FlxCard

Generated by Doxygen 1.12.0

Chapter 1

Introduction

This note defines an application program interface (API) for the use of the FLX PCIe I/O card in the ATLAS read-out system. The intention of the API is to satisfy the needs of simple test programs as well as the requirements of the FelixApplication.

This note defines an application program interface (API) for the use of the FLX PCIe I/O card in the ATLAS read-out system. The intention of the API is to satisfy the needs of simple test programs as well as the requirements of the FelixApplication.

1.1 Copyright

This is the header file for the FlxCard object

Author

Markus Joos, CERN Markus.Joos@cern.ch
Maintainance: Henk Boterenbrood, Nikhef boterenbrood@nikhef.nl

Copyright

1.2 Description of the API

This note defines an application program interface (API) for use with FLX PCIe cards.

The API can also be used for the Wupper release. Wupper is the OpenCore version of the PCIe DMA engine of the FLX firmware. See:

https://opencores.org/project/virtex7_pcie_dma

The main requirements that drove the design of this API as well as the underlying software were:

- Simplicity
  - The functions shall be easy to understand
  - The functions shall only provide basic operations. Complex functions can be built on the basis of this API in higher level code.
- Safety
  - The functions shall perform plausibility checks on user input and provide a detailed error reporting mechanism
- Speed
  - The implementation shall keep an eye on performance
- Portability
  - The implementation shall not use constructs that complicate the porting of the software to other platforms
- Flexibility
  - The API shall allow for the addition of features as required by the evolution of the FELIX project

1.3 Implementation Issues

The following issues are related to the implementation of the API:

1. Layered implementation
   - The source code that implements this API will be kept in one .cpp and one .h file. The layer below this API are the "flx" and "cmem_rcc" device drivers as well as the Linux OS. The functions of the API may also use selected low level packages from the ATLAS TDAQ project.
2. Multi-processing and multi-threading
   - The implementation of the API shall allow for several application programs and multiple threads within the same application program to use all functions of the API concurrently. It is up to the application programmers to add synchronisation as and where required.
3. Language binding
   - C++ was chosen for the language binding of the API.
4. Binary
   - The source code will be compiled into a single, shared library object named "libFlxCard.so". The library will exist in the form of a dynamically loadable object (shared library): libFlxCard.so. Applications should not use static linking
5. Buffer allocation
   - The API (i.e. libFlxCard.so) will not provide methods for buffer allocation. Application programmers are requested to use the cmem_rcc library directly in their code. If needed a wrapper for libcmem_rcc.so as a separate object (e.g. FelixContiguousBuffer) and library (e.g. lib FelixContiguousBuffer.so) can be added at a later time.
6. cmake will be used in order to build the library from the source code

1.4 Building the Software

Links to documentation on how to install additional software and build FlxCard can be found in:

https://gitlab.cern.ch/atlas-tdaq-felix/flxcard/blob/master/README.md

1.5 Support for debugging

The source code of the library (FlxCard.cpp) uses the DEBUG_TEXT macro

Example:

```
DEBUG_TEXT(DFDB_FELIX, 15, "card number = " << cardno);
```

If the code gets compiled without –DDEBUG_LEVEL=1 the macros will disappear (i.e. will not impact performance)
If the macros are enabled the generation of the debug text can be controlled via two variables:

```
#include "DFDebug/DFDebug.h"
  DF::GlobalDebugSettings::setup(dblevel, dbpackage);
```

dbpackage defines the package for which text is output. In order to activate line in the example above one has to set dbpackage=DFDB_FELIX dblevel defines the level of verbosity. Only messages that have a level equal or smaller than dblevel will be printed.

1.6 Error reporting

All errors will be reported through exceptions. For that purpose an exception class, FlxException.cpp, has been added. The FlxException object inherits from std::runtime_error and provides some additional data members such as an error code.

At the level of the FlxCard source code a macro is used for throwing exceptions:

```
#define THROW_FLX_EXCEPTION(errorCode, message) throw FlxException(FlxException::errorCode, message);
```

1.7 Organization of this Document

Section 'Application Program Interface' contains the definition of the API. For each function the section gives a detailed description of all input and output parameters, a description of the functionality and the return codes. The section contains sub-section for the type definitions used by the API as well as functions concerning the return codes.

1.8 Application Program Interface

The methods of the API are grouped by function into several categories.

For the implementation of the API it will be necessary to develop service functions for internal use. They are not described in this API document.

The remarks below apply to all functions defined in the API:

- If not stated otherwise, all functions of this API are non-blocking, i.e. they return immediately indicating an error code if necessary. Wherever functions are blocking, i.e. waiting on external events, e.g. end of block transfer, this is stated explicitly.
- The implementation of the API is in the following called the “FLX library”.

1.9 Driver calls

In order to optimize user code for performance and to spot potential race conditions between processes and threads it is important to know details about the interface between this library and the device driver underneath. The methods listed in the table below make a call to the driver (typically via an `ioctl()` function or to `mmap()`).

Method	Type of driver access
card_open()	open() and ioctl(SETCARD)
card_close()	close()
dma_max_tlp_bytes()	ioctl(GET_TLP)
irq_enable()	ioctl(UNMASK_IRQ)
irq_disable()	ioctl(MASK_IRQ)
irq_wait()	ioctl(WAIT_IRQ)
irq_cancel()	ioctl(CANCEL_IRQ_WAIT)
irq_clear()	ioctl(CLEAR_IRQ)
irq_reset_conters()	ioctl(RESET_IRQ_COUNTERS)
map_memory_bar()	mmap()
number_of_cards()	ioctl(GETCARDS)

1.10 Background information about I2C transfers

This section provides some information about how I2C transfers work in the FELIX system. At the level of the software tools several of the applications in the flxcard package perform I2C transfers. The program flx-i2c allows for the execution of single transfers. Program `flx-init` provides functions that execute lists of I2C transactions in order to e.g. set the clock frequency of the TTC interface.

Description of flx-i2c:

Command “flx-i2c list” produces a list of I2C devices. Below is the output for an FLX-712. FLX-709, FLX-710, FLX-711 and FLX-128 cards are of course also supported.

```
> flx-i2c list
Card model: FLX-712
Switch I2C address: 0x70
=> List of I2C devices:
Device              Model                  Switch port(s)    Address
======================================================================
ADN2814             ADN2814                  1:-:-            0x40
```

```
SIS53154            SI53154             1:-:-           0x6b
LTC2991_1           LTC2991             1:-:-           0x48
LTC2991_2           LTC2991             1:-:-           0x49
SI5345              SI5345              2:-:-           0x68
TCA6408A            TCA6408A            2:-:-           0x20
MINIPOD-TX1         AFBR-814PxyZ        3:-:-           0x2c
MINIPOD-TX2         AFBR-814PxyZ        3:-:-           0x2d
MINIPOD-TX3         AFBR-814PxyZ        3:-:-           0x2e
MINIPOD-TX4         AFBR-814PxyZ        3:-:-           0x2f
MINIPOD-RX1         AFBR-824PxyZ        3:-:-           0x30
MINIPOD-RX2         AFBR-824PxyZ        3:-:-           0x31
MINIPOD-RX3         AFBR-824PxyZ        3:-:-           0x32
MINIPOD-RX4         AFBR-824PxyZ        3:-:-           0x33
```

The function of the FELIX can be implemented on several types of PCI cards. They are known as FLX-709, FLX-710, FLX-711, FLX-712 and FLX-128. Each type of PCI card has its own I2C tree. This tree connects to the FPGA. Therefore all I2C commands originate in the firmware and the used programs I2C transactions by reading from and writing to registers of the firmware (via PCIe).

The switch I2C addresses are the address of the single I2C switches on the FELIX cards. All I2C devices (see below) connect to these switches. Hence every I2C access from the firmware is first routed to a switch and from there reaches the endpoint.

On a FLX with more than one switch, the first one is connected to the firmware (primary switch) and the second connects to one of the ports of the first; therefore, it is a secondary switch that cannot be reached directly from the firmware.

The output of “flx-i2c list” is what we call the “I2C model”. At the level of the software it is defined as an array of structures of type “i2c_device_t” in FlxCard.cpp.

Below is a description of the columns of the table above:

Column	Description
Device	This is the name of a discrete chip with a I2C interface
Model	This is the actual part number of the device
Switch port	An I2C switch has 8 ports. Each port has its own range of 128 device addresses. Therefore up to 128 I2C devices could be connected to each port in principle. The first number is the switch port (0..7). The number after the “:” characters is the secondary and tertiary switch number in case of cascaded switches (present on certain hardware platforms), or '-' if that number does not apply. Therefore e.g. “7:-:-” refers to a port that connects directly to I2C endpoint devices while “4:3:-” means that two switches are cascaded. The first number (“4”) is the port of the first switch and the second number (“3”) refers to the port of the second switch.
Address	This is a 7-bit address value that selects the I2C endpoint device. The addresses of the endpoints are hardwired in the hardware design.

The applications internally use functions of the flxcard API: i2c_devices_read(), i2c_devices_write(), i2c_read_byte() and i2c_write_byte(). See the appropriate section in another chapter for the details of these methods.

