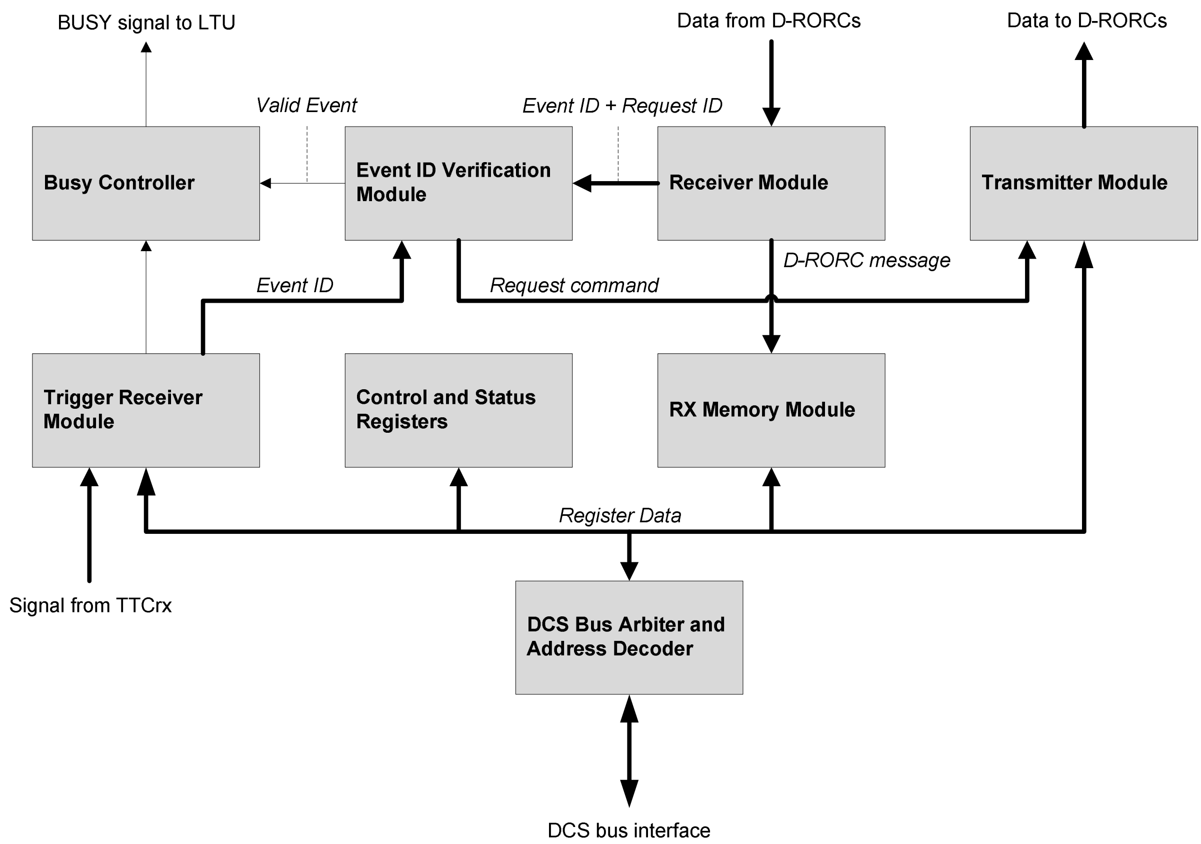
Document name: *User Guide BusyBox*

Revision: *1.0*

Date: *2009 03.06 15:15*

Author: *Rikard Bølgen*

1 Block diagram BusyBox



Features

- Decoding of serial B and L1a line
- CDH FIFO
- Event ID extractor
- D-RORC communication
- Event ID verification
- Busy signal indicating when Fee buffers are full
- BRAM module to store up to 1024 D-RORC messages

Contents

Preface

The BusyBox is an FPGA based system developed at the University of Bergen. The first of three development phases was done by Anders Rossebø and Bjørn Pommeresche. He designed the BusyBox hardware including the 19” rack case which holds all the electronics. Then Magne Munkejord developed part of the firmware and PhD student Johan Alme contributed with the Trigger Receiver Module to make the firmware complete. Test and upgrades of firmware was done by Rikard Bølgen.

This User Guide is about the whole BusyBox system. The intention is to give users and future designers an intuitive understanding of the BusyBox. The first part of this user manual is an overview of the BusyBox. Hardware, Firmware, DCS board and communication systems will be discussed. The second part is how to interact with the BusyBox; How to program, read/write registers and to test the hardware.

For more information related to the BusyBox, check out the wiki at:

https://wikihost.uib.no/ift/index.php/Busy_Box_and_related

1 Document Control

1.1 *Revision History*

Revision number	Revision date	Summary of changes	Author
1.0	03.04.09	N/A	Rikard Bølgen

1.2 *Firmware Version*

Package	Version
Firmware BusyBox	1.01
Trigger Receiver Module	1.3
DCS	2.84UiB

1.3 *References*

Ref. No.	Doc. Name	Rev/Rev date	Title
01	master_thesis_magne_munkejord.pdf	October 2007	Development of the ALICE Busy Box
02	TTC receiver requirement specification_v1.2.doc	v0.12, 12. june 2008	TTC receiver requirement specification
03	Anders_Rossebo.pdf	2006	Busy-logic for ALICE TPC

2 Motivation

The scope of this combined technical paper and user guide will be to collect all the information necessary to understand, use and modify the BusyBox.

ALICE is one of four large detectors situated at the collision points in the Large Hadron Collider (LHC) at CERN. The BusyBox is used by four of ALICE's sub-detectors: Time Projection Chamber (TPC), Photon Spectrometer (PHOS), Forward Multiplicity Detector (FMD) and Electromagnetic Calorimeter (EMCal)

Triggers initiate data readout from ALICE's sub-detectors and are received by the DCS board via an optical cable interface. The triggers and associated data are routed from the TTCrx ASIC on the DCS board to the BusyBox FPGA(s). Here, the L1a and Serial B line raw data is decoded by the Trigger Receiver firmware module.

Every time a trigger sequence starts the Fee starts buffering data, i.e. a buffer in the Fee is used. A valid trigger sequence ends with an L2a trigger and the event data along with the event ID is sent to the D-RORCs.

The purpose of BusyBox is to let the Central Trigger Processor (CTP) know when the Fee's buffers are full by asserting a *busy* signal which prevents further issuing of triggers. The BusyBox and D-RORCs receives a unique event ID from the Fee after an event. After a valid trigger sequence ends the BusyBox will ask the D-RORCs if they have received the same event ID as the BusyBox did. If they do not reply with the same ID it means data has not been shipped from the Fee to the D-RORC, hence, the buffer in the Fee still holds event data.

The Fee buffers can hold 4 or 8 events and the BusyBox keeps track of free buffers. The *busy* is asserted if the buffers are full.

Interaction with the BusyBox is done through the DCS board, either via Ethernet or UART.

3 Project Setup

3.1 *Project Files*

The complete design is checked into the SVN Repository[1], under the folder /trunk/. The Trigger Receiver module uses CVS Repository[2], under /vhdlcvs/trigger_receiver/vhdl/. Any updates for that module will be uploaded to that Repository.

File	Folder	Description
busybox_fpga1.bit	/busybox_files	Bit file to programme FPGA 1
busybox_fpga1_solo.bit	/busybox_files	Bit file to programme FPGA 1
busybox_fpga2.bit	/busybox_files	Bit file to programme FPGA 2
fpga2_dummy.bit	/busybox_files	Bit file needed to programme just one FPGA
busybox_fpga1.bit	/ISE_projects/busybox_fpga1	-
project_setup.tcl	/ISE_projects/busybox_fpga1	TCL script to set up ISE project for FPGA 1 (TPC)
busybox_fpga1_solo.bit	/ISE_projects/busybox_fpga1_solo	-
project_setup.tcl	/ISE_projects/busybox_fpga1_solo	TCL script to set up ISE project for FPGA 1 (PHOS)
busybox_fpga2.bit	/ISE_projects/busybox_fpga2	
project_setup.tcl	/ISE_projects/busybox_fpga2	TCL script to set up ISE project for FPGA 2 (TPC)
project_setup.tcl	/simulation	TCL script to set up QuestaSim project to simulate project
backbone_controller.vhd	/source	
branch_controller.vhd	/source	
busybox_fpga1.vhd	/source	
busybox_fpga1_solo.vhd	/source	
busybox_fpga2.vhd	/source	
busylogic_pkg.vhd	/source	
busylogic_top.vhd	/source	
busylogic_top.vhd.bak	/source	
busy_controller.vhd	/source	
busy_controller.vhd.bak	/source	
ctrl_regs.vhd	/source	
dcs_arbit_addr_dec.vhd	/source	
digital_clock_manager.vhd	/source	
digital_clock_manager.xaw	/source	
drorc_inbox_buffer.vhd	/source	
eventidfifo.vhd	/source	
eventid_control.vhd	/source	
eventid_extractor.vhd	/source	
event_processor.vhd	/source	
event_validator_top.vhd	/source	
multi_channel_receiver.vhd	/source	
payload_fifo.vhd	/source	
piso.vhd	/source	
receiver_memory_module.vhd	/source	

[1] http://svn.ift.uib.no/svn/busybox_firmware/
[2] http://web.ift.uib.no/kjekscgi-bin/viewcvs.cgi/

File	Folder	Description
rx_bram.vhd	/source	
rx_mem_filter.vhd	/source	
serial_decoder.vhd	/source	
serial_encoder.vhd	/source	
serial_rx.vhd	/source	
single_channel_receiver.vhd	/source	
single_channel_transmitter.vhd	/source	
transmitter_module.vhd	/source	
trigger_eventid_queue.vhd	/source	
addressed_msg_decoder.vhd	/source/trigger_module	
broadcast_msg_decoder.vhd	/source/trigger_module	
counters.vhd	/source/trigger_module	
event_fifo.vhd	/source/trigger_module	
fifo_wrapper.vhd	/source/trigger_module	
hamming_decoder.vhd	/source/trigger_module	
L1_line_decoder.vhd	/source/trigger_module	
phase_check.vhd	/source/trigger_module	
rcu_com.vhd	/source/trigger_module	
rcu_com_release.vhd	/source/trigger_module	
sequence_validator.vhd	/source/trigger_module	
serialb_com.vhd	/source/trigger_module	
test_pattern_generator.vhd	/source/trigger_module	
trigger_receiver.vhd	/source/trigger_module	
trigger_receiver_busy_logic.vhd	/source/trigger_module	
trigger_receiver_pkg.vhd	/source/trigger_module	

Table 3-1: Files checked in the SVN repository.

3.2 *Software*

Editor:	Notepad++ v4.5
Simulation:	QuestaSim 6.1d
Synthesis and Place and Route for test:	Xilinx ISE v10.1

Note: The Core's may have to be regenerated for a different ISE version or different Xilinx FPGA series.

4 External Interface

4.1 *Generic interface*

Generic name	Type	Range	Default value	Description
fpga_id	std_logic	-	-	Identify the current fpga
num_of_cahnnels	natural	0 to 119	119	Specifies the number of channels (-1) that will be instantiated at compile-time.
num_of_baranches	positive	1 to 8	8	Specifies the number of branches at compile-time.
num_of_modules	positive	1 to 8	8	Specifies the number of main firmware modules in the BusyBox.
cycles_per_bit	positive	1 to 9	5	Specifies the number of cycles each bit will be sampled.

Table 4-1: Description of generic interface.

4.2 *Signal interface*

Signal name	Type	Direction	Range	Sync	Description
clock_lvds_P	std_ulogic	IN	-	-	Reference clock, positive component of differential signal.
clock_lvds_N	std_ulogic	IN	-	-	Reference clock, negative component.
areset_n	std_logic	IN	-	No	Asynchronous active low global design reset.
serial_in	std_logic	IN	-	Falling edge	Channel B from TTCrx chip
L1Trig_P	std_logic	IN	-	Yes	Channel A from TTCrx chip. Positive component of differential signal.
L1trig_N	std_logic	IN	-	Yes	Negative component of differential signal.
serial_rx_P	std_logic_vector	IN	0 to num_of_channels	No	Serial channels input. Positive component of differential signal.
serial_rx_N	std_logic_vector	IN	0 to num_of_channels	No	Serial channel input. Negative component of differential signal.
serial_tx_P	std_logic_vector	IN	0 to num_of_channels	Yes	Serial channel output. Positive component of differential signal.
serial_tx_N	std_logic_vector	IN	0 to num_of_channels	Yes	Serial channel output. Negative component of differential signal.
dcs_data	std_logic_vector	INOUT	15 downto 0	No	Bidirectional 16 bit data bus to/from DCS board.
dcs_addr	std_logic_vector	IN	15 downto 0	No	Address line for bus to DCS board.

Signal name	Type	Direction	Range	Sync	Description
dcs_strobe_n	std_logic_vector	IN	-	No	Active low strobe signal for bus interface.
dcs_RnW	std_logic	IN	-	No	Bus control signal. Read_not_Write.
dcs_ack_n	std_logic	OUT	-	No	Bus control signal. Active low acknowledgement to bus master.
intercom_busy	std_logic	IN	-	Yes	Busy status from secondary FPGA.
BUSY_1	std_logic	OUT	-	No	Busy status output 1.
BUSY_2	std_logic	OUT	-	No	Busy status output 2.
lesds	std_logic_vector	OUT	3 downto 0	Yes	Control LEDs on the main board.

Table 4-2: Description of signal interface.

5 Register Interface

All registers for the BusyBox are listed below.

5.1 *BusyBox Register Interface*

Register Name	Address	Type [3]	Decription
TX Register[15:0]	0x0001	RW	Transmits a message on serial ports when written to. Bit 7:0 is TX Data. Bit 15:8 gives channel number in hexadecimal. Any value greater than the actual number of channels will result in a broadcast on all channels.
RX Memory1[15:0]	0x1000-0x1FFF	RW	All addresses in range where the 2 LSBs are '00'. Holds DRORC message [47:32].
RX Memory2[15:0]	0x1000-0x1FFF	RW	All addresses in range where the 2 LSBs are '01'. Holds DRORC message [31:16].
RX Memory3[15:0]	0x1000-0x1FFF	RW	All addresses in range where the 2 LSBs are '10'. Holds DRORC message [15:0].
RX Memory4[15:8]	0x1000-0x1FFF	RW	All addresses in range where the 2 LSBs are '11'. Holds DRORC channel number.
RX Memory Pointer[11:0]	0x2000	R	Value indicates where next message from DRORC will be written in RX Memory.
Event ID Count[8:0]	0x2001	R	Number of Event Ids extracted from triggers and stored in FIFO.
Current EventID[3:0]	0x2002	R	Bit 35:32 of Event ID currently being matched.
Current EventID[15:0]	0x2003	R	Bit 31:16 of Event ID currently being matched.
Current EventID[15:0]	0x2004	R	Bit 15:0 of Event ID currently being matched.
Newest EventID[3:0]	0x2005	R	Bit 35:32 of Event ID most recently received from triggers.
Newest EventID[15:0]	0x2006	R	Bit 31:16 of Event ID most recently received from triggers.
Newest EventID[15:0]	0x2007	R	Bit 15:0 of Event ID most recently received from triggers.
L0 trigger timeout	0x2008	RW	Number of clock cycles 'busy' will be asserted after an L0 trigger. Note: The busy will not be deasserted if the buffers are full.
FEE Buffers Available[3:0]	0x2009	RW	Configuration register which indicates how many events can be stored in the buffers on the FEE.
Halt FSM[0]	0x200A	RW	When set to '1' the FSM that controls the Event ID matching will halt in a known state.
Force Event ID Match[0]	0x200B	W	Writing '1' to this register when the FSM has been halted will cause the FSM to move on to the next Event ID.
Re-request Timeout[15:0]	0x200C	RW	Number of clock cycles (40 MHz domain) to wait in between sending requests to the DRORCs.
Current Request ID[3:0]	0x200D	R	Holds the Request ID the Busy Box uses to request Event Ids from the DRORCs.
Request Retry Count[15:0]	0x200E	R	Number of iterations the FSM has made while trying to match the current Event ID.
Busy Timer[15:0]	0x2010	R	Bit 31:16 of register that holds number of cycles the BUSY has been asserted.
Busy Timer[15:0]	0x2011	R	Bit 15:0 of register that holds number of cycles

[3] Legend: W=write, R=read, T= write trigger (not physical registers)

Register Name	Address	Type [3]	Decription
			the BUSY has been asserted.
RX Memory Filter[15:0]	0x2012	RW	Filters which messages will be stored in RX Memory by channel number. Bit 7:0 is matching pattern. Bit 15:8 is matching mask to enable/disable matching of individual bits.
L0 deadtime offset[15:0]	0x2013	RW	Bit 31:16 of register that holds number of cycles the 'busy' will be asserted after a L0 trigger.
L0 deadtime offset[15:0]	0x2014	RW	Bit 15:0 of register that holds number of cycles the 'busy' will be asserted after a L0 trigger.
Firmware version	0x2015	R	Gives version number in format x.xx
Channel Register[1:0]	0x21XX	RW	Provides information on status of channel 'XX'. Bit 0: '1' receiver for the channel is enabled and a matching Event ID from this channel is required. Bit 1: '1' indicates that the current Event ID has been matched for this channel. This bit is read only.

Table 5-1: List of registers that can be accessed externally.

