

Introduction to ApplicationCore.



Martin Hierholzer

MicroTCA Workshop - Tutorial "Introduction to ChimeraTK"

2017-12-05

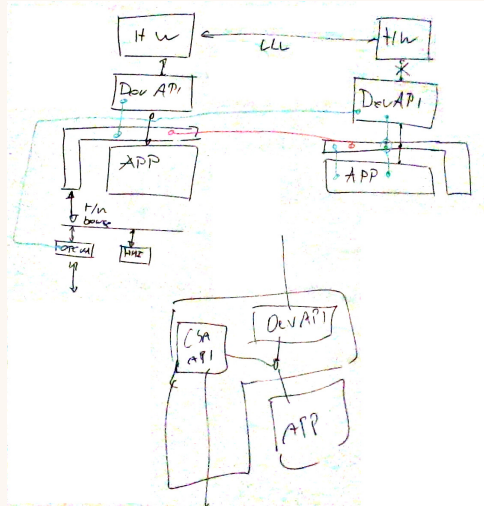
- ▶ Kick-off meeting for the OPC UA adapter with TU-DD (Summer 2016)
- ▶ Why do we treat control system variables and registers differently?

- ▶ Kick-off meeting for the OPC UA adapter with TU-DD (Summer 2016)
- ▶ Why do we treat control system variables and registers differently?
- ▶ Control system variable \Rightarrow Server
- ▶ DeviceAccess register \Rightarrow Client
- ▶ Conceptionally the same, can sometimes even be exchanged!

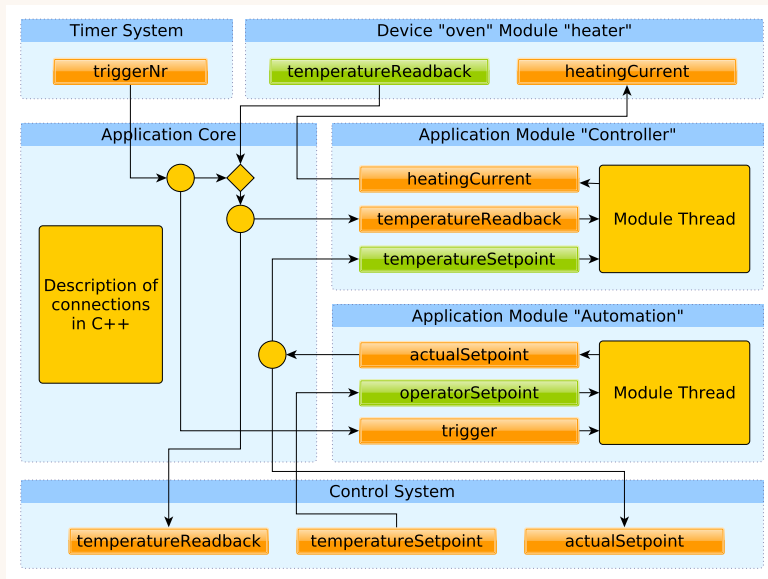
- ▶ Kick-off meeting for the OPC UA adapter with TU-DD (Summer 2016)
- ▶ Why do we treat control system variables and registers differently?
- ▶ Control system variable \Rightarrow Server
- ▶ DeviceAccess register \Rightarrow Client
- ▶ Conceptionally the same, can sometimes even be exchanged!
- ▶ DOOCS example:
 - ▶ Client $\xrightarrow{\text{write via RPC}}$ Server
 - ▶ Server $\xrightarrow{\text{send via OMQ}}$ Client

- ▶ Kick-off meeting for the OPC UA adapter with TU-DD (Summer 2016)
- ▶ Why do we treat control system variables and registers differently?
- ▶ Control system variable \Rightarrow Server
- ▶ DeviceAccess register \Rightarrow Client
- ▶ Conceptionally the same, can sometimes even be exchanged!
- ▶ DOOCS example:
 - ▶ Client $\xrightarrow{\text{write via RPC}}$ Server
 - ▶ Server $\xrightarrow{\text{send via OMQ}}$ Client
- ▶ Of course one has to decide for one implementation, but no fundamental difference in the application

How it all started



The structure of ApplicationCore



Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!

Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- ▶ Separate actual application code (algorithms etc.) from control system and device implementation details

Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- ▶ Separate actual application code (algorithms etc.) from control system and device implementation details
- ▶ Encourage modular applications (→ conceptual abstraction)

Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- ▶ Separate actual application code (algorithms etc.) from control system and device implementation details
- ▶ Encourage modular applications (→ conceptual abstraction)
- ▶ Clean and simple interface, avoid boiler plate code as much as C++11 allows

Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- ▶ Separate actual application code (algorithms etc.) from control system and device implementation details
- ▶ Encourage modular applications (→ conceptual 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)

Goal: provide framework for implementing applications which are “naturally” control-system independent

- ▶ If we abstract away differences between control system and device variables, we will less likely make our application sensitive to specific control systems!
- ▶ Separate actual application code (algorithms etc.) from control system and device implementation details
- ▶ Encourage modular applications (→ conceptual 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

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer
- ▶ Conceptual abstraction:

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer
- ▶ Conceptual abstraction:
 - ▶ Make your application independent even of concepts implied by the use of a particular technology

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer
- ▶ Conceptual abstraction:
 - ▶ Make your application independent even of concepts implied by the use of a particular technology
 - ▶ Example: printer library drawing line on vector plotters and pixel printers

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer
- ▶ Conceptual abstraction:
 - ▶ Make your application independent even of concepts implied by the use of a particular technology
 - ▶ Example: printer library drawing line on vector plotters and pixel printers
 - ▶ This is a difficult one! Often not possible on level of entire applications
 - ▶ Instead: break down application into abstract parts, combine them with little code

Two types of abstraction (strongly related):

- ▶ API abstraction:
 - ▶ Make your application independent of 3rd-party APIs
 - ▶ Example: use one common API for access to any register-based hardware - independent of the transport protocol
 - ▶ Other example: printer library drawing line on a HPGL and a PCL printer
- ▶ Conceptual abstraction:
 - ▶ Make your application independent even of concepts implied by the use of a particular technology
 - ▶ Example: printer library drawing line on vector plotters and pixel printers
 - ▶ This is a difficult one! Often not possible on level of entire applications
 - ▶ Instead: break down application into abstract parts, combine them with little code
- ▶ Conceptual abstraction usually involves also API abstraction
- ▶ Both improve code quality a lot!

