Fine Delay User's Manual

July 2012 – Release 1.1

FMC Delay 1ns-4cha hardware and software manual

CERN BE-CO-HT / Tomasz Wlostowski, Alessandro Rubini

Table of Contents

Introduction 1

1 Repositories and Releases 1

2 Hardware Description 1
- 2.1 Requirements and Supported Platforms 1
- 2.2 Modes of Operation 2
- 2.3 Mechanical/Environmental 2
- 2.4 Electrical 2
- 2.5 Timing 3
- 2.6 Principles of Operation 4

3 Driver Features 5

4 Installation 5
- 4.1 Gateware Dependencies 5
- 4.2 Gateware Installation 5
- 4.3 Software Dependencies 6
- 4.4 Software Installation 6
- 4.5 Module Parameters 7

5 Source Code Conventions 7

6 Using the Driver Directly 8
- 6.1 The device 8
- 6.2 Device Attributes 9
  - 6.2.1 List of Commands to the Device 9
  - 6.2.2 Reading Board Time 10
  - 6.2.3 Writing Board Time 10
- 6.3 The Input cset 10
  - 6.3.1 Input Device Attributes 11
  - 6.3.2 Reading with zio-dump 11
  - 6.3.3 Reading with fd-raw-input 12
  - 6.3.4 Using fd-raw-perf 13
  - 6.3.5 Configuring the Input Channel 13
  - 6.3.6 Pulsing from the Parallel Port 13
- 6.4 The Output cset 13
  - 6.4.1 Using fd-raw-output 15

7 Using the Provided API 15
- 7.1 Initialization and Cleanup 15
- 7.2 Time Management 16
- 7.3 Input Configuration 17
- 7.4 Reading Input Time-stamps 17
- 7.5 Output Configuration 18
- 7.6 White-Rabbit Configuration 19

8 Calibration **20**

9 Known Bugs and Missing Features **20**

9.1 Bugs in Related Packages 20
9.2 Bugs in This Package 21
9.3 Wish List 21

10 Troubleshooting **21**

10.1 ZIO Doesn't Compile 21
10.2 make modules_install misbehaves 21
10.3 Wrong FPGA Image 21
10.4 Version Mismatch 22

Introduction

This is the user manual for the "fmc-delay-1ns-4cha" board developed on `ohwr.org`. Please note that the ohwr project is misnamed as `fmc-delay-1ns-8cha`; thus the web references include this wrong naming and it's not a typo in the documentation.

1 Repositories and Releases

This version of the software package is release 1.1. The code and documentation is distributed in the following places:

`http://www.ohwr.org/projects/fine-delay-sw/documents`
: This place hosts the pdf documentation for every official release.

`http://www.ohwr.org/projects/fine-delay-sw/files`
: Here we place the *.tar.gz* file for every release, including the *git* tree and compiled documentation (for those who lack TeX).

`git://ohwr.org/fmc-projects/fmc-delay-1ns-8cha/fine-delay-sw.git`
`git://gitorious.org/fine-delay/fine-delay.git`
: Read-only repositories for the software and documentation. The former is authoritative, the latter is a backup.

`git@ohwr.org:fmc-projects/fmc-delay-1ns-8cha/fine-delay-sw.git`
`git@gitorious.org:fine-delay/fine-delay.git`
: Read-write repositories, for those authorized. Again, OHWR is the authoritative place, but we tend to push to gitorious as well.

The repository has a tag called called `fine-delay-sw-v1.1`. The same tag is used in related repositories (*zio*, *spec-sw* and the hardware repository). Any official hot fixes, if any, for this release will live in the branch called `fine-delay-sw-v1.1-fixes`, in each repository.

Note: If you got this from the repository (as opposed to a named *tar.gz* or *pdf* file) it may happen that you are looking at a later commit than the tagged release: it is a fact of life that developers forget to re-read and fix documentation while updating the code. In that case, plase run "`git describe HEAD`" to ensure where you are.

2 Hardware Description

The *FMC Delay 1ns-4cha* is an FPGA Mezzanine Card (FMC - VITA 57 standard), whose main purpose is to produce pulses delayed by a user-programmed value with respect to the input trigger pulse. The card can also work as a Time to Digital converter (TDC) or as a programmable pulse generator triggering at a given TAI time.

For the sake of clarity of this document, the card's name will be further abbreviated as *FmcDelay*.

2.1 Requirements and Supported Platforms

FmcDelay can work with any VITA 57-compliant FMC carrier, provided that the carrier's FPGA has enough logic resources. So far, FmcDelay has been only tested with the CERN's SPEC (Simple PCI-Express Carrier) board and the provided drivers currently work only with that carrier. A VME version using the SVEC carrier is currently being developed and should be available soon.

In order to operate *FmcDelay*, the following hardware/software components are required:

- A standard PC with at least one free 4x (or wider) PCI-Express slot,

- A SPEC PCI-Express FMC carrier (supplied with the *FmcDelay*),
- 50-ohm cables with 1-pin LEMO 00 plugs for connecting the I/O signals,
- Any Linux (kernel 2.6 or 3.0+) distribution,

2.2 Modes of Operation

FmcDelay can work in one or more of the following modes:

- **Pulse Delay**: produces one or more pulse(s) on selected outputs a given time after an input trigger pulse (fig. 1a)
- **Pulse Generator**: produces one or more pulse(s) on selected outputs starting at an absolute time value programmed by the user (fig. 1b). In this mode, time base is usually provided by the White Rabbit network.
- **Time to Digital Converter**: tags all trigger pulses and delivers the timestamps to the user's application.

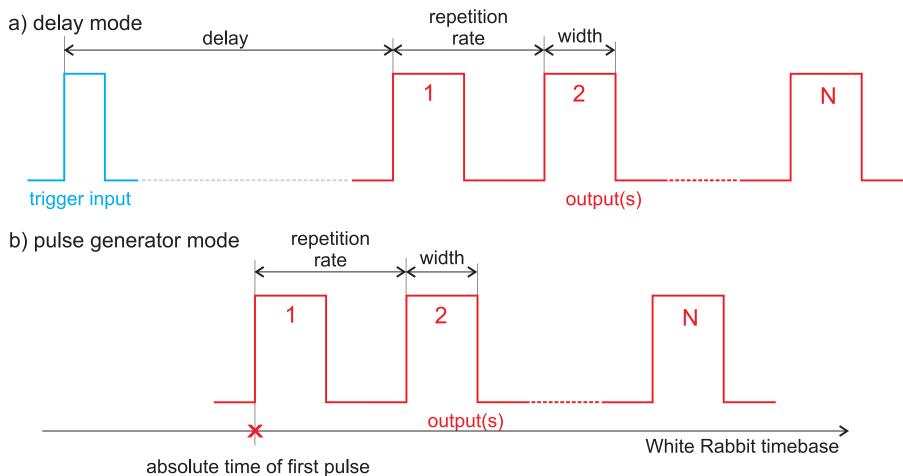


Fig. 1. *FmcDelay* operating modes.

Modes (pulse delay/generator) can be selected independently for each output. For example, one can configure the output 1 to delay trigger pulses by 1 us, and the output 2 to produce a pulse at the beginning of each second. The TDC mode can be enabled for the input at any time and does not interfere with the operation of the channels being time tagged.

