CARLOSrx v4 rel 2 reference manual

Samuele Antinori, Filippo Costa, Davide Falchieri,
Alessandro Gabrielli, Enzo Gandolfi,
Massimo Masetti, Samuele Zannoli

Department of Physics and INFN Bologna

June 2004

Outline

What's new in CARLOSrx v4 rel 2

CARLOSrx v4 rel 2 acts as an interface between 2 ASICs CARLOS v4 and 1 DDL in the context of the SDD readout chain. CARLOSrx v4 rel 2 has changed from the previous release both for what concerns the firmware and the decoding software.
Here follows a list of the new features in CARLOS rx v4 rel 2.

Hardware upgrades:

- The hardware used for CARLOSrx v4 rel 2 is the same as the one used for release 1.

Firmware upgrades:

- Nearly every block contained in CARLOSrx v4 rel 1 has been doubled, compatibly with the FPGA available resources. This is the reason why the available RAM resources have been assigned half to a processing channel and half to the other.
- A queue manager has been added in order to fairly allocate the available bandwidth towards the SIU to the 2 incoming data streams. In particular the available bandwidth is allocated dynamically, so to provide it to the channel requiring it most.
- The data transmission protocol has been changed in order to accomplish the merging of two processed data streams. In particular an event of CARLOSrx rel 2 corresponds to the sum of the 2 events coming from the 2 ASICs CARLOS. Furthermore before a new event is sent in output, the whole current event has to be processed and sent out.
- The busy signal towards the trigger system is obtained as the logical OR of the two busy signals of the two incoming channels.
- The JTAG programming is the same for both CARLOS and the related front-end electronics. This means that CARLOSrx receives JTAG information on the JTAG port and forwards it towards both the CARLOS chips.

Software upgrades:

- A unique version of the C++ decoding software *carlosrx* is able to decode data encoded both in rel 1 and in rel 2.

Main features

- XC2V1000 Xilinx Virtex2 FPGA;
- 40 MHz working frequency;
- 1.8 V core power supply; 2.5 V I/O pads power supply;
- standard IEEE 1149.1 JTAG implemented;
- interface towards 2 CARLOS v4 ASICs implemented;
- interface towards the SIU implemented (with flow control);
- interface towards the trigger system implemented;
- it can be directly interfaced either to 2 CARLOS chips or to 2 optical links

Known limitations

- The CARLOSrx board can be connected to the CARLOS chips in either of 2 ways:
    - direct connection;
    - with the optical links (GOL + optical fiber + TLK1501)

  If used in the second configuration, the clock and the serial back-link running from CARLOSrx to CARLOS have to be carried with wires (not with optical fibers as expected in ALICE). In fact no optical transceiver for these two signals has been foreseen on the CARLOSrx board.
- The current version of the firmware of CARLOSrx allows to interface a 1-level trigger with a simple trigger – busy scheme. This version is not able to interface the 3-levels trigger system expected for ALICE and it does not interface the TTCrx device.
- No mechanism has been foreseen in order to avoid double (dummy) events from AMBRA when working in multi-buffer mode.

General description

CARLOSrx v4 rel 2 is a Xilinx Virtex2 FPGA-based device with the main purpose of concentrating data coming from two SDD detectors on one LDC.
For each of the two incoming streams, CARLOSrx packs data coming from the related front-end electronics and CARLOS through the optical links into 32-bit words, stores them in a large data FIFO and then sends them towards the DDL system, after a transaction has been opened by the SIU and when they are assigned the output bandwidth by the queue manager. The use of a FIFO per channel allows not to lose any data even when the DDL asserts the flow control: in fact CARLOSrx asserts the back-pressure towards one of the two CARLOS (or both) and the related CARLOS will stop AMBRA, thus freezing the data acquisition process until the DDL is ready to accept data again.
CARLOSrx also drives 2 serial backlink ports towards the CARLOS chips. They are used for the following purposes:

- for sending JTAG information to PASCAL, AMBRA, CARLOS and GOL. The same JTAG information is sent on both the serial backlink ports (➔ the 2 CARLOS chips have to be programmed in the same way).
- for sending to CARLOS reset commands, trigger signals and control commands. Even if the two serial back-link ports are driven by two separate and completely independent state machines, the trigger commands are sent to both CARLOS chips at the same time.

CARLOSrx also interfaces the trigger system by receiving the *trigger* signal and asserting the *busy* signal obtained as a logical OR of the 2 busy signals received from the 2 CARLOS chips. It also asserts the *tdc1* and *tdc0* signals that will be used by the TDC chip in order to determine the latency occurred from the moment the trigger arrives to the moment it is actually sent to AMBRA.
Fig. 1 shows a schematic representation of the chain to be implemented in our Lab. This firmware implemented on the FPGA contains 7 major logic blocks:

1. *data packing1*: CARLOSrx receives the 16-bit data words coming from CARLOS 1, groups them depending on their type, packs them into 32-bit words and stores them into a FIFO, before they are sent towards the SIU.
2. *data packing0*: CARLOSrx receives the 16-bit data words coming from CARLOS 0, groups them depending on their type, packs them into 32-bit words and stores them into a FIFO, before they are sent towards the SIU.
3. *SIU interface*: this block manages the protocol interface towards the SIU. It is able to recognize the commands sent from the SIU and then to send packed data towards the SIU.
4. *trigger interface*: this block directly interfaces the trigger system by receiving the trigger input and asserting the busy signal. When both CARLOS are in RUN mode, the busy signal value reflects the value received from both CARLOS error flag words.

5. *JTAG interface to SIU*: this block receives the JTAG signals from the SIU and encapsulates them on the 2 serial back-link ports towards both CARLOS ASICs. It also implements 2 JTAG instructions: "Put CARLOS in JTAG mode" and "Put CARLOS in RUN mode". When one of these instructions is detected, a signal is sent to both the serial backlink blocks in order to send to CARLOS the corresponding command.
6. *serial backlink1*: this block drives the *serial backlink* signal from CARLOSrx to CARLOS 1. It is used to send JTAG commands, reset commands, trigger signals, prepulse and testpulse signals, the commands "Enter JTAG mode" and "Enter RUN mode".
7. *serial backlink0*: this block drives the *serial backlink* signal from CARLOSrx to CARLOS 0. It is used to send JTAG commands, reset commands, trigger signals, prepulse and testpulse signals, the commands "Enter JTAG mode" and "Enter RUN mode".

