THE SLAC COMMON-PLATFORM FIRMWARE FOR HIGH-PERFORMANCE SYSTEMS

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

Abstract

LCLS-II's high beam rate of almost 1MHz and the requirement that several "high-performance" systems (such as MPS, BPM, LLRF, timing etc.) shall resolve individual bunches precludes the use of a traditional software based control system but requires many core services to be implemented in FPGA logic. SLAC has created a comprehensive open-source firmware framework which implements many commonly used blocks (e.g., timing, globally-synchronized fast data buffers, MPS, diagnostic data capture), libraries (Ethernet protocol stack, AXI interconnect, FIFOs, memory etc.) and interfaces (e.g., for timing, diagnostic data etc.) thus providing a versatile platform on top of which powerful high-performance systems can be built and rapidly integrated.

INTRODUCTION

The next generation Linac Coherent Light Source (LCLS) has many High Performance System (HPS) sub-systems:

- Beam Charge Monitor (BCM)
- Beam Length Monitor (BLEN)
- Beam Position Monitor (BPM)
- Low-Level Radio Frequency (LLRF)
- Machine Protection System (MPS)
- Timing System

While each specific HPS sub-system has some unique requirements, they all share the same base requirements:

- Intelligent Platform Management Interface (IPMI)
- Timing Network
- Experimental Physics and Industrial Control System (EPICS) Network
- MPS network

The motivation for developing a HPS common platform is to do the following:

- Identify areas of commonality within LCLS-II, LCLS-I, and SLAC in general
- Define intra HPS interfaces and interconnects
- Ensure application specific firmware and software is consistent with core library, portable, parameterized, and reusable in future systems

The HPS common platform provides the base hardware, base firmware, and base software for all LCLS-II (and eventually LCLS-I) sub-systems [1]. The common platform is an Ethernet Network Access Device (NAD) based module. Figure 1 shows the default packaging solution, which is a 7-slot Advanced Telecommunications Computing Architecture (ATCA) crate.

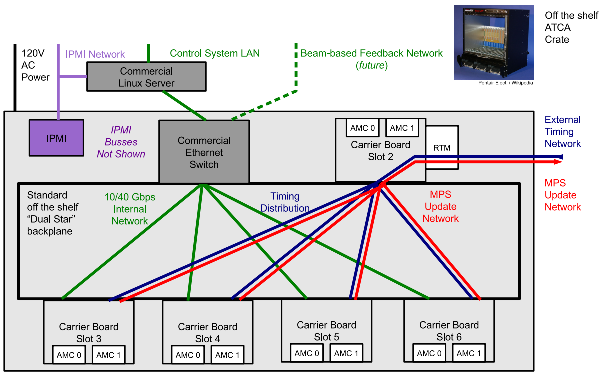

Figure 1: Crate Network Block Diagram.

COMMON PLATFORM HARDWARE

The common platform hardware is an ATCA Advanced Mezzanine Card (AMC) carrier board. A block diagram of the carrier board is shown in Fig. 2, and a photograph of the AMC carrier is shown in Fig. 3. The AMC carrier supports two double-wide, full height mezzanine cards. An AMC is where the application specific hardware exists for a given HPS sub-system. Figure 4 shows some examples of application specific hardware that has been developed for the common platform carrier board. The AMC carrier provides the following interconnects and power to each of the AMCs:

- Unfiltered, switching +12VDC power (up to 9A)
- Filtered, switching +2VDC power (up to 3A)
- Filtered, switching +4VDC power (up to 3A)
- Filtered, switching +6VDC power (up to 3A)
- Filtered, switching +15VDC power (up to 0.5A)
- Filtered, switching -15VDC power (up to 0.5A)
- 7 - 10 high speed FPGA I/Os, (Up to 10Gbps)
- 86 low speed FPGA I/Os, (Up to 1Gbps)
- 2 differential pairs between AMCs on the same carrier
- 2 differential pairs between AMC and RTM

The main controller on the AMC carrier is a Xilinx Kintex Ultrascale Field-Programmable Gate Array (FPGA). There are two loading options for this FPGA: XCKU040-2FFVA1156E (7 high speed links per AMC) or XCKU060-2FFVA1156E (10 high speed links per AMC). XCKU040-2FFVA1156E is the default loading option. Attached to the FPGA is a standard 8GB DDR3 SODIMM for local buffering of large amounts of data.

