FMC TDC 1ns 5 Channel Documentation

Release v8.0.0.rc2-16-g4e02f30

Federico Vaga <federico.vaga@cern.ch>

Jan 24, 2022

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

This document describes the gateware developed to support the FmcTDC 1n 5channel (later refered to as fmc-tdc) mezzanine card on the SPEC and SVEC carrier cards. The gateware is the HDL code used to generate the bitstream that configures the FPGA on the carrier (sometimes also called firmware). The gateware architecture is described in detail. The configuration and operation of the fmc-tdc is also explained. The Linux driver and basic tools are explained as well. On the other hand, this manual is not intended to provide information about the hardware design.

1.1 Repositories and Releases

The FMC TDC 1ns 5 Channels is hosted on the Open HardWare Repository. The main development happens here. You can clone the GIT project with the following command:

```
git clone https://ohwr.org/project/fmc-tdc.git
```

Within the GIT respository, releases are marked with a TAG named using the Semantic Versioning. For example the latest release is `v8.0.0`. You can also find older releases with a different versioning mechanism.

For each release we will publish the FPGA bitstream for all supported carrier cards (FPGA Bitstream Page). For the Linux driver we can't release the binary because it depends on the Linux version on which it will run. For details about how to build the Linux driver for your kernel please have a look at *Compile And Install* section in *Driver's Documentation*.

1.2 Documentation License

CHAPTER

TWO

THE GATEWARE

2.1 About Source Code

2.1.1 Build from Sources

The fmc-tdc hdl design make use of the `hdlmake` tool. It automatically fetches the required hdl cores and libraries. It also generates Makefiles for synthesis/par and simulation.

Here is the procedure to build the FPGA binary image from the hdl source.:

```
# Install ``hdlmake`` (version 3.4).
# Get fmc-adc hdl sources.
git clone git://ohwr.org/project/fmc-adc-100m14b4cha.git <src_dir>

# Goto the synthesis directory.
cd <src_dir>/hdl/<carrier>/syn/

# Fetch the dependencies and generate a synthesis Makefile.
hdlmake

# Perform synthesis, place, route and generate FPGA bitstream.
make
```

2.1.2 Source Code Organisation

hdl/rtl/ ADC specific hdl sources.

hdl/cheby/ ADC specific `cheby` sources, html documentation and C header file.

hdl/ip_cores/ Location of fetched hdl cores and libraries.

hdl/platform/<platform> Platform related hdl sources.

hdl/top/<design> Top-level hdl module for selected design.

hdl/syn/<design> Synthesis directory for selected design. This is where the synthesis top manifest, the design constraints and the ISE project are stored. For each release, the synthesis, place&route and timing reports are also saved here.

hdl/testbench/ Simulation files and testbenches.

2.1.3 Dependencies

The fmc-adc gateware depends on the following hdl cores and libraries: General Cores, DDR3 SP6 core, GN4124 core (SPEC only), SPEC (SPEC only) VME64x Slave (SVEC only), SVEC (SVEC only), WR Cores.

These dependencies are managed with GIT submodules. Whenever you checkout a different branch remember to update the submodules as well.:

```
git submodule sync
git submodule update
```

CHAPTER

THREE

THE SOFTWARE

3.1 Driver

3.1.1 Driver Features

3.1.2 Requirements

The fmcadc100m14b4ch device driver has been developed and tested on Linux 3.10. Other Linux versions might work as well but it is not guaranteed.

This driver depends on the zio framework and fmc library; we developed and tested against version zio 1.4 and fmc 1.1.

The FPGA address space must be visible on the host system. This requires a driver for the FPGA carrier that exports the FPGA address space to the host. As of today we support SPEC and SVEC.

3.1.3 Compile And Install

The compile and install the fmctdc1ns5ch device driver you need first to export the path to its direct dependencies, and then you execute `make`. This driver depends on the zio framework and fmc library; on a VME system it depends also on the VME bridge driver from CERN BE-CEM.

```
$ cd /path/to/fmc-tdc/software/kernel
$ export LINUX=/path/to/linux/sources
$ export ZIO=/path/to/zio
$ export FMC=/path/to/fmc-sw
$ export VMEBUS=/path/to/vmebridge
$ make
$ make install
```

Note: Since version v8.0.0 the fmctdc1ns5ch device driver does not depend anymore on fmc-bus subsystem, instead it uses a new fmc library

The building process generates 3 Linux modules: *kernel/fmc-tdc.ko*, *kernel/fmc-tdc-spec.ko*, and *kernel/fmc-tdc-svec.ko*.

3.1.4 Top Level Driver

The fmctdc is a generic driver for an FPGA device that could be instanciated on a number of FMC carriers. For each carrier we write a little Linux module which acts as a top level driver (like the MFD drivers in the Linux kernel). In these modules there is the knowledge about the virtual memory range, the IRQ lines, and the DMA engine to be used.

The top level driver is a platform driver that matches a string containing the application identifier. The carrier driver builds this identification string from the device ID embedded into the FPGA (https://ohwr.org/project/fpga-dev-id).

