

A Special-Purpose Processor System with Software-Defined Connectivity

Benjamin Miller, Sara Siegal, James Haupt, Huy Nguyen, and Michael Vai
MIT Lincoln Laboratory, Lexington, MA 02420
{bamiller, ssiegal, jhaupt, hnguyen, mvai}@ll.mit.edu

Introduction

An open-architecture special-purpose processor (SPP) system that can host a variable number of receivers, transmitters, and processors is being developed for a broad class of DoD systems (ISR, COMMS, EW/EA, etc.). The salient feature of this SPP system is the component connectivity, which is defined by a low-latency, point-to-point, high-speed switch fabric that is user programmable. As the functionality of this system is completely defined by its software (CPUs), firmware (FPGAs), and connectivity, it can be readily constructed as a high-performance, heterogeneous processing system. In addition, such a system is inherently anti-tamper as the critical program information is entirely contained in software and firmware.

Motivation

One example application of the SPP system is a reactive jammer. Development challenges of this application are numerous. The implementation has to meet SWaP requirements under extreme environmental constraints. The operation must have very low latency. In addition, the asymmetric warfare nature of “Overseas Contingency Operation” requires a jammer to be adaptable to new technologies (e.g., new devices) and operating environments (e.g., rural vs. urban). A successful jammer system must thus be scalable, modular, upgradeable, and serviceable.

Architecture

Figure 1 shows the architecture of an exemplar SPP system. The communication paths between components are fully user definable. The general data flow path of this architecture is as follows. The receivers pick up signals in their tuned bands. The signals are digitized and passed on to the processors for detection and discrimination. Based on the results, appropriate response waveforms are sent to the transmitters.

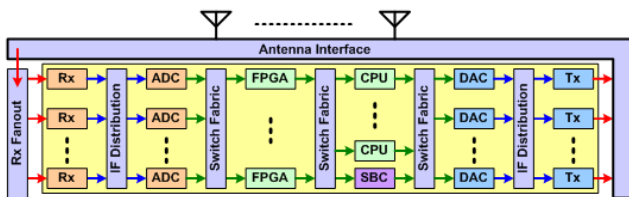


Figure 1: The architecture of an SPP system

With software-defined connectivity, the architecture shown in Fig. 1 is fully configurable in different situations and environments. Depending on the received signal type, different processing paths may be needed. For example, in

This work is sponsored by the United States Navy under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

one configuration, the output of a receiver is connected to

an FPGA processor, which generates and sends a response to a transmitter. In another case, the output of the FPGA may be sent to a CPU for further discrimination. It is the role of the CPU to provide an adequate response to the transmitter.

The architecture also provides the flexibility and scalability for different form factors (e.g., hand-held vs. fixed-site) and operations (e.g., urban vs. rural). For example, a hand-held device may have a single receiver-transmitter pair while a fixed-site device may have a whole band of receivers and transmitters to cover a wider spectrum. The reconfigurability of the architecture also provides fault tolerance and optimal use of limited resources.

Most of the applications targeted by the SPP system require low communication latency between components. The implementation of an SPP must thus have minimal overhead. After evaluating a number of general-purpose products, the development team chose to follow the radar open system architecture (ROSA) model [1] and create a bare-bone middleware library to minimize overhead. Like the radar thin communications layer (RTCL) library, this library uses a publish/subscribe paradigm to release applications from the need to keep track of their receiving clients.

In the SPP software architecture shown in Figure 2, the thin communications layer is the key component. In addition to the implementation of a low-overhead publish/subscribe communication paradigm, the thin communications layer also isolates HW components (Rx/Tx, ADC, etc.) from the system control components. Similarly, the application components (software) are also isolated from the hardware details. The application interface is implemented in two levels. First, it communicates to the thin communications layer. Second, it employs signal processing functions from a library that has been adapted to the thin communications layer.

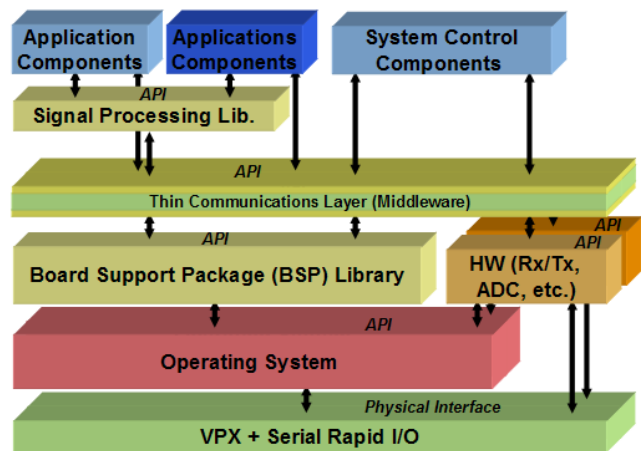


Figure 2: SPP software architecture

The middleware library has a few abstract classes that can be used to implement the functionality that allows

components to communicate with each other. The DataWriter class allows a publisher component to send data to all components that subscribe to its topic. A subscriber receives a notification from a DataReaderListener when data is published to a topic of interest, then uses a DataReader to read in the data. Since serial RapidIO (sRIO) is used for communication in the current system, concrete sRIO-specific classes SrioDataWriter, SrioDataWriter and SrioDataReaderListener have been derived. Applications will call member functions of these concrete classes for data transfer between components.

