IPbus: a flexible Ethernet-based control system for xTCA hardware

C. Ghabrous Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea and T. Williams

Dept. of Physics, University of Wisconsin-Madison, 1150 University Ave., Madison, WI 53706, U.S.A.
STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot, OX11 0QX, U.K.
H.H. Wills Physics Laboratory, University of Bristol, Tyndall Avenue, Bristol, BS8 1TL, U.K.
Blackett Laboratory, Imperial College, Prince Consort Road, London, SW7 2BW, U.K.

E-mail: t.williams@cern.ch

ABSTRACT: The ATCA and 𝜇TCA standards include industry-standard data pathway technologies such as Gigabit Ethernet which can be used for control communication, but no specific hardware control protocol is defined. The IPbus suite of software and firmware implements a reliable high-performance control link for particle physics electronics, and has successfully replaced VME control in several large projects. In this paper, we outline the IPbus control system architecture, and describe recent developments in the reliability, scalability and performance of IPbus systems, carried out in preparation for deployment of 𝜇TCA-based CMS upgrades before the LHC 2015 run. We also discuss plans for future development of the IPbus suite.

KEYWORDS: Control and monitor systems online; Modular electronics
1 Introduction

The electronics systems of particle physics experiments constructed during the last few decades have typically been based on the VMEbus standard. However, new electronics systems within many particle physics experiments are based on the newer ATCA and µTCA standards (henceforth collectively referred to as xTCA). The xTCA specifications incorporate industry-standard serial communication technologies such as Gigabit Ethernet; however, unlike the VMEbus standard, they do not specify a hardware access protocol for reading and modifying the memory spaces of xTCA boards from external software applications.

Several important requirements must be considered when designing the architecture and implementation of a hardware control system. Control systems must have reliable and predictable behaviour under all conditions, since they form the main link by which hardware is configured, monitored, and debugged in case of problems. The control system architecture for large experiments should be highly scalable, ideally with the same ease of setup and use from the simple ‘board on benchtop’ scenario to the final system with hundreds of boards. In modern experiments, the same electronics setup is often used for decades before being replaced, and the associated control infrastructure must have the same maintainable lifetime. Hence, it is beneficial to use pervasive industry-standard technologies, in order to avoid the risk of reliance on a single vendor. Experience from the CMS experiment’s online systems in LHC Run 1 also shows that for monitoring and debugging issues in complex scenarios, in general it is helpful to move complexity away from hardware/firmware into software running on commercial PC hardware.
The IPbus protocol — first developed by J. Mans et al. in 2009 — is a simple packet-based control protocol for reading and modifying memory-mapped resources within FPGA-based IP-aware hardware. A tightly-integrated suite of IPbus software and firmware components which can be used to construct reliable, scalable, high-performance control systems has previously been presented in ref. [1]. This IPbus suite will be used to control the xTCA off-detector electronics in the upgrades for Run 2 of the CMS experiment [2], as well as the ATLAS experiment’s upgrades for Runs 2 and 3 [3]. In this paper, we present recent improvements in the reliability, scalability and performance of the IPbus suite, based on a new version of the protocol.

2 IPbus protocol

The IPbus protocol is a simple protocol for controlling IP-aware hardware devices which have an A32/D32 bus. It defines the following operations:

**Read** A read of user-definable depth. Two types are defined: address-incrementing (for multiple continuous registers in the address space) and non-address-incrementing (for a port or FIFO).

**Write** A write of user-definable depth. As with reads, two types of write are defined: incrementing and non-incrementing.

**Read-Modify-Write bits (RMWbits)** An atomic bit-masked write, defined as \( X := (X \& A) \mid B \). This allows one to efficiently set/clear a subset of bits within a 32-bit register.

**Read-Modify-Write sum (RMWsum)** An atomic increment operation, defined as \( X := X + A \), which is useful for adding values to a register (or subtracting, using two’s complement).