3.1.5 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:

irq_timeout_ms=NUMBER It sets the IRQ coalesing timeout expressed in milli-seconds (ms). By default the value is set to 10ms.

test_data_period=NUMBER It sets how many fake timestamps to generate every seconds on the first channel, 0 to disable. By default the value is set to 0.

dma_buf_ddr_burst_size=NUMBER It sets DDR size coalesing timeout expressed in number of timestamps. By default the value is set to 16 timestamps.

wr_offset_fix=NUMBER It overwrites the White-Rabbit calibration offset for calibration value computed before 2018. By default this is set to 229460 ps.

3.1.6 Device Abstraction

This driver is based on the ZIO framework. It supports initial setup of the board; it allows users to manually configure the board, to start and stop acquisitions, to force trigger, and to read all the acquired time-stamps.

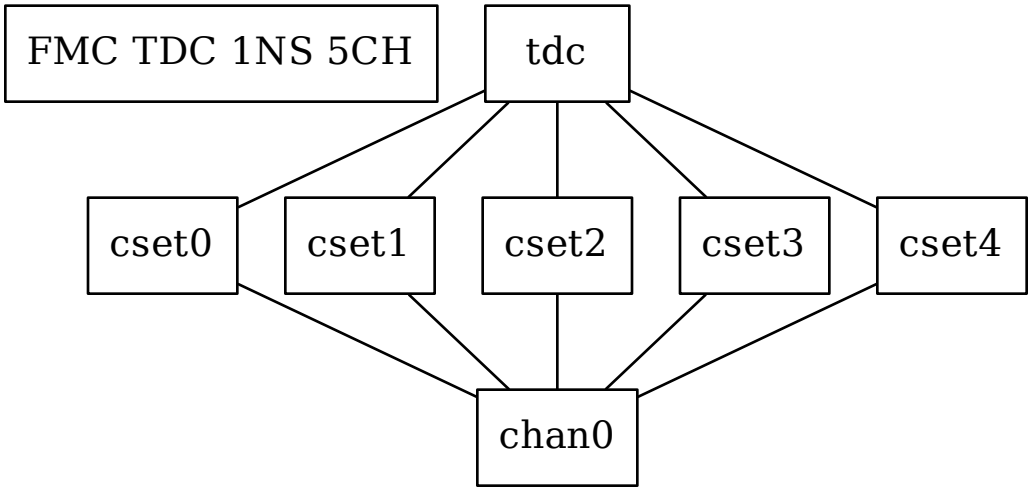
The driver is designed as a ZIO driver. ZIO is a framework for input/output hosted on http://www.ohwr.org/projects/zio.

ZIO devices are organized as csets (channel sets), and each of them includes channels. All channels belonging to the same cset trigger together. This device offers a channel-set for each channel.

Note: Unless specified, the units are the same as for the TDC HDL design. Therefore, this driver does not perform any data processing.

The Overall Device

As said, the device has 5 cset with 1 channel each. Channel sets from 0 to 4 represent the physical channels 1 to 5. In other words a channel set represents a single TDC channel.



The TDC registers can be accessed in the proper sysfs directory::

```
cd /sys/bus/zio/devices/tdc-1n5c-${ID}.
```

The overall device (*tdc-1n5c*) provides the following attributes:

calibration_data It is a binary attribute which allows the user to change the runt-time calibration data (the EEPROM will not be touched). The `fmc-tdc-calibration` tool can be used to read write calibration data. To be consistent, this binary interface expects **only** little endian values because this is the endianess used to store calibration data for this device.

coarse

command

seconds

temperature It shows the current temperature

transfer-mode

wr-offset

The Channel Set

The TDC has 5 Channel Sets named `cset[0-4]`. Its attributes are used to control and monitor each TDC channel individually. All channel specific attributes are available at the channel set level.

The Channels

Because there is a one-to-one relation with the channel set, we have decided to put all custom attributes at the channel set level. So, at this level you will find only default ZIO attributes.

The Trigger

TODO fix this section

In ZIO, the trigger is a separate software module, that can be replaced at run time. This driver includes its own ZIO trigger type, that is selected by default when the driver is initialized. You can change trigger type (for example use the timer ZIO trigger) but this is not the typical use case for this board.

This is the list of attributes (excluding kernel-generic and ZIO-generic ones):

enable This is a standard zio attribute, and the code uses it to enable or disable the hardware trigger (i.e. internal and external). By default the trigger is enabled.

post-samples, pre-samples Number of samples to acquire. The pre-samples are acquired before the actual trigger event (plus its optional delay). The post samples start from the trigger-sample itself. The total number of samples acquired corresponds to the sum of the two numbers. For multi-shot acquisition, each shot acquires that many sample, but pre + post must be at most 2048.

The Buffer

TODO fix this section

In ZIO, buffers are separate objects. The framework offers two buffer types: kmalloc and vmalloc. The former uses the kmalloc function to allocate each block, the latter uses vmalloc to allocate the whole data area. While the kmalloc buffer is linked with the core ZIO kernel module, vmalloc is a separate module. The driver currently prefers kmalloc, but even when it preferred vmalloc (up to mid June 2013), if the respective module wad not loaded, ZIO would instantiate kmalloc.

