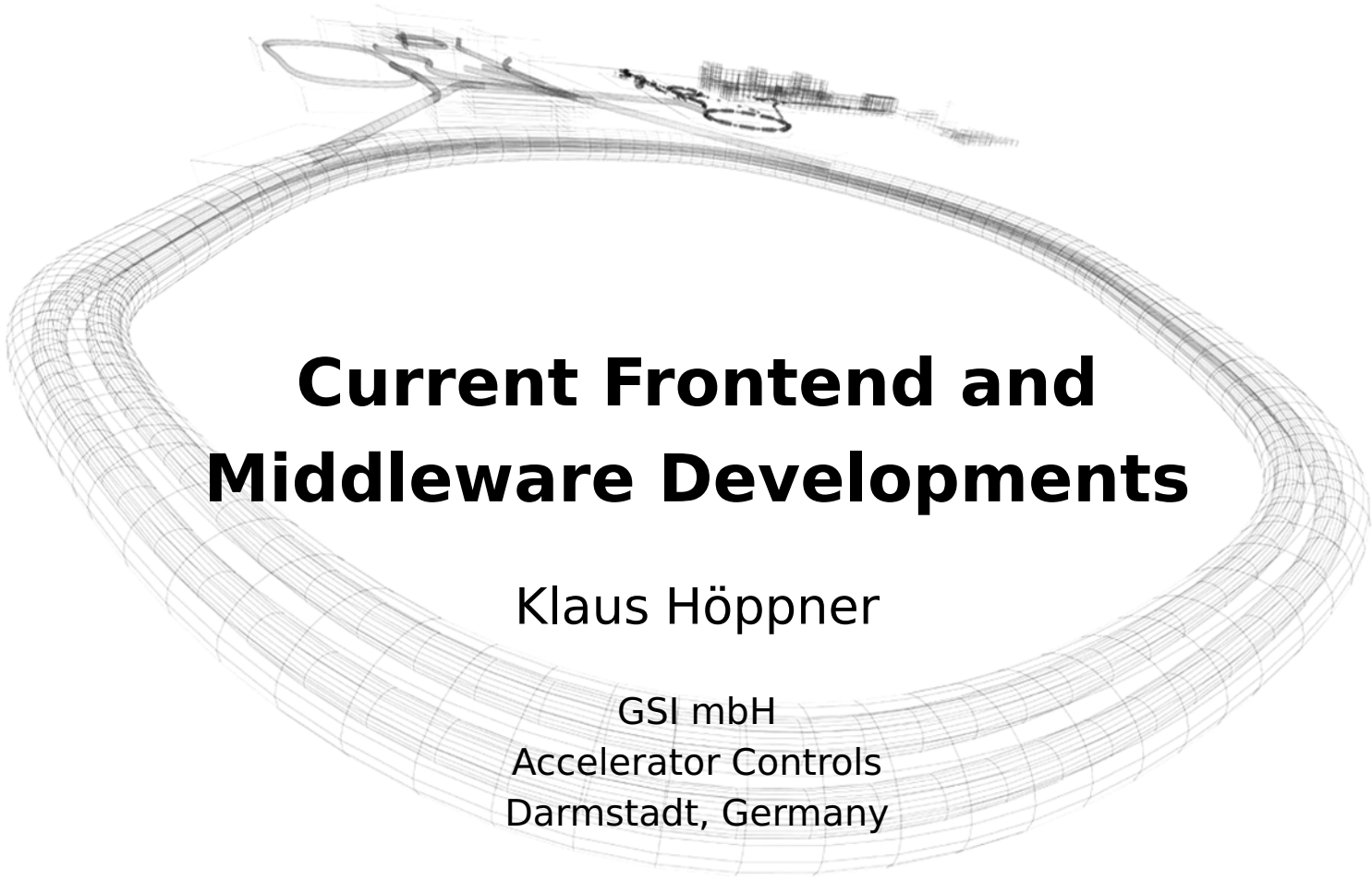


Current Frontend and Middleware Developments

Klaus Höppner

GSI mbH
Accelerator Controls
Darmstadt, Germany

Overview

Replacement of SC Boards

Currently, M68020 boards are used as Supervisor Controllers (SC), running an in-house Micro Operating System (MOPS) on top of pSOS, responsible for the non-realtime controls of devices.

