Fine Delay User's Manual

April 2014 (fine-delay-sw-v2014-04)
FMC Delay 1ns-4cha hardware and software manual

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

Table of Contents

Revision history

Revision	Date	Author	Changes
1.0	April 2012	AR, TW	Initial version
2.0	August 2013	AR, TW	Updated to 2.0 software/gateware release

Introduction

This is the user manual for the "fmc-delay-1ns-4cha" board developed on `ohwr.org`. Please note that the ohwr hardware project is misnamed as `fmc-delay-1ns-8cha`; even if the board has 4 channels; the references to `8ch` below are thus correct, even if the may seem wrong.

1 Repositories and Releases

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 some official release, but we prefer to use the *files* tab, below.

`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), as well as manuals.

`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.

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 release this manual claims to document. It is a fact of life that developers forget to re-read and fix documentation while updating the code. In that case, please 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 FINE-DELAY.

2.1 Requirements and Supported Platforms

FINE-DELAY can work with any VITA 57-compliant FMC carrier, provided that the carrier's FPGA has enough logic resources. The current software/gateware release officially supports the following carrier and mezzanine combinations:

- CERN's SPEC (Simple PCI-Express Carrier) with one FINE-DELAY mezzanine.
- CERN's SVEC (Simple VME64x Carrier) with one or two FINE-DELAY mezzanines. Note that if only one FINE-DELAY is in use, the other slot should be left empty.

Aside from the FMC and its carrier, the following hardware/software components are required:

- For the PCI version: a standard PC with at least one free 4x (or wider) PCI-Express slot.
- For the VME version: a VME64x crate with a MEN A20 CPU (fixme: can the driver work on RIO or something else?).
- 50-ohm cables with 1-pin LEMO 00 plugs for connecting the I/O signals.
- Any Linux (kernel 2.6 or 3.0+) distribution. Backports are provided down to kernel `2.6.24`.

2.2 Modes of Operation

FINE-DELAY 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.

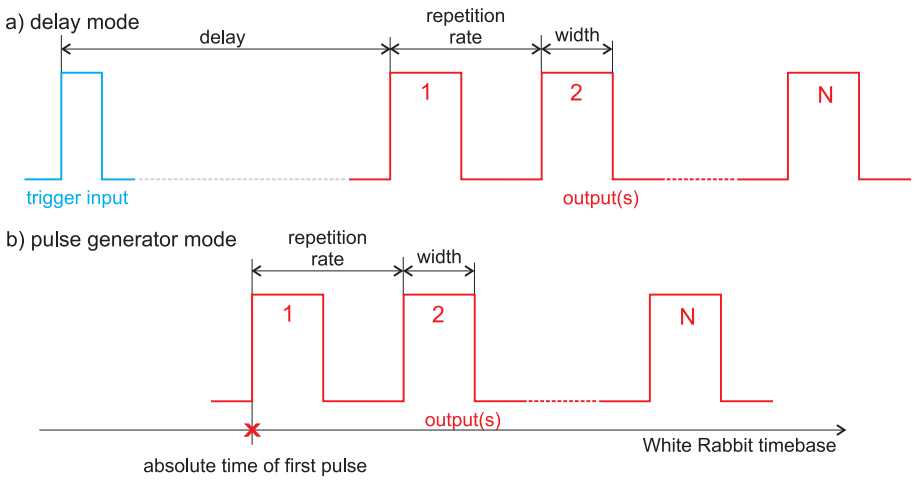

Fig. 1. FINE-DELAY operating modes.

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

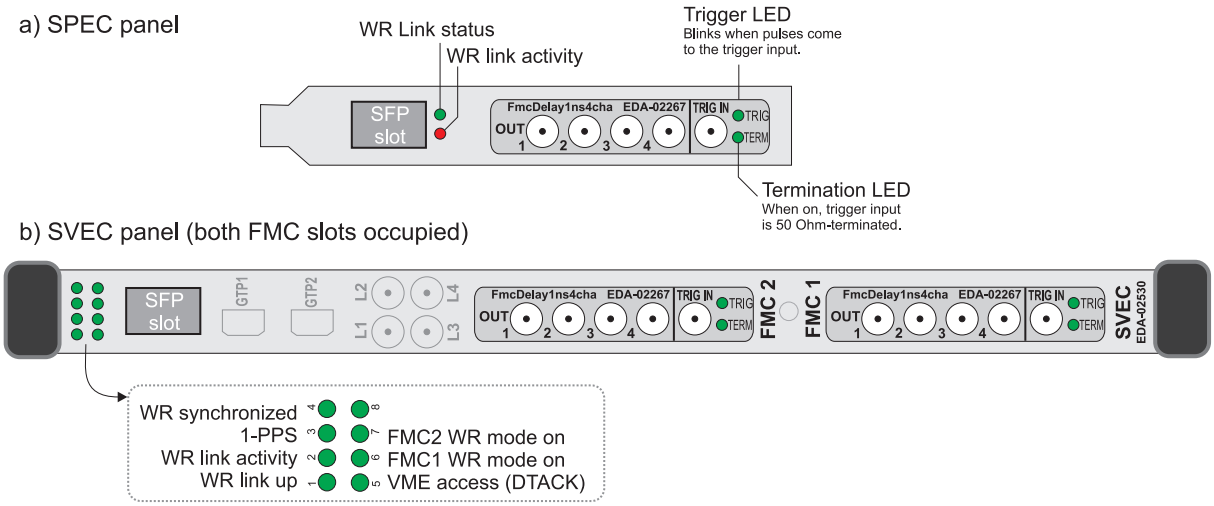

Fig. 2. FINE-DELAY front panel connector layout.

