Available online at www.sciencedirect.com

SciVerse ScienceDirect

Physics Procedia 37 (2012) 1892 – 1899

Physics Procedia

TIPP 2011 - Technology and Instrumentation in Particle Physics 2011

Software and firmware for controlling CMS trigger and readout hardware via gigabit Ethernet

Robert Frazier[1 a], Greg Iles[b], Dave Newbold[a], Andrew Rose[b]

[a] *University of Bristol, H.H. Wills Physics Laboratory, Tyndall Avenue, Bristol, BS8 1TL, UK*
[b] *Imperial College, Blackett Laboratory, London, SW7 2BW, UK*

Abstract

Forthcoming hardware upgrades to the CMS experiment trigger and readout system are based upon the ATCA or μTCA bus standards, giving them the opportunity to be controlled via commodity gigabit Ethernet. These hardware upgrades supersede existing systems largely based upon the VME-bus standard, and thus a requirement has arisen to provide a new low-level control infrastructure for use by trigger and readout subsystem developers. This paper details the recent research and development into a tightly-integrated suite of software and firmware based upon the IPbus protocol that allows such Ethernet-attached hardware to be controlled in an efficient and highly-scalable manner.

© 2012 Published by Elsevier B.V. Selection and/or peer review under responsibility of the organizing committee for TIPP 11. Open access under CC BY-NC-ND license.

Keywords: CMS, trigger, control, ATCA, μTCA, IPbus

1. Introduction

The majority of forthcoming and future upgrades to CMS [1] trigger and readout hardware will be based upon the ATCA or μTCA bus standards (henceforth denoted as xTCA), replacing existing systems based upon antiquated standards such as VME-bus. Taking μTCA as an example, a typical μTCA crate allows slots for up to 12 custom hardware boards controlled via a carrier hub known as an MCH. Communication with the MCH can use a variety of different protocols, including gigabit Ethernet, which it then distributes to the boards via the crate backplane. With an appropriate protocol and associated firmware and software, it is then possible to communicate with and control any boards within the μTCA crate via this gigabit Ethernet connection.

This concept of controlling hardware via gigabit Ethernet forms the motivation behind all of the work described herein. As a solid foundation to this aim, J. Mans *et al.* should be credited with providing us with their preliminary work on an appropriate protocol, known as the IPbus protocol [2], and associated firmware in 2009. Although this firmware has since been radically redesigned and ported to a different language, the IPbus protocol itself remains a stable core component, and over the last two years a tightly-integrated suite of software implementing this protocol has been developed to allow end-users to control

[1] Corresponding author. Email: rob.frazier@bristol.ac.uk

1875-3892 © 2012 Published by Elsevier B.V. Selection and/or peer review under responsibility of the organizing committee for TIPP 11. Open access under CC BY-NC-ND license. doi:10.1016/j.phpro.2012.02.516

their hardware. Henceforth, this collective work is just referred to as IPbus, consisting of the following four discrete components that will be covered in greater detail in later sections:

- *IPbus protocol*: a custom protocol for controlling host hardware from a client computer.
- *IPbus firmware*: a firmware module to enable the IPbus protocol within end-user hardware.
- *MicroHAL*: software implementation of the IPbus protocol and end-user programming interface.
- *Control Hub*: packet-handling software to allow separation of hardware and control networks.

2. The IPbus concept

2.1. Choice of link layer

Gigabit Ethernet was chosen as the preferred form of link-layer communication with an xTCA crate over available alternatives such as PCIe or SATA/SAS, for a number of reasons. Gigabit Ethernet is inexpensive, very commonplace, convenient, and can provide a reasonably large amount of bandwidth per board (~80 Mbit/s) when averaged across a whole crate. If future bandwidth requirements are greater, 10-gigabit Ethernet can be used. The flexibility and scalability of Ethernet also meshes well with requirements of an upgraded CMS level-1 trigger and readout system, which is likely to consist of order ~hundreds of crates of electronics spread over multiple floors of an underground cavern, ultimately controlled from a control room on the surface roughly 100 meters above.

