

BusyBox User Guide

The purpose of the 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 receive 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 BusyBoxes are located in the DAQ counting room and is a FPGA based system developed at the University of Bergen. The first of three development phases was done by Anders Rossebø. He designed the BusyBox hardware including the 19" rack case. Then Magne Munkejord developed the firmware and PhD student Johan Alme contributed with the Trigger Receiver Module to make the firmware complete. And finally a full test, integration and commissioning was done by Rikard Bølgren and Magne Munkejord.

This User Guide is about the whole BusyBox system. The intention is to give newcomers to the system an intuitive understanding without going too much in detail.

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 everything. How to program and read registers in the FPGA, to emulate triggers from an emulator.

For more latest firmware release and latest discussion about the BusyBox, check out the wiki at:

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

1	Table of content	
1	TABLE OF CONTENT	2
2	INTRODUCTION	3
2.1	Document History	3
2.2	Abstract	3
3	SYSTEM OVERVIEW	4
4	BUSYBOX HARDWARE	6
4.1	Xilinx Virtex IV FPGA	7
4.2	DCS	7
5	BUSYBOX FIRMWARE	8
5.1	Introduction	8
5.2	BusyBox top-level wrappers	10
5.3	Module digital_clock_manager	12
5.4	Busylogic_top Module	14
5.5	Reset_logic module	14
5.6	DCS Bus Arbiter and Address Decoder	15
5.7	Receiver Module	16
5.8	Transmitter Module	20
5.9	RX Memory Module	24
5.10	RX Memory Filter Module	26
5.11	Trigger Receiver Module	26
5.12	Event ID Verification Module	29
5.13	Busy Controller Module	34
5.14	Control and Status Registers	35
6	FUNCTIONAL VERIFICATION OF THE BUSYBOX FIRMWARE	37
6.1	Introduction	37
6.2	Support packages	38
6.3	The RCU and DRORC emulator module	40
6.4	Testbench execution flow	41
6.5	Running the simulation in QuestaSim/ModelSim	43
7	BUSYBOX DCS BOARD	44
7.1	Setting up DCS board Firmware to use with BusyBox	45
8	BUSYBOX COMMUNICATION	46
8.1	BusyBox - DRORC Communication	46
8.2	TTCrx Communication	49
8.3	DCS Communication	49
8.4	LTU Communication	49
9	GETTING STARTED WITH THE BUSYBOX	50
9.1	Introduction	50
9.2	SVN Repository and Project Setup	50
9.3	Hardware Setup	51
9.4	Logging on to the DCS board	51
9.5	RCU Shell	52
9.6	Programming the FPGA	52
9.7	Configuring the Firmware	52
9.8	Monitoring the BusyBox registers	52
9.9	Resetting the BusyBox	53
9.10	CTP Emulator	53
10	REGISTER	55
10.1	BusyBox Register Interface	55
10.2	Trigger Receiver Module Register Interface	56
10.3	TPC Channel Register Interface	58

2 Introduction

The purpose of this document is to give users of the BusyBox an understanding of what it is, by discussing the hardware, firmware and software which is the key components of the ALICE BusyBox system.

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

2.1 Document History

Revision number	Revision date	Summary of changes	Author
1.0	09.12.08	N/A	Rikard Bølgen
1.1	08.12.09	<ul style="list-style-type: none"> Merged into one file. Added chapter Feil! Fant ikke referanseilden. Feil! Fant ikke referanseilden.. Changed document style. Updated some interface tables and figures to latest firmware revision. 	Magne Munkejord
1.2	14.09 2010	<ul style="list-style-type: none"> Updated first page Edited layout 	Kjetil Ullaland

Table 2-1: Revision history.

Package	Version
Firmware BusyBox	41
Trigger Receiver Module	1.5
DCS card	Version 2.84 BUSYBOX

Table 2-2: Firmware versions corresponding to this guide.

2.2 Abstract

ALICE is one of four large detectors situated at the collision points in the LHC at CERN. The BusyBox utilizes the TPC, PHOS, FMD and EMCAL sub-detectors and it is an FPGA based device.

It will verify the transfer of event data from the sub-detectors Front End Electronics (FEE) to the Data Acquisition system (DAQ). The BusyBox also keeps track of free buffers in the FEE. If the buffers are full or a collision is detected the BusyBox will flag a busy signal to a Central Trigger Processor (CTP), which halts further triggers from being issued.

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

3 System Overview

The BusyBox is a part of the data acquisition in four of the ALICE sub-detectors, namely: TPC, PHOS, FMD and EMCal. The latter is currently in development.

There are some minor differences between the BusyBoxes for each sub-detector because of the different numbers of D-RORCs they use.

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

Table 3-1: Number of D-RORCs per detector

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 senses 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 3-1 below illustrates the BusyBox place in the readout chain.

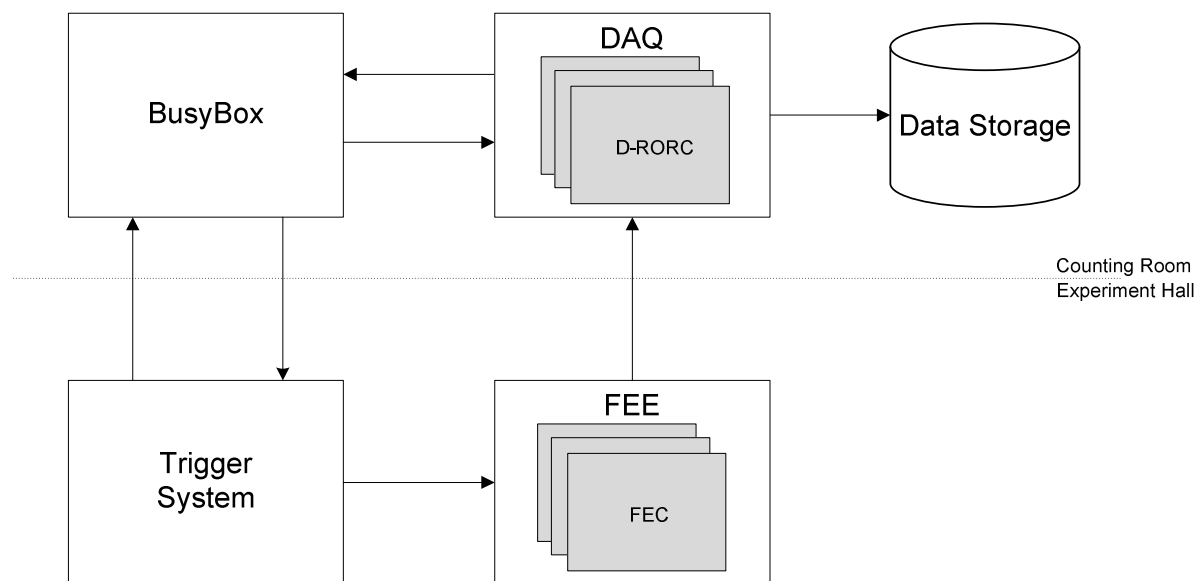


Figure 3-2: Illustration of the data flow for the BusyBox system. The BusyBox and D-RORCs are placed in the counting rooms above the experiment hall.

4 BusyBox Hardware

This chapter discusses the hardware and some key components.

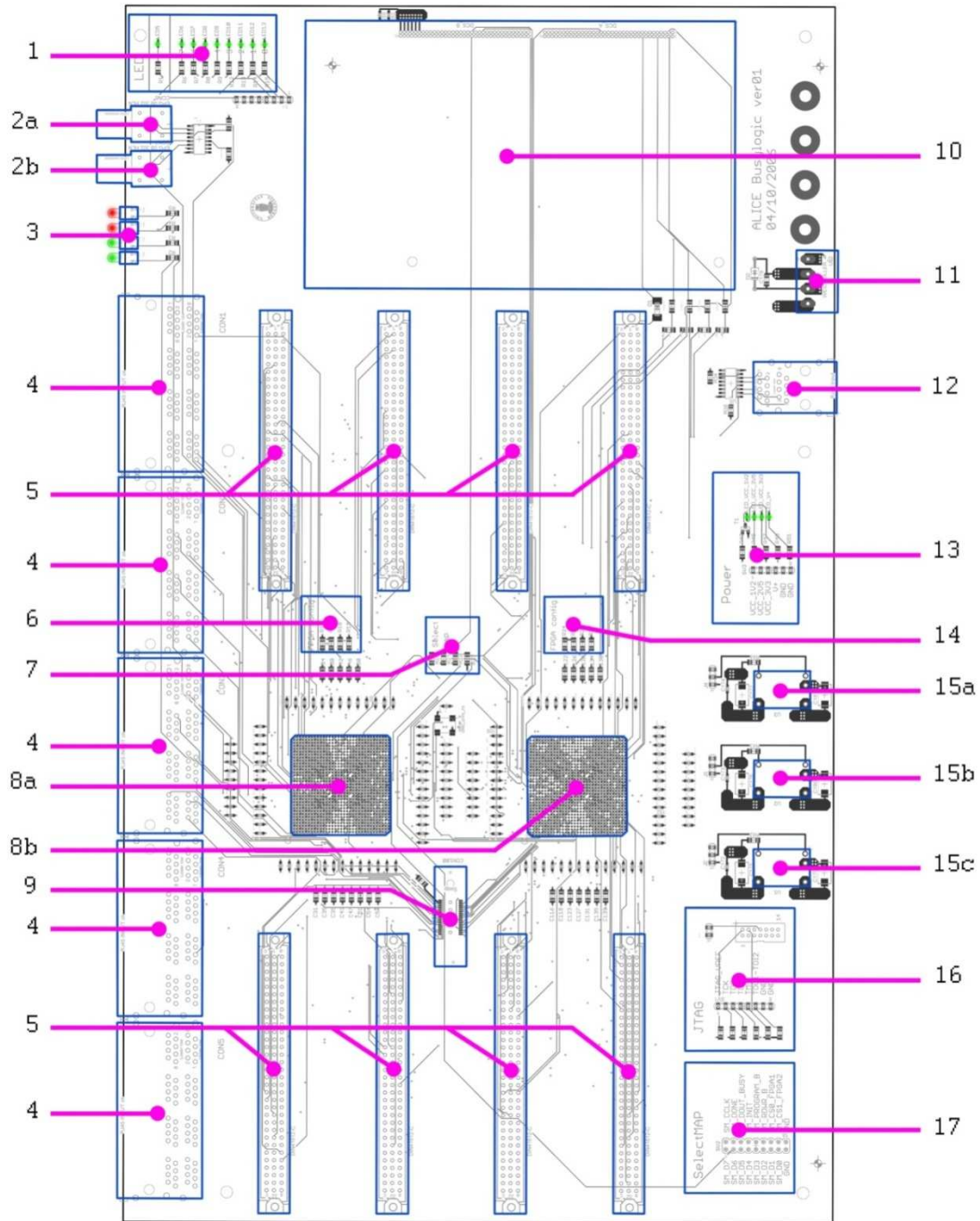


Figure 4-1 BusyBox PCB

#	Type	Description
1	LED indicators	Indicates numbers of buffers used
2a	LEMO contact	LVDS busy signal from FPGA 1
2b	LEME contact	LVDS busy signal from FPGA 1
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	Resistors	FPGA1 configuration resistors. Sets the configuration mode to 8bit SelectMAP slave.
7		
8a	FPGA 1	Xilinx Virtex IV
8b	FPGA 2	Xilinx Virtex IV (TPC only)
9		Connector for digital analyzer. Also serves as inter connect between FPGAs
10	DCS board	Connectors for DCS board
11	Power supply connector	Connector for external power supply. Power supply: 5V, 12 A
12		SelectMAP resistor for pull-ups/pulldowns and thevenin termination of clock (CCLK).
13	Pin connectors	GND, 1.2V, 2.5V and 3.3V output
14		FPGA2 configuration resistors. Sets the configuration mode to 8bit SelectMAP slave.
15abc	Voltage regulator	PTH05000W voltage regulators from Texas Instruments
16	Test point JTAG interface	Standard JTAG access port. Used for configuration and debug.
17	Test point SelectMap interface	BUG: SelectMAP CCLK must be cut at certain point on PCB to prevent ringing.

Table 4-1: List of components on the PCB.

4.1 Xilinx Virtex IV FPGA

The BusyBox use the Virtex-4 LX-40 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 program 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.

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

4.2 DCS

The DCS board was originally designed for the TRD and TPC sub-detectors, but because it was very versatile it has been adapted for the BusyBox and other instrumenst in ALICE experiment. It is running a lightweight version of Linux and implements TCP/IP network protocol. 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 the control room at Point 2.

5 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.

5.1 Introduction

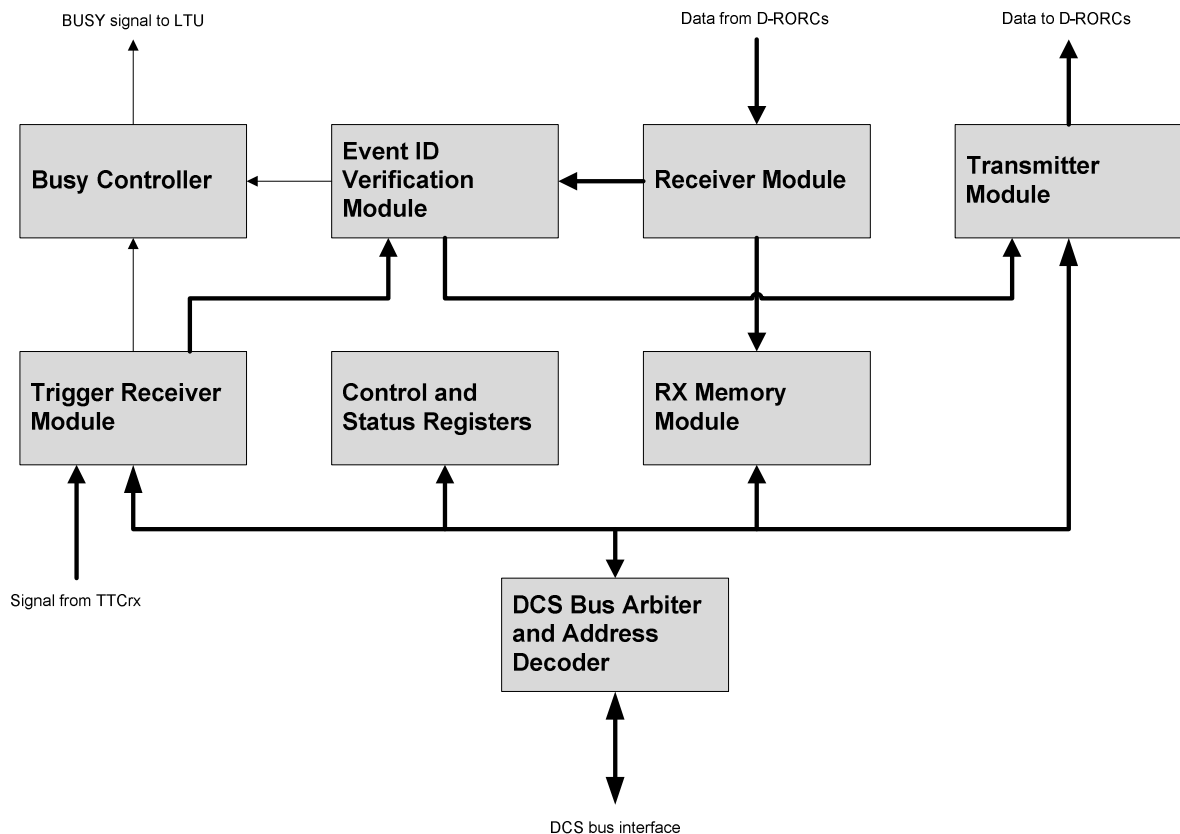


Figure 5-1: Main BusyBox firmware modules.

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 L1 trigger has been issued by the CTP.

5.1.1 An intuitive explanation of how the BusyBox firmware works

It all starts with a collision of hadrons in the LHCs ALICE detector. The CTP detects this collision and notifies the LTU, which issues a L0 trigger to all four BusyBoxes via its optical fiber network. The L0 trigger is the start of a sequence of triggers and ends with either an L2a or L2r trigger.

The LTU broadcasts the BC, Channel A and Channel B to the BusyBox through its fiber network and is converted by the TTCrx chip, on the DCS card, to electrical signals. Then the information is decoded by the trigger receiver module.

Not all of the decoded messages are useful for the BusyBox. Hence, the trigger receiver module only extracts the bunch count, event ID and triggers.

The triggers are forwarded to the busy controller module, which asserts the busy when a L1 trigger is received.

The bunch count and event ID is used to assure that all D-RORCs have received data from an event and with that information in hand the BusyBox can keep track of the FEE buffers. If all D-RORCs have received event data, this will imply that the event data have 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). So if there is a L0 trigger 1 buffer is occupied, and if all the D-RORCs have responded with the same event ID and bunch count the EventID is OK (EIDOK). Then the event data have been read out and the buffer is free.

A control and status register can as the name implies, control and check the status of registers in the BusyBox. Registers in the trigger receiver module and stored data from the receiver module in the RX memory module can also be accessed. All this is done via the FeeServer on the DCS card mounted on the BusyBox PCB.

