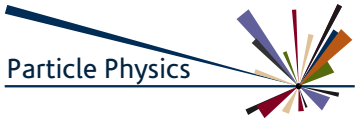
Science & Technology Facilities Council
Rutherford Appleton Laboratory



Update on IPbus v2

David Sankey

Tuesday, 9 April 2013

IPbus team:

Marc Magrans, Dave Newbold, Andy Rose, DS, Tom Williams

original incarnation from Jeremiah Mans, Erich Frahm and Eric Hazen

Much of this talk plundered from previous talks by the above and Rob Frazier

Overview of talk

Brief summary of IPbus

- Why IPbus
- Design Goals
- Why IP
- Why not TCP
- IPbus components

Current status

- Now implements reliable transport over UDP

Why IPbus

Need a mechanism to control xTCA hardware

Replacing the access over VME in historical systems

- and the need for VME SBC (ATLAS) and/or CAEN drivers/controllers (CMS)

General philosophy:

Based on an A32 D32 virtual bus

- Wishbone, that VME history…

No reliance on proprietary systems

- LHC experiments operate on much longer timescales than industry

Keep hardware and firmware as simple as possible

- push (any) complexity into software

Encapsulate complexity away from end-users

- simple API
- users should not be dealing with resource locks, threading *etc.*

Flexibility and scalability must be built in

- bench top to Point *n* deployment

Design Goals

Cover wide range of use cases

- single board through to 100s of xTCA modules

Good performance

- approach GB Ethernet wire speed?

Latency not an overarching priority

- but addressed to a degree by queuing and grouping requests in packets

Efficient FPGA resource usage

- fit into small FPGA or small % of large FPGA
- no requirement for CPU

Not dependent on xTCA features

- not tied to µTCA or ATCA
- can use IPMI for address assignment, RARP elsewhere

Work across many devices

- minimal vendor specific or family specific features
- Ethernet MAC and FIFO

Simplicity!

Why IP

Target application

Medium speed control traffic

- perhaps some local DAQ function

Pervasive technology

IP over Ethernet is everywhere (including xTCA Base Interface)

- physical layer components are cheap
- every computing device can use it trivially

key advantage: cost-effective scalability

- setup of a large (switched) distributed hardware control system is trivial
- setup of a many-to-many client/server system is trivial

key advantage: no drivers required!

Comparison with other 'control bus' technologies

not 'low performance' (*cf*. USB, SPI, VME)

not really 'high performance' (*cf*. PCIe, RapidIO)

- but 10 GB Ethernet could change that…

Why not TCP

Design choice is UDP/IP

essentially equivalent to 'raw IP'

- but UDP allows user space drivers (at the expense of some latency)

Pros:

- simplicity, efficiency, stateless

Cons:

- no guaranteed delivery

Why not TCP

in practise, modern switched Ethernet is almost 100% reliable

- in tests, dropping 1 packet in 200 million, in test beam, 1 in 30 million
- UDP has significantly lower latency for small packets

TCP effectively requires a CPU at the device end

- FPGA soft CPUs not good for performance or logic requirements
- FPGA hard CPUs require external RAM

