SpaceWire-HS Host Adapter – An FPGA based PCI Express Device for Versatile High-Speed Channels

Poster Session

S. Jörg, M. Nickl, T. Bahls, and S. Strasser
Robotics and Mechatronics Center
German Aerospace Center (DLR)
Oberpfaffenhofen, Germany
stefan.joerg@dlr.de

***Abstract*—Robotic systems like the DLR Hand Arm System that feature control cycles beyond 1 kHz demand a deterministic and low latency communication. Therefore, DLR is working on high-speed SpaceWire. This paper presents the SpaceWire-HS host adapter, a FPGA driven PCI Express device for high-speed SpaceWire. The adapter provides a generic host interface for QNX real-time hosts, supported by a client C++ library. Two implementation variants of the adapter's communication architecture and host interface are presented. The performance of both variants in terms of bandwidth and latency is discussed.**

***Index Terms*—host adapter, high-speed SpaceWire, robotics**

I. Introduction

DLR has been using SpaceWire as communication backbone for several of its lightweight robots. The latest and most complex system using SpaceWire is the DLR Hand Arm System, an anthropomorphic arm that comprises of 52 motors and over 430 sensors. To operate that many actuators and sensors precisely at high feedback control cycles beyond 1 kHz requires deterministic system behavior and low communication latencies. To achieve this, DLR implemented a 1 GBit/s SpaceWire modification (see [1]). Currently, DLR is working on a more efficient implementation of high-speed SpaceWire links capable of providing more than 1 GBit/s bandwidth [2].

The interface of the SpaceWire communication backbone to the PC-based real-time control hosts is a crucial point of the communication infrastructure. There the network implementation meets the non-determinism of a state-of-the-art workstation and its memory-based peripheral interface, PCI Express (PCIe).

To benefit from the performance of the high-speed links the authors have designed the SpaceWire-HS host adapter, a PCIe interface card that features four physical high-speed links (see Fig. 1). The SpaceWire-HS host adapter is equipped with a Xilinx Virtex-5 FPGA that implements the SpaceWire communication architecture. Thus, the communication architecture can be easily adapted to application requirements. Nevertheless, the SpaceWire-HS host adapter is designed as a general-purpose SpaceWire network endpoint.

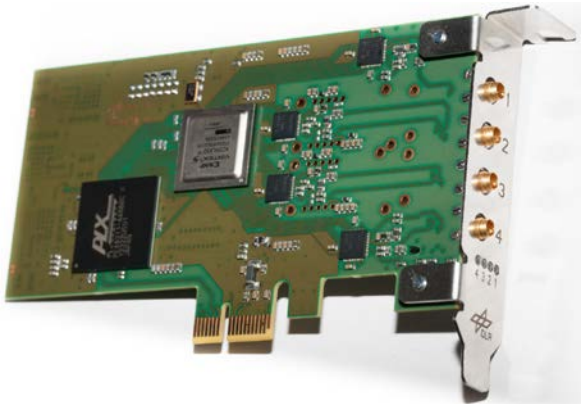

Fig. 1. The SpaceWire-HS Host Adapter Board with four copper HS-Links

Instead of increasing the performance by implementing application-specific algorithms and data structures on the host adapter FPGA, the focus is on the implementation of general-purpose packet channels whose performance in terms of bandwidth and latency can be configured by only a small set of parameters (e.g. buffer size).

This paper presents the communication architecture of the SpaceWire-HS adapter including two implementation variants. The next section introduces the hardware architecture of the SpaceWire-HS host adapter. Section III and IV present two variants of host interface implementation, which consists of the FPGA firmware and a C++ library. Section V presents the experimental evaluation of both implementation variants.

II. The SpaceWire-HS Host Adapter Architecture

The hardware design of the SpaceWire-HS host adapter is based on the design presented in [3]. Fig. 2 depicts the host adapter architecture. The PCI Express interface is implemented with a PLX PEX 8311 ExpressLane Bridge chip (see [5]), a one-lane master-capable host interface. A Xilinx Virtex-5 (5VLX50) FPGA connects PCIe interface, flash memory, housekeeping infrastructure and four physical layer interfaces. Four Texas Instruments TLK1221 IEEE802.3 Gigabit Ethernet compliant physical layer circuits implement the physical layer

interfaces. Character encoding and link layer are implemented by the firmware on the FPGA. Therefore, the support of the new SpaceWire-HS high-speed link protocol (see [2]) only required the adaption of the FPGA firmware.