The protocol is transactional — for each read, write or RMW operation, the IPbus client (typically software) sends a request to the IPbus device; the device then sends back a response message containing an error code (equal to 0 for a successful transaction), followed by return data in case of reads. In order to minimise latency, multiple transactions can be concatenated into a single IPbus packet.

The protocol lies in the application layer of the networking model and is network protocol agnostic. TCP exhibits various highly-desirable features of a transport protocol, such as reliable, ordered data transmission and congestion avoidance; however, the underlying algorithm is significantly more complex than for the other ubiquitous transport protocol, UDP. Since the IPbus device implementation must have a low FPGA resource usage, UDP has been chosen as the transport protocol. Version 2.0 of the IPbus protocol [4] (finalised in early 2013) includes a reliability mechanism over UDP, through which the client can correct for any packet loss, duplication or re-ordering. This mechanism is credit-based with a fixed number of packets in flight, giving implicit traffic shaping which can avoid congestion-based performance degradation, such as TCP Incast.

3 Firmware and software suite

The IPbus software and firmware suite consists of the following components:

**IPbus firmware** A module that implements the IPbus protocol within end-user hardware
Table 1. Resource usage of IPbus firmware core in a Xilinx Virtex-7 (XC7VX690T) FPGA.

| Element type    | Minimal configuration | Fully-featured |
|-----------------|-----------------------|----------------|
| Flip flops      | 2000                  | 3500 (0.4 %)   |
| Slices          | 1000                  | 1078 (1.0 %)   |
| Block RAMs      | 5                     | 17 (0.6 %)     |

**ControlHub** Software application that mediates simultaneous hardware access from multiple µHAL clients, and implements the IPbus reliability mechanism over UDP

**µHAL** C++ and Python end-user programming interface for writes, reads and RMW operations

End-user instructions and source code for these components are available through the CERN CAC-TUS (Code Archive for CMS Trigger UpgradeS) website and SVN repository [5]. The software is packaged as RPMs for Scientific Linux versions 5 and 6, and available through a YUM repository.

### 3.1 IPbus firmware

The IPbus 2.0 firmware module is a reference system-on-chip implementation of an IPbus 2.0 UDP server in VHDL; it interprets IPbus transactions on an FPGA. It has been designed as a common module to run alongside a device’s main processing logic (e.g. trigger algorithms) on the same FPGA, only using resources from within the FPGA. Any loss, re-ordering or duplication of the IPbus UDP packets is automatically corrected by the ControlHub using the IPbus reliability mechanism.

The IPbus firmware module has been designed to be simple to integrate into variety of platforms, and there are example designs for several development boards and standard platforms. The source code is currently Xilinx-specific, but has been successfully adapted for Altera devices. The firmware is modular, with a core protocol decoder and bus master controlling the interface to the IPbus slaves, and a number of interfaces into the decoder with simple arbitration between them. As well as the UDP interface, there are SPI/I2C interfaces and chip-to-chip bridges allowing control from microcontrollers and between FPGAs. The UDP interface is monolithic, operating at the network layer in order to eliminate unnecessary internal buffering. It also implements: the echo request/reply semantics from ICMP (RFC 792, used in the Unix `ping` command); ARP (RFC 826, used for resolving IP addresses into MAC addresses); and RARP (RFC 903, used for requesting an IP address on startup). Several parameters are configurable at build time, including: the Ethernet frame MTU; the number of buffers for incoming/outgoing IPbus packets which determines the maximum possible control throughput; and the method used for IP address assignment — fixed IP address, RARP, or a secondary out-of-band IPbus controller (for instance an onboard microcontroller). The resource usage of the IPbus firmware core under ‘minimal’ and ‘fully-featured’ configurations is shown in table 1.

### 3.2 ControlHub

The ControlHub is a software application that forms a single point of access for IPbus control of each device; specifically, it arbitrates simultaneous access from multiple control applications to one
or more devices, and it implements the IPbus reliability mechanism for the ControlHub-device UDP packets. Since the ControlHub is a software application, the µHAL-ControlHub communication uses TCP, which has sophisticated congestion mitigation and flow-control algorithms.