5.1.2 VHDL Entity Hierarchy

- *busybox_fpga1_solo* || *busybox_fpga1* || *busybox_fpga2*
 - *busylogic_top*
 - *ctrl_regs* (Control and Status Registers)
 - *dcs_arbit_addr_dec* (DCS Bus Arbiter and Address Decoder)
 - *transmitter_module*(Transmitter Module)
 - *serial_encoder*
 - *multi_channel_receiver* (Multi Channel Receiver Module)
 - *single_channel_receiver*
 - *serial_rx*
 - *branch_controller*
 - *backbone_controller*
 - *rx_mem_filter*
 - *receiver_memory_module* (RX Memory Module)
 - *rx_bram*
 - *event_validator_top* (Event ID Verification Module)
 - *drorc_inbox_buffer*
 - *FIFOsync108x512*
 - *daq_header_extractor*
 - *eventid_control*
 - *eventid_processor*
 - *trigger_receiver_busylogic* (Trigger Receiver Module)
 - *busy_controller* (Busy Controller)

The firmware can be implemented by using one of three different top-level wrappers. The different top-level wrappers are necessary to adapt the firmware to fit different hardware configurations. If two FPGAs are present then they must work in parallel and be coordinated. If only one FPGA is present it must operate in standalone mode.

Figure 5-2 shows a graphical representation of the same information as above. In addition the IO buffers that are instantiated in the top-level wrappers are included in the figure.

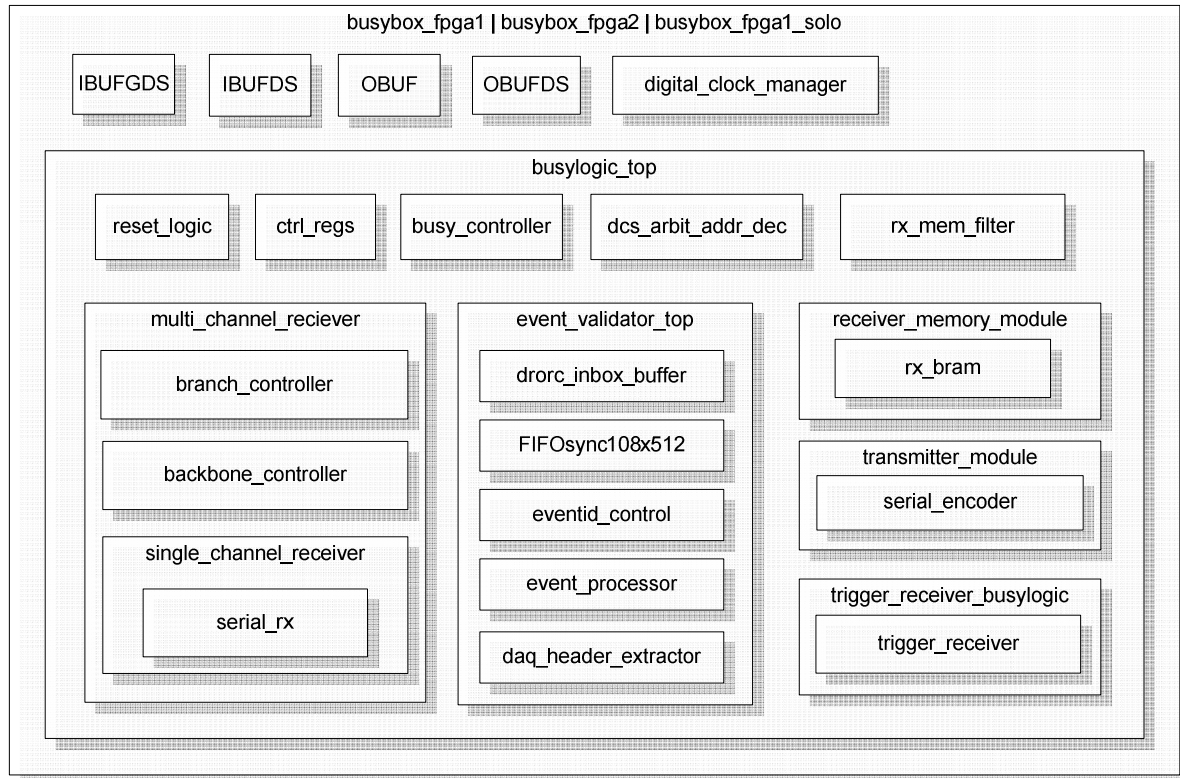


Figure 5-2: Module instantiation hierarchy.

5.2 BusyBox top-level wrappers

The BusyBox PCB can be fitted with one or two FPGAs depending on the number of channels required. The different hardware configurations require slightly modified versions of the firmware at the toplevel HDL source. A BUSY signal from FPGA2 is combined in FPGA1 among other things. There are three different toplevel HDL files: `busybox_fpga1.vhd`, `busybox_fpga2` and `busybox_fpga1_solo`. There three are different usage scenarios:

- For BusyBoxes with 2 FPGAs mounted, FPGA1 should be programmed with firmware based on `busybox_fpga1` and FPGA with `busybox_fpga2`.
- If the second FPGA is not going to be used then FPGA1 can be programmed with `busybox_fpga1_solo`, but FPGA2 must then be programmed with a dummy configuration for the programming operation to finish without errors.
- For BusyBoxes with only one FPGA firmware based on `busybox_fpga1_solo` must be used.

At the toplevel HDL wrappers it is possible to specify the number of channels to be implemented. Each channel is connected to the rest of the design through a branch. One branch controller can support up to 16 channels. There must be enough branches to connect the number of channels specified or the implementation will fail.

5.2.1 Entity BusyBox FPGA Modules

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 instantiate the *BusyBox_top* module with the required generic parameters and extra logic. The wrapper also adds and configures the necessary Virtex-4 IO buffers and Digital Clock Manager (DCM) around the *busylogic_top* module.

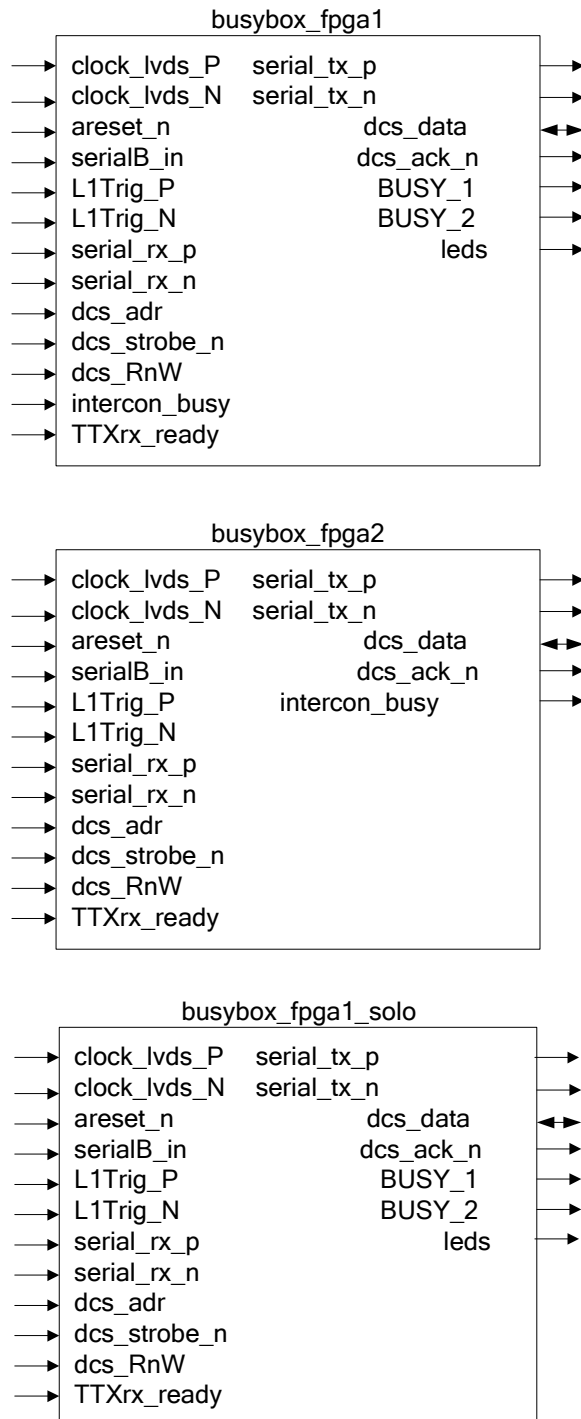


Figure 5-3: Entity for BusyBox FPGA modules.

- *BUSY1* and *BUSY2* outputs only exist on *FPGA1*. These outputs are the same logical signal.

- *Busybox_fpga1* wrapper takes busy input (*intercon_busy*) from FPGA2 and combines it with its own *BUSY* output through an OR gate.

Generic name	Type	Legal range	Default value	Description
<i>num_of_channels</i>	<i>natural</i>	0 to 119	119	Specifies the number of channels (-1) that will be instantiated at compile-time.
<i>num_of_branches</i>	<i>natural</i>	1 to 8	8	Specifies number of branches at compile-time. Each branch can connect 16 serial receiver channels.

Table 5-1: Generics at the HDL top-level wrappers.

BusyBox wrapper	<i>num_of_channels</i>	<i>num_of_branches</i>
<i>busybox_fpga1</i>	119	8
<i>busybox_fpga2</i>	96	6
<i>busybox_fpga1_solo</i>	39	3

Table 5-2: Default values for generic parameters for BusyBox wrappers.

Port Name	Direction	# Bit	Description
<i>clock_lvds_P</i>	Input	1	<i>std_logic</i> ;
<i>clock_lvds_N</i>	Input	1	<i>std_logic</i> ;
<i>areset_n</i>	Input	1	<i>std_logic</i> ;
<i>serialB_in</i>	Input	1	<i>std_logic</i> ;
<i>L1Trig_P</i>	Input	1	<i>std_logic</i> ;
<i>L1Trig_N</i>	Input	1	<i>std_logic</i> ;
<i>serial_rx_p</i>	Input	120 ¹	<i>std_logic_vector</i> (0 to <i>num_of_channels</i>);
<i>serial_rx_n</i>	Input	120	<i>std_logic_vector</i> (0 to <i>num_of_channels</i>);
<i>dcs_adr</i>	Input	16	<i>std_logic_vector</i> (15 downto 0);
<i>dcs_strobe_n</i>	Input	1	<i>std_logic</i>
<i>dcs_RnW</i>	Input	1	<i>std_logic</i> ;
<i>intercom_busy</i>	Input/Output ²	1	<i>std_logic</i> ;
<i>serial_tx_p</i>	Output	1	<i>std_logic_vector</i> (0 to <i>num_of_channels</i>)
<i>serial_tx_n</i>	Output	1	<i>std_logic_vector</i> (0 to <i>num_of_channels</i>)
<i>dcs_data</i>	Bidirectional	16	<i>std_logic_vector</i> (15 downto 0)
<i>dcs_ack_n</i>	Output	1	<i>std_logic</i> ;
<i>BUSY_1</i>	Output ³	1	<i>std_logic</i> ;
<i>BUSY_2</i>	Output ²	1	<i>std_logic</i> ;
<i>leds</i>	Output ²	13	<i>std_logic_vector</i> (1 to 13);

Table 5-3: I/O details for BusyBox FPGA Modules.

5.3 Module *digital_clock_manager*

This core has been generated by the Xilinx tool Architect Wizard available through the CoreGen GUI. It a single DCM configured to deskew and output two clock signals generated from the incoming clock. The incoming clock from the DCS board is approximately 40 MHz. The DCM multiplies this clock signal by 5 to generate a 200 MHz output used in the design. The 40MHz and 200 MHz are routed to global clock buffers that drive global clock nets to distribute the clock signals around the chips.

The DCM is setup in a system synchronous configuration. This means that the clock is fed back to the DCM after it has gone through the clock distribution network. The DCM will regulate the phase of its output clocks so that

¹ Number of channels implemented is configurable.

² This is an input on FPGA1 and an output on FPGA1. It does not exist in the *busybox_fpga1_solo* wrapper.

³ Only exists on FPGA1.

the feedback clock's rising edge and incoming's clock rising edge are aligned. This is done to compensate for the delay of the clock distribution network in the chip. This configuration ensures that the chip/FPGA is clocked synchronously with the rest of the system (other synchronous elements).

During startup of the FPGA (right after the configuration data has been loaded) the internal circuits of the DCM will try to lock on the incoming clock signal. This is an operation that might take several thousand clock cycles to complete and may fail if the incoming clock contains glitches and/or sporadic behavior. The clocks supplied by the DCM are not reliable until the DCM has acquired lock. A dedicated output signal named "lock" from the DCM indicates when lock has been acquired. This signal is forwarded to the reset logic which will hold the design in reset until the lock signal is deasserted.

The wizard that generates the DCM core does not support enabling of the `DIFF_TERM` attribute of the differential global clock input buffer (`IBUFGDS`). Therefore the clock input buffer is disabled in the wizard and instead the input buffer is instantiated in the `BusyBox` top-level 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 behavior.

NOTE: The differential termination could/can also be enabled by constraints in the User Constraints File (UCF).

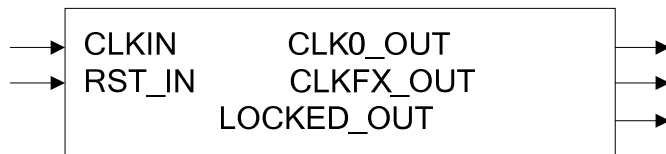


Figure 5-4: Entity for `digital_clock_manager`

5.4 Busylogic_top Module

The `busylogic_top` module is the common top level for all three firmware versions. It has a structural architecture where all main modules are instantiated and connected.

5.4.1 Entity for busylogic_top Module

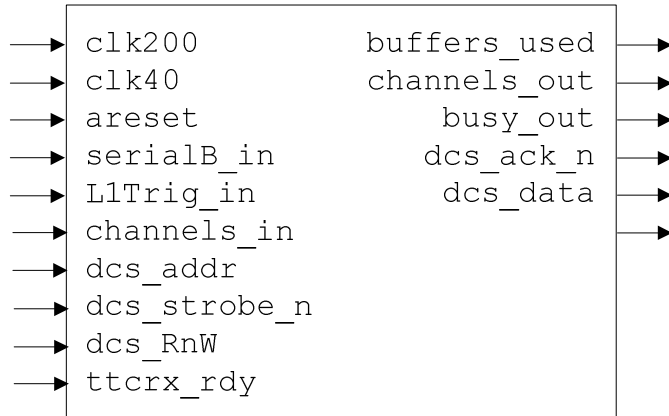


Figure 5-5: Entity for BusyBox top module.

Port Name	Direction	# Bit	Description
<code>clk200</code>	Input	1	<code>std_logic</code> ;
<code>clk40</code>	Input	1	<code>std_logic</code> ;
<code>areset</code>	Input	1	<code>std_logic</code> ;
<code>serialB_in</code>	Input	1	<code>std_logic</code> ;
<code>L1Trig_in</code>	Input	1	<code>std_logic</code> ;
<code>channels_in</code>	Input	120	<code>std_logic_vector(0 to num_of_channels)</code> ;
<code>dcs_addr</code>	Input	16	<code>std_logic_vector(15 downto 0)</code> ;
<code>dcs_strobe_n</code>	Input	1	<code>std_logic</code> ;
<code>dcs_RnW</code>	Input	1	<code>std_logic</code> ;
<code>ttcrx_rdy</code>	Input	1	<code>std_logic</code> ;
<code>buffers_used</code>	Output	4	<code>std_logic_vector(3 downto 0)</code> ;
<code>channels_out</code>	Output	120	<code>std_logic_vector(0 to num_of_channels)</code> ;
<code>busy_out</code>	Output	1	<code>std_logic</code> ;
<code>dcs_ack_n</code>	Output	1	<code>std_logic</code> ;
<code>dcs_data</code>	In/Out	16	<code>std_logic_vector(15 downto 0)</code> ;

Table 5-4: I/O details for BusyBox Top Module.

5.5 Reset_logic module

This module implements some simple reset logic to generate proper reset signals to the design. The reset signals will be asserted asynchronously whenever the DCM's locked signal is low. In other words, the design will not activate before the DCM locks on the incoming clock. After the DCM achieves lock it will assert the lock signal. The reset logic implements a shift register that is used to delay a synchronous release of the reset signals.

Reset signals for both clock domains are generated. This is done to let the synthesis tools employ "register duplication" to reduce fanout and routing delay of the resets in both clock domains.

Generic Name	Type	Comment
<code>g_rst_length</code>	<code>natural</code>	Sets the time in number of clock (<code>clk40</code>) cycles before the resets are released.

Table 5-5: Tabel of generic parameters for `reset_logic` module.

Port Name	Direction	Type	Comment
<code>clk200</code>	<code>in</code>	<code>std_logic</code>	200 MHz clock input
<code>clk40</code>	<code>in</code>	<code>std_logic</code>	40 MHz clock input
<code>clk_lock</code>	<code>in</code>	<code>std_logic</code>	acts active low asynchronous reset
<code>rst200</code>	<code>out</code>	<code>std_logic</code>	synchronous reset for the 200 MHz clock domain
<code>rst40</code>	<code>out</code>	<code>std_logic</code>	synchronous reset for the 40 MHz clock domain

Table 5-6: IO table for `reset_logic` module.

5.6 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.

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

Table 5-7: Bit-mapping of DCS bus address.