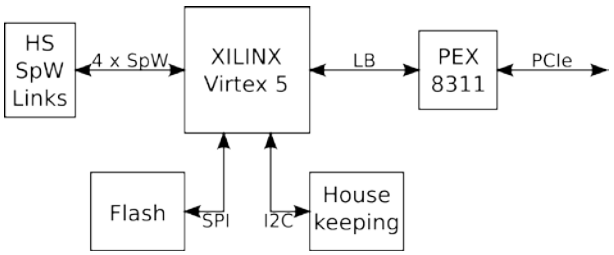

Fig. 2. The SpaceWire-HS Host Adapter Architecture

The Printed Circuit Board (PCB) format of 68.9x119.0mm conforms to the PCIe low profile form factor. Thus, it also fits into small-form factor cases. The board can alternatively be equipped with up to four fiber and/or copper links. Fig. 1 shows the adapter PCB with four copper HS-Links.

III. The Communication Architecture

The communication architecture has two main parts: The local SpaceWire Routing Switch and the Host Interface (see Fig. 3). The four physical SpaceWire-HS links are connected to the local Routing Switch, which is implemented as a standard SpaceWire wormhole routing switch. All HS-Links are configured for a fixed bandwidth and start automatically if connected to a peer.

The FPGA's configuration flash memory is connected via a SpaceWire/SPI bridge to the local routing switch. Thus, the FPGA's firmware can be programmed via the SpaceWire network.

For the implementation of clock synchronization, a configurable TimeCode (TC) master is connected to the local router. A Spacewire/I2C bridge provides access to the adapter's housekeeping infrastructure.

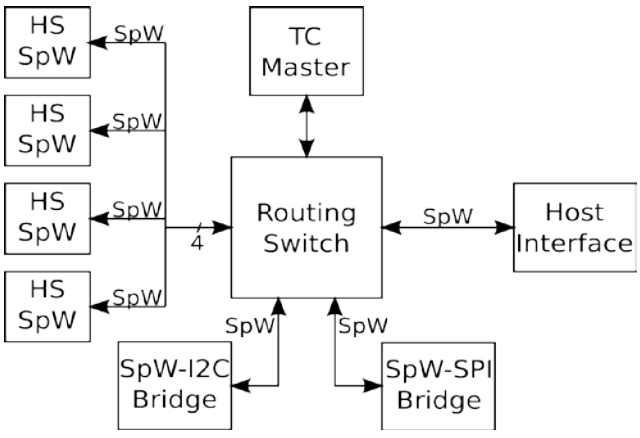

Fig. 3. The communication architecture consists of two main parts: A Routing Switch and the Host Interface. Two implementation variants of the Host Interface are discussed (see Fig. 4 and Fig. 5).

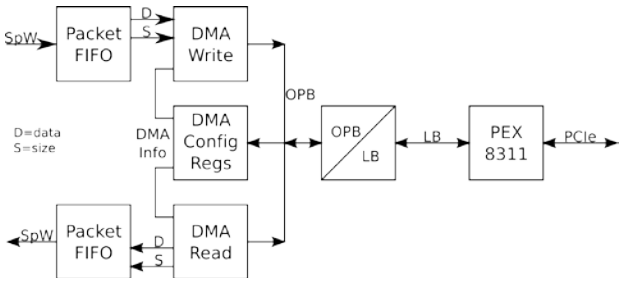

Fig. 4. Single-Channel Host Interface Architecture

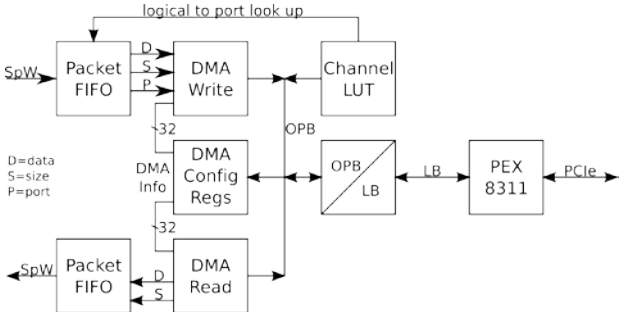

Fig. 5. Multi-Channel Host Interface Architecture

In the following, two implemented and evaluated variants of the Host Interface are presented. For both variants, the Host Interface consists of memory mapped status, configuration registers (DMA Config Regs in Figs. 4/5), and a number of DMA Read/Write interfaces. Those interfaces on the FPGA are connected via a CoreConnect On-Chip-Peripheral Bus (OPB). An OPB/PEX Local Bus bridge connects the FPGA's OPB to the PEX 8311, which implements the bridge to PCIe.

