## Table of Contents

**Introduction**  ................................................................................. 1

1 History and Overview ............................................................. 1

2 Compiling the Drivers .............................................................. 1

3 Role of spec.ko ................................................................. 2
  3.1 SPEC Initialization ........................................................... 2
  3.2 SPEC Module Parameters .............................................. 2
  3.3 Interrupts in spec.ko ...................................................... 3

4 The WR-NIC ........................................................................ 4
  4.1 Code Layout ................................................................. 5
  4.2 Writing the DIO EEPROM ............................................... 6
  4.3 Overview of the Driver ................................................... 7
  4.4 Module Parameters ....................................................... 7
  4.5 Controlling the White Rabbit Core ................................. 8
  4.6 Transferring Data .......................................................... 9
  4.7 Timestamping Frames .................................................... 9
  4.8 Accessing the DIO Channels .......................................... 10
  4.9 WR-NIC Command Tool ............................................... 11
  4.10 Timestamping Fast Input Signals ................................... 13
  4.11 WR-DIO Pulse per Second ............................................ 13
  4.12 Distributing Output Pulses ............................................. 15
  4.13 The Future of WR-NIC .................................................. 16

5 User-Space Tools ............................................................... 17

6 Bugs and Missing Features .................................................. 18

7 Portability ............................................................................ 18
Introduction

This is the manual for the SPEC device driver. SPEC is the *Simple PCI-Express Carrier* for FMC cards, developed at [http://www.ohwr.org/projects/spec](http://www.ohwr.org/projects/spec). This manual is part of the associated software project, hosted at [http://www.ohwr.org/projects/spec-sw](http://www.ohwr.org/projects/spec-sw). The latest version of this work is always available in the *git* repository at [ohwr.org](http://www.ohwr.org).

1 History and Overview

This driver is pretty different from the initial implementation, such as the one used by fine-delay-sw-v1.1. If you are using that version, please compile the manual you find in your source code repository and avoid reading this one.

The driver is a client of the FMC software bus abstraction. The package *fmc-bus* is dowloaded as a *git* submodule of this package, and its latest version can be found at [http://www.ohwr.org/projects/fmc-bus](http://www.ohwr.org/projects/fmc-bus).

2 Compiling the Drivers

The kernel modules that are part of this package live in the *kernel* subdirectory. To compile them, you need to set the following variables in your environment:

**LINUX**

The top directory of the kernel sources for the version you are going to run the driver under. I'm testing mostly with 3.4, but this version compiles against Linux-2.6.37 and later ones (2.6.36 had a different interface for hardware timestamping, so *fmc* and *spec* compile fine, but *wr-nic* does not).

**CROSS_COMPILE**

If you are cross-compiling, you need to set this variable. It is not usually needed for the PC, but if you are using the *Powec* board, you'll most likely need this. It is not needed if you compile for a different-sized PC (see below).

**ARCH**

If you are cross-compiling, set this variable. Use *powerpc* for the *Powec*, *x86-64* if you compile on a 32-bit PC to run on a 64-bit PC and *i386* if you compile on a 64-bit PC to run on a 32-bit PC.

To compile run “make” with the previous variables set. To install run “make install” to install under `/lib/modules/3.4.0` (or other version-based directory). You can set INSTALL_MOD_PATH to force and installation directory (as a prefix followed by `/lib/modules/...`). For example, if your target computer’s filesystem is mounted under `/mnt/target` you can run

```
make install INSTALL_MOD_PATH=/mnt/target
```

The modules are installed under the subdirectory *extra*. In the case shown above your driver will end up being installed (together with the other modules) as

```
/mnt/target/lib/modules/3.4.0/extra/spec.ko
```

Please note that by default the package compiles an installs the *fmc-bus* modules, too (the project is a *git* submodule).
3 Role of spec.ko

The spec.ko driver depends on fmc.ko, that must be loaded first (unless you rely on automatic dependencies).

spec.ko registers as a PCI driver, using both the “old” identifiers (i.e. the Gennum vendor and GN4124 device) and the new ones (CERN vendor and SPEC device). It has a module device table, so after installation it will be automatically loaded at the next boot.

3.1 SPEC Initialization

For each new SPEC device found on the system, the driver performs the following steps:

- It enables MSI interrupts if enabled with a parameter.
- It loads the fmc/spec-init.bin “golden” gateware file.
- It checks that the content of the binary is as expected (using a minimal sdb-based verification).
- It reads the whole I2C EEPROM found on the mezzanine.
- It allocates an fmc_device structure and registers as a new device in the fmc bus.

Failure of any of the above steps is fatal.

The suggested spec-init.bin gateware binary is always available from the files area of the spec-sw project on ohwr.org. The current binary version to be used with this software version is http://www.ohwr.org/attachments/download/1756/spec-init.bin-2012-12-14.

The EEPROM content that the carrier driver reads at initialization time is still available to mezzanine drivers after they are loaded, but a mezzanine driver cannot write to the EEPROM using carrier methods after reprogramming the FPGA. The carrier refuses to act on registers after the golden gateware is replaced by a new mezzanine-specific binary, which by definition is unknown to the carrier itself.

Note: currently the SPEC driver does not re-write the golden binary file when the sub-driver releases control of the card. This allows a further driver to make use of an existing binary, which may be useful during development.