2.3 Mechanical/Environmental

Mechanical and environmental specs:

- Format: FMC (VITA 57), with rear zone for conduction cooling
- Operating temperature range: 0 - 90 degC.
- Carrier connection: 160-pin Low Pin Count FMC connector

2.4 Electrical

Inputs/Outputs:

- 1 trigger input (LEMO 00)
- 4 pulse outputs (LEMO 00)
- 2 LEDs

- Carrier communication via 160-pin Low Pin Count FMC connector

Trigger input:

- TTL/LVTTL levels, DC-coupled. Reception of a trigger pulse is indicated by blinking the "TRIG" LED in the front panel.
- 2 kOhm or 50 Ohm input impedance (programmable via software). 50 Ohm termination is indicated by the "TERM" LED in the front panel.
- Power-up input impedance: 2 kOhm.
- Protected against short circuit, overcurrent (> 200 mA) and overvoltage (up to +28 V).
- Maximum input pulse edge rise time: 20 ns.

Outputs:

- TTL-compatible levels DC-coupled: Voh = 3 V, Vol = 200 mV (50 Ohm load), Voh = 6 V, Vol = 400 mV (high impedance).
- Output impedance: 50 Ohm (source-terminated)
- Rise/fall time: 2.5 ns (10%% - 90%%, 50 Ohm load)
- Power-up state: LOW (2 kOhm pulldown), guaranteed glitch-free.
- Protected against continuous short circuit, overcurrent and overvoltage (up to +28 V).

Power supply:

- Used power supplies: P12V0, P3V3, P3V3_AUX, VADJ (voltage monitor only).
- Typical current comsumption: 200 mA (P12V0) + 1.5 A (P3V3).
- Power dissipation: 7 W. Forced cooling is required.

2.5 Timing

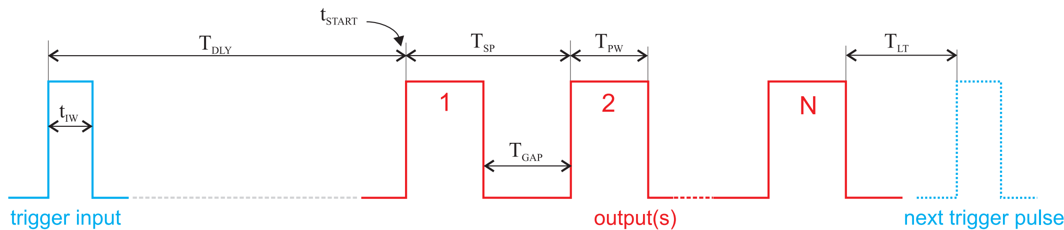


Fig. 2. *FmcDelay* timing parameter definitions.

Time base:

- Onboard oscillator accuracy: +/- 2.5 ppm (i.e. max. 2.5 ns error for a delay of 1 ms).
- When using White Rabbit as the timing reference: depending on the characteristics of the grandmaster clock and the carrier used. On SPEC v 4.0 FMC carrier, the accuracy is better than 1 ns.

Input timing:

- Minimum pulse width: t_{IW} = 50 ns. Pulses below 24 ns are rejected.
- Minimum gap between the last delayed output pulse and subsequent trigger pulse: T_{LT} = 50 ns
- Input TDC performance: 400 ps pp accuracy, 27 ps resolution, 70 ps trigger-to-trigger rms jitter (measured at 500 kHz pulse rate)

Output timing:

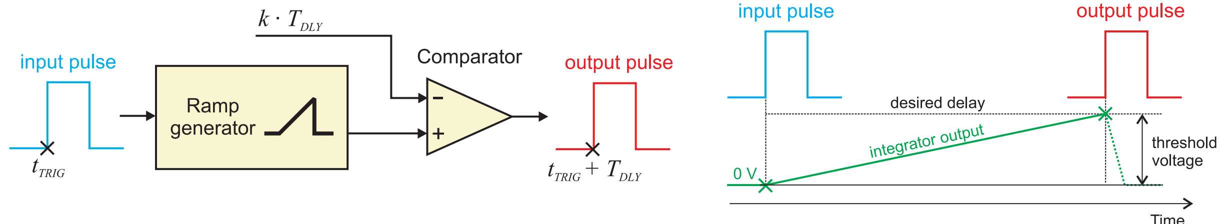
- Resolution: 10 ps.
- Accuracy (pulse generator mode): 300 ps.
- Train generation: trains of 1-65536 pulses or continuous square wave up to 10 MHz.
- Output-to-output jitter (outputs programmed to the same delay): 10 ps rms.
- Output-to-output jitter (outputs programmed to to different delays, worst case): 30 ps rms.
- Output pulse spacing (T_{SP}) : 100 ns - 16 s. Adjustable in 10 ps steps when both T_{PW}, T_{GAP} > 200 ns. Outside that range, T_{SP} resolution is limited to 4 ns.
- Output pulse start (t_{START}) resolution: 10 ps for the rising edge of the pulse, 10 ps for subsequent pulses if the condition above is met, otherwise 4 ns.

Delay mode specific parameters:

- Delay accuracy: < 1 ns.
- Trigger-to-output jitter: 80 ps rms.
- Trigger-to-output delay: minimum T_{DLY} = 500 ns, maximum T_{DLY} = 120 s.
- Maximum trigger pulse rate: $T_{DLY} + N * (T_{SP} + T_{GAP})$+ 100 ns, where N = number of output pulses.
- Trigger pulses are ignored until the output with the biggest delay has finished generation of the pulse(s).

2.6 Principles of Operation

Note: If you are an electronics engineer, you can skip this section, as you will most likely find it rather boring.



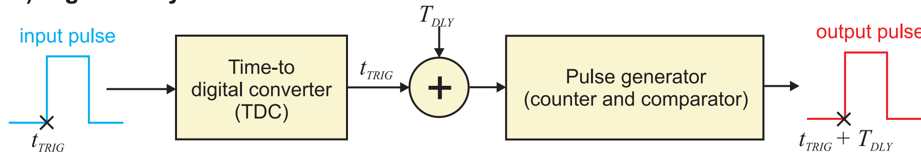


Fig. 3. Principle of operation of analog and digital delay generators.

Contrary to typical analog delay cards, which work by comparing an analog ramp triggered by the input pulse with a voltage proportional to the desired delay, *FmcDelay* is a digital delay generator, which relies on time tag arithmetic. The principle of operation of both generators is illustrated in figure 3.

When a trigger pulse comes to the input, *FmcDelay* first produces its' precise time tag using a Time-to-Digital converter (TDC). Afterwards, the time tag is summed together with the delay preset and the result is passed to a digital pulse generator. In its simplest form, it consists of a free running counter and a comparator. When the counter reaches the value provided on the input, a pulse is produced on the output. Note that in order for the system to work correctly,

both the TDC and the Pulse Generator must use exactly the same time base (not shown on the drawings).

Digital architecture brings several advantages compared to analog predecessors: Timestamps generated by the TDC can be also passed to the host system, and the Pulse Generators can be programmed with arbitrary pulse start times instead of $t_{TRIG} + T_{DLY}$. Therefore, *FmcDelay* can be used simultaneously as a TDC, pulse generator or a pulse delay.

3 Driver Features

This driver is based on ZIO and *spec-sw*. It supports initial setup of the board, setting and reading time, run-time continuous calibration, input timestamping and output pulse generation. It supports user-defined offsets, so our users can tell the driver about channel-specific delays (for example, to account for wiring) and ignore the issue in application code.

For each feature offered the driver (and documentation) tries to offer the following items; sometimes however one of them is missing for a specific driver functionality, if considered unneeded.

- A description of how the features works at low level;
- A low-level user-space program to test the actual mechanism;
- A C-language API to access the feature with data structures;
- An example program based on that API.

Additionally, the `NewLogger` directory includes the (uncommented, undocumented) program that has been used (at least for a while) in the Gran Sasso labs to log neutrino catches.