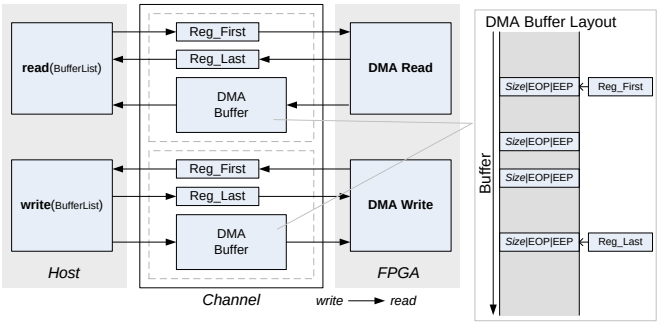

Fig. 6. A bi-directional ring buffer channel connects host software (read/write) to FPGA firmware (DMA Read/Write) via PCIe. The DMA Buffer layout is an in-place linked packet list.

The first variant is the Single-Channel Interface (Fig. 4). This implementation was already used in the SpaceWire-1Gb of the Hand Arm System [1] and is presented in more detail in [3]. The Single-Channel has one large ring buffer interface for each communication direction. Both the DMA Write and DMA read channel are a 1Mbyte fixed-sized ring buffer. The resulting implementation creates a bi-directional FIFO-channel from host software to FPGA firmware as depicted by Fig. 6.

The ring buffer synchronization is implemented as follows: Host Software and Firmware communicate via the shared

REG_FIRST, REG_LAST ring buffer registers (see Fig. 6). The ring buffer concept governs the concurrent access to these registers. The firmware is synchronized by simply polling those ring buffer registers. The host software is synchronized by interrupt. If enabled by the host software, the firmware raises a PCIe interrupt if the DMA read ring buffer is no longer empty or the DMA write ring buffer is no longer full.

Most applications require more than one endpoint at the host. With only a single channel, the host software needs to implement packet routing, which is an expensive operation. To avoid the packet routing on the host, the second variant provides 32 parallel endpoints to the host software.

Therefore, the Multi-Channel variant (Fig. 5) implements 32 configurable DMA read/write ring buffer interfaces. The functionality of each channel is the same as for the Single-Channel implementation. However, the host software configures the size of each ring buffer. Similar to a routing switch, the channel lookup table (Channel LUT) implements a routing table for the mapping of physical/logical ids to one of the 32 channels. The host software also configures the Channel LUT. Channel arbitration is implemented as a fair round-robin scheme.

IV. Client Packet Interfaces

The client packet interface implements the host software SpaceWire end-point interface. It consists of a POSIX I/O driver and a C++ software library. Both are implemented for the QNX 6.x real-time operating system. The I/O driver provides an open/close/read/write POSIX I/O interface for each of the host adapter's DMA channels. The driver uses the efficient message passing implementation of the QNX kernel to copy the data packets via read/write function calls. Thus, shared memory between client and driver is avoided.

As depicted by Fig. 6, packets are stored as an in-place linked list layout in the DMA ring buffer. Each packet in the ring buffer starts with a 4-byte header that contains a 16-bit packet size and packet tags, such as EOP, EEP.

The I/O driver read/write interface uses the same data structure to communicate with its clients. Thus, each call to read/write is able to transfer more than one packet. This is supported by the C++ client library class *over::pci::BufferList*, which implements the in-place linked list layout of the DMA ring buffer. Additionally, the C++ client library provides protocol-specific packet data structures, a routing switch implementation, end-point classes, a network topology configurator, and more convenient functionality. Therefore, an application does not need to use the I/O interface directly.

