

# Controlling $\mu$ TCA Hardware

**Robert Frazier**

**Greg Iles, Marc Magrans, Dave Newbold,  
Andrew Rose, Dave Sankey, Tom Williams**

**Institutes: Bristol, Imperial, RAL, CERN**



# Talk Overview

- **Objective**
- **The basics:**
  - IPbus Protocol
  - Basic design principles, decisions and constraints
  - IPbus Firmware overview and status
- **Frequently Asked Questions on the design**
- **Software and use-cases**
- **Testing IPbus**
- **Issues still to resolve**



# Objective

- **A mechanism to control  $\mu$ TCA-based hardware**
  - Replacing the VME HAL, and associated CAEN drivers/controllers
- **I think it's safe to say:**
  - It's best if we're in full control of this mechanism
    - Whatever it may be...
  - CMS has to operate hardware on much longer timescales than industry
    - The solution still has to work in 10+ years' time.
    - Must avoid industry-based proprietary solutions/components
    - Stick to established standards to avoid any lock-in.



# Introducing IPbus

- **IPbus Protocol** – for controlling hardware via IP over Ethernet
  - UDP (or TCP, etc) as the transport protocol
  - Originally developed by J. Mans, *et al.*
- **Implementing this protocol, we have:**
  - ***IPbus Firmware*** (VHDL) – using UDP as the transport protocol
  - ***μHAL*** – C++ HAL for end-users to build upon
- **Supplementary to this, we also have:**
  - ***ControlHub*** – to form a single point of contact with the hardware
    - Currently implemented in Erlang
  - ***PyChips*** – a simple Python HAL (mainly for non-CMS use)

# Development effort and external interest



- **We already have reasonable size team from multiple institutes**
  - With substantial current/previous experience in online software
- **UK is prepared to take a substantial role in the long-term support**
  - Assuming continued funding for upgrade activities
- **IPbus has generated much external interest:**
  - Already in use for the AIDA test-beam project
  - Baseline solution for the Atlas L1 trigger upgrade (ATCA)
  - Daresbury/CERN accelerator control groups
  - Several smaller projects



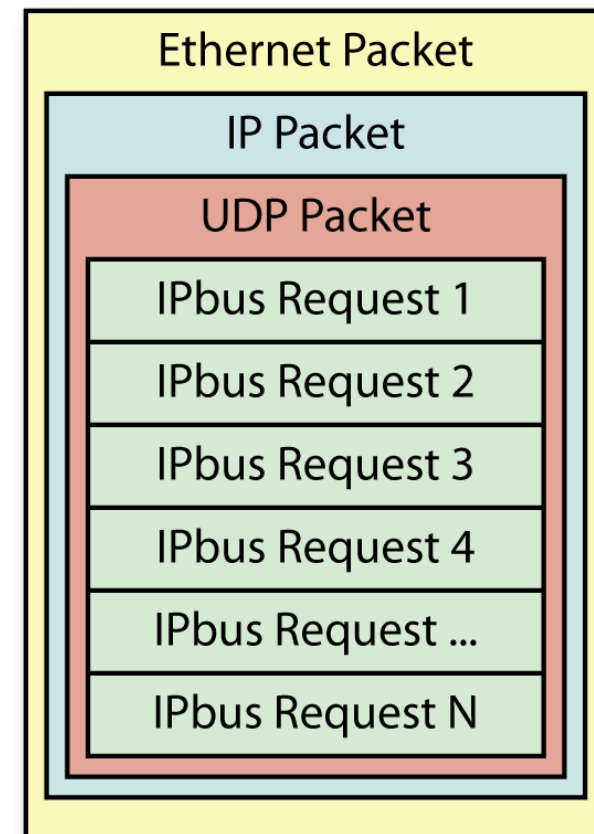
# IPbus Protocol basics 1

- **The protocol describes the basic transactions:**
  - Read
  - Write
  - Non-incrementing Read/Write
  - Atomic Masked Write (Read/Modify/Write)
- **A32/D32**
  - Word-addressable, not byte-addressable
  - 16 GiB maximum addressable space per AMC



# IPbus Protocol basics 2

- **Each transaction request/response is self-contained**
  - Has its own header and body
  - Transport protocol agnostic
- **Transactions can be concatenated together into the same packet**
  - Queue requests and dispatch when necessary
  - Improves network transport efficiency
  - Major difference to VME!



