Mordicus-hw
Framework for backend electronics control and configuration

G. Stelmakh

CEA Irfu Saclay

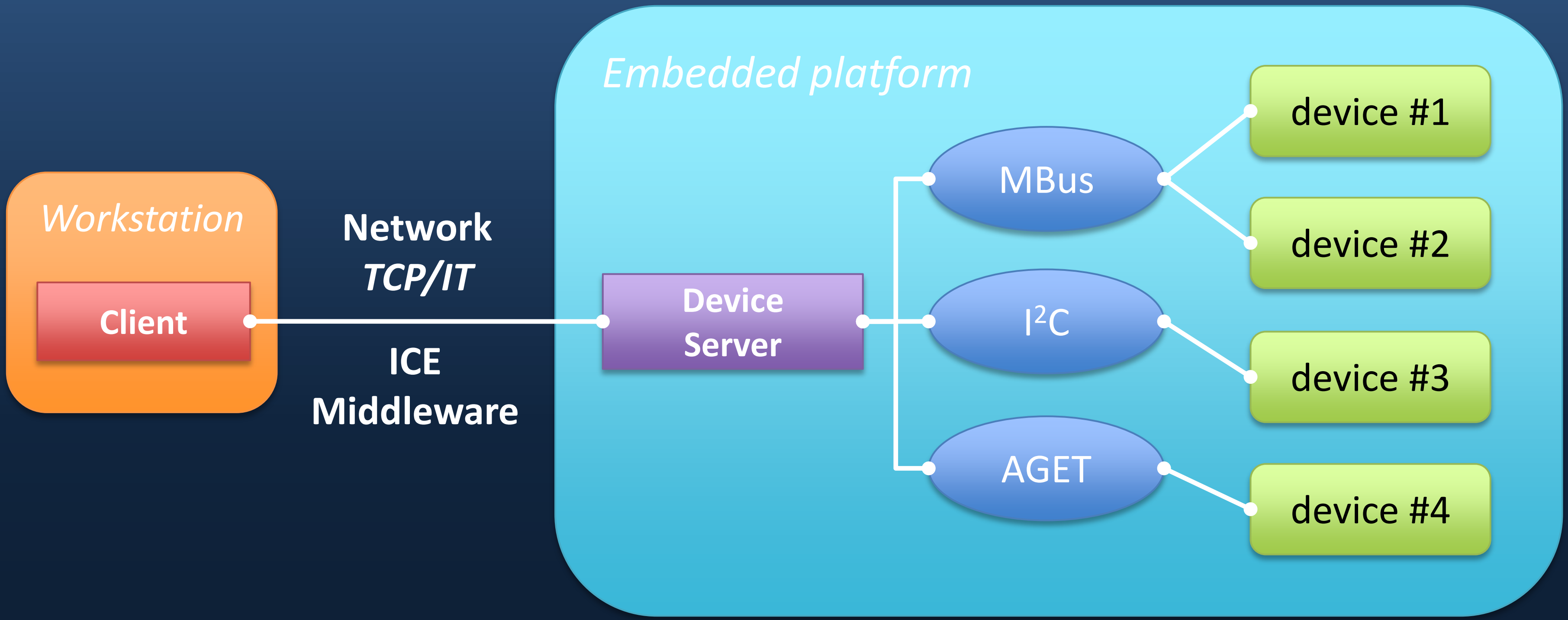
Workshop on picosecond photon sensors

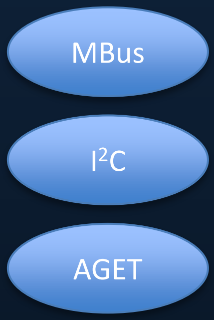
March 2014

What is Mordicus-hw?

Mordicus-hw is a C++ framework designed to optimize collaborative development between electronics and software engineers by providing them software tools that are adapted to their respective activities.