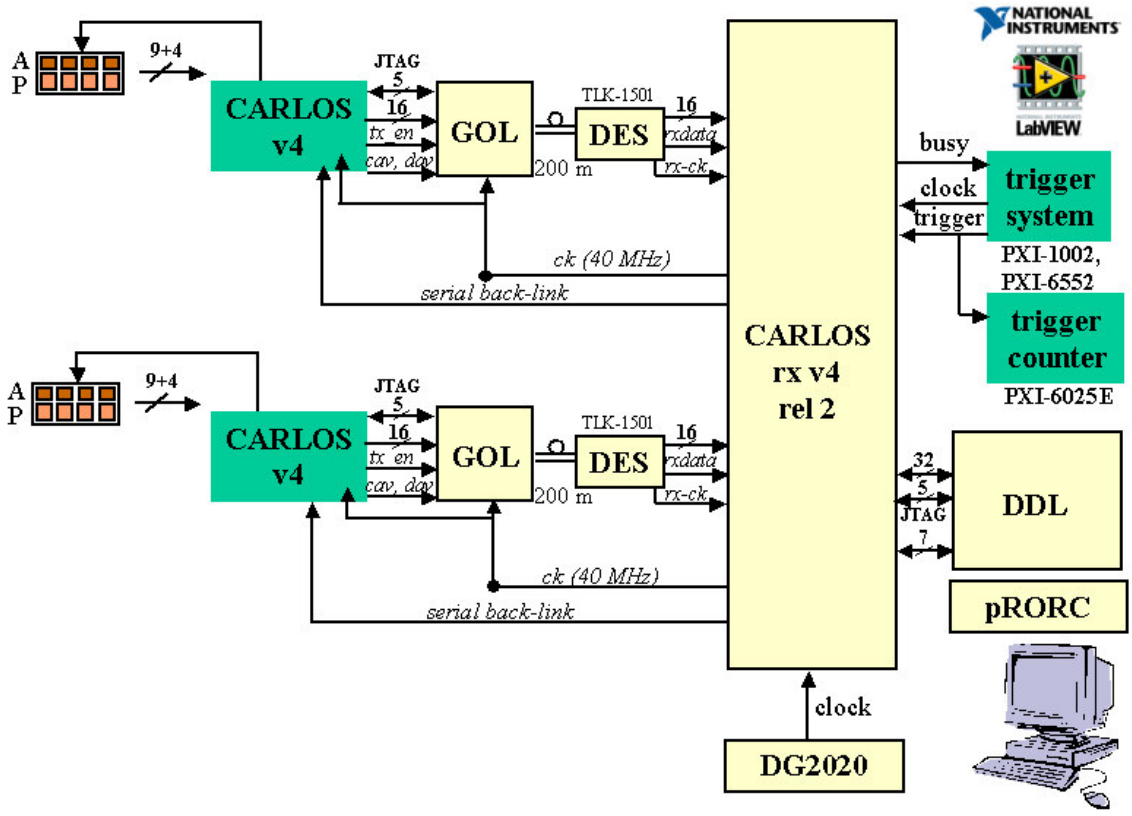


Fig. 1: Schematic representation of the SDD readout chain to be implemented in Lab.

Interface to CARLOS 0 and CARLOS 1

CARLOSrx receives data coming from CARLOS 0 and CARLOS 1. These are the interface signals for each CARLOS:

- *output_data#* (16 bits): this is the 16-bit bus containing the data coming from CARLOS#;
- *tx_en#*: it is a strobe signal, active high. When active the *output_data* bus contains a valid value, whichever its type (header, footer, data from ch1, data from ch0, error flag word, JTAG word).
- *cav#*: it is a strobe signal, active high. When active, the *output_data* bus contains a valid control word, that is either a JTAG word or an error flag word.
- *serial backlink#*: it is a serial link from CARLOSrx to CARLOS. It is used to send 8-bit commands to CARLOS, such as reset signals, trigger signals, JTAG information and commands for putting CARLOS in JTAG mode or in RUN mode.

The *serial backlink1* and *serial backlink0* blocks on CARLOSrx are used to drive the *serial backlink1* and *serial backlink0* signals. After the reset is activated, each block sends a number of IDLE codes for link synchronization, then it sends the commands for resetting all the front-end chips (PASCAL, AMBRA, CARLOS and GOL). Then it continues sending IDLE codes until it has to send one of the following commands (from highest priority to lowest: if two commands have to be sent at the same time, the highest priority is sent first)):

- enter JTAG mode;
- enter RUN mode;
- trigger signal;
- JTAG information;
- L1reject;
- L2reject;
- prepulse25;
- testpulse;
- prepulse50, 75, 100, 125, 150, 175, 200
- stop acquisition;
- restart acquisition.

Since these commands are to be sent on a 8-bit serial link, there is a variable jitter of a few clock cycles from the moment a signal is received to the moment it is actually sent to CARLOS over the serial back-link.

This is the list of actions occurring when a signal is about to be transferred over the serial back-link (refer to Fig. 2):

- a 3-bit counter, *ibit*, continuously runs from 0 to 7: a complete 8-bit command is sent in output over the serial back-link in 8 clock cycles, from *ibit* = 0 to *ibit* = 7. In each of these 8-period slots a command is sent in output.
- a trigger is received: two clock cycles are necessary to get a synchronous trigger signal (*trigger_long*) that is used by CARLOSrx internal blocks.
- the internal state machine driving the serial back-link output continuously checks for the *trigger_long* signal to be high. When it does, it changes state from 5 to 6.
- When state is 6, the 8-bit register *vector1* is updated with the trigger command value AC.
- Then when *ibit* = 7, the 8-bit register *vector* is updated with *vector1* value and then the command is transmitted.

In this case the latency from the trigger arrival to the moment in which the related commands starts to be sent over the serial back-link is about 5 clock cycles.
Fig. 3 shows that this latency can be higher (9 clock cycles) depending on the relative timing between the trigger arrival time and the *ibit* counter value. The latency might also be larger if the trigger occurs while an other command is being transmitted over the serial link.
The *tdc* signal is activated for one clock cycle when a trigger command is being sent over the serial back-link and when *ibit* = 2.
A proposal of upgrade of the CARLOSrx firmware suggests to store the latency of the trigger signal (as well as the testpulse and prepulse) in a FIFO and send them towards the DDL on the DDL header.

N.B. Since the two *serial backlink* blocks receive the reset signal at the same time, the *ibit1* and *ibit0* counter are identical. So far the trigger commands are sent to the 2 CARLOS chips at the same time, also considering that the trigger command is the highest priority one. Thus no jitter problem has to be considered between the two CARLOS chips for what concerns the trigger arrival time. The testpulse and prepulse commands should not suffer this kind of problem as well: it is sufficient to be sure to send these commands when no back-pressure is being activated or de-activated.

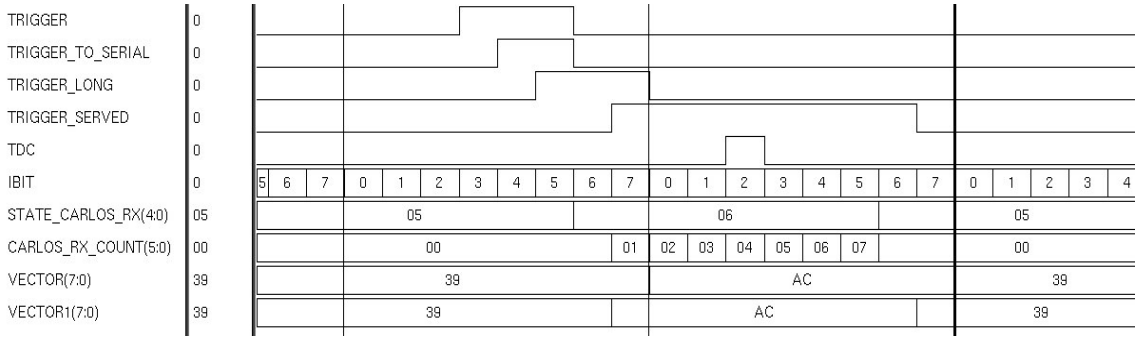


Fig. 2: In each 8-cycle timing slot (between vertical bars) a command is sent over the serial back-link. In this case the trigger – command latency is 5 clock cycles.

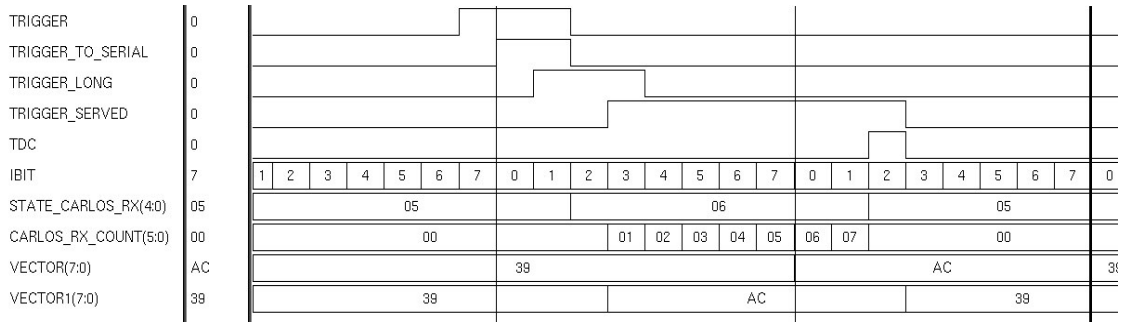


Fig. 3: In this case the trigger command latency is 9 clock cycles.