- ▶ Default arguments to member constructors with braces

```
struct SomeClass {  
    std::string myText{"Hello World"};  
};
```

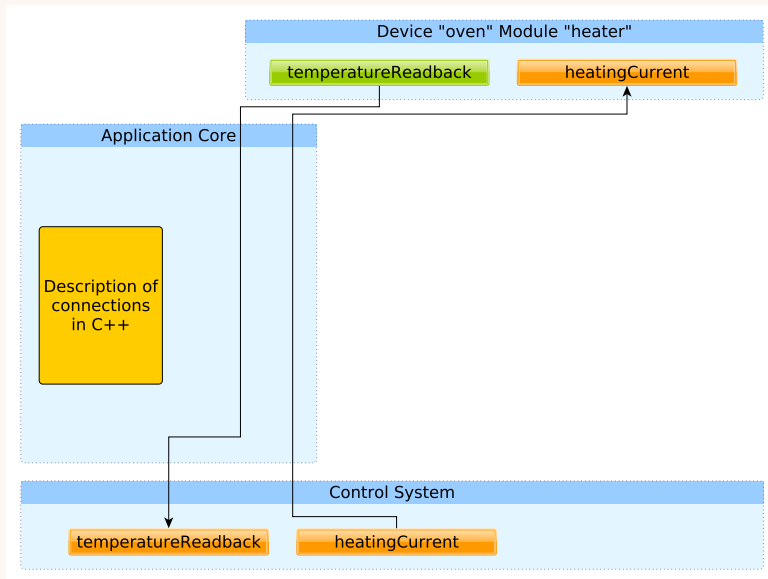
- ▶ Default arguments to member constructors with braces

```
struct SomeClass {  
    std::string myText{"Hello World"};  
};
```

- ▶ Brace-initialisers when passing arguments

```
void someFunction(std::vector<int> values);  
void main() {  
    someFunction({1, 42, 33});  
}
```

Direct access to device registers




```
struct ExampleApp : public ctk::Application {  
  
    ctk::DeviceModule heater{"oven", "heater"};  
  
    ctk::ControlSystemModule cs{"Bakery"};  
  
    void defineConnections();  
};  
ExampleApp theExampleApp;
```

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

    ctk::DeviceModule heater{"oven", "heater"};

    ctk::ControlSystemModule cs{"Bakery"};

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

```
void ExampleApp::defineConnections() {  
  
    cs("heatingCurrent") >> heater("heatingCurrent");  
    heater("temperatureReadback") >> cs("temperatureReadback");  
  
}
```

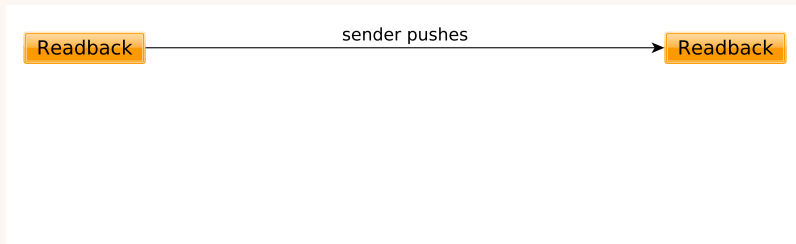
```
void ExampleApp::defineConnections() {  
  
    cs("heatingCurrent") >> heater("heatingCurrent");  
    heater("temperatureReadback") >> cs("temperatureReadback");  
  
}
```

- ▶ Neither DeviceModule nor ControlSystemModule define their variable types
- ▶ DeviceAccess allows to read a register as any type
- ▶ ControlSystemAdater variables are created on-the-fly
- ▶ We need to specify the type and array length here!

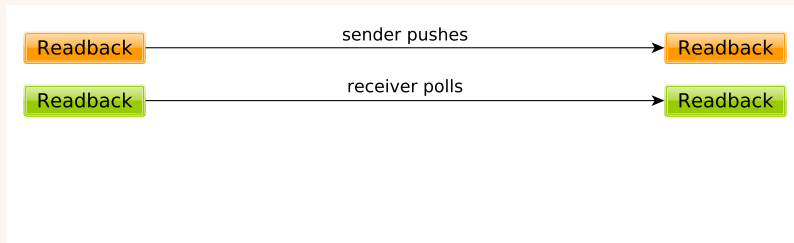
```
void ExampleApp::defineConnections() {  
  
    cs("heatingCurrent", typeid(int), 1) >> heater("heatingCurrent");  
    heater("temperatureReadback", typeid(int), 1) >> cs("temperatureReadback");  
  
}
```

- ▶ Neither DeviceModule nor ControlSystemModule define their variable types
- ▶ DeviceAccess allows to read a register as any type
- ▶ ControlSystemAdater variables are created on-the-fly
- ▶ We need to specify the type and array length here!