Device Server





MBus: Memory-mapped register access

I²C: I^2C register access

AGET: Proprietary register access

Highly Distributed Application

Distributed processes

Client-Server Architecture

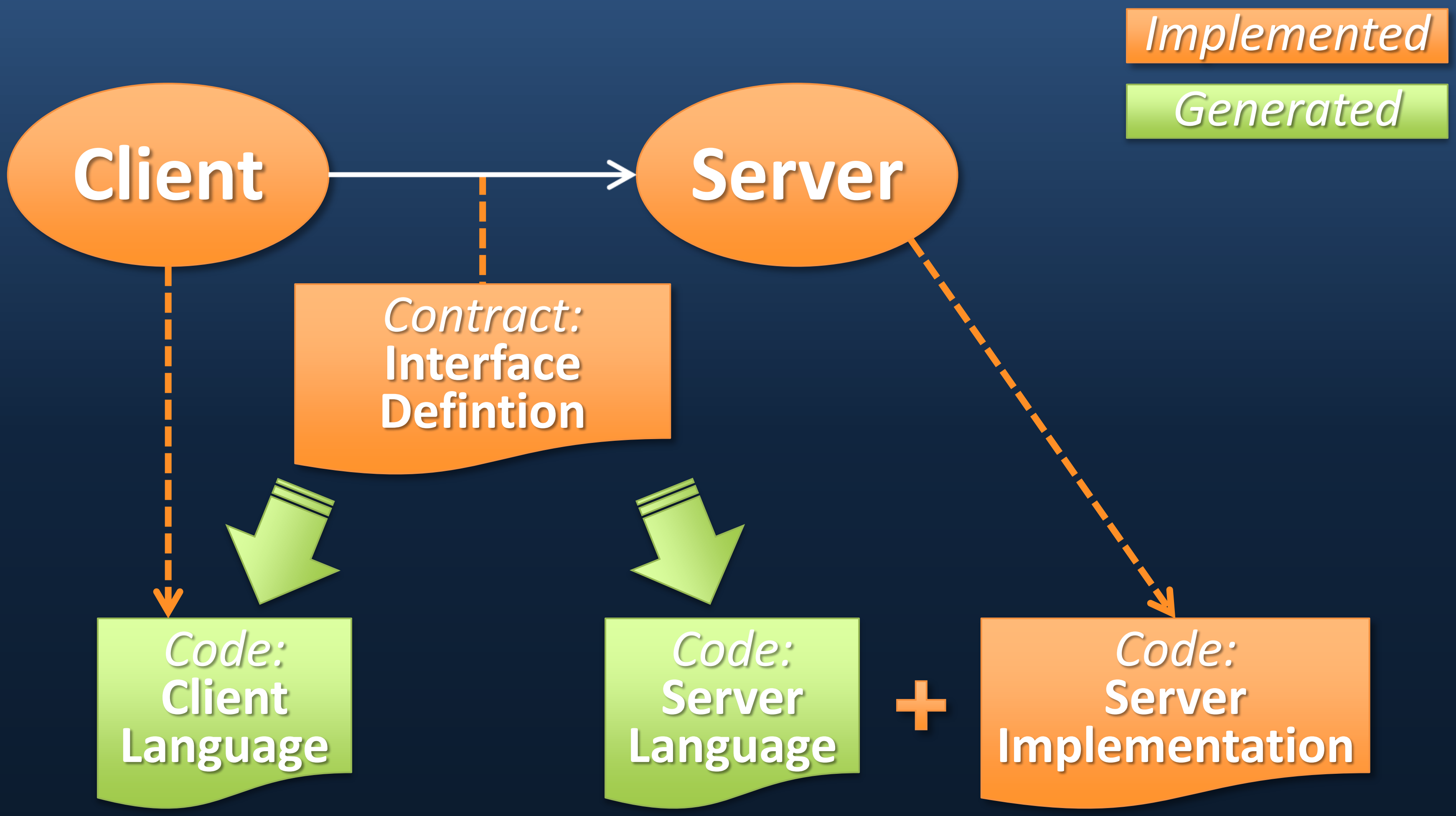
Middleware: Internet Communication Engine

