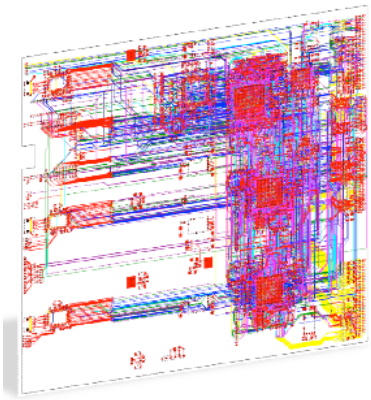
Design document

IBL ROD BOC Manual

Developer Version



B. Chen[1], D. Falchieri[2], A. Kugel[3], J. Mayer[1]
1) University of Washington EE/Physics, Seattle
2) University of Bologna and INFN, Bologna
3) ZITI, University of Heidelberg

August 2014
Version 1.2.2

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Version management

No	Date	Changes	Author
0.1	20120829	Initial version	A. Kugel
0.3	20120917	Development and register bits	A. Kugel
0.4	20130606	Initial merge with ROD control+datapath	A. Kugel
0.5	20140305	Adding specifications for all the blocks	D. Falchieri
0.6	20140318	Adding specifications for the histogrammer	M. Kretz
0.7	20140402	Updating specifications for the EFB	B. Chen
0.8	20140416	Master-PRM communication and XVC JTAG	A. Kugel
0.9	20140502	Adding busy blocks specifications	M. Backaus
0.91	20140520	Output format condensed mode flag now “c”	A. Kugel
1.0	20140702	Updated slave output data format	D. Falchieri
1.1	20140723	Adding the HPI specification paragraph	D. Falchieri
1.2	20140725	Update on the formatter paragraph	D. Falchieri
1.2.2	20140828	Further update on the formatter paragraph	D. Falchieri
1.2.3	20150122	Updated slave error reporting data format and registers.	Joe Mayer

Contents

Index

1. Introduction

An additional inner layer for the existing ATLAS Pixel detector, named insertable B-layer (IBL), is under design and will be installed by LHC-PHASE1. The IBL will consist of 14 staves equipped with 16 double chip modules for which a new front-end readout ASICs (FE-I4) is been developed to accommodate the higher hit occupancy and a smaller pixel area.

The staves will be organized in half staves to form readout groups, which are the modularity of connection to the outside world. The data transmission connection for control and physics data will be done via an optical link. This optical link will consist of optoboards as on-detector optical interfaces, fibres, and off-detector optical components located on the back of crate card (BOC).

The off-detector components of the readout system, as they are the readout driver (ROD and the back of crate card (BOC) are to be redesigned. The functionality and its realization in terms of logical blocks are to be described in this document.

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

The main functional blocks of the combined ROD/BOC system are shown in Error: Reference source not found.

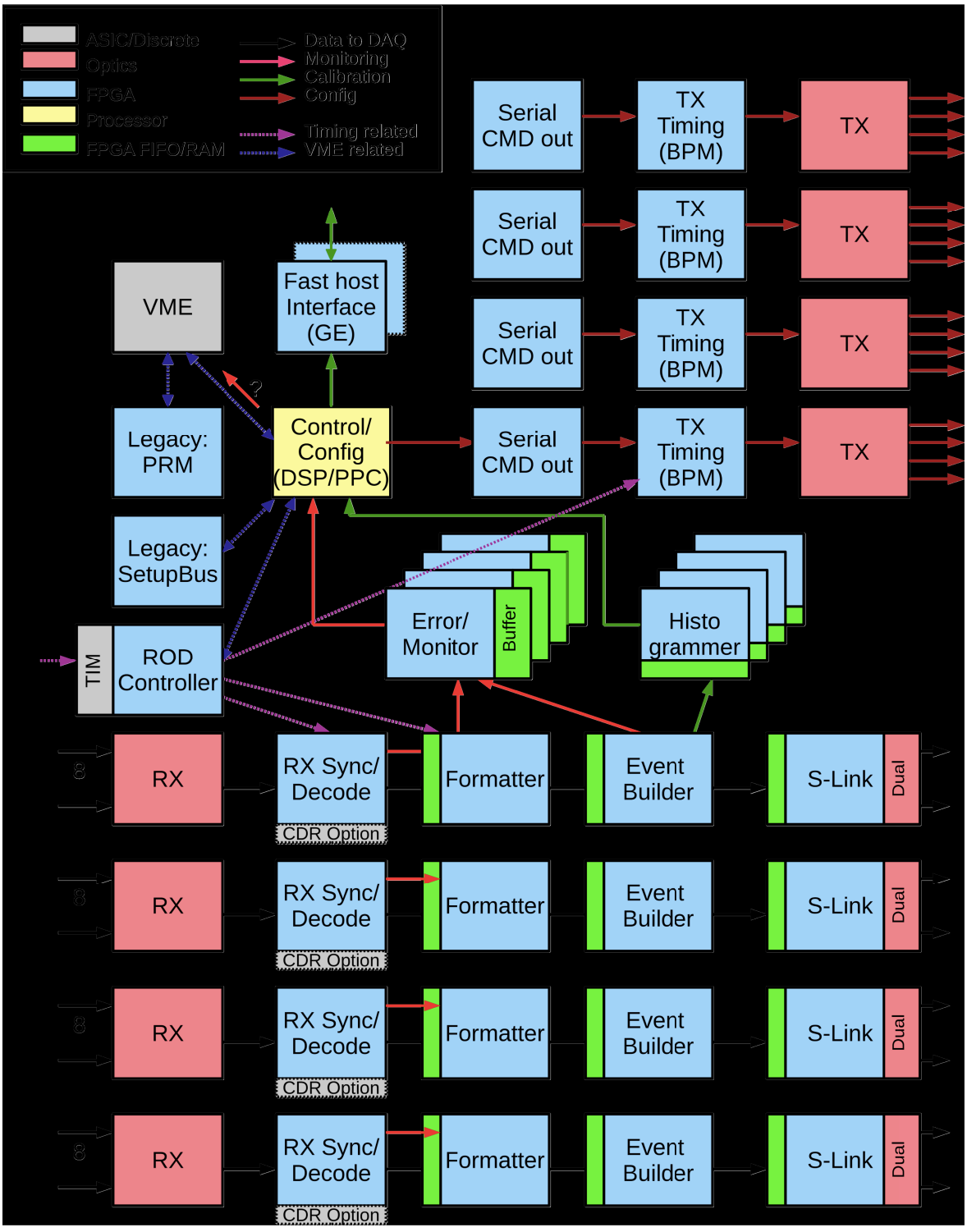
On the ROD a set of four FPGAs implements all required functionality:

- 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.

On the BOC there are another 3 Spartan-6 FPGAs two of which comprise the data-path logic and the third one the local control, steered by the ROD.

This document describes the basic card features related to code development, configuration and debugging as well as the various communication mechanisms between the two cards and between the elements of the cards, in particular the communication between master and slave processors on the ROD. The document also provides some guidelines and examples on how to access the ROD from the higher level “RodCrate” software.



igure 1: Main Functional Blocks

1.1. Firmware structure

Both cards include reconfigurable logic devices (FPGAs) which must be initialized after power-on, as the FPGA on-chip memory is volatile. The ROD includes a digital signal processor (DSP) which needs a program binary. In addition, all FPGAs may include

general purpose processor cores (CPUs), either in the form of a dedicated on-chip device (Virtex-5 FPGA on ROD) or created via the FPGA configuration (all Spartan-6 devices).

All of the elements mentioned above (FPGA, DSP, CPU) require binary firmware images to be downloaded into (FPGA bitstreams) or made available to (CPU/DSP program binaries) in order to function properly. Some of the firmware files will develop over time and will need to be upgraded in-situ, with minimal disruption to regular operation.

Finally, there will be some configuration parameters individual[1] to each card, for example a hardware serial number or the Ethernet MAC address etc. Both the structure and the content of this parameter set must be available for in-situ upgrades.

1.1.1. FPGA Bitstreams

All FPGAs used on ROD/BOC provide for different configuration mechanisms. The most basic one is to access a dedicated JTAG configuration port via an external JTAG programmer which is attached through a 4 wire electrical interface[2]. The JTAG port also serves as a versatile test and debug interface, which makes JTAG the primary access mechanism during the development phase.

Secondly, an FPGA can initialize itself from an attached FLASH memory. With an appropriate selection of devices the FPGA can also modify the content of it's configuration memory. Certain FPGAs – for example the Spartan-6 devices – provide for a so-called "multi-boot" feature. Multi-boot allows to start with a known-good bitstream after power-on. An alternative bitstream may subsequently be loaded from a different address range of the configuration ROM, any time under control of the initial firmware.

Finally, an FPGA can be attached as a peripheral to another "intelligent" device (CPU, PLD, FPGA) and be configured under external control.

1.1.2. Processor Binaries

Processor binaries store the program code for the processing devices on ROD and BOC. Typically this code is stored in a FLASH memory and copied into the processor main memory by some boot-loading hardware (e.g. an SPI memory controller). In case of processors embedded into FPGAs the boot-code may be part of the FPGA configuration bitsream. For complex applications (e.g. embedded Linux) the boot code most likely is embedded in the FPGA bitstream and will subsequently load the final application from external storage (e.g. separate FLASH chip, network, SD-card, etc.)

1.1.3. Parameters

Non-volatile card specific configuration is used to identify hardware items for administrative purposes (error tracking, ATLAS hardware database, etc) and to configure functions like Ethernet MAC addresses which must be unique within the system. This information will most probably be used by an on-board or a remote processor and be accessed via one of the FPGAs. Reprogrammability is required in order to allow for field-upgrades.

[1] In contrast, we expect the bitstream and program binaries to be uniform for all cards of the same kind.

[2] Actually, the JTAG cable has power and ground wires in addition to the 4 JTAG signals.

1.1.4. Approach

The general approach to configuration and re-configuration of the ROD/BOC system is to boot the cards into a state which enables the elementary functions including communication to the remote controlling host system (via VME and/or network). Subsequently new configuration binaries can be loaded, either transmitted from the remote system or from local non-volatile memory using the multi-boot mechanism (BOC only). In case of functional errors in the updated configuration a fall-back to the initial one can be forced at least via a power-cycle. Configuration data stored in the FLASH memories can in general[3] be updated from the remote system without needing physical access to the cards with the help of appropriate local firmware (e.g. FLASH update via embedded processor).

2. FPGA Programming

2.1. ROD

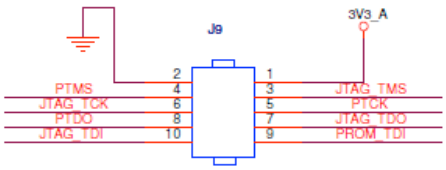
The ROD contains a collection of 4 different XILINX FPGAs

- Spartan-6 XC6SLX45FGG484, VME and low-level control (PRM)
- Two Spartan-6 XC6SLX150FGG900, data path
- Virtex-5 XC5VFX70TFF1136, control and local processing

2.1.1. PRM FPGA

PRM is a XC6SLX45FGG484 FPGA. It controls the VME interface, the MDSP host port and the corresponding interface of the Virtex-5 FPGA. PRM configuration data is stored in a Xilinx platform FLASH memory XCF16PVO48. Both the FLASH and the FPGA constitute a JTAG chain which is attached to connector J3 (right column), see Figure 4.

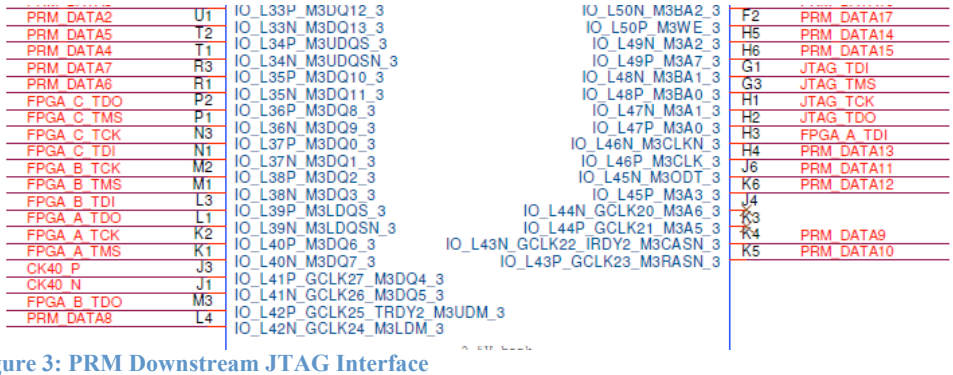
Figure 2: PRM JTAG Header



The left column of J9 connects to I/O pins of the PRM. A separate external JTAG port can be created through the PRM to access FPGA_A, B (both Spartan-6) and C (Virtex-5). As shown in Figure 3 the 4-wire JTAG interface of each FPGA (A, B and C) are connected to PRM I/O signals, allowing to create arbitrary interconnects between the devices.

The platform FLASH can only be programmed from an external JTAG programmer, thus field-upgrades require physical access to the ROD hardware. On the other hand the PRM functionality is relatively simple and well defined such that a field-upgrade is an unlikely situation. The corresponding connector is available at the ROD front-panel.

[3] Physical access with a JTAG programmer is required to update the PRM and DSP FLASH memories.



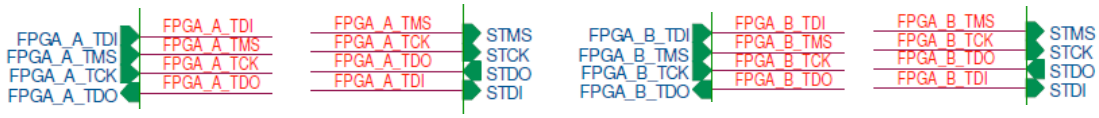
igure 3: PRM Downstream JTAG Interface

2.1.2. Spartan-6 Data Path FPGAs

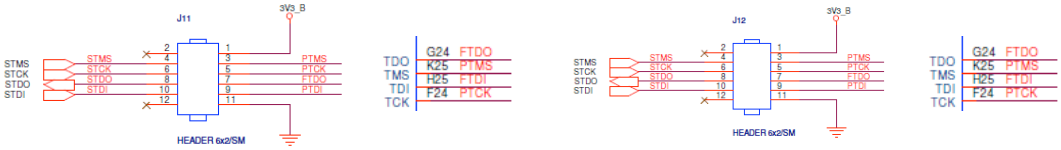
The two data-path FPGAs have identical programming interfaces. Each consists of a JTAG port which can be driven from an external programmer of from the PRM plus a FLASH memory capable to store a single bitstream. The FLASH is updated using the same JTAG interface[4]. Selecting external or PRM connectivity is done by installing (PRM) or removing (external) jumpers on 2x6 pin headers.

The following images show the interconnects at the level of the top-level schematics, the individual JTAG ports and the program memories.

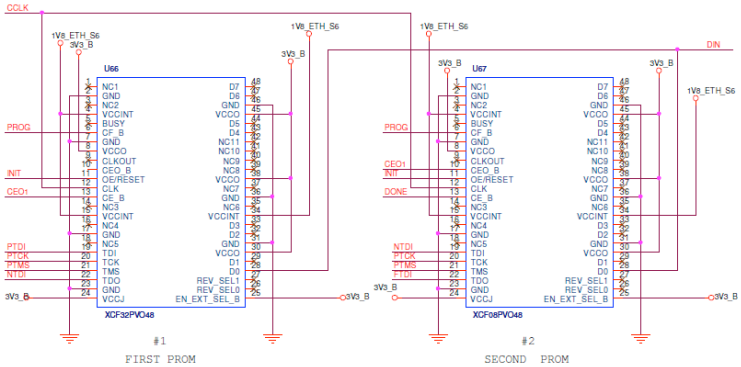
Top-level ports:



JTAG ports:



FLASH:



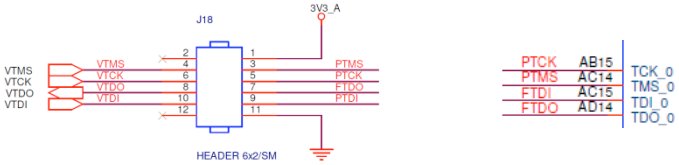
2.1.3. Virtex-5 Control FPGA

Configuration access to the Virtex-5 FPGA is done in the same way as described in section Spartan-6 Data Path FPGAs Only one memory device is used however.

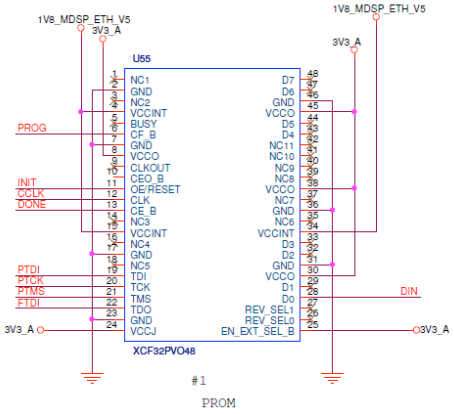
[4] Both FPGA and FLASH constitute a JTAG chain – multiple (here: two) devices connected with parallel control signals and serially cascaded data lines.

Top-level ports:

JTAG ports:



FLASH:



2.1.4. XVC-Mode

A remote JTAG mode will be available soon (~April 2014) which allows to use Impact, Chipscope and XMD via a network connection through the ROD master FPGA. This feature is based upon the "Xilinx Virtual Cable" (XVC) mechanism which implements an TCP/IP based cable server. The same mechanism is already in use on the BOC (see BOC documentation for more details, chapter 2.1.4.). XVC mode requires firmware features on both master and PRM FPGA and corresponding master software.

At the firmware level an 4 bit GPIO port is available to the master software, mapping the 4 JTAG signals TCK, TMS, TDI (to FPGA) and TDO (from FPGA) under software control. The GPIO port is driven by the XVC software. The signals are routed via the prm_control_in/out interface between master and PRM. At the PRM a multiplexer is implemented which selects either the JTAG cable header as endpoint for the main JTAG chain (master, slaves and assosciated PROMs) or the XVC interface. The selection defaults to cable mode upon reset and can be changed by the master software via the master-PRM SPI channel (see Chapter 7.).

2.2. BOC

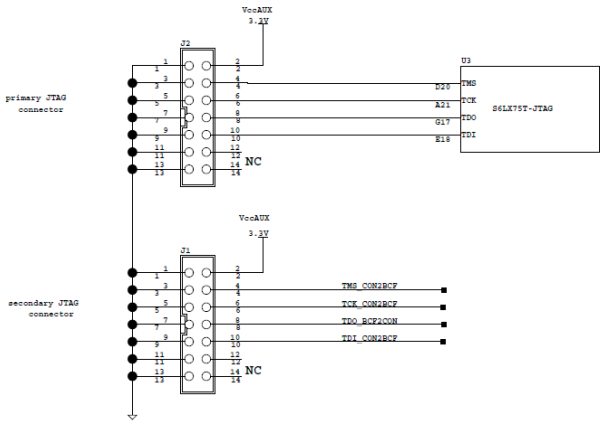
The BOC contains a collection of 3 different XILINX FPGAs

- Spartan-6 XC6SLXT75FGG484, ROD-interface and low-level control (BCF)
- Two Spartan-6 XC6SLXT150FGG900, data path (BMF)

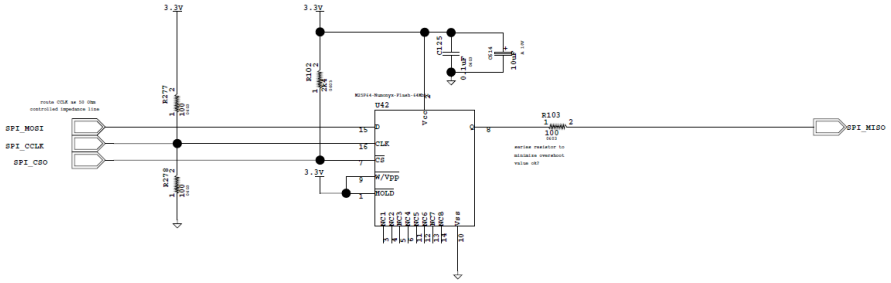
2.2.1. JTAG

The basic JTAG connectivity is implemented very similar to the ROD. The BCF is the primary device with a private JTAG connector (Figure 5, top) and a private FLASH memory (Figure 6). The two BMFs are connected to general I/O pins of the BCF with their JTAG ports (as above) and a secondary JTAG connector (Figure 5, bottom) attached to the BCF allows to create individual or daisy-chained JTAG access to the two BMFs.

Figure 4: BCF JTAG



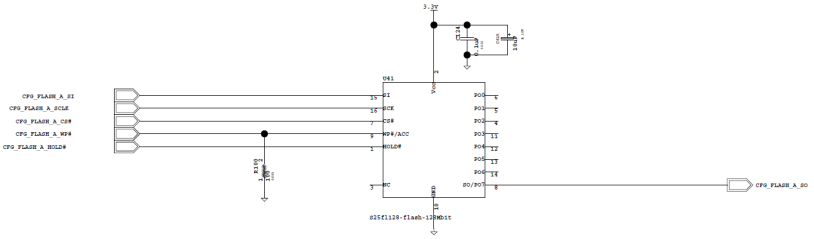
Connectors (top: primary, bottom: secondary)



igure 5: BCF Primary FLASH

Configuration data for the BMFs is stored in two serial FLASH memories attached to the BCF (Figure 2).

F



igure 6: BCF Secondary FLASH (1 of 2)

2.2.2. Configuration

The FPGA configuration mechanism is different from that on the ROD in the following respects:

- FLASH memories are serially attached (SPI mode), not via the JTAG chain. This allows that the BCF has read-write access to it's own configuration memory.

- All configuration memories are attached to the BCF, hence the BMFs are allways programmed by or via the BCF.
- The primary BCF FLASH memory supports the XILINX multi-boot feature.

During a regular BOC power-up sequence the BCF is configured from its primary FLASH memory with the default configuration. An alternative configuration stored in the same FLASH memory may be loaded subsequently under control of the initial binary (multi-boot feature). Next, configuration data for the two BMFs are read by the BCF (hardware or software) from the secondary FLASH memories and sent to the BMFs.

Using appropriate firmware at the BCF the BMFs can alternatively be configured from the ROD, from a remote host via the BOC Ethernet connection or via the USB to serial port located at the BOC front-panel. Configuration updates can be done via the same mechanisms.

2.2.3. BOC XVC JTAG

*** To Do: write this section ***

3. CPU Programming

All FPGAs on ROD and BOC allow to instantiate embedded processors (100MHz “soft”-cpu. Virtex-5 with option for 450MHz PowerPC core). The binaries for the embedded processors can be embedded within the FPGA bit-stream, provided the memory footprint is sufficiently small. In case of larger memory requirements (e.g. to boot Linux) a small embedded boot-loader has to copy the final binary into the external DDR2-memory of the FPGA (e.g. from the FLASH or from VME).

4. DSP Programming

4.1. VME Mode

The recommended and from experience most stable way of reprogramming the DSP residing on the ROD is via VME using the HPI of the DSP itself. A dedicated executable within the PixelDAQ Software framework called MdspFlashLoad can be used to load a binary file onto the DSP's Flash. Five BootMode lines set by the PRM determine the bootmode of the DSP. Currently there are two options implemented in the PRM. A BootMode of 01101 can be set by writing 0 to the PRM register address C00008 and stands for a ROM boot. Writing 1 to the mentioned register address a BootMode of 00111 is selected which lets the DSP boot from HPI.

4.2. JTAG Mode

An alternative may to program the DSP is via a dedicated JTAG interface. The Texas-Instruments JTAG debugger also allows programming of the attached FLASH memory (see Figure 2). After power-up the DSP boots from the FLASH memory.

There is no alternative JTAG path from the PRM to the DSP. Invalid FLASH content requires to reprogram the memory with the JTAG debugger.