UDP easy to implement in firmware

IPbus components

IPbus firmware

VHDL, using UDP as transport protocol

- ARP, ping

μHAL

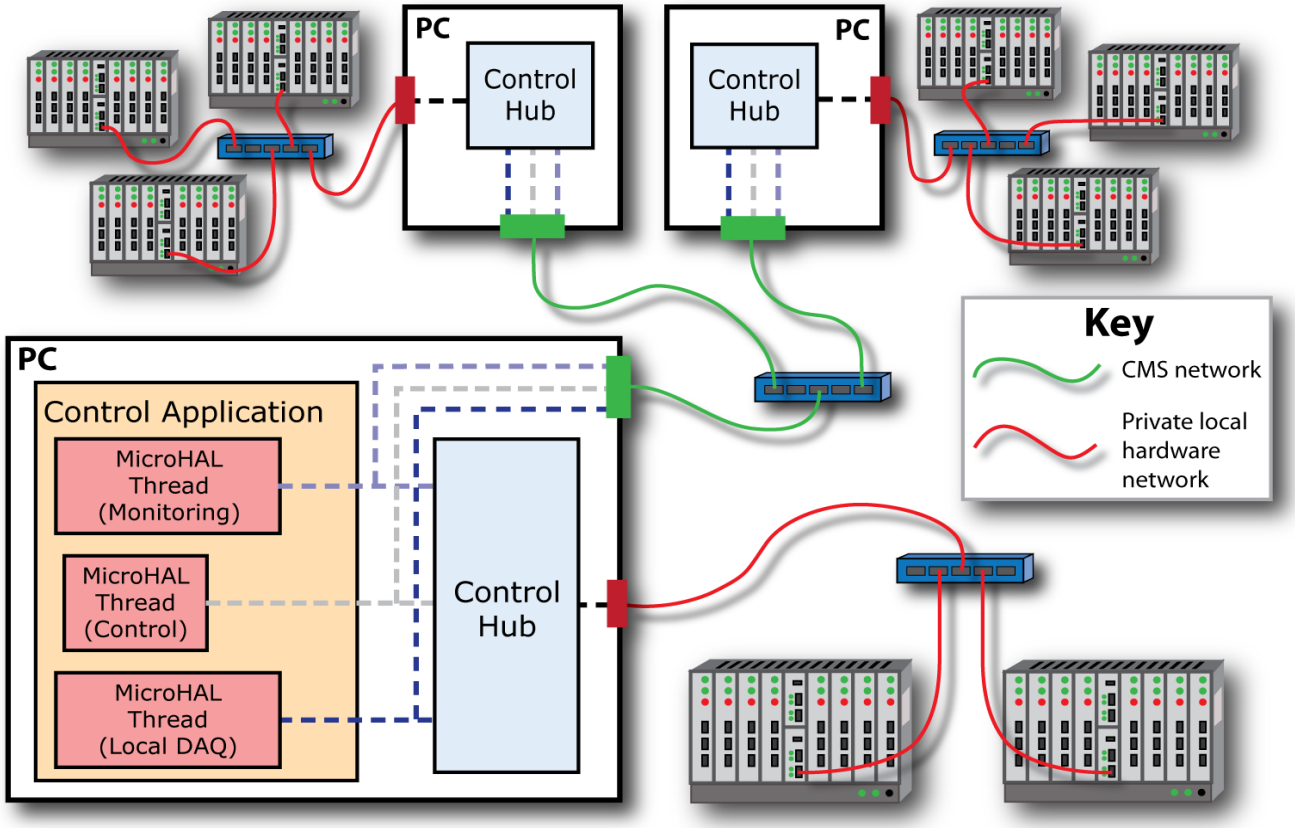
C++ Hardware Access Library

- Python bindings for easy scripting

ControlHub

single point of contact with private hardware network

- serialises/routes IPbus transactions
- implements reliability over UDP
- communication between μHAL and remote ControlHub over TCP



IPbus status

Version 1.3

Last version in 1.x series, in extensive use

- not protected against packet loss
- only single packet in flight

PyChips (standalone only) replaced by pycohal (compatible with ControlHub)

Version 2.0

Specification finalised

- https://svnweb.cern.ch/trac/cactus/browser/trunk/doc/ipbus_protocol_v2_0.pdf

First release on Friday

- https://svnweb.cern.ch/trac/cactus/blog/released_cactus_ipbus_2_0_0

Control behaviour unchanged from C++/Python API perspective

Consolidation in back-end code of all components

- transport layer in firmware rewritten from scratch

Transport between ControlHub and firmware now reliable

Multiple packets in flight next on agenda

IPbus 2.0 firmware

Full functionality for single packet in flight

Tested on SP605 at Bristol and RAL

- millions of pings
- 150000 control packets to and from board
  i.e. 500k reads/writes (random sequence)