# Design principles, decisions and constraints



- I shall discuss these as we go...
- However the major ones I list quickly here:
  1. **No proprietary hardware/firmware/software from external sources**
    - Unless based on established/commonplace standards or components
  2. **Where possible, complexity should be pushed into software**
    - Keep hardware and firmware as simple as possible
  3. **Where possible, complexity should be pushed away from end-users**
    - Simple API
    - Users should not be dealing with resource locks, threading, etc.
  4. **Flexibility and scalability must be built in**
    - Bench-top testing all the way up to full Point 5 deployment!





# IPbus Firmware basics 1

- **Implemented in VHDL**
  - No usage of Microblaze or other CPU
  - IPbus transactions delivered directly to FPGA
    - Slaves attached to simple 1 Gbit/s parallel bus
  - UDP as the transport protocol
- **Multiple concrete implementation examples available**
  - Xilinx only – no Altera yet
  - Example implementations:
    - SP601 (Spartan 6)
    - SP605 (Spartan 6)
    - Avnet AES-V5FXT-EV30 (Virtex 5)
    - Mini-T (Vertex 5)



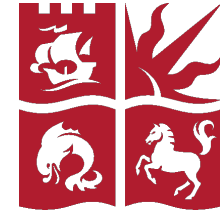
# IPbus Firmware basics 2

- **Small footprint**
- **Real-world resource usage in a low-end FPGA**
  - Xilinx Spartan 6 (XC6LX16-CS324) FPGA
  - As found on the SP601 evaluation board

Resource	Usage
Registers	7%
Lookup Tables	18%
Block RAMs	10%

# IPbus Firmware

## - current status



- A stable release has been available for many months now
  - <https://projects.hepforge.org/cactus/trac/wiki/IPbusFirmware>
  - Implements IPbus protocol v1.3
- **Current limitations:**
  - Standard-sized (1500 byte) Ethernet frames only
  - Single packet-in-flight per board
    - Implication: there must be a single controlling process per board
  - Bandwidth limited to about ~60 Mbit/s per board.

# IPbus Firmware

## - developmental



- **In final testing stages:**
  - Supports Jumbo Frames
  - Still a single packet-in-flight per board
  - Read/write bandwidth to a single board measured at 155 Mbit/s
- **Early development phase:**
  - Multiple packets in flight on read
  - Aiming for 1 Gbit/s read bandwidth
- **Draft specification:**
  - Reliability/retry mechanism

# Firmware design considerations and FAQ 1



- **Implementing complex network protocols in firmware is non-trivial**
  - Rules out TCP
  - Probably rules out things like DHCP
    - Overkill anyway?...
  - However, UDP *is* possible to implement in firmware...

# Firmware design considerations and FAQ 2



- ***“But... isn’t UDP an unreliable protocol?!”***
  - Strictly, yes – it is unreliable
  - In reality, for our use-case, we lose about **1 in 200 million** packets
    - Simple network topology – very different to a LAN/Internet, etc.
    - Network is “owned” solely by control software
- ***“Ok... but I still don’t think that’s good enough”***
  - IPbus is still evolving
    - We are adding a mechanism to cope with packet loss.
    - Many real-world examples of reliability on top of UDP (*cf.* UDT)
  - Note that:
    - Once a packet has been delivered, UDP is just as good as TCP
    - Same level of CRC checking

# Firmware design considerations and FAQ 3



- ***“So, why not use an on-chip, on-board, or on-crate CPU running a lightweight Linux TCP/IP stack? Then I can use TCP!”***
  - Certainly possible...
    - Solves the transport layer packet-loss “issue” (overkill?!)
    - Doesn’t solve access arbitration, transaction serialisation, etc
    - TCP has higher latency and lower bandwidth
  - We feel this route adds complexity too near to the hardware
  - It’s easier to develop/maintain software on a commercial rack PC
    - Using a standard SLCx Linux distro
    - Safer in the long term