5.6.1 Entity DCS bus arbiter and address decoder

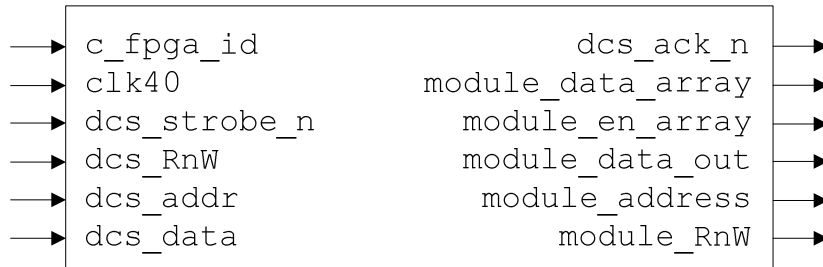


Figure 5-6: Entity for DCS Bus Arbiter and Address Decoder.

Generic name	Type	Comment
<code>c_fpga_id</code>	<code>std_logic</code>	This bit sets the slave FPGA MSB address. '0' for FPGA1 or '1' for FPGA2.

Table 5-8: Generic parameters for `dc_s_arbit_addr_dec`

Port Name	Direction	# Bit	Description
<code>clk40</code>	<code>Input</code>	1	<code>std_logic</code> ; the <code>clk40</code> frequency is 40.08 MHz
<code>dc_s_strobe_n</code>	<code>Input</code>	1	<code>std_logic</code> ; the asynchronous handshake is done with <code>STROBE_N</code> from the DCS board.
<code>dc_s_RnW</code>	<code>Input</code>	1	<code>std_logic</code> ; '1' read and '0' write.
<code>dc_s_addr</code>	<code>Input</code>	16	<code>std_logic_vector(15 downto 0)</code> ; address module and submodule register.
<code>dc_s_data</code>	<code>Inout</code>	16	<code>std_logic_vector(15 downto 0)</code> ; bi-directional data line.
<code>dc_s_ack_n</code>	<code>Output</code>	1	<code>std_logic</code> ; the asynchronous handshake is done with <code>ACK_N</code> from the busy board.

<i>module_data_array</i>	Output	7	<i>std_logic_vector</i> (0 to <i>num_of_modules</i> -1); communication with modules.
<i>module_en_array</i>	Output	7	<i>std_logic_vector</i> (0 to <i>num_of_modules</i> -1); communication with modules.
<i>module_address</i>	Output	12	<i>std_logic_vector</i> (11 downto 0); communication with modules.
<i>module_RnW</i>	Output	1	<i>std_logic</i> ; communication with modules.

Table 5-9:I/O details for DCS Bus Arbiter and Address Decoder.

5.7 Receiver Module

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

Table 5-10: Numbers of channels per detector pr FPGA.

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 data word, one word after another, when the serial decoder flags that data is ready to be sent. 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, and no parity errors and timeouts were found, the state machine will send the data to a multiplexer tree.

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

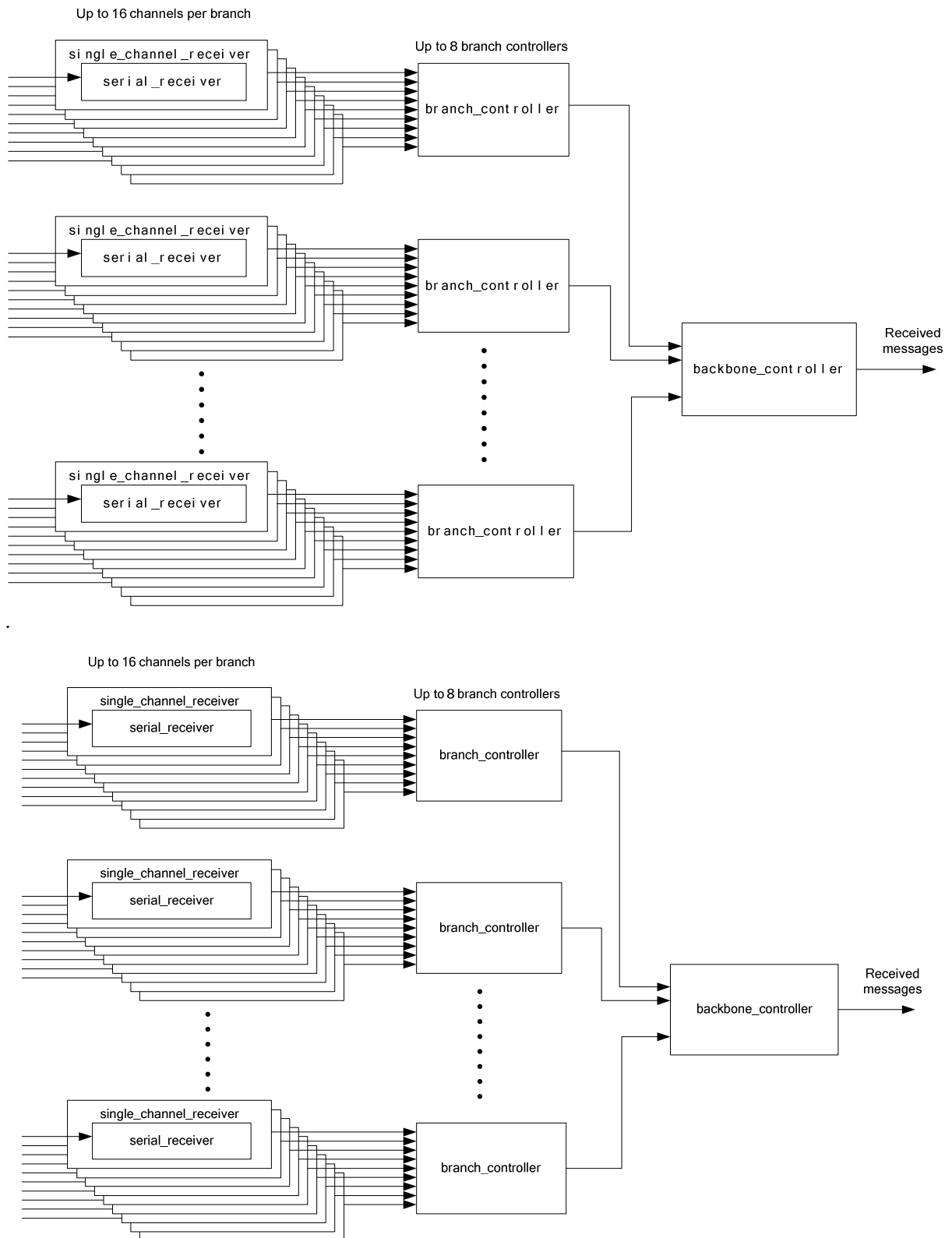


Figure 5-7: Architecture of multi channel receiver.

5.7.1 Receiver Module VHDL Entity Hierarchy

1) Multi Channel Receiver

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

5.7.2 Entity Multi Channel Receiver Module

The Multi Channel Receiver module has structural architecture that instantiates and connects the correct numbers of single channel receivers and branch controllers. The single channel serial receivers are connected to branch controllers and the branch controllers are then connected to the backbone controller. See Figure 5-8. The number of channels and number of branches to instantiate is defined through generics.

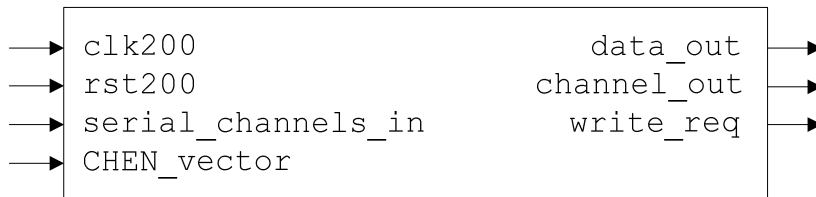


Figure 5-8: Entity for Channel Receiver Module.

Port Name	Direction	# Bit	Description
clk200	Input	1	std_logic; the clk200 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

Table 5-11: I/O details for Channel receiver Module.

5.7.3 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. Three 16 bit words are concatenated to a 48 bit message and stored temporary in three different registers. If the registers are not read out fast enough they will be overwritten.

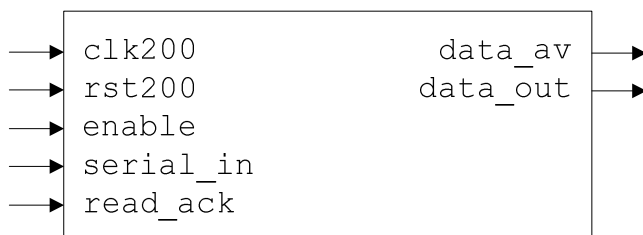


Figure 5-9: Entity for Single Channel Receiver.

Port Name	Direction	# Bit	Description
<i>clk200</i>	<i>Input</i>	<i>1</i>	<i>std_logic; the clk200 frequency is 200 MHz.</i>
<i>rst200</i>	<i>Input</i>	<i>1</i>	<i>std_logic; synchronous rest.</i>
<i>enable</i>	<i>Input</i>	<i>1</i>	<i>std_logic;</i>
<i>serial_in</i>	<i>Input</i>	<i>1</i>	<i>std_logic; data bit from serial decoder.</i>
<i>data_out</i>	<i>Output</i>	<i>48</i>	<i>std_logic_vector(47 downto 0); 48 bit data from D-RORC</i>
<i>read_ack</i>	<i>Input</i>	<i>1</i>	<i>std_logic;</i>
<i>data_av</i>	<i>Output</i>	<i>1</i>	<i>std_logic;</i>

Table 5-12: I/O details for Single Channel Receiver.

5.7.4 Entity Serial Decoder

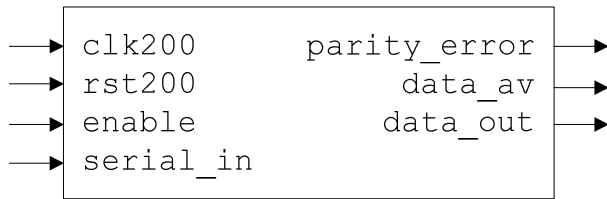


Figure 5-10: Entity for Serial Decoder.

Port Name	Direction	# Bit	Description
clk200	Input	1	std_logic; the clk200 frequency is 200 MHz.
rst200	Input	1	std_logic; synchronous reset
enable	Input	1	std_logic;
serial_in	Input	1	std_logic; 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

Table 5-13: I/O details for Serial Decoder.

If the Serial Decoder is enabled it listens to the transmission line for serial data. Each data word is packed into a frame and encoded on the serial signal as illustrated in

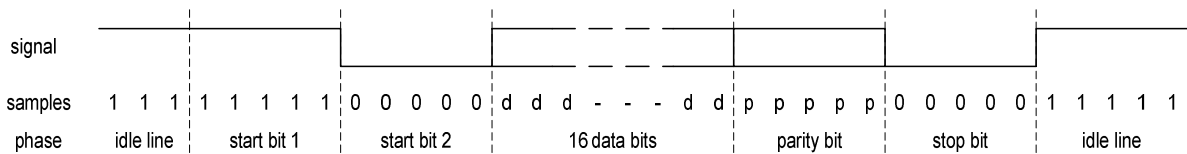


Figure 5-11. When the line is idle it pulled to logic 1. A frame starts with two start bits to create 1-0 transition. The decoder looks for this transition to lock on to the data frame. Each bit is sampled 5 times. This is necessary to detect the bit phase of the incoming serial bit stream. Once the 1-0 transition of the start bits are found a state machine in the decoder triggers and start capturing data. The state machine picks the sample that is believed to be the middle of each bit by counting samples at the local clock rate. After 16 data bits have been sampled, a parity bit and a stop bit are sampled. Both must have the correct logic value before the frame is accepted and data forwarded. The parity bit is a even parity generated by XOR'ing the data bits as they are received. The stop bit is always logic 0.

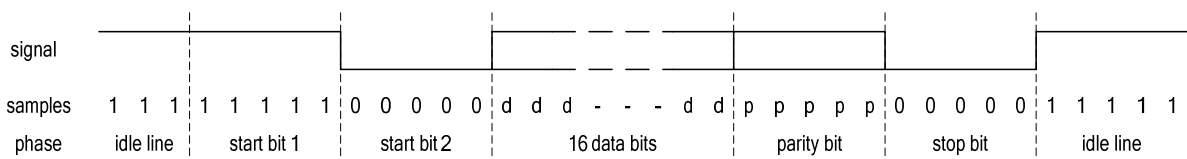


Figure 5-11: Encoding of serial data on transmission line.

5.7.5 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 until the Backbone Controller has verified that it has read the message.

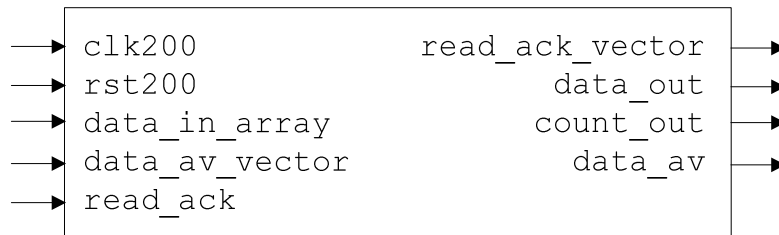


Figure 5-12: Entity for Branch Controller.

Port Name	Direction	# Bit	Description
clk200	Input	1	std_logic; the clk200 frequency is 200 MHz.
Rst200	Input	1	std_logic; synchronous 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

Table 5-14: I/O details for Branch Controller.

5.7.6 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.

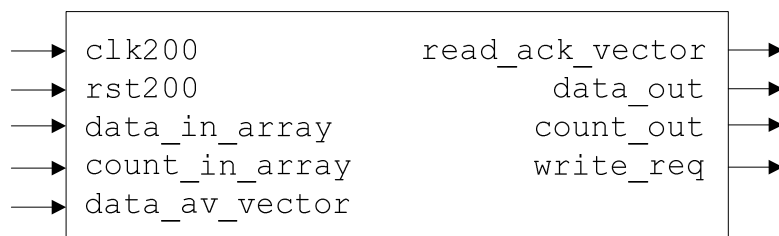


Figure 5-13: Entity for Backbone Controller.

Port Name	Direction	# Bit	Description
clk200	Input	1	std_logic; the clk200 frequency is 200 MHz.
rst200	Input	1	std_logic; synchronous 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
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;

Table 5-15: I/O details for Backbone Controller.

5.8 Transmitter Module

The transmitter module transmits serial data to the D-RORCs and consists of a controller, a serial decoder and a masking vector. A message register and a channel register are available for the DCS bus module and 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 channel number in the channel register. The masking vector lets the Event ID Verification module and DCS bus module select which channels to enable or disable. The controller handles requests from the Event ID Verification module and DCS bus module to prevent communication conflicts.

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

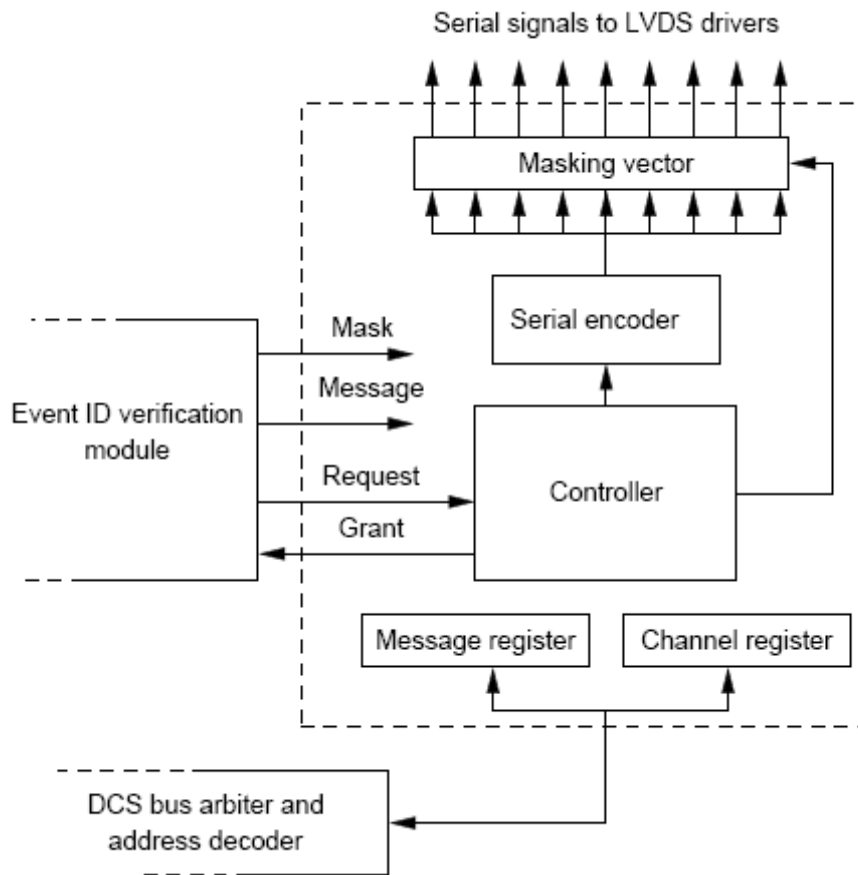


Figure 5-14: Transmitter system. From [Magne]

The transmitter module will request eventIDs from the D-RORCs. The request is a 16 bit word and is sent to all D-RORCs.

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

Table 5-16: Bit map for Trigger module request.

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.

Table 5-17: Request commands.

5.8.1 Transmitter module VHDL Entity Hierarchy

- Transmitter module
 - Serial encoder

The Transmitter module is initiating the serial encoder and setting the masking vector.

5.8.2 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 5-15. The register contains a message register and a channel register.

