SpaceWire Device Driver for the Remote Terminal Controller

Albert Ferrer Florit
Wahida Gasti

ESA-ESTEC

Motivation

- The Remote Terminal Controller provides two SpW Interfaces

Their complexity requires a specific software library to simplify user operation and hide implementation details.

- BepiColombo team was interested in the RTC to be used as a Front-End solution for data handling system.

There was no software available to handle the SpaceWire interfaces in a simple way, with flexibility and high performance.

- RTC is one of the first ESA ASICs to implement a processor-based SoC with SpW interfaces.

Good opportunity to define a suitable SpW API for embedded systems. (data link layer specifications of the protocol stack)

RTC SpW interface

The Remote Terminal Controller provides two SpW Interfaces.

- They are highly configurable with multiple modes of operation and more than thirty configuration registers.
- The implementation is interrupt driven and performs DMA transfers for the reception and transmission of SpaceWire packets.
- Multiple packets can be stored in the same receive buffer but the information about their lengths is not preserved.
- Provides an extra virtual channel reserved for VCTP packets, and a hardware implementation of the RMAP protocol.
- Basic hardware support for TimeCodes and link errors reporting.

Software Design Considerations

SpaceWire Network characteristics

- Support variable packet length. ⇒ Do not restrict to memory available.
- Avoid network congestion. ⇒ Discard data when receive buffer is full.

Application constrains

- Limited resources (30Mhz Processor Speed) ⇒ Avoid non DMA data transfers.
  Provide early identification of packets.
- Support integration of higher level protocols. ⇒ Provide independent send request with different priorities.
- Extended functionalities ⇒ Link error reporting
  Time-code functionality
  Time stamp information

Specifications (1)

- Performance and efficiency:
  - Support for **sustained bidirectional high data rate transfers** (up to 159Mbit/s, for low demanding data processing applications)
  - User application can obtain the length of a packet and may read a complete packet without performing any memory copy.
- Memory requirements
  - Driver implementation has a small code and data footprint, and does not require any external library.
  - Receive buffers can have arbitrary sizes and can be dynamically adjusted by the user application. A receive buffer may contain multiple SpW packets.

Specifications (2)

- SpaceWire functionality
  - Three different packet transmission functions, including **multicast** packet function. Information about **time transmission** is provided upon completion.
  - **Multiple send requests** can be queued and an identifier is provided to supervise their status. They can have two levels of **priority** and be cancelled before being executed.
  - Information provided on packet reception: packet length, protocol ID, EOP marker, CRC and other errors.
  - **Multiple receive buffers** can be **queued** or added dynamically. Receive buffers are actually implemented as optimised receive FIFOs.
  - There is **no limitation in packet sizes**. Packets that are bigger than the size of receives buffers available can be read in multiple chunks of bytes.

Specifications (3)

- Capability for hardware packet rejection and **software packet filtering** on packet reception. Statistical information about packets rejected and filtered is provided.
- Configurable notification of SpaceWire events.
- It can be configured to **automatically discard** incoming **data** in case there is no more memory available for packet reception.
- It can configure the hardware to act as a **time-master** (sending Time Codes periodically) or time-receiver (retrieving the last received Time-Code).
- Provides complete **link configuration and error notification** and recovering. It also provides configuration functions for the RMAP and VCTP hardware support.

SpW driver Operation

Initialization

1) Initialize SpW drivers and set the receive packet mode.
2) Set packet filtering and error reporting options.
3) Provide a receive buffer to the Driver (it starts receiving packets)