This package is currently available from `git://gnudd.com/fine-delay.git`, as well as `git://gitorious.org/fine-delay/fine-delay.git`. Snapshots are released on the "Files" and "Documents" tabs of the `ohwr` project for the related hardware and gateware.

4 Installation

This driver depends on two other drivers, as well as the Linux kernel. Also, it must talk to a specific FPGA binary file running in the carrier card.

4.1 Gateware Dependencies

This driver has been developed from the FPGA binary included in the package as `binaries/spec_top.bin`, which is White-Rabbit-enabled. You should use it with `binaries/wrc.bin`.

If the gateware is updated, I'll take care to always include in this package the exact binary the software is developed and verified against, until the *spec-sw* package will offer a hash-based approach and this package will use it.

4.2 Gateware Installation

To install the FPGA image in the target system, please follow the instruction in the documentation of *spec-sw*. To summarize, you'll need to place the `.bin` file, properly renamed, in */lib/firmware/fmc*.

If you have several *fine-delay* cards in the same host, you can install several copies of the binary, renamed to match the bus and slot number of the various SPEC cards, or you can use the default filename if there are no other SPEC cards (i.e. no cards hosting something else than the *fine-delay* module).

If you use the White-Rabbit version of the firmware, you also need the `wrc.bin` file, renamed as *spec-sw* looks for it. In my system, where the *fine-delay* card is at bus 2 slot 0, I use the following two files in */lib/firmware/fmc*:

```
-rw------- 1 root   root    58628 May  4 21:15 spec-B0002-cpu.bin
-rw------- 1 root   root  1485788 May  4 21:14 spec-B0002.bin
```

Also, please note that you must pass the parameter `lm32=0xc0000` to the *spec.ko* module, because the default address (0x80000) is not appropriate to the gateware of this device.

Note: if you are running a slightly earlier version of *spec-sw*, the files for the firmware loaded in *spec-sw* live in `/lib/firmware` instead of `/lib/firmware/fmc/`.

4.3 Software Dependencies

The kernel versions I used during development are 2.6.32 and 2.6.24 (in its *preempt-rt* incantation), because these are the ones where installed boards have been running.

The driver, then is based on the ZIO framework, available from `ohwr.org`. The version being used during development is a development version, back-ported to Linux-2.6.32 and 2.6.24. Similarly, this is a sub-module for the SPEC board, and thus relies on code from the *spec-sw* package, again from `ohwr.org`.

4.4 Software Installation

First of all, please note that you need to define the following environment variables to be able to compile this driver. All of them are assumed to be already set when running the commands shown.

`LINUX`

The top-level directory of the Linux kernel you are compiling against. This document assumes it is already set.

`ZIO`

The top-level directory of the ZIO repository checkout.

`SPEC_SW`

The top-level directory of the *spec-sw* repository checkout.

To install ZIO you should download it and install the tag or branch called fine-delay-sw-v1.1, which has been backported to work on Linux-2.6.24 and Linux-2.6.21.

The commands here are reported without prompt for easy cut-and-paste.

```
test -d zio/.git || git clone git://ohwr.org/misc/zio.git
cd zio
export ZIO=$(/bin/pwd)
git checkout -b fine-delay fine-delay-sw-v1.1
make
sudo make modules_install
```

The procedure for *spec-sw* is similar, and again the branch is called “fine-delay-sw-v1.1”:

```
test -d spec-sw/.git || \
    git clone git://ohwr.org/fmc-projects/spec/spec-sw.git
cd spec-sw
export SPEC_SW=$(/bin/pwd)
git checkout -b fine-delay fine-delay-sw-v1.1
cd kernel
make
sudo make modules_install
```

At this point all the software modules are ready to be loaded. Actually, the right set will be auto-loaded when you *modprobe* for `spec-fine-delay` if you installed everything.

Note: When loading `spec.ko` you **must** pass the argument `lm32=0xc000`. The script used by Tomasz is available as `tools/load.sh` for your reference.

This is an example of the kernel messages you'll get back over a few seconds (initializing the fine-delay card above, takes almost 4 seconds, including the calibration.

```
spec_probe (device 0002:0000)
spec_probe: current 2893 (modprobe)
spec 0000:02:00.0: PCI INT A -> GSI 18 (level, low) -> IRQ 18
spec_load_files
spec 0000:02:00.0: firmware: requesting spec-B0002.bin
spec_load_fpga: got binary file "spec-B0002.bin", 1485788 (0x16abdc) bytes
spec 0000:02:00.0: firmware: requesting spec-B0002-cpu.bin
spec_load_lm32: got program file "spec-B0002-cpu.bin", 59568 (0xe8b0) bytes
LM32 has been restarted
spec_load_submodule: load "spec-B0002": 256
fd_onewire_init: Found DS18xx sensor: 28:85:8c:61:03:00:00:0e
fd_read_temp: Scratchpad: 12:05:4b:46:7f:ff:0e:10:42
fd_read_temp: Temperature 0x512 (12 bits: 81.125)
fd_calibrate_outputs: ch1: 8ns @846 (f 811, off 35, t 81.12)
fd_calibrate_outputs: ch2: 8ns @854 (f 811, off 43, t 81.12)
fd_calibrate_outputs: ch3: 8ns @842 (f 811, off 31, t 81.12)
fd_calibrate_outputs: ch4: 8ns @846 (f 811, off 35, t 81.12)
spec_fine_delay: Found i2c device at 0x50
```

4.5 Module Parameters

The driver accepts a few load-time parameters for configuration. You can pass them to *insmod* directly, or write them in `/etc/modules.conf` or the proper file in `/etc/modutils/` .

The following parameters are used:

`regs=`

The *regs* parameter defaults to 0x80000 and is the offset of fine-delay registers within the PCI memory area. You shouldn't use it unless you are Tomasz Wlostowski or you otherwise changed the FPGA design

`verbose=`

The parameter defaults to 0. If set, it enables more diagnostic messages during probe.

`timer_ms=`

The period of the internal timer. The timer is used to poll for input events. We currently have no interrupt support so we must poll, but the parameter will remain even when interrupt is available, to disable it on request and reuse the current polling code. The interval by default is 10ms and currently only one timestamp is retrieved at each timer execution.

`calib_s=`

The period, in seconds, of temperature measurement to recalibrate the output delays. Defaults to 30. If set to zero, the timer is not activated.

5 Source Code Conventions

This is a random list of conventions I use in this package

- All internal symbols in the driver begin with `fd_` (excluding local variables like *i* and similar stuff). So you know if something is local or comes from the kernel.

- All library functions and public data begin with `fdelay_`.
- The board passed as a library token (`struct fdelay_board`) is opaque, so the user doesn't access it. Internally it is called `userb` because `b` is the real one being used. If you need to access library internals from a user file just define `FDELAY_INTERNAL` before including `fdelay-lib.h`.
- The driver header is called `fine-delay.h` while the user one is `fdelay-lib.h`. The latter includes the former, which user programs should not refer to. Both are different from the original implementation (`fdelay_lib.h`) to avoid trying to compile older code with new headers.
- The *tools* directory includes standalone tools that access ZIO directly. Their name begins with `fd-raw-` (but there is a non-fine-delay tool to generate pulses on the parallel port, which has a different name pattern).
- The *lib* directory includes example programs for the library. Unfortunately sources of programs and library files both begin with `fdelay-` – this is an overlook of mine but I won't fix it.

6 Using the Driver Directly

The driver is designed as a ZIO driver that offers 1 input channel and 4 output channels. Since each output channel is independent (they do not output at the same time) the device is modeled as 5 separate *csets*.