To help minimize cabling to the common platform hardware, the ATCA back plane is highly utilized. We are using a dual-star back plane. In ATCA slot#1 (1st star connection), we use a Commercial Off The Self (COTS) Ethernet switch

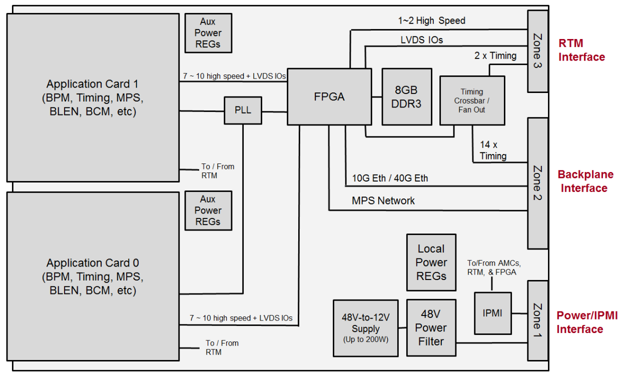

Figure 2: Hardware Block Diagram.

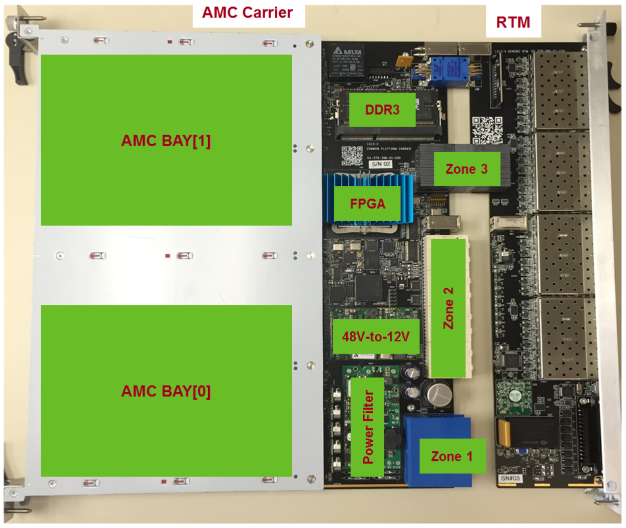

Figure 3: Photograph of the AMC carrier with RTM.

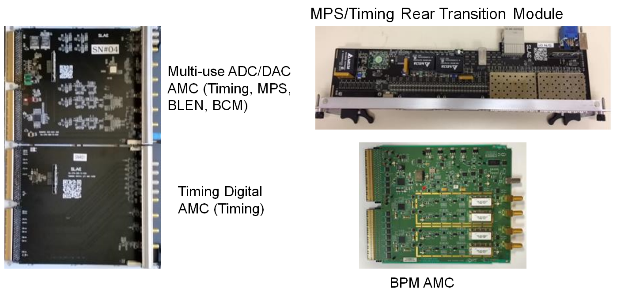
MPS/Timing Rear Transition Module
Multi-use ADC/DAC AMC (Timing, MPS, BLEN, BCM)
Timing Digital AMC (Timing)
BPM AMC

Figure 4: Examples of Application Specific Hardware.

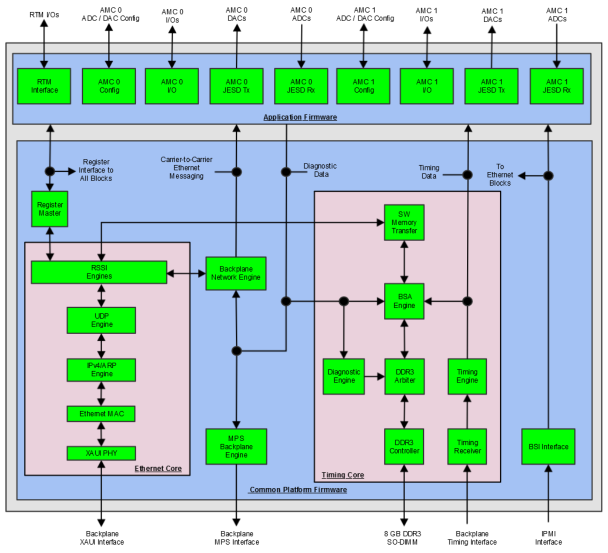

Figure 5: Overall Firmware Block Diagram.

