. in EPICS records files
- ✓ Works v… complex
- ✓ Fast updates e.g. when firmware changes
- ✗ Hard to introduce real abstraction from hardware details
- ✗ Typically doesn't scale well with growing complexity
- ✗ Difficult to maintain

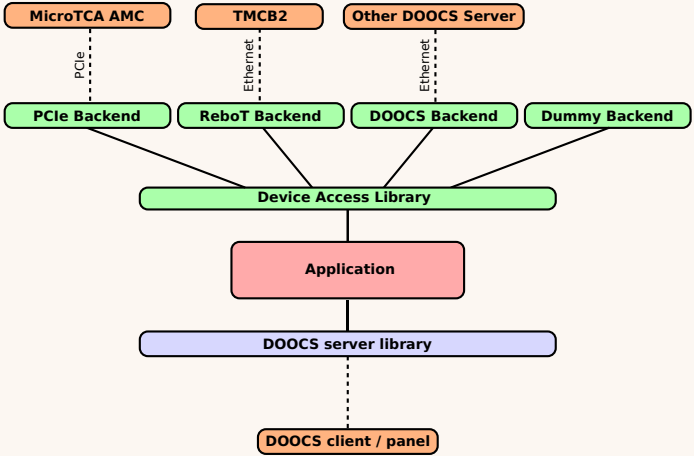
In this tutorial...

Can we build a fully fledged server without most of its downsides?

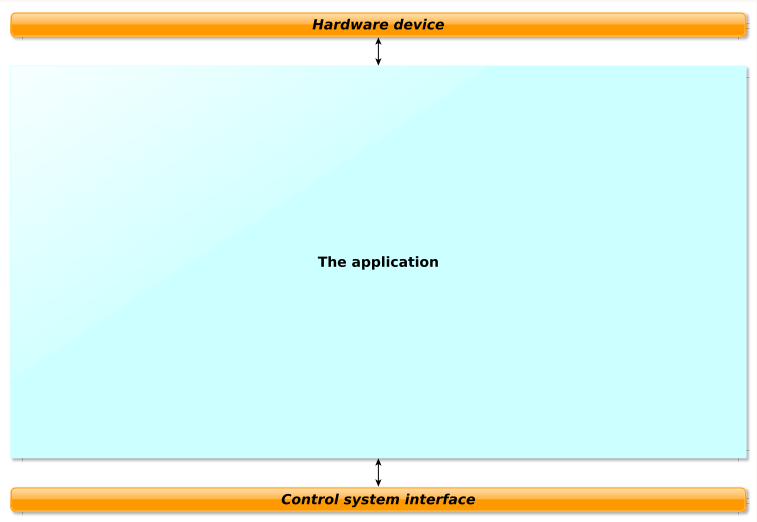
Fully fledged server

- Usually C++ application based on control system middleware
- …mplemented
- ✓ Full freedom to implement abstraction
- ● Contineous integration tests possible, if framework supports it
- ✗ High initial implementation effort required
- ✗ Typically doesn't scale well for small complexity
- ✗ Not so flexible

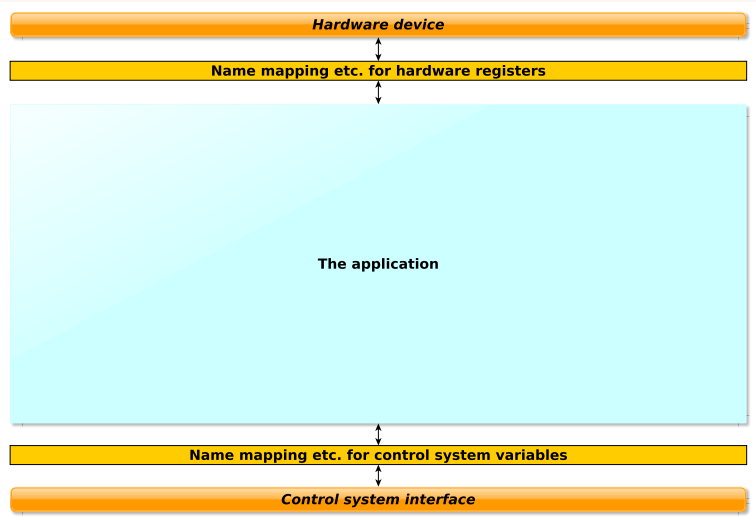
Design of a control application



Design of a control application



Design of a control application

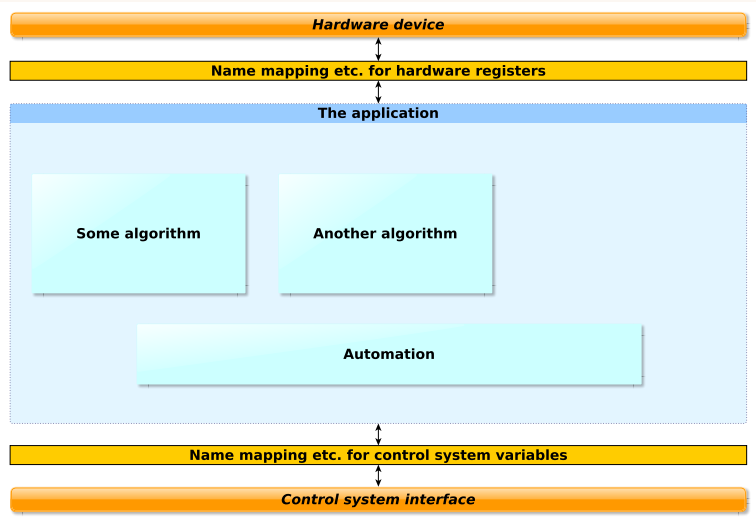


← logical name mapping

- Mapping layers give freedom to change parts independently
- ChimeraTK: both can be introduced later and thus are optional at first