The reader of this chapter is expected to be confident with basic ZIO concepts, available in ZIO documentation (ZIO is an `ohwr.org` project).

6.1 The device

The overall device includes a few device attributes and a few attributes specific to the csets (some attributes for input and some attributes for output). The attributes allow to read and write the internal timing of the card, as well as other internal parameters, documented below. Since ZIO has no support for *ioctl*, all the attributes appear in *sysfs*. For multi-valued attributes (like a time tag, which is more than 32 bits) the order of reading and writing is mandated by the driver (e.g.: writing the seconds field of a time must be last, as it is the action that fires hardware access for all current values).

The device appears in */dev* as a set of char devices:

```
spusa# ls -l /dev/zio/*
crw------- 1 root root 249,   0 Apr 26 00:26 /dev/zio/fd-0200-0-0-ctrl
crw------- 1 root root 249,   1 Apr 26 00:26 /dev/zio/fd-0200-0-0-data
crw------- 1 root root 249,  32 Apr 26 00:26 /dev/zio/fd-0200-1-0-ctrl
crw------- 1 root root 249,  33 Apr 26 00:26 /dev/zio/fd-0200-1-0-data
crw------- 1 root root 249,  64 Apr 26 00:26 /dev/zio/fd-0200-2-0-ctrl
crw------- 1 root root 249,  65 Apr 26 00:26 /dev/zio/fd-0200-2-0-data
crw------- 1 root root 249,  96 Apr 26 00:26 /dev/zio/fd-0200-3-0-ctrl
crw------- 1 root root 249,  97 Apr 26 00:26 /dev/zio/fd-0200-3-0-data
crw------- 1 root root 249, 128 Apr 26 00:26 /dev/zio/fd-0200-4-0-ctrl
crw------- 1 root root 249, 129 Apr 26 00:26 /dev/zio/fd-0200-4-0-data
```

The actual pathnames depend on the version of *udev*, and the support library tries both names (the new one shown above, and the older one, without a `zio` subdirectory). Also, please note that a still-newer version of *udev* obeys device permissions, so you'll have read-only and write-only device files.

In this drivers, *cset* 0 is for the input signal, and *csets* 1..4 are for the output channels.

If more than one board is probed for, you'll have two or more similar sets of devices, differing in the *dev_id* field, i.e. the `0200` that follows the device name `zio-fd` in the stanza above. The

dev_id field is built using the PCI bus and the *devfn* octet; the example above refers to slot 0 of bus 2.

For remotely-controlled devices (e.g. Etherbone) the problem will need to be solved differently.

Device (and channel) attributes can be accessed in the proper *sysfs* directory. For a card in slot 0 of bus 2 (like shown above), the directory is */sys/bus/zio/devices/fd-0200*:

```
spusa# ls -Ff /sys/bus/zio/devices/fd-0200/
./      enable           utc-l      power/     fd-ch2/
../     resolution-bits  coarse     driver     fd-ch3/
uevent  version          command    fd-input/  fd-ch4/
name    utc-h            subsystem  fd-ch1/
```

6.2 Device Attributes

Device-wide attributes are the three time tags (*utc-h*, *utc-l*, *coarse*), a read-only *version* and a write-only *command*. To read device time you should read *utc-h* first. Reading ŭtc-h will atomically read all values from the card and store them in the software driver: when reading *utc-l* and *coarse* you'll get such cached values.

Example:

```
spusa# cd /sys/bus/zio/devices/fd-0200/
spusa# cat coarse coarse utc-h coarse
75136756
75136756
0
47088910
```

To set the time, you can write the three values leaving *utc-h* last: writing *utc-h* atomically programs the hardware:

```
spusa# echo 10000 > coarse; echo 10000 > utc-l; echo 0 > utc-h
spusa# cat utc-h utc-l
0
10003
```

If you write 0 to *command*, board time will be synchronized to the current Linux clock within one microsecond (reading Linux time and writing to the *fine-delay* registers is done with interrupts disabled, so the actual synchronization precision depends on the speed of your CPU and PCI bus):

```
spusa# cat utc-h utc-l; echo 0 > command; cat utc-h utc-l; date +%s
0
50005
0
1335948116
1335948116
```

However, please note that the times will diverge over time. Also, if you are using White-Rabbit mode, host time is irrelevant to the board.

I chose to offer a *command* channel, which is opaque to the user, because there are several commands that you may need to send to the device, and we need to limit the number of attributes. The command numbers are enumerated in `fine-delay.h` and described here below.

6.2.1 List of Commands to the Device

The following commands are currently supported for the `command` write-only file in *sysfs*:

`0 = FD_CMD_HOST_TIME`
: Set board time equal to host time.

`1 = FD_CMD_WR_ENABLE`
: Enable White-Rabbit mode.

`2 = FD_CMD_WR_DISABLE`
: Disable White-Rabbit mode.

`3 = FD_CMD_WR_QUERY`
: Tell the user the status of White-Rabbit mode. This is a hack, as the return value is reported using error codes. Success means White-Rabbit is synchronized. `ENODEV` means WR mode is inactive, `EAGAIN` means it is not synchronized yet. The error is returned to the *write* system call.

`4 = FD_CMD_DUMP_MCP`
: Force dumping to log messages (using a plain *printk* the GPIO registers in the MCP23S17 device.

`5 = FD_CMD_PURGE_FIFO`
: Empty the input fifo and reset the sequence number.

6.2.2 Reading Board Time

The program *fd-raw-gettime*, part of this package, allows reading the current board time using *sysfs* directly:

```
spusa# ./tools/fd-raw-gettime ; sleep 1; ./tools/fd-raw-gettime
3303.076543536
3304.082016080
```

In the example above the time has never been set, so the epoch if FPGA load time.

Note: the tool is bugged as of year 2038 because it assumes utc-h is 0.

6.2.3 Writing Board Time

The program *fd-raw-settime*, part of this package, allows setting the current board time using *sysfs* directly:

```
spusa# ./tools/fd-raw-settime  123; ./tools/fd-raw-gettime
123.000541696
spusa# ./tools/fd-raw-settime  123 500000000; ./tools/fd-raw-gettime
123.500570096
```

Note: the tool is bugged as of year 2038 because it assumes utc-h is 0.

No tool is there to sync the board to Linux time, because writing 0 to the *command* attribute is atomic by itself, but there is an example program using the official API (see Section 7.2 [Time Management], page 16).

6.3 The Input cset

The input cset returns fake data, with timestamp information in the control block (the meta-information associated to data). This is suboptimal, but it is a "good enough" first implementation until time permits to refine it.

Currently, no input timestamp is collected until some process calls the *read* function on the control or data char device.

In a perfect world we would have a custom *trigger* module that stuffs the timestamp information directly in the proper place within the ZIO control block. This version of the code uses the default ZIO trigger, which is user-driven. In other words, data is only requested to hardware if a user process is actually reading. This "software" trigger sticks a software timestamp in the control block, so the hardware timestamp must be provided elsewhere.

The hardware timestamp and other information is returned as *channel attributes*, which you can look at using *zio-dump* (part of the ZIO package) or *tools/fd-raw-input* which is part of this package.