3.2 SPEC Module Parameters

The module can receive the following parameters to customize its operation:

fw_name
This string parameter can be used to override the default name (fmc/spec-init.bin) for the initialization binary file.

use_msi
This forces the driver to use message signalled interrupts. MSI interrupts have some advantages over conventional wire-level interrupts, but with the GN4124 we had serious issues. See Section 3.3 [Interrupts in spec.ko], page 3 for details. By default the driver uses the old-fashioned wire-level signalling method; to experiment with MSI pass use_msi=1.

test_irq
If not zero, this parameter requests to self-test interrupt generation, using the software-driven interrupt source in Gennum registers. This usually does not work on my host if I use MSI, for yet unknown reasons (and that’s why it is disabled by default).
Chapter 3: Role of spec.ko

3.3 Interrupts in spec.ko

Up to version 2.0 of this package, I enabled MSI (message signalled interrupts) in the Gennum chip. Now the default is using normal wire-level PCI interrupts; there is a module parameter called use_msi for those who want to experiment with MSI.

Unless you want to help with making MSI work, you are not interested in this section, and you can skip over it.

While MSI should perform better than conventional interrupts, there's a misbehaviour in the GN4124 that makes their use pretty difficult.

The message in MSI includes a data field, chosen by the operating system for its own use. Linux stores the so-called vector number in the data field, which is an index in an kernel-internal array. The PCI-E standards allows for peripheral devices to support several MSI signals, if the operating system enables them to do so; the Gennum chip supports this, and is able to fire 4 different MSI signals, however, it is not able to send the base MSI signal if the multiple-MSI feature is kept disabled.

The 4124 is found to choose the MSI interrupt to send according to the two least significant bits of the data field. For this reason, we must enable the multiple-MSI feature, and enable

---

i2c_dump
If not zero, this parameter requests to printk the content of the FMC eeprom, for diagnostic purposes.

show_sdb
If not zero, the SDB internal structure of the golden binary is reported through kernel messages. It is disabled by default.

eeprom
The parameter accepts one or more comma-separated filenames, to be used as fake EEPROM images for the various cards in their probe order. The probe order is unpredictable, but it's always the same when you unload and reload the module. This is a tool for developers, to allow experimenting with eeprom images (likely related to SDB configurations) without the need to wear the real i2c device. Please note that the fake eeprom is only used with the golden firmware: mezzanine-specific drivers that read and write the EEPROM with their own gateware-specific code will still access the real I2C device.

Any mezzanine-specific action must be performed by the driver for the specific FMC card, including reprograming the FPGA with the final gateware file. Similarly, the spec driver is not concerned with programming the LM32 image, when it makes sense to. This is different from the role splitting in previous versions of the driver.

Note: the driver looks for the gateware binary in /lib/firmware/fmc, which is where all fmc-related external files are expected to live. That's because our own installations share firmware for COTS peripherals but mount a host-specific NFS subdirectory.

Please refer to the fmc-bus manual for details about the overall design of the interactions of carriers and mezzanines.

Warning: currently the match function of the bus always returns success: the mezzanines I currently have for testing have no ID records written in their internal EEPROM, so I'm not able to setup the associated data structures and code. For this reason there is no module_alias support nor autoprobe of drivers: any fmc driver you load will drive all SPEC cards found on the system unless it limits itself through parameters (see below)
in the chip the corresponding interrupt configuration register. The former operation involves a standard registers, the latter is Gennum-specific.

Linux, however, does not know that multiple-MSI is enabled, and thus it configures the standard register for a single MSI. When this happens, interrupts stop working (because the Gennum would send msi 2, for example, but only msi 0 is enabled).

The code of this package, under “if (use_msi)” stanzas, fixes the control register after Linux touches it, but it may not be enough. We found, for example, that the irqbalance tool (which is installed and runs in background in a number of recent installations) will make some configuration attempt after interrupts starts to flow. When this happens, Linux will rewrite the control register thus disabling SPEC interrupts. Killing irqbalance in advance was found to fix the problem, but clearly this isn’t production-ready. Note: Linux doesn’t currently support enabling multiple-MSI unless they are MSI-X (a further standard) so we can’t just enable all 4 of them.

If you want to help with taming the MSI problem, you should load the spec driver with use_msi=1 and run the wr-nic driver. If you are unable to exchange frames, or data transfer just stops, please grep for wr-nic in ‘/proc/interrupts’ to see if the counters are moving or not. If not, the problem is most likely in register 0x4a (MSI_CONTROL) that lost its correct value of 0xa5. You can check the high 16 bits in the output of specmem -g 48 to verify. To unlock the situation, you need to fix this register and force an edge in the interrupt line from the FPGA to the Gennum, by acting on VIC registers. The “magic” sequence is the same you find at the end of wrn_handler(), in ‘wr-nic-eth.c’.

The following snapshot come from a system with two SPEC cards plugged. No irqbalance is running and both kind of interrupts work. With non-msi interrupts, the lines are shared with other peripherals, and you’ll see something like this:

```
spusa.root# grep nic /proc/interrupts
16: 236 0 0 805 IO-APIC-fasteoi snd_hda_intel, wr-nic
18: 0 14721 16 33 IO-APIC-fasteoi ahci, ohci_hcd:usb4, wr-nic
```

If you enable MSI, interrupts have higher numbers, allocated specifically for the peripheral:

```
spusa.root# grep wr-nic /proc/interrupts
47: 0 0 0 124 PCI-MSI-edge wr-nic
48: 70470 0 0 26 PCI-MSI-edge wr-nic
```

4 The WR-NIC

This package is expected to only host the carrier driver, i.e. the one that registers a device for the FMC bus. However, at the time being the wr-nic mezzanine driver is part of this package, for various reasons.

The wr-nic.ko mezzanine driver is designed to run with the simple 5-Channel simple-DIO mezzanine board, because it includes both code to access the network card and code to act on the DIO channels, using time tags that are provided by the White Rabbit mechanism. Similarly, both incoming and outgoing frames can be time-stamped by White Rabbit.