All responses correct

Still to do (short term)

- test correct behaviour in all error cases (e.g. write to non-writable address)
- long term soak tests

Longer term

- implement multiple packets in flight
- cope with little-endian requests (the one instance where the fix is in the firmware, not the software!)
- implement RARP *(definitely not DHCP)*

IPbus 2.0 μHAL

Only one or two additions since last release

- new protocol version already in, has been passing unit tests for last month

Some changes to back end for consolidation/ ease of future maintenance

- again passing unit tests

Still to do (short term)

- add remaining IPbus 2.0 functionality to dummy hardware (software that replicates hardware IPbus interface)
- a few other smallish items

IPbus 2.0 ControlHub

Large overhaul of one of the modules to implement packet loss recovery and ease path to multiple packets in flight

Testing

- force 0.1% packet loss to/from hardware
- far higher loss than on any sensible network

Sent 5 million packets to board so far

- *i.e.* 5000 packets dropped

All successfully recovered

Still to do (short term)

- residual tests

Longer term

- multiple packets in flight
- a handful of other changes for P5-style deployment

Packet loss recovery and multiple packet implementation

New in version 2 protocol is an IPbus packet header

This contains an incrementing packet ID

- hardware will only accept next packet ID (or 0), dropping any other packet

For *n* packets in flight, ControlHub and hardware buffer all *n* outbound packets

Packet loss on way to hardware

Hardware drops subsequent packets

- ControlHub confirms next expected packet ID and resends from there

⇒ *communication resumed*

Packet loss on return from hardware

ControlHub spots missing response

- ControlHub confirms next expected packet ID and requests resend of missing packet

⇒ *communication resumed*

Conclusion

First version of IPbus 2.0 has just been released

Implements reliable transport over UDP between hardware and ControlHub

Specification at

- https://svnweb.cern.ch/trac/cactus/browser/trunk/doc/ipbus_protocol_v2_0.pdf

Release notes

- https://svnweb.cern.ch/trac/cactus/blog/released_cactus_ipbus_2_0_0

Will be used for integration testing at RAL and in 904

In next few months will add multiple packets in flight

- will improve block read/write bandwidth

Addendum: sharing the Ethernet MAC

The IPbus interface has the following AXI4 signals to and from the Ethernet MAC

On receive side:

```
mac_rx_data:   IN    std_logic_vector(7 DOWNTO 0);
mac_rx_error:  IN    std_logic;
mac_rx_last:   IN    std_logic;
mac_rx_valid:  IN    std_logic;
```

On transmit side:

```
mac_tx_data:   OUT   std_logic_vector(7 DOWNTO 0);
mac_tx_error:  OUT   std_logic;
mac_tx_last:   OUT   std_logic;
mac_tx_valid:  OUT   std_logic;
mac_tx_ready:  IN    std_logic;
```

On the receive side as there is no flow control, sharing the MAC is trivial

- any number of clients can listen, discarding the packets they don't want
- an upcoming change is a configuration switch to disable ARP and ping in an IPbus end point, so that multiple IPbus end points can share a MAC, each listening on a different UDP port

Sharing the Ethernet MAC (*cont.*)

On the transmit side sharing the MAC just requires a simple arbiter

- this picks which end point to route through to the real MAC
- all other end points see tx_ready low, so don't send

Chosen end point sees real tx_ready

- so starts to send when real MAC goes ready

Arbiter switches to 'next' end point at end of packet

'Other' end points can use the same arbiter

- if they have more intelligence then they could be the one responsible for ARP and ping

Caveat:

As described this isn't 'quite' the AXI4 flow control

- first two bytes should be accepted before MAC goes ready
- IPbus end point will cope with this (it's more like the old style flow control)

If an 'other' end point insists on true AXI4 flow control then it's just a bit more work in the arbiter to always buffer these two bytes