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 (termination status and trigger indicator).
- 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 consumption: 200 mA (P12V0) + 1.5 A (P3V3).
- Power dissipation: 7 W. Forced cooling is required.

2.5 Timing

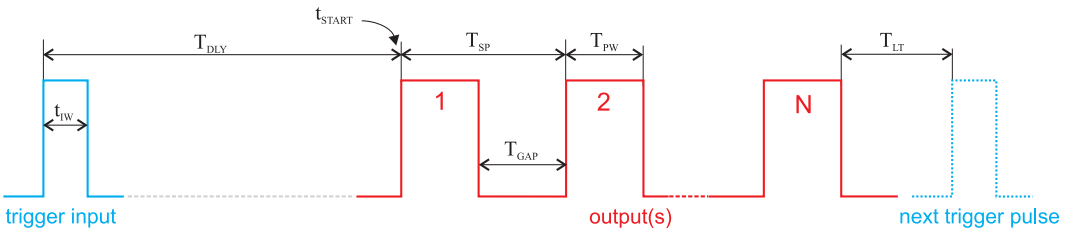

Fig. 2. FINE-DELAY timing parameter definitions.

Time base:

- On-board 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} = 600 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.

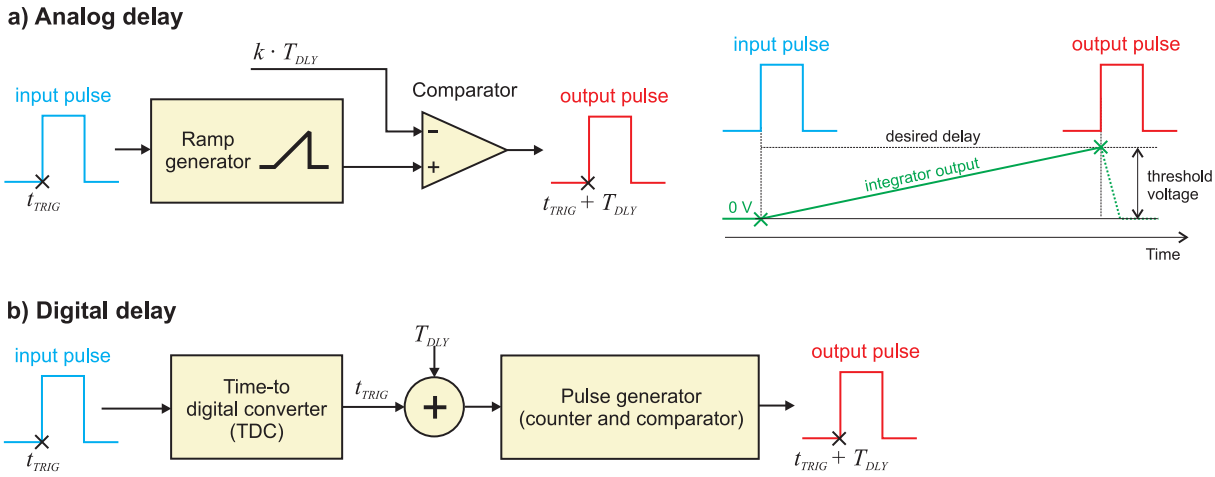

Fig. 3. Principle of operation of analog and digital delay generators.

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, FINE-DELAY 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, FINE-DELAY 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, FINE-DELAY can be used simultaneously as a TDC, pulse generator or a pulse delay.

3 Driver Features

This driver is based on ZIO and *fmc-bus*. It supports initial setup of the board, setting and reading time, run-time continuous calibration, input timestamping, output pulse generation and readback of output settings from the hardware. 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) the driver tries to offer the following items; sometimes however one of them is missing for a specific driver functionality, if we don't consider it important enough.

- 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.

4 Installation

This driver depends on four other modules (four `ohwr.org` packages), as well as the Linux kernel. Also, it must talk to a specific FPGA binary file running in the device.

4.1 Gateware Dependencies

While previous versions of this package included a gateware binary in the `binaries/` subdirectory, in Jan 2014 we decided not to do that any more. Release versions of this package are expected to point to "current" gateware images for different carriers. Clearly the driver is expected to work on any FMC carrier, even those ignored to us, and we can't provide all binaries.

The up-to-date gateware binaries for the SVEC and SPEC carriers will be always available in the *Releases* section of the Fine Delay project: http://www.ohwr.org/projects/fmc-delay-1ns-8cha/wiki/Releases

Note that the release gateware contains a stable version of the White Rabbit PTP Core firmware. This firmware may be reloaded dynamically at any time using carrier-specific tools.