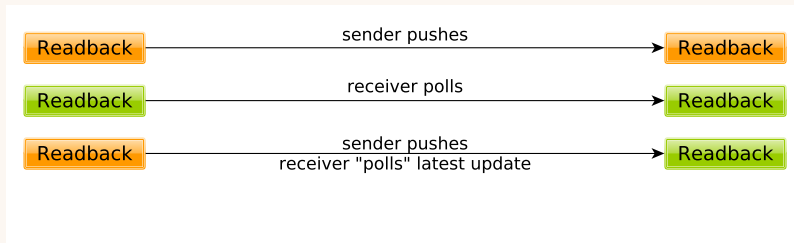
- Generalise the client/server concept



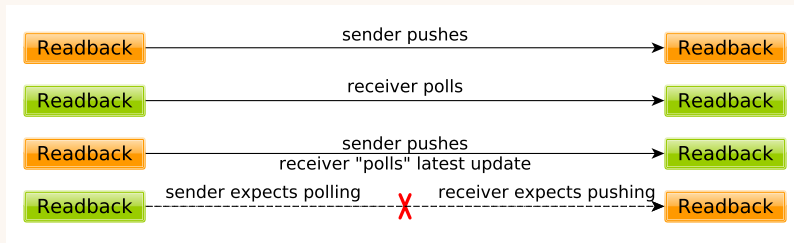
- Generalise the client/server concept

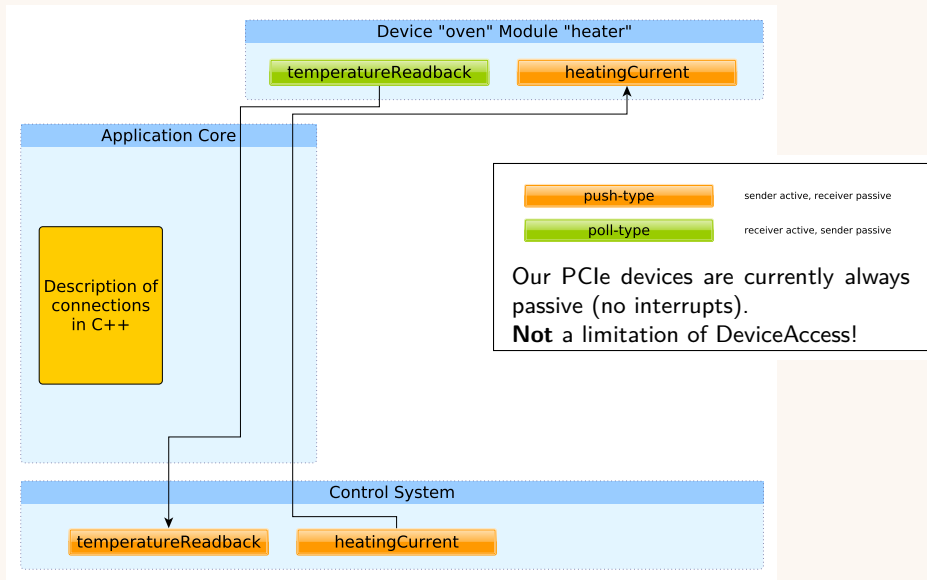


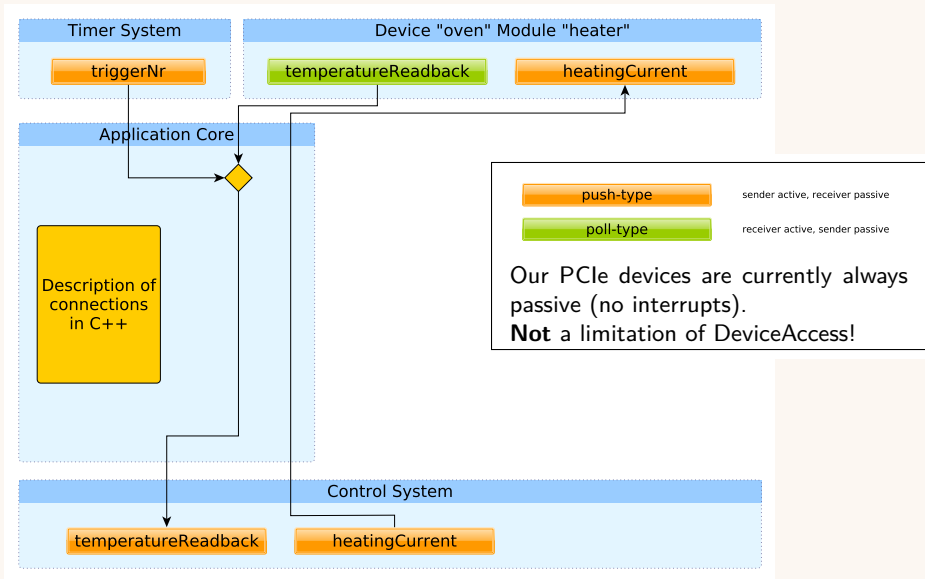
- Generalise the client/server concept



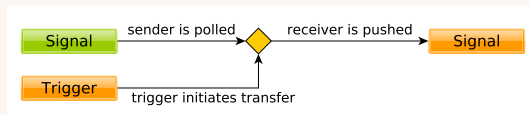
- Generalise the client/server concept







- ▶ When connecting a poll-type output (e.g. a PCIe register) with a push-type input, a trigger for the transfer is needed



- ▶ Any push-type variable can act as a trigger, its value will be ignored
- ▶ It can come from a device, another ApplicationModule or even the ControlSystemAdapter

```
void ExampleApp::defineConnections() {  
    mtca4u::setDMapFilePath("devices.dmap");  
  
    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);  
  
    cs("heatingCurrent", typeid(int), 1) >> heater("heatingCurrent");  
    heater("temperatureReadback", typeid(double), 1) [ triggerNr ]  
        >> cs("temperatureReadback");  
}
```

```
void ExampleApp::defineConnections() {  
    mtca4u::setDMapFilePath("devices.dmap");  
  
    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);  
  
    cs("heatingCurrent", typeid(int), 1) >> heater("heatingCurrent");  
    heater("temperatureReadback", typeid(double), 1) [ triggerNr ]  
        >> cs("temperatureReadback");  
}
```

- ▶ DeviceAccess variables are by default poll-type
- ▶ Need to change this to push-type

Define the application (actual code!)



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

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

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

Define the application (actual code!)



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

    ctk::DeviceModule heater, "heater";
    ctk::DeviceModule timer;
    ctk::ControlModule cs{"Bakery"};

    void DefineConnections();
};

ExampleApp theExampleApp;
```

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

Define the application (actual code!)



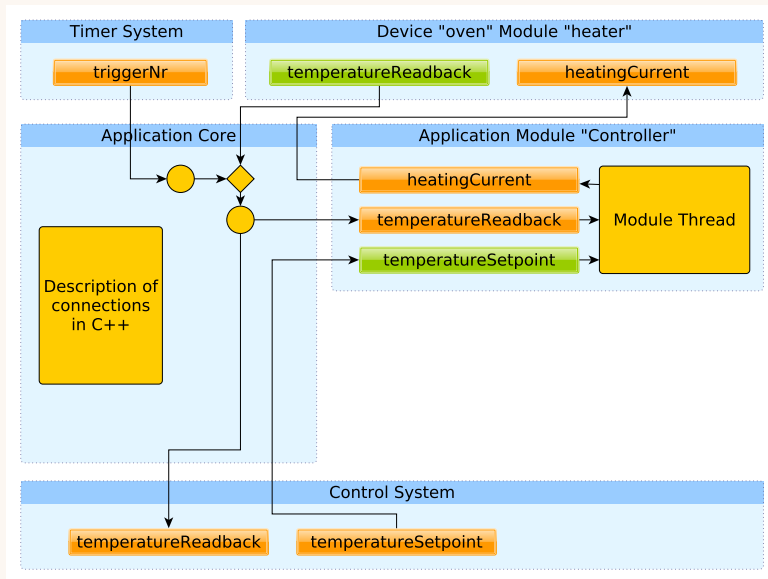
```
struct ExampleApp : public ctk::Application {
    ExampleApp() : Application("exampleApp") {}
    ~ExampleApp() { shutdown(); }