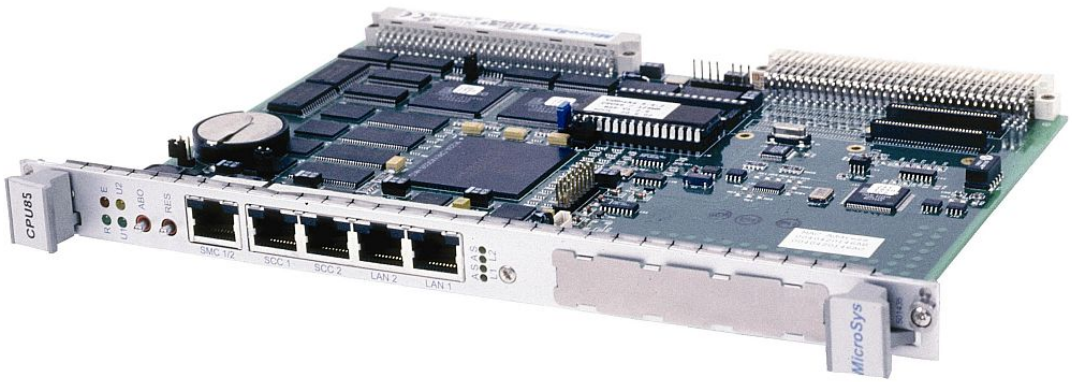
Replacement of these outdated boards is in progress.

Future:

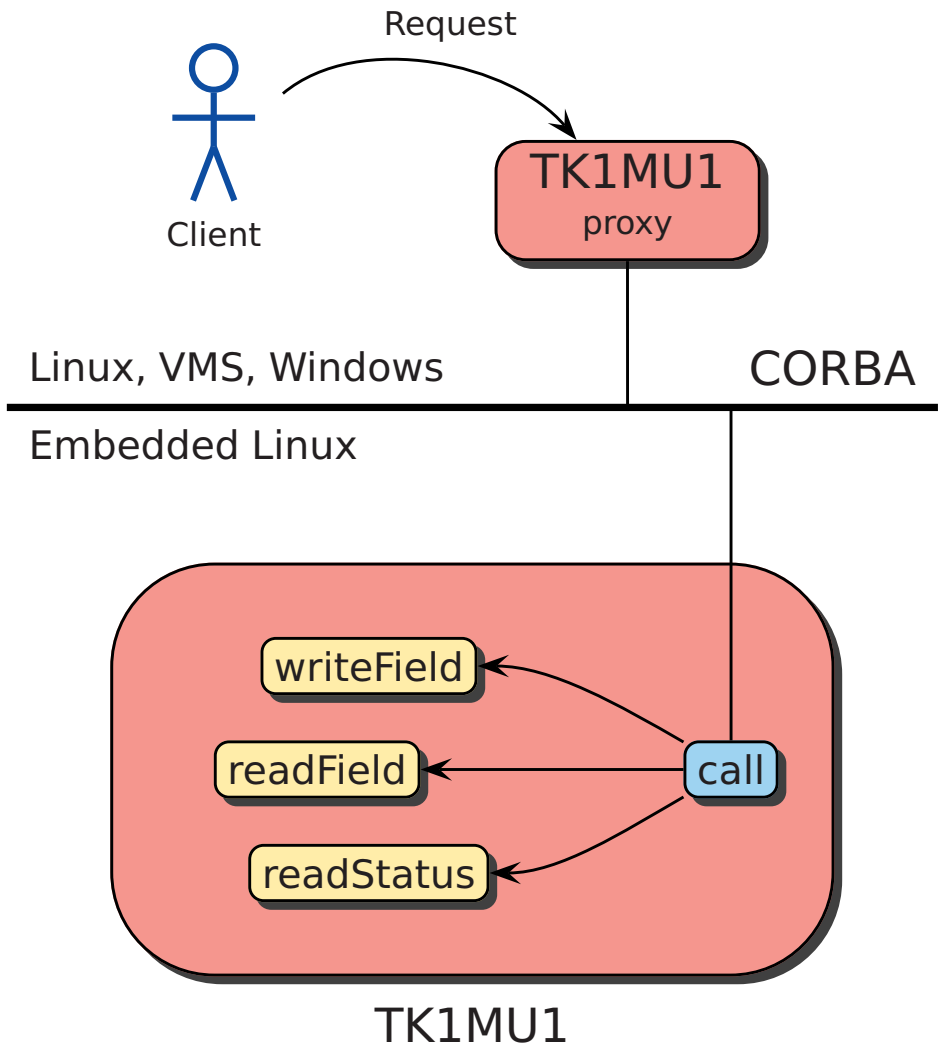
- Single board computers with VME bus
- CPU: PowerPC
- Operating system: Embedded Linux