4.2 Gateware Installation

By default, the driver looks for a gateware file named `/lib/firmware/fmc/[carrier]-fine-delay.bin`, where `[carrier]` is the carrier's name (lowercase - currently `svec` or `spec`). There are two ways to install the gateware:

- The easy way: run `make gateware_install` in the target system. This will automatically download all required files and install them in the right place.
- The difficult way: download the bitstreams from the Release page (or build your own, as you wish) and put them in `/lib/firmware/fmc`. You may have to strip the version/date attached to the file names or create symlinks.

If you have several *fine-delay* cards in the same host, you can load different binaries for different cards, using appropriate module parameters. Loading custom gateware files is advised only for advanced users/developers.

4.3 Software Dependencies

The kernel versions I am using during development is 3.4. Everything used here is known to build with all versions from 2.6.35 to 3.12.

The driver, then, is based on the ZIO framework, available from `ohwr.org`. This version of ZIO doesn't build with 3.13, for a minor incompatibility, so this version of *fine-delay-sw* is limited to Linux 3.12 as well.

The FMC mezzanine is supported by means of the *fmc-bus* software project. This *fine-delay* kernel module registers as a *driver* for the FMC bus abstraction, and is verified with version `v2014-02` of the FMC package. The same kernel range applies.

Both packages (ZIO and *fmc-bus*) are currently checked out as *git submodules* of this package, and each of them is retrieved at the right version to be compatible with this driver. This means you may just ignore software dependencies and everything should work.

FMC support is a *bus* in the Linux way, so you need both a *device* and a *driver*. This driver is known to work both with the SPEC carrier on PCI and the SVEC carrier on VME. The software packages that provide the respective *device* are called *spec-sw* and *svec-sw*; both are hosted on `ohwr.org`.

Most of the non-CERN users are expected to run the SPEC carrier, so a compatible version of *spec-sw* is downloaded as a submodule, too.

4.4 Software Installation

To install this software package, you need to tell it where your kernel sources live, so the package can pick the right header files. You need to set only one environment variable:

`LINUX`

> The top-level directory of the Linux kernel you are compiling against. If not set, the default may work if you compile in the same host where you expect to run the driver.

Most likely, this is all you need to set. After this, you can run:

```
make
sudo make install gateware_install LINUX=$LINUX
```

In addition to the normal installation procedure for `fmc-fine-delay.ko` you'll see the following message:

```
WARNING: Consider "make prereq_install"
```

The *prerequisite* packages are *zio* and *fmc-bus*; unless you already installed your own preferred version, you are expected to install the version this packages suggests. This step can be performed by:

```
make
sudo make prereq_install LINUX=$LINUX
```

The step is not performed by default to avoid overwriting some other versions of the drivers. After `make prereq_install`, the warning message won't be repeated any more if you change this driver and `make install` again.

After installation, your carrier driver should load automatically (for example, the PCI bus will load `spec.ko`), but `fmc-fine-delay.ko` must be loaded manually, because support for automatic loading is not yet in place. The suggested command is one or the other of the following two:

```
modprobe fmc-fine-delay [<parameter> ...]         # after make install
insmod kernel/fmc-fine-delay.ko [<parameter> ...]  # if not installed
```

Available module parameters are described in Section 4.5 [Module Parameters], page 8. Unless you customized or want to customize one of the three related packages, you can skip the rest of this section.

In order to compile *fine-delay* against a specific repository of one of the related packages, ignoring the local *submodule* you can use one or more of the following environment variables:

`ZIO`
`FMC_BUS`
`SPEC_SW`

> The top-level directory of the repository checkout of each package. Most users won't need to set them, as the Makefiles point them to the proper place by default.

If any of the above is set, headers and dependencies for the respective package are taken from the chosen directory. If you `make prereq_install` with any of these variables set, they are be used to know where to install from, instead of using local submodules.

4.5 Module Parameters

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

The following parameters are used:

`verbose=`

The parameter defaults to 0. If set, it enables more diagnostic messages during probe (you may find it is not used, but it is left in to be useful during further development, and avoid compile-time changes like use of `DEBUG`).

`timer_ms=`

The period of the internal timer, if not zero. The timer is used to poll for input events instead of enabling the interrupt. The default interval is 0, which means to use interrupt support. You may want to use the timer while porting to a different carrier, before sorting out IRQ issues.

`calib_s=`

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

The module also uses the two parameters provided by the *fmc* framework:

`busid=`

A list of bus identifiers the driver will accept to driver. Other identifiers will lead to a failure in the *probe* function. The meaning of the identifiers is carrier-specific; the SPEC uses the bus number and *devfn*, where the latter is most likely zero.

`gateware=`

A list of gateware file names. The names passed are made to match the *busid* parameters, in the same order. This means that you can't make the driver load a different gateware file without passing the respective *busid*. Actually, to change the gateware for all boards, you may just replace the file in '`/lib/firmware`'. (Maybe I'll add an option to change the name at load time for all boards).