<i>temp_dcs_data</i>	
<i>dcs_tx_channel</i>	<i>dcs_tx_data</i>
15 – 8	7 - 0

Table 5-18: Bit map for DCS data.

The channel register selects which channel to be masked 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. A flag is raised in to indicate that data are available to be written from the DCS board to the message register. A state machine, see figure 5-16, 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.

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 EventID module.

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*.

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

Table 5-19: Hamming code table

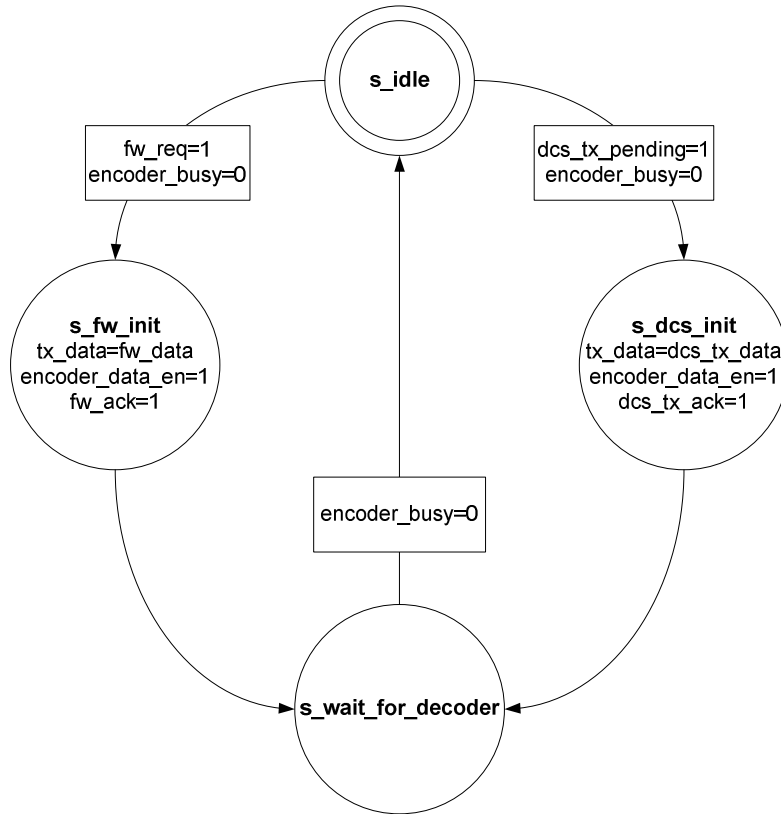


Figure 5-17: State diagram for TX controller

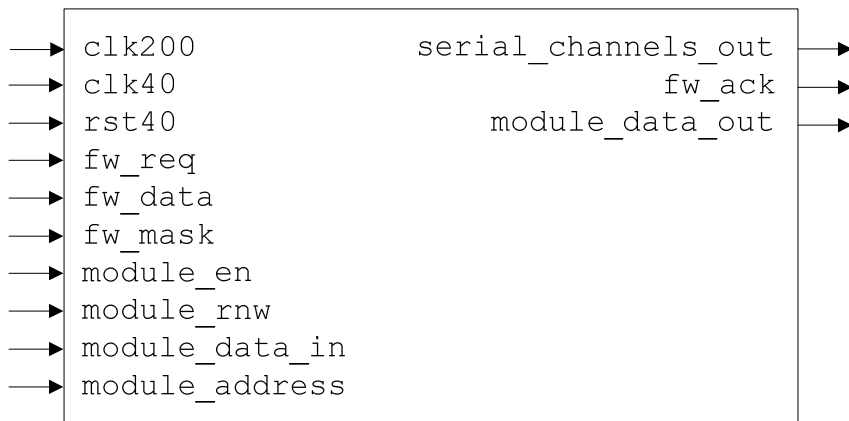


Figure 5-18: Entity for Transmitter Module.

Port Name	Direction	# Bit	Description
rst40	Input	1	std_logic; synchronous reset
clk200	Input	1	std_logic; the clk200 frequency is 200 MHz
clk40	Input	1	std_logic; the clk40 frequency is 40.08 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);
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

Table 5-20: I/O details for Transmitter Module.

5.8.3 Entity Serial Encoder

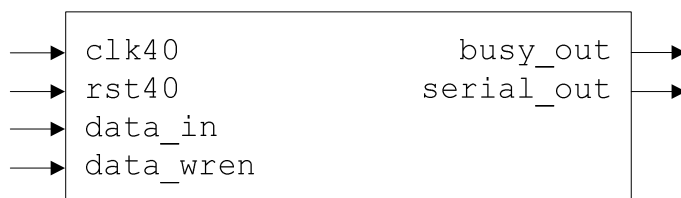


Figure 5-19: Entity for Serial Encoder.

Port Name	Direction	# Bit	Description
rst40	Input	1	std_logic; synchronous reset
clk40	Input	1	std_logic; ; the clock_in frequency is 200 MHz
data_in	Input	1	std_logic_vector;
data_wren	Input	1	std_logic;
busy_out	Output	1	std_logic;
serial_out	Output	1	std_logic;

Table 5-21: I/O details for Serial Encoder.

5.9 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 domains⁴. 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.

⁴ The Receiver module operates in the 200 MHz domain while the internal logic of the BusyBox runs in the 40 MHz domain.

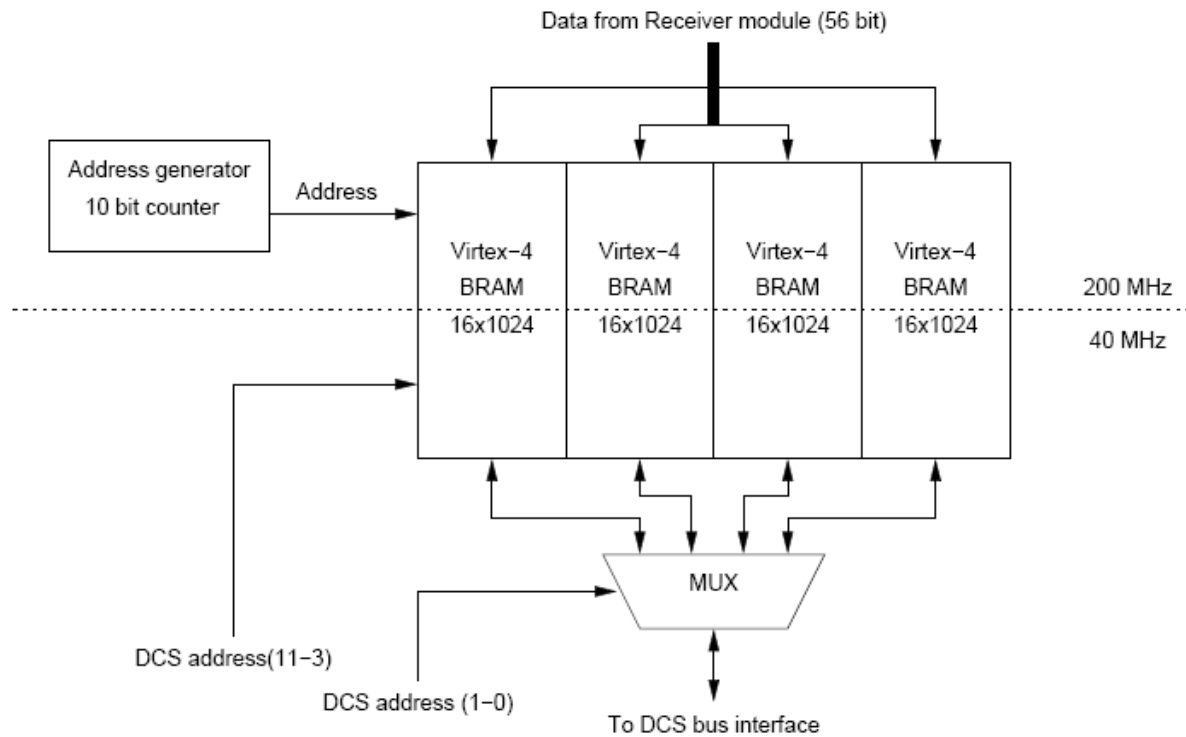


Figure 5-20: Illustration of the RX Memory module. From [Magne].

5.9.1 Entity RX Memory Module

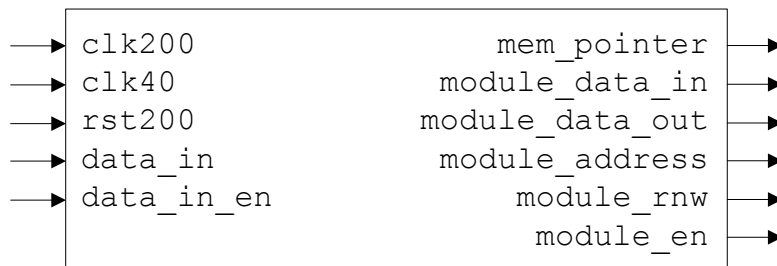


Figure 5-21: Entity for RX Memory Module.

Port Name	Direction	# Bit	Description
clk200	Input	1	std_logic; the clk200 frequency is 200 MHz
clk40	Input	1	std_logic; the clk40 frequency is 40.08 MHz
rst200	Input	1	std_logic; synchronous reset
rst40	Input	1	std_logic; synchronous 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);
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;

Table 5-22: I/O details for RX Memory Module.

5.10 RX Memory Filter Module

The RX Memory filter can be used to filter which messages from specific channels will trigger the write enable signal from 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.

5.10.1 Entity RX Memory Filter Module

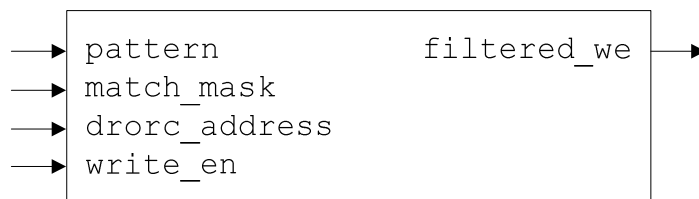


Figure 5-22: Entity for RX Memory Filter.

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

Table 5-23: I/O details for RX Memory Filter.

5.11 Trigger Receiver Module

The optical signals from the CTP are converted to electrical signals by the TTCrx chip on the DCS board into Channel A and Channel B. The Trigger Receiver module decodes the information and stores it in a FIFO in the CDH (Common Data Header) format. Triggers will appear as individual signal at the module outputs.

Channel A transmits the L0 and L1 triggers. Channel B transmits the broadcast message and the individually addressed messages. The addressed messages are decoded into the CDH format and put in a FIFO. The BusyBox extracts the event ID (OrbitID + BunchCountID), event info and event errors from the CDH. Figure 5-23 shows an overview of the Trigger Receiver module.

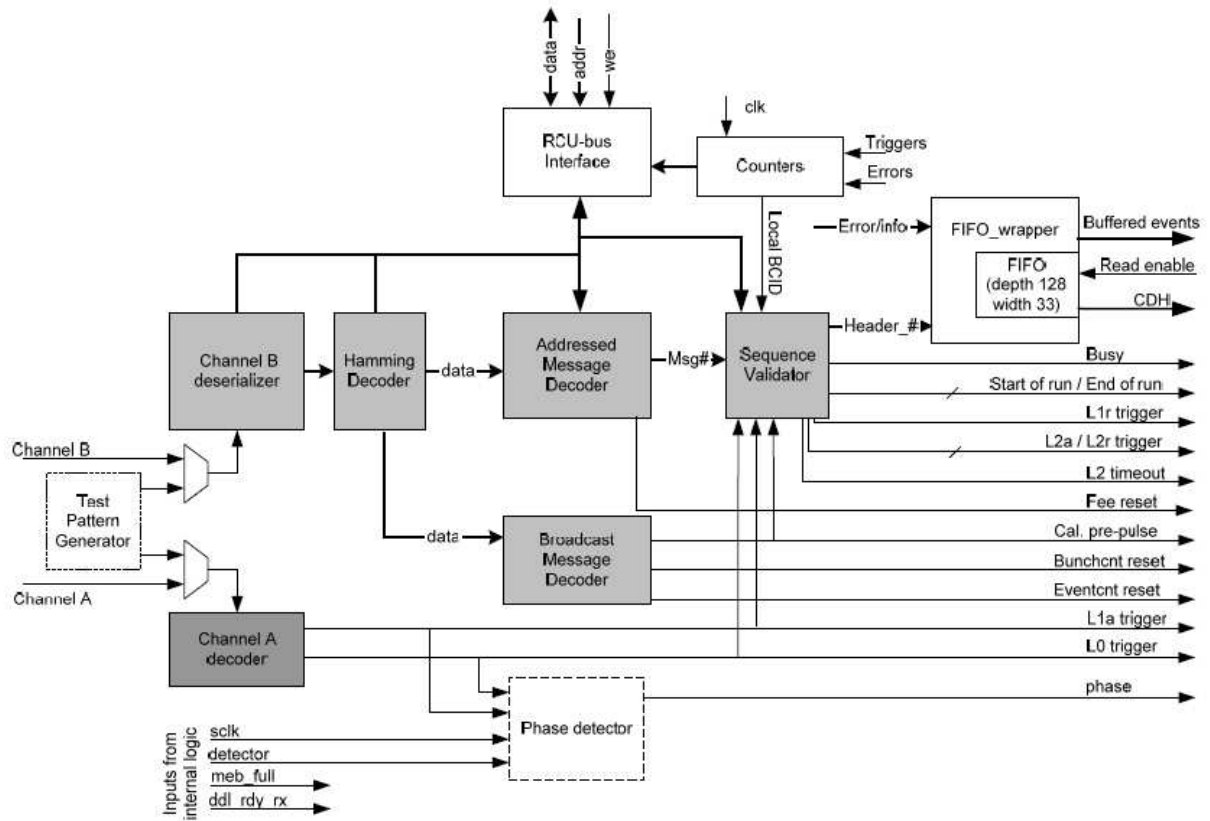


Figure 5-24: Block diagram of the Trigger Receiver module. From [Johan].

5.11.1 Entity Trigger Receiver Module

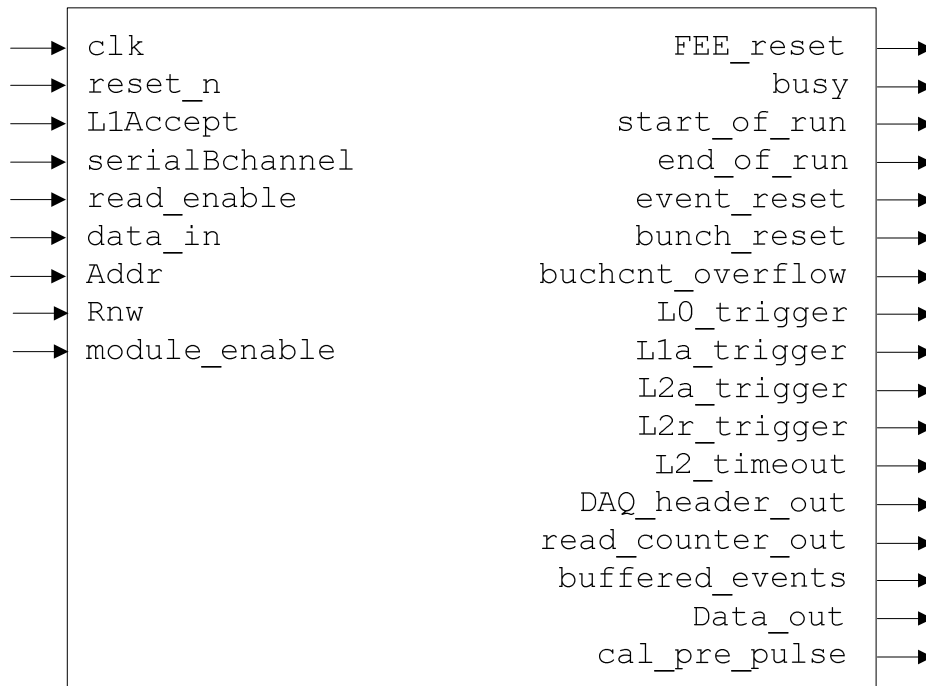


Figure 5-25: Entity for Trigger Receiver Module.

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

Table 5-24: I/O details for Trigger Receiver Module.

5.12 Event ID Verification Module

The Trigger Receiver module's 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 then requests the Transmitter module to read out the data and send it to the D-RORCs. The Receiver module forwards D-RORC data to the D-RORC Inbox Buffer. The Inbox operates in both frequency domains⁵ 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 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 3-1

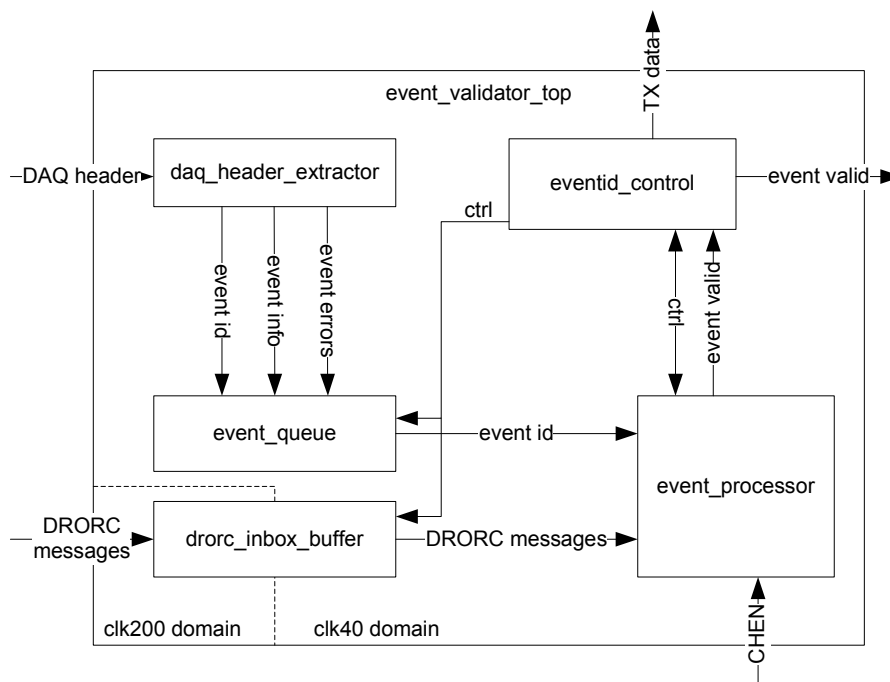


Figure 5-26: Illustration of the structure of event_validator_top module.