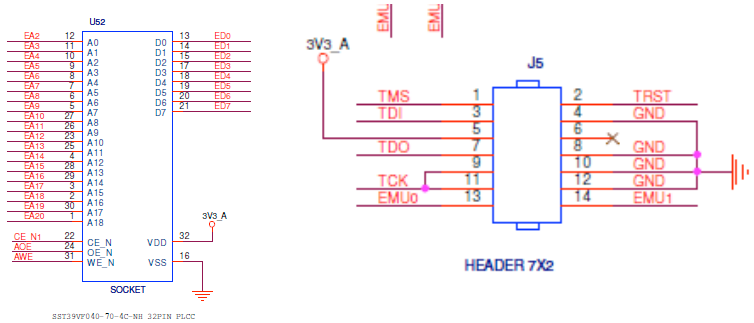


Figure 7: MSDP JTAG Connector and FLASH

5. Parameters

Non-volatile parameters can be stored on the BOC in any of the FLASH memories. Reading and updating can be done via a simple SPI interface. The JTAG-based FLASH memories on the ROD do not provide similar support.

In addition, the BOC provides a silicon-serial-number device (Maxim DS2401) and a 1kB EEPROM (24AA01) to read/write non-volatile parameters. Both components are attached to the BCF.

On the ROD a dedicated data FLASH memory (Atmel AT45DB642D, 8MB) attached to the Virtex-5 FPGA is used to store the binary for the embedded processor (e.g. PowerPC), but can also accommodate non-volatile parameters. In addition, the ROD has a mechanical 4-bit switch (e.g. for providing unique board identification within a crate).

6. ROD 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.

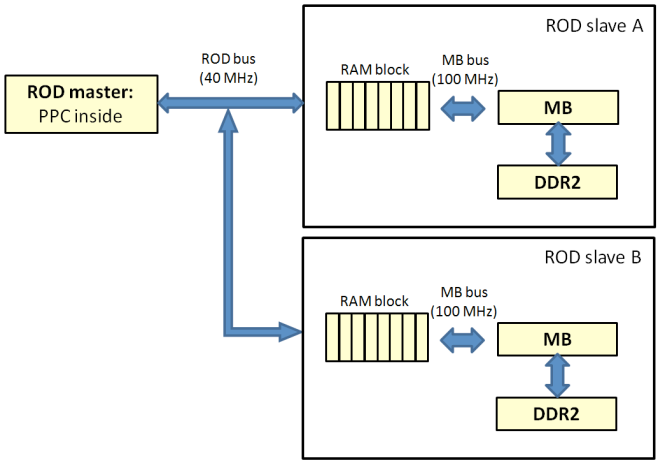


Figure 8: Master-Slave Interconnect

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 uses the same mechanism.

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

6.1. 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.

6.2. 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 Status).

6.3. Commands

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

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

6.3.1. 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    // histogrammer 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.

6.3.2. 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 IblSlvNetCfg. There the local configuration of the slave is set (IP address, subnet mask, gateway) as well as separate target IP address and port number combinations for both slave histogramming units.

The configuration can be retrieved via the command SLV_CMD_NET_CFG_GET.

6.3.3. 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 Master control commands).

6.3.4. Histogram control

See section Histogram Control Mechanism

6.4. Controller level command access

Access to the master and slave commands is done via the traditional “primitives”. 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.

6.4.1. Slave commands

*** TODO: Update – new way of sending commands to PPCpp software on PPC ***

Karolos, Steve, Laura

6.4.2. 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

7. ROD Master-PRM Communication

A simple communication channel using an SPI interface is available to transfer data between the two FPGAs. An SPI transfer is initiated by driving the SlaveSelect line of the PRM interface (device spi_1) low. PRM data comprises a 32 bit status word followed by the 160 bit git id. The SPI configured in 32 bit mode and the data must be read in 6 successive read cycles without toggling SlaveSelect. At the end, SlaveSelect has to be braucht to the inactive state.

The status word contains the following information:

- (MSB) 4 bit “magic” nibble: 0x5
- 4 bit SPI_CTL_MODE (see below), default 0x0
- 8 bit master watchdog trigger counter
- 8 bit serial id (hex switches)
- 2 bit “00”
- 1 bit VME geographical address parity bit (GA_P)
- (LSB) 5 bit VME geographical address (inverted) (GA_N0..4)

The watchdog trigger count is reset by VME or power-up reset and increments with every watchdog reset event generated by the master.

SPI is a bi-directional interface and information can also be sent to the PRM. To date, there is only a 4 bit register available at the PRM which is filled with the trailing 4 bits of the 192 bit SPI word. Be warned that the bit order of the 4 bits is reversed, transmitting bit 3 controls bit 0 of the mode word (likewise 2 -> 1, 1 -> 2, 0 -> 3). So far, the only bit in use is PRM mode bit 0 (transmitted via SPI bit 3) which controls the XVC mode.

- Bit 0 = 0: cable mode (default after reset)
- Bit 0 = 1: XVC mode

The JTAG cable cannot be used in XVC mode.

7.1. HPI

HPI is a mechanism to access the PowerPC address space from VME: all the PowerPC address space can be accessed except the DDR2 region.

HPI is composed by 2 firmware blocks:

1. A PLB Master peripheral implemented into the RodMaster, directly connected to the PLB PowerPC bus; its external interface has been implemented with a custom protocol (very similar to the MSDP HPI).
2. A block into the PRM that translates VME access into operations on that HPI-like protocol.

4 VME registers have been implemented:

VME_ADDRESS[23:0]	NAME	Read/Write	# of bits	Note
0x100000	HPIC	RW	16	dummy
0x300000	HPIA	RW	32	
0x500000	HPID_AUTO	RW	32	
0x700000	HPID_NOAUTO	RW	32	

When a read/write VME cycle is performed on the HPID_NOAUTO register, the same cycle is done on the PowerPc memory space at the address whose value is stored in HPIA.

Example 1: if you want to read `PPC_CTL_REG` reg, you have to write: `XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x1000004` into the HPIA (= 0x40000000 + `0x1000004` = 0x41000004) and read HPID_NOAUTO.

Example 2: if you want to write the same register, just write 0x41000004 on HPIA (as before) and write the wanted value into HPID_NOAUTO; this value will be written into `PPC_CTL_REG` reg.

For coding example look at IblUtils/Vme/src/libs/IsfFlash , class PpcVmeInterface, methods ppcSingleRead and ppcSingleWrite.

If you perform multiple operation on the same PowerPc register, you don't need to rewrite the address into HPIA.

When a read/write cycle is executed on HPID_AUTO, it behaves like HPID_NOAUTO, but at the end the HPIA value is incremented by 0x4, so that at the following cycle the operation is performed on the adjacent PowerPC address. HPID_AUTO reg is useful for reading/writing contiguous memory regions.

Important note: HPI access to the PowerPc memory is **disabled** by default. You have to enable it by writing 1 into both bits PPC_CTL_HPIENABLE_BIT and PPC_CTL_MASTER_BIT of the PPC_CTL_REG register. This is already done for instance into the rodMaster_flash.bit programming file, generated with the embedded code in RodMaster/Firmware/sw/flashloader.

8. ROD Design

This document arises from the need of a clear and ordered list of the features for the IBL ROD firmware. The ultimate goal is to collect a list of most of the needed functionalities, so to be able to assign the design of the blocks to different teams.

The idea is to stick as much as possible to the firmware architecture of the Pixel ROD. The main architectural changes with respect to it are due to the following differences:

1. the Pixel ROD has to manage data coming from up to 96 links, while the IBL ROD has to manage data from 32 links (= 32 FeI4 chips). So, for instance, the format of the “EFB Event ID and Dynamic Mask Data” features 192 bits in the Pixel ROD (2 bits per link), while it features 64 bits in the IBL ROD.
2. With the new front end architecture (FeI4 vs FeI3-MCC) all the error diagnosis blocks need to be changed accordingly.

3.

9. ROD Control and Data Path

9.1. ROD firmware

The ATLAS IBL Read Out Driver (IBL ROD) electronics board, used in the ATLAS IBL subsystem, is designed specifically to interface with IBL modules and their front-end electron ics (FeI4). Each IBL ROD interfaces directly with one IBL Back-of-Crate (IBL BOC) electronics board that connects to the IBL modules using fiber optic links. One Timing Interface Module (TIM) electronics board:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3055&rep=rep1&type=pdf

distributes the global timing and trigger information to 15` IBL ROD modules. The IBL ROD communicates with a system host through VME and Ethernet connection. Formatted event data are transmitted off the IBL ROD back to the IBL BOC through the crate backplane.

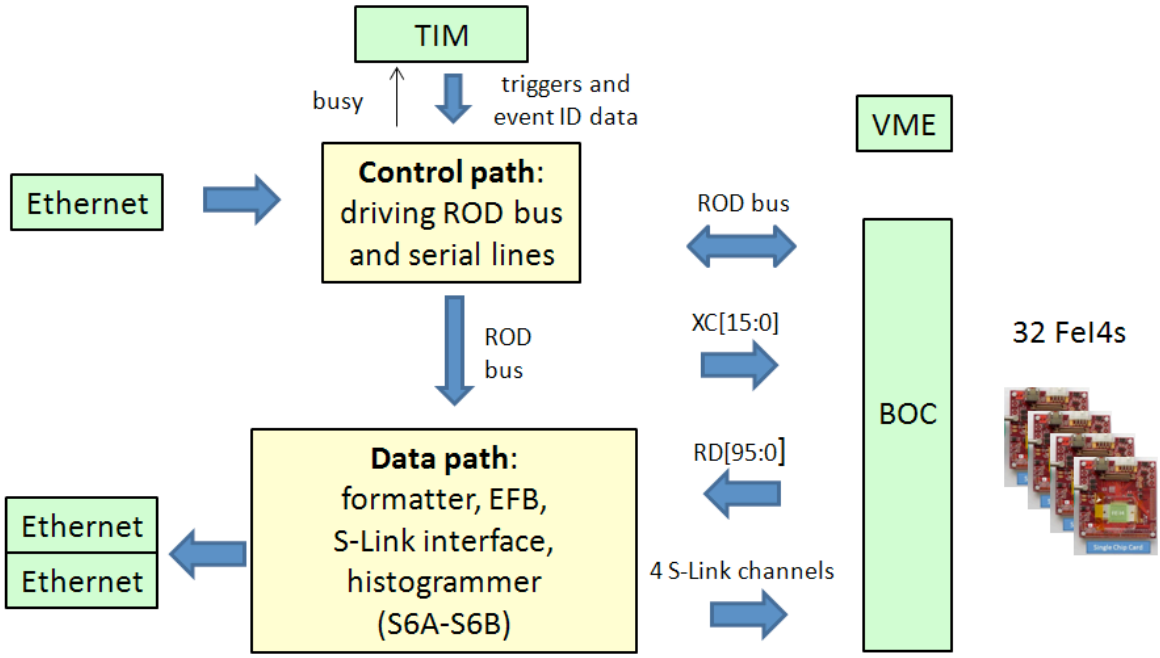


Figure 9: IBL ROD general structure

The IBL ROD-BOC system deals with 32 FeI4 chips and 4 S-Links.

One basic function of the IBL ROD is to transmit control commands and configuration data through the IBL BOC to the FeI4 chips. The transmit data is sent to the modules as 40 MHz serial data streams and can be Level 1 triggers, Event Counter resets, Bunch-Counter resets, calibration commands or module register data.

A second basic function of the IBL ROD is to receive data streams from the FE chips, after the IBL BOC performs 8B/10B decoding on 160 Mb/s serial data lines (one per chip). The Field Programmable Gate Arrays (FPGA) on the IBL ROD are designed to detect and

mark format errors in the event data, build event fragments and transmit the event fragments to the Read Out Link (ROL) system (via the IBL BOC).

In normal running (physics data taking) the TIM supplies the trigger and event description data to the IBL ROD. The trigger is detected and expanded into the module trigger commands required by the IBL FE modules. The FE Trigger code is sent to a 16-bit wide mask gate that sends the trigger to the active modules.

Here's the sequence of operations:

- At the start of the run, the PPC sets the active links for the default mask;
- after the IBL ROD sends the Trigger code to the FE modules, a set of dynamic mode bits are sent to all of the Formatters;
- after sending the L1 pulse to the IBL ROD, the TIM transmits serial data that describes the rest of the ATLAS trigger information.
- when the TIM data for a specific trigger has been transmitted, that data (a dynamic mask and the number of event counter resets) are sent to the Event Fragment Builder (EFB) block. After a delay of a few clock periods to allow the event description data to propagate to the EFB, a token is sent to the Formatters to start the readout of the incoming data.

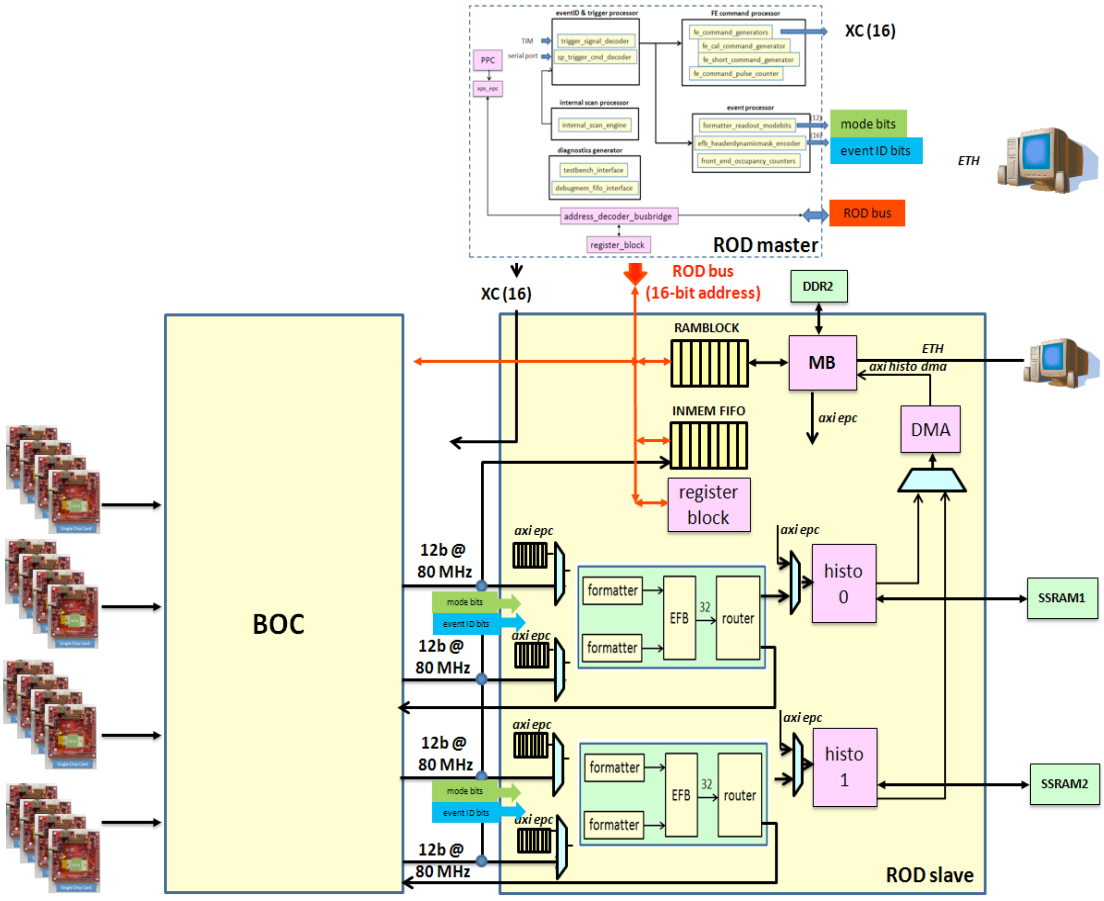


Figure 1: IBL ROD complete firmware (only one slave is shown)

9.2. ROD master: VHDL blocks

The ROD Virtex5 FPGA coordinates all the real-time control operations that are required for normal data taking, module calibration and on-board diagnostics; it connects the data path FPGAs (the slaves) and the BOC board to the PPC and interfaces with the TIM module for access to the TTC data and commands.

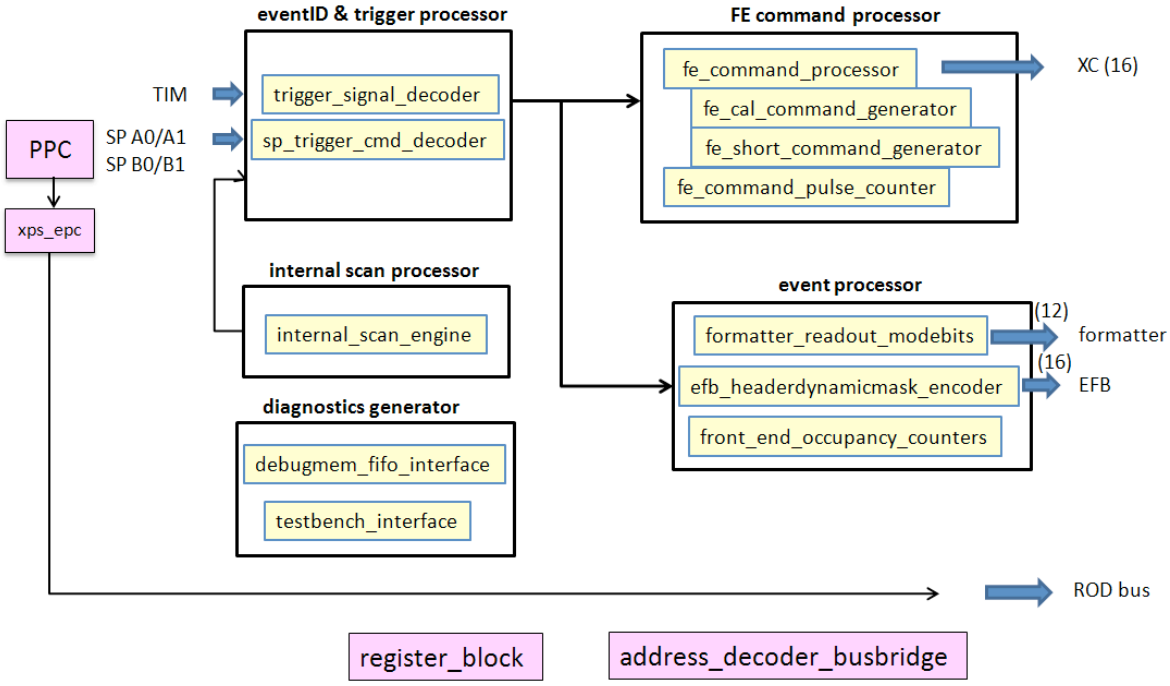


Figure 10: ROD control path

9.2.1. Functionality

9.2.2. External interfaces

9.2.3. Input data format

9.2.4. Output data format

9.2.5. Performances

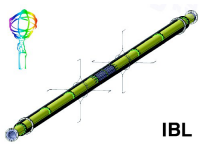
9.2.6. Implementation

9.2.7. Error handling

9.2.8. Registers

Master main firmware blocks:

Event ID & trigger processor:
trigger signal decoder
sp trigger cmd decoder
FE command processor:
fe command processor
fe cal command generator
fe short command generator
fe command pulse counter
Event processor:
formatter readout modebits
EFB header dynamic mask encoder
front end occupancy counters
Internal scan processor



Diagnostics generator
Address decoder bus bridge
Register block
Busy block
PPC peripherals (with watchdog)

9.3. EventID & trigger processor

9.3.1. Functionality

The Event ID and Trigger processor is the primary collection of real-time event building operations in the ROD. Its primary function is to gather triggers and event description data from all of the trigger sources, format the data and transmit triggers to the FE Command Processor and data to the Event Processor functions in the V5 FPGA. Trigger sources available to and on the ROD include the TIM module, the Internal Scan Processor, the PPC and the Diagnostics Generator.

9.3.2. External interfaces

9.3.3. Input data format

9.3.4. Output data format

9.3.5. Performances

9.3.6. Implementation

Trigger_signal_decoder

Implements the serialL1ID and TriggerType serial decoders with simple no error detection shift registers. There is no examination for consistency with local L1/BC counters, proper size or any sort of bit error or CRC checking.

Serial input data streams are all positive logic, LSB first:

Serial ID: leading zero(s) & 1 & 24bits L1ID & 12bits BCID & trailing_zero(s)

Serial TT: leading zero(s) & 1 & 8+2bits trigger type & trailing_zero(s)

Sp_trigger_cmd_decoder

This block counts L1 Triggers issued by the PPC during calibration and special testing modes. Command bit streams to front end modules are:

l1accept:	"11101"	
bcr:	"101100001"	Bunch Counter Reset
ecr:	"101100010"	Event Counter Reset
cal:	"101100100"	Calibration Trigger

When a L1A is detected on one of the serial port inputs, the ID values are loaded into the queue FIFO on the rising edge of the next system clock. Each serial port input has a L1 ID counter. The L1ID counters are reset to 0 when an ECR command is detected and

incremented by 1 when and L1 trigger is detected. The block also contains 2 free running Bunch Counter ID (BCID) counters. When a Bunch Counter Reset (BCR) is detected, the BCID counters are reset to 0. When a L1A is detected, the value of the BCID counter at that time is latched into register and can be used in the Event Header as the event BCID.

9.3.7. Error handling

9.3.8. Registers

9.4. Formatter_readout_modebits

9.4.1. Functionality

The Formatter Read Out Mode Bits Encoder sends the appropriate Mode Bit mask to the Formatters when the ROD detects a L1 trigger from any source.

The Formatter Read Out Mode Bits are stored in a LUT in internal Block RAM in the V5.

The ROD has 2 sets of Read Out Mode Bits: the default set and the corrective set.

The default mode bits are sent to the Formatters for all "normal" (no correction required) triggers received from the TIM. If the ROD is using the PPC Serial Port Outputs (SP_A0/A1 & SP_B0/B1) for trigger generation, the Default Mode Bits are sent when a trigger is detected on SP_A and the Corrective Mode Bits are sent when a trigger is detected on SP_B.

MB1	MB0	Formatter Read Out Mode Bits Definitions
0	0	Play 1st event from Link FIFO. Normal data flow from Formatter to EFB.
0	1	Mask this link for 1 event. In this mode, the Formatter will dump one event then move on to the next active link.
1	0	Skip read out of this link for 1 event for re-synchronization.
1	1	Read out 1st event from FIFO and dump it on the floor, play the 2nd event to the EFB.

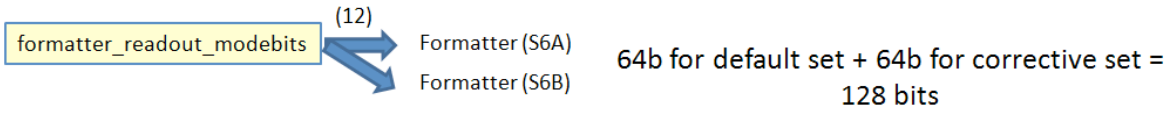


Figure 11: Formatter Mode Bits

9.4.2. External interfaces

9.4.3. Input data format

9.4.4. Output data format

9.4.5. Performances

9.4.6. Implementation

9.4.7. Error handling

9.4.8. Registers

9.5. Efb_headerdynamicmask_encoder

9.5.1. Functionality

The EFB Header ID and Dynamic Mask Encoder is used to transmit the trigger information that is sent from the TTC system or generated internally on the ROD to the EFB prior to the arrival of event data from the FE modules. The EFB Header/Dynamic Mask Encoder is implemented in the V5 and consists of 5 FIFOs, 2 Look-Up-Tables (LUT) and a FSM to control the Event Header building task. The primary function is to collect all of the trigger description data, format and send the Event ID Header and the default or corrective Dynamic Mask information to the EFB.

The Header ID values are used to generate the Event Fragment Header and consists of the event specific L1ID, ECRID, BCID, Atlas Trigger Type, TIM Trigger Type and the ROD Event Type. The source for the Header ID values depends on the current mode of the ROD.

