GATEWARE AND SOFTWARE FRAMEWORKS FOR SIRIUS BPM ELECTRONICS

L. M. Russo*, J. V. F. Filho, LNLS, Campinas, SP, Brazil

Abstract

The Brazilian Synchrotron Light Laboratory (LNLS) is developing a BPM system based on the MicroTCA.4 standard comprised of AMC FPGA boards carrying FMC digitizers and an AMC CPU module. In order to integrate all of the boards into a solution and to support future applications, two frameworks were developed. The first one, gateware framework, is composed of a set of Wishbone B4 compatible modules and tools that build up the system foundation, including: PCIe Wishbone master; FMC digitizer interfaces; data acquisition engines and trigger modules. The gateware also supports the Self-Describing Bus (SDB), developed by CERN/GSI. The second one, software framework, is based on the ZeroMQ messaging library and aims to provide an extensible way of supporting new functionalities to different boards. To achieve this, this framework has a multilayered architecture, decoupling its four main components: (i) hardware communication protocol; (ii) reactor-based dispatch engine; (iii) business logic, comprising of the specific board functionalities; (iv) standard RPC-like interface to clients. In this paper, motivations, challenges and limitations of both frameworks will be discussed.

INTRODUCTION

Sirius is a new 3 GeV synchrotron light source under construction in Brazil, with a 0.27 nm.rad natural emittance and 518 meters circumference. The beginning of machine installation is scheduled to the end of 2017 [1].

In this context, a BPM electronics system has been specified, designed and developed by the Beam Diagnostics team at LNLS, reaching its final phase of long-term testing and hardware manufacturing in the following year [2].

As the system was developed from scratch, employing new high-performance data acquisition and communication technologies (e.g., MicroTCA.4, FPGA, FMC, PCIe) [3], the need for an FPGA gateware and software infrastructure frameworks emerged. For that matter, it was sought the use of consolidated codebases and collaborative development through an open source approach. Examples of this were the initial collaboration with the Warsaw University of Technology, the use of a community-driven set of repositories aimed at building generic software/gateware modules from the OHWR collaboration [4] and the use of projects such as the ZeroMQ messaging library [5], the CZMQ High-Level C Binding library [6] and the Malamute Messaging Broker [7] which leveraged many years of development.

In the next sections, the requirements and the details of the these two frameworks, licensed under the copyleft GPLv3 and LGPLv3 licenses, will be described.

* lucas.russo@lnls.br

GATEWARE FRAMEWORK

The gateware framework consists of a set of modules, mainly written in VHDL, that interconnect with each other and to external interfaces. It can be used as a basis to other designs. The general architecture is as depicted in Fig. 1.

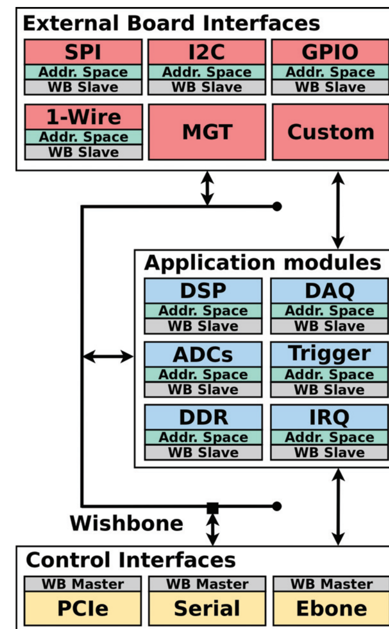

Figure 1: Gateware Framework Architecture.

General Description

The first basic component of the framework is the *Control Interfaces*, as shown in Fig. 1, and it encompasses standard communication interfaces to a controlling node, acting as a Wishbone Master to the gateware side. Currently, 3 types are supported: PCIe Gen1 (for Xilinx Artix7 FPGA) and Gen2 (for Xilinx Kintex7 FPGA), with accompanying linux driver with PIO, single buffer and scatter-gather DMA; RS232 Syscon for simple serial communication supporting auto-baud generation; preliminary support for Ethernet MAC + Etherbone [8] for UDP communication and userspace software library.