# Firmware design considerations and FAQ 4



- ***“What about using protocol X, Y, Z, or use that UDT thing you mentioned?”***
  - These other protocols (SCTP, AoE, FCoE, iSCSI, ...) are not widely supported in software
    - See design principle number 1
  - Also, protocols such as SCTP, UDT require big buffers for reliability
    - Block RAM is a limited resource, particularly in Spartan FPGAs
  - If we use UDP, we have to make our own 100% reliability solution
- ***“Ok, well I still don’t like it. We’re still gonna use TCP with a CPU”***
  - Fine - our software already supports IPbus over TCP ☺





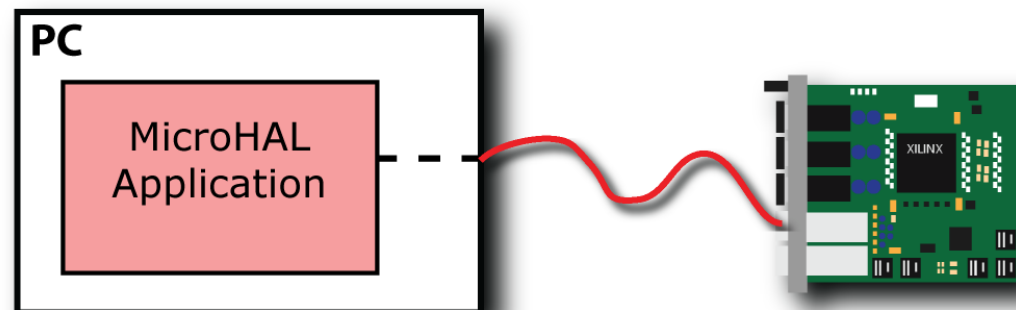
# Software: $\mu$ HAL

- **$\mu$ HAL is a C++ front-end to IPbus**
  - Supports TCP & UDP as the transport protocol
  - Designed to support the recursive modularity of your firmware
    - Hierarchical address tables (*i.e.* a tree, not a flat address space)
- **API is currently under wider review from various parties**
  - See Marc's talk
  - Please get involved if you are interested:
    - <https://svnweb.cern.ch/trac/cactus/ticket/6>
  - Review aims to provide an inclusive interface
    - Other protocols can be added
    - No reason for it to be exclusive to our IPbus implementation

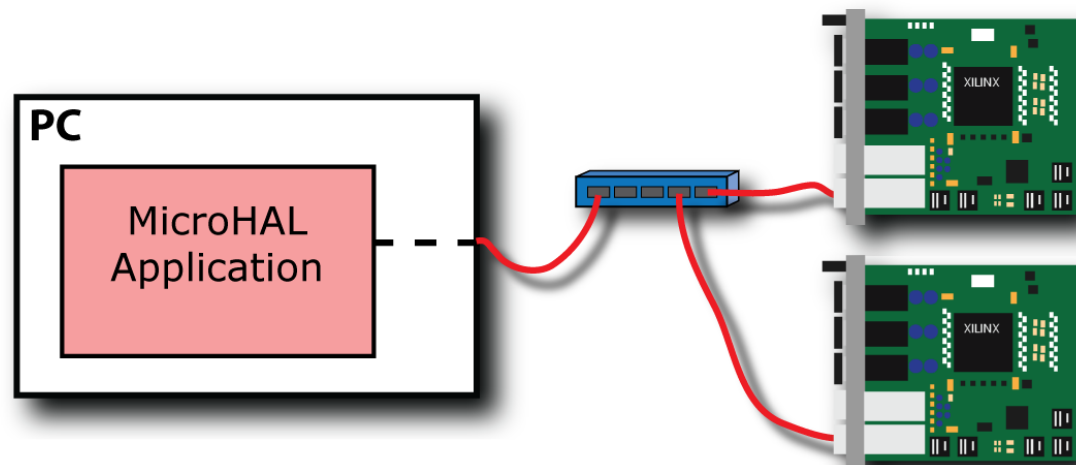


# Use-cases: bench-top 1

- With what I've outlined so far, we can certainly do this:



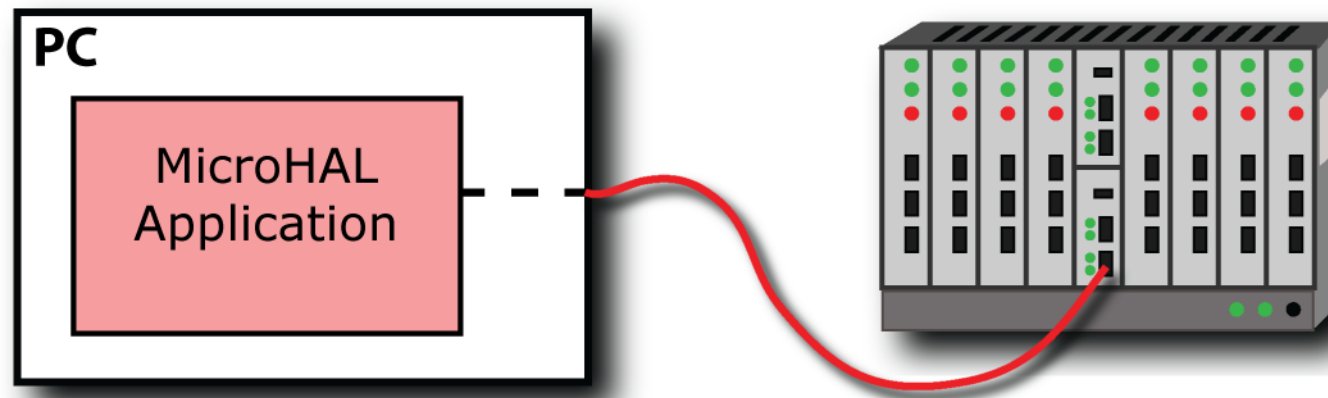
- And this:





# Use-cases: bench-top 2

- And this as well:



- **Bench-top use-case is then defined as:**
  - Single PC
  - A single, simple software process/thread controlling ~few boards
  - Maybe a crate MCH



