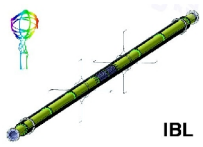


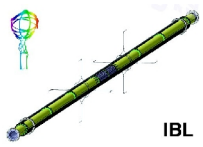
Design document

IBL ROD Communication
Overview

Prepared by
A. Kugel

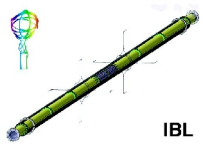
August 2012
Version 0.3

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

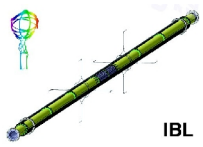


Version management

No	Date	Changes	Author
0.1		Initial version	A. Kugel
0.2		Updates on communication mechanics	A. Kugel
0.3	20120824	Updates with slave emu and hist test	A. Kugel



Contents



1. Introduction

The new IBL ROD employs hardware and mechanisms quite different from the traditional Pixel ROD, whilst maintaining a largely compatible API towards the higher level software.

A set of four FPGAs implements all required functionality:

- Xilinx Spartan-6 LX45: low-level board control (PRM, program reset manager)
- Xilinx Virtex-5 FX70T: master device with on-chip processor (PPC) and main control register set. An off-chip pixel-style DSP processor is available as well.
- 2 * Xilinx Spartan-6 LX150: slave devices for histogramming and I/O

The PPC processor executes the master software, which provides the traditional API, however considering the specific properties of IBL (e.g. front-end ganularity, command set, etc.). Only a small fraction of the code is processor specific (DSP vs PPC). Both off-chip and on-chip processor access the master control registers and the inter-chip communication (ROD-bus, setup-bus) in the same fashion and either can be selected to operate the ROD. However, the DSP can be used through the VME interface only, while the PPC provides an alternative network interface.

This document describes the communication between master and slave processors on the ROD and also provides some guidelines and examples on how to access the ROD from the higher level "RodCrate" software.

2. Master-Slave communication

Communication between master and slave operates via a small dual-ported memory (DPR, 4kB size) inside the slave FPGA, accessible to the master via the ROD-bus.

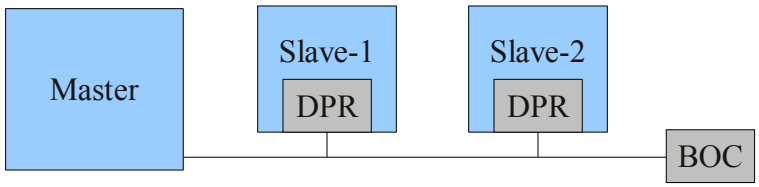


Figure 1: Master-Slave Interconnect

The Pixel-sytle "xface" data structure is not used any more. The effective size of the DPR is currently restricted to 2kB by the address range definition in the master FPGA. After power-up, the slaves run a small boot-loader program residing in the internal memory. The real slave code must be downloaded by the master. Further operation is done by exchanging commands and data via the DPR. The download procedure used the same mechanism.

A special control register can be used to reset the slaves, which re-enables the boot-loader program.

2.1. Command mechanism

The first location of the DPR is used to indicate the command to be executed. A value of 0 indicates no action (idle). Subsequent locations are used to transfer data corresponding to the specific command, defined by certain data structures. In case the slave provides data to the master, the data words are written to a command dependent location and an

acknowledge words is written to the command location. Once the master has processed the data it clears the command word.

The following slave-emulation code-snippet demonstrates the command processing using the default command structure IblSlvRdWr. (pCtl is a pointer to the control register).

```
typedef struct {
      UINT32 cmd;         // command code
      UINT32 addr;
      UINT32 count;
      UINT32 data;        // first data word. more may follow
} IblSlvRdWr;


    IblSlvRdWr *cmd = (IblSlvRdWr*)dprStartAddress;

    while (1){
        // check reset
        if (*pCtl & (1 << SLV_CTL_RESET_BIT)){
            printf("Slave %d reset\n ",slvId);
            booted = 0;
            loaded = 0;
            verbose = 1;
            continue;
        }
        // command loop
        switch (cmd->cmd){
        case SLV_CMD_IDLE:
        case SLV_CMD_BUSY:
        case SLV_CMD_ACK:
        case SLV_CMD_ALIVE:
            continue;

        case SLV_CMD_BOOT:
             // set active status
             booted = 1;
             cmd->cmd = SLV_CMD_ALIVE;
             break;
        case SLV_CMD_STAT:
            stat = (iblSlvStat*)&cmd->data;
            stat->progType = loaded? SLV_TYPE_PROG : SLV_TYPE_LOADER;
            stat->progVer = 1;
            stat->statType = SLV_STAT_TYPE_STD;
            stat->status = SLV_STAT_OK;
            cmd->count = sizeof(iblSlvStat)/sizeof(long);
            cmd->cmd = SLV_CMD_ACK;
            break;
        case SLV_CMD_WRITE:
            addr = (unsigned long*)progBuf + cmd->addr/sizeof(long);
            count = cmd->count;
            data = (unsigned long*)&cmd->data;
            for (i=0;i<count;i++){
                addr[i] = data[i];
            }
            cmd->cmd = SLV_CMD_IDLE;
            break;
        case SLV_CMD_READ:
            addr = (unsigned long*)progBuf + cmd->addr/sizeof(long);
            count = cmd->count;
            data = (unsigned long*)&cmd->data;
            for (i=0;i<count;i++){
                data[i] = addr[i];
            }
            cmd->cmd = SLV_CMD_ACK;
```

```
            break;
        case SLV_CMD_CRC:
             crc32((char *)&progBuf[cmd->addr], \
                    cmd->data * sizeof(UINT32), &crcVal);
            data = (unsigned long*)&cmd->data;
            data[0] = ENDIAN_FLAG;
            data[1] = crcVal;
            cmd->count = 2;
            cmd->cmd = SLV_CMD_ACK;
             break;
        case SLV_CMD_START:
            addr = (unsigned long*)progBuf + cmd->addr/sizeof(long);
            startAddr = (void (*)())addr; // this is the JUMP address
            loaded = 1;
            cmd->cmd = SLV_CMD_ALIVE;
            // (*startAddr)(); //this would start the code on a real slave
            break;
        default:
            printf("Slv %d: Invalid command 0x%x\r\n",slvId,cmd->cmd);
            cmd->cmd = SLV_CMD_IDLE;
            break;
        }
    }
```