← system integration

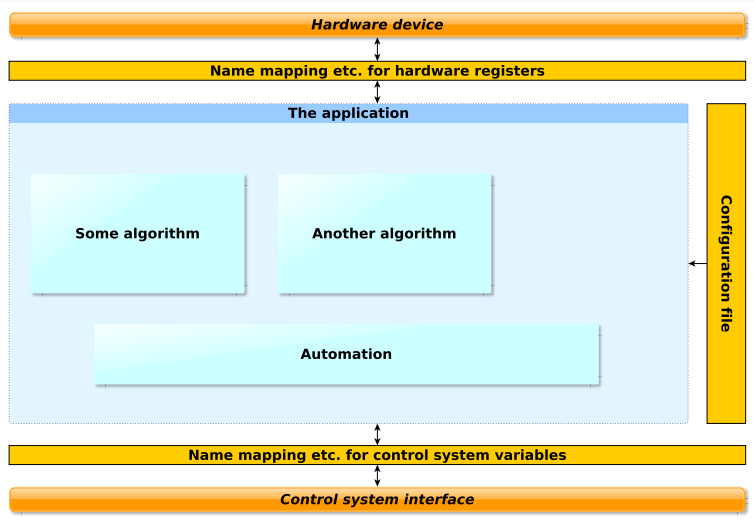
Design of a control application



Modularise the application

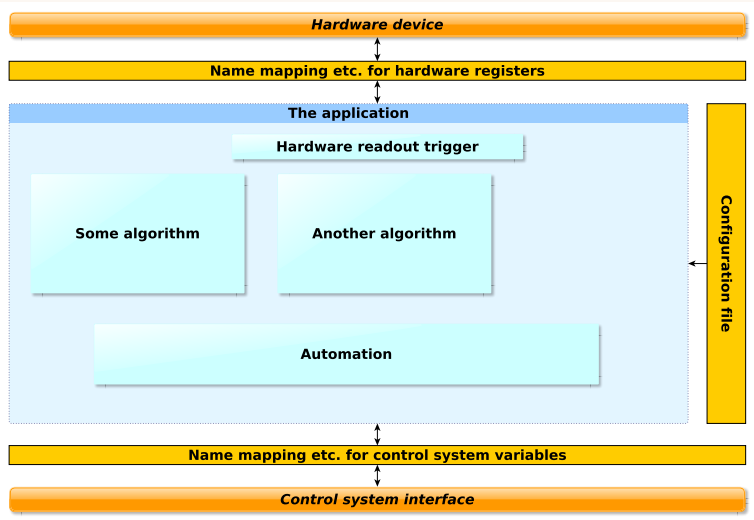
- Self-containment reduces complexity!
- Code easier to understand and to maintain
- Modules might be reused
- Flexibility to enabled/disable features where needed

Design of a control application



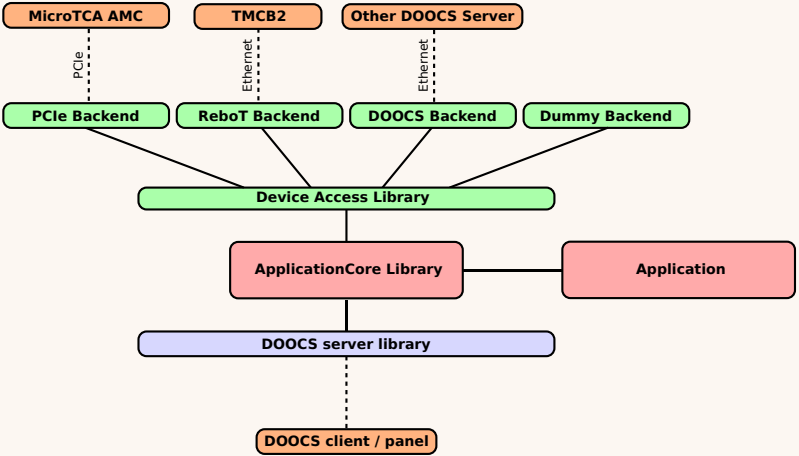
- Configuration files add flexibility:
- Same generic application might work in different places with different configuration

Design of a control application

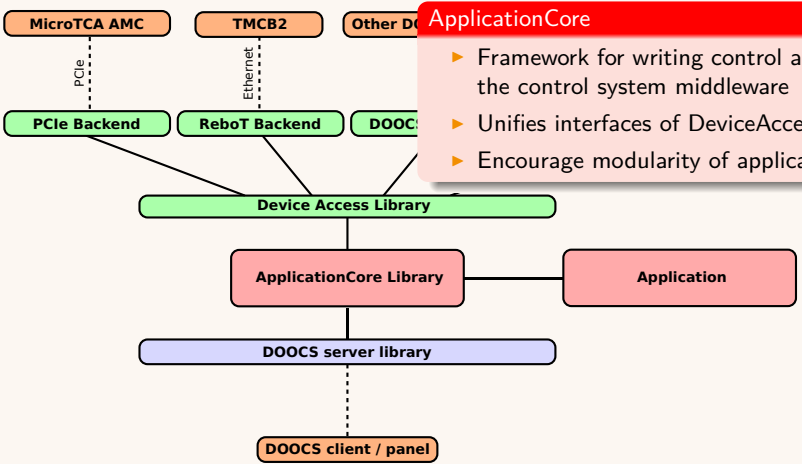


- Trigger all hardware readouts by a common trigger
- Data consistency
- Ideally already realised in hardware (but must still be handled in software properly)