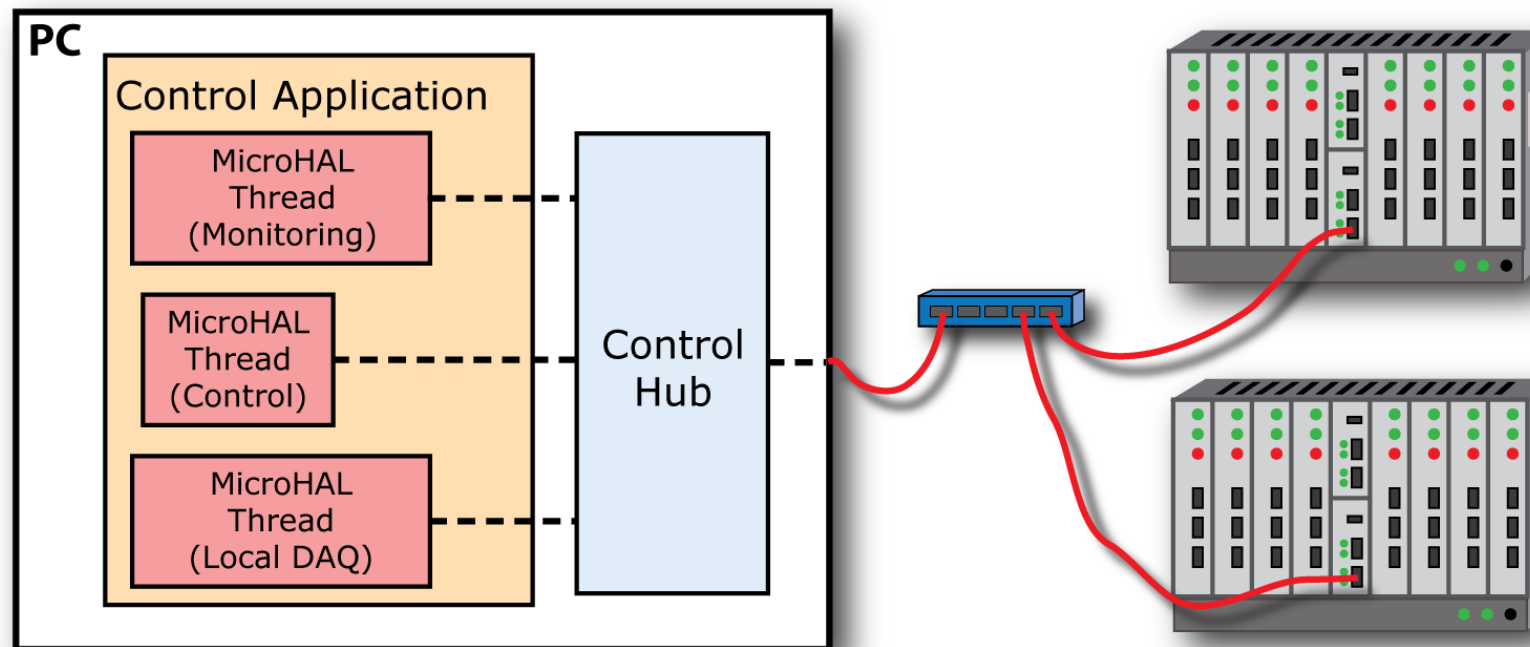
# Use-cases: test-beam 1

- **What about something more complicated for test beams?**
- **Multiple crates**
- **More complex software – multiple threads/processes**
  - Remember: with IPbus, there can only be a single client in communication with any single board at any one time
- **We don't want end-users to implement locks/mutexes...**
- **... but we might need:**
  - Configuration process
  - Monitoring process
  - Hardware spy process
  - Local DAQ process



# Use-cases: test-beam 2

- “Control Hub” acts as single point of contact with hardware



- Control app. can now safely instantiate independent client threads



# Use-cases: test-beam 3

- **Test-beam use-case is then defined as:**
  - More complicated control app (TS, XDAQ app, etc)
  - Tens of boards / few crates
  - Probably a single PC
  - Usage of crate MCHs
  - Probably a Control Hub