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

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

Live demo now! (01)

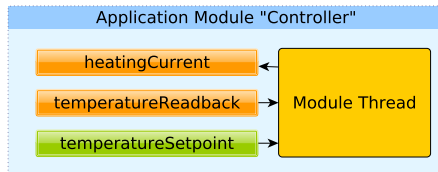
Make the application smarter!



```
struct Controller : public ctk::ApplicationModule {

    ctk::ScalarInput<double> sp{"sp", "degC", "Description"};
    ctk::ScalarInput<double> rb{"rb", "degC", "..."};
    ctk::ScalarOutput<double> cur{"cur", "mA", "..."};

    void mainLoop() {
        const double gain = 100.0;
        while(true) {
            rb.read();
            sp.read();
            cur = gain * (sp - rb);
            cur.write();
        }
    }
};
```



Add an ApplicationModule (towards actual code - step 1)



```
struct Controller : public ctk::ApplicationModule {  
  
    ctk::ScalarInput<double> sp{this, "sp", "degC", "Description"};  
    ctk::ScalarInput<double> rb{this, "rb", "degC", "..."};  
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "..."};  
  
    void mainLoop() {  
        const double gain = 100.0;  
        while(true) {  
            rb.read();  
            sp.read();  
            cur = gain * (sp - rb);  
            cur.write();  
        }  
    }  
};
```

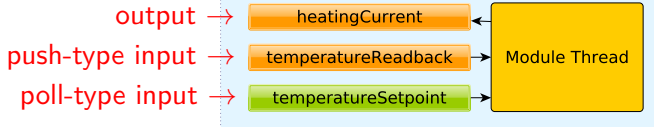
Need to know the owner!

```
struct Controller : public ctk::ApplicationModule {  
    using ctk::ApplicationModule::ApplicationModule;    ← Inherit constructor  
    ctk::ScalarInput<double> sp{this, "sp", "degC", "Description"};  
    ctk::ScalarInput<double> rb{this, "rb", "degC", "..."};  
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "..."};  
  
    void mainLoop() {  
        const double gain = 100.0;  
        while(true) {  
            rb.read();  
            sp.read();  
            cur = gain * (sp - rb);  
            cur.write();  
        }  
    }  
};
```

```
struct Controller : public ctk::ApplicationModule {  
    using ctk::ApplicationModule::ApplicationModule;  
    ctk::ScalarPollInput<double> sp{this, "sp", "degC", "Description"};  
    ctk::ScalarPushInput<double> rb{this, "rb", "degC", "..."};  
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "..."};  
    Define the update mode. Outputs are always push-type.  
    void mainLoop() {  
        const double gain = 100.0;  
        while(true) {  
            rb.read();    // waits until update arrives  
            sp.read();    // just reads latest value  
            cur = gain * (sp - rb);  
            cur.write();  
        }  
    }  
};
```

```
struct Controller : public ctk::ApplicationModule {  
    using ctk::ApplicationModule::ApplicationModule;  
    ctk::ScalarPollInput<double> sp{this, "sp", "degC", "Description"};  
    ctk::ScalarPushInput<double> rb{this, "rb", "degC", "..."};  
    ctk::ScalarOutput<double> cur{this, "cur", "mA", "..."};  
    void mainLoop() {  
        const double gain = 100.0;  
        while(true) {  
            rb.read();    // waits until update arrives  
            sp.read();    // just reads latest value  
            cur = gain * (sp - rb);  
            cur.write();  
        }  
    }  
};
```

Define the update mode. Outputs are always push-type.



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

Interface from DeviceAccess

- ▶ ScalarPollInput / ScalarPushInput / ScalarOutput
⇒ ScalarRegisterAccessor
- ▶ ArrayPollInput / ArrayPushInput / ArrayOutput
⇒ OneDRegisterAccessor

```
};
```



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

Interface from DeviceAccess

- ▶ ScalarPollInput / ScalarPushInput / ScalarOutput
⇒ ScalarRegisterAccessor
- ▶ ArrayPollInput / ArrayPushInput / ArrayOutput
⇒ OneDRegisterAccessor
- ▶ Actual inheritance!
- ▶ Only adds *inversion of control*

```
};
```

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

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

Live demo now! (02)

```
void ExampleApp::defineConnections() {  
    mtca4u::setDMapFilePath("devices.dmap");  
  
    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);  
  
    cs("temperatureSetpoint") >> controller.temperatureSetpoint;  
    controller.heatingCurrent >> heater("heatingCurrent");  
    heater("temperatureReadback") [ triggerNr ] >> controller.temperatureReadback  
        >> cs("temperatureReadback");  
}
```

- ▶ Triggers are in many cases not strictly needed, one could do periodic polling in the ApplicationModule instead (i.e. poll-type input and sleep)

- ▶ Triggers are in many cases not strictly needed, one could do periodic polling in the ApplicationModule instead (i.e. poll-type input and sleep)
- ▶ If consistency (or at least near consistency) is required, periodic polling must be done in the same ApplicationModule

- ▶ Triggers are in many cases not strictly needed, one could do periodic polling in the ApplicationModule instead (i.e. poll-type input and sleep)
- ▶ If consistency (or at least near consistency) is required, periodic polling must be done in the same ApplicationModule
- ▶ Conceptual abstraction:
 - ▶ Keep a variable out an ApplicationModule if it doesn't belong there!
Don't place `supplyVoltages` into the `Controller` module

- ▶ Triggers are in many cases not strictly needed, one could do periodic polling in the ApplicationModule instead (i.e. poll-type input and sleep)
- ▶ If consistency (or at least near consistency) is required, periodic polling must be done in the same ApplicationModule
- ▶ Conceptual abstraction:
 - ▶ Keep a variable out an ApplicationModule if it doesn't belong there!
Don't place `supplyVoltages` into the `Controller` module
 - ▶ If an ApplicationModule acts upon change of an input, make that clear by using a push-type input