6.3.1 Input Device Attributes

The attributes are all 32-bit unsigned values, and their meaning is defined in *fine-delay.h* for libraries/applications to use them:

```
enum fd_zattr_in_idx {
        FD_ATTR_TDC_UTC_H,
        FD_ATTR_TDC_UTC_L,
        FD_ATTR_TDC_COARSE,
        FD_ATTR_TDC_FRAC,
        FD_ATTR_TDC_SEQ,
        FD_ATTR_TDC_CHAN,
        FD_ATTR_TDC_FLAGS,
        FD_ATTR_TDC_OFFSET,
        FD_ATTR_TDC_USER_OFF,

};
/* Names have been chosen so that 0 is the default at load time */
#define FD_TDCF_DISABLE_INPUT 1
#define FD_TDCF_DISABLE_TSTAMP 2
#define FD_TDCF_TERM_50 4
```

The attributes are also visible in */sys*, in the directory describing the cset:

```
spusa# ls -Ff /sys/bus/zio/devices/fd-0200/fd-input/
./      enable           utc-l   chan         power/
../     current_trigger  coarse  flags        trigger/
uevent  current_buffer   frac    offset       chan0/
name    utc-h            seq     user-offset
```

The timestamp-related values in this file reflect the last stamp that has been enqueued to user space (this may be the next event to be read by the actual reading process).

The *offset* attribute is the stamping offset, in picoseconds, for the TDC channel. The *user-offset* attribute, which currently defaults to 0, is a signed value that users can write to represent a number of picoseconds to be added (or subtracted) to the hardware-reported stamps. This is used to account for delays induced by cabling (range: -2ms to 2ms).

The *flags* attribute can be used to change three configuration bits, defined by the respective macros. Please note that the default at module load time is zero: some of the flags bits are inverted over the hardware counterpart, but the `DISABLE` in flag names is there to avoid potential errors.

6.3.2 Reading with zio-dump

This is an example read sequence using *zio-dump*: data must be ignored and only the first few extended attributes are meaningful. This can be used to see low-level details, but please note that the programs in `tools/` and `lib/` in this package are in general a better choice to timestamp input pulses.

```
spusa# zio-dump /dev/zio/fd-0200-0-0-*
Ctrl: version 0.5, trigger user, dev fd, cset 0, chan 0
Ctrl: seq 1, n 16, size 4, bits 32, flags 01000001 (little-endian)
Ctrl: stamp 1335737285.312696982 (0)
Device attributes:
    [...]
    Extended: 0x0000003f
    0x0 0x30 0x640f20d 0x60a 0x0 0x0 0x0 0x0
    [...]
    Extended: 0x0000003f
    0x0 0x40 0x454b747 0x1d3 0x1 0x0 0x0 0x0
    [...]
```

```
Extended: 0x0000003f
0x0 0x47 0xf04c57 0x772 0x2 0x0 0x0 0x0
```

6.3.3 Reading with fd-raw-input

The *tools/fd-raw-input* program, part of this package, is a low-level program to read input events. It reads the control devices associated to *fine-delay* cards, ignoring the data devices which are known to not return useful information. The program can receive file names on the command line, but reads all fine-delay devices by default – it looks for filenames in */dev* using *glob* patterns (also called "wildcards").

This is an example run:

```
spusa# ./tools/fd-raw-input
/dev/zio/zio-fd-0200-0-0-ctrl: 00000000 0000001a 00b9be2b 00000bf2 00000000
/dev/zio/zio-fd-0200-0-0-ctrl: 00000000 0000001b 00e7f5c2 0000097d 00000001
/dev/zio/zio-fd-0200-0-0-ctrl: 00000000 0000001b 02c88901 00000035 00000002
/dev/zio/zio-fd-0200-0-0-ctrl: 00000000 0000001b 03e23c26 000006ce 00000003
```

The program offers a "float" mode, that reports floating point time differences between two samples (this doesn't use the *frac* delay value, though, but only the integer second and the coarse 8ns timer).

This is an example while listening to a software-generated 1kHz signal:

```
spusa# ./tools/fd-raw-input -f
/dev/zio/zio-fd-0200-0-0-ctrl:    1825.903957552 (delta   0.001007848)
/dev/zio/zio-fd-0200-0-0-ctrl:    1825.904971384 (delta   0.001013832)
/dev/zio/zio-fd-0200-0-0-ctrl:    1825.905968648 (delta   0.000997264)
/dev/zio/zio-fd-0200-0-0-ctrl:    1825.906980376 (delta   0.001011728)
/dev/zio/zio-fd-0200-0-0-ctrl:    1825.907997128 (delta   0.001016752)
```

The tool reports lost events using the sequence number (attribute number 4). This is an example using a software-generated burst with a 10us period:

```
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.385815880 (delta   0.000010024)
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.385825832 (delta   0.000009952)
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.385835720 (delta   0.000009888)
/dev/zio/zio-fd-0200-0-0-ctrl: LOST 2770 events
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.412775304 (delta   0.026939584)
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.412784808 (delta   0.000009504)
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.412794808 (delta   0.000010000)
/dev/zio/zio-fd-0200-0-0-ctrl:    1958.412804184 (delta   0.000009376)
```

The "pico" mode of the program (command line argument `-p`) is used to get input time stamps with picosecond precision. In this mode the program doesn't report the "second" part of the stamp. This is an example run of the program, fed by 1kHz generated from the board itself:

```
spusa.root# ./tools/fd-raw-input -p | head -5
/dev/zio/zio-fd-0800-0-0-ctrl: 642705121635
/dev/zio/zio-fd-0800-0-0-ctrl: 643705121647 - delta 001000000012
/dev/zio/zio-fd-0800-0-0-ctrl: 644705121656 - delta 001000000009
/dev/zio/zio-fd-0800-0-0-ctrl: 645705121647 - delta 000999999991
/dev/zio/zio-fd-0800-0-0-ctrl: 646705121664 - delta 001000000017
```

If is possible, for diagnostics purposes, to run several modes at the same time: while `-f` and `-p` disable raw/hex mode, the equivalent options `-r` and `-h` reinstantiate it. If the input event is reported in more than one format, the filename is only printed once, and later lines begin with a single blank space (you may see more blanks because they are part of normal output, for alignment purposes).

Finally, the program uses two environment variables, if set to any value: `FD_SHOW_TIME` make the tool report the time difference between sequential reads, which is mainly useful to debug the driver workings; `FD_EXPECTED_RATE` makes the tool report the difference from the expected data rate, relative to the first sample collected:

```
spusa.root# FD_EXPECTED_RATE=1000000000 ./tools/fd-raw-input -p | head -5
/dev/zio/zio-fd-0800-0-0-ctrl: 139705121668
```

```
/dev/zio/zio-fd-0800-0-0-ctrl: 140705121699 - delta 001000000031 - error  31
/dev/zio/zio-fd-0800-0-0-ctrl: 141705121661 - delta 000999999962 - error  -7
/dev/zio/zio-fd-0800-0-0-ctrl: 142705121671 - delta 001000000010 - error   3
/dev/zio/zio-fd-0800-0-0-ctrl: 143705121689 - delta 001000000018 - error  21
```

Please note that the expected rate is a 32-bit integer, so it is limited to 4ms; moreover it is only used in "picosecond" mode.

6.3.4 Using fd-raw-perf

The program *tools/fd-raw-perf* gives trivial performance figures for a train of input pulses. It samples all input events and reports some statistics when a burst completes (i.e., no pulse is received for at least 300ms):

```
spusa#  ./tools/fd-raw-perf
59729 pulses (0 lost)
   hw: 1000000000ps (1.000000kHz) -- min 999999926 max 1000000089 delta 163
   sw: 983us (1.017294kHz) -- min 7 max 18992 delta 18985
```

The program uses the environment variable `PERF_STEP`, if set, to report information every that many seconds, even if the burst is still running:

spusa.root# PERF_STEP=5 ./tools/fd-raw-perf

```
4999 pulses (0 lost)
   hw: 1000000000ps (1.000000kHz) -- min 999999933 max 1000000067 delta 134
   sw: 1000us (1.000000kHz) -- min 8 max 10001 delta 9993

4999 pulses (0 lost)
   hw: 1000000000ps (1.000000kHz) -- min 999999926 max 1000000081 delta 155
   sw: 1000us (1.000000kHz) -- min 7 max 18995 delta 18988
```

6.3.5 Configuring the Input Channel

There is no support in *tools/* to change channel configuration (but see Section 7.3 [Input Configuration], page 17 for the official API). The user is expected to write values in the *flags* file directly. For example, to enable the termination resistors, write 4 to the *flags* file in *sysfs*.

