DESIGNING A REUSABLE INSTRUMENT INTERFACE

T. Karcnik, M. Sprogar
Instrumentation Technologies, Solkan, Slovenia

ABSTRACT

The paper describes one possible approach to designing a reusable instrument interface applicable to a family of individual devices. The main objective is to offer a uniform top level interface accross the entire family while maintaining the flexibility by utilising clearly designed generic building blocks, tied together in a flexible way. This approach works also for devices where firmware is an integral part of the system design. The proposed approach is demonstrated on an actual case.

INTRODUCTION

A control group is faced with several issues when integrating an instrument or a device into a complex control system (CS). The first step is usually building a standard CS stack for each instrument, usually involving EPICS, TANGO, etc., and a screen/display for the operator. A typical hierarchy in a modern device may look like:

- CS software,
- interface between the CS software and instrument system software,
- instrument system software,
- firmware (*e.g.*: FPGA) and,
- hardware.

However, the glue layer or interface between the CS software and the instrument is typically very much case specific. For each instrument a dedicated interface has to be written; typically this means at least a device driver for the selected CS stack. With the recent advancements in the reconfigurable hardware, e.g. FPGA, it is now possible to use the same physical hardware to architect various instruments. It would be highly inefficient if all the software up to the CS interface had to be rewritten for each new application; not to mention maintenance of all the variants.

The following sections explain the principles and demonstrate one solution to the above stated problem.

STATE OF THE ART

In general, an instrument/device has several input and output channels: e.g. Beam Position Monitor (BPM) receives pick-up signals and passes calculated results to the CS and/or feedback system. In addition to obvious elements, there are some common low-level elements built into each device, that are transparent to a CS user:

- DSP chains,
- temporary high-rate data storage,
- timing and triggering subsystems,
- house-keeping (hardware control) and
- monitoring (e.g: failure control).

Data storage and timing/triggering subsystems are the most generic modules. Regardless of instrument usage, the timing system remains the same in an accelerator. The data storage subsystem is used to provide buffering to slow and non real-time data access from the CS.

Contrary to the mentioned subsystems, the DSP chains are of course device dependant. They connect the hardware defined input/output data channels. Thus, a generalized approach is possible if the DSP chains are parameterized in a generic way.

Housekeeping and monitoring subsystems are the same if identical hardware is used. Otherwise, the interface to the higher software layers should be preserved to the maximum extent possible.

All of the above mentioned tasks are executed in a hard real-time environment and are thus not suitable for a general purpose computer. For modest speeds, a real-time computer suffices, while for

cutting edge performance these tasks are implemented in the hardware, using FPGA's in the most cases.

The functionality implemented in hardware is complemented by the software stack. A System-on-Chip (SoC) or a Single Board Computer (SBC) is used extensively to provide a standard interface to the control system. Both are running an embedded operating system with network support. The system software interfaces hardware on one side and CS on the other. System software relies on well defined interfaces on both sides in order to assure portability of the system software and facilitate maintenance of the CS interface across a range of instruments. An excellent example of a portable interface connecting data consumers to data providers is ODBC, which is well established and a de-facto standard for cross-operability in the database business. Since the system software mirrors the hardware functionality it is therefore essential that the low level functionality adaptations are kept at the minimum.

As a side benefit, the amount of work needed to maintain the system and low level software is kept at the minimum, thus enabling faster development cycles and easier maintenance.

CASE STUDY

Instrumentation Technologies delivers a family of instruments under the Libera trademark. Two representatives are the Libera Electron Beam Position Processor (EBPP) and Libera Bunch-by-Bunch Feedback Processor (BBFP). The two are based on similar hardware platforms: a digital board designed around the Xilinx FPGA Virtex II Pro and a Single Board Computer. However, the analogue front-ends are different. In both cases, the signals are eventually delivered to the FPGA after the A/D conversion.

The data flows are similar in both cases. Figure 1 shows the data flow for the Libera EBPP.

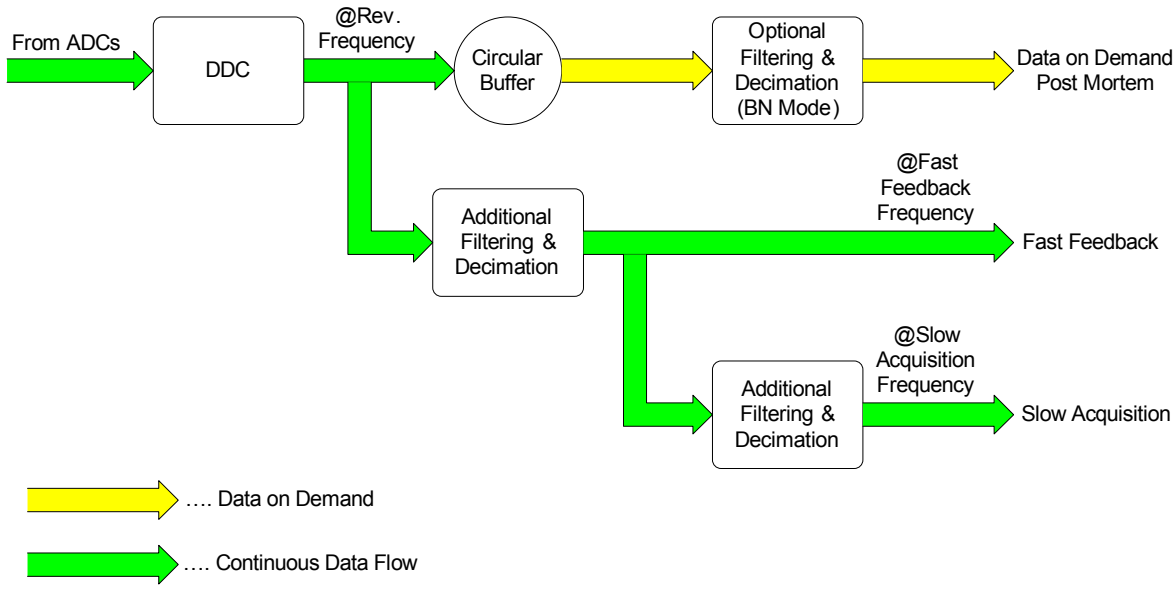

Figure 1: Libera EBPP data flow

The sampled data is passed through the DDC and then divided into three branches:

- history buffer for Data-on-Demand,
- two continuous data streams for:
  - Fast Feedback and,
  - Slow Acquisition.