- ▶ Triggers are in many cases not strictly needed, one could do periodic polling in the `ApplicationModule` instead (i.e. poll-type input and sleep)
- ▶ If consistency (or at least near consistency) is required, periodic polling must be done in the same `ApplicationModule`
- ▶ Conceptual abstraction:
 - ▶ Keep a variable out an `ApplicationModule` if it doesn't belong there!
Don't place `supplyVoltages` into the `Controller` module
 - ▶ If an `ApplicationModule` acts upon change of an input, make that clear by using a push-type input
- ▶ NB: Application-wide periodic triggers are like the cycle time in PLCs

Let's add some monitoring!



```
void ExampleApp::defineConnections() {
    mtca4u::setDMapFilePath("devices.dmap");

    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);

    cs("temperatureSetpoint") >> controller.temperatureSetpoint;
    controller.heatingCurrent >> heater("heatingCurrent");
    heater("temperatureReadback") [ triggerNr ] >> controller.temperatureReadback
        >> cs("temperatureReadback");

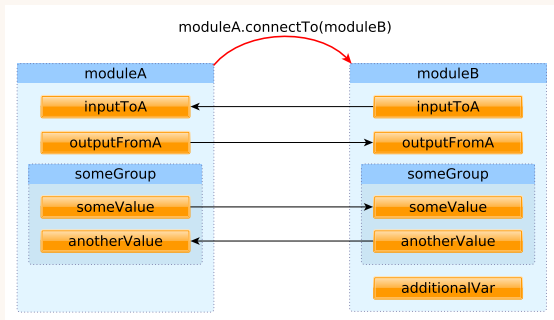
    heater("supplyVoltages", typeid(int), 4) [ triggerNr ] >> cs("supplyVoltages");
    triggerNr >> cs("triggerNr");
}
```

Live demo now! (03)

Shortcut: do many connections at a time with `connectTo()`

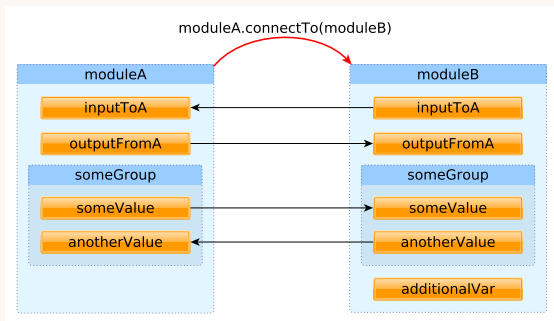


- ▶ Connect all variables of module with variables of same name in other module
- ▶ Works recursively (variables can be grouped with `VariableGroup`)



Shortcut: do many connections at a time with connectTo()

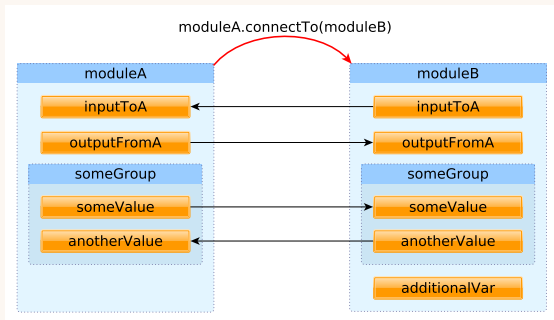
- ▶ Connect all variables of module with variables of same name in other module
- ▶ Works recursively (variables can be grouped with VariableGroup)
- ▶ Does not care about additional variables in the target module
- ▶ Even acts on inputs and outputs simultaneously



Shortcut: do many connections at a time with connectTo()



- ▶ Connect all variables of module with variables of same name in other module
- ▶ Works recursively (variables can be grouped with VariableGroup)
- ▶ Does not care about additional variables in the target module
- ▶ Even acts on inputs and outputs simultaneously
- ▶ Works nicely with ControlSystemModule which creates variables on demand



Shortcut: do many connections at a time with connectTo()



```
void ExampleApp::defineConnections() {
    mtca4u::setDMapFilePath("devices.dmap");

    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);

    cs("temperatureSetpoint") >> controller.temperatureSetpoint;
    controller.heatingCurrent >> heater("heatingCurrent");
    heater("temperatureReadback") [ triggerNr ] >> controller.temperatureReadback
        >> cs("temperatureReadback");

    heater("supplyVoltages", typeid(int), 4) [ triggerNr ] >> cs("supplyVoltages");
    triggerNr >> cs("triggerNr");
}
```

Shortcut: do many connections at a time with connectTo()



```
void ExampleApp::defineConnections() {
    mtca4u::setDMapFilePath("devices.dmap");

    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);

    cs("temperatureSetpoint") >> controller.temperatureSetpoint;
    controller.heatingCurrent >> heater("heatingCurrent");
    heater("temperatureReadback") [ triggerNr ] >> controller.temperatureReadback;
    controller.temperatureReadback >> cs("temperatureReadback");

    heater("supplyVoltages", typeid(int), 4) [ triggerNr ] >> cs("supplyVoltages");
    triggerNr >> cs("triggerNr");
}
```

Shortcut: do many connections at a time with connectTo()



```
void ExampleApp::defineConnections() {
    mtca4u::setDMapFilePath("devices.dmap");

    auto triggerNr = timer("triggerNr", typeid(int), 1, ctk::UpdateMode::push);

    // cs("temperatureSetpoint") >> controller.temperatureSetpoint;
    controller.heatingCurrent >> heater("heatingCurrent");
    heater("temperatureReadback") [ triggerNr ] >> controller.temperatureReadback;
    // controller.temperatureReadback >> cs("temperatureReadback");

    heater("supplyVoltages", typeid(int), 4) [ triggerNr ] >> cs("supplyVoltages");
    triggerNr >> cs("triggerNr");

    controller.connectTo(cs);    // could connect arbitrary number of variables
}
```

Live demo now! (04)