Interface to the SIU

A list follows of the signals involved in the CARLOSrx- SIU interface (see Fig. 4):

- *fidir*: it is an input to CARLOSrx. It asserts the direction of the data flow between CARLOSrx and the SIU: when 0 the direction is from the SIU to CARLOSrx, when 1 the direction is from CARLOSrx to the SIU.
- *fiben_n*: it is an input to CARLOSrx, active low. It enables the communication on the bidirectional bus between CARLOSrx and the SIU. When 0 the communication is enabled, when 1 the communication is disabled.
- *filf_n*: it is an input to CARLOSrx, active low, "lf" stands for link full. When the SIU is no longer able to accept data coming from CARLOSrx it asserts this signal. When this happens CARLOSrx sends an other valid data word, then stops transmitting waiting for the *filf_n* signal to switch back to 1. This is the signal used by the SIU to implement the back-pressure on the data flow running from the front-end to the data acquisition system.
- *foclk*: it is a free running clock generated on CARLOSrx and driving the CARLOSrx-SIU interface. It is a 20 MHz clock generated by dividing the system clock frequency by two. Interface signals coming from the SIU change state on the falling edge of *foclk*.
- *fbten_n*: it is a bidirectional signal, active low, it can be driven by CARLOSrx or by the SIU, "ten" stands for transfer enable. When CARLOSrx is assigned to drive the bidirectional buses (when *fidir* is 1 and *fiben_n* is 0) *fbten_n* value is asserted from CARLOS: it turns to its active state when CARLOSrx is transmitting valid data to the SIU, otherwise it is inactive. When the SIU is assigned to drive the bidirectional buses (when *fidir* is 0 and *fiben_n* is 0) *fbten_n* value is asserted from the SIU: it turns to its active state when the SIU is transmitting valid commands to CARLOSrx, otherwise it is inactive.
- *fbctrl_n*: it is a bidirectional signal, active low, it can be driven by CARLOSrx or by the SIU, "ctrl" stands for control. When CARLOSrx is assigned to drive the bidirectional buses (when *fidir* is 1 and *fiben_n* is 0) *fbctrl_n* value is asserted from CARLOSrx: it turns to its active state when CARLOSrx is transmitting a Front End Status Word to the SIU, otherwise, when in the inactive state, CARLOS is sending normal data to the SIU. When the SIU is assigned to drive bidirectional buses (when *fidir* is 0 and *fiben_n* is 0) *fbctrl_n* value is asserted from the SIU: it turns to its active state when sending command words to CARLOSrx, to its inactive state when sending data words. The second option has not been implemented on CARLOSrx since we decided that CARLOSrx needs only commands and not data from the SIU.

- *fobsy_n*: it is an input signal to the SIU, active low, "bsy" stands for busy. CARLOS should put this signal active when not able to accept data coming from the SIU. Since CARLOSrx has not to receive data from the SIU, this signal has been fixed at 1, meaning that CARLOSrx will never be in a busy state. In fact it always has to accept command words coming from the SIU.
- *fbd*: it is a bidirectional 32-bit bus on which data or command words are exchanged between CARLOSrx and the SIU.

This is the way the communication protocol works:
The SIU acts as the master and CARLOSrx acts as the slave, that is the SIU sends commands to CARLOSrx and CARLOSrx sends data and front end status words to the SIU. At first the link CARLOSrx - SIU has to be initialized and the SIU acts as the master of the bidirectional buses. So CARLOSrx waits for the bidirectional buses to be driven from the SIU (*fidir* is 0 and *fiben_n* is 0) and waits for a valid (*fbten_n* = 0) command (*fbctrl_n* = 0) named Ready to Receive (RDYRX). This command is always used in order for a new event transaction to begin. The RDYRX command contains a transaction identifier (bits 11 to 8) and the string "00010100" as the less significant bits. As the command is accepted and recognized CARLOSrx waits for the *fidir* signal to change value in order to take possession of the bidirectional buses, then, if the *filf_n* is not active, it is able to send valid data on the *fbd* bus if it has any.
Each data packet begins with the DDL header. At the end of a data packet CARLOSrx puts in output the Front End Status Word, a word that confirms that no errors occurred and that the whole event has been successfully transferred to the SIU. The Front End Status Word contains the Transaction Id code received upon the opening of the transaction (bits 11 to 8) and the 8-bit FESTW code "01100100". After this happens CARLOSrx begins waiting for some action of the SIU to be taken: it means that the SIU can decide to take back its control on the bidirectional buses and close the data link towards the data acquisition system, or the SIU can leave the bidirectional buses control to CARLOSrx for an other data event to be sent. So far CARLOSrx begins waiting 16 *foclk* periods: if nothing happens CARLOSrx is able to begin sending data again without the need to receive some other commands from the SIU; if the SIU takes back the possession of the bidirectional buses CARLOSrx closes the link towards the SIU and keeps waiting for an other RDYRX command asserted from the SIU itself.

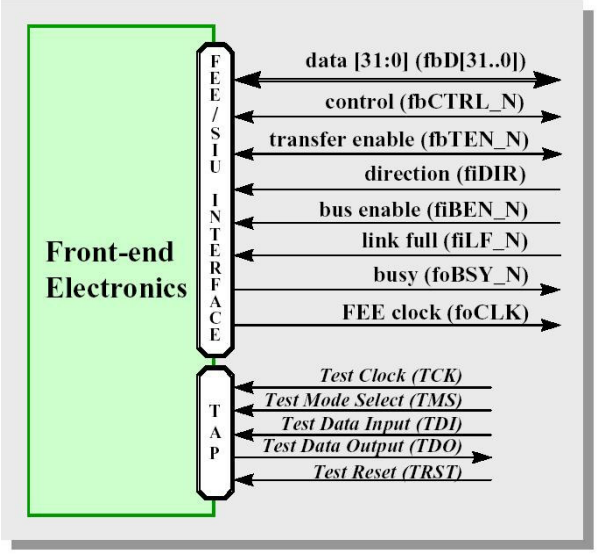


Fig. 4: CARLOSrx – SIU interface

Interface to the trigger system

CARLOSrx interfaces the trigger system with the 2 following signals:

- *trigger*: it is the trigger signal received from the trigger system, active high. It is completely asynchronous with respect to the incoming clock and its width is 80 ns.
- *busy*: it is the busy signal from CARLOSrx to the trigger system. When the CARLOS chips are in JTAG mode, CARLOSrx asserts the *busy* signal high. When the CARLOS chips are in RUN mode the busy value is obtained by performing the logical OR of the ones received from the CARLOS chips in the error flag words (one every 64 clock cycles).

JTAG instruction set

CARLOSrx interfaces the JTAG signals from the SIU and encapsulates them on the serial back-link towards CARLOS as it is. The JTAG *tck* frequency has to be at most 5 MHz (suggested value = 5 MHz). Higher *tck* frequencies will lead to errors in JTAG programming and reading back register values.
Beside that, CARLOSrx internal JTAG unit monitors the input JTAG port looking for the JTAG instructions reported in Table 1.

JTAG instruction	JTAG IR value	Length of scan register involved

Put CARLOS in JTAG mode	10001	5
Put CARLOS in RUN mode	10010	5

Table 1: List of CARLOSrx JTAG instructions

After decoding the instruction "Put CARLOS in JTAG mode", the command "Enter JTAG mode" is sent to both CARLOS through the serial backlink ports.
After decoding the instruction "Put CARLOS in RUN mode", the command "Enter RUN mode" is sent to both CARLOS through the serial backlink ports.

Back-pressure

CARLOSrx v4 rel 2 makes use of the back-pressure as soon as it needs to, that is as soon as its internal FIFOs are going to get full. CARLOSrx internal FIFOs contain a 10k 32-bit word dual-clock RAM for each channel. Since the allowed number of words is a power of 2, each FIFO is obtained by putting in series 2 FIFOs:

- *small fifo*: one 2k 32-bit words FIFO;
- *large fifo*: one 8k 32-bit words FIFO.