5.2 *Trigger Receiver Module Register Interface*

Register name	Address	Type [4]	Description
Control[15:0]	0x3000	RW	[0] Serial B channel on/off *Default: 1* [1] Disable_error_masking *0* [2] Enable RoI decoding *0* [3] L0 support *1* [4:7] *(Not Used)* [8] L2a FIFO storage mask *1* [9] L2r FIFO storage mask *1* [10] L2 Timeout FIFO storage mask *1* [11] L1a message mask *1* [12] Trigger Input Mask Enable *0* [13:15] *(Not Used)*
Control[7:0]	0x3001	R	[16] Bunch_counter overflow - [17] Run Active - [18] Busy (receiving sequence) - [19] *Not Used* [23:20] CDH version 0x2
Module reset	0x3002	T	Reset Module
RoI_Config1[15:0]	0x3004	RW	RoI-Definition. Bit 15:0
RoI_Config1[1:0]	0x3005	RW	RoI Definition. Bit 17:16
RoI_Config2[15:0]	0x3006	RW	RoI Definition. Bit 33:18
RoI_Config2[1:0]	0x3007	RW	RoI Definition. Bit 35:34
Reset_Counters	0x3008	T	Write to this registers will reset the counters in the module
Issue_TestMode	0x300A	T	Debug: Issues testmode sequence. Note that serialB channel input MUST be disabled when using this feature.
L1_Latency[15:0]	0x300C	RW	[15:12] Uncertainty region +- N. default value 0x2 (50 ns) [11:0] Latency from L0 to L1, default value 0x30D4 (5.3 us)
L2_Latency[15:0]	0x300E	RW	[15:0] Max Latency from BC0 to L2

[4] Legend: W=write, R=read, T= write trigger (not physical registers)

Register name	Address	Type[4]	Description
L2_Latency[15:0]	0x300F	RW	[31:16] Min Latency from BC0 to L2
PrePulse_Latency[7:0]	0x3010	RW	
RoI_Latency[15:0]	0x3012	RW	[15:0] Max Latency from BC0 to RoI msg
RoI_Latency[15:0]	0x3013	RW	[31:16] Min Latency from BC0 to RoI msg
L1_msg_latency[15:0]	0x3014	RW	[15:0] Max Latency from BC0 to L1 msg
L1_msg_latency[15:0]	0x3015	RW	[15:0] Max Latency from BC0 to L1 msg
Pre_pulse_counter[15:0]	0x3016	RW	Number of decoded pre-pulses.
BCID_Local[11:0]	0x3018	R	Number of bunchcrossings at arrival of L1 trigger.
L0_counter[15:0]	0x301A	R	Number of L0 triggers
L1_counter[15:0]	0x301C	R	Number of L1 triggers
L1_msg_counter[15:0]	0x301E	R	Number of successfully decoded L1 messages
L2a_counter[15:0]	0x3020	R	Number of successfully decoded L2a messages
L2r_counter[15:0]	0x3022	R	Number of successfully decoded L2r messages
RoI_counter[15:0]	0x3024	R	Number of successfully decoded RoI messages
Bunchcounter[11:0]	0x3026	R	Debug: Number of bunchcrossings
hammingErrorCnt[15:0]	0x302C	R	[15:0] Number of single bit hamming errors
hammingErrorCnt[15:0]	0x302D	R	[31:16] Number of double bit hamming errors
ErrorCnt[15:0]	0x302E	R	[15:0] Number of message decoding errors
ErrorCnt[15:0]	0x302F	R	[31:16] Number of errors related to sequence and timeouts.
Buffered_events[4:0]	0x3040	R	Number of events stored in the FIFO.
DAQ_Header01[15:0]	0x3042	R	Latest received DAQ Header 1 [15:0]
DAQ_Header01[15:0]	0x3043	R	Latest received DAQ Header 1 [31:16]
DAQ_Header02[15:0]	0x3044	R	Latest received DAQ Header 2 [15:0]
DAQ_Header02[15:0]	0x3045	R	Latest received DAQ Header 2 [31:16]
DAQ_Header03[15:0]	0x3046	R	Latest received DAQ Header 3 [15:0]
DAQ_Header03[15:0]	0x3047	R	Latest received DAQ Header 3 [31:16]
DAQ_Header04[15:0]	0x3048	R	Latest received DAQ Header 4 [15:0]
DAQ_Header04[15:0]	0x3049	R	Latest received DAQ Header 4 [31:16]
DAQ_Header05[15:0]	0x304A	R	Latest received DAQ Header 5 [15:0]
DAQ_Header05[15:0]	0x304B	R	Latest received DAQ Header 5 [31:16]
DAQ_Header06[15:0]	0x304C	R	Latest received DAQ Header 6 [15:0]
DAQ_Header06[15:0]	0x304D	R	Latest received DAQ Header 6 [31:16]
DAQ_Header07[15:0]	0x304E	R	Latest received DAQ Header 7 [15:0]
DAQ_Header07[15:0]	0x304F	R	Latest received DAQ Header 7 [31:16]
Event_info[11:0]	0x3050	R	[0] RoI enabled [1] Region of Interest announced (=ESR) [2] RoI received [3] Within region of interest [4:7] Calibration/SW trigger type (= RoC) [8] Software trigger event [9] Calibration trigger event [10] Event has L2 Reject trigger [11] Event has L2 Accept trigger

Register name	Address	Type[4]	Description
Event_error[15:0]	0x3052	R	[0] Serial B Stop Bit Error [1] Single Bit Hamming Error Individually Addr. [2] Double Bit Hamming Error Individually Addr. [3] Single Bit Hamming Error Broadcast. [4] Double Bit Hamming Error Broadcast. [5] Unknown Message Address Received [6] Incomplete L1 Message [7] Incomplete L2a Message [8] Incomplete RoI Message [9] TTCrx Address Error (not X"0003") [10] Spurious L0 [11] Missing L0 [12] Spurious L1 [13] Boundary L1 [14] Missing L1 [15] L1 message arrives outside legal timeslot
Event_error[11:0]	0x3053	R	[16] L1 message missing/timeout [17] L2 message arrives outside legal timeslot [18] L2 message missing/timeout [19] RoI message arrives outside legal timeslot [20] RoI message missing/timeout [21] Prepulse error (=0; possible future use) [22] L1 message content error [23] L2 message content error [24] RoI message content error
L1_MessageHeader[11:0]	0x3060	R	Debug: Latest received L1 Message
L1_MessageData1[11:0]	0x3062	R	Debug: Latest received L1 Message
L1_MessageData2[11:0]	0x3064	R	Debug: Latest received L1 Message
L1_MessageData3[11:0]	0x3066	R	Debug: Latest received L1 Message
L1_MessageData4[11:0]	0x3068	R	Debug: Latest received L1 Message
L2aMessageHeader[11:0]	0x306A	R	Debug: Latest received L2a Message
L2aMessageData1[11:0]	0x306C	R	Debug: Latest received L2a Message
L2aMessageData2[11:0]	0x306E	R	Debug: Latest received L2a Message
L2aMessageData3[11:0]	0x3070	R	Debug: Latest received L2a Message
L2aMessageData4[11:0]	0x3072	R	Debug: Latest received L2a Message
L2aMessageData5[11:0]	0x3074	R	Debug: Latest received L2a Message
L2aMessageData6[11:0]	0x3076	R	Debug: Latest received L2a Message
L2aMessageData7[11:0]	0x3078	R	Debug: Latest received L2a Message
L2rMessageHeader[11:0]	0x307A	R	Debug: Latest received L2r Message
RoIMessageHeader[11:0]	0x307C	R	Debug: Latest received RoI Message
RoIMessageData1[11:0]	0x307E	R	Debug: Latest received RoI Message
RoIMessageData2[11:0]	0x3080	R	Debug: Latest received RoI Message
RoIMessageData3[11:0]	0x3082	R	Debug: Latest received RoI Message
FIFO_read_enable	0x3100	T	Debug: Triggers a readout pulse to FIFO
FIFO_DAQHeader[15:0]	0x3102	R	Debug: Output of FIFO [15:0]
FIFO_DAQHeader[15:0]	0x3103	R	Debug: Output of FIFO [31:16]

Table 5-2: List of registers that can be accessed externally. Note that the registers marked debug can be excluded by setting the generic include_debug_registers to false, but during the development of HW/FW they come in handy for testing and verification. The module address is not given in this table.

5.3 *TPC Channel Register Interface*

Ch	Address	TPC	Patch	Ch	Address	TPC	Patch
0	0X2100	C00	RCU0	108	0x216c	A00	RCU0
1	0X2101	C00	RCU1	109	0x216d	A00	RCU1
2	0X2102	C00	RCU2	110	0x216e	A00	RCU2
3	0X2103	C00	RCU3	111	0x216f	A00	RCU3
4	0X2104	C00	RCU4	112	0x2170	A00	RCU4
5	0X2105	C00	RCU5	113	0x2171	A00	RCU5
6	0X2106	C01	RCU0	114	0x2172	A01	RCU0
7	0X2107	C01	RCU1	115	0x2173	A01	RCU1
8	0X2108	C01	RCU2	116	0x2174	A01	RCU2
9	0X2109	C01	RCU3	117	0x2175	A01	RCU3
10	0X210a	C01	RCU4	118	0x2176	A01	RCU4
11	0x210b	C01	RCU5	119	0x2177	A01	RCU5
12	0x210c	C02	RCU0	0	0xC100	A02	RCU0
13	0x210d	C02	RCU1	1	0xC101	A02	RCU1
14	0x210e	C02	RCU2	2	0xC102	A02	RCU2
15	0x210f	C02	RCU3	3	0xC103	A02	RCU3
16	0x2110	C02	RCU4	4	0xC104	A02	RCU4
17	0x2111	C02	RCU5	5	0xC105	A02	RCU5
18	0x2112	C03	RCU0	6	0xC106	A03	RCU0
19	0x2113	C03	RCU1	7	0xC107	A03	RCU1
20	0x2114	C03	RCU2	8	0xC108	A03	RCU2
21	0x2115	C03	RCU3	9	0xC109	A03	RCU3
22	0x2116	C03	RCU4	10	0xC10a	A03	RCU4
23	0x2117	C03	RCU5	11	0xC10b	A03	RCU5
24	0x2118	C04	RCU0	12	0xC10c	A04	RCU0
25	0x2119	C04	RCU1	13	0xC10d	A04	RCU1
26	0X211a	C04	RCU2	14	0xC10e	A04	RCU2
27	0x211b	C04	RCU3	15	0xC10f	A04	RCU3
28	0x211c	C04	RCU4	16	0xC110	A04	RCU4
29	0x211d	C04	RCU5	17	0xC111	A04	RCU5
30	0x211e	C05	RCU0	18	0xC112	A05	RCU0
31	0x211f	C05	RCU1	19	0xC113	A05	RCU1
32	0x2120	C05	RCU2	20	0xC114	A05	RCU2
33	0x2121	C05	RCU3	21	0xC115	A05	RCU3
34	0x2122	C05	RCU4	22	0xC116	A05	RCU4
35	0x2123	C05	RCU5	23	0xC117	A05	RCU5
36	0x2124	C06	RCU0	24	0xC118	A06	RCU0
37	0x2125	C06	RCU1	25	0xC119	A06	RCU1
38	0x2126	C06	RCU2	26	0xC11a	A06	RCU2
39	0x2127	C06	RCU3	27	0xC11b	A06	RCU3
40	0x2128	C06	RCU4	28	0xC11c	A06	RCU4
41	0x2129	C06	RCU5	29	0xC11d	A06	RCU5
42	0X212a	C07	RCU0	30	0xC11e	A07	RCU0
43	0x212b	C07	RCU1	31	0xC11f	A07	RCU1
44	0x212c	C07	RCU2	32	0xC120	A07	RCU2
45	0x212d	C07	RCU3	33	0xC121	A07	RCU3
46	0x212e	C07	RCU4	34	0xC122	A07	RCU4
47	0x212f	C07	RCU5	35	0xC123	A07	RCU5
48	0x2130	C08	RCU0	36	0xC124	A08	RCU0
49	0x2131	C08	RCU1	37	0xC125	A08	RCU1
50	0x2132	C08	RCU2	38	0xC126	A08	RCU2
51	0x2133	C08	RCU3	39	0xC127	A08	RCU3
52	0x2134	C08	RCU4	40	0xC128	A08	RCU4
53	0x2135	C08	RCU5	41	0xC129	A08	RCU5
54	0x2136	C09	RCU0	42	0xC12a	A09	RCU0

Ch	Address	TPC	Patch	Ch	Address	TPC	Patch
55	0x2137	C09	RCU1	43	0xC12b	A09	RCU1
56	0x2138	C09	RCU2	44	0xC12c	A09	RCU2
57	0x2139	C09	RCU3	45	0xC12d	A09	RCU3
58	0X213a	C09	RCU4	46	0xC12e	A09	RCU4
59	0x213b	C09	RCU5	47	0xC12f	A09	RCU5
60	0x213c	C10	RCU0	48	0xC130	A10	RCU0
61	0x213d	C10	RCU1	49	0xC131	A10	RCU1
62	0x213e	C10	RCU2	50	0xC132	A10	RCU2
63	0x213f	C10	RCU3	51	0xC133	A10	RCU3
64	0x2140	C10	RCU4	52	0xC134	A10	RCU4
65	0x2141	C10	RCU5	53	0xC135	A10	RCU5
66	0x2142	C11	RCU0	54	0xC136	A11	RCU0
67	0x2143	C11	RCU1	55	0xC137	A11	RCU1
68	0x2144	C11	RCU2	56	0xC138	A11	RCU2
69	0x2145	C11	RCU3	57	0xC139	A11	RCU3
70	0x2146	C11	RCU4	58	0xC13a	A11	RCU4
71	0x2147	C11	RCU5	59	0xC13b	A11	RCU5
72	0x2148	C12	RCU0	60	0xC13c	A12	RCU0
73	0x2149	C12	RCU1	61	0xC13d	A12	RCU1
74	0X214a	C12	RCU2	62	0xC13e	A12	RCU2
75	0x214b	C12	RCU3	63	0xC13f	A12	RCU3
76	0x214c	C12	RCU4	64	0xC140	A12	RCU4
77	0x214d	C12	RCU5	65	0xC141	A12	RCU5
78	0x214e	C13	RCU0	66	0xC142	A13	RCU0
79	0x214f	C13	RCU1	67	0xC143	A13	RCU1
80	0x2150	C13	RCU2	68	0xC144	A13	RCU2
81	0x2151	C13	RCU3	69	0xC145	A13	RCU3
82	0x2152	C13	RCU4	70	0xC146	A13	RCU4
83	0x2153	C13	RCU5	71	0xC147	A13	RCU5
84	0x2154	C14	RCU0	72	0xC148	A14	RCU0
85	0x2155	C14	RCU1	73	0xC149	A14	RCU1
86	0x2156	C14	RCU2	74	0xC14a	A14	RCU2
87	0x2157	C14	RCU3	75	0xC14b	A14	RCU3
88	0x2158	C14	RCU4	76	0xC14c	A14	RCU4
89	0x2159	C14	RCU5	77	0xC14d	A14	RCU5
90	0X215a	C15	RCU0	78	0xC14e	A15	RCU0
91	0x215b	C15	RCU1	79	0xC14f	A15	RCU1
92	0x215c	C15	RCU2	80	0xC150	A15	RCU2
93	0x215d	C15	RCU3	81	0xC151	A15	RCU3
94	0x215e	C15	RCU4	82	0xC152	A15	RCU4
95	0x215f	C15	RCU5	83	0xC153	A15	RCU5
96	0x2160	C16	RCU0	84	0xC154	A16	RCU0
97	0x2161	C16	RCU1	85	0xC155	A16	RCU1
98	0x2162	C16	RCU2	86	0xC156	A16	RCU2
99	0x2163	C16	RCU3	87	0xC157	A16	RCU3
100	0x2164	C16	RCU4	88	0xC158	A16	RCU4
101	0x2165	C16	RCU5	89	0xC159	A16	RCU5
102	0x2166	C17	RCU0	90	0xC15a	A17	RCU0
103	0x2167	C17	RCU1	91	0xC15b	A17	RCU1
104	0x2168	C17	RCU2	92	0xC15c	A17	RCU2
105	0x2169	C17	RCU3	93	0xC15d	A17	RCU3
106	0X216a	C17	RCU4	94	0xC15e	A17	RCU4
107	0x216b	C17	RCU5	95	0xC15f	A17	RCU5

Table 5-3: List registers for all BusyBox channel numbers in decimal, the address to their registers and which RCU-DRORC pair should be connected to this channel.

5.4 ***BusyBox Channel Layout***

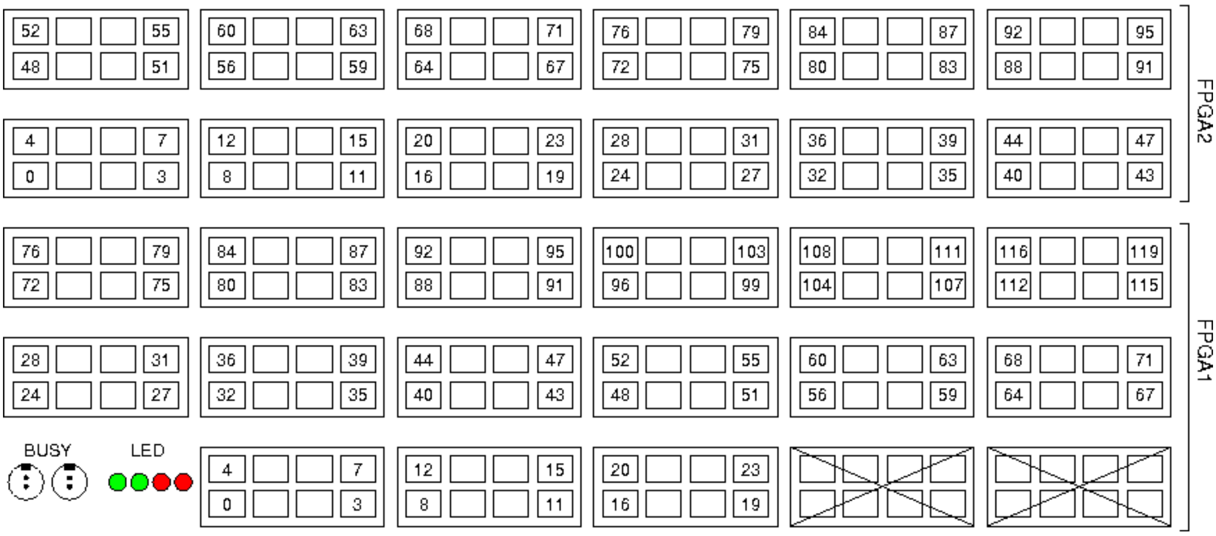


Figure 5-1: Layout of 5 U front panel for the BusyBox.

6 System Overview

The BusyBox is a part of the data acquisition in four of the ALICE sub-detectors, namely: TPC, PHOS, FMD and EMCal. There are some minor differences between the BusyBoxes for each sub-detector because of the different numbers of D-RORCs they use.

Table 6-1: Number of D-RORCs per detector.

Detector	D-RORCS	Panel height
TPC	216	5 units
PHOS	20	1 unit
FMD	24	1 unit
EMCal	3	1 unit

Data acquisition in ALICE is trigger based and is controlled by a Central Trigger Processor (CTP). The CTP distributes a trigger sequence starting with a L0 trigger when it detects a collision. Then, depending on the quality of the collision a L1 followed by an L2a or L2r trigger is issued by the CTP via the LTU.

The TPC Fee starts buffering data upon receiving a L1 trigger and PHOS a L0 trigger. The Fee on the four sub-detectors can buffer 4 or 8 events depending on number of samples configured.

So, the BB has two main tasks, keep track of available buffers and maintain a past-future protection. If the buffers are full or a L1 trigger is issued the BusyBox asserts a *busy* signal to the CTP, which will halt further triggers. The *busy* is then removed if these conditions are no longer true.

The BusyBox has no direct communication with the Fee and keeps track of available buffers by communicating with the D-RORCs. The Trigger System sends triggers to the BusyBox and the Fee. Figure 6-1 below illustrates the BusyBox place in the readout chain.