If the TIM Trigger Signal Decoder is enabled, the L1ID, BCID and Atlas and TIM Trigger Types will be sent to the ROD on the TIM TTC Bus inputs and, when received, loaded to the Header/Dynamic Mask Output FIFO. A L1A from the TIM will load the ROD Trigger Type from the internal memory location to the Header/Dynamic Mask Output FIFO. If the ROD is using the PPC Serial Port Outputs (SP_A & SP_B) for triggers, the Header ID information is generated using internal counters and registers in the V5 for each input.

The Dynamic Mask Bits, 2 bits per link, enable dynamic synchronization error correction in the EFB on a link by link basis. A corrective mask, used for error correction, can be written to the Corrective Dynamic Mask Bits Register and sent to the EFB when the ROD receives an L1A from the TIM and the “Corrective Mode Bits and Dynamic Mask Ready” bit is set in the V5 Main Control Register (0x00404410, Bit 23). When a corrective mask is sent to the EFB, the Corrective Dynamic Mask bits are written into the Default Mask LUT.

When the ROD receives a “normal” (no correction required) L1A, the Default Dynamic Mask Registers are sent as the Dynamic Mask. If the ROD is using the Serial Port Inputs (SP_A & SP_B), the Default Dynamic Mask Bits are sent when a trigger is detected on SP_A and the Corrective Dynamic Mask Bits are sent when a trigger is detected on SP_B. The following table shows the definitions of the Dynamic Mask Bits.

DMB[1:0]	EFB Dynamic Mask Bits Definitions
00	No Change to L1 ID
01	Increment L1 ID by 1
10	Decrement L1 ID by 1

Figure 12: Dynamic Mask Bits

word 0 : L1 ID[15:0]

word 1 : ECR ID[7:0] & L1 ID[23:16]

word 2 : BC ID[11:0] & RoL & BOC OK & TIM OK

word 3 : TT - ROD[5:0] & TIM[1:0] & ATLAS[7:0]

word 4 : Dynamic Mask 0 for Links [7: 0]

word 5 : Dynamic Mask 1 for Links [15: 8]

word 6 : Dynamic Mask 2 for Links [23:16]

word 7 : Dynamic Mask 3 for Links [31:24]

Figure 13: EFB data distribution

9.5.2. External interfaces

9.5.3. Input data format

9.5.4. Output data format

9.5.5. Performances

9.5.6. Implementation

9.5.7. Error handling

9.5.8. Registers

9.6. Front_end_occupancy_counters

9.6.1. Functionality

For each link, this block increments a 4-bit counter when a L1 trigger is issued to the FE chips and decrements it when a trailer is written in to the Link Formatter FIFO. The FE Occupancy Counters can be used as a diagnostic tool to monitor the occupancy of the Formatters and the status of data flow from a FE module.

The expectation is that the Front End Occupancy Counters are to be used with periodic resets to determine when the ROD can start configuring FE modules that are not acting properly. The counter can also be used as a diagnostic on the performance of the system and they are used by the Internal Scan Engine to indicate when a new trigger can be issued to the FE Modules during a calibration run.

The “FE Occupancy Counters All Zero” bit in the V5 Main Status Register is set if all of the FE Occupancy Counters have the value 0. The “All Zero” bit can be used to indicate that all modules have finished sending event data to the ROD. It will have a value of 0 if any Counter has a value greater than 0.

Each link counter can be reset independently of all others by setting the bits in the FE Occupancy Counter Reset Register that correspond to the inactive link. The counters can be disabled individually by setting the bits in the FE Dead Data Link Mask Register that correspond to links that are turned off at the Formatter. For diagnostic purposes, the FE Occupancy Counters can be loaded with a pre-set value, the same for all links.

9.6.2. External interfaces

9.6.3. Input data format

9.6.4. Output data format

9.6.5. Performances

9.6.6. Implementation

9.6.7. Error handling

9.6.8. Registers

9.7. Fe_command_processor

9.7.1. Functionality

The FE Command Processor accepts trigger and timing commands from the TIM module, PPC, Internal Scan Engine, Diagnostics Generator and distributes these signals to the BOC module (via the slaves) and internally to specific ROD functions. The primary operation of the FE Command Processor block is to detect all input signals that correspond to L1 Accept (L1A) triggers, Event Counter Reset commands, Bunch Counter Reset commands and Calibration Strobe commands as they arrive on the ROD.

The values set in the FE Command Link Mask Registers determine which FE Modules will receive command streams.

TTC Command	Output to Pixel FE Chip
L1A	"11010"
BCR	"101100001"
ECR	"101100010"
CAL	"101100100" followed by a L1A packet after a preset delay equal to the value stored in the Cal Strobe Delay Register (0x00404440).

igure 14: FE Command Encoding

9.7.2. External interfaces

9.7.3. Input data format

9.7.4. Output data format

9.7.5. Performances

9.7.6. Implementation

9.7.7. Error handling

9.7.8. Registers

9.8. Internal_scan_engine

9.8.1. Functionality

The Internal Scan Engine in the V5 provides a speed-optimized system for running calibration scans on the ROD. The primary structure of the process allows only L1 triggers and CAL commands to be sent on the FE command links, so all ROD setup, FE module configuration data, BCR and ECR commands must be controlled by the PPC. The configuration of the Scan Engine is accomplished using VME accessible static registers that allow the user to set the specific scan variables.

9.8.2. External interfaces

9.8.3. Input data format

9.8.4. Output data format

9.8.5. Performances

9.8.6. Implementation

9.8.7. Error handling

9.8.8. Registers

9.9. Diagnostics generator

9.9.1. Functionality

9.9.2. External interfaces

9.9.3. Input data format

9.9.4. Output data format

9.9.5. Performances

9.9.6. Implementation

9.9.7. Error handling

9.9.8. Registers

9.10. estbench_interface

9.10.1. Functionality

The Test Bench Interface is used to run internal, self-test diagnostics on the ROD that will be used primarily in production to verify the function and performance of all of the fabricated RODs. This allow for testing of the ROD with trigger rates of 100kHz. There is a second variant where triggers come from the serial data stream from the PPC and the data from slave internal RAM.

9.10.2. External interfaces

9.10.3. Input data format

9.10.4. Output data format

9.10.5. Performances

9.10.6. Implementation

9.10.7. Error handling

9.10.8. Registers

9.11. ROD Master busy block

9.11.1. Functionality

The ROD busy block in the ROD master collects the state of all possible busy sources (see 8.?.2) and forwards the busy signal to the TIM (active low format). A histogram to monitor the different busy sources is generated. The histogram can be read for information/debugging purposes by the host. A 16 bit counter is implemented for each input and incremented on the negative edge (active low format).

An 8-bit mask is implemented to provide masking functionality of single busy inputs. Mask is active high logic: mask(x) == 1 —> input masked. The mask is controlled by the busy register bits 7:0.

Each individual input can be overwritten using the 8-bit input bus force_input(7:0), which is connected to the busy register bits 15 to 8. All force_input bits follow the input logic of the same logic as the corresponding inputs. Therefore force_input(5, 4, 2, 1) need to be set to '1' to be inactive!

Bit force_input(0) is used to force the busy output, implemented as active high logic. It is masked consequently by mask(0).

9.11.2. External interfaces

Inputs:

The ROD busy receives one input signal per slave FPGA. The busy generation of the slave is defined by the slave busy block, see section 8.?. Additionally to the inputs from the slave, the ROD busy module has one input for calibration, one per reset type. An 8 bit bus (connected to the register block with default set to 8xff) masks individual busy inputs.

Mask connection:

force_rod_busy	mask(0) — force_rod_busy is NOT an output signal
rod_busy_n_in_p(0)	mask(1)
rod_busy_n_in_p(1)	mask(2)
efb_header_pause_i	mask(3)
formatter_mb_fef_n_in_p(0)	mask(4)
formatter_mb_fef_n_in_p(1)	mask(5)
rol_test_pause_i	mask(6)

(force input bits similarly)

Outputs:

The ROD busy has one output link to the TIM module. Additionally, the current status of the ROD busy is written to a register, which can be read by the host PC. This register

contains one bit for the status of the output, and one bit for the status of each input in order to get information of the source of the current busy.

9.11.3. Input data format

All input links are single wire inputs, active high (or low? Let's stick to the TIM module format...)

9.11.4. Output data format

The output signal to the TIM is also a single wire with active high logic. The current status register is read only. (How to read it ?)

9.11.5. Performances

9.11.6. Implementation

9.11.7. Error handling

9.11.8. Registers

9.12. ROD slave: VHDL Blocks

The ROD data path is physically implemented over the 2 Spartan6 devices (slaves) on the IBL ROD. The following paragraphs show the blocks which are part of the overall firmware and what are their functions. The main blocks inside the slave are:

- formatter
- event fragment builder (EFB)
- router
- histogrammer
- INMEM FIFO
- RAM block
- Microblaze peripherals
- register block

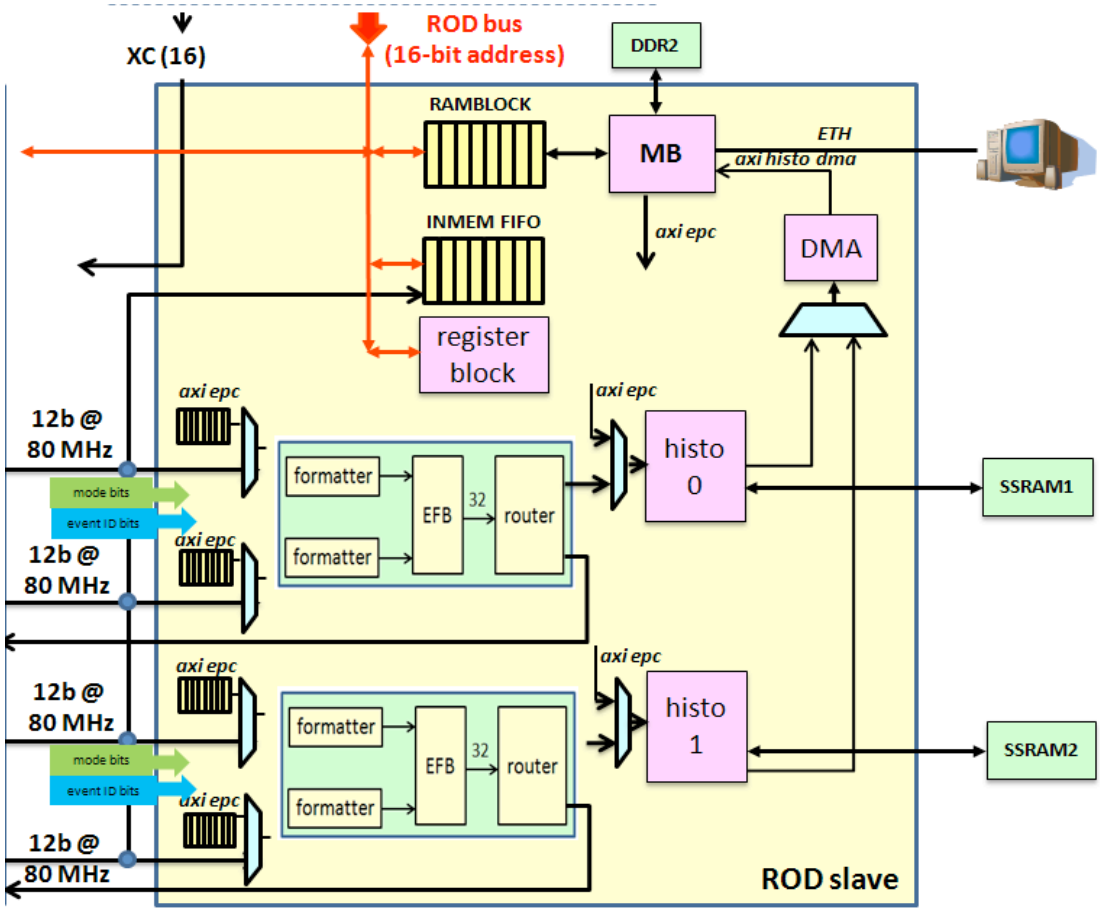


Fig. 28: Slave schematic diagram

Slave register table

Description	Address	Access	Width
Formatter Link Enable	0x0000	RW	16
ROD Slave ID	0x0005	RW	1
Calibration Mode (dump S-Link data)	0x0006	RW	1
Trigger Count Number	0x0007	RW	4
Formatter Readout Timeout Limit	0x0010	RW	32
Formatter Data Overflow Limit	0x0014	RW	16
Formatter Header Trailer Limit	0x0018	RW	32
Formatter ROD Busy Limit	0x001C	RW	32
Formatter Trailer Timeout Limit	0x001F	RW	32
Formatter Control Register (Reserved)	0x0020	RW	32
Formatter0 Status Register (Reserved)	0x0022	R	32
Formatter1 Status Register (Reserved)	0x0023	R	32
Formatter2 Status Register (Reserved)	0x0024	R	32
Formatter3 Status Register (Reserved)	0x0025	R	32
Formatter0 Occupancy Register	0x0027	R	40
Formatter1 Occupancy Register	0x0029	R	40
Formatter2 Occupancy Register	0x002B	R	40
Formatter3 Occupancy Register	0x002D	R	40
Formatter0 Frames Register	0x0030	R	16

Formatter1 Frames Register	0x0034	R	16
Formatter2 Frames Register	0x0038	R	16
Formatter3 Frames Register	0x003C	R	16
Mode bits… **(still missing ...)**			
INMEM FIFO Channel Select Select BOC to ROD channel	0x0040	RW	2
INMEM FIFO Reset	0x0042	W	1
INMEM FIFO Data	0x0044	R	12
Formatter Readout Timeout Error	0x0070	R	16
Formatter Data Overflow Error	0x0074	R	16
Formatter Header Trailer Error	0x0078	R	16
Formatter ROD Busy Error	0x007A	R	16
RAM Block Select	0x1---	WR	
Slave0, Half-Slave 0 Error Masks Link 0 Link 1 Link 2 Link 3 Link 4 Link 5	0x2000 0x2004 0x2008 0x200C 0x2010 0x2014	RW	32

Link 6 Link 7	0x2018 0x201C		
Slave 0, Half-Slave 1 Error Masks Link 8 Link 9 Link 10 Link 11 Link 12 Link 13 Link 14 Link 15	0x2020 0x2024 0x2028 0x202C 0x2030 0x2034 0x2038 0x203C	RW	32
Slave 1, Half-Slave 0 Error Masks Link 16 Link 17 Link 18 Link 19 Link 20 Link 21 Link 22 Link 23	0x2100 0x2104 0x2108 0x210C 0x2110 0x2114 0x2118 0x211C	RW	32
Slave 1, Half-Slave 1 Error Masks Link 24 Link 25 Link 26	0x2120 0x2124 0x2128	RW	32

Link 27 Link 28 Link 29 Link 30 Link 31	0x212C 0x2130 0x2134 0x2138 0x213C		
EFB Format Version 0x0301UUUU U = user defined, top half fixed	0x2200	R RW	32 16
EFB Source ID 0xUU1SMMMM S = Sub Detector ID, M = Module ID	0x2204	R RW	32 23
EFB Run Number	0x2208	RW	32
EFB1 Command Register Bit[0]: 0 (send empty events) Bit[1]: Mask BCID Error Bit[2]: Group Event Counter Enable Bit[3]: Mask L1ID Error Bit[4:7]: 0 (link input test select) Bit[8]: Mask TIM Clock Error Bit[9]: Mask BOC Clock Error Bit[10]: 0 (mask sweeper error) Bit[11]: Enable L1 and BC ID Trap Bit[12]: TIM BCID Readout Link Enable Bit[13]: BCID Rollover Select Bit[14:15]: 0 Bit[16:25]: BCID Offset	0x2210	RW	32
EFB2 Command Register	0x2212	RW	32

See above description (should revise)			
EFB1 Run-Time Status Register Bit[0]: FIFO1 "almost_full_n" Status Flag Bit[1]: err_sumry_fifo1_almost_full Bit[2]: ev_id_fifo_empty_error1 Bit[3]: fifo 1 pause to Formatter Bit[4]: FIFO2 "almost_full_n" Status Flag Bit[5]: err_sumry_fifo2_almost_full Bit[6]: ev_id_fifo_empty_error2 Bit[7]: fifo 2 pause to Formatter Bit[8]: halt output from router Bit[9]: rod_event_type_empty Bit[10]: rod_event_type_afull Bit[11]: rod_event_type_full	0x2218	R	12
EFB2 Run-Time Status Register See above description	0x221A	R	32
ROD Code Version and Board Version Bit[0:7]: HDL Code Version Bit[8]: 0 Bit[9:15]: 00 & Board Revision Number	0x221C	R	16
EFB1 Event Header Data	0x2220	R	16
EFB2 Event Header Data	0x2222	R	16
EFB1 Group Event Count	0x2230	R	32
EFB2 Group Event Count	0x2232	R	32
EFB1 Out FIFOs Reset	0x2248	W	1

EFB2 Out FIFOs Reset	0x224A	W	1
EFB1 Out FIFO Status Flags Bit[0]: FIFO1 NOT "empty_n" Status Flag Bit[1]: FIFO1 NOT "almost_empty_n" Status Flag Bit[2]: FIFO1 NOT "full_n" Status Flag Bit[3]: FIFO1 NOT "almost_full_n" Status Flag Bit[4]: FIFO2 NOT "empty_n" Status Flag Bit[5]: FIFO2 NOT "almost_empty_n" Status Flag Bit[6]: FIFO2 NOT "full_n" Status Flag Bit[7]: FIFO2 NOT "almost_full_n" Status Flag Bit[8]: 0 (HTFIFO "empty" Status Flag) Bit[9]: 0 (HTFIFO "full" Status Flag) Bit[10]: ev_id_empty1 Bit[11]: ev_id_almost_full1 Bit[12]: ev_id_empty2 Bit[13]: ev_id_almost_full2 Bit[14]: header_ev_id_empty Bit[15]: header_ev_id_almost_full Bit[16]: header_ev_id_full Bit[17]: ev_data_empty Bit[18]: ev_id_full1 Bit[19]: ev_id_full2 Bit[20]: count_fifo1_full Bit[21]: count_fifo2_full Bit[22]: err_sumry_fifo1_full	0x224C	R	24

Bit[24]: err_sumry_fifo2_full			
EFB2 Out FIFO Status Flags See above description	0x224E	R	32
EFB1 L1/BCID Trapped Values Bit[15:0]: latched_bcid0 & 0 & latched_l1id0 Bit[31:16]: latched_bcid1 & 0 & latched_l1id1	0x227C	R	32
EFB2 L1/BCID Trapped Values See above description	0x227D	R	32
EFB1 Miscellaneous Status Register Bit[0]: err_sumry_fifo1_almost_full Bit[1]: err_sumry_fifo2_almost_full Bit[2]: ev_id_fifo_empty_error1 Bit[3]: ev_id_fifo_empty_error2 Bit[4]: ev_id_empty1 Bit[5]: ev_id_empty2 Bit[6]: header_ev_id_empty Bit[7]: fifo1_pause_o Bit[8]: fifo2_pause_o Bit[9]: gatherer_halt_output Bit[10:17]: 0 Bit[18]: header_ev_id_data_out(35) Bit[19]: ev_id_almost_full1 Bit[20]: ev_id_almost_full2 Bit[21]: header_ev_id_almost_full Bit[22]: ev_data_almost_full_n_o	0x227E	R	32

Bit[23]: header_ev_id_full Bit[24]: ev_id_full1 Bit[25]: ev_id_full2 Bit[26]: ev_data_empty_o Bit[27]: err_sumry_fifo1_full Bit[28]: err_sumry_fifo2_full Bit[29]: count_fifo1_full Bit[30]: count_fifo2_full			
EFB2 Miscellaneous Status Register See above description	0x227F	R	32
Link 0 Errors Register	0x4000	R	32
Link 1 Errors Register	0x4004	R	32
Link 2 Errors Register	0x4008	R	32
Link 3 Errors Register	0x400C	R	32
Link 4 Errors Register	0x4010	R	32
Link 5 Errors Register	0x4014	R	32
Link 6 Errors Register	0x4018	R	32
Link 7 Errors Register	0x401C	R	32
Link 8 Errors Register	0x4020	R	32
Link 9 Errors Register	0x4024	R	32
Link 10 Errors Register	0x4028	R	32
Link 11 Errors Register	0x402C	R	32
Link 12 Errors Register	0x4030	R	32

Link 13 Errors Register	0x4034	R	32
Link 14 Errors Register	0x4038	R	32
Link 15 Errors Register	0x403C	R	32
Header Errors Register	0x4040	R	32
Trailer Errors Register	0x4044	R	32
Readout Timeout Errors Register	0x4048	R	32
Header Trailer Limit Errors Register	0x404C	R	32
Row Column Errors Register	0x4050	R	32
L1ID Errors Register	0x4054	R	32
BCID Errors Register	0x4058	R	32

9.13. Formatter

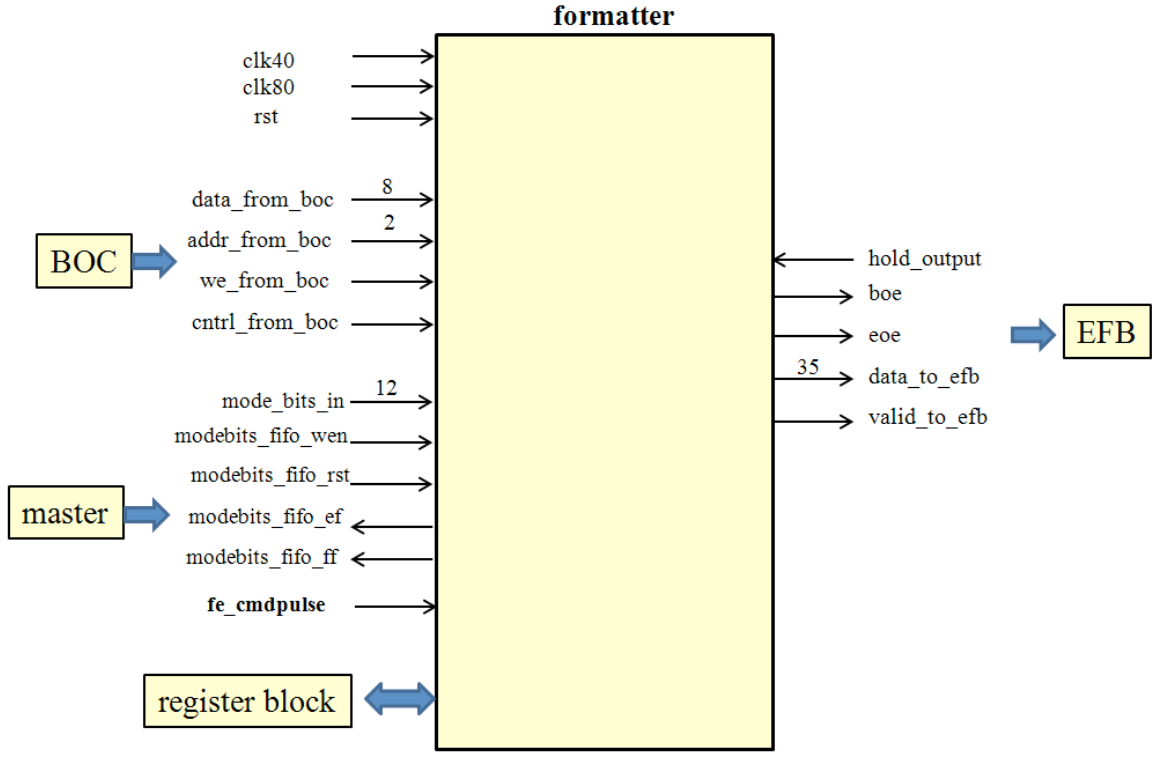
9.13.1. Functionality

The formatter block is the first stage of the data path on the ROD slave. It takes care of the data coming directly through the BOC from up to 4 FeI4 chips on a 12-bit bus. It performs:

- 32-bit word encoding,
- data buffering,
- mode bits data managing,
- 4 queues scheduling in a fixed sequential order (from link 0 to link 3)

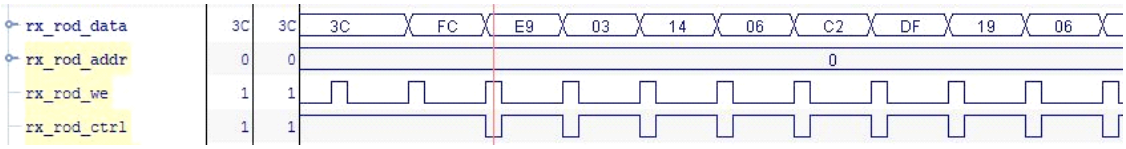
The formatter block extracts 4 streams out of the incoming BOC data for each link channel. A proper formatting is given to the hits (both expanded or condensed modes can be chosen), plus headers, service records and trailers. The formatter block sends one event per link channel in a fixed order starting from link 0 up to link 3 (in case all the 4 links are enabled) after receiving the *fe_cmdpulse* signal from the master (which means that a trigger has been received and the mode bits and the EFB header dynamic bits have been sent to the slave). The block also receives the mode bits from the master for every trigger, which are used to be able to dump events on the fly in case synchronization problems between different links occur. In case multiple frames per trigger are expected from the FeI4, the formatter block will send to the EFB as many frames as required per trigger.