You can change the buffer type, while not acquiring, by writing its name to the proper attribute. For example:

```
echo vmalloc > /sys/bus/zio/devices/adc-100m14b-0200/cset0/current_buffer
```

The disadvantage of kmalloc is that each block is limited in size. usually 128kB (but current kernels allows up to 4MB blocks). The bigger the block the more likely allocation fails. If you make a multi-shot acquisition you need to ensure the buffer can fit enough blocks, and the buffer size is defined for each buffer instance, i.e. for each channel. In this case we acquire only from the interleaved channel, so before making a 1000-long multishot acquisition you can do:

```
export DEV=/sys/bus/zio/devices/adc-100m14b-0200
echo 1000 > $DEV/cset0/chani/buffer/max-buffer-len
```

The vmalloc buffer allows mmap support, so when using vmalloc you can save a copy of your data (actually, you save it automatically if you use the library calls to allocate and fill the user-space buffer). However, a vmalloc buffer allocates the whole data space at the beginning, which may be unsuitable if you have several cards and acquire from one of them at a time.

The vmalloc buffer type starts off with a size of 128kB, but you can change it (while not acquiring), by writing to the associated attribute of the interleaved channel. For example this sets it to 10MB:

```
export DEV=/sys/bus/zio/devices/adc-100m14b-0200
echo 10000 > $DEV/cset0/chani/buffer/max-buffer-kb
```

3.1.7 The debugfs Interface

The fmctdc1ns5cha driver exports a set of debugfs attributes which are supposed to be used only for debugging activities. For each device instance you will see a directory in `/sys/kernel/debug/fmc-tdc.*`.

regs It dumps the FPGA registers

3.1.8 Reading Data with Char Devices

To read data from user-space, applications should use the ZIO char device interface. ZIO creates 2 char devices for each channel (as documented in ZIO documentation). The TDC can acquire data on each channel independently, so ZIO creates ten char device, as shown below:

```
$ ls -l /dev/zio/tdc-*
  cr--r----- 1 root root 241, 0 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-0-0-ctrl
  cr--r----- 1 root root 241, 1 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-0-0-data
  cr--r----- 1 root root 241, 2 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-1-0-ctrl
  cr--r----- 1 root root 241, 3 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-1-0-data
  cr--r----- 1 root root 241, 4 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-2-0-ctrl
  cr--r----- 1 root root 241, 5 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-2-0-data
  cr--r----- 1 root root 241, 6 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-3-0-ctrl
  cr--r----- 1 root root 241, 7 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-3-0-data
  cr--r----- 1 root root 241, 8 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-4-0-ctrl
  cr--r----- 1 root root 241, 9 Jan 13 13:36 /dev/zio/tdc-1n5c-000b-4-0-data
```

If more than one board is probed for, you'll have more similar pairs of devices, differing in the dev_id field, i.e. the `000b` shown above. The dev_id field is assigned by the Linux kernel platform subsystem.

The char-device model of ZIO is documented in the ZIO manual; basically, the ctrl device returns metadata and the data device returns data. Items in there are strictly ordered, so you can read metadata and then the associated data, or read only data blocks and discard the associated metadata.

The `zio-dump` tool, part of the ZIO distribution, turns metadata and data into a meaningful grep-friendly text stream.

3.1.9 User Header Files

Both the kernel and the user make use of the same header file `fmc-tdc.h`. This because they need to share some data stracture and constants use to interpret data and meta-data in the library or by an application

Troubleshooting

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

Installation issue with modules_install

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

```
make: *** /lib/modules/<kernel-version>/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
```

3.2 Tools

The driver is distributed with a few tools living in the `tools/` subdirectory and they do not use any dedicated library, instead they do raw accesses to the driver. The programs are meant to provide examples about the use of the driver and library interface

3.2.1 Termination Configuration

3.2.2 Reading Temperature

3.2.3 Getting And Setting Board Time

3.2.4 Read Timestamps

3.2.5 User Offset Configuration

3.2.6 Calibration Data

3.3 The Library

Here you can find all the information about the *fmc-tdc* API and the main library behaviour that you need to be aware of to write applications.

This document introduces the developers to the development with the ADC library. Here you can find an overview about the API, the rational behind it and examples of its usage. It is not the purpose of the document to describe the API details. The complete API is available in the Library API document.

Note: The TDC hardware design diverged into different buffering structures. One based on FIFOs for SVEC, and one based on double-buffering in DDR for SPEC. The API tries to provide the same user-experience, however this is not always possible. Functions having different behaviour are properly declaring it in their documentation.

Note: This document provides also snippet of code from *example.c*. This is only to show you an example, please avoid to blindly copy and paste.

3.3.1 Initialization and Cleanup

The library may keep internal information, so the application should call its initialization function *fmctdc_init()*. After use, it should call the exit function *fmctdc_exit()* to release any internal data.

Note: *fmctdc_exit()* is not mandatory, the operating system releases anything in any case – the library doesn't leave unexpected files in persistent storage.