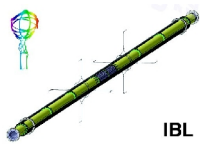
The default read and write commands target the memory area at the slave which keeps the program executable and are used to load and possibly verify the program code. Different commands can be implemented to access different memory areas, e.g. histogram data. The provided address is always a byte offset into the specific area. Access to arbitrary memory locations is not foreseen.

2.2. Boot process

The boot procedure involves 5 stages, which are triggered by certain operations of the master.

- Reset. A slave is started by asserting/deasserting the reset line, accessible from the DSP/PPC via the slave control register. The reset starts the boot loader program, resident in the FPGA internal memory. The boot loader waits for the command SLV_CMD_BOOT to appear at the command location of the shared memory. Subsequently, the slave responds with SLV_CMD_ALIVE to the command location. Once the master resets the command location to SLV_CMD_IDLE the initial handshake is complete. Command processing continues.
- Load binary. The binary program file for the slave is transferred to the boot loader program. This is done in portions, fitting into the small DPR, e.g. in 1kB block.
- Check CRC of loaded binary. The following “start” step is only executed when the local and remote (slave) CRC32 values match.
- Start binary. Performs the actual branch to the loaded binary. does not wait for program to start
- Boot verify. Waits until the new slave program is operational by checking the “program-type” field of the status response.

The boot-loader responds to commands in the same style the real code does. The master can verify the type of the slave code via the progType field of the status structure, returned in response to a status command (see section 2.3.1.).



2.3. Commands

Slave commands are defined in a file (common/DspInterface/PIX/iblSlaveCmds.h) used by the ROD and RodCrate software. At present, the following commands are defined:

```
// command definitions
#define SLV_CMD_BOOT        0xdeaddead // boot marker from master
#define SLV_CMD_ALIVE       0x00c0ffee // alive marker from slave
#define SLV_CMD_IDLE        0x000      // if command completed
#define SLV_CMD_BUSY        0x001      // working on command
#define SLV_CMD_ACK         0x002      // command finished and data available.
#define SLV_CMD_VERBOSE     0x010      // following word: rdwr structure
#define SLV_CMD_WRITE       0x011      // following data: rdwr structure
#define SLV_CMD_READ        0x012      // following data: rdwr structure
#define SLV_CMD_CRC         0x013      // following data: rdwr structure
#define SLV_CMD_START       0x014      // following data: rdwr structure
#define SLV_CMD_ID_SET      0x015      // following data: rdwr structure
#define SLV_CMD_ID_GET      0x016      // following data: rdwr structure
#define SLV_CMD_HIST_CFG_SET   0x020   // following data: histo cfg structure
#define SLV_CMD_HIST_CFG_GET   0x021   // returns data: histo cfg structure
#define SLV_CMD_HIST_CMPLT  0x022      // following data: hist term structure
#define SLV_CMD_HIST_TEST   0x023      // following data: rdwr structure
#define SLV_CMD_HIST_READ   0x024      // following data: rdwr structure
#define SLV_CMD_BOC_TEST    0x025      // following data: rdwr structure
#define SLV_CMD_NET_CFG_SET 0x030      // following data: net cfg structure
#define SLV_CMD_NET_CFG_GET 0x031      // return data: net cfg structure
#define SLV_CMD_CTL         0x099      // following data: rdwr structure
#define SLV_CMD_STAT        0x09a      // following data: rdwr structure
```

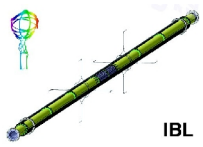
Commands with “GET” and “SET” flavours normally use the same data structure for both directions.