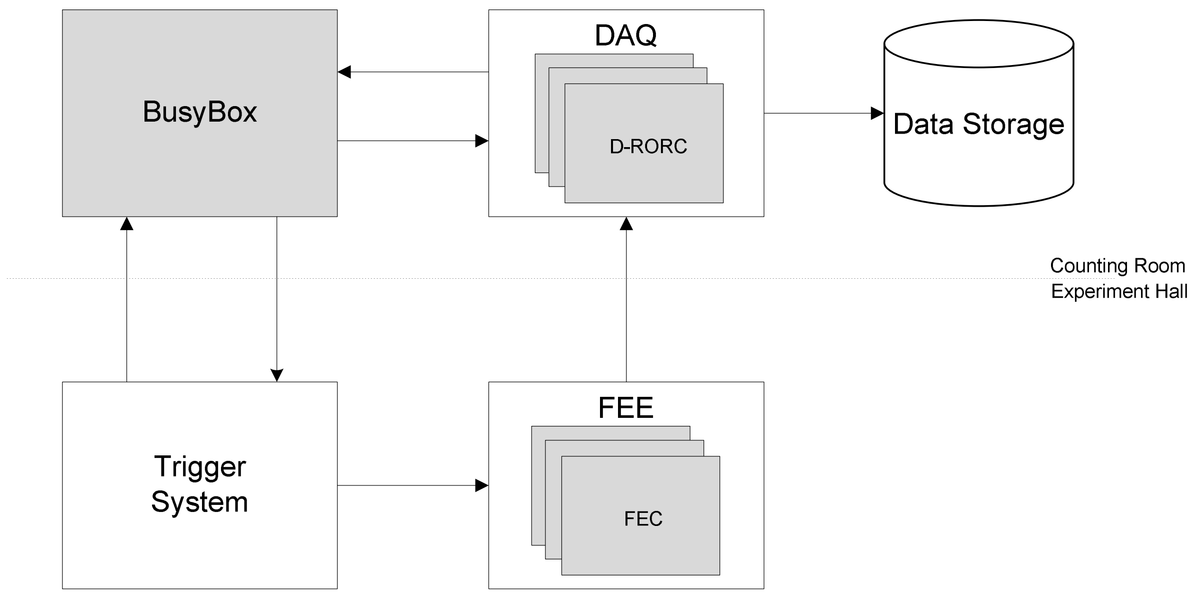


Figure 6-1: Illustration of the data flow for the BusyBox system. The BusyBox and D-RORCs are placed in the counting rooms above the experiment hall.

7 BusyBox Firmware

This chapter discusses the functionality of the firmware and gives a description of each module with sub-modules. The firmware modules are described with text, pictures, entities and port details.

7.1 *Introduction*

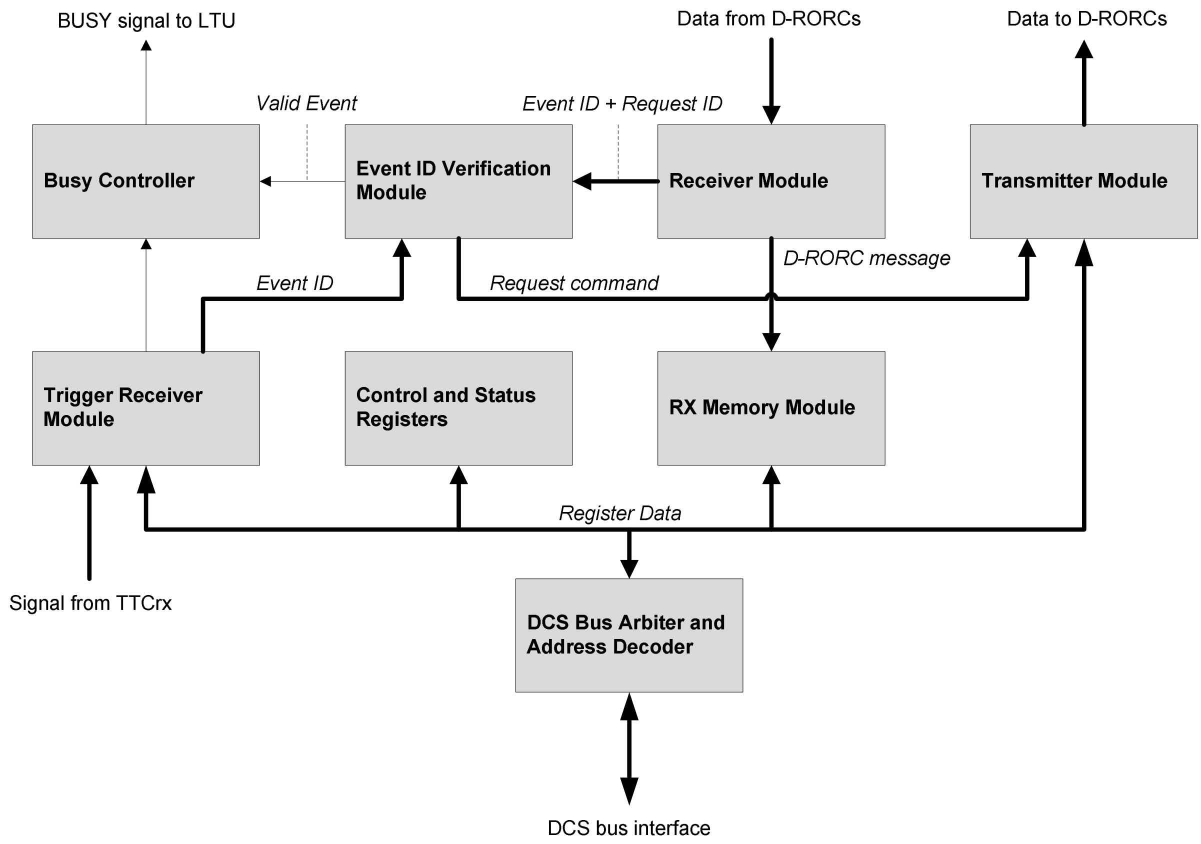


Figure 7-1: Main BusyBox firmware modules.

The firmware controls the BusyBox and executes its designed purpose based on inputs from three sources: TTCrx, BusyBox DCS card and the D-RORCs. The above figure shows the main firmware modules of the BusyBox and will be discussed in more detail. As mentioned before the BusyBox has two main functions: assert the *busy* signal if Fee buffers are full or when a L0 trigger has been issued by the CTP.

7.1.1 An intuitive explanation of how the BusyBox firmware works

It all starts with a collision of hadrons in the LHC's ALICE detector. The CTP detects this collision and notifies the LTU, which sends a L0 trigger to the BusyBox via its optical fibre network. The L0 trigger is the start of a sequence of triggers which ends with either an L2a, L2r trigger or a L2 timeout.

The LTU broadcasts the system clock that is directly dependent on the bunchcrossing frequency in LHC, in addition to L1a and Serial B line, to the BusyBox. This is done through a fibre network and converted by the TTCrx chip on the DCS card to electrical signals. Then the information is decoded by the Trigger Receiver firmware module in the BusyBox.

Not all of the decoded messages are useful for the BusyBox. Hence, the Trigger Receiver module only extracts the event ID and triggers from the LTU broadcasts. The triggers are forwarded to the Busy Controller module, which decides when to assert the *busy*.

The event ID is used to verify that all D-RORCs have received data from an event and with only that information in hand; the BusyBox can keep track of the Fee buffers. The BusyBox sends a message to all D-RORCs requesting them to send back the last event ID they have. If the event ID received from the D-RORCs is the same event ID as the one the BusyBox received from LTU, it implies that the event data has been read out from the Fee buffers.

It is the Busy Controller module that keeps track of the Fee buffers. Fee buffers can hold 4 or 8 events and starts buffering data on a L0 trigger (TPC starts on a L1). If there is a L0 trigger 1 buffer is occupied and the buffer is freed if the D-RORC corresponding to that Fee has replied with the same event ID. This is then checked as OK in a register named EIDOK. The EIDOK register is AND'ed with the CHEN register (Channel Enable) giving a 1 if all event ID's from the D-RORCs are verified.

A control and status register can as the name implies, control and check the status of registers in the BusyBox. This is done via the DCS board mounted on the BusyBox PCB.

7.1.2 VHDL Entity Hierarchy

- busybox_fpga1_solo || busybox_fpga1 || busybox_fpga2
  - busylogic_top
    - ctrl_reg
    - dcs_arbit_addr_dec
    - transmitter_module
      - serial_encoder
        - PISO
    - multi_channel_receiver
      - signle_channel_receiver
        - serial_rx
      - branch_controller
      - backbone_controller
    - rx_mem_filter
    - receiver_memory_module
    - rx_bram
    - event_validator_top
      - drorc_inbox_buffer
      - trigger_eventid_queue
        - eventide_fifo
        - eventide_extractor
      - eventide_control
      - eventide_processor
    - trigger_receiver_busy_logic
    - busy_controller

7.2 *BusyBox FPGA Modules*

The BusyBox can have one or two FPGAs depending on which detector it is used for. There are only minor differences in the source code for the three firmware versions (FPGA1, FPGA2 and FPGA solo). The differences are the number of channels instantiated and extra logic to coordinate the BUSY signal from the second FPGA to the first when two FPGAs are used.

7.2.1 Entity BusyBox FPGA Modules

This module acts as a wrapper for each version of the three firmware versions: busybox_fpga1.vhd, busybox_fpga2.vhd and busybox_fpga1_solo.vhd. These wrappers instantiates the BusyBox Top module with the required generic parameters and extra logic. The wrapper also adds and configures the Virtex-4 IO buffers and Digital Clock Manager (DCM) around the BusyBox Top module. Since the wizard that generates the wrapper does not support enabling of the DIFF_TERM attribute of the differential input buffer, it is not included. Instead the input buffer (IBUFGDS) is instantiated in the BusyBox wrapper files where the DIFF_TERM attribute is enabled. This is essential for the design to operate reliable, otherwise the DCM may not lock on the incoming reference clock and the internal clock signals will be full of glitches and spurious behaviour.

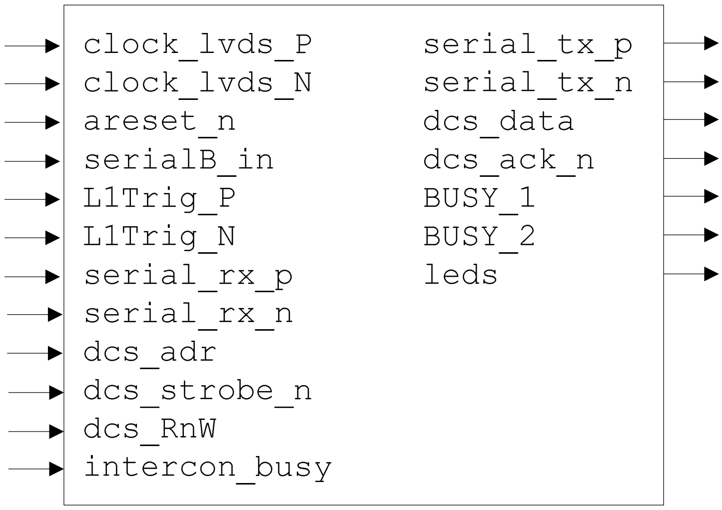


Figure 7-2: Entity for BusyBox FPGA modules.

Table 7-1: I/O details for BusyBox FPGA Modules.

Port Name	Direction	# Bit	Description
clock_lvds_P	Input	1	std_logic;
clock_lvds_N	Input	1	std_logic;
areset_n	Input	1	std_logic;
serialB_in	Input	1	std_logic;
L1Trig_P	Input	1	std_logic;
L1Trig_N	Input	1	std_logic;
serial_rx_p	Input	120	std_logic_vector(0 to num_of_channels);
serial_rx_n	Input	120	std_logic_vector(0 to num_of_channels);
dcs_adr	Input	16	std_logic_vector(15 downto 0);
dcs_strobe_n	Input	16	std_logic_vector(15 downto 0);
dcs_RnW	Input	1	std_logic;
intercom_busy	Input	1	std_logic;
serial_tx_p	Output	1	std_logic_vector(0 to num_of_channels)
serial_tx_n	Output	1	std_logic_vector(0 to num_of_channels)
dcs_data	In/Out	1	std_logic;
dcs_ack_n	Output	1	std_logic;
BUSY_1	Output	1	std_logic;
BUSY_2	Output	1	std_logic;
leds	Output	13	std_logic_vector(1 to 13);

7.3 *BusyBox Top module*

7.3.1 Entity BusyBox Top module

This is the top level structural module of the BusyBox design. All eight main modules are instantiated and connected in this module.

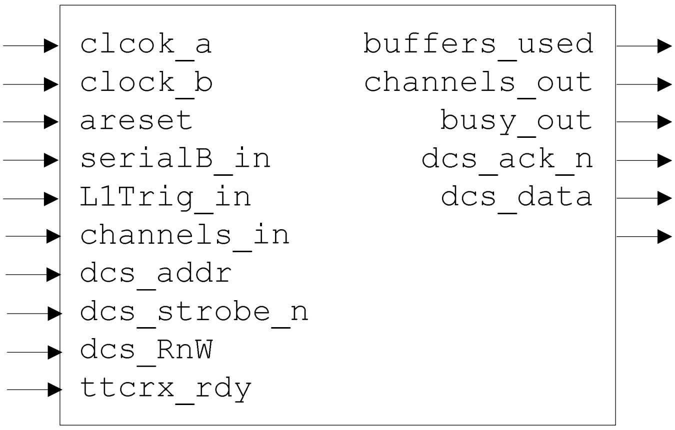


Figure 7-3: Entity for BusyBox top module.

Table 7-2:I/O details for BusyBox Top Module.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
clock_b	Input	1	std_logic; the clock_b frequency is 40 MH;
areset	Input	1	std_logic;
serialB_in	Input	1	std_logic;
L1Trig_in	Input	1	std_logic;
channels_in	Input	120	std_logic_vector(0 to num_of_channels);
dcs_addr	Input	16	std_logic_vector(15 downto 0);
dcs_strobe_n	Input	1	std_logic;
dcs_RnW	Input	1	std_logic;
ttcrx_rdy	Input	1	std_logic;
buffers_used	Output	4	std_logic_vector(3 downto 0);
channels_out	Output	120	std_logic_vector(0 to num_of._channels);
busy_out	Output	1	std_logic;
dcs_ack_n	Output	1	std_logic;
dcs_data	In/Out	16	std_logic_vector(15 downto 0);

7.4 *DCS Bus Arbiter and Address Decoder*

The DCS Bus Arbiter and Address Decoder module is an asynchronous 16 bit data/address handshake protocol for communication between the FPGA and DCS board. This protocol is used to read and write registers in the BusyBox firmware. The MSB of the 16 bits DCS bus address selects which FPGA to communicate with. Then each module can be accessed with the next three bits and the remaining bits are used to target specific sub-module registers.

Table 7-3: Bit-mapping of DCS bus address.

FPGA address	Module address	Sub module address
15	14 – 12	11 – 0

7.4.1 Entity DCS bus arbiter and address decoder

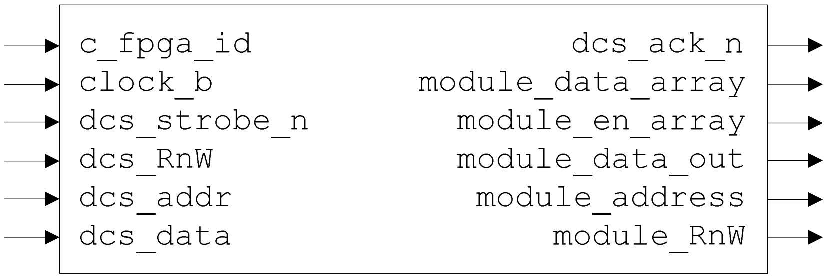


Figure 7-4: Entity for DCS Bus Arbiter and Address Decoder.

Table 7-4: IO details for DCS Bus Arbiter and Address Decoder.

Port Name	Direction	# Bit	Description
c_fpga_id	Input	1	std_logic; '0' fpga1 or '1' fpga2.
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
dcs_strobe_n	Input	1	std_logic; the asynchronous handshake is done with STROBE_N from the DCS board.
dcs_RnW	Input	1	std_logic; '1' read and '0' write.
dcs_addr	Input	16	std_logic_vector(15 downto 0); address module and submodule register.
dcs_data	Inout	16	std_logic_vector(15 downto 0); bi-directional data line.
dcs_ack_n	Output	1	std_logic; the asynchronous handshake is done with ACK_N from the busy board.
module_data_array	Output	7	std_logic_vector(0 to num_of_modules-1); communication with modules.
module_en_array	Output	7	std_logic_vector(0 to num_of_modules-1); communication with modules.
module_address	Output	12	std_logic_vector(11 downto 0); communication with modules.
module_RnW	Output	1	std_logic; communication with modules.

7.5 *Receiver Module*

Serial data from the D-RORCs are handled by the Receiver module and up to 120 single communication channels can be implemented in one FPGA.

Table 7-5: Numbers of channels per detector pr FPGA.

Detector	# Channels on FPGA 1	# Channels on FPGA 2
TPC	120	96
PHOS	20	N/A
FMD	24	N/A
EMCal	3	N/A

Incoming serial data is sampled at 200 MHz by Serial Decoder modules and shifted through a 100 bit shift register (98 bit in firmware due to capture conditions) as shown in Figure 7-5. Each bit is sampled five times and then the middle three bits are run through a MAJ (Majority) gate where the majority bit is selected to be the data bit which is captured as part of the final 16 bit data.

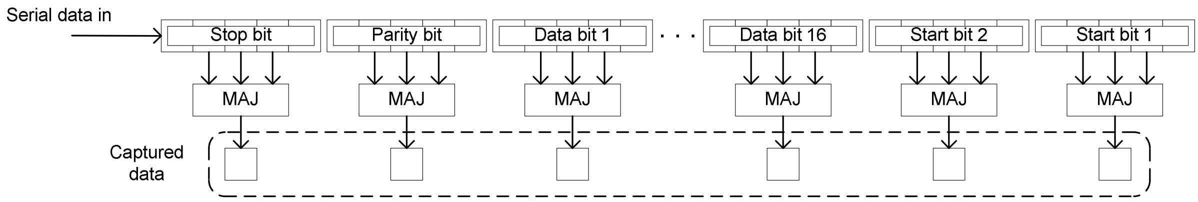


Figure 7-5: Internal architecture of the implementation of a serial decoder.

In order to implement error tolerance, the 48 bit word from the D-RORC is sampled in a 16 bit data frame. A state machine in the Single Channel Receiver module reads out the 16 bit data words, one word after another. Whenever the serial decoder flags that a data word is received it is sent to the Branch Controller. A countdown timer in the state machine discards the data if the strict timing between data readout is compromised. In that case the next word is then considered the first in the readout sequence of three words.

If all three words have been read out successfully to the Brach Controller, and no parity errors and timeouts were found, the state machine will concatenate the three words to a 48 bit message and send it to the Backbone Controller.