These functions don't do anything at this point, but they may be implemented in later releases. For example, the library may scan the system and cache the list of peripheral cards found, to make later *open* calls faster. For this reason it is **recommended** to, at least, initialize and release the library before starting.

Following an example from the `example.c` code available under `tools`

```
        err = fmctdc_init();
        if (err)
                exit(EXIT_FAILURE);

        err = use_fmctdc_library();
        if (err)
                exit(EXIT_FAILURE);
        fmctdc_exit(); /* optional, indeed in the error condition
                          we do not do it */
```

3.3.2 Error Reporting

Each library function returns values according to standard *libc* conventions: -1 or NULL (for functions returning `int` or pointers, resp.) is an error indication. When error happens, the *errno* variable is set appropriately.

The *errno* values can be standard Posix items like `EINVAL`, or library-specific values, for example `ADC_ERR_VMALLOC` (*driver vmalloc allocator not available*). All library-specific error values have a value greater than 4096, to prevent collision with standard values. To convert such values to a string please use *fmctdc_strerror()*

Following an example from the `example.c` code available under `tools`

```
                fprintf(stderr, "%s: Cannot open device: %s\n",
                        prog_name, fmctdc_strerror(errno));
```

3.3.3 Opening and closing

Each device must be opened before use by calling *fmctdc_open()*, and it should be closed after use by calling *fmctdc_close()*.

Note: *fmctdc_close()* is not mandatory, but it is recommended, to close if the process is going to terminate, as the library has no persistent storage to clean up – but there may be persistent buffer storage allocated, and *fmctdc_close()* may release it in future versions.

The data structure returned by *fmctdc_open()* is an opaque pointer used as token to access the API functions. The user is not supposed to use or modify this pointer.

Another kind of open function has been provided to satisfy CERN's developers needs. Function *fmctdc_open_by_lun()* is the open by LUN (*Logic Unit Number*); here the LUN concept reflects the *CERN* one. The usage is exactly the same as *fmctdc_open()* only that it uses the LUN instead of the device ID.

No automatic action is take by *fmctdc_open()*. Hence, you may want to flush the buffers before starting a new acquisition session. You can do this with *fmctdc_flush()*

```
        tdc = fmctdc_open(0x0000);
        if (!tdc) {
                fprintf(stderr, "%s: Cannot open device: %s\n",
                        prog_name, fmctdc_strerror(errno));
                return -1;
        }

        err = fmctdc_flush(tdc, channel);
        if (err)
                return err;

        err = config_and_acquire(tdc);
        if (err) {
                fprintf(stderr, "%s: Error: %s\n",
                        prog_name, fmctdc_strerror(errno));
                return -1;
        }

        fmctdc_close(tdc);
```

3.3.4 Configuration And Status

The TDC configuration API is based on a number of getter and setter function for each option. These include: *termination*, *IRQ coalescing timeout*, *board time*, *white-rabbit*, *timestamp mode*.

The *termination* options allows you to set the 50 Ohm channel termination. You can use the following getter and setter: *fmctdc_get_termination()*, *fmctdc_set_termination()*.

```
        err = fmctdc_set_termination(tdc, channel, termination);
        if (err)
                return err;
        termination_rb = fmctdc_get_termination(tdc, channel);
        if (termination_rb < 0)
                return termination_rb;
```

The *IRQ coalescing timeout* option allows to force an IRQ when the timeout expire to inform the driver that there is at least one pending timestamp to be transfered. You can use the following getter and setter: *fmctdc_coalescing_timeout_get()*, *fmctdc_coalescing_timeout_set()*.

```
        err = fmctdc_coalescing_timeout_set(tdc, channel, coalescing_timeout);
        if (err)
                return err;
        err = fmctdc_get_coalescing_timeout(tdc, channel, &coalescing_timeout_rb);
        if (err)
                return err;
```

The TDC main functionality is to timestap incoming pulses. To assign a timestamp the board needs a time reference.

This can be provided by the on-board clock, or by the more accurate white-rabbit network. You can enable or disable white-rabbit using *fmctdc_wr_mode()*. You can check the white-rabbit status with *fmctdc_check_wr_mode()*. When working with white-rabbit the time reference is handled by the white-rabbit network.

```
        err = fmctdc_wr_mode(tdc, wr_mode)
        if (err)
                return err;
        wr_mode_rb = fmctdc_check_wr_mode(tdc);
        if (wr_mode_rb < 0)
                return wr_mode_rb;
```

If you do not have white-rabbit connected to the TDC, or simply this is not what you want, then be sure to disable. When white-rabbit is disabled the TDC will use the on-board clock to keep a time reference. However, in this scenario the user is asked to set first the time using *fmctdc_set_time()* or *fmctdc_set_host_time()*.

```
        err = fmctdc_set_time(tdc, channel, time);
        if (err)
                return err;
```

Whater you are using white-rabbit or not, you can get the current board time with *fmctdc_get_time()*.

```
        err = fmctdc_get_time(tdc, channel, &time_rb);
        if (err)
                return err;
```