When loaded as a kernel module, the wr-nic driver will only take hold of simple-DIO mezzanines whose EEPROM is correctly programmed. So, if your computer hosts several spec devices, only the ones with a DIO card will become wr-nic. If your EEPROM is blank, you are urged to reprogram it as described in Section 4.2 [Writing the DIO EEPROM], page 6.

This driver is the most important driver in this package, from the user’s point of view. It is a generic implementation of the spec-sw framework which can be useful by itself as a White Rabbit starter kit. Moreover, it is a complete driver that can serve as a model for other developments.
Within White Rabbit we have other full-featured drivers for specialized FMC mezzanines hosted on the SPEC carrier. They are not part of this package because of their specialized nature; all of them are nonetheless hosted on www.ohwr.org, usually as a software subproject of the related gateware project.

4.1 Code Layout

This section is mainly for the developers who look in the code, and for me to make order in my own mind. SPEC users are expected to skip to the next section.

The `wr-nic.ko` is built using a number of headers and source files, spread over several directories:

`'wbgen-regs/'`

The directory hosts the register definitions for the various core that are included in the FPGA binary. The headers are generated by running `wbgen2` over the `.wb` files that are part of the VHDL source repositories; unfortunately some minor editing is needed on the `wbgen2` output, so there is a `Makefile` that takes care of this. This package includes both the input file and the output header; the log messages details the upstream origin of each `.wb` file. The directory started out as a direct copy of the directory with the same name found in `wr-switch-sw`, release 3.0.

`'wr_nic/'`

The directory started out as an unchanged copy of the driver used in the `wr-switch-sw` package, release 3.0. The directory name is the same in both projects. All later commits take care of differences in the SPEC with regard to the switch, but we plan to clean up those later changes and reach a unified code base between the White Rabbit switch and the White Rabbit node. The NIC driver itself is a platform driver, instantiated by platform devices defined externally.

The `wr-nic` driver refers to several headers, in addition to the register definitions. This is the role of each of them:

`'fmc-bus/kernel/include/linux/fmc.h'`
`'fmc-bus/kernel/include/linux/sdb.h'`
`'fmc-bus/kernel/include/linux/fmc-sdb.h'`

These three headers are part of the `fmc-bus` project, whose repository is a git submodule of this one. They are used to define the interface to the FMC bus abstraction and the SDB self-description of the internal FPGA bus. They are used by other spec-sw files as well. We include them as `<linux/fmc.h>` because we plan to have them upstreamed to the official kernel, and we don’t want to introduce incompatibilities in the related source files.

`'wr_nic/nic-mem.h'`
`'wr_nic/nic-hardware.h'`
`'wr_nic/wr-nic.h'`

These headers come from `wr-switch-sw/wr_nic` with minor SPEC-related modifications. `nic-mem.h` defines the memory map and is now almost obsoleted by SDB; `nic-hardware.h` is a collection of inline functions used by the driver; `wr-nic.h` defines all the important data structures and ioctl commands. Because of ioctl commands, it has a rather generic name and is meant to be included by user space as well as kernel space.

`'spec.h'`

Definitions related to the SPEC carrier (Gennum registers and other SPEC-internal stuff). It is currently used by `wr-nic-eth.c`, which is not completely carrier-independent.
The header defines the SDB vendor and device values used in wr-nic as well as the data structures and prototypes used internally by the driver.

wr-dio.h

The header hosts the user interface to access the DIO channels. It is included by wr-nic-dio.c as well as the user-space tools that want to configure DIO operation.

The source code of the driver itself is split in several files, in addition to the NIC platform driver hosted in 'wr_nic/':

wr-nic-core.c

The file is the fmc driver: it implements the probe and remove methods and deals with loading the firmware file and the LM32 program binary (called wrc: White Rabbit Core).

wr-nic-eth.c

This is concerned with creating the platform device for the network interface card. It maps the needed device memory, allocates the platform data and sets up the internal interrupt controller to route interrupts to the platform driver.

wr-nic-dio.c

This is the mezzanine-specific driver. It implements the ioctl commands that allow user space to talk with the mezzanine. It only implements the ioctl method and support functions for it (e.g., interrupt management). If you want to port wr-nic to a different mezzanine, this is the file you need to replace.

4.2 Writing the DIO EEPROM

As said, the wr-nic driver requires that the EEPROM of the mezzanine is correctly programmed. While recently-shipped cards are preprogrammed, the early ones were shipped with a blank eeprom.

Upon loading spec.ko, the carrier driver identifies all mezzanines. If your EEPROM is properly programmed, you’ll see messages like the following ones:

spec 0000:02:00.0: FPGA programming successful
spec 0000:02:00.0: mezzanine 0
Manufacturer: CERN
Product name: FmcDio5cha

If the memory is blank, you’ll see this:

spec 0000:04:00.0: FPGA programming successful
spec 0000:04:00.0: mezzanine 0
EEPROM has no FRU information

In this case, if you are sure the spec device in PCIe slot number 4 hosts a DIO card, you can write the proper binary image into /sys/bus/fmc/devices/fmc-0400/eeprom. A suitable memory image is shipped as binaries/fmc-dio-eeprom.bin, so the following commands, run as superuser, will work:

cat binaries/fmc-dio-eeprom.bin > /sys/bus/fmc/devices/fmc-0400/eeprom
rmmod spec
insmod kernel/spec.ko

Please remember to check the bus/slot number: the 0400 above must be adapted to your own system.

By removing and reloading the spec driver, as suggested, you force it to read the new identifiers. The new device in sysfs, therefore, will be called FmcDio5cha-0400 instead of fmc-0400.

The provided EEPROM image has been generated with fru-generator, part of the package fmc-bus, with the following command:
The above serial number is fake, and users with a pre-programmed EEPROM are not expected to overwrite it.