Up to sixteen Single Channel Receivers can be connected to a Branch Controller module. The Branch Controller buffers data from the Single Channel Receivers and stops further buffering until data have been read out by the Backbone Controller. The Backbone Controller may have up to eight Branch Controllers and the concept is illustrated in Figure 7-6.

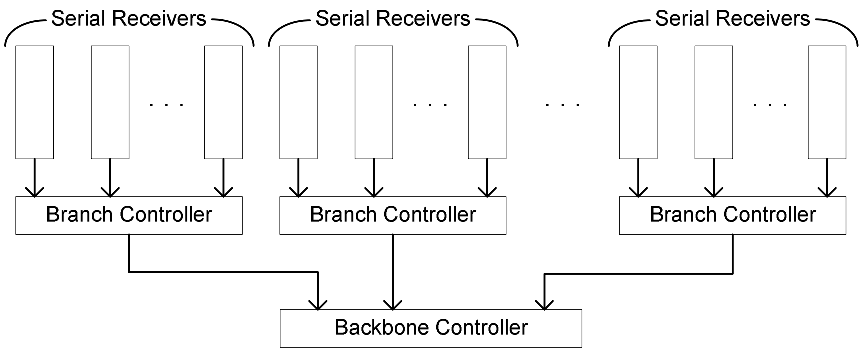


Figure 7-6: Concept of data collector architecture.

7.5.1 Receiver Module VHDL Entity Hierarchy

- Multi Channel Receiver
    - Single Channel Receiver
        - Serial Decoder
    - Branch Controller
    - Backbone Controller

7.5.2 Entity Multi Channel Receiver module

The CHEN (Channel Enable) register tells this module how many serial receivers to enable. Three generics can be set before synthesizing the firmware:

- Numbers of channels (120 max)
- Numbers of branches (8 max)
- Cycles per bit (5 by default)

Based on the CHEN register the Multi Channel Receiver then enables/disables the correct numbers of sub modules to be instantiated. Channels that are not in use will be grounded to avoid electromagnetic noise. This noise would have produced a lot of garbage data if left floating.

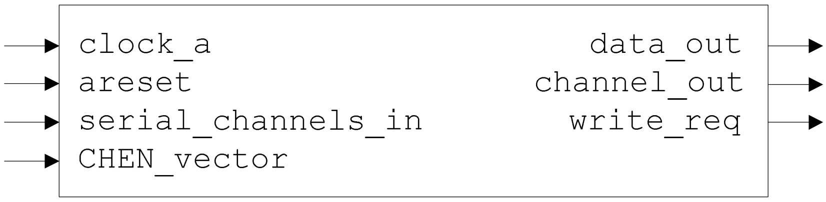


Figure 7-7: Entity for Channel Receiver Module.

Table 7-6: I/O details for Channel receiver Module.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
areset	Input	1	std_logic; asynchronous reset
serial_channel_in	Input	120	std_logic_vector(0 to num_of_channels); LVDS serial channels from D-RORCs
CHEN_vector	Input	120	std_logic_vector(0 to num_of_channels); CHEN vector is a register in the Control and Status Register module , one bit set or disable channels.
data_out	Output	48	std_logic_vector(47 downto 0); 48 bit data from D-RORCs
channel_out	Output	8	std_logic_vector(7 downto 0); toggles the data from the different channels to be outputted
write_req	Output	1	std_logic; ‘1’ D-RORC data ready to send

7.5.3 Entity Single Channel Receiver

A state machine checks for parity errors and make sure that the 16 bit words from the serial decoder is within the allowed time limit. The three 16 bit words are concatenated to a 48 bit message and stored temporary in a registers. If the register is not read out fast enough it will be overwritten.

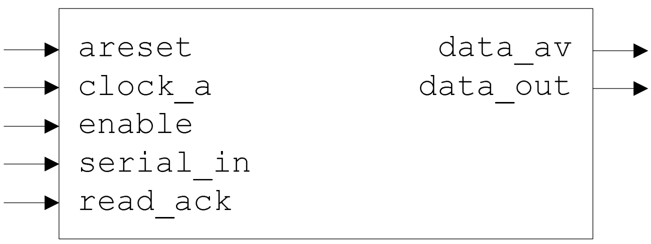


Figure 7-8: Entity for Single Channel Receiver.

Table 7-7: I/O details for Single Channel Receiver.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
Areset	Input	1	std_logic; asynchronous rest.
Enable	Input	1	std_logic;
serial_in	Input	1	std_logic; data bit from serial decoder.
data_out	Output	48	std_logic_vector(47 downto 0); 48 bit data from D-RORC
read_ack	Input	1	std_logic;
data_av	Output	1	std_logic;

7.5.4 Entity Serial Decoder

If the Serial Decoder is enabled by the CHEN register it will wait for the start transition from four 1's to four 0's and the stop condition of three 0's before the data is captured.
Two functions in the busylogic_pkg package, the majority and parity, will take the three middle samples of each bit period[5] and determine the logic value, see Figure 7-5. The parity is then calculated from the extracted 16 bit word and compared with the received parity word. Parity error flag is raised if any parity errors and data available flag is raised when data is available.

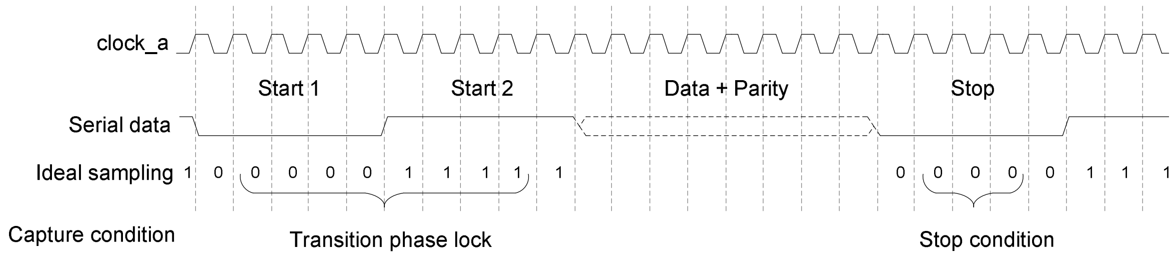


Figure 7-9: Capture conditions for a data frame.

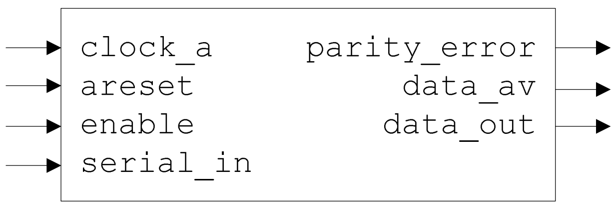


Figure 7-10: Entity for Serial Decoder.

Table 7-8: I/O details for Serial Decoder.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
areset	Input	1	std_logic; asynchronous reset
enable	Input	1	std_logic;
serial_in	Input	1	std_logic; LVDS serial signal from D-RORC
parity_error	Output	1	std_logic;
data_av	Output	1	std_logic;
data_out	Output	16	std_logic_vector(15 downto 0); data from D-RORC

[5] Each bit period is sampled 5 times.

7.5.5 Entity Branch Controller

The Branch Controller reads data from up to 16 Single Channels Receiver's and feed the data to the backbone controller. It scans the receivers for data available flag and copies the data to a buffer when the flag is raised. The branch controller will hold the flag until the Backbone Controller has verified that it has read the message.

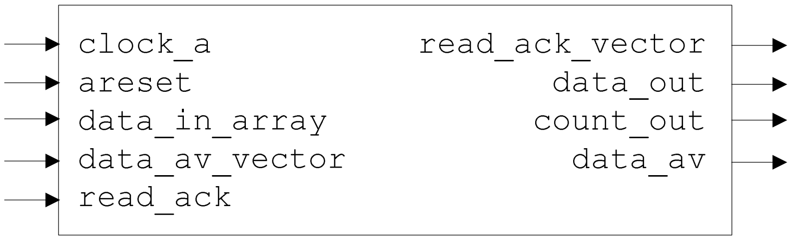


Figure 7-11: Entity for Branch Controller.

Table 7-9: I/O details for Branch Controller.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
areset	Input	1	std_logic; asynchronous reset
data_in_array	Input	16	receiver_busy_array(0 to 15);
data_av_vector	Input	16	std_logic_vector(0 to 15); '1' when data is available
read_ack	Input	1	std_logic; from backbone controller
read_ack_vector	Output	16	std_logic_vector(0 to 15);
data_out	Output	48	std_logic_vector(47 downto 0); 48 bit data
count_out	Output	4	std_logic_vector(3 downto 0); counter to keep track of serial channel being scanned
data_av	Output	1	std_logic; '1' when data from serial receiver is ready to be sent

7.5.6 Entity Backbone Controller

The Backbone Controller reads data from up to 8 Branch Controller's and writes the data to the RX Memory module and the D-RORC inbox buffer in the Event Validator Top module.

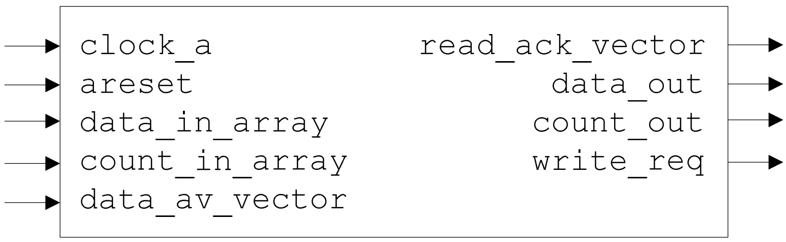


Figure 7-12: Entity for Backbone Controller.

Table 7-10: I/O details for Backbone Controller.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz.
areset	Input	1	std_logic; asynchronous reset
data_in_array	Input	8	receiver_bus_array(0 to 7); work.busylogic_pkg
count_in_array	Input	8	count_array(0 to 7); work.busylogic_pkg
read_ack_vector	Output	8	std_logic_vector(0 to 7);
data_out	Output	48	std_logic_vector(47 downto 0); 48 bit data

Port Name	Direction	# Bit	Description
count_out	Output	8	std_logic_vector(7 downto 0);
data_av_vector	Output	8	std_logic_vector(0 to 7);
write_req	Output	1	std_logic;

7.6 *Transmitter Module*

The transmitter module requests the event ID from the D-RORCs and consists of a controller, a serial encoder and a masking vector. A message register and a channel register are available for the DCS Bus Arbiter and Address Decoder module and the Event ID Verification module. Data from the message register will be loaded into the serial encoder and the masking vector will be created based on the channels enabled in the Channel register. The masking vector lets the Event ID Verification module or the DCS bus module select which channels to enable or disable. The controller handles requests from the Event ID Verification module and the DCS bus module to prevent communication conflicts, but the DCS bus module is just used for debugging purposes.

A state machine in the serial encoder module sends a 16 bit word to the PISO (Parallell In – Serial Out) module by request from the controller.

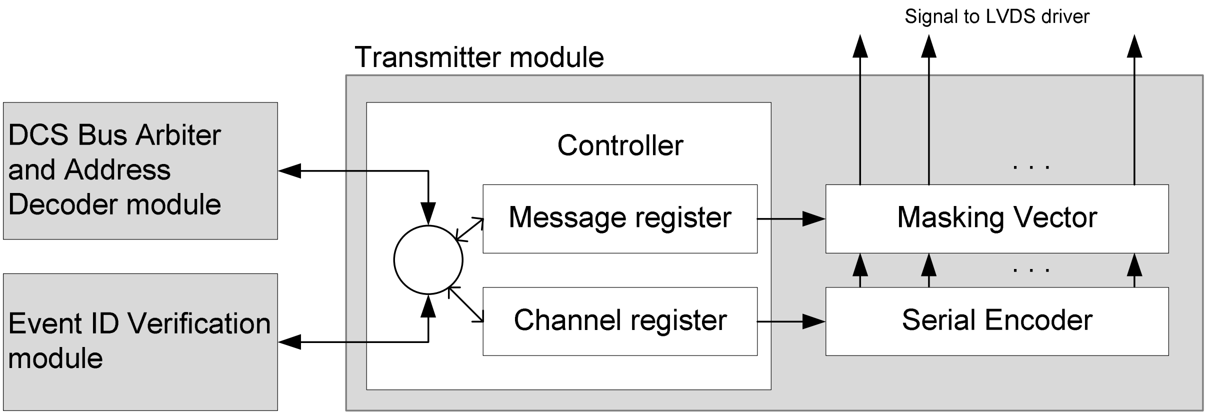


Figure 7-13: Transmitter system.

The Transmitter module will request event IDs from the D-RORCs. The request is a 16 bit word and is sent to all D-RORCs, see Table 7-11 and Table 7-12.

Table 7-11: Bit map for Trigger module request.

15 – 12	11 – 8	7 - 0
Command type	Request ID	Unused

Table 7-12: Request commands.

Command type	Bit Code	Description
Request Event ID	0100	Request an Event ID from the D-RORC.
Resend last message	0101	Command the D-RORC to re-transmit the last message sent.
Force pop Event ID	0110	Command the D-RORC to pop one Event ID from its local queue.
Force Request ID	0111	Command the D-RORC to store the attached Request ID.

7.6.1 Transmitter module VHDL Entity Hierarchy

- Transmitter module
  - Serial encoder
    - PISO

7.6.2 Entity Transmitter module

The Transmitter module is initiating the serial encoder and setting the masking vector. A 16 bit register can be accessed from the DCS bus as shown in Figure 7-13. The register contains a message register and a channel register.

temp_dcs_data	
dcs_tx_channel	dcs_tx_data
15 – 8	7 - 0

Table 7-13: Bit map for DCS data.

The channel register selects which channel to be masked, based on the CHEN register in the Status and Control module, and unmasked the other channels. If the value in the channels register does not specify a specific channel, all channels are unmasked and the message is broadcasted to all channels.

The Event ID module sends a request to the Transmitter module and the request is granted if there is no pending flag from the DCS bus. The controller loads data and the masking vector from the Event ID Verification module.

A flag is raised to indicate if data are available to be written from the DCS board to the message register. A state machine, see Figure 7-14, in the controller sees the flag and starts loading data into the serial encoder and sets the masking vector. The flag is removed and the procedure is executed.

Messages are Hamming coded in the Transmitter module in an 8:4 code applied to the 4 bit command word and request ID. The receiver (D-RORC) will discard data if it finds any errors. The Hamming function is in the busylogic_pkg.

Table 7-14: Hamming code table

Bit position	8 P4	7 D4	6 D3	5 D2	4 P3	3 D1	2 P2	1 P1
P1		X		X		X		P1
P2		X	X			X	P2	
P3		X	X	X	P3			
P4	P4	X	X	X	X	X	X	X

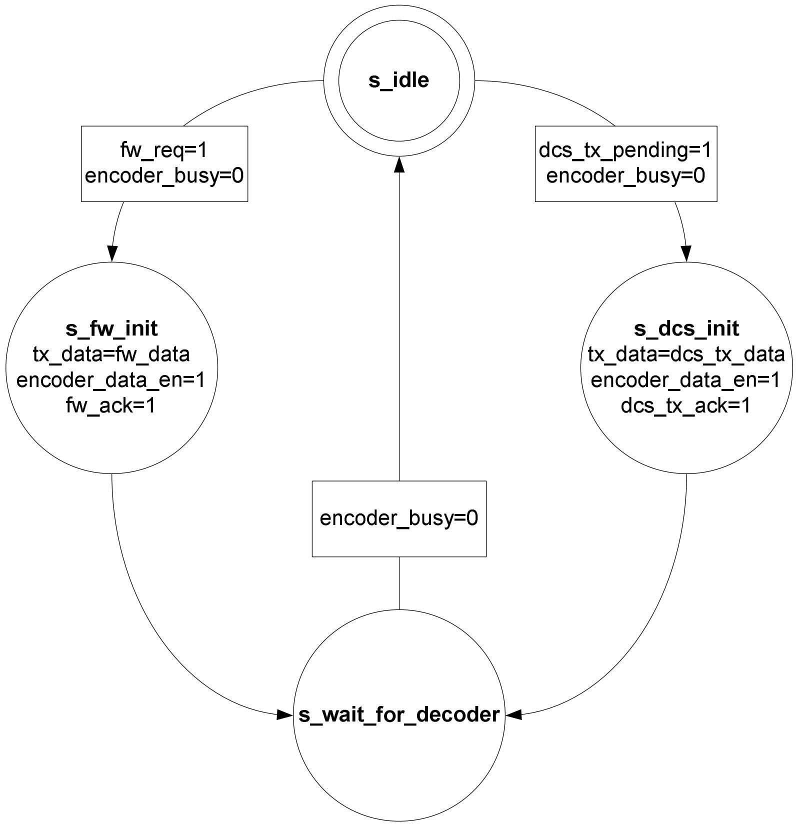


Figure 7-14: State diagram for TX controller

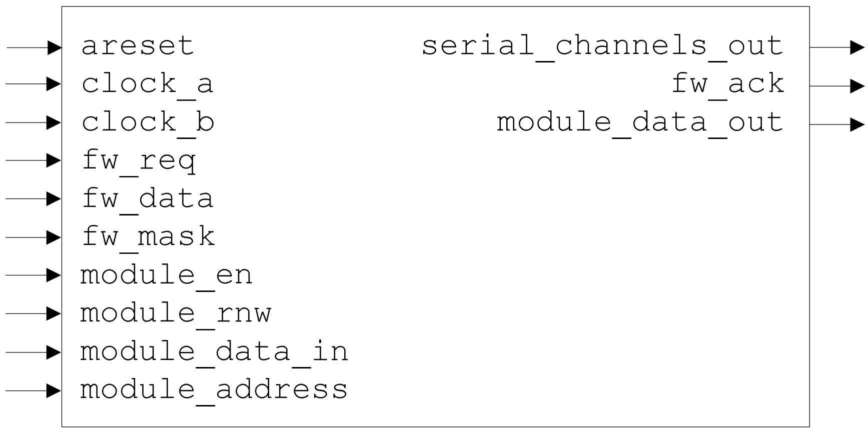


Figure 7-15: Entity for Transmitter Module.

Table 7-15: I/O details for Transmitter Module.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous reset
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
fw_req	Input	1	std_logic;
fw_data	Input	8	std_logic_vector(7 downto 0);
fw_mask	Input	120	std_logic_vector(0 to num_of_channels);
module_en	Input	1	std_logic;
module_rnw	Input	1	std_logic;
module_data_in	Input	16	std_logic_vector(15 downto 0);
module_address	Input	12	std_logic_vector(11 downto 0);

Port Name	Direction	# Bit	Description
serial_channels_out	Output	120	std_logic_vector(0 to num_of_channels);
fw_ack	Output	1	std_logic;
module_data_out	Output	16	std_logic_vector(15 downto 0); 16 bit request data to D-RORCs

7.6.3 **Entity Serial Encoder**

Encodes the command type word which is sent to the D-RORCs

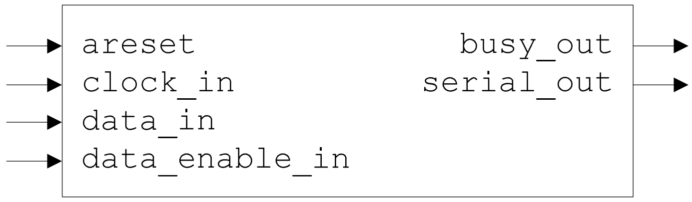


Figure 7-16: Entity for Serial Encoder.