Chapter 2

Topic Index

2.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Topic Documentation

5.1 Driver interaction

Functions

- void FlxCard::card_open (int device_nr, u_int lock_mask, bool read_config=false, bool ignore_version=false)
- void FlxCard::card_close ()
- static u_int FlxCard::number_of_cards ()
- static u_int FlxCard::number_of_devices ()
- static device_list_t FlxCard::device_list ()
- static int FlxCard::card_to_device_number (int card_number)
- u_int FlxCard::lock_mask (int device_nr)
- u_int **FlxCard::get_lock_mask** (int device_nr)
- u_long FlxCard::openBackDoor (int bar)
- u_long FlxCard::bar0Address ()
- u_long FlxCard::bar1Address ()
- u_long FlxCard::bar2Address ()

5.1.1 Detailed Description

A set of functions that interact with the driver, there are methods to open the API for instance, but also to open an alternative (backdoor) way to reading and writing the registers by memory-mapping a struct directly on the register map (BAR space).

5.1.2 Function Documentation

5.1.2.1 card_open()

```
void FlxCard::card_open (
            int device_nr,
            u_int lock_mask,
            bool read_config = false,
            bool ignore_version = false)
```

Opens the *flx* driver and links one Flx PCIe card to the FlxCard object and has to be called before any other reference to the method is made. In case of problems, check (more /proc/flx) if the driver is running and if it can see all cards. The method also checks if the major version of the register map of the F/W running on the FLX card matches with the version of the "regmap" library. This is to prevent running the API on incompatible FLX cards.

Each FLX device has a number of resources that cannot be shared by multiple processes, such as a DMA controller or the onboard I2C-bus. The purpose of the resource locking bits is to allow a process to declare to the driver that it requires exclusive access to such a resource. If a resource is locked by a process, the driver will refuse other processes that request access to the same resource. In such a case card_open() throws an exception. The parameter *lock_mask* passes a collection of bits to the driver. These bits are defined in FlxCard.h. Currently the following resources are defined:

```
#define LOCK_DMA0     1
#define LOCK_DMA1     2
#define LOCK_I2C      4
#define LOCK_FLASH    8
#define LOCK_ELINK    16
#define LOCK_ALL      0xffffffff
```

Note that DMA-related lock bits are currently also located in bits 16 and up: there is copy of LOCK_DMA0 and LOCK_DMA1 (in bit 16 and 17 resp.) and space for more DMA lock bits (required for next-generation firmware versions).

Example: To lock access to DMA1 and the FLASH memory of the first FLX device:

```
card_open( 0, LOCK_DMA1 | LOCK_FLASH )
```

The lock will be held until the owning process calls card_close(). If a process terminates with an error the driver will release the resources that were locked by that process. Users can find an overview of what resources are locked by looking at the content of */proc/flx*. Processes that do not request any locks still have full access to the respective resources. That is to say that a process that does not lock DMA1 still can call the DMA related methods of the API. The method lock_mask() can be called to figure out what resources are locked. This was done on purpose because the application programmers know best if their applications (for example for diagnostic purposes) should be able to run even if a resource is locked. Consequently, the application programmers bear the responsibility for using the resource locking in a correct way.

Errors In case of an error, an exception is thrown.

Parameters

device_nr	The number of the FLX device (0. . . N-1) that is to be linked to the object
lock_mask	The resources that are to be locked, defined as a bit mask
read_config	Read and locally store the (e-)link configuration
ignore_version	For some tools it may be useful to be able to open the FELIX device despite a mismatch between software and firmware version (in that case results in a warning, but does not throw an exception)

5.1.2.2 card_close()

```
void FlxCard::card_close ()
```

Unlinks a Flx PCIe card. It must be called before closing the application.

Errors In case of an error, an exception is thrown.

5.1.2.3 number_of_cards()

```
static u_int FlxCard::number_of_cards ()  [static]
```

Returns the number of FLX *cards* that are installed in the host computer, taking into account single-device FLX-709 cards. As it is a static method you do not have to instantiate an FlxCard object to call it. For errors see method number_of_devices().

5.1.2.4 number_of_devices()

```
static u_int FlxCard::number_of_devices ()  [static]
```

Returns the number of FLX *devices* (PCIe endpoints) that are installed in the host computer. As it is a static method you do not have to instantiate an FlxCard object to call it.

Errors In case of an error, no exception is thrown but “0” is returned. Therefore “zero cards found” can also mean that:

- the flx driver was not running
- the device node `/dev/flx` was missing
- the "GETCARDS" ioctl() of the flx driver did not work

5.1.2.5 device_list()

```
static device_list_t FlxCard::device_list ()  [static]
```

Returns a structure with information about the cards and devices installed in the PC. As it is a static method you do not have to instantiate an FlxCard object to call it.

The parameter `device_list_t.n_devices` gives the total number of devices. It is identical to the value returned by number_of_devices() The value of `device_list_t.cdmap[device][0]` (with device = 0...(`device_list_t.n_devices`-1) is the type of the card as read from the *CARD_TYPE* register (offset 0xA0 of BAR2) The value of `device_list_t.cdmap[device][1]` (with device = 0...(`device_list_t.n_devices`-1) is the relative number of the device with respect to the card. That means that for a FLX-709 and FLX-710 it will always be "0" because these are single device cards. For a FLX-711, FLX-712 or FLX-128 it can be "0" or "1".

Errors In case of an error, an exception is thrown.

5.1.2.6 card_to_device_number()

```
static int FlxCard::card_to_device_number (
            int card_number)  [static]
```

Returns the FELIX device number associated with the first device of the selected card number. if for example the host machine contains two FLX-712 cards then card 0 is accessed through device 0, and card 1 through device 2. If on the other hand there is one FLX-709 and one FLX-712 card in the machine, then it could be either devices 0 and 1 or devices 0 and 2, depending on the order in which the cards are enumerated by the host system. This function is used by tools that act on a card rather than a device (for example, card I2C-bus access is through device 0 of the card only).

Parameters

card_number	The number of the FLX card (0. . . N-1) in the host machine

5.1.2.7 lock_mask()

```
u_int FlxCard::lock_mask (
            int device_nr)
```

Returns information about a device's resources that are currently locked by *other* instances of FlxCard objects, which may reside in either the same process or in different processes. The value returned is a global resource-lock value retrieved from the driver. The individual lock bits are declared in FlxCard.h

Parameters

device_nr	The number of the FLX device (0. . . N-1)

Example:

```
FlxCard flxcard;
u_int lock_bits = flxcard.lock_mask( 0 );
if( lock_mask & LOCK_DMA1 )
  cout « "Device 0 DMA1 is locked (by others)"
```

Note This method can be called without first calling open_card().

Errors In case of an error, an exception is thrown.

5.1.2.8 openBackDoor()

```
u_long FlxCard::openBackDoor (
            int bar)
```

Returns the PCI base address of the specified PCI register block.

Parameters

bar	The number of a BAR register block; legal values are 0, 1 or 2

Returns

the PCI base address of the specified PCI register block

5.1.2.9 bar0Address()

```
u_long FlxCard::bar0Address ()  [inline]
```

Is an alias for `openBackDoor(0)`.

5.1.2.10 bar1Address()

```
u_long FlxCard::bar1Address ()  [inline]
```

Is an alias for `openBackDoor(1)`.

5.1.2.11 bar2Address()

```
u_long FlxCard::bar2Address ()  [inline]
```

Is an alias for `openBackDoor(2)`.

5.2 DMA

Functions

- int FlxCard::dma_max_tlp_bytes ()
- void FlxCard::dma_to_host (u_int dma_id, u_long dst, size_t size, u_int flags)
- void FlxCard::dma_from_host (u_int dma_id, u_long src, size_t size, u_int flags)
- bool FlxCard::dma_enabled (u_int dma_id)
- void FlxCard::dma_wait (u_int dma_id)
- void FlxCard::dma_stop (u_int dma_id)
- void FlxCard::dma_advance_ptr (u_int dma_id, u_long dst, size_t size, size_t bytes)
- void FlxCard::dma_set_ptr (u_int dma_id, u_long dst)
- u_long FlxCard::dma_get_ptr (u_int dma_id)
- u_long **FlxCard::dma_get_read_ptr** (u_int dma_id)
- void FlxCard::dma_reset ()
- u_long FlxCard::dma_get_fw_ptr (u_int dma_id)
- u_long **FlxCard::dma_get_current_address** (u_int dma_id)
- bool FlxCard::dma_cmp_even_bits (u_int dma_id)

5.2.1 Detailed Description

A set of functions to setup, start and stop DMA transfers and to interact with the circular buffer PC pointers.

5.2.2 Function Documentation

5.2.2.1 dma_max_tlp_bytes()

```
int FlxCard::dma_max_tlp_bytes ()  [inline]
```

Calls the flx driver in order to determine the maximum number of TLP bytes that the H/W can support.

Errors In case of an error, an exception is thrown.

5.2.2.2 dma_to_host()

```
void FlxCard::dma_to_host (
            u_int dma_id,
            u_long dst,
            size_t size,
            u_int flags)
```

First stops the DMA channel identified by dma_id. It does not check if the channel is busy. Then it programs the channel and starts a new DMA from-device-to-host operation.

Note: The bits 10:0 of the DMA descriptor (bitfield NUM_WORDS in the Wupper documentation) will be set to the maximum TLP size supported by the system. Therefore, the transfer size has to be a multiple of that value.