2.3.1. Status

A default status command is implemented for boot-loader and regular program code. It allows to request different status types, e.g. a “standard” status, network status and histogrammer status.

```
typedef struct {
    UINT32 progType;        // program type: loader, slave, ???
    UINT32 progVer;         // program version, > 0
    UINT32 status;          // actual status word, or size of status words
following. Depends on statType
} iblSlvStat;
// program types
#define SLV_TYPE_LOADER     0x10ad  // boot loader
#define SLV_TYPE_PROG       0xface  // real slave program
// status types
#define SLV_STAT_TYPE_STD   0x01    // default status
#define SLV_STAT_TYPE_NET   0x02    // network status
#define SLV_STAT_TYPE_HIST  0x03    // hisotgrammer status
// status codes
#define SLV_STAT_OK         SLV_CMD_IDLE     // no errors, idle
#define SLV_STAT_BUSY       SLV_CMD_BUSY     // no errors, busy
#define SLV_STAT_DATA       SLV_CMD_ACK     // no errors, data available
#define SLV_STAT_ERROR      0xffffffff  // something wrong
```

The progType field identifies the type of program. ProgVer is a user defined version number. StatType identifies the status type and allows to access additional information, e.g. error statistics etc., if implemented. The default status uses corresponding values to the command field for OK, BUSY and DATA.



2.3.2. Basic slave network configuration

The SLV_CMD_NET_CFG_SET command is used to set-up the basic network parameters of the slave and to start network operation. Without this command, only local read-back of histogram data is possible. The configuration is transferred via the structure

```
typedef struct {
      UINT32 cmd;          // command code
      char localMac[8];    // 2 0-chars, followed by 6 byte MAC address
      char localIp[4];     // e.g. 192.168.1.100
      char maskIp[4];      // e.g. 255.255.255.0
      char gwIp[4];        // e.g. 192.168.1.1
} IblSlvNetCfg;
```

The configuration can be retrieved via the command SLV_CMD_NET_CFG_GET.

2.3.3. Utility commands

A couple of utility commands are available to help in testing and debugging:

- Set/get ID: each of the slaves can be assigned a unique id, which may help when debugging with a terminal.
- Verbose mode: terminal output can be turned on or off (default). The rodMaster must map the appropriate serial port to the main terminal output (see 2.4.2.).

2.3.4. Histogram control

See section 3.

2.4. Controller level command access

Access to the master and slave commands is done via the traditional “primitivs”. All of the pixel-style master primitives are still there, however some are no longer in use and some have been modified. The primitive mechanism has not been changed, however.

2.4.1. Slave commands

Access to the slave commands is done via two of traditional slave primitives: SLAVE_CTRL and SEND_SLAVE_PRIMITIVE.

Loading and starting of the slaves is identical to the pixel-style:

```
// provide the name to the slave binary file
SlaveFile = "bootTest.bin";
// call load function of the RodController class
rod0->loadAndStartSlaves(SlaveFile,0x01);
```

The individual slave commands are accessible via the primitive “ SendSlavePrimitiveIn”. A sample sequence of writing 4 words and reading 2 words from the program memory area is shown below.

```
  // send a slave primitive
  SendSlavePrimitiveIn *slvCmdPrim;
  IblSlvRdWr *slvCmd;
  RodPrimitive *sendSlvCmdPrim;
  // UINT32* slvPrimBuf = (UINT32*)malloc(sizeof(SendSlavePrimitiveIn) +
sizeof(IblSlvRdWr));
  UINT32* slvPrimBuf = (UINT32*)malloc(2048); // large enough for DPR
  // we need an outlist too
```

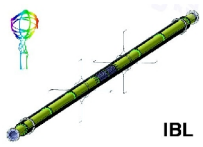
```
RodOutList* outList;
unsigned long *outBody;

cout << "Sending slave command write via SEND_SLAVE_PRIMITIVE " << endl;
// NB: nWords needs to be adapter to actual data size
slvCmdPrim = (SendSlavePrimitiveIn*)slvPrimBuf;
slvCmdPrim->slave = 0; // this is a number, not a mask
slvCmdPrim->primitiveId = 0;   // ignored
slvCmdPrim->fetchReply = 0;
slvCmd = (IblSlvRdWr*)&slvCmdPrim[1];
slvCmd->cmd = SLV_CMD_WRITE;
slvCmd->addr = 0;
slvCmd->count = 4;
slvCmdPrim->nWords = SIZEOF(IblSlvRdWr) + slvCmd->count - 1;
((UINT32*)&slvCmd->data)[0] = 0x5aa5c66c;
((UINT32*)&slvCmd->data)[1] = 0x12345678;
((UINT32*)&slvCmd->data)[2] = 0xdeadface;
((UINT32*)&slvCmd->data)[3] = 0x0101a0a0;
sendSlvCmdPrim = new RodPrimitive(SIZEOF(SendSlavePrimitiveIn) +
    SIZEOF(IblSlvRdWr) + slvCmd->count – 1, 1, SEND_SLAVE_PRIMITIVE,
    0, (long int*)slvCmdPrim);
rod0->executeMasterPrimitiveSync(*sendSlvCmdPrim);
delete sendSlvCmdPrim;

cout << "Sending slave command read via SEND_SLAVE_PRIMITIVE " << endl;
slvCmdPrim = (SendSlavePrimitiveIn*)slvPrimBuf;
slvCmdPrim->slave = 0; // this is a number, not a mask
slvCmdPrim->primitiveId = 0;   // ignored
slvCmdPrim->nWords = SIZEOF(IblSlvRdWr);
slvCmdPrim->fetchReply = 1;
slvCmd = (IblSlvRdWr*)&slvCmdPrim[1];
slvCmd->cmd = SLV_CMD_READ;
slvCmd->addr = 2*sizeof(UINT32);
slvCmd->count = 2;
slvCmd->data = 0;
sendSlvCmdPrim = new RodPrimitive(SIZEOF(SendSlavePrimitiveIn) + \
    SIZEOF(IblSlvRdWr),1, SEND_SLAVE_PRIMITIVE, 0, (long int*)slvCmdPrim);
rod0->executeMasterPrimitiveSync(*sendSlvCmdPrim, outList);
outBody = (unsigned long *) outList->getMsgBody(1);
UINT32 location = 0;
for(int i = 0; i < outList->getMsgLength(1); ++i) {
  cout << "Message data " << i << ": 0x" << hex << outBody[i] << endl;
}
rod0->deleteOutList();
delete sendSlvCmdPrim;

free(slvPrimBuf);
```