Table 7-16: I/O details for Serial Encoder.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous reset
clock_in	Input	1	std_logic; ; the clock_in frequency is 200 MHz
data_in	Input	1	std_logi_vectorc;
data_enable_in	Input	1	std_logic;
busy_out	Output	1	std_logic;
serial_out	Output	1	std_logic;

7.6.4 **Entity PISO (Parallel In – Serial Out)**

Serialize the command type message which is sent to the D-RORCs.

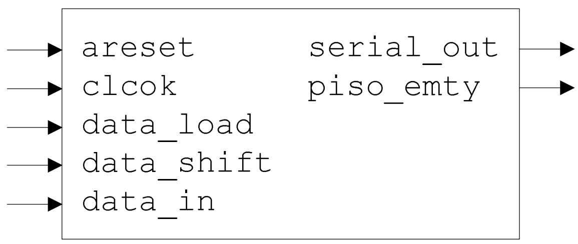


Figure 7-17: Entity for PISO.

Table 7-17: I/O details for PISO.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous reset
clock	Input	1	std_logic; ; the clock frequency is 200 MHz
data_load	Input	1	std_logic;
data_shift	Input	1	std_logic;
data_in	Input	1	std_logic_vector;
serial_out	Output	1	std_logic;
piso_emty	Output	1	std_logic;

7.7 *RX Memory Module*

The BusyBox can store up to 1024 D-RORC messages from the Receiver module in the RX Memory module. Four BRAM modules are instantiated in the FPGA and can be accessed from both clock domains[6]. Data from the Receiver module is 56 bit and is written into memory at the address given by a 10 bit counter. The DCS bus is limited to read 16 bit at a time, and needs four read operations to get the whole word from memory. The RX Memory module can be written to by the DCS bus for testing and verification purposes.

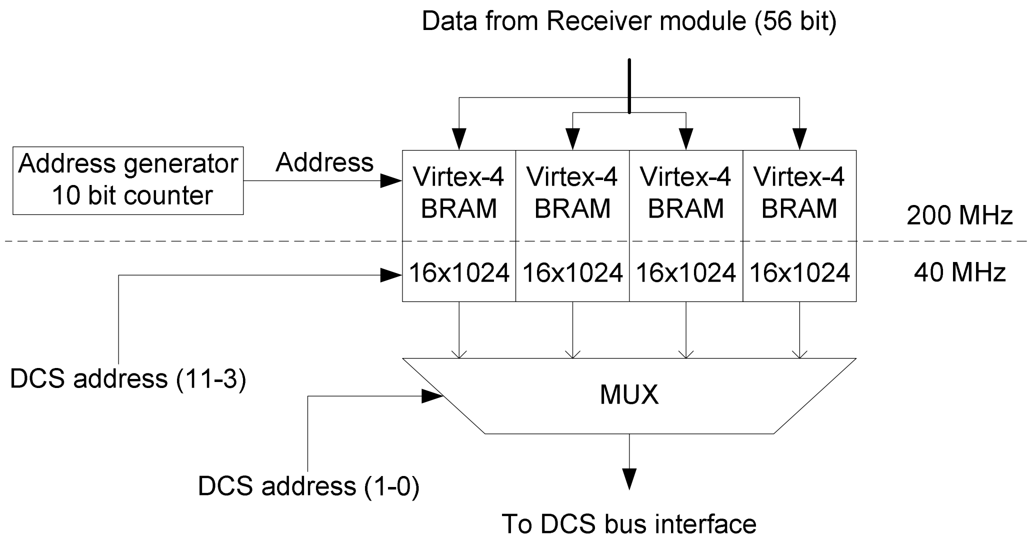


Figure 7-18: Illustration of the RX Memory module.

7.7.1 Entity RX Memory Module

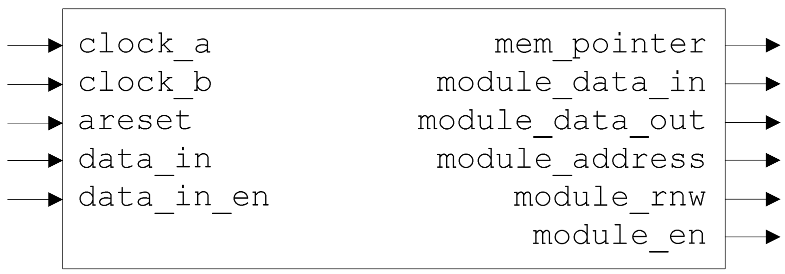


Figure 7-19: Entity for RX Memory Module.

Table 7-18: I/O details for RX Memory Module.

Port Name	Direction	# Bit	Description
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
areset	Input	1	std_logic; asynchronous reset
data_in	Input	64	std_logic_vector(63 downto 0);
data_in_en	Input	1	std_logic;
mem_pointer	Output	10	std_logic_vector(9 downto 0);
module_data_in	Output	16	std_logic_vector(15 downto 0);

[6] The Receiver module operates in the 200 MHz domain while the internal logic of the BusyBox runs in the 40 MHz domain.

Port Name	Direction	# Bit	Description
module_data_out	Output	16	std_logic_vector(15 downto 0);
module_address	Output	12	std_logic_vector(11 downto 0);
module_rnw	Output	1	std_logic;
module_en	Output	1	std_logic;

7.8 *RX Memory Filter Module*

The RX Memory filter can be used to filter which messages from specific channels will trigger the write enable signal form the RX Memory Module. Each message from the Receiver Module will have an 8 bit channel number appended to it. Each individual bit of this 8 bit word can be compared with bits in a register in the RX Memory Filter that is accessible from the DCS bus interface. The RX Memory Filter has registers with 16 bits. The first 8 bits are used to toggle matching individual bits. The last 8 bits are the bits that will be compared with the channel number bits of the message. This feature makes it easier to see the response of only a subset of channels in the RX Memory without disabling the other channels in the CHEN registers.

7.8.1 Entity RX Memory Filter Module

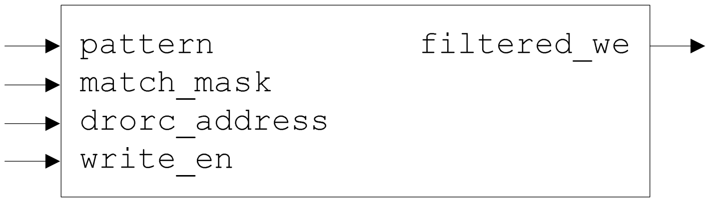


Figure 7-20: Entity for RX Memory Filter.

Table 7-19: I/O details for RX Memory Filter.

Port Name	Direction	# Bit	Description
pattern	Input	8	std_logic_vector(7 downto 0);
match_mask	Input	8	std_logic_vector(7 downto 0);
drorc_address	Input	8	std_logic_vector(7 downto 0);
write_en	Input	1	std_logic;
filtered_we	Output	1	std_logic;

7.9 *Trigger Receiver Module*

The Trigger Receiver module is responsible for decoding all the information sent from the Central Trigger Processor (CTP). The information is sent on two communication lines, L1 accept line and Serial B line, from the TTCrx chip which converted the optical information from CTP (distributed by LTU). The L0 and L1a triggers are transmitted on the L1 accept line while the trigger messages, L1 accept message, L2 accept message and L2 reject message are sent on the Serial B line. All information sent is synchronous with the LHC clock.

L1 accept line is decoded by the Channel A decoder and the L0 and L1 triggers are sent to the Busy Controller. Serial B line is decoded by the Channel B decoder, checked for hamming errors, address messages are then decoded and validated before a CDH header is generated and stored in a CDH FIFO.

The CDH header holds the event ID used by the Event ID Verification module to verify an event readout from the Fee. A *buffered events* counter is outputted from the CDH FIFO to notify the Event ID Verification module that an event ID is ready to be read out once the *buffered events* counter is incremented. See Figure 7-22 and the *TTC receiver requirement specification* document for more information [2].

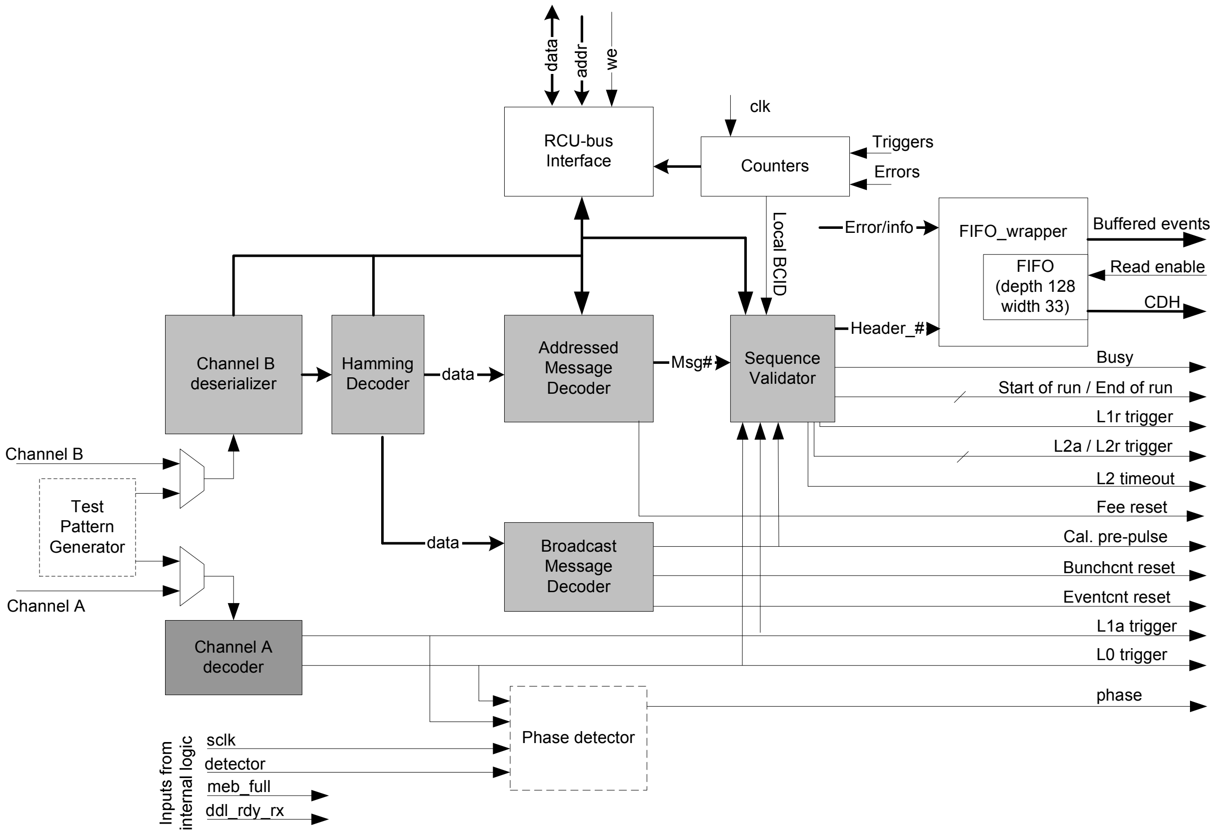


Figure 7-21: Block diagram of the Trigger Receiver module. From [2].

7.9.1 **Entity Trigger Receiver Module**

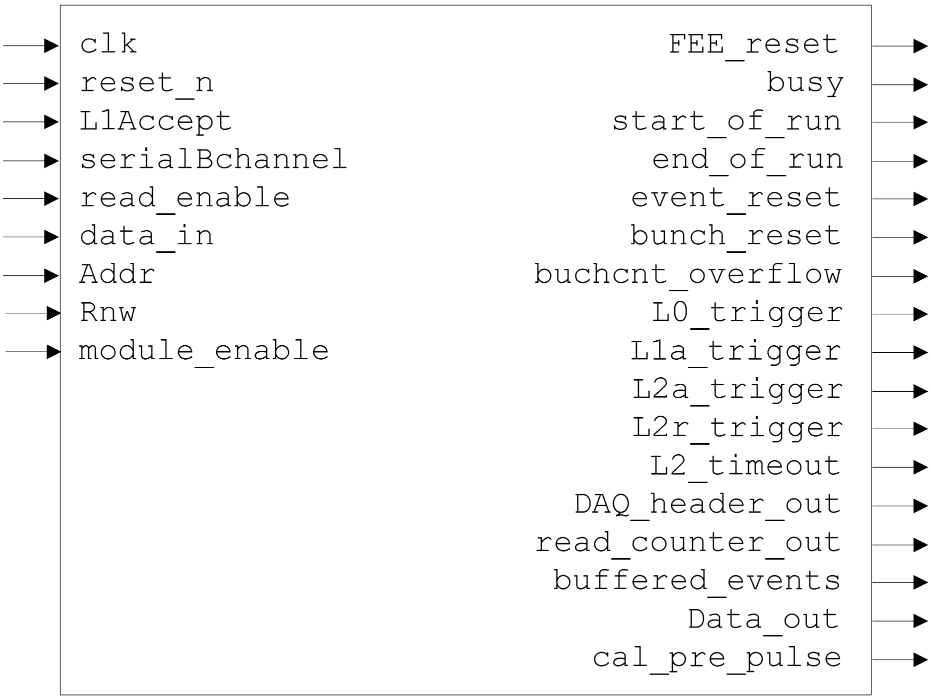


Figure 7-22: Entity for Trigger Receiver Module.

Table 7-20: I/O details for Trigger Receiver Module.

Port Name	Direction	# Bit	Description
clk	Input	1	std_logic; the clk frequency is 40.08 MHz
reset_n	Input	1	std_logic;
L1Accept	Input	1	std_logic;
serialBchannel	Input	1	std_logic;
read_enable	Input	1	std_logic;
data_in	Input	16	std_logic_vector(15 downto 0);
addr	Input	12	std_logic_vector(11 downto 0);
rnw	Input	1	std_logic;
module_enable	Input	1	std_logic;
FEE_reset	Output	1	std_logic; N/A
busy	Output	1	std_logic;
cal_pre_pulse	Output	1	std_logic; N/A
start_of_run	Output	1	std_logic; N/A
end_of_run	Output	1	std_logic; N/A
event_reset	Output	1	std_logic; N/A
bunch_reset	Output	1	std_logic; N/A
bunchcnt_overflow	Output	1	std_logic; N/A
L0_trigger	Output	1	std_logic;
L1a_trigger	Output	1	std_logic;
L2a_trigger	Output	1	std_logic;
L2r_trigger	Output	1	std_logic;
L2_timeout	Output	1	std_logic;
DAQ_header_out	Output	33	std_logic_vector(32 downto 0);
read_counter_out	Output	4	std_logic_vector(3 downto 0);
buffered_events	Output	4	std_logic_vector(3 downto 0);
data_out	Output	16	std_logic_vector(15 downto 0);

7.10 *Event ID Verification Module*

The Trigger Receiver module's CDH FIFO is constantly monitored by the Event ID Verification module. Data from an L2a/L2r or L2 timeout trigger is stored in the CDH format in the FIFO and will be read out by the Event ID Queue module.

The event controller requests the Transmitter module to read out the data and send it to the D-RORCs. The Receiver module forwards the received D-RORC data to the D-RORC Inbox Buffer. The Inbox operates in both frequency domains[7] and makes the data available for the Event processor, which compares the event ID.

The Event Processor has a register called EIDOK (Event ID OK), and together with the CHEN vector it compares the two event IDs from the Event ID Queue module and the D-RORC Inbox buffer. If the ID matches, the verification gate will assert an event verified signal. An overview of the ID verification model is shown in Figure 7-25.

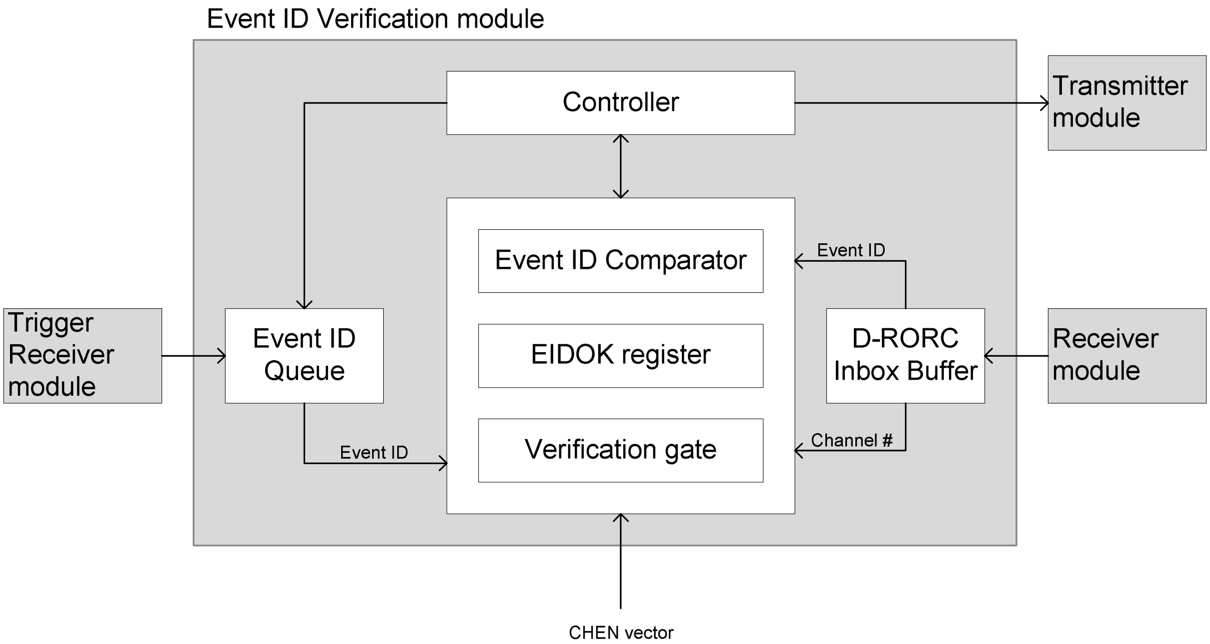


Figure 7-23: Overview of the Event ID Verification module.

[7] The Receiver module operates in the 200 MHz domain while the internal logic of the verification module runs in the 40 MHz domain.

