MicroHAL - Powered by Redwood

Hardware Access Library for TCP/IP or UDP/IP

Greg Iles on behalf of Andy Rose & Rob Frazier

7 March 2011

What is this all about?

- xTCA has no standardized protocol for register access
- Common method to access registers
- J. Mans (CMS HCAL) proposed a packet format specification compatible with TCP and UDP
  - Also supplied firmware core (UDP)
- **MicroHAL** – Advanced software layer for register access
  - **Redwood** package provides infrastructure

Simple example

- FPGA with multiple registers on a bus

```
<chip name="FPGA1">
        <leaf name="LEDs" address="0x0010" />
        <leaf name="epim_lut" address="0x00100000" />
        <leaf name="ecal_lut" address="0x00100800" />
        <leaf name="hcal_lut" address="0x00101000" />
</chip>
```

This is as far as the current HAL table goes

Hierarchical

- Include calls to other module files

```
<chip name="FPGA1">
  <leaf name="LEDs" address="0x0010" />
  <array count="12" spacing="0x00004000" >
    <module path="examples/algo.xml"
            name="algo"
            address="0x0000"/>
  </array>
</chip>
```

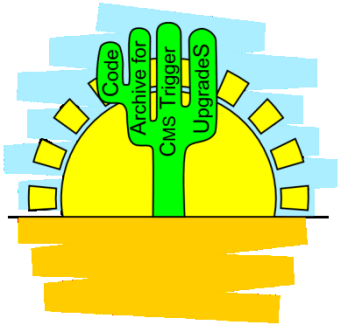
- Where examples/algo.xml contains

```
<module name="algo" mask="0x001FFFFF" >
  <branch name="module_algo">
    <leaf name="epim_lut" address="0x00000000" />
    <leaf name="ecal_lut" address="0x00000800" />
    <leaf name="hcal_lut" address="0x00001000" />
  </branch>
</module>
```

Also a lot more, but must go...

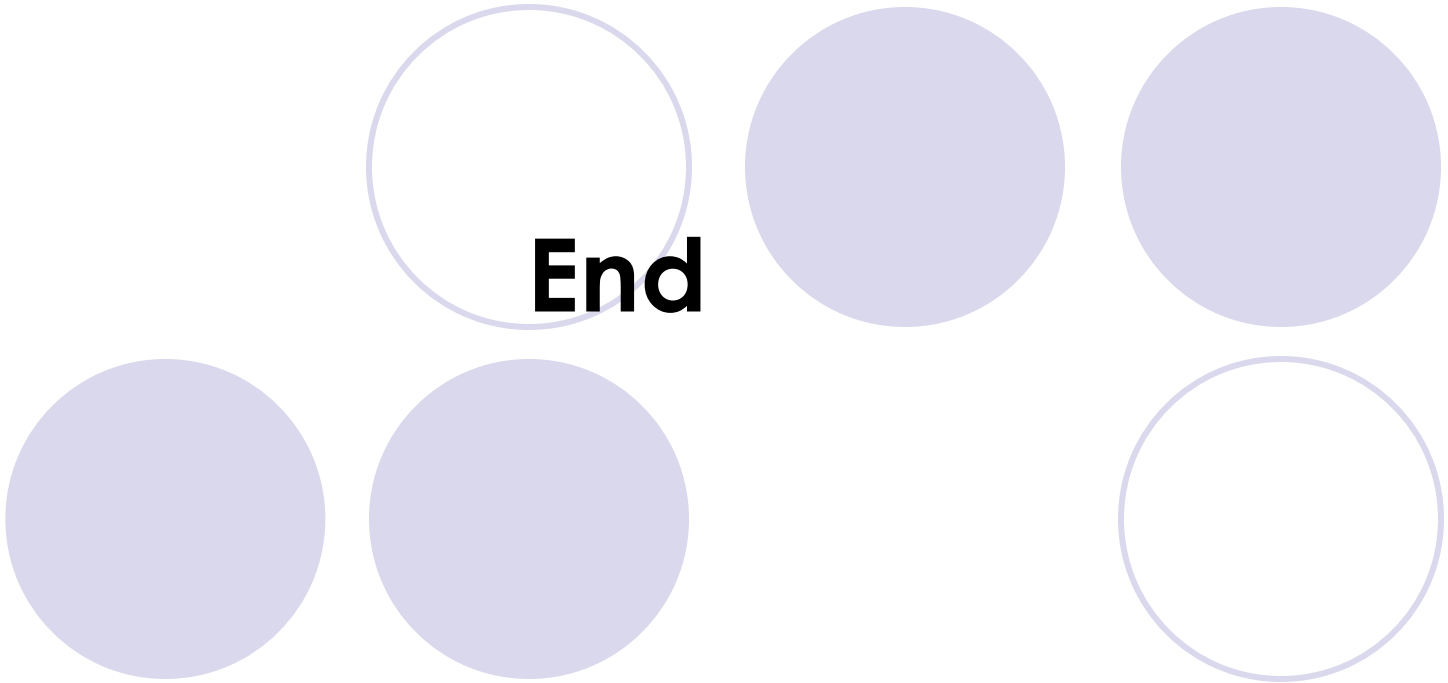
- Supports all basic commands
  - e.g. Read/Write, RMW, Block Read/Write
- Automatically concatenates commands together
  - Important given IP latency
  - Commands executed sequentially by firmware or micro-processor
- Hierarchical nature
  - Allows modular design
  - e.g. VHDL design accompanied by XML address table and C++ code.
  - Reference address table and include C++
- Scalable design (not discussed here)
- Software has full knowledge of FPGA memory / registers
  - Can map onto database