The second parameter to "RodPrimitive" is now void, as the command is included in the payload slave command structure and just kept for backward compatibility.

2.4.2. Master control commands

So far, only one new function has been added in order to control the ROD terminal output. The "EXPERT" primitive with the new function "UART_SRC" allows to set the source of the output as follows:

- 0 = rodMaster
- 1 = slave 0
- 2 = slave 1



3. Histogram Control Mechanism

Histograms are part of a higher-level “scan” procedure. A typical scan requires to iterate over a number of settings (“bins”, e.g. charge values) for each of which a number of data are collected (e.g. by sending triggers to the front-end and reading the returning data). A scan may be further divided into sub-scans, each operating on a subset of pixels (e.g. by “mask stepping” of the front-end). This allows to overlapp post-processing (e.g. “fitting”) of the results from one sub-scan with the data-acquisition (e.g. “histogramming”) for a subsequent sub-scan.

The histograms are collected by the Spartan-6 slave devices, each of which serves 2 groups of 8 front-end devices. The accumulated data have to be transferred to a remote fit-server via Ethernet after each charge bin. The size of the histogram is defined by the number of concurrently active front-end chips and the selected subset of active pixels per front-end. The initial implementation is limited (per group) to 1/8 of the full number of pixels, which can be arranged as one entire chip or as 8 chips each with 1/8 of active pixels. The latter is implemented by selecting a specific quarter of double-columns (DC 1-10, 11-20, 21-30, 31-40) and a mask-stepping of 2 (even/odd row number).

Figure 2 shows the intended sequence of operations:

- The controller selects the desired scan and instructs the fitServer and the rodMaster with appropriate configuration messages, which for example include the ROD network configuration.
- The rodMaster configures the network of the slaves
- Now, the rodMaster should know which sequence of sub-scans and hsitogram-loops have to be performed.
- The rodMaster issues for each of the inner histogram loops a histogram configuration command (see 3.1.) to the slaves, selecting the active front-end(s) and pixels. Also, it sends a scan configuration to the front-ends setting the register values (e.g. active pixels and charge).
- The actual data-acquisition is done by sending a number of triggers to the front-ends.
- Once the histogram is accumulated the rodMaster sends a histogram termination command (see 3.2.) to the slaves, which includes the network parameters of the fitServer (might be NULL). The slaves copy the histogram data from the acquisition memory to a processors accessible area, allowing to process a new histogram in parallel to the readout of the previous one.
- If the network is activated, the slaves send the histogram configuration and the data to the fitServer.
- The fitServer should know from the global configuration when and how to process the data. It returns the fit results to the controller at appropriate times.

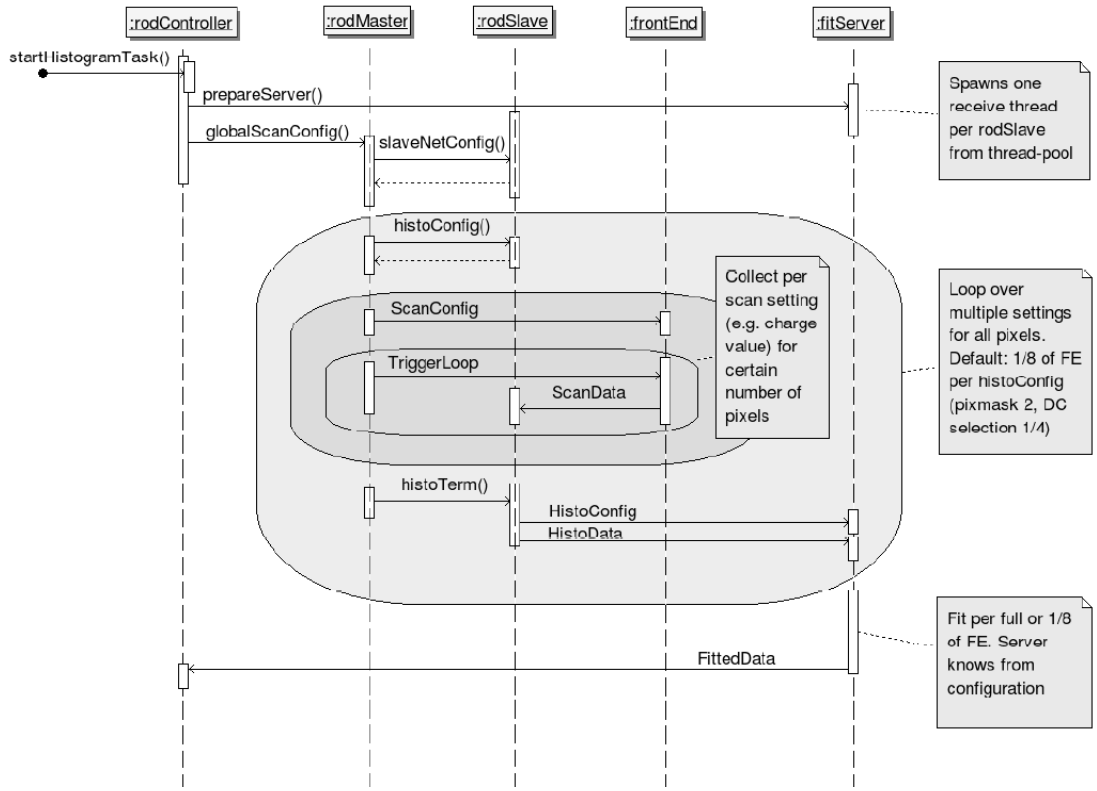
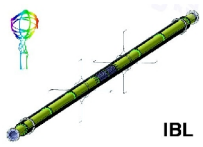