Introducing ApplicationCore



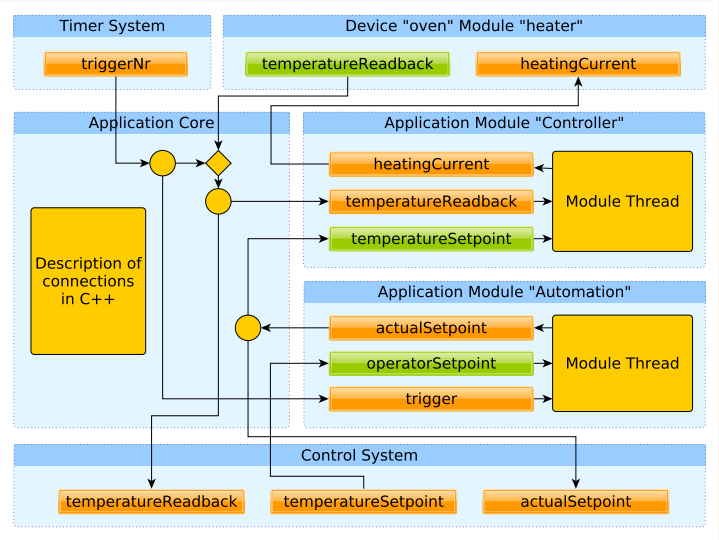
Introducing ApplicationCore



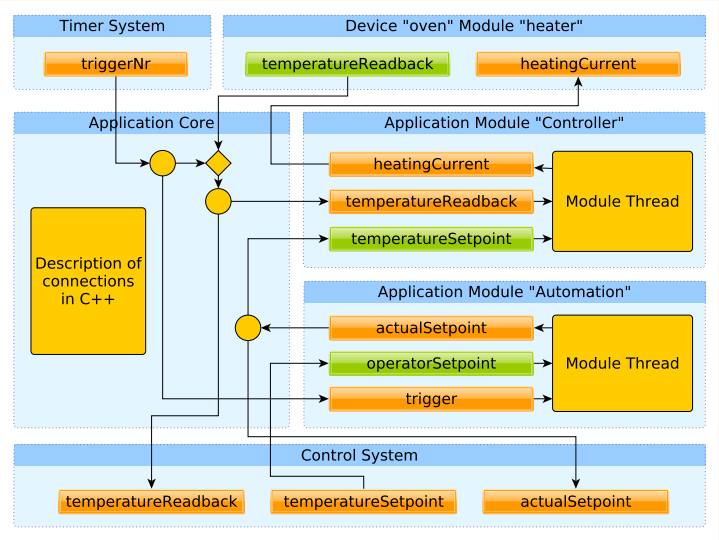
ApplicationCore

- Framework for writing control applications independent of the control system middleware
- Unifies interfaces of DeviceAccess and the control system
- Encourage modularity of applications

The structure of ApplicationCore

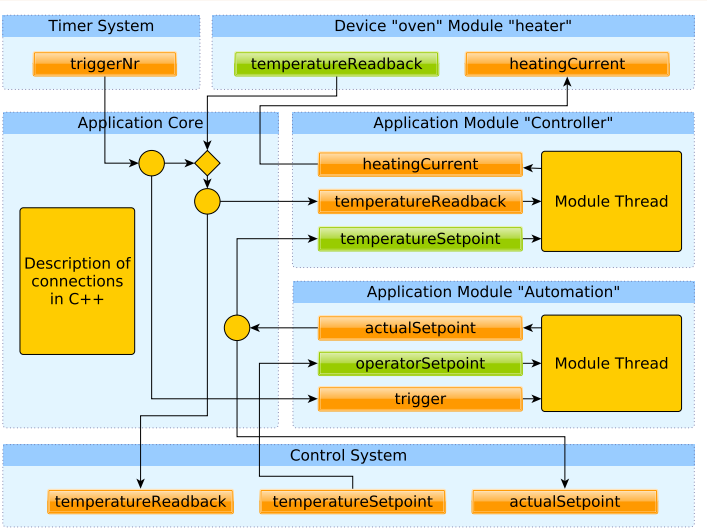


The structure of ApplicationCore



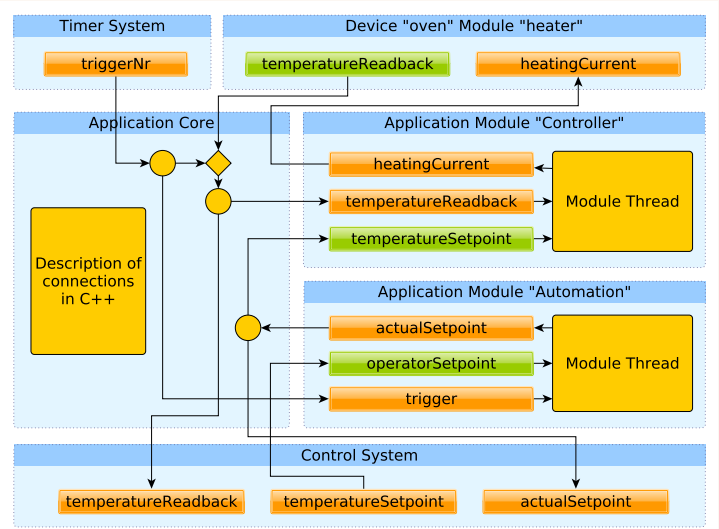
- Modules merely have inputs and outputs
- Implementations of algorithms do not need to know how its variables are connected

The structure of ApplicationCore



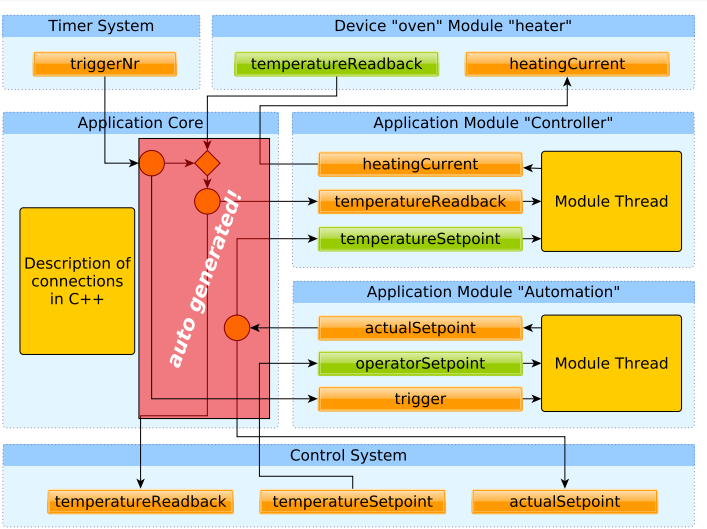
- Modules merely have inputs and outputs
- Implementations of algorithms do not need to know how its variables are connected
- Modern multithreading: lock-free communication between modules (“for free”)
- Perfect modularity, as modules are self-contained

The structure of ApplicationCore



- Modules merely have inputs and outputs
- Implementations of algorithms do not need to know how its variables are connected
- Modern multithreading: lock-free communication between modules (“for free”)
- Perfect modularity, as modules are self-contained
- Simpler code!