For example, this host has two SPEC cards:

```
spusa.root# lspci | grep CERN
02:00.0 Non-VGA unclassified device: CERN/ECP/EDU Device 018d (rev 03)
04:00.0 Non-VGA unclassified device: CERN/ECP/EDU Device 018d (rev 03)
```

One of the cards hosts a *fine-delay* mezzanine and the other does not. FMC identifiers are not yet used by this driver at this point in time. (They will be there in the next release: code is there but not finalized). So, here you can use `busid=` to choose which SPEC must use *fine-delay*, leaving the other one alone:

```
spusa.root# insmod fmc-fine-delay.ko busid=0x0200
[ 4603.994936] spec 0000:02:00.0: Driver has no ID: matches all
[ 4604.000624] spec 0000:02:00.0: reprogramming with fmc/fine-delay.bin
[ 4604.206515] spec 0000:02:00.0: FPGA programming successful
[ 4604.212442] spec 0000:02:00.0: Gateware successfully loaded
[ 4604.218037] spec 0000:02:00.0: fd_regs_base is 80000
[ 4604.223023] spec 0000:02:00.0: fmc_fine_delay: initializing
[ 4604.228624] spec 0000:02:00.0: calibration: version 3, date 20130427
[ 4605.691404] fd_read_temp: Scratchpad: 9f:04:4b:46:7f:ff:01:10:89
[ 4605.697615] fd_read_temp: Temperature 0x49f (12 bits: 73.937)
[ 4606.645545] fd_calibrate_outputs: ch1: 8ns @859 (f 827, off 32, t 71.00)
[ 4606.815228] fd_calibrate_outputs: ch2: 8ns @867 (f 827, off 40, t 71.00)
[ 4607.001027] fd_calibrate_outputs: ch3: 8ns @854 (f 827, off 27, t 71.00)
[ 4607.187007] fd_calibrate_outputs: ch4: 8ns @859 (f 827, off 32, t 71.00)
```

```
[ 4607.356103] fmc_fine_delay: Found i2c device at 0x50
[ 4607.364039] spec 0000:02:00.0: Using interrupts for input
[ 4607.369549] spec 0000:04:00.0: Driver has no ID: matches all
[ 4607.375243] spec 0000:04:00.0: not using "fmc_fine_delay" according to modparam
```

If you use `show_sdb=1`, you'll get the following dump of the internal SDB structure to *printk*. The *Self Describing Bus* data structure is described in the documentation of the *fpga-config-space* project, under `ohwr.org`.

```
SDB: 00000651:e6a542c9 WB4-Crossbar-GSI
SDB: 0000ce42:f19ede1a Fine-Delay-Core     (00010000-000107ff)
SDB: 0000ce42:f19ede1a Fine-Delay-Core     (00020000-000207ff)
SDB: 0000ce42:00000013 WB-VIC-Int.Control  (00030000-000300ff)
SDB: 00000651:eef0b198 WB4-Bridge-GSI      (bridge: 00040000)
SDB:    00000651:e6a542c9 WB4-Crossbar-GSI
SDB:    0000ce42:66cfeb52 WB4-BlockRAM        (00040000-00055fff)
SDB:    00000651:eef0b198 WB4-Bridge-GSI      (bridge: 00060000)
SDB:       00000651:e6a542c9 WB4-Crossbar-GSI
SDB:       0000ce42:ab28633a WR-Mini-NIC         (00060000-000600ff)
SDB:       0000ce42:650c2d4f WR-Endpoint         (00060100-000601ff)
SDB:       0000ce42:65158dc0 WR-Soft-PLL         (00060200-000602ff)
SDB:       0000ce42:de0d8ced WR-PPS-Generator    (00060300-000603ff)
SDB:       0000ce42:ff07fc47 WR-Periph-Syscon    (00060400-000604ff)
SDB:       0000ce42:e2d13d04 WR-Periph-UART      (00060500-000605ff)
SDB:       0000ce42:779c5443 WR-Periph-1Wire     (00060600-000606ff)
SDB:       0000ce42:779c5445 WR-Periph-AuxWB     (00060700-000607ff)
SDB: Bitstream 'svec-fine-delay' synthesized 20140317 by twlostow \
               (ISE version 133), commit e95b10c776f5f7603f49fcf1330e0c07
SDB: Synthesis repository: git://ohwr.org/fmc-projects/fmc-delay-1ns-8cha.git
```

The module also supports some more parameters that are calibration-specific. They are described in the Section B.6 [Calibration], page 26 section.

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.
- The *tools* directory hosts the suggested command-line tools to use the device for testing and quick access. They demonstrate use of the library functions using internally-consistent command line conventions. All command names begin with `fmc-fdelay-`
- The *oldtools* directory includes tools that access ZIO directly; they are not built by default any more because they are now deprecated; we also removed documentation for them, for the same reason. We keep them for our previous users, in case they still want to run previous scripts the saved in the past. The directory also includes tools that used to be built withing `lib/` and are deprecated as well. The old tools use the name patterns `fd-raw-` and `fdelay-`
- The *lib* directory contains the userspace library, providing a simple C API to access the driver.
- The *tools* directory contains a number of tools built on top of that library that let you access all features of the FINE-DELAY mezzanine.