In order to implement an open system architecture, the middleware library provides a Builder class to isolate applications from the vendor-specific physical interface drivers, commonly referred to as board support packages (BSP). Following the methodology described in [2] with the DataWriters and DataReaders acting as directors, the user derives a class from Builder that implements a number of virtual functions such as reserveChannel() and performDMAxfer(). These functions serve as the interfaces to board-specific BSP libraries. As these interfaces are defined in the base class, the user-level DataWriters and DataReaders do not need knowledge of these implementation details.

The architecture openness of the system is ensured by the fact that a new component, in addition to comply with the physical interface (i.e., pin compatability), only needs to supply its board-specific implementation of the pure virtual functions. Its BSP must be compatible with the API defined in the thin communications layer middleware library.

The data communication between applications is handled by the middleware. The current implementation employs sRIO to communicate between components, which allows fast transmission of data between them. The component BSP only has to provide component-specific functions for carrying out basic sRIO operations (e.g., setting up inbound and outbound windows, triggering doorbells, etc.). The details of communications between publisher and subscribers are hidden from the users.

Initial Implementation and Ongoing Activities

The middleware library has been initially implemented and verified in a development system. In the current implementation, the connectivity between components is defined with a user-created XML (extensible markup language) file. An example XML file is shown in Figure 3. The file is read in by the system to set up topics and their publishers and subscribers. This feature allows the connectivity of a system to be rapidly redefined as required by the situation. A new XML file can be loaded into the system either manually or automatically (e.g., through a network), allowing a quick turn-around time for reconfigurability.

A common concern when using communications layer middleware is its overhead and associated latency. Experiments were performed to measure the data transfer rates and compare them against the native sRIO data rates. The results shown in Figure 4 document the overhead of the thin communications layer, which is practically negligible with larger packet sizes.

```

<Topic>
  <Name>Detection</Name>
  <ID>3</ID>
  <Sources>
    <Source>
      <SourceID>8</SourceID>
      <NumBufs>2</NumBufs>
    </Source>
  </Sources>
  <Destinations>
    <Destination>
      <DSTID>0</DSTID>
    </Destination>
  </Destinations>
</Topic>

```

Figure 3: Example XML configuration file

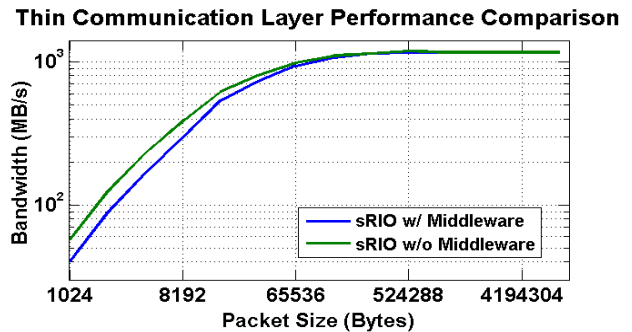


Figure 4: sRIO data rate with and without the thin communications layer

Figure 5 illustrates an approach that allows the bypassing of the thin communications layer. In extremely demanding cases, a real-time streaming device (e.g., an ADC followed by an FPGA) can bypass the communications layer and send data directly to its destination.

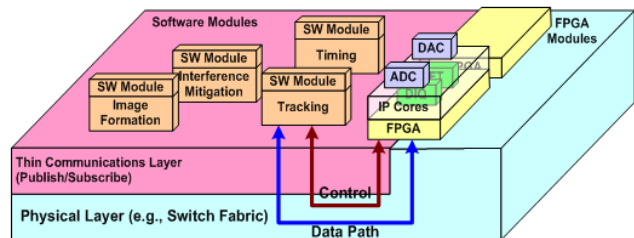


Figure 5: A streaming device can bypass the thin communication layer to minimize latencies

The development team is in the process of integrating the middleware library with an application developed in parallel. Other ongoing activities include the development of an exploration function to automatically detect and configure components on power-up (i.e., plug and play).

References

- [1] A. S. Rhoades, G. Schrader, and P. Poulin, "Benchmarking publish/subscribe middleware for radar applications," *Proc. Eleventh Annual HPEC Workshop*, 2007.
- [2] E. Gamma et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.