4.3 Overview of the Driver

The wr-nic driver is basically an Ethernet driver with support for hardware time stamping. The simple-DIO mezzanine card can be used by means of ioctl commands. Such commands are designed to be portable, so user-space programs should able to identify which mezzanine is connected to the SPEC network card and act accordingly.

The driver loads two binaries, using the firmware loader mechanism offered by the Linux kernel: one is the gateware file, that is requested through the reprogram carrier method; the other is the LM32 program binary, which however is only loaded on user request. With the current version of the gateware binary, the LM32 program is already included in its most current version, so you won’t need to load it.

The default file names are as follows:

‘fmc/wr_nic_dio.bin’
This is the gateware file.

‘fmc/wr_nic_dio-wrc.bin’
This is the LM32 program file, or White Rabbit Core, WRC. The file is not loaded automatically, because the official gateware file already includes the correct LM32 program. To request loading the file you should pass wrc=1. To request loading a different WRC file name, you should pass the actual file name. For example “wrc=recompiled-wrc.bin”.

The binaries suggested for this software release are available from the files tab of the Open Hardware Repository. The direct links are as follows (the former is required, the latter is optional):

http://www.ohwr.org/attachments/download/2687/wr_nic_dio.bin-2014-02-14
http://www.ohwr.org/attachments/download/2688/wr_nic_dio-wrc.bin-2014-02-14

The date is included in the binary name so we won’t need to remove the binaries when they are obsoleted by newer ones: spec-sw releases are expected to keep working in the future, with no compulsory updates: if you are using this version you need those very files to be available over time. You can copy the following command sequence to your shell in order to fill your ‘/lib/firmware/fmc’ with everything that’s needed to run wr-nic:

    cd /tmp
    wget -O wr_nic_dio.bin http://www.ohwr.org/attachments/download/2687/wr_nic_dio.bin-2014-02-14
    wget -O wr_nic_dio-wrc.bin http://www.ohwr.org/attachments/download/2688/wr_nic_dio-wrc.bin-2014-02-14
    sudo mv wr_nic_dio wr_nic_dio-wrc /lib/firmware/fmc

4.4 Module Parameters

The module receives the following parameters:

busid=

These are standard parameters. You can either limit the driver to only bind to some specific SPEC cards (not all of the ones selected by the match function of fmc-bus),
or to assign different gateware files for each card. They are fully described in the fmc-bus manual.

show_sdb= This is another standard parameter. Set to 1 to get the SDB tree printed to kernel messages.

file= This parameter is a strings, and it is a global override for the default gateware file name. Please note that gateware= takes precedence.

wrc= The name of the LM32 program to load, if any. There is no support currently to load different LM32 programs to different cards.

4.5 Controlling the White Rabbit Core

In this release the driver is not controlling the White Rabbit Core and the default mode of operation is slave. You can use the serial port and interact with the WRC shell to change the operation mode and do other supported interaction with the PTP daemon. If you want, you can use the serial port to configure a card as free running master storing such request in EEPROM to make it persistent.

The complete reference of the shell commands is included in the wrpc-sw manual in the files tab of the project. The direct link is http://www.ohwr.org/attachments/download/1586/wrpc-v2.0.pdf. (Actually, you should get the latest available or, even better, the documentation that is part of the package you are running).

The most useful commands are repeated here for your convenience

mode grandmaster
mode master
mode slave

The commands change the current PTP mode. mode with no arguments reports the current mode.

ptp stop
ptp start

Stop and start the daemon running on LM32. You’ll most likely need to stop and restart the PTP daemon after changing mode.

time raw

Prints the internal device time as seconds and nanoseconds.

gui

Start a self-refreshing informative display of the White Rabbit synchronization status. Press <ESC> to return to command-line mode.

mac get

Reports the MAC address used by WRPC (it should match what is reported by ifconfig in Linux).

Please note that you may also need to configure the SFP module you are using, with the sfp WRC command, as described in the wrc-v2.0.pdf manual referenced above.
4.6 Transferring Data

The \textit{wr-nic} driver registers a Linux network interface card for each SPEC device it drives. The cards are called \texttt{wr%d} (i.e., \texttt{wr0, wr1,} \ldots).

The MAC address of the device is retrieved from the internal \textit{White Rabbit} registers, because at the time when Linux configures the interface the WRC code has already configured the Ethernet port and generated a valid MAC address using the serial number of the internal thermometer.

The user is thus only expected to assign an Internet address to the Ethernet port to be able to use it to transfer data. Note however that \textit{White Rabbit} synchronization happens even if the interface is not configured in Linux. In case you need to change the MAC address as seen by Linux, the command to type is something like the following:

\begin{verbatim}
ifconfig wr0 hw ether 12:34:56:78:9a:bc
\end{verbatim}

The fiber controlled by the SPEC can carry normal data traffic in addition to the PTP frames of \textit{White Rabbit}, that remain invisible to the host computer. The examples related to the simple \textit{DIO} device use this data channel to exchange Ethernet frames so you'll need to assign IP addresses to your \texttt{wr} interfaces.

4.7 Timestamping Frames

The SPEC Ethernet interface supports hardware timestamping for user frames through the standard Linux mechanisms. Time stamps are currently reported with a resolution of 8ns only (\textit{White Rabbit} does much better, but we don't have the code in place for this demo, yet).

Unfortunately the Linux mechanisms are not trivial: the application must enable timestamping on both the hardware interface and the specific socket it is using, and it must issue several \texttt{ioctl} and \texttt{setsockopt} commands. Moreover, timestamps are returned to user space using the \texttt{recvmsg} system call, which is more difficult to deal with than the normal \texttt{send} or \texttt{recv}.

