NEW CONTROLS PLATFORM FOR SLAC HIGH-PERFORMANCE SYSTEMS*

Till Straumann, R. Claus, J. M. D'Ewart, J.C. Frisch, G. Haller, R.T. Herbst, B. Hong, U. Legat, L. Ma, J.J. Olsen, B.A. Reese, L. Ruckman, L. Sapozhnikov, S.R. Smith, J. Vasquez, M. Weaver, E. Williams, C. Xu, A. Young, SLAC, Menlo Park, California, USA

Abstract

The 1 MHz beam rate of LCLS-2 precludes the use of a traditional software solution for controls of "high-performance systems" which operate at this rate, such as BPMs, LLRF or MPS. Critical algorithms are ported into FPGA logic and administered by ordinary PCs via commodity Ethernet. SLAC has developed a controls architecture which is based on FPGA technology interconnected by 10G Ethernet and commercially available ATCA shelves. A custom ATCA carrier board hosting an FPGA, memory and other resources provides a "common platform" for many applications which can be implemented on AMC cards which are plugged into the carrier. A library of firmware modules including e.g., timing, history buffers and reliable network communication together with corresponding software packages complement the common platform hardware and provide a standardized environment which can be employed for a variety of high-performance applications across the laboratory.

INTRODUCTION

The LCLS-2 XFEL is currently under construction at SLAC. A superconducting linac supplies electron bunches at a rate of up to 1 MHz. Several diagnostics, controls and protection systems must be able to resolve individual bunches and process data in real-time. A conventional control system based on computers with peripheral devices and software based algorithms is – at the current state of the art – not capable of meeting the necessary throughput and latency requirements (considering that the actual data rate some subsystems have to handle is much higher than the beam rate of 1 MHz).

Therefore, the bulk of processing must be implemented in programmable logic, leveraging a high level of parallelism in hardware. SLAC has developed a new platform which is based on the following technologies:

- ATCA [1] form-factor and commercial shelves.
- IPMI management [1].
- 10 G Ethernet communication.
- FPGA on generic ATCA carrier.
- Application-specific AMC cards.

An library of firmware components and a "common-platform" software framework complement this versatile and powerful instrumentation and controls platform.

* Work supported by the US Department of Energy, Office of Science under contract DE-AC02-76SF00515

Since the hardware design has already been presented elsewhere [2, 3] this paper shall focus on the communication protocols and software framework.

HARDWARE OVERVIEW

The platform employs a COTS ATCA shelf that provides standardized mechanical support, power, cooling, management and interconnect. The backplane's fabric interface is configured in a dual-star topology supporting up to 10 Gbps per channel (with a 40 Gbps upgrade path). A block diagram of the system is depicted in Fig. 1.

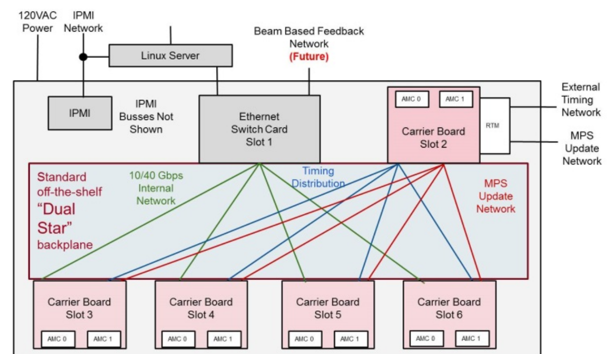

Figure 1: Common Platform System Diagram.

One star is concentrated at a 10 Gb Ethernet switch that provides the main connectivity between ATCA boards as well as an external Linux server computer (Fig. 1) which is responsible for interacting with the firmware.

The second star is used to broadcast timing data and to gather MPS (machine-protection) information that is further aggregated by the custom general-purpose carrier board (as described in the next paragraph) in slot 2'. Timing and MPS connectivity to the outside is provided by a custom RTM.

One core element is the custom carrier board that hosts a Xilinx Kintex Ultrascale XCKU040 or -060 FPGA and 8 GB of DDR3 memory. The FPGA's SerDes interfaces are connected to the Fabric interface as well as four AMC bays. These bays can receive single- or dual-wide, full-height AMCs. Additional LVDS and high-speed I/O is available via the zone-3 connector to RTMs.

A variety of AMC cards (commercial and custom) can be plugged into the carrier. A common use case are high-speed ADCs for applications like BPMs or bunch-length monitors. Placing analog components on well-shielded AMC cards ensures that EMI effects can be minimized.