2.2. The IPbus protocol

Before giving an architectural overview of the IPbus concept, it is important to understand some basic features of the IPbus protocol itself. The protocol is transactional between client (computer) and host (hardware), such that each request has a specific response message to allow confirmation (or otherwise) of the request. Requests, and their respective responses, can be concatenated into a single packet to allow efficient transport over the network. The current implementation uses the User Datagram Protocol (UDP) as the transport protocol, and thus the IPbus requests are within the UDP packet payload, which itself is wrapped up within an Internet Protocol packet and then an Ethernet packet for transmission over the network.

UDP was selected as the transport protocol for two main reasons. Firstly, it is a very simple protocol, and thus relatively easy to implement in firmware, and secondly it is widely supported within the software world. Although UDP is classed as an “unreliable” protocol when compared to complex protocols such as TCP, the fact that each IPbus request has an explicit confirmation largely mitigates this issue, and packet loss on a dedicated network is in reality very low; more detail is given on these issues in section 6. Each IPbus host device has an IP address and port number on which it responds to IPbus client requests, and importantly, only a single request/response packet may be in flight between the client and any single host device at any one time. This greatly simplifies the design of the software and firmware as well as keeping the firmware input buffer sizes to reasonable levels, at the cost of slightly reducing the peak bandwidth to any one board. This is a largely inconsequential cost when considering the overall communication bandwidth to an entire crate of cards that share a single gigabit Ethernet link.

The protocol defines a number of different possible transactions, including:

- *Read*: a read of a user-definable depth, and a non-incrementing variety for reading FIFOs.
- *Write*: a write of a user-definable depth, and a non-incrementing variety for writing to FIFOs
- *Read-Modify-Write bits*: for efficiently altering a subset of bits at a single address location.
- *Read-Modify-Write sum*: for efficiently adding a given value to a single address location.

More information on the protocol can be found in the specification [2].

2.3. Architectural overview

The overall IPbus architecture is heavily motivated by the topology of the existing CMS level-1 trigger and readout system. Described here using gross generalities, this consists of many hundred crates of electronics, each of which is controlled via a rack PC that communicates with the VME (or alternative) crate controller. Another tier of systems above this is responsible for distributing via the network global commands such as “configure”, and collating monitoring information. Most importantly, many software processes or threads may be querying any one board or crate at any one time (for instance, monitoring will still occur whilst configuration is taking place), with the crate-controller driver software organising these concurrent requests into an orderly queue of reads and writes that get sent to the crate controller itself, and then onwards to the hardware.

The key points here are that there is a single point of contact with the hardware that isolates it from everything above, and that the hardware is attached via its own private connection. These features are particularly important when considering controlling hardware via a conventional networking technology such as Ethernet, as it is clearly undesirable to clutter up the connection to the hardware with other general-purpose network communication. Also, for reasons previously given, the firmware implementation of IPbus does not support concurrent requests to any single device. All of these factors motivated the need for what is now termed a Control Hub. A Control Hub forms a single point of contact with the hardware, receiving and queuing requests from whatever higher-level IPbus software processes may exist above it, and dispatching them to the hardware on its own private gigabit Ethernet connection. In many respects, an IPbus Control Hub is analogous to a VMEbus crate controller and its associated driver software.

