White Rabbit PTP Core User's Manual

wrpc-v4.2 December 18, 2017

Grzegorz Daniluk
Dimitrios Lampridis
Adam Wujek CERN (BE-CO-HT)

Contents

1 Introduction

This is the user manual for the White Rabbit PTP Core (WRPC), part of the White Rabbit project. It describes the building and running process, and it provides a guide to instantiating the WRPC in your own HDL design.

If you don't want to get your hands dirty and prefer to start with the demo binaries available at `http://www.ohwr.org/projects/wr-cores/files` for officially supported boards, please skip Section 2 and move forward directly to Section 3. For help with instantiating the WRPC in your own HDL design, see Section 8.

1.1 Software and hardware requirements

1.1.1 Repositories and Releases

This manual is about the official wrpc-v4.2 stable release of the White Rabbit PTP Core (WRPC). The code and documentation for the project is distributed in the following places:

- `http://www.ohwr.org/projects/wr-cores/documents`
  hosts the pdf documentation for every official release.
- `http://www.ohwr.org/projects/wr-cores/files`
  place where you can find already synthesized demo bitstreams, ready to be downloaded to one of the officially supported boards
- `git://ohwr.org/hdl-core-lib/wr-cores.git`
  repository with HDL sources of the WRPC
- `git://ohwr.org/hdl-core-lib/wr-cores/wrpc-sw.git`
  repository with the LM32 software running inside the WRPC

Other tools useful for building and running the WRPC can be downloaded from the following locations:

- `git://ohwr.org/misc/hdl-make.git`
  hdlmake is used in the HDL synthesis process to create a Makefile and Xilinx ISE / Altera Quartus project file
- `http://www.ohwr.org/attachments/download/1133/lm32.tar.xz`
  LM32 toolchain used to compile the LM32 software running inside the WRPC. This is a 32-bit version of the toolchain. If you encounter problems running this toolchain on modern 64bit machines, try 64 version described below.
- `http://www.ohwr.org/attachments/download/3868/lm32_host_64bit.tar.xz`
  LM32 toolchain used to compile the LM32 software running inside the WRPC (64-bit version of the toolchain).

Repositories containing the WRPC gateware and software (*wr-cores*, *wrpc-sw*) are tagged with `wrpc-v4.1` tag. Other tools used to build the core and load it into the FPGA should be used in their newest stable releases, unless otherwise stated.

1.1.2 Required hardware

The minimum hardware set required to run the WR PTP Core reference firmware depends on the hardware platform you want to use. One of the following setups can be chosen:

- SPEC PCIe board + FMC DIO card + PC computer running Linux
- SVEC VME board + VME crate with a single board computer running Linux[1]
- VFC-HD VME board + FMC DIO card + VME crate with a single board computer running Linux[1]
- FASEC board + FMC DIO card

To be able to test White Rabbit synchronization you would also need additional components regardless of the reference platform chosen from the list above:

- another WR node (e.g. one of the reference boards listed above) or a White Rabbit Switch
- a pair of WR-supported SFP transceivers[2]
- a roll of G652, single mode fiber to connect your WR devices

[1] In our test setup we used MEN A20 board

[2] The list of supported SFPs can be found on our wiki page http://www.ohwr.org/projects/white-rabbit/wiki/SFP

2 Building the Core

Note: you can skip this chapter if you want to use the release binaries available from *ohwr.org*.

Depending on your needs, building the WRPC can be a one- or two-step process. In most of the cases you only need to synthesize the FPGA firmware (section 2.1). This way, you get a working WRPC with the default/release LM32 software running inside the core. If, for some reasons, you need to modify the LM32 software, please check also section 2.2 which contains a description of the software compilation process.

2.1 HDL synthesis

Before running the synthesis process you have to make sure your environment is set up correctly. You will need a synthesis software from your FPGA vendor. Depending on whether you want to run the WRPC on Xilinx (e.g. SPEC, SVEC boards) or Altera/Intel (e.g. VFC-HD) FPGA, you should install either Xilinx ISE or Quartus Prime software.

2.1.1 Setting up Xilinx ISE on Linux

To synthesize the FPGA firmware containing the WRPC, Xilinx ISE with free of charge WebPack license is enough. ISE provides a set of scripts: `settings32.sh`, `settings32.csh`, `settings64.sh` and `settings64.csh` that configure all the system variables to let you easily run the software. Depending on the shell you use and whether your Linux is 32 or 64-bits you should execute one of them before the other tools are used. For 64-bit system and BASH shell you should call (assuming that ISE is installed in the default */opt* directory):

```
$ /opt/Xilinx/<version>/ISE_DS/settings64.sh
```

You can check if the shell is configured correctly by verifying if the `$XILINX` variable contains path to your ISE installation directory.

2.1.2 Setting up Quartus Prime on Linux

To be able to synthesize the WRPC for Arria V FPGA (which is used on the VFC-HD board) you need at least a license for the Quartus Prime Standard Edition with the support of Arria V family. To set up the software after it is installed, you should add the location of its binaries to your `$PATH` environment variable. Assuming you have installed the software in */opt/altera*, the following command should be executed:

```
$ export PATH=/opt/altera/16.0/quartus/bin:$PATH
```

2.1.3 Downloading the sources

Thanks to the *hdlmake* tool, the synthesis process for the reference designs does not differ between Xilinx and Altera/Intel based boards. The tool creates synthesis Makefile as well as ISE/Quartus project file based on a set of Manifest.py files that you will find in the

wr-cores repository.

First, please clone the *hdlmake* repository from its location given in section 1.1.1:

```
$ git clone git://ohwr.org/misc/hdl-make.git <your_location>/hdl-make
$ cd <your_location>/hdl-make
$ git checkout c4789c4
```

This checks out *hdlmake* version 2.1 patched with the Arria V FPGA support (VFC-HD board).

Then, using your favorite editor, you should create an *hdlmake* script in /usr/bin to be able to call it from any directory. The script should have the following content:

```
#!/usr/bin/env bash
python2.7 <path_to_hdlmake_sources>/hdl-make/hdlmake/__main__.py #@
```

Please, make your script executable:

```
$ chmod a+x /usr/bin/hdlmake
```

Having all the tools in place, you can now clone the main WR PTP Core git repository for the v4.1 release. The set of commands below clones the WR PTP Core repository, checks out the release tag, and downloads other HDL repositories (submodules) needed to synthesize the core:

```
$ git clone git://ohwr.org/hdl-core-lib/wr-cores.git <your_location>/wr-cores
$ cd <your_location>/wr-cores
$ git checkout wrpc-v4.1
$ git submodule init
$ git submodule update
```

The local copies of the submodules are stored to:
`<your_location>/wr-cores/ip_cores`

Note: If you use the WRPC within another project (like *wr-nic*), you may need to checkout another release tag for this repository. Please refer to the project's documentation to find out which version of this package you need to build.

2.1.4 Running the synthesis (ISE, Quartus)

The subdirectory you should enter to run the synthesis depends on the hardware platform you use:

- **SPEC**: `<your_location>/wr-cores/syn/spec_ref_design`
- **SVEC**: `<your_location>/wr-cores/syn/svec_ref_design`
- **VFC-HD**: `<your_location>/wr-cores/syn/vfchd_ref_design`

After selecting a proper location from the list above, please call *hdlmake* without any arguments to create the Makefile and project file:

```
$ hdlmake
```

After that, the actual synthesis is just the matter of executing:

```
$ make
```

This takes (depending on your computer) about 10 minutes and should generate bitstream files in various formats depending on your selected reference hardware:

- **SPEC**: `spec_wr_ref_top.bin`, `spec_wr_ref_top.bit`
- **SVEC**: `svec_wr_ref_top.bin`, `svec_wr_ref_top.bit`
- **VFC-HD**: `vfchd_wr_ref.sof`

You can select the bitstream format to be downloaded to FPGA depending on the programming method:

- ***.bin** files to program the Xilinx FPGA on SPEC or SVEC board using the official software support package (*spec-sw*, *svec-sw*). See section 3 for more information.
- ***.bit** files to program the Xilinx FPGA with Xilinx USB Platform Cable (using e.g. Xilinx Impact tool)
- ***.sof** file to program the Intel FPGA (VFC-HD board) using using the Altera / Intel JTAG cable

If, you would like to clean-up the repository to start building everything from scratch you can use the following commands:

- `$ make clean` - removes all synthesis reports and log files;
- `$ make mrproper` - removes `*.bin`, `*.bit` and `*.sof` files;

2.1.5 Running the synthesis (Vivado)

The workflow in Xilinx Vivado is different than for ISE and Quartus. It is heavily based on packed IP cores and schematic entry. Therefore, if you would like to synthesize the reference design for Zynq (FASEC board), you need to go through two Vivado projects:

- `<your_location>/wr-cores/syn/wrc_board_fasec_ip` - is a project created with *hdlmake* from all HDL files necessary to synthesize the WR PTP Core with all its peripherals for Zynq (Kintex-7 FPGA). The project contains also IP-XACT module description to generate Vivado IP Core for further synthesis with a complete FASEC reference design.
- `<your_location>/wr-cores/syn/fasec_ref_design` - is a main reference design project for FASEC board. It is made with the Vivado Block Design instantiating Processing System, reset circuits, AXI interconnects and WR PTP Core IP generated from `wrc_board_fasec_ip` project.

First, please execute the `build.tcl` script in Vivado batch mode [3] to create WR PTP Core IP:

```
$ cd <your_location>/wr-cores/syn/wrc_board_fasec_ip
$ vivado -mode batch -source build.tcl
```

After this step, you can generate the main reference design Vivado project, using Tcl scripts in the repository:

[3] You can also use Tools->Run Tcl Script... from Vivado gui

```
$ cd <your_location>/wr-cores/syn/fasec_ref_design
$ vivado -mode batch -source build.tcl
```

Open the project file `fasec_ref_design.xpr` and continue with regular Vivado flow to synthesize the bitstream.

2.2 LM32 software compilation

Note: By default, the LM32 software for a stable release is embedded inside the FPGA bitstream you've downloaded from *ohwr.org* or synthesized in section 2.1. You can skip this section, unless you need to make some custom changes to the LM32 software and compile it yourself.

Note: To compile also the host tools that come with this software package you will need a *readline-dev* library. In some Linux distributions you would have to install it manually. E.g. in Ubuntu, please install *libreadline-dev* package.

First, you need to download and unpack the LM32 toolchain from the location mentioned in section 1.1.1. The following example uses 32bit version of a toolchain. If you encounter problems running it, please use 64bit version.

```
$ wget http://www.ohwr.org/attachments/download/1133/lm32.tar.xz
$ tar xJf lm32.tar.xz -C <your_location>
```

Then you need to set a `CROSS_COMPILE` environment variable in order to compile the software for the LM32 processor:

```
$ export CROSS_COMPILE="<your_location>/lm32/bin/lm32-elf-"
```

To get the sources of the WRPC software, please clone the *wrpc-sw* git repository tagged with `wrpc-v4.1` tag. The commands in the listing below clone the *wrpc-sw* repository together with submodules needed for this software.

```
$ git clone git://ohwr.org/hdl-core-lib/wr-cores/wrpc-sw.git \
<your_location>/wrpc-sw
$ cd <your_location>/wrpc-sw
$ git checkout wrpc-v4.1
```

Note: If you use WRPC within another project, you may need to checkout a different tag or a specific commit. If this applies, please refer to a documentation for this project.

Before you can compile *wrpc-sw*, you can make some configuration choices. The package uses *Kconfig* as a configuration engine, so you may run one of the following commands (the first two are text-mode, the third uses a KDE GUI and the fourth uses a Gnome GUI):

```
$ make menuconfig
$ make config
$ make xconfig
$ make gconfig
```

Other `Kconfig` target applies, like `config`, `oldconfig` and so on. A few default known-good configurations are found in `./configs` and you choose one by `make`ing it by name. For all three boards mentioned in this manual `spec_defconfig` can be used.

```
$ make spec_defconfig
```

After the package is configured, just run `make` without parameters to build your binary file:

```
$ make
```

The first time you build, the *Makefile* automatically downloads the *git submodules* of this package, unless you already did that by hand. The second and later build won't download anything from the network.

The compilation process produces the binary in 3 different formats:

- *wrc.bin* can be then used with the loader from *spec-sw* or *svec-sw* software package to program the LM32 inside the White Rabbit PTP Core (see section 3 for details).
- *wrc.bram* you can use to initialize WRPC internal RAM with LM32 software during the synthesis for Xilinx FPGAs.
- *wrc.mif* you can use to initialize WRPC internal RAM with LM32 software during the synthesis for Altera FPGAs.

The location of *wrc.bram*/*wrc.mif* files should be passed to the WR PTP Core HDL using the `g_dpram_initf` generic.

3 Programming FPGA with WRPC firmware

3.1 Programming FPGA on SPEC

Note: If you use a more recent version of the `spec-sw` than the one described in this manual, the up-to-date documentation can always be found in *doc/* subdirectory of the `spec-sw` git repository.

First you need to clone the SPEC board software support package (*spec-sw*) from *ohwr.org*. It is a set of Linux kernel drivers and userspace tools, that interacts with a SPEC board plugged into a PCI-Express slot.

```
$ git clone git://ohwr.org/fmc-projects/spec/spec-sw.git <your_location>/spec-sw
$ cd <your_location>/spec-sw
$ git checkout 0745464
$ make
```

You have to download also the "golden" firmware for SPEC card. It is used by the drivers to recognize the hardware:

```
$ wget http://www.ohwr.org/attachments/download/4057/spec-init.bin-2015-09-18
$ sudo mv spec-init.bin-2015-09-18 /lib/firmware/fmc/spec-init.bin
```

Now you can load the drivers necessary to access the SPEC board from your system:

```
$ sudo insmod fmc-bus/kernel/fmc.ko
$ sudo insmod kernel/spec.ko
```

You can use the *dmesg* Linux command to verify if this step succeeded. Among plenty of messages you should be able to find something very similar to:

```
[1275526.738895] spec 0000:20:00.0: probe for device 0020:0000
[1275526.738906] spec 0000:20:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
[1275526.738913] spec 0000:20:00.0: setting latency timer to 64
[1275526.743102] spec 0000:20:00.0: got file "fmc/spec-init.bin", 1485236 (0x16a9b4) bytes
[1275526.934710] spec 0000:20:00.0: FPGA programming successful
[1275527.296754] spec 0000:20:00.0: mezzanine 0
[1275527.296756]     Manufacturer: CERN
[1275527.296757]     Product name: FmcDio5cha
```

Now, you are ready to program the FPGA with your synthesized bitstream from `<your_location>/wr-cores/syn/spec_ref_design/spec_wr_ref_top.bin`

```
$ sudo tools/spec-fwloader \
<your_location>/wr-cores/syn/spec_ref_design/spec_wr_ref_top.bin
```

If everything went right up to this moment you have your board running the FPGA bitstream with the WR PTP Core. If you need to load your own `wrc.bin` compiled in section 2.2, you can use the `spec-cl` tool:

```
$ sudo tools/spec-cl <your_location>/wrpc-sw/wrc.bin
```

After all these steps, you should be able to start a Virtual-UART tool (also part of the *spec-sw* package) that will be used to interact with the WRPC shell:

```
$ sudo tools/spec-vuart
```

If you are able to see the shell prompt *wrc#* this means the Core is up and running on your SPEC card. Congratulations !

Starting from version 3.0, WR PTP Core uses a flash memory chip on the carrier as a default place for storing calibration parameters and an init script. The storage format of this information is organized in SDBFS filesystem. Therefore, starting from v3.0 you have to write the empty SDBFS filesystem image to the flash before running the WRPC. The simplest way of doing this is by calling a WR PTP Core shell command:

```
wrc# sdb fs 0
```

You should see the output similar to:

```
filename: .              ; first: 2e0000; last: 32007f
filename: wr-init        ; first: 2f0000; last: 2f00ff
filename: calibration    ; first: 300000; last: 30007f
filename: mac-address    ; first: 310000; last: 310005
filename: sfp-database   ; first: 320000; last: 32007f
Formatting SDBFS in Flash(0x2e0000)...
```

The other two methods: through the PCIe bus and using a Xilinx JTAG cable are described in appendix C.

3.2 Programming FPGA on other boards

Currently the driver packages for SVEC and VFC-HD boards depend on other CERN drivers and front-end infrastructure. We are working to make them exportable. If you are a CERN user of SVEC, VFC-HD board or SPEC in a Front-end computer, please check https://wikis.cern.ch/display/HT/WR+Nodes (accessible **only** from CERN network) for more instructions. If you are outside CERN and would like to use one of these boards, please contact us.

4 Using WRPC shell

4.1 Writing configuration

First, you should perform a few configuration steps through the WRPC shell before using the core.
Note: the examples below describe only a subset of the WRPC Shell commands. The full list of supported commands can be found in Appendix A.

Before making any configuration changes, it is recommended to stop the PTP daemon. Then, the messages from the daemon will not show up to the console while you interact with the shell.

```
wrc# ptp stop
```

First you should make sure your board has a proper MAC address assigned:

```
wrc# mac get
```

If the result of above command is `MAC-address: 22:33:ww:xx:yy:zz` (where ww, xx, yy, zz are hex values), this means MAC was not yet configured and stored in the Flash/EEPROM. The value is based on thermometer serial number as is unique among SPEC devices. This is globally accepted as "locally assigned", but you might want to assign your own address. A value `22:33:44:55:66:77` is the final fallback if no thermometer is found.

You should get the MAC for your board from its manufacturer. To configure the address and store it into the Flash/EEPROM (so that it's automatically loaded every time the WRPC starts) you should type two commands in the shell:

```
wrc# mac set xx:xx:xx:xx:xx:xx
wrc# mac setp xx:xx:xx:xx:xx:xx
```

where `xx:xx:xx:xx:xx:xx` is the MAC address of your board.

Next, you should input calibration fixed delays values and alpha parameters. The example below clears any existing entries and adds two Axcen transceivers with Δ_{TX}, Δ_{RX} and α parameters associated with them.

```
wrc# sfp erase
wrc# sfp add AXGE-1254-0531 180750 148326 72169888
wrc# sfp add AXGE-3454-0531 180750 148326 -73685416
```

To check the content of the SFP database you can execute the *sfp show* shell command.

Note: The Δ_{TX} and Δ_{RX} parameters above are the defaults for wrpc-v4.1 release bitstream available on *ohwr.org*, running on the SPEC v4 board and calibrated to port 1 of a WR Switch v3.3. These values as well as the default parameters for other boards are available on the calibration wiki page (http://www.ohwr.org/projects/white-rabbit/wiki/Calibration). If you re-synthesize the firmware or want to use the most accurate estimation of the fixed delays and alpha for your fiber, you should read and perform the WR Calibration procedure (http://www.ohwr.org/documents/213).

The WRPC mode of operation (GrandMaster/Master/Slave) can be set using the *mode* command:

```
wrc# mode gm     # for GrandMaster mode
wrc# mode master # for Master mode
wrc# mode slave  # for Slave mode
```

This stops the PTP daemon, changes the mode of operation, but does not start it automatically. Therefore, after calling it, you need to start the daemon manually:

```
wrc# ptp start
```

Note: For running the GrandMaster mode, you need to have the `g_with_external_clock_input` generic enabled in your FPGA firmware. You also must provide 1-PPS and 10MHz signal from an external source (e.g. GPS receiver or Cesium clock). Depending on your board you should connect:

- **SPEC**: DIO mezzanine LEMO No.4 for 1-PPS, LEMO No.5 for 10MHz
- **SVEC**: SVEC LEMO No.4 for 1-PPS, LEMO No.3 for 10MHz
- **VFC-HD**: DIO mezzanine LEMO No.4 for 1-PPS, LEMO No.5 for 10MHz

WRPC has a possibility to load user-defined intialization script on every startup. If you run WRPC release v4.1 or higher, the default LM32 software comes with a built-in init script. The default script disables VLAN support, loads calibration values from the SFP database (provided that it was written earlier by the user) and starts PTP in slave mode. Here are the actual WRPC shell commands executed from the built-in init script:

```
vlan off
ptp stop
sfp match
mode slave
ptp start
```

You can still define your own init script that will be saved in the Flash/EEPROM memory attached to the WRPC. In that case, user-defined set of commands is executed after the built-in script. For example, if you would like to expand the default configuration by your own script that assigns an IP address, you need to to execute the following WRPC shell commands:

```
wrc# init erase
wrc# init add ip set 192.168.1.5
```

Another typical situation would be configuring the WRPC to run in GrandMaster or Master mode. In this case, assuming you also want to configure the IP address (like in the previous example) you would need to run the following WRPC shell commands:

```
wrc# init erase
wrc# init add mode master # for Free-running Master mode or
wrc# init add mode gm   #for GrandMaster mode
wrc# init add ptp start
wrc# init add ip set 192.168.1.5
```

You can always check the content of the built-in and user-defined init scripts by calling `init show` command:

```
wrc# init show
-- built-in script --
vlan off
ptp stop
sfp match
mode slave
ptp start
-- user-defined script --
ip set 192.168.1.5
```

Note: This simple configuration disables VLANs configuration in the WR PTP Core. If, for your network configuration you need to configure VLANs, please check the instructions in section 7.

4.2 Running the Core

Having the SFP calibration database, and eventually a user-defined init script created in 4.1 you can now restart the WRPC by typing the shell command:

```
wrc# init boot
```

You should see log messages that confirm the execution of the initialization script:

```
executing: vlan off
current vlan: 0 (0x0)
executing: ptp stop
PTP stop
executing: sfp match
AXGE-3454-0531
SFP matched, dTx=180750 dRx=148326 alpha=-73685416
executing: mode slave
PTP stop
Locking PLL
executing: ptp start
PTP start
Slave Only, clock class set to 255
executing: ip set 192.168.1.5
IP-address: 192.168.1.5 (static assignment)
```

Now, your device runs as a WR Slave.

Important!!! WRPC needs to make a calibration of t24p phase transition value. It has to be done only once for a new bitstream and is performed automatically when WRPC runs in the Slave mode. That is why it is very important, even if WRPC is meant to run in the Master mode, to configure it to Slave for a moment and connect to any WR Master. This has to be repeated every time a new bitstream (gateware) is deployed. The measured value is automatically stored to Flash/EEPROM and used later in the Master or GrandMaster mode.

The Shell also contains a monitoring function which you can use to check the WR synchronization status:

```
wrc# gui
```

The information is presented in a clear, auto-refreshing screen (see the figure blow). The information is refreshed at every WR iteration or periodically if nothing else happens (so you see an up-to-date timestamp). The period defaults to 1 second and can be changed using the `refresh` command. To exit from this console mode press <Esc>. A full description of the information reporter by *gui* is provided in Appendix B.