The ControlHub must be at least as reliable and transparent as a VME crate controller, since failure or crash within the ControlHub could disrupt the communications of several upstream control or monitoring applications. Additionally its design must allow multiple clients to communicate with multiple targets reliably, efficiently and independently.

The ControlHub is implemented in Erlang [6], a general-purpose, concurrent programming language, designed by Ericsson to build high-availability, fault-tolerant applications. The main structural unit in Erlang is the process: Erlang processes are lightweight compared to operating system processes; they share no state, instead communicating by message passing. These features are well-suited to the ControlHub’s requirements for high reliability, performance, and scalability in routing IPbus transactions. The ControlHub uses a separate Erlang process for each connected µHAL client and each IPbus device, ensuring workload can be spread across multiple CPU cores; its internal structure is described in more detail in ref. [1].

3.3 µHAL library

µHAL is the Hardware Access Library (HAL) providing an end-user C++/Python API for IPbus reads, writes and RMW transactions. It is based on a delayed dispatch model, in which multiple transactions are queued and concatenated within the transport layer payload buffers until the dispatch method is called, or the command queue exceeds the maximum packet size.

In µHAL each device’s register layout is specified by XML files. Each node of the XML tree represents either a single register, block RAM, FIFO, or a collection of these; the nodes in one file can reference other address files, such that the interfaces to repeated instances of a firmware module can be generated with minimal copy-paste of address file contents. This enables the user to write control software in a manner that intuitively mirrors the modular, hierarchical structure of large firmware designs.

The µHAL interface to each device (based on the methods of the HwInterface and Node classes) can run in one of two modes of operation. In the local-client mode, the µHAL library communicates directly with device over UDP. In the remote-client mode, the µHAL library communicates with hardware exclusively via a ControlHub. These differing modes of operation are implemented through the inheritance of a common interface, such that users can switch between the modes of operation by simply changing the prefix of a single string when creating a HwInterface instance.

µHAL is also packaged with an example GUI that is useful for monitoring the values of a subset of registers on a device during hardware development.

4 Control system topology

The topologies of an IPbus control system in some common scenarios are shown in figure 1. The simplest system (upper left) is a single target running the IPbus firmware, directly connected by a single Ethernet cable to a computer running a C++/Python control application based on the µHAL library. This is the typical layout during early hardware development.
In a more complex scenario such as a beam test or integration tests, there will typically be several devices, with multiple control, monitoring and DAQ applications, as shown in figure 1 (upper right). Due to multiple applications simultaneously communicating with the devices, the IPbus traffic would be routed via a ControlHub, which would also recover any lost packets making the IPbus communication 100% reliable.

For a full-scale IPbus system at a large experiment (such as ATLAS or CMS) there would be hundreds of IPbus devices spread across many crates, and the control/monitoring applications would be spread across many computers, as shown in figure 1 (lower). In this case the use of an Ethernet network naturally allows scalability with the ease of extending the network using multiple switches and routers. Additionally the recovery from computer failure is simplified with the possibility of having redundant computers already connected to the network. Notably, the network will typically be divided into a separate subnet for each subdetector so that the network’s logical segmentation matches the typical IPbus dataflow. The exact number of devices per ControlHub will typically be adapted based on performance requirements.

**IPbus test system.** A test system was set up in the CMS electronics integration centre at CERN, in order to investigate the reliability and performance of the IPbus suite using very similar network layout and hardware to that planned for final deployment in the CMS experiment. The test system consisted of network infrastructure, two computers, and one μTCA crate containing 12 μTCA boards (AMCs), each running the IPbus 2.0 firmware core. The computers were Dell PowerEdge R300 rack PCs; three of the AMCs were GLIBs [7] and the other nine were Mini-T5s [8].
5 System reliability

The reliability and robustness of the IPbus suite has been ensured by extensive testing of both the software and firmware in a range of scenarios.