As soon as a word enters a FIFO, it is first written in the *small fifo*. Then an automatic process scans the *small fifo* and, as soon as the *small fifo* is no longer empty, it pops the *small fifo* and pushes the word in the *large fifo*. Then the *large fifo* is popped when the SIU is ready to accept data (the SIU has opened a transaction and the flow control has not been activated) and when the queue manager assigns the necessary bandwidth.
The back-pressure is activated when the *small fifo* has only 1k 32-bit free locations available (1k on 2k are used cells), see Fig. 5. Then the back-pressure is de-activated when the *small fifo* is completely empty.
Back-pressure is needed when the input data rate is larger than the output data rate, that is DDL input data rate. Backpressure is very important especially:

- when data streams coming from 2 CARLOS chips have to be concentrated on one LDC;
- when transmitting large events (anode length = 256);
- when no compression is operated on CARLOS;
- when the trigger rate is high (= when AMBRA works in multi-buffer mode).

So far the back-pressure can be activated very often during an acquisition run when these conditions apply. Of course it is sufficient to use CARLOS as a compressor chip in order to avoid using the back-pressure so often, thus decreasing data processing and transmission delay.
No mechanism has been foreseen in order to avoid double (dummy) events from AMBRA when working in multi-buffer mode.

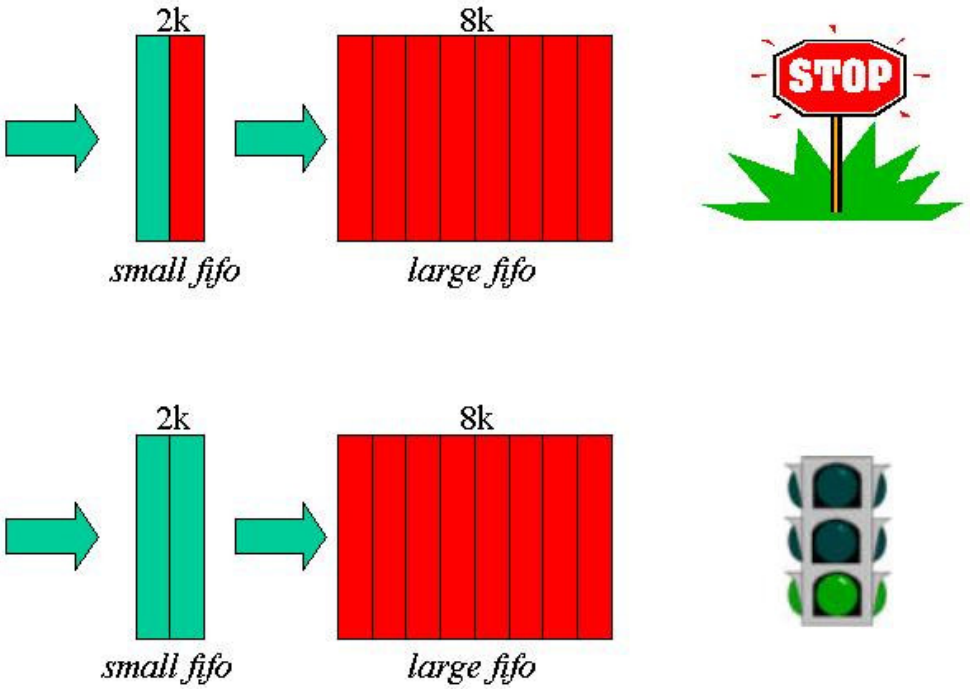


Fig. 5: Back-pressure is activated when the small fifo is half full and de-activated when the small fifo is completely empty.

CARLOSrx v4 rel 2 pin function

Terminal name	Type	Description
carlos4_0_output(15-0)	I	Input data bus coming from CARLOS 0
tx_en0	I	Input signal coming from CARLOS 0: when 1 it means that CARLOS is sending a valid word
cav0	I	Input signal coming from CARLOS 0: when 1 it means that CARLOS is sending a valid error flag word or JTAG word
dav0	I	Input signal coming from CARLOS 0: when 1 it means that CARLOS is sending a valid data
carlos4_1_output(15-0)	I	Input data bus coming from CARLOS 1
tx_en1	I	Input signal coming from CARLOS 1: when 1 it means that CARLOS is sending a valid word
cav1	I	Input signal coming from CARLOS 1: when 1 it means that CARLOS is sending a valid error flag word or JTAG word
dav1	I	Input signal coming from CARLOS 1: when 1 it means that CARLOS is sending a valid data
serial backlink0	O	Output signal towards CARLOS 0: it carries JTAG information and commands (reset, trigger, backpressure, …)
reset_carlos0	O	Output signal resetting CARLOS 0 (active low reset)
gol_ready0	O	Output signal towards CARLOS 0 (when the GOL is not used): its value is fixed to 1
serial backlink1	O	Output signal towards CARLOS 1: it carries JTAG information and commands (reset, trigger, backpressure, …)
reset_carlos1	O	Output signal resetting CARLOS 1 (active low reset)
gol_ready1	O	Output signal towards CARLOS 1 (when the GOL is not used): its value is fixed to 1
ck	I	Input clock
reset_n	I	Active low reset
fidir	I	From the SIU: it decides the bidirectional port direction
fiben_n	I	From the SIU: it decides if the bus is enabled or not (active low)
filf_n	I	From the SIU: it decides if the link is full or not (active low)
foclk	O	To the SIU: 20 MHz free running clock
fbten_n	INOUT	To and from the SIU: when active (low) the data *fbd* is

		valid
fbctrl_n	INOUT	To and from the SIU: when active (low) *fbd* contains the FESTW
fobsy_n	O	To the SIU: when active (low) CARLOSrx is not ready to accept data from the SIU
fbd	INOUT	32-bit data bus between CARLOSrx and SIU
tdi_from_siu	I	JTAG *tdi* from SIU
tck_from_siu	I	JTAG *tck* from SIU
tms_from_siu	I	JTAG *tms* from SIU
trst_from_siu	I	JTAG *trst* from SIU
tdo_to_siu	O	JTAG *tdo* to SIU
trigger	I	Input trigger, active high, 80 ns wide
busy	O	Output busy
L1reject	I	Input L1reject, active high, 80 ns wide
testpulse	I	Input L1reject, active high, 80 ns wide
tdc1	O	Output signal, active for one clock cycle when the trigger command is being sent over the *serial backlink1*
tdc0	O	Output signal, active for one clock cycle when the trigger command is being sent over the *serial backlink0*

CARLOSrx v4 rel 2 operation

This section contains an explanation of the sequence of actions needed to program and run CARLOSrx operationally.

Using CARLOSrx

CARLOSrx utilization should include the following sequence of actions:

1) power supply to the front-end electronics, to the CARLOS chips, CARLOSrx and the DDL is turned on. CARLOSrx receives a signal reset (active low) either from an external RC network or from the outside on the *reset_n* pin. After being reset, CARLOSrx begins sending reset commands to the front-end chips and to the CARLOS chips, one after the other using the 2 serial backlink ports. Busy = 1. See Timing 1.
2) Using the JTAG port the SIU sends to CARLOSrx the command "Put CARLOS in JTAG mode". As a consequence CARLOSrx sends to both CARLOS the command "Enter JTAG mode" using the serial backlink ports. Busy = 1.
3) Using the JTAG port (*tck* = 5 MHz) the SIU sends to CARLOSrx commands and data for addressing, programming and reading back the selected device (the chosen PASCAL, AMBRA or CARLOS). CARLOSrx encapsulates this information over the serial back-link ports towards CARLOS. After each chip has been programmed, the JTAG connection is closed by asserting the *trst* signal. After this step is over all the selected front-end chips have been programmed. At this point all the JTAG information read from the front-end chips and from CARLOS are stored in the internal FIFOs of CARLOSrx. Busy = 1.
4) Using the JTAG port the SIU sends to CARLOSrx the command "Put CARLOS in RUN mode". As a consequence CARLOSrx sends to both CARLOS the command "Enter RUN mode" using the serial backlink ports. After this step, CARLOSrx begins waiting for the error flag words coming from the 2 CARLOS chips one every 64 clock cycles containing the busy value. So far when CARLOS is in RUN mode, the busy value asserted towards the trigger system is obtained by putting in OR the ones received by CARLOS. Should CARLOS be brought back in JTAG mode, then the busy signal would be fixed to 1 again by CARLOSrx, meaning that no trigger signal can be accepted.
5) CARLOSrx keeps waiting until the SIU opens a transaction by sending the RDYRX (Ready to Receive) command to CARLOSrx on the 32-bit bidirectional bus *fbd*. Then CARLOSrx takes possession of the bidirectional bus until the transaction is closed. Busy = 1. See Timing 2.