The second layer, *Wishbone Infrastructure*, is a set of Wishbone modules and functions that provide the general interconnection between controllable modules (i.e., that can receive/transmit configuration parameters and/or low-bandwidth data) and address space enumeration with SDB [9] support. This is the standard adopted within the framework for consistently interfacing with all *Control Interfaces*.

The third layer, *Application Modules*, is where all of the framework functionalities reside, comprising components from third-parties and LNLS Beam Diagnostics team. A

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

ISBN 978-3-95450-189-2

variety of modules are supplied, ranging from DSP modules (e.g., adders, multipliers, dividers, filters, CORDIC), DDR interface and Interrupt Management, to Data Acquisition, ADC interfaces and Trigger logic.

The forth and last layer, *External Board Interfaces*, is a set of board communication protocols used to interface with Multigigabit standards (i.e., MGT) and to configure or interface with board peripherals, such as: SPI (ADC, clock circuit configuration, etc.), I2C (EEPROM, temperature/-voltage/current sensors, etc.), 1-Wire (EEPROM) and GPIO (LEDs, generic interfaces, etc.). Also, there are some custom interfaces implemented, like a parallel ADC interfaces for data acquisition.

Development

The gateware framework is a development and integration effort from a diverse set of modules that took place in the context of the Sirius BPM electronics focusing on creating generic modules, written in portable VHDL/Verilog to avoid vendor lock-in and to encourage reusability. This encouraged that, besides the primary target platform (i.e., BPM electronics), the Sirius MicroTCA.4 timing receiver also started porting its gateware to this framework.

The development is coordinated through the GitHub platform [10] following a simple branch model and fork+pull requests. Currently, the framework is being versioned together with its main application (i.e., BPM), but a separate repository is planned for simpler integration with other projects.

SOFTWARE FRAMEWORK

The software framework, called *HALCS* (Hardware Abstraction Layer for Control Systems) [11], can be defined as a software daemon that abstracts away a given hardware platform and its functionalities by means of a common interface to hardware and generalized set of *specific application modules*. It is written in C99, implementing a simple, yet effective, scalable object-oriented API, following the guidelines in [12]. The framework was thought as a way of simplifying the development and deployment of applications, while keeping it extensible and flexible. The general framework architecture can be viewed in Fig. 2.

General Description

HALCS main components are: *Hardware Abstraction Layer* (yellow solid box in Fig. 2), defining a common interface for all *Board Support* (yellow dashed box in Fig. 2) implementations, such as PCIe and TCP/IP; *Dispatch Engine* (green box in Fig. 2), that receives message requests from upper-layers and safely demultiplexes them according to the selected *Board Support*; *Specific Modules Layer* (blue boxes in Fig. 2), implementing application-specific logic, receiving message requests from external clients through an RPC interface and ordering lower layers to perform a desired set of operations; *Client Interface* (red boxes in Fig. 2), providing an external API for external clients to communicate with the application.

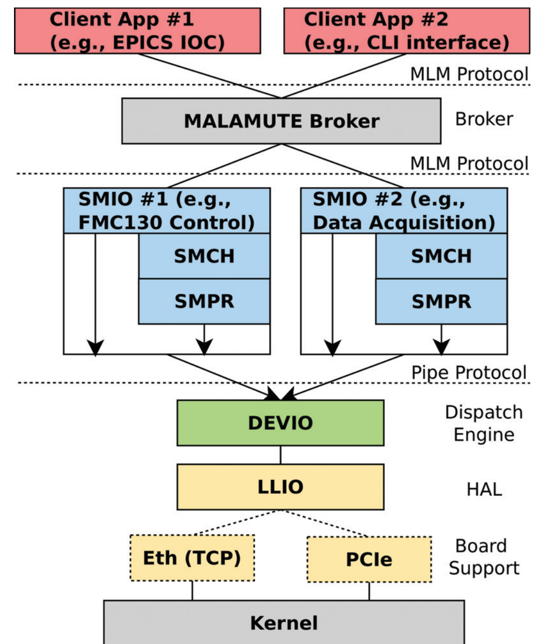

Figure 2: Software Framework Architecture.