Figure 2: Histogram control

The low-level commands sent from rodMaster to slave are exposed to the rodController and can be used directly from the higher level directly. This helps in testing and to develop new scan-mechanics.

3.1. HistoConfig

HistoConfig (SLV_CMD_HIST_CFG_SET) sets the parameters for the histogramming engine, for example single chip or partial chip operation, DC quarter selection etc. All parameters are forwarded to the fit-server. The histogram configuration set/get commands carry the important parameters (see Table chapter 3.1) for the two histogram units of each slave individually.



Variable	Type	Comment
Enable	UINT32	1 = histogram unit enabled
HistoType	UINT32	Type of histogram, e.g. occupancy, ToT, etc 0: occupancy 1: ToT
ColStep	UINT32	Histogram parameter: DC selection 0: all columns (single chip) 1: first quarter 2: second quarter 3: third quarter 4: fourth quarter
MaskStep	UINT32	Histogram parameter: row selection 0: all rows (single chip) 1: divide row# by 2 (odd/even case)
ChipSel	UINT32	Number of active chips 0: single chip. (force chip = 0) 1: all chips
AddrRange	UINT32	Address range (=size) of histogram 0: single chip range (< 32k) 1: full range (< 1M)
ScanId	UINT32	Identifies the current scan (remote use only)
BinId	UINT32	Identifies the current bin. (remote use only)
Ntrigs	UINT32	Number of triggers. (remote use only)

The actual configuration can be read from the slaves with the command SLV_CMD_HIST_CFG_GET.

3.2. HistoComplete

HistoComplete (SLV_CMD_HIST_CMPLT) instructs the slave to copy the collected data from the internal histogram buffer to the processor memory and to subsequently send the data to the fit-server. The network parameters of the fit-server are provided.

Variable	Type	Comment
TargetIP	IP-Address, char[4]	IP Address of fit-server
TargetPort	UINT32	IP port number
Protocol	UINT32	1 = TCP, 0 = UDP

If the target IP number is 0 or the network is not enabled, the histogram data just remain in the slave memory and may be retrieved (see 3.3.) by the rodMaster.

3.3. Histogram readback

The acquired histogram can be read in chunks by the master processor via the histogram read commands (SLV_CMD_HIST_READ) using the data structure “IblSlvRdWr”. Typcal chunk size is 1kB. The maximum address can be computed according to the configuration

setting “addrRange”. The address corresponds to the address of the individual pixels in the histogram, starting with pixel number 0 at row=1, col=1, chip=0. Column/row sizes of the front-end are 80 and 336 respectively. The front-end generates col/row indices starting at 1, such that the address is computed by: pixNum = (col – 1) + (row – 1)*80 + chip * 336*80.

Occupancy/ToT histogram results data are indivudual 32-bits words formatted as follows:

Bits	Meaning
6 .. 0	Occupancy value
17 .. 8	Sum of ToT value
31 .. 18	Sum of ToT² value

Note: the available number of bits limits the number of triggers for a ToT scan to 63. For a plain occupancy scan up to 127 triggers can be issued.

3.4. Histogramming test data

Test data can be inserted into the histogramming engines with the command SLV_CMD_HIST_TEST. The histogramming unit is selected via the address field (0 or 1). The data format is identical to the format generated by the FPGA module “event-fragment-builder” - one 32 bit word per hit with the following composition:

Bits	Meaning
8 .. 0	Row number, 1..336
15 .. 9	Column number, 1..80
18 .. 16	Chip number, 0 .. 7
22 .. 19	ToT value, 0 .. 15

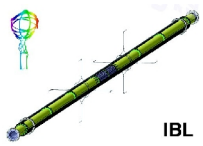
4. Development and targets

The code for the rodMaster natively targets two different architectures – a DSP and a PowerPC processor (PPC), both of which are available on the standard IBL ROD and which may be used alternatively. Using the DSP requires a VME infrastructure, while the PPC can be used via a network interface instead of VME, allowing for simple table-top test installations. Both processors share a very large portion of their source code, residing in a common repository. In addition, there is an emulator available - meaning the master code can be compiled and run on a PC and be controlled in the same way the network based PPC implementation is. In order to allow a maximum of code testing and debugging the emulator features two threads implementing the basic slave-style command processing, even including generation of histograms using the test data upload command.