Copyright © 2017 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-189-2

Firmware Library

Many applications require a large set of complex, common functionality such as DDR-memory, JESD, Ethernet communication, timing-system support, MPS uplink, diagnostics, etc. These components are supported by an extensive library of firmware modules (Fig. 2), thus greatly simplifying the task of an application developer.

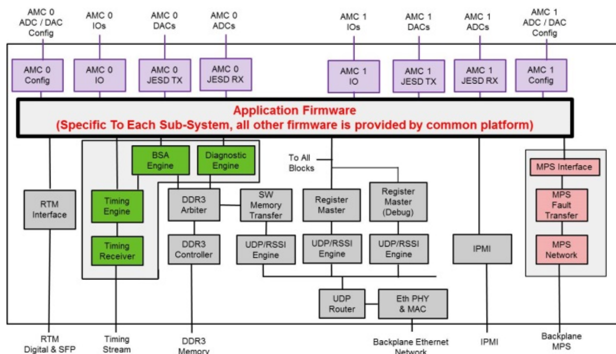

Figure 2: Common Platform Firmware.

COMMUNICATION PROTOCOLS

Communication between firm- and software is achieved by means of a stack of protocols as shown in Fig. 3. On top of Ethernet there is a layer of pseudo-IP/UDP. This is (in firmware) a non-standard version of the widely-known IP protocol that supports just enough functionality (e.g., no fragmentation or IP options, no ICMP, etc.) to interoperate with a standard network stack.

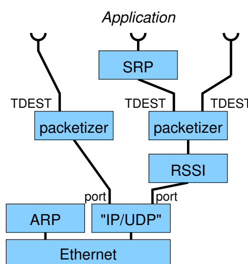

Figure 3: Protocol Stack; some layers are optional (e.g., no RSSI at LHS UDP port).

The *RSSI* layer implements a reliable transport. It is based on [4–6] with minor deviations, one of which adds a flow-control feature. RSSI provides functionality similar to TCP but is much simpler and adequate in the high-performance, very reliable, switched LAN environment of the common platform's ATCA backplane.

The next, "packetizer", layer handles datagram fragmentation to allow datagrams bigger than the Ethernet MTU. Note the inverted arrangement compared with TCP/IP where fragmentation is handled *below* reliability. Obviously, the task of reassembling large datagrams is much easier to do and more efficient when layered on top of reliable delivery.

The packetizer header features a so-called "TDEST" ID (having origin in the AXI [7] technology which is widely used in firmware). TDEST serves a similar purpose as UDP port numbers do, i.e., datagrams can be multiplexed to/from different endpoints based on TDEST.

The top layer is the "SLAC Register Protocol" (SRP) which defines simple read- and write- operations to addressable entities such as registers or memory. The use of SRP is optional, i.e., there are use-cases where raw datagrams (e.g., "events" or "streams") are delivered to or received from the application.

With TDEST multiplexing it is possible to share a UDP/RSSI channel between multiple endpoints which may or may not use SRP.

In software, the protocol layers are handled by *protocol modules* which are assembled into a "stack" using the so-called "builder-API" (see below).

COMMON PLATFORM SOFTWARE

While all high-performance operations which require a high, sustained throughput and/or very low latency are implemented in firmware there are still plenty of tasks left for implementation in software: configuration, slow monitoring, diagnostics, etc. A Linux server communicates with firmware over 10 GbE (Fig. 1) and acts as a gateway to a traditional control system such as EPICS. However, rather than directly interfacing the control system to the firmware, an abstraction layer, the "Common Platform Software" (*CPSW*) was designed.

- Provide abstracted access to firmware entities (mostly: device registers) hiding details of Ethernet communication, endianness, offsets, etc.
- Present firmware entities as a hierarchical name space (similar to Linux' sysfs).
- Offers a standardized interface to firmware. Can be accessed e.g., from python (development) and from EPICS (production).

CPSW APIs

An important design goal for CPSW was a clear separation of APIs from implementation and partitioning of APIs according to their core functionality. CPSW is written in C++ and APIs (except for the "developer-API") are defined as abstract classes. Access to an API and the objects it defines is obtained via factories which produce "smart" shared pointers. This approach alleviates the user from having to manage object lifetime explicitly and completely encapsulates implementation details. There are three layers of APIs:

- "User-API". All interactions of applications with the firmware use this API. It offers "lookup" and "access" operations in a hierarchical namespace (e.g., "find a register", "get value").