Still about time, the user can add it's own offset without changing the timebase using *fmctdc_get_offset_user()* and *fmctdc_set_offset_user()*.

```
        err = fmctdc_set_offset_user(tdc, channel, offset_user);
        if (err)
                return err;
        err = fmctdc_get_offset_user(tdc, channel, &offset_user_rb);
        if (err)
                return err;
```

Finally, you can monitor the board temperature using *fmctdc_read_temperature()*, and pulse and timestamps statistics with *fmctdc_stats_recv_get()* and *fmctdc_stats_trans_get()*.

```
        err = fmctdc_stats_recv_get(tdc, channel, &recv);
        if (err)
                return err;
        err = fmctdc_stats_trans_get(tdc, channel, &trans);
        if (err)
                return err;
```

Note: If it can be useful there is one last status function in the API used to detect the transfer mode between the driver and the board. This function is *fmctdc_transfer_mode()*

Timestamp buffering has its own set of options. Buffering in hardware is fixed, it can't be configured, so what we are going to describe here is the Linux device driver buffering configuration. Because the TDC driver is based on ZIO, then you can choose the buffer allocator type. You can handle this option with the pair: *fmctdc_get_buffer_type()* and *fmctdc_set_buffer_type()*.

```
        err = fmctdc_set_buffer_type(tdc, buffer_type);
        if (err)
                return err;
        buffer_type_rb = fmctdc_get_buffer_type(tdc);
        if (buffer_type_rb < 0)
                return buffer_type_rb;
```

You can configure - and get - the buffer size (number of timestamps) with: *fmctdc_get_buffer_len()* and *fmctdc_set_buffer_len()*. Beware, that this function works only when using FMCTDC_BUFFER_VMALLOC.

```
        err = fmctdc_set_buffer_len(tdc, channel, buffer_len);
        if (err)
                return err;
        buffer_len_rb = fmctdc_get_buffer_len(tdc, channel);
        if (buffer_len_rb < 0)
                return buffer_len_rb;
```

Finally, you can select between to modes to handle buffer's overflows: FMCTDC_BUFFER_CIRC and FMCTDC_BUFFER_FIFO. The first will discard old timestamps to make space for the new ones, the latter will discard any new timestamp until the buffer get consumed. To configure this option you can use: *fmctdc_get_buffer_mode()* and *fmctdc_set_buffer_mode()*.

```
        err = fmctdc_set_buffer_mode(tdc, channel, buffer_mode);
        if (err)
                return err;
        buffer_mode_rb = fmctdc_get_buffer_mode(tdc, channel);
        if (buffer_mode_rb < 0)
                return buffer_mode_rb;
```

3.3.5 Acquisiton

Before actually being able to get timestamps, the TDC acquisition must be enabled. The acquisition can be *enabled* or *disabled* through its gateware using, respectivily, *fmctdc_channel_enable()* and *fmctdc_channel_disable()*.

```
        err = fmctdc_channel_enable(tdc, channel);
        if (err)
                return err;

        err = fetch_and_process(tdc);
        if (err)
                return err;

        err = fmctdc_channel_disable(tdc, channel);
        if (err)
                return err;
```

You have to may functions to read timestamp *fmctdc_read()* and *fmctdc_fread()*. As the name may suggest, the first behaves like *read* and the second as *fread*.

```
        do {
                n = fmctdc_read(tdc, channel, ts, max, O_NONBLOCK);
        } while (n < 0 && errno == EAGAIN);
```

(continues on next page)

(continued from previous page)

```
        if (n < 0)
                return n;
```

If you need to flush the buffer, you can use *fmctdc_flush()*.

```
        err = fmctdc_flush(tdc, channel);
        if (err)
                return err;
```

3.3.6 Timestamp Math

The TDC library API has functions to support timestamp math. They allow you to *add*, *subtract*, *compare*, *normalize*, and *approximate*. These functions are: *fmctdc_ts_add()*, *fmctdc_ts_sub()*, *_fmctdc_tscmp()*, *fmctdc_ts_norm()*, *fmctdc_ts_ps()*,and *fmctdc_ts_approx_ns()*.

3.4 The Library API

Defines

PRItsps
printf format for timestamps with pico-second resolution

PRItspsVAL(_ts)
printf value for timestamps with pico-second resolution

PRItswr
printf format for timestamps with White-Rabbit notation

PRItswrVAL(_ts)
printf value for timestamp with White-Rabbit notation

__FMCTDC_ERR_MIN

Enums

enum **fmctdc_error_numbers**
Values:

enumerator **FMCTDC_ERR_VMALLOC**

enumerator **FMCTDC_ERR_UNKNOWN_BUFFER_TYPE**

enumerator **FMCTDC_ERR_NOT_CONSISTENT_BUFFER_TYPE**

enumerator **FMCTDC_ERR_VERSION_MISMATCH**

enumerator **__FMCTDC_ERR_MAX**

enum **fmctdc_channel**
Enumeration for all TDC channels

Values:

enumerator **FMCTDC_CH_1**

enumerator **FMCTDC_CH_2**

enumerator **FMCTDC_CH_3**

enumerator **FMCTDC_CH_4**

enumerator **FMCTDC_CH_5**

enumerator **FMCTDC_CH_LAST**

enumerator **FMCTDC_NUM_CHANNELS**

enum **fmctdc_buffer_mode**
Enumeration of all buffer modes

Values:

enumerator **FMCTDC_BUFFER_FIFO**
FIFO policy: when buffer is full, new time-stamps will be dropped

enumerator **FMCTDC_BUFFER_CIRC**
circular buffer policy: when the buffer is full, old time-stamps will be overwritten by new ones

enum **fmctdc_buffer_type**
Enumeration of all buffer types

Values:

enumerator **FMCTDC_BUFFER_KMALLOC**
kernel allocator: kmalloc

enumerator **FMCTDC_BUFFER_VMALLOC**
kernel allocator: vmalloc

enum **fmctdc_channel_status**
Enumeration for all possible status of a channel

Values:

enumerator **FMCTDC_STATUS_DISABLE**
The cannel is disable

enumerator **FMCTDC_STATUS_ENABLE**
the channel is enable

enum **ft_transfer_mode**
Values:

enumerator **FT_ACQ_TYPE_FIFO**

enumerator **FT_ACQ_TYPE_DMA**

enum **fmctdc_ts_mode**
Enumeration for all possible time-stmap mode

Values:

enumerator **FMCTDC_TS_MODE_POST**
after post-processing

enumerator **FMCTDC_TS_MODE_RAW**
directly from ACAM chip. This should be used ONLY when debugging low level issues

Functions

const char ***fmctdc_strerror**(int err)
It returns the error message associated to the given error code

Parameters **err** – **[in]** error code

int **fmctdc_init**(void)
Init the library. You must call this function before use any other library function.

Returns 0 on success, otherwise -1 and errno is appropriately set

void **fmctdc_exit**(void)
It releases all the resources used by the library and allocated by fmctdc_init().

int **fmctdc_set_time**(struct fmctdc_board *b, struct *fmctdc_time* *t)
It sets the TDC base-time according to the given time-stamp. Note that, for the time being, it sets only seconds. Note that, you can set the time only when the acquisition is disabled.

Parameters

- **userb** – **[in]** TDC board instance token
- **t** – **[in]** time-stamp

Returns 0 on success, otherwise -1 and errno is set

int **fmctdc_get_time**(struct fmctdc_board *b, struct *fmctdc_time* *t)
It gets the base-time of a TDC device. Note that, for the time being, it gets only seconds.

Parameters

- **userb** – **[in]** TDC board instance token
- **t** – **[out]** time-stamp

Returns 0 on success, otherwise -1 and errno is set

int **fmctdc_set_host_time**(struct fmctdc_board *b)
It sets the TDC base-time according to the host time

Parameters **userb** – **[in]** TDC board instance token

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_wr_mode**(struct fmctdc_board *b, int on)
It enables/disables the WhiteRabbit timing system on a TDC device

Parameters

- **userb** – **[in]** TDC board instance token
- **on** – **[in]** white-rabbit status to set

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_check_wr_mode**(struct fmctdc_board *b)
It check the current status of the WhiteRabbit timing system on a TDC device

Parameters **userb** – **[in]** TDC board instance token

Returns 0 if it properly works, -1 on error and errno is set appropriately.

- ENOLINK if it is not synchronized and
- ENODEV if it is not enabled

float **fmctdc_read_temperature**(struct fmctdc_board *b)
It reads the current temperature of a TDC device

Parameters **userb** – **[in]** TDC board instance token

Returns temperature

int **fmctdc_channel_status_set**(struct fmctdc_board *userb, unsigned int channel, enum *fmctdc_channel_status* status)
The function enables/disables timestamp acquisition for the given channel.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to which we want change status
- **status** – **[in]** enable status to set

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_channel_enable**(struct fmctdc_board *userb, unsigned int channel)
It enables a given channel. NOTE: it is just a wrapper of fmctdc_channel_status_set()

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to which we want change status

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_channel_disable**(struct fmctdc_board *userb, unsigned int channel)
It disable a given channel. NOTE: it is just a wrapper of fmctdc_channel_status_set()

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to which we want change status

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_channel_status_get**(struct fmctdc_board *userb, unsigned int channel)
It gets the acquisition status of a TDC channel

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to which we want read the status

Returns the acquisition status (0 disabled, 1 enabled), otherwise -1 and errno is set appropriately

int **fmctdc_set_termination**(struct fmctdc_board *b, unsigned int channel, int enable)
The function enables/disables the 50 Ohm termination of the given channel. Termination may be changed anytime.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use
- **on** – **[in]** status of the termination to set

Returns 0 on success, otherwise a negative errno code is set appropriately

int **fmctdc_get_termination**(struct fmctdc_board *b, unsigned int channel)
The function returns current temrmination status: 0 if the given channel is high-impedance and positive if it is 50 Ohm-terminated.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use

Returns termination status, otherwise a negative errno code is set appropriately

int **fmctdc_get_buffer_type**(struct fmctdc_board *userb)
The function returns current buffer type: 0 for kmallo, 1 for vmalloc.