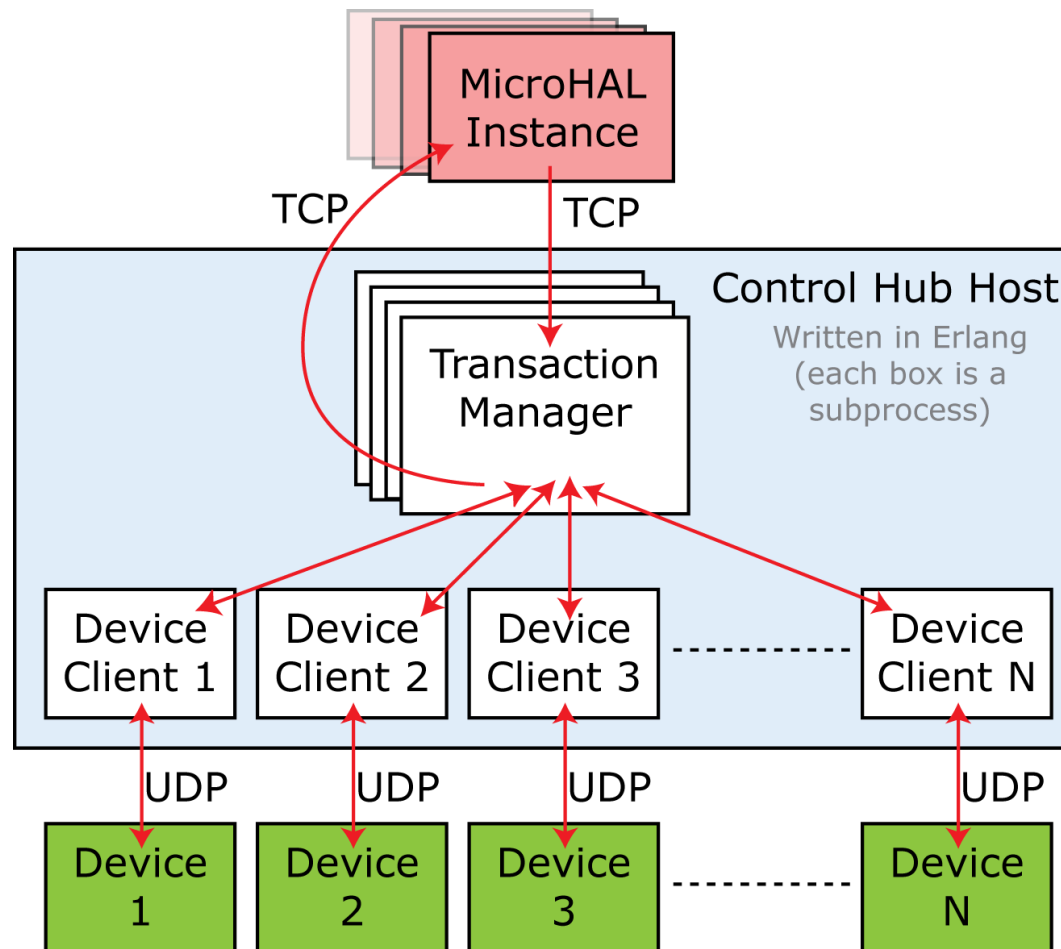


# Software: Control Hub 1

- **Analogous to a VME crate controller + driver software**
  - The Control Hub forms a single point of contact with the hardware
  - “Owns” the hardware network
  - Control Hub can also be thought of as an IPbus packet router
- **Current functionality**
  - Accepts up to 16 simultaneous microHAL clients
  - Makes best use of available bandwidth
    - Bandwidth to each board is multiplexed across gigabit connection
  - Highly scalable – can control multiple crates of boards
    - Pretty much only limited by number of Ethernet connections and CPU cores you can cram in a single rack PC.



# Software: Control Hub 2







# Software: Control Hub 3

- **Currently implemented in Erlang**
  - Concurrent programming language designed by telecoms industry
    - Transparent usage of multi-core CPUs
  - Erlang now in extensive use in wider industry
    - Database industry (CouchDB), Facebook Chat, etc, etc.
  - Yes... I appreciate Erlang is not a widely-known language in CMS
    - Consider the current implementation a prototype
    - Long-term maintainability – maybe better to re-implement in C++
- **Plan to add features such as:**
  - Auto IP address assignment
  - Board “heartbeat” monitoring