To simplify use of timestamps for Ethernet frames, this package includes the \texttt{‘stamp-frame’} program in the \texttt{‘tools’} directory. The leading part of its source file includes generic library-like functions that deal with the intricacies of timestamping; the final part is the actual example, which is designed to be simple and readable.

The program is a minimal implementation of the basic time-synchronization protocols (like NTP and PTP), but excluding the synchronization itself. The idea is sending a frame from one host to another, and receiving a second frame back; the departure and arrival times are recorded and collected at a single place, so they can be reported to the user.

The \texttt{‘stamp-frame’} example supports two modes of operations. In \texttt{listen} mode, it binds to an Ethernet interface and listens forever: it waits for the forward frames and replies to them; in normal mode it sends the forward frame and reports data as soon as it gets a reply. This is an example running on two different hosts:

\begin{verbatim}
tornado.root# stamp-frame wr0 listen
stamp-frame: Using interface wr0, with all timestamp options active

spusa.root# stamp-frame wr1
stamp-frame: Using interface wr1, with all timestamp options active
timestamp T1: 1476.381349032
timestamp T2: 1476.381403352
timestamp T3: 1476.391563248
timestamp T4: 1476.391617560
round trip time: 0.000108632
forward time: 0.000054320
backward time: 0.000054312
\end{verbatim}

The four times are departure and arrival of the forward frame, followed by departure and arrival of the backward frame. Thus, time stamps T1 and T4 are collected at the original sender (here: \textit{spusa}) while T2 and T3 are collected at the remote host (here: \textit{tornado}). The times
above are all consistent because the two SPEC cards are synchronized with White Rabbit. The reported forward and backward times match the fact that I used a 10km fiber to connect the two cards; the difference between them is due to the different speed of light in the two directions, because the two SFP transceivers I plugged use different wave lengths.

The following example shows the output for two forcibly-unsynchronized cards. The difference between the two clocks is clearly a few seconds; the round trip time is correct nonetheless, because it is a difference of differences:

<table>
<thead>
<tr>
<th>Timestamp T1:</th>
<th>13.225249168</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp T2:</td>
<td>9.130237600</td>
</tr>
<tr>
<td>Timestamp T3:</td>
<td>9.140438816</td>
</tr>
<tr>
<td>Timestamp T4:</td>
<td>13.235559016</td>
</tr>
<tr>
<td>Round trip time:</td>
<td>0.000108632</td>
</tr>
<tr>
<td>Forward time:</td>
<td>-5.904988432</td>
</tr>
<tr>
<td>Backward time:</td>
<td>4.095120200</td>
</tr>
</tbody>
</table>

The code in `stamp-frame` is designed to be simple to be reused, but there is one non-obvious detail that is worth describing here. Whereas the receive timestamp is returned to user-space together with the frame it refers to, the transmit timestamp is only known after the relevant frame left the computer. For this reason, in order to communicate the TX timestamp of a frame to your peer, you'll need to send another message which carries the departure time of the previous frame. This further message is usually called follow-up, and `stamp-frame` respects this tradition.

### 4.8 Accessing the DIO Channels

In order to access the DIO channels, user-space programs are expected to issue device-specific `ioctl` commands. The driver supports two commands, allocated at the end of the range of command numbers reserved for device-specific use:

**PRIV_MEZZANINE_ID**

The command is used to identify the features of the specific NIC device. It tells user space which mezzanine is currently plugged and also which type of carrier you are talking to. The command exchanges a data structure with user space in order to be able to extend its functionality over time, and such data structure includes a sub-command field. (For example, we may return EEPROM contents to user space on request).

**Warning:** the command is not implemented because we still have no mezzanine identification in place. The error being returned is `EAGAIN`; user code can rely on that error to know it is actually talking with a SPEC device, even if no identification is currently possible.

**PRIV_MEZZANINE_CMD**

The command is based on the exchange of a data structure: by means of sub-commands included in such structure user space programs can request different services to the mezzanine driver. In the case of the DIO mezzanine this includes generating pulses and timestamping input events; other mezzanine drivers will be able to use the command in a different way. The application is expected to first run `PRIV_MEZZANINE_ID` to ensure the NIC device is connected to the right mezzanine.

In the specific case of this *wr-nic* driver, the data structure is defined and explained in `wr-dio.h` and is not repeated here.

The structure includes a few integer fields and an array of `struct timespec`. Such structures define time stamps with nanosecond precision, but the simple-DIO mezzanine and its gateware are able to time-stamp input events and generate output events with a resolution of 8ns.

When the device is asked to timestamp input events, the array of `struct timespec` is used to return such events to user space. When the device is asked to generate output pulses at specific
points in time, the array is used to pass three values: the beginning of the pulse, the duration of the pulse and (optionally) the period of the pulse train.

Specifics about the use of individual fields are shown in the header (in a big comment block), in the driver itself and in the user-space programs that call *ioctl*.

In lab environments you may be concerned about the duration of the *ioctl* implementation, because it sometimes seems to do more work than needed. To verify whether we have an over-engineering problem in kernel space, I provided a simple measurement of how much time is spent in the *ioctl* itself. The make variable `WR_NIC_CFLAGS` can be used to pass extra flags to the compiler, and the macro `DIO_STAT` enables the time measurement. Compiling with the following command thus enable such measurement and associated *printk* – the time is usually 5 microseconds for me:

```
make WR_NIC_FLAGS=-DDIO_STAT
```

Previous versions of this manual described how to command pulses on several channels with a single *ioctl* command, but that feature has never been implemented (one of the reasons is that *ioctl* revealed fast, so calling it several times is acceptable).