to connect Ethernet to all the AMC carriers via the back plane's zone2 interface. In ATCA slot#2 (2nd star connection), we use an AMC carrier to connect all the other AMC carriers to the timing and MPS networks via the back plane's zone2 interface as well. The slot#2 AMC carrier's external connection to the timing and MPS networks is via a Rear Transition Module (RTM). There is only one RTM required per ATCA crate in the HPS common platform system. The ATCA Zone1 back place interface provides the AMC carriers with -48VDC power (up to 300W) and an IPMI network interface.

COMMON PLATFORM FIRMWARE

The purpose of the common platform firmware is to provide the common firmware interfaces and functional modules required by all or many applications. A block diagram of the carrier board is shown in Fig. 5. The top-level firmware is partitioned into two sections: Common Platform Core and Application Core. The Common platform core firmware is used in all firmware builds whereas the application core is the unique firmware specific to the application.

The SURF Library

The backbone of the common platform firmware framework is the SLAC Ultimate RTL Framework (SURF). SURF is an open source firmware library that is developed, supported and maintained by SLAC on Github [2]. Here are some of the common SURF IP core libraries used application engineers:

- Ethernet: 1000BASE, 10G-BASE, XAUI, IPv4, ARP, DHCP, ICMP, UDP
- AXI4: Crossbar, DMA, FIFO, etc.
- AXI4-Lite: Crossbar, AXI4-to-AXI4-Lite bridge, etc.
- AXI4 stream: DMA, MUX, FIFO, etc.
- Devices: ADI, Linear, Micron, TI, etc.
- Synchronization: Synchronize bits, buses, vectors, resets, etc.
- Wrapped Xilinx IPs: Clock managers, SEM, DNA, IPROG
- Serial Protocols: I2C, SPI, UART, JESD204B, etc.

Application firmware engineers are able to develop their firmware faster by leveraging this firmware library resource.

Networking Support The common platform Ethernet core uses a firmware-based 5-layer Ethernet stack (see Fig. 6). The first layer is a Xilinx 10 Gigabit Attachment Unit Interface (XAUI) physical layer (PHY). We are using the Xilinx XAUI PHY IP core [3]. The Ethernet Media Access Controller (MAC) is the 2nd Ethernet layer. The MAC is part of the SURF firmware library and has the following features:

- MAC address filtering
- Pause Flow control
- Cyclic Redundancy Check (CRC) generation/verification

The MAC address for this firmware module is stored an onboard IPMI PROM. Each AMC carrier is programmed with an unique MAC address.

The IPv4 engine is the 3rd Ethernet layer. The IPv4 engine is part of the SURF firmware library and has the following features:

- Integrated ARP engine for ARP requests/replies
- Integrated ICMP engine to reply back to standard Ethernet pings
- IP address filtering
- IPv4 protocol type filtering
- IPv4 header checksum generation/verification

The UDP engine is the 4th Ethernet layer. The UDP engine is part of the SURF firmware library and has the following features:

- UDP checksum generation/verification
- UDP port filtering/routing

SLAC Streaming Interface (SSI) is a particular messaging definition on top of AXI4 streaming. It uses the AXI4 Stream base protocol with definitions (within the allowable AXI4 Stream standard) for EOFE (end of frame with error) and SOF (start of frame) in the USER fields.

Reliable SLAC Streaming Interface (RSSI) engine is the 5th Ethernet layer and is the reliable communications layer based upon RUDP (Cisco implementation: refer to RFC-908 and RFC-1151) [4]. The RSSI engine handles the handshaking for re-transmission to another RSSI firmware engine (or RSSI software driver) and provides a flow control mechanism. The user application interface to the RSSI engine is a SSI interface.