Microsys PowerPC board with VME bus

- MPC8270 PowerPC CPU
- 450MHz Core Speed
- 256 MB RAM, 64 MB Flash
- 2 $\times$ 10/100 MBit Ethernet
- running Embedded Linux based on Yellowdog Linux, with board support package provided by Denx (`http://www.denx.de`)



Overview: Software Architecture



Architecture

PowerPC supervisor controllers will still act as bridge between realtime device control on ECs and operating level.

→ Separation of non-realtime control on SC and realtime control on EC will remain.

Communication between SC and ECs via dual port RAM (VMEbus)

New: Communication between SC and operating level via CORBA; replacement of in-house communication protocol.

Device Manager

On PowerPC SC a Device Manager (DevMan) is running, that

- parses a device table file, containing a list of devices on SC (name, equipment model, device data)
- creates `AccDevice` device server instances, loading equipment software from shared objects at runtime
- creates CORBA server objects making the AccDevice instances accessible from outside
- registering the device in NameService

CORBA IDL definition

The GSI CorbaInterface IDL file defines a API with a *narrow interface*

for

- synchronous (`read`, `write`, `call`)
- asynchronous (single non-blocking or periodic/event specific read, write and call requests)

communication with devices

Excerpt from IDL file

```
module CorbaInterface {
  // Access to all devices in the control system
  //-----------------------------------------
  interface CorbaIfc {
    // Synchronous actions
    //----------------------
    AccDevErr read(in AccessId access,
                   in Property prop, in long vrtAcc,
                   in AccData para, out AccData data,
                   out AccStamp stamp, out AccEFICD eficd)
        raises(AccDevExc);

    AccDevErr write(in AccessId access,
                   in Property prop, in long vrtAcc,
                   in AccData para, in AccData data)
        raises(AccDevExc);
```

Excerpt from IDL file (cont.)

```
AccDevErr call(in AccessId access, in Property prop,
               in long vrtAcc, in AccData para)
    raises(AccDevExc);

// Asynchronous actions
//-------------------------
AsynchId requestRead(in AccessId access,
             in Property prop, in long vrtAcc,
             in AccData para, in Callback cb)
    raises(AccDevExc);

AsynchId requestWrite(in AccessId access,
             in Property prop, in long vrtAcc,
             in AccData para, in AccData data,
             in Callback cb)
    raises(AccDevExc);
```

Excerpt from IDL file (cont.)

```
    AsynchId requestCall(in AccessId access,
                         in Property prop, in long vrtAcc,
                         in AccData para, in Callback cb)
        raises(AccDevExc);

    // [...]
  };
};
```

Narrow Interface

Read, write and call requests to properties are realized with a narrow interface, sending the name of property, optionally sending parameters, and reading or sending data, both using a universal data container.

Advantage: Flexible interface, new properties can be implemented without change of IDL file and API.

Disadvantage: Parameters and data can be an arbitrary structure of arbitrary data types, that have to be interpreted by property implementation.