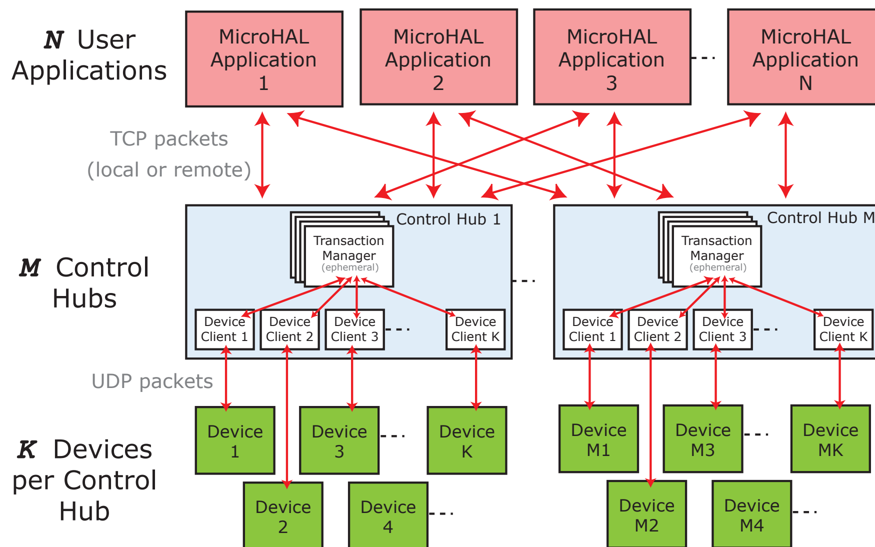

Fig. 1. Example of a large-scale IPbus system, with multiple MicroHAL end-users applications communicating with many crates of hardware via their respective Control Hubs.

For end-users to make use of the IPbus protocol, an application programming interface (API) known as MicroHAL has been created and substantially documented [3]. This software implementation of the protocol provides developers with a transparent interface to communicate with their hardware either in a local-client mode, which is suitable for bench-top testing and development, or in a remote-client mode for more complex setups where the IPbus transactions are routed via Control Hubs. When used in local-client mode, MicroHAL communicates directly with the hardware with UDP packets containing the IPbus

requests/responses. When the remote-client mode is used, MicroHAL communicates via TCP with the Control Hubs for safe transport of commands across what might be a general-purpose network.

Putting these components together, a large-scale system can be achieved with multiple user-level nodes running MicroHAL-based applications that communicate on the general-purpose network with several Control Hubs. Control Hub nodes each then control one or more xTCA crates, according to the power of the host machine and availability of gigabit Ethernet outputs. See figure 1 for a graphical representation of such a large-scale system. If preferable, it is also possible to have MicroHAL applications running on the same physical machine as a Control Hub, with the MicroHAL to Control Hub network traffic going through the localhost.

3. IPbus firmware

The IPbus firmware is implemented in VHDL and is simple to integrate into a wide variety of physical solutions. Real-world usage has already been achieved on a number of different platforms, ranging from sub $300 Xilinx Spartan 6 FPGA demonstrator boards, to custom boards using Virtex 5 or 6 FPGAs, and these example designs are provided to the end-user. Resource usage for the baseline IPbus firmware is reasonably low, with the major constraint being the availability of block RAMs needed for packet input/output buffers. Example resource usage figures for one of the lower-end Spartan 6 FPGAs can be seen in table 1.

Resource	Usage
Registers	7%
Lookup tables	18%
Block RAMS	25%

Table 1. Real-world resource usage of the IPbus firmware as seen in a Xilinx Spartan 6 (XC6LX16-CS324) FPGA

The firmware is well modularised, with a clean separation between the Ethernet interface, protocol decoders, and bus master. This allows any of these components to be easily swapped out if desired—for instance, if a different transport protocol other than UDP was preferred. Another key feature is a general-purpose interface to the bus master, allowing arbitrated control of the IPbus from a number of sources other than the usual Ethernet/UDP route. For instance, this allows the IPbus to be controlled by a microcontroller or external SPI/I2C interface, providing the convenience of being able to set the IP address of any individual IPbus host via IPMI.

4. Control Hub

A Control Hub is a software instance that forms a single point of contact with what may be one or more crates of IPbus-enabled hardware. It mediates transaction requests coming from several end-user software processes into orderly queues for dispatch to the individual hardware devices, and then passes the transaction responses back to the originator. Users software processes can be on the same physical host as the Control Hub, or on remote machines communicating across a user network, with the Control Hub acting as a middle-man between these processes and the hardware devices connected on a private hardware network. It is largely analogous to—but much more flexible than—a VME crate controller and its associated driver software.