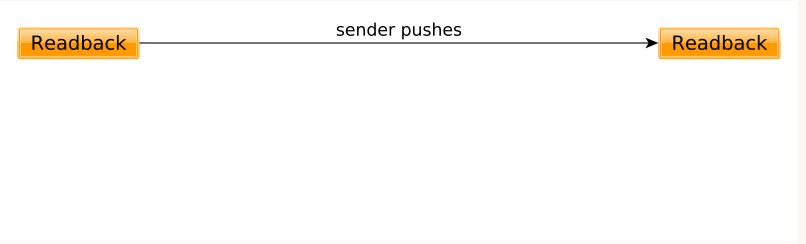
The structure of ApplicationCore



- Modules merely have inputs and outputs
- Implementations of algorithms do not need to know how its variables are connected
- Modern multithreading: lock-free communication between modules (“for free”)
- Perfect modularity, as modules are self-contained
- Simpler code!

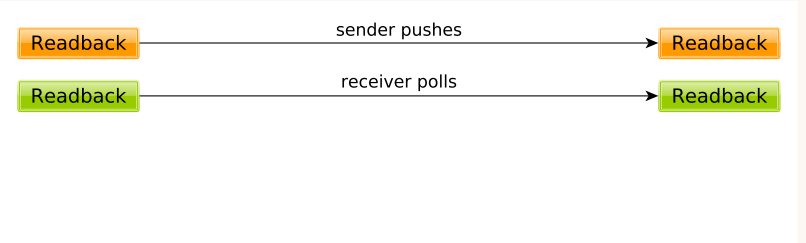
Excursus: Update modes

- Analog to the client/server concept:
  - Client/server is about who initiates a connection
  - Update mode is about who initiates a data transfer



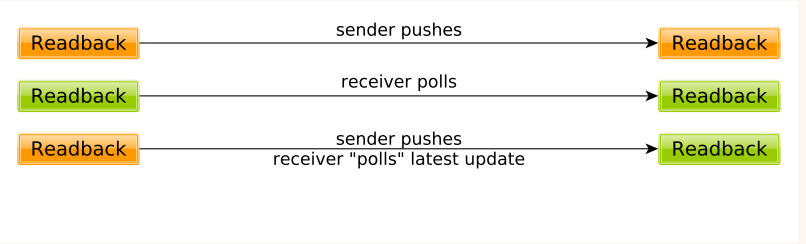
Excursus: Update modes

- Analog to the client/server concept:
  - Client/server is about who initiates a connection
  - Update mode is about who initiates a data transfer



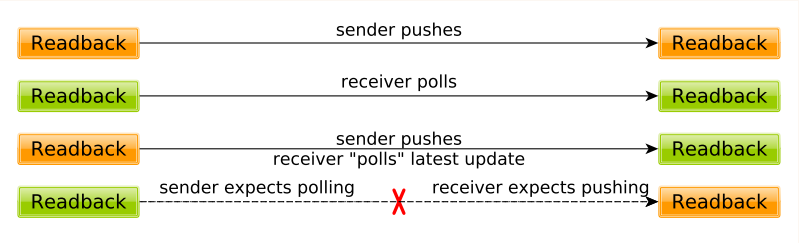
Excursus: Update modes

- Analog to the client/server concept:
  - Client/server is about who initiates a connection
  - Update mode is about who initiates a data transfer



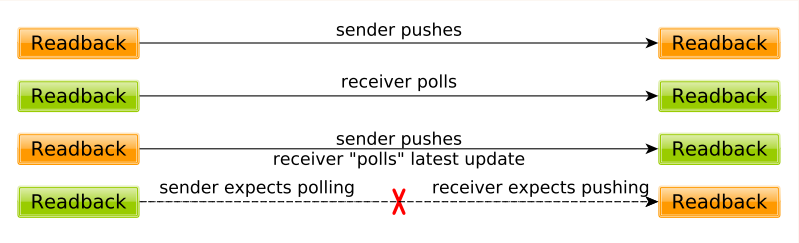
Excursus: Update modes

- Analog to the client/server concept:
  - Client/server is about who initiates a connection
  - Update mode is about who initiates a data transfer

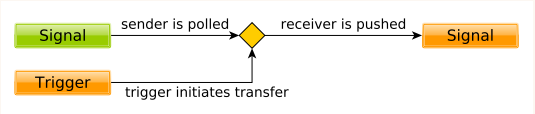


Excursus: Update modes

- Analog to the client/server concept:
  - Client/server is about who initiates a connection
  - Update mode is about who initiates a data transfer



- A trigger makes it possible:

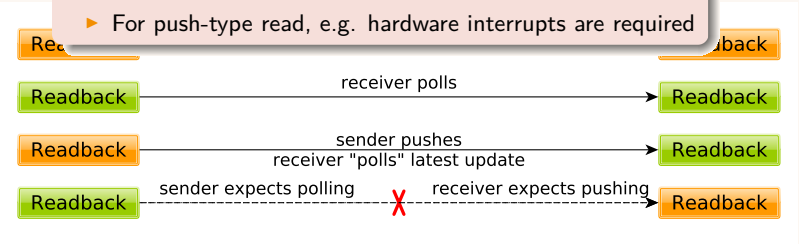


Excursus: Update modes

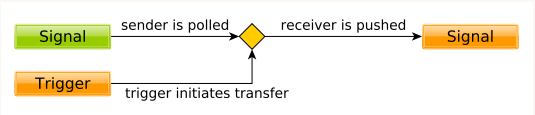
- Analog to the client/server concept:
  - Client/se
  - Update m

Typical devices (and most control system applications)

- A typical “passive” device has poll-type read variables and push-type write variables
- For push-type read, e.g. hardware interrupts are required



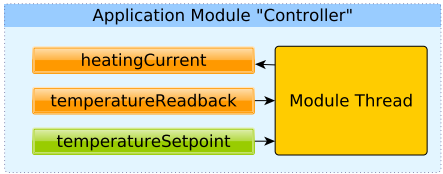
- A trigger makes it possible:



A simple ApplicationModule

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

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



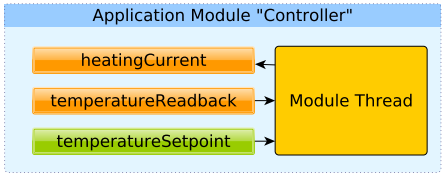
A simple ApplicationModule

Rule of thumb

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

```
struc
    us
    ctk::ScalarPollInput<double> sp{this, "temperatureSetpoint", "degC", "Description"};
    ctk::ScalarPushInput<double> rb{this, "temperatureReadback", "degC", "..."};
    ctk::ScalarOutput<double> cur{this, "heatungCurrent", "mA", "..."};

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

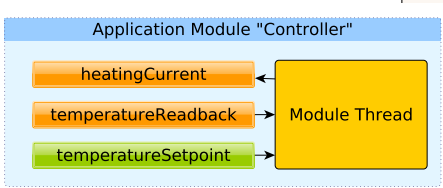


A simple ApplicationModule

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

    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
      }
    }
};
```

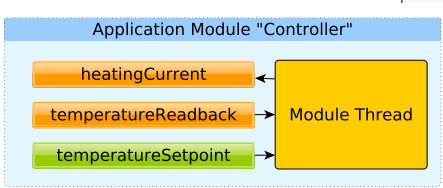


A simple ApplicationModule

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

    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
      }
    }
};
```



see example2/demoApp.cc in ApplicationCore source code on github (incl. following slides)

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 (1000ms period)", 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", "Periodic timer", 1000};

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

    void defineCo
};
static ExampleApp theExampleApp;
```

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

Connect the module with device and control system - Step I

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

    controller.connectTo(heater, timer.tick);    // use periodic timer as trigger for readout
    controller.connectTo(cs);
}
```

Something is missing...

- `A.connectTo(B)` will connect all variables in A with the variables of the same name in B
- `controller.sp` is a control system variable and does not exist in the hardware device

Connect the module with device and control system - Step I

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

    controller.connectTo(heater, timer.tick);    // use periodic timer as trigger for readout
    controller.connectTo(cs);
}
```

Something is missing...

- `A.connectTo(B)` will connect all variables in A with the variables of the same name in B
- `controller.sp` is a control system variable and does not exist in the hardware device
- Explicitly connecting each variable is possible, but annoying
- We need to filter variables based on additional information

Connect the module with device and control system - Step II

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

tags

Connect the module with device and control system - Step II

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

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

Tags

- Tag names can be arbitrarily chosen
- Any number of tags can be attached to each process variable
- Fancy searches possible as well...

Connect the module with device and control system - Step II

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

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

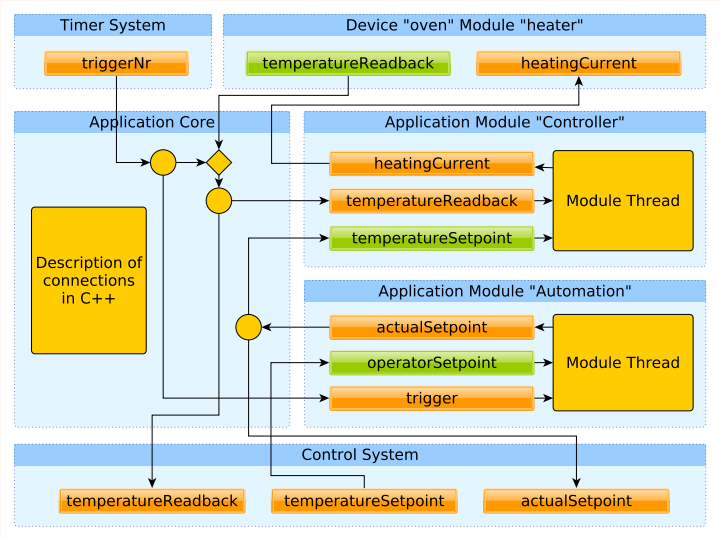
Tags

- Tag names can be arbitrarily chosen
- Any number of tags can be attached to each process variable
- Fancy searches possible as well...

That's it!

- At this point, the application is complete and can run!

Adding an automation module



Adding an automation module

```
struct Automation : public ctk::ApplicationModule {
    using ctk::ApplicationModule::ApplicationModule;
    ctk::ScalarPollInput<double> opSp{this, "operatorSetpoint", "degC", "...", {"CS"}};
    ctk::ScalarOutput<double> actSp{this, "temperatureSetpoint", "degC", "...", {"Controller"}};
    ctk::ScalarPushInput<uint64_t> trigger{this, "trigger", "", "..."};

    void mainLoop() {
      const double maxStep = 0.1;
      while(true) {
        readAll();     // waits until trigger received, then read opSp
        actSp += std::max(std::min(opSp - actSp, maxStep), -maxStep);    // ramp sp slowly
        writeAll();
      }
    }
};
```

see example2a/demoApp.cc in ApplicationCore source code

Adding an automation module

```
struct Automation : public ctk::ApplicationModule {
    using ctk::ApplicationModule::ApplicationModule;
    ctk::ScalarPollInput<double> opSp{this, "operatorSetpoint", "degC", "...", {"CS"}};
    ctk::ScalarOutput<double> actSp{this, "temperatureSetpoint", "degC", "...", {"Controller"}};
    ctk::ScalarPushInput<uint64_t> trigger{this, "trigger", "", "..."};

    void mainLoop() {
      const double maxStep = 0.1;
      while(true) {
        readAll();     // waits until trigger received, then read opSp
        actSp += std::max(std::min(opSp - actSp, maxStep), -maxStep);    // ramp sp slowly
        writeAll();
      }
    }
};
```

```
struct ExampleApp : public ctk::Application {
    [...]
    Controller controller{this, "Controller", "The Controller"};
    Automation automation{this, "Automation", "Slow setpoint ramping algorithm"};
    [...]

    void defineConnections();
};
```

unchanged!

added

unchanged!

