ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-THPV033

REUSABLE REAL-TIME SOFTWARE COMPONENTS FOR THE SPS LOW LEVEL RF CONTROL SYSTEM

M. Suminski[*], K. Adrianek, B. Bielawski, A. C. Butterworth, J. Egli, G. Hagmann, P. Kuzmanovic, S. Novel Gonzalez, A. Rey, A. Spierer, CERN, Geneva, Switzerland

Abstract

In 2021 the Super Proton Synchrotron has been recommissioned after a complete renovation of its low level RF system (LLRF). The new system has largely moved to digital signal processing, implemented as a set of functional blocks (IP cores) in Field Programmable Gate Arrays (FPGAs) with associated software to control them. Some of these IP cores provide generic functionalities such as timing, function generation and signal acquisition, and are reused in several components, with a potential application in other accelerators.

To take full advantage of the modular approach, IP core flexibility must be complemented by the software stack. In this paper we present steps we have taken to reach this goal from the software point of view, and describe the custom tools and procedures used to implement the various software layers.

INTRODUCTION

The new LLRF system for the SPS has largely replaced old VME-based hardware with a modern installation designed around microTCA crates using PCI-express as the bus.

The update has brought many benefits, one of them being higher hardware density: what used to take a VME crate filled with modules, now is replaced with one or two microTCA cards. It has also affected the control system, as component size has shrunk from a card to an IP core.

Previously, each component was implemented as a VME module, with its own driver, user-space library and application. With the new hardware, the said approach was no longer applicable, therefore a new solution had to be defined.

WORKFLOW

This section will illustrate typical steps needed to develop reusable firmware and software. Overview of the layers constituting a component is presented in Fig. 1.

Memory Map

The process begins with hardware interface definition, starting with individual IP cores and ending with the top level map containing all components used in a card. The interface is called memory map since it defines mapping registers and memories to offsets in a card memory space. Memory map serves as the primary data source for both firmware and software developers.

Each register is described by a number of attributes, such as name, access mode (read-write/read-only), bit width, valid value range or conversion functions between raw register value and physical units.

[*] maciej.suminski@cern.ch

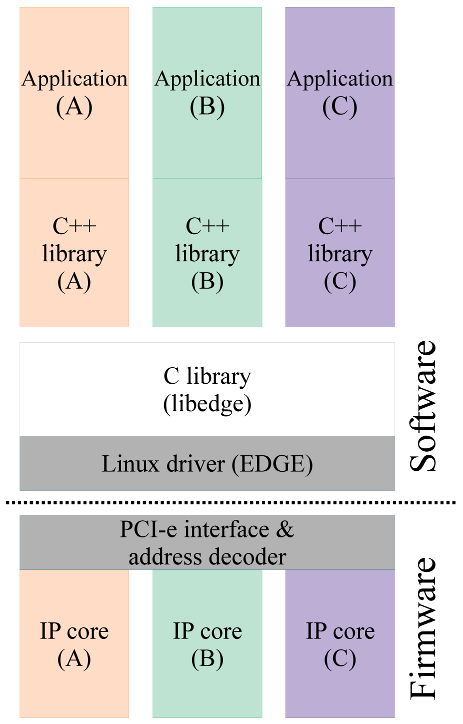

Figure 1: Example of a card implementation with several components. The white block (C library) is common to all cards and components. The grey blocks (Linux driver & address decoder) are card specific. The remaining colors show parts of reusable component stacks, each color representing a different component.

Registers may have subcomponents called fields, providing a way to give different meaning to a set of bits belonging to a particular register. This method is frequently used for status and control registers, where each bit represents a different part of the logic.

Each memory map may also include other memory maps, allowing the designer to reuse existing ones and establish a tree-like structure. The latter method is commonly applied for composing IP cores to define a card interface, also known as the top level map.

Memory maps are edited using a dedicated tool named Reksio (former Cheburashka [1]). The tool offers a graphical user interface aiding the users in memory map creation and validation. It also provides straightforward access to external tools, such as generators for various layers of the component stack.

Once a memory map is finished, it serves as the source to generate several layers of the component stack:

- FPGA firmware template
- Linux driver
- C++ library

Generated components will be discussed in detail in the following sections.

FPGA Firmware Template

The firmware designer begins with executing the Cheby [2] tool to generate skeleton code in VHDL, creating the foundation for firmware development. The obtained code supplies a bus interface used for data transfers and a set of signals directly representing registers. The latter part is used by the firmware designer to implement the actual component logic.

This step may be executed at the component level, resulting in self-contained, component-specific code or at the top level in order to generate an address decoder for a card.

Linux Driver

Another part of the toolset is the Encore Driver Generator (EDGE) [4], which translates a memory map file to Linux driver source code.

While in certain cases it would be possible to bypass the driver by directly accessing memory-mapped device registers, having a driver brings certain benefits. In a microTCA module, the driver is needed to enable Direct Memory Access (DMA) transfers or handle hardware interrupts.

