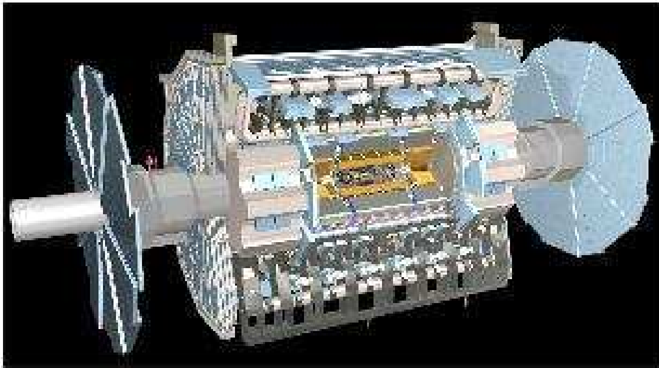
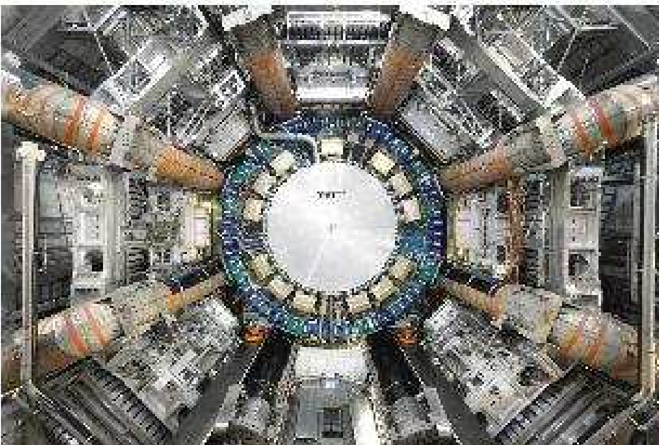