Note: the *Synchronization status* and *Timing parameters* in `gui` are available only in the WR Slave mode. When running as WR Master, you can see only the current date and time, link status, Tx and Rx packet counters, IP, lock and calibration status.

```
WR PTP Core Sync Monitor wrpc-v4.0
Esc = exit

TAI Time:                   Thu, Mar 2, 2017, 08:59:27

Link status:
wru1: Link up   (RX: 812, TX: 719) IPv4: 192.168.5.6 (static assignment)
Mode: WR Slave   Locked Calibrated

PTP status: slave

Synchronization status:
Servo state:                TRACK_PHASE
Phase tracking:             ON
Aux clock 0 status:         enabled

Timing parameters:
Round-trip time (mu):               840412 ps
Master-slave delay:                 400910 ps
Master PHY delays:            TX:   224455 ps, RX:   234079 ps
Slave PHY delays:             TX:   180625 ps, RX:   151651 ps
Total link asymmetry:                38592 ps
Cable rtt delay:                     49602 ps
Clock offset:                            3 ps
Phase setpoint:                       7509 ps
Skew:                                   -2 ps
Update counter:                         12
```

If you want to log statistics from the WRPC operation, it is better to use a `stat` shell command. It reports the same information as `gui`, but in a single line, which is easier to parse and analyze:

```
wrc# stat
lnk:1 rx:172338 tx:151811 lock:1 ptp:slave sv:1 ss:'TRACK_PHASE' aux0:1 sec:6047 \
nsec:828412744 mu:836453 dms:398530 dtxm:224455 drxm:232479 dtxs:180667 drxs:149251 \
asym:39393 crtt:49643 cko:1 setp:5082 ucnt:270 hd:31734 md:46228 ad:0 temp: 52.6875 C
lnk:1 rx:172392 tx:151860 lock:1 ptp:slave sv:1 ss:'TRACK_PHASE' aux0:1 sec:6049 \
nsec:399776360 mu:836452 dms:398530 dtxm:224455 drxm:232479 dtxs:180667 drxs:149251 \
asym:39392 crtt:49642 cko:2 setp:5082 ucnt:271 hd:31730 md:46211 ad:0 temp: 52.6875 C
(...)
```

Unlike `gui`, the `stat` command runs asynchronously: you can still issue shell commands while stats are running. You can stop statistics by running `stat` again. As an alternative to the toggling action of `stat` alone, you can use "`stat on`" or "`stat off`".

Statistics are printed every time the WR servo runs; thus no statistics are reported when the node is running in master and GrandMaster mode, nor when your node is running as slave and the master disappeared.

You can verify the synchronization performance by observing the offset between 1-PPS signals from the WR Master and your WRPC running in the Slave mode. Please remember to use oscilloscope cables of the same length and type (with the same delay), or take their delay difference into account in your measurements. Depending on your board, 1-PPS output is produced on:

- **SPEC**: DIO mezzanine LEMO No.1
- **SVEC**: SVEC LEMO No.1
- **VFC-HD**: VFC-HD LEMO L3

5 WRPC diagnostics

5.1 SNMP

Up to the version 4.0 of WRPC the only way to perform diagnostics of the `wrpc-sw` was to use serial console with tools like *gui*, *stat*, etc. For some set-ups, like standalone node, it is very inconvenient to use external console for diagnostics.

Starting with version 4.0 of WRPC, it is possible to include the *Mini SNMP responder*, which allows to perform remote diagnostics using *SNMP* via a port connected to the *Write Rabbit* network.

The configuration file of WRPC contains the following SNMP-related options:

- `CONFIG_SNMP` – include the *Mini SNMP responder* into WRPC
- `CONFIG_SNMP_SET` – enable the support of SNMP *SET* packets
- `CONFIG_SNMP_VERBOSE` – enable verbose output from the *Mini SNMP responder* on the WRPC's console

The MIB file describing WRPC's OIDs can be found in the `lib` directory of the `wrpc-sw` repo. So far, the *Mini SNMP responder* supports version 1 and a subset of version 2c of the SNMP protocol. The following types of requests are supported:

- GET – get value of a given OID
- GETNEXT – get value of a next OID after the given OID (this is used for `snmpwalk`s)
- SET – change the value of a given OID (so far used only for adding SFP's to the database and PTP restarts)

The *Mini SNMP responder* does not support:

- bulk requests packets (GETBULK)
- more than one OID in the request packet
- `trap` and `inform` packets
- encryption
- authentication
- SNMPv2c return error types; all returned error types follows SNMPv1

To make examples more readable, listings below use `SNMP_OPT` environment variable. Make sure you set it properly in your shell.

```
$ SNMP_OPT="-c public -v 2c -m WR-WRPC-MIB -M +/var/lib/mibs/ietf:lib 192.168.1.20"
```

where:

- `-c public` – sets SNMP community as "*public*"
- `-v 2c` – specifies SNMP version
- `-m WR-WRPC-MIB` – specifies MIBs to be loaded
- `-M +/var/lib/mibs/ietf:lib` – contains path to MIBs in the host system (`/var/lib/mibs/ietf`) and path to `WR-WRPC-MIB` (`lib`); on Debian-like systems default MIBs can be downloaded using `download-mibs` command (package `snmp-mibs-downloader`); on CentOS and RedHat MIBs are included in the `libsmi` package

- `192.168.1.20` – the IP address of the target board

For example, to get the system uptime please execute the `snmpget` command:

```
$ snmpget $SNMP_OPT wrpcTimeSystemUptime.0
```

To get a dump of all available OIDs please execute the `snmpwalk` command:

```
$ snmpwalk $SNMP_OPT wrpcCore
```

Part of the `snmpwalk`'s output:

```
WR-WRPC-MIB::wrpcVersionHwType.0 = STRING: spec
WR-WRPC-MIB::wrpcVersionSwVersion.0 = STRING: wrpc-v3.0-251-g14e952e
WR-WRPC-MIB::wrpcVersionSwBuildBy.0 = STRING: Adam Wujek
WR-WRPC-MIB::wrpcVersionSwBuildDate.0 = STRING: Jun 7 2016 18:12:24
WR-WRPC-MIB::wrpcTimeTAI.0 = Counter64: 1465375022
WR-WRPC-MIB::wrpcTimeTAIString.0 = STRING: 2016-06-08-08:37:02
WR-WRPC-MIB::wrpcTimeSystemUptime.0 = Timeticks: (18186) 0:03:01.86
WR-WRPC-MIB::wrpcTemperatureName.1 = STRING: pcb
WR-WRPC-MIB::wrpcTemperatureValue.1 = STRING: 38.5625
WR-WRPC-MIB::wrpcSpllMode.0 = INTEGER: slave(3)
WR-WRPC-MIB::wrpcSpllIrqCnt.0 = Counter32: 1259605
[...]
WR-WRPC-MIB::wrpcPortSfpInDB.0 = INTEGER: inDataBase(2)
WR-WRPC-MIB::wrpcPortInternalTx.0 = Counter32: 452
WR-WRPC-MIB::wrpcPortInternalRx.0 = Counter32: 869
WR-WRPC-MIB::wrpcSfpPn.1 = STRING: AXGE-1254-0531
WR-WRPC-MIB::wrpcSfpDeltaTx.1 = INTEGER: 180750
WR-WRPC-MIB::wrpcSfpDeltaRx.1 = INTEGER: 148326
WR-WRPC-MIB::wrpcSfpAlpha.1 = INTEGER: 72169888
End of MIB
```

It is recommended to use SNMP v2c for communication with a WRPC. Please note that when the version 1 of SNMP is used, 64 bit counters are not supported. This makes impossible to read some WRPC's objects with SNMPv1.

5.1.1 Managing SFP database via SNMP

The SFPs database can be displayed using the `sfp show` command from the WRPC's console:

```
wrc# sfp show
1: PN:AXGE-1254-0531 dTx:  180750 dRx:  148326 alpha: 72169888
2: PN:AXGE-3454-0531 dTx:  180750 dRx:  148326 alpha: -73685416
```

The same data is exported by the *Mini SNMP responder* via the table `wrpcSfpTable`:

```
$ snmpwalk $SNMP_OPT wrpcSfpTable
WR-WRPC-MIB::wrpcSfpPn.1 = STRING: AXGE-1254-0531
WR-WRPC-MIB::wrpcSfpPn.2 = STRING: AXGE-3454-0531
WR-WRPC-MIB::wrpcSfpDeltaTx.1 = INTEGER: 180750
WR-WRPC-MIB::wrpcSfpDeltaTx.2 = INTEGER: 180750
WR-WRPC-MIB::wrpcSfpDeltaRx.1 = INTEGER: 148326
WR-WRPC-MIB::wrpcSfpDeltaRx.2 = INTEGER: 148326
WR-WRPC-MIB::wrpcSfpAlpha.1 = INTEGER: 72169888
WR-WRPC-MIB::wrpcSfpAlpha.2 = INTEGER: -73685416
End of MIB
```

When the SET support is compiled into the *Mini SNMP responder*, it is possible to erase or add/replace SFP entires to the SFPs database via SNMP.

Addition (or modification) of one SFP to the database can be done by a row of SNMP SETs. First, please set the delta Tx (`wrpcPtpConfigDeltaTx.0`), the delta Rx (`wrpcPtpConfigDeltaRx.0`) and the alpha (`wrpcPtpConfigAlpha.0`) with new values. Then, to commit the change to the SFP database, perform the SNMP SET on the `wrpcPtpConfigApply.0` with the value `writeToFlashCurrentSfp`. It will add/update values for the currently plugged SFP.

To add or update entries for other SFPs, you shoud set deltas and alpha like above, set PN of an SFP to the `wrpcPtpConfigSfpPn.0` and commit the change by setting `writeToFlashGivenSfp` to the `wrpcPtpConfigApply.0`.

It is also possible to update parameters of the currently used SFP without storing them to the Flash/EEPROM. For that, please set delta Tx, delta Rx and alpha as described above, then set `writeToMemoryCurrentSfp` to the `wrpcPtpConfigApply.0`. Please remember that these changes are made only in RAM and will be lost after a power cycle of a board, soft reset of WRPC or unplug/plug of a fiber/SFP.

If a database entry or values in RAM of the currently used SFP are updated, it is necessary to perform a restart of the PTP daemon (set `wrpcPtpConfigRestart.0` with the value `restartPtp`). Such restart is necessary because currently PTP does not support on-the-fly changes of deltas nor alpha. It is expected that this behavior will change in the future.

Each SNMP SET of `wrpcPtpConfigApply.0` or `wrpcPtpConfigRestart.0` returns the status of a performed action. For details please check `WR-WRPC-MIB` file.

Commands below add an SFP with PN as "`NEW-SFP`", delta Tx "`1111`", delta Rx "`2222`" and alpha "`3333`".

```
$ snmpset $SNMP_OPT wrpcPtpConfigDeltaTx.0 = 1111
WR-WRPC-MIB::wrpcPtpConfigDeltaTx.0 = INTEGER: 1111
$ snmpset $SNMP_OPT wrpcPtpConfigDeltaRx.0 = 2222
WR-WRPC-MIB::wrpcPtpConfigDeltaRx.0 = INTEGER: 2222
$ snmpset $SNMP_OPT wrpcPtpConfigAlpha.0 = 3333
WR-WRPC-MIB::wrpcPtpConfigAlpha.0 = INTEGER: 3333
$ snmpset $SNMP_OPT wrpcPtpConfigSfpPn.0 = NEW-SFP
WR-WRPC-MIB::wrpcPtpConfigSfpPn.0 = STRING: "NEW-SFP"
$ snmpset $SNMP_OPT wrpcPtpConfigApply.0 = writeToFlashGivenSfp
WR-WRPC-MIB::wrpcPtpConfigApply.0 = INTEGER: applySuccessful(100)
```

In case when the SFP database does not contain the currently plugged SFP, the last `snmpset` command will return `applySuccessfulMatchFailed(101)`.

Optionally restart the PTP:

```
$ snmpset $SNMP_OPT wrpcPtpConfigRestart.0 = restartPtp
WR-WRPC-MIB::wrpcPtpConfigRestart.0 = INTEGER: restartPtpSuccessful(100)
```

Simple verification of performed actions:

```
wrc# sfp show
1: PN:AXGE-1254-0531 dTx:  180750 dRx:  148326 alpha: 72169888
2: PN:AXGE-3454-0531 dTx:  180750 dRx:  148326 alpha: -73685416
3: PN:NEW-SFP        dTx:    1111 dRx:    2222 alpha:   3333
```

The same add can also be achieved by performing `sfp add` command in the WRPC's console:

```
wrc# sfp add NEW-SFP 1111 2222 3333
Update existing SFP entry
3 SFPs in DB
```

Verify the result via SNMP:

```
$ snmpwalk $SNMP_OPT wrpcSfpTable
WR-WRPC-MIB::wrpcSfpPn.1 = STRING: AXGE-1254-0531
WR-WRPC-MIB::wrpcSfpPn.2 = STRING: AXGE-3454-0531
WR-WRPC-MIB::wrpcSfpPn.3 = STRING: NEW-SFP
WR-WRPC-MIB::wrpcSfpDeltaTx.1 = INTEGER: 180750
WR-WRPC-MIB::wrpcSfpDeltaTx.2 = INTEGER: 180750
WR-WRPC-MIB::wrpcSfpDeltaTx.3 = INTEGER: 1111
WR-WRPC-MIB::wrpcSfpDeltaRx.1 = INTEGER: 148326
WR-WRPC-MIB::wrpcSfpDeltaRx.2 = INTEGER: 148326
WR-WRPC-MIB::wrpcSfpDeltaRx.3 = INTEGER: 2222
WR-WRPC-MIB::wrpcSfpAlpha.1 = INTEGER: 72169888
WR-WRPC-MIB::wrpcSfpAlpha.2 = INTEGER: -73685416
WR-WRPC-MIB::wrpcSfpAlpha.3 = INTEGER: 3333
End of MIB
```

It is also possible to erase the SFPs database via SNMP (equivalent of the `sfp erase` command):

```
$ snmpset $SNMP_OPT wrpcPtpConfigApply.0 = eraseFlash
WR-WRPC-MIB::wrpcPtpConfigApply.0 = INTEGER: applySuccessful(100)
```

To verify that database is empty:

```
wrc# sfp show
SFP database empty
```

5.2 Syslog

The node can act as a *syslog* client, though only on the UDP protocol. To activate it, you must build with `CONFIG_SYSLOG` and pass proper parameters at run time.

To configure *syslog* you can run the `syslog` shell command, which receives two parameters (`ipaddr` and `macaddr`), or a single `off` subcommand.

When deploying a network of nodes, you can choose to put the `syslog <ip> <mac>` command in the build-time init command. To do so, you must activate `CONFIG_BUILD_INIT` and then pass your command string as `CONFIG_INIT_COMMAND`. In that context, you can use "`;`" as a command separator as no newlines are permitted in `Kconfig` strings.

The strings that a WR node sends to the *syslog* server are always using the format: "<*level*> *Jan 01 00:00:00 192.168.1.1 msg*" where *level* is usually 14 (type "user", priority "info") and *msg* is a free-format message strings. The *syslog* client sends strings to the server in the following situations:

Boot time	The node sends "`(ma:ca:dd:rr:ee:ss) Node up since` *X* `seconds`" as soon as the network link is up and the *syslog* server is configured with the shell command or init script. The message is re-sent, with an updated uptime value, if you change the *syslog* server parameters.
Link up after link down	The message is "`"Link up after` *2.345* `s`". The time printed is the duration of the link-down interval that has just passed – no lost-by-design message is sent at link-down time. The message is not sent the first time the link goes up, because the boot message is already there.
Synchronization, first time	When the node reaches WR synchronization (i.e. "track phase" state), it sends "`Tracking after` *5..678* `s`". The reported time is the lapse since power-on.
Synchronization lost	Whenever WR looses *track-phase* status, the node reports `Lost track`.
Synchronization recovered	When the WR servo is in *track phase* state after loosing synchronization, the node sends "*45*`-th re-track after` *23.456* `s`". The time reported is the amount of time during which the node has not been synchronized since previous synchronization. The seconth and thirth re-sync are reported as `2-th` and `3-th`, to make you smile. At the `4-th` you should stop smiling and be concerned.

Temperature over threshold

The node monitors various thermometers every few seconds. If `CONFIG_TEMP_POLL_INTERVAL` and related parameters are set, any over-temperature event is reported to *syslog*. If any temperature in the collected set is over threshold, the message is "`Temperature high:`" followed by the list of all collected temperatures. The message is repeated every few seconds (`CONFIG_TEMP_HIGH_RAPPEL`, default 60) until all temperatures are under-threshold. When temperature is recovered the node sends "`Temperature ok:`" followed by the current list of temperatures.

5.3 Diagnostic Tools

This section describes various diagnostics tools that come with the WR PTP Core. These tools are foreseen for the hosted environments as they access various WR PTP Core registers over the External Wishbone interface[4] (through PCIe or VME bridge).

5.3.1 wrpc-diags

A direct access to synchronization status of the WR PTP Core is possible via WB registers. Such access can be used by the application-specific logic or by any software running on the host machine, provided the PCIe/VME bridge is instantiated in your design and lets you access the WR PTP Core external Wishbone Slave interface (see 8.1. Every reference design contains appropriate bridge, so you can use one of them to access conveniently the Wishbone registers for WRPC diagnostics using the `wrpc-diags` tool.

You will find `wrpc-diags` tool in the *wrpc-sw* repository. If you have not yet cloned it, please see section 2.2 for more instructions how this should be done, as well as how to compile it.

The tool is self documented in the sense that a help `h` command lists all the currently supported commands, and to get detailed information for a given command, one can type `h cmd`.

To use the `wrpc-diags` tool interactively, call it with appropriate parameters, depending on your hardware type. For example for a SPEC board at address 01:00.0:

```
$ sudo <your_wrpc-sw_location>/tools/wrpc-diags -o 0x20800 \
   -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

The prompt `wrcdiag[00] >` will appear and you will be able to input commands. All the available commands are listed with `h`:

```
cfv-774-cbt:wrcdiag[00] > h
Valid COMMANDS:
Idx  Name    Params          Description
# 1: q       [          ] -> Quit test program
# 2: h       [ o c      ] -> Help on commands
# 3: a       [          ] -> Atom list commands
# 4: his     [          ] -> History
# 5: s       [ Seconds  ] -> Sleep seconds
# 6: ms      [ MilliSecs ] -> Sleep milliseconds
# 7: sh      [ Unix Cmd ] -> Shell command
# 8: diags   [          ] -> show all wrc diags
# 9: sstat   [          ] -> get servo status
#10: ptpstat [          ] -> get PTP state
#11: pstat   [          ] -> get port status
#12: astat   [          ] -> get auxiliare state
#13: txfcnt  [          ] -> get TX PTP frame count
#14: rxfcnt  [          ] -> get RX frame count
#15: ltime   [          ] -> Local Time expressed in sec since epoch
#16: rttime  [          ] -> Round trip time in picoseconds
#17: msdelay [          ] -> Master slave link delay in picoseconds
#18: asym    [          ] -> Total link asymmetry in picoseonds
#19: cko     [          ] -> Clock offset in picoseonds
#20: setp    [          ] -> Current slave's clock phase shift value
#21: ucnt    [          ] -> Update counter
#22: temp    [          ] -> get board temperature

Type "h name" to get complete command help
```

[4] Please check section 8 for information about the WR PTP Core interfaces.

In order to see all the wrc diagnostics, the `diags` command inside the prompt should be executed as follows:

```
cfc-774-cbt:wrcdiag[00] > diags
servo status:         Track phase
Port status:          Link up, PLL locked,
PTP state:            PPS slave
Aux state:            ch0:enabled
TX frame count:       1593970
RX frame count:       6883447
TAI time:             Wed May 31 15:31:56 2017
Round trip time:      848002 ps
Master slave delay:   417483 ps
Total Link asymmetry: 13036 ps
Clock offset:         2 ps
Phase setpoint:       7883 ps
Update counter:       812977
temp:                 47.3750 C
```

Some advanced commands can be also executed. For example, one can call `temp` command every 1 second for 5 times:

```
cfc-774-cbt:wrcdiag[00] > 5(temp, s 1)
temp:                 47.3750 C
temp:                 47.3750 C
temp:                 47.3750 C
temp:                 47.3750 C
temp:                 47.3750 C
```

The commands that are available from the interactive prompt can be also executed directly from the host's shell prompt and via ssh. Both cases can be useful when writing shell scripts using the tool. To call one of the commands from the host shell prompt, use `echo cmd | wrpc-diags` format. For example, to execute `diags` command for SPEC:

```
echo 'diags' | sudo <your_wrpc-sw_location>/tools/wrpc-diags -o 0x20800 \
   -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

In order to call `temp` command every 1 second for 5 times:

```
echo '5(temp, 1s)' | sudo <your_wrpc-sw_location>/tools/wrpc-diags -o 0x20800 \
   -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

5.3.2 wr-streamers

Note: This tool can be used only if you've selected `STREAMERS` mode for the external WRPC fabric interface or if you have manually instantiated WR Streamers module in your HDL design. See section 8 for more details on WR PTP Core and Board Support Packages interfaces.

The WR Streamers module provides information about transmitted and received frames, which can be useful for diagnostics and debugging. One way of accessing this information is by using the `diag` command of the WR PTP Core Shell. An alternative is to use the `wr-streamers` tool described in this section. You will find it in the *wrpc-sw*

repository. If you have not yet cloned it, please see section 2.2 for more instructions how this should be done, as well as how to compile it.

The available statistical information is identical to the one that can be accessed via the `diag` command of WR PTP Core's Shell. It includes statistics collected since the most recent reset: max/min latency, number of transmitted/received frames, number of lost frames/blocks, number of latency values accumulated, accumulated latency, and the time of the last reset. The wishbone registers can also be used to override the default network configuration of the WR Streamers.

The `wr-streamers` tool is self documented in the sense that a help `h` command lists you all the currently supported commands, and to get detailed information for a given command, one can type `h cmd`.

To use the `wr-streamers` tool interactively, call the tool with appropriate parameters, depending on your hardware type. For example for a SPEC board at address 01:00.0:

```
$ sudo <your_wrpc-sw_location>/tools/wr-streamers -o 0x20700 \
   -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

The prompt `wrstm[00] >` will appear and you will be able to input commands. All the available commands are listed with `h`:

```
cfv-774-cbt:wrstm[00] > h
Valid COMMANDS:
Idx  Name      Params                Description
# 1: q         [                   ] -> Quit test program
# 2: h         [ o c               ] -> Help on commands
# 3: a         [                   ] -> Atom list commands
# 4: his       [                   ] -> History
# 5: s         [ Seconds           ] -> Sleep seconds
# 6: ms        [ MilliSecs         ] -> Sleep milliseconds
# 7: sh        [ Unix Cmd          ] -> Shell command
# 8: stats     [ [0/1/2/3/4/5/6/7] ] -> show streamers statistics
# 9: reset     [                   ] -> show time of the latest reset / time elapsed since then
#10: resetcnt  [                   ] -> reset tx/rx/lost counters and avg/min/max latency values
#11: resetseqid [                  ] -> reset sequence ID of the tx streamer
#12: lat       [ [latency]         ] -> get/set config of fixed latency in integer [us] (-1 to disable)
#13: qtagf     [ [0/1]             ] -> QTags flag on off
#14: qtagvp    [ [VID,prio]        ] -> QTags Get/Set VLAN ID and priority
#15: qtagor    [ [0/1]             ] -> get/set overriding of default qtag config with WB config (set
                                        using qtagf, qtagvp)
#16: ls        [ [leapseconds]     ] -> get/set leap seconds

Type "h name" to get complete command help
```