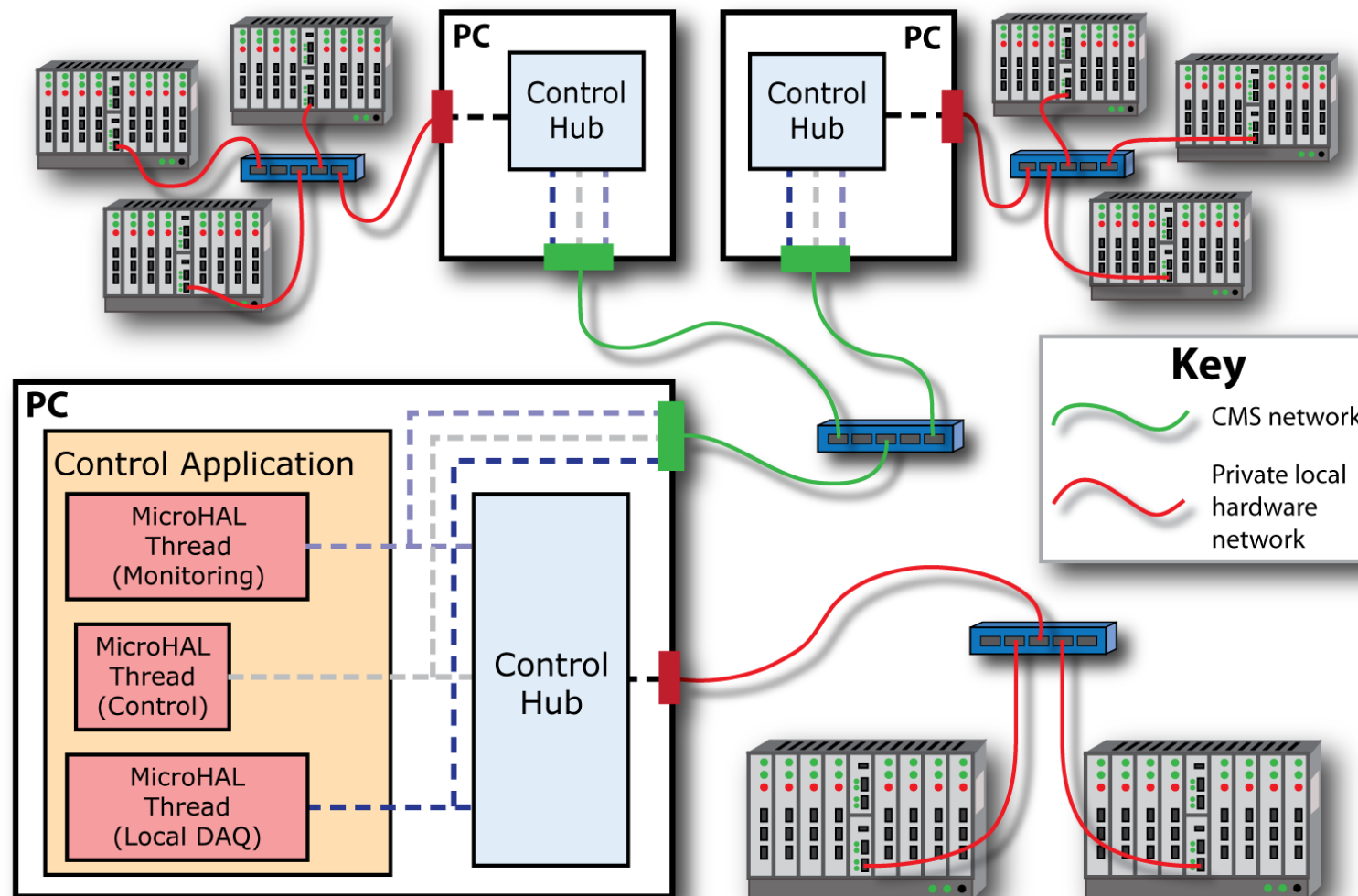


# Use-cases: full-scale 1

- **Full Point 5 use-case defined as:**
  - Multiple rack PCs
  - Full-scale control software
    - Multiple IPbus client threads/processes
  - Several Control Hubs
  - Many crates
  - DCS
- **There are various possible configurations...**
  - Software + PC + network topology largely up to end-user
  - Perhaps something like this...



# Use-cases: full-scale 2





# Testing IPbus 1

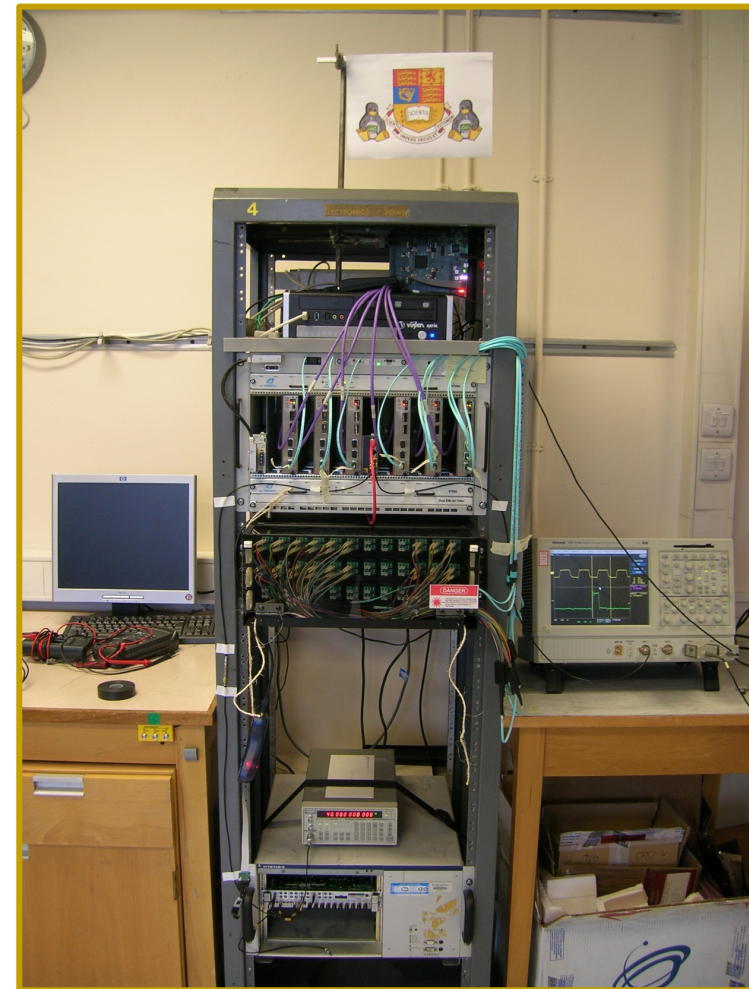
- Extensive test system in Bristol
- On three rack PCs, we run:
  - Multiple  $\mu$ HAL instances
  - Single Control Hub
- 20 (logical) IPbus hosts...
  - Running on six physical development boards
    - 3 x SP601
    - 2 x SP605
    - 1 x Avnet V5