TCP Protocol

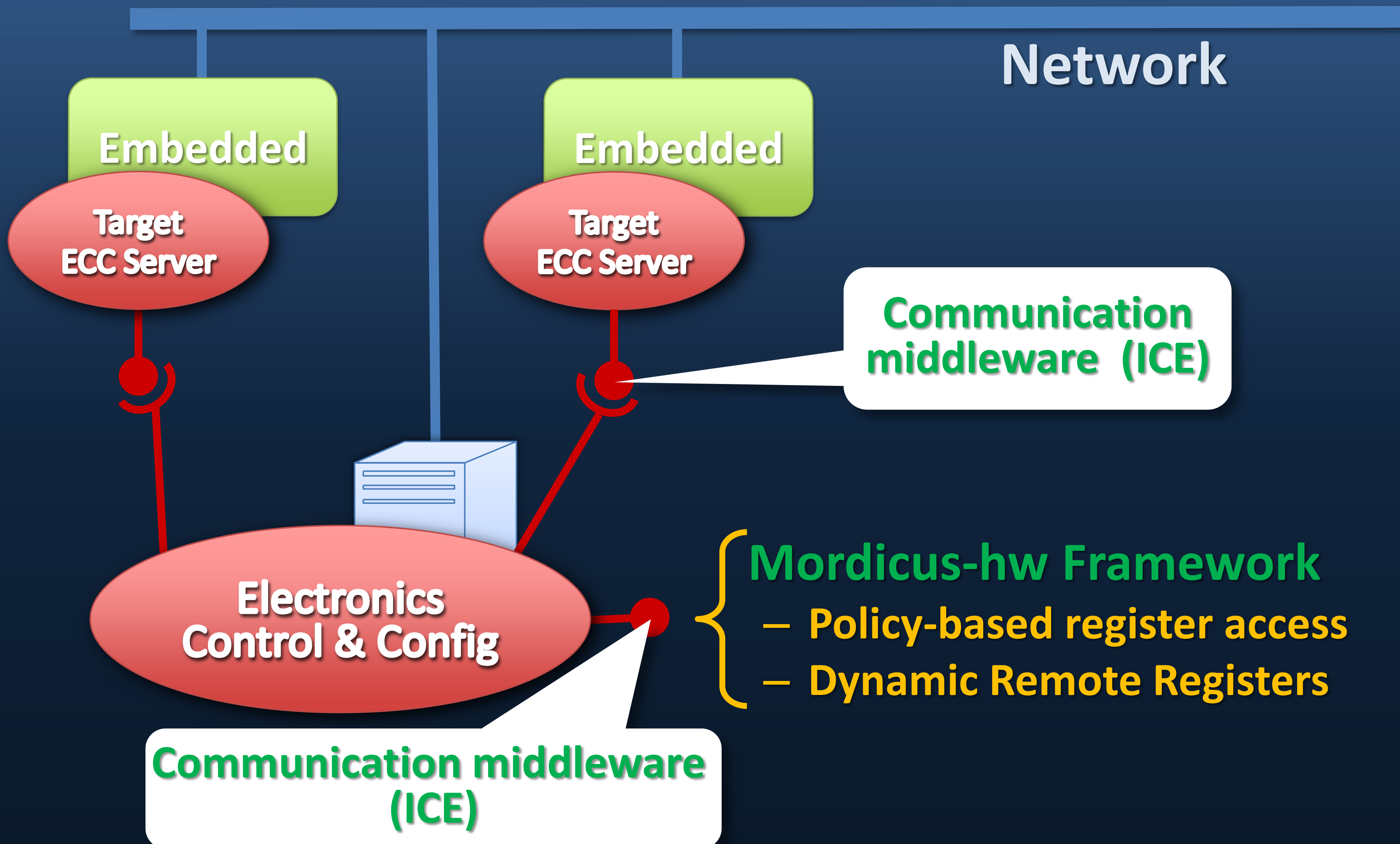
IP Networking

Switched Ethernet

Client-Server over ICE



Electronics Control & Configuration

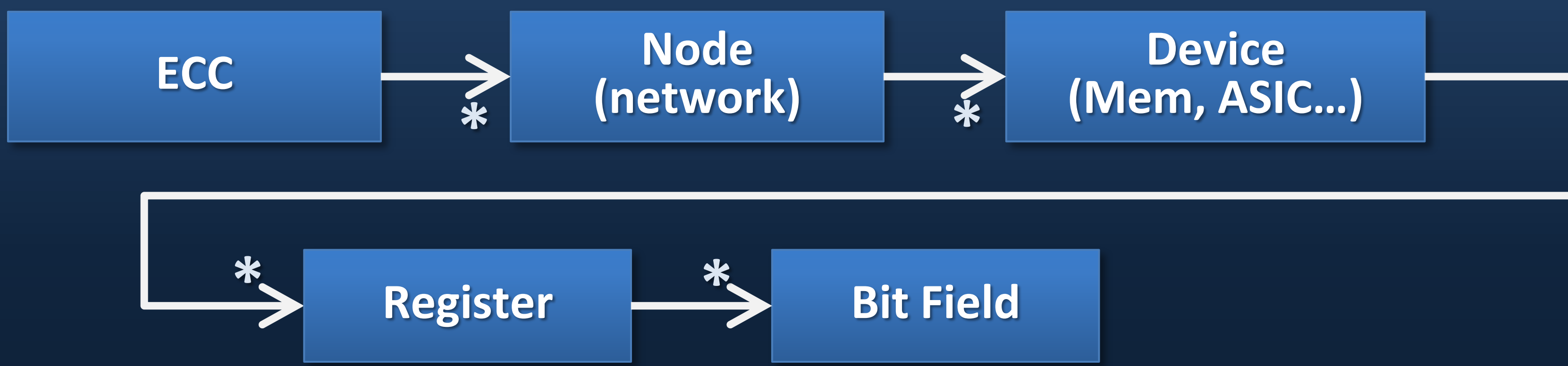


Mordicus-hw Framework
- Policy-based register access
- Dynamic Remote Registers

Electronics Control & Config

Mordicus-HW

- Policy-based register access (C++ templates)
- Dynamic Remote Registers (CConfig framework)



ICE Interface & data definitions
Embedded C++ library (VxWorks & Linux)
Host C++ library (Linux, MacOS)

Optimal collaborative work between electronics and software engineers

Register Access Policies

Every device is associated to a "register access policy" representing the protocol through which hardware registers are read from or written to.

The framework architecture confines the specification of the register access policy to a single Policy class that basically implements the 4 elements:

- the type that represents a register reference: `Policy::AddrType`;
- the data type that is read from / written to the register: `Policy::DataType`;
- the register write function: `void Policy:: poke(const AddrType& addr, const DataType& value);`
- the register read function: `void Policy::peek(const AddrType& addr, DataType& value);`

Scripting tools for electronics design

Based on the described architecture, we can develop powerful clients running on general-purpose workstations capable with dynamic description of target hardware devices and then running any sequence of register accesses in the form of scripts.

Once firmware reaches a sufficient level of maturity, the scripts themselves can be either directly reused or ported to the final system.

Optimization Issues

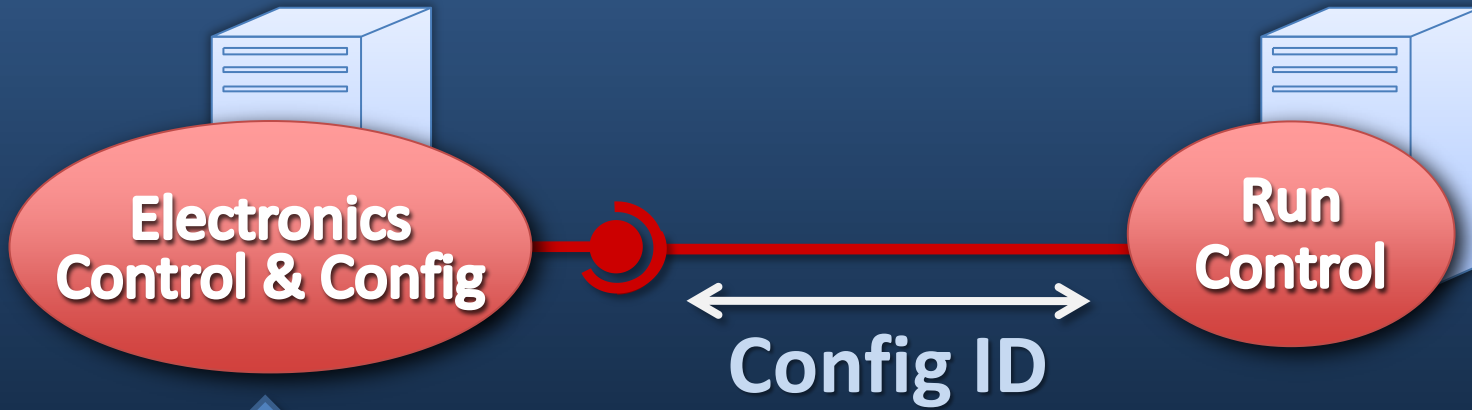
Bit-field access optimization was actually implemented in *Mordicus-hw* resulting in significant acceleration of control sequences.