Apart from driver generation, EDGE also provides a user-space shared C library to communicate with the hardware. The library interface allows the software developer to access any register by its name, using a driver specific memory map file. Thanks to that, the library is common to all components and is not included in the component stack.

It is worth mentioning that EDGE supports both PCI/PCI-express and VME, which is especially convenient when a component needs to be migrated to a different platform.

Please note that the Linux driver is card specific and does not belong to any particular component.

C++ Library

The next layer of the software stack is a C++ library, also generated from a memory map. It uses the EDGE library for accessing the hardware, and as such requires a driver generated with the aforementioned tool. While the C++ library is not strictly required for the application development, it facilitates the software development process.

First of all, generated C++ libraries provide a user-friendly interface, where the tree-like hierarchy of a memory map is reflected in a set of classes, each representing a submap, register or field. Such an approach allows the developer to easily traverse the structure by accessing appropriate fields in the generated classes.

Moreover, the generated library takes advantage of certain properties defined in the memory map:

- Data type (bit width and signedness): ensures correct register value interpretation.
- Valid value range: prevents setting registers to values which are not considered correct.
- Functions to convert between raw register values and physical units: enables writing application code using values expressed in physical units without any additional effort.

Real-Time Application

The last layer of the software stack is the real-time application. At CERN, such applications are developed using the Front-end Software Architecture (FESA [5]) framework with C++ as the programming language.

Most applications built using FESA consist of two parts:

- Server: used for communication with the user or higher level applications. It provides a way to specify new settings or read data acquired from the hardware.
- Real-time: controls the hardware synchronously to the timing system. This part most often configures the card for a particular accelerator cycle and reads back data from the hardware.

Application code might also be generated, but the interface would correspond directly to the memory map. In most cases, applications are expected to implement a user-friendly interface and not one that gives a direct access to the hardware. Usually this layer takes most time to implement.

VERSION VALIDATION

It is usual for components to evolve as new requirements arise. If a change is made to hardware, then it needs to be propagated through each layer upwards in order to avoid unexpected behaviour, which is often difficult to analyze.

For this reason it is essential to verify if all layers of a component are compatible with each other. The simplest way to achieve that is by assigning each layer a version number. In the new SPS LLRF system, the numbers are assigned in accordance with semantic versioning [5] scheme, which defines clear rules regarding which versions are compatible with each other.

For hardware, it has been decided that the first registers in each memory map constitute a standard header providing firmware and memory map version numbers. This is true for both top level and component memory maps.

The Linux driver version is expected to match the firmware version of the card. EDGE offers a way to specify an automatic version check by specifying a register and its expected value range. If the register value is outside of the range, the driver will refuse to load. This step assures memory map compatibility between software and hardware domains.

The user-space EDGE library (libedge) executes another test. When a device is opened, the library reads its driver version and loads the corresponding memory map file, which will be used for obtaining register offsets. At this stage the library also computes the memory map file checksum and compares it with the one stored in the driver. Further access to the card is possible only when the two numbers match.

Finally, each user-space application is also expected to validate the hardware version. Normally this is done by comparing the component firmware version against the version supported by the application.

EXAMPLES

During the SPS LLRF upgrade, several common components have been developed to be applied in Cavity Controllers and Beam Control microTCA cards:

- Timing generator core: programmable timer generating pulses with requested delays.
- Acquisition core: generic component for sampling data inside the FPGA and transferring it to a memory.
- Resampler core: converts acquired data between fixed sampling rate and beam synchronous rate.
- Function generator core: delivers time series data according to a programmed pattern.
- Numerically controlled oscillator core: generates samples corresponding to a sine wave at the requested frequency, with certain customizations for application in accelerators.
- Gigabit link core: controller for a gigabit serial link used for data transfer.

Each of the above components has an implementation in all the presented layers apart from the Linux driver, which is card specific. If one of them needs to be instantiated in another system, the whole stack can be reused.

CONCLUSION

The described workflow has been successfully applied during the SPS LLRF renovation. The process has delivered several self-contained components, covering all layers from hardware to software application. The components have been reused in two cards without any modification and are potentially applicable in other systems, including non-microTCA ones.

The new approach has improved task coordination in the development process, since several components could be developed independently and tested with any available hardware.

REFERENCES

[1] P. Plutecki, B. Bielawski, and A.C. Butterworth, "Code Generation Tools and Editor for Memory Maps", in *Proc. ICALEPCS'19*, New York, NY, USA, Oct. 2019, pp. 493-496. doi:10.18429/JACoW-ICALEPCS2019-MOPHA115

[2] Cheby, https://gitlab.cern.ch/cohtdrivers/cheby

[3] EDGE Driver Generator, https://gitlab.cern.ch/cohtdrivers/encore

[4] M. Arruat *et al.*, "Front-End Software Architecture", in *Proc. ICALEPCS'07*, Knoxville, Tennessee, USA, paper WOPA04, p. 310-312.

[5] Semantic Versioning, https://semver.org