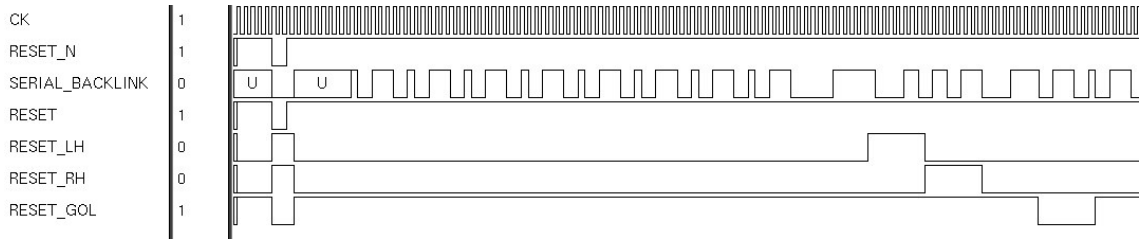
6) After the transaction is opened, CARLOSrx sends a DDL header (8 32-bit words whose structure is explained later in this paragraph), then it begins emptying its internal FIFOs by sending the JTAG words towards the SIU. First the CARLOS 1 identifier followed by 128 JTAG words coming from CARLOS 1, then the CARLOS 0 identifier followed by 128 JTAG words coming from CARLOS 0 and so on until the FIFOs are empty again. See Timing 3.
7) After CARLOSrx receives a trigger, it sends the related command to the CARLOS chips using the serial backlink ports and it asserts the busy output value. Then CARLOSrx begins waiting for the data packets coming from the CARLOS chips.
8) Valid words coming from CARLOS 1 are grouped into 32-bit words depending on their meaning:
    - header words;
    - footer words;
    - data from channel 1 (right hybrid);
    - data from channel 0 (left hybrid);
    - error flag words;
    - JTAG words.

   The same holds for CARLOS 0.
   After coding, 32-bit words are stored into a dual clock FIFO containing 10K 32-bit words per channel. The FIFOs allow to implement the flow control between CARLOSrx and the SIU. The choice of a dual clock FIFO is due to the fact that data are written into the FIFOs with a 40 MHz clock, while they are read with an internally-generated 20 MHz clock (*foclk*). In fact 32-bit data are sent to the SIU with a 20 MHz clock (= 640 Mbit/s) since the total bandwidth of the DDL is 800 Mbit/s. See Timing 4.
   Each data packet begins with the DDL header (8 32-bit words) containing information on the orbit number and on errors occurred during the transmission. Then data coming from the 2 channels are transmitted in the following way:
   - CARLOS 1 identifier;
   - 128 words coming from CARLOS 1 (if the FIFO gets empty before sending 128 words, the bandwidth is allocated to CARLOS 0);
   - CARLOS 0 identifier,
   - 128 words coming from CARLOS 0 (if the FIFO gets empty before sending 128 words, the bandwidth is allocated to CARLOS 1);
   - then after the current event coming from CARLOS 1 and CARLOS 0 have been sent in output, the FESTW follows. In the following 16 *foclk* cycles the SIU might send the EOBTR (End Of Block Transfer) command. Otherwise CARLOSrx begins sending data again. See Timing 5.

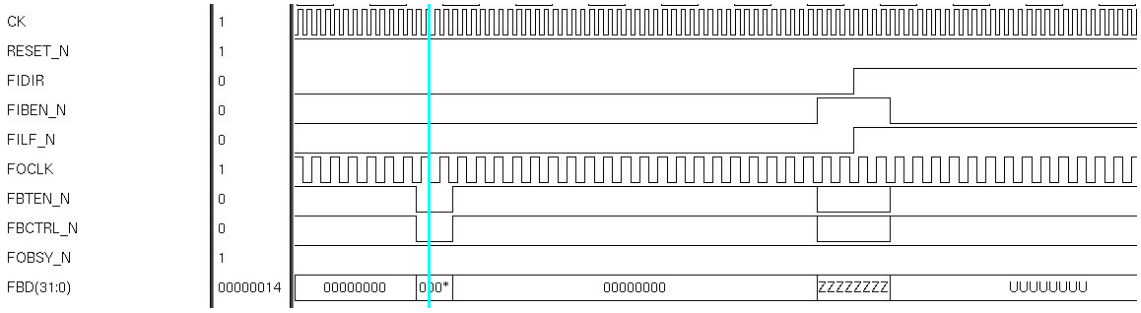
- if CARLOS 1 is still sending data to CARLOSrx, while CARLOS 0 has already completed the corresponding event (same event number), CARLOSrx is ready to accept the new event coming from CARLOS 0, storing data in the related FIFO, but without popping it.

See Timing 4, 5 and 6.

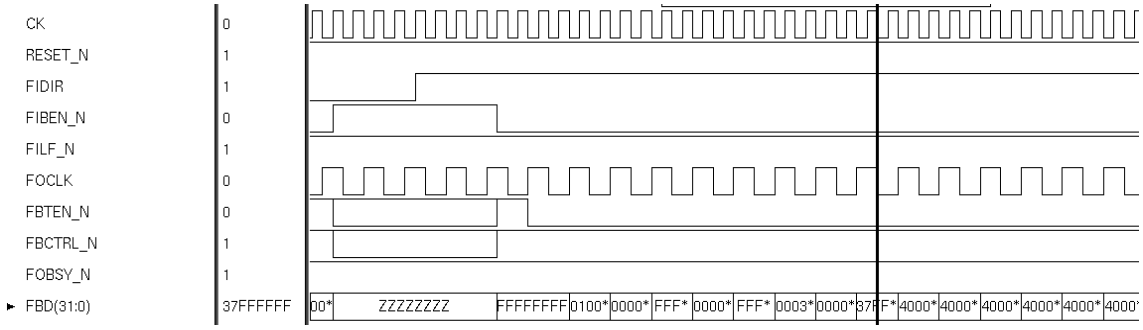
9) If any errors occurred during the transmission of an event, after the event has been completely transmitted a dummy event follows with the error bits asserted in the DDL header and a FESTW (see Timing 7).
10) CARLOSrx implements the flow control. This means that each time the SIU board can no longer accept input data from CARLOSrx and it asserts the *filf_n* signal, CARLOSrx stops sending data, while it continues to receive data from CARLOS. After the SIU board is ready to accept data again, the *filf_n* signal is de-asserted and CARLOSrx begins to send data to the SIU again. See Timing 8 and 9.
11) If an error occurs in the communication between one AMBRA -CARLOS pair (for instance if CARLOS 0 does not receive the *data_end* signal in the expected time slot) and if StopIfError is 1, after closing the data packet, CARLOS 0 asserts both the *data_stop* signals, thus stopping the acquisition of further events from AMBRA. In this case, after sending the dummy event, CARLOSrx also stops since it receives data coming from CARLOS 1, but it does not receive data coming from CARLOS 0. In this case using the JTAG port the SIU has to send the command "Put CARLOS in JTAG mode" and, if necessary, reprogram some of the front-end chips. Then the SIU will send the command "Put CARLOS in RUN mode", so that CARLOS will de-assert the data-stop signals and the data acquisition will begin again.
12) If an error occurs in the communication between one AMBRA – CARLOS pair and if StopIfError is 0, a dummy event is sent in output, but the transmission goes on.