- ▶ Variables can have any number of tags attached
- ▶ A tag is any alpha-numeric string (no spaces)
- ▶ Not visible outside the application
- ▶ Can be used to search for a subset of variables


```
// ApplicationModule "Controller":  
ctk::ScalarPollInput<double> temperatureSetpoint{..., {"CS"}};  
ctk::ScalarOutput<double> heatingCurrent{..., {"HEATER"}};  
ctk::ScalarPushInput<double> temperatureReadback{..., {"CS", "HEATER"}};  
  
// defineConnections():  
controller.findTag("HEATER").connectTo(heater, triggerNr);  
controller.findTag("CS").connectTo(cs);
```

```
// ApplicationModule "Controller":
ctk::ScalarPollInput<double> temperatureSetpoint{..., {"CS"}};
ctk::ScalarOutput<double> heatingCurrent{..., {"HEATER"}};
ctk::ScalarPushInput<double> temperatureReadback{..., {"CS", "HEATER"}};

// defineConnections():
controller.findTag("HEATER").connectTo(heater, triggerNr;
controller.findTag("CS").connectTo(cs);
```

```
/**
 * Connect the entire module into another module. [...]
 *
 * If an optional trigger node is specified, this trigger node is applied to all
 * poll-type output variables of the target module, which are being connected during
 * this operation, if the corresponding variable in this module is push-type.
 */
void Module::connectTo(const Module &target, VariableNetworkNode trigger={}) const;
```

```
// ApplicationModule "Controller":  
ctk::ScalarPollInput<double> temperatureSetpoint{..., {"CS"}};  
ctk::ScalarOutput<double> heatingCurrent{..., {"HEATER"}};  
ctk::ScalarPushInput<double> temperatureReadback{..., {"CS", "HEATER"}};  
  
// defineConnections():  
controller.findTag("HEATER").connectTo(heater, triggerNr);  
controller.findTag("CS").connectTo(cs);
```

The order matters!

- ▶ The order of connectTo() statements may matter
- ▶ Device and control system variables decide their direction from the context
- ▶ Swapping the two statements would feed temperatureReadback from the control system and pass a copy to the device

```
// ApplicationModule "Controller":  
ctk::ScalarPollInput<double> temperatureSetpoint{..., {"CS"}};  
ctk::ScalarOutput<double> heatingCurrent{..., {"HEATER"}};  
ctk::ScalarPushInput<double> temperatureReadback{..., {"CS", "HEATER"}};  
  
// defineConnections():  
controller.findTag("HEATER").connectTo(heater, triggerNr);  
controller.findTag("CS").connectTo(cs);
```

Live demo now! (05)

- ▶ A pipe is a standard ApplicationModule simply passing on a variable
- ▶ Very useful for direct connections between devices and control system

```
// Application:
ctk::ArrayPipe<int> supplyVoltages{ this, "supplyVoltages", "mV", 4, "...",
                                   {"HEATER"}, {"CS"} };

//                               ^^          ^^ output tags

// defineConnections():
controller.findTag("HEATER").connectTo(heater, triggerNr);
controller.findTag("CS").connectTo(cs);
supplyVoltages.findTag("HEATER").connectTo(heater, triggerNr);
supplyVoltages.findTag("CS").connectTo(cs);
```

- ▶ A pipe is a standard ApplicationModule simply passing on a variable
- ▶ Very useful for direct connections between devices and control system

```
// Application:
ctk::ArrayPipe<int> supplyVoltages{ this, "supplyVoltages", "mV", 4, "...",
                                   {"HEATER"}, {"CS"} };

//                               ^^          ^^ output tags

// defineConnections() smarter:
findTag("HEATER").connectTo(heater, triggerNr);
findTag("CS").connectTo(cs);
```

- ▶ A pipe is a standard ApplicationModule simply passing on a variable
- ▶ Very useful for direct connections between devices and control system

```
// Application:
ctk::ArrayPipe<int> supplyVoltages{ this, "supplyVoltages", "mV", 4, "...",
                                   {"HEATER"}, {"CS"} };

//                               ^^          ^^ output tags

// defineConnections() smarter:
findTag("HEATER").connectTo(heater, triggerNr);
findTag("CS").connectTo(cs);
```

Live demo now! (06)

- ▶ A pipe is a standard ApplicationModule simply passing on a variable
- ▶ Very useful for direct connections between devices and control system

```
// Application:
ctk::ArrayPipe<int> supplyVoltages{ this, "supplyVoltages", "mV", 4, "...",
                                     {"HEATER"}, {"CS"} };

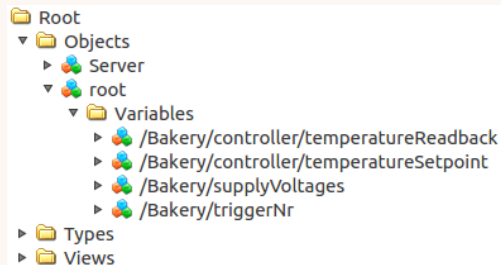
//                               ^^          ^^ output tags

// defineConnections() smarter:
findTag("HEATER").flatten().connectTo(heater, triggerNr);
findTag("CS").connectTo(cs);
```

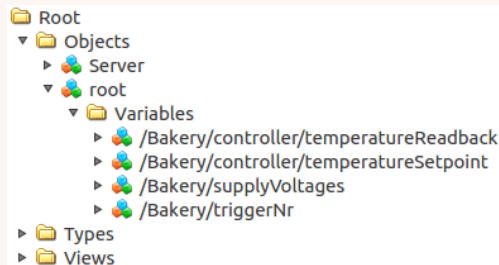

flatten()

- ▶ Eliminates all hierarchies (just for the current operation)
- ▶ Common use case: device with flat hierarchy but register access scattered over many modules

```
// defineConnections() smarter:  
findTag("HEATER").flatten().connectTo(heater, triggerNr);  
findTag("CS").connectTo(cs);
```



- ▶ Model already improved a bit by introducing the “controller” directory
- ▶ Still, a bakery does not has a controller - the oven has it!