The caching mechanism uses C++ transient objects which accomplish the single register read in their constructor and the final write-back in their destructor, doing all the bit-field access operations in the form of chained method calls such as (bit-fields are referenced here as strings):

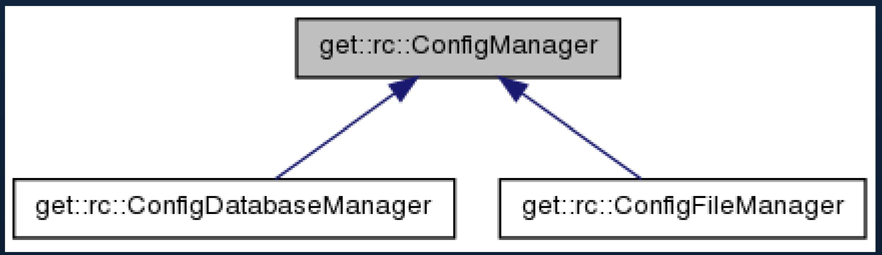
```
myReg.poke("ctrl",2).poke("status",11).poke("cs",1);
```

In this example, myReg is a register object and the first call to the `poke()` method returns the transient object on which, from then on, the subsequent `poke()` calls are made.

Configuration Framework



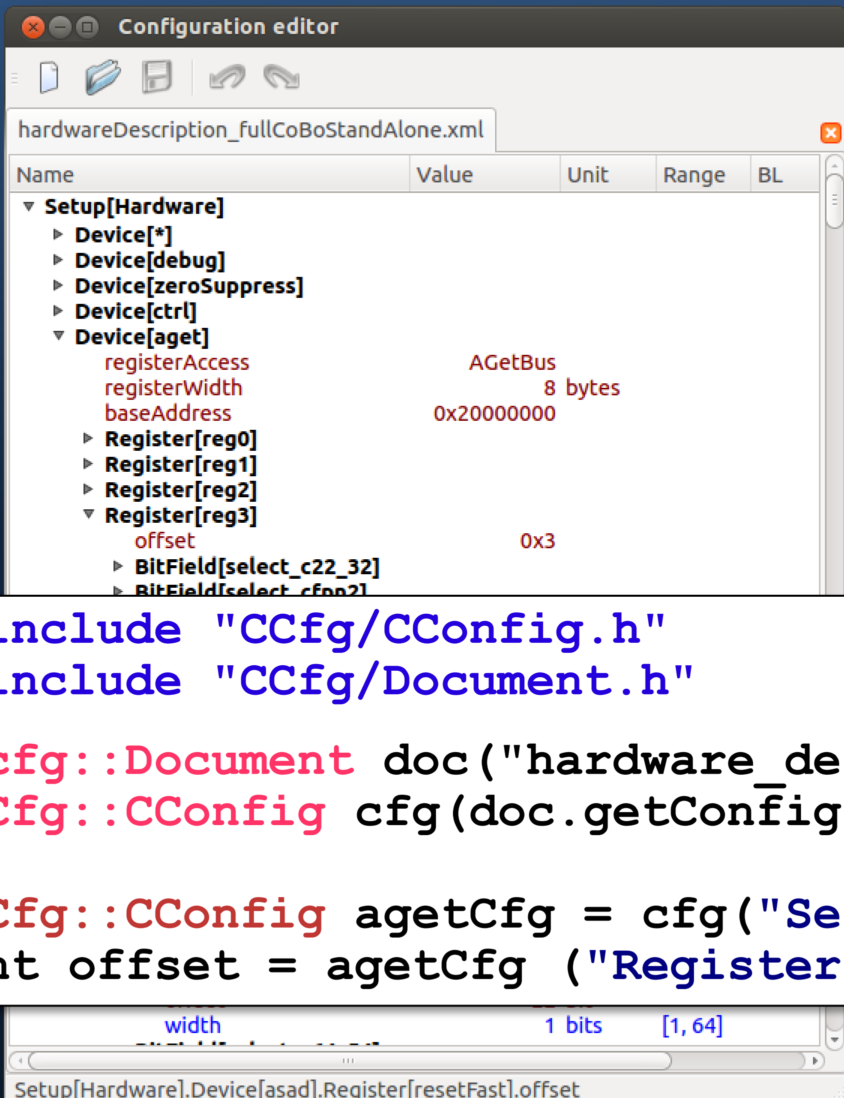
- C++ framework (Std & embedded OS)
- XML File & Database persistence