The eventid_control block controls the flow of new eventIDs from the event_queue and DRORC messages from the drorc_inbox_buffer to the event_processor. It is also commands the TX controller to transmit requests to the DRORCs. The event_processor block determines when the current event has been verified/validated.

5.12.1 Event_validator_top VHDL Entity Hierarchy

- Event_validator_top
 - i_daq_header_extractor: daq_header_extractor
 - i_event_queue: FIFOsync108x512(Core)
 - i_eventid_control: eventid_control
 - i_event_processor : event_processor

⁵ The Receiver module operates in the 200 MHz domain while the internal logic of the verification module runs in the 40 MHz domain.

- o `i_drorc_inbox_buffer`: `drorc_inbox_buffer` (Core)

5.12.2 Entity `Event_validator_top`

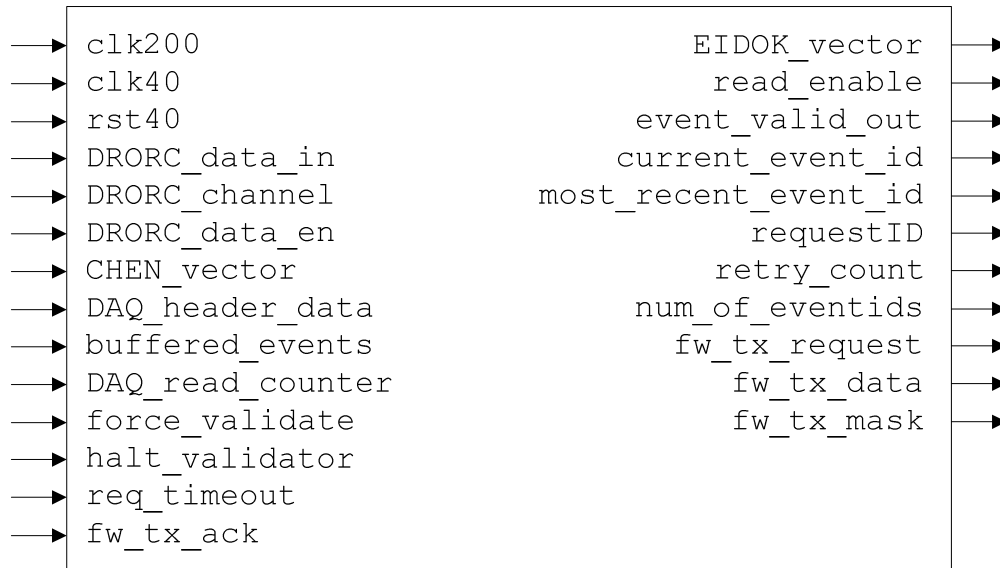


Figure 5-27: Entity for `event_validator_top` module.

Port Name	Direction	# Bit	Description
<code>rst40</code>	Input	1	<code>std_logic</code> ; synchronous reset
<code>clk200</code>	Input	1	<code>std_logic</code> ; the <code>clk200</code> frequency is 200 MHz
<code>clk40</code>	Input	1	<code>std_logic</code> ; the <code>clk40</code> frequency is 40.08 MHz
<code>DRORC_data_in</code>	Input	48	<code>std_logic_vector</code> (47 downto 0);
<code>DRORC_channel</code>	Input	8	<code>std_logic_vector</code> (7 downto 0);
<code>DRORC_data_en</code>	Input	1	<code>std_logic</code> ;
<code>CHEN_vector</code>	Input	120	<code>std_logic_vector</code> (0 to <code>num_of_channels</code>);
<code>DAQ_header_data</code>	Input	33	<code>std_logic_vector</code> (32 downto 0);
<code>buffered_events</code>	Input	4	<code>std_logic_vector</code> (3 downto 0);
<code>DAQ_read_counter</code>	Input	4	<code>std_logic_vector</code> (3 downto 0);
<code>force_validate</code>	Input	1	<code>std_logic</code> ;
<code>halt_validator</code>	Input	1	<code>std_logic</code> ;
<code>req_timeout</code>	Input	16	<code>std_logic_vector</code> (15 downto 0);
<code>fw_tx_ack</code>	Input	1	<code>std_logic</code> ;
<code>EIDOK_vector</code>	Output	120	<code>std_logic_vector</code> (0 to <code>num_of_channels</code>);
<code>read_enable</code>	Output	1	<code>std_logic</code> ;
<code>event_valid_out</code>	Output	1	<code>std_logic</code> ;
<code>current_event_id</code>	Output	36	<code>std_logic_vector</code> (35 downto 0);
<code>most_recent_event_id</code>	Output	36	<code>std_logic_vector</code> (35 downto 0);
<code>requestID</code>	Output	4	<code>std_logic_vector</code> (3 downto 0);
<code>retry_count</code>	Output	16	<code>std_logic_vector</code> (15 downto 0);
<code>num_of_eventids</code>	Output	4	<code>std_logic_vector</code> (3 downto 0);
<code>fw_tx_request</code>	Output	1	<code>std_logic</code> ;
<code>fw_tx_data</code>	Output	8	<code>std_logic_vector</code> (7 downto 0);
<code>fw_tx_mask</code>	Output	120	<code>std_logic_vector</code> (0 to <code>num_of_channels</code>);

Table 5-25: I/O details for Event Validator.

5.12.3 Entity DAQ Header Extractor

This module reads out the DAQ header (also called CDH (Common Data Header)) that the trigger receiver module generates and buffers. The Trigger Receiver Module stores the DAQ header as 9 32 bit words in an internal FIFO. Each word contains information about the received trigger sequence. The DAQ Header Extractor reads out all 9 words and outputs selected fields of information in parallel. Most importantly the EventID is extracted from the header. The extracted information is to a FIFO. The information is also forwarded to the Control and Status Register module.

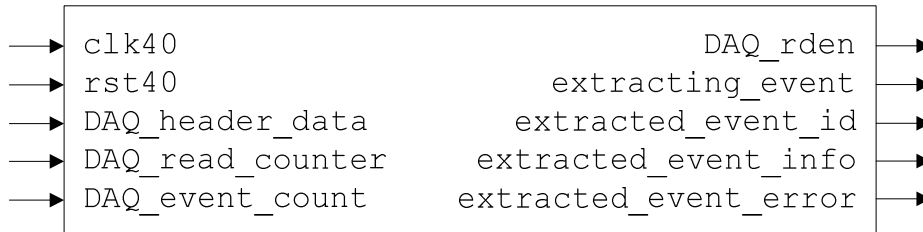


Figure 5-28: Entity for EventID Extractor.

Port Name	Direction	# Bit	Description
clk40	Input	1	std_logic; the clock_in frequency is 40.08 MHz
rst40	Input	1	std_logic; synchronous reset
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
DAQ_event_count	Input	4	std_logic_vector(3 downto 0); counts numbers of buffered events in the FIFO
DAQ_rden	Output	1	std_logic;
extracting_event	Output	1	std_logic; status output
extracted_event_id	Output	36	std_logic_vector(35 downto 0); the extracted orbit end bunch cross IDs
extracted_event_info	Output	13	std_logic_vector(12 downto 0);
extracted_event_error	Output	25	std_logic_vector(

Table 5-26: I/O details for EventID Extractor.

5.12.4 Entity EventID Control

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

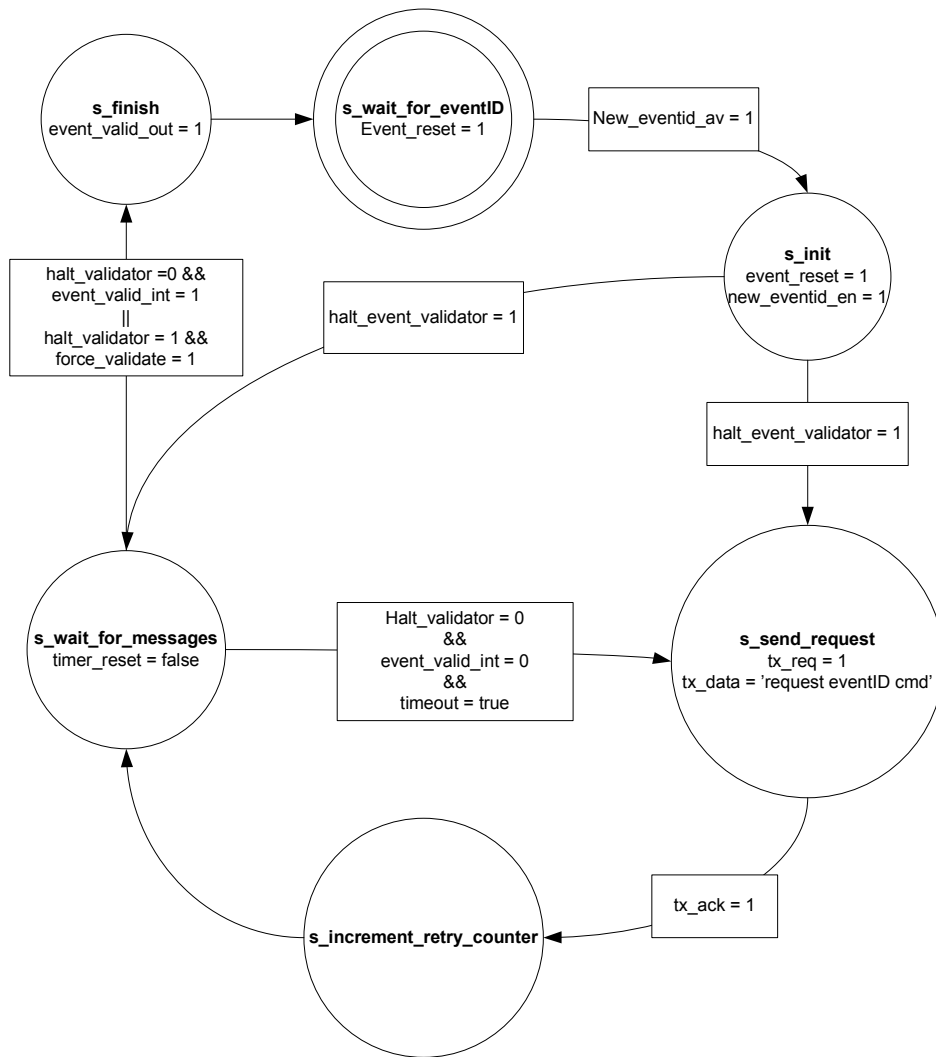


Figure 5-29: State diagram for EventID Controller.

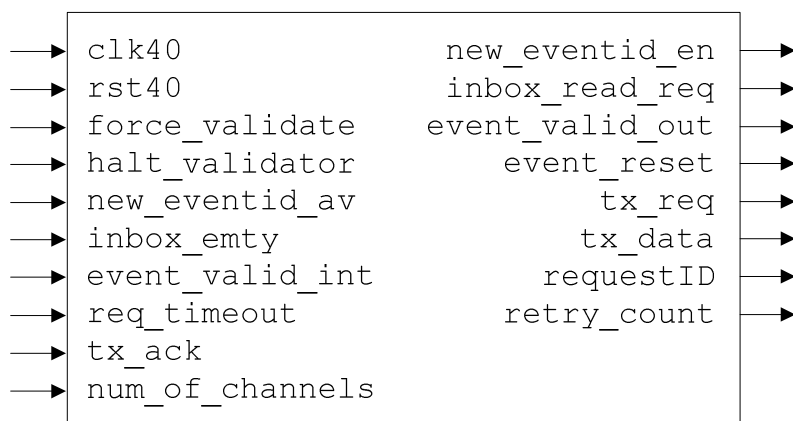


Figure 5-30: Entity for EventID Control.

Port Name	Direction	# Bit	Description
clk40	Input	1	std_logic; the clk40 frequency is 40.08 MHz
rst40	Input	1	std_logic; synchronous reset
force_validate	Input	1	std_logic;
halt_validator	Input	1	std_logic;
new_eventid_av	Input	1	std_logic;
inbox_empt	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_eventid_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);

Table 5-27: I/O details for EventID Control.

5.12.5 Entity EventID Processor

In this module all the verification occurs and based on the CHEN 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 5-31: Entity for EventID Processor.

Port Name	Direction	# Bit	Description
clk40	Input	1	std_logic; the clk40 frequency is 40.08 MHz
rst40	Input	1	std_logic; synchronous 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;

Table 5-28: I/O details for EventID Processor.

5.13 Busy Controller Module

The `busy_controller` module is responsible for generating the `BUSY` signal. It does this by evaluating inputs from other modules and counting occupied Multi Event Buffers (MEB) in the FEE.

There are four conditions that sets the busy signal high and only one have to be true:

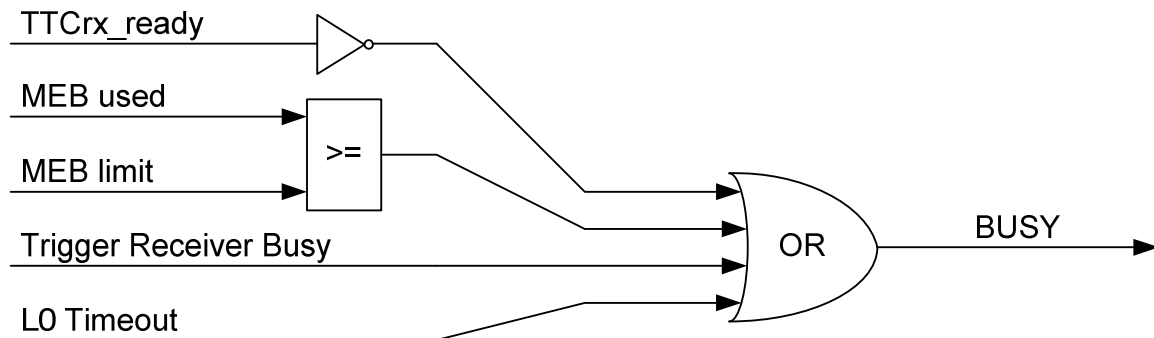


Figure 5-32: BUSY generation.

1. The TTCrx ready (`ttxcrx_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 this time period. The busy time can be set manually with a register in the Control and Status Register module.
2. The `busy_controller` keeps track of how many Multi Event Buffers (MEB) are occupied in the FEE. If the number of occupied MEBs is greater or equal than then a programmable limit then `BUSY` is asserted.
3. The `BUSY` is asserted whenever the trigger receiver is busy, which means we are in the middle of a trigger sequence.
4. 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.

The number of occupied MEBs is calculated by monitoring the triggers and the validation of event IDs process.

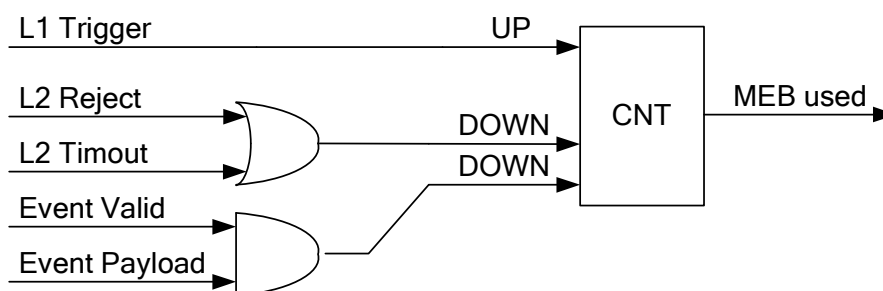


Figure 5-33: MEB counter.

The counter is incremented when a L1 trigger is seen. It decrements on a L2 Reject or L2 Timeout and when event valid and event payload is active at the same time. See Figure 5-33.

5.13.1 Entity Busy Controller Module

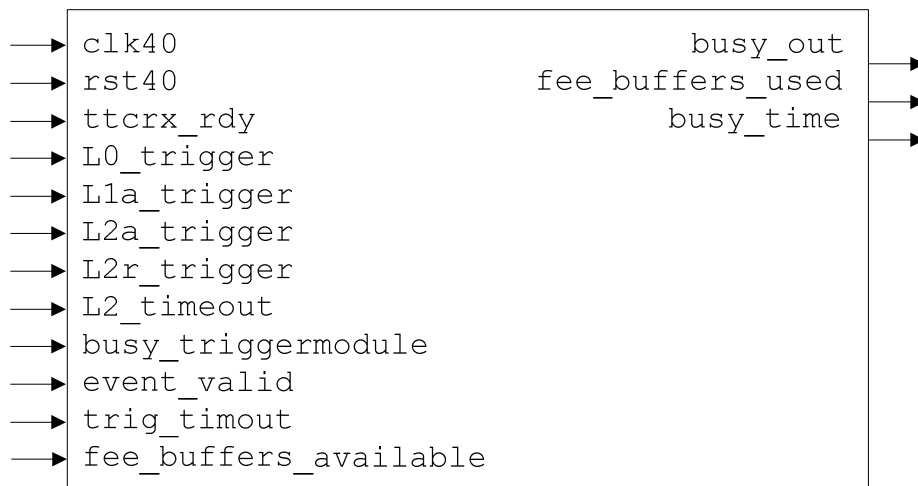


Figure 5-34: Entity for Busy Controller Module.