Generic Device Access $\leftrightarrow$ Middleware

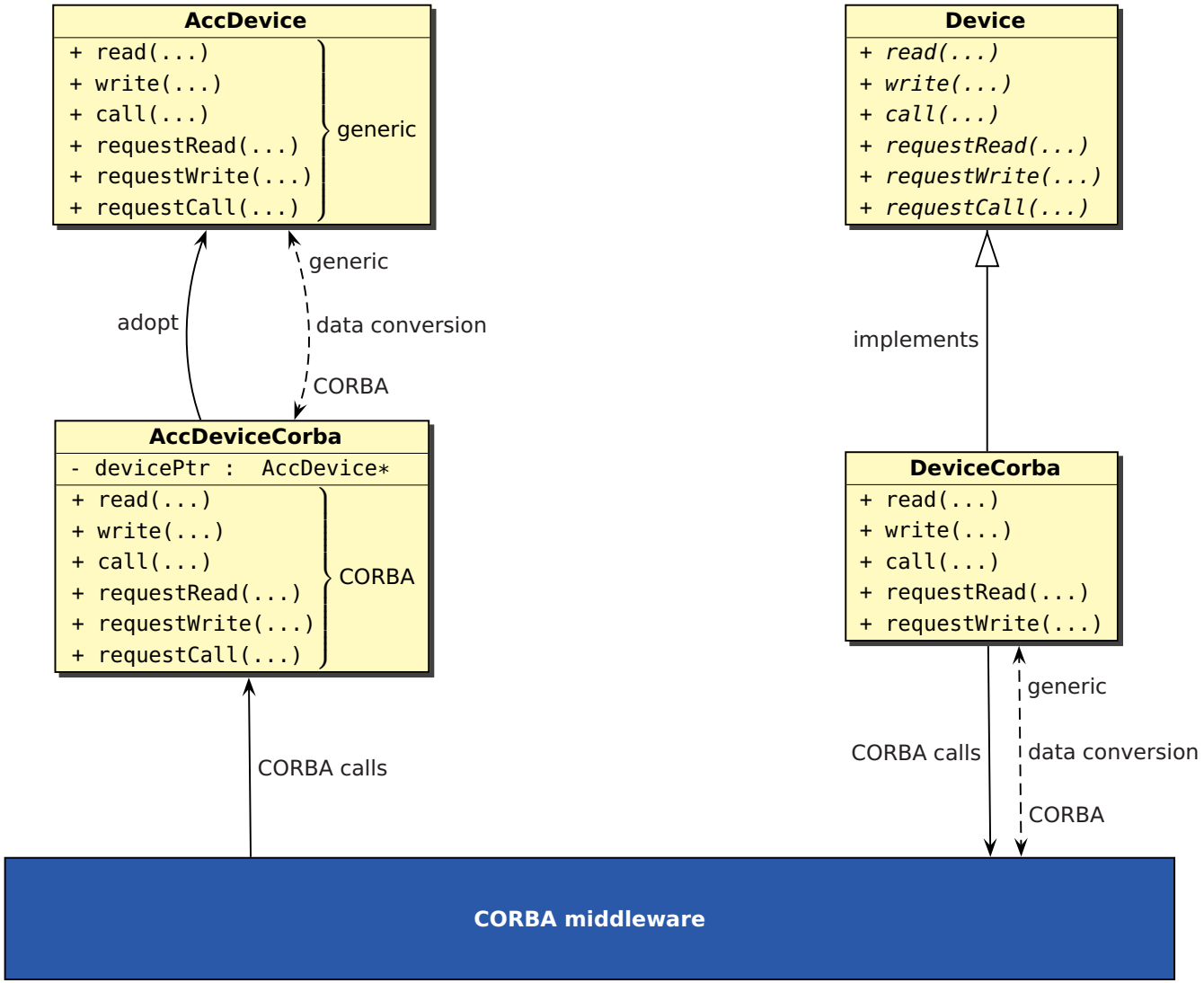
To become independent from a specific middleware/communication protocol, the API for device access on frontend level and middleware API are decoupled.

Property implementation for device access use a generic API for read, write and call, using a generic `AccData` container.

Device Manager creates CORBA server objects using an adapter class, listening for read, write and call requests from clients and translating them to generic API.

Advantage: Replacing middleware or using other communication protocols will be possible without change in equipment software.