A typical usage scenario might involve ~four end-user processes communicating via a Control Hub with a single crate of hardware (12 devices). Forming the boundary between two gigabit networks obviously results in a lot of data throughput at the Control Hub. Handling such a data-rate in software requires careful programming to minimise data copies, and even if this is achieved, still results in considerable processing load. Good performance of the Control Hub implementation is thus highly desirable, not only to offer flexibility in terms of how many cards or crates a Control Hub can support, but also looking to the future and the widespread emergence of 10-gigabit Ethernet. Given that present-day CPUs typically contain four

physical CPU cores and this will only increase with time, the performance of the Control Hub must not be artificially constrained by its implementation utilising only a single or small number of available processing cores. The ability for the Control Hub software to scale across an arbitrary number of CPU cores according to workload is very important, both today and in the future, particularly bearing in mind the long potential usage lifetimes of the software.

Reliability and transparency are also of utmost importance—a Control Hub instance should be at least as reliable and transparent to use as the VME crate controller it replaces. A failure or crash at a Control Hub can disrupt the communications of several upstream user processes, making the consequences significantly worse than a user-level software failure. Ideally, the user should need to know nothing about a Control Hub instance other than the IP address and port number it resides at.

Key requirements for the implementation of the Control Hub software are thus a high level of reliability, performance, scalability, and transparency, with its main role being the handling and routing of gigabit levels of IPbus data across a network boundary. Although it is undoubtedly possible—but rather difficult—to implement these requirements using conventional languages such as C++ or Java, it was decided that a better solution would be to use the somewhat unconventional language Erlang, which provides many of the desired features out of the box. Developed by the telecoms industry, Erlang is a programming language used to build massively scalable systems with requirements on high availability, with built-in support for concurrency and fault tolerance. Erlang is built upon the concept of very lightweight sub-processes with no shared state, and these sub-processes can be spawned and destroyed very rapidly. Sub-processes communicate with each other via message passing, but are otherwise completely independent entities, allowing them to take complete advantage of modern multi-core CPU architectures. It was considered the many benefits Erlang provides with respect to implementing the Control Hub far outweighed the risk of using a language not common within the field of high-energy physics.

A simplified overview of the Control Hub implementation can be seen within the blue/grey boxes of figure 1. An incoming request from a MicroHAL application results in an ephemeral Transaction Manager process being spawned that will exist only until a response has been returned to the requestor. The request contains a header so the Transaction Manager can determine which devices the IPbus transactions are headed for, allowing it to direct the transactions to the appropriate Device Client. The Device Clients deal with the request, and returns the result to the originating Transaction Manager, which then responds to the originating MicroHAL instance. There is a single Device Client process per physical device attached to the Control Hub host machine, and each client queues and handles the IPbus requests/responses completely independently from the other Device Clients. This makes very efficient use of the Ethernet connection to the hardware, as each Device Client can effectively overlap its maximum device bandwidth with the other clients that share the connection. Also, in the event of very heavy demand, there are at least as many Erlang sub-processes as there are devices attached, allowing the processing load to scale across many CPU cores if necessary. These features make the Control Hub software very flexible and future-proof, and depending on the power of the host machine, multiple crates can be serviced by a single host. To provide maximum Control Hub reliability, supervisor processes can restart any Device Client sub-process in the unlikely event of it failing, minimising device communication downtime to the sub-milli-second range.

5. MicroHAL

At the most basic level of description, MicroHAL is a C++ hardware access library (HAL) for IPbus, allowing end-users to write applications to control their IPbus-enabled hardware. However, MicroHAL also provides a great deal more than this baseline IPbus functionality, consisting of two main packages known as the IPbusClient and Redwood. The IPbusClient code encapsulates the implementation of the protocol, whilst Redwood provides higher-level functionality that enables the end-user to write software in a manner that logically and intuitively mirrors the structure of the firmware. Both of these packages are described further in the respective sub-sections below. For detail beyond that provided here, and for information on the other features and packages provided by MicroHAL, see the extensive MicroHAL user documentation [3].

5.1. IPbusClient

Typically, end-user hardware is first developed in the lab, and then put into a production environment. The IPbusClient package reflects this need by providing two corresponding modes of operation. For simple bench-top operation in the early R&D stage, a local client mode is provided that communicates directly with a device (i.e. no Control Hub required). For later on in the R&D phase, and for use in the final production environment when multiple devices or crates of devices must be controlled, a remote client is provided that communicates with the hardware exclusively via a Control Hub. The local and remote clients are implemented through inheritance of a common interface, making it trivial for the end-user to move between the two different modes when the time is appropriate.