General information



- Project website
  - http://projects.hepforge.org/cactus/index.php
- HepForge repository
  - http://projects.hepforge.org/cactus/trac/browser/trunk
- Software User Manual, Instant Start Tutorials and Developers Guide
  - http://projects.hepforge.org/cactus/trac/export/148/trunk/doc/user_manual/Redwood.pdf

More info: Please contact Andrew Rose: andrew.rose01@imperial.ac.uk


End

CMS Electronics Upgrade Software and Firmware Framework for Off Detector Control and Configuration

Andrew W. Rose

Interpreted and Presented by Magnus Hansen

Disclaimer

- I am not the expert on this subject, thus please bear with small errors etc. By the way, please bear with big errors too...
- Questions may have to be addressed to Andrew Rose in order to be correctly answered
- Please TAKE NOTES and send them to me (magnus.hansen@cern.ch)such that all questions eventually get answered and answers published together with these slides on Indico.

What is this all about?

- xTCA has no standardized protocol for register access
- The temptation is large to go ahead and develop personalized solutions to the "problem", leading to a large number of different register access methods, all which need to be maintained long term
- In order to start the work towards a common software framework J. Mans, who had already started working with the CMS HCAL off-detector upgrade in mind, was asked to propose a specification of a packet format compatible with TCP and UDP. So was done, and a firmware package was also supplied together with a clear statement that long term maintenance could not be supplied.
- The initial specification was picked up by CMS UK in view of CMS trigger upgrades and developed further.

- And now to the Talk...

Philosophy

If a board's firmware is built from library modules, then so should its corresponding software be.

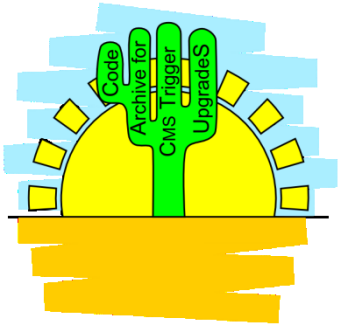
Infrastructure tasks quickly become boring.

Aims

- Limit code duplication by promoting module reuse
- All infrastructure code transparent to the end user
  - details of network architecture
  - details of transport layer
  - database interfaces
  - trigger supervisor interface
- All code fully documented
- Each class includes a formal unit test