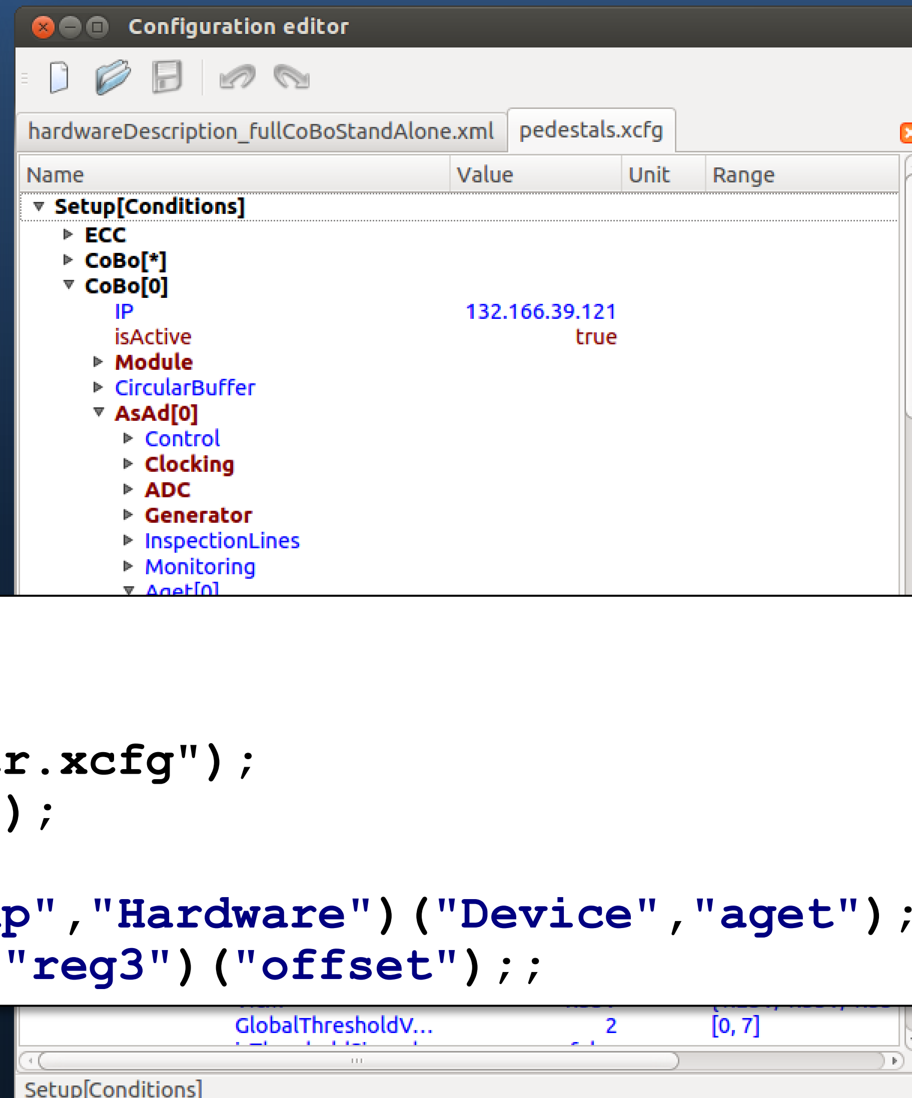


- Graphical Editor (based on Qt4)
- Parameter default value mechanism
- Multilanguage server access (using ICE)

…

Configuration Framework






```
#include "CCfg/CConfig.h"
#include "CCfg/Document.h"

Ccfg::Document doc("hardware_descr.xcfg");
CCfg::CConfig cfg(doc.getConfig());

CCfg::CConfig agetCfg = cfg("Setup","Hardware")("Device","aget");
int offset = agetCfg ("Register","reg3")("offset");;
```

Further developments

- *“Batch” objects*

  Series of remote register access instructions than would be transported in a single network operation to their target node and then locally interpreted and executed.

- *Advanced device parameterization*

  The possibility to instantiate register devices of any kind with an arbitrary number of parameters.

- *Parameters in more than 64-bit values.*

  The possibility to transfer in a single network operation the whole configDB.