Illustration: Decoupling from Middleware



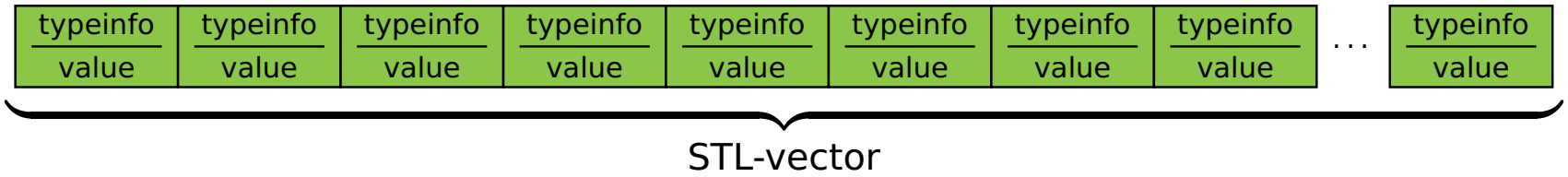
Example: AccData container

Generic Device Access: The generic device access API uses an `AccData` container similar to `vector` class from STL, implementing index operator, iterators, and providing methods to access data elements (with and without data type conversion).

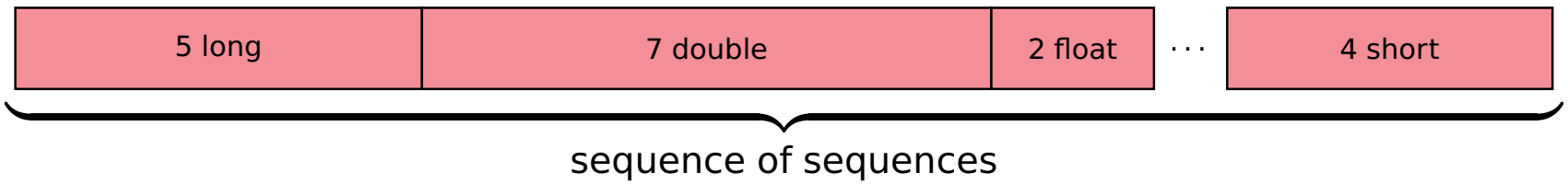
CORBA IDL/API: CORBA specific `AccData` is a *sequence of sequences*, i. e. a container of type “5 long, 10 float, 2 unsigned short, 3 double”.

Illustration: AccData

Generic AccData:



CORBA specific AccData:



Example Code: Generic AccData

```
AccData data;
data.push_back( (long) 7 );
data.push_back( (double) 3.1 );
try {
    // access without conversion: exception if different data type
    cout << data[1].value<double>() << endl;
    // access with conversion:
    // exception if value out of target range
    cout << data[0].convert<short>() << endl;
    // STL like iterators
    for (AccData::iterator it=data.begin(); it!=data.end(); ++it) {
        cout << it->convert<double>() << endl;
    }
} catch (const AccDevException& e) {
    cerr << e.what() << endl;
}
```

Decoupling from VME bus

M68k legacy software is dependent on in-house communication protocol *and* VME bus (dual port RAM access, VME interrupt handling).

New `AccDevice` class for device server objects is decoupled from VME bus by delegating bus dependent functionality to a `DeviceSupport` pointer, pointing to an instance of a hardware/bus specific child class of `DeviceSupport`.

As proof of principle, we created devices (simulated magnets) running on a normal x86 Linux PC.
So, connecting non-VME devices handled by a normal PC will be possible by simply installing a x86 Device Manager and omniORB.