The software is tested by itself (independent of the hardware) during development using a dummy hardware executable which emulates the response of an IPbus device. A suite of unit test executables are run in order to test \( \mu \)HAL and the ControlHub with basic read/write/RMW operations to the dummy hardware running on the same machine. By configuring the operating system to randomly drop IP packets, the ControlHub’s reliability mechanism is also tested.

The full IPbus control link (\( \mu \)HAL-ControlHub-firmware) has been tested with a variety of \( \mu \)TCA boards, using a \( \mu \)HAL-based C++ executable. This executable issues random sequences of reads, writes and RMW transactions to a device using random addresses, random depths for the reads and writes, and random values for the data written and the RMW parameters. The executable checks that all of the returned error codes indicate success, and checks that the values returned by the reads and RMW transactions are always correct. The released version of the firmware core was validated by running the executable for over 20 hours (corresponding to over 10 billion transactions) against the IPbus firmware core loaded on each of the Mini-T5, GLIB and MP7 [9] boards. No errors were observed during this final testing.

6 Performance

The latency and block transfer throughput are two important parameters of a control system: latency is defined as the total round-trip time taken to perform an IPbus transaction, as measured in the \( \mu \)HAL client application; and throughput is defined as the amount of user data transferred or received per unit of time.

In order to predict the performance of the future CMS IPbus control system, and verify the design of the IPbus components and their planned layout, the system performance was measured in several benchmark scenarios. These measurements were carried out in the IPbus test system, with the \( \mu \)HAL clients running on one computer and the ControlHub on the other computer.

1-to-1 block transfers. The block read/write latency and throughput for one \( \mu \)HAL client controlling one device via the ControlHub is shown in figure 2. The median single-word write/read latency is approximately 250 \( \mu \)s. Although this single-word latency is significantly larger than with VME/PCIe-based control, for multiple transactions or large block transfers this is compensated by concatenating multiple transactions into each packet, and by having multiple packets in flight around the system at any given time. Together, concatenation and multiple packets in flight increase throughput by a factor of approximately 20 to 2000, depending on transaction type. Hence, the 1-to-1 block read/write throughput for payloads larger than 1 MByte is above 0.5 Gbit/s.

n-to-m polling. The system performance for multiple \( \mu \)HAL clients polling a single-word register in multiple devices via one ControlHub was also measured. The mean polling latency, and total system polling frequency, for 1, 2 or 4 clients per device are shown in figure 3 as a function of the number of devices. The latency experienced by each client gradually increases with the number of clients or devices, due to the the increasing load of network interrupts on the computers. However,
Figure 2. The median write/read latency and throughput as a function of depth, for one software client controlling one IPbus device, via the ControlHub.

Figure 3. The latency and total system polling frequency for \( n \) clients each simultaneously polling a register in one of the \( m \) targets.

The total polling frequency increases with the number of clients or devices in the system, as the ControlHub spreads its increasing workload over the four CPU cores.

**n-to-n block transfers.** The performance for continuous block reads and writes of all 12 boards in the \( \mu \)TCA crate was also measured. The Ethernet connection to a \( \mu \)TCA crate is via a Gigabit Ethernet socket on the front panel of the crate management module, the MCH (MicroTCA Carrier Hub). Each AMC in a \( \mu \)TCA crate is connected to the MCH’s Ethernet switch by a separate bidirectional 1 Gbit/s link. In theory, this network topology can lead to congestion in the MCH switch during simultaneous block reads from multiple AMCs. For block reads, the reply packets are significantly larger than the request packets, and so the total instantaneous return bandwidth from the 12 AMCs into the MCH can exceed the 1 Gbit/s capacity of the link from the MCH to the local network. However, within the IPbus protocol only a limited number of requests are in flight at any given time, which imposes an upper limit on the total size of packets that have to be buffered within the MCH switch. Hence, the IPbus protocol implicitly contains a simple congestion avoidance mechanism, allowing graceful performance degradation in congested scenarios.
Figure 4. The total system throughput for \( n \) IPbus clients each simultaneously writing to / reading from one of \( n \) devices, via one ControlHub, using a NAT MCH (left) or a Vadatech MCH (right).