- ▶ Model already improved a bit by introducing the “controller” directory
- ▶ Still, a bakery does not has a controller - the oven has it!
- ▶ Assuming this application represents an entire bakery, we shall introduce the oven an entity in both the source code and the control system hierarchy
- ▶ In other words: pick the right level of conceptional abstraction for each part of the application

```
struct Oven : public ctk::ModuleGroup {  
    using ctk::ModuleGroup::ModuleGroup;  
  
    Controller controller{this, "controller", "Proportional controller..."};  
    ctk::ArrayPipe<int> supplyVoltages{ this, "supplyVoltages", "mV", 4, "...",  
                                       {"HEATER"}, {"CS"} };  
  
};
```

Live demo now! (07)

```
// Application:
std::vector<Oven> ovens;
std::vector<ctk::DeviceModule> heaters;

// defineConnections():
for(size_t i=0; i<2; ++i) {
    ovens.emplace_back(this, "oven"+std::to_string(i), "Oven "+std::to_string(i));
    heaters.emplace_back("oven"+std::to_string(i), "heater");

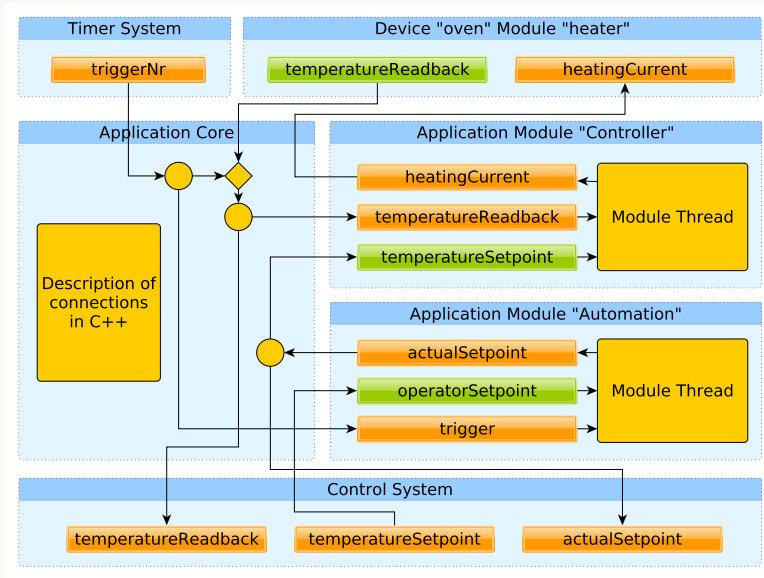
    ovens[i].findTag("HEATER").flatten().connectTo(heaters[i], triggerNr);
}
```

Live demo now! (08)

- ▶ `excludeTag()` is the opposite of `findTag()`
- ▶ `findTag()` and `excludeTag()` both support regular expressions
- ▶ Any level in the hierarchy might be “eliminated” if introduced only for technical reasons (e.g. need for `readAll()` on a `VariableGroup`), see `Module::setEliminateHierarchy()`
- ▶ In case of doubt: use `Module::dump()` or the result of `findTag()` etc., e.g.:

```
ovens[0].dump();  
findTag("CS").dump();
```

- ▶ Or use `Module::dumpGraph()` to generate GraphViz dot code



- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)

- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)
 - ▶ Often improves performance (on multi-core CPUs)

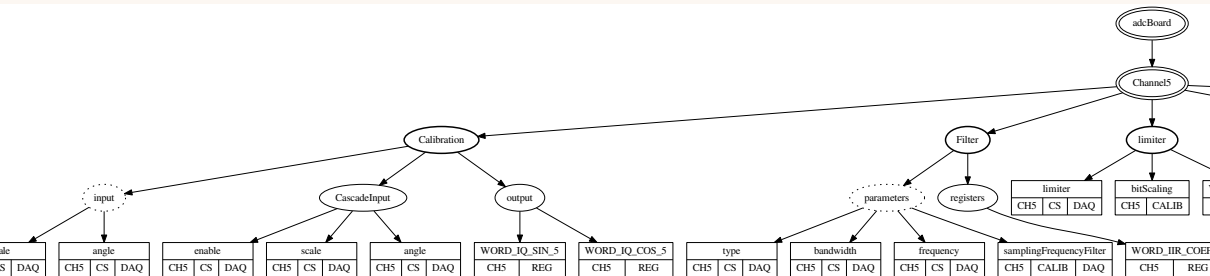
- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)
 - ▶ Often improves performance (on multi-core CPUs)
- ▶ Effectively quite advanced threading system based on futures and lock-free queues
- ▶ Message-based: values of variables are passed to other modules

- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)
 - ▶ Often improves performance (on multi-core CPUs)
- ▶ Effectively quite advanced threading system based on futures and lock-free queues
- ▶ Message-based: values of variables are passed to other modules
- ▶ Lock-free: no dead locks can occur

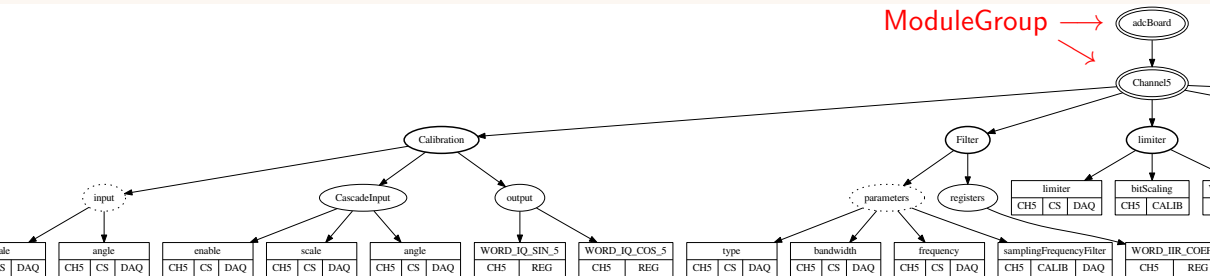
- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)
 - ▶ Often improves performance (on multi-core CPUs)
- ▶ Effectively quite advanced threading system based on futures and lock-free queues
- ▶ Message-based: values of variables are passed to other modules
- ▶ Lock-free: no dead locks can occur
- ▶ Futures: can effectively wait without using the cpu

- ▶ Advised to build applications from many small modules:
 - ▶ Improves application structure (easier conceptual abstraction, reusing code)
 - ▶ Often improves performance (on multi-core CPUs)
- ▶ Effectively quite advanced threading system based on futures and lock-free queues
- ▶ Message-based: values of variables are passed to other modules
- ▶ Lock-free: no dead locks can occur
- ▶ Futures: can effectively wait without using the cpu
- ▶ No mutexes: no non-concurrent code sections which would spoil scalability