The framework also relies on an external standalone messaging broker called *Malamute* [7] to provide reliability, authentication and mailbox messaging pattern. In the future, *HALCS* is planned to use, besides the mailbox pattern, a higher-efficiency, asynchronous, stream pattern (ideal for data acquisition applications) and, possibly, a service pattern for modules implementing logic that can be replicated, like some calculation engine or some data processing logic. Both of these patterns are also available in the broker API.

Hardware Abstraction Layer

The *Hardware Abstraction Layer* is implemented as a standalone software library, called *LLIO*, for Low-Level Input/Output, and acts as a generic interface for hardware operations. The selection of which interface to use (currently PCIe or TCP/IP) is done at compile time and can be easily changed by opting for another implemented set of *LLIO* operations, as long as it follows the same *LLIO* interface.

Dispatch Engine

The *Dispatch Engine*, called *DEVIO* in Fig. 2, is the core of the framework. It acts both as the entry point for a constructed application using the framework and a serialization engine to safely access the hardware without the use of traditional synchronization points (e.g., mutexes, semaphores). Instead, all of the communication is done by sending/receiving messages between layers, improving decoupling, flexibility and avoiding programmer errors in protecting shared resources. Typically, the *DEVIO* layer performs the following steps:

1. Calling thread calls devio_new () method for creating a new instance of *DEVIO* layer
2. *DEVIO* registers the selected *LLIO* operations

ISBN 978-3-95450-189-2

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

3. (Optionally) *DEVIO* tries to parse an SDB structure located in hardware to dynamically enumerate its modules as *SMIO* modules
   3.1. If successful, spawns each *SMIO* module as a new CZMQ actor, executing its entry point routine and opening an inter-thread communication (inproc), using PAIR-PAIR ZeroMQ sockets (Pipe Protocol in Fig. 2)
4. *DEVIO* opens a control socket to the calling thread, so it can send commands to *DEVIO* at anytime, such as: REGISTER_SMIO, to register a new *SMIO* (executing the actions of step 3.1.) and UNREGISTER_SMIO, to unregister an *SMIO* (executing the opposite actions of step 3.1.)
5. *DEVIO* starts its event-driven reactor engine to wait for commands from the registered *SMIOs* and dispatching the appropriate messages to the *LLIO* layer

Specific Modules Layer

This layer is where all of the business logic resides. It is composed of self-contained modules that implement specific operations and communicates with lower layers by sending/receiving messages through the PAIR-PAIR socket opened by *DEVIO*. For higher layers, it uses the MLM protocol [7] and, on top of that, a simple RPC protocol to export its functionalities to external clients.

In order to register new functions, one would have to fill a description structure informing its opcode, how many and what type of the arguments it receives and the type and size for the return value. After that, the exported function can be implemented by following the function signature: `int (*disp_table_func_fp)(void *owner, void * args, void *ret)`. Finally, all that is left is associating this function to the description structure and registering it in the SMIO exported operations. All of these actions are available in the API, through functions.

There are also two optional layers inside *SMIO*, called *SMPR* and *SMCH*, showed in blue boxes Fig. 2. They can be used to implement specific protocols (e.g., SPI, I2C, 1-wire, GPIO) and specific operations on top of it (e.g., Clock-distribution IC using SPI protocol, Programmable Crystal Oscillator using I2C protocol for configuration).