6 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).

6.1 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.

Example code: all tools in `tools/` subdirectory.

6.2 Time Management

These are the primitives the library offers for time management, including support for White Rabbit network synchronization.

`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. This function is only provided to

coarsely correlate the board time with the system time. Relying on system time for synchronizing multiple FINE-DELAYS is strongly discouraged.

`int fdelay_wr_mode(struct fdelay_board *b, int on);`
The function enables/disables White Rabbit mode. It may fail with `ENOTSUPP` if there's no White Rabbit support in the gateware.

`int fdelay_check_wr_mode(struct fdelay_board *b);`
The function returns 0 if the WR slave is synchronized, `EAGAIN` if it is enabled by not yet synchronized, `ENODEV` if WR-mode is currently disabled and `ENOLINK` if the WR link is down (e.g. unconnected cable).

Example code: `fmc-fdelay-board-time` tool.

6.3 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 bit-mask 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 timestamps, and the last enables the 50-ohm termination on the input channel.

Example code: `fmc-fdelay-term` tool.

6.4 Reading Input Timestamps

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.

The test programs for the functions are in *oldtools/*, not built by default.

6.5 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 for pulse or delay mode. The channel numbers are 0..3 (that is, the number of the output on the front panel minus 1, you may use `FDELAY_OUTPUT` macro to convert). 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_get_config_pulse(board, channel, pulse_cfg);`
`int fdelay_get_config_pulse_ps(board, channel, pulse_ps_cfg);`
: The two functions return the configuration of the channel (numbered 0..3) read from the hardware. They may be used to check the correctness of outputs' programming. The former function returns `struct fdelay_pulse` (with split utc/coarse/frac times) while the latter returns `struct fdelay_pulse_ps`, with picosecond-based time values.

`int fdelay_has_triggered(struct fdelay_board *b, int channel);`
: The function 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`.

Example code: `fmc-fdelay-pulse` tool.

6.6 Miscellanous functions

`void fdelay_pico_to_time(uint64_t *pico, struct fdelay_time *time);`
: Splits a time value expressed in picoseconds to FINE-DELAY's internal time format (utc/coarse/frac).

`void fdelay_time_to_pico(struct fdelay_time *time, uint64_t *pico);`
: Converts from FINE-DELAY's internal time format (utc/coarse/frac) to plain picoseconds.

`float fdelay_read_temperature(struct fdelay_board *b);`
: Returns the temperature of the given board, in degrees Celsius.

7 Command Line Tools

This chapter describes the command line tools that come with the driver and reside in the `tools/` subdirectory. They are provided as diagnostic utilities and to demonstrate how to use the library.

7.1 General Command Line Conventions

Most tools accept the following command-line options, in a consistent way:

`-d <devid>`
`-i <index>`
: Used to select one board among several. See the description of *fdelay_open* in Section 6.1 [Initialization and Cleanup], page 10. If no argument is given, the "first" board is used (index is 0).

7.2 fmc-fdelay-list

The command takes no arguments. It reports the list of available boards in the current system:

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

7.3 fmc-fdelay-term

The command can be used to activate or deactivate the 50 ohm termination resistor.

In addition to the `-i` or `-d` arguments, mandatory if more than one board is found on the host system, the command receives one optional argument, either `1` or `on` (activate termination) or `0` or `off` (deactivate termination).

```
spusa# ./tools/fmc-fdelay-term on
./tools/fmc-fdelay-term: termination is on
```

If no arguments are passed the termination status is reported back but not changed.

7.4 fmc-fdelay-board-time

The command is used to act on the time notion of the FINE-DELAY card.

In addition to the `-i` or `-d` arguments, mandatory if more than one board is found on the host system, the command receives one mandatory argument, that is either a command or a floating point number. The number is the time, in seconds, to be set in the card; the command is one of the following ones:

`get`
: Read board time and print to *stdout*.

`host`
: Set board time from host time

`wr`

Lock the boards to White Rabbit time. It may block if no White Rabbit is there. No timeout is currently available.

`local`

Detach the board from White Rabbit, and run local time instead.

Examples:

```
spusa# ./lib/fdelay-board-time  25.5; ./lib/fdelay-board-time get
25.504007360
spusa# ./lib/fdelay-board-time  get
34.111048968
spusa# ./lib/fdelay-board-time  host
spusa# ./lib/fdelay-board-time  get
1335974946.493415600
```

7.5 fmc-fdelay-input

The tool reports input pulses to stdout. It receives the usual `-i` or `-d` arguments to select one board, mandatory if more than one FINE-DELAY card is found.

It receives the following options:

`-c <count>`

Number of pulses to print. Default (0) means run forever.

`-n`

Nonblocking mode: just print what is pending in the buffer.

`-f`

Floating point: print as a floatingpoint seconds.pico value. The default is a human-readable string, where the decimal part is split.

`-r`

Raw output: print the three hardware timestamps, in decimal.