Demonstration of the RCE Interfaced to the Pixel FE

RCE Training Workshop

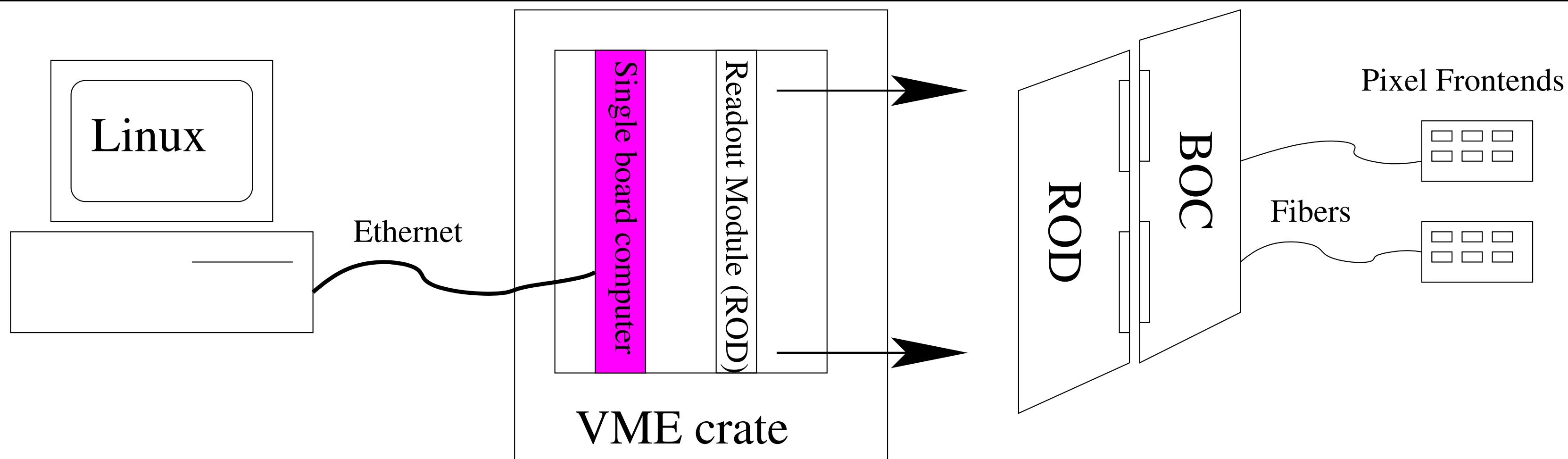
Martin Kocian, kocian@slac.stanford.edu

June 15, 2009

Introduction

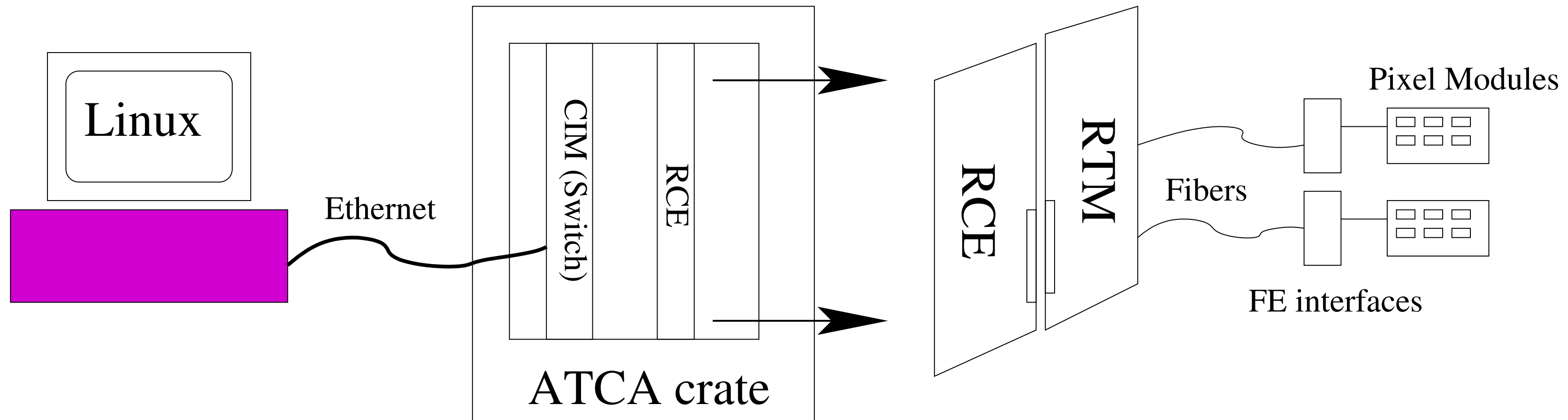
- The goal was to implement a pixel calibration on RCE system to explore the calibration aspect of an RCE based DAQ system for ATLAS.
- The calibration chosen is the "digital test".
  - Injects a "digital charge" into the digital circuitry of each pixel.
  - The test is done in 32 mask stages.
  - The output is an occupancy histogram with one bin per pixel.
- The following slides contain a detailed discussion of:
  - RCE system setup compared to present ROD calibration setup.
  - PGP protocol for FE communication.
  - FPGA side of FE communication.
  - RCE side of FE communication.
  - DSP code conversion.
  - Linux control executable.

ROD Based Calibration



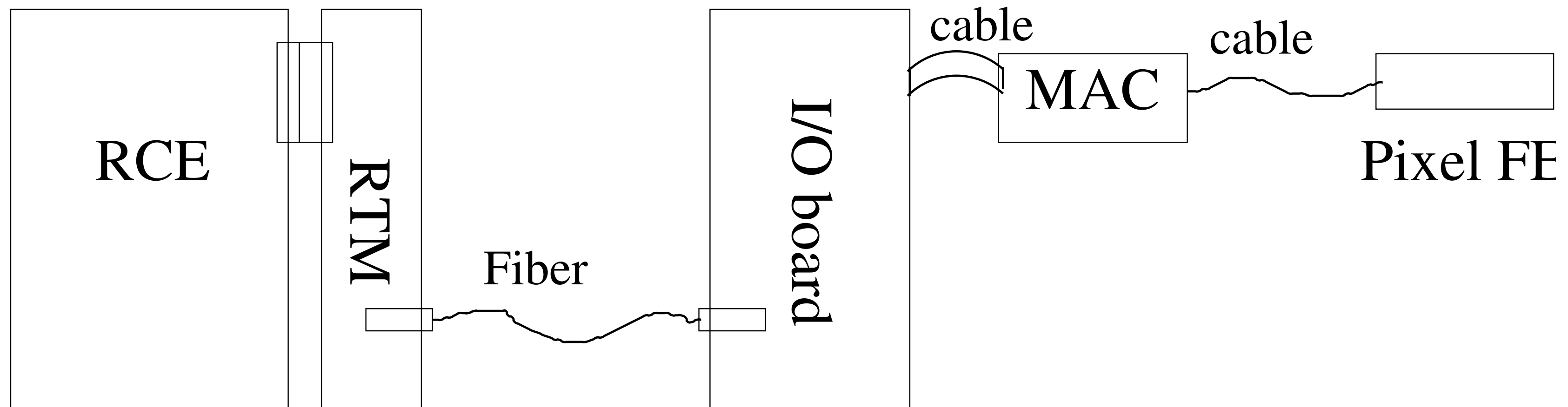
- The pixel system has a number of electronics calibrations that run on the RODs.
- Infrastructure (Databases etc.) run on Linux.
- Communication is done over an SBC that talks to the DSPs inside the ROD over VME using "primitives," data blocks that can easily be transferred over VME.
- DSPs run central part of calibration (e.g. mask stage loop) and do histogramming.
- The master DSP talks to the FEs through a back-of-crate card which is connected to the pixel frontends through optical fibers.
- Histograms are shipped out through the SBC to Linux.