Errors In case of an error, an exception is thrown.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7
dst	The value for the start_address field of the descriptor. NOTE: You have to provide a physical memory address
size	The size of the transfer in bytes
flags	The value for the wrap_around field of the descriptor; "1" means: enable wrap around, i.e. use continuous DMA

5.2.2.3 dma_from_host()

```
void FlxCard::dma_from_host (
             u_int dma_id,
             u_long src,
             size_t size,
             u_int flags)
```

First stops the DMA channel identified by *dma_id*. It does not check if the channel is busy. Then it programs the channel and starts a new DMA from-host-to-device operation.

Note: The size of the transfer has to be a multiple of 32 bytes.

The method internally computes the optimal value for the bits 10:0 of the DMA descriptor (bitfield NUM_WORDS in the Wupper documentation).

The algorithm used is this:

1. if transfersize % 32 != 0: error("number of bytes transferred must be a multiple of 32")
2. tlp = get_max_tlp()
3. if transfersize % tlp == 0: do transfer
4. else: tlp = tlp >> 1 && goto 3

Note: The algorithm above is currently not used: the transfer unit size is set to the minimum of 32 bytes, to make sure small messages are always sent, in particular when continuous DMA is enabled. The upper 16 bits of the flags parameter can be used to force a transfer size unequal to (so larger than) 32 bytes (but must be a multiple of 32).

Errors In case of an error, an exception is thrown.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7
src	The value for the start_address field of the descriptor. NOTE: You have to provide a physical memory address
size	The size of the transfer in bytes
flags	The value for the wrap_around field of the descriptor. “1” means: enable wrap around, i.e. use continuous DMA

5.2.2.4 dma_enabled()

```
bool FlxCard::dma_enabled (
             u_int dma_id)
```

Returns whether the DMA channel identified by dma_id is enabled or not.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7

5.2.2.5 dma_wait()

```
void FlxCard::dma_wait (
            u_int dma_id)
```

Is blocking. It returns once the DMA on channel dma_id has ended. The method has an internal (hard wired) time out of 1 second.

Errors In case the DMA has not ended after 1 second an exception is thrown.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7

5.2.2.6 dma_stop()

```
void FlxCard::dma_stop (
            u_int dma_id)
```

Clears the DMA channel identified by dma_id. It does not check the status of that channel.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7

5.2.2.7 dma_advance_ptr()

```
void FlxCard::dma_advance_ptr (
            u_int dma_id,
            u_long dst,
            size_t size,
            size_t bytes)
```

Advances the internal read pointer of the DMA channel by the number of bytes given. This is used when operating the DMA engine in "endless DMA" / "circular buffer" mode. This method is to be called by the user after processing data at the head of the circular buffer to free the buffer for DMA writes or reads of the FLX card again.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7
dst	See below. NOTE: You have to provide a physical memory address
size	If the value of the read_ptr field, after adding bytes is larger than dst + size, size will be subtracted from read_ptr
bytes	This value will be added to the *read_ptr* field of the DMA descriptor

5.2.2.8 dma_set_ptr()

```
void FlxCard::dma_set_ptr (
            u_int dma_id,
            u_long dst)
```

Directly writes the read pointer of a DMA channel.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7
dst	The read pointer register of the DMA channel will be set to the value in *dst*

5.2.2.9 dma_get_ptr()

```
u_long FlxCard::dma_get_ptr (
            u_int dma_id)
```

Returns the current value of the SW_POINTER. That is to say the bits 127:64 of the DMA_DESC_[0..7]a as defined in the Wupper documentation.

Parameters

dma_id	The DMA channel (descriptor) to be used; valid numbers are 0..7

5.2.2.10 dma_reset()

```
void FlxCard::dma_reset ()
```

Triggers a reset by the DMA_RESET register in BAR0.

5.2.2.11 dma_get_fw_ptr()

```
u_long FlxCard::dma_get_fw_ptr (
            u_int dma_id)
```

Reads the status register of a DMA descriptor (i.e. one of the DMA_DESC_STATUS_∗ registers) and returns the value of the FW_POINTER bit field (bits 63:0).

Parameters

dma_id	The DMA channel (descriptor) to be queried; valid numbers are 0..7

5.2.2.12 dma_cmp_even_bits()

```
bool FlxCard::dma_cmp_even_bits (
            u_int dma_id)
```

Reads the *even_addr_dma* and *even_pc_dma* bits of a DMA channel and compares them.

Parameters

dma↩ _id	The DMA channel (descriptor) to be used; valid numbers are 0..7

Returns

Returns *true* if the bits have the same value and *false* if not.

5.3 I2C

Functions

- void FlxCard::i2c_read (const char ∗device_str, u_char reg_addr, u_char ∗value, int nbytes=1)
- void FlxCard::i2c_write (const char ∗device_str, u_char reg_addr, u_char data)
- void **FlxCard::i2c_write** (const char ∗device_str, u_char reg_addr, u_char ∗data, int nbytes)
- void FlxCard::i2c_write_byte (u_char dev_addr, u_char byte)
- void FlxCard::i2c_write_byte (u_char dev_addr, u_char reg_addr, u_char byte)
- u_char FlxCard::i2c_read_byte (u_char dev_addr, u_char reg_addr)
- bool FlxCard::i2c_flush (u_int ∗count=0)
- void FlxCard::i2c_write_bytes (uint8_t dev_addr, uint8_t reg_addr, int nbytes, uint8_t ∗bytes)
- void **FlxCard::i2c_read_bytes** (uint8_t dev_addr, uint8_t reg_addr, int nbytes, uint8_t ∗bytes)

5.3.1 Detailed Description

A set of functions that interact with the onboard I2C bus.

5.3.2 Function Documentation

5.3.2.1 i2c_read()

```
void FlxCard::i2c_read (
            const char * device_str,
            u_char reg_addr,
            u_char * value,
            int nbytes = 1)
```

These methods read and write 8-bit data words from/to a named I2C device.

Errors In case of an error, an exception is thrown.

Parameters

∗*device_str*	A string of the form “p1:p2:p3:addr”: see table below for details. Alternatively the string can be the name of the I2C endpoint device You must use the predefined names: find out by running "flx-i2c list"
reg_addr	The address of the register within the device
data	The data that is to be written to the register
value	A pointer to a value that will be filled with the data read from the I2C device

Description of the sub-parameters in parameter *device_str:*

Sub-parameter	Description
p1	The port number on the primary I2C switch to select; run "flx-i2c list" to find out where the I2C device you want to access, is located in the I2C tree
p2	In case of cascaded switches the secondary switch's port number to select
p3	In case of cascaded switches this is the third switch's port number
addr	The I2C address of the endpoint device behind the switch(es)

As an alternative to a string of the format defined above, parameter *device_str* can a symbolic name of an I2C endpoint device; get a listing of all available I2C devices on all platforms by running "flx-i2c list all".

5.3.2.2 i2c_write()

```
void FlxCard::i2c_write (
            const char * device_str,
            u_char reg_addr,
            u_char data)
```

See i2c_read() for details

5.3.2.3 i2c_write_byte() [1/2]

```
void FlxCard::i2c_write_byte (
            u_char dev_addr,
            u_char byte)
```

These methods read or write one byte of data to/from an I2C address. In case the I2C is stuck the method will abort after 1 second with a time-out error. Before transferring the byte the methods call private function *i2c_wait_not_full()* in order to make sure that the I2C interface is not busy.

Errors In case of an error, an exception is thrown.

Parameters

dev_addr	The address of the I2C device
byte	The byte that is to be written to the I2C device

5.3.2.4 i2c_write_byte() [2/2]

```
void FlxCard::i2c_write_byte (
            u_char dev_addr,
            u_char reg_addr,
            u_char byte)
```

Parameters

dev_addr	The address of the I2C device
reg_addr	The register address inside the I2C device
byte	The byte that is to be written to the register in the I2C device

5.3.2.5 i2c_read_byte()

```
u_char FlxCard::i2c_read_byte (
            u_char dev_addr,
            u_char reg_addr)
```

Parameters

dev_addr	The address of the I2C device
reg_addr	The register address inside the I2C device to read from

5.3.2.6 i2c_flush()

```
bool FlxCard::i2c_flush (
            u_int * count = 0)
```

Can be used to flush the I2C read FIFO. Up to 16 bytes are flushed, the exact number returned optionally in 'count'.
Errors Returns false if after 16 bytes flushed the FIFO is still not empty.

Parameters

count	Number of bytes flushed

5.3.2.7 i2c_write_bytes()

```
void FlxCard::i2c_write_bytes (
            uint8_t dev_addr,
            uint8_t reg_addr,
            int nbytes,
            uint8_t * bytes)
```

Read or write one or two bytes of data to/from an I2C address plus register address.

5.4 GBT and transceiver related methods

Functions

- u_int FlxCard::gbt_setup (int alignment, int channel_mode)
- long int FlxCard::gbt_version_delay (u_long svn_version, char *filename)
- u_long FlxCard::rxusrclk_freq (u_int channel)
- void FlxCard::gth_rx_reset (int quad=-1)

5.4.1 Detailed Description

A set of functions for setting up GBT links.

5.4.2 Function Documentation

5.4.2.1 gbt_setup()

```
u_int FlxCard::gbt_setup (
            int alignment,
            int channel_mode)
```

Is used to initialize the GBT Wrapper registers, and to perform the TX and RX configuration.

Errors In case of an error, an exception is thrown.