This an example output, reading a pps signal through a 16ns cable:

```
spusa.root# ./tools/fmc-fdelay-input -c 3
seq 10921:     time      11984:000,000,015,328 ps
seq 10922:     time      11985:000,000,015,410 ps
seq 10923:     time      11986:000,000,015,248 ps
spusa.root# ./tools/fmc-fdelay-input -c 3 -r
seq 10924:      raw   utc      11987,  coarse         1,  frac      3773
seq 10925:      raw   utc      11988,  coarse         1,  frac      3814
seq 10926:      raw   utc      11989,  coarse         1,  frac      3794
spusa.root# ./tools/fmc-fdelay-input -c 3 -f
seq 10927:     time      11990.000000015328
seq 10928:     time      11991.000000015410
seq 10929:     time      11992.000000015410
```

In a future release we'll support reading concurrently from several boards.

7.6 fmc-fdelay-pulse

The program can be used to program one of the output channels to output a sequence of pulses. It can parse the following command-line options:

`-o <output>`

Output channels are numbered 1 to 4, as written on the device panel. Each command invocation can set only one output channel; the last `-o` specified takes precedence.

`-c <count>`
: Output repeat count: 0 is the default and means forever

`-m <mode>`
: Output mode. Can be `pulse`, `delay` or `disable`.

`-r <reltime>`
: Output pulse at a relative time in the future. The time is a fraction of a second, specified as for `-T` and `-w`, described below. For delay mode the time is used as a delay value from input events; for pulse mode the time represents a fraction of the next absolute second.

`-D <date>`
: Output pulse at a specified date. The argument is parsed as `<seconds>:<nanoseconds>`.

`-T <period>`
`-w <width>`
: Period and width of the output signal. A trailing `m`, `u`, `n`, `p` means milli, micro, nano, pico, resp. The parser supports additions and subtractions, e.g. `50m-20n`. The period defaults to 100ms and the width defaults to 8us

`-t`
: Wait for the trigger to happen before returning. The boards reports a trigger event when the requested pulse sequence is initiated, either because the absolute time arrived or because an input pulse was detected and the requested delay elapsed.

`-p`
`-1`
: Pulse-per-seconds and 10MHz. These are shorthands setting many parameters.

`-v`
: Verbose: report action to stdout before telling the driver.

This is, for example, how verbose operation reports the request for a single pulse 300ns wide, 2 microseconds into the next second.:

```
spusa.root# ./tools/fmc-fdelay-board-time get; \
            ./tools/fmc-fdelay-pulse -i 0 -o 1 -m pulse -r 2u -w 300n -c 1 -t -v
WR Status: disabled.
Time: 13728.801090400
Channel 1, mode pulse, repeat 1
  start  time      13729:000,002,000,000 ps
  end    time      13729:000,002,300,000 ps
  loop   time          0:100,000,000,000 ps
Channel 1, mode pulse, repeat 1
  start   raw   utc      13729,  coarse       250,  frac         0
  end     raw   utc      13729,  coarse       287,  frac      2048
  loop    raw   utc          0,  coarse  12500000,  frac         0
```

7.7 fmc-fdelay-status

The program reports the current output status of the four channels, both in human-readable and raw format. The receives no arguments besides the usual `-i` or `-d`.

Please note that the tool reads back hardware values, which are already fixed for calibration delays. For example, this is the output I get after the previously-shown activation command, that was for 13729 + 2 microseconds:

```
spusa.root# ./tools/fmc-fdelay-status
Channel 1, mode already-triggered, repeat 1
  start  time      13729:000,001,961,814 ps
  end    time      13729:000,002,261,814 ps
  loop   time          0:100,000,000,000 ps
Channel 1, mode already-triggered, repeat 1
  start   raw   utc      13729,  coarse       245,  frac       929
  end     raw   utc      13729,  coarse       282,  frac      2977
  loop    raw   utc          0,  coarse  12500000,  frac         0
[...]
```

The difference in value depends on the `delay-offset` value for the channel, according to calibration.

Appendix A Troubleshooting

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

A.1 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
```

A.2 Version Mismatch

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

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

Appendix B Additional Information for Developers

This appendix contains some extra information on the internals of the driver and how to access it bypassing the library. If you are not a FINE-DELAY, FMC-BUS or ZIO developer, you can skip this chapter and just use the library.

B.1 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).

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

The actual pathnames depend on the version of *udev*, and the support library tries the three names that have been used over time (the newest name is shown above; the oldest didn't have the two `zio`). 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/zio-fd-0200*:

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

B.3 Device Attributes

Device-wide attributes are the three time tags (*utc-h*, *utc-l*, *coarse*), a read-only *version*, a read-only *temperature* 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/zio-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
```

The temperature value is scaled by four bits, so you need divide it by 16 to obtain the value in degrees. In this example:

```
spusa# cat temperature
1129
```

Temperature is 70.5625 degrees.

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.

B.3.1 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 not supported or 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 (fixme: is it really needed).

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

B.3.2 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.

B.3.3 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 6.2 [Time Management], page 10).