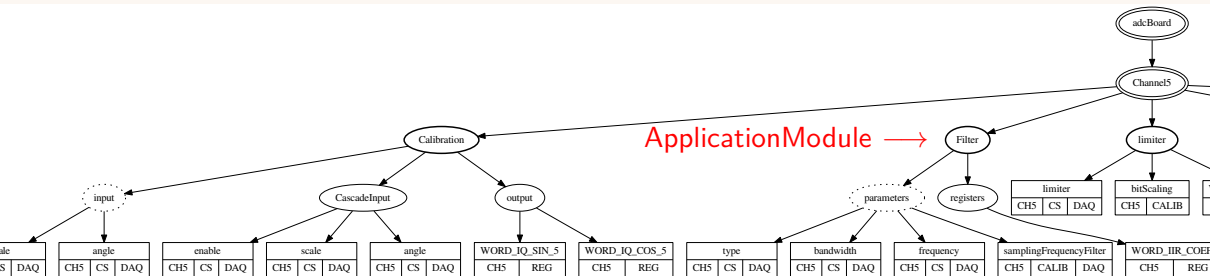
Example: LLRF server module adcBoard, Channel 5 (partial)



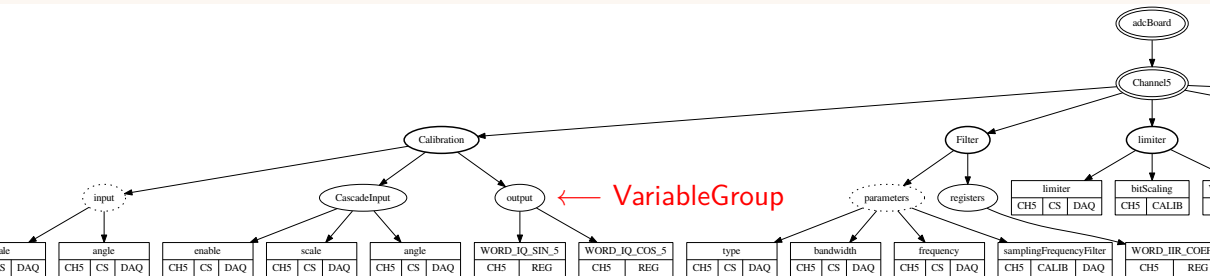
Example: LLRF server module adcBoard, Channel 5 (partial)



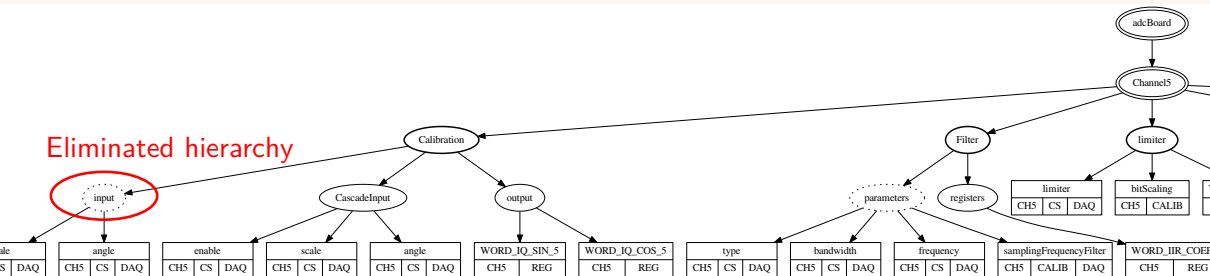
Example: LLRF server module adcBoard, Channel 5 (partial)



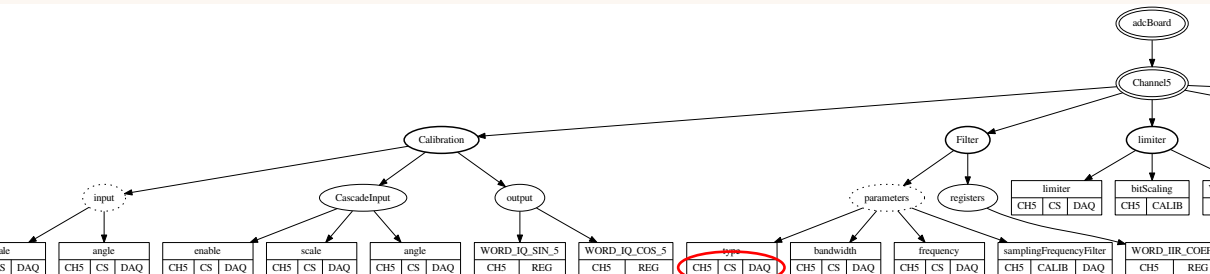
Example: LLRF server module adcBoard, Channel 5 (partial)



Example: LLRF server module adcBoard, Channel 5 (partial)



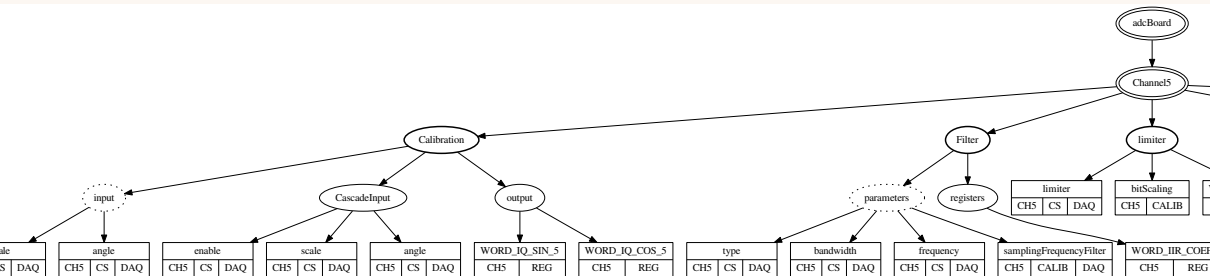
Example: LLRF server module adcBoard, Channel 5 (partial)



Tags

- ▶ Tag “CH5”: other modules also provide variables for Channel 5
- ▶ Tags “REG” and “CS”: device registers and control system variables
- ▶ Tag “CALIB”: global calibration values
- ▶ Tag “DAQ”: variables go into the internal DAQ system (if enabled)

Example: LLRF server module adcBoard, Channel 5 (partial)



- For more details about the LLRF server, come to my poster:
ApplicationCore: A Framework for Modern Control Applications at the Example of a Facility Independent LLRF Server

Git repository with source code used in this demo

- ▶ URL: <https://github.com/mhier/ApplicationCoreLiveDemo>
- ▶ Full source code incl. cmake files
- ▶ Also includes these slides