Parameters

alignment	Tx time domain crossing mode selection for each channel (GBT TX_TC_METHOD)
channel_mode	Set GBT mode (normal FEC or Wide-bus mode) for all channels. (GBT_DATA_TXFORMAT, GBT_DATA_RXFORMAT)

5.4.2.2 gbt_version_delay()

```
long int FlxCard::gbt_version_delay (
            u_long svn_version,
            char * filename)
```

Is used to load the tx phase values from a file, for subsequent configuration of the registers *GBT_TX_TC_DLY_VALUE1* to *GBT_TX_TC_DLY_VALUE4* before the GBT initialization.

Errors In case of an error, an exception is thrown.

Parameters

svn_version	The current SVN (old) of GIT (new) version
filename	The file to load the tx phase values from

5.4.2.3 rxusrclk_freq()

```
u_long FlxCard::rxusrclk_freq (
            u_int channel)
```

This function will select the channel through the register RXUSRCLK_FREQ_CHANNEL. Then it will wait for the register RXUSRCLK_FREQ_VALID to go high, indicating that the measurement is done. The return value will be read from RXUSRCLK_FREQ_VAL and it represents the measured frequency in Hz.
Errors In case of an error, an exception is thrown; if the functionality has not (yet) been implemented in the FPGA firmware, the function throws a TIMEOUT exception. This also happens when a non-existing channel is selected.

Parameters

channel	The transceiver channel number on which the recovered receiver clock (rxusrclk) has to be measured
value	Frequency of the recovered receiver clock of the selected channel in Hz

5.4.2.4 gth_rx_reset()

```
void FlxCard::gth_rx_reset (
            int quad = -1)
```

Resets the GTH receivers. It method is meant for FULL mode F/W only.

Parameters

quad	The quad to be reset; legal values are 0..5

5.5 Interrupt

Functions

- void FlxCard::irq_enable (u_int interrupt=ALL_IRQS)
- void FlxCard::irq_disable (u_int interrupt=ALL_IRQS)
- void FlxCard::irq_wait (u_int interrupt)
- void FlxCard::irq_clear (u_int interrupt=ALL_IRQS)
- void FlxCard::irq_cancel (u_int interrupt=ALL_IRQS)
- void FlxCard::irq_reset_counters (u_int interrupt=ALL_IRQS)

5.5.1 Detailed Description

This section describes the interrupt handler (IRQ) related functions.

5.5.2 Function Documentation

5.5.2.1 irq_enable()

```
void FlxCard::irq_enable (
            u_int interrupt = ALL_IRQS)
```

Enables one interrupt channel or all channels of one FLX card. If called with interrupt set to the number of an interrupt only this channel will be enabled. If called with interrupt set to ALL_IRQS or if the interrupt parameter is omitted, all channels of the given card will be enabled.

This method calls a function of the flx device driver. **Errors** In case of an error, an exception is thrown.

Parameters

interrupt	The number of the interrupt; legal values are 0..7 and ALL_IQRS

5.5.2.2 irq_disable()

```
void FlxCard::irq_disable (
            u_int interrupt = ALL_IRQS)
```

Disables one interrupt channel or all channels of one FLX card. If called with interrupt set to the number of an interrupt only this channel will be disabled. If called with interrupt set to ALL_IRQS or if the interrupt parameter is omitted, all channels of the given card will be disabled.

This method calls a function of the flx device driver.

Errors In case of an error, an exception is thrown.

Parameters

interrupt	The number of the interrupt; legal values are 0..7 and ALL_IRQS

5.5.2.3 irq_wait()

```
void FlxCard::irq_wait (
             u_int interrupt)
```

Is blocking. It suspends the execution of a user application until the interrupts of the number specified in interrupt has been received by the flx driver.

The waiting takes place in the driver.

Errors In case of an error, an exception is thrown.

Parameters

interrupt	The number (0..7) of the interrupt that is to be waited for

5.5.2.4 irq_clear()

```
void FlxCard::irq_clear (
            u_int interrupt = ALL_IRQS)
```

Instructs the driver to clear unsolicited interrupts. These are interrupts that may have been received by the driver while no user application was waiting for them. If the driver has received an unsolicited interrupt and irq_wait() is called by an application without clearing the interrupt, the application will continue immediately instead of waiting for an interrupt.

As this method interferes with the interrupt flags of the device driver it should only be called if no applications are waiting for the interrupts specified in interrupt.

Errors In case of an error, an exception is thrown.

Parameters

interrupt	The number (0..7) of the interrupt that is to be cleared or ALL_IQRS to clear all interrupts of that given card

5.5.2.5 irq_cancel()

```
void FlxCard::irq_cancel (
            u_int interrupt = ALL_IRQS)
```

Instructs the driver to cancel wait requests for one particular interrupt or for all interrupts of one FLX card. It will therefore unblock applications that are waiting for an interrupt. As the function is setting an internal flag of the driver in order to simulate an interrupt by S/W it should only be called if an application is blocked.

Errors In case of an error, an exception is thrown.

Parameters

interrupt	The number (0..7) of the interrupt that is to be cancelled or ALL_IQRS to cancel all interrupts of that given card

5.5.2.6 irq_reset_counters()

```
void FlxCard::irq_reset_counters (
            u_int interrupt = ALL_IRQS)
```

Resets the counters of one or all interrupts of the given FLX card. The counters count the number of times a given interrupt has been signalled. Their values are shown in the file /proc/flx. This method makes a call to the driver.

Errors In case of an error, an exception is thrown.

Parameters

interrupt	The number (0..7) of the interrupt that is to be reset or ALL_IQRS to reset all interrupts of that given card

5.6 Register and bitfield access

Functions

- u_long FlxCard::cfg_get_option (const char ∗key, bool show_options=false)
- void FlxCard::cfg_set_option (const char ∗key, u_long value, bool show_options=false)
- u_long FlxCard::cfg_get_reg (const char ∗key)
- void FlxCard::cfg_set_reg (const char ∗key, u_long value)
- void FlxCard::registers_reset ()
- static regmap_register_t ∗ FlxCard::cfg_register (const char ∗name)
- static regmap_bitfield_t ∗ FlxCard::cfg_bitfield (const char ∗name)
- static std::string FlxCard::cfg_bitfield_options (const char ∗name, bool include_all_substr=true)

5.6.1 Detailed Description

A set of functions for looking up registers and bitfields by name (string) and to get and set individual bitfields and full 64-bits registers, by name.

5.6.2 Function Documentation

5.6.2.1 cfg_get_option()

```
u_long FlxCard::cfg_get_option (
            const char * key,
            bool show_options = false)
```

Reads the value of a bit field and returns its value as the value of the method.
The method is only for the registers in BAR2. For BAR0 and BAR1 register access use the bar0Address() and bar1Address() functions and map the pointer to the respective structure from FlxCard.h.

Errors In case of an error, an exception is thrown.

Parameters

key	A string with the name of the bit field
show_options	If true displays (on standard output) a list of bitfield options that match 'key' if a unique match was not found

5.6.2.2 cfg_set_option()

```
void FlxCard::cfg_set_option (
            const char * key,
            u_long value,
            bool show_options = false)
```

Writes a value to a bit field. The method is only for the registers in BAR2. For BAR0 and BAR1 register access use the bar0Address() and bar1Address() functions and map the pointer to the respective structure from FlxCard.h.

Errors In case of an error, an exception is thrown.

Parameters

∗key	A string with the name of the register or bit field
value	The value that is to be written to the register
show_options	If true displays (on standard output) a list of bitfield options that match 'key', if a unique match was not found

5.6.2.3 cfg_get_reg()

```
u_long FlxCard::cfg_get_reg (
            const char * key)
```

Reads the value of a register or and returns its value as the value of the method.
The method is only for the registers in BAR2. For BAR0 and BAR1 register access use the bar0Address() and bar1Address() functions and map the pointer to the respective structure from FlxCard.h.

Errors In case of an error, an exception is thrown.

Parameters

key	A string with the name of the register

5.6.2.4 cfg_set_reg()

```
void FlxCard::cfg_set_reg (
            const char * key,
            u_long value)
```

Writes a value to a register. The method is only for the registers in BAR2. For BAR0 and BAR1 register access use the bar0Address() and bar1Address() functions and map the pointer to the respective structure from FlxCard.h.

Errors In case of an error, an exception is thrown.

Parameters

∗key	A string with the name of the register or bit field
value	The value that is to be written to the register

5.6.2.5 registers_reset()

```
void FlxCard::registers_reset ()
```

Triggers a reset by the REGISTERS_RESET register in BAR0. This causes all registers in the BAR2 area to be reset to their power-on default values. The registers in BAR0 and BAR1 will keep their values.

5.6.2.6 cfg_register()

```
static regmap_register_t * FlxCard::cfg_register (
            const char * name)  [static]
```

Returns a pointer to a `regmap_register_t` struct in the array of these structs describing all the FELIX registers. Internally used by cfg_get_reg() and cfg_set_reg().

Parameters

name	The string containing the register name: is case-insensitive and '_' characters may be replaced by '-'

5.6.2.7 cfg_bitfield()

```
static regmap_bitfield_t * FlxCard::cfg_bitfield (
            const char * name)  [static]
```

Returns a pointer to a `regmap_bitfield_t` struct in the array of these structs describing all the FELIX register bitfield. Internally used by cfg_get_option() and cfg_set_option().