The IBL software repository[1] contains all mentioned code variations plus test programs for the communication mechanism. The main makefile (see Figure 3) allows generation of all targets:

- make: (no parameters) builds the DSP binary, provided the TI tools are installed properly
- make gcc: compiles the DSP sources with the systems gcc, to help understand compile errors

[1] Currently: svn.cern.ch/reps/atlaspixeldaq/branches/IBLDAQ/IBLDAQ-0-0-0



- make ppc: builds the PPC binary (iblDsp_ppc.elf) into the PPC subdirectory, provided the Xilinx tools are installed and available in the PATH. For the moment, the binary must be loaded with the Xilinx debugger (xmd) using the following instructions (assuming xmd is started in the PPC directory)
    - [kugel@pcakulap IblDsp]$ xmd
    - Xilinx Microprocessor Debugger (XMD) Engine
    - Xilinx EDK 14.1 Build EDK_P.15xf
    - Copyright (c) 1995-2009 Xilinx, Inc. All rights reserved.
    - XMD% conn ppc hw -debugdevice devicenr 8
    - XMD% dow iblDsp_ppc.elf
    - XMD% run
- make ppcemu: builds the emulator binary (iblDsp_ppcemu) into the PPC subdirectory. The emulator can alternatively be build and debugged with the codeblocks IDE (http://www.codeblocks.org/), the project file is available in the PPC subdirectory.

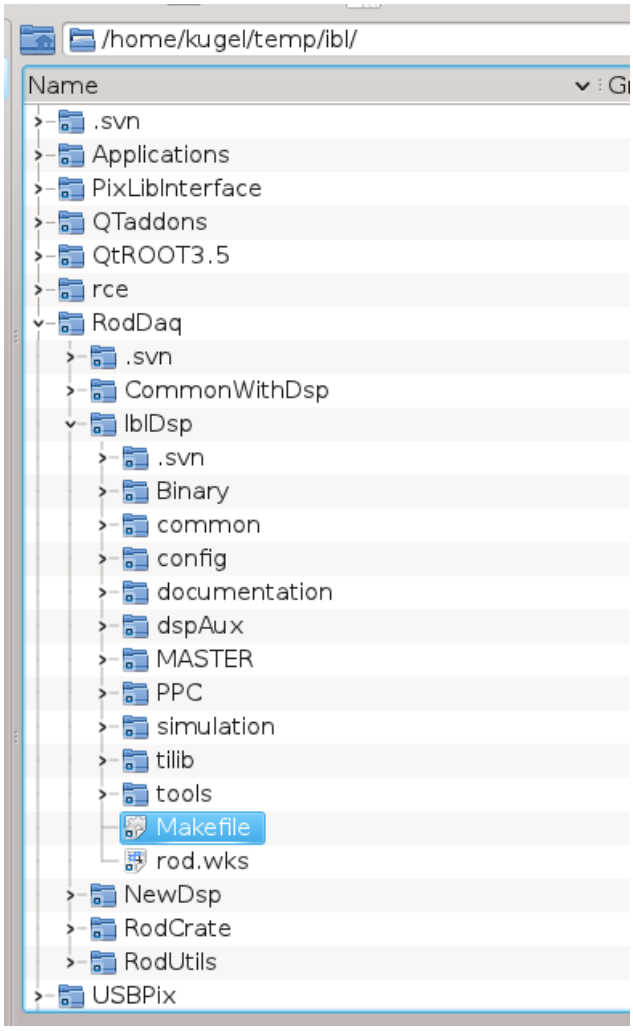
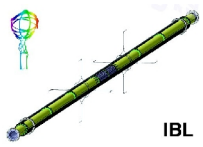


Figure 3: IBL repo structure

4.1. Hardware dependent files

Hardware dependent files are located in the PPC subdirectory, next to the PPC specific sources. All files are copies of the original files in the hardware repository and need to be kept up-to-date by the VHDL developers.

- rodSlave.hxx and rodMaster.hxx are C-headers generated from the corresponding VHDL sources. They identify address mappings, bit positions etc.
- rodMaster.bit: the FPGA binary for the Virtex-5 (master) FPGA
- sp6fmt.bit: the FPGA binary for the two Spartan-6 (slave) FPGAs
- rodMaster.elf: a stand-alone executable for the PPC which tests the slave communication at a low level. It can be downloaded with the Xilinx debugger (xmd) if desired.