ISBN 978-3-95450-189-2

Copyright © 2017 CC-BY-3.0 and by the respective authors

- "Builder-API". This API is used to create the hierarchy of object instances which represents the entities implemented in firmware. It uses abstract representations of basic building blocks ("memory-mapped container", "register").
- "Developer-API". This API exposes the headers of the underlying classes and allows existing classes to be extended, etc. It is used to implement new building blocks that can be instantiated via the builder-API and accessed via the user-API.

The user- and builder- APIs are independent views of the underlying classes and are abstract. This guarantees that implementation details can be changed without affecting an application. The separation between user- and builder- API allows for a complete separation of the application proper that accesses and manipulates firmware from the code which constructs the tree of drivers. I.e., when the layout of a firmware register changes then only the "build" process needs to be modified but the core application is guaranteed to be unaffected.

User-API If a user wants to manipulate e.g., a firmware register then the corresponding entry is first located by name (similar to a directory lookup; blue arrows in Fig. 4) yielding a "Path" handle. Access to the functionality offered by an item is performed via a so-called *interface*. A CPSW *interface* is also an abstract class that offers access methods (e.g., "get integer value"). The user attempts to "create" or "open" an interface to the target entry from the Path handle. If the underlying class indeed implements the desired interface then a smart pointer to the interface is passed to the user (green arrow in Fig. 4); otherwise an exception is raised. The user may then go on and access the interface.

Basic interfaces include "ScalVal" (get/set integer value), "DoubleVal" (get/set floating-point value), "Command" (execute an operation).

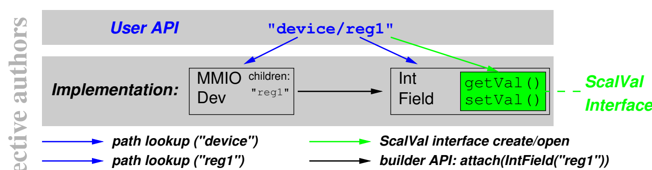

Figure 4: CPSW User-API and Implementation Layers.

YAML Firmware/Hardware Description

Traditionally, a driver-writer often has to extract detailed information about a device (such as register offsets, bitmasks, etc.) from a manual and translate it into definitions which would typically be coded into a header file. This process is cumbersome and error-prone.

The separation between user- and builder- API was also driven by the desire to automate these steps. This is accomplished by the firmware engineer providing device descriptions in the *YAML* [8] data serialization format which meets our requirement of a standardized, human- and machine-readable interchange format. See Fig. 5 for an example.

```
device:
  class: MMIODev
  children:
    reg1:
     class:    IntField
     sizeBits: 32
     at:       { offset: 0x00 }
```

Figure 5: Excerpt YAML Description of Example in Fig. 4.

Larger systems are assembled from device descriptions and result in a YAML file which defines the complete hierarchy implemented in firmware.

A CPSW application can then simply load a YAML description without the need to explicitly use the builder-API. Any firmware modification (as long as it doesn't directly affect functionality "seen" by the user-API) can then be handled in a completely transparent way – provided it is properly reflected in the YAML description.

CPSW Extensions and Dynamic Loading

CPSW supports dynamic loading of driver classes making it possible to extend the core functionality in a modular way.

Python Bindings

Bindings for Python2.7 and Python3 are provided.

CONCLUSION

The new platform supports a lot of common functionality at the hard- firm- and software level which can be leveraged by specialized high- performance systems in order to reduce development time and cost as well as simplify maintenance.

REFERENCES

[1] PICMG, "AdvancedTCA Overview," https://www.picmg.org/openstandards/advancedtca/

[2] J. Frisch *et al.*, "A FPGA Based Common Platform for LCLS2 Beam Diagnostics and Controls," in *Proc. IBIC'16*, Barcelona, Sep. 2016, paper WEBPG15.

[3] R. Herbst, "ATCA Based Accelerator Controls & Detector Platform", EPICS Collaboration Meeting, Oak Ridge, September 2016, https://conference.sns.gov/event/11/session/1/contribution/39

[4] D. Velten, R. Hinden, and J. Sax, "Reliable Data Protocol," July 1984, https://tools.ietf.org/html/rfc908

[5] C. Partridge and R. Hinden, "Version 2 of the Reliable Data Protocol (RDP)," April 1990, https://tools.ietf.org/html/rfc1151

[6] T. Bova and T. Krivoruchka, "Reliable UDP Protocol," *DRAFT*, February 1999, https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00

[7] ARM Ltd., "AMBA Documentation," http://infocenter.arm.com/help/topic/com.arm.doc.set.amba/index.html

[8] O. Ben-Kiki, C. Evans, and I.d Net, "YAML Ain't Markup Language Version 1.2," http://www.yaml.org/spec/1.2/spec.html

Copyright © 2017 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-189-2