9.13.2. External interfaces



The formatter block external interfaces are four:

- **BOC**: The BOC sends data from 4 FeI4 chips on a 12-bit bus synchronous with the 80 MHz clock (clk80). The 12-bit bus contains an 8-bit data bus, a 2-bit address bus, a write enable and a control signal, as in the following picture:



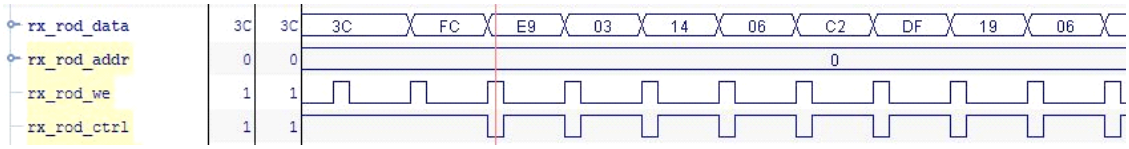
  The incoming data is considered valid when we = '1': in this case if cntrl = '0' then a data is received, otherwise a control word, which is 0xFC as the start of frame and 0xBC as the end of frame. Multiple frames (from 1 to 16) for a single trigger can be expected.

- **Master**: the master sends the *fe_cmdpulse* signal when the mode bits and the EFB header dynamic mask bits have been sent to the slaves. After the formatter receives this signal, a token is generated for reading out one event from the internal buffers.

- **EFB**: Data to the EFB is synchronous with the 80 MHz clock. Output data is valid when *valid_to_efb* = '1'. The *boe* (beginning of event) and *eoe* (end of event) are usually 0, except during the first and last word belonging to an event respectively. When hold_output is 1, no valid data is sent towards the EFB.

- **register block**: The formatter block receives and provides information on the formatter registers as outlined later in this paragraph.

9.13.3. Input data format



9.13.4. Output data format

Name	Bits[31:0]
header	001xxxnnFLLLLLLLLLLLLBBBBBBBBB
hit (long)	100xxxnnTTTTTTTTCCCCCCCRRRRRRRRR
hit (condensed mode) (***not implemented yet***)	101RRRRRTTTTTTTTCCCCCCCRRRRRRRRR 1CCCRRRRRRRRRTTTTTTTTCCCCCCCRRRR 1TTTCCCCCCCRRRRRRRRRTTTTTTTTCCCC 111TTTTTTTTCCCCCCCRRRRRRRRRTTTTT
FE flag error	000xxxnnxSSSSSSxxxxxxxDDDDDDDDDD
trailer	010xxxnnEcPplbzhvMMMMMMMMMMBBBBB

n: link number

F: FeI4B flag bit

L: L1ID

B: BCID

T: ToT

C: hit column

R: row column

S: service code

D: service code counter

c: condensed mode

M: number of skipped triggers on the FE

l/b: L1ID/BCID error
h: header/trailer limit error

E: readout timeout error bit

P: link masked by PPC

p: preamble error (header error)

z: trailer timeout error
v: row/column error

bit 32	TimeOut Error Bit (for all words)
bit 33	Condensed Mode (**not yet implemented**)
bit 34	Link masked by PPC (**not yet implemented**)

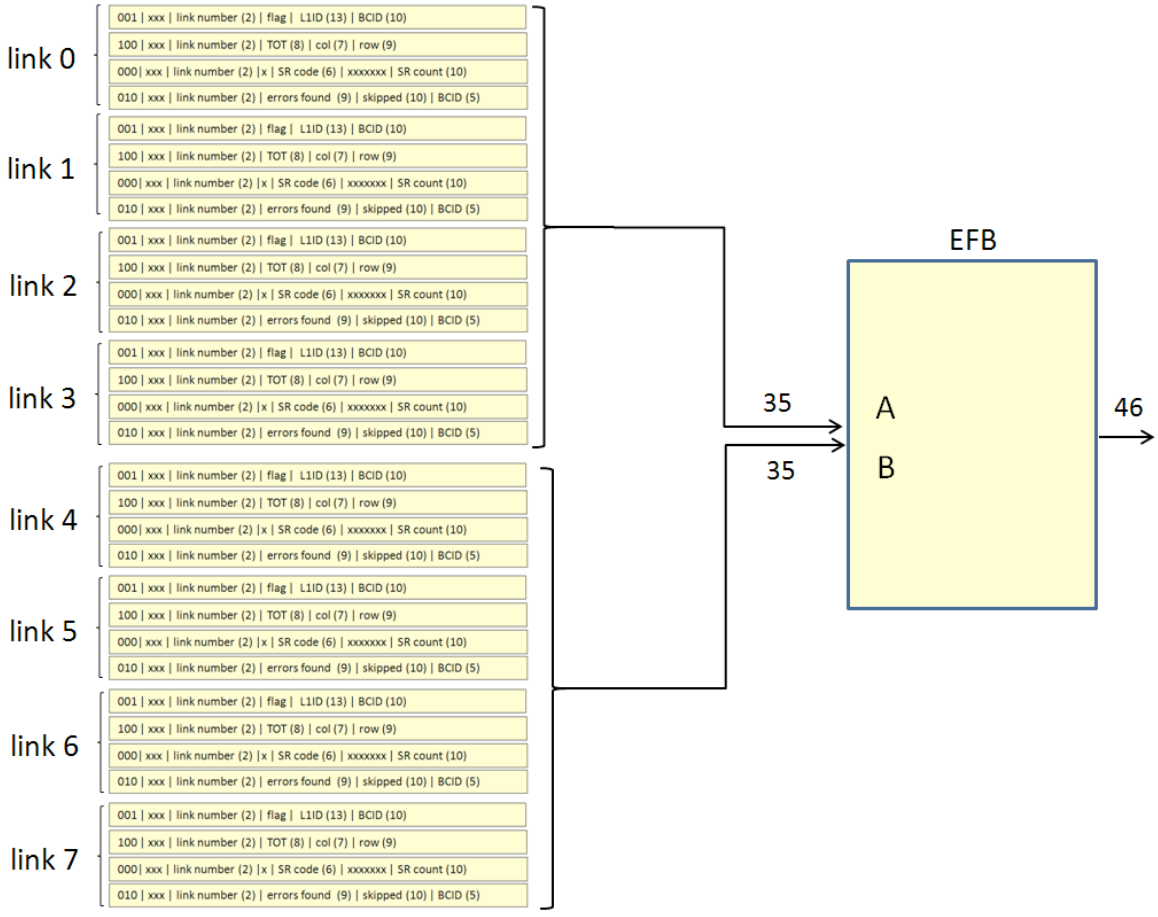


Figure 18: Data packet from two formatter blocks to the EFB in case all 8 modules are enabled (only one hit per event is shown)

When a timeout condition is reached the following event is sent in output for the faulty FE:

Name	Bits[31:0]
header	001xxxnn00000000000101110101101 (0xBAD)
trailer	010xxxnn1cPplbzhv00010111010101 (0xBAD)

with the timeout error bit set and the 0xBAD tag as less significant bits.

9.13.5. Performances

9.13.6. Implementation

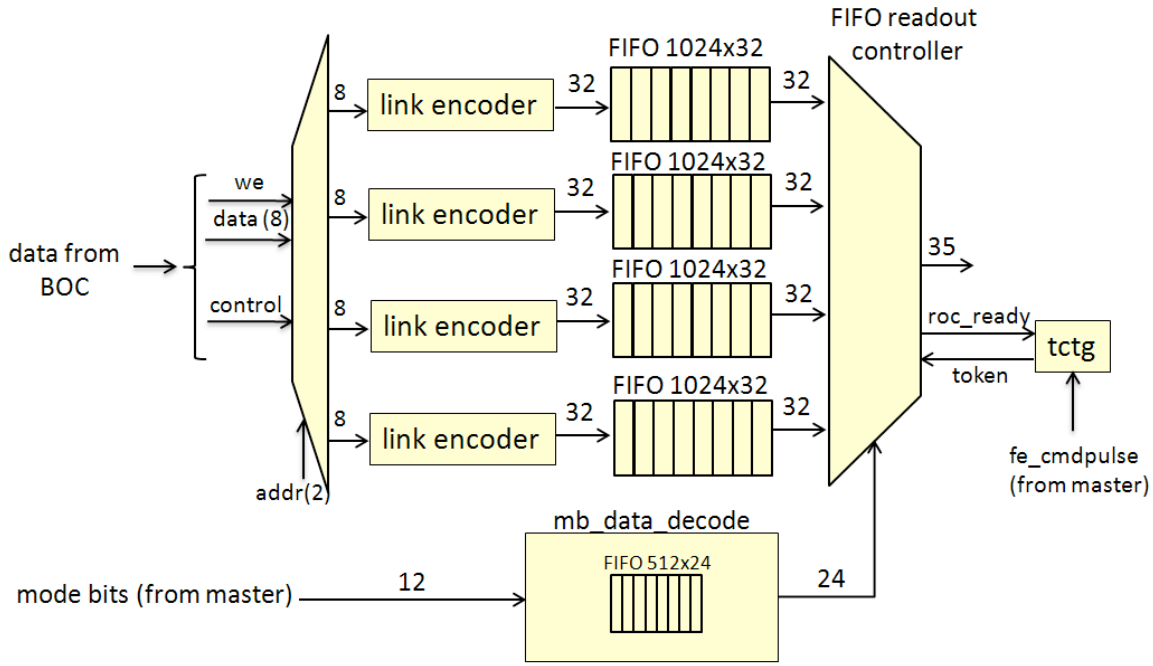


Figure 15: Formatter Schematic Diagram

link encoder

The link encoder blocks takes care of:

- decoding the incoming 12-bit bus from the BOC;
- encoding the 32-bit data words into:
  - header (flag value, BCID and L1ID are received from the Data Header of the FeI4 event)
  - hit
  - FE flag error (service records received inside events are encoded as FE flag errors)
  - trailer

- implement expanded/condensed format,
- insert the appropriate number of events, in case they have not been received from the FE,

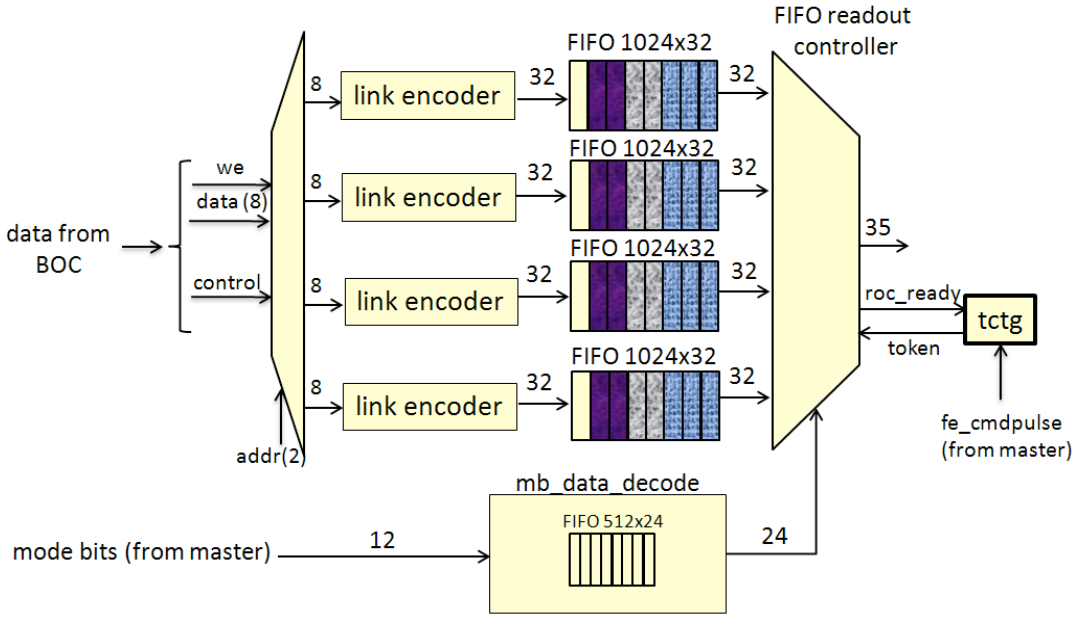
	31 … 0
header	001 | xxx | link number (2) | flag| L1ID (13) | BCID (10)
hit (expanded format)	100 | xxx | link number (2) | TOT (8) | col (7) | row (9)
FE flag error	000| xxx | link number (2) |x | SR code (6) | xx (7) | SR count (10)
trailer	010 | xxx | link number (2) |errors found (9) | skipped (10) | BCID (5)

Figure 16: link_encoder output data format

FIFO readout controller

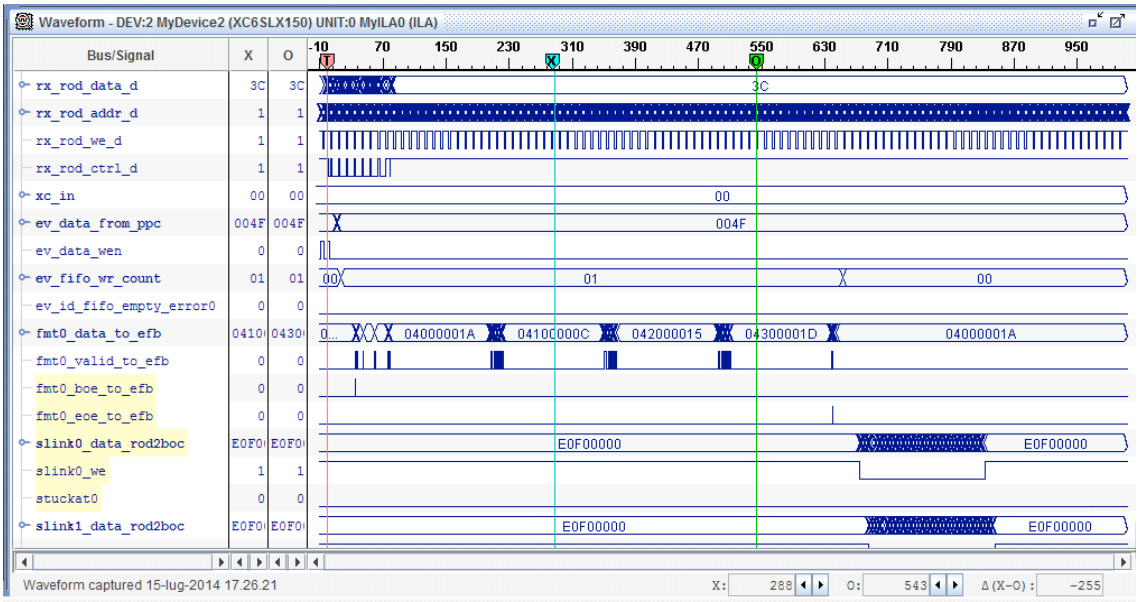
The FIFO readout controller block:

- receives the mode bits from the *mb_data_decode* block, which, in turn, receives the MBs from the master;
- waits for the *token* signal from the *tctg* block;
- reads out as many frames as expected for each L1a per link starting from 0 to 3 (only the enabled links), as in the following (valid if all 4 links are enabled):
  - event# from link0
  - event# from link1
  - event# from link2
  - event# from link3
- deals with the mode bits depending on the values received in order to allow for resynchronization between links (see *doe* and *roe* states in the FIFO readout controller state machine);
- deals with timeout and overflow error conditions.



The FIFO readout controller block can read multiple frames per FIFO, as instructed by the register ***TRIGGER_COUNT_NUMBER*** (address: 0x0007). So if this register's value is 3, the FIFO readout controller reads 3 frames per FIFO per token received.

Caution: the ***TRIGGER_COUNT_NUMBER*** value shall be the same as FeI4 register #2 (bits 15-12) .



If the FeI4 is configured for sending one frame per trigger and ***TRIGGER_COUNT_NUMBER*** = 2, we get one good event plus one bad event (0xBAD) per channel after the timeout period.

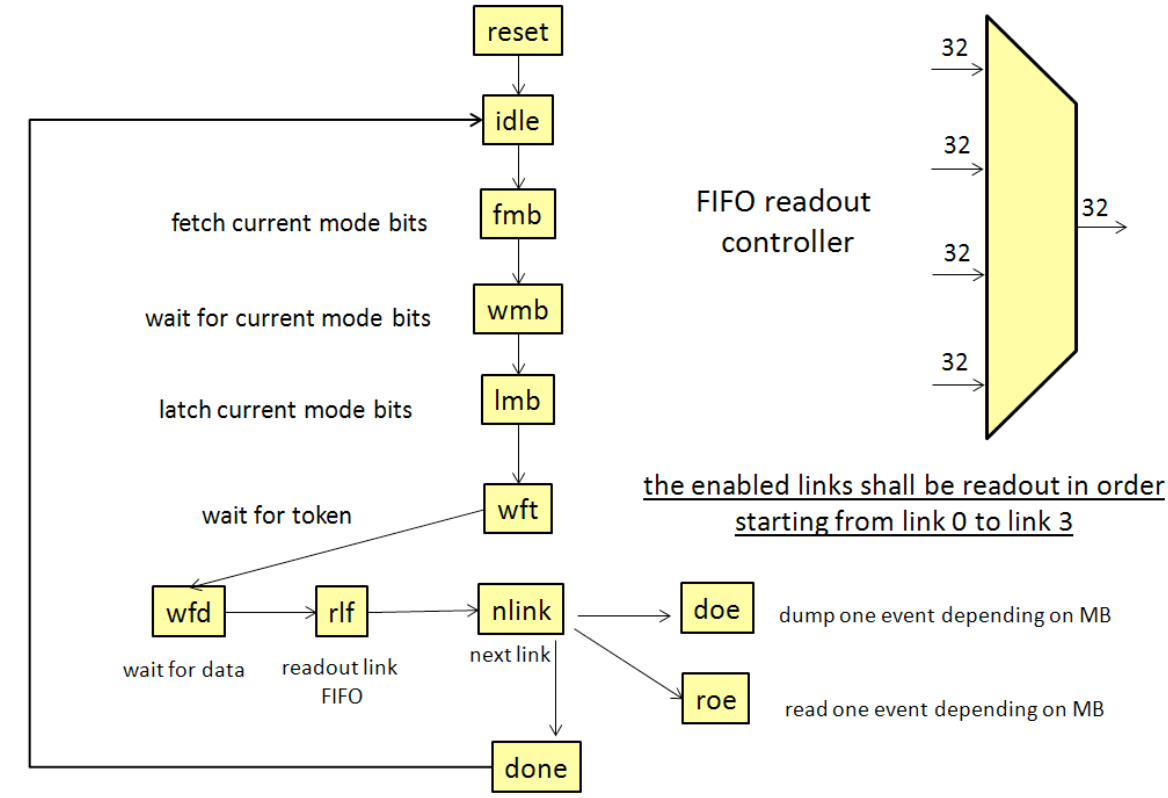


Figure 17: FIFO readout controller state machine

tctg

This block receives the *fe_cmdpulse* signal from the master and increments a trigger counter accordingly. When the trigger counter is greater than 0 and the FIFO readout controller is ready to accept it, a **token** is generated and sent. After this happens, the trigger counter is decremented by 1.

mb_data_decoder

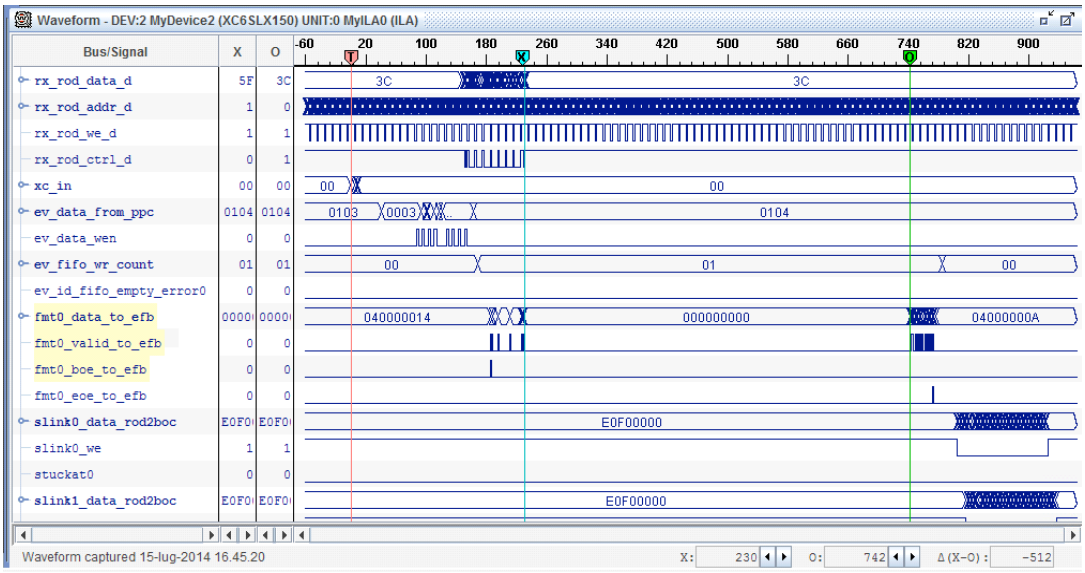
This block receives the mode bits (64 bits for the default set + 64 bits for the corrective set = 128 bits) from the V5 on a 12-bit data bus, thus requiring 11 clock periods for the transfer. The mode bits are temporarily stored on a 512x24 bits FIFO before being used by the FIFO readout controller.