# Testing IPbus 2

- **And at Imperial we have**
  - The TMT calorimeter-trigger demonstrator!
    - 6 x Mini-T V5 boards
- **IPbus over UDP has so far proved very solid**
  - Extensive soak-testing performed
  - Good understanding of packet lost rates (1 in ~200 million)





# Issues still to resolve 1

- **IP address assignment**
  - How to map MAC addresses to IP addresses?
  - Boards have IPs assigned from the Control Hub?
    - How to deal with hot-swap?
  - What about crate MCHs? (Two network interfaces?)
- **How should the software refer to boards?**
  - User software should not need to be changed across use-cases
  - *i.e.* no recompile when going from lab to P5





# Issues still to resolve 2

- **How should the software refer to boards? (cont.)**
  - Perhaps refer to boards within  $\mu$ HAL something like:
    - `protocol://subsystem.crateX.slotY`
    - `protocol://subsystem.crateX.boardY`
  - How do we resolve names to IP addresses or Control Hubs, etc?
    - Bench-top and test-beam, it can be done by config files
    - Point 5... some kind of name-server?
    - How do we support hot-swap, fail-over, etc?
- **DCS? (well outside my area of knowledge)**
  - Crates controlled via secondary MCH network interface?



# Conclusions

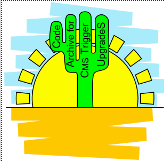
- **Stable release of IPbus firmware + software already available**
  - Tested at a wide variety of scales
- **C++ API undergoing review and ratification (see next talk)**
  - New release with approved API once review is complete
- **IPbus is under active development and is still evolving**
  - It will get faster – 155 Mbit/s per board and beyond
  - Retry mechanism for 100% reliability will be added
    - If 99.9999995% reliability not enough for you ;-)
  - More documentation, examples, *etc.*, once review is complete
- **Starting to think about wider system issues**
  - Much work still needed here; write CMS Internal Note?



# Want more info?

## - see the website





### CODE ARCHIVE FOR CERN TRIGGER UPGRADES

#### HOME


- Home
- Wiki
- Repository/Trac
- About
- Quality Control
- Contact

Installation instructions and documentation for *all* projects within CACTUS can be found on the **CACTUS Wiki**.

Direct links to all the IPbus-related projects are also included below:


#### IPbus Firmware

- Firmware implementation of the IPbus protocol.
- VHDL sourcecode and example designs.
- Compact — uses minimal FPGA resources.




#### microHAL

- C++ Hardware Access Library for IPbus.
- Fast and highly scalable.
- Mimics the recursive modularity of firmware.




#### Control Hub

- Reliable, scalable packet-handling for IPbus.
- Allows separation of the microHAL user network from the hardware network.
- The IPbus equivalent of a VME crate controller.



#### PyChips

- Python-based Hardware Access Library for IPbus.
- Simple and easy to use, but not designed with scalability in mind.
- Great for small, low-complexity projects.



This project is unrelated to the Cactus Code Numerical Relativity project which may be found [here](#).

- <https://projects.hepforge.org/cactus/index.php>