In the case of the local client mode, communication with the hardware is done via UDP packets exactly as described in the protocol document [2]. When the remote client is used, communication between the user application and Control Hub may be traversing a general-purpose network, and thus TCP is used for the network transport. In both modes, local or remote, the IPbusClient utilises a delayed despatch methodology that concatenates user requests until an explicit despatch method is called. This dramatically improves network transport efficiency, particularly in situations such as the monitoring of many individual status registers on a single device. Remote client communication with a Control Hub instance does not strictly follow the IPbus protocol itself, for it has no need to; instead, it uses a variant of the protocol (described here [3]) that allows MicroHAL to intelligently package up transactions for the most efficient transport across the network. To continue the previously used example of monitoring many status registers, if these status registers were common to every device in a crate of hardware, MicroHAL would package up a single copy of the individual register reads together with the relevant device identifiers, leaving it to the Control Hub to replicate these requests to the individual devices over the private hardware network.

From an end-user perspective, once a choice of local or remote client mode is made, little direct interaction is needed with the IPbusClient package, as it is further encapsulated by the more intuitive user-interface provided by Redwood. The IPbusClient then operates behind the scenes, performing the required network transactions in the most efficient way possible.

5.2. Redwood

Redwood builds upon the IPbusClient package to provide the end-user with a hierarchical software framework that best represents the firmware/hardware being controlled, whilst strongly promoting modularity and code reuse. When writing firmware, whether in VHDL or Verilog, the programmer is naturally led toward a hierarchical structure, some levels of which often contain many repeated elements. Traditional hardware access libraries often provide the user with a flat representation of the device address space, which—although fine for small firmware designs—proves increasingly inadequate when trying to represent in software the sizeable and very hierarchical designs that can be contained in large modern FPGAs. Redwood overcomes this limitation by providing the user with a truly recursive software structure known as a Redwood *tree*. Redwood *leaves* make up the bus endpoints (i.e. registers, etc), and Redwood *branches* can contain other branches, or any number of leaves. Each leaf has access to the relevant IPbusClient object, and so operations performed on the leaf are transparently performed on the hardware itself.

The address space of the device characterised by a Redwood tree is described using XML, the obvious choice for representing the hierarchical and modular nature of the underlying firmware's address space. Each discrete firmware component (i.e. leaf) has its own XML description, detailing any registers, etc, and their bitmasks. A complete description of the bus is then made up by referencing these module descriptions along with an offset for the module. Thus all components—firmware, software and address table—reflect exactly the same structure. To take this methodology to its ultimate conclusion, a Redwood tree-builder is also included, making it possible for the end-user to construct a full object-oriented representation of their firmware, including the transparent communication layer, with only three lines of C++ and the associated XML description of the firmware.

6. Reliability and performance

Extensive reliability and performance tests have been run on a dedicated test setup consisting of 20 IPbus host devices, a single Control Hub host PC (also capable of running local MicroHAL application instances), and three dedicated MicroHAL application PCs. Communication between the MicroHAL machines and the Control Hub takes place on a separate network to the communication between the Control Hub and the hardware, as would be expected in a production environment. To measure the UDP packet loss on the dedicated hardware network, 250 million iterations of a 350-deep block read request were sent concurrently between the Control Hub and each of 20 IPbus host devices, which represents 10 billion UDP packets traversing the hardware network, and 7 terabytes of IPbus payload data received by the Control Hub. In total, only 53 UDP packets were lost, making the average packet loss 1 in ~189 million packets transmitted. Currently when a packet is lost, a timeout occurs in the software, an exception gets thrown to inform the user, and it is left to user to resolve the issue. Based on the packet loss rate and the developmental nature of IPbus at this time, this is not currently considered a particularly onerous task for the end-user. Nonetheless, this issue is being addressed in the next version of the protocol, which will incorporate automated retries in the event of packet loss.