For example, the packet data structures allow an application to pre-allocate packets as required at an initialization phase. Then, during operation, only the payload of the packets in the pre-allocated data structure has to be updated. Fig. 7 exemplifies how the pre-allocation, update and send is implemented using the C++ library. Packet reception works similar. A blocking read on a DMA channel (done in link.receive()) yields all packets available in the DMA. Fig. 8 exemplifies how to iterate through all received packets and route them to their destination node (i.e. buffer).

```
// create packet buffer list
over::pci::BufferListInstance<> tx_pkts(10xspacewire::MAX_PACKET_SIZE);
…
network::Packet packet(25,2);  // 25 payload and 2 address bytes
packet.address()[0] = 42;
packet.address()[1] = 2;
tx_pkts.push_back( packet );  // add packet to send buffer
…
tx_pkts.push_back( packet2 );  // add another packet to send buffer
…
// update payload of every packet to be send
for ( over::pci::BufferList<>::iterator p= tx_pkts.begin(); p!= tx_pkts.end(); ++p )
      std::memcpy((*p).payload.data(),src.data(),(*p).payload.size());
link.send( tx_pkts );   // deliver all packets with one write() to driver
```

Fig. 7. Example of pre-allocated packets. Only the payload needs to be updated before sending all pre-allocated packets at once.

```
// create packet buffer list
over::pci::BufferListInstance<> rx_pkts(10xspacewire::MAX_PACKET_SIZE);
…
link.receive( rx_pkts );   // blocks until at least one packet arrives
for ( over::pci::BufferList<>::iterator p =  rx_pkts.begin(); p != rx_pkts.end(); ++p )
 {   // route all received packets
    network::Packet<BufferReference<> > recv(*p);
    Node& node = router_table->find(recv.address()[0]);
    node.push(recv);
 }
```

Fig. 8. More than one packet can be received with one read(). This example demonstrates, how to route every packet to its destination by simply iterating through the received packets.

A different I/O driver implementation is required for the Single-Channel and Multi-Channel variants since the I/O driver has to provide an open/close/read/write interface for each of the host adapters DMA channels. Since both variants use the same DMA channel layout, the same C++ library is used for the Single-Channel and Multi-Channel implementation variants.

V. Experimental Results

The performance in terms of bandwidth and latency of the two implementation variants Single-Channel and Multi-Channel has been experimentally evaluated. Therefore, the following experiments have been conducted for each variant:

1. **Roundtrip Latency:** Packets are looped via a HS link.
   measured: Roundtrip time for each packet
   parameters: packet size: 5-1017,
   parallel communication: 1-9 channels
2. **Receive Bandwidth:** External source sends packet to host. Packet load is increased to find the stable limit.
   measured: received packet bytes per second
   parameters: packet size: 9-1017,
   number of packets per second
   parallel communication: 1-4 channels

Both experiments were conducted with a DELL Optiplex Intel i7-3770 host running QNX 6.5 and a SpaceWire-HS card with four copper links. Fig. 9 depicts the setup for both experiments for the Single-Channel (top) and Multi-Channel (bottom) variant. The main difference in the test setup is the additional client library router required for the Single-Channel.

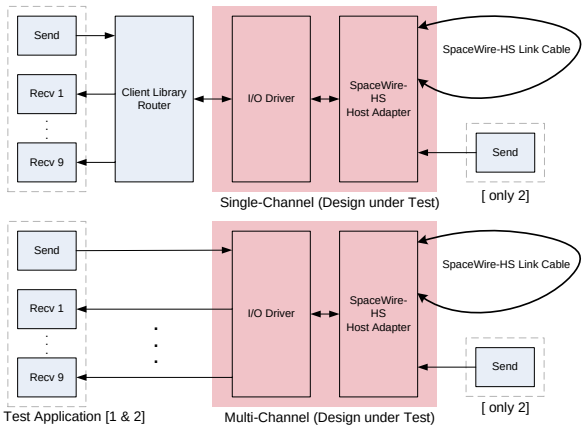

Fig. 9. The Experiment Setup for the Single-Channel (top) and Multi-Channel (bottom) variant

1) Roundtrip Latency

The test software consists of a packet source, a packet receiver for each channel, and a time measurement. The packet source sends a packet to 1-9 receivers, looped through a HS-link cable. The total time was measured it took to send all packets until the reception of all sent packets. Depending on the number of channels, 1-9 packets were send/received in one cycle. Each measurement was conducted for 30 seconds, which means 200.000-300.000 cycles.