While our firmware Ethernet stack does support DHCP in the UDP engine, for network management simplicity we do not enable this feature in our common platform firmware implementation. Instead we automatically generate the IP address from the ATCA slot number and ATCA crate ID:

- IP Address: 10.x.y.z
    - x = upper 8 bits of crate ID
    - y = lower 8 bits of crate ID
    - z = 100 + ATCA logical slot #

Each ATCA crate has a unique 16-bit crate ID in our system.

Timing and BSA Support

A block diagram of the Beam Synchronous Acquisition (BSA)/Timing firmware core is shown in Fig. 7. BSA allows synchronously time-stamped measurements across multiple distributed sub-systems (e.g., capture of synchronous orbits involving many BPMs). Measurements are acquired into a deep on-board buffer in real-time and eventually read out by software for off-line processing.

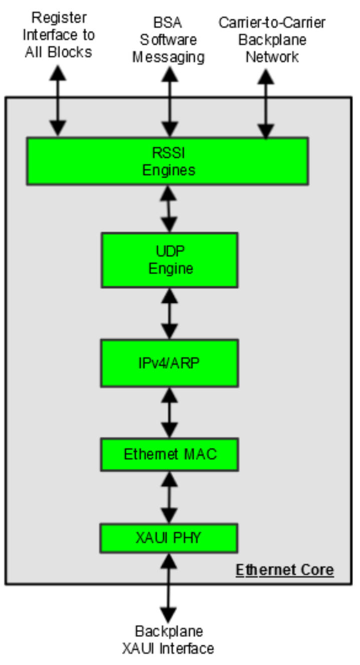

Figure 6: Firmware Ethernet Stack.

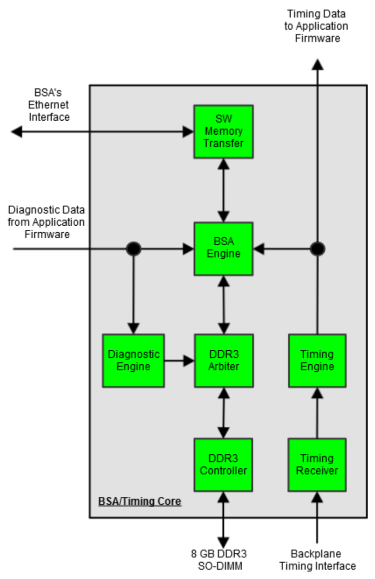

Figure 7: Timing Core Block Diagram.

The timing core is connected to the zone2 back plane timing network and forwards the decoded timing messages to the application firmware and the BSA engine. The BSA engine receives 1 MHz measurement results from the application. The BSA Engine supports up to 64 individual acquisition arrays. The engine queues the application's 1MHz data to DDR3 memory for some subset of the 64 arrays as indicated by the timing message. The timing message also provides an update request to the BSA Engine to notify the application software to initiate read back of the arrays.

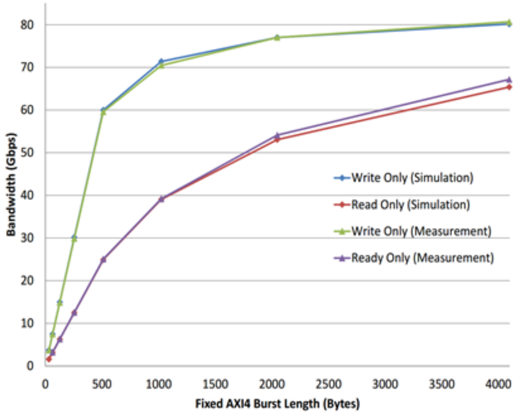

Figure 8: DDR Memory Performance.

We have performed both simulated and non-simulated measurements of the FPGA's DDR3 memory performance. Figure 8 shows the DDR3 memory performance for both simulation and measurement as a function of the AXI burst length. The 64-bit DDR3 memory runs at 1600 MT/s (102.4 Gbps peak throughput). While there are faster than 1600 MT/s DDR3 SODIMMs on the market, the 1600 MT/s is limited by the switching spead of the FPGA I/Os. If we write/read at maximum burst size (4kB), we are able to achieve 80 Gbps writes and 65 Gbps read.