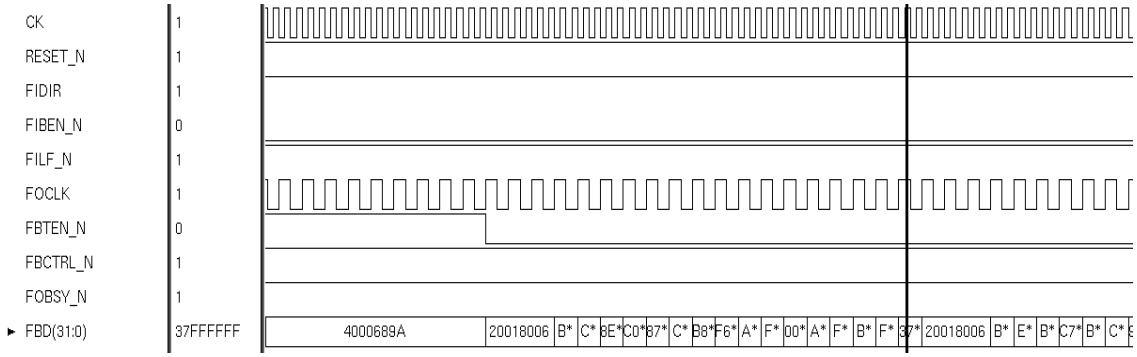
Timing 1: CARLOSrx receives the reset signal (*reset_n*). Then it sends IDLE commands on the *serial backlink*, followed by the reset commands for the front-end chips.



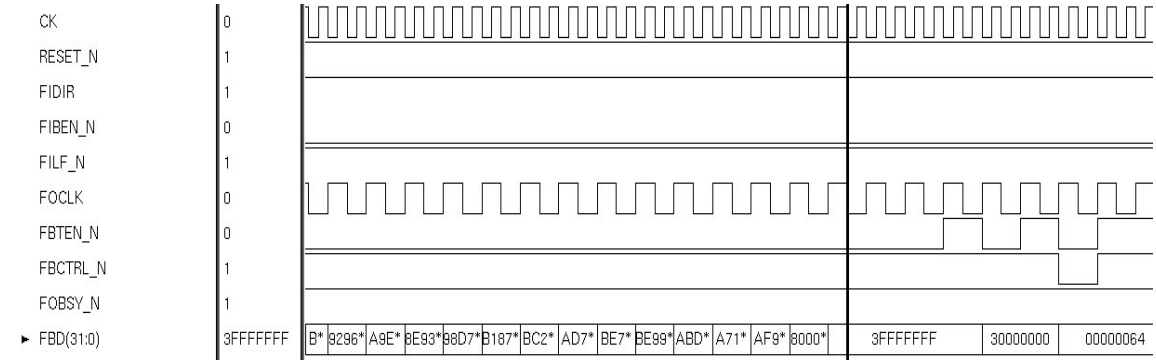
Timing 2: CARLOSrx receives the RDYRX command on the *fbd* bus, then it waits until the SIU changes the *fidir* value. Then CARLOSrx takes possession of the bidirectional bus.



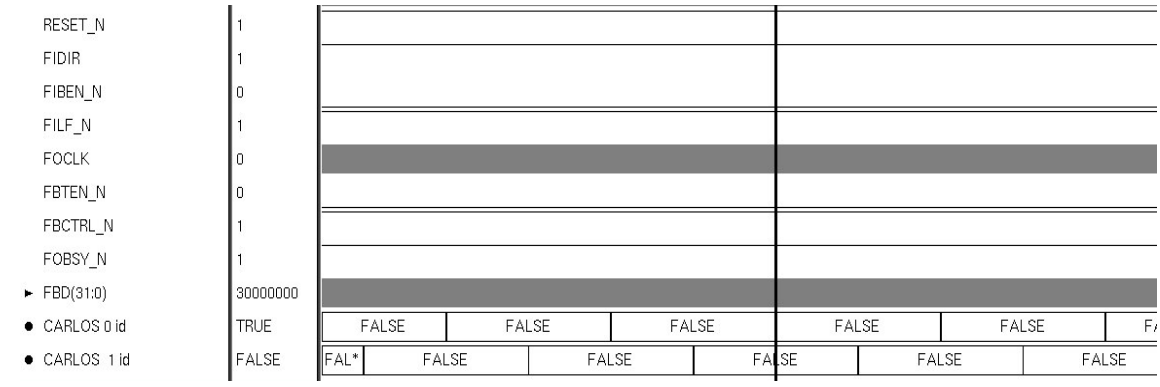
Timing 3: After the opening of a transaction, CARLOSrx sends in output the 8 32-bit words of the DDL header, followed by the JTAG words stored in the FIFOs starting from CARLOS 1.



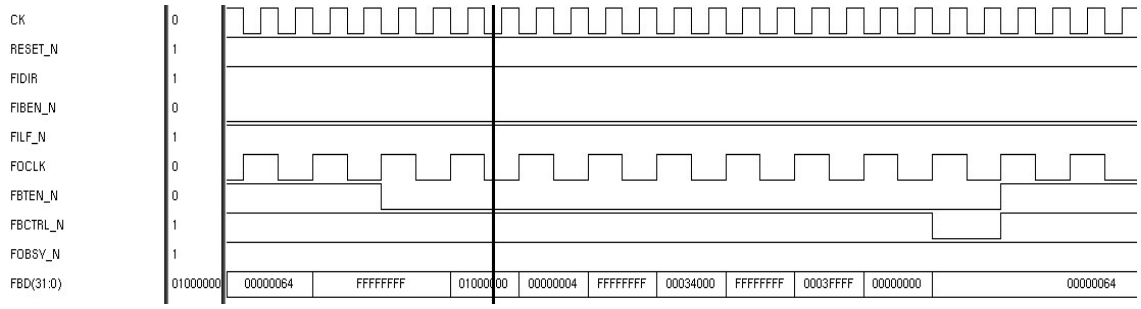
Timing 4: After the JTAG words, CARLOSrx begins sending in output data coming from CARLOS 0, followed by data coming from CARLOS 1, trying to fairly assign bandwidth to both channels.



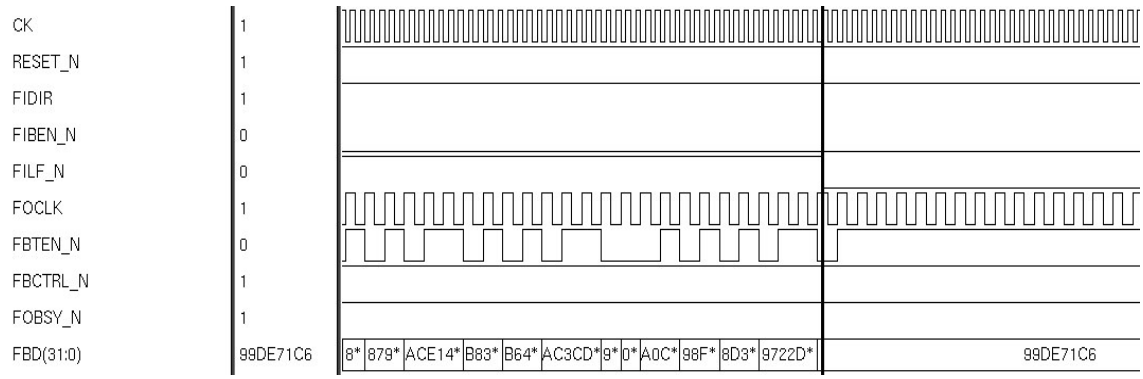
Timing 5: CARLOSrx closes a data packet with 3 footer words, then it sends the FESTW to the SIU.