RCE Based Calibration



- The SBC is gone.
- The VME crate is replaced by an ATCA crate.
- The RCEs replace the RODs and connect to Linux via Ethernet through 1 of 2 CIM switches inside the crate.
- A CIM is connected to Linux through a 10 Gbit/s fiber.
- Each RCE contains a powerpc processor that runs the calibration.
- The RCE connects to the frontends through optical fibers at multi Gbit/s rates.

Frontend Chain



- The RCE is connected to the fiber module on the RTM through the zone 3 ATCA connector.
- The RTM connects to a custom high-speed I/O fpga board (HSIO) through a pair of optical fibers at 3 Gbits/s.
- The I/O board communicates with one frontend module through a Module Adapter Card which contains LVDS drivers.

Frontend Interface Design

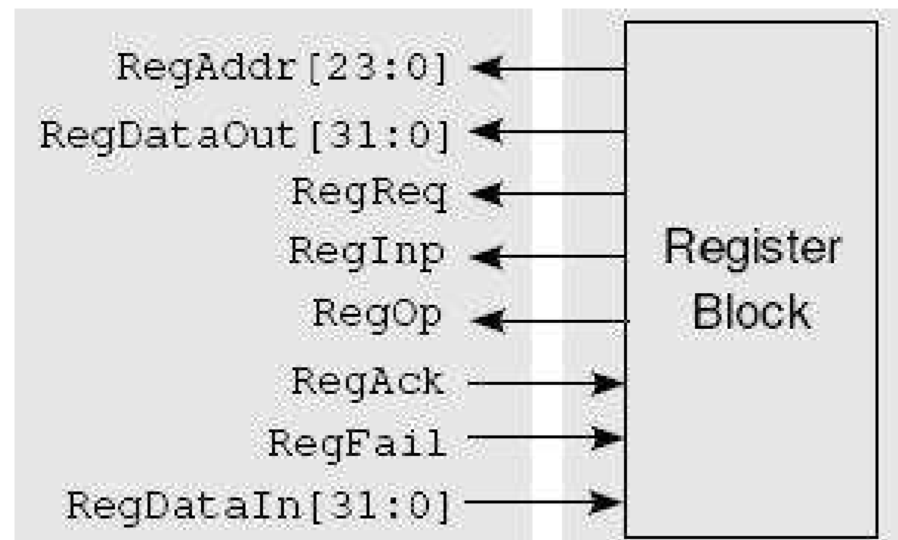
- The protocol that was used for communication between the RCE and the HSIO is PGP (pretty good protocol) which was developed at SLAC by Mike Huffer et al.
- Data rate of 3 Gbits/s.
- 4 virtual channels per lane (i.e. per pair of optical fibers).
- Multiple lanes available per RCE depending on configuration.
- RCDI (Register Command Data interface).
  - Convenient user interface.
  - Vhdl design available for frontend fpga implementation.
  - Software support in RCE.
- The PGP is one example of a protocol that can be used. Other protocols (e.g. GBT) can be used as well.

RCDI

- Through the RCDI interface PGP can be used to:
  - Read and write registers remotely.
  - Send opcodes.
  - Receive (and send) data blocks.