6.3.6 Pulsing from the Parallel Port

For my initial tests, some of which are shown above, I generated bursts of pulses with a software program (later I used the board itself, for a much better precision). To do so, I connected a pin of a parallel port plugged on the PCI bus to the input channel of the *fine-delay* card.

The program *tools/parport-burst*, part of this package, generates a burst according to three command line parameters: the I/O port of the data byte of the parallel port, the repeat count and the duration of each period. This example makes 1000 pulses of 100 usec each, using the physical address of my parallel port (if yours is part of the motherboard, the address is `378`):

```
./parport-burst d080 1000 100
```

6.4 The Output cset

The output channels need some configuration to be provided. This is done using attributes. Attributes can either be written in *sysfs* or can be passed in the control block that accompanies data.

This driver defines the sample size as 4 bytes and the trigger should be configured for a 1-sample block (the library does it at open time). We should aim at a zero-size data block, but this would require a patch to ZIO, and I'd better not change version during development.

The output is configured and activated by writing a control block with proper attributes set. Then a write to the data channel will push the block to hardware, for it to be activated.

The driver defines the following attributes:

```
/* Output ZIO attributes */
enum fd_zattr_out_idx {
        FD_ATTR_OUT_MODE = FD_ATTR_DEV__LAST,
        FD_ATTR_OUT_REP,
        /* Start (or delay) is 4 registers */
        FD_ATTR_OUT_START_H,
        FD_ATTR_OUT_START_L,
        FD_ATTR_OUT_START_COARSE,
        FD_ATTR_OUT_START_FINE,
        /* End (start + width) is 4 registers */
        FD_ATTR_OUT_END_H,
        FD_ATTR_OUT_END_L,
        FD_ATTR_OUT_END_COARSE,
        FD_ATTR_OUT_END_FINE,
        /* Delta is 3 registers */
        FD_ATTR_OUT_DELTA_L,
        FD_ATTR_OUT_DELTA_COARSE,
        FD_ATTR_OUT_DELTA_FINE,
        /* The two offsets */
        FD_ATTR_OUT_DELAY_OFF,
        FD_ATTR_OUT_USER_OFF,
        FD_ATTR_OUT__LAST,
};
enum fd_output_mode {
        FD_OUT_MODE_DISABLED = 0,
        FD_OUT_MODE_DELAY,
        FD_OUT_MODE_PULSE,
};
```

To disable the output, you must assign 0 to the mode attribute and other attributes are ignored. To configure pulse or delay, all attributes must be set to valied values.

Note: writing the output configuration (mode, rep, start, end, delta) to *sysfs* is not working with this version of ZIO. And I've been too lazy to add code to do that. While recent developments in ZIO introduced more complete consistency between the various places where attributes live, with this version you can only write these attributes to the control block.

The *delay-offset* attribute represents an offset that is subtracted from the user-requested delay (*start* fields) when generating output pulses. It represents internal card delays. The value can be modified from *sysfs*.

Note: the *delay-offset* is used for delay mode but not for pulse-generation mode.

The *user-offset* attribute, which currently defaults to 0, is a signed value that users can write to represent a number of picoseconds to be added (or subtracted) to the every user-command (for both delay and pulse generation). This is used to account for delays induced by cabling (range: -2ms to 2ms). The value can be modified from *sysfs*.

This is the unsorted content of the *sysfs* directory for each of the output csets:

```
spusa# ls -fF /sys/bus/zio/devices/zio-fd-0200/fd-ch1
./               mode          end-l          user-offset
../              rep           end-coarse     power/
uevent           start-h       end-fine       trigger/
name             start-l       delta-l        chan0/
enable           start-coarse  delta-coarse
current_trigger  start-fine    delta-fine
current_buffer   end-h         delay-offset
```

As said, only *delay-offset* and *user-offset* are designed to be read and written by the user. Additionally, *mode* can be read to know whether the channel output or delay event has triggered. As of this version, the other attributes are not readable nor writable in *sysfs* – they are meant to be used in the control block written to */dev*.

6.4.1 Using fd-raw-output

The simplest way to generate output is using the tools in `lib/`. You are therefore urged to skip this section and read Section 7.5 [Output Configuration], page 18 instead.

For the bravest people, the low level way to generate output is using *fd-raw-output*, part of the *tools* directory of this package. The tool writes a control block to the ZIO control file, setting the block size to 1 32-bit sample; it then writes 4 bytes to the data file to force output of the attributes.

The tool acts on channel 1 (the first) by default, but uses the environment variable `CHAN` if set. All arguments on the command line are passed directly in the attributes. Thus, it is quite a low-level tool.

To help the user, any number that begins with `+` is added to the current time (in seconds). It is thus recommended to set the card to follow system time.

The following example sets card time to 0 and programs 10 pulses at the beginning of the next second. The pulses are 8usec long and repeat after 16usec. The next example runs 1s of 1kHz square wave. For readability, numbers are grouped as *(mode, delay), (start – utc-h, utc-l, coarse, frac), (stop – utc-h, utc-l, coarse, frac), (delta - utc-l, coarse, frac)*.

```
spusa# ./tools/fd-raw-settime 0 0; \
       ./tools/fd-raw-output 2 10   0 1 0 0   0 1 1000 0   0 2000 0

spusa# ./tools/fd-raw-settime 0 0; \
       ./tools/fd-raw-output 2 500   0 1 0 0   0 1 62500 0   0 125000 0
```

The following example sets board time to host time and programs a single 40us pulse at the beginning of the next second (note use of `+`)

```
spusa# echo 0 > /sys/bus/zio/devices/fd-*/command; \
       ./tools/fd-raw-output 2 0   0 +1 0 0   0 +1 5000 0
```

The following example programs a pps pulse (1ms long) on channel 1 and a 1MHz square wave on channel 2, assuming board time is already synchronized with host time:

```
spusa# CHAN=1 ./tools/fd-raw-output 2 -1   0 +1 0 0   0 +1 125000 0  1 0 0; \
       CHAN=2 ./tools/fd-raw-output 2 -1   0 +1 0 0   0 +1 64 0   0 125 0
```

7 Using the Provided API

This chapter describes the higher level interface to the board, designed for user applications to use. The code lives in the *lib* subdirectory of this package. The directory uses a plain Makefile (not a Kbuild one) so it can be copied elsewhere and compiled stand-alone. Only, it needs a copy of `fine-delay.h` (which it currently pulls from the parent directory) and the ZIO headers, retrieved using the `ZIO` environment variable).

7.1 Initialization and Cleanup

The library offers the following structures and functions:

`struct fdelay_board;`

This is the "opaque" token that is being used by library clients. If you want to see the internals define `FDELAY_INTERNAL` and look at *fdelay_list.c*.

`int fdelay_init(void);`
`void fdelay_exit(void);`

The former function allocates its internal data and returns the number of boards currently found on the system. The latter releases any allocated data. If *init* fails, it returns -1 with a proper `errno` value. If no boards are there it returns 0. You should not load or unload drivers between *init* and *exit*.