7.10.1 **Event Id Verification VHDL Entity Hierarchy**

- Event Validator
  - Trigger EventID Queue
    - EventID FIFO
    - EventID Extractor
  - EventID Control
  - Event Processor
  - D-RORC Inbox Buffer

7.10.2 **Entity Event Validator**

This is the top module that concatenates all the sub modules and also instantiates the D-RORC Inbox Buffer.

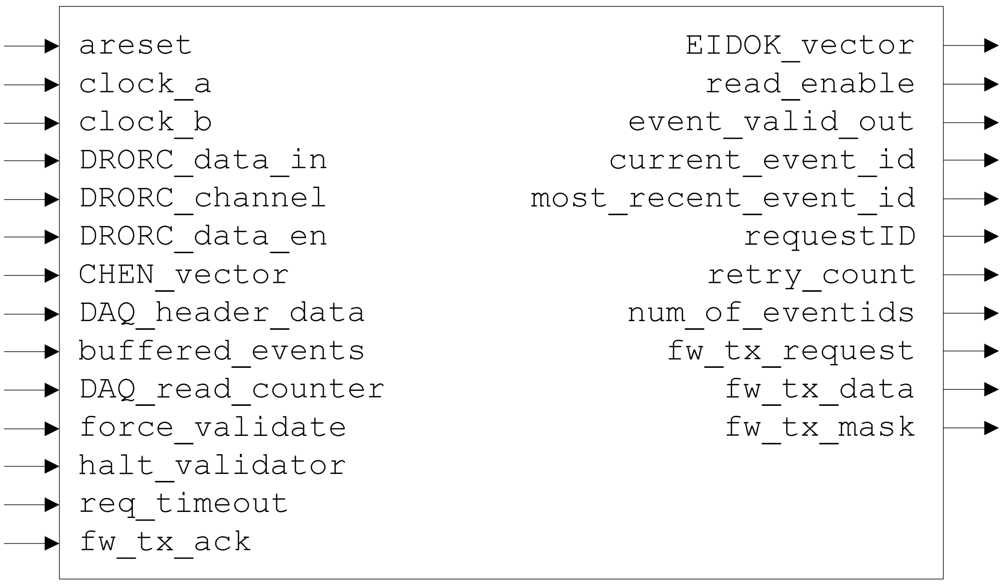


Figure 7-24: Entity for Event Validator.

Table 7-21: I/O details for Event Validator.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous reset
clock_a	Input	1	std_logic; the clock_a frequency is 200 MHz
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
DRORC_data_in	Input	48	std_logic_vector(47 downto 0);
DRORC_channel	Input	8	std_logic_vector(7 downto 0);
DRORC_data_en	Input	1	std_logic;
CHEN_vector	Input	120	std_logic_vector(0 to num_of_channels);
DAQ_header_data	Input	33	std_logic_vector(32 downto 0);
buffered_events	Input	4	std_logic_vector(3 downto 0);
DAQ_read_counter	Input	4	std_logic_vector(3 downto 0);
force_validate	Input	1	std_logic;
halt_validator	Input	1	std_logic;

Port Name	Direction	# Bit	Description
req_timeout	Input	16	std_logic_vector(15 downto 0);
fw_tx_ack	Input	1	std_logic;
EIDOK_vector	Output	120	std_logic_vector(0 to num_of_channels);
read_enable	Output	1	std_logic;
event_valid_out	Output	1	std_logic;
current_event_id	Output	36	std_logic_vector(35 downto 0);
most_recent_event_id	Output	36	std_logic_vector(35 downto 0);
requestID	Output	4	std_logic_vector(3 downto 0);
retry_count	Output	16	std_logic_vector(15 downto 0);
num_of_eventids	Output	4	std_logic_vector(3 downto);
fw_tx_request	Output	1	std_logic;
fw_tx_data	Output	8	std_logic_vector(7 downto 0);
fw_tx_mask	Output	120	std_logic_vector(0 to num_of_channels);

7.10.3 **Entity Trigger EventID Queue**

This module is a structural architecture to concatenate the EventID FIFO and the EventID Extractor. The Trigger EventID Queue extracts the bunchcount and orbit ID from the CDH message stored in the Trigger Receiver CDH FIFO. It then forwards the two messages to the Event Processor for comparison with the bunchcount and orbit ID from the all the D-RORCs.

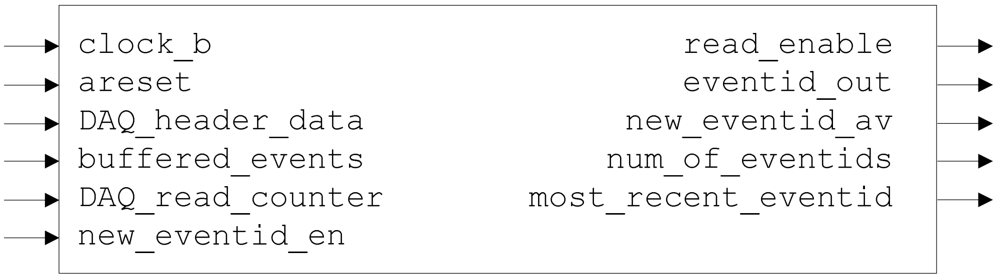


Figure 7-25: Entity for Trigger EventID Queue.

Table 7-22: I/O details for Trigger EventID Queue.

Port Name	Direction	# Bit	Description
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
areset	Input	1	std_logic; asynchronous reset
DAQ_header_data	Input	33	std_logic_vector(32 downto 0);
buffered_events	Input	4	std_logic_vector(3 downto 0);
DAQ_read_counter	Input	4	std_logic_vector(3 downto 0);
new_eventid_en	Input	1	std_logic;
read_enableventid_out	Output	1	std_logic;
eventide_out	Output	1	std_logic;
new_eventid_av	Output	1	std_logic;
num_of_eventids	Output	4	std_logic_vector(3 downto 0);
most_recent_eventid	Output	36	std_logic_vector(35 downto 0);

7.10.4 **Entity EventID Extractor**

An L2a trigger generates a CDH message in the Trigger Receiver module with the bunchcount and orbit ID of the event. The CDH message holds more information than just these two messages. Thus, the EventID Extractor needs to sort through the CDH message to get the information needed. When the messages are read out they are sent for verification in the Event Processor module. The messages are also forwarded to the Control and Status Register module.

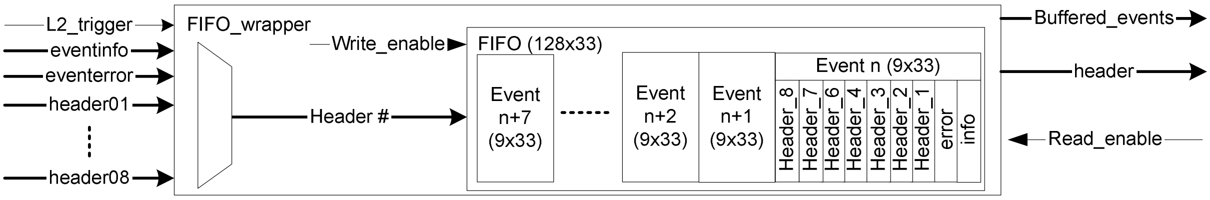


Figure 7-26: Structure of the CDH FIFO. From [2]

The figure above shows the structure of the CDH FIFO and the Event message. Header_1 (12 bit) contains the bunchcross ID and Header_2 (24 bit) the orbit ID. The two words are then concatenated to a 36 bit word named event ID.

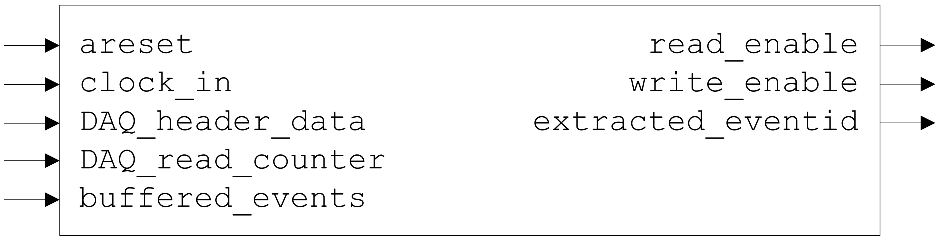


Figure 7-27: Entity for EventID Extractor.

Table 7-23: I/O details for EventID Extractor.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous reset
clock_in	Input	1	std_logic; the clock_in frequency is 40.08 MHz
DAQ_header_data	Input	33	std_logic_vector(32 downto 0); 33 bit word
DAQ_read_counter	Input	4	std_logic_vector(3 downto 0); counts through the 9 words in the CDH message
buffered_events	Input	4	std_logic_vector(3 downto 0); counts numbers of buffered evnts in the FIFO
read_enable	Output	1	std_logic;
write_enable	Output	1	std_logic;
extracted_eventid	Output	36	std_logic_vector(35 downto 0); the extracted orbit end bunch cross IDs

7.10.5 **Entity Event ID Control**

The Event ID Control module is a state machine that monitors and controls the event verification process. Under is a state diagram of the controller.

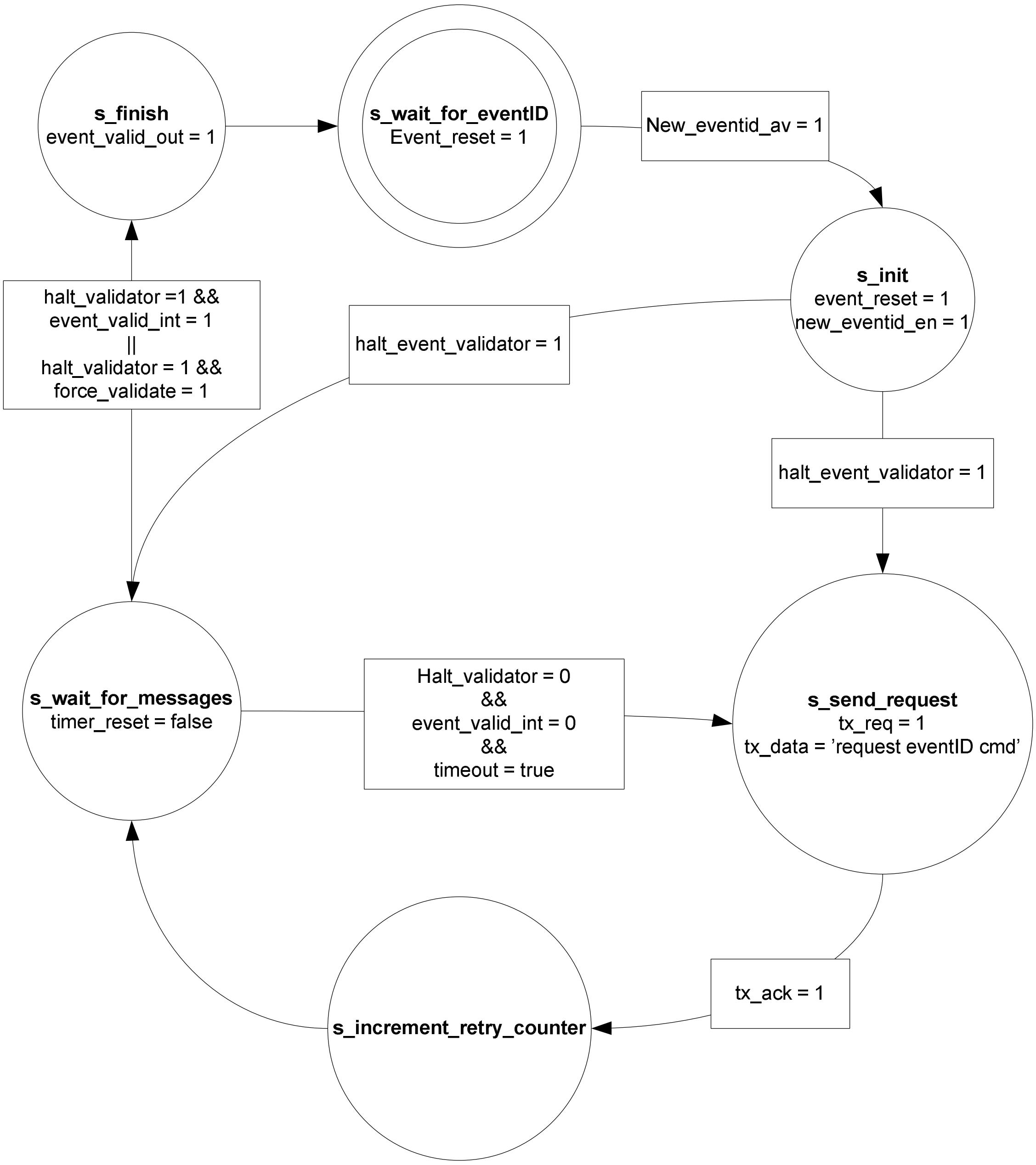


Figure 7-28: State diagram for EventID Controller.

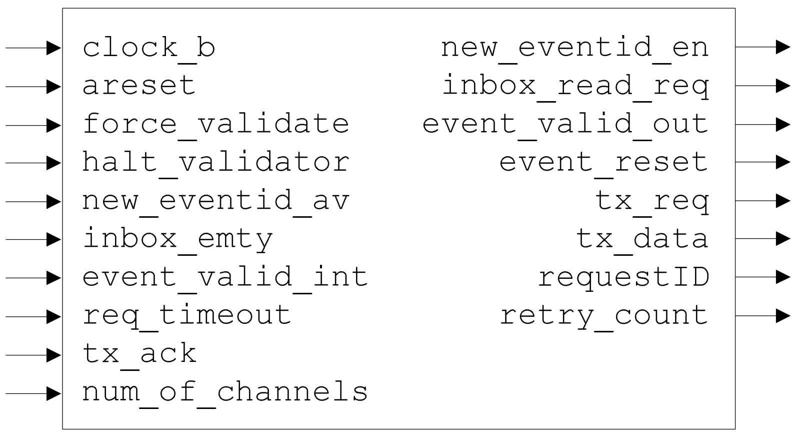


Figure 7-29: Entity for EventID Control.

Table 7-24: I/O details for EventID Control.

Port Name	Direction	# Bit	Description
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
areset	Input	1	std_logic; asynchronous reset
force_validate	Input	1	std_logic;
halt_validator	Input	1	std_logic;
new_eventid_av	Input	1	std_logic;
inbox_emty	Input	1	std_logic;
event_valid_int	Input	1	std_logic;
req_timeout	Input	16	std_logic_vector(15 downto 0);
tx_ack	Input	1	std_logic;
new_evetid_en	Output	1	std_logic;
inbox_read_req	Output	1	std_logic;
event_valid_out	Output	1	std_logic;
event_reset	Output	1	std_logic;
tx_req	Output	1	std_logic;
tx_data	Output	8	std_logic_vector(7 downto 0);
requestID	Output	4	std_logic_vector(3 downto 0);
retry_count	Output	16	std_logic_vector(15 downto 0);

7.10.6 Entity Event ID Processor

In this module all the verification occurs and based on the CEHN register it will continuously compare the event IDs and set each individual channel with '1' if match or '0' if mismatch in a register called EIDOK. A verification gate will flag an event verified signal if either the CHEN register is disabled or all channels where checked in the EIDOK register.

Figure 7-30: Entity for EventID Processor.

Table 7-25: I/O details for EventID Processor.

Port Name	Direction	# Bit	Description
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
areset	Input	1	std_logic; asynchronous reset
trigger_eventid	Input	36	std_logic_vector(35 downto 0);
DRORC-message	Input	56	std_logic_vector(55 downto 0);
CHEN_vector	Input	120	std_logic_vector(0 to num_of_channels);
local_requestID	Input	4	std_logic_vector(3 downto 0);
event_reset	Input	1	std_logic;
EIDOK_vector	Output	120	std_logic_vector(0 to num_of_channels);
tx_mask	Output	120	std_logic_vector(0 to num_of_channels);
event_valid	Output	1	std_logic;

7.11 *Busy Controller Module*

There are four conditions that sets the busy signal high and only one have to be true to set the *busy*. The TTCrx ready (ttcrx_rdy) is added to the BusyBox since each sub-detector should report busy if this is not asserted. If there is a physical problem with the connection to the LTU or the CTP is issuing a global reset, the busy is set [JohanA]. Every time a L0 trigger is detected a countdown timer (timeout_active) starts and the *busy* is set for a pre set time period. The busy time can be set manually with a register in the Control and Status Register module.

The TPCs Fee starts buffering data when a L1 trigger is issued and the other detectors starts buffering upon a L0 trigger. The FEE can buffer 4 or 8 events depending on how many samples are configured in the ALTRO chip and the *busy* is set when the buffers are full.

The Busy Controller module increment a register (buffer_count) when a L0 is detected (L1 for TPC), decrements the register when a L2 Reject trigger is asserted and when the Event ID Verification module asserts the event valid signal.

7.11.1 Entity Busy Controller Module

Figure 7-31: Entity for Busy Controller Module.

Table 7-26: I/O details for Busy Controller Module.

Port Name	Direction	# Bit	Description
areset	Input	1	std_logic; asynchronous resets
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz.
ttcrx_rdy	Input	1	std_logic; ttcrx_rdy out from dcs_ctrl7 (physical line on the DCS-RCU connector). If not asserted it implies a physical problem with connection to the LTU, or that the CTP is issuing a global reset via the TTCrx.
L0_trigger	Input	1	std_logic; N/A
L1a_trigger	Input	1	std_logic; L1a_trigger output from trigger_receiver_busy_model. Starts buffering data in Fee if L1a_trigger signal is asserted.
L2a_trigger	Input	1	std_logic; N/A
L2r_trigger	Input	1	std_logic; L1r_trigger output from trigger_receiver_busy_model. Overwrites buffers in Fee if L2r_trigger signal is asserted.

Port Name	Direction	# Bit	Description
L2_timeout	Input	1	std_logic; L2_timeout output from trigger_receiver_busy_model. Overwrites buffers in Fee if L2_timeout signal is asserted.
busy_triggermodule	Input	1	std_logic; busy_triggermodule output from trigger_receiver_module. Asserted when Fee buffers are full (4-8 depending on the number of samples configured in the ALTRO).
event_valid	Input	1	std_logic;
trig_timeout	Input	16	std_logic_vector(15 downto 0); programmable timeout following the start of a trigger sequence. 10 us resolution. Register 0x2008 in Control and Status Register. Set Register to A (10 decimal) to get 100 us timeout.
fee_buffers_available	Input	4	std_logic_vector(3 downto 0); Holds the numbers of buffers assumed on the FEE. Register 0x2009. Default is 4.
busy_out	Output	1	std_logic; busy_out is asserted when busy conditions are met.
fee_buffers_used	Output	4	std_logic_vector(3 downto 0);
busy_time	Output	32	std_logic_vector(31 downto 0); busy_time count numbers of clock cycles busy signal is asserted.

7.12 *Control and Status Registers*

This module has information about register and control signals available for the BusyBox. See chapter 3 for more information.