In order to see all the WR Streamers statistics, the `stats` command inside the prompt should be executed as follows:

```
cfc-774-cbt:wrstm[01] > stats
Latency [us]   : min=    3.736 max=    8.216 avg =  3.88176
Frames [number]: tx =0 rx =61897834620 lost=0 (lost blocks =0)
```

The commands that are available from the interactive prompt can be also executed directly from the host's shell prompt and via ssh. Both cases can be useful when writing shell scripts using the tool. To call one of the commands from the host shell prompt, use `echo cmd | wr-streamers`. For example, to execute `stats` command for SPEC:

```
echo 'stats' | sudo <your_wrpc-sw_location>/tools/wr-streamers -o 0x20700 \
```

```
    -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

In order see the number of received frames every 1 second for 5 times, call:

```
echo '5(stats 1, 1s)' | sudo <your_wrpc-sw_location/tools/wr-streamers -o 0x20700 \
    -f /sys/bus/pci/devices/0000:01:00.0/resource0
```

5.4 Other Diagnostic Methods

5.4.1 Latency Test

The configuration choice `CONFIG_LATENCY_PROBE` activates a mechanism for *wr* nodes to measure network latency, base on the same hardware timestamps that are used for synchronization – the mechanism assumes the nodes are synchronized.

ltest frames can be used to verify whether the network is overloaded and/or has strange inconsistencies in node-to-node delays.

ltest frames use a special ethernet type, `CONFIG_LATENCY_ETHTYPE`, which defaults to 0x0123 (291). If *vlans* are active, these frames are sent and received in the same *vlans* as other CPU frames.

The `ltest` sender periodically sends three frames, with a sequence number. One frames is at priority 7, one is at priority 6, and one is at priority 0. The last frame reports the departure timestamps of the previous frames. The *ltest* receiver uses ingress timestamps to measure latency and report lost frames.

Every node that is built with `CONFIG_LATENCY_PROBE` listens for frames belonging to `CONFIG_LATENCY_ETHTYPE`. A single node in the network is expected to send *ltest* frames; use the `ltest` shell command to select how often to send the *ltest* tuple of three frames. To enable sending every second use "`ltest 1`", to enable sending every 100ms use "`ltest 0 100`", to stop sending use `ltest 0`.

In the sender node, a reminder is sent to the console very 10s reporting that the node is currently sending *ltest* frames. All non-sending nodes report every minute to *syslog*. The report message includes the number of samples received as well as the minimum, average and maximum latency, in nanoseconds. Any lost frames are reported both to the console and to *syslog*.

You can use *ltest* without `CONFIG_SYSLOG`. In that case the receiver nodes print the exact latency (picosecond resolution) for every received event.

5.4.2 wrpc-dump

When trying to diagnose software issues, especially lockup situations, it may be useful to look at the current values or critical variables within *wrpc-sw/ppsi*.

To this aim, you can pass a memory image (dump of RAM content) of *wrpc* to `tools/wrpc-dump`. The tool will print information for softpll, ppsi data structures, ptp data sets and version information.

For example, for the *spec* board, you can use the resource file in *sysfs* to look at a live system, or copy the file for off-line analysis. The following command line show both uses:

```
# Look for "resource0" in /sys/devices/pci, choose your bus number.
FILE=/sys/devices/pci0000:00/0000:00:04.0/0000:04:00.0/resource0

sudo ./tools/wrpc-dump $FILE

sudo tools/mapper $FILE 0 0x20000 > wrpc-memory-image
./tools/wrpc-dump wrpc-memory-image
```

The *mapper* tool used above, that is part of *wrpc-sw*, reads a file using *mmap()*. The kernel doesn't allow plain *read()* from a resource file. If you encounter problem using `wrpc_dump` directly, please use `mapper` then `wrpc_dump`.

Note: Data read by `mapper` may look as if it has wrong endianness compared to the file used for programming LM32 by `spec-cl`, but `spec-cl` is the one which change the endianness of the binary data during the programming.

With *etherbone*, you can get a snapshot for *wrpc* memory using `eb-get` and the proper address (the size is always 128kB)

```
eb-get dev/wbm2 0x4040000/0x20000 wrpc-memory-image
eb-get dev/ttyUSB2 0x4040000/0x20000 wrpc-memory-image
```

We won't show the 180 output lines here, to save some paper, but they are readable dumps of the data structures, including the PTP timestamps.

5.4.3 Softpll Timing

To help understanding the CPU time spent in the *softpll*, you can set `CONFIG_SPLL_FIFO_LOG` in the configuration. The option depends on `CONFIG_DEVELOPER` and is disabled by default.

When the configuration option is set, `wrpc-dump` will also show information about the last 16 *softpll* iterations. Both the `tstamp` and `duration` fields come from reading the `PPSG` nanosecond counter.

```
fifo log at 0x17258
    trr:                        0x0126d921
    tstamp:                     0x06b58d6a
    duration:                   0x00000c7e
    irq_count:                  2305
    tag_count:                  2304
    [... repeats for 5 more events ...]
```

5.4.4 Uptime Counter

Wrpc-sw now maintains an "uptime" counter, in seconds. It lives at binary address 0xa0. It can be queried by *etherbone*, or seen in memory maps.

In addition to knowing how much the node has been up, it can be used to know roughly when a node got stuck, and whether software is still running when a node is not active on the network. Neither of this events happens in production, but the tool is useful during development.

5.4.5 Profiling

The node now has a *ps* command, that shows the number of iterations and time spent in each *task*. Each task reports when it did something (as opposed to just polling the clock or network socket and seeing that nothing is there to do); the *iterations* count shows how many times the task did something. The *max_ms* show the longest execution time of a particular task.

```
wrc# ps
 iterations    seconds.micros  max_ms name
     145560         29.007144      75 idle
          0          0.000000       0 spll-bh
          6          0.001630       1 shell+gui
```

```
       642          2.540416        8 ptp
        31          0.000361        1 uptime
         1          0.050424       51 check-link
        32          0.011172        9 diags
         0          0.000000        0 stats
       235          0.023562        1 net-bh
        32          0.029157        2 ipv4
         0          0.000000        0 arp
         0          0.000000        0 snmp
         6          0.054566       12 temperature
```

By using "`ps reset`" you can zero all counters to start a new test run.

It is possible to configure `ps` in such way that it prints information when any task runs longer than any run before since reset (or `ps reset`) and when it runs longer than a specified value in miliseconds. For this please use command "`ps max <msecs>`", where `<msecs>` is a number of miliseconds triggering printouts.

```
wrc# ps max 10
task temperature, run for 11 ms
task temperature, run for 12 ms
task temperature, run for 11 ms
wrc# ps
[...]
New max run time for a task shell+gui, old 1, new 75
task shell+gui, run for 75 ms
```

5.4.6 Pfilter rules

By setting `CONFIG_PFILTER_VERBOSE`, which depends on `CONFIG_DEVELOPER`, you can get a dump of packet-filter rules whenever they are activated. This happens at initialization time and whenever you change the MAC address or *vlan* choice.

6 Network Services

If built with `CONFIG_IP=y`, *wrpc-sw* implements the following udp-based network services, in addition to *ptp*:

bootp	The node will get an IPV4 address by making *bootp* queries once per second, until it gets a reply from a server. As an alternative, the address can be set using the shell command, and in this case the node won't send *bootp* queries, or will stop sending them.
rdate	The node can be queried using `rdate -u`, once it has an IP address.
syslog	If `CONFIG_SYSLOG` is set, the node is a syslog client. See section 5.2 for details.
snmp	If `CONFIG_SNMP` is set, the node is a snmp agent. See section 5.1 for details.

7 VLAN Support

If you are using the White Rabbit PTP Core version 4.1 or later, it comes by default with VLAN support enabled. You can always disable them in run time using a WRPC shell command:

```
wrc# vlan off
```

Current implementation allows to configure up to 4 VLAN IDs (VIDs) at build time. The following Kconfig options are available in the LM32 software:

CONFIG_VLAN	This is the top-level choice. It can be enabled or disabled. If VLANs are disabled, all incoming tagged frames will be discarded.
CONFIG_VLAN_NR	The default *vlan* number for the CPU (class 0). All network traffic directed to (and originating from) the LM32 processor will belong to this VLAN. The value can be changed at run time using the `vlan set` shell command.
CONFIG_VLAN_1_FOR_CLASS7 *CONFIG_VLAN_2_FOR_CLASS7*	The packet-filter rules are setup to deliver frames belonging to two specific VLANs to class 7, usually Etherbone. To deliver one VLAN only, set the two options to the same number. These VIDs can be modified only at build time.
CONFIG_VLAN_FOR_CLASS6	The packet-filter rules are setup to deliver frames belonging to one specific VLAN to class 6, usually routed to the Streamer module. This VID can be modified only at build time.

Frame classes are assigned inside the WRPC and are used to distinguish received frames between these that should be processed by the WR PTP Core (e.g. PTP frames, SNMP requests) and frames that should be passed to the external WR Fabric interface. Currently, class 0 is used for all frames that should be processed by the WRPC, class 6 is used for Streamers traffic, class 7 is used for Etherbone traffic (see HDL documentation for boards HDL modules and selection between Streamers, Etherbone and Plain modes).

The default WR PTP Core v4.1 release firmware comes with the following VIDs:

- **VID 1** - for class 0, traffic processed inside the WR PTP Core
- **VID 10** and **VID 11** - for class 7, Etherbone traffic
- **VID 20** - for class 6, Streamers traffic

This of course means that all the traffic from VIDs 10, 11, 20 is forwarded and available on the WR PTP Core external fabric interface. The user is free to use it for any other HDL module, not necessary Etherbone or Streamers.

Currently, only VID for the traffic processed by the WRPC (class 0) can be modified in the run time. You can use the WRPC shell command `vlan set <vid>` to modify it. If, for example, your WR node is supposed to synchronize and respond to SNMP requests in VLAN 5, you should add this to the user-defined init script:

```
wrc# init add vlan set 5
```

8 Instantiating WRPC in your own HDL design

This section describes the various options available to the users for instantiating and parametrising the WRPC in their designs.

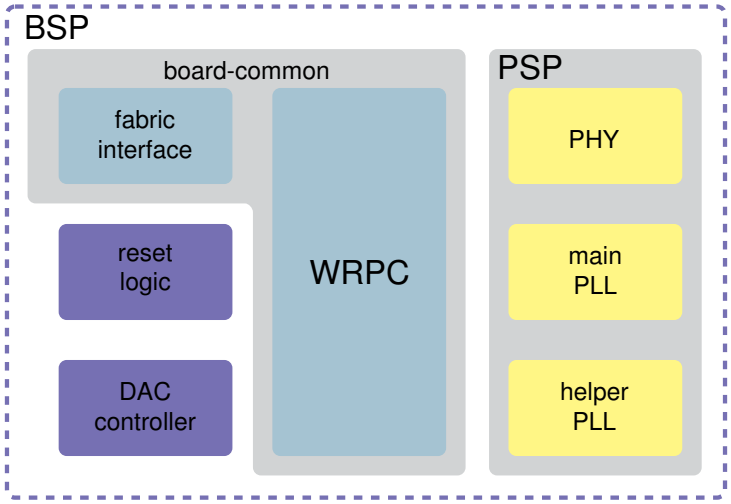


Figure 1: WRPC HDL abstraction hierarchy

The WRPC provides several levels of abstractions and VHDL modules, depending on the target system. These are presented in Figure 1. At the highest level of abstraction, the WRPC provides Board Support Packages (BSPs), available for all officially supported boards. All BSP modules share a common part (the "board-common" module) which encapsulates the WRPC core itself, together with a selection of interfaces for connecting the core the the user FPGA logic. Furthermore, each BSP also makes use of a Platform Support Package (PSP), which groups together and instantiates all the FPGA-specific parts (typically hard IP provided by the FPGA vendor), such as PHY, PLLs and clock buffers.

Thus, depending on the users' systems and needs, several scenarios might be available for instantiating the WRPC into their designs.

Option 1: Supported board. In this simplest of scenarios, it will be enough to just instantiate the provided BSP into the users' designs and configure it via the provided generics.

Option 2: Supported FPGA platform. The users could draw inspiration from an existing BSP based on the same platform, reusing the board-common module and PSP, while adapting the parts that are unique to their designs.

Option 3: Unsupported FPGA platform. There is significant work involved in this scenario. In addition to providing the details for their board (just like for option 2), users also have to write their own PSP. It should be possible though to reuse the board-common module. Furthermore, if the unsupported platform is related to a supported one, it could be that the PHY and/or PLLs will also be reused, perhaps with minor adjustments.

When writing a new BSP or PSP, it's worth discussing it first in the white-rabbit-dev mailing list. Perhaps there is already some preliminary support underway. It's also worth considering sharing your work so that it can be merged with the project and added to the list of supported platforms/boards.

The rest of this section describes the various modules in more detail. The WRPC module is presented in Section 8.1. The platform support modules are presented in Section 8.2, while the board support modules are presented in Section 8.3.

8.1 WR PTP Core component

This section describes the input and output ports of the WRPC IP-core and VHDL generic parameters that can be used to personalize the core.

The top-level VHDL module is located under:
`modules/wrc_core/wr_core.vhd`

A wrapper for the top-level VHDL module which makes use of VHDL records to reduce the number of ports can be found under:
`modules/wrc_core/xwr_core.vhd`

Figure 2 is an example on how to instantiate the WRPC component inside a Xilinx Spartan6-based project. It contains few additional modules besides the WRPC:

- *wr_gtp_phy_spartan6*: module wrapping Xilinx GTP SerDes to improve its determinism
- *PLL_BASE*: Xilinx Spartan6 PLL primitive[5], used to produce 62.5 MHz system clock from 125 MHz local reference clock and to produce the DMTD offset clock from a local 20 MHz oscillator
- *spec_serial_dac_arb*: converts DACs tuning values to serial interface and arbitrates access to two DACs used for reference and DMTD clock tuning.

A very similar example can be found in the WRPC reference design for PCI-Express SPEC board (see Section 8.3.2).

8.1.1 Generic parameters

name	type	default	description
`g_simulation`	integer	`0`	setting to '1' speeds up the simulation, must be set to '0' for synthesis
`g_with_external_clock_input`	boolean	`false`	enable external clock and 1-PPS inputs. The PLL inside WRPC will lock to external 10 MHz and 1-PPS signal when operating in GrandMaster mode
`g_board_name`	string	`"NA "`	board name, exported by WRPC software as SNMP object for diagnostics

[5]see also Xilinx Spartan-6 FPGA Clocking Resources, User Guide

name	type	default	description
g_flash_secsz_kb	integer	256	Flash memory sector size in kilobytes. Available through a Wishbone register, used by WRPC software to read/write SDBFS image
g_flash_sdbfs_baddr	integer	0x600000	Default base address in Flash memory where `sdb fs` command should store an empty SDBFS image
g_phys_uart	boolean	true	enable physical UART interface
g_virtual_uart	boolean	false	enable virtual UART interface
g_aux_clks	integer	0	number of aux clocks syntonized by WRPC to WR timebase
g_rx_buffer_size	integer	1024	size of Rx buffer in WRPC MAC module, default value is 1024 and should not be changed
g_tx_runt_padding	boolean	true	when set to true, all user frames transmitted from the external fabric interface are padded if shorter than minimal Ethernet frame size (60B with header)
g_dpram_initf	string	""	filename of compiled WRPC software, to be stored in WRPC memory during the synthesis (default is *wrc.ram* created by compiling WRPC software from *wrpc-sw* git repository)
g_dpram_size	integer	32768	size of RAM used by WRPC software (in 32-bit words), default value is 22528 and should not be changed
g_interface_mode	enum	PIPELINED	external Wishbone Slave interface mode [PIPELINED/CLASSIC]
g_address_granularity	enum	BYTE	granularity of address bus in external Wishbone Slave interface [BYTE/WORD]
g_aux_sdb	rec	c_wrc_periph3_sdb	structure providing an SDB descriptor for the peripheral attached to the WRPC auxiliary WB interface. This parameter is optional and can be left unassigned. The default value corresponds to an undocumented device with an address space of 256 bytes
g_softpll_enable_debugger	boolean	false	when set to true, additional FIFO is instantiated in the SoftPLL for collecting DMTD tags. It can be read out by the host and analyzed for SoftPLL debugging.

name	type	default	description
g_vuart_fifo_size	integer	1024	size (in bytes) for the virtual UART FIFO
g_pcs_16bit	boolean	false	when set to true, make use of 16-bit PCS, otherwise use 8-bit PCS
g_records_for_phy	boolean	false	when set to true, all the PHY-related signals will be grouped in the phy8/phy16 VHDL records, otherwise the individual standard logic signals will be used
g_diag_id	integer	0	auxiliary diagnostics module ID
g_diag_ver	integer	0	auxiliary diagnostics version for a given module ID
g_diag_ro_size	integer	0	number of read-only registers fed to auxiliary diagnostics
g_diag_rw_size	integer	0	number of read-write registers fed to auxiliary diagnostics

8.1.2 Ports

name	dir	size	description
Clocks and resets			
clk_sys_i	in	1	main system clock, can be any frequency $\leq f_{clk\_ref\_i}$ e.g. 62.5 MHz
clk_dmtd_i	in	1	DMTD offset clock (close to 62.5 MHz, e.g. 62.49 MHz)
clk_ref_i	in	1	125 MHz reference clock
clk_aux_i	in	var	[optional] vector of auxiliary clocks that will be disciplined to WR timebase. Size is equal to g_aux_clks
clk_ext_mul_i	in	1	125 MHz clock, derived from clk_ext_i
clk_ext_mul_locked_i	in	1	PLL locked indicator for clk_ext_mul_i
clk_ext_stopped_i	in	1	PLL stopped indicator for clk_ext_mul_i
clk_ext_rst_o	out	1	Reset output to be used for clk_ext_mul_i
clk_ext_i	in	1	[optional] external 10 MHz reference clock input for GrandMaster mode
pps_ext_i	in	1	[optional] external 1-PPS input used in GrandMaster mode
rst_n_i	in	1	main reset input, active-low (hold for at least 5 clk_sys_i cycles)

name	dir	size	description
Timing system			
dac_hpll_load_p1_o	out	1	validates DAC value on data port
dac_hpll_data_o	out	16	DAC value for tuning helper (DMTD) VCXO
dac_dpll_load_p1_o	out	1	validates DAC value on data port
dac_dpll_data_o	out	16	DAC value for tuning main (ref) VCXO
PHY inteface (when g_records_for_phy = false)			
phy_ref_clk_i	in	1	TX clock
phy_tx_data_o	out	var	TX data. If g_pcs_16bit = true, then size = 16, else size=8
phy_tx_k_o	out	var	1 when phy_tx_data_o contains a control code, 0 when it's a data byte. If g_pcs_16bit = true, then size = 2, else size=1
phy_tx_disparity_i	in	1	disparity of the currently transmitted 8b10b code (1 for positive, 0 for negative)
phy_tx_enc_err_i	in	1	TX encoding error indication
phy_rx_data_i	in	var	RX data. If g_pcs_16bit = true, then size = 16, else size=8
phy_rx_rbclk_i	in	1	RX recovered clock
phy_rx_k_i	in	var	1 when phy_rx_data_i contains a control code, 0 when it's a data byte. If g_pcs_16bit = true, then size = 2, else size=1
phy_rx_enc_err_i	in	1	RX encoding error indication
phy_rx_bitslide_i	in	var	RX bitslide indication. If g_pcs_16bit = true, then size = 5, else size=4
phy_rst_o	out	1	PHY reset, active high
phy_rdy_i	in	1	PHY is ready: locked and aligned
phy_loopen_o	out	1	local loopback enable (TX→RX), active high
phy_loopen_vec_o	out	3	
phy_tx_prbs_sel_o	out	3	PRBS select (see Xilinx UG386 Table 3-15; "000" = Standard operation, pattern generator off)
phy_sfp_tx_fault_i	in	1	SFP TX fault indicator
phy_sfp_los_i	in	1	SFP Loss Of Signal indicator
phy_sfp_tx_disable_o	out	1	SFP TX disable control
PHY inteface (when g_records_for_phy = true)			
phy8_o	out	rec	input/output records for PHY signals when g_pcs_16bit = false
phy8_i	in	rec	
phy16_o	out	rec	input/output records for PHY signals when g_pcs_16bit = true
phy16_i	in	rec	

name	dir	size	description
GPIO			
led_act_o	out	1	signal for driving Ethernet activity LED
led_link_o	out	1	signal for driving Ethernet link LED
sda_i	in	1	I2C interface for EEPROM memory storing calibration
sda_o	out	1	
scl_i	in	1	
scl_o	out	1	
sfp_sda_i	in	1	I2C interface for EEPROM inside SFP module
sfp_sda_o	out	1	
sfp_scl_i	in	1	
sfp_scl_o	out	1	
sfp_det_i	in	1	SFP presence indicator
btn1_i	in	1	two microswitch inputs, active low, currently not used in official WRPC software
btn2_i	in	1	
spi_sclk_o	out	1	Flash SPI SCLK
spi_ncs_o	out	1	Flash SPI $\overline{SS}$
spi_mosi_o	out	1	Flash SPI MOSI
spi_miso_i	in	1	Flash SPI MISO
UART			
uart_rxd_i	in	1	[optional] serial UART interface for interaction with WRPC software
uart_txd_o	out	1	
OneWire			
owr_pwren_o	out	1	[optional] 1-Wire interface used to read the temperature of hardware board from digital thermometer (e.g. Dallas DS18B20)
owr_en_o	out	1	
owr_i	in	1	
External WB interface			
wb_adr_i	in	32	Wishbone slave interface that operates in Pipelined or Classic mode (selected with g_interface_mode), with the address bus granularity controlled with g_address_granularity
wb_dat_i	in	32	
wb_dat_o	out	32	
wb_sel_i	in	4	
wb_we_i	in	1	
wb_cyc_i	in	1	
wb_stb_i	in	1	
wb_ack_o	out	1	
wb_err_o	out	1	
wb_rty_o	out	1	
wb_stall_o	out	1	
wb_slave_o	out	rec	Alternative record-based ports for the WB slave interface (available in xwr_core.vhd)
wb_slave_i	in	rec	

name	dir	size	description
Auxiliary WB master			
aux_adr_i	in	32	Auxilirary Wishbone pipelined master interface
aux_dat_o	out	32	
aux_dat_i	in	32	
aux_sel_o	out	4	
aux_we_o	out	1	
aux_cyc_o	out	1	
aux_stb_o	out	1	
aux_ack_i	in	1	
aux_stall_i	in	1	
aux_master_o	out	rec	Alternative record-based ports for the aux WB master interface (available in xwr_core.vhd)
aux_master_i	in	rec	
External fabric interface			
ext_snk_adr_i	in	2	External fabric Wishbone pipelined interface, direction Sink→Source
ext_snk_dat_i	in	16	
ext_snk_sel_i	in	2	
ext_snk_cyc_i	in	1	
ext_snk_stb_i	in	1	
ext_snk_we_i	in	1	
ext_snk_ack_o	out	1	
ext_snk_err_o	out	1	
ext_snk_stall_o	out	1	
ext_src_adr_o	out	2	External fabric Wishbone pipelined interface, direction Source→Sink
ext_src_dat_o	out	16	
ext_src_sel_o	out	2	
ext_src_cyc_o	out	1	
ext_src_stb_o	out	1	
ext_src_we_o	out	1	
ext_src_ack_i	in	1	
ext_src_err_i	in	1	
ext_src_stall_i	in	1	
wrf_src_o	out	rec	Alternative record-based ports for the fabric interface (available in xwr_core.vhd)
wrf_src_i	in	rec	
wrf_snk_o	out	rec	
wrf_snk_i	in	rec	

name	dir	size	description
External TX timestamp interface			
txtsu_port_id_o	out	5	physical port ID from which the timestamp was originated. WRPC has only one physical port, so this value is always 0.
txtsu_frame_id_o	out	16	frame ID for which the timestamp is available
txtsu_ts_value_o	out	32	Tx timestamp value
txtsu_ts_incorrect_o	out	1	Tx timestamp is not reliable since it was generated while PPS generator inside WRPC was being adjusted
txtsu_stb_o	out	1	strobe signal that validates the rest of signals described above
timestamps_o	out	rec	Alternative record-based output ports for the TX timestamp interface (available in xwr_core.vhd)
txtsu_ack_i	in	1	acknowledge, indicating that user-defined module has received the timestamp
Pause frame control			
fc_tx_pause_req_i	in	1	Ethernet flow control, request sending Pause frame
fc_tx_pause_delay_i	in	16	Pause quanta
fc_tx_pause_ready_o	out	1	Pause acknowledge - active after the current pause send request has been completed
Timecode/Servo control			
tm_link_up_o	out	1	state of Ethernet link (up/down), 1 means Ethernet link is up
tm_dac_value_o	out	24	DAC value for tuning auxiliary clock (clk_aux_i)
tm_dac_wr_o	out	var	validates auxiliary DAC value. Size is equal to g_aux_clks
tm_clk_aux_lock_en_i	in	var	enable locking auxiliary clock to internal WR clock. Size is equal to g_aux_clks
tm_clk_aux_locked_o	out	var	auxiliary clock locked to internal WR clock. Size is equal to g_aux_clks
tm_time_valid_o	out	1	if 1, the timecode generated by the WRPC is valid
tm_tai_o	out	40	TAI part of the timecode (full seconds)
tm_cycles_o	out	28	fractional part of each second represented by the state of counter clocked with the frequency 125 MHz (values from 0 to 124999999, each count is 8 ns)

name	dir	size	description
pps_p_o	out	1	1-PPS signal generated in clk_ref_i clock domain and aligned to WR time, pulse generated when the cycle counter is 0 (beginning of each full TAI second)
pps_led_o	out	1	1-PPS signal with extended pulse width to drive a LED
rst_aux_n_o	out	1	Auxiliary reset output, active low
link_ok_o	out	1	Link status indicator
Auxiliary diagnostics to/from external modules			
aux_diag_i	in	var	Arrays of 32-bit vectors, to be accessed from WRPC via SNMP or uart console. Input array contains g_diag_ro_size elements, while output array contains g_diag_rw_size elements
aux_diag_o	out	var	

8.1.3 PHY interface

The interface connects WRPC with the Ethernet PHY layer IP-core. The interface is generic, but currently four Gigabit Ethernet PHYs are tested and supported:

1. Xilinx Spartan6 8-bit GTP SerDes
2. Xilinx Virtex6 16-bit GTX SerDes
3. Xilinx Kintex7 16-bit GTX SerDes
4. Altera "Deterministic Latency" Transceiver PHY (tested on Arria V)

Depending on the value of the g_records_for_phy and g_pcs_16bit generic parameters, WRPC expects to find the PHY signals in one of the following ports:

g_records_for_phy	g_pcs_16bit	PHY ports
false	false	individual standard logic ports
false	true	
true	false	phy8 record-based ports
true	true	phy16 record-based ports

Important: If a WRPC user wants to use one of the supported PHYs, they have to be taken from the WRPC repository instead of manually generating them with the Xilinx/Altera tools. That is because WR developers have attached additional logic to them to improve their determinism. The easiest way of doing so is to make use of the provided Platform/Board Support Packages (see Sections 8.2 and 8.3).

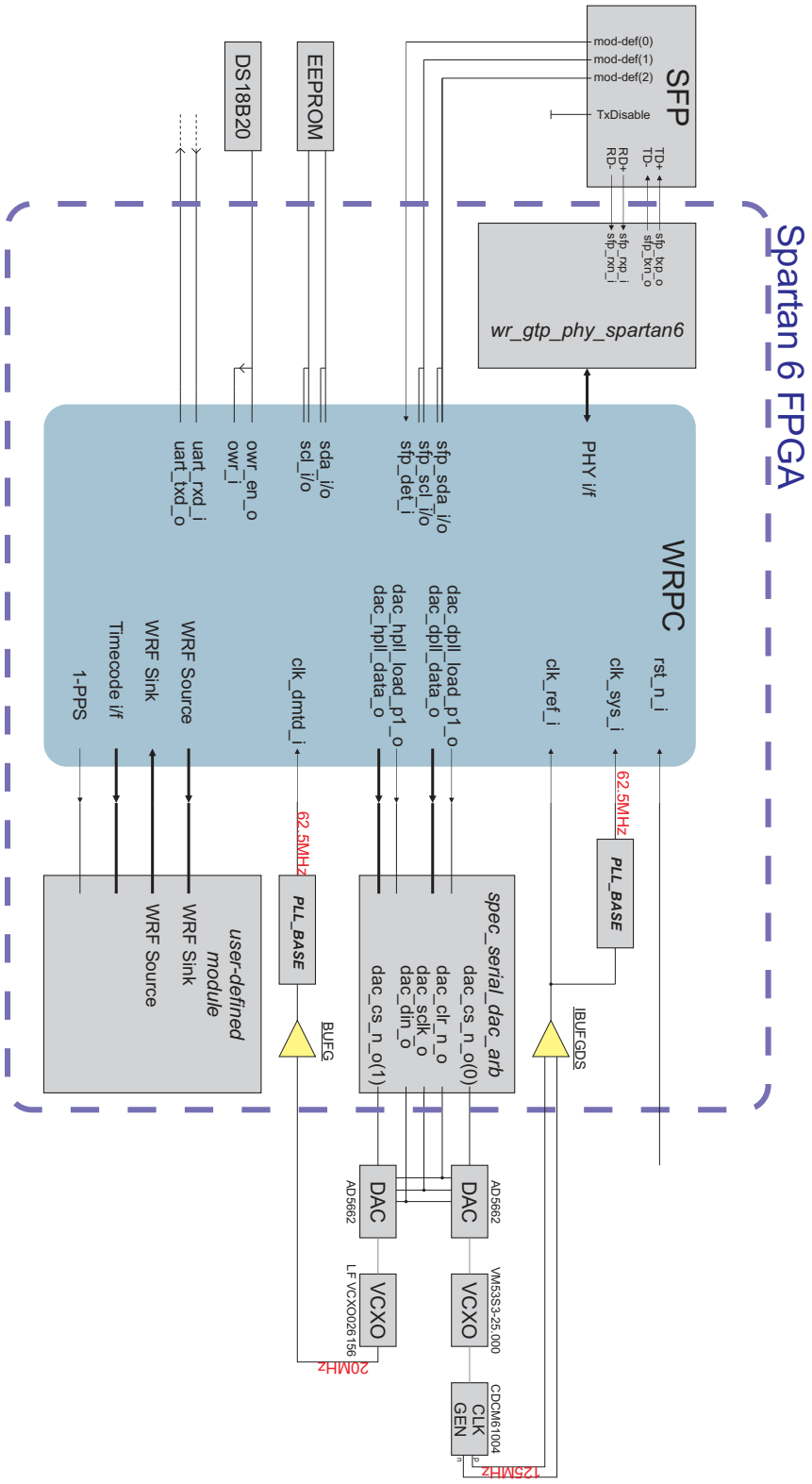


Figure 2: Simple top design with WRPC

8.1.4 Peripherals

Several hardware peripherals can be connected to the White Rabbit PTP Core. It has:

- UART - provides access to the WR PTP Core user shell
- 1-Wire - access to a digital thermometer for an on-board temperature and unique ID (used to generate a default MAC address of the WR port)
- SFP I^2C - access to the SFP EEPROM, to read its ID and math with the calibration values
- SPI - access to the Flash memory, used to store calibration parameters and init script
- EEPROM I^2C - [optional] access to the EEPROM memory, used to store calibration parameters and init script - currently SPI Flash is the preferred storage, however, EEPROM can still be used if needed.

8.1.5 External Wishbone Slave/Master interface

Ext WB Slave is a Wishbone slave interface[6]. It controls the primary Wishbone Crossbar insisde the WRPC and thus provides access to all the WRPC internals.

In most designs, this slave interface should be connected to the host (if any), via an aprropriate bridge. As an example, in the SPEC WRPC reference design it is connected to a Gennum GN4124 IP core, and in the SVEC/VFC-HD reference designs it is connected to the VME64x IP core. In all three reference designs, this interface is used to upload WRPC software to its internal memory and to access the WRPC VUART.

HDL modules accessible through *Ext WB Slave* interface include:

module name	**offset (bytes)**
WRPC internal memory	0x00000
Mini NIC	0x20000
Endpoint	0x20100
Soft PLL	0x20200
PPS generator	0x20300
Syscon	0x20400
UART	0x20500
1-Wire Master	0x20600
Aux WB Master	0x20700

Aux WB Master is a Pipelined Wishbone Master interface. It is connected through the Wishbone Crossbar inside the White Rabbit PTP Core to the LM32 soft-core processor (instantiated inside the WRPC). It can optionally be used to control any user-defined module having a Pipelined Wishbone Slave interface. In that case, the WRPC software has to be modified to control additional modules connected to the *Aux WB Master* interface. An alternative is to access the Aux WB master interface from the host (via the external Wishbone slave interface).

[6] see the Wishbone bus specification (rev.B4) for more details

8.1.6 Fabric interface

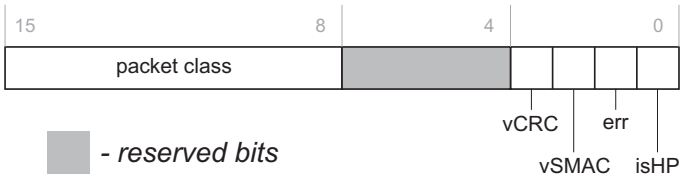
The Fabric interface is used for sending and receiving Ethernet frames. It consists of two pipelined Wishbone interfaces operating independently:

- *WRF Source*: pipelined Wishbone Master, passes all the Ethernet frames received from a physical link to WRF Sink interface implemented in a user-defined module.
- *WRF Sink*: pipelined Wishbone Slave, receives Ethernet frames from the WRF Source implemented in the user-defined module, and sends them to a physical link.

Address bus can have one of the following values:

decimal value	meaning of data word on data bus
0	regular data (packet header and payload)
1	OOB (Out-of-band) data
2	status word
3	currently not used

Status word (sent when the value of address bus is *2*) contains various information about Ethernet frame's structure and type:



isHP - if *1*, the frame is high priority

err - if *1*, the frame contains an error

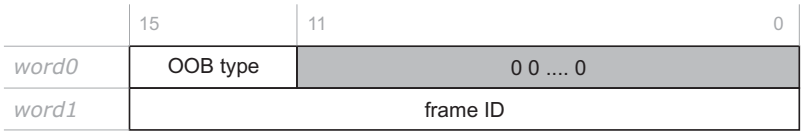
vSMAC - the frame contains a source MAC address (otherwise it will be assigned from WRPC configuration)

vCRC - the frame contains a valid CRC checksum

packet class - the packet class assigned by the classifier inside WRPC MAC module

OOB data is used for passing the timestamp-related information for the incoming and outgoing Ethernet frames. Each frame received from a physical link is timestamped inside the WRPC and this value is passed as Rx OOB data. On the other hand, for each transmitted frame the Tx timestamp can be read from the Tx Timestamping Interface (section 8.1.7) together with a unique frame number assigned in Tx OOB. Therefore, the format of OOB differs between Rx and Tx frames.

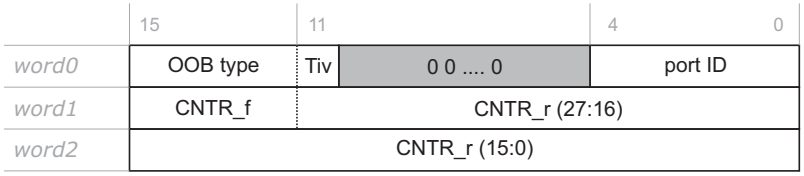
Tx OOB format:



OOB type: "0001" means Tx OOB

frame ID: ID of the frame being sent. It is later output through the *Tx Timestamping interface* to associate Tx timestamp with appropriate frame. Frame ID = 0 is reserved for PTP packets inside WRPC and cannot be used by user-defined modules.

Rx OOB format:



OOB type: "0000" means Rx OOB

Tiv: timestamp invalid. When this bit is set to '1', the PPS generator inside WRPC is being adjusted which means the Rx timestamp is not reliable.

port ID: the ID of a physical port on which the packet was received. In case of WRPC, this field is always 0, because there is only one physical port available.

CNTR_f: least significant bits of the Rx timestamp generated on the falling edge of the reference clock.

CNTR_r: Rx timestamp generated on the rising edge of the reference clock.

Figure 3 presents data words fed to the WRF data bus by the sender and the information got at the receiving side. Please note that the CRC checksum is calculated and inserted automatically inside the WRPC and user-defined module doesn't care about it. The Ethernet frame received from the WR Fabric interface may contain additional OOB data suffixed. It has to be received (acknowledged) by the user-defined module, but can be simply discarded.

Examples

Figure 4 shows a very simple WR Fabric cycle. The WRF Source of user-defined module sends there an Ethernet frame containing even number of bytes.

1. The WRF Source in user-defined module starts the cycle by asserting *cyc_o*, *stb_o* and putting a status word to the data bus. However, since WRF Sink set *stall* signal to active state, Source has to wait until Sink is ready to receive data.

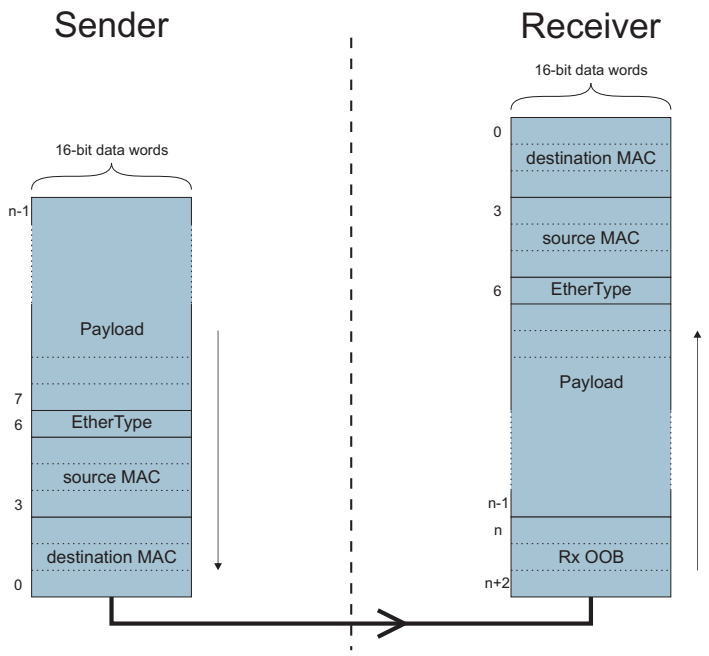


Figure 3: Data words that make the Ethernet frame

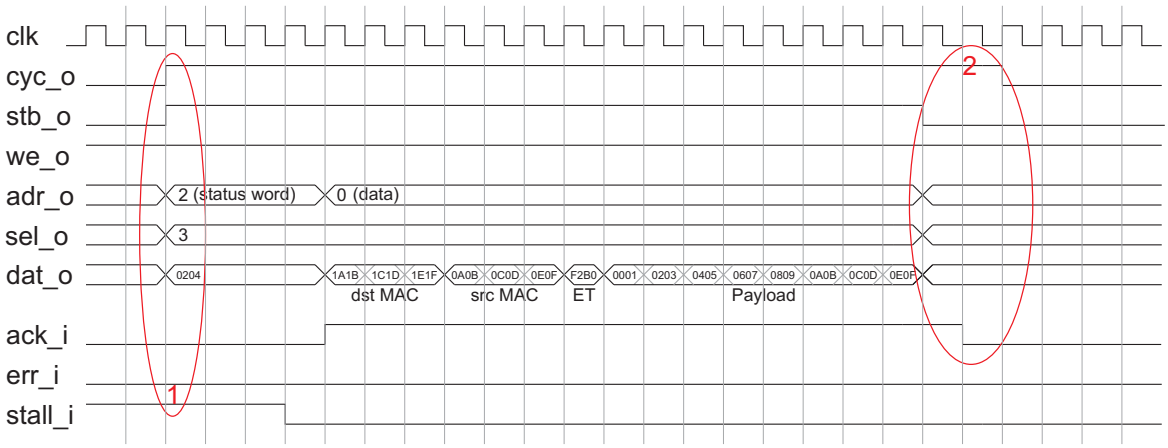


Figure 4: Simple WR Fabric cycle - user-defined module sending packet

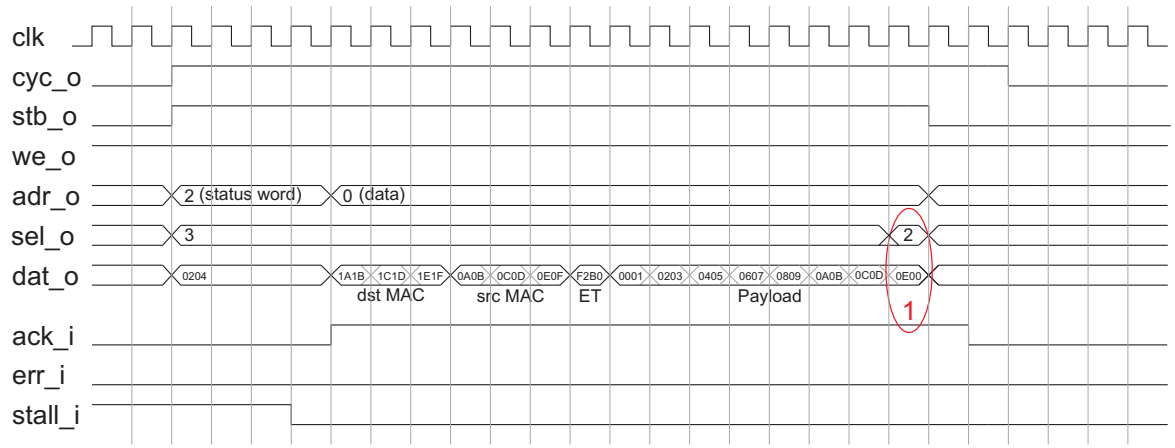


Figure 5: Simple WR Fabric cycle - user-defined module sending packet(odd number of bytes in the payload)

2. After the last word is transmitted, the WRF Source sets *stb_o* back to *0*, but waits until Sink acknowledges all the words transmitted in the cycle (*ack_i* line). The cycle ends when *cyc_o* goes back to the low state.

Figure 5 shows again a very simple WR Fabric cycle where user-defined WRF Source sends an Ethernet frame to the WRPC. This time though, the frame contains odd number of bytes, therefore the *sel* line is used to signal this fact to WRF Sink inside the WRPC (1).

Figure 6 presents more complicated Fabric cycle where an Ethernet frame is received from WRF Source in the WRPC (output signals in the diagram are driven by WRF Source on the WRPC side):

1. The WRF Source starts the cycle by asserting *cyc_o*, *stb_o* and putting a status word to the data bus. However, since WRF Sink set *stall* signal to active state, Source has to wait until Sink is ready to receive data.

2. While the payload of the Ethernet frame is being transmitted, Sink stalls the cycle. The WRF Source pauses the transmission until Sink becomes ready to process the rest of the data. During that time *stb_o* has to remain in a high state.

3. The Ethernet frame contains an odd number of bytes, so only half of last word of payload carries a valid data. *Sel_o* is used to signal this fact to WRF Sink.

4. After the whole payload is transmitted, Source may additionally sent Rx OOB data. It contains some internal WRPC data that should be acknowledged by Sink, but discarded in the user's module.

5. After the last word is transmitted, the WRF Source sets *stb_o* back to *0*, but waits until Sink acknowledges all the words transmitted in the cycle (*ack_i* line). The cycle ends when *cyc_o* goes back to the low state.

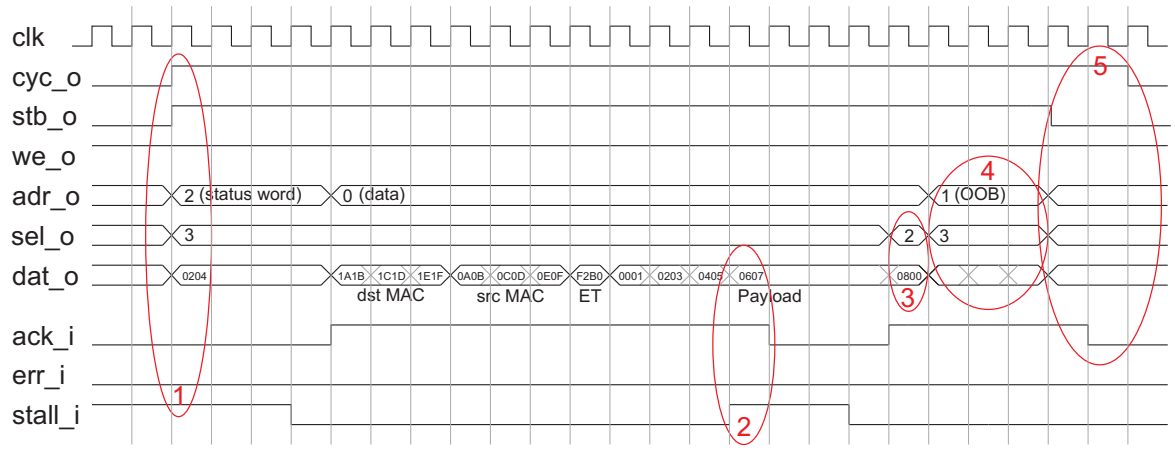


Figure 6: WR Fabric cycle

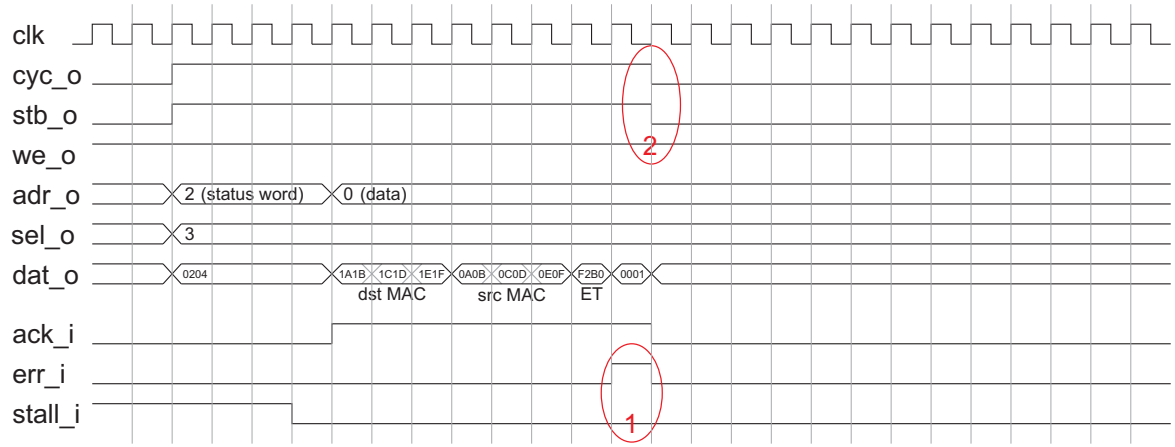


Figure 7: WR Fabric cycle interrupted with an error line

WRF Sink can use the *stall* line to pause the frame transmission if it cannot process the flow of data coming from WRF Source. However, if some more serious problem appears on the receiving side, the *err* line can be used to immediately break the cycle. This situation is presented in figure 7:

1. WRF Sink wants to break a bus cycle, so it drives *err_i* high.
2. WRF Source breaks the cycle immediately after receiving an error indicator from the WRF Sink.

SystemVerilog model

The SystemVerilog simulation model of the WR Fabric interface (both WRF Source and WRF Sink) can be found in the *wr-cores* git repository (git://ohwr.org/hdl-core-lib/wr-cores.git) and consists of the files:

- *sim/if_wb_master.svh*
- *sim/if_wb_slave.svh*
- *sim/wb_packet_source.svh*
- *sim/wb_packet_sink.svh*

The testbench example using the simulation model of WR Fabric interface can be found in the zip archive attached to this documentation.

8.1.7 Tx Timestamping interface

The Tx Timestamping interface provides the timestamps generated inside WRPC for each Ethernet frame transmitted from user-defined module through the WRF Sink interface.

8.1.8 Aux clocks

The WRPC can syntonize auxiliary clock signals to the White Rabbit timebase. It is done with a similar PLL that is used to discipline the local reference clock. WRPC provides tuning values for the VCXO producing clock signal which is connected to *clk_aux_i*.

8.1.9 Timecode interface

Timecode interface provides current time to the other HDL modules in a form that can be easily used. It consists of: a 1-PPS and a TAI timecode aligned to the time of WR Master.

8.1.10 Auxiliary diagnostics interface

Auxiliary diagnostics interface can be used if a user would like to benefit from the WR PTP Core diagnostics capabilities to export some registers from his/her IP core. The interface consists of two 32-bit `std_logic_vector` arrays. User-defined registers that are to be read from the WRPC SNMP agent (SNMP GET requests), should be connected to the `aux_diag_i` vector. User-defined values that are written from the WRPC SNMP agent (SNMP SET requests) will be available in the `aux_diag_o` vector.

Two VHDL generics `g_diag_id` and `g_diag_ver` are used to let the user uniquely identify given application (user-defined set of registers) to match it with appropriate, custom SNMP MIB file.

8.2 Platform Support Packages

The White Rabbit (WR) PTP core project provides platform support packages (PSPs) for Altera and Xilinx FPGAs.

By using these modules, users gain the benefit of instantiating all the platform-specific support components for the WR PTP core (PHY, PLLs, etc.) in one go, without having to delve into the implementation details, using a setup that has been tested and is known to work well on the supported FPGAs.

8.2.1 Common

This section describes the generic parameters and ports which are common to all provided PSPs.

Generic parameters

name	type	default	description
`g_with_external_clock_input`	boolean	`false`	Select whether to include the external 10MHz reference clock input (used in WR Grandmaster mode)
`g_use_default_plls`	boolean	`true`	Set to FALSE if you want to instantiate your own PLLs

Each PSP provides two generic parameters of boolean type , which allow the users to configure the PLLs in their designs. As such, four different PLL setups can be achieved by changing the values of these parameters.

PLL setup 1: Use default PLLs, no external reference clock. In this setup, the PSP expects one 20MHz and one 125MHz clock, and it will instantiate all the required PLLs internally. This is the default mode.

PLL setup 2: Use default PLLs, with external reference clock. This is the same as PLL setup 1, with the addition of the external 10MHz reference clock input, which will be multiplied internally by the PSP to 125MHz.

PLL setup 3: Use custom PLLs, no external reference clock. In this setup, the PSP will not instantiate any PLLs internally. It is up to the user to provide the 62.5MHz system clock, the 125MHz reference clock and the 62.5MHz DMTD clock.

PLL setup 4: Use custom PLLs, with external reference clock. This is the same as PLL setup 3, with the addition of the external reference clock input, which should be provided as is (10MHz) and also multiplied to 125MHz.

Ports

name	dir	size	description
`areset_n_i`	in	`1`	asynchronous reset (active low)
`clk_10m_ext_i`	in	`1`	10MHz external reference clock input (used when `g_with_external_clock_input = true`)

name	dir	size	description
Clock inputs for default PLLs (used when `g_use_default_plls = true`)			
`clk_20m_vcxo_i`	in	1	20MHz VCXO clock
`clk_125m_pllref_i`	in	1	125MHz PLL reference
Interface with custom PLLs (used when `g_use_default_plls = false`)			
`clk_62m5_dmtd_i`	in	1	62.5MHz DMTD offset clock and lock status
`clk_dmtd_locked_i`	in	1	
`clk_62m5_sys_i`	in	1	62.5MHz Main system clock and lock status
`clk_sys_locked_i`	in	1	
`clk_125m_ref_i`	in	1	125MHz Reference clock
`clk_125m_ext_i`	in	1	125MHz derived from 10MHz external reference, locked/stopped status inputs and reset output (used when `g_with_external_clock_input = true`)
`clk_ext_locked_i`	in	1	
`clk_ext_stopped_i`	in	1	
`clk_ext_rst_o`	out	1	
Interface with SFP			
`sfp_tx_fault_i`	in	1	TX fault indicator
`sfp_los_i`	in	1	Loss Of Signal indicator
`sfp_tx_disable_o`	out	1	TX disable control
Interface with WRPC			
`clk_62m5_sys_o`	out	1	62.5MHz system clock output
`clk_125m_ref_o`	out	1	125MHz reference clock output
`clk_62m5_dmtd_o`	out	1	62.5MHz DMTD clock output
`pll_locked_o`	out	1	logic AND of system and DMTD PLL lock
`clk_10m_ext_o`	out	1	10MHz external reference clock output
`phy8_o`	out	`rec`	input/output records for PHY signals when `g_pcs_16bit = false`
`phy8_i`	in	`rec`	
`phy16_o`	out	`rec`	input/output records for PHY signals when `g_pcs_16bit = true`
`phy16_i`	in	`rec`	
`ext_ref_mul_o`	out	1	125MHz derived from 10MHz external reference, locked/stopped status outputs and reset input
`ext_ref_mul_locked_o`	out	1	
`ext_ref_mul_stopped_o`	out	1	
`ext_ref_rst_i`	in	1	

8.2.2 Altera

The Altera PSP currently supports the Arria V family of FPGAs.

The top-level VHDL module is located under:
`platform/altera/xwrc_platform_altera.vhd`