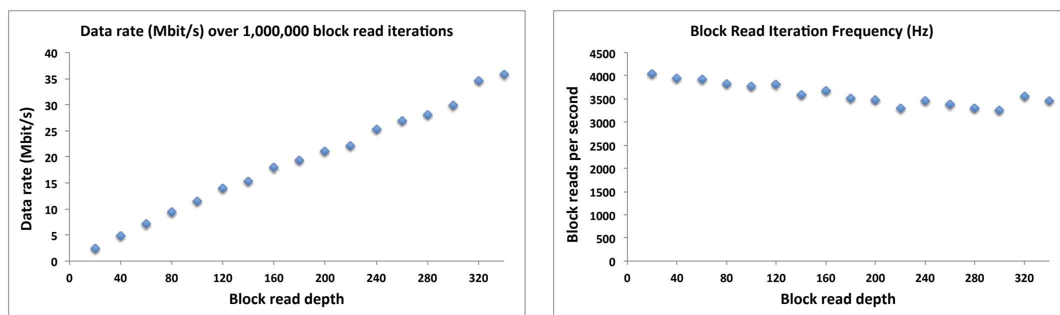

Fig. 2. Read-bandwidth performance for various block-read depths between a MicroHAL instance and a single IPbus host device, going via a Control Hub. Both the MicroHAL instance and the Control Hub instance were running on the same physical machine.

To give an idea of performance, figure 2 shows the read bandwidth and number of transactions per second for a variety of block read depths. The setup used consisted of a MicroHAL instance communicating with a single IPbus host device, with the requests going via a Control Hub running on the same physical host as the MicroHAL application. The block read depth is deliberately capped at a maximum of 340 deep as this is approaching the limit of the standard Ethernet frame payload size of 1500 bytes, and although block reads of any size can be requested, they must be suitably divided up once they exceed the limits of a single Ethernet frame. It can be seen that there is a linear relationship between the bandwidth and the block read size with a gradient of approximately 0.1, and there is only a weak relationship between the size/depth of a transaction and the number of transaction iterations that can be performed per second. On the test machine used here (2.4 GHz Intel Xeon E5620 CPU), the maximum read bandwidth for the software configuration specified is ~35Mbit/s. The same is true for the write bandwidth. Whilst this is more than sufficient for control applications, particularly when considering a single gigabit Ethernet connection is shared by one or more crates of boards, end-users desiring higher bandwidth can move to using jumbo Ethernet frames, thus increasing the maximum bandwidth 6-fold to in excess of 200 Mbit/s per board at the expense of increased block RAM usage within their FPGAs. Other optimisations to increase the bandwidth are yet to be fully explored at this stage in the development of IPbus, but include changes to the way the Control Hub dispatches large block read/write requests, and tuning the kernel networking parameters of the host PC for lower-latency network throughput.

7. Conclusions

A suite of firmware and software that implements the IPbus protocol has been developed to allow control of hardware via gigabit Ethernet. This suite has been designed with scalability in mind, and can operate in a number of configurations, from small systems through to very large ones. Extensive testing has confirmed the successful operation of IPbus on a system representative of a production-scale system for CMS. Performance tests show that approximately 4000 network dispatch iterations can be performed per second to a single board when using the full software chain, which given the ability to concatenate individual IPbus requests before dispatch, makes it possible to perform well in excess of 100,000 bus transactions per second. Whilst packet-loss does inevitably occur, loss rates are very low, and at this stage in the development of IPbus they cause almost no inconvenience to the end-user. As development continues, a retry mechanism will be introduced, along with other measures to increase the overall performance of IPbus. These envisaged reliability and performance improvements will be provided to the end-user in a largely seamless manner, as all components of IPbus are being co-developed and integrated by a single team.

References

[1] R. Adolphi, et al., The cms experiment at the cern lhc, JINST 3 (S08004).
[2] J. Mans, et al., IPBus v1.3: Simple IP-based μTCA Control System, `https://projects.hepforge.org/cactus/trac/wiki/IPbusIntro` (July 2011).
[3] A. Rose, R. Frazier, Redwood and co. User Guide, `https://projects.hepforge.org/cactus/trac/wiki/microHAL` (April 2011).
[4] IPbus project website: `https://projects.hepforge.org/cactus`.