Parameters

name	The string containing the bitfield name: is case-insensitive and '_' characters may be replaced by '-'

5.6.2.8 cfg_bitfield_options()

```
static std::string FlxCard::cfg_bitfield_options (
            const char * name,
            bool include_all_substr = true)  [static]
```

Compiles and returns a string containing a list of register bitfield names that match the given name, meaning that string 'name' is either a substring of the full bitfield name or that the full bitfield name starts with 'name'. Internally used by cfg_get_option() and cfg_set_option().

Parameters

name	The string with the name substring: is case-insensitive and '_' characters may be replaced by '-'
include_all_substr	If false only bitfield names that start with the given substring are returned, if true names are returned that contain the substring anywhere in their name

5.7 Tools

Functions

- u_int FlxCard::card_model ()
- u_int **FlxCard::card_type** ()
- u_int FlxCard::firmware_type ()
- std::string FlxCard::firmware_type_string ()
- std::string FlxCard::firmware_string ()
- bool FlxCard::lpgbt_type ()
- bool FlxCard::lpgbt_fec12 ()
- bool FlxCard::lpgbt_5gbps ()
- bool FlxCard::fullmode_type ()
- void FlxCard::soft_reset ()
- void FlxCard::cr_fromhost_reset ()
- u_int FlxCard::number_of_channels ()
- std::pair< int, int > FlxCard::ec_elink_indices ()
- std::pair< int, int > FlxCard::ic_elink_indices ()

5.7.1 Detailed Description

A set of functions that can mostly be described as methods that interact with the card or device endpoint.

5.7.2 Function Documentation

5.7.2.1 card_model()

```
u_int FlxCard::card_model ()  [inline]
```

Returns the H/W type of the FLX card, read in card_open(). Currently these types are known (and supported): FLX-709, FLX-710, FLX-711, FLX-712, FLX-128, FLX-181, FLX-182, FLX-155

Returns

integer value of the card model, a number like 709, 712, etc.

5.7.2.2 firmware_type()

```
u_int FlxCard::firmware_type ()  [inline]
```

Returns the firmware type of the FLX card, read in card_open().

5.7.2.3 firmware_type_string()

```
std::string FlxCard::firmware_type_string ()  [inline]
```

Returns the firmware type of the FLX card as a (short) string.

5.7.2.4 firmware_string()

```
std::string FlxCard::firmware_string ()  [inline]
```

Returns a descriptive name of the firmware type and version running on the FLX card.

5.7.2.5 lpgbt_type()

```
bool FlxCard::lpgbt_type ()  [inline]
```

Returns whether the firmware type is an lpGBT-type firmware.

5.7.2.6 lpgbt_fec12()

```
bool FlxCard::lpgbt_fec12 ()  [inline]
```

Returns whether the lpGBT-type firmware uses FEC12 (otherwise FEC5)

5.7.2.7 lpgbt_5gbps()

```
bool FlxCard::lpgbt_5gbps ()  [inline]
```

Returns whether the lpGBT-type firmware has lpGBT at 5Gbps (default 10Gbps)

5.7.2.8 fullmode_type()

```
bool FlxCard::fullmode_type ()  [inline]
```

Returns whether the firmware type is a FULLMODE-type firmware.

5.7.2.9 soft_reset()

```
void FlxCard::soft_reset ()
```

Triggers a reset by the SOFT_RESET register in BAR0.

5.7.2.10 cr_fromhost_reset()

```
void FlxCard::cr_fromhost_reset ()
```

Triggers a reset by the CRFROMHOST_RESET register in BAR2.

5.7.2.11 number_of_channels()

```
u_int FlxCard::number_of_channels ()  [inline]
```

Returns

Number of physical links connected to the PCIe endpoint.

5.7.2.12 ec_elink_indices()

```
std::pair< int, int > FlxCard::ec_elink_indices ()
```

Returns

Indices of the e-link number for EC ToHost (receive) and FromHost (send).

5.7.2.13 ic_elink_indices()

```
std::pair< int, int > FlxCard::ic_elink_indices ()
```

Returns

Indices of the e-link number for IC ToHost (receive) and FromHost (send).

5.8 Monitoring

Functions

- fpga_monitoring_t **FlxCard::fpga_monitoring** ()
- ltc_monitoring_t **FlxCard::power_monitoring_ltc2991** ()
- ina_monitoring_t **FlxCard::power_monitoring_ina226** (int *ina_count=0)
- tmp_monitoring_t **FlxCard::temperature_monitoring_tmp435** (int *tmp_count=0)
- ltm_monitoring_t **FlxCard::power_monitoring_ltm4700** ()
- adm_monitoring_t **FlxCard::powersequencer_monitoring_adm1x66** (int *adm_count=0)
- minipod_monitoring_t **FlxCard::minipod_monitoring** (u_int selection, int *number_of_pods=0)
- firefly_monitoring_t **FlxCard::firefly_monitoring** (int *tx_count=0, int *rx_count=0, int *tr_count=0)
- monitoring_data_t FlxCard::get_monitoring_data (u_int mon_mask)
- float FlxCard::fpga_temperature ()
- int FlxCard::fpga_fanspeed ()
- std::vector< int > FlxCard::minipods_temperature ()
- std::vector< int > FlxCard::minipods_optical_power ()
- bool FlxCard::firefly_detect (const std::string &device_name, std::string &vendor_part, std::string &vendor_sn)

5.8.1 Detailed Description

A set of functions for FLX card monitoring

5.8.2 Function Documentation

5.8.2.1 get_monitoring_data()

```
monitoring_data_t FlxCard::get_monitoring_data (
            u_int mon_mask)
```

Retrieves monitoring data from various devices on the FLX card (FPGA, MiniPODs, power monitoring ICs, temperature ICs, FireFly devices), like status, temperatures, currents, voltages, power readings. The following bits are defined in 'device_mask':

- FPGA_MONITOR Retrieve information about the FPGA
- POD_MONITOR_ALL Retrieve various sets of information from the (upto 8) MiniPODs
- POD_MONITOR_LOS Retrieve Loss-of-Signal information from the MiniPODs
- POD_MONITOR_TEMP_VOLTS Retrieve temperature and voltage readings from the MiniPODs
- POD_MONITOR_POWER Retrieve optical link power readings from the MiniPODs
- POWER_MONITOR Retrieve information from the LTC2991 or INA226 power monitoring devices
- FIREFLY_MONITOR Retrieve information from the FireFly devices The method returns a structure of type `monitoring_data_t` which is filled in accordance with the given 'monitor mask' bits (parts of the structure may remain empty). See the data structures defined in FlxCard.h for the organisation of the data and names of the parameters. See application flx-info.cpp for an example of decoding the structure. Note: It is likely that additional parameters and readout options will be added in the future.

Errors In case of an error, an exception is thrown.

Parameters

mon_mask	A mask of bits to select the parts to be monitored.

Returns

The method returns a structure of type monitoring_data_t.

5.8.2.2 fpga_temperature()

```
float FlxCard::fpga_temperature ()
```

Returns

The temperature of the FPGA (in degrees celcius).

5.8.2.3 fpga_fanspeed()

```
int FlxCard::fpga_fanspeed ()
```

Returns

The speed of the fan installed on the FPGA (in RPM), -1 returned in case of an invalid reading.

5.8.2.4 minipods_temperature()

```
std::vector< int > FlxCard::minipods_temperature ()
```

Returns

The temperatures of the MiniPODs (in degrees celcius), -1 returned for missing devices. MiniPOD order: TX1, RX1, TX2, RX2, etc.

5.8.2.5 minipods_optical_power()

```
std::vector< int > FlxCard::minipods_optical_power ()
```

Returns

The optical power readings of the MiniPODs (in microWatts), 0 returned for unconnected channels and missing devices. MiniPOD order: 12 channels TX1, 12 channels RX1, 12 channels TX2, 12 channels RX2, etc.

5.8.2.6 firefly_detect()

```
bool FlxCard::firefly_detect (
            const std::string & device_name,
            std::string & vendor_part,
            std::string & vendor_sn)
```

Determines if the FireFly identified by a name is accessible via I2C.

Returns

Whether the FireFly device with the given name is accessible, and if true, device part number and serial number are returned as strings in vendor_part and vendor_sn respectively.