9.13.7. Error handling

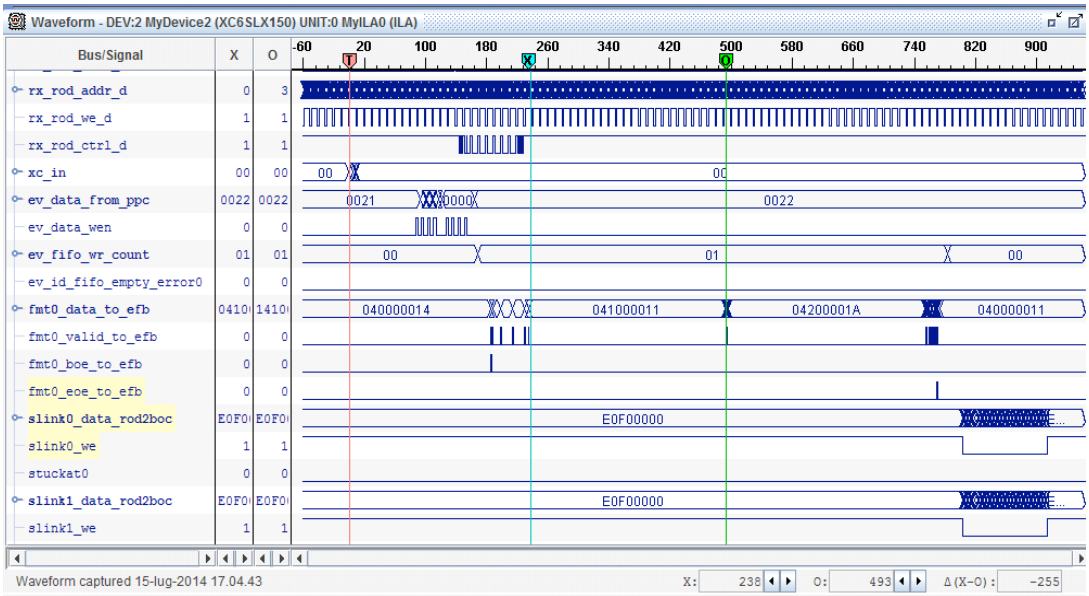
The formatter shall implement the following mechanisms:

- **Readout Timeout Error:** there is a maximum allowed time for a module to transmit event data following a L1 trigger. This value is user configurable in the register_block (its name is ***FMT_READOUT_TIMEOUT_LIMIT***). If a module does not respond with data before the counter in the Formatter reaches the timeout limit, a timeout error event is inserted into the event fragment and the following

link begins to be readout next. The default value at reset for ***FMT_READOUT_TIMEOUT_LIMIT*** on the Pixel ROD is 0x800h (51.2 us). The value for ***FMT_READOUT_TIMEOUT_LIMIT*** used with IBL ROD during calibration runs is 0x1FF (10.25 us measured in 80MHz clock periods).



Timeout example 1: channel 1 is is not sending data



Timeout example 2: channels 1 and 2 are not sending data

☒ **Skipped events:** On the FeI4B chip, when the 16-word deep buffer of pending triggers is full, any additional trigger (due to a new L1A command or to multiple triggers from a multiplier greater than 1) will result in skipped triggers. These triggers are ignored, but the FE-I4B chip keeps track of how many have been ignored with the 10-bit skipped trigger counter. This counter is read out with Service Record 15, which comes out automatically if there are skipped triggers (unless explicitly disabled). The formatter shall extract and save the number of lost

events from the appropriate header and insert the required number of empty events into the data flow. The inserted empty events will contain the correct L1ID and sequential BCID and will contain a flag in the header that will indicate the number of events that were skipped for the current L1 trigger. This process performs the required function of keeping events in synchronization and informs the offline system that an error has occurred on a module. ***To be implemented yet.***

- **Header Error:** When the first 24-bit word received by the link_encoder does not contain the 8-bit MSB header qualifier xE9. This could signify sampling errors from the BOC. The error is marked with x000BAD in the least significant bits and written out to the link FIFO, allowing data to continue to be taken. This error does not currently have a dedicated error flag in the trailer. Lvinfo is not updated as in normal header write.
- **Trailer Timeout Error:** When the hexadecimal trailer xBC is never received by the link_encoder, the trailer error flag is marked in a generated pseudo-trailer after a programmable amount of time, measured in 80MHz clocks. This could signify sampling errors. This error is reported by creation of a pseudo-trailer in the link in place of a trailer coming from the BOC. It is flagged with the trailer error flag set to 1 (flag_errors bit 2: z). The data format is identical to that of a normal trailer, with the exception of the flag being set. The timeout limit can be programmed through the slave register ***FMT_TRAILER_TIMEOUT_LIMIT***, it is 32-bit long and has a default value of x"00000050", which is 1us.
- **Link Occupancy/Rod Busy Limit:** The amount of data stored in the link FIFOs is tracked and stored in registers for debugging and monitoring purposes. Since the FIFOs have a depth of 1024 32-bit words, all occupancy counters are 10 bits long and counters from all 4 links in a Formatter are concatenated together in one slave register in ascending order 3 → 0 (MSB → LSB). The occupancy levels are also compared to the user programmable ROD Busy limit to assert ROD Busy when the FIFOs become close to full. ROD Busy Limit is logical OR'ed with the independent fifo_full signal from the core generated FIFO in order to form a fifo_busy signal for each link FIFO. All 16 fifo_busy signals are concatenated and sent to the ROD BUSY module in each slave. There are four link occupancy registers, one for each of the Formatters in a ROD slave, they are 40-bits wide and are named ***FMT_#_OCCUPANCY_REG***, where # represents the formatter number 0 → 3. The ROD busy limit is stored in the 32-bit programmable register ***FMT_ROD_BUSY_LIMIT*** and has a default value of max 1024.
- **Header-Trailer Limit:** The header-trailer limit allows a cap to be placed on the amount of data allowed to be written into the link FIFO in between a header and a trailer. If the HTL condition is true, the Formatter will change to Header-Trailer Limit mode operation. If the Formatter is currently receiving an event when the HTL condition occurs, the encoder will stop writing data to the FIFO until a trailer is detected and stored in the link FIFO with the HTL error bit set (flag_errors bit 1: h), a bit is also written to the ***FMT_HEADER_TRAILER_ERR*** register, with the bit corresponding to a given link held high while the error is occurring. While similar the key difference between HT limit and Trailer timeout is whether or not a trailer will be detected. Trailer timeout counts clocks, HT Limit counts number of hits. It is possible for both errors to occur in the same frame. The HT Limit can be

programmed in the ROD Slave register file using the 32-bit ***FMT_HEADER_TRAILER_LIMIT*** register.

9.13.8. Formatter: registers

Description	Address	Access	Width
Formatter Link Enable	0x0000	RW	16
Formatter Trigger Count Number	0x0007	RW	4
Formatter Readout Timeout Limit	0x0010	RW	32
Formatter Data Overflow Limit	0x0014	RW	16
Formatter Header Trailer Limit	0x0018	RW	32
Formatter ROD Busy Limit	0x001C	RW	32
Formatter Trailer Timeout Limit	0x001F	RW	32
Formatter Control Register (Reserved)	0x0020	RW	32
Formatter0 Status Register (Reserved)	0x0022	R	32
Formatter1 Status Register (Reserved)	0x0023	R	32
Formatter2 Status Register (Reserved)	0x0024	R	32
Formatter3 Status Register (Reserved)	0x0025	R	32
Formatter0 Occupancy Register	0x0027	R	40
Formatter1 Occupancy Register	0x0029	R	40
Formatter2 Occupancy Register	0x002B	R	40
Formatter3 Occupancy Register	0x002D	R	40
Formatter0 Frames Register	0x0030	R	16
Formatter1 Frames Register	0x0034	R	16
Formatter2 Frames Register	0x0038	R	16
Formatter3 Frames Register	0x003C	R	16
Formatter Readout Timeout Error	0x0070	R	16
Formatter Data Overflow Error	0x0074	R	16
Formatter Header Trailer Error	0x0078	R	16
Formatter ROD Busy Error	0x007A	R	16

9.14. EFB

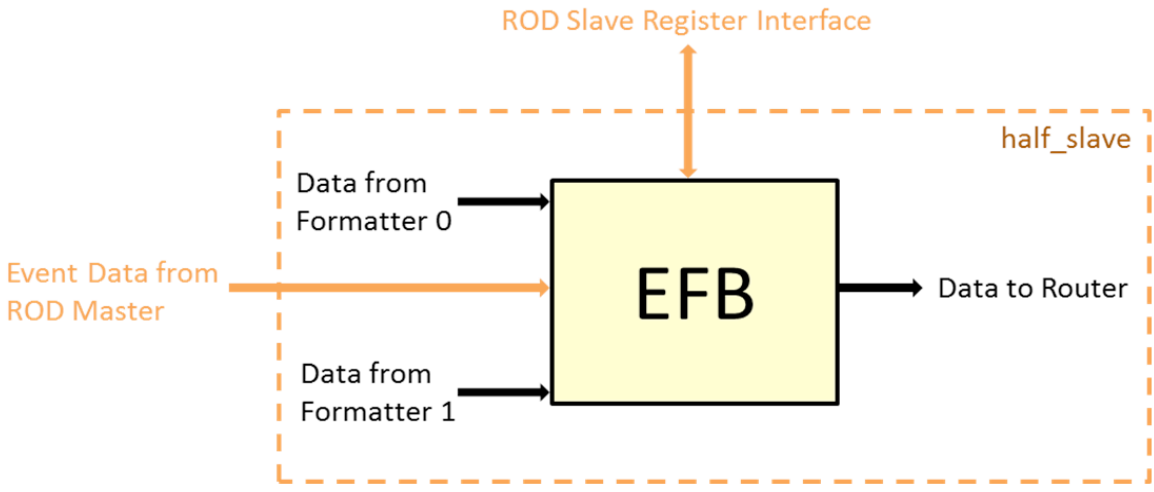
9.14.1. Functionality

Each ROD Slave contains two “half slave” modules and each “half slave” contains one EFB with each EFB capable of processing data coming from two Formatters (eight front-tend chips). The top level group signal interfaces of the EFB are found in the diagram below with details in the following “Interface and Data Format” section.

While FIFOs are heavily used in the synchronization of data inside the EFB, the input stage of the EFB (going into the bc_l1_check from a Formatter) does not have any FIFOs to help synchronize. Interfacing signals between Formatter and EFB are one-way not both ways. If the event data from the Formatter arrives before the Event ID arrives from ROD Master, there will be catastrophic sync error (with no catastrophic fail penalty) since bc_l1_check cannot check ID mismatches and will mark such error, as seen in the S-Link trailer0 (i.e. 0x00030001). I suggest using the event ID FIFO inside the ev_data_decode block inside the EFB as part of the state machine control signal in the Formatter FIFO output controller so Formatter would hold the event data until event ID arrives.

9.14.2. External interfaces

The EFB interfaces with the two Formatters inside the same half_slave module. The ROD Slave register interface signals simple forward the signals with the slv_register block outside of the half_slave module. The event data from ROD Master also forwarded directly to the EFB directly from the ROD Slave top.



Register Interface Signals (ROD Bus)

The register interface signals are listed as follows:

```
err_mask_wen1        : out STD_LOGIC;
err_mask_din1        : out STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
err_mask_dout1        : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_sel1         : out STD_LOGIC_VECTOR( 2 DOWNTO 0);
err_mask_wen2         : out STD_LOGIC;
err_mask_din2         : out STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_dout2        : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_sel2         : out STD_LOGIC_VECTOR( 2 DOWNTO 0);
err_mask_wen3         : out STD_LOGIC;
err_mask_din3         : out STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_dout3        : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_sel3         : out STD_LOGIC_VECTOR( 2 DOWNTO 0);
err_mask_wen4         : out STD_LOGIC;
err_mask_din4         : out STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_dout4        : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
err_mask_sel4         : out STD_LOGIC_VECTOR( 2 DOWNTO 0);
-- Information going into efb gen_fragment block
source_id             : out STD_LOGIC_VECTOR(31 DOWNTO 0);
format_version        : out STD_LOGIC_VECTOR(31 DOWNTO 0);
run_number            : out STD_LOGIC_VECTOR(31 DOWNTO 0);
-- output evtmem fifos and EFB signals
efb1_out_fifo_rst     : out STD_LOGIC;
efb1_out_fifo1_empty_n  : in  STD_LOGIC;
efb1_out_fifo1_aempty_n : in  STD_LOGIC;
efb1_out_fifo1_full_n   : in  STD_LOGIC;
efb1_out_fifo1_afull_n  : in  STD_LOGIC;
efb1_out_fifo1_play_done : in  STD_LOGIC;
efb1_out_fifo2_empty_n  : in  STD_LOGIC;
efb1_out_fifo2_aempty_n : in  STD_LOGIC;
efb1_out_fifo2_full_n   : in  STD_LOGIC;
efb1_out_fifo2_afull_n  : in  STD_LOGIC;
efb1_out_fifo2_play_done : in  STD_LOGIC;
efb2_out_fifo_rst     : out STD_LOGIC;
efb2_out_fifo1_empty_n  : in  STD_LOGIC;
efb2_out_fifo1_aempty_n : in  STD_LOGIC;
efb2_out_fifo1_full_n   : in  STD_LOGIC;
efb2_out_fifo1_afull_n  : in  STD_LOGIC;
efb2_out_fifo1_play_done : in  STD_LOGIC;
efb2_out_fifo2_empty_n  : in  STD_LOGIC;
```

```
                efb2_out_fifo2_aempty_n  : in  STD_LOGIC;
                efb2_out_fifo2_full_n    : in  STD_LOGIC;
                efb2_out_fifo2_afull_n   : in  STD_LOGIC;
                efb2_out_fifo2_play_done : in  STD_LOGIC;
                efb1_misc_stts_reg      : in  STD_LOGIC_VECTOR(30 DOWNTO 0);
                efb2_misc_stts_reg      : in  STD_LOGIC_VECTOR(30 DOWNTO 0);
                efb1_ev_header_data     : in  STD_LOGIC_VECTOR(15 DOWNTO 0);
                efb2_ev_header_data     : in  STD_LOGIC_VECTOR(15 DOWNTO 0);
                efb1_ev_fifo_data1      : in  STD_LOGIC_VECTOR(26 DOWNTO 0);
                efb1_ev_fifo_data2      : in  STD_LOGIC_VECTOR(26 DOWNTO 0);
                efb2_ev_fifo_data1      : in  STD_LOGIC_VECTOR(26 DOWNTO 0);
                efb2_ev_fifo_data2      : in  STD_LOGIC_VECTOR(26 DOWNTO 0);
                efb_mask_bcid_error     : out STD_LOGIC;
                efb_mask_l1id_error     : out STD_LOGIC;
                efb_bcid_offset         : out STD_LOGIC_VECTOR( 9 DOWNTO 0);
                efb_grp_evt_enable      : out STD_LOGIC_VECTOR( 1 downto 0);
                efb1_event_count        : in  STD_LOGIC_VECTOR(31 downto 0);
                efb2_event_count        : in  STD_LOGIC_VECTOR(31 downto 0);
                mask_boc_clock_err      : out STD_LOGIC;
                mask_tim_clock_err      : out STD_LOGIC;
                efb1_l1id_fifo_data                 : in  std_logic_vector(39 downto 0);
                efb1_l1id_fifo_addr                 : out std_logic_vector( 3 downto 0);
                efb2_l1id_fifo_data           : in  std_logic_vector(39 downto 0);
                efb2_l1id_fifo_addr                 : out std_logic_vector( 3 downto 0);
                efb_enable_l1_trap      : out STD_LOGIC_VECTOR( 1 downto 0);
                efb_l1_trap_full_in     : in  STD_LOGIC_VECTOR( 1 downto 0);
                efb1_latched_l1id0      : in  STD_LOGIC_VECTOR( 4 DOWNTO 0);
                efb1_latched_l1id1      : in  STD_LOGIC_VECTOR( 4 DOWNTO 0);
                efb1_latched_bcid0      : in  STD_LOGIC_VECTOR( 9 DOWNTO 0);
                efb1_latched_bcid1      : in  STD_LOGIC_VECTOR( 9 DOWNTO 0);
                efb2_latched_l1id0      : in  STD_LOGIC_VECTOR( 4 DOWNTO 0);
                efb2_latched_l1id1      : in  STD_LOGIC_VECTOR( 4 DOWNTO 0);
                efb2_latched_bcid0      : in  STD_LOGIC_VECTOR( 9 DOWNTO 0);
                efb2_latched_bcid1      : in  STD_LOGIC_VECTOR( 9 DOWNTO
0);
                tim_bcid_in_rol         : out STD_LOGIC;
                bcid_rollover_select    : out STD_LOGIC
```

For the complete register table with address values and read/write contents, refer to Table X.

Event ID from ROD Master

```
ev_data_in          : in  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
ev_data_wen_n       : in  STD_LOGIC ;
ev_data_almost_full_n : out STD_LOGIC ;
ev_data_empty       : out STD_LOGIC ;
header_ev_id_data   : out STD_LOGIC_VECTOR(63 DOWNTO 0) ; -- first 4 words concatenated as header
header_ev_id_wen    : out STD_LOGIC ;
ev_id_data1         : out STD_LOGIC_VECTOR(26 DOWNTO 0) ;
ev_id_wen1          : out STD_LOGIC ;
ev_id_data2         : out STD_LOGIC_VECTOR(26 DOWNTO 0) ;
ev_id_wen2          : out STD_LOGIC ;
ev_fifo1_af         : in  STD_LOGIC ; -- header_ev_id_almost_full
ev_fifo2_af         : in  STD_LOGIC ; -- ev_id_almost_full1,
ev_fifo3_af         : in  STD_LOGIC ; -- ev_id_almost_full2,
bcid_offset         : in  STD_LOGIC_VECTOR(9 DOWNTO 0) ;  -- this will need to change
bcid_rollover_select  : in  STD_LOGIC;
        -- new addition, used for distinguishing dynamic mask bit selection for EFBs on different slave
        mask_sel            : in STD_LOGIC_VECTOR(1 DOWNTO 0)
```

Formatter Signals

```
data_in1            : in  STD_LOGIC_VECTOR(34 DOWNTO 0) ;---
data_in2            : in  STD_LOGIC_VECTOR(34 DOWNTO 0) ;---
        fmt_data_valid_in1    : in  STD_LOGIC ;
        fmt_data_valid_in2    : in  STD_LOGIC ;
fmt_beg_of_event1     : in STD_LOGIC ;
fmt_beg_of_event2     : in STD_LOGIC ;
fmt_end_of_event1     : in STD_LOGIC ;
fmt_end_of_event2     : in STD_LOGIC ;
```

Router Signals

```
data_valid_out        : out STD_LOGIC ;
out_data_to_router    : out STD_LOGIC_VECTOR(31 DOWNTO 0) ;
        data_chip_id        : out STD_LOGIC_VECTOR( 2 DOWNTO 0) -- for the histogrammer
```

9.14.3. EFB input data format

INPUT DATA FORMAT: data_in[34:0] (from Formatter)	
data_in[31:0]	event data
- header	=> [001| xxx | link#(2) | flag | L1ID(13) | BCID(10)]
- hit	=> [100 | xxx | link#(2) | ToT(8) | col(7) | row(9)]
- error_flags	=> [000 | xxx | link#(2) | x | SR code(6) | xxxxxxx | SR count(10)]
- trailer	=> [010 | xxx | link#(2) | errors found(9) | skipped (10) | BCID(5)]
data_in[32]	timeout error
data_in[33]	condensed mode
data_in[34]	link masked by PPC

Event ID from the master to the slaves

Event ID from the V5
word 0 : L1 ID[15:0]
word 1 : ECR ID[7:0] & L1 ID[23:16]
word 2 : BC ID[11:0] & RoL & BOC OK & TIM OK
word 3 : TT - ROD[5:0] & TIM[1:0] & ATLAS[7:0]
word 4 : Dynamic Mask 0 for Links [7: 0]
word 5 : Dynamic Mask 1 for Links [15: 8]
word 6 : Dynamic Mask 2 for Links [23:16]
word 7 : Dynamic Mask 3 for Links [31:24]

9.14.4. EFB output data format

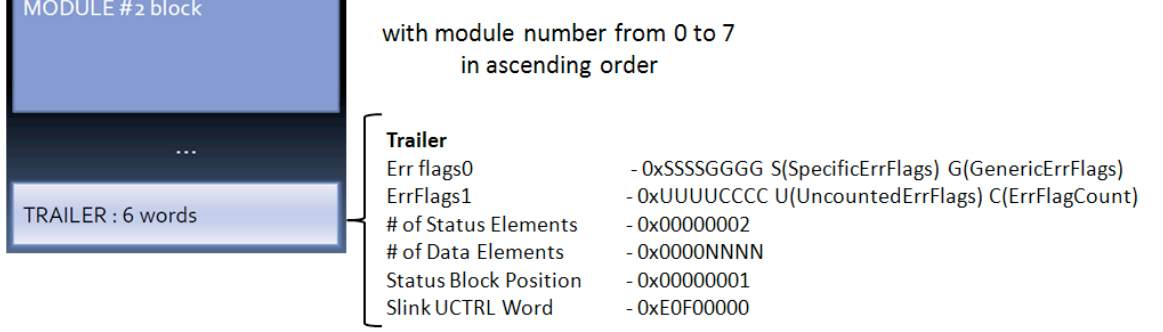


Figure 21: Fragment Format

D(SubdetectorID) = 0x14, M(ModuleID)(1-0)=ROL ID (2 bits)

MODULE ## block	data_out[31:0] (to the router)
header	001nnnnnFLLLLLLLLLLLLLBBBBBBBBB
hit (long)	100nnnnnTTTTTTTTCCCCCCCRRRRRRRR
hit (condensed mode)	101RRRRRTTTTTTTCCCCCCCRRRRRRRR 1CCCRRRRRRRRTTTTTTTCCCCCCCRRR 1TTTCCCCCCCRRRRRRRRTTTTTTTCCCC 111TTTTTTTCCCCCCCRRRRRRRRTTTTT
FE flag error	000nnnnnxSSSSSSxxxxxxxDDDDDDDDD
trailer	010nnnnnEcPplbzhvMMMMMMMMMBBBBB

n: link number
F: FeI4B flag bit
L: L1ID
B: BCID
T: hit ToT
C: hit column
R: row column
S: service code
D: service code counter
E: readout timeout error bit
c: condensed mode
P: link masked by PPC
p: preamble/header error
l/b: L1ID/BCID error
z: trailer timeout error
h: header/trailer limit error
v: row/column error
M: skipped trigger counter

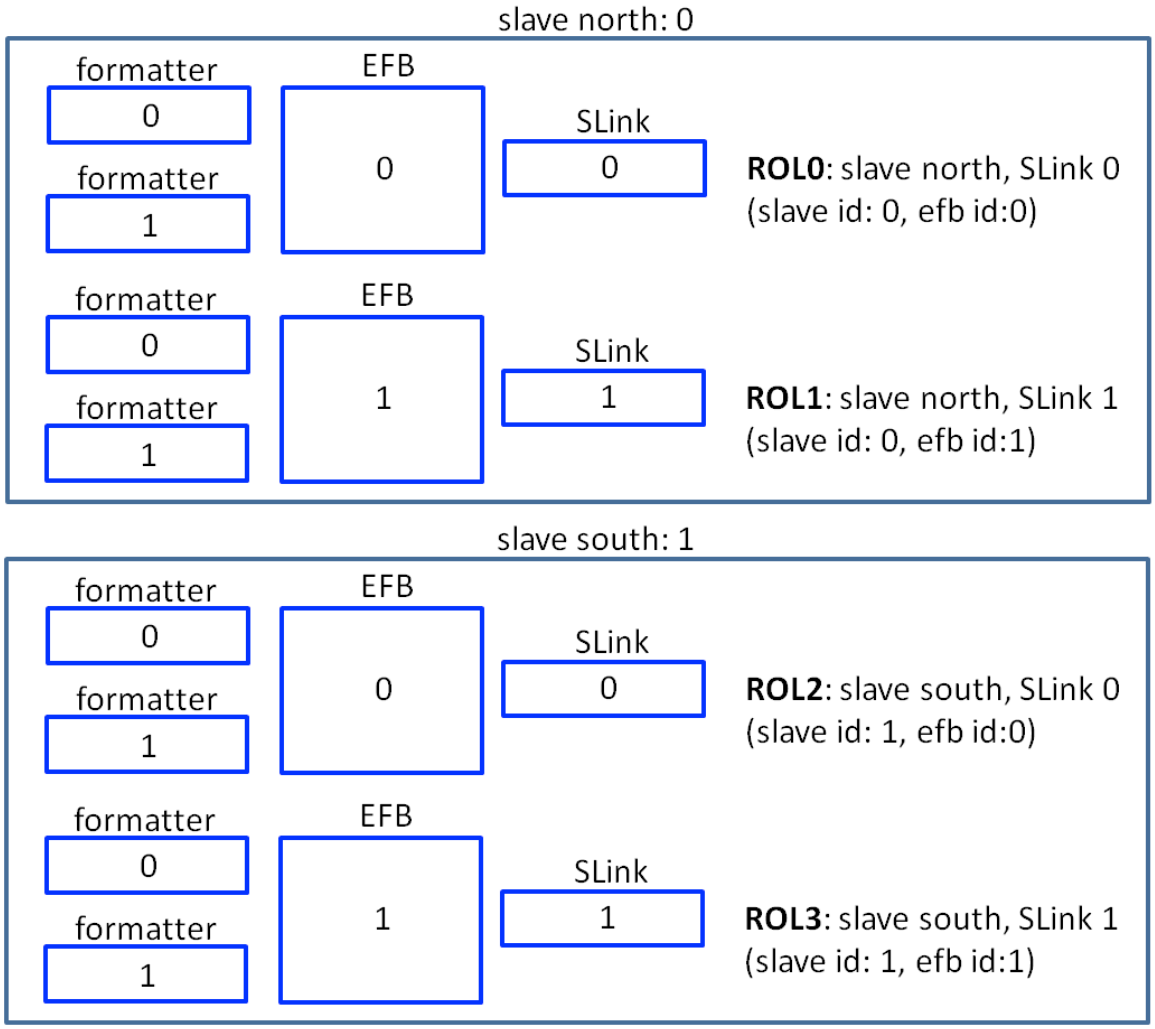
nnnnn: slave ID (1b) – efb ID (1b) – formatter ID (1b) – link ID (2b)

where:

slave ID: 0 for slave north, 1 for slave south

efb ID: 0 for first EFB, 1 for the second EFB

formatter ID: 0 for the first formatter, 1 for the second formatter (as related to one EFB)