FPGA Boot Process

The FPGA uses a two-stage booting process for loading the firmware. A flow cart of this two stage process is shown in Fig. 9. At power up the FPGA will always load the First Stage Bootloader (FSBL). The FSBL does a DDR memory test before booting into the second stage. The firmware-base memory tester writes pseudo-random data to the entire memory space then reads back the entire memory space to verify pseudo-random data was written. This memory test takes less than 2 seconds after power up. If the memory test failed, the FSBL does not boot into the next stage. If the DDR memory test passes, the FSBL will boot into the second stage. The second stage base address is determined by the IPMI interface. The hardware supports up to 4 firmware images (1 FSBL + 3 non-FSBL) in a 1 Gb FLASH PROM. The non-FSBL images can be remotely reprogramming via the Zone2 Ethernet interface. The FSBL image is protected by the PROM's hardware write protection pin [5] and cannot be reprogrammed without installing a physical jump on the board. If the non-FSBL image gets corrupted during a remote reprogramming process, the IPMI can set the second stage base address to zero to keep the FSBL from transitioning to non-FSBL image. While running the FSBL image the system can remotely reprogram the non-FSBL image to recover from such an failure state.

SOFTWARE SUPPORT

A software block-diagram for a typical application is shown in Fig. 10.

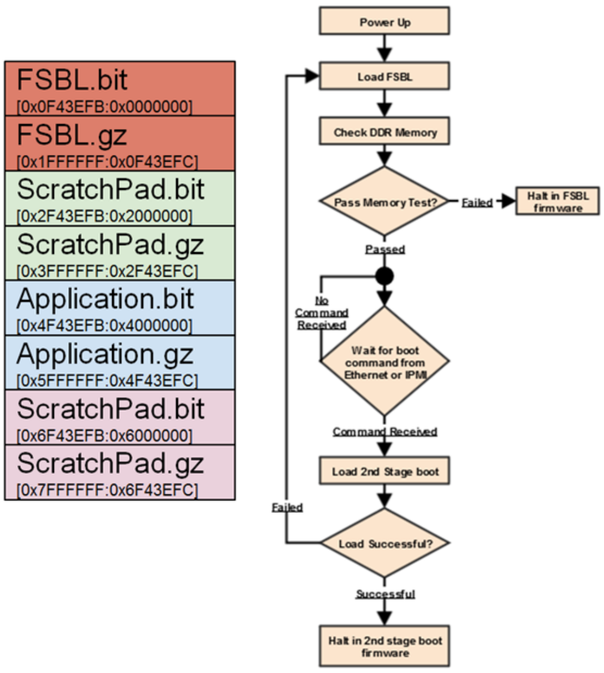

Figure 9: Two-Stage FPGA Bootloading Process.

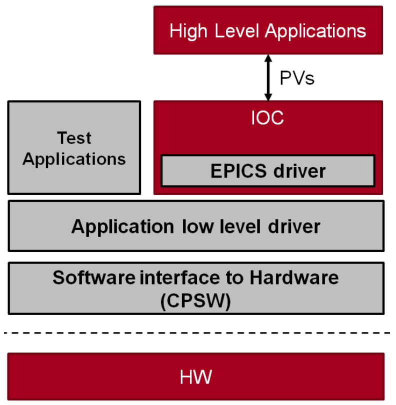
High Level Applications
PVs
Test Applications
IOC
EPICS driver
Application low level driver
Software interface to Hardware (CPSW)
HW

Figure 10: Software Organization.

The Common Platform Software (CPSW) provides a common interface to the FPGA for all high-level software [6]. It is written in C++ and highly objected oriented making extensive use of smart pointers and RAII. It is designed for access to services with varying degree of exposed detail and offers three different APIs with decreasing level of abstraction:

- User API: provides access to a device hierarchy, previously defined using the builder API, without explicit knowledge about details of communication (ports, register addresses, endianness, ...).
- Builder API: it is used for assembling a hierarchy from predefined blocks, defining the topology and providing the necessary parameters, like ports, register offsets, endianness, among others.
- Developer API: it gives access to all details, and can be used for defining more complex objects and extend CPSW itself.