Parameters **userb** – **[in]** TDC board instance token

Returns buffer type, otherwise a negative errno code is set appropriately

int **fmctdc_set_buffer_type**(struct fmctdc_board *userb, enum *fmctdc_buffer_type* type)
The function sets the buffer type for a device

Parameters

- **userb** – **[in]** TDC board instance token

- **type** – **[in]** buffer type to use

Returns 0 on success, otherwise a negative errno code is set appropriately

int **fmctdc_get_buffer_mode**(struct fmctdc_board *userb, unsigned int channel)

The function returns current buffer mode: 0 for FIFO, 1 for circular buffer.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use

Returns buffer mode, otherwise a negative errno code is set appropriately

int **fmctdc_set_buffer_mode**(struct fmctdc_board *userb, unsigned int channel, enum *fmctdc_buffer_mode* mode)

The function sets the buffer mode for a channel

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use
- **mode** – **[in]** buffer mode to use

Returns 0 on success, otherwise a negative errno code is set appropriately

int **fmctdc_get_buffer_len**(struct fmctdc_board *userb, unsigned int channel)

The function returns current driver buffer length (number of timestamps)

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use

Returns buffer lenght, otherwise a negative errno code is set appropriately

int **fmctdc_set_buffer_len**(struct fmctdc_board *userb, unsigned int channel, unsigned int length)

The function set the buffer length

Internally, the buffer allocates memory in chunks of minimun 1KiB. This means, for example, that if you ask for 65 timestamp the buffer will allocate space for 128. This because 64 timestamps fit in 1KiB, to store 65 we need 2KiB (128 timestamps).

NOTE: it works only with the VMALLOC allocator.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** to use
- **length** – **[in]** maximum number of timestamps to store (min: 64)

Returns 0 on success, otherwise a negative errno code is set appropriately

int **fmctdc_set_offset_user**(struct fmctdc_board *userb, unsigned int channel, int32_t offset)

It sets the user offset to be applied on incoming timestamps. All the timestamps read from the driver (this means also from this library) will be already corrected using this offset.

Parameters

- **userb** – **[in]** TDC board instance token

- **channel** – **[in]** target channel [0, 4]
- **offset** – **[in]** the number of pico-seconds to be added

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_get_offset_user**(struct fmctdc_board *userb, unsigned int channel, int32_t *offset)
It get the current user offset applied to the incoming timestamps

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **offset** – **[out]** the number of pico-seconds to be added

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_transfer_mode**(struct fmctdc_board *userb, enum *ft_transfer_mode* *mode)
It gets the current transfer mode

Parameters

- **userb** – **[in]** TDC board instance token
- **mode** – **[out]** transfer mode

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_coalescing_timeout_set**(struct fmctdc_board *userb, unsigned int channel, unsigned int timeout_ms)

It sets the coalescing timeout on a given channel

It does not work per-channel for the following acquisition mechanism:

- FIFO (it will return the global IRQ coalescing timeout)

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **timeout_ms** – **[in]** ms timeout to trigger IRQ

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_coalescing_timeout_get**(struct fmctdc_board *userb, unsigned int channel, unsigned int *timeout_ms)

It gets the coalescing timeout from a given channel

It does not work per-channel for the following acuqisition mechanism:

- FIFO: there is a global configuration for all channels

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **timeout_ms** – **[out]** ms timeout to trigger IRQ

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_ts_mode_set**(struct fmctdc_board *userb, unsigned int channel, enum *fmctdc_ts_mode* mode)
It sets the timestamp mode

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **mode** – **[in]** time-stamp mode

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_ts_mode_get**(struct fmctdc_board *userb, unsigned int channel, enum *fmctdc_ts_mode* *mode)
It gets the timestamp mode

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **mode** – **[out]** time-stamp mode

Returns 0 on success, otherwise -1 and errno is set appropriately

struct fmctdc_board ***fmctdc_open**(int dev_id)

struct fmctdc_board ***fmctdc_open_by_lun**(int lun)
It opens one specific device by logical unit number (CERN/BE-CO-like). The function uses a symbolic link in /dev that points to the standard device. The link is created by the local installation procedure, and it allows to get the device id according to the LUN. Read also fmctdc_open() documentation.

Parameters **lun** – **[in]** Logical Unit Number

Returns an instance token, otherwise NULL and errno is appripriately set

int **fmctdc_close**(struct fmctdc_board*)
It closes a TDC instance opened with fmctdc_open() or fmctdc_open_by_lun()

Parameters **userb** – **[in]** TDC board instance token

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_fread**(struct fmctdc_board *b, unsigned int channel, struct *fmctdc_time* *t, int n)
this "fread" behaves like stdio: it reads all the samples. Read fmctdc_read() for more details about the function.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to use
- **t** – **[out]** array of time-stamps
- **n** – **[in]** number of elements to save in the array

Returns number of acquired time-stamps, otherwise -1 and errno is set appropriately

int **fmctdc_fileno_channel**(struct fmctdc_board *b, unsigned int channel)
It get the file descriptor of a TDC channel. So, for example, you can poll(2) and select(2). Note that, the file descriptor is the file-descriptor of a ZIO control char-device.