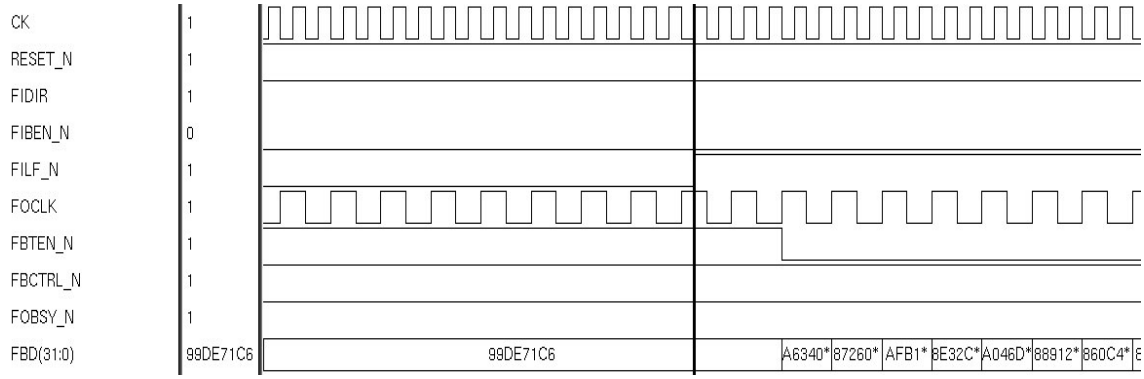
Timing 6: CARLOSrx sends 128 valid words from CARLOS 0, then from CARLOS 1 and so on, until a FIFO does gets empty or one of the CARLOS chips completes its event.



Timing 7: Dummy event after a faulty event: the orbit number is the same as the previous event and the status and error bits are asserted.



Timing 8: Flow control: the SIU asserts the *filf_n* signal. CARLOSrx stops sending valid data on *fbd* (*fbten_n* = 1). Data coming from CARLOS are written into the internal FIFO.



Timing 9: Flow control: when *filf_n* is switched back to 1, then CARLOSrx begins emptying the internal FIFO.

Data processing

CARLOSrx contains 2 parallel pipeline blocks for reading, processing and storing data coming from CARLOS 1 and CARLOS 0. For each processing channel :

- CARLOSrx fetches the 16-bit data bus coming from CARLOS# when it contains a valid value (*tx_en#* = 1).
- then, depending on the type of data word, it groups together words of the same type into 32 bit words. Error flag words coming from CARLOS are stored only during the timing frame of a data event: error flag words arriving outside these frames are discarded. In the same way error flag words are discarded when CARLOSrx is asserting the back-pressure towards CARLOS.
- it also changes the MSBs of these words in order to be able to recognize them after the packing process.

The 32-bit words resulting from the packing process have the following format:

bit 31	bit 30	bit 29	bit 28	bit 27 – 0	word type
0	0	1	0	header[13-0], header[13-0]	header
0	0	1	1	footer[13-0], footer[13-0]	footer
0	1	0 & JTAG word [14-0]			JTAG word
0	0	0	0 & errflag [13-0]		error flag word
1	0	II output_data[14-0] & I output_data[14-0]			data from ch0
1	1	II output_data[14-0] & I output_data[14-0]			data from ch1

Table 2: 32-bit format from CARLOSrx to the SIU

In the 32-bit word, the first data received from CARLOS is packed as LSBs, while each second data is packed as MSBs.
When a 32-bit word is complete, it is written into the related 10K 32-bit words long FIFO (see Fig. 6). Then a queue manager decides when to pop data from the FIFOs synchronously with *foclk*, that is half the system frequency (20 MHz).
It may also happen that one of the data channels from CARLOS 1 or 0 sends an odd number of data, so that the related 32-bit register on CARLOS remains incomplete. In this case its value is put in output with the II *output_data[14-0]* containing all zeros before the footer words are sent in output.

Data transmission protocol

The main change in the data transmission protocol between the current release and release 1 is in the fact that data coming from two different streams have to be merged. An other issue is to make as few modifications as possible to the data format. For this reason we decided to allocate the available bandwidth in time frames with the following features (see Fig. 7):

- each time frame begins with the CARLOS identifier;
- data contained in each time frame have the same format than in CARLOSrx rel 1;
- each time frame SHOULD contain 128 words with the following exceptions:
    - **LESS THAN 128**: if the corresponding FIFO gets empty, the time frame is closed and the bandwidth is assigned to the other CARLOS, if it has data to send;
    - **MORE THAN 128**: if the other CARLOS chip has already completed its event, the current CARLOS can keep the bandwidth until it has completed the event too.
- the time frame ends when a new CARLOS identifier is found.

After the transaction has been opened, CARLOSrx begins sending JTAG data towards the SIU using timing frames. Then data coming from physical events are sent out in the following way:

- first event CARLOS 1 – first event CARLOS 0
- second event CARLOS 1 – second event CARLOS 0
- …

This means that a CARLOSrx event corresponds to the sum of the related events on CARLOS 1 and CARLOS 0. This also means that CARLOS 1 does not start sending a new event, until CARLOS 0 has completely sent out the current event and viceversa.

DDL header

The DDL header is required by the DATE v4 SW and later releases. It usually contains information regarding block length, L1 message, event ID, orbit number, participating sub-detectors, status and error bits, mini-event ID, ROI (Region of Interest). Since CARLOSrx does not receive most of this information, most of these fields are not used and, for this reason, they are filled with 0 or 1 as required by DATE v4.
The only meaningful information fields are (see Fig. 8):

- **format version**: DATE v4 requires this field to be 00000001;
- **orbit number**: this is an incremental number associated to every event;
- **status & error bits**: this field is usually filled with 0s when transmitting error-free events. If any error is detected during the transmission of an event, it is followed by a dummy event (see Fig. 8) containing the same orbit number as the previous event and the following bits asserted in the status & error bits field:
    - bit 5: ask DATE to stop the run;
    - bit 4: no central trigger present;
    - bit 2: data parity error.