B.4 The Input cset

The input cset returns blocks with no data and timestamp information in the control structure (the meta-information associated to data). Before January 2014 the driver was suboptimal, but now those limitations are gone and the driver uses the "self-timed" ZIO abstraction, which allows it to push blocks to the buffer even if no process is yet reading.

Collecting event in empty blocks, with full meta-data description, brings some overhead in the data flow, mainly for the marshalling of meta-data. If you need to stamp pulse rates higher than 10kHz we advise you to rely on the *raw_tdc* support, which on an average computer can timestamp up to 100-150 kHz without data loss. This is described in Section B.4.7 [Raw TDC], page 22. The internals of the input data flow are described in Section B.4.8 [The Input Data Flow], page 22, that may help fine-tune driver parameters to match your timestamping needs.

For normal ZIO blocks, with meta-data and no data, 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.

B.4.1 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/zio-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 hardware timestamper's time-base is shifted backwards, so the driver adds this offset to the raw timestamps it collects. Users should not change this value, that depends on how hardware and HDL is designed.

The *user-offset* attribute, which defaults to 0 every time the driver is loaded, is a signed value that users can write to represent a number of picoseconds to be added (or subtracted, if negative) 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.

B.4.2 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/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
```

B.4.3 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 timestamps 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).

If you are using the tool in a script, and you want to capture all the samples in a burst and then terminate, you can specify a timeout, in microseconds, using `-t`. The timeout is only applied after the first pulse is received.

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.

B.4.4 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
```

B.4.5 Configuring the Input Channel

There is no support in *tools/* to change channel configuration (but see Section 6.3 [Input Configuration], page 11 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*.

B.4.6 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
```

B.4.7 Raw TDC

If your rate of input pulses is above a dozen kHz, the overheader of setting up a full ZIO block with proper control information may cause some data loss; the actual threshold depends on the speed of your computer and the amount of other activities that are going on.

By loading the module with the parameter "`raw_tdc=1`", you force the input channel to carry timestamps in the data area; only the first timestamp is properly converted to meta-data for the control structure. This allow timestamping without data loss trains of pulses of up to 150kHz; again, the actual limit depends on the performance of your host computer and concurrent load.

Timestamps are returned as 24-byte-long data samples, i.e. `struct fd_time`, as defined in the header file:

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

For a simple pulse logging, the following shell command will work:

```
insmod kernel/fmc-fine-delay.ko raw_tdc=1 fifo_len=16384
cat /dev/zio/fd-0200-0-0-data > logfile
```

Anders Walling provided tools for use with `raw_tdc=1`. I'll try to merge them with this package; meanwhile please find them in https://github.com/aewallin/fine-delay-sw.

B.4.8 The Input Data Flow

This section described the input data flow, after a summary about the basic ZIO concept, because most readers are not expected to be confident with it.

Fdelay-sw implements a ZIO device. ZIO is a framework to transport I/O data, its own atomic unit is a "block", i.e. meta-information (*control*, or `ctrl`) and actual samples (*data*). Each block is like a network frame, in a way: header and payload. The header/ctrl is 512 bytes and includes a very sharp timestamp plus both standardized and device-specific "attribute" values.

TDC/DTC devices are best represented as an empty block: the header carries the timestamp and the attributes, and no data is associated with the event. This however has an overhead: each timestamp is 512 bytes big, and is delivered as a separate object. With `fd-raw-input` I can collect 30-40kHz square waves, but not more than that. This means my computer takes 25-30 microseconds per sample, including the user-space overhead. This time is mainly taken by the data conversion and attribute setting to provide high-level information; the overhead of a ZIO block is less than one microsecond, as documented elsewhere.

By using the new module parameter `raw_tdc=1` the data flow is slightly modified and timestamps are delivered to user space in a much lower-level format. The sample-size of the input channel is now 24 bytes (`struct fd_time`, defined in the header) and each block can transport several samples in its data area. Thus, if configured for N samples per block, ZIO allocates payload areas of `24*N` bytes; when the input interrupt is served, the driver fills as many samples as it can, up to N, it then stores the block to the ZIO buffer. Thus, each block in the buffer will host 1 or more "raw" timestamps, up to the configured value N. This lowers the computational load and allows capturing fast bursts of many thousands pulses.

The data path is then split in the following steps

- In the gateware, timestamps are placed in a ring buffer (FIFO) that is currently 1024-samples long (set by `c_RING_BUFFER_SIZE_LOG2` in `fine_delay_pkg.vhd`).
- The irq handler pulls the hardware fifo and places samples into a software ring buffer (fifo). The software fifo is an array of "struct fd_time". Its size is configured by the insmod parameter `fifo_len=` (default is 1024 as I write this). The handler finally sends acknowledgement to the hardware and awakes the software interrupt.
- The software interrupt handler pulls the software fifo and fills the already-allocated ZIO block, finally storing it to the buffer. Both the block size and the number of blocks in the buffer are configurable at run time. When ZIO allocates the next block, the driver pulls the software fifo too, so any sample received in the store-allocate interval is recovered in the new block. When using `raw_tdc=1`, the ZIO control represents the first timestamp (so consistency of the meta-information is preserved), and all stamps including the first are included in the data area after a simple normalization step. So the samples are not *very* raw, some calculation is still performed, but much less than setting all the ZIO attributes.

Thus, the critical points are the following ones:

- Hardware can timestamp up to its maximum speed (I tested 1MHz with no issues) as long as the burst fits in the hw fifo.
- The irq handler moves the samples to the software fifo, while splitting bit fields. Several samples are handled by each interrupt. I think I can pull up to 300-500 kilosamples per second. But I didn't prepare a specific test. This works with no loss as long as the software fifo is not overflown. Clearly the sw fifo can be increased at will: making it 64-ksample or more is not a problem, but the size is constrained to be a power of two.
- Moving the samples from the software fifo to the ZIO buffer is another step, which requires a little more data conversion (normalization and addition of the user-defined constant offset). There is a per-sample overhead and a (bigger) per-block overhead. This step detects if an overflow of the software fifo happened. IF so, it discards half of the fifo size to recover some margin.

The number of samples per ZIO block is configured by the "post-samples" attribute (or pre-samples, which is usually left as 0 because stamps are taken after the trigger event):