### 4.9 WR-NIC Command Tool

In the ‘tools/’ subdirectory of this project, you find the ‘wr-dio-cmd’ program, which is a command-line interface to the *ioctl* command that acts on the simple-DIO mezzanine card. Other *wr-dio-* tools are provided (and described below) but this is the most generic one.

Please note that neither timestamping nor pulse generation work if the WR core is not running or has an invalid time: it must either be a master or a synchronized slave.

Moreover, please note that this tool is just a demonstration to quickly test the I/O features of the device (and for me to verify the kernel part is actually working): for serious use you should call *ioctl* by yourself with proper arguments, and avoid all the parsing of ASCII and repeated invocation of this program.

This is the general syntax of the command:

```
wr-dio-cmd <ifname> <cmd> [arg] ...
```

The arguments have the following meaning:

**ifname**

The name of the network interface, most likely wr0 (if you have more than one SPEC card, the other interfaces are called wr1, wr2 and so on).

**cmd**

The specific command. Supported commands are listed below. Each command takes zero or more of arguments. If you pass a wrong number of arguments you’ll get help, and if one argument is wrong (e.g., not a number) the error message is meant to be directly helpful.

**Note:** This command, like everything else in this package, numbers channels from 0 through 4, whereas the back panel of the mezzanine numbers them 1 through 5. My devices have no panel, so I just made the wrong guess.

The current version of the tool supports the following commands:

```
stamp [channel] [wait]
stamppm [channel-mask]
```

The commands are used to retrieve timestamps from the card. If no arguments are passed, the tool reports to *stdout* all timestamps for all channels (they are ordered by channel, not by time). If one integer argument is passed, it can be a channel number in the range 0 to 4 (*stamp* command) or a mask in the range 0 to 0x1f
(stampm command). If getting stamps for an individual channel, you can add the
wait option to have the tool wait for (and report) new timestamps until killed.

**Warning:** use of wait is dangerous because it has been implemented against the
rules. You must terminate any waiting process before you unload the driver, or
your PC will explode and will destroy your academic career.

**pulse <channel> <duration> <when> [period <count>]**

Channel is an integer in the range 0 to 4. The duration must be specified as a
fraction of a second (decimal number, less than one second), the **when argument**
can be the string now, an absolute time (<seconds>.<fraction>) or a relative
time (+<seconds>.<fraction>). In the last case, the current second is added to
<seconds> while the fraction is used unchanged. The + form is useful for simple
checks with visual inspection. **period**, if specified, requests for a pulse train, with
the specified time period between raising edges; **count** is the number of instances
to run (-1 means forever, 0 means “stop generating pulses”).

**mode <channel> <mode> [channel <mode> ...]**

Configure one or more channels for the specified mode. Each mode is represented
by one character; the latter form above thus configures all 5 channels by means of
a 5-bytes-long string, each characters specifies a mode according to the next table.

This is the list of supported modes for channels:

**I i**

Input mode, with termination or without termination. Uppercase forces the termi-
nation on (the rationale is you need a stronger signal to drive the input). Termination
is 50 Ohm. Pulses on an input channel are timestamped.

**O 1**

Digital output mode, from the GPIO logic core, resp. low and high. Both modes
disable the termination resistor.

**D d**

DIO core output (not the GPIO core). With this mode the channel is driven by the
WR-aware digital I/O, and can thus generate pulses and so on. Uppercase enables
the termination. A channel managed by the DIO core is normally low and can pulse
high on request (see pulse command. The termination resistor doesn’t make much
sense for output, but the code is provided for consistency with input modes.

**P p (Channel 0)**

Pulse per second output from the White Rabbit core that can be used only for the
first channel (ch0). This pps is sharper in its absolute time than the one that can be
generated by software using DIO pulses. Again, upper case selects the termination,
for symmetry with input modes.

**C c (Channel 4)**

“Clock” input. This mode (with or without termination) is used to feed a clock
to the White Rabbit core (currently the WR core supports a 10MHz input on
channel 4 (the last one). For other channel, the mode is just like I or i but without
timestamping (and thus without a software interrupt).

**Note:** The first channel (channel 0) has been modified and now support only the P/p as output
mode. You will not be able to use D,d,1,0 modes.

**Note:** At startup, the DIO channels are configured by default as: wr-dio-cmd mode p00ic.
Generation of a pulse train is performed by software running at interrupt time, because the simple DIO card and gateware can only emit a single pulse at a requested White Rabbit time. For this reason you’ll observe that the pulse period cannot be too short, according to how powerful your computer is and how much you loaded it with other tasks. In the future, new gateware files may perform pulse trains in hardware.

There is no command to flush the timestamp FIFOs, but you can always “wr-dio-cmd stamp > /dev/null” if needed.

Example uses of the tool follow:

```bash
# Pulse channel 4 for 0.1 seconds now
wr-dio-cmd wr0 pulse 4 .1 now

# Pulse for 10 microseconds in the middle of the next second
wr-dio-cmd wr0 pulse 4 .00001 +1.5

# Pulse for 1ms at 17:00 today
wr-dio-cmd wr0 pulse 4 .001 $(date +%s --date 17:00)

# Get timestamps for the output events above
wr-dio-cmd wr0 stamp 4

# Make a train of 5 pulses, 0.5ms wide, every ms at next second
wr-dio-cmd wr0 pulse 4 0.0005 +1 .001 5

# Configure channel 0 as input with termination, 1 as input, 4 as low
wr-dio-cmd wr0 mode Ii--0
```

### 4.10 Timestamping Fast Input Signals

When timestamping pulses in the simple-DIO mezzanine board, an interrupt is generated to notify the driver that a new timestamp is pending. On recent computers this works reliably up to more than 100kHz, but clearly there is a point where the system locks up, because it spends all of its time in interrupt handling.