A VHDL package with the definition of the module can be found under:
platform/wr_altera_pkg.vhd

An example (VHDL) instantiation of this module can be found in the VFC-HD board support package (see also Section 8.3.4):
board/vfchd/xwrc_board_vfchd.vhd

This section describes the generic parameters and ports which are specific to the Altera PSP. Parameters and ports common to all PSPs are described in Section 8.2.1.

Generic parameters

name	type	default	description
g_fpga_family	string	arria5	Defines the family/model of Altera FPGA. Recognized values are "arria5" (more will be added)
g_pcs16_bit	boolean	false	Some FPGA families provide the possibility to configure the PCS of the PHY as either 8bit or 16bit. The default is to use the 8bit PCS, but this generic can be used to override it

Ports

name	dir	size	description
Interface with SFP			
sfp_tx_o	out	1	PHY TX and RX. These are single ended and should be mapped to the positive half of each differential signal. Altera tools will infer both the negative half and the differential receiver
sfp_rx_i	in	1	

8.2.3 Xilinx

The Xilinx PSP currently supports the Spartan 6 and Kintex 7 (also inside Zynq) family of FPGAs.

The top-level VHDL module is located under:
platform/xilinx/xwrc_platform_xilinx.vhd

A VHDL package with the definition of the module can be found under:
platform/wr_xilinx_pkg.vhd
Examples of (VHDL) instantiation of this module can be found in the SPEC, SVEC and FASEC board support packages (see also Sections 8.3.2 and 8.3.3):
board/spec/xwrc_board_spec.vhd
board/svec/xwrc_board_svec.vhd
board/fasec/xwrc_board_fasec.vhd

This section describes the generic parameters and ports which are specific to the Xilinx PSP. Parameters and ports common to all PSPs are described in Section 8.2.1.

Generic parameters

name	type	default	description
g_fpga_family	string	spartan6	Defines the family/model of Xilinx FPGA. Recognized values are "spartan6" (more will be added)
g_simulation	integer	0	setting to '1' speeds up the simulation, must be set to '0' for synthesis

Ports

name	dir	size	description
clk_125m_gtp_p_i	in	1	125MHz GTP reference differential clock input
clk_125m_gtp_n_i	in	1	
Interface with SFP			
sfp_txn_o	out	1	differential pair for PHY TX
sfp_txp_o	out	1	
sfp_rxn_i	in	1	differential pair for PHY RX
sfp_rxp_i	in	1	

8.3 Board Support Packages

The White Rabbit (WR) PTP core project provides board support packages (BSPs) for the following boards:

- SPEC, a PCIe single FMC carrier board based on a Xilinx Spartan 6 FPGA.

- SVEC, a VME dual FMC carrier board based on a Xilinx Spartan 6 FPGA.
- VFC-HD, a VME single FMC carrier board based on an Altera Arria V FPGA.

By using these modules, users gain the benefit of instantiating all the necessary components of the WR PTP core (including the core itself, the PHY, PLLs, etc.) in one go, without having to delve into the implementation details, using a setup that has been tested and is known to work well on the supported boards.

Each BSP is split in two modules: the common module, which is shared across all BSPs, and the board-specific module. The common module instantiates the WRPC itself, together with a selection of interfaces for connecting the core the the user FPGA logic. The board-specific module instantiates all the FPGA- and system-specific parts (related to WR), such as hard IP provided by the FPGA vendor, interfaces to DACs, reset inputs, etc.

The BSPs make use internally of the appropriate FPGA family platform support packages (PSPs, see also Section 8.2). For users who need more control and flexibility over their designs, it is suggested to use the BSP as a reference, and to consider instantiating directly the respective PSP for their FPGA family.

8.3.1 Common

Most of the generic parameters and ports of the board-common module map directly to those of the WRPC. One notable exception to this rule is that of the parameters and ports related to the selected interface for connecting the core to the user FPGA logic.

The board-common module provides the `g_fabric_iface` generic parameter, an enumeration type with three possible values:

PLAIN: No additional module is instantiated and the "raw" WRPC fabric interface (see also Section 8.1.6) is provided on the board's ports.

STREAMERS: A set of TX/RX streamers is attached to the WRPC fabric interface.

ETHERBONE: An Etherbone slave node is attached to the WRPC fabric interface.

Sections 8.3.1 and 8.3.1 list the generic parameters and ports of the board-common module which are shared across the BSPs.

Note: the board-common module defines more parameters and ports than the ones mentioned in the following sections. Those that are not exposed by any of the BSPs have been left out to keep the tables short and to the point. Users interested in studying the board-common module and/or writing their own BSP, can find the board-common module under:
`board/common/xwrc_board_common.vhd`

Generic parameters

name	type	default	description
g_simulation	integer	0	These map directly to generic parameters with the same name in the WRPC module (see Section 8.1.1)
g_with_external_clock_input	boolean	true	
g_board_name	string	"NA "	
g_flash_secsz_kb	integer	256	
g_flash_sdbfs_baddr	integer	0x600000	
g_aux_clks	integer	0	
g_dpram_initf	string	""	
g_diag_id	integer	0	
g_diag_ver	integer	0	
g_diag_ro_size	integer	0	
g_diag_rw_size	integer	0	
g_streamers_op_mode	enum	TX_AND_RX	Selects whether both TX and RX streamer modules should be instantiated or only one of them when g_fabric_iface = STREAMERS (otherwise ignored)
g_tx_streamer_params	record	default record	various TX/RX streamers parameters when g_fabric_iface = STREAMERS (otherwise ignored)[7]
g_rx_streamers_params	record	default record	
g_fabric_iface	enum	PLAIN	optional module to be attached to the fabric interface of WRPC [PLAIN/STREAMERS/ETHERBONE]

Ports

name	dir	size	description
Clocks and resets			
clk_aux_i	in	var	[optional] vector of auxiliary clocks that will be disciplined to WR timebase. Size is equal to g_aux_clks

[7] See Streamers wiki page for detailed description of the configuration records: http://www.ohwr.org/projects/wr-cores/wiki/TxRx_Streamers

name	dir	size	description
clk_10m_ext_i	in	1	10MHz external reference clock input (used when g_with_external_clock_input = true)
pps_ext_i	in	1	external 1-PPS input (used when g_with_external_clock_input = true)
clk_sys_62m5_o	out	1	62.5MHz system clock output
clk_ref_125m_o	out	1	125MHz reference clock output
rst_62m5_n_o	out	1	Active low reset output, synchronous to clk_sys_62m5_o
rst_125m_n_o	out	1	Active low reset output, synchronous to clk_ref_125m_o
Interface with SFP			
sfp_tx_fault_i	in	1	TX fault indicator
sfp_los_i	in	1	Loss Of Signal indicator
sfp_tx_disable_o	out	1	TX disable control
I2C EEPROM interface			
eeprom_sda_i	in	1	EEPROM I2C SDA
eeprom_sda_o	out	1	
eeprom_scl_i	in	1	EEPROM I2C SCL
eeprom_scl_o	out	1	
Onewire interface (UID and temperature)			
onewire_i	in	1	OneWire data input
onewire_oen_o	out	1	OneWire data output enable (when asserted, OneWire tri-state output buffer should be enabled and driven to ground)
External WishBone interface			
wb_slave_o	out	rec	Mapped to WRPC external WB slave interface (see also Section 8.1.5)
wb_slave_i	in	rec	
aux_master_o	out	rec	Mapped to WRPC auxiliary WB master interface (see also Section 8.1.5)
aux_master_i	in	rec	
WR fabric interface (when g_fabric_iface = plain)			
wrf_src_o	out	rec	Mapped to WRPC fabric interface (see also Section 8.1.6)
wrf_src_i	in	rec	
wrf_snk_o	out	rec	
wrf_snk_i	in	rec	
WR streamers (when g_fabric_iface = streamers)			
wrs_tx_data_i	in	var	Data to be sent. Size is equal to g_tx_streamer_width
wrs_tx_valid_i	in	1	Indicates whether wrs_tx_data_i contains valid data
wrs_tx_dreq_o	out	1	When active, the user may send a data word in the following clock cycle

name	dir	size	description
wrs_tx_last_i	in	1	Can be used to indicate the last data word in a larger block of samples
wrs_tx_flush_i	in	1	When asserted, the streamer will immediatly send out all the data that is stored in its TX buffer
wrs_tx_cfg_i	in	rec	Networking configuration of Tx/Rx Streamers[8]
wrs_rx_cfg_i	in	rec	
wrs_rx_first_o	out	1	Indicates the first word of the data block on wrs_rx_data_o
wrs_rx_last_o	out	1	Indicates the last word of the data block on wrs_rx_data_o
wrs_rx_data_o	out	var	Received data. Size is equal to g_rx_streamer_width
wrs_rx_valid_o	out	1	Indicates that wrs_rx_data_o contains valid data
wrs_rx_dreq_i	in	1	When asserted, the streamer may output another data word in the subsequent clock cycle
Etherbone WB master interface (when g_fabric_iface = etherbone)			
wb_eth_master_o	out	rec	WB master interface for the Etherbone core. Normally this is attached to a slave port of the primary WB crossbar in the design, in order to provide access to all WB peripherals over Etherbone
wb_eth_master_i	in	rec	
Generic diagnostics interface			
aux_diag_i	in	var	Arrays of 32 bit vectors, to be accessed from WRPC via SNMP or uart console. Input array contains g_diag_ro_size, while output array contains g_diag_rw_size elements.
aux_diag_o	out	var	
Auxiliary clocks control			
tm_dac_value_o	out	24	DAC value for tuning auxiliary clock (*clk_aux_i*)
tm_dac_wr_o	out	var	validates auxiliary DAC value. Size is equal to g_aux_clks
tm_clk_aux_lock_en_i	in	var	enable locking auxiliary clock to internal WR clock. Size is equal to g_aux_clks
tm_clk_aux_locked_o	out	var	auxiliary clock locked to internal WR clock. Size is equal to g_aux_clks

[8] See Streamers wiki page for detailed description of the network configuration: http://www.ohwr.org/projects/wr-cores/wiki/TxRx_Streamers

name	dir	size	description
External TX timestamp interface			
timestamps_o	out	rec	Record-based output ports for the TX timestamp interface (see also Section 8.1.7)
txtsu_ack_i	in	1	acknowledge, indicating that user-defined module has received the timestamp
abscal_txts_o	out	1	[optional] Endpoint timestamping triggers used in the absolute calibration procedure
abscal_rxts_o	out	1	
Pause frame control			
fc_tx_pause_req_i	in	1	[optional] Ethernet flow control, request sending Pause frame
fc_tx_pause_delay_i	in	16	[optional] Pause quanta
fc_tx_pause_ready_o	out	1	[optional] Pause acknowledge - active after the current pause send request has been completed
WRPC timecode interface			
tm_link_up_o	out	1	state of Ethernet link (up/down), *1* means Ethernet link is up
tm_time_valid_o	out	1	if *1*, the timecode generated by the WRPC is valid
tm_tai_o	out	40	TAI part of the timecode (full seconds)
tm_cycles_o	out	28	fractional part of each second represented by the state of counter clocked with the frequency 125 MHz (values from 0 to 124999999, each count is 8 ns)
Buttons, LEDs and PPS output			
led_act_o	out	1	signal for driving Ethernet activity LED
led_link_o	out	1	signal for driving Ethernet link LED
btn1_i	in	1	two microswitch inputs, active low, currently not used in official WRPC software
btn2_i	in	1	
pps_p_o	out	1	1-PPS signal generated in clk_ref_i clock domain and aligned to WR time, pulse generated when the cycle counter is 0 (beginning of each full TAI second)
pps_led_o	out	1	1-PPS signal with extended pulse width to drive a LED
link_ok_o	out	1	Link status indicator

8.3.2 SPEC

The SPEC BSP provides a ready-to-use WRPC wrapper for the SPEC carrier board.

The top-level VHDL module is located under:

board/spec/xwrc_board_spec.vhd

An alternative top-level VHDL module which only makes use of standard logic for ports and integers and strings for generic parameters (ideal for instantiation in Verilog-based designs) can be found under:
board/spec/wrc_board_spec.vhd

A VHDL package with the definition of both modules can be found under:
board/spec/wr_spec_pkg.vhd

An example (VHDL) instantiation of this module can be found in the SPEC WRPC reference design:
top/spec_ref_design/spec_wr_ref_top.vhd

This section describes the generic parameters and ports which are specific to the SPEC BSP. Parameters and ports common to all BSPs are described in Section 8.3.1.

Generic parameters

No additional generic parameters are declared in the SPEC BSP. See Section 8.3.1 for a the list of common BSP parameters.

Ports

name	dir	size	description
Clocks and resets			
areset_n_i	in	1	Reset input (active low, can be async)
areset_edge_n_i	in	1	[optional] Reset input edge sensitive (active rising-edge, can be async). Should be connected to PCIe reset if the board should be able to operate both in hosted and standalone configuration.
clk_20m_vcxo_i	in	1	20MHz clock input from board VCXO
clk_125m_pllref_p_i	in	1	125MHz PLL reference differential clock input from board
clk_125m_pllref_n_i	in	1	
clk_125m_gtp_p_i	in	1	125MHz GTP reference differential clock input from board
clk_125m_gtp_n_i	in	1	
SPI interface to DACs			
plldac_sclk_o	out	1	SPI SCLK, common to both DACs
plldac_din_o	out	1	SPI MOSI, common to both DACs
pll25dac_cs_n_o	out	1	SPI $\overline{SS}$ for DAC controlling 25MHz oscillator
pll20dac_cs_n_o	out	1	SPI $\overline{SS}$ for DAC controlling 20MHz oscillator
SFP interface			
sfp_txn_o	out	1	differential pair for PHY TX
sfp_txp_o	out	1	
sfp_rxn_i	in	1	differential pair for PHY RX
sfp_rxp_i	in	1	

name	dir	size	description
sfp_det_i	in	1	Active low, indicates presence of SFP (corresponds to SFP MOD-DEF0)
sfp_sda_i	in	1	SFP I2C SDA
sfp_sda_o	out	1	
sfp_scl_i	in	1	SFP I2C SCL
sfp_scl_o	out	1	
sfp_rate_select_o	out	1	SFP rate select
Physical UART interface			
uart_rxd_i	in	1	UART RXD (serial data to WRPC)
uart_txd_o	out	1	UART TXD (serial data from WRPC)
Flash memory SPI interface			
flash_sclk_o	out	1	Flash SPI SCLK
flash_ncs_o	out	1	Flash SPI $\overline{SS}$
flash_mosi_o	out	1	Flash SPI MOSI
flash_miso_i	in	1	Flash SPI MISO

8.3.3 SVEC

The SVEC BSP provides a ready-to-use WRPC wrapper for the SVEC carrier board.

The top-level VHDL module is located under:

board/svec/xwrc_board_svec.vhd

An alternative top-level VHDL module which only makes use of standard logic for ports and integers and strings for generic parameters (ideal for instantiation in Verilog-based designs) can be found under:

board/svec/wrc_board_svec.vhd

A VHDL package with the definition of both modules can be found under:

board/svec/wr_svec_pkg.vhd

An example (VHDL) instantiation of this module can be found in the SVEC WRPC reference design:

top/svec_ref_design/svec_wr_ref_top.vhd

This section describes the generic parameters and ports which are specific to the SVEC BSP. Parameters and ports common to all BSPs are described in Section 8.3.1.

Generic parameters

No additional generic parameters are declared in the SVEC BSP. See Section 8.3.1 for a the list of common BSP parameters.

Ports

name	dir	size	description
Clocks and resets			
areset_n_i	in	1	Reset input (active low, can be async)
areset_edge_n_i	in	1	[optional] Reset input edge sensitive (active rising-edge, can be async).
clk_20m_vcxo_i	in	1	20MHz clock input from board VCXO
clk_125m_pllref_p_i	in	1	125MHz PLL reference differential clock input from board
clk_125m_pllref_n_i	in	1	
clk_125m_gtp_p_i	in	1	125MHz GTP reference differential clock input from board
clk_125m_gtp_n_i	in	1	
SPI interface to DACs			
pll20dac_sclk_o	out	1	SPI SCLKfor DAC controlling 20MHz oscillator
pll20dac_din_o	out	1	SPI MOSI for DAC controlling 20MHz oscillator
pll20dac_cs_n_o	out	1	SPI $\overline{SS}$ for DAC controlling 20MHz oscillator
pll25dac_sclk_o	out	1	SPI SCLK for DAC controlling 25MHz oscillator
pll25dac_din_o	out	1	SPI MOSI for DAC controlling 25MHz oscillator
pll25dac_cs_n_o	out	1	SPI $\overline{SS}$ for DAC controlling 25MHz oscillator
SFP interface			
sfp_txn_o	out	1	differential pair for PHY TX
sfp_txp_o	out	1	
sfp_rxn_i	in	1	differential pair for PHY RX
sfp_rxp_i	in	1	
sfp_det_i	in	1	Active low, indicates presence of SFP (corresponds to SFP MOD-DEF0)
sfp_sda_i	in	1	SFP I2C SDA
sfp_sda_o	out	1	
sfp_scl_i	in	1	SFP I2C SCL
sfp_scl_o	out	1	
sfp_rate_select_o	out	1	SFP rate select
Physical UART interface			
uart_rxd_i	in	1	UART RXD (serial data to WRPC)
uart_txd_o	out	1	UART TXD (serial data from WRPC)
Flash memory SPI interface			
spi_sclk_o	out	1	Flash SPI SCLK
spi_ncs_o	out	1	Flash SPI $\overline{SS}$
spi_mosi_o	out	1	Flash SPI MOSI
spi_miso_i	in	1	Flash SPI MISO

8.3.4 VFC-HD

The VFC-HD BSP provides a ready-to-use WRPC wrapper for the VFC-HD carrier board.

The top-level VHDL module is located under:

`board/vfchd/xwrc_board_vfchd.vhd`

An alternative top-level VHDL module which only makes use of standard logic for ports and integers and strings for generic parameters (ideal for instantiation in Verilog-based designs) can be found under:

`board/vfchd/wrc_board_vfchd.vhd`

A VHDL package with the definition of both modules can be found under:

`board/vfchd/wr_vfchd_pkg.vhd`

An example (VHDL) instantiation of this module can be found in the VFC-HD WRPC reference design:

`top/vfchd_ref_design/vfchd_wr_ref_top.vhd`

This section describes the generic parameters and ports which are specific to the VFC-HD BSP. Parameters and ports common to all BSPs are described in Section 8.3.1.

Generic parameters

name	type	default	description
`g_pcs16_bit`	boolean	`false`	Altera Arria V FPGAs provide the possibility to configure the PCS of the PHY as either 8bit or 16bit. The default is to use the 8bit PCS. Currently, 16bit PCS is not supported for Arria V.

Ports

name	dir	size	description
Clocks and resets			
`areset_n_i`	in	1	Reset input (active low, can be async)
`areset_edge_n_i`	in	1	[optional] Reset input edge sensitive (active rising-edge, can be async).
`clk_board_20m_i`	in	1	20MHz clock input from board
`clk_board_125m_i`	in	1	125MHz reference clock input from board
SPI interface to DACs			
`dac_sclk_o`	out	1	SPI SCLK, common to both DACs
`dac_din_o`	out	1	SPI MOSI, common to both DACs
`dac_ref_sync_n_o`	out	1	SPI $\overline{SS}$ for DAC controlling 125MHz oscillator

name	dir	size	description
dac_dmtd_sync_n_o	out	1	SPI $\overline{SS}$ for DAC controlling 20MHz oscillator
SFP interface			
sfp_tx_o	out	1	PHY TX
sfp_rx_i	in	1	PHY RX
sfp_det_i	in	1	Active high, asserted if all of the following are true: * SFP is detected (plugged in) * The part number has been successfully read
sfp_data_i	in	128	16 byte SFP vendor Part Number (ASCII encoded, first character byte in bits 127 downto 120)

9 Troubleshooting

My computer hangs on loading spec.ko or programming the FPGA.

This will occur when you try to load the driver or program the FPGA while your *spec-vuart* is running and trying to get messages from Virtual-UART's registers inside the WRPC. Please remember to quit *spec-vuart* before reloading the driver or programming the FPGA.

I want to synthesize WRPC but hdlmake says I don't have the tools.

If you have installed the synthesis tool (ISE for SPEC/SVEC or Quartus for VFC-HD), but you still see the message, please make sure you have correctly set up your environment. See sections 2.1.1 and 2.1.2 for the instructions.

WR PTP Core seems to work but the 1-PPS skew on the oscilloscope between WR Master and WR Slave is more than 1ns.

Check if the oscilloscope cables you use have the same delays. If not, you should either change the cables or take the delay difference into account in your measurements. If that's not the case, please make sure you have the correct calibration values loaded to both of your devices (see section 4.1). If you made your own synthesis of the WRPC, then the default calibration values are no longer valid and you need to perform the White Rabbit Calibration procedure[9].

10 Questions, reporting bugs

If you have found a bug, you have problems with the WR PTP Core or one of the tools used to build and run it, you can write to our mailing list ***white-rabbit-dev@ohwr.org***

[9] Use the latest version of the document available at: http://www.ohwr.org/documents/213

A WRPC Shell Commands

`calibration`	(legacy) tries to read t2/4 phase transition value from the Flash/EEPROM (in WR Master or GrandMaster mode), or executes the t24p calibration procedure and stores its result to the Flash/EEPROM (in WR Slave mode)
`config`	prints the dot-config file used to build this instance of WRPC and ppsi. It is an optional command, enabled at build time by `CONFIG_CMD_CONFIG`.
`delays`	
`delays <tx> <rx>`	gets or sets the constant delays in frame transmission, only available if `CONFIG_CMD_LL`. Delays are expressed in picoseconds.
`devmem <addr>`	
`devmem <addr> <value>`	reads or writes a 32-bit word from memory and/or FPGA registers. Only available if `CONFIG_CMD_LL` is set.
`diag`	diagnostics for auxiliary user registers in HDL. Only available if `CONFIG_AUX_DIAG`.
`diag ro <reg_no>`	reads register <reg_no> from read-only bank.
`diag rw <reg_no>`	reads register <reg_no> from read-write bank.
`diag w <reg_no>`	writes register <reg_no> to read-write bank.
`faketemp`	
`faketemp <T1> <T2> <T3>`	reads or sets the three fake temperatures, used to test the temperature/syslog mechanism. Available if `CONFIG_FAKE_TEMPERATURES` is set.
`gui`	starts GUI WRPC monitor
`help`	lists available commands in this instance of the WRPC
`init add <cmd>`	adds shell command at the end of the initialization script
`init boot`	executes the script stored in Flash/EEPROM (the same action is done automatically when WRPC starts after resetting LM32)
`init erase`	erases the initialization script in Flash/EEPROM
`init show`	prints all commands from the script stored in Flash/EEPROM
`ip get`	

`ip set <ip>`	reports or sets the IPv4 address of the WRPC (only available if `CONFIG_IP` is set at build time
`ltest`	
`ltest fake <nsecs>`	fakes delay to trigger latency failures (for testing).
`ltest <secs> <msecs>`	reads or sets the latency-test sending interval. See 5.4.1. Available if `CONFIG_LATENCY_PROBE` is set.
`ltest quiet`	disables sending latency messages to Syslog.
`ltest verbose`	enables sending latency messages to Syslog.
`mac getp`	reads the persistent MAC address stored in Flash/EEPROM
`mac get`	prints WRPC's MAC address
`mac setp <mac>`	stores the persistent MAC address in Flash/EEPROM
`mac set <mac>`	sets the MAC address of WRPC
`mode gm|master|slave`	(legacy) sets WRPC to operate as Grandmaster clock (requires external 10MHz and 1-PPS reference), Master or Slave. After setting the mode, `ptp start` must be re-issued
`pll cl <channel>`	checks if SoftPLL is locked for the channel
`pll gdac <index>`	gets dac's value
`pll gps <channel>`	gets current and target phase shift for the channel
`pll init <mode> <ref_channel> <align_pps>`	manually runs `spll_init()` function to initialize SoftPll
`pll sdac <index> <val>`	sets the dac
`pll sps <channel> <picoseconds>`	sets phase shift for the channel
`pll start <channel>`	starts SoftPLL for the channel
`pll stop <channel>`	stops SoftPLL for the channel
`ps`	prints the list of running tasks (processes) in the CPU. For each task you get the number of iterations, the maximum execution time (measured with the monotonic clock) and the CPU time consumed (using the RT clock) since boot or last reset of values
`ps reset`	zeroes the profiling information reported by the `ps` command
`ps max <msecs>`	starts printing all tasks executing longer than a given number of miliseconds. Additionally, it triggers printing messages if particular task runs longer than ever before. Passing "`0`" as a parameter stops the further printouts.

`ptp <e2e|p2p>`	selects PTP delay mechanism: end-to-end or peer-to-peer. If configured, you can set `p2p` mode. Alternatively you can use also aliases: `delay` (instead of `e2e`) or `pdelay` (instead of `p2p`).
`ptp gm|master|slave`	sets WRPC to operate as Grandmaster clock (requires external 10MHz and 1-PPS reference), Master or Slave. After setting the mode, `ptp start` must be re-issued
`ptp start`	starts WR PTP daemon
`ptp stop`	stops WR PTP daemon
`ptp <cmd> <cmd> ...`	you can concatenate several of the above subcommands in a single `ptp` command. With no arguments, the command reports the current values.
`refresh`	changes the update time period of the gui and stat commands. Default period is 1 second. If you set the period to 0, the log message is only generated one time.
`sdb`	prints devices connected to the Wishbone bus inside WRPC
`sdb fs <memtype> <baseadr> <param>`	creates SDBFS image under specified `<baseadr>` in selected storage depending on `<memtype>` (**0** - Flash, **1** - I2C EEPROM, **2** - 1-Wire EEPROM). The meaning of last parameter `<param>` depends on the type of selected storage. It is either the sector size in kilobytes (for Flash) or I2C chip address (for I2C EEPROM). Command `sdb` is available if `CONFIG_GENSDBFS` is set.
`sdb fs 0`	creates SDBFS image in Flash memory. Base address and sector size are taken from HDL Syscon registers for SPEC/SVEC boards. If you want to use it for custom board, base address and sector size must be specified as VHDL generic parameters of the WR PTP Core. Command `sdb` is available if `CONFIG_GENSDBFS` is set.
`sdb fse <memtype> <baseadr> <param>`	erases SDBFS image under specified `<baseadr>` from selected storage depending on `<memtype>` (**0** - Flash, **1** - I2C EEPROM, **2** - 1-Wire EEPROM). The meaning of last parameter `<param>` depends on the type of selected storage. It is either the sector size in kilobytes (for Flash) or I2C chip address (for I2C EEPROM). Command `sdb` is available if `CONFIG_GENSDBFS` is set.

`sdb fse 0`	erases SDBFS image from Flash memory. Base address and sector size are taken from HDL Syscon registers for SPEC/SVEC boards. If you want to use it for custom board, base address and sector size must be specified as VHDL generic parameters of the WR PTP Core. Command `sdb` is available if `CONFIG_GENSDBFS` is set.
`sfp add <PN> <deltaTx> <deltaRx> <alpha>`	stores calibration parameters for SFP to a file in Flash/EEPROM
`sfp erase`	erases the SFP database stored in the Flash/EEPROM
`sfp match`	prints the ID of a currently used SFP transceiver and tries to load the calibration parameters for it
`sfp show`	prints all SFP transceivers stored in database
`stat`	toggles reporting of loggable statistics. You can pass `on` or `off` as argument as an alternative to toggling
`stat bts`	prints bitslide value for established WR Link, needed by the calibration procedure
`syslog off`	
`syslog <ipaddr> <macaddr>`	disables or sets your *syslog* server. See 5.2. Available if `CONFIG_SYSLOG` is set.
`temp`	reports current temperatures.
`time`	prints current time from WRPC
`time raw`	prints current time in a raw format (seconds, nanoseconds)
`time setnsec <nsec>`	sets only nanoseconds of the WRPC time
`time setsec <sec>`	sets only seconds of the WRPC time (useful for setting time in GrandMaster mode, when nanoseconds counter is aligned to external 1-PPS and 10 MHz)
`time set <sec> <nsec>`	sets WRPC time
`verbose <digits>`	sets PPSi verbosity. See the PPSi manual about the meaning of the digits (hint: `verbose 1111` is a good first bet to see how the PTP system is working)
`ver`	prints which version of wrpc is running
`vlan`	
`vlan set <n>`	reports or sets the active vlan used for sending/receiving network frames. The command exists only if `CONFIG_VLAN` is set.
`vlan off`	disables vlans. Available if `CONFIG_VLAN` is set.

`w1`	lists onewire devices on the bus
`w1w <offset> <byte> [<byte> ...]`	
`w1r <offset> <len>`	if `CONFIG_W1` is set and a OneWire EEPROM exists, write and read data. For writing, `byte` values are decimal

B WRPC GUI elements

`TAI Time:`	Current state of device's local clock
`RX: / TX:`	Rx/Tx packets counters
`IPv4:`	IP address; also whether it is statically configured or acquired via BOOTP (and the status of BOOTP)
`mode:`	Operation mode of the WR PTP Core - `<WR Master, WR Slave>`
`<Locked, NoLock>`	SoftPLL lock state
`<Calibrated, Uncalibrated>`	Status of PHY calibration; not used anymore
`PTP status:`	Current state of PTP state machine
`Servo state:`	Current state of WR servo state machine - `<Uninitialized, SYNC_SEC, SYNC_NSEC, SYNC_PHASE, TRACK_PHASE>`
`Phase tracking:`	Is phase tracking enabled when WR Slave is synchronized to WR Master - `<ON, OFF>`
`Aux clock <N> status:`	Statuses of AUX clocks; one status line per available AUX clock; can contain <enabled> and <locked>
`Round-trip time (mu):`	Round-trip delay in picoseconds $(delay_{MM})(delay_{MM})$
`Master-slave delay:`	Estimated one-way (master to slave) link delay $(delay_{MS})$
`Master PHY delays:`	Transmission/reception delays of WR Master's hardware $(\Delta_{TXM}, \Delta_{RXM})$
`Slave PHY delays:`	Transmission/reception delays of WR Slave's hardware $(\Delta_{TXS}, \Delta_{RXS})$
`Total link asymmetry:`	WR link asymmetry calculated as $delay_{MM} - 2 \cdot delay_{MS}$
`Cable rtt delay:`	Round-trip fiber latency
`Clock offset:`	Slave to Master offset calculated by PTP daemon $(offset_{MS})$
`Phase setpoint:`	Current Slave's clock phase shift value
`Skew:`	The difference between current and previous estimated one-way link delay
`Update counter:`	The value of a counter incremented every time the WR servo is updated

C Other ways to write SDBFS image to your Flash memory

C.1 Writing SDBFS image through PCIe bus

To write SDBFS filesystem image to the Flash memory of a hosted SPEC card, you can use the `flash-write` tool available in the *spec-sw* drivers package.

First, please download the SDBFS image from *ohwr.org*:

```
$ wget http://www.ohwr.org/attachments/download/4060/sdbfs-flash.bin
```

It contains all the files required by the core. They are empty, but have to exist in the SDBFS structure to be filled later from the WR PTP Core shell or SNMP.

Before calling the tool, you need to have SPEC drivers loaded in your system:

```
$ cd <your_location>/spec-sw
$ sudo insmod fmc-bus/kernel/fmc.ko
$ sudo insmod kernel/spec.ko
```

To write the filesystem image to flash, please execute the following command:

```
$ sudo tools/flash-write -c 0x0 0 1507712 < <your_location>/sdbfs-flash.bin
```

Note: If you have more than one SPEC board in your computer, you can use `-b` parameter which takes the PCI bus address of the card you want to program. This can be found in the list of the PCI devices installed in the system (`lspci`).

C.2 Writing SDBFS image in standalone configuration

If you use SPEC board in a host-less environment, or you use custom hardware and SPEC drivers/tools cannot be used, there is still a possibility of writing SDBFS through Xilinx JTAG.

In the case when you want to run on the the SPEC a reference bitstream provided with a stable WRPC release, you can simply program your Flash with *spec_top.mcs* provided with the release binaries using for example Xilinx ISE Impact tool. This *mcs* file already includes both SDBFS image and FPGA bitstream.

In the case when you want to run a custom gateware or you have a custom hardware, you can download a standalone SDBFS image:

```
$ wget http://www.ohwr.org/attachments/download/4558/sdbfs-standalone-160812.bin
```

and generate a custom **.mcs* file with your own FPGA bitstream. You should use the following layout:

`0x000000`	your FPGA bitstream
`0x170000`	SDBFS image

For example, to generate the **.mcs* file for M25P32 Flash on SPEC, the following *promgen* parameters should be used:

```
promgen -w -spi -p mcs -c FF -s 32768 -u 0 <your_bitstream>.bit \
-bd sdbfs-standalone-160812.bin start 0x170000 -o output.mcs
```

After that, you can use the Xilinx JTAG cable and Xilinx ISE Impact tool to write your *output.mcs* file to the Flash memory.

D wrpc-dump with Older WRPC Binaries

The tool has another working mode, that you can use with older *wrpc* builds, where the special table is missing (but please be aware that the data structures may have changed slightly). In this mode, you provide the address of the data structure and its name. It supports the names *pll*, *fifo* (the circular log described above), *ppg*, *ppi*, *servo_state*, *ds* and *stats*. The *ds* name refers to the PTP data sets, and for this the pointer needed is `ppg` (ppsi global data). This is an example:

```
$ ./tools/wrpc-dump dump-file 00015798 servo_state

servo_state at 0x15798
       if_name:                ""
       flags:                  1
       state:                  4
       delta_tx_m:             10
       delta_rx_m:             174410
       delta_tx_s:             0
       delta_rx_s:             3200
       fiber_fix_alpha:        73622176
       clock_period_ps:        8000
       t1:                     correct 1: 1448628309.390213200:0000
       t2:                     correct 1: 1448628309.390213520:2921
       t3:                     correct 1: 1448628310.419072616:0000
       t4:                     correct 1: 1448628310.419073104:6041
[...]
```

The address is specified as a hex number, and can be retrieved running "`lm32-elf-nm wrc.elf`".

Please note that the LM32 soft-core processor inside the WRPC has a different endianness than your host machine, thus a special endian conversion is needed. With current *wrpc* it is autodetected, but if you dump an older binary you'll need to set the `WRPC_SPEC` environment variable (to any value you like) to properly access the PCI memory or a dump taken from PCI:

```
$ export WRPC_SPEC=y
```

or

```
$ WRPC_SPEC=y ./tools/wrpc-dump dump-file 00015798 servo_state
```

This is not needed if the dump is retrieved using Etherbone.

E Adding new objects to the SNMP

The *Mini SNMP responder* can be easily expanded to export new objects. Values of new objects can come from WRPC's variables or other HDL modules as long as there is a proper interface to the WRPC to read these values.

This section contains the instruction on how to export new objects with the given variables' content.

The *Mini SNMP responder* internally divides all OIDs into two parts. The first part is called *limb*. The *limb* part of the incoming OID is matched by a function `snmp_respond`, with the defined *limb* parts of OIDs in the structure `oid_limb_array`. When the *limb* part is matched then the corresponding function from the structure `oid_limb_array` is called to try to match the second part of OID (the *twig* part).

The example below adds to the *Mini SNMP responder* an `int32_t` variable (`example_i32var`) with OID `1.3.6.1.4.1.96.102.1.1.0` and a string (`example_string`) with OID `1.3.6.1.4.1.96.102.1.2.0`. Before assigning new OIDs in your projects please contact the maintainer of `wrpc-sw` repo to avoid conflicts.

- First declare `example_i32var` and `example_string`:

```
static int32_t example_i32var;
static char example_string[] = "test string";
```

- Define the *limb* part of the OID:

```
static uint8_t oid_wrpcExampleGroup[] = {0x2B,6,1,4,1,96,101,99};
```

- Define the *twig* part of the OID:

```
static uint8_t oid_wrpcExampleV1[] = {1,0};
static uint8_t oid_wrpcExampleV2[] = {2,0};
```

- Add a group definition to the `struct snmp_oid_limb oid_limb_array`. Please note that this structure has to be sorted by ascending OIDs.

```
OID_LIMB_FIELD(oid_wrpcExampleGroup, func_group, oid_array_wrpcExampleGroup),
```

  The macro `OID_LIMB_FIELD` takes the following arguments:

  - `oid_wrpcExampleGroup` – an array with the *limb* part of the OID
  - `func_group` – a function to be called when the *limb* part of the OID is matched; this function will try to match the *twig* part of the OID within a table or a group.
  - `oid_array_wrpcExampleGroup` – an array of *twig* parts of OIDs

- Declare a previously used `oid_wrpcExampleGroup`. Please note that this structure has to be sorted by ascending *twig* part of OIDs.