n.b. The End User is the application code developer

General information

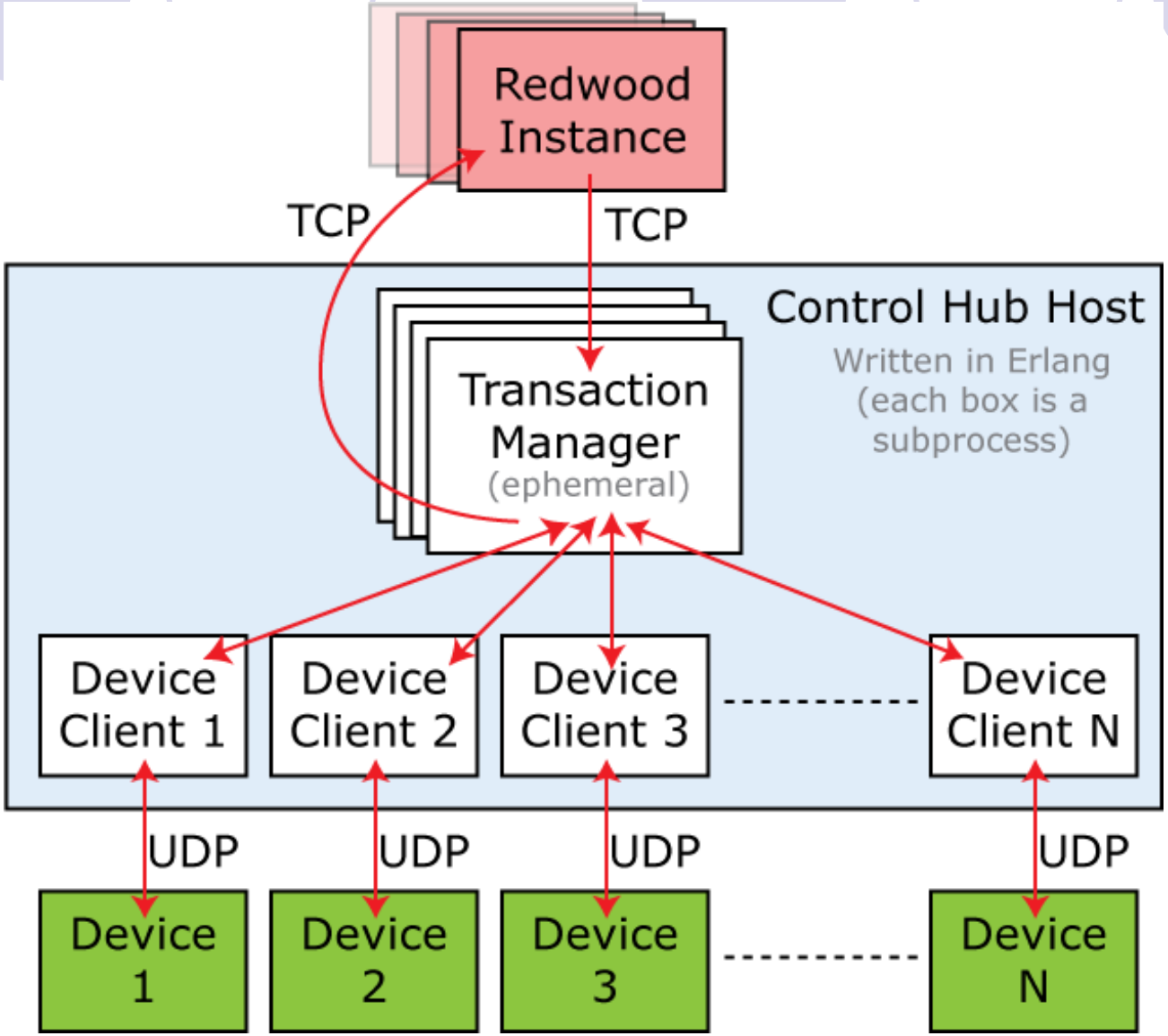


- Project website
  - http://projects.hepforge.org/cactus/index.php
- HepForge repository
  - http://projects.hepforge.org/cactus/trac/browser/trunk