Currently, the *SMPR* layer implements interfaces with the OpenCores SPI and OpenCores I2C gateware modules. These modules are controllable through a simple set of registers and implements the respective protocol. The *SMCH* layer, which is independent of the *SMPR* and only relies on a standard interface to communicate with it, implements various IC interfaces, such as: ISLA216P ADC, AD9510 PLL and clock-distribution, EEPROM, Si57x crystal oscillator, I2C switches, etc. In fact, one can change the *SMPR* protocol of any *SMCH* chip by simply choosing another protocol for it. Any specificities can be controllable through the specific *SMPR* protocol API, while the implementation of both layers remain unaffected.

Client Interface

The *Client Interface* is basically a set of wrapper functions that encapsulates the necessary arguments and return types for each SMIO exported function. There is also a generic API for sending/receiving commands to SMIO functions that relies on the description structures for checking the type, arity and size of arguments. This layer forms the API in which client programs use to communicate with a given application implemented with *HALCS*.

FUTURE WORK

Currently, *SMIOs* only have the mailbox interface, implementing a synchronous request-reply RPC, available through *Malamute*. In this pattern, each *SMIO* receives a mailbox in the broker and can send/receive messages to that specific mailbox. However, in some cases, like data acquisition, a more suitable pattern is a data stream (i.e., publish-subscribe) in which each client that wants to receive some published data registers to a certain *topic* and any control can be set through the regular mailbox pattern API. This in fact creates 2 protocols that implement specific use cases: a high-performance protocol for large amounts of data data (e.g., data acquisition); low-performance protocol for control and monitoring (e.g., configuration parameters). In order to fix these issues, an event API for asynchronous communication and a stream API for high-performance data transmission is under study and should be implemented soon.

CONCLUSION

Two frameworks for developing interfaces to hardware application boards were presented. Currently, two applications are using the frameworks inside LNLS, namely BPM electronics and MicroTCA timing receiver, but more are envisioned. New features are included with relatively ease and new APIs for high-performance data streaming and asynchronous communication will be implemented over the next months.

ACKNOWLEDGEMENTS

The author of this paper would like to acknowledge the collaborative work of Adrian Byszuk from Creotech Instruments SA, for the development of the FPGA PCIe core and the linux driver, Andrzej Wojeński from the Warsaw University of Technology, for the initial gateware and software codes, OpenCores and Open Hardware (OHWR) collaborations, for an excellent combined set of software/gateware/hardware projects and the initiative to embrace and push forward the development of free and open source projects.

REFERENCES

[1] A. R. D. Rodrigues *et al.*, "Sirius Accelerators Status Report", in *Proc. IPAC'15*, Richmond, VA, USA, May 2015, paper TUPWA006, pp. 1403–1406.

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

ISBN 978-3-95450-189-2

[2] S. R. Marques *et al.*, "Status of the Sirius RF BPM Electronics", in *Proc. IBIC'14*, Monterey, CA, USA, Sep. 2014, paper WECYB3, pp. 505–509.

[3] D. O. Tavares *et al.*, "Development of an Open-Source Hardware Platform for Sirius BPM and Orbit Feedback", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, paper WECOCB07, p. 1039.

[4] Open Hardware Collaboration, `http://www.ohwr.org`

[5] ZeroMQ Messaging Library Project, `http://zeromq.org`

[6] CZMQ High-Level C Binding Project, `http://czmq.zeromq.org`

[7] Malamute ZeroMQ Messaging Broker Project, `https://github.com/zeromq/malamute`

[8] Etherbone Core Communication Project, `http://www.ohwr.org/projects/etherbone-core`

[9] Self-Describing Bus Project, `www.ohwr.org/projects/fpga-config-space`

[10] Beam Position Monitor Gateware Project, `https://github.com/lnls-dig/bpm-gw`

[11] Hardware Abstraction Layer for Control Systems Project, `https://github.com/lnls-dig/halcs`

[12] CLASS ZeroMQ RFC, `https://rfc.zeromq.org/spec:21/CLASS`

ISBN 978-3-95450-189-2

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