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to use

Returns a file descriptor, otherwise -1 and errno is set appropriately

int **fmctdc_read**(struct fmctdc_board *b, unsigned int channel, struct *fmctdc_time* *t, int n, int flags)

It reads a given number of time-stamps from the driver. It will wait at most once and return the number of samples that it received from a given input channel.

Timestamps are to the base time.

This "read" behaves like the system call and obeys O_NONBLOCK

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** channel to use [0, 4]
- **t** – **[out]** array of time-stamps
- **n** – **[in]** number of elements to save in the array
- **flags** – **[in]** tune the behaviour of the function. O_NONBLOCK - do not block

Returns number of acquired time-stamps, otherwise -1 and errno is set appropriately.

- EINVAL for invalid arguments
- EIO for invalid IO transfer
- EAGAIN if nothing ready to read in NONBLOCK mode

int **fmctdc_flush**(struct fmctdc_board *userb, unsigned int channel)

It removes all samples from the channel buffer. In order to doing this, the function temporary disable any active acquisition, only when the flush is completed the acquisition will be re-enabled

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_stats_recv_get**(struct fmctdc_board *userb, unsigned int channel, uint32_t *val)

It gets the number of received pulses (on hardware)

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **val** – **[out]** number of received pulses

Returns 0 on success, otherwise -1 and errno is set appropriately

int **fmctdc_stats_trans_get**(struct fmctdc_board *userb, unsigned int channel, uint32_t *val)

It gets the number of transferred timestamps

Parameters

- **userb** – **[in]** TDC board instance token
- **channel** – **[in]** target channel [0, 4]
- **val** – **[out]** number of transferred timestamps

Returns 0 on success, otherwise -1 and errno is set appropriately

uint64_t **fmctdc_ts_approx_ns**(struct *fmctdc_time* *a)
Set of mathematical functions on time-stamps

It provides a nano-second approximation of the timestamp.

Parameters **a** – **[in]** timestamp

Returns it returns the time stamp in nano-seconds

uint64_t **fmctdc_ts_ps**(struct *fmctdc_time* *a)
It provides a pico-seconds representation of the time stamp. Bear in mind that it may overflow. If you thing that it may happen, check the timestamp

Parameters **a** – **[in]** timestamp

Returns it returns the time stamp in pico-seconds

void **fmctdc_ts_norm**(struct *fmctdc_time* *a)
It normalizes the timestamp

Parameters **a** – **[inout]** timestamp

int **fmctdc_ts_sub**(struct *fmctdc_time* *r, const struct *fmctdc_time* *a, const struct *fmctdc_time* *b)
It perform the subtraction: r = a - b

Parameters

- **r** – **[out]** result
- **a** – **[in]** normalized timestamp
- **b** – **[in]** normalized timestamp

Returns 1 if the difference is negative, otherwise 0

void **fmctdc_ts_add**(struct *fmctdc_time* *r, const struct *fmctdc_time* *a, const struct *fmctdc_time* *b)
It perform an addiction: r = a + b

Parameters

- **r** – **[out]** result
- **a** – **[in]** normalized timestamp
- **b** – **[in]** normalized timestamp

int **_fmctdc_tscmp**(struct *fmctdc_time* *a, struct *fmctdc_time* *b)

Variables

const char *const **libfmctdc_version_s**
libfmctdc version string

const char *const **libfmctdc_zio_version_s**
zio version string used during compilation of libfmctdc

struct **fmctdc_time**
#include <fmctdc-lib.h> FMC-TDC time-stamp descriptor

Public Members

uint64_t **seconds**

```
TAI seconds. Note this is *not* an UTC time;
```

the counter does not support leap seconds. The internal counter is also limited to 32 bits (2038-error-prone).

uint32_t **coarse**

```
number of ticks of 8ns since the beginning of
```

the last second

uint32_t **frac**

```
fractional part of an 8 ns tick, rescaled
```

to (0..4095) range - i.e. 0 = 0 ns, and 4095 = 7.999 ns.

uint32_t **seq_id**
channel sequence number

uint32_t **debug**
debug stuff, driver/firmware-specific

CHAPTER

FOUR

THE MEMORY MAP

Following the memory map for the part of the ADC design that drives the FMC-TDC-1NS-5CH modules.

Warning: Unfortunatelly we are not able to include the memory map in PDF format. Please for the memory map refer to the online documentation,

4.1 Supported Designs

Here you can find the complete memory MAP for the supported designs. This will include the TDC register as well as the carrier registers and any other component used in an FMC-TDC-1NS-5CH design.

4.1.1 SPEC FMC-TDC-1NS-5CHA

The memory map is divided in two parts: the *Carrier* part common to all SPEC designs, and the *FMC-TDC-1NS-5CHA* part specific to the FMC-TDC-1NS-5CHA mezzanine.

Carrier

FMC-TDC-1NS-5CHA

4.1.2 SVEC FMC ADC 100M

The memory map is divided in two parts: the *Carrier* part common to all SPEC designs, and the *FMC-TDC-1NS-5CHA* part specific to the FMC-TDC-1NS-5CHA mezzanine.

Carrier

FMC-TDC-1NS-5CHA

INDEX

Symbols

F

L

P