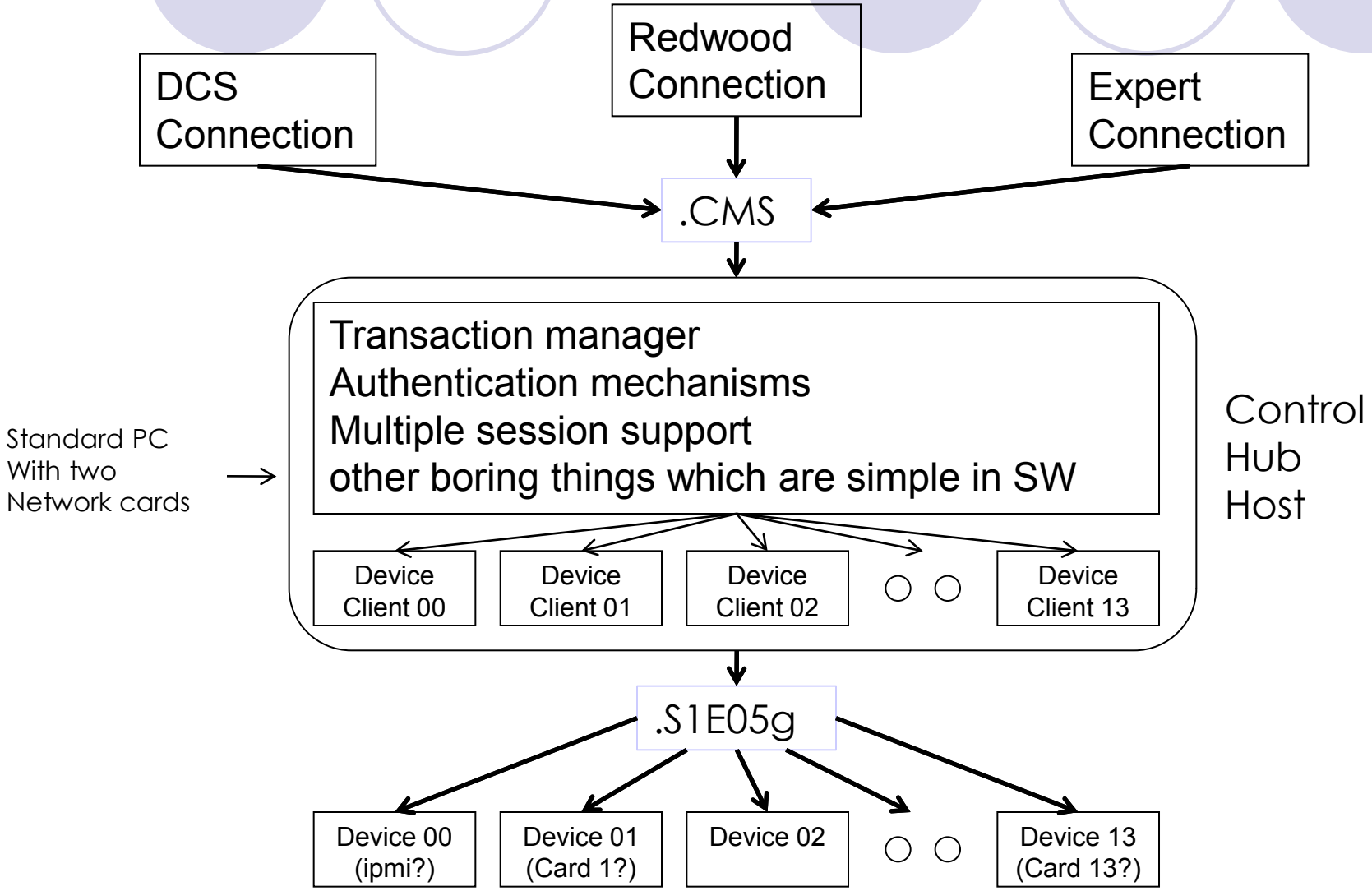
- Redwood & co.

  The Software User Manual, Instant Start Tutorials and Developers Guide
  http://projects.hepforge.org/cactus/trac/export/148/trunk/doc/user_manual/Redwood.pdf

Overview


Redwood Instance
TCP
TCP
Control Hub Host
Written in Erlang (each box is a subprocess)
Transaction Manager (ephemeral)
Device Client 1
Device Client 2
Device Client 3
Device Client N
UDP
UDP
UDP
UDP
Device 1
Device 2
Device 3
Device N

Another Overview



Control Hub Host

- Basic code is complete!
  - Full chain is working:
  - Redwood → CHH → Device → CHH → Redwood
- Now undergoing more rigorous testing:
  - Scalability/throughput testing:
    - Increasing size/complexity of transactions
    - Increase number of Redwood clients
    - Increase number of hardware devices
- Once happy with scalability/throughput…:
  - Further work on "bulletproofing" + error reporting to do.
  - Control Hub Host needs to both very reliable and transparent for end users.

Firmware

Core Firmware

- Work from Dave Newbold and Andrew Rose based on the original work of Jeremy Mans and Greg Iles

- Aim is to produce a library of out of the box components. This has meant:
  - Splitting the EMAC from the IP comms module so that the comms module can be used for boards without hard TEMAC
  - Work is ongoing on an ICMP reset module

    i.e. very hard remote reset of board subsystems, independent of the bus
  - IPbus module needs discussion (so we shall)…

IPbus Firmware

- There are known issues with the original verilog firmware
- There is a VHDL version from Eric Hazen which is not tested to date
- A working VHDL version has been developed and is currently accessible in a semi-public repository
- UK-CMS Upgrades is willing to commit to supporting it long term, i.e. succession will be organized

IPbus Firmware

- The available VHDL version attempts to make the bus self-consistent and so differs in the reply word-count for the write command
- An error protocol is defined
- Software handles both the original and the current implementation by setting of a bus version flag at compile time
  - But not Eric Hazen's version at this point

Firmware conclusion

- Rather than three different versions, we should agree on using one.
- Is UK offer to assume responsibility for IPbus acceptable?