Packet reception

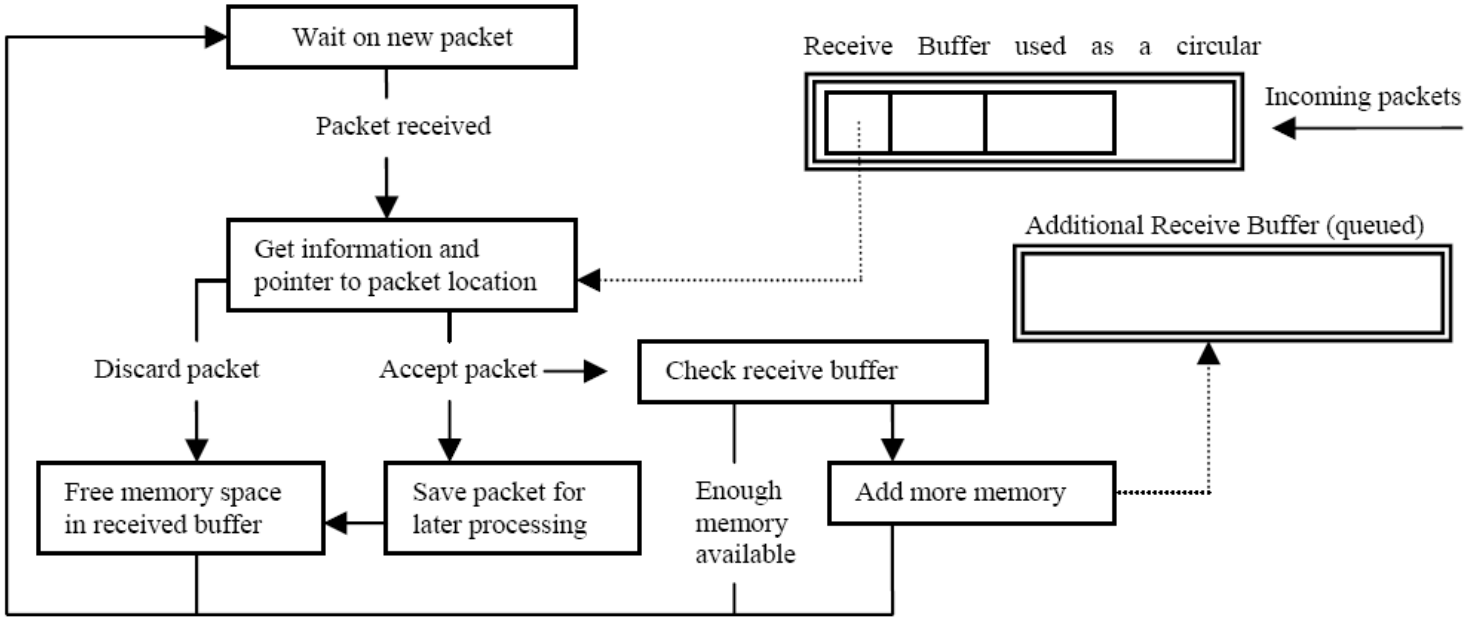
1) Request information about the next packet received. (including errors, packet length, time stamp, and pointer to its memory location)
2) Packet processing.
3) Free memory used by the packet or provide more memory.

Packet Transmission

1) Perform a send request (unicast or multicast)
2) Check status of request. (get time transmission)
3) Free request identifier.

SpW driver Operation (2)

Simplified program flow for packets reception:



Note that it is possible to keep current packet in the Driver receive buffer and only free memory used by old packets.

RTC development suite

Hardware:

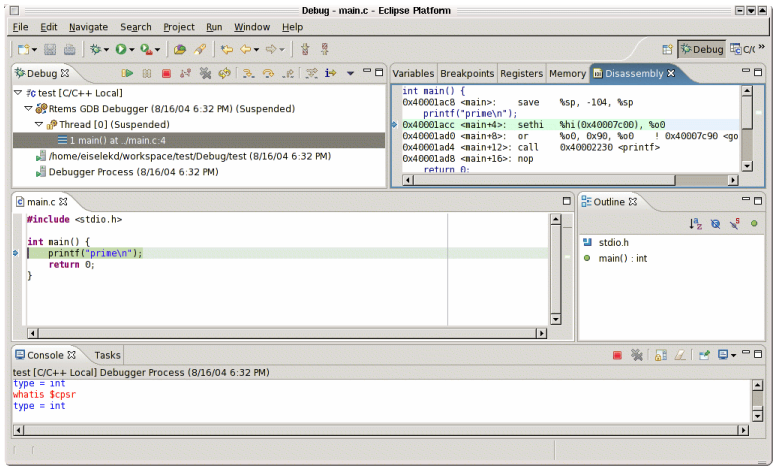
GR-4M-CAN2-SPW3 board

+ GR-CPCI-XC2V Development Board

+ SpaceWire-USB Brick

Software:

Eclipse+ Grmon+GNU compiler



Example application: Echo Server (1)

```
// Initialize driver
spw_Open(0);

 // Set receive mode: accept all non VCTP packets without report
 //options.
spw_SetReceptionMode(SPW_LINK_0, SPW_DISCARD_VCTP_PACKETS, 0  );

// Set link speed
spw_SetLinkSpeed(SPW_LINK_0, SPW_100MBits);

// Assign logical address of the node
spw_SetLogicalAddress(SPW_LINK_0, RTC_DLA);

// Try to start link 1
if (spw_StartLink(SPW_LINK_0) == SPW_LINK_DISCONNECTED)
{
        // Wait until link is running
        while (SPW_IS_LINK_RUNNING(SPW_LINK_0) == SPW_FALSE ) { };
}

// Setup Receive FiFo
spw_AddNewRxFifo(SPW_LINK_0, rxFifo, BYTESIZE_RXFIFO);
```

Example application: Echo Server (2)

```
    // Main loop
    for (;;) {
            spw_WaitOnNewPacket(SPW_LINK_0, 0);
            retCode = spw_GetNewPacket(SPW_LINK_0,&rxPacket);
            if (retCode == SPW_SUCCESS) {
                if (rxPacket.length <= BYTESIZE_RXBUF) {
                   // Discard packets too big.
                   if (rxPacket.length != rxPacket.bufSize) {
                       spw_SavePacket(SPW_LINK_0, rxBufferU32, BYTESIZE_RXBUF/4, 0);
                       pBuf = (BYTE *)rxBufferU32;
                   } else {
                      pBuf = rxPacket.pBuf;
                   }
                   // Send packet without the RTC logical address
                   spw_SendPacketTo(SPW_LINK_0,DEST_DLA, pBuf+1,rxPacket.length-1,0,&sendId);
                   spw_WaitOnSendCompleting(sendId, 0);
                   spw_FreeSend(sendId);
                }
            }
            // Free receive fifo memory
            spw_FreeFifoInUse(SPW_LINK_0,0);
    }
    return (0);
}
```

Conclusions

- The SpW device driver provides an API with a **full set of data link layer services** including time stamp information, priorities, packet filtering, link error reporting, and time-codes.
- It can handle **SpW packets of any size** with functionalities to **avoid network congestion**.
- It provides enough **flexibility** to support a wide range of applications using limited memory and processor power resources.
- **Application examples** provided to BepiColombo team has proved its capability to **support sustained high data transfers**.

SpW made even more simple and powerful !!!

Thank you for your attention!

SpW driver Operation (3)

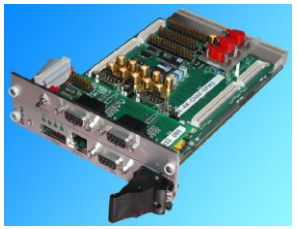
Receive thread

RTC Software development environment

LEON2 multi-platform framework, plus specific libraries to handle RTC interfaces (i.e. SpaceWire drivers)

Eclipse framework with:
- C developers Tool-Kit
- Gaisler Research plugin for LEON

→ **GRMON** (debug monitor) → UART →



CygWin (only on windows hosts)

LEON Bare-C Cross Compilation System

Includes:
-GNU C/C++ cross-compiler
-Newlib (Embedded C-library)
-Pthreads library

RTC interfaces libraries
SpW drivers:
- Continuous high data rate transfers.
- Link error recovering.
- Packet length and time stamp info
- Transparent interrupt handling.
- Multiple protocols supported
- Time-Codes.
- Easy-to-use operation
- Compact memory footprint