```
echo 1000 > /sys/bus/zio/devices/fd-0200/fd-input/trigger/post-samples
```

A bigger size for the block means more wasted memory if pulses are slow (the block is used almost-empty); a smaller size means more overhead and thus a smaller maximum bursts frequency.

The buffer length (number of blocks), can be increased at will:

```
echo 1000 > /sys/bus/zio/devices/fd-0200/fd-input/chan0/buffer/max-buffer-len
```

There is nothing against using a very long list of blocks in the buffer, if user-space is slow in pulling data: blocks are only allocated when needed. Federico recently added an attribute to monitor buffer usage: `allocated-buffer-len` (which is always at least 1, because one block is always ready to be filled by the next interrupt).

Data can be read by user-space simply by reading

```
/dev/zio/fd-0200-0-0-data
```

The file is a continuous stream of samples. Meta-information is delivered to another device name: by reading data alone, the application ignores the control structures that are properly released.

Each sample includes a 16-bit sequence number, so the final consumer can detect overflows. This doesn't apply if the software fifo is 128k samples, because samples are dropped half-a-fifosize each time – maybe I can change this). If the ZIO buffer is overflown, ZIO must discard one or more blocks. This is reported in the *alarms* field of the control, also readable as `alarms` in sysfs. The sysfs attribute is write-1-to-clear and there's no other way to clear alarms.

In order to see how ZIO blocks flow, you can

```
./zio/tools/zio-dump /dev/zio/fd-0200-0-0-*
```

or just *grep* the number of samples in each block, without even reading the payload:

```
./zio/tools/zio-dump /dev/zio/fd-0200-0-0-* | grep ", n "
```

You'll get something like this:

```
Ctrl: seq 2257, n 26, size 24, bits 32, flags 01000001 (little-endian)
Ctrl: seq 2258, n 436, size 24, bits 32, flags 01000001 (little-endian)
Ctrl: seq 2259, n 2684, size 24, bits 32, flags 01000001 (little-endian)
Ctrl: seq 2260, n 4000, size 24, bits 32, flags 01000001 (little-endian)
[...]
Ctrl: seq 2268, n 4000, size 24, bits 32, flags 01000001 (little-endian)
Ctrl: seq 2269, n 854, size 24, bits 32, flags 01000001 (little-endian)
```

The log above is 40000 samples streamed at 200kHz into 4000-big ZIO blocks. In the log above, `n` is the number of samples in each block, `seq` is the ZIO sequence number for the block. The number of bits (32) is wrong, I apologize.

B.5 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 valid 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*.

The *user-offset* attribute, which defaults to 0 at module load time, is a signed value that users can write to represent a number of picoseconds to be added (or subtracted) to 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*.

B.5.1 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 6.5 [Output Configuration], page 12 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, count), (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
```

B.6 Calibration

Calibration data for a fine-delay card is stored in the I2C FMC EEPROM device, using the SDB filesystem. Previous versions used a constant offset of 6kB, but the calibration format was different, so no compatibility is retained. The driver will refuse to work with cards that have incompatible EEPROMs, these must be re-calibrated.

The driver automatically loads calibration data from the flash at initialization time, but only uses it if its hash 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 during initialization. It is mainly a debug tool.

`calibration_default`
: This option should only be used by developers. If not zero, it tells the driver to ignore calibration data found on the EEPROM, thus enacting a build-time default (which is most likely wrong for any board).

`calibration_load`
: This parameter is a file name, and it should only be used by developers. The name is used to ask the *firmware loader* to retrieve a file from '`/lib/firmware`'. The data, once read, is used only if the size is correct. The hash is regenerated by the driver. Please remember that all values in the calibration structure are stored as big-endian.

`calibration_save`
: This option should only be used by developers, and is not supported in this release. If you are a developer and need to change the calibration, please check the current master branch on the repository, or a later release. 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 currently offers no tool to generate the binary file for the calibration.