Adding an automation module

```
struct Automation : public ctk::ApplicationModule {
    using ctk::ApplicationModule::ApplicationModule;
    ctk::ScalarPollInput<double> opSp{this, "operatorSetpoint", "degC", "...", {"CS"}};
    ctk::ScalarOutput<double> actSp{this, "temperatureSetpoint", "degC", "...", {"Controller"}};
    ctk::ScalarPushInput<uint64_t> trigger{this, "trigger", "", "..."};

    void mainLoop() {
      const double maxStep = 0.1;
      while(true) {
        readAll();     // waits until trigger received, then read opSp
        actSp += std::max(std::min(opSp - actSp, maxStep), -maxStep);    // ramp sp slowly
        writeAll();
      }
    }
};
```

```
void ExampleApp::defineConnections() {  // (setDMapFilePath omitted)
    automation.findTag("Controller").connectTo(controller);
    automation.findTag("CS").connectTo(cs);
    timer.tick >> automation.trigger;

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

added

unchanged!

Configuration

- Configuration can be used in `defineConnections()` for static configuration

```
struct ExampleApp : public ctk::Application {
    [...]
    ctk::ConfigReader config{this, "Configuration", "demoApp2a.xml"};
    Automation automation;
    [...]
};
```

```
<configuration>
  <variable name="enableAutomation" type="int32" value="1"/>
</configuration>
```

```
void ExampleApp::defineConnections() {
    [...]

    if(config.get<int>("enableAutomation")) {
      automation = Automation(this, "Automation", "Slow setpoint ramping algorithm");
      automation.findTag("Controller").connectTo(controller);
      automation.findTag("CS").connectTo(cs);
      timer.tick >> automation.trigger;
    }
    [...]
}
```

Configuration

- Configuration can be used in `defineConnections()` for static configuration
- It can also be connected as (never changing) variables to modules or the control system

```
struct ExampleApp : public ctk::Application {
    [...]
    ctk::ConfigReader config{this, "Configuration", "demoApp2a.xml"};
    Automation automation;
    [...]
};
```

```
<configuration>
  <variable name="enableAutomation" type="int32" value="1"/>
</configuration>
```

```
void ExampleApp::defineConnections() {
    [...]
    config.connectTo(cs);
    if(config.get<int>("enableAutomation")) {
      automation = Automation(this, "Automation", "Slow setpoint ramping algorithm");
      automation.findTag("Controller").connectTo(controller);
      automation.findTag("CS").connectTo(cs);
      timer.tick >> automation.trigger;
    }
    [...]
}
```

Hierarchies

- Current variables in the control system:
  - `operatorSetpoint` (writeable)
  - `temperatureSetpoint` (read only)
  - `temperatureReadback` (read only)
  - `enableAutomation` (read only)

Hierarchies

- Current variables in the control system:
  - `operatorSetpoint` (writeable)
  - `temperatureSetpoint` (read only)
  - `temperatureReadback` (read only)
  - `enableAutomation` (read only)
- Names are missing the context! Want to have:
  - `Automation/operatorSetpoint`
  - `Controller/temperatureSetpoint`
  - `Controller/temperatureReadback`
  - `Configuration/enableAutomation`

Hierarchies

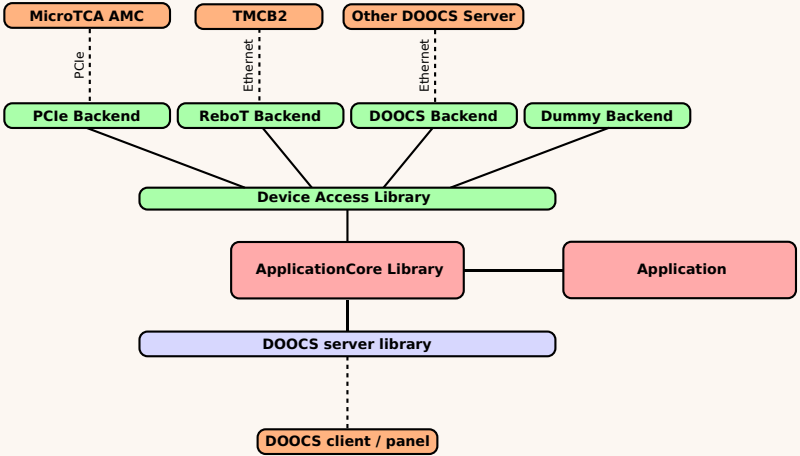
- Current variables in the control system:
  - `operatorSetpoint` (writeable)
  - `temperatureSetpoint` (read only)
  - `temperatureReadback` (read only)
  - `enableAutomation` (read only)
- Names are missing the context! Want to have:
  - `Automation/operatorSetpoint`
  - `Controller/temperatureSetpoint`
  - `Controller/temperatureReadback`
  - `Configuration/enableAutomation`
- In the example, we just need to connect to the control system differently:

```
findTag("CS").connectTo(cs);
```

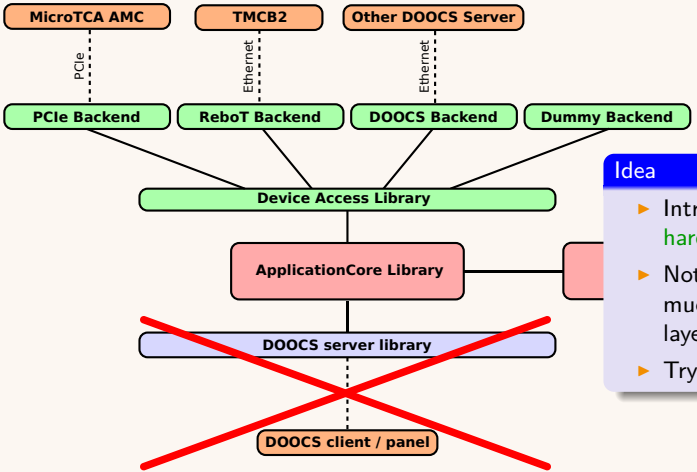
  This works on all modules in the application (and even saves two lines of code)!
- ApplicationCore allows to build arbitrary hierarchies:
  - VariableGroup
  - ModuleGroup

  Should be used to structure the application logically!

Control system integration



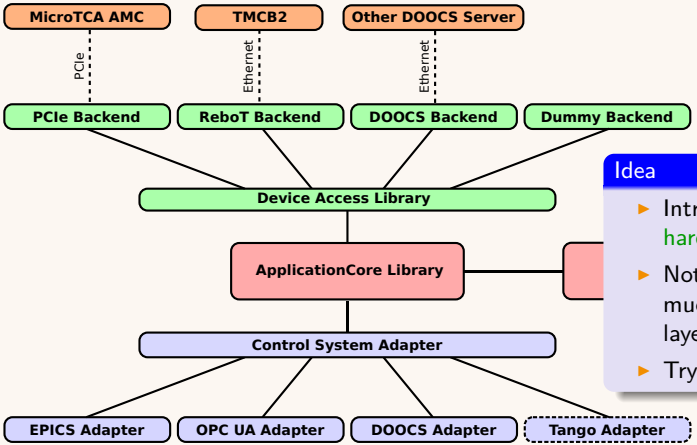
Control system integration



Idea

- Introduce similar abstraction as for the hardware access
- Not trivial - control system middleware much more than just a communication layer
- Try **not** to create a new control system!

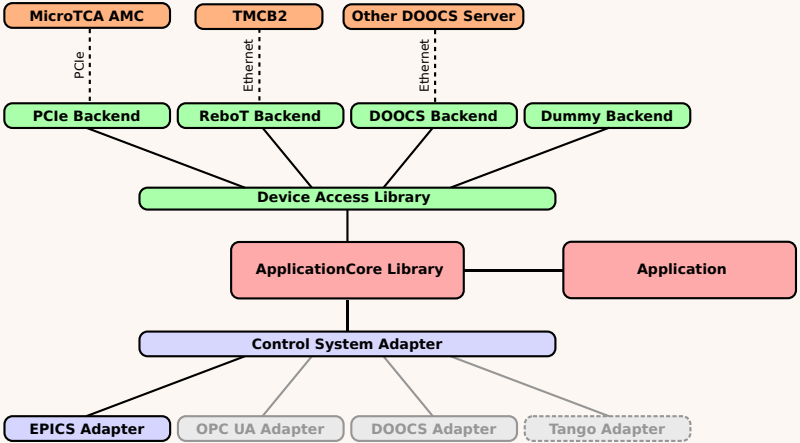
Control system integration



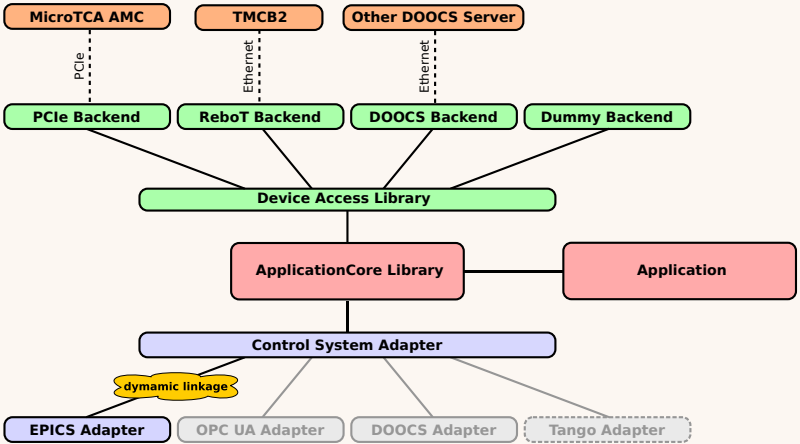
Idea

- Introduce similar abstraction as for the hardware access
- Not trivial - control system middleware much more than just a communication layer
- Try **not** to create a new control system!

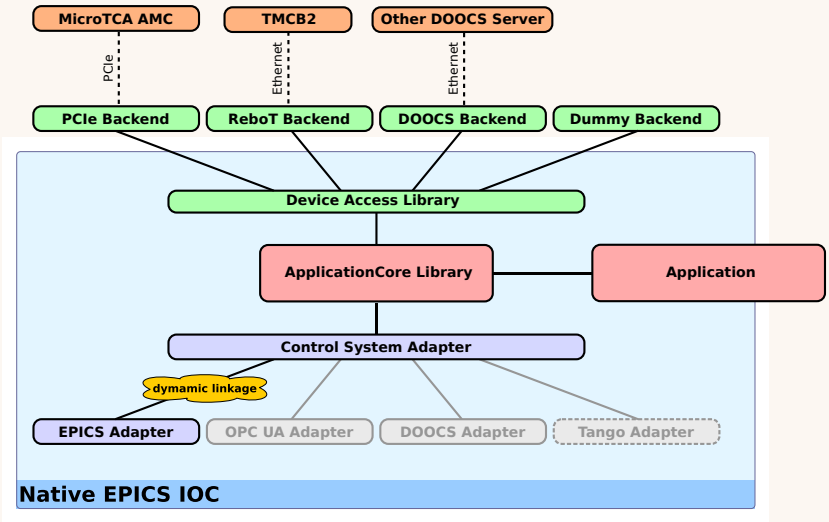
Control system integration



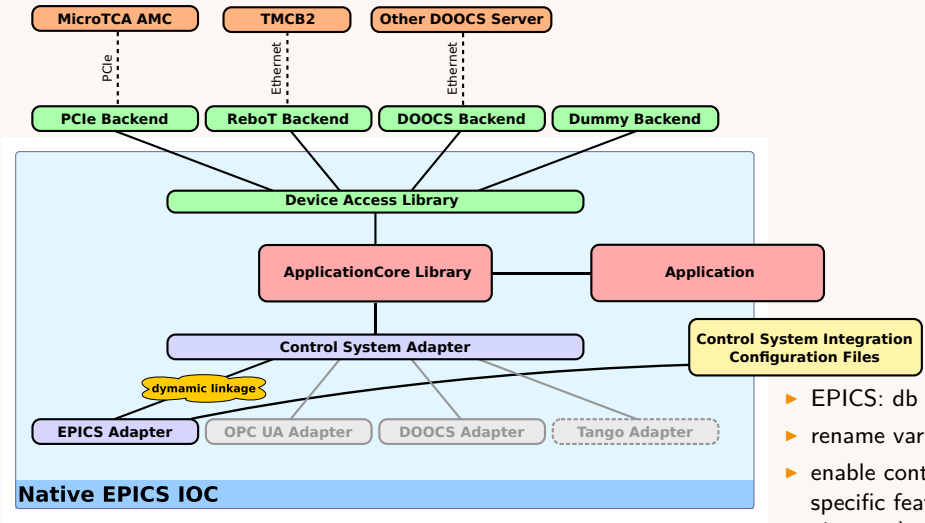
Control system integration



Control system integration

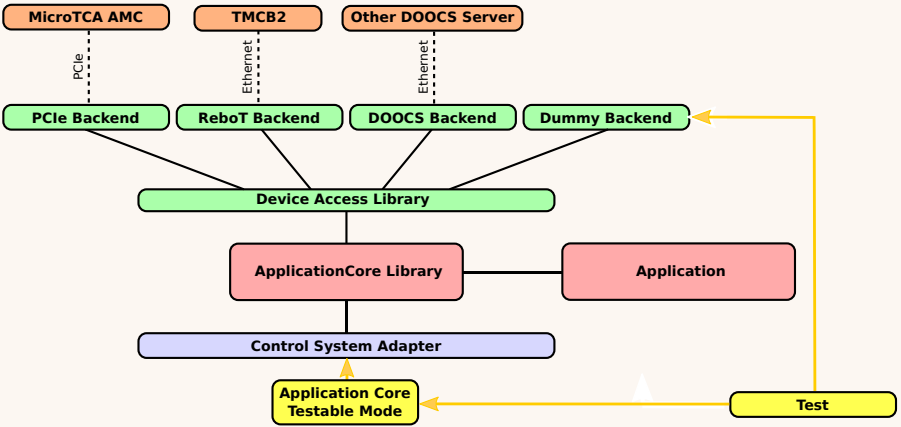


Control system integration

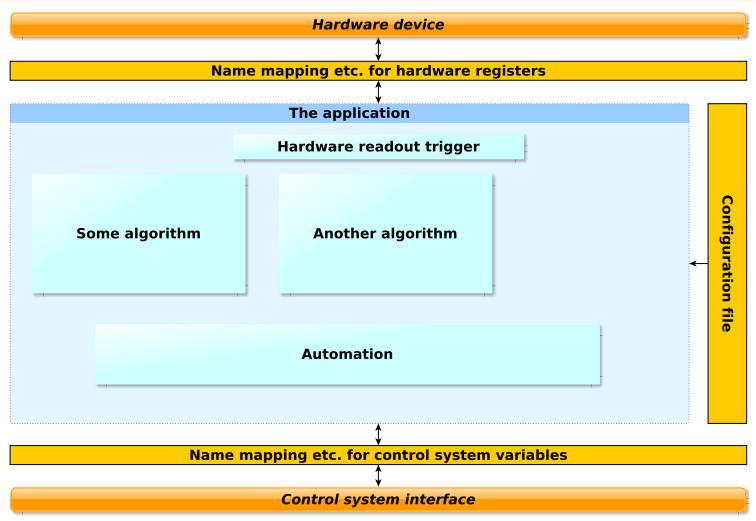


- EPICS: db files etc.
- rename variables
- enable control system specific features (histories, alarms...)

Testing the application

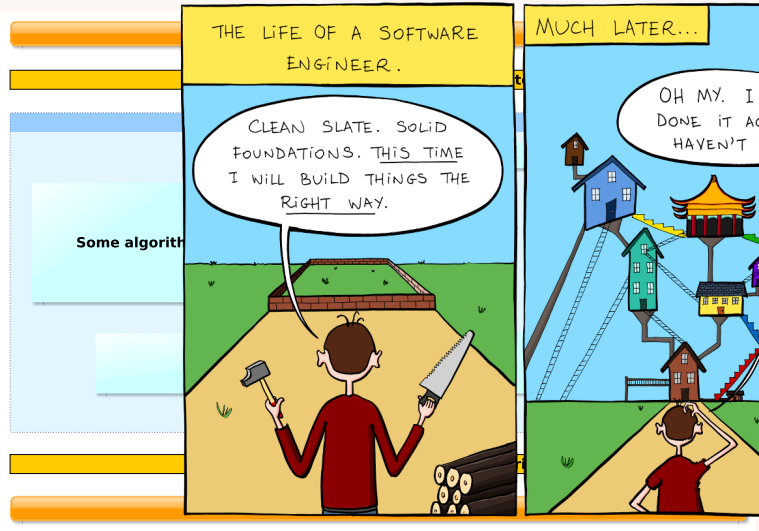


Summary



- ChimeraTK-ApplicationCore unifies DeviceAccess and ControlSystemAdapter
- Self-contained modules
- Modern multi-threading (“for free”)
- Optional mapping layers to catch interface changes

Summary



by Manu Cornet (www.bonkersworld.net)

eraTK-ApplicationCore
s DeviceAccess and
olSystemAdapter

ontained modules

rn multi-threading
free")

nal mapping layers to
interface changes

How to get started...