5.9 Configuration

Functions

- void FlxCard::configure (std::string filename, bool do_links=true, bool do_registers=true)
- void FlxCard::readConfiguration (std::string filename)
- void FlxCard::readConfiguration ()
- u_int FlxCard::number_of_elinks_tohost (int dma_index=-1)
- u_int FlxCard::number_of_elinks_toflx ()
- u_int **FlxCard::number_of_elinks_fromhost** ()
- std::vector< flxcard::elink_descr_t > FlxCard::elinks_tohost (int dma_index=-1)
- std::vector< flxcard::elink_descr_t > FlxCard::elinks_toflx ()
- std::vector< flxcard::elink_descr_t > **FlxCard::elinks_fromhost** ()
- std::vector< flxcard::elink_descr_t > **FlxCard::elinks_toflx_broadcast** ()
- std::vector< flxcard::elink_descr_t > **FlxCard::elinks_fromhost_broadcast** ()
- u_int FlxCard::number_of_dma_tohost ()
- bool FlxCard::uses_dma_index_mask ()
- void **FlxCard::set_hdlc_instant_timeout** (bool b)
- bool **FlxCard::hdlc_instant_timeout** ()
- unsigned int FlxCard::link_mode ()
- bool FlxCard::is_elink_enabled (u_int channel, u_int egroup, u_int epath, bool is_to_flx)
- unsigned int FlxCard::tohost_elink_dmaid (u_int channel, u_int egroup, u_int epath)
- unsigned int FlxCard::tohost_elink_dmaid (uint16_t elinknr)
- bool FlxCard::has_tohost_elink_streams (u_int channel, u_int egroup, u_int epath)
- uint8_t FlxCard::detector_id ()
- uint16_t FlxCard::connector_id (uint16_t channel)

5.9.1 Detailed Description

Functions pertaining to configuration of an FLX device and its (e-)links

5.9.2 Function Documentation

5.9.2.1 configure()

```
void FlxCard::configure (
            std::string filename,
            bool do_links = true,
            bool do_registers = true)
```

Reads a FELIX device (e-)link configuration -plus additional register settings, if any- from a .yelc configuration file, stores it in a member structure, and writes the configuration to this FELIX device (or optionally, not, and/or optionally skipping any register settings present in the file).

5.9.2.2 readConfiguration() [1/2]

```
void FlxCard::readConfiguration (
            std::string filename)
```

Reads a FELIX device (e-)link configuration -plus additional register settings, if any- from a .yelc file, and stores it in a member structure.

5.9.2.3 readConfiguration() [2/2]

```
void FlxCard::readConfiguration ()
```

Reads a FELIX device (e-)link configuration from this FELIX device and stores it in a member structure. Leaves any stored additional register settings untouched.

5.9.2.4 number_of_elinks_tohost()

```
u_int FlxCard::number_of_elinks_tohost (
            int dma_index = -1)
```

Returns the number of enabled ToHost e-link numbers of this FELIX device, optionally only those assigned to the given DMA index.

5.9.2.5 number_of_elinks_toflx()

```
u_int FlxCard::number_of_elinks_toflx ()
```

Returns the number of enabled FromHost e-link numbers of this FELIX device, including TTC-type e-links.

5.9.2.6 elinks_tohost()

```
std::vector< flxcard::elink_descr_t > FlxCard::elinks_tohost (
            int dma_index = -1)
```

Returns a list of the enabled ToHost e-link numbers of this FELIX device, optionally only those assigned to the given DMA index.

5.9.2.7 elinks_toflx()

```
std::vector< flxcard::elink_descr_t > FlxCard::elinks_toflx ()
```

Returns a list of the enabled FromHost e-link numbers of this FELIX device, not including TTC-type e-links.

5.9.2.8 number_of_dma_tohost()

```
u_int FlxCard::number_of_dma_tohost ()  [inline]
```

Returns the number of ToHost DMA descriptors of this FELIX device.

5.9.2.9 uses_dma_index_mask()

```
bool FlxCard::uses_dma_index_mask ()  [inline]
```

Returns true if this FELIX device uses a DMA index bitmask for its e-links, rather than a single DMA index number.

5.9.2.10 link_mode()

```
unsigned int FlxCard::link_mode ()
```

Returns the link mode, defined in LinkConfig.h

5.9.2.11 is_elink_enabled()

```
bool FlxCard::is_elink_enabled (
            u_int channel,
            u_int egroup,
            u_int epath,
            bool is_to_flx)
```

Returns true if the e-link is enabled.

5.9.2.12 tohost_elink_dmaid() [1/2]

```
unsigned int FlxCard::tohost_elink_dmaid (
            u_int channel,
            u_int egroup,
            u_int epath)
```

Returns the DMA id assigned to the given e-link

5.9.2.13 tohost_elink_dmaid() [2/2]

```
unsigned int FlxCard::tohost_elink_dmaid (
            uint16_t elinknr)
```

Returns the DMA id assigned to the given e-link

5.9.2.14 has_tohost_elink_streams()

```
bool FlxCard::has_tohost_elink_streams (
            u_int channel,
            u_int egroup,
            u_int epath)
```

Returns true if the e-link is enabled.

5.9.2.15 detector_id()

```
uint8_t FlxCard::detector_id ()
```

Returns

the detector id set up in the register map.

Returns -1 if the VALID bit is not set.

5.9.2.16 connector_id()

```
uint16_t FlxCard::connector_id (
            uint16_t channel)
```

Returns

the connector id set up in the register map.

Returns the connector number from 0 to 3 if the register is not set

Parameters

channel	the channel number to resolve the connector ID in case of an FLX-155 card with more than 24 links.

5.10 Public member variables

Variables

- volatile flxcard_bar0_regs_t * **FlxCard::m_bar0**
- volatile flxcard_bar1_regs_t * **FlxCard::m_bar1**
- volatile flxcard_bar2_regs_t * **FlxCard::m_bar2**
- i2c_device_t const * **FlxCard::m_i2cDevices**

5.10.1 Detailed Description

Public member variables of class FlxCard.

Chapter 6

Class Documentation

6.1 adm1066_parameters_t Struct Reference

Public Attributes

- std::string **name**
- bool **valid**
- uint32_t **manufacturer_id**
- uint32_t **silicon_rev**
- uint8_t **input_cfg** [10][8]
- uint32_t **fault_stat** [6]
- uint32_t **adc** [12]
- float **adc_fullscale** [12]
- std::string **adc_name** [12]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.2 adm1266_parameters_t Struct Reference

Public Attributes

- std::string **name**
- bool **valid**
- uint32_t **ic_device_id**
- uint64_t **ic_device_rev**
- uint32_t **vout** [17]
- uint32_t **vout_mode** [17]
- uint32_t **status_vout** [17]
- std::string **vout_name** [17]
- uint32_t **vout_uv_warn** [17]
- uint32_t **vout_uv_fault** [17]
- uint32_t **vout_ov_warn** [17]
- uint32_t **vout_ov_fault** [17]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.3 adm_monitoring_t Struct Reference

Public Attributes

- adm1066_parameters_t **adm1066** [2]
- adm1266_parameters_t **adm1266**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.4 board_id_t Struct Reference

Public Attributes

- volatile u_long **date_time**: 40
- volatile u_long **reserved**: 24
- volatile u_long **revision**: 16

The documentation for this struct was generated from the following file:

- FlxCard.h

6.5 device_list_t Struct Reference

Public Attributes

- u_int **n_devices**
- u_int **cdmap** [MAXCARDS][2]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.6 dma_descriptor_t Struct Reference

Public Attributes

- volatile u_long **start_address**
- volatile u_long **end_address**
- volatile u_long **tlp**:11
- volatile u_long **fromhost**: 1
- volatile u_long **wrap_around**: 1
- volatile u_long **reserved**:51
- volatile u_long **sw_pointer**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.7 dma_status_t Struct Reference

Public Attributes

- volatile u_long **fw_pointer**
- volatile u_long **descriptor_done**: 1
- volatile u_long **even_addr_dma**: 1
- volatile u_long **even_addr_pc**: 1

The documentation for this struct was generated from the following file:

- FlxCard.h

6.8 ffly_rx_parameters_t Struct Reference

Public Attributes

- std::string **name**
- bool **absent**
- uint32_t **status**
- uint32_t **status_summ**
- uint32_t **latched_alarms_rx_los**
- uint32_t **latched_alarms_rx_power**
- uint32_t **latched_alarms_temp**
- uint32_t **latched_alarms_vcc33**
- uint32_t **latched_alarms_cdr_lol**
- int32_t **case_temp**
- uint32_t **vcc**
- uint32_t **elapsed_optime**
- uint32_t **chan_disable**
- uint32_t **output_disable**
- uint32_t **polarity_invert**
- uint64_t **output_swing**
- uint64_t **output_preemphasis**
- uint32_t **cdr_enable**
- uint32_t **firmware_version**
- std::string **vendor_part**
- std::string **vendor_serial**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.9 ffly_tr_parameters_t Struct Reference

Public Attributes

- std::string **name**
- bool **absent**
- uint32_t **status**
- uint32_t **latched_alarms_tx_los**
- uint32_t **latched_alarms_tx_fault**
- uint32_t **latched_alarms_cdr_lol**
- uint32_t **latched_alarms_temp**
- uint32_t **latched_alarms_vcc**
- uint32_t **latched_alarms_rx_power**
- uint32_t **elapsed_optime**
- int32_t **case_temp**
- uint32_t **vcc_3v3**
- uint32_t **vcc_1v8**
- uint32_t **rx_optical_power** [4]
- uint32_t **firmware_revision**
- uint32_t **tx_chan_disable**
- uint32_t **cdr_enable**
- uint32_t **cdr_rate_select**
- uint32_t **mask_los_alarms**
- uint32_t **mask_tx_fault_flags**
- uint32_t **mask_cdr_lol_alarms**
- uint32_t **mask_temp_alarms**
- uint32_t **mask_vcc_alarms**
- std::string **vendor_part**
- std::string **vendor_serial**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.10 ffly_tx_parameters_t Struct Reference

Public Attributes

- std::string **name**
- bool **absent**
- uint32_t **status**
- uint32_t **status_summ**
- uint32_t **latched_alarms_tx_los**
- uint32_t **latched_alarms_tx_fault**
- uint32_t **latched_alarms_temp**
- uint32_t **latched_alarms_vcc33**
- uint32_t **latched_alarms_cdr_lol**
- int32_t **case_temp**
- uint32_t **vcc**
- uint32_t **elapsed_optime**
- uint32_t **chan_disable**