9.14.5. Performances

9.14.6. Implementation

The implementation of the EFB used many FIFOs to synchronize between incoming decoded event IDs, data packets, flagged error words, and output S-Link data generation. The top level diagram of the can be found below:

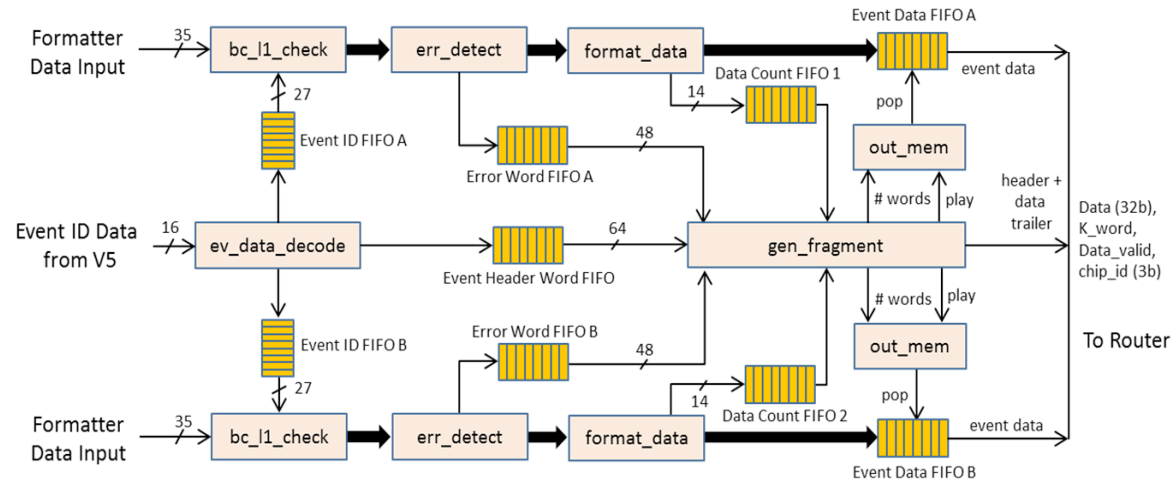


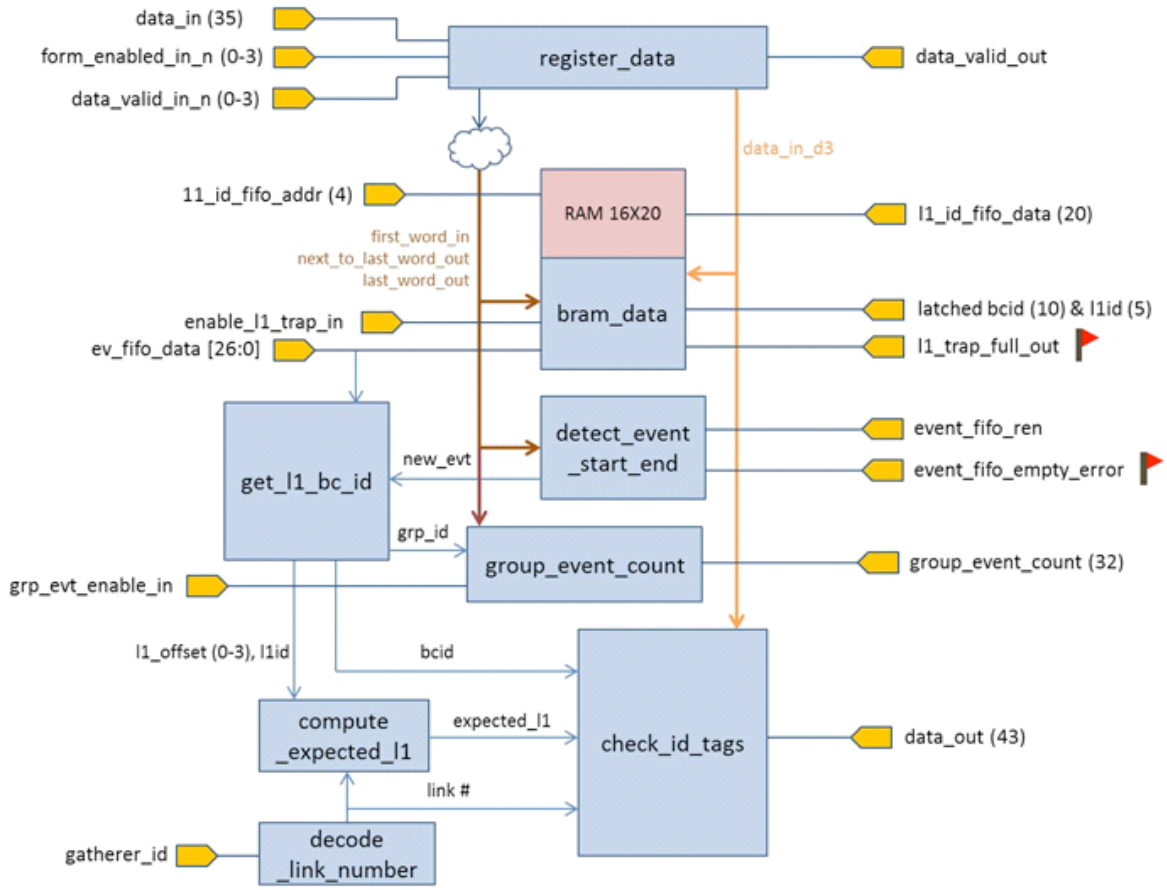
Figure 19: EFB schematic diagram

It should be noted that there are two clock sources used in the EFB. While most of the blocks utilize the 80MHz generated by the DCM (digital clock manager at top level that takes in differential 40 MHz signal and regenerated to 80 MHz), the block with direct interfacing to the ROD Master (ev_data_decode for receiving event ID and EFB link dynamic mask bits) and the register block (err_detect for configuring link error masking) use the 40MHz clock. One should find FIFOs or block RAMs in these blocks with dual clock operations.

At the final output of the EFB to the router, a MUX is used to switch between gen_fragment block (S-Link headers + trailers), Event Data FIFO A (stored data from frontend link i to i+3), and Event Data FIFO B (stored data from frond end link i+4 to i+7). The other two outputs are data valid (also MUXed), and chip_id (3 bits). The data_chip_i is retrieved from the data being output from the MUX and sent to the Router block so to simplify the task for the Router to forward histogrammer data.

bc_l1_check

This block receives data from formatter, decodes link number, calculates L1ID offset, resynchronizes the four FE data links (individually) with dynamic masked bits, checks errors (BCID, L1ID, non-sequential FE order), provides an event group counter, flags the end of an event, and allows single event ID trapping.



event_data_decode

This block receives eight 16-bit event ID data from the V5 and assigns corresponding event ID and link dynamic masked bits to each **bc_l1_check** block. It allows BCID offset calculation with roll over select option and generates event header words for the **gen_fragment** block to generate headers at the output of the EFB.

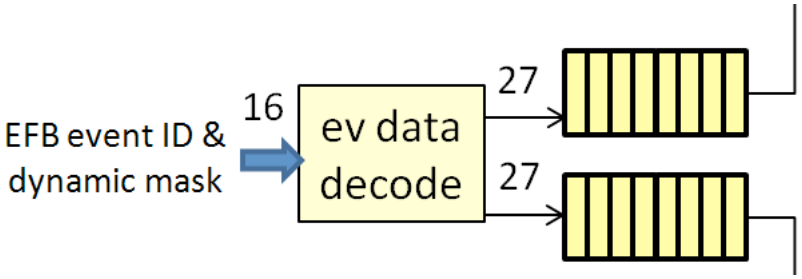
EVENT ID FIFO INPUT DATA FORMAT: ev_id_data1[26:0]

EVENT ID FIFO INPUT DATA FORMAT: ev_id_data1[26:0]
ev_fifo_data1[9:0] BCID(10) ev_fifo_data1[14:10] L1ID(5) ev_fifo_data1[18:15] ROD Trigger Type(4) ev_fifo_data1[26:19] Dynamic Mask # (8) supports links 0 to 3 with 2-bits each - L1 Offset Link 0 <= ev_fifo_data (20 downto 19) - L1 Offset Link 1 <= ev_fifo_data (22 downto 21) - L1 Offset Link 2 <= ev_fifo_data (24 downto 23) - L1 Offset Link 3 <= ev_fifo_data (26 downto 25)

EVENT ID FIFO INPUT DATA FORMAT: ev_id_data2[26:0]

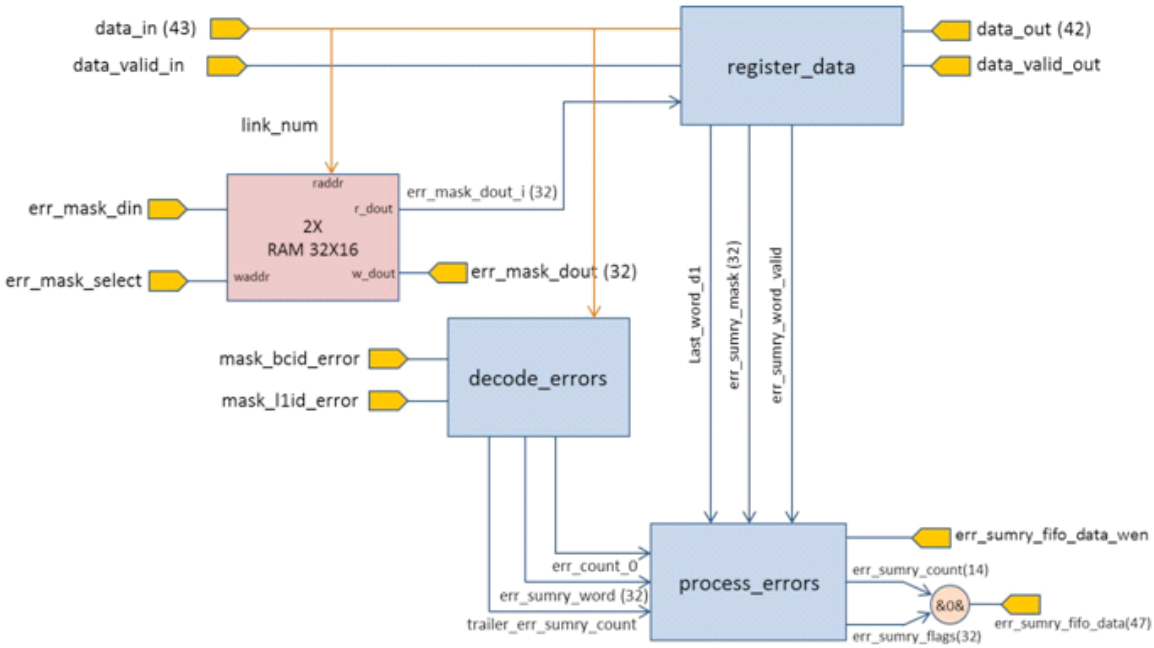
EVENT ID FIFO INPUT DATA FORMAT: ev_id_data2[26:0]

ev_fifo_data1[9:0] BCID(10)
ev_fifo_data1[14:10] L1ID(5)
ev_fifo_data1[18:15] ROD Trigger Type(4)
ev_fifo_data2[26:19] Dynamic Mask # (8) supports links 4 to 7 with 2-bits each
- L1 Offset Link 4 <= ev_fifo_data (20 downto 19)
- L1 Offset Link 5 <= ev_fifo_data (22 downto 21)
- L1 Offset Link 6 <= ev_fifo_data (24 downto 23)
- L1 Offset Link 7 <= ev_fifo_data (26 downto 25)



igure 20: EFB event ID distribution

err_detect_new



format_data

This is a simple data forwarding block that keeps count of how many data words there are in a given event. This block receives the 42 bit incoming data (see table below) and valid signal and writes to the EFB event data output FIFO (only write the lower 32 bit). As the

data words are passing through the blocks, a counter keeps count of how many data words have been written to the FIFO. As data_in[41] goes high, which is the end of event fragment marked previously by bc_l1_check block, the count of words is written to the data count FIFO (for later to determine how many words should be played to the S-Link) and the counter is then reset.

data_in[31:0] event data

data_out[35:32] link address

data_in[36] timeout error

data_in[37] condensed mode

data_in [38] L1ID error

data_in[39] BCID error

data_in[40] Link masked by PPC

data_in[41] end of event fragment

out_mem

This is a FIFO readout controller with delicate timing. The functionality of this block is simply: "receive how many words to pop out of the event data FIFO, play those many words when prompted by gen_fragment, and signals play done when done playing". Since each out_mem block is responsible for 4 formatter frontend links, if all of the 4 links are disabled, the out_mem block will then never receive any play commands from the gen_fragment block.

gen_fragment

This block takes in the ATLAS event information (trigger type, event type, run number, detector module ID), as well as the collected data words from the front end chips, and organize them into a 33 bit S-Link data packet (K word + 32b data word). This block coordinates and interfaces with the FIFOs holding processed event data, event ID, data word counts, error summary words, and the out_mem (output FIFO controller block) to piece together S-Link data in the timely order of:

- **headers** (beginning-of-frame word and ATLAS event information)
- **data from Event Data FIFO A** (frontend data provided by Formatter 0)
- **data from Event Data FIFO B** (frontend data provided by Formatter 1)
- **trailers** (error summary, size of event, end-of-frame word)

Comprehensive data format and bit definitions are described in the previous "Router Signals" section.

Begin

State = idle

If(event header word FIFO is NOT empty)

Check which links are enabled (link_en are grouped by 4 links)

If (something is in error word FIFO and data count FIFO)

Read and latch event header

Read and latch data count

Read and latch error word

State -> wait for event data

State = wait for event data

Wait for 2 clock cycles for event to latch from FIFO

State -> test event type

State = test event type

Check and mark if there are any errors in event (error count > 0)

State -> send event fragment

State = send event fragment

If (no S-Link backpressure) – currently not connected

State -> header0

State = header0 -> header9

Send headers

If(readout link test enabled)

State -> rol test data

Else if (lower 4 links enabled)

State -> out mem 1

Else if (NOT lower 4 links enabled AND upper 4 links enabled)

State -> out mem 2

State = rol test data

Generate S-Link test data

State -> trailer0-5

State = out_mem1

Ask out_mem 1 to play #latched word count (from FIFO1)

If(upper 4 links enabled AND play done)

State -> out_mem2

Else

State -> trailer0-5

State = out_mem2

Ask out_mem 2 to play #latched word count (from FIFO1)

If(play done)

State -> trailer0-5

State = trailer0 -> trailer5

Send trailers 0-5

State -> idle

End

9.14.7. Error handling

9.14.8. EFB registers

Description	Address	Access	Width
Slave0, Half-Slave 0 Error Masks			
Link 0	0x2000		
Link 1	0x2004		
Link 2	0x2008	RW	32
Link 3	0x200C		
Link 4	0x2010		
Link 5	0x2014		
Link 6	0x2018		
Link 7	0x201C		
Slave 0, Half-Slave 1 Error Masks			
Link 8	0x2020		
Link 9	0x2024		
Link 10	0x2028	RW	32
Link 11	0x202C		
Link 12	0x2030		
Link 13	0x2034		
Link 14	0x2038		
Link 15	0x203C		
Slave 1, Half-Slave 0 Error Masks			
Link 16	0x2100		
Link 17	0x2104		
Link 18	0x2108	RW	32
Link 19	0x210C		
Link 20	0x2110		
Link 21	0x2114		
Link 22	0x2118		
Link 23	0x211C		
Slave 1, Half-Slave 1 Error Masks			
Link 24	0x2120		
Link 25	0x2124		
Link 26	0x2128	RW	32
Link 27	0x212C		
Link 28	0x2130		
Link 29	0x2134		
Link 30	0x2138		
Link 31	0x213C		
EFB Format Version	0x2200	R	32
0x0301UUUU U = user defined, top half fixed		RW	16
EFB Source ID	0x2204	R	32
0xUU1SMMMM S = Sub Detector ID, M = Module ID		RW	23

EFB Run Number	0x2208	RW	32
EFB1 Command Register Bit[0]: 0 (send empty events) Bit[1]: Mask BCID Error Bit[2]: Group Event Counter Enable Bit[3]: Mask L1ID Error Bit[4:7]: 0 (link input test select) Bit[8]: Mask TIM Clock Error Bit[9]: Mask BOC Clock Error Bit[10]: 0 (mask sweeper error) Bit[11]: Enable L1 and BC ID Trap Bit[12]: TIM BCID Readout Link Enable Bit[13]: BCID Rollover Select Bit[14:15]: 0 Bit[16:25]: BCID Offset	0x2210	RW	32
EFB2 Command Register See above description (should revise)	0x2212	RW	32
EFB1 Run-Time Status Register Bit[0]: FIFO1 "almost_full_n" Status Flag Bit[1]: err_sumry_fifo1_almost_full Bit[2]: ev_id_fifo_empty_error1 Bit[3]: fifo 1 pause to Formatter Bit[4]: FIFO2 "almost_full_n" Status Flag Bit[5]: err_sumry_fifo2_almost_full Bit[6]: ev_id_fifo_empty_error2 Bit[7]: fifo 2 pause to Formatter Bit[8]: halt output from router Bit[9]: rod_event_type_empty Bit[10]: rod_event_type_afull Bit[11]: rod_event_type_full	0x2218	R	12
EFB2 Run-Time Status Register See above description	0x221A	R	32
ROD Code Version and Board Version Bit[0:7]: HDL Code Version Bit[8]: 0 Bit[9:15]: 00 & Board Revision Number	0x221C	R	16
EFB1 Event Header Data	0x2220	R	16
EFB2 Event Header Data	0x2222	R	16
EFB1 Group Event Count	0x2230	R	32
EFB2 Group Event Count	0x2232	R	32
EFB1 Out FIFOs Reset	0x2248	W	1
EFB2 Out FIFOs Reset	0x224A	W	1
EFB1 Out FIFO Status Flags Bit[0]: FIFO1 NOT "empty_n" Status Flag Bit[1]: FIFO1 NOT "almost_empty_n" Status Flag Bit[2]: FIFO1 NOT "full_n" Status Flag Bit[3]: FIFO1 NOT "almost_full_n" Status Flag Bit[4]: FIFO2 NOT "empty_n" Status Flag Bit[5]: FIFO2 NOT "almost_empty_n" Status Flag Bit[6]: FIFO2 NOT "full_n" Status Flag Bit[7]: FIFO2 NOT "almost_full_n" Status Flag Bit[8]: 0 (HTFIFO "empty" Status Flag) Bit[9]: 0 (HTFIFO "full" Status Flag) Bit[10]: ev_id_empty1 Bit[11]: ev_id_almost_full1 Bit[12]: ev_id_empty2 Bit[13]: ev_id_almost_full2 Bit[14]: header_ev_id_empty Bit[15]: header_ev_id_almost_full Bit[16]: header_ev_id_full Bit[17]: ev_data_empty Bit[18]: ev_id_full1 Bit[19]: ev_id_full2	0x224C	R	24

Bit[20]: count_fifo1_full Bit[21]: count_fifo2_full Bit[22]: err_sumry_fifo1_full Bit[24]: err_sumry_fifo2_full			
EFB2 Out FIFO Status Flags See above description	0x224E	R	32
EFB1 L1/BCID Trapped Values Bit[15:0]: latched_bcid0 & 0 & latched_l1id0 Bit[31:16]: latched_bcid1 & 0 & latched_l1id1	0x227C	R	32
EFB2 L1/BCID Trapped Values See above description	0x227D	R	32
EFB1 Miscellaneous Status Register Bit[0]: err_sumry_fifo1_almost_full Bit[1]: err_sumry_fifo2_almost_full Bit[2]: ev_id_fifo_empty_error1 Bit[3]: ev_id_fifo_empty_error2 Bit[4]: ev_id_empty1 Bit[5]: ev_id_empty2 Bit[6]: header_ev_id_empty Bit[7]: fifo1_pause_o Bit[8]: fifo2_pause_o Bit[9]: gatherer_halt_output Bit[10:17]: 0 Bit[18]: header_ev_id_data_out(35) Bit[19]: ev_id_almost_full1 Bit[20]: ev_id_almost_full2 Bit[21]: header_ev_id_almost_full Bit[22]: ev_data_almost_full_n_o Bit[23]: header_ev_id_full Bit[24]: ev_id_full1 Bit[25]: ev_id_full2 Bit[26]: ev_data_empty_o Bit[27]: err_sumry_fifo1_full Bit[28]: err_sumry_fifo2_full Bit[29]: count_fifo1_full Bit[30]: count_fifo2_full	0x227E	R	32
EFB2 Miscellaneous Status Register See above description	0x227F	R	32

9.15. Router

9.15.1. Functionality

The router block performs the task of receiving event fragments from the EFB and routing data towards the Slink lines (→ BOC) and towards the related histogrammer block. During calibration runs event fragments are routed just towards the histogrammer block, while during normal data taking event fragments are routed both to the Slink and the histogrammer blocks. The router expects to receive data from the EFB already formatted as packets with Slink header, module data and Slink trailer: no further data formatting/processing in the router block is expected. In case condensed mode is used, the router shall be able to decode the hits (value + address) and send them to the histogrammer one per clock cycle. The router also creates histograms of the error bits found in the trailer; histograms of both individual errors (row/coloumn errors) as well as for each link in the ROD Slave.

Incoming data are stored in an internal FIFO before being read out and distributed to the outputs. In case the FIFO occupancy becomes larger than a programmable value, a busy signal is asserted, in order to stop the generation of triggers.