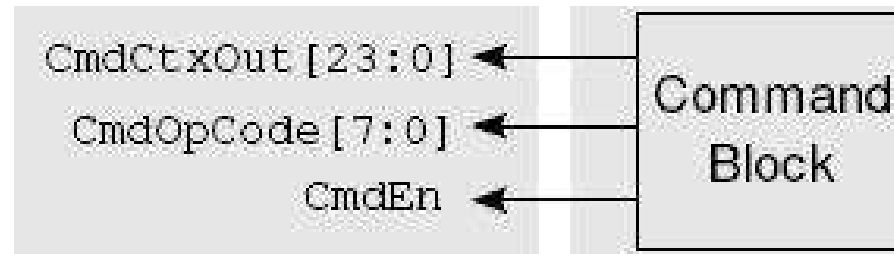
Register Block

- Write and read a 32-bit wide register to 24 bit wide address space.
- Returns a reply message to RCE.
- User can set fail bit for reply message.
- Timeout error is returned of the user logic does not acknowledge the request.
- Vhdl I/O port:



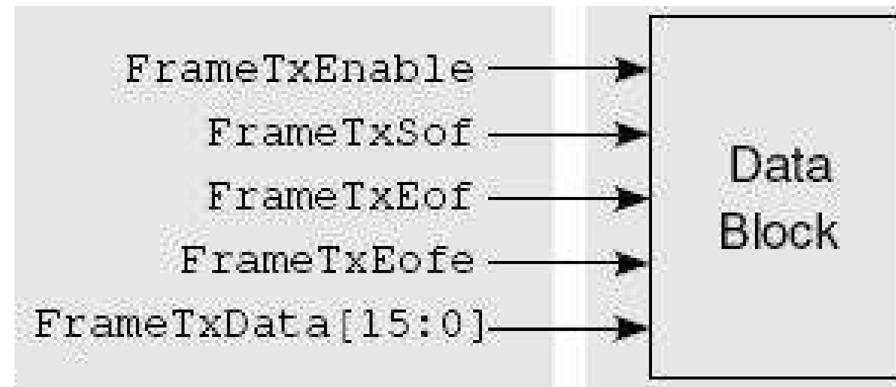
Command Block

- Sends an 8-bit opcode to the user logic.
- Sends a 24 bit context word.
- No reply to RCE.
- Can be used for fast commands and for triggering.
- Highest priority. Data block transmissions are paused to allow for instant command transmission.
- VHDL I/O port:



Data Block

- User logic can send a stream of 16-bit words to the RCE.
- First and last word have to be flagged as start and end of frame.
- User can set error flag.
- In the current design the maximum data length is 2 MB (but this can be configured to be a different size).
- VHDL I/O port:



RCDI with Pixel FE

- HSIO board contains a Xilinx FX60 FPGA.
- One MGT (Gbit/s transceiver) on HSIO is connected to a fiber optics module.
- The RCDI vhdl design is implemented on the FX60.
- Custom logic on FX60 interfaces with pixel module.
- The following slides discuss how the RCDI is used for communication with the pixel frontend.

Pixel Frontend Configuration

- Pixel FE is configured through serial bitstreams at 40 Mbits/s.
- The register block is used to write configuration data to a FIFO inside the HSIO board.
- The command block is used to trigger a serializer in the HSIO that will then serialize the data inside the FIFO and send it to the frontend at 40 Mbits/s.
- The HSIO sends a short data stream back to the RCE as a handshake to signal that the FIFO contents have been sent.

Calibration Pulse and Trigger

- The calibration command and the trigger command for the pixel frontend are bitstreams in the same way as the configuration.
- The same mechanism to send and serialize the CAL/Trigger command as for configuration is therefore used.
- Instead of the serialization command at the end a “serialize and wait for data” command is sent through the command block.

Data

- The event data consists of one or two serial bitstreams at 40 or 80 Mbits/s (depending on how the module is configured).
- This example calibration uses 1 bitstream at 40 Mbits/s.
- A deserializer in the HSIO creates 16-bit words from the bitstream without any formatting.
- The deserializer is activated by the "serialize and wait" command.
- It starts running as soon as the first "1" appears at the input.
- It stops running when it receives 32 "0"s in a row.
- The output of the deserializer is the input to the data block of the RCDI.
- When the deserializer is done it asserts "End-Of-Frame" for the RCDI.

PGP On the RCE Side

- The RCE firmware supports the PGP protocol.
- A C++ driver that interacts with the hardware is the software interface for raw PGP.
- Handler classes can be written as high level interfaces.
- The RCDImaster class is such a handler class that implements the RCDI interface in software.
- The following slides show how to use the PGP software on the RCE.