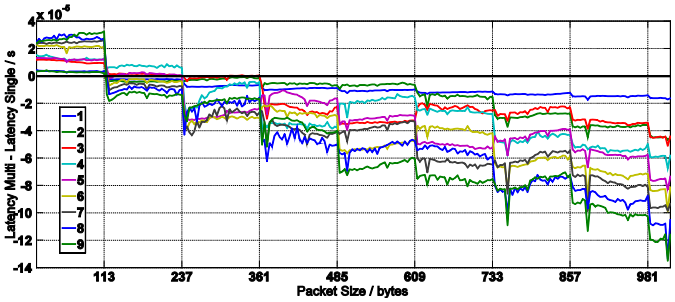

Fig. 10. Multi-Channel – Single Channel Roundtrip latency over Packet Size

Fig. 10 depicts the difference between the measured latencies $L_{Multi\text{-}Channel}$-$L_{Single\text{-}Channel}$ for 1-9 used channels over rising packet size. The steps that appear in the graph at 128 byte intervals are directly related to the PEX8311's 128 byte maximum payload size of a Transaction Layer Packet (TLP) (see [4]). For packets>237 bytes, the Multi-Channel is always faster, even if only one channel is used. This is because no software router is required. For packets of 113–237 bytes both implementations yield nearly the same latency. For packets<113 bytes the Single-Channel is slightly faster (~2us).

2) Receive Bandwidth

The second experiment uses an external packet source that sends packets of varying size at deterministic intervals to up to four destinations. For each packet size, the time interval between each packet was reduced until the stability limit of the host was reached. Fig. 11 depicts the measured maximum stable bandwidth over increasing packet size. The achieved bandwidth saturates at 57.7x10 bytes/s for the Single-Channel and at 61.0x10^6 bytes/s for the Multi-Channel. Both variants come close to the maximum input bandwidth, which is determined by the internal FPGA SpaceWire link rate of 62.5x10^6 bytes/s. Due to higher routing and DMA data handling effort small packets achieve only a lower bandwidth.

Since the Multi-Channel variant does not require packet routing by the host software, it saturates at a higher bandwidth and reaches that bandwidth for smaller packets.

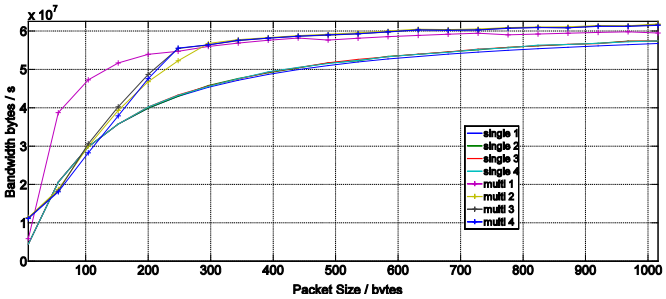

Fig. 11. Bandwith over packet size: Multi-Channel achieves higher bandwith

VI. Conclusions

The demand for a deterministic communication backbone with low latency motivated DLR's work on high-speed SpaceWire. For robotic applications, a deterministic host interface is an essential component of a high-speed SpaceWire network. Starting from the already available Single-Channel implementation, the goal has been to get a more flexible and efficient host interface, which features multiple endpoints. The challenge has been to achieve a deterministic and efficient behaviour despite the higher complexity of the Multi-Channel implementation. The results show, that not only the Multi-Channel implementation is deterministic but also matches the Single-Channel in performance.

Future work will be to add Quality-Of-Service parameters to the Multi-Channel implementation such as channel priority or guaranteed channel bandwidth. Furthermore, we intend to evaluate how the SpaceWire-HS host adapter with its four physical links can be used as a building block for very complex networks such as multi-robot systems.

References

[1] M. Nickl and S. Jörg, T. Bahls A. Nothhelfer, S. Strasser, "SpaceWire, A Backbone For Humanoid Robotic Systems", Int. SpaceWire Conference, San Antonio, 2011

[2] M. Nickl, S. Jörg, T. Bahls and B. Cook, "Towards High-Speed SpaceWire Links", Int. SpaceWire Conference, Gothenburg, 2013

[3] T. Bahls, "Entwicklung einer latenz- und bandbreitenoptimierten Bridge zur transparenten Anbindung von FPGAs an Standard-CPUs", Master Thesis (in German)

[4] R. Budruk, D. Anderson, T. Shanley, „PCI Express System Architecture", MindShare, Inc., Addison Wesley, 2004

[5] "ExpressLane PEX 8311 PCI Express-to Generic Local Bus Bridge Data Book", PLX Technology, www.plxtech.com, 2009