9.15.2. External interfaces

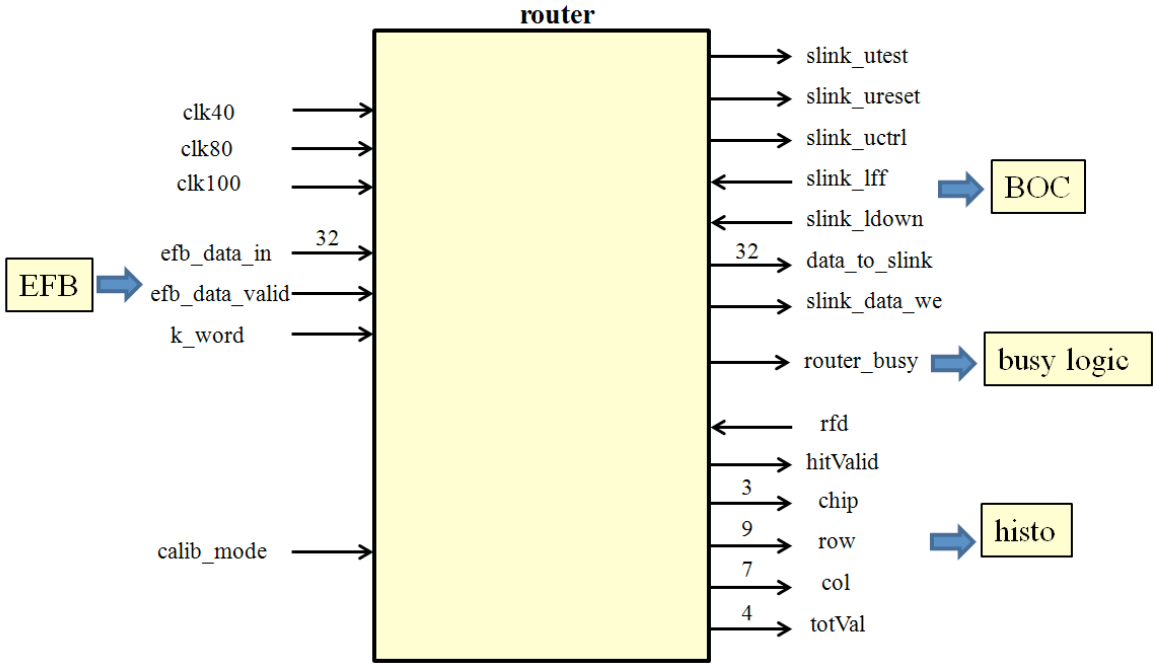


Figure 2: Router block I/O interfaces

The router block external interfaces are four:

- **EFB**: Event fragments come from the EFB on a 32-bit bus (*efb_data_in*) synchronous with the 80 MHz clock (*clk80*). An incoming data is considered valid

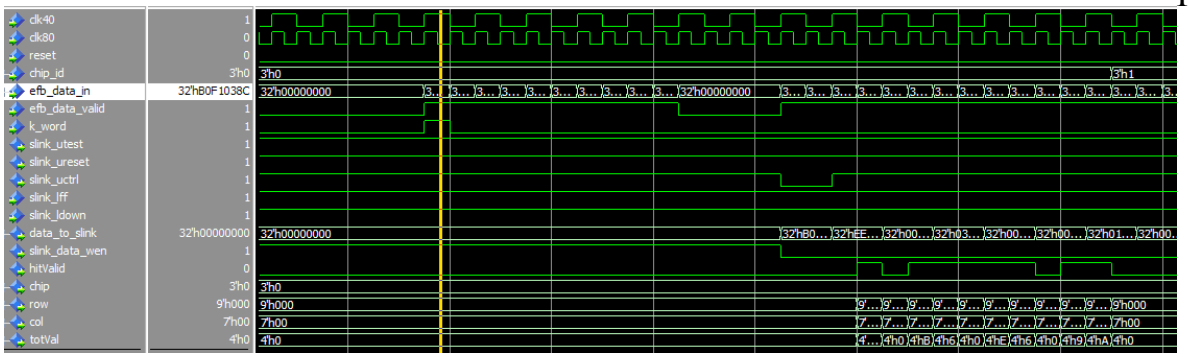
when *efb_data_valid* is 1. The *k_word* signal is usually 0: when 1 the first word of the fragment or the last one is being sent.

- **BOC (Slink lines)**: Event fragments are sent to the Slink lines only during normal data taking (i.e. not during calibration runs, *calib_mode* = 0). 32-bit data towards the Slink lines (*data_to_slink*) are synchronous with the 40 MHz clock (in the toplevel of the slave ODDR2 buffers are used to send the 32-bit bus over a 16-bit bus using Double Data Rate). Control signals are active low:
  - *slink_utest*: fixed to 1
  - *slink_ureset*: same as master reset (but active low)
  - *slink_uctrl*: active during the first and last word of a data packet
  - *slink_data_we*: active when valid data are sent to the ROS
  - *slink_lff* (link full flag): when low, the SLink is not operational, so no data shall be transmitted
  - *slink_ldown* (link down): when low, the SLink is not operational, so no data shall be transmitted
- **histogrammer**: Event fragments are sent to the histogrammer both during normal data taking and during calibration runs (regardless of the *calib_mode* value). The data taking run will not be affected by the histogrammer being not ready to received data (*rfd* = 0). The histogrammer expects to receive hits information with which to build the histograms synchronous with the 100 MHz clock (*clk100*) in the following way:
  - *hitValid*: high when a hit is being sent to the histogrammer
  - *chip*: 3-bit FeI4 identifier (this info is contained in the 32-bit input data)
  - *row*: 9-bit row identifier
  - *col*: 7-bit column identifier
  - *totVal*: 4-bit toT value
  - *rfd*: when high, the histogrammer is not ready to receive data

IMPORTANT: two adjacent hits are encoded as a single 24-bit FeI4 word. In this case only one word is sent to the Slink, but the histogrammer need to receive this information as a pair of words (it just accept one hit per clock cycle).

- **busy logic**: the busy signal *router_busy* is asserted when the internal FIFO occupancy goes beyond a programmable value (actually this value is chosen at design level, not at run time, so a proper value has to be chosen after some simulations/tests). The busy signal shall be synchronous with the 40 MHz clock.

F



igure 3: Router I/O waves

9.15.3. Router input data format

Slink header (10 words) +

MODULE ## block	data_out[31:0] (to the router)
header	001nnnnnFLLLLLLLLLLLLLBBBBBBBBBB
hit (long)	100nnnnnTTTTTTTTCCCCCCCRRRRRRRRR
hit (condensed mode)	101RRRRRTTTTTTTTCCCCCCCRRRRRRRRR 1CCCRRRRRRRRRTTTTTTTTCCCCCCCRRRR 1TTTCCCCCCCRRRRRRRRRTTTTTTTTCCCC 111TTTTTTTTCCCCCCCRRRRRRRRRTTTTT
FE flag error	000nnnnnxSSSSSSxxxxxxxxDDDDDDDDDD
trailer	010nnnnnEcPplbzhvMMMMMMMMMMBBBBB

+ Slink trailer (6 words)

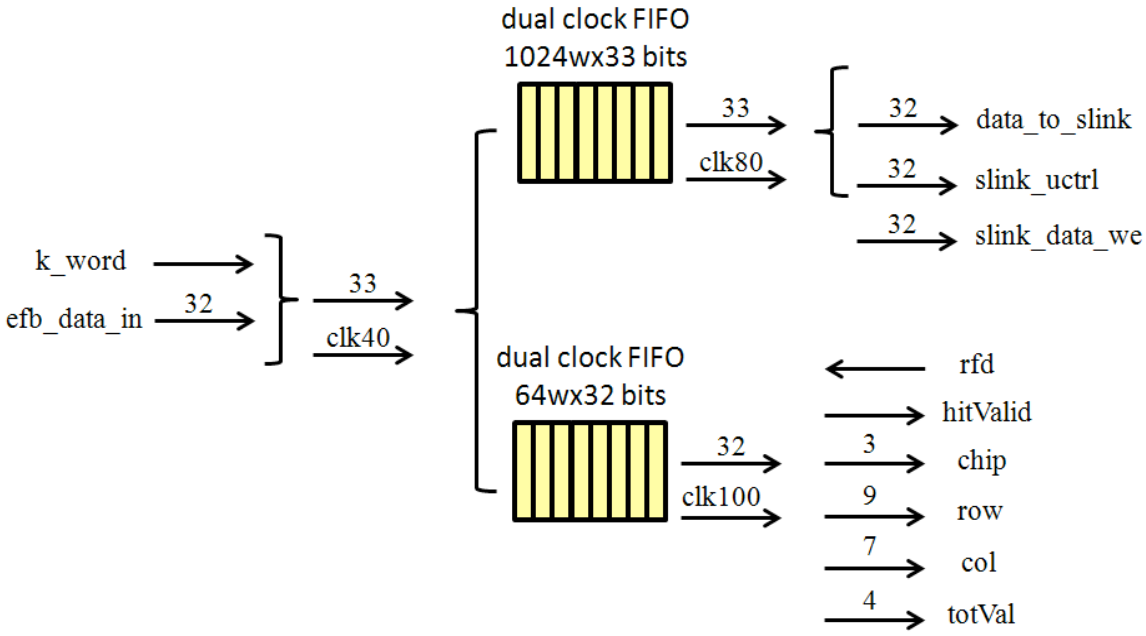
9.15.4. Router output data format

- The output data format towards the Slink is the same as the input data format from the EFB.
- Data towards the histogrammer are just hits: one single hit per clock cycle. In case condensed mode is used, the router shall be able to decode the hits (value + address) and send them to the histogrammer one per clock cycle.

9.15.5. Performances

The performances of the router block are almost completely defined by the functionality required for the block itself. So during normal data taking runs when no backpressure comes from the ROS (i.e. *slink_lff = '1' AND slink_ldown = '1'*) the input-to-output latency is just given by the write/read time of a dual clock FIFO (being written at 80 MHz and read out at 40 MHz). The same holds for calibration runs when *rfd* = '1' (in this case the latency is given by the access time of a FIFO being written at 40 MHz and read out at 100 MHz). A careful choice has to be taken for the programmable full flag of the Slink FIFO (which causes the busy assertion), so to optimize the number of times in which the busy signal needs to be asserted.

9.15.6. Implementation



9.15.7. Error handling

The router block does not perform any data integrity check on event fragments coming from the EFB. Data is received and forwarded to the Slink and histogrammer blocks without any further formatting/processing/error checking.

9.15.8. Registers

The only register used by the router block is the *calib_mode* mode, which instructs the block on the type of run being taken. No other register access is currently foreseen for this block.

Description	Address	Access	Width
Calibration Mode (dump S-Link data)	0x0006	RW	1
Link 0 Error Register	0x4000	R	32
Link 1 Error Register	0x4004	R	32
Link 2 Error Register	0x4008	R	32
Link 3 Error Register	0x400C	R	32
Link 4 Error Register	0x4010	R	32
Link 5 Error Register	0x4014	R	32
Link 6 Error Register	0x4018	R	32
Link 7 Error Register	0x401C	R	32
Link 8 Error Register	0x4020	R	32
Link 9 Error Register	0x4024	R	32
Link 10 Error Register	0x4028	R	32
Link 11 Error Register	0x402C	R	32
Link 12 Error Register	0x4030	R	32
Link 13 Error Register	0x4034	R	32

Link 14 Error Register	0x4038	R	32
Link 15 Error Register	0x403C	R	32
Header Limit Error Register	0x4040	R	32
Trailer Error Register	0x4044	R	32
Readout Timeout Error Register	0x4048	R	32
Header-Trailer Limit Error Register	0x404C	R	32
Row Column Error Register	0x4050	R	32
L1ID Error Register	0x4054	R	32
BCID Error Register	0x4058	R	32

9.16. Histogrammer

The histogrammer block of the ROD slave firmware provides the means to generate per-pixel histograms of the occupancy, the ∑ToT and ∑ToT² values that are needed for detector calibration, but also for online-monitoring. These histograms are being populated within the histogrammer block during a sampling phase, before they are transferred to an external memory during a readout phase. Once the histograms are available in the external memory, a soft-core CPU (MicroBlaze) that is implemented on the slave FPGA transfers them via TCP/IP to a compute farm for further analysis.

There are two histogramming units per slave FPGA. Thus each histogrammer is fed data from up to 8 FE-I4s (totaling 215,040 pixels) from the EFB, and that data is then stored in private SSRAMs.

9.16.1. Functionality

The histogram collection for threshold and ToT scan require simple arithmetic operations which can easily be executed in the ROD slave FPGA devices. For the threshold scan one 8-bit adder is needed to accumulate the pixel response during 100 iterations per charge step. The FPGA can perform histogram updates at a speed of 100MHz (see comments below on the histogrammer SSRAM), using the logic shown in figure 4. Each FE-I4 can generate hits at a maximum rate of 11MHz.

Serializing data from 8 FE-I4 chips results in a maximum hit rate of 85MHz. The chip number together with row and column number from the data stream are translated into an absolute address for the internal histogrammer memory. For every hit the corresponding value is read from the memory, incremented and written back to the memory in a pipelined fashion, such that one input

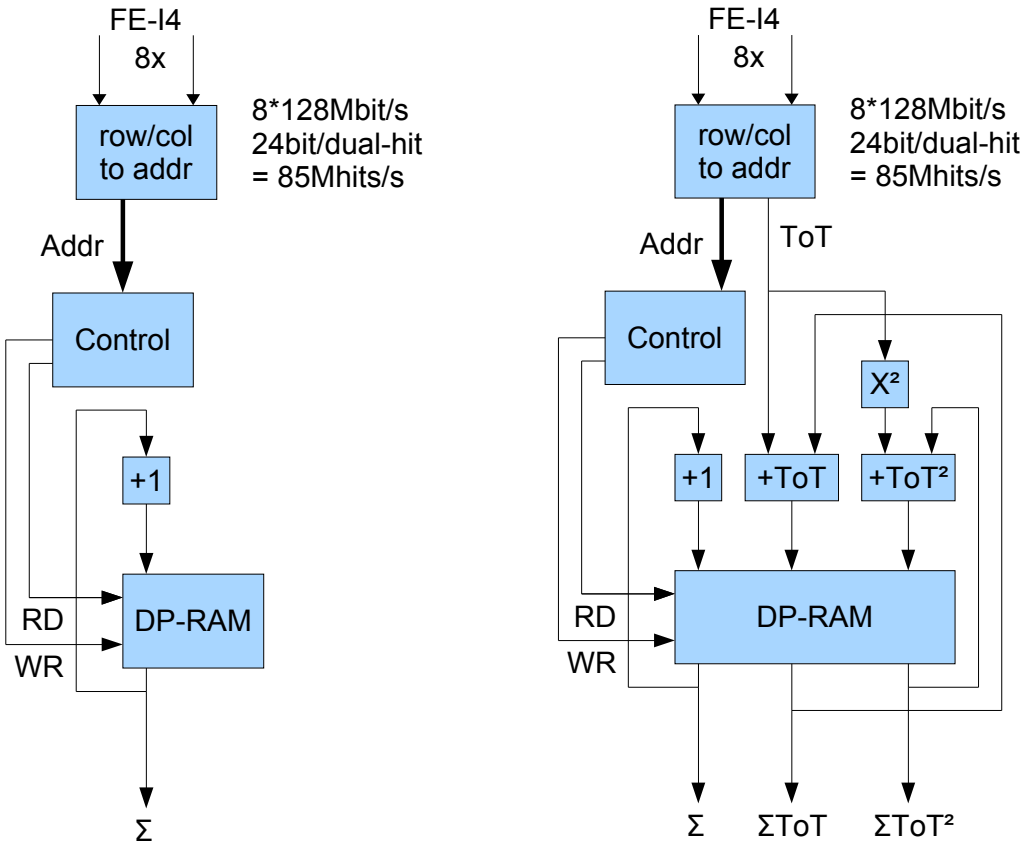


Figure 4: FPGA histogramming logic. Threshold (left) and ToT (right)

hit can be processes in every clock cycle. In case of the ToT scan, 2 additional operations are needed, summing the 4-bit ToT values and summing of the squared ToT values.

Internally the data for a pixel is stored in a 36-bit wide word, resulting in the need to adjust the format during readout to account for the 32-bit word data-width used by the DDR-RAM and MicroBlaze.

[35 .. 20]: $\sum ToT^2$ (16 bits)

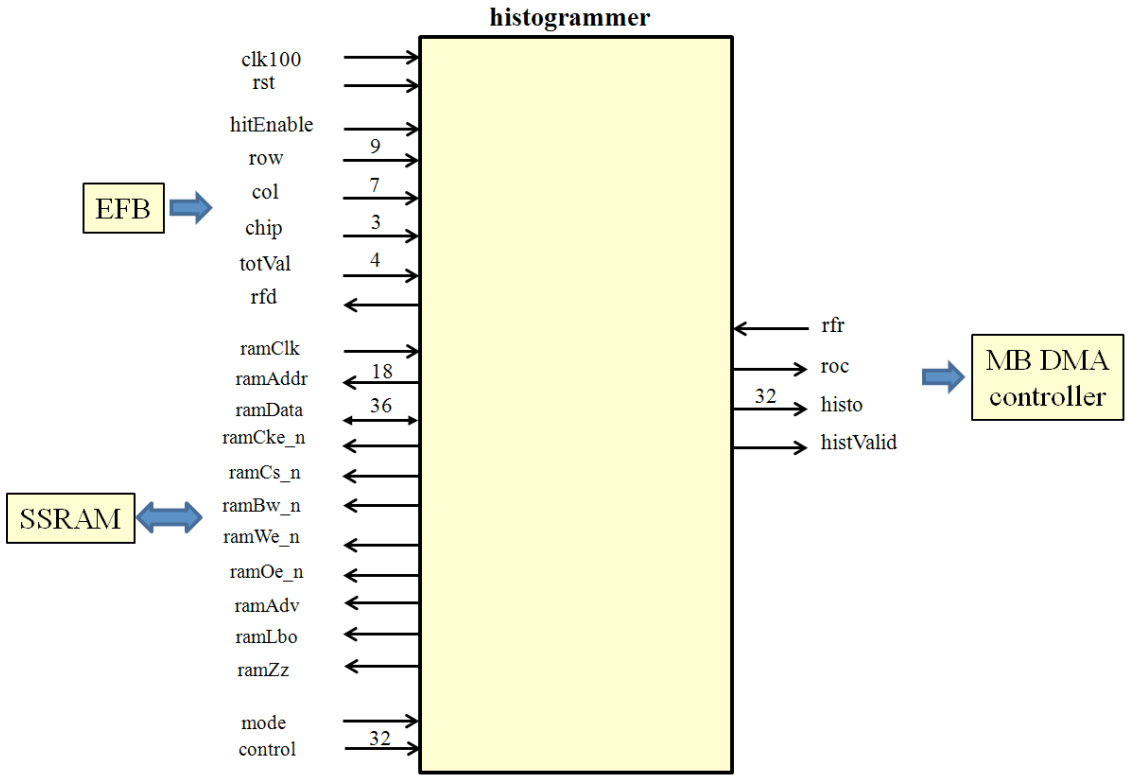
[19 .. 8]: $\sum ToT^2$ (12 bits)

[7 .. 0]: occupancy (8 bits)

Each histogramming unit is configured via a separate control register to change options such as number of chips or mode of operation. The status of both units can be accessed by reading the slave status register, while their current state (sampling, readout, idle) is controlled via the slave control register.

9.16.2. External interfaces

The histogrammer's main interfaces are to the EFB for its data input, to the SSRAM for internal storage of values during histogramming, and to the Microblaze DMA interface for readout to the DDR memory.



9.16.3. Input data format

The histogrammers input is coming from the EFB and contains the geological address of a pixel that produced a hit (row, column, pixel number, chip ID) as well as the ToT value. Valid input values for rows range from 1 to 336, for columns from 1 to 80 and for chip ID from 0 to 7.

A valid signal indicates that the data is valid.

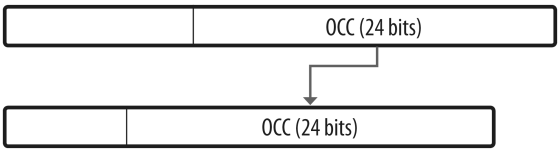
9.16.4. Output data format

While the internal representation of the data is the same for each mode of operation (expect for the non-calibration mode ONLINE_OCCUPANCY), the format of the data that is transferred to the external memory via DMA depends on the configuration of the histogrammer. This way we are able to optimize the readout and data transfer by only reading out the required information, thus accelerating the data transfer to the FitFarm.

In the following the output formats of the different modes of histogrammer operation are presented, including a graphical mapping of the internal storage format of the data in the SSRAM to the data format in the DDR-memory.

ONLINE_OCCUPANCY

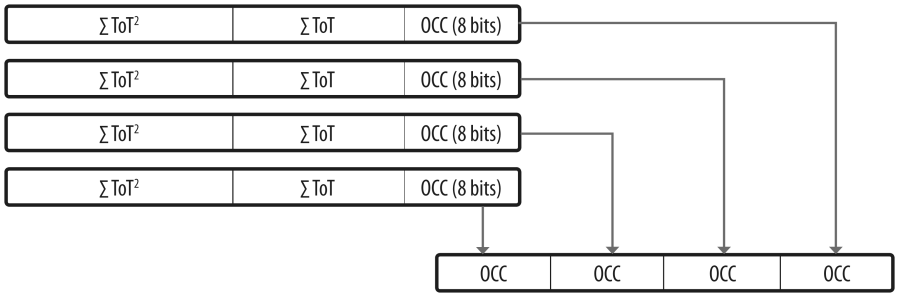
This mode is supposed to be used during data taking to create occupancy-histograms "on-the-fly". Internally only occupancy information is stored, but the maximum number of hits per pixel is with 16M a lot larger than for the other modes of operation. This way data can be accumulated over a longer period of time (a few minutes) before readout becomes necessary.



We exploit the internal structure of the FPGA (DSP blocks) by using only 24 bits for accumulating occupancy information.

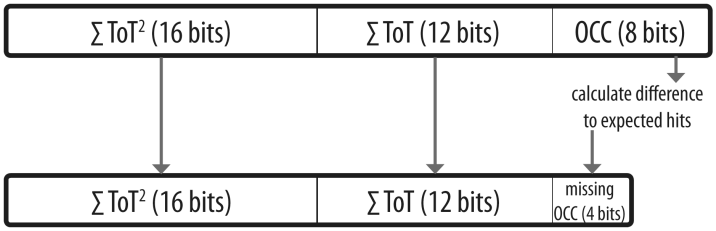
OFFLINE_OCCUPANCY

Here only the occupancy information for each pixel is read out. This is supposed to be the default readout-mode for basic scan types such as ANALOG, DIGITAL or THRESHOLD, where no ToT information is needed.



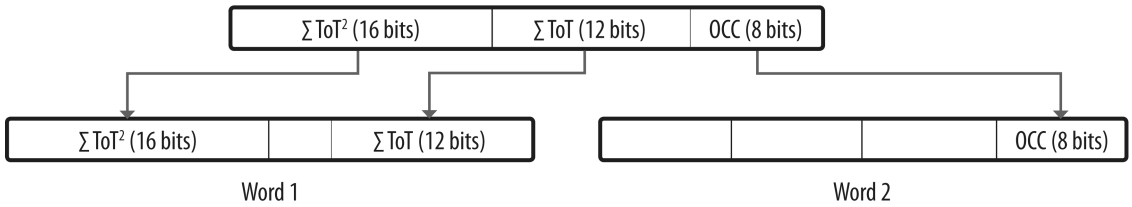
SHORT_TOT

In this mode we assume that the front-end is operated in the "well-above-threshold"-regime so that every injected charge results in a hit. Therefore the occupancy information is reduced by not reading out the occupancy a pixel but the missing hits. The expected number of hits needs to be configured beforehand and the difference between expected and actual hits is read out (with 4 bits accuracy).



LONG_TOT

For this mode all the information that is stored in the internal histogrammer SSRAM is also read out by expanding the data to two 32 bit words, resulting in 8 bytes per pixel.



9.16.5. Performances

Histogrammer performance is mostly defined by three parts: access speed of the internal memory, the readout phases, and the histogram transfer via Gigabit Ethernet.

While the calculation in section 8.1 showed that we expect a maximum hit rate of 85 MHz per histogrammer, this means that access to the internal memory needs to be happening twice as fast, as each hit results in a read and write operation. Work on the SSRAM is ongoing, but for the moment its clock is set to 100 MHz, thus constraining the operation.

During the readout-phase of the histogrammer no data can be sampled. The speed of the DMA transfer basically limits the speed of this phase and access speed of the SSRAM is a factor again.

The last major parameter that has an influence on the overall performance is the network transfer of data from the DDR-RAM to the FitFarm. One quick way to reduce the time this part takes is to select a fitting readout mode of the histogrammer in order to reduce the amount of data that needs to be transferred. The clock frequency of the MicroBlaze as well as its cache sizes have a grand influence on the network speed. Further optimization is possible by tuning the IP stack.