E.g., CPSW provides an elementary building block called *ScalVal*. It represents a generic scalar value with programmable parameters (Bit-width, Bit-offset, Endianness, Signed-ness, Arrays of multiple elements with definable stride), visible to the builder API. From the User API, *ScalVal* provides simple access via the methods *getVal()* and *setVal()*, for reading and writing respectively while all details remain hidden.

Builder API

When building the hierarchy, two main steps are required:

- Create an entity with the desired parameters.

```
IntField property = IIntField::create(
  "property", // name
  16,         // size in bits
  false ,     // is_signed?
  0);         // bit-offset of lsb
```

- Attached the entity to a parent.

```
mmio->addAtAddress(
  property ,  // child
  0x030 ,     // offset
  1);         // number of elements
```

CPSW provides the following building blocks:

- ScalVal: a “leaf”, representing a scalar value/number; directly manipulated by the user API.
- Stream: another “leaf”. Provides raw access to network protocols above the transport layer (RSSI) for special applications.
- MMIO: a memory-mapped container block. MMIOs can be nested.
- NetIO; a container block which implements network communication using a configurable “stack” of protocol modules.

CPSW provides an alternative way of describing the hardware by using YAML files [7]. The YAML files are provided by the firmware build system and represent a concise description of all firmware entities including all relevant parameters (protocol stack details, register offset and sizes, etc.). A CPSW YAML interpreter uses the builder API to automatically construct the system.

User API

In order to access an entity in the hierarchy, three main steps are required by the user:

- Navigate the hierarchy via Path objects which identify an entity using names (string), similar to file system paths. The most basic operation is the lookup, which allows the user to find a specific entity.

```
Path p = root->findByName("/some/dev");
```

- An object which instantiates the desired user interface of the entity can be created.

```
ScalVal property = IScalVal::create(p);
```

- Subsequently, the property may be accessed via the instantiated interface.

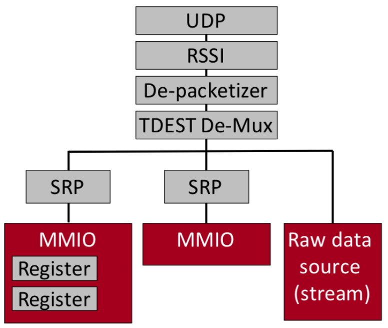

Figure 11: Communication Stack.

```
property->getVal(&int32_variable);
```

FPGA Communication stack:

Communication with FPGAs is established using proprietary protocols layered on top of UDP (see Fig. 11):

- SRP (Slac Register Protocol): RPC-style (but much more primitive) messages fitting into a single MTU. Request/reply for reading/writing 32bit words.
- Packetizer/Depacketizer: Messages which can be much larger than a MTU and are fragmented/reassembled.
- TDEST De/mux: Allows multiple destinations to share a common (De-)Packetizer/RSSI/UDP channel.
- RSSI (see subsection “Networking Support”)

CONCLUSION

The SLAC common platform is a comprehensive framework of hardware, firmware and software components for supporting FPGA-based control- and data-acquisition systems. It greatly simplifies application development, reduces redundant efforts and enhances efficient use of engineering resources.

REFERENCES

[1] J. Frisch *et al.*, “A FPGA Based Common Platform for LCLS2 Beam Diagnostics and Controls, in *Proc. IBIC’16*, Barcelona, Spain, 2016, paper WEPG15.

[2] SLAC Ultimate RTL Framework (SURF) on Github: https://github.com/slaclab/surf

[3] Xilinx’s XAUI PHY IP core Homepage: https://www.xilinx.com/products/intellectual-property/xaui.html

[4] SLAC’s RSSI Documentation Homepage: https://confluence.slac.stanford.edu/x/1IyfD

[5] MT25QL01 Datasheet: https://www.micron.com/~/media/documents/products/data-sheet/nor-flash/serial-nor/mt25q/die-rev-b/mt25q_qlkt_l_01g_bbb_0.pdf

[6] T. Straumann *et al.*, “New Controls Platform for SLAC High-Performance Systems,” in *Proc. PCAPAC’16*, Campinas, Brazil, 2016.

[7] O. Ben-Kiki, C. Evans and I. döt Net, “YAML Ain’t Markup Language,” http://yaml.org/spec/1.2/spec.html