7.12.1 Entity Control and Status Register

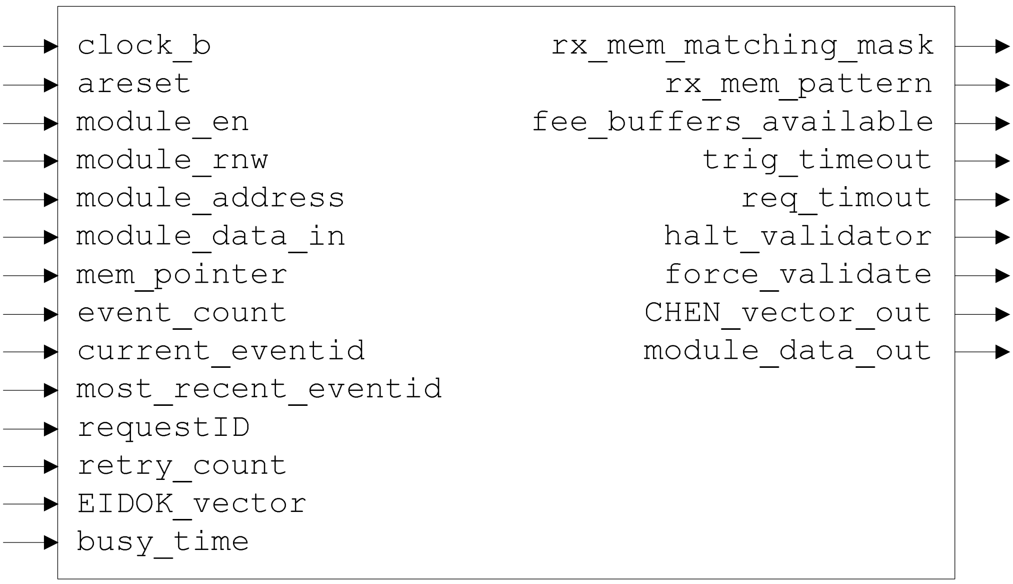


Figure 7-32: Entity for Control and Status Registers.

Table 7-27: I/O details for Control and Status Registers.

Port Name	Direction	# Bit	Description
clock_b	Input	1	std_logic; the clock_b frequency is 40 MHz
areset	Input	1	std_logic; asynchronous resets

Port Name	Direction	# Bit	Description
module_en	Input	1	std_logic;
module_rnw	Input	1	std_logic;
module_address	Input	12	std_logic_vector(11 downto 0);
module_data_in	Input	16	std_logic_vector(15 downto 0);
mem_pointer	Input	10	std_logic_vector(9 downto 0);
event_count	Input	4	std_logic_vector(3 downto 0);
current_eventid	Input	36	std_logic_vector(35 downto 0);
most_recent_eventid	Input	36	std_logic_vector(35 downto 0);
requestID	Input	4	std_logic_vector(3 downto 0);
retry_count	Input	16	std_logic_vector(15 downto 0);
EIDOK_vector	Input	120	std_logic_vector(0 to num_of_channels);
busy_time	Input	32	std_logic_vector(31 downto 0);
module_data_out	Output	16	std_logic_vector(15 downto 0);
rx_mem_matching_mask	Output	8	std_logic_vector(7 downto 0);
rx_mem_pattern	Output	8	std_logic_vector(7 downto 0);
fee_buffers_available	Output	4	std_logic_vector(3 downto 0);
trig_timeout	Output	16	std_logic_vector(15 downto 0);
req_timout	Output	16	std_logic_vector(15 downto 0);
halt_validator	Output	1	std_logic;
force_validate	Output	1	std_logic;
CHEN_vector_out	Output	120	std_logic_vector(0 to num_of_channels);

8 BusyBox Hardware

This chapter discusses the hardware on the BusyBox PCB.

8.1 *PCB*

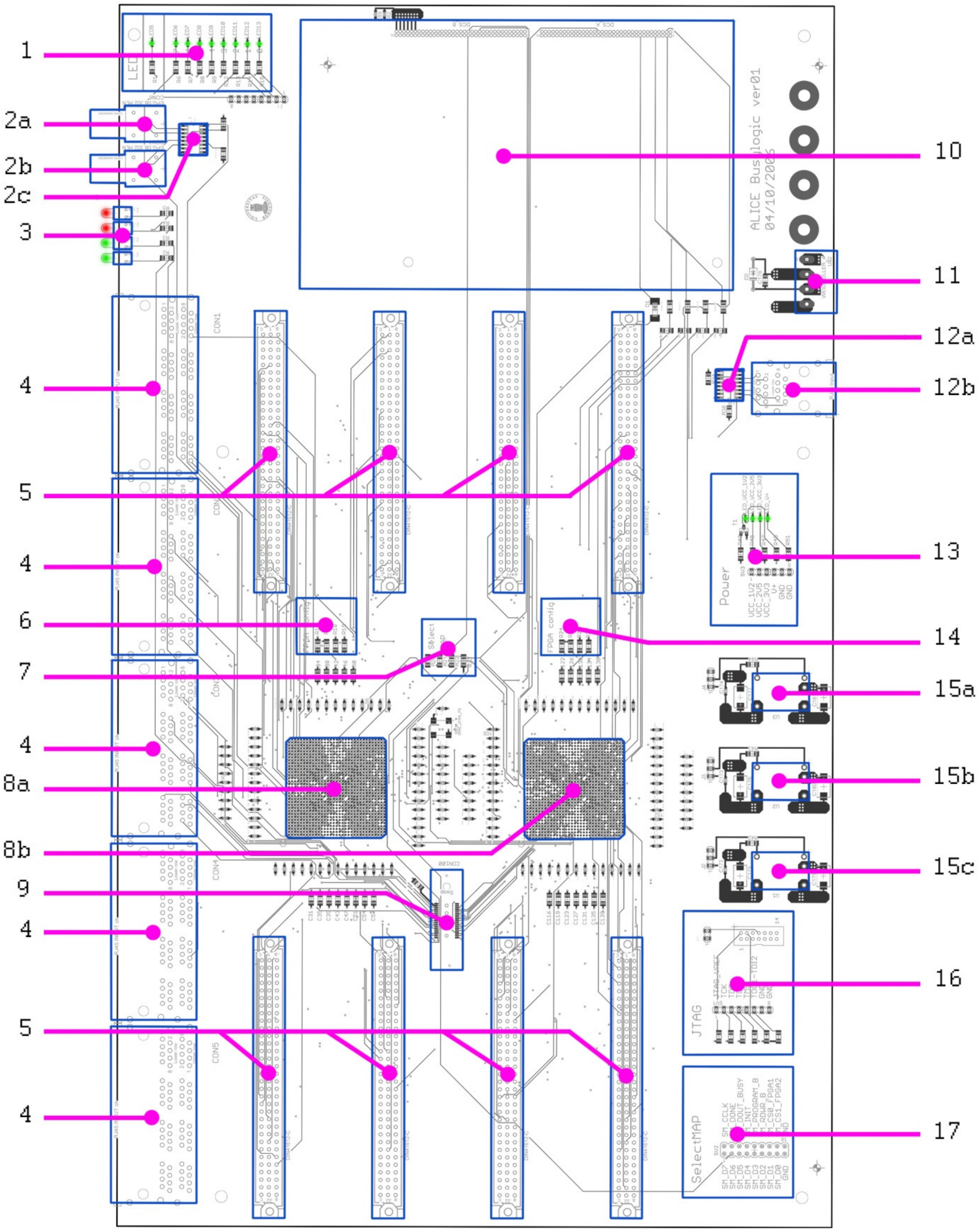


Figure: 8-1: PCB layout of BusyBox.

Table 8-1: List of components on the PCB.

#	Type	Description
1	LED indicators	-
2a	LEMO contact	Lemo EPG.0b.302.HLN
2b	LEME contact	Lemo EPG.0b.302.HLN
2c	LVDS driver	TI SN65LVDM31
3	LED indicators	-
4	RJ-45 contact	RJ-45 contacts for D-RORCs
5	Mezzaine connectors	Mezzaine card holders to additional RJ-45 connectors
6	FPGA 1/solo config	-
7	Select MAP	-
8a	FPGA 1	Xilinx Virtex IV
8b	FPGA 2	Xilinx Virtex IV (TPC only)
9	38-Pin Low-Voltage Probe	Agilent Technologies E5339A
10	DCS board	Connectors for DCS board
11	Power supply connector	Connector for external power supply. Power supply: 5V, 12 A. XP POWER Model: ECM60US05
12a	LVDS driver	-
12b	RJ-45 connector	Block with two RJ-45 connectors
13	Power interface	GND, 1.2V, 2.5V and 3.3V output
14	FPGA 2 config	
15abc	Voltage regulator	PTH05000W voltage regulators from Texas Instruments
16	JTAG interface	-
17	SelectMap interface	-

8.2 *Description Details*

7.2.1 Led Indicators

The BusyBox board has four LEDs {3} in front and nine LEDs {1} in the upper left corner. The four LEDs in front, two red (busy) and two green, are used as indicators under system operations and the other nine green LEDs in indicates how many of the Fee buffers are occupied.

7.2.2 LVDS

The *busy* output is an LVDS output to two LEMO {2a, 2b} connectors in front and goes via a LVDS driver {2c} from FPGA1. The driver is an IC with four LVDS drivers, but only two are used. In the back of the board a block with two RJ-45 {12b} connectors with an LVDS driver {12a} and with the same setup as for the LEMO connectors in front.

7.2.3 Mezzanine card for RJ-45 connectors

Mounted on the BusyBox board are five blocks with eight RJ-45 {4} connectors used for communication with the D-RORCs. Four mezzanine cards can be connected to contact points {5} on the BusyBox board and each mezzanine cards holds six blocks with eight RJ-45 connectors in each block.

7.2.4 FPGA, SelectMAP and JTAG

The BusyBox use the Virtex-4 LX-40 {8a, 8b} with the ff1148 package from Xilinx. There are 640 user programmable I/O pins that support LVDS 2.5 standard used to communicate with the D-RORCs. The Virtex-4 can run on clock speeds up to 500 MHz, store 18 Kbits in 96 BRAM modules and has DCM to provide flexible clocking and synchronization.

A “Multiple device SelectMap bus” is used to programme the FPGAs, since two FPGAs can be used with different firmware. Linux kernel device drivers have been developed so that the Linux OS running on the DCS board can redirect the programming bit file to the FPGA. There is a SelectMAP {17} interface on the BusyBox board which can be used to program and read data from the configuration memory on the FPGA(s).

The BusyBox can also be programmed via JTAG {16} interface on the PCB. When one FPGA is used a jumper on the PCB needs to be applied to bypass the missing JTAG chain.

7.2.5 Voltage Regulators and Power Supply

The BusyBox has three voltage regulators {15a, 15b, 15c} to supply power to the BusyBox electronics. These regulators can be controlled via control inputs connected to the DCS board via jumpers J1, J2 and J3. The jumpers select whether the voltage regulators; are controlled by the DCS, always on or always off.

The power is supplied {11} from an XP Power AC-DC converter with 230 VAC input and 5 V/60 W output.

9 BusyBox DCS board

The DCS board was originally designed for the TRD and TPC sub-detector, but because it was very versatile it has been adapted for the BusyBox and other instrument in ALICE experiment. It is running a lightweight version of Linux and implements TCP/IP network protocol and UART interface. The DCS board has a TTCrx chip to receive the LHC clock, first level trigger accept and trigger messages. Each card runs a Fee server that interfaces with the system it is connected to. Thus, it makes it possible to program the FPGA(s) and read/write registers remotely from any location with an Ethernet connection.

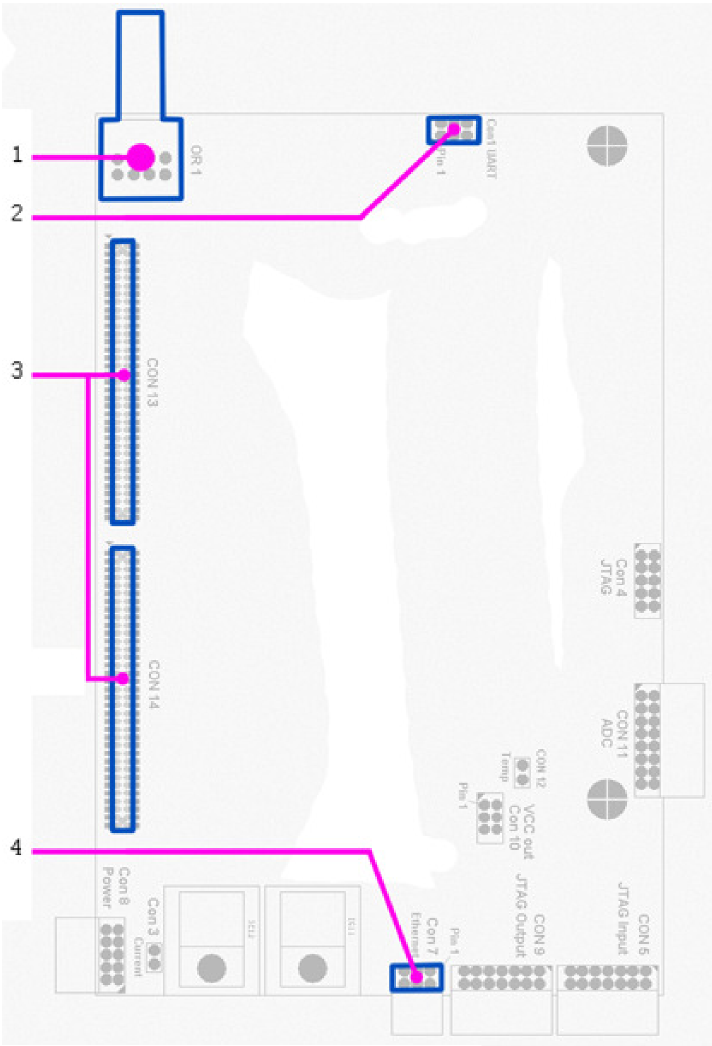


Figure 9-1: PCB layout of DCS board.

Table 9-1: Connectors on the DCS board.

#	Type	Description
1	Optical input	Optical input from LTU or CTP emulator
2	UART	RS-422 connection
3	Connector	DCS bus connector to BusyBox PCB
4	Ethernet	Ethernet link to communicate with DCS board

9.1 *Communication with the DCS board*

There are 54 general IO pins and 8 dedicated control pins used to connect {7} the DCS board to the BusyBox board. The pins are for the DCS bus, clock, reset, L1 accept, Serial B, voltage regulators and the SelectMAP interface.

9.2 *Setting up DCS board Firmware to use with BusyBox*

The DCS board firmware needs to be adapted for the BusyBox. If the DCS board is not already modified, it needs to be reprogrammed to fit the BusyBox.

A description on how to update the DCS board flash device to work with the BusyBox is given her:

http://web.ift.uib.no/~kjeks/wiki/index.php?title=Electronics_for_the_Time_Projection_Chamber_(TPC)#Update_of_the_DCS_Board_Flash_Device

10 The BusyBox Communication Protocol

This Chapter describes the communication protocol which is used by the BusyBox and the D-RORCs to send commands and receive event IDs.

10.1 *Introduction*

Besides decoding trigger information the BusyBox must also be able to communicate with the D-RORCs. The communication is necessary in the sense that the BusyBox needs to know when to set the *busy* signal to the TTC system.
The BusyBox communication protocol was developed by Magne Munkejord as part of his master thesis. His work included investigation of serial communication protocols, implementation and testing. A robust serial communication protocol with the D-RORCs was then achieved.
The protocol defines the mechanical, electrical and functional characteristics of a serial data bus. It feature a LVDS coupled network interface, NRZ encoding, RS-232 like message format and full duplex command/response protocol. The communication link has a data rate of 40 Mbps.

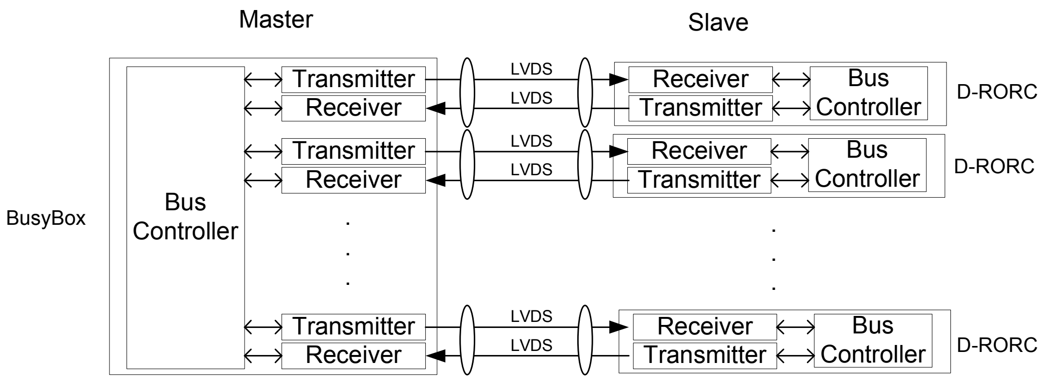


Figure 10-1: BusyBox - D-RORC bus structure.

10.2 *Physical Layer*

The communication between BusyBox and the D-RORCs are done with LVDS and the transmission lines are twisted pair cables with RJ-45 connectors. The TP cables are not longer than 15 m, thus providing good signal integrity.

7.2.1 LVDS

LVDS (Low-Voltage Differential Signalling) is an electrical signalling system that can run at very high speed over inexpensive twisted pair copper cables. LVDS is a differential signalling system, which means that it transmits two differential voltages which are compared at the receiver. LVDS uses this difference in voltage between the two wires to encode the information.
The Virtex-4 FPGA is configured with the LVDS I/O standard specified as LVDS_25 for the output and the input I/O block, DIFF_TERM, is enabled to set the internal differential resistor.

7.2.2 Twisted Pair and RJ-45

Twisted pair cabling is a form of wiring in which two conductors, the forward and return conductor of a single circuit, are twisted together for the purpose of cancelling out electromagnetic interference from external sources. The RJ-45 is a standard eight wire connector.
Standard straight Cat-5 twisted pair cables with RJ-45 connectors are used in the BusyBox – D-RORC communication lines and the connection scheme is shown in Figure 10-2. The wiring scheme is the same as used for 10/100 BASE-T Ethernet.