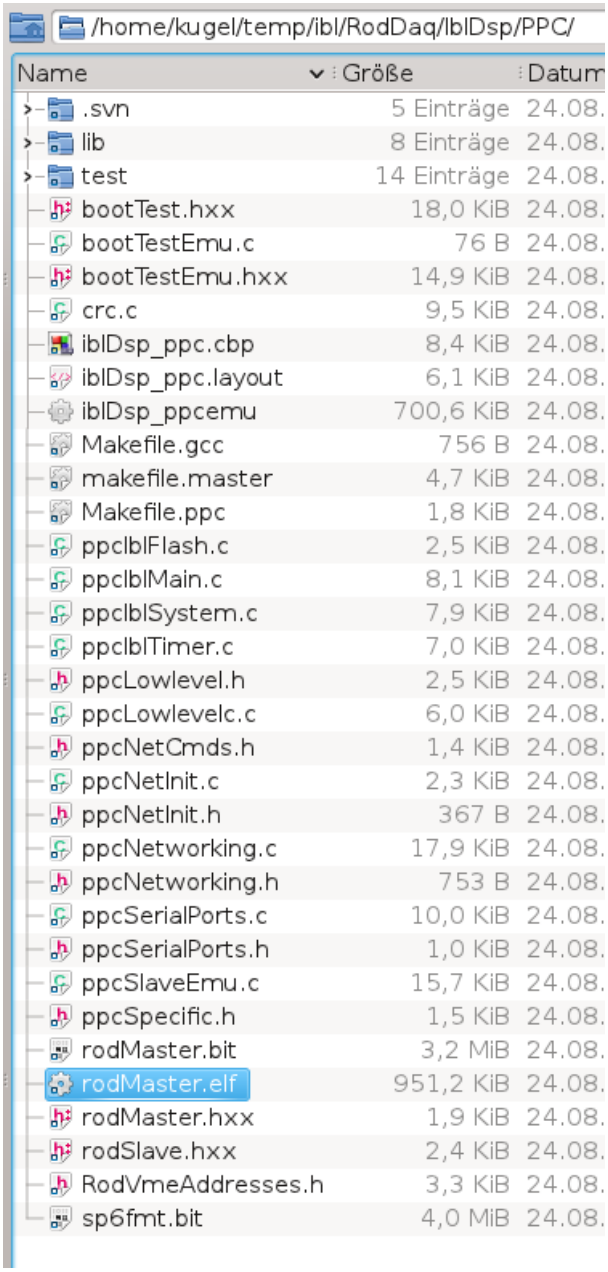


Figure 4: Hardware related files

4.2. Test programs

The folder PPC/test contains (at present) two test programs: rodCtl and histTest, which can be built either via the makefile or via the corresponding codeblocks projects. The makefile needs to be supplied with parameters to build for network and IBL (make NETWORK=1 IBL=1). The other options (VME, without network) and Pixel (non IBL) are not working at the moment.

With the network option enabled the programs request a "slot" number upon startup. Slot number 1 connects to the ROD emulator[2] on "localhost". The other slot numbers connect to

[2] Emulation is not visible to the test programs. The emulator is just an ordinary ROD.

real IP addresses, e.g. number 3 connects to the default IP address of the Xilinx rodMaster at 192.168.1.10.

RodCtl provides the raw basics of the primitive handling, histTest is the more recent tools with a couple a convenience functions and a simple histogram test.

4.3. Using the ROD Emulator

Emulation can be used to debug DSP/PPC code on the ROD as well as the communication mechanics. Starting the ROD emulator produces the following output

```
[kugel@pcakulap IblDsp]$ PPC/iblDsp_ppcemu
Starting ROD master software. Compile date Aug 24 2012, time 17:43:55
EPC emulation area allocated at 0xf5764008, size 4194304 bytes
EPC base from offset 0xf5764008
EPC size 0x400000 bytes
Local registers from offset 0xf5768408
Local ctl registers from offset 0xf676400c
Local stat registers from offset 0xf6764010
Local spmask registers from offset 0xf6764014
Status register: 0x0
Clt register (0x5aa5c66c): 0x5aa5c66c
Clt register (0): 0x0
Acquiring DSP register area
Controller version (at 0xf5768414): 0x0
Formatter0 version (at 0xf5764090): 0x0
Formatter1 version (at 0xf5765090): 0x0
Formatter0 ram (at 0xf576480c): 0x5aa56cc1
Formatter1 ram (at 0xf576580c): 0xc66ca551
Formatter0 ram (at 0xf5764810): 0x5aa56cc2
Formatter1 ram (at 0xf5765810): 0xc66ca552
Formatter0 ram (at 0xf5764814): 0x5aa56cc3
Formatter1 ram (at 0xf5765814): 0xc66ca553
Formatter0 ram (at 0xf5764818): 0x5aa56cc4
Formatter1 ram (at 0xf5765818): 0xc66ca554
Formatter0 ram (at 0xf576481c): 0x5aa56cc5
Formatter1 ram (at 0xf576581c): 0xc66ca555
Formatter0 ram (at 0xf5764820): 0x5aa56cc6
Formatter1 ram (at 0xf5765820): 0xc66ca556
Formatter0 ram (at 0xf5764824): 0x5aa56cc7
Formatter1 ram (at 0xf5765824): 0xc66ca557
Formatter0 ram (at 0xf5764828): 0x5aa56cc8
Formatter1 ram (at 0xf5765828): 0xc66ca558
Formatter0 ram (at 0xf576482c): 0x5aa56cc9
Formatter1 ram (at 0xf576582c): 0xc66ca559
EPC thread A created
Starting SLAVE A
Starting thread epcThread, slvId 0

Entering slave loop
EPC thread B created
Starting SLAVE B
Starting thread epcThread, slvId 1

Entering slave loop
Creating socket for port 5001
```