Port Name	Direction	# Bit	Description
<i>rst40</i>	<i>Input</i>	<i>1</i>	<i>std_logic; synchronous resets</i>
<i>clk40</i>	<i>Input</i>	<i>1</i>	<i>std_logic; the clk40 frequency is 40.08 MHz.</i>
<i>ttcrx_rdy</i>	<i>Input</i>	<i>1</i>	<i>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.</i>
<i>L0_trigger</i>	<i>Input</i>	<i>1</i>	<i>std_logic; N/A</i>
<i>L1a_trigger</i>	<i>Input</i>	<i>1</i>	<i>std_logic; L1a_trigger output from trigger_receiver_busy_model. Starts buffering data in Fee if L1a_trigger signal is asserted.</i>
<i>L2a_trigger</i>	<i>Input</i>	<i>1</i>	<i>std_logic; N/A</i>
<i>L2r_trigger</i>	<i>Input</i>	<i>1</i>	<i>std_logic; L1r_trigger output from trigger_receiver_busy_model. Overwrites buffers in Fee if L2r_trigger signal is asserted.</i>
<i>L2_timeout</i>	<i>Input</i>	<i>1</i>	<i>std_logic; L2_timeout output from trigger_receiver_busy_model. Overwrites buffers in Fee if L2_timeout signal is asserted.</i>
<i>busy_triggermodule</i>	<i>Input</i>	<i>1</i>	<i>std_logic; busy_triggermodule output from trigger_receiver_module.</i>
<i>event_valid</i>	<i>Input</i>	<i>1</i>	<i>std_logic;</i>
<i>trig_timeout</i>	<i>Input</i>	<i>16</i>	<i>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.</i>
<i>fee_buffers_available</i>	<i>Input</i>	<i>4</i>	<i>std_logic_vector(3 downto 0); Holds the numbers of buffers assumed on the FEE. Register 0x2009. Default is 4.</i>
<i>busy_out</i>	<i>Output</i>	<i>1</i>	<i>std_logic; busy_out is asserted when busy conditions are met.</i>
<i>fee_buffers_used</i>	<i>Output</i>	<i>4</i>	<i>std_logic_vector(3 downto 0);</i>
<i>busy_time</i>	<i>Output</i>	<i>32</i>	<i>std_logic_vector(31 downto 0); busy_time count numbers of clock cycles busy signal is asserted.</i>

Table 5-29: I/O details for Busy Controller Module.

5.14 Control and Status Registers

This module has information about register and control signals available for the BusyBox. See chapter 8 for more information.

5.14.1 Entity Control and Status Register

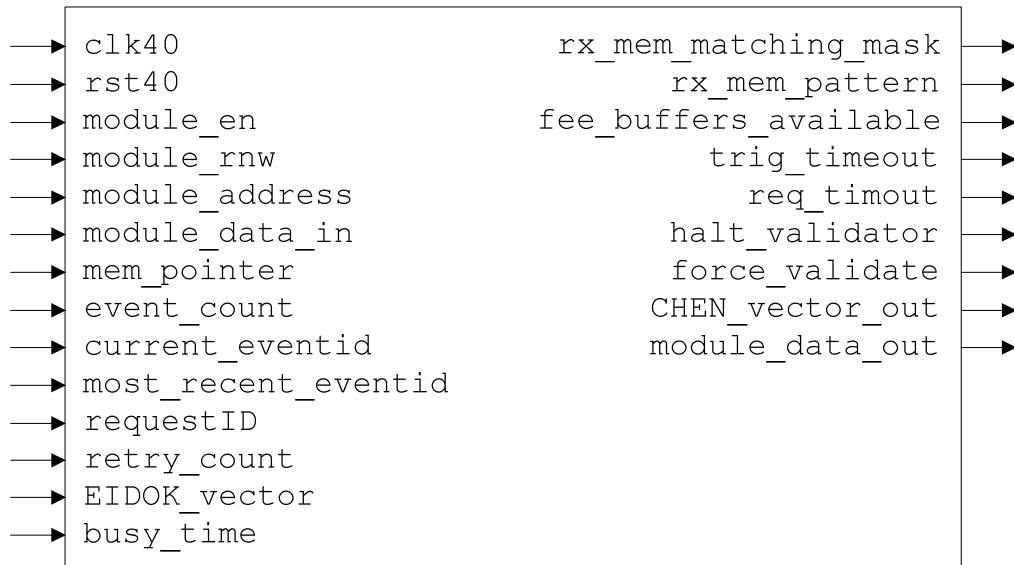


Figure 5-35: Entity for Control and Status Registers.

Port Name	Direction	# Bit	Description
clk40	Input	1	std_logic; the clk40 frequency is 40.08 MHz
rst40	Input	1	std_logic; synchronous resets
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_timeout	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);

Table 5-30: I/O details for Control and Status Registers.

6 Functional verification of the BusyBox firmware

6.1 Introduction

This chapter describes a testbench for the BusyBox firmware. The testbench is designed to emulate a complete environment around the BusyBox top level module `busylogic_top`. The FPGA specific wrappers are not included in the test setup.

The BusyBox interacts with the outside world through numerous interfaces:

1. Clocks & Reset
2. Trigger System; L1Trig and SerialB
3. DCS bus
4. Serial RX and TX to DRORCs

The testbench is implemented in the VHDL file `tb_trigger_busybox.vhd`. It emulates the outside world of all these interfaces, providing stimuli and interaction. Figure 6-1 shows a structural overview of the testbench. The Unit Under Test (UUT) is the rectangular box in the middle. Other rectangular boxes are modules instantiated in the testbench. Elliptic shapes are processes defined in the testbench.

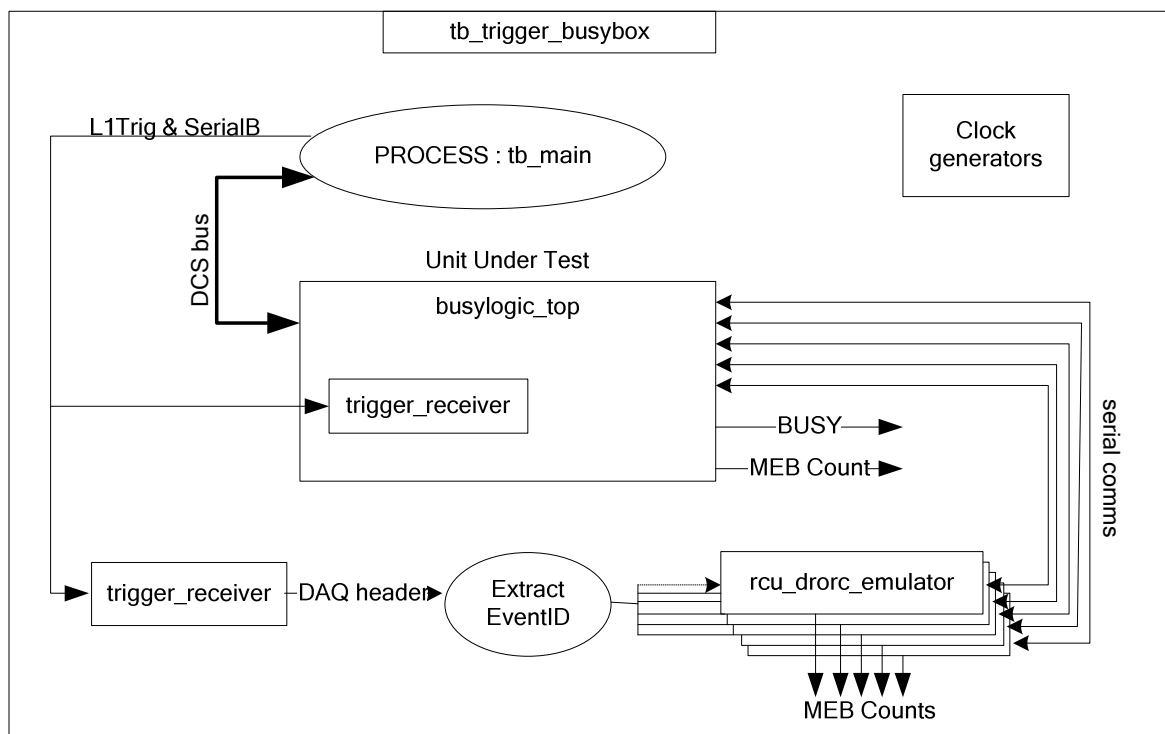


Figure 6-1: Structural overview of the testbench for the BusyBox logic.

The clock generators are simply concurrent procedure calls to `clk_gen` in the `busybox_tb_pkg`. See section 6.2.1 for more details on the procedure. The system needs two clocks to operate, one at 200 MHz and one at 40 MHz.

```
p_clockA : clkgen(clkA_period, clk_en, clk200);
p_clockB : clkgen(clkB_period, clk_en, clk40);
```

The process `tb_main` generates trigger sequences and access registers in the UUT over the DCS bus. An extra `trigger_receiver` is instantiated in the testbench to acquire the same trigger information as produced by the `trigger_receiver` in the UUT. The decoded triggers are read out from the extra `trigger_receiver` and the `eventID` is extracted and pushed to the `rcu_drorc_emulator` modules which communicate with the BusyBox over serial links.

The trigger sequences are easily generated by a procedure call to `run_sequence` defined in the `busybox_tb_pkg` VHDL package which is described in section *The busybox_tb_pkg package* 6.2.1.

6.2 Support packages

The testbench makes use of the following packages:

- `ieee.std_logic_1164.all`; IEEE package defining `std_logic` type and derived types.
- `ieee.numeric_std.all`; - IEEE package for numeric operations on `std_logic` based (sub)types
- `ieee.math_real.all`; - IEEE package for math operations, used for random generation.
- `std.textio.all`; - Package for modifying and printing text strings.
- `work.tb_pkg.all`;
- `work.busybox_tb_pkg.all`; - Package containing various definitions of types, functions and procedures used in the testbench.

6.2.1 The busybox_tb_pkg package

This package was written specifically for this testbench and holds definitions required by it.

A record type is defined that contains all signals related to the bus interface with the DCS board. Procedures for performing read and write operations on this interface are also defined.

```
type t_bb_bus_record is record
  strobe_n      : std_logic;
  RnW           : std_logic;
  addr          : std_logic_vector(15 downto 0);
  data          : std_logic_vector(15 downto 0);
  ack_n        : std_logic;
end record t_bb_bus_record;
```

Procedures for performing master read and write bus transactions using the bus record types defined in the package. The procedures are used for accessing and testing the internal busybox registers during simulation. They must be called from a sequential process and will block the calling process until they return.

```
procedure bb_bus_read (
  constant addr  : in    std_logic_vector;
  variable data  : out   std_logic_vector;
  signal bb_bus  : inout t_bb_bus_record);

procedure bb_bus_write (
  constant addr, data : in    std_logic_vector;
  signal bb_bus      : inout t_bb_bus_record);
```

The `clk_gen` procedure makes it easy to generate clocks in the testbench. The procedure is called from the concurrent section of a module and will act as a process.

- The period is specified by a constant and cannot be changed afterwards.
- The clock output can be enabled or disabled by the `clk_en` input.

```

procedure clkgen(
constant period      : in time;
signal clk_en       : in boolean;
signal clk           : out std_logic);

```

For a convenient and compact definition of a trigger sequence a record type is defined that holds all relevant parameters of a trigger sequence. A defined sequence is executed by calling a procedure (`run_sequence`) with an instance of this record type is one of its parameters.

```

type sequence_type is record
  description      : string(1 to 80);
  L0_enable       : boolean;
  L1_enable       : boolean;
  L1Msg_enable    : boolean;
  L2Msg_enable    : boolean;
  L2Accept       : boolean;
  L1_latency      : time;
  L1Msg_latency   : time;
  L2Msg_latency   : time;
  L1_ESR         : std_logic;
  L1_CIT         : std_logic;
  L1_SwC         : std_logic;
  L1_RoC         : std_logic_vector(3 downto 0);
  L1_Class       : std_logic_vector(49 downto 0);
  L2_ESR         : std_logic;
  L2_CIT         : std_logic;
  L2_SwC         : std_logic;
  L2_Cluster     : std_logic_vector(5 downto 0);
  L2_Class       : std_logic_vector(49 downto 0);
  BCID           : std_logic_vector(11 downto 0);
  OrbitID        : std_logic_vector(23 downto 0);
end record sequence_type;

```

A more detailed description of the trigger sequence parameters follow:

Parameter	Description	Default value
<i>description</i>	A string of 80 characters that can be used to describe the sequence. This string will be printed to console when the sequence is executed during simulation runs.	"Default sequence L0-L1A-L1Msg-L2AMsg."
<i>L0_enable</i>	Include the L0 trigger in the sequence	true
<i>L1_enable</i>	Include the L1 trigger in the sequence	true
<i>L1Msg_enable</i>	Include the L1 Message in the sequence	true
<i>L2Msg_enable</i>	Include the L2 Message in the sequence	true
<i>L2Accept</i>	true => Level 2 Accept Message false => Level 2 Reject Message	true
<i>L1_latency</i>	Latency from rising edge of L0 trigger to rising edge of L1 trigger on the L1Trig signal	5.3 μ s
<i>L1Msg_latency</i>	Latency from rising edge of L0 trigger to the start bit of the L1 Message on the serialB signal	6.3 μ s
<i>L2Msg_latency</i>	Latency from rising edge of L0 trigger to the start bit of the L2	85 μ s

	Message	
L1_ESR	Level 1 Message content : Enable Segemented Readout	'0'
L1_CIT	Level 1 Message content: Calibration trigger flag	'0'
L1_SwC	Level 1 Message content: Software trigger	'0'
L1_RoC	Level 1 Message content: ReadOut Chamber	"0000"
L1_Class	Level 1 Message content: Level 1 Class trigger state flag	(others => '0')
L2_ESR	Level 2 Message content: Enable Segmented Readout	'0'
L2_CIT	Level 2 Message content: Calibration trigger flag	'0'
L2_SwC	Level 2 Message content: Software trigger flag	'0'
L2_Cluster	Level 2 Message content: Cluster trigger flag	(others => '0')
L2_Class	Level 2 Message content: Level 2 Class trigger flag	(others => '0')
BCID	Level 2 Message content: Bunch Crossing ID	X"789"
OrbitID	Level 2 Message content: Orbit ID	X"123456"

- It is a required relationship that $L1_latency < L1Msg_latency < L2Msg_latency$.
- The BCID and OrbitID values combined make up the eventID.
- Message content entries besides the eventID are not relevant for the BusyBox operation.
- A constant with default values for all parameters is defined in the busybox_tb_pkg. This constant can be used as init value to instantiate a record with default values.

Refer to documentation for the ALICE trigger system for more information on the message contents.

Procedure for executing a sequence and driving the L1Trig and SerialB signals.

```

procedure run_sequence (
signal clk      : in std_logic;
constant seq    : in sequence_type;
signal L1trig   : out std_logic;
signal serialb  : out std_logic);

```

The procedure produces a sequence in the following order L0 trigger – L1 trigger – L1 Message – L2 Message.

The different stages can be enabled and disabled by setting the flags in the sequence record type. The L0 trigger is executed at the first rising edge of the testbench clock after the procedure is called. All other latencies are referenced from this time. If e.g. the L2 Message latency has already passed when the L1 message is finished, it will be executed right away. This is true for all latencies defined in the trigger sequence record type.

6.3 The RCU and DRORC emulator module

This module has been designed to emulate the RCU and DRORC devices from the BusyBox's point of view. Only the functions required for the operation of the BusyBox are emulated. In the real system the RCUs will read event data from the Front End Cards and push it to the DRORCs. The event data is preceded by a Common Data Header (CDH) which contains information related to the event, amongst others the EventID. When the DRORC has received an event, including the CDH, from the RCU the EventID is extracted and put in a FIFO queue. When the BusyBox requests a new EventID, the DRORC will look for it in this FIFO queue. Since the size of the event data will vary between events and FEE patches (RCU with FECs attached) the time taken from an event has been triggered until the EventID ends up in the DRORCs EventID queue will vary as well.

To emulate this behaviour the rcu_drorc_emulator module has to FIFOs for eventIDs. The first is used for storing EventIDs as they are triggered and the second to hold EventIDs that have been transmitted to the DRORCs. Internally the module generates random times from 100 to 500 μ s that it will wait before moving an EventID

from the first FIFO to the second. The random time emulates the variance in event data sizes. The concept is illustrated in Figure 6-2.