```
static struct snmp_oid oid_array_wrpcExampleGroup[] = {
  OID_FIELD_VAR(oid_wrpcExampleV1, get_p, set_p, ASN_INTEGER, &example_i32var),
  OID_FIELD_VAR(oid_wrpcExampleV2, get_p, set_p, ASN_OCTET_STR, &example_string),
  { 0, }
};
```

  The macro `OID_FIELD_VAR` takes the following arguments:

  - `oid_wrpcExampleV1` – an array with *twig* part of the OID
  - `get_p` (or `get_pp`) – a function to be called when *twig* part of the OID is matched for SNMP GET requests;
  - `set_p` (or `set_pp`) – a function to be called when a *twig* part of the OID is matched for SNMP SET requests; if no SET functionality is planned, please use NULL
  - `ASN_INTEGER, ASN_OCTET_STR` – type of the OID; please check the source for other possible types
  - `&example_i32var, &example_string` – addresses to the data in memory

In the case when the address of variable is not known at boot, it is possible to define a pointer to the variable which will be initialized (e.g. in the `snmp_init` the function) at the boot time. In that case function `get_pp` (`set_pp`) has to be used instead of `get_p` (`set_p`). For variables that are a part of a structure and have to be accessed via a pointer, a macro `OID_FIELD_STRUCT` is available.
For more complex extraction of variables or run-time value corrections, it is possible to use a custom *get* function. It is possible to pass a constant number to the custom function instead of an address. For example:

```
OID_FIELD_VAR(oid_wrpcPtpServoUpdateTime, get_servo, NO_SET, ASN_COUNTER64, \
              SERVO_UPDATE_TIME),
```

Perform a `snmpwalk` to get new OIDs:

```
$ snmpwalk -On $SNMP_OPT 1.3.6.1.4.1.96.102.1
.1.3.6.1.4.1.96.102.1.1.0 = INTEGER: 123432
.1.3.6.1.4.1.96.102.1.2.0 = STRING: "test string"
End of MIB
```

Trying to set too long string into the `example_string` results in an error:

```
$ snmpset -On $SNMP_OPT 1.3.6.1.4.1.96.102.1.2.0 s "new long string"
Error in packet.
Reason: (badValue) The value given has the wrong type or length.
Failed object: .1.3.6.1.4.1.96.102.1.2.0
```

A short enough (not longer than defined `"test string"`) value succeeds:

```
$ snmpset -On $SNMP_OPT 1.3.6.1.4.1.96.102.1.2.0 s "new value12"
.1.3.6.1.4.1.96.102.1.2.0 = STRING: "new value12"
```

Set 999 to the `example_i32var`:

```
$ snmpset -On $SNMP_OPT 1.3.6.1.4.1.96.102.1.1.0 i 999
.1.3.6.1.4.1.96.102.1.1.0 = INTEGER: 999
```

Perform `snmpwalk` to verify changes:

```
$ snmpwalk -On $SNMP_OPT 1.3.6.1.4.1.96.102.1
.1.3.6.1.4.1.96.102.1.1.0 = INTEGER: 999
.1.3.6.1.4.1.96.102.1.2.0 = STRING: "new value12"
End of MIB
```

E.1 Mini SNMP responder's tests

In the `wrpc-sw` repo, automatic tests are available to validate the *Mini SNMP responder* implementation. These tests are placed in the `test/snmp` directory. They use the `bats` framework (https://github.com/sstephenson/bats).

To run these tests, please go to `test/snmp` directory, then set `TARGET_IP` environment varaible to the IP of your target board, then type `make`. For example:

```
$ TARGET_IP=192.168.1.20 make
bats/libexec/bats snmp_tests_*.bats
  Host up 192.168.1.20
  Check the presence of snmpget
  Check the presence of snmpgetnext
  Check the presence of snmpwalk
  Check the presence of snmpset
```