`struct fdelay_board *fdelay_open(int index, int dev_id);`
`int fdelay_close(struct fdelay_board *);`

The former function opens a board and returns a token that can be used in subsequent calls. The latter function undoes it. You can refer to a board either by index or by `dev_id`. Either argument (but not both) may be -1. If both are different from -1 the index and dev_id must match. If a mismatch is found, the function return NULL with `EINVAL`; if either index or `dev_id` are not found, the function returns NULL with `ENODEV`.

`struct fdelay_board *fdelay_open_by_lun(int lun);`

The function opens a pointer to a board, similarly to *fdelay_open*, but it uses the Logical Unit Number as argument instead. The LUN is used internally be CERN libraries, and the function is needed for compatibility with the installed tool-set. The function uses a symbolic link in *dev*, created by the local installation procedure.

The sample program *fdelay-list* lists the boards currently on the system, using *fdelay_init*:

```
spusa# ./lib/fdelay-list
./lib/fdelay-list: found 1 boards
  dev_id 0200, /dev/fd-0200, /sys/bus/zio/devices/fd-0200
```

7.2 Time Management

These are the primitives the library offers for time management.

`struct fdelay_time;`

The structure has the same fields as the one in the initial user-space library. All but *utc* are unsigned 32-bit values whereas they were different types in the first library.

`int fdelay_set_time(struct fdelay_board *b, struct fdelay_time *t);`
`int fdelay_get_time(struct fdelay_board *b, struct fdelay_time *t);`

The functions are used to set board time from a user-provided time, and to retrieve the current board time to user space. The functions return 0 on success. They only use the fields *utc* and *coarse* of `struct fdelay_time`.

`int fdelay_set_host_time(struct fdelay_board *b);`

The function sets board time equal to host time. The precision should be in the order of 1 microsecond, but will drift over time.

The current API clearly lacks a function to report the offset from board-time and system-time.

The program *fdelay-board-time* is a command-line front-end to the library, to validate the library works as expected:

```
spusa# ./lib/fdelay-board-time  25.5; ./lib/fdelay-board-time get
25.500661824
spusa# ./lib/fdelay-board-time  get
34.111048968
spusa# ./lib/fdelay-board-time  host
spusa# ./lib/fdelay-board-time  get
1335974946.493415600
```

7.3 Input Configuration

To configure the input channel for a board, the library offers the following function and macros:

`int fdelay_set_config_tdc(struct fdelay_board *b, int flags);`
`int fdelay_get_config_tdc(struct fdelay_board *b);`

The function configures a few options in the input channel. The *flags* argument is a bitmask of the following three values (note that 0 is the default at initialization time). The function returns -1 with `EINVAL` if the *flags* argument includes undefined bits.

`FD_TDCF_DISABLE_INPUT`
`FD_TDCF_DISABLE_TSTAMP`
`FD_TDCF_TERM_50`

The first bit disables the input channel, the second disables acquisition of time-stamps, and the last enables the 50-ohm termination on the input channel.

The example program *fdelay-term* demonstrates use of the function. It just enables or disables the 50-ohm resistor. The effect is usually verifiable by hooking a scope to the input signal:

7.4 Reading Input Time-stamps

The library offers the following functions that deal with the input stamps:

`int fdelay_fread(struct fdelay_board *b, struct fdelay_time *t, int n);`

The function behaves like *fread*: it tries to read all samples, even if it implies sleeping several times. Use it only if you are aware that all the expected pulses will reach you.

`int fdelay_read(struct fdelay_board *b, struct fdelay_time *t, int n,` int flags);

The function behaves like *read*: it will wait at most once and return the number of samples that it received. The *flags* argument is used to pass 0 or `O_NONBLOCK`. If a non-blocking read is performed, the function may return -1 with `EAGAIN` if nothing is pending in the hardware FIFO.

`int fdelay_fileno_tdc(struct fdelay_board *b);`

This returns the file descriptor associated to the TDC device, so you can *select* or *poll* before calling *fdelay_read*. If access fails (e.g., for permission problems), the functions returns -1 with `errno` properly set.

There are two example programs here: one using *read* and one using *fread*.

Unfortunately, even if there are samples pending, *read* will only return one of them, because the ZIO device will only see the next sample slightly after returning the previous one. This is a buffering problem with our use of ZIO. Here below there were three stamps enqueued, 1ms spaced in time:

```
spusa# ./lib/fdelay-read 10
./lib/fdelay-read: reading 10 pulses in blocking mode... got 1 of them
seq   179: time 1447.218417376 + 0ebf
./lib/fdelay-read: reading 10 pulses in non-blocking mode... got 1 of them
seq   180: time 1447.219415872 + 07b5

spusa# ./lib/fdelay-read 10
./lib/fdelay-read: reading 10 pulses in blocking mode... got 1 of them
seq   181: time 1447.220418000 + 0187
./lib/fdelay-read: reading 10 pulses in non-blocking mode... got -1 of them
```

This is an example with *fread*, where it received two bursts of 5 pulses: the function didn't return before getting all 10 of them:

```
spusa# ./lib/fdelay-fread 10
./lib/fdelay-fread: reading 10 pulses using fread... got 10 of them
seq   182: time 1587.441758984 + 0a4f
seq   183: time 1587.442757840 + 03b1
seq   184: time 1587.443757712 + 08f3
seq   185: time 1587.444757616 + 0a71
seq   186: time 1587.445757344 + 0480
seq   187: time 1592.530255160 + 05b7
seq   188: time 1592.531253896 + 0173
seq   189: time 1592.532253704 + 0db6
seq   190: time 1592.533253672 + 0646
seq   191: time 1592.534253336 + 0752
```

There is no example for *fdelay_fileno_tdc* using *select*.

7.5 Output Configuration

The library offers the following functions for output configuration:

`int fdelay_config_pulse(board, channel, pulse_cfg);`
`int fdelay_config_pulse_ps(board, channel, pulse_ps_cfg);`

The two functions configure the channel (numbered 0..3) for pulse of delay mode. The former function receives `struct fdelay_pulse` (with split utc/coarse/frac times) while the latter receives `struct fdelay_pulse_ps`, with picosecond-based time values. The functions return 0 on success, -1 and an error code in `errno` in case of failure.

`int fdelay_has_triggered(struct fdelay_board *b, int channel);`

The funcion returns 1 of the output channel (numbered 0..3) has triggered since the last configuration request, 0 otherwise.

The configuration functions receive a time configuration. The starting time is passed as `struct fdelay_time`, while the pulse end and loop period are passed using either the same structure or a scalar number of picoseconds. These are the relevant structures:

```
struct fdelay_time {
        uint64_t utc;
        uint32_t coarse;    uint32_t frac;
        uint32_t seq_id;    uint32_t channel;
};

struct fdelay_pulse {
        int mode;           int rep;    /* -1 == infinite */
        struct fdelay_time start, end, loop;
};

struct fdelay_pulse_ps {
        int mode;          int rep;
        struct fdelay_time start;
        uint64_t length, period;
};
```

The `rep` field represents the repetition count, to output a train of pulses. The mode field is one of `FD_OUT_MODE_DISABLED`, `FD_OUT_MODE_DELAY`, `FD_OUT_MODE_PULSE`.

The example program to test output generation is called `lib/fdelay-pulse`, and receives several arguments. The invocation pattern is the following:

```
fdelay-pulse [-w] [<dev>] <mode> <ch> <rep> <t1> <t2> <t3>"
```

The meaning of the arguments is as follows:

`[-w]`

The switch requests the program to wait for the trigger to happen before it returns. If missing, the program returns immediately.

`[<dev>]`

The optional device number is needed if more than one card is plugged in the computer. The form is `0200` (bus 02, slot 00).