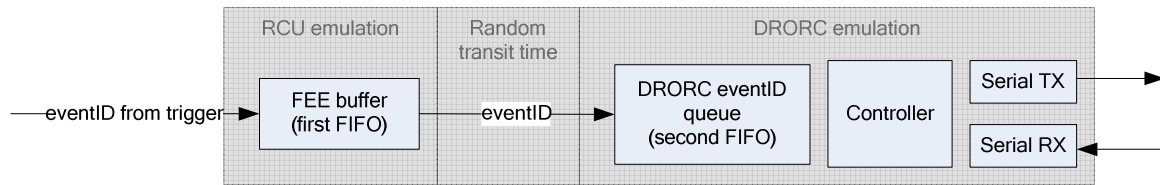


Figure 6-2: Illustration of the RCU and DRORC emulation.

The number of EventIDs stored in the FEE buffer corresponds to Multi Event Buffers (MEBs) in use on the real system. This number is monitored by the testbench to check that it never exceeds the specified maximum numbers events that can be stored on the FEE. The DRORC emulator takes care of the communications with the BusyBox. Requests/commands from the BusyBox are handled in the same manner as for the real DRORC. Serial receiver and transmitter modules are used for communication with the BusyBox.

6.4 Testbench execution flow

The process `tb_main` controls the simulation. It begins by setting some signals to default values and resetting the UUT and any support modules. At line 7 the `dcm_lock` signal is asserted to indicate to the `busylogic_top` module the the DCM has locked on the external clock. Then the reset is released. At lines 11 and 12 it writes to two configuration registers to set the MEB limit to 4 and release the busybox FSM (set the halt FSM bit to 0). All other registers are left at their default value. Now the BusyBox should release the BUSY and be ready to receive triggers.

```

1  clk_en <= true;
2  dcm_lock <= '0';
3  serialb <= '1';
4  Lltrig <= '0';
5  areset <= '1';
6  wait for 200 ns;
7  dcm_lock <= '1';
8  wait for 10 us;
9  areset <= '0';
10 header("Testbench for BusyBox", "A sequence of basic triggers.");
11 bb_bus_write(X"2009", X"0004", dcs_bus); -- set MEB to 4
12 bb_bus_write(X"200A", X"0000", dcs_bus); -- release busybox FSM
13 wait for 500 ns;
14 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
15 wait for 100 us;
16 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
17 wait until busy_out = '0';
18 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
19 wait until busy_out = '0';
20 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
21 wait until busy_out = '0';
22 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
23 wait until busy_out = '0';
24 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
25 wait until busy_out = '0';

```

```

26 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
27 wait until busy_out = '0';
28 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
29 wait until busy_out = '0';
30 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
31 wait until busy_out = '0';
32 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
33 wait until busy_out = '0';
34 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
35 wait until busy_out = '0';
36 run_sequence(clk40, trigger_sequence1, Lltrig, serialB);
37 wait until busy_out = '0' and BusyBox_MEB_cnt = "0000";
38 clk_en <= false;
39 wait;

```

Next, the process starts sending trigger sequences by calling the `run_sequence` procedure. A wait statement ensures that `BUSY` is not asserted before sending a trigger sequence. The simulation will end when all statements in this process have completed. The final statements wait until the `BUSY` has been de-asserted and the `BusyBox` MEB count has reached 0. Then the clocks are disabled and the process goes to sleep.

During the entire simulation a concurrent assert statement makes sure that the MEB counts from all the `rcu_drorc_emulator` modules are never higher than the MEB count of the `BusyBox`. This is important as it tests whether the `BusyBox` will toggle the `BUSY` output correctly so that the `FEE` cannot experience buffer overflows during operation.

```

assert RCU_MEB_cnt_array(i) <= BusyBox_MEB_cnt
    report "Possible Buffer overflow detected!"
    severity warning;

```

Below is an excerpt from the output/console when running the testbench.

```

# 10400 ns: bb_bus_write: ADDR 0x2009 DATA 0x0004
# 10650 ns: bb_bus_write: ADDR 0x200A DATA 0x0000
# 11200 ns: Running trigger sequence number 1
# 11200 ns: Sending Level 0 Trigger.
# 16500 ns: Sending Level 1 Trigger. Time since sequence init : 5300 ns
# 17500 ns: Sending Level 1 Message. Time since sequence init : 6300 ns
# 96200 ns: Sending Level 2 Accept Message. Time since sequence init : 85000 ns
# ===== Trigger Receiver DAQ event FIFO =====
# Event Info : 0xA9501800
#           + Bit 11 : L2 Accept received.
#           + Bit 12 : Include payload.
# Event Error : 0x00002000
#           + Bit 13 : Buondary L1.
# DAQ1       : 0x02000789
# DAQ2       : 0x00123456
# DAQ3       : 0x00000000
# DAQ4       : 0x0100028E
# DAQ5       : 0x00000000
# DAQ6       : 0x00000000
# DAQ7       : 0x00000000
# =====
# 204850 ns: Running trigger sequence number 2
# 204850 ns: Sending Level 0 Trigger.
# 210150 ns: Sending Level 1 Trigger. Time since sequence init : 5300 ns
# 211150 ns: Sending Level 1 Message. Time since sequence init : 6300 ns
# 289850 ns: Sending Level 2 Accept Message. Time since sequence init : 85000 ns
# ===== Trigger Receiver DAQ event FIFO =====
# Event Info : 0xA9501800

```

```
# + Bit 11 : L2 Accept received.
# + Bit 12 : Include payload.
# Event Error : 0x00002000
# + Bit 13 : Buondary L1.
# DAQ1 : 0x02000789
# DAQ2 : 0x00123456
# DAQ3 : 0x00000000
# DAQ4 : 0x010000D0
# DAQ5 : 0x00000000
# DAQ6 : 0x00000000
# DAQ7 : 0x00000000
# =====
```

6.5 Running the simulation in QuestaSim/ModelSim

- 1) *start questa/modelsim*
- 2) *cd to trunk/simulation*
- 3) *source project_setup.tcl*
- 4) *vsim tb_trigger_busybox*
- 5) *add wave **
- 6) *run -all*

Some files (only testbench files) may fail compilation because they use some features from VHDL 2008. Set the compiler to use VHDL 2008 to compile these files.

7 BusyBox DCS board

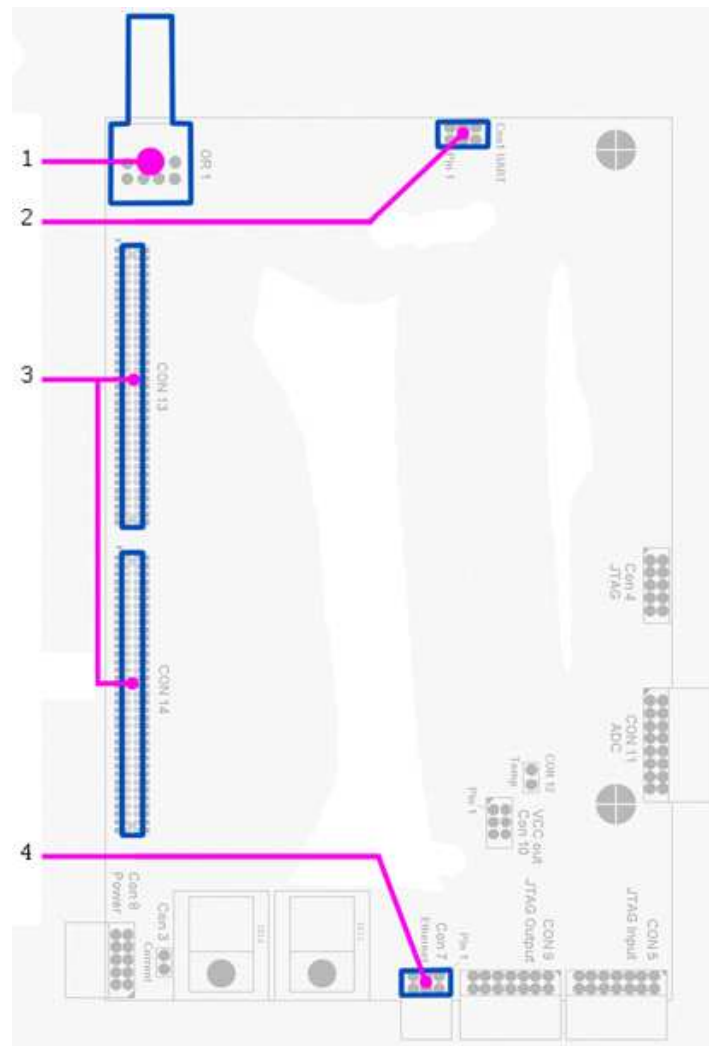


Figure 7-1: PCB layout of DCS board.

#	Type	Description
1	Optical input	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

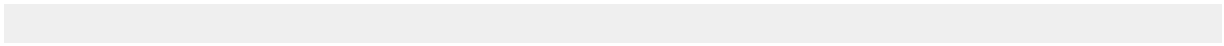
Table 7-1: Connectors on the DCS board.

7.1 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. The latest firmware is available here:

https://wikihost.uib.no/ift/index.php/Electronics_for_the_Time_Projection_Chamber_%28TPC%29#DCS_board_firmware_for_TPC_and_PHOS

Here a description of how to update the DCS board flash device is also given.



8 BusyBox Communication

This chapter discusses the how the BusyBox handles communication with the D-RORCs.

8.1 BusyBox - DRORC Communication

A robust serial communication has been developed for communication between the BusyBox and the D-RORCs.

- Serial, 48 bit oriented, bidirectional data transfer can be made at up to

8.1.1 Physical Layer

Category 5 cables with RJ-45 connectors are the communication channel between the BusyBox and D-RORCs. Built in I/O blocks in the Virtex-4 FPGA is set to I/O standard specified as LVDS_25 and DIFF_TERM attribute is set to true.

Pin	Pair	Wire	Color
1	3	1	 white/green
2	3	2	 green
3	2	1	 white/orange
4	1	2	 blue
5	1	1	 white/blue
6	2	2	 orange
7	4	1	 white/brown
8	4	2	 brown

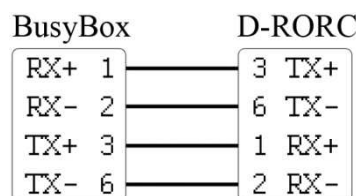


Figure: 8-1: RJ-45 pin connection scheme.

8.1.2 Data Link Layer

Serial data is transmitted with NRZ encoding at a 40 MHz bit rate. Three 16 bit data packages are sent from the D-RORCs to the BusyBox on request from the BusyBox. The packages are concatenated to one 48 bit message in the BusyBox receiver module. The messages from the BusyBox to the D-RORCs are 16 bit long.

The receiver samples the incoming serial data at 200 MHz into a 98 bit shift register. When bits 1 to 8 are '00001111' and bits 96 to 98 are '000' in the shift register, the capture condition is valid, and the data is passed to the majority vote.

Bit range	1	2	3	4	5	6	7	8	.	.	.	96	97	98
Bit value	0	0	0	0	1	1	1	1	x	x	x	0	0	0

Table 8-1: Capture condition for a data frame.

One bit period is 5 cycles and the three middle bits are run through a majority gate. The captured data word is 17 bit long with the LSB as the parity bit. If there is any parity error or timeout error during the data capture data is discarded. The timeout clock counts 110 cycles from the first received bit before a timeout error is issued. When three 16 bit data packages are received without error they are concatenated into a 48 bit package and stored.

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

Table 8-2: Hamming code table.

Messages from the BusyBox to the D-RORCs are Hamming encoded with an 8:4 code applied and sent in series. Each bit is cycled five times and the transmission starts with two start bits followed by 16 data bit, one parity bit and a stop bit.

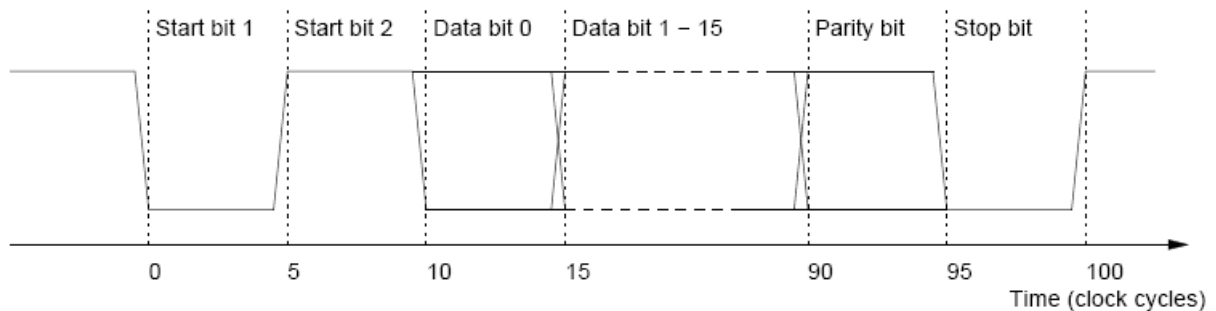


Figure 8-1: One serial data transmission.

8.1.3 Network Layer

The BusyBox will request event IDs from all active D-RORCs when a Level 2 Accept trigger is issued. The transmitter will keep sending new request at a programmable time interval until all the D-RORC have replied. Request can either be issued automatically by the firmware or manually by the DCS software.

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

Table 8-3: Bit-map for BusyBox Messages.

The messages from the D-RORCs are 48 bits long and the time to receive one message is 1.2 μ s. Every channel is checked every 120 clock cycle, i.e. 3 μ s.

47 - 44	43 - 32	31 - 8	7 - 0
Request ID	Bunch Count ID	Orbit ID	D-RORC ID

Table 8-4: Bit-map for D-RORC messages.

8.1.4 Sequence Diagrams

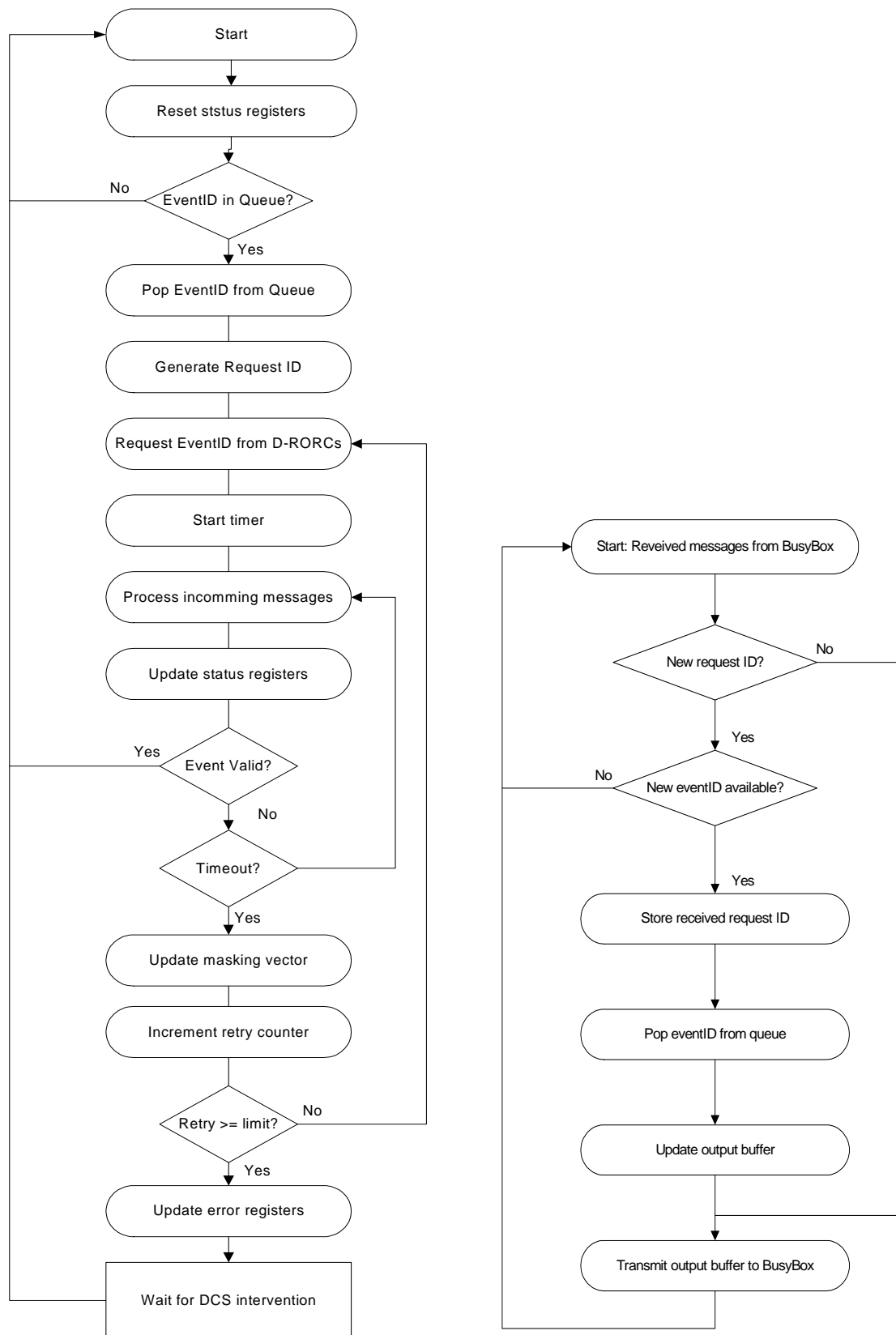


Figure 8-2: Sequence diagram for the BusyBox (on the left) and D-RORC (on the right)

See chapter 4.5 and 4.6 for more information on how the sending and receiving hardware modules work.

8.2 TTCrx Communication

The trigger information from the LTU is acquired by the TTC receiver chip (TTCrx) on the DCS board. The Trigger Receiver module on the BusyBox decodes the information into three parts: Channel A, Channel B and BC.