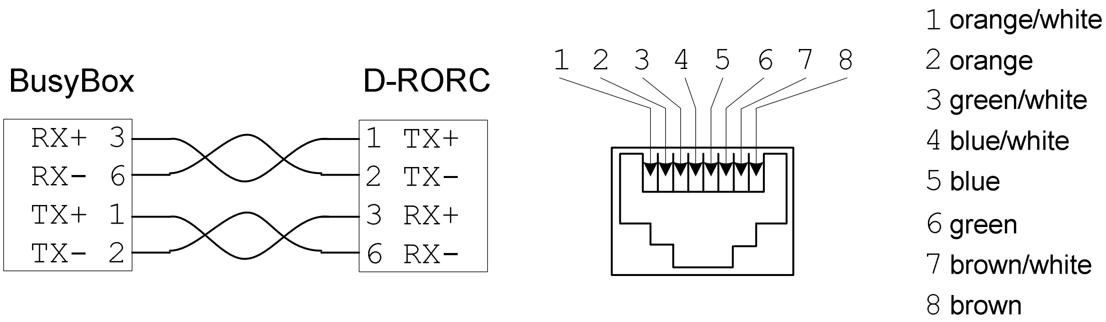


Figure 10-2: RJ-45 pin connection for BusyBox and D-RORC.

10.3 *Message Formats*

The communication on the bus consists of two types of messages: 1) Message sent from the BusyBox, and 2) Message sent from the D-RORCs. All words sent are 16 bit long with an RS232-like message format: 2 start bits, 16 data bits, 1 parity bit and 1 stop bit.

10.3.1.1 BusyBox message

The BusyBox message has two 4 bit words: The *Command type* word and the *Request ID* word. The remaining 8 LSB bit of the message are unused.

Table 10-1: Bit map for BusyBox message.

15 – 12	11 – 8	7 - 0
Command type	Request ID	Unused

Command type The command type word is used to command the D-RORCs to transmit event ID or to do error handling in relation to debugging.

Table 10-2: Command types.

Command type	Bit Code	Description
Request Event ID	0100	Request an Event ID from the D-RORC.
Resend last message	0101	Command the D-RORC to re-transmit the last message sent.
Force pop Event ID	0110	Command the D-RORC to pop one Event ID from its local queue.
Force Request ID	0111	Command the D-RORC to store the attached Request ID.

Request ID The request ID word is generated by the BusyBox to control the event ID queue in the D-RORCs.

10.3.1.2 D-RORC message

The D-RORC message is 48 bit long with 4 words: Request ID, Bunchcount ID, Orbit ID and D-RORC ID. The message is divided into three 16 data bits before it is sent.

Table 10-3: Bit map for D-RORC message.

47 – 44	43 – 32	31 – 8	7 – 0
Request ID	Bunchcount ID	Orbit ID	D-RORC ID

Request ID described in section 10.3.1.1.

Bunchcount ID The bunchcount ID is the number of bunch that is involved in the collision.

Orbit ID The orbit ID is the number of times all bunches have rotated since the start of the run.

D-RORC ID The D-RORC ID is the unique ID given to each D-RORC.

10.4 *Transmission*

Both the D-RORC and BusyBox run on the same nominal BC frequency, but do not share the same clock source. This is defined as a plesiochronous system and refers to the fact that this system runs in a state where different parts of the system are almost, but not quite perfectly in sync, i.e. a sender and a receiver operate at the same nominal frequency and might have slight frequency mismatch, which leads to a drifting phase.
The communication between BusyBox and D-RORC use NRZ line coding. A NRZ (non-return-to-zero) code is a binary code in which 1's are represented by one significant condition and 0's are represented by some other significant condition, with no other neutral or rest condition. NRZ is not inherently a self-synchronous code, and needs some kind of synchronisation technique to avoid bit slip.
The BusyBox has two clock domains, clock A and clock B. Clock A is 200 MHz and is derived from clock B, 40 MHz, which is the nominal BC frequency in LHC. Clock B is used for serial communication with the D-RORCs.
Messages sent from the D-RORCs are 48 bit long and commands sent from the BusyBox are 16 bit long. To avoid that the two communication devices get out of synch due to the system being plesiochronous, long bit streams are avoided by dividing the D-RORC messages into three 16 bit messages before they are sent to the BusyBox. In addition to this each bit is cycled 5 times with respect to clock A, giving a 40 Mbps rate. At the receiving end the bit stream is sampled into a shift register long enough to hold a complete messages. Then the message is run through majority gates to determine the logic values of the capture data.

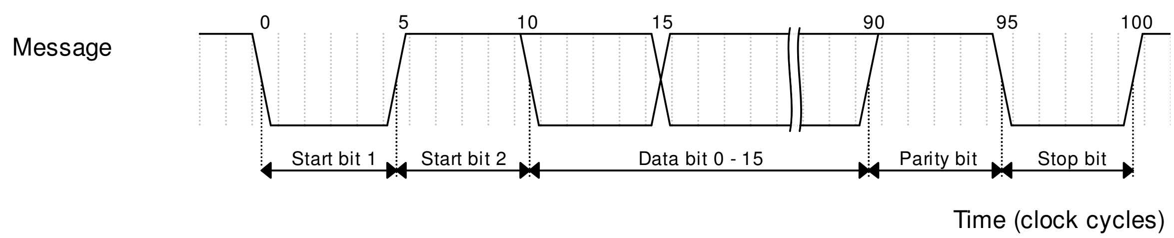


Figure 10-3: Message format.

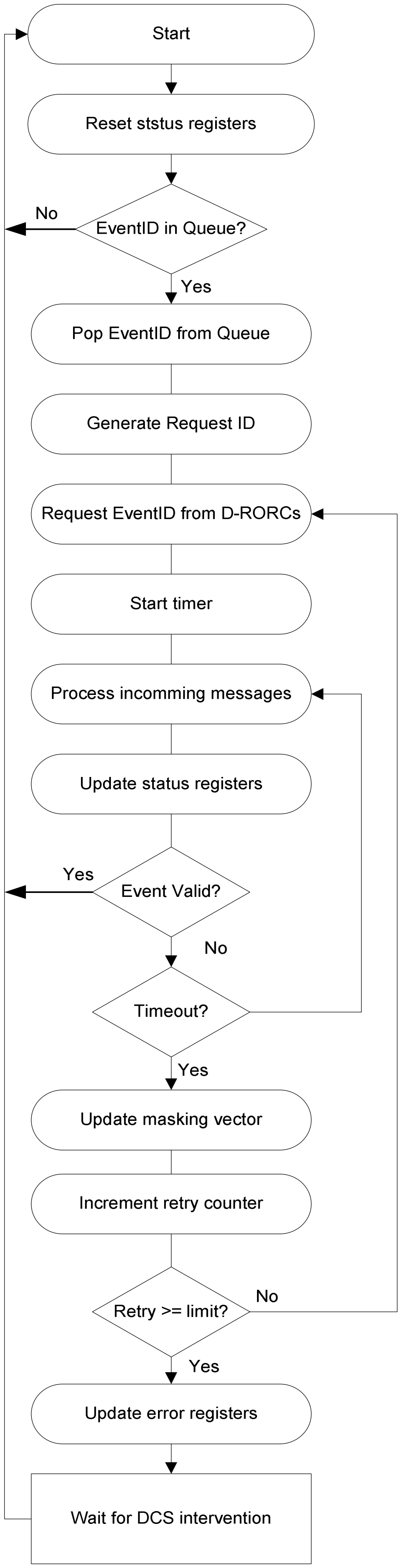


Figure 10-4: Sequence diagram for the BusyBox.

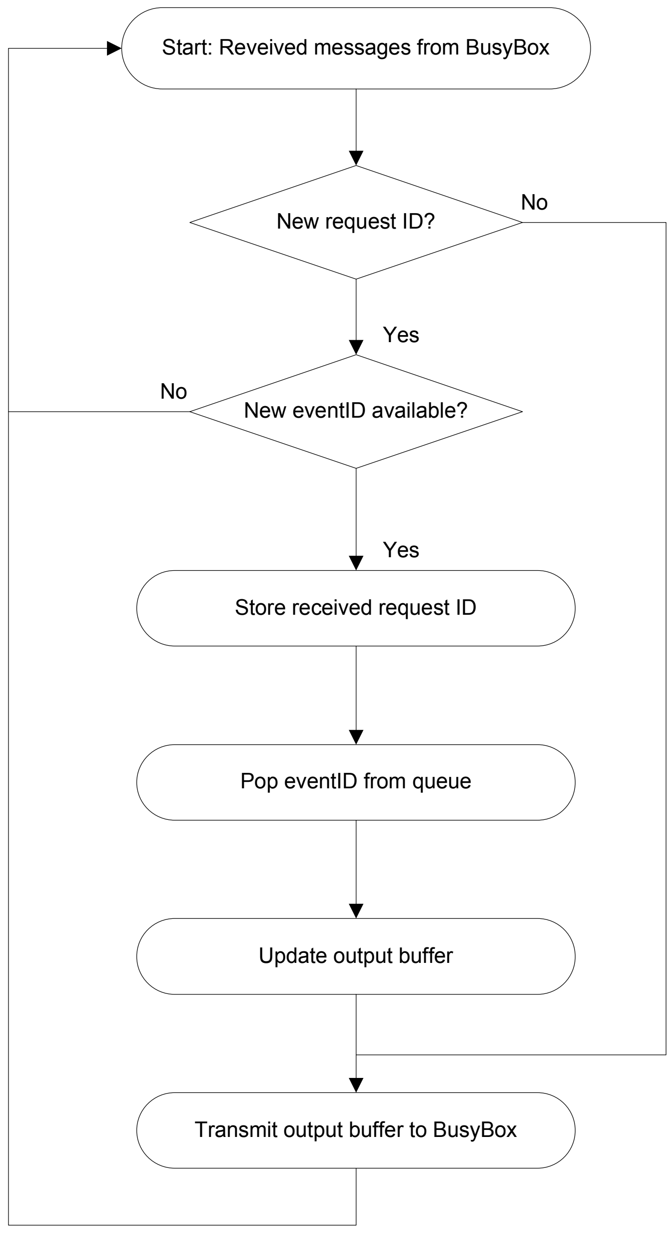


Figure 10-5: Sequence diagram for the D-RORC.

11 BusyBox User Guide

This chapter will give some insight in how to get started with the BusyBox, make changes to the firmware and do hardware/firmware tests/simulations.

11.1 *Xilinx ISE and QuestaSim files*

File	Folder	Description
project_setup.tcl	/trunk/ISE_projects/busybox_fpga1	TCL scrip to set up the project for FPGA 1 in ISE (TPC)
project_setup.tcl	/trunk/ISE_projects/busybox_fpga1_solo	TCL scrip to set up the project for FPGA 1 solo in ISE (PHOS)
project_setup.tcl	/trunk/ISE_projects/busybox_fpga2	TCL scrip to set up the project for FPGA 2 (TPC)
project_setup.tcl	/trunk/simulation	TCL scrip to set up the project for simulation in QuestaSim
trigger_receiver.mpf	/vhdlcvs	Project file for QuestaSim (located in the CVS repository, see chapter 3.1)

11.2 *Introduction*

TCL scripts sets up projects in Xilinx ISE Design Suit and Mentor Graphics' QuestaSim. All the files needed for the BusyBox are in the SVN Repository[8]. The scripts automatically make a project for each firmware versions and add all the files needed to simulate the design in QuestaSim.

Furthermore, knowledge about the interaction with the BusyBox hardware is given on how to program, read/write registers and test the design with triggers from a trigger emulator.

7.2.1 Project Setup

There are two main TCL scripts to setup a project. There is one script for QuestaSim to do simulations, and three scripts for ISE Design Suite to make changes to the three different firmwares[9] and compile it.

In QuestaSim the TCL scrip is run under *Tools* > *TCL* > *Execute Macro* In Xilinx ISE navigate to the directory where the TCL file is located and type *xtclsh project_setup.tcl* in the transcript window.

The Trigger Receiver module uses CVS Repository and the *trigger_receiver.mpf* file setup this QuestaSim project for simulation.

Note: Some editing may be done in the TCL files to setup the project correctly, i.e. the location of where the repository is located on your computer.

---

[8] http://svn.ift.uib.no/svn/busybox_firmware

[9] There are three different firmware version for the BusyBox: busybox_fpga1_solo, busybox_fpga1 and busybox_fpga2

11.3 *Hardware Setup*

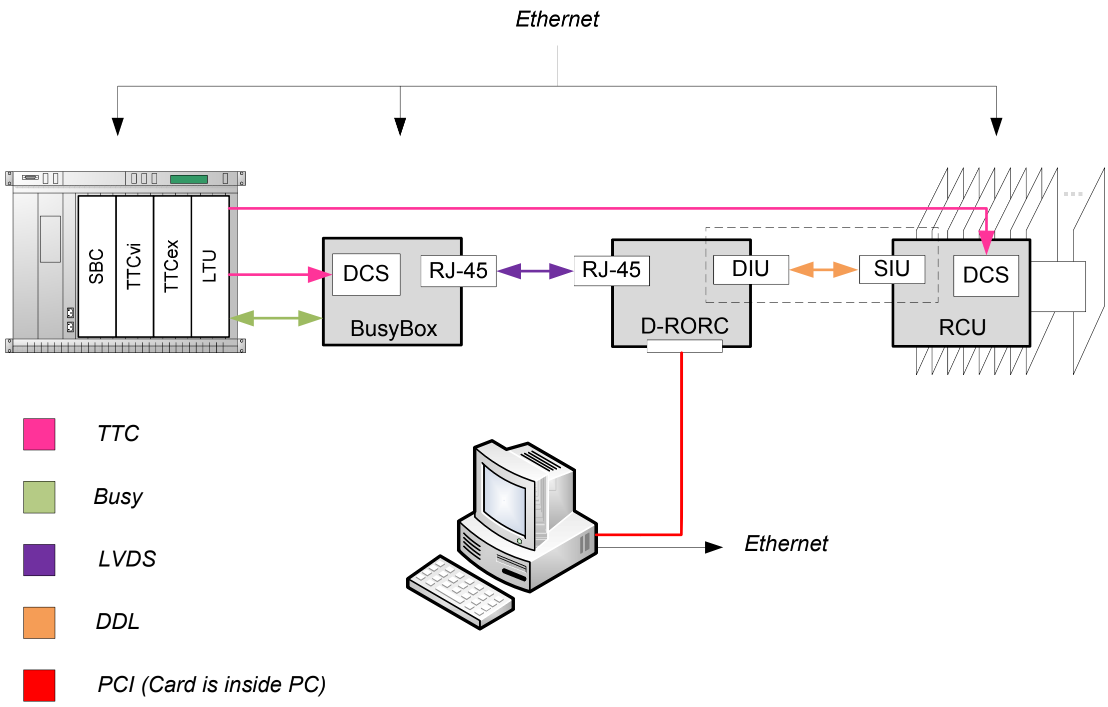


Figure 11-1: Hardware setup for experimenting with the BusyBox.

The hardware setup for testing the BusyBox is shown in Figure 11-2. All the devices has interface to Ethernet and are Linux based. Thus, they can all be controlled by one PC with Linux or Window operating system and Ethernet connection.

The CTP Emulator is connected to the BusyBox and Fee via optical cables to the DCS board. TP cables from the *busy* outputs are connected from the BusyBox to the CTP Emulator with LEMO plugs. TP cables are also used between the BusyBox an LDC (D-RORCs) and optical cables from the Fee to the LDC.

Note:

The Fee must utilize a whole section with Fec (A or B) to work.

The optical cables connected to the CTP Emulator must have an attenuator.

TP connection scheme for DCS Ethernet connector:
http://www.kip.uni-heidelberg.de/ti/DCS-Board/current/mechanic/DCS160Ethernet01.htm

11.4 *Logging on to the DCS board*

The DCS board mounted on the BusyBox is the easiest way to interact with the firmware. From here registers can be accessed and new firmware can be programmed to the Virtex-4 chip(s).

Interfacing with the DCS board is done either trough Ethernet or UART. The DCS board runs on a lightweight version of Linux and is access through SSH.

To login type:
*ssh root@dcs**xxxx***,

where ***xxxx*** is substituted with the number of the DCS card. You will then be prompted for a password.

11.5 *RCU Shell*

The rcu-sh, RCU shell, is a software that is "built around" the BusyBox firmware and provides an interface for users to interact with it. The shell is used to read and write registers in the BusyBox firmware.

Type rcu-sh after login on the DCS board. Type “h” and press “enter” to see available commands in the RCU shell.

To send commands to the D-RORCs type:

rcu-sh w 0x1 0x ‘--‘--‘,

where the first ‘--‘ (without the quotes) is the channel number in hex and the second ‘--‘ (without the quotes) is the command type in hex. E.g., to command the D-RORC connected to channel 0 to pop its event ID, type: rcu-sh 0x1 0x0006.

Note: If the channel number provided is greater than the actual number of channels the message is transmitted on all channels. The reply from the D-RORCs is stored in the RX Memory module.

11.6 *Programming the FPGA*

The FPGA(s) will not be programmed automatically when the box is powered up. To check if the FPGA(s) are programmed the right green LED {3} will be on. One can try to read some register with the RCU shell, e.g: shell prompt on DCS board: rcu-sh r 0x1000. If the result is “**no target answer**” then the FPGA is not programmed. Otherwise you should get the value of the register.

The easiest way to program the FPGAs is to use the shell script "**program**". This script should be located with the programming files for the FPGAS (*.bit) in the directory "**/mnt/dcscard/busybox-files/**"

Prompt on DCS board: ./program <programmingfile1.bit> [<programmingfile2.bit>]
There should be four programming files in the directory:
1. busybox_fpga1.bit for the first of two FPGAs
2. busybox_fpga2.bit for the second of two FPGAs
3. busybox_fpga1_solo.bit for FPGAs on boards/boxes where only on FPGA is mounted.
4. busybox_dummy.bit will be used by the script to program the second FPGA if no second programming file is given.

Note:

When two FPGAs are mounted then both must be programmed, or else the firmware will not work.

The bit files to be programmed into the FPGA(s) must be put in the folder: /nfs_export/dcscard, on kjekspc7.

11.7 *Configuring the Firmware*

Modify the shell script **bbinit.sh** to fit your setup.

11.8 *Monitoring the BusyBox registers*

Use **regpoll.sh status** to view most of the status registers of the BusyBox.
Type:
./regpoll.sh status
To display the channel registers use **regpoll.sh channels**.
Type:
./regpoll.sh channels

11.9 *Resetting the BusyBox*

To activate the global asynchronous reset of the Busy Box firmware, both FPGA(s), run "rcu-sh fw r". This will reset all registers in the Busy Box (except for the block RAMs). The configuration registers must be set again, including channel registers.

11.10 *CTP Emulator*

When testing or debugging with the BusyBox the CTP trigger emulator can be used.

Open a terminal window in Linux:

> Type: **ssh –X ltu@vme1**, and enter the password when prompted.
> Type: vmecrate ltu.

Then the VME menu is displayed:

> Click *Configuration* and *LTUinit*.
> Click *Configuration* and *TTCreset*.
> Click *CTP Emulator*.

The CTP Emulator window pops up.

Next to the *Sequence* tag click **L2a.seq** and click **Load sequence**.

Click *Start emulation.*

So, when you click the **Generate SW ‘Start signal(s)’** trigger sequences are sent to the BusyBox.