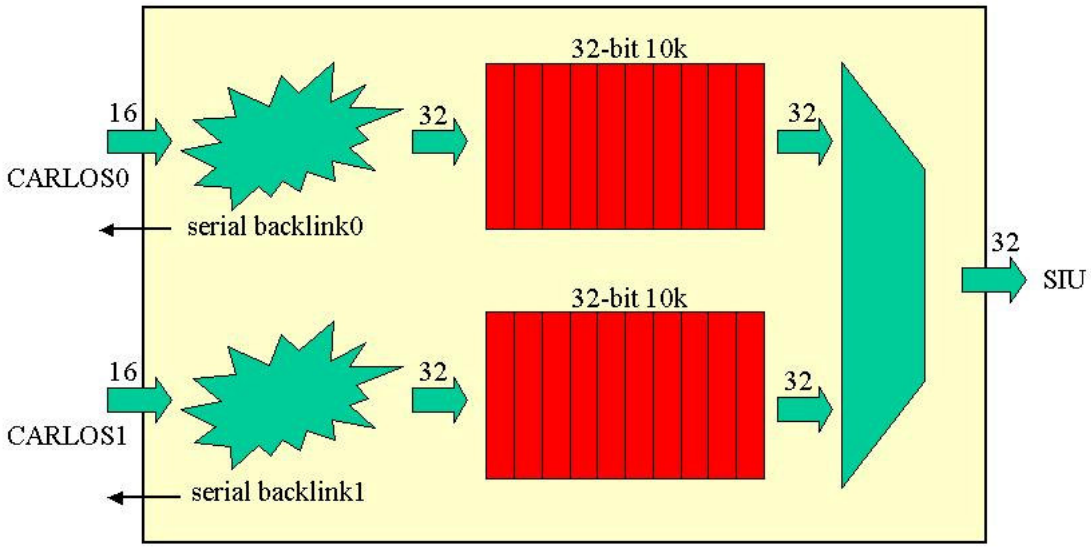


Fig. 6: Data packing and storing on CARLOSrx

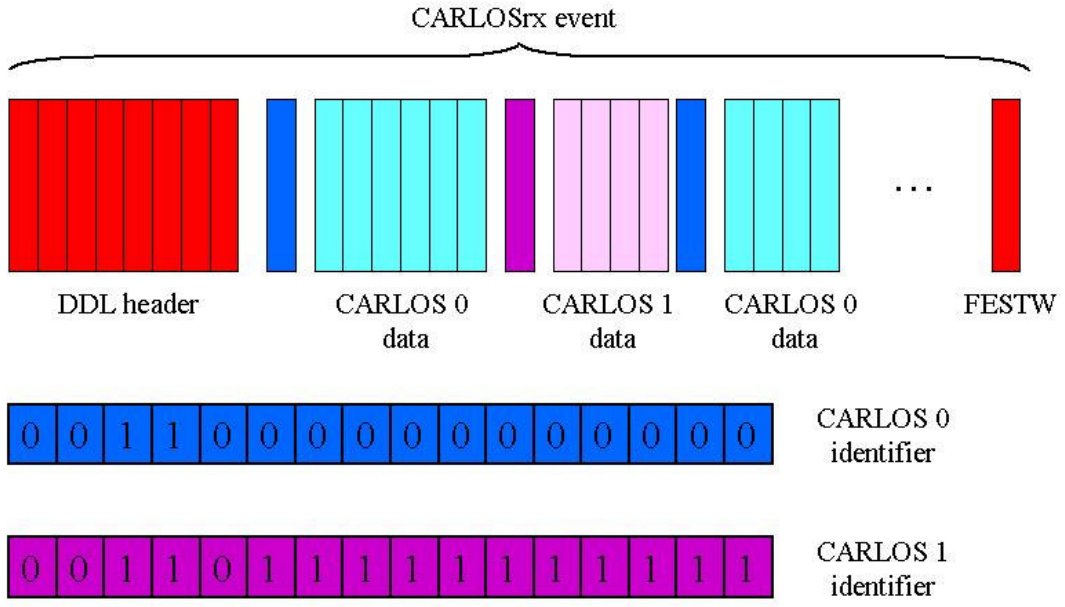


Fig. 7: CARLOSrx output data protocol

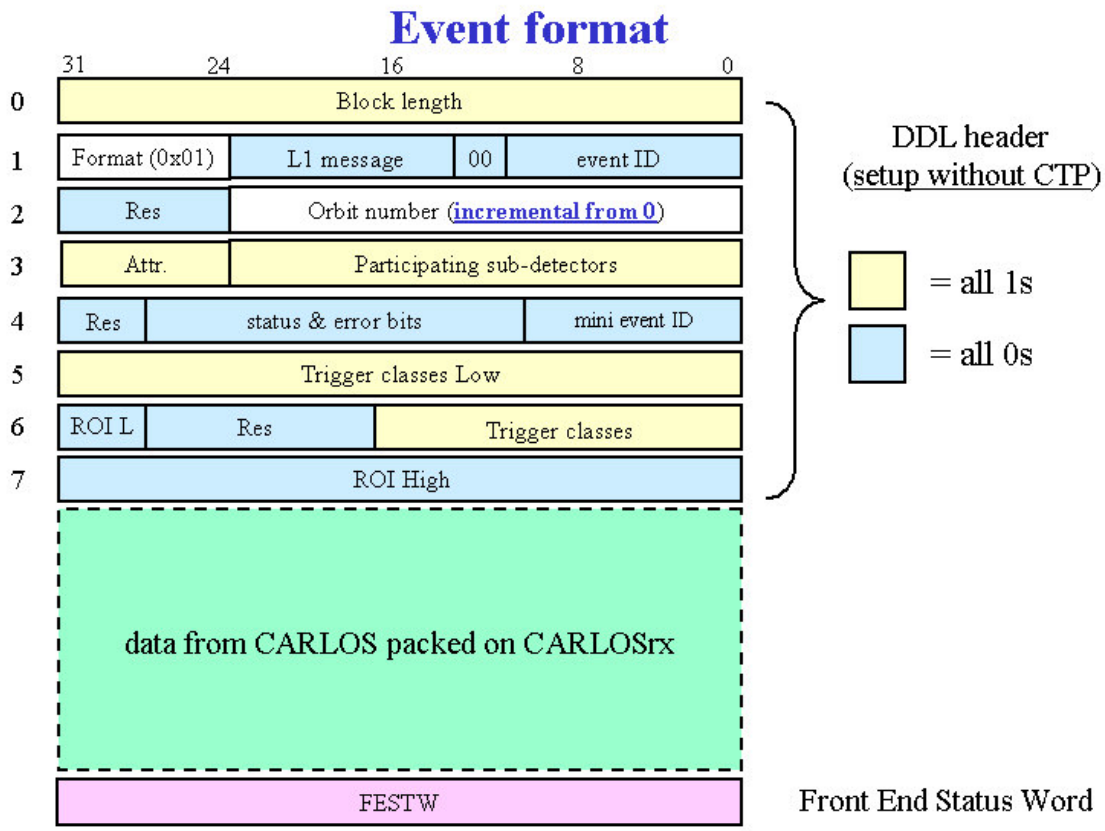


Fig. 8: CARLOSrx event data format

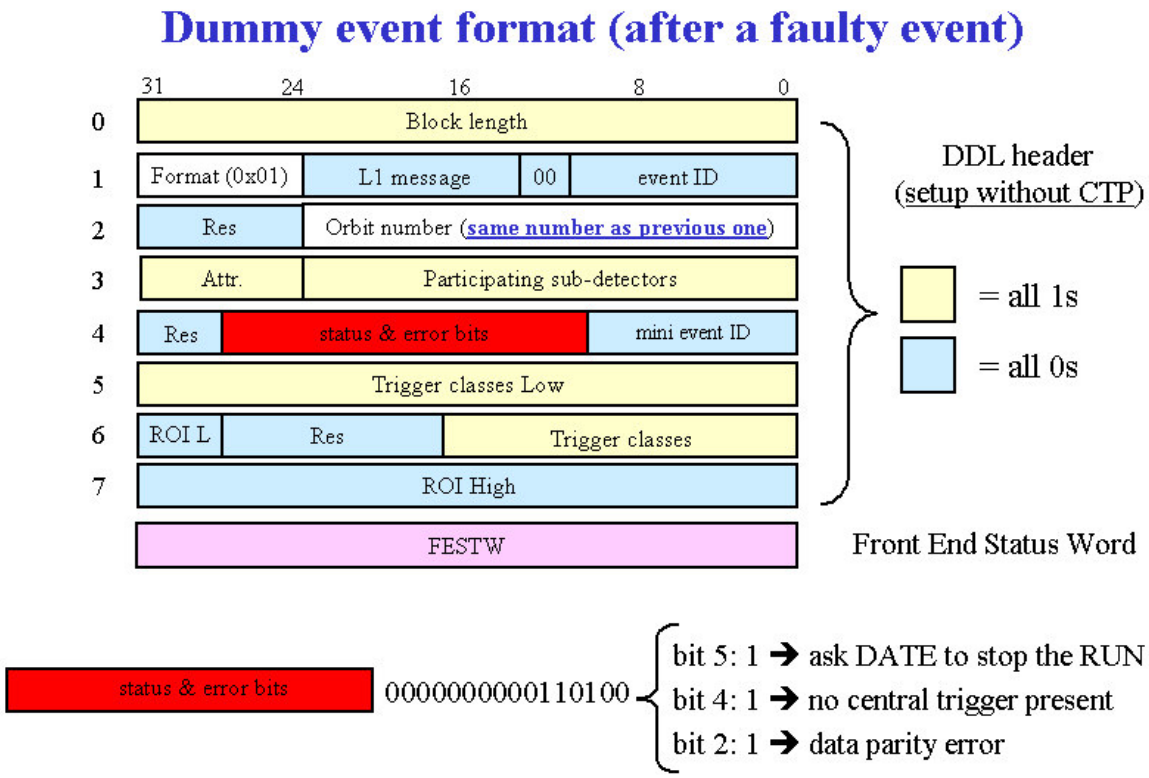


Fig. 9: CARLOSrx dummy event format

Software tool

A C++ software tool has been written in order to ease CARLOSrx debugging stage. The SW decodes data coming out from CARLOSrx and automatically compares the decoded data with the actual CARLOS outputs.