- uint32_t **squelch_disable**
- uint32_t **polarity_invert**
- uint64_t **input_equalization**
- uint32_t **cdr_enable**
- uint32_t **mask_tx_los_alarms**
- uint32_t **mask_fault_flags**
- uint32_t **mask_temp_alarms**
- uint32_t **mask_vcc33_alarms**
- uint32_t **mask_cdr_lol_alarms**
- uint32_t **firmware_version**
- std::string **vendor_part**
- std::string **vendor_serial**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.11 firefly_monitoring_t Struct Reference

Public Attributes

- ffly_tx_parameters_t **ffly_tx** [2+2]
- ffly_rx_parameters_t **ffly_rx** [2+2]
- ffly_tr_parameters_t **ffly_tr** [1+1]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.12 FlxCard Class Reference

Public Member Functions

- **FlxCard** ()

  The constructor of an FlxCard object only initializes a few class variables. It does not interact with the FELIX H/W.
- void card_open (int device_nr, u_int lock_mask, bool read_config=false, bool ignore_version=false)
- void card_close ()
- u_int lock_mask (int device_nr)
- u_int **get_lock_mask** (int device_nr)
- u_long openBackDoor (int bar)
- u_long bar0Address ()
- u_long bar1Address ()
- u_long bar2Address ()
- int dma_max_tlp_bytes ()
- void dma_to_host (u_int dma_id, u_long dst, size_t size, u_int flags)
- void dma_from_host (u_int dma_id, u_long src, size_t size, u_int flags)
- bool dma_enabled (u_int dma_id)
- void dma_wait (u_int dma_id)

- void dma_stop (u_int dma_id)
- void dma_advance_ptr (u_int dma_id, u_long dst, size_t size, size_t bytes)
- void dma_set_ptr (u_int dma_id, u_long dst)
- u_long dma_get_ptr (u_int dma_id)
- u_long **dma_get_read_ptr** (u_int dma_id)
- void dma_reset ()
- u_long dma_get_fw_ptr (u_int dma_id)
- u_long **dma_get_current_address** (u_int dma_id)
- bool dma_cmp_even_bits (u_int dma_id)
- void i2c_read (const char *device_str, u_char reg_addr, u_char *value, int nbytes=1)
- void i2c_write (const char *device_str, u_char reg_addr, u_char data)
- void **i2c_write** (const char *device_str, u_char reg_addr, u_char *data, int nbytes)
- void i2c_write_byte (u_char dev_addr, u_char byte)
- void i2c_write_byte (u_char dev_addr, u_char reg_addr, u_char byte)
- u_char i2c_read_byte (u_char dev_addr, u_char reg_addr)
- bool i2c_flush (u_int *count=0)
- void i2c_write_bytes (uint8_t dev_addr, uint8_t reg_addr, int nbytes, uint8_t *bytes)
- void **i2c_read_bytes** (uint8_t dev_addr, uint8_t reg_addr, int nbytes, uint8_t *bytes)
- u_int gbt_setup (int alignment, int channel_mode)
- long int gbt_version_delay (u_long svn_version, char *filename)
- u_long rxusrclk_freq (u_int channel)
- void gth_rx_reset (int quad=-1)
- void irq_enable (u_int interrupt=ALL_IRQS)
- void irq_disable (u_int interrupt=ALL_IRQS)
- void irq_wait (u_int interrupt)
- void irq_clear (u_int interrupt=ALL_IRQS)
- void irq_cancel (u_int interrupt=ALL_IRQS)
- void irq_reset_counters (u_int interrupt=ALL_IRQS)
- u_long cfg_get_option (const char *key, bool show_options=false)
- void cfg_set_option (const char *key, u_long value, bool show_options=false)
- u_long cfg_get_reg (const char *key)
- void cfg_set_reg (const char *key, u_long value)
- void registers_reset ()
- u_int card_model ()
- u_int **card_type** ()
- u_int firmware_type ()
- std::string firmware_type_string ()
- std::string firmware_string ()
- bool lpgbt_type ()
- bool lpgbt_fec12 ()
- bool lpgbt_5gbps ()
- bool fullmode_type ()
- void soft_reset ()
- void cr_fromhost_reset ()
- u_int number_of_channels ()
- std::pair< int, int > ec_elink_indices ()
- std::pair< int, int > ic_elink_indices ()
- fpga_monitoring_t **fpga_monitoring** ()
- ltc_monitoring_t **power_monitoring_ltc2991** ()
- ina_monitoring_t **power_monitoring_ina226** (int *ina_count=0)
- tmp_monitoring_t **temperature_monitoring_tmp435** (int *tmp_count=0)
- ltm_monitoring_t **power_monitoring_ltm4700** ()
- adm_monitoring_t **powersequencer_monitoring_adm1x66** (int *adm_count=0)
- minipod_monitoring_t **minipod_monitoring** (u_int selection, int *number_of_pods=0)
- firefly_monitoring_t **firefly_monitoring** (int *tx_count=0, int *rx_count=0, int *tr_count=0)

- monitoring_data_t get_monitoring_data (u_int mon_mask)
- float fpga_temperature ()
- int fpga_fanspeed ()
- std::vector< int > minipods_temperature ()
- std::vector< int > minipods_optical_power ()
- bool firefly_detect (const std::string &device_name, std::string &vendor_part, std::string &vendor_sn)
- void configure (std::string filename, bool do_links=true, bool do_registers=true)
- void readConfiguration (std::string filename)
- void readConfiguration ()
- u_int number_of_elinks_tohost (int dma_index=-1)
- u_int number_of_elinks_toflx ()
- u_int **number_of_elinks_fromhost** ()
- std::vector< flxcard::elink_descr_t > elinks_tohost (int dma_index=-1)
- std::vector< flxcard::elink_descr_t > elinks_toflx ()
- std::vector< flxcard::elink_descr_t > **elinks_fromhost** ()
- std::vector< flxcard::elink_descr_t > **elinks_toflx_broadcast** ()
- std::vector< flxcard::elink_descr_t > **elinks_fromhost_broadcast** ()
- u_int number_of_dma_tohost ()
- bool uses_dma_index_mask ()
- void **set_hdlc_instant_timeout** (bool b)
- bool **hdlc_instant_timeout** ()
- unsigned int link_mode ()
- bool is_elink_enabled (u_int channel, u_int egroup, u_int epath, bool is_to_flx)
- unsigned int tohost_elink_dmaid (u_int channel, u_int egroup, u_int epath)
- unsigned int tohost_elink_dmaid (uint16_t elinknr)
- bool has_tohost_elink_streams (u_int channel, u_int egroup, u_int epath)
- uint8_t detector_id ()
- uint16_t connector_id (uint16_t channel)

Static Public Member Functions

- static u_int number_of_cards ()
- static u_int number_of_devices ()
- static device_list_t device_list ()
- static int card_to_device_number (int card_number)
- static regmap_register_t * cfg_register (const char *name)
- static regmap_bitfield_t * cfg_bitfield (const char *name)
- static std::string cfg_bitfield_options (const char *name, bool include_all_substr=true)

Public Attributes

- volatile flxcard_bar0_regs_t * **m_bar0**
- volatile flxcard_bar1_regs_t * **m_bar1**
- volatile flxcard_bar2_regs_t * **m_bar2**
- i2c_device_t const * **m_i2cDevices**

The documentation for this class was generated from the following file:

- FlxCard.h

6.13 flxcard_bar0_regs_t Struct Reference

Public Attributes

- dma_descriptor_t **DMA_DESC** [8]
- u_char **unused1** [256]
- dma_status_t **DMA_DESC_STATUS** [8]
- u_char **unused2** [128]
- volatile u_int **BAR0_VALUE**
- u_char **unused3** [12]
- volatile u_int **BAR1_VALUE**
- u_char **unused4** [12]
- volatile u_int **BAR2_VALUE**
- u_char **unused5** [220]
- volatile u_int **DMA_DESC_ENABLE**
- u_char **unused7** [28]
- volatile u_int **DMA_RESET**
- u_char **unused8** [12]
- volatile u_int **SOFT_RESET**
- u_char **unused9** [12]
- volatile u_int **REGISTERS_RESET**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.14 flxcard_bar1_regs_t Struct Reference

Public Attributes

- int_vec_t **INT_VEC** [8]
- u_char **unused1** [128]
- volatile u_int **INT_TAB_ENABLE**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.15 fpga_parameters_t Struct Reference

Public Attributes

- float **temperature**
- float **vccint**
- float **vccaux**
- float **vccbram**
- u_long **fanspeed**
- u_long **dna**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.16 ina226_parameters_t Struct Reference

Public Attributes

- std::string **name**
- uint32_t **config_reg**
- uint32_t **shunt_volt_reg**
- uint32_t **bus_volt_reg**
- uint32_t **power_reg**
- uint32_t **current_reg**
- uint32_t **calib_reg**
- uint32_t **manufacturer_id**
- float **r_shunt**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.17 ina_monitoring_t Struct Reference

Public Attributes

- ina226_parameters_t **ina226** [10+1]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.18 int_vec_t Struct Reference

Public Attributes

- volatile u_int **control**
- volatile u_int **data**
- volatile u_long **address**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.19 ltc2991_1_parameters_t Struct Reference