The emulator now waits for commands from a controller of the network (port 5001), e.g. by one of the testprograms. Starting the testprogram in another terminal windows results in the following:

```
Connected on socket 4
rxThread running, receiving from socket 4
```

```
Starting rx thread succeeded
Updating status register[0] at: 0x8f137c0
Regsiter now 0x1f
Register read value: 0x1f
Updating status register[2] at: 0x8f137c8
Regsiter now 0x3e
Register read value: 0x3e
Updating master register at: 0x8f13640
Regsiter now 0xc0ffee
Register read value: 0xc0ffee
Register read value: 0x0
Register read value: 0x8f136c0
Verbose set to 1
Set uart to slave 0
Slave 0 reset
 Slv 0 booting: indicate run mode

Slave 0 alive
slave init complete
Real branch will not work on emulator. Jump skipped
slave 0 boot ok
Verbose set to 0
Verbose set to 1
Set uart to slave 1
Slave 1 reset
 Slv 1 booting: indicate run mode

Slave 1 alive
```

This shows the reset of the rodMaster, followed by the reset of slave0 and the subsequent download of the binary.

The corresponding output of the testprogram (histTest) is as follows:

```
[kugel@pcakulap test]$ ./histTest_net_ibl
Enter slot number (decimal):1
Creating socket
Connecting to 127.0.0.1, port 5001
creating of BOC supressed
LED Start init
RodModule::initialize
S1 RodModule::reset
S1 RodModule::reset MAGIC_LOCATION 80000000
S1 RodModule::reset RESET
reset status: 0x1f
S1 reading MEMORYMAP_REG: 80000004 start 8f136c0
Total number of text buffers: 10
S1 RodModule::initialized finished
Using slave0
Status: program type     10ad
Status: program version  1
Status: status value     0
CWD is /home/kugel/temp/ibl/RodDaq/IblDsp/PPC/test


histClient.bin crc:
b61f404f
Local crc32: 0xb61f404f
Loading
file ..........................................................................
.....................................................
File loaded
Slave CRC: b61f404f
CRC check OK
```

```
Starting slave
Waiting for slave boot
SLAVE 0 LOADED!
Status: program type      face
Status: program version   1
Status: status value      0
Using slave1
Status: program type      10ad
Status: program version   1
```

Slave0 is reset followed by a status request. The program-type is “10ad”, meaning boot-loader. The slave binary is transferred (the many dots), which takes some time. The CRC check succeeds and is followed by a branch to the loaded binary (not executed on the emulator). The status is requested again and shows a program-type “face”, which means application. Slave0 is now ready.

Depending on the use case either the rod code or the test program is run by the debugger (e.g. codeblocks). Figure 5 shows debugging of the test program with a breakpoint just in before sending the master control primitive.

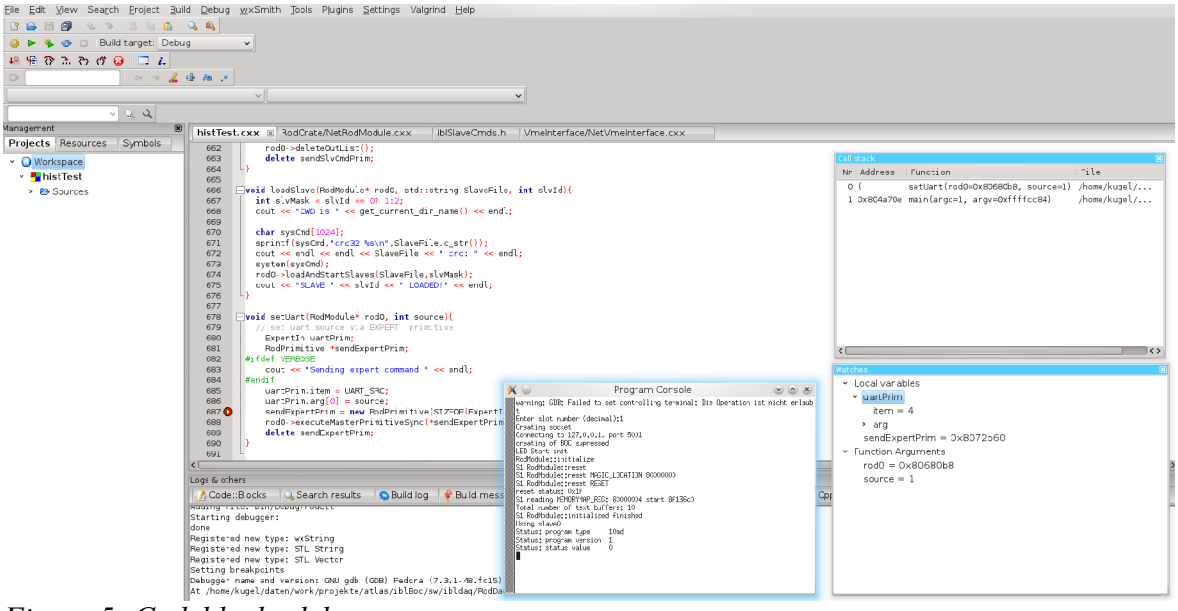


Figure 5: Codeblocks debugger

In order to boot the slaves the testprograms need a slave binary file, e.g. the histogram client “histClient.bin”, located in the program directory. This binary has to be created from the Xilinx software project via the following commands:

- mb-objcopy -O binary -R .vectors.reset -R .vectors.sw_exception -R .vectors.interrupt -R .vectors.hw_exception *program.elf program.bin*