9.16.6. Implementation

9.16.7. Error handling

Currently the histogrammer does not implement sophisticated error handling.

Tt expects the pixel row and column address at its inputs to be in a valid range and there is currently no error handling in case of overflows.

One should also note that the histogrammer expects the incoming hit data to be ordered in such a way that hits resulting in a read/write combination to the same memory address are unproblematic – i.e. while an operation is performed on the pixel data there must not be another operation on the same data due to the pipelining. Such an access pattern would lead to missing data.

9.16.8. Registers

The slave offers two global registers that are accessible by the MicroBlaze: the control register and the status register. If needed, the control register is also readable and writable by the ROD master, but this is not foreseen during normal operation. Additionally each histogrammer is configured via a separate histogrammer control register.

The definitions of the registers for use in the slave software can be found in the generated files rodSlave.hxx and rodHisto.hxx.

Status Register (SLV_STAT_REG)

The status register can be read by the MicroBlaze and contains status information of the histogramming units. They indicate RFD (ready-for-data), RFR (ready-for-readout), and ROC (read-out-complete) to enable the software to control the histogrammers.

Control Register (SLV_CTL_REG)

The control can be made accessible by the master, via the CTL_MASTER_MODE_BIT, but the register is foreseen to be used directly by the MicroBlaze on the slave FPGA. It is used to configure the data-path of the slave. Different data sources for formatter and histogrammer can be selected for debugging and testing purposes.

The second major function of the control register is to change the states of the histogrammers (idle, sample, wait, readout) by setting the corresponding mode bits.

Histogrammer Control Register (MB_HIST0_CTL_REG & MB_HIST1_CTL_REG)

Each histogramming unit features a control register, that is used to configure the histogrammer. There are several options available:

- Operation mode: select one of the modes that are later discussed in section 8.9.5:
- ONLINE_OCCUPANCY, OFFLINE_OCCUPANCY, SHORT_TOT, LONG_TOT.
- Expected occupancy: set the number of expected hits per pixel which is used to calculate the missing triggers in SHORT_TOT mode
- Chip control: select whether we sample one chip or all 8 chips that are connected to one unit. In the first configuration the chip number as supplied by the EFB is simply ignored.
- Row mask: compresses the address space of a chip by a factor of 2, 4, or 8 in the rows to accommodate for scans that use mask-stepping.

9.17. Slave busy block

9.17.1. Functionality

The ROD busy block in the ROD slave collects the status of the almost full signal of each FIFO, generates the slave busy signal and forwards the busy signal to the ROD master busy module. A histogram to monitor the different inputs is generated. The histogram can be read for information/debugging purposes by the host. For this a 16 bit counter is

implemented for each input and incremented on the positive edge. The current status of the output and of each input is written to a read only register to allow the check of the current busy source by the host.

A 11 bit mask register is implemented to allow ignoring individual inputs.

Similarly to the master busy, a force busy input bit is implemented for all inputs and an additional bit forces the output.

9.17.2. External interfaces

Inputs:

The ROD busy receives one almost full signal per FIFO in the data path. All of these are single wire, active high signals.

A 32 bit bus from the slave register block is used for input masking and input overwriting. The mask connection is as follows:

force_rod_busy	busy_mask(0)
slink0_lff	busy_mask(1)
slink0_ldown	busy_mask(2)
slink1_lff	busy_mask(3)
slink1_ldown	busy_mask(4)
efb1_misc_stts_reg	busy_mask(5)
efb1_misc_stts_reg	busy_mask(6)
efb2_misc_stts_reg	busy_mask(7)
efb2_misc_stts_reg	busy_mask(8)
router0_busy	busy_mask(9)
router1_busy	busy_mask(10)

(force input bits similarly)

Outputs:

The ROD busy has one output link to the ROD master busy module. This signal is a single wire active high logic.

Additionally, the current status of the ROD busy is written to a register, which can be read by the host PC. This register contains one bit for the status of the output, and one bit for the status of each input in order to get information of the source of the current busy.

9.17.3. Input data format

All input links are single wire inputs, active high. The eleven mask and force input bits are connected to a 32 bit bus connected to a 32 bits of the slave register.

9.17.4. Output data format

The output signal to the ROD Master is a single wire with active low logic. The current status register is read only and connected to the ROD Slave register.

9.17.5. Performances

9.17.6. Implementation

9.17.7. Error handling

9.17.8. Registers

9.18. Firmware implementation/test status

	Implementation	Test	Designer
ROD master			
Event ID & trigger processor:			
trigger signal decoder			
sp trigger cmd decoder			
FE command processor:			
fe command processor			
fe cal command generator			
fe short command generator			
fe command pulse counter			
Event processor:			
formatter readout modebits			
EFB header dynamic mask encoder			
front end occupancy counters			
Internal scan processor			Malte
Diagnostics generator			
Address decoder bus bridge			
Register block			
Busy block			Malte- Bing
PPC peripherals			Andreas-Riccardo
ROD slave			

formatter			Bing
EFB			Bing
router			Bing
histogrammer			Andreas-Moritz
INMEM	YES	YES	
RAM block	YES	YES	
Microblaze peripherals			Andreas-Moritz

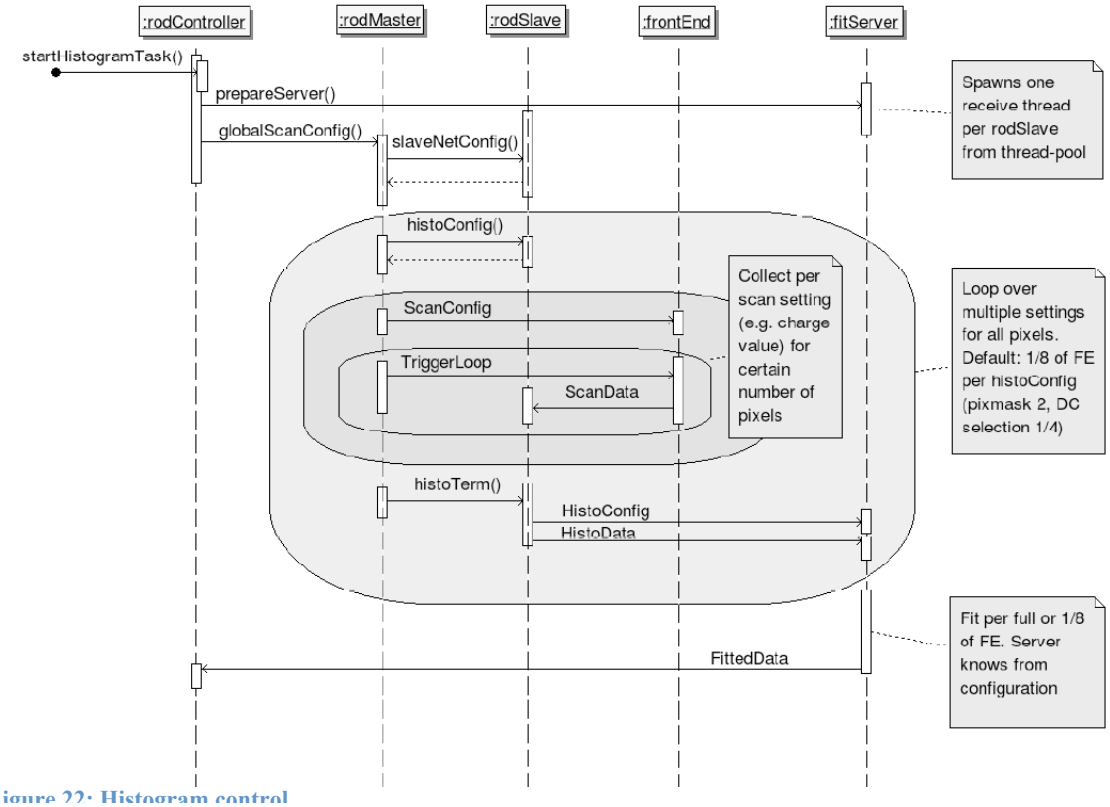
10. Histogram Control Mechanism

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 22 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 histogram-loops have to be performed.
- The rodMaster issues for each of the inner histogram loops a histogram configuration command (see HistoConfig) 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 HistoComplete) 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.



igure 22: 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.

10.1. 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) 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

Variable	Type	Comment
		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.

10.2. 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 Histogram readback) by the rodMaster.

10.3. 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

Bits	Meaning
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.

10.4. 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

11. 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.

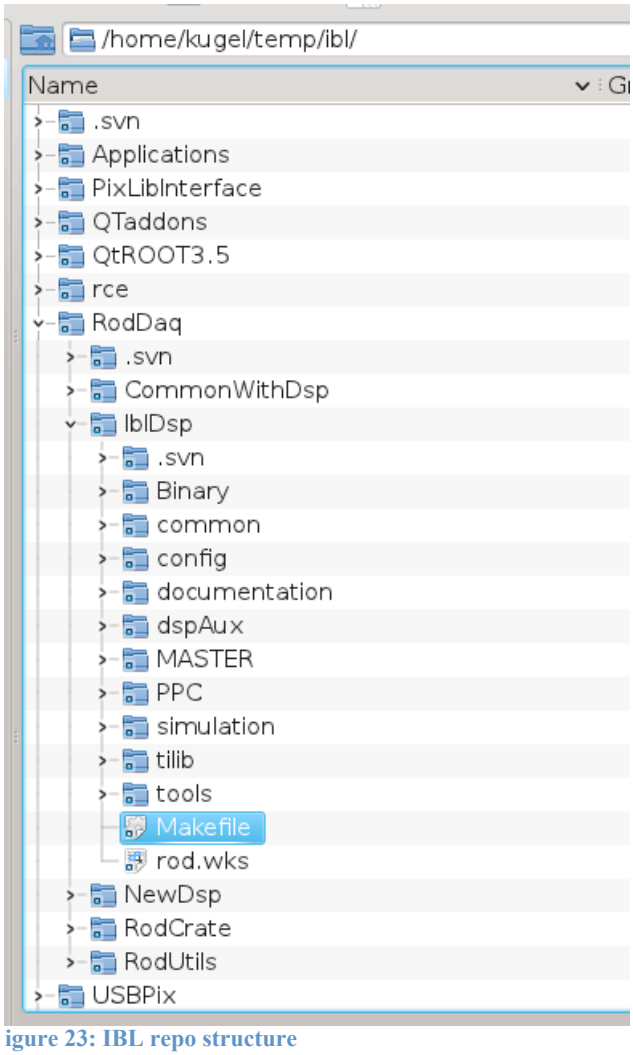
The IBL software repository[5] contains all mentioned code variations plus test programs for the communication mechanism. The main makefile (see Figure 23) allows generation of all targets:

- 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
- 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). Note that the FPGA needs to be configured first, see section Hardware dependent files
  - [kugel@pcakulap IblDsp]$ xmd

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

- 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



igure 23: IBL repo structure

PPC subdirectory.

The slave code is available in the “SLAVE” subdirectory, parallel to “PPC”. The makefile in SLAVE/src/histClient can be used to generate the slave binary file for download via VME or network, provided the Xilinx EDK tools are installed.

11.1. 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[6] 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.

The .bit files have to be programmed into the FPGAs using the Xilinx programmer cable and tools, either “impact” or “xmd”. As “xmd” has to be used anyway to download the processor binaries, it is convenient to use it for FPGA programming as well.

Start xmd:

- xmd%

Programm Virtex-5 and the two Spartan-6 FPGAs:

- xmd% fpga -f PPC/rodMaster.bit -debugdevice devicenr 8
- xmd% fpga -f PPC/sp6fmt.bit -debugdevice devicenr 3
- xmd% fpga -f PPC/sp6fmt.bit -debugdevice devicenr 6

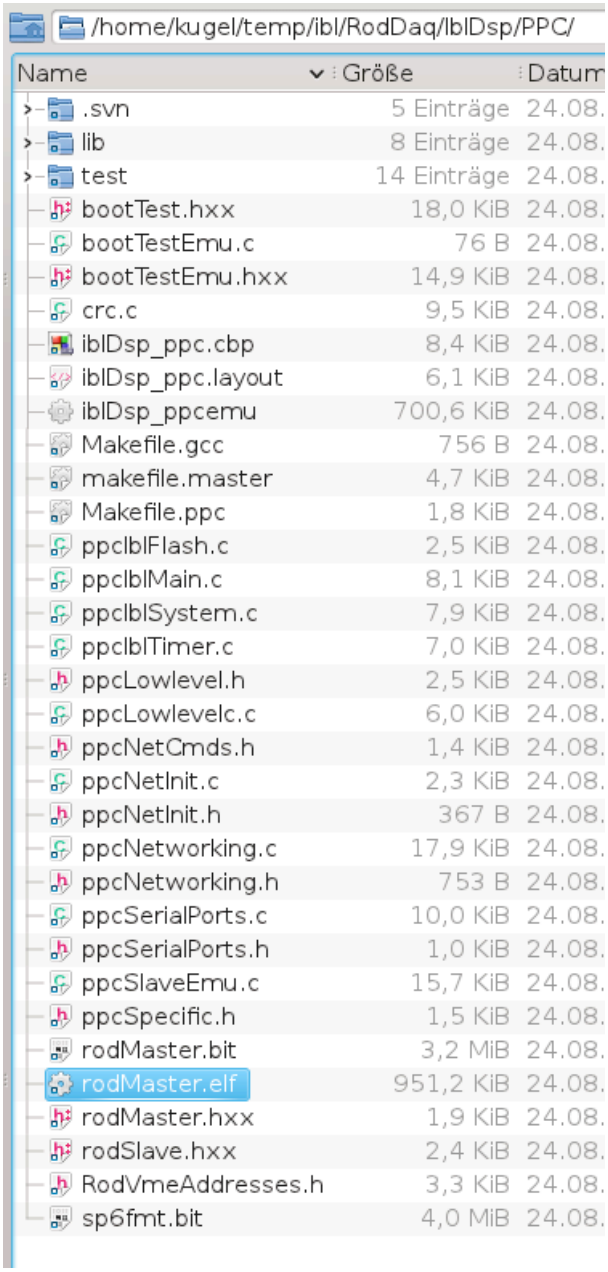
Connect to PPC on Virtex-5:

- xmd% conn ppc hw -debugdevice devicenr 8

Stop PPC, download and star binary:

- xmd% stop
- xmd% dow PPC/iblDsp_ppc.elf
- xmd% run

[6] svn.cern.ch/reps/atlasgroups/Detectors/Pixel/IBLdev/RODcode/trunk



igure 24: Hardware related files

11.2. Control and status registers

Address offsets and bit definitions are exported by the VHDL into FPGA specific header files, rodMaster.hxx and rodSlave.hxx. Many address offsets need to be combined with processor specific (DSP vs. PPC) base address definitions in order to give fully qualified addresses.

Master

```
// IBL ROD rodMaster address and bit definitions
#ifndef ROD_MASTER_H
#define ROD_MASTER_H
// This is a generated file, do not edit!
// Design identification
```

```
#define HDL_FPGA_VERSION 0x00010001
// Design parameters
// The following register addresses sit on top of the EPC area, defined in
xparameters.h
#ifndef XPAR_XPS_EPC_0_PRH0_BASEADDR
#include "xparameters.h"
#endif
// Number of EPC address bits is an alternative way to identify the address range
#define EPC_ADDRESS_BITS 25
// Design identification
#define PPC_DESIGN_REG (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x1000000)
// PPC control register
#define PPC_CTL_REG (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x1000004)
// Master: PPC is master if 1, else DSP is master
#define PPC_CTL_MASTER_BIT 31
// HPI enable: HPI port is activated if 1
#define PPC_CTL_HPIENABLE_BIT 30
// Serial port local link: local link implementation selected if 1, else EDK
peripheral version (Bologna)
#define PPC_CTL_SPLOCLINK_BIT 29
// Invert serial port clock if 1
#define PPC_CTL_SPCLKINV_BIT 28
#define PPC_CTL_UARTA_BIT 27
#define PPC_CTL_UARTB_BIT 26
// PPC status register
#define PPC_STAT_REG (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x1000008)
// Simulation mode if 1
#define PPC_STAT_SIMULATION_BIT 31
// Serial port mask. Bit 0..7 enable output from local link serial port to SP6A,
Bits 8..15 to SP6B. Output goes to XC signals
#define PPC_SPORT_REG (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x100000C)
// Common PPC+DSP registers
// Base of common (PPC+DSP) control registers, aka RCF registers
#define COMMON_REG_BASE (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x4400)
// Base of common (PPC+DSP) slave-A registers, aka formatter 0
#define SLV_A_REG_BASE (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x0)
// Base of common (PPC+DSP) slave-B registers, aka formatter 4
#define SLV_B_REG_BASE (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x1000)
// Base of common (PPC+DSP) BOC registers
#define BOC_REG_BASE (XPAR_XPS_EPC_0_PRH0_BASEADDR + 0x8000)
// Base of common (PPC+DSP) inmem registers
#endif //ROD_MASTER_H
```

Slave

```
// IBL ROD spartan-6 formatter slave address and bit definitions// IBL ROD
spartan-6 formatter slave address and bit definitions
#ifndef ROD_SLAVE_H
#define ROD_SLAVE_H
// This is a generated file, do not edit!

// make size of ram available for master PPC
#define SLV_RAM_WORDS 512
// formatter register numbers, for conveniency
#define SLV_VERSION_REG 0x22
#define SLV_INMEM_REG 0x40
#define SLV_CTL_REG 0x42
#define SLV_CTL_RESET_BIT 0
#define SLV_STAT_REG 0x43
// if not PPC, define all other things as well

#ifndef PPC // Design parameters
// The following register addresses sit on top of the EPC area, defined in
xparameters.h
#ifndef XPAR_AXI_EPC_0_PRH0_BASEADDR
```

```
#include "xparameters.h"
#endif
// Number of EPC address bits is an alternative way to identify the address range
#define MB_EPC_ADDRESS_BITS 22
// EPC address range for registers
#define MB_EPC_REG_BASE (XPAR_AXI_EPC_0_PRH0_BASEADDR)
// EPC address range for registers
#define MB_EPC_RAM_BASE (XPAR_AXI_EPC_0_PRH0_BASEADDR + 0x10000)
// Design identification
#define MB_DESIGN_REG (MB_EPC_REG_BASE + 0x0)
// Design identification
#define HDL_FPGA_VERSION 0x00010004
// Control register
#define MB_CTL_REG (MB_EPC_REG_BASE + 0x4)
// Control register bits
#define CTL_KEEP_BIT 0
#define CTL_VALID_BIT 1
#define CTL_RESET_BIT 2
#define CTL_FAKE_BOC0_BIT 3
#define CTL_FAKE_BOC1_BIT 4
#define CTL_BOC_MUX0_BIT 5
#define CTL_BOC_MUX1_BIT 6
#define CTL_HIST_MODE0_BIT 7
#define CTL_HIST_MODE1_BIT 8
#define CTL_HIST_MUX0_BIT 9
#define CTL_HIST_MUX1_BIT 10
#define CTL_DMA_TEST_BIT 11
#define CTL_DMA_MUX_BIT 12
#define CTL_BOC_ENABLE_BIT 13
// Status register
#define MB_STAT_REG (MB_EPC_REG_BASE + 0x8)
// Status bits
#define STAT_SIMULATION_BIT 31
#define STAT_EXT_HISTRAM_BIT 30
#define STAT_DMA_RDY_BIT 29
#define STAT_HIST_RFR0_BIT 28
#define STAT_HIST_RFR1_BIT 27
#define STAT_HIST_RFD0_BIT 26
#define STAT_HIST_RFD1_BIT 25
#define STAT_HIST_ROC0_BIT 24
#define STAT_HIST_ROC1_BIT 23
#define STAT_LAST_BIT 23
// BOC test input address
#define MB_BOC_TEST_REG (MB_EPC_REG_BASE + 0xC)
// BOC serial transmit to master address
#define MB_TX_REG (MB_EPC_REG_BASE + 0x10)
// Histogrammer 0 test input address
#define MB_HIST0_TEST_REG (MB_EPC_REG_BASE + 0x18)
// Histogrammer 1 test input address
#define MB_HIST1_TEST_REG (MB_EPC_REG_BASE + 0x1C)
// Histogrammer 0 control address
#define MB_HIST0_CTL_REG (MB_EPC_REG_BASE + 0x20)
// Histogrammer 1 control address
#define MB_HIST1_CTL_REG (MB_EPC_REG_BASE + 0x24)

#endif // PPC
#endif //ROD_SLAVE_H
```

11.3. Monitoring

The USB-serial port on the ROD can be used during the development to display information output from the master or the slave processors (selection see Master control

commands). Terminal parameters are 115200 bit/s, 8 bit, no parity, no carrier, no flow-control. Appropriate terminal programs are e.g. kermit, minicom etc.

11.4. 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[7] on "localhost". The other slot numbers connect to 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.

11.5. 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
```

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

```
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:

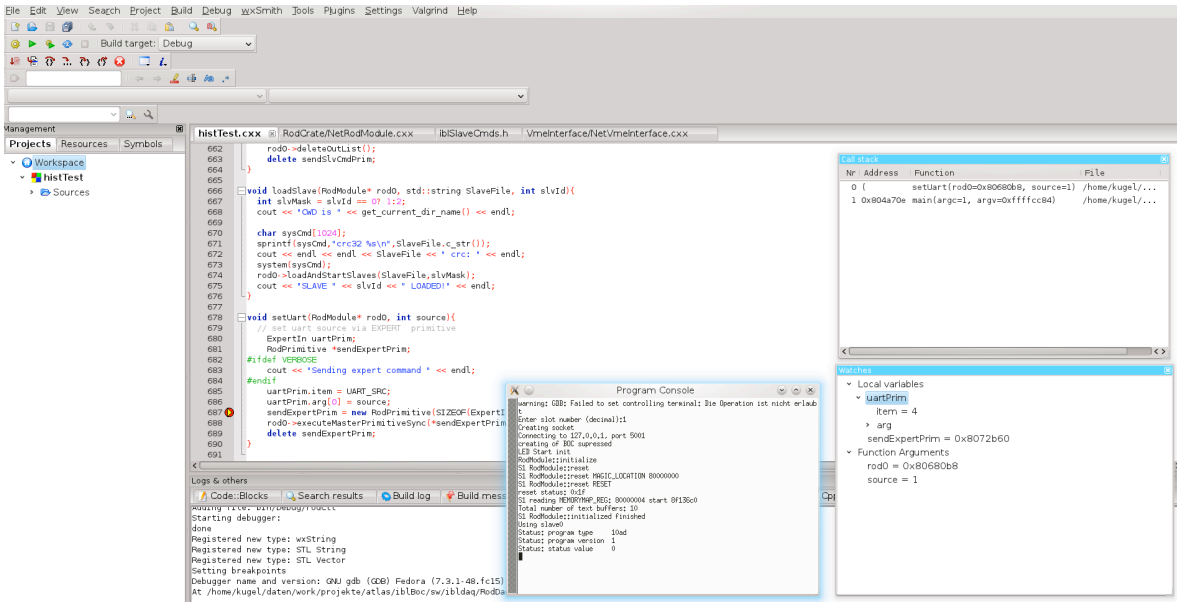
```
[kugel@pcakulap test]$ ./histTest_net_ibl
Enter slot number (decimal):1
Creating socket
Connecting to 127.0.0.1, port 5001
creating of BOC supressed
LED Start init
RodModule::initialize
S1 RodModule::reset
S1 RodModule::reset MAGIC_LOCATION 80000000
S1 RodModule::reset RESET
reset status: 0x1f
S1 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 25 shows debugging of the test program with a breakpoint just in before sending the master control primitive.



igure 25: 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 is created via make in the SLAVE/src/histClient directory (see above, chapter Development and targets)

References

[1] https://twiki.cern.ch/twiki/bin/view/Main/AtlasSiliconRodGroup
[2] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3055&rep=rep1&type=pdf