```
  snmpget existing oid 1.3.6.1.4.1.96.101.1.1.1.0
[...]
  snmpwalk 1.3.6.1.4.1.95
  snmpwalk 1.3.6 count all OIDs
100 tests, 0 failures
```

On the left of each test there will be a tick symbol shown or an `x` depending of the test's result (not included in the example above).

Be aware that it might be necessary to clone the `bats` repo first. `make` command will inform whether this is needed.

In the case when the number of OIDs changes please correct variable `TOTAL_NUM_OIDS` in file `snmp_test_config.bash`.

These tests have to be executed after any changes are made to the *Mini SNMP responder*.

F Wishbone Memory Maps

This appendix provides a register map of Wishbone modules that are exposed to external application-s/HDL core. In the table below you can see the list of modules available on the External Wishbone interface of the WR PTP Core. Please bare in mind that WR Streamers Diagnostics module is available only when `STREAMERS` external fabric mode is selected.

module name	base address
WR Core Diagnostics	0x20800
WR Streamers Diagnostics	0x20700

F.1 WR Core Diagnostics

[version 0x00000001]
Diagnostics information accessible via WR

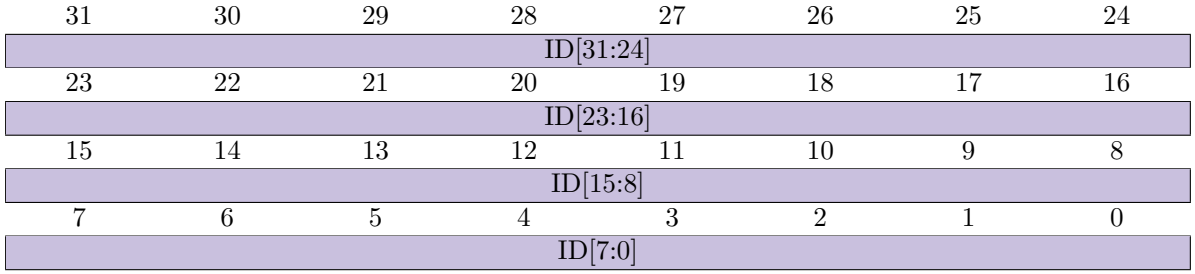
F.1.1 Memory map summary

SW Offset	Type	Name	HW prefix	C prefix
0x0	REG	Version register	wrc_diags_ver	VER
0x4	REG	Ctrl	wrc_diags_ctrl	CTRL
0x8	REG	WRPC Diag: servo status	wrc_diags_wdiag_sstat	WDIAG_SSTAT
0xc	REG	WRPC Diag: Port status	wrc_diags_wdiag_pstat	WDIAG_PSTAT
0x10	REG	WRPC Diag: PTP state	wrc_diags_wdiag_ptpstat	WDIAG_PTPSTAT
0x14	REG	WRPC Diag: AUX state	wrc_diags_wdiag_astat	WDIAG_ASTAT
0x18	REG	WRPC Diag: Tx PTP Frame cnts	wrc_diags_wdiag_txfcnt	WDIAG_TXFCNT
0x1c	REG	WRPC Diag: Rx PTP Frame cnts	wrc_diags_wdiag_rxfcnt	WDIAG_RXFCNT
0x20	REG	WRPC Diag:local time [msb of s]	wrc_diags_wdiag_sec_msb	WDIAG_SEC_MSB
0x24	REG	WRPC Diag: local time [lsb of s]	wrc_diags_wdiag_sec_lsb	WDIAG_SEC_LSB
0x28	REG	WRPC Diag: local time [ns]	wrc_diags_wdiag_ns	WDIAG_NS
0x2c	REG	WRPC Diag: Round trip (mu) [msb of ps]	wrc_diags_wdiag_mu_msb	WDIAG_MU_MSB
0x30	REG	WRPC Diag: Round trip (mu) [lsb of ps]	wrc_diags_wdiag_mu_lsb	WDIAG_MU_LSB
0x34	REG	WRPC Diag: Master-slave delay (dms) [msb of ps]	wrc_diags_wdiag_dms_msb	WDIAG_DMS_MSB
0x38	REG	WRPC Diag: Master-slave delay (dms) [lsb of ps]	wrc_diags_wdiag_dms_lsb	WDIAG_DMS_LSB
0x3c	REG	WRPC Diag: Total link asymmetry [ps]	wrc_diags_wdiag_asym	WDIAG_ASYM
0x40	REG	WRPC Diag: Clock offset (cko) [ps]	wrc_diags_wdiag_cko	WDIAG_CKO
0x44	REG	WRPC Diag: Phase setpoint (setp) [ps]	wrc_diags_wdiag_setp	WDIAG_SETP
0x48	REG	WRPC Diag: Update counter (ucnt)	wrc_diags_wdiag_ucnt	WDIAG_UCNT
0x4c	REG	WRPC Diag: Board temperature [C degree]	wrc_diags_wdiag_temp	WDIAG_TEMP

F.1.2 Register description

Version register

HW prefix: wrc_diags_ver
HW address: 0x0
SW prefix: VER
SW offset: 0x0



- **ID** [*read/write*]: Version identifier
  Version identifier for the peripheral

Ctrl

HW prefix: wrc_diags_ctrl
HW address: 0x1
SW prefix: CTRL
SW offset: 0x4

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	DATA_SNAPSHOT
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	DATA_VALID

- **DATA_VALID** [*read-only*]: WR DIAG data valid
  0: valid
  1:transcient
- **DATA_SNAPSHOT** [*read/write*]: WR DIAG data snapshot
  1: snapshot data (data in registers will not change aveter VALID becomes true)

WRPC Diag: servo status

HW prefix: wrc_diags_wdiag_sstat
HW address: 0x2
SW prefix: WDIAG_SSTAT
SW offset: 0x8

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	SERVOSTATE[3:0]			
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	WR_MODE

- **WR_MODE** [*read-only*]: WR valid
  0: not valid
  1:valid
- **SERVOSTATE** [*read-only*]: Servo State
  0: Uninitialized
  1: SYNC_NSEC
  2: SYNC_TAI
  3: SYNC_PHASE
  4: TRACK_PHASE
  5: WAIT_OFFSET_STABLE

WRPC Diag: Port status

HW prefix: wrc_diags_wdiag_pstat
HW address: 0x3
SW prefix: WDIAG_PSTAT
SW offset: 0xc

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	-	-	LOCKED	LINK

- **LINK** [*read-only*]: Link Status
  0: link down
  1: link up
- **LOCKED** [*read-only*]: PLL Locked
  0: not locked
  1: locked

WRPC Diag: PTP state

HW prefix: wrc_diags_wdiag_ptpstat
HW address: 0x4
SW prefix: WDIAG_PTPSTAT
SW offset: 0x10

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
PTPSTATE[7:0]							

- **PTPSTATE** [*read-only*]: PTP State
  0: NONE
  1: PPS_INITIALIZING
  2: PPS_FAULTY
  3: disabled
  4: PPS_LISTENING
  5: PPS_PRE_MASTER
  6: PPS_MASTER
  7: PPS_PASSIVE
  8: PPS_UNCALIBRATED
  9: PPS_SLAVE
  100-116: WR STATES
  see: ppsi/proto-ext-whiterabbit/wr-constants.h
  ppsi/include/ppsi/ieee1588_types.h

WRPC Diag: AUX state

HW prefix: wrc_diags_wdiag_astat
HW address: 0x5
SW prefix: WDIAG_ASTAT
SW offset: 0x14

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
AUX[7:0]							

- **AUX** [*read-only*]: AUX channel
  A vector of bits, one bit per channel
  0: not valid
  1:valid

WRPC Diag: Tx PTP Frame cnts

HW prefix: wrc_diags_wdiag_txfcnt
HW address: 0x6
SW prefix: WDIAG_TXFCNT
SW offset: 0x18

Number of transmitted PTP Frames

31	30	29	28	27	26	25	24
WDIAG_TXFCNT[31:24]							
23	22	21	20	19	18	17	16
WDIAG_TXFCNT[23:16]							
15	14	13	12	11	10	9	8
WDIAG_TXFCNT[15:8]							
7	6	5	4	3	2	1	0
WDIAG_TXFCNT[7:0]							

- **WDIAG_TXFCNT** [*read-only*]: Data

WRPC Diag: Rx PTP Frame cnts

HW prefix: wrc_diags_wdiag_rxfcnt
HW address: 0x7
SW prefix: WDIAG_RXFCNT
SW offset: 0x1c

Number of received PTP Frames

31	30	29	28	27	26	25	24
WDIAG_RXFCNT[31:24]							
23	22	21	20	19	18	17	16
WDIAG_RXFCNT[23:16]							
15	14	13	12	11	10	9	8
WDIAG_RXFCNT[15:8]							
7	6	5	4	3	2	1	0
WDIAG_RXFCNT[7:0]							

- **WDIAG_RXFCNT** [*read-only*]: Data

WRPC Diag:local time [msb of s]

HW prefix: wrc_diags_wdiag_sec_msb
HW address: 0x8
SW prefix: WDIAG_SEC_MSB
SW offset: 0x20

Local Time expressed in seconds since epoch (TAI)

31	30	29	28	27	26	25	24
WDIAG_SEC_MSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_SEC_MSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_SEC_MSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_SEC_MSB[7:0]							

- **WDIAG_SEC_MSB** [*read-only*]: Data

WRPC Diag: local time [lsb of s]

HW prefix: wrc_diags_wdiag_sec_lsb
HW address: 0x9
SW prefix: WDIAG_SEC_LSB
SW offset: 0x24

Local Time expressed in seconds since epoch (TAI)

31	30	29	28	27	26	25	24
WDIAG_SEC_LSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_SEC_LSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_SEC_LSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_SEC_LSB[7:0]							

- **WDIAG_SEC_LSB** [*read-only*]: Data

WRPC Diag: local time [ns]

HW prefix: wrc_diags_wdiag_ns
HW address: 0xa
SW prefix: WDIAG_NS
SW offset: 0x28

Nanoseconds part of the Local Time expressed in seconds since epoch (TAI)

31	30	29	28	27	26	25	24
WDIAG_NS[31:24]							
23	22	21	20	19	18	17	16
WDIAG_NS[23:16]							
15	14	13	12	11	10	9	8
WDIAG_NS[15:8]							
7	6	5	4	3	2	1	0
WDIAG_NS[7:0]							

- **WDIAG_NS** [*read-only*]: Data

WRPC Diag: Round trip (mu) [msb of ps]

HW prefix: wrc_diags_wdiag_mu_msb
HW address: 0xb
SW prefix: WDIAG_MU_MSB
SW offset: 0x2c

31	30	29	28	27	26	25	24
WDIAG_MU_MSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_MU_MSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_MU_MSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_MU_MSB[7:0]							

- **WDIAG_MU_MSB** [*read-only*]: Data

WRPC Diag: Round trip (mu) [lsb of ps]

HW prefix: wrc_diags_wdiag_mu_lsb
HW address: 0xc
SW prefix: WDIAG_MU_LSB
SW offset: 0x30

31	30	29	28	27	26	25	24
WDIAG_MU_LSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_MU_LSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_MU_LSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_MU_LSB[7:0]							

- **WDIAG_MU_LSB** [*read-only*]: Data

WRPC Diag: Master-slave delay (dms) [msb of ps]

HW prefix: wrc_diags_wdiag_dms_msb
HW address: 0xd
SW prefix: WDIAG_DMS_MSB
SW offset: 0x34

31	30	29	28	27	26	25	24
WDIAG_DMS_MSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_DMS_MSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_DMS_MSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_DMS_MSB[7:0]							

- **WDIAG_DMS_MSB** [*read-only*]: Data

WRPC Diag: Master-slave delay (dms) [lsb of ps]

HW prefix: wrc_diags_wdiag_dms_lsb
HW address: 0xe
SW prefix: WDIAG_DMS_LSB
SW offset: 0x38

31	30	29	28	27	26	25	24
WDIAG_DMS_LSB[31:24]							
23	22	21	20	19	18	17	16
WDIAG_DMS_LSB[23:16]							
15	14	13	12	11	10	9	8
WDIAG_DMS_LSB[15:8]							
7	6	5	4	3	2	1	0
WDIAG_DMS_LSB[7:0]							

- **WDIAG_DMS_LSB** [*read-only*]: Data

WRPC Diag: Total link asymmetry [ps]

HW prefix: wrc_diags_wdiag_asym
HW address: 0xf
SW prefix: WDIAG_ASYM
SW offset: 0x3c

31	30	29	28	27	26	25	24
WDIAG_ASYM[31:24]							
23	22	21	20	19	18	17	16
WDIAG_ASYM[23:16]							
15	14	13	12	11	10	9	8
WDIAG_ASYM[15:8]							
7	6	5	4	3	2	1	0
WDIAG_ASYM[7:0]							

- **WDIAG_ASYM** [*read-only*]: Data

WRPC Diag: Clock offset (cko) [ps]

HW prefix: wrc_diags_wdiag_cko
HW address: 0x10
SW prefix: WDIAG_CKO
SW offset: 0x40

31	30	29	28	27	26	25	24
WDIAG_CKO[31:24]							
23	22	21	20	19	18	17	16
WDIAG_CKO[23:16]							
15	14	13	12	11	10	9	8
WDIAG_CKO[15:8]							
7	6	5	4	3	2	1	0
WDIAG_CKO[7:0]							

- **WDIAG_CKO** [*read-only*]: Data

WRPC Diag: Phase setpoint (setp) [ps]

HW prefix: wrc_diags_wdiag_setp
HW address: 0x11
SW prefix: WDIAG_SETP
SW offset: 0x44

31	30	29	28	27	26	25	24
WDIAG_SETP[31:24]							
23	22	21	20	19	18	17	16
WDIAG_SETP[23:16]							
15	14	13	12	11	10	9	8
WDIAG_SETP[15:8]							
7	6	5	4	3	2	1	0
WDIAG_SETP[7:0]							

- **WDIAG_SETP** [*read-only*]: Data

WRPC Diag: Update counter (ucnt)

HW prefix: wrc_diags_wdiag_ucnt
HW address: 0x12
SW prefix: WDIAG_UCNT
SW offset: 0x48

31	30	29	28	27	26	25	24
WDIAG_UCNT[31:24]							
23	22	21	20	19	18	17	16
WDIAG_UCNT[23:16]							
15	14	13	12	11	10	9	8
WDIAG_UCNT[15:8]							
7	6	5	4	3	2	1	0
WDIAG_UCNT[7:0]							

- **WDIAG_UCNT** [*read-only*]: Data

WRPC Diag: Board temperature [C degree]

HW prefix: wrc_diags_wdiag_temp
HW address: 0x13
SW prefix: WDIAG_TEMP
SW offset: 0x4c

31	30	29	28	27	26	25	24
WDIAG_TEMP[31:24]							
23	22	21	20	19	18	17	16
WDIAG_TEMP[23:16]							
15	14	13	12	11	10	9	8
WDIAG_TEMP[15:8]							
7	6	5	4	3	2	1	0
WDIAG_TEMP[7:0]							

- **WDIAG_TEMP** [*read-only*]: Data

F.2 WR Streamers, status and debug

[version 0x00000001]

This WB registers allow to diagnose transmission and reception of
data using WR streamers.
In particular, these registers provide access to streamer's
statistics that can be also access from SNMP, if supported.

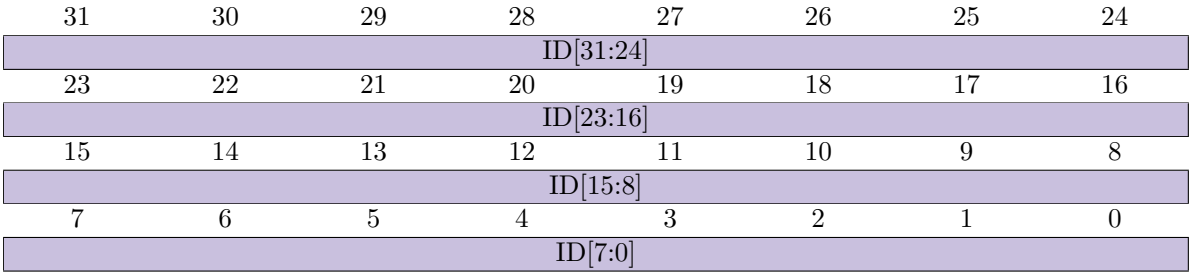
F.2.1 Memory map summary

SW Offset	Type	Name	HW prefix	C prefix
0x0	REG	Version register	wr_streamers_ver	VER
0x4	REG	Statistics status and ctrl register	wr_streamers_sscr1	SSCR1
0x8	REG	Statistics status and ctrl register	wr_streamers_sscr2	SSCR2
0xc	REG	Statistics status and ctrl register	wr_streamers_sscr3	SSCR3
0x10	REG	Rx statistics	wr_streamers_rx_stat0	RX_STAT0
0x14	REG	Rx statistics	wr_streamers_rx_stat1	RX_STAT1
0x18	REG	Tx statistics	wr_streamers_tx_stat2	TX_STAT2
0x1c	REG	Tx statistics	wr_streamers_tx_stat3	TX_STAT3
0x20	REG	Rx statistics	wr_streamers_rx_stat4	RX_STAT4
0x24	REG	Rx statistics	wr_streamers_rx_stat5	RX_STAT5
0x28	REG	Rx statistics	wr_streamers_rx_stat6	RX_STAT6
0x2c	REG	Rx statistics	wr_streamers_rx_stat7	RX_STAT7
0x30	REG	Rx statistics	wr_streamers_rx_stat8	RX_STAT8
0x34	REG	Rx statistics	wr_streamers_rx_stat9	RX_STAT9
0x38	REG	Rx statistics	wr_streamers_rx_stat10	RX_STAT10
0x3c	REG	Rx statistics	wr_streamers_rx_stat11	RX_STAT11
0x40	REG	Rx statistics	wr_streamers_rx_stat12	RX_STAT12
0x44	REG	Rx statistics	wr_streamers_rx_stat13	RX_STAT13
0x48	REG	Tx Config Reg 0	wr_streamers_tx_cfg0	TX_CFG0
0x4c	REG	Tx Config Reg 1	wr_streamers_tx_cfg1	TX_CFG1
0x50	REG	Tx Config Reg 2	wr_streamers_tx_cfg2	TX_CFG2
0x54	REG	Tx Config Reg 3	wr_streamers_tx_cfg3	TX_CFG3
0x58	REG	Tx Config Reg 4	wr_streamers_tx_cfg4	TX_CFG4
0x5c	REG	Tx Config Reg 4	wr_streamers_tx_cfg5	TX_CFG5
0x60	REG	Rx Config Reg 0	wr_streamers_rx_cfg0	RX_CFG0
0x64	REG	Rx Config Reg 1	wr_streamers_rx_cfg1	RX_CFG1
0x68	REG	Rx Config Reg 2	wr_streamers_rx_cfg2	RX_CFG2
0x6c	REG	Rx Config Reg 3	wr_streamers_rx_cfg3	RX_CFG3
0x70	REG	Rx Config Reg 4	wr_streamers_rx_cfg4	RX_CFG4
0x74	REG	Rx Config Reg 5	wr_streamers_rx_cfg5	RX_CFG5
0x78	REG	TxRx Config Override	wr_streamers_cfg	CFG
0x7c	REG	DBG Control register	wr_streamers_dbg_ctrl	DBG_CTRL
0x80	REG	DBG Data	wr_streamers_dbg_data	DBG_DATA
0x84	REG	Test value	wr_streamers_dummy	DUMMY

F.2.2 Register description

Version register

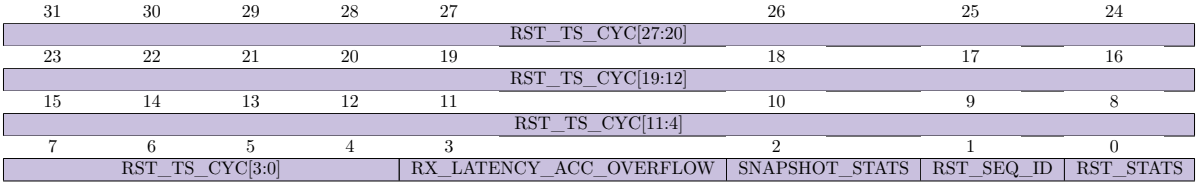
HW prefix: wr_streamers_ver
HW address: 0x0
SW prefix: VER
SW offset: 0x0



- **ID** [*read/write*]: Version identifier
  Version identifier for the peripheral

Statistics status and ctrl register

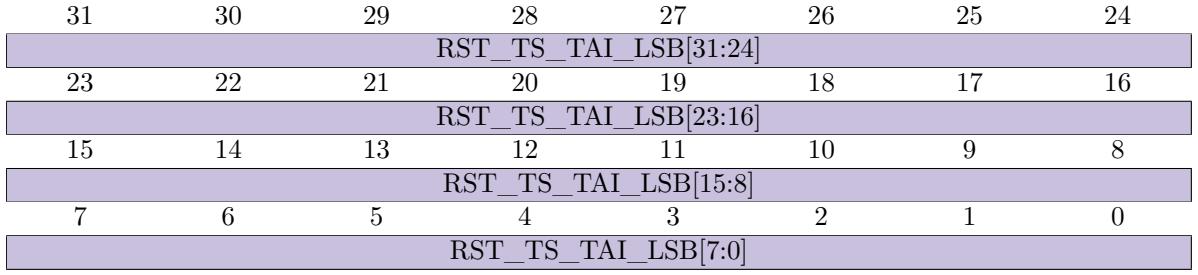
HW prefix: wr_streamers_sscr1
HW address: 0x1
SW prefix: SSCR1
SW offset: 0x4



- **RST_STATS** [*write-only*]: Reset statistics
  Writing 1 reset counters, latency acc/max/min. This reset is timestamped
- **RST_SEQ_ID** [*write-only*]: Reset tx seq id
  Writing 1 reset sequence ID of transmitted frames
- **SNAPSHOT_STATS** [*read/write*]: Snapshot statistics
  Writing 1 snapshots statistics for reading, it means that all the counters are copied at the same instant to registers and this registers can be read via wishbone/snmp while the counters are still running in the background. this allows to read coherent data
- **RX_LATENCY_ACC_OVERFLOW** [*read-only*]: Latency accumulator overflow
  Latency accumulator overflow - the lateny accumulator value is invalid
- **RST_TS_CYC** [*read-only*]: Reset timestamp cycles
  Timestamp of the last reset of stats (RST_STAT) – count of clock cycles

Statistics status and ctrl register

HW prefix: wr_streamers_sscr2
HW address: 0x2
SW prefix: SSCR2
SW offset: 0x8



- **RST_TS_TAI_LSB** [*read-only*]: Reset timestamp 32 LSB of TAI
  Timestamp of the last reset of stats (RST_STAT) – LSB 32 bits of TAI

Statistics status and ctrl register

HW prefix: wr_streamers_sscr3
HW address: 0x3
SW prefix: SSCR3
SW offset: 0xc

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
RST_TS_TAI_MSB[7:0]							

- **RST_TS_TAI_MSB** [*read-only*]: Reset timestamp 8 MSB of TAI
  Timestamp of the last reset of stats (RST_STAT) – MSB 8 bits of TAI

Rx statistics

HW prefix: wr_streamers_rx_stat0
HW address: 0x4
SW prefix: RX_STAT0
SW offset: 0x10

31	30	29	28	27	26	25	24
-	-	-	-	RX_LATENCY_MAX[27:24]			
23	22	21	20	19	18	17	16
RX_LATENCY_MAX[23:16]							
15	14	13	12	11	10	9	8
RX_LATENCY_MAX[15:8]							
7	6	5	4	3	2	1	0
RX_LATENCY_MAX[7:0]							

- **RX_LATENCY_MAX** [*read-only*]: WR Streamer frame latency
  Maximum latency of received frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat1
HW address: 0x5
SW prefix: RX_STAT1
SW offset: 0x14

31	30	29	28	27	26	25	24
-	-	-	-	RX_LATENCY_MIN[27:24]			
23	22	21	20	19	18	17	16
RX_LATENCY_MIN[23:16]							
15	14	13	12	11	10	9	8
RX_LATENCY_MIN[15:8]							
7	6	5	4	3	2	1	0
RX_LATENCY_MIN[7:0]							

- **RX_LATENCY_MIN** [*read-only*]: WR Streamer frame latency
  Minimum latency of received frames since reset

Tx statistics

HW prefix: wr_streamers_tx_stat2
HW address: 0x6
SW prefix: TX_STAT2
SW offset: 0x18

31	30	29	28	27	26	25	24
TX_SENT_CNT_LSB[31:24]							
23	22	21	20	19	18	17	16
TX_SENT_CNT_LSB[23:16]							
15	14	13	12	11	10	9	8
TX_SENT_CNT_LSB[15:8]							
7	6	5	4	3	2	1	0
TX_SENT_CNT_LSB[7:0]							

- **TX_SENT_CNT_LSB** [*read-only*]: WR Streamer frame sent count (LSB)
  Number of sent wr streamer frames since reset

Tx statistics

HW prefix: wr_streamers_tx_stat3
HW address: 0x7
SW prefix: TX_STAT3
SW offset: 0x1c

31	30	29	28	27	26	25	24
TX_SENT_CNT_MSB[31:24]							
23	22	21	20	19	18	17	16
TX_SENT_CNT_MSB[23:16]							
15	14	13	12	11	10	9	8
TX_SENT_CNT_MSB[15:8]							
7	6	5	4	3	2	1	0
TX_SENT_CNT_MSB[7:0]							

- **TX_SENT_CNT_MSB** [*read-only*]: WR Streamer frame sent count (MSB)
  Number of sent wr streamer frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat4
HW address: 0x8
SW prefix: RX_STAT4
SW offset: 0x20

31	30	29	28	27	26	25	24
RX_RCVD_CNT_LSB[31:24]							
23	22	21	20	19	18	17	16
RX_RCVD_CNT_LSB[23:16]							
15	14	13	12	11	10	9	8
RX_RCVD_CNT_LSB[15:8]							
7	6	5	4	3	2	1	0
RX_RCVD_CNT_LSB[7:0]							

- **RX_RCVD_CNT_LSB** [*read-only*]: WR Streamer frame received count (LSB)
  Number of received wr streamer frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat5
HW address: 0x9
SW prefix: RX_STAT5
SW offset: 0x24

31	30	29	28	27	26	25	24
RX_RCVD_CNT_MSB[31:24]							
23	22	21	20	19	18	17	16
RX_RCVD_CNT_MSB[23:16]							
15	14	13	12	11	10	9	8
RX_RCVD_CNT_MSB[15:8]							
7	6	5	4	3	2	1	0
RX_RCVD_CNT_MSB[7:0]							

- **RX_RCVD_CNT_MSB** [*read-only*]: WR Streamer frame received count (MSB)
  Number of received wr streamer frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat6
HW address: 0xa
SW prefix: RX_STAT6
SW offset: 0x28

31	30	29	28	27	26	25	24
RX_LOSS_CNT_LSB[31:24]							
23	22	21	20	19	18	17	16
RX_LOSS_CNT_LSB[23:16]							
15	14	13	12	11	10	9	8
RX_LOSS_CNT_LSB[15:8]							
7	6	5	4	3	2	1	0
RX_LOSS_CNT_LSB[7:0]							

- **RX_LOSS_CNT_LSB** [*read-only*]: WR Streamer frame loss count (LSB)
  Number of lost wr streamer frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat7
HW address: 0xb
SW prefix: RX_STAT7
SW offset: 0x2c

31	30	29	28	27	26	25	24
RX_LOSS_CNT_MSB[31:24]							
23	22	21	20	19	18	17	16
RX_LOSS_CNT_MSB[23:16]							
15	14	13	12	11	10	9	8
RX_LOSS_CNT_MSB[15:8]							
7	6	5	4	3	2	1	0
RX_LOSS_CNT_MSB[7:0]							

- **RX_LOSS_CNT_MSB** [*read-only*]: WR Streamer frame loss count (MSB)
  Number of lost wr streamer frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat8
HW address: 0xc
SW prefix: RX_STAT8
SW offset: 0x30

31	30	29	28	27	26	25	24
RX_LOST_BLOCK_CNT_LSB[31:24]							
23	22	21	20	19	18	17	16
RX_LOST_BLOCK_CNT_LSB[23:16]							
15	14	13	12	11	10	9	8
RX_LOST_BLOCK_CNT_LSB[15:8]							
7	6	5	4	3	2	1	0
RX_LOST_BLOCK_CNT_LSB[7:0]							

- **RX_LOST_BLOCK_CNT_LSB** [*read-only*]: WR Streamer block loss count (LSB)
  Number of indications that one or more blocks in a frame were lost (probably CRC error) since reset

Rx statistics

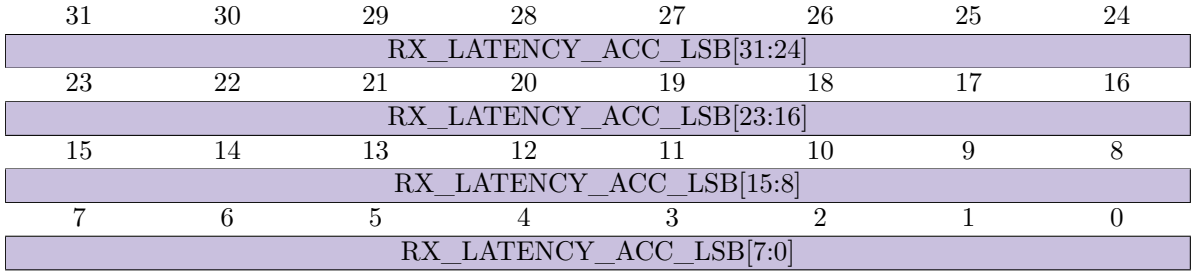
HW prefix: wr_streamers_rx_stat9
HW address: 0xd
SW prefix: RX_STAT9
SW offset: 0x34

31	30	29	28	27	26	25	24
RX_LOST_BLOCK_CNT_MSB[31:24]							
23	22	21	20	19	18	17	16
RX_LOST_BLOCK_CNT_MSB[23:16]							
15	14	13	12	11	10	9	8
RX_LOST_BLOCK_CNT_MSB[15:8]							
7	6	5	4	3	2	1	0
RX_LOST_BLOCK_CNT_MSB[7:0]							

- **RX_LOST_BLOCK_CNT_MSB** [*read-only*]: WR Streamer block loss count (MSB) Number of indications that one or more blocks in a frame were lost (probably CRC error) since reset

Rx statistics

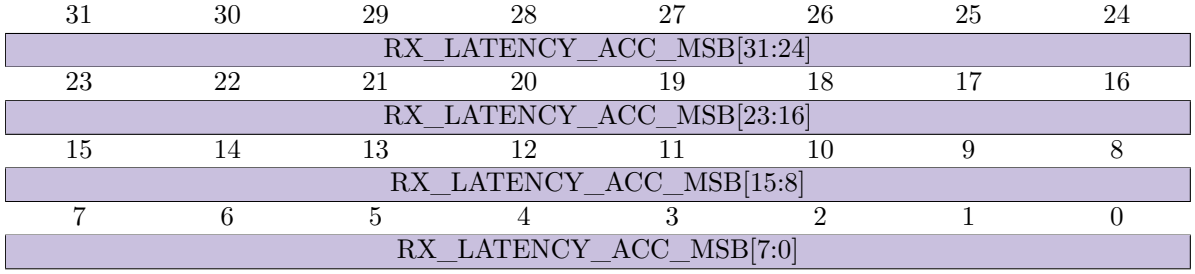
HW prefix: wr_streamers_rx_stat10
HW address: 0xe
SW prefix: RX_STAT10
SW offset: 0x38



- **RX_LATENCY_ACC_LSB** [*read-only*]: WR Streamer frame latency (LSB) Accumulated latency of received frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat11
HW address: 0xf
SW prefix: RX_STAT11
SW offset: 0x3c



- **RX_LATENCY_ACC_MSB** [*read-only*]: WR Streamer frame latency (MSB) Accumulated latency of received frames since reset

Rx statistics

HW prefix: wr_streamers_rx_stat12
HW address: 0x10
SW prefix: RX_STAT12
SW offset: 0x40

31	30	29	28	27	26	25	24
RX_LATENCY_ACC_CNT_LSB[31:24]							
23	22	21	20	19	18	17	16
RX_LATENCY_ACC_CNT_LSB[23:16]							
15	14	13	12	11	10	9	8
RX_LATENCY_ACC_CNT_LSB[15:8]							
7	6	5	4	3	2	1	0
RX_LATENCY_ACC_CNT_LSB[7:0]							

- **RX_LATENCY_ACC_CNT_LSB** [*read-only*]: WR Streamer frame latency counter (LSB)
  Counter of the accumulated frequency (so avg can be calculated in SW) since reset

Rx statistics

HW prefix: wr_streamers_rx_stat13
HW address: 0x11
SW prefix: RX_STAT13
SW offset: 0x44

31	30	29	28	27	26	25	24
RX_LATENCY_ACC_CNT_MSB[31:24]							
23	22	21	20	19	18	17	16
RX_LATENCY_ACC_CNT_MSB[23:16]							
15	14	13	12	11	10	9	8
RX_LATENCY_ACC_CNT_MSB[15:8]							
7	6	5	4	3	2	1	0
RX_LATENCY_ACC_CNT_MSB[7:0]							

- **RX_LATENCY_ACC_CNT_MSB** [*read-only*]: WR Streamer frame latency counter (MSB)
  Counter of the accumulated frequency (so avg can be calculated in SW) since reset

Tx Config Reg 0

HW prefix: wr_streamers_tx_cfg0
HW address: 0x12
SW prefix: TX_CFG0
SW offset: 0x48

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
ETHERTYPE[15:8]							
7	6	5	4	3	2	1	0
ETHERTYPE[7:0]							

- **ETHERTYPE** [*read/write*]: Ethertype

Tx Config Reg 1

HW prefix: wr_streamers_tx_cfg1
HW address: 0x13
SW prefix: TX_CFG1
SW offset: 0x4c

31	30	29	28	27	26	25	24
MAC_LOCAL_LSB[31:24]							
23	22	21	20	19	18	17	16
MAC_LOCAL_LSB[23:16]							
15	14	13	12	11	10	9	8
MAC_LOCAL_LSB[15:8]							
7	6	5	4	3	2	1	0
MAC_LOCAL_LSB[7:0]							

- **MAC_LOCAL_LSB** [*read/write*]: MAC Local LSB

Tx Config Reg 2

HW prefix: wr_streamers_tx_cfg2
HW address: 0x14
SW prefix: TX_CFG2
SW offset: 0x50

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
MAC_LOCAL_MSB[15:8]							
7	6	5	4	3	2	1	0
MAC_LOCAL_MSB[7:0]							

- **MAC_LOCAL_MSB** [*read/write*]: MAC Local MSB

Tx Config Reg 3

HW prefix: wr_streamers_tx_cfg3
HW address: 0x15
SW prefix: TX_CFG3
SW offset: 0x54

31	30	29	28	27	26	25	24
MAC_TARGET_LSB[31:24]							
23	22	21	20	19	18	17	16
MAC_TARGET_LSB[23:16]							
15	14	13	12	11	10	9	8
MAC_TARGET_LSB[15:8]							
7	6	5	4	3	2	1	0
MAC_TARGET_LSB[7:0]							

- **MAC_TARGET_LSB** [*read/write*]: MAC Target LSB

Tx Config Reg 4

HW prefix: wr_streamers_tx_cfg4
HW address: 0x16
SW prefix: TX_CFG4
SW offset: 0x58

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
MAC_TARGET_MSB[15:8]							
7	6	5	4	3	2	1	0
MAC_TARGET_MSB[7:0]							

- **MAC_TARGET_MSB** [*read/write*]: MAC Target MSB

Tx Config Reg 4

HW prefix: wr_streamers_tx_cfg5
HW address: 0x17
SW prefix: TX_CFG5
SW offset: 0x5c

31	30	29	28	27	26	25	24
-	-	-	-	-	QTAG_PRIO[2:0]		
23	22	21	20	19	18	17	16
-	-	-	-	QTAG_VID[11:8]			
15	14	13	12	11	10	9	8
QTAG_VID[7:0]							
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	QTAG_ENA

- **QTAG_ENA** [*read/write*]: Enable tagging with Qtags
- **QTAG_VID** [*read/write*]: VLAN ID
- **QTAG_PRIO** [*read/write*]: Priority

Rx Config Reg 0

HW prefix: wr_streamers_rx_cfg0
HW address: 0x18
SW prefix: RX_CFG0
SW offset: 0x60

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	FILTER_REMOTE	ACCEPT_BROADCAST
15	14	13	12	11	10	9	8
ETHERTYPE[15:8]							
7	6	5	4	3	2	1	0
ETHERTYPE[7:0]							

- **ETHERTYPE** [*read/write*]: Ethertype

- **ACCEPT_BROADCAST** [*read/write*]: Accept Broadcast
  0: accept only unicasts;
  1: accept all broadcast packets
- **FILTER_REMOTE** [*read/write*]: Filter Remote
  0: accept streamer frames with any source MAC address;
  1: accept streamer frames only with the source MAC address defined in mac_remote

Rx Config Reg 1

HW prefix: wr_streamers_rx_cfg1
HW address: 0x19
SW prefix: RX_CFG1
SW offset: 0x64

31	30	29	28	27	26	25	24
MAC_LOCAL_LSB[31:24]							
23	22	21	20	19	18	17	16
MAC_LOCAL_LSB[23:16]							
15	14	13	12	11	10	9	8
MAC_LOCAL_LSB[15:8]							
7	6	5	4	3	2	1	0
MAC_LOCAL_LSB[7:0]							

- **MAC_LOCAL_LSB** [*read/write*]: MAC Local LSB

Rx Config Reg 2

HW prefix: wr_streamers_rx_cfg2
HW address: 0x1a
SW prefix: RX_CFG2
SW offset: 0x68

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
MAC_LOCAL_MSB[15:8]							
7	6	5	4	3	2	1	0
MAC_LOCAL_MSB[7:0]							

- **MAC_LOCAL_MSB** [*read/write*]: MAC Local MSB

Rx Config Reg 3

HW prefix: wr_streamers_rx_cfg3
HW address: 0x1b
SW prefix: RX_CFG3
SW offset: 0x6c

31	30	29	28	27	26	25	24
MAC_REMOTE_LSB[31:24]							
23	22	21	20	19	18	17	16
MAC_REMOTE_LSB[23:16]							
15	14	13	12	11	10	9	8
MAC_REMOTE_LSB[15:8]							
7	6	5	4	3	2	1	0
MAC_REMOTE_LSB[7:0]							

- **MAC_REMOTE_LSB** [*read/write*]: MAC Remote LSB

Rx Config Reg 4

HW prefix: wr_streamers_rx_cfg4
HW address: 0x1c
SW prefix: RX_CFG4
SW offset: 0x70

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
MAC_REMOTE_MSB[15:8]							
7	6	5	4	3	2	1	0
MAC_REMOTE_MSB[7:0]							

- **MAC_REMOTE_MSB** [*read/write*]: MAC Remote MSB

Rx Config Reg 5

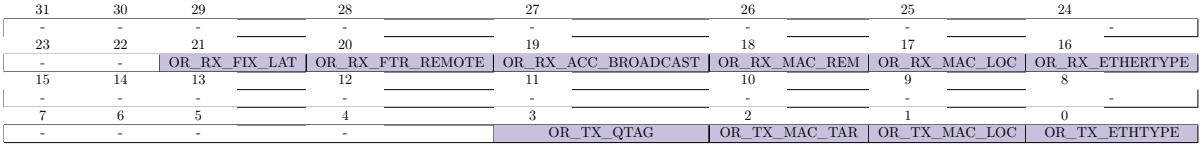
HW prefix: wr_streamers_rx_cfg5
HW address: 0x1d
SW prefix: RX_CFG5
SW offset: 0x74

31	30	29	28	27	26	25	24
-	-	-	-	FIXED_LATENCY[27:24]			
23	22	21	20	19	18	17	16
FIXED_LATENCY[23:16]							
15	14	13	12	11	10	9	8
FIXED_LATENCY[15:8]							
7	6	5	4	3	2	1	0
FIXED_LATENCY[7:0]							

- **FIXED_LATENCY** [*read/write*]: Fixed Latency
  This register allows to configure fixed-latency. If the value is other than zero, the instant of outputing the received data from the rx streamer to the user application is delayed, so that the time-difference between the transmission fo the data and the output to the user matches the provided value. If the configured latency value is smaller than the network latency, the data is provided to the user instantly. The configuration value is expressed in clock cycles (16ns)

TxRx Config Override

HW prefix: wr_streamers_cfg
HW address: 0x1e
SW prefix: CFG
SW offset: 0x78



- **OR_TX_ETHTYPE** [*read/write*]: Tx Ethertype
  Overrides default/application Tx Ethertype configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_TX_MAC_LOC** [*read/write*]: Tx MAC Local
  Overrides default/application Tx local MAC configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_TX_MAC_TAR** [*read/write*]: Tx MAC Target
  Overrides default/application Tx target MAC configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_TX_QTAG** [*read/write*]: QTAG
  Overrides default/application QTAG values with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_RX_ETHERTYPE** [*read/write*]: Rx Ethertype
  Overrides default/application Rx Ethertype configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_RX_MAC_LOC** [*read/write*]: Rx MAC Local
  Overrides default/application Rx MAC Local configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_RX_MAC_REM** [*read/write*]: Rx MAC Remote
  Overrides default/application Rx MAC Remote configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_RX_ACC_BROADCAST** [*read/write*]: Rx Accept Broadcast
  Overrides default/application Rx Accept Broardcast configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register

- **OR_RX_FTR_REMOTE** [*read/write*]: Rx Filter Remote
  Overrides default/application Rx Filter Remote configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register
- **OR_RX_FIX_LAT** [*read/write*]: Rx Fixed Latency
  Overrides default/application Rx fixed latency configuration with configuration in the proper register:
  0: Default/set by application;
  1: Value from WB register

DBG Control register

HW prefix: wr_streamers_dbg_ctrl
HW address: 0x1f
SW prefix: DBG_CTRL
SW offset: 0x7c

This register is meant to control simple debugging of transmitted or received data.
It allows to sniff a 32-bit word at a configurable offset from received or transmitted data.

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
START_BYTE[7:0]							
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	MUX

- **MUX** [*read/write*]: Debug Tx (0) or Rx (1)
- **START_BYTE** [*read/write*]: Debug Start byte
  The offset, in bytes, from which the 32-bit word is read.

DBG Data

HW prefix: wr_streamers_dbg_data
HW address: 0x20
SW prefix: DBG_DATA
SW offset: 0x80

31	30	29	28	27	26	25	24
DBG_DATA[31:24]							
23	22	21	20	19	18	17	16
DBG_DATA[23:16]							
15	14	13	12	11	10	9	8
DBG_DATA[15:8]							
7	6	5	4	3	2	1	0
DBG_DATA[7:0]							

- **DBG_DATA** [*read-only*]: Debug content

Test value

HW prefix: wr_streamers_dummy
HW address: 0x21
SW prefix: DUMMY
SW offset: 0x84

31	30	29	28	27	26	25	24
DUMMY[31:24]							
23	22	21	20	19	18	17	16
DUMMY[23:16]							
15	14	13	12	11	10	9	8
DUMMY[15:8]							
7	6	5	4	3	2	1	0
DUMMY[7:0]							

- **DUMMY** [*read-only*]: DUMMY value to read