```
#include <ApplicationCore.h>
#include <PeriodicTrigger.h>
namespace ctk = ChimeraTK;

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

    ctk::PeriodicTrigger timer{this, "Timer", "Periodic timer (1000ms period)", 1000};

    ctk::DeviceModule dev{"oven"};
    ctk::ControlSystemModule cs{"Bakery"};

    void defineConnections();
};
static ExampleApp theExampleApp;

void ExampleApp::defineConnections() {
    ChimeraTK::setDMapFilePath("example2.dmap");
    dev.connectTo(cs, timer.tick);
}
```

Yes, this is 100% complete! (but requires the head revision of ChimeraTK...)

How to get started...

```
#include <ApplicationCore.h>
#include <PeriodicTrigger.h>
namespace ctk = ChimeraTK;

struc
    E
    ~
    c
    c
    c
    v
};
static ExampleApp theExampleApp;

void ExampleApp::defineConnections() {
    ChimeraTK::setDMapFilePath("example2.dmap");
    dev.connectTo(cs, timer.tick);
}
```

Availability of ChimeraTK

- All ChimeraTK libraries are released as open source under LGPL license
- Source code: `https://github.com/ChimeraTK`
- Documentation: `https://chimeratk.github.io`
- Debian packages for Ubuntu 16.04: `https://chimeratk.github.io`
- Launchpad-hosted PPA (for all Ubuntu versions) is in preparation

Yes, this is 100% complete! (but requires the head revision of ChimeraTK...)

Backup slides.

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?)

Direct connections between devices and the control system

(Experimental feature, not yet released...)

- ApplicationCore can 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 (poll/push) depends on register capabilities
- Planned: Exceptions from these automatic rules can still be made by providing the information per-variable (or maybe per `connectTo()`?)