This problem is transient: as soon as you remove the offending cable the system recovers. However, you need a 10MHz input signal if you want to run your SPEC device to be a White Rabbit grandmaster. In order to support that, the driver disables DIO interrupts when the time spent in interrupt management exceeds 80% of the total time, averaged over one thousand interrupt events. Ethernet interrupts are not affected. The fact is reported by a kernel message, using the PCI address of the card that triggered the problem.

```
spec 0000:04:00.0: DIO irq takes > 80% CPU time: disabling
```

This choice allows stamping your pulse trains up to a few dozen kilohertz and still be able to feed higher frequencies without manual intervention. However, after DIO interrupts are disabled, the only way to re-enable them is removing and reloading the device driver.

**Note:** if you run two SPEC cards, and one is fed with high frequency pulses, it may happen that interrupts are disabled on both boards. The safeguard is currently not very refined, as it was implemented in a hurry.

### 4.11 WR-DIO Pulse per Second

To better show how to write your own application with the SPEC driver and the simple-DIO mezzanine card, this package includes ‘wr-dio-pps’, which features a very small and readable source file.
The program is meant as a source code example, more than a real PPS signal. If you need a real WR-driven pulse-per-second, you should use the channel0 which is "hard-wired" to the PTP core and can be configured by executing: "\texttt{wr-dio-cmd mode 0 p}".

The program just fires a 1ms-long \texttt{pps} pulse on one of the output channels. The device name defaults to \texttt{wr0} but can specify a different one; the channel number is a mandatory argument.

\textbf{Note:} This command, like everything else in this package, numbers channels from 0 through 4, whereas the back panel of the mezzanine numbers them 1 through 5. My devices have no panel, so I just made the wrong guess.

\begin{verbatim}
# run pps on channel 2 of the default SPEC card
./wr-dio-pps 2

# run pps on channel 0 of the "second" card
./wr-dio-pps wr1 0
\end{verbatim}

The following two figures show two such pulse-per-second signals retrieved from two different simple-DIO cards, connected by a 10km roll of fiber, after syncing with White Rabbit. In some cases, it may happen that the leading edges differ by almost exactly 8ns; this happens because in the simple-DIO all times are quantized by 8ns-long clock ticks. The differences in internal delays (which depend on the carrier, the mezzanine and the FPGA binary), are not self-measured and calibrated in this simple design and may appear in the output after quantization. A more complete experimental setup would include calibration of the internal delays of the boards, like the mechanism in place for the fine-delay mezzanine card (see \url{http://www.ohwr.org/projects/fmc-delay-1ns-8cha} and \url{http://www.ohwr.org/projects/fine-delay-sw}).
4.12 Distributing Output Pulses

A typical application for White Rabbit (or any time synchronization system) is being able to generate output signals at the same time in different output boards; another typical application is time stamping input events.

By using the Ethernet interface included in the SPEC, an application can exchange data with other White Rabbit devices; thus, it can easily request output event to other output peripherals, or collect remote input events. The tool-set offered by the driver is made up of the PRIV_MEZZANINE_CMD ioctl command, and the usual Posix API for network communication.

The DIO-specific ioctl command is the one used by the wr-dio-cmd tool described above, while network communication should be known to most users of this package. In order to ease new White Rabbit users, though, this package includes sample code to implement a simple dual-headed system with concurrent output. The examples are also meant to show the basic code that uses the provided ioctl command, without all the boring parameter parsing that is required in more generic tools like wr-dio-cmd. Because of this simplification of parameter passing, the pulse width is hardwired to 1ms.

The example is made up of two programs: wr-dio-agent and wr-dio-ruler (the former is a dumb actor, while the latter states the rules). To keep things simple the two programs assume that the SPEC is connected point-to-point to another SPEC and both carry the simple-DIO mezzanine.

Under this simplified assumptions, the ruler transmits raw Ethernet frames to the broadcast address, while the agent receives almost everything that appears on the cable. This choice allows plugging two SPEC cards in the same computer and run the example; if the example used an IP-based protocol (like UDP) it couldn’t be used with two cards on a single PC – and a fiber through them. The simplification above, however, most likely prevents the programs from working within a more complex network topology. I expect real White Rabbit users to add proper network addressing in their applications.

If you have a single SPEC card, you can still use the ruler by itself to mirror an input channel to an output channel of the same card, with a specified delay.

The agent program silently listens to the network interface and receives a data structure ready to be passed to ioctl. Its only command line argument is the name of the White Rabbit interface to use (for most users it is wr0):

   wr-dio-agent wr0
Chapter 4: The WR-NIC

The *ruler* command, on the other hand, waits for timestamps to appear on the specified input channel; when notified about a positive-going edge, it replicates the edge on one or more outputs. Each output can be local or remote, and can use a different delay from the input pulse.

If you lack an input signal, you can make an output pulse with *wr-dio-pps* or other means and use it as a trigger. Please note that the *ruler* does not configure the channel mode, so you might want to use the *mode* command of *wr-dio-cmd* in advance.

The following command waits for events on channel 0 of the card connected to *wr1*, and replicates the event with a delay of 1ms on channel 3 of both the local and the remote card; it also replicates with a 2ms delay to local channel 4. Please note that the delays should be no more than the interval between input pulses, because the tools reprograms all output triggers at each input event.

**Note:** As usual, channels are numbered 0 through 4.

```
wr-dio-ruler wr1 IN0 L3+0.001 R3+0.001 L4+0.002
```

There is no sample code that generates trains of pulses as a response to events, nor support for other than 1ms-long output pulses; anyways, the code is thoroughly commented in order to serve as a starting point for more complex lab environments.