The output from Data-on-Demand and Slow Acquisition is passed to the CS, using SBC. Contrary to that, the Fast Feedback output is too fast and is dealt with at the FPGA level instead.

The data flow for Libera BBFP is similar.

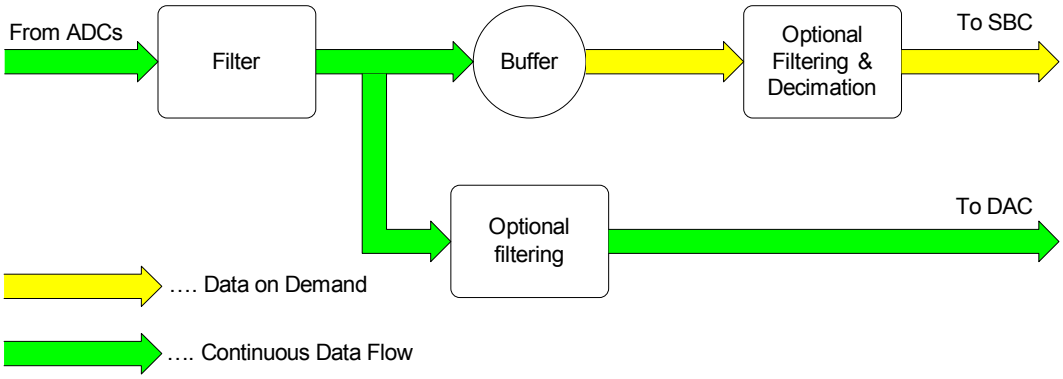

Figure 2: Libera BBFP data flow

The fast continuous stream is again dealt with at the FPGA level, while the Data-on-Demand is again passed to the SBC and then CS.

Although the functionality and hardware of both instruments are vastly different, some similarities exist:

- data flow is similar and,
- data handling resembles.

The differences, for instance, include initialization and configuration.

The instrument software relies on a clearly designed software structure. The software structure is shown in figure 3.

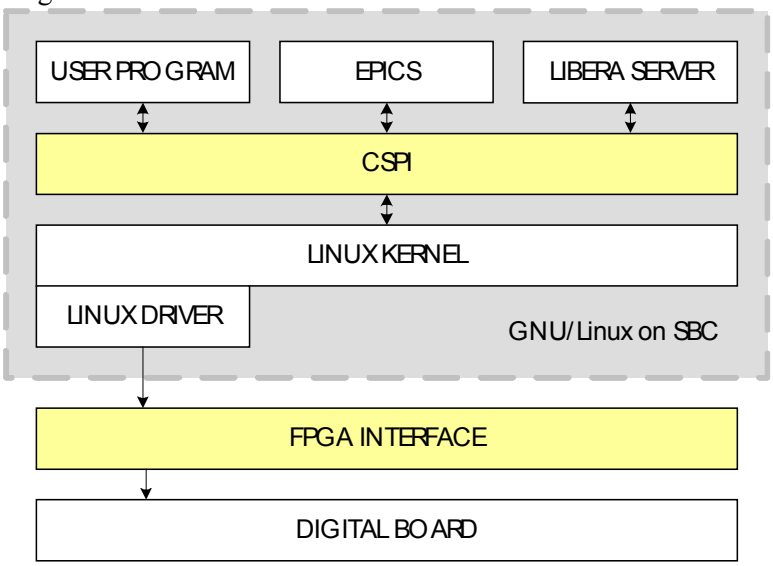

Figure 3: Software structure

The SBC computer runs a stripped down GNU/Linux operating system. All software interfacing the CS runs in user-space and connects to the hardware using a device driver integrated into the kernel. The most important and visible part from the end user point of view is the Control System Programming Interface (CSPI) library. It provides a consistent C API, making low level details transparent for an application developer. The core of the CSPI is the same across the entire family of Libera instruments. Instrument specifics such as fast feedback configuration for Libera EBPP are modularized and tied to the core using an internal interface. CSPI together with a standard environment and GCC based development tool-chain forms a highly efficient way of instrument integration into the accelerator control system.

Since the CSPI is consistent across the range of instruments, the same is then true for a custom application interfacing the CS. A new member of the Libera family requires only minor changes to, for example, the EPICS device driver. The necessary modifications can be easily dealt with at compile time. Thus the same code base can be used for all instruments using the CSPI library.

CONCLUSIONS

In order to minimize the variations required by various tasks several guidelines must be followed:

- the system software must provide a consistent, well thought-out interface;
- the real-time and non real-time tasks must be clearly separated;
- FPGA code must use well defined building blocks and,
- all components must be designed with modularity as one of the primary goals.

With all these requirements met, it is possible to cover a range of functionalities with a near-zero modifications to the high-level, end-user software.

A case study was presented where the principles described above were utilized to achieve the desired goals for a family of products that share similar hardware platform and are based on the same source code. The design follows the principles described above.