Within the CMS collaboration, MCH modules are currently being purchased from two vendors: NAT and Vadatech. The IPbus system throughput for multi-client block reads and writes with multiple targets are shown in figure 4 for both the NAT and Vadatech MCHs. For the NAT MCH (V3.4), the read and write throughputs are similar; over 75\% of the Gigabit Ethernet bandwidth is utilised with three or more devices. However, using the Vadatech MCH (model UTC002-210-440-010), the system throughput degrades for simultaneous block reads from four or more devices due to congestion in the MCH switch, with read throughput approximately 20\% lower than write throughput for 8 or more targets. In order to reduce congestion, the throughput was re-measured with fewer packets in flight, achieved by editing one line in the ControlHub configuration file. With 11 packets in flight to each device (default value is 16), there is less congestion-induced packet loss, and so the simultaneous read throughput is above 0.75 Gbit/s for three or more devices; however, the maximum 1-client-to-1-target throughput decreases by approximately 12\%.

7 Conclusions

A new reliable, high-performance version of the IPbus protocol has been developed along with the associated suite of software and firmware, in order to control xTCA hardware via Gigabit Ethernet. An IPbus test system with realistic network topology was set up in the CMS electronics integration centre in order to verify the control system’s reliability, and investigate its performance. For one software client controlling one device, the single-word read/write latency is approximately 250 \( \mu s \) and the block read/write throughput is above 0.5 Gbit/s for payloads larger than 1 MByte; the total block read/write throughput is above 0.75 Gbit/s for three or more boards in a single \( \mu \)TCA crate.

The first large-scale IPbus system in the CMS experiment was deployed in August 2014, in preparation for the start of LHC Run 2 in 2015. Development is now focused on simplifying the monitoring of IPbus dataflows in large systems of hundreds of devices. The IPbus software and firmware suite will be optimised in order to improve performance with 10 Gigabit Ethernet. Additionally, an IPbus locking mechanism is being considered in order to provide exclusive access to IPbus devices from a single client for extended configuration sequences.
Acknowledgments

We would like to thank the Science and Technology Facilities Council (STFC) for funding the U.K. contribution to this research.

References

[1] R. Frazier, G. Iles, D. Newbold and A. Rose, Software and firmware for controlling CMS trigger and readout hardware via gigabit Ethernet, Phys. Proc. 37 (2012) 1892.

[2] CMS Collaboration, Technical proposal for the upgrade of the CMS detector through 2020, CERN-LHCC-2011-006 (2011).

[3] ATLAS collaboration, Letter of Intent for the Phase-I Upgrade of the ATLAS Experiment, CERN-LHCC-2011-012 (2011).

[4] R. Frazier et al., The IPbus protocol: version 2.0, https://svnweb.cern.ch/trac/cactus/browser/trunk/doc/ipbus_protocol_v2_0.pdf.

[5] The CMS Level-1 trigger project, The CACTUS (Code Archive for CMS Trigger UpgradeS) SVN repository, http://cactus.web.cern.ch/.

[6] J. Armstrong, Making reliable distributed systems in the presence of errors, Ph.D. Dissertation, The Royal Institute of Technology, Stockholm (2003).

[7] P. Vichoudis et al., The Gigabit Link Interface Board (GLIB) ecosystem, 2013 JINST 8 C03012.

[8] C. Foudas et al., A demonstrator for a level-1 trigger system based on MicroTCA technology and 5Gb/s optical links, 2010 JINST 5 C11015.

[9] A. Rose et al., Development and testing of an upgrade to the CMS level-1 calorimeter trigger, 2014 JINST 9 C01006.