Software

- Firmware is intrinsically hierarchical
- Would like some way of treating software similarly

Software structure

So how does this all fit together?

which is constructed from lower level modules down to registers

which references other XML files and module definitions down to the register description

which is a container for child branches and leaves, which represent registers on the bus

VHDL/verilog

XML

Object-Oriented C++

XML example

- FPGA with one register on a bus

```
<chip name="FPGA1">
        <leaf name="LEDs" address="0x0010" />
</chip>
```

Leaf is the name for register

XML example

- Can also give labels to bits or groups of bits within each register / leaf

```
<chip name="FPGA1">
  <leaf name="LEDs" address="0x0010" >
    <bitfield name="a" offset="0" width="1"/>
    <bitfield name="b" offset="1" width="1"/>
    <bitfield name="c" offset="2" width="2"/>
    <bitfield name="d" offset="4" width="2"/>
  </leaf>
</chip>
```

XML example

- FPGA with multiple registers on a bus

```
<chip name="FPGA1">
        <leaf name="LEDs" address="0x0010" />
        <leaf name="epim_lut" address="0x00100000" />
        <leaf name="ecal_lut" address="0x00100800" />
        <leaf name="hcal_lut" address="0x00101000" />
</chip>
```

This is as far as the current HAL table goes

XML example

- FPGA with multiple registers on a bus, divided into subsystems

```
<chip name="FPGA1">
  <leaf name="LEDs" address="0x0010" />
  <branch name="module_algo"  address="0x00100000"
                              mask="0x000FFFFF" >
    <leaf name="epim_lut" address="0x00000000" />
    <leaf name="ecal_lut" address="0x00000800" />
    <leaf name="hcal_lut" address="0x00001000" />
  </branch>
</chip>
```

branch is defining hierarchy

XML example

- FPGA with multiple identical (ex. address) subsystems on a bus

```
<chip name="FPGA1">
  <leaf name="LEDs" address="0x0010" />
  <array count="12" spacing="0x00004000" >
    <branch name="module_algo">
      <leaf name="epim_lut" address="0x00000000" />
      <leaf name="ecal_lut" address="0x00000800" />
      <leaf name="hcal_lut" address="0x00001000" />
    </branch>
  </array>
</chip>
```

- 12 copies of module_algo, with addresses spaced by 0x4000
- Labelled in OO model as “module_algo[0]” to “module_algo[11]”
- c.f. VHDL generate statement

Array is the key for multiple instances

XML example

- Include calls to other module files

```
<chip name="FPGA1">
  <leaf name="LEDs" address="0x0010" />
  <array count="12" spacing="0x00004000" >
    <module path="examples/algo.xml"
            name="algo"
            address="0x0000"/>
  </array>
</chip>
```

- Where examples/algo.xml contains

```
<module name="algo" mask="0x001FFFFF" >
  <branch name="module_algo">
    <leaf name="epim_lut" address="0x00000000" />
    <leaf name="ecal_lut" address="0x00000800" />
    <leaf name="hcal_lut" address="0x00001000" />
  </branch>
</module>
```

Software structure

So how does this all fit together?

which is constructed from lower level modules down to registers

which references other XML files and module definitions down to the register description

which is a container for child branches and leaves, which represent registers on the bus

What is missing?

→ Specialization of functionality

Specialization of functionality

- Split operations into two types:
  - standard operations:
    - **CONFIGURE** , **ENABLE** , **SUSPEND** , **RESUME**
  - custom – for example, bench-top tests

- Each Redwood branch can hold an OperationObject for each of the standard operations
  - Written in C++ (but other languages in principle possible)
- For custom set-ups, the user has full control over everything from their test executable

Specialization of functionality

- Split operations into two types:
  - standard operations:
    - **CONFIGURE** , **ENABLE** , **SUSPEND** , **RESUME**
  - custom – for example, bench-top tests

- If a branch needs to do nothing for a particular operation:
  - no OperationObject is necessary
  - the branch just invokes that operation on its child branches

XML example

- FPGA with a self configuring branch

```
<chip name="FPGA1">
  <branch name="module_gtx"   address="0x00100000"
                              mask="0x000FFFFF" >
    <configure plugin="gtx_config" />
    <enable plugin="gtx_enable" />
    <suspend plugin="gtx_suspend" />
    <resume plugin="gtx_resume" />
    <leaf name="…" address="0x00000000" />
        ⋮
  </branch>
</chip>
```

Specialization of functionality

- As such, the engineering team writes:
  1. A firmware module
  2. OperationObjects to perform standard operations
  3. An XML description file for the module

- Unless the user wishes to do something non-standard, they may never need to look into the guts of any of the three.
  - Can just call `some_module_name.configure()`, etc.

Software conclusion

- What is needed now is
  - Agreement about principle with a common access structure and that this is a good attempt
  - Try it out!
  - Give Constructive Feedback