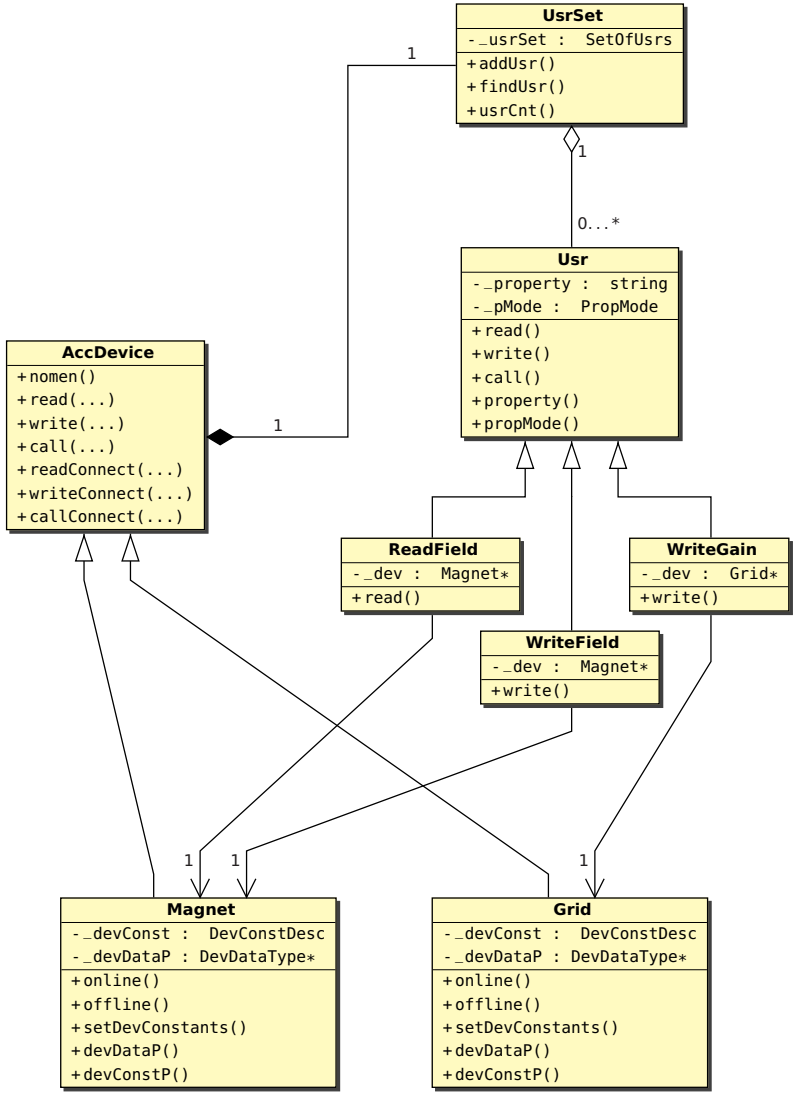
Reusing Existing USR Code

Functionality of equipment model software on SC side is implemented using a set of pointers to USR objects, each providing a (read, write or call) property.

Property specific USR classes can be rewritten from existing M68k USR code, reusing about 80 % of old code, while code changes are mainly limited to reflecting the new API by reading and filling generic `AccData` container and using the `DeviceSupport` pointer for bus specific operations.

Transition process of a USR is done in a standardized way following a “cookbook”.

Illustration: Device Access using old USRs



Accessing a property (`read`, `write`, `call`—implemented in base class `AccDevice`):

- find in `UsrSet` (set of pointers to USRs) a pointer to a USR object with requested property name and property mode,
- execute `read`, `write` or `call`, respectively, for the object referenced by the USR pointer,
- where the object is an instance of an equipment model specific child class (e. g. `WriteField`), implemented using legacy USR code.

Side View: Client APIs – C++

Currently, most progress was done for the C++ client API.

It is decoupled from specific middleware by accessing devices via an abstract `Device` class, that define synchronous and asynchronous `read`, `write` and `call` methods using generic `AccData` container.

Concrete implementation is done in CORBA specific child class `DeviceCorba`, that translates the property access to the CORBA specific method calls and converts parameters to CORBA `AccData` before sending and received data back to generic `AccData`.

Client APIs (cont.) – Python, Java

A Python module provides device access for clients, currently mainly used for test purposes. It depends on omniORBpy libraries for CORBA–Python bindings, decoupling of client API and CORBA has to be done in the future.

→ Usage of Python as scripting language in accelerator controls, possibly as a successor of Nodal

Client Java API is implemented by Cosylab, together with some demo applications. API still dependent on CORBA, decoupling API from middleware has to be done. As first step, a generic `AccData` container was implemented in Java, similar to generic C++ container.

Tool Chain

- Manual transition from existing USRs (M68k) to PowerPC USRs
- Set of Makefiles for
  - Compiling and linking libraries, shared objects, executables; creating dependency files by just specifying list of source files
  - Copying files to system directories
  - Nightly build of complete software from current development trunk
- Set of scripts (mainly Python) for supplying PowerPC SCs with equipment model software
- Subversion as software archive and version control system

Future Plans

- Automatic creation of USR source/header file skeletons from XML definition of new equipment models
- Device Manager configuration (device list, device constants) using XML files
- Introduction of new property: Clients can get information about device constants (e. g. hard/soft limits) directly from a device object (request from Cosylab)
- Access Rights