PGP Initialization

- To use PGP:
  - Define transmit buffer parameters.
  - Create a pool of transmit buffers.
  - Get the pgp driver for the desired lane from the system.
  - Register your handler with the driver.

```
const RcePic::Params Tx = {
    RcePic::NonContiguous,
    16, // Header
    64*132, // Payload
    128 // Number of buffers };
PgpTrans::RCDImaster *rcdi=new PgpTrans::RCDImaster; // RCDI interface
RcePic::Pool *pool = new RcePic::Pool::Pool(Tx);
rcdi->SetPool(pool); // tell the handler so it can put back the buffers
RcePgp::DriverList *driverList = RcePgp::DriverList::instance();
RcePgp::Driver *pgpd = driverList->handler(RcePgp::RTM_0, rcdi);
```

RCDImaster Class I

- RCDImaster class has the following accessor functions:
- Register read and write:

```
unsigned RCDImaster::writeRegister(unsigned address, unsigned value);
unsigned RCDImaster::readRegister(unsigned address, unsigned& value);
```

- The return value is the PGP status (0 if no error).
- Send command:

```
void RCDImaster::sendCommand(unsigned char opcode, unsigned
context=0);
```

RCDImaster Class II

- For the data block the user has 2 choices:

1. Callback function:
   - User provides a class that inherits from PgpTrans::Receiver
   - User class implements `void receive(RcePic::Tds* tds);`
   - User registers class with interface: `rcdi->setReceiver(&myReceiver);`
   - Receive function is called every time data arrives.
2. RCDImaster class stores data if no callback function is registered:
   - As data blocks are received they are stored in buffers.
   - Number of currently stored data buffers: `unsigned nBuffers();`
   - Retrieve current buffer (in the order received):
     `unsigned int currentBuffer(unsigned char*& header, unsigned& headerSize, unsigned char*& payload, unsigned& payloadSize);`
   - Return value is 0 if OK 1 if no data buffer available.
   - User has to invalidate buffer when it is no longer needed:
     `int discardCurrentBuffer();`
   - After that the next buffer (if there is one) becomes the current buffer.
   - If the buffers are not discarded RCDI will run out of buffers.

Ethernet

- Initialization:

```
RceInit::configure_network_from_dhcp();
```

- Set up a UDP server:

```
enum {MY_PORT = 1201};
_clientaddr=RceNet::IpAddress(0,MY_PORT);
_servaddr=RceNet::IpAddress(INADDR_ANY,MY_PORT);
_socket=new RceNet::SocketUdp;
_socket->bind(_servaddr);
```

- Send and receive messages:

```
sendMsg(&buffer,bufferLength);
ret=_socket->recvfrom(&buffer,bufferLength,_clientaddr);
```

Porting DSP Code to RCE

- DSP code:
  - 1 master DSP and 4 slave DSPs per ROD.
  - Master code is a single endless loop.
  - Controlled from SBC by writing to memory in VME.
  - Hardware access (e.g. serial interface or FPGA logic configuration) throughout the code.
  - Written in C.
- Ported to RCE as:
  - 1 powerpc processor runs master and slave code.
  - Several tasks in parallel one of which is the main loop.
  - Controlled via ethernet from Linux. On the RCE side there is a separate task that converts the ethernet packets into memory blocks.
  - Serial interface uses pgp through RCDImaster class.
  - Formatting of the data is done in software in this example (instead of FPGA).
  - Code is C with a few C++ classes (ethernet and pgp).

Linux Code

- Control executable runs on Linux host.
  - Reads configuration and sends it to RCE.
  - Starts calibration.
  - Reads histograms from the RCE and displays them in root.
- Uses PixLib library in a standalone version without databases.
- VME access functions replaced by functions that use ethernet.
- Configuration read from TurboDAQ file.

Experience

- It is straightforward to compile C/C++ code on RTEMS.
- The usual c/c++ libraries including STL are available.
- The system is very stable.
- Debugging is convenient through gdb.
- The only complication was endianness: Powerpc is big-endian, the DSPs are little-endian.

Summary

- Pixel digital test was ported successfully to run on RCE.
- Communication with the Pixel FE is done through PGP protocol.
- On the FE side there is a PGP interface to the user logic consisting of a serializer/deserializer.
- On the RCE side a C++ user interface for PGP exists.
- DSP code was ported without any major changes.
- Demonstration to follow...