As a final remark, please note that all pulse generation is driven by host software, after an hardware interrupt reports the input event. For this reason, you’ll not be able to reliably replicate pulses with delays smaller than a few hundred microseconds, depending on the processing power of your computer and the load introduced by other processes. For remote connections, you must also count the overhead of network communication as well as transmission delays over the fiber (a 10km fiber introduces a delay of 50 microseconds).

The following example shows use of the *ruler* and *agent* on two hosts, called *spusa* and *tornado*. The input events on *spusa* are replicated to one local channel and two remote channels, with a delay of 1ms. The input events in this case are from a *pulse-per-second* signal:

```
tornado.root# /tmp/wr-dio-agent wr0 &
spusa.root# wr-dio-ruler wr1 IN4 L3+.001 R4+.001 R2+.001
wr-dio-ruler: configured for local channel 3, delay 0.001000000
wr-dio-ruler: configured for remote channel 4, delay 0.001000000
wr-dio-ruler: configured for remote channel 2, delay 0.001000000
[... wait a few seconds ...]
spusa.root# wr-dio-cmd wr1 stamp 3
ch 3, 385.001000000
ch 3, 386.001000000
ch 3, 387.001000000
ch 3, 388.001000000

tornado.root# wr-dio-cmd wr0 stamp 2
ch 2, 385.001000000
ch 2, 386.001000000
ch 2, 387.001000000
ch 2, 388.001000000

tornado.root# wr-dio-cmd wr0 stamp 4
ch 4, 385.001000000
ch 4, 386.001000000
ch 4, 387.001000000
ch 4, 388.001000000
```

### 4.13 The Future of WR-NIC

The *grand plan* for this driver is to detach the NIC functionality from the mezzanine driver.

Some future version of this package will thus feature a different layout of the code, and the NIC will be a mezzanine-independent feature that may be activated on request of the mezzanine
driver – it’s the mezzanine driver that requests its own gateware to be loaded, and only that
driver can know whether or not NIC functionality is part of its feature set.

In any case, no change to user-space access is expected, because the current way to handle
mezzanine-specific ioctl commands is already portable to the new code arrangement.

5 User-Space Tools

The tools subdirectory of this package includes a few host-side programs that may be useful
while working with the SPEC device. This section does not describe the wr-nic specific tool;
for that see Section 4.9 [WR-NIC Command Tool], page 11.

They are all base on the same speclib, part of the same directory, so all of them accept some
parameters in common, in order to identify one specific SPEC card if you have more than one:

\[-b \text{ <bus>}\]

This option specifies the bus number

\[-d \text{ <devfn>}\]

This is used to specify the device and function number, but it is expected to be 0 on
most if not all the computers. You won’t generally need to specify the devfn value.

If no arguments are supplied, the tools act on the first device if more than one is plugged.
The meaning of first is actually undefined and system-dependent.

The tools currently available are:

specmem

The program acts like devmem or devmem2 but in a simplified way. It receives
one or two command line arguments: one for reading and two for writing. Both
arguments are used as hex numbers, whether or not the leading 0x is specified. The
program makes a single 32-bit access to BAR0 of the Gennum PCI bridge; the first
argument is the address, and the second argument is the value to be written. The
VERBOSE environment variable makes the tool slightly more verbose. If you pass -g
you will access the Gennum registers (for example, for GPIO access).

spec-cl

This is the cpu loader. It is not called lm32-loader to avoid confusion with other
tools we have been using. It loads a program at offset 0x80000 in BAR0. This
is where we usually have RAM memory for the soft-core running in the SPEC. If
the program lives at aa different address, you can pass -c <number> to specify a
different address (note that the leading 0x is required to pass an hex address).

spec-fwloader

This is a user-space loader for the gateware file. It simply receives a file name
argument (after the optional bus number for the device).

spec-vuart

A simple tool to talk with the virtual-uart device inside the SPEC. The default
base address for the peripheral is 0xe0500 but you can can change it passing -u
<address>.
6 Bugs and Missing Features

• Identification of the mezzanine is completely missing; every fmc driver at this point takes hold of every device. We are working on this, and the next version of spec-sw will support identification, with a flag to run without identification for users whose EEPROM has not been programmed.

• Both spec and wr-nic should have GPIO support with gpiolib; there is skeletal support but no real code for actual I/O. This is not a priority, just a wish list for better Linux integration.

• The NIC driver should directly support setting the White Rabbit mode for each card (grandmaster, free-running master or slave). This will be supported at module load time, not at runtime (for that please use the UART).

• DIO support in wr-nic is missing some of the features listed in ‘wr-dio.h’ (i.e. DAC control)

• The wr-nic functionality should be completely detached from the specific mezzanine. This is a longer-term desire.

• Locking in kernel code should be verified with a serious audit effort. There are no known issues at this point, but some code may be made safer.

7 Portability

This package should be portable. However I didn’t test it on a wide variety of systems. Currently most of my use is on a 32-bit x86 host, running version 3.4 of the kernel.

The complete package builds without any warning from version 2.6.37 up to 3.13 (I didn’t try later versions, yet). Frame timestamping changed seriously after 2.6.36, so the wr-nic.ko driver is not easily backward portable.

To allow use of the core spec driver, to drive custom mezzanines, the Makefile supports the configuration variable CONFIG_WR_NIC, which you may set to n before compiling:

```
export CONFIG_WR_NIC=n
```

With this in place, the package compiles without any warning on a 32-bit PC from version 2.6.30 onwards. By using the backport branch of the 2013-05 release you can build for all kernels back to 2.6.27 and also 2.6.24 (the one we were still using in production). Later releases have no associated backport branch.