`<mode>`

A string: `disable`, `pulse` or `delay`.

`<ch>`

The channel number: 0 through 3.

`<rep>`

The repetition count: how many pulses to output: -1 means forever.

`<t1>`
`<t2>`
`<t3>`

The three times specifying the pulse burst: *t1* is the starting time, *t2* is the pulse length and *t3* is the period of the square wave (thus, *t1* is absolute and the others are relative times.

The syntax to specify times is of the form `second.micro+pico`, where the *seconds.micro* part resembles a floating point number and *pico* is scalar. This syntax has been chosen to split the decimal part in two fields of 6 digits and ease readability. If the *seconds* field begins with `+`, the current host utc seconds are added to the number.

The following example programs a pps pulse (1ms long) on channel 0 and a 1MHz square wave on channel 1, assuming board time is already synchronized with host time:

```
spusa# ./lib/fdelay-pulse  pulse 0 -1  +2.0  0.001    1.0       ;  \
       ./lib/fdelay-pulse  pulse 1 -1  +2.0  0.0+500  0.000001
```

This example outputs a train of pulses, 100us long every 1ms - 10ps:

```
./lib/fdelay-pulse  pulse 0 -1  +2.0  0.0001 0.000999+999990
```

This final example uses delay functionality, requesting 5 pulses on channel 0, 10ms long with a delay of 100ms from the input pulses and 100ms period. At the same time channel 1 outputs 6 pulses, 10ms long with a delay of 150ms; period is again 100ms (it's something pretty clear to looks at on the scope).

```
./lib/fdelay-pulse delay 0 5   0.10 0.01 0.1; \
./lib/fdelay-pulse delay 1 6   0.15 0.01 0.1; \
```

7.6 White-Rabbit Configuration

The following functions are offered:

`int fdelay_wr_mode(struct fdelay_board *b, int on);`
The function receives 0 to disable WR mode or non-0 to enable it. It is expected to never fail if the driver is loaded.

`int fdelay_check_wr_mode(struct fdelay_board *b);`
The function returns 0 if the WR slave is synchronized, `EAGAIN` (an integer) if it is enabled by not yet synchronized and `ENODEV` if WR-mode is currently disabled.

8 Calibration

Calibration data for a fine-delay card is stored in the I2C FMC EEPROM device. The FMC standard reserves the initial part of the memory for its own data structures, so our calibration is stored at offset 0x1800 (6kB over a size of 8kB).

Future releases of this driver are expected to use SDB for describe flash layout, but the offset of 6kB will be retained for compatibility.

The driver automatically loads calibration data from the flash at initialization time, but only uses it if its has is valid. The calibration data is in `struct fd_calib` and the on-eeprom structure is `fd_calib_on_eeprom`; both are on show in '`fine-delay.h`'.

If the hash of the data structure found on eeprom is not valid, the driver will use the compile-time default values. You can act on this configuration using a number of module parameters; please note that changing calibration data is only expected to happen at production time.

`calibration_check`
: This integer parameter, if not zero, makes the driver dump the binary structure of calibration data in a few places during initialization. It is mainly a debug tool.

`calibration_default`
: The user can set this integer parameter to tell the driver to ignore calibration data found on the EEPROM, thus enacting the compile-time defaults.

`calibration_load`
: This parameter is a string. The name is used to ask the *firmware loader* to retrieve a file from '`/lib/firmware`'. The driver tries both the name it receives, and if that fails a name with bus and *devfn* number appended – so you can reconfigure a card even if more than one of them is plugged on the same host. The data, once read, is used only if the size is correct (0x54 == 84).

`calibration_save`
: The integer parameter is used to request saving calibration data to EEPROM, whatever values are active after the other parameters have been used. You can thus save the compiled-in default, the content of the firmware file just loaded, or the value you just read from EEPROM – not useful, but not denied either.

This package offers no tool to generate the binary file (the 84 bytes of calibration).

9 Known Bugs and Missing Features

This package is still work in progress, and unfortunately the same applies to the packages it depends on – ZIO and *spec-sw*.

9.1 Bugs in Related Packages

The current package set (i.e., *zio*, *spec-sw* and this one) has the following known issues exposed by *fine-delay*:

- The auto-loading of *spec* submodules is not really working: the *modprobe* command will be stuck sleeping if you try to use it. Run '`insmod spec-fine-delay`" (or *modprobe*) by hand instead. *This will be addressed in an upcoming release of spec-sw.*
- The *user* trigger of ZIO is really user-driven, so the driver can't push stuff to the buffer until asked to. Also, a related buglet prevents to return data immediately when asked. This will be fixed, but it currently results in the *read* function only returning one sample, and an immediately-following non-blocking *read* will say nothing is there, yet.

9.2 Bugs in This Package

This is the list of known bugs and missing features over what hardware allows:

- We should use interrupts. The input is currently performed with a kernel timer.
- Calibration information in the EEPROM is not fixed for endianness, so it only works on hosts of the same endiannes as the one where it has been programmed.
- We need a module parameter to avoid probing non-fine-delay SPEC cards. Reading the magic number from an SPEC that is not programmed (or likely that is programmed with a different gateware) may lock up the system. *Actually, this is going to be addressed by a new approach to loading gateware, set forth in the upcoming spec-sw.*

9.3 Wish List

Other less important issues may be dealt with over time, but are not urgent as I write this:

- The driver should register its own ZIO trigger, or use the new attribute for "greedy-input" planned in new versions of ZIO (thank you Federico). Currently there's no buffering and reading is slower than it could be.
- Most example programs only use the "first" board in the system.
- Kernel messages are not very consistent: the code uses both *dev_info* and plain *kern_info*.
- There is clear inconsistency between the various uses of bus and *devfn* numbers, in firmware naming and *sysfs* naming. The inconsistency is mainly between this driver and the SPEC driver, but this one is more correct.

10 Troubleshooting

This chapters lists a few errors that may happen and how to deal with them.

10.1 ZIO Doesn't Compile

Compilation of ZIO ma fail with error like:

```
zio-ad788x.c:180: error: implicit declaration of function "spi_async_locked"
```

This happens because the function wasn't there in your older kernel version, and your system is configured to enable `CONFIG_SPI`.

To fix, please just remove the *zio-ad788x* line from `drivers/Makefile`.

10.2 make modules_install misbehaves

The command *sudo make modules_install* may place the modules in the wrong directory or fail with an error like:

```
make: *** /lib/modules/2.6.37+/build: No such file or directory.
```

This happens when you compiled by setting `LINUX=` and your *sudo* is not propagating the environment to its child processes. In this case, you should run this command instead

```
sudo make modules_install  LINUX=$LINUX
```

10.3 Wrong FPGA Image

Thanks to Tomasz, the *fine-delay* FPGA binary has a magic number at a magic address, so the driver can detect if the *spec* includes no gateware or a different binary image.

In this case, a message like the following one is reported:

```
fd_probe: card at 0002:0000 has wrong gateware
```

If this happens, please put the binary image in */lib/firmware* where the spec driver can find it. In my case the file name is `spec-B0002.bin` (refer to *spec-sw* documentation for details.

10.4 Version Mismatch

The *fdelay* library may report a version mismatch like this:

```
spusa# ./lib/fdelay-board-time  get
fdelay_init: version mismatch, lib(1) != drv(2)
./lib/fdelay-board-time: fdelay_init(): Input/output error
```

This reports a difference in the way ZIO attributes are laid out, so user space may exchange wrong data in the ZIO control block, or may try to access inexistent files in */sys*. I suggest recompiling both the kernel driver and user space from a single release of the source package.