Channel A

L0 and L1 triggers are transmitted on Channel A using LVDS from the TTCrx chip

Channel B

Channel B transmits two types of messages: Broadcast messages and individual address messages. The broadcast message is decoded into a pre-pulse bit, an event count reset bit and a bunchcount reset bit. The individual addressed messages are decoded into L1a, L2a, L2r and RoI messages.

BC

The Bunch Count, 40 MHz, is distributed to the BusyBox logic and DCS board.

8.3 DCS Communication

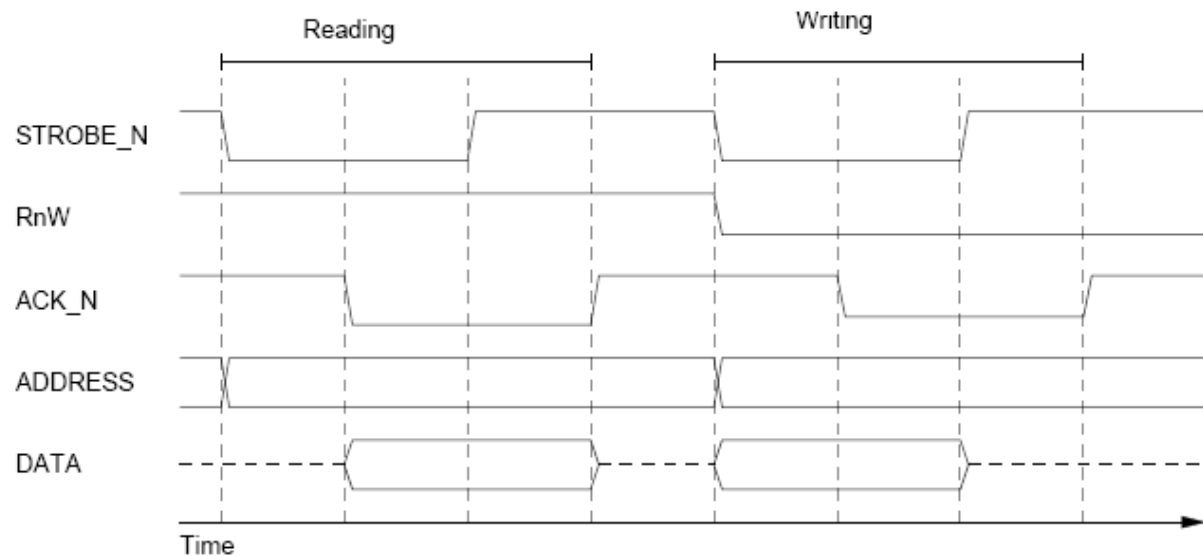


Figure 8-3: Read and write transactions on the DCS bus. From [Magne]

Communication between the DCS board FPGA and the BusyBox motherboard FPGA is a 16 bit asynchronous handshake protocol. The DCS card is the master and the FPGA is the slave. A transaction starts with STROBE_N on the DCS and ACK_N from the BusyBox. RnW indicates read/write and when reading the BusyBox drives the data lines. When writing, the DCS board drives the data lines. If an address given by the DCS board does not point to any module/sub-module in the BusyBox firmware the request is be ignored, and the transaction times out.

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

Table 8-5: Bit-mapping of DCS bus address.

8.4 LTU Communication

The BusyBox has two LEMO contacts. The busy signal is driven through a CERN certified LVDS driver via coaxial cables from the counting room to the LTU in the experiment hall.

9 Getting started with the BusyBox

This chapter will give some insight in how to get started with the BusyBox and make changes to the firmware. In addition the CTP emulator will be discussed.

9.1 Introduction

TCL scripts sets up projects in Xilinx ISE Design Suit and Mentor Graphics' QuestaSim. All the files needed for the BusyBox are in a repository. The scripts are written to implement the files to checkout and build or simulate the design.

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.

9.2 SVN Repository and Project Setup

The BusyBox firmware has been written in Xilinx ISE Design Suite and tested in Mentor Graphics' QuestaSim. The source code and other relevant files for the BusyBox are stored in a SVN repository to make it easy to see the latest work done.

9.2.1 SVN Repository

A repository has been created to have a place where data is stored and maintain for future retrieval to manage ongoing development of digital documents in the BusyBox project. Any changes in the document are identified by incrementing an associated number, termed the revision number. The University of Bergen uses a SVN repository.

TortoiseSVN is an open source revision control software for Microsoft Windows. To download the software, go to:

<http://tortoisesvn.net/downloads>

TortoiseSVN is a shell extension that is integrated into the Windows explorer. So, after you have downloaded and installed TortoiseSVN, open the explorer and right-click on any folder you like to bring up the context menu where you will find all TortoiseSVN commands.

Select "TortoiseSVN" and "Create repository here".

Then you right-click on the folder again and select "Import". The URL for the repository is:

https://svn.ift.uib.no/svn/busybox_firmware

Username and password can be acquired by contacting the microelectronics division at IFT, UIB.

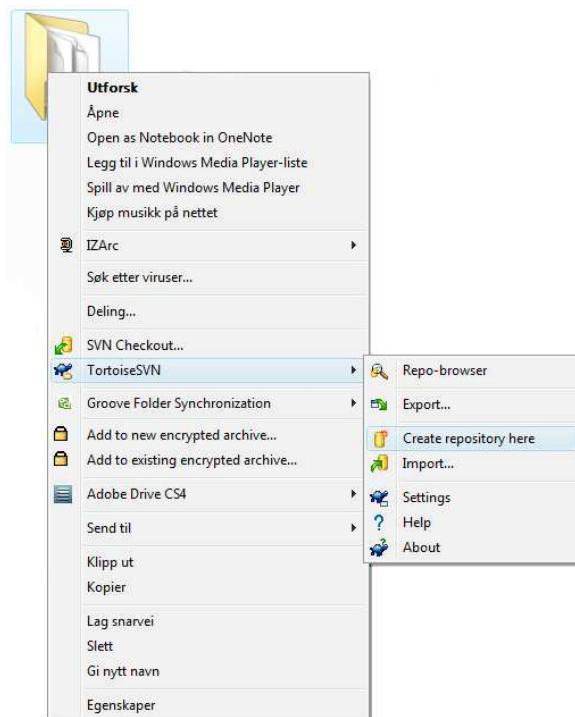


Figure 9-1: TortoiseSVN example.

9.3 Hardware Setup

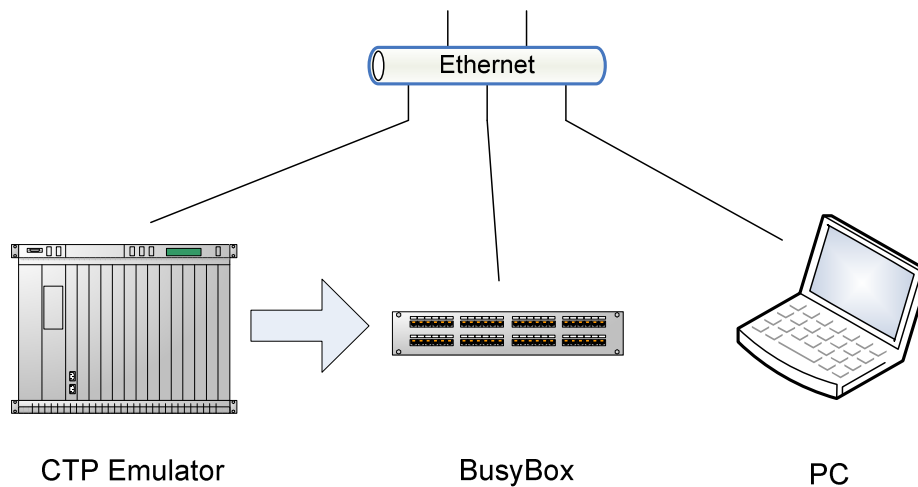


Figure 9-2: Principle hardware setup for experimenting with the BusyBox

The BusyBox is connected to the Ethernet and the CTP Emulator. The physical distance between the two devices are limited by the optical cable, and the PC can be anywhere, but for practical reasons it should be in near proximity to the others.

The optical fibre cable from the CTP Emulator goes into #1 in figure 5-1.

The Ethernet cable goes into #4 in figure 5-1.

The Ethernet connector on the DCS card is a 6 pin Milli-Grid crimp housing and the cable connections can be found here:

<http://www.kip.uni-heidelberg.de/ti/DCS-Board/current/mechanic/DCS160Ethernet01.htm>

9.4 Logging on to the DCS board

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

Interfacing with the DCS board is done either through Ethernet or UART, but Ethernet is the preferred way. The DCS board runs on a lightweight version of Linux and is access through a SSH shell. For Microsoft Windows users a SSH client can be used like the PuTTY SSH client.

Open a terminal window on a Linux computer or a SSH client on a Windows computer.

To login type:

```
ssh root@dcsxxxx
```

The **xxxx** is substituted with the number on the DCS card. Then you will be prompted for a password.

To get the password contact:

9.5 RCU Shell

The RCU Shell is a small command-line software for communicating with the firmware on the BusyBox. It is started by simply typing 'rcu-sh' at the command-line. This will bring you to the shell mode, where different types of commands can be executed, for example for reading and writing registers in the BusyBox firmware. Type "h" and press "enter" to see available commands in the RCU shell.

9.6 Programming the FPGA

The FPGA(s) will not be programmed automatically when the box is powered up. To check if the FPGAs are programmed 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/dcsbro/busy/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 one 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 the firmware will not start up.

The bit files to be programmed into the FPGA(s) must be put in the folder:

/nfs_export/dcscard on kjekspc7.

9.7 Configuring the Firmware

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

9.8 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

9.9 Resetting the BusyBox

To activate the global asynchronous reset of the Busy Box firmware (both FPGAs) 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. Sending and Receiving messages to/from the D-RORCs.

9.10 CTP Emulator

When testing or debugging the BusyBox a 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`.

```
bash-3.2$ ssh -X ltu@vme1
Scientific Linux CERN Release 3.0.8 (SL)
ltu@vme1's password:
Last login: Thu Dec 18 15:28:56 2008 from iftmikro039165.klientdrift.uib.no
```

This version of Scientific Linux CERN (SLC3) is obsolete and no longer receives automatic security updates.
See <http://cern.ch/linux/slc4/docs/migration-campaign> for more information.

```
[vme1] /home/ltu > vmecrate ltu
Use right mouse button to get help
```

Figure 9-3: Example of login.

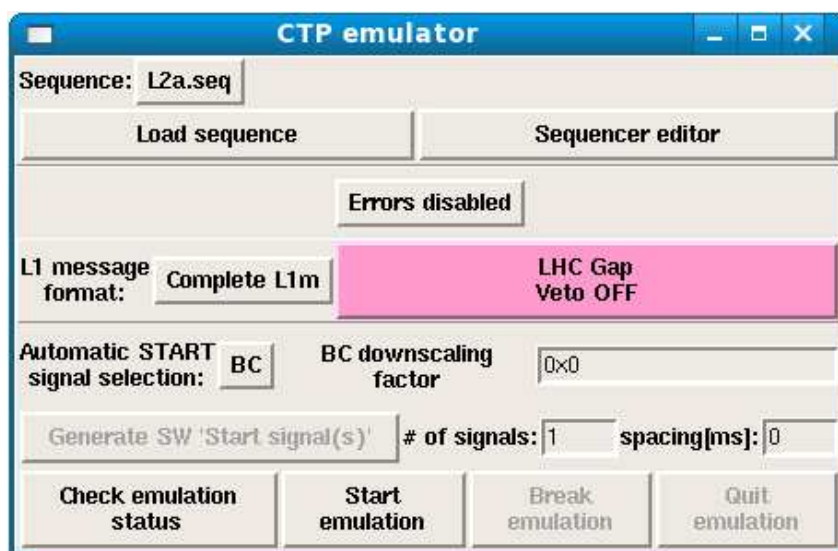
Then the VME menu is displayed.

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.

To get the password contact:



10 Register

All registers for the BusyBox are listed below.

10.1 BusyBox Register Interface

Register Name	Address	Type ⁶	Description
TX Register[15:0]	0x30001	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.

⁶ Legend: W=write, R=read, T= write trigger (not physical registers)

Busy Timer[15:0]	0x2011	R	Bit 15:0 of register that holds number of cycles 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.
Channel Register[3:0]	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 10-1: List of registers that can be accessed externally.

10.2 Trigger Receiver Module Register Interface

Register name	Address	Type ⁷	Description
Control[15:0]	0x3000	RW	[0] Serial B channel on/off Default: 1 [1] Disable_error_masking 0 [2] Enable Rol 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
Rol_Config1[15:0]	0x3004	RW	Rol-Definition. Bit 15:0
Rol_Config1[1:0]	0x3005	RW	Rol Definition. Bit 17:16
Rol_Config2[15:0]	0x3006	RW	Rol Definition. Bit 33:18
Rol_Config2[1:0]	0x3007	RW	Rol 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
L2_Latency[15:0]	0x300F	RW	[31:16] Min Latency from BC0 to L2
PrePulse_Latency[7:0]	0x3010	RW	

⁷ Legend: W=write, R=read, T= write trigger (not physical registers)

<i>Rol_Latency</i> [15:0]	0x3012	RW	[15:0] Max Latency from BC0 to Rol msg
<i>Rol_Latency</i> [15:0]	0x3013	RW	[31:16] Min Latency from BC0 to Rol msg
<i>L1_msg_latency</i> [15:0]	0x3014	RW	[15:0] Max Latency from BC0 to L1 msg
<i>L1_msg_latency</i> [15:0]	0x3015	RW	[15:0] Max Latency from BC0 to L1 msg
<i>Pre_pulse_counter</i> [15:0]	0x3016	RW	Number of decoded pre-pulses.
<i>BCID_Local</i> [11:0]	0x3018	R	Number of bunchcrossings at arrival of L1 trigger.
<i>L0_counter</i> [15:0]	0x301A	R	Number of L0 triggers
<i>L1_counter</i> [15:0]	0x301C	R	Number of L1 triggers
<i>L1_msg_counter</i> [15:0]	0x301E	R	Number of successfully decoded L1 messages
<i>L2a_counter</i> [15:0]	0x3020	R	Number of successfully decoded L2a messages
<i>L2r_counter</i> [15:0]	0x3022	R	Number of successfully decoded L2r messages
<i>Rol_counter</i> [15:0]	0x3024	R	Number of successfully decoded Rol messages
<i>Bunchcounter</i> [11:0]	0x3026	R	Debug: Number of bunchcrossings
<i>hammingErrorCnt</i> [15:0]	0x302C	R	[15:0] Number of single bit hamming errors
<i>hammingErrorCnt</i> [15:0]	0x302D	R	[31:16] Number of double bit hamming errors
<i>ErrorCnt</i> [15:0]	0x302E	R	[15:0] Number of message decoding errors
<i>ErrorCnt</i> [15:0]	0x302F	R	[31:16] Number of errors related to sequence and timeouts.
<i>Buffered_events</i> [4:0]	0x3040	R	Number of events stored in the FIFO.
<i>DAQ_Header01</i> [15:0]	0x3042	R	Latest received DAQ Header 1 [15:0]
<i>DAQ_Header01</i> [15:0]	0x3043	R	Latest received DAQ Header 1 [31:16]
<i>DAQ_Header02</i> [15:0]	0x3044	R	Latest received DAQ Header 2 [15:0]
<i>DAQ_Header02</i> [15:0]	0x3045	R	Latest received DAQ Header 2 [31:16]
<i>DAQ_Header03</i> [15:0]	0x3046	R	Latest received DAQ Header 3 [15:0]
<i>DAQ_Header03</i> [15:0]	0x3047	R	Latest received DAQ Header 3 [31:16]
<i>DAQ_Header04</i> [15:0]	0x3048	R	Latest received DAQ Header 4 [15:0]
<i>DAQ_Header04</i> [15:0]	0x3049	R	Latest received DAQ Header 4 [31:16]
<i>DAQ_Header05</i> [15:0]	0x304A	R	Latest received DAQ Header 5 [15:0]
<i>DAQ_Header05</i> [15:0]	0x304B	R	Latest received DAQ Header 5 [31:16]
<i>DAQ_Header06</i> [15:0]	0x304C	R	Latest received DAQ Header 6 [15:0]
<i>DAQ_Header06</i> [15:0]	0x304D	R	Latest received DAQ Header 6 [31:16]
<i>DAQ_Header07</i> [15:0]	0x304E	R	Latest received DAQ Header 7 [15:0]
<i>DAQ_Header07</i> [15:0]	0x304F	R	Latest received DAQ Header 7 [31:16]
<i>Event_info</i> [11:0]	0x3050	R	[0] Rol enabled [1] Region of Interest announced (=ESR) [2] Rol 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
<i>Event_error</i> [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 Rol 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] Rol message arrives outside legal timeslot [20] Rol message missing/timeout [21] Prepulse error (=0; possible future use) [22] L1 message content error [23] L2 message content error [24] Rol 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
RolMessageHeader[11:0]	0x307C	R	Debug: Latest received Rol Message
RolMessageData1[11:0]	0x307E	R	Debug: Latest received Rol Message
RolMessageData2[11:0]	0x3080	R	Debug: Latest received Rol Message
RolMessageData3[11:0]	0x3082	R	Debug: Latest received Rol 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 10-2: 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.

10.3 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
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 10-3: 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.