Public Attributes

- std::string **name**
- float **vccint_current**
- float **vccint_voltage**
- float **mgtavcc_current**
- float **mgtavcc_voltage**
- float **fpga_internal_diode_temperature**
- float **mgtavtt_current**
- float **mgtavtt_voltage**
- float **mgtavttc_voltage**
- float **mgtvccaux_voltage**
- float **internal_temperature**
- float **vcc**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.20 ltc2991_2_parameters_t Struct Reference

Public Attributes

- std::string **name**
- float **pex0p9v_current**
- float **pex0p9v_voltage**
- float **sys18_current**
- float **sys18_voltage**
- float **sys12_voltage**
- float **sys25_current**
- float **sys25_voltage**
- float **pex8732_internal_diode_temperature**
- float **internal_temperature**
- float **vcc**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.21 ltc_monitoring_t Struct Reference

Public Attributes

- ltc2991_1_parameters_t **ltc2991_1**
- ltc2991_2_parameters_t **ltc2991_2**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.22 ltm4700_parameters_t Struct Reference

Public Attributes

- std::string **name**
- uint32_t **mfr_special_id**
- uint32_t **vin**
- uint32_t **iin**
- uint32_t **pin**
- uint32_t **vout** [2]
- uint32_t **iout** [2]
- uint32_t **pout** [2]
- uint32_t **temp_die**
- uint32_t **temp_ext** [2]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.23 minipod_monitoring_t Struct Reference

Public Attributes

- minipod_parameters_t **minipod** [8]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.24 minipod_parameters_t Struct Reference

Public Attributes

- bool **absent**
- char **name** [10]
- int **temp**
- float **v33**
- float **v25**
- int **los**
- char **vname** [16]
- char **voui** [3]
- char **vpnum** [16]
- char **vrev** [2]
- char **vsernum** [16]
- char **vdate** [8]
- int **optical_power_uw** [12]

The documentation for this struct was generated from the following file:

- FlxCard.h

6.25 monitoring_data_t Struct Reference

Public Attributes

- fpga_monitoring_t **fpga**
- ltc_monitoring_t **ltc**
- ina_monitoring_t **ina**
- ltm_monitoring_t **ltm**
- tmp_monitoring_t **tmp**
- adm_monitoring_t **adm**
- minipod_monitoring_t **pod**
- firefly_monitoring_t **firefly**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.26 tmp435_parameters_t Struct Reference

Public Attributes

- std::string **name**
- uint32_t **t_local_hi**
- uint32_t **t_local_lo**
- uint32_t **t_remote_hi**
- uint32_t **t_remote_lo**
- uint32_t **status**
- uint32_t **device_id**
- uint32_t **manufacturer_id**

The documentation for this struct was generated from the following file:

- FlxCard.h

6.27 tmp_monitoring_t Struct Reference

Public Attributes

- tmp435_parameters_t **tmp435** [7+1]

The documentation for this struct was generated from the following file:

- FlxCard.h

Chapter 7

File Documentation

7.1 FlxCard.h

```
00001 /*****************************************************************
00002  * \mainpage                                                      *
00003  *                                                               *
00004  * @author: Markus Joos, CERN                                    *
00005  *  Maintainer: Henk Boterenbrood, Nikhef                        *
00006  *                                                               *
00007  * @brief                                                        *
00008  * This note defines an application program interface (API) for  *
00009  * the use of the FLX PCIe I/O card in the ATLAS read-out system. *
00010  * The intention of the API is to satisfy the needs of simple    *
00011  * test programs as well as the requirements of the              *
00012  * FelixApplication.                                             *
00013  *                                                               *
00014  * @copyright CERN, Nikhef                                       *
00015  *****************************************************************/
00016
00017 #ifndef FLXCARD_H
00018 #define FLXCARD_H
00019
00020 #include <cstdint>
00021 #include <vector>
00022 #include <string>
00023 #include <sys/types.h>
00024 #include <utility> // std::pair
00025
00026 #include "cmem_rcc/cmem_rcc.h"
00027 #include "flxcard/flx_common.h"
00028 #include "regmap/regmap.h"
00029 #include "flxcard/LinkConfig.h"
00030
00031 // Constants
00032 #define WAIT_TIME_600                    600
00033 #define WAIT_TIME_200                    200
00034 #define NUM_INTERRUPTS                   8
00035 #define FLX_GBT_FILE_NOT_FOUND           -1
00036 #define FLX_GBT_VERSION_NOT_FOUND        -2
00037 #define FLX_GBT_TMODE_FEC                0
00038 #define FLX_GBT_ALIGNMENT_CONTINUOUS     0
00039 #define FLX_DMA_WRAPAROUND               1
00040 #define FLX_GBT_ALIGNMENT_ONE            1
00041 #define FLX_GBT_CHANNEL_AUTO             1
00042 #define FLX_GBT_TMODE_WideBus            1
00043
00044 // Firmware types
00045 #define FIRMW_GBT                        0
00046 #define FIRMW_FULL                       1
00047 #define FIRMW_LTDB                       2
00048 #define FIRMW_FEI4                       3
00049 #define FIRMW_PIXEL                      4
00050 #define FIRMW_STRIP                      5
00051 #define FIRMW_FELIG_GBT                  6
00052 #define FIRMW_FELIG                      6 // Backwards compatible
00053 #define FIRMW_FMEMU                      7
00054 #define FIRMW_MROD                       8
00055 #define FIRMW_LPGBT                      9
00056 #define FIRMW_INTERLAKEN                 10
00057 #define FIRMW_FELIG_LPGBT                11
00058 #define FIRMW_HGTD_LUMI                  12
```

```
00059 #define FIRMW_BCM_PRIME                  13
00060 #define FIRMW_FELIG_PIXEL                14
00061 #define FIRMW_FELIG_STRIP                15
00062 #define FIRMW_WUPPER                     16
00063 #define FIRMW_TPX4                       17
00064 const std::vector<std::string> FIRMW_STR{ "GBT",   "FULL",  "LTDB",   "FEI4", "PIXEL",
00065                                           "STRIP", "FELIG", "FM-EMU", "MROD", "LPGBT",
00066                                           "INTERLAKEN", "FELIG-LPGBT", "HGTD-LUMI",
00067                                           "BCM-PRIME",  "FELIG-PIXEL", "FELIG-STRIP",
00068                                           "WUPPER", "TPX4" };
00069
00070 // Interrupts
00071 #define ALL_IRQS                         0xFFFFFFFF
00072
00073 // Monitoring devices
00074 #define FPGA_MONITOR                     0x001
00075 #define POD_MONITOR_ALL                  0x002
00076 #define POWER_MONITOR                    0x004
00077 #define FIREFLY_MONITOR                  0x008
00078 #define POD_MONITOR_LOS                  0x100
00079 #define POD_MONITOR_TEMP_VOLT            0x200
00080 #define POD_MONITOR_POWER                0x400
00081 #define POD_MONITOR_POWER_RX             0x800
00082 // (Backwards compatability)
00083 #define POD_MONITOR                      POD_MONITOR_ALL
00084
00085 // Resource locking
00086 #define LOCK_NONE                        0x0000
00087 #define LOCK_DMA0                        0x0001
00088 #define LOCK_DMA1                        0x0002
00089 #define LOCK_I2C                         0x0004
00090 #define LOCK_FLASH                       0x0008
00091 #define LOCK_ELINK                       0x0010
00092 #define LOCK_READ_CONFIG                 0x0020
00093 #define LOCK_ALL                         0xFFFFFFFF
00094 #define LOCK_DMA(n)                      (1<<(16+n))
00095
00096 // Other constants
00097 #define ALL_BITS                         0xFFFFFFFFFFFFFFFF
00098
00099 // Register model
00100 typedef struct
00101 {
00102   volatile u_long start_address;        /*  low half, bits  63:00 */
00103   volatile u_long end_address;          /*  low half, bits 127:64 */
00104   volatile u_long tlp         :11;      /* high half, bits  10:00 */
00105   volatile u_long fromhost    : 1;      /* high half, bit      11 */
00106   volatile u_long wrap_around : 1;      /* high half, bit      12 */
00107   volatile u_long reserved    :51;      /* high half, bits  63:13 */
00108   volatile u_long sw_pointer;           /* high half, bits 127:64 */
00109 } dma_descriptor_t;
00110
00111 typedef struct
00112 {
00113   volatile u_long fw_pointer;           /* bits  63:00 */
00114   volatile u_long descriptor_done : 1;  /* bit      64 */
00115   volatile u_long even_addr_dma   : 1;  /* bit      65 */
00116   volatile u_long even_addr_pc    : 1;  /* bit      66 */
00117 } dma_status_t;
00118
00119 typedef struct
00120 {
00121   volatile u_int  control;              /* bits  63:00 */
00122   volatile u_int  data;                 /* bits  95:64 */
00123   volatile u_long address;              /* bits 127:96 */
00124 } int_vec_t;
00125
00126 typedef struct
00127 {
00128   volatile u_long date_time : 40;       /* bits  39:00 */
00129   volatile u_long reserved  : 24;       /* bits  63:40 */
00130   volatile u_long revision  : 16;       /* bits  79:64 */
00131 } board_id_t;
00132
00133 typedef struct
00134 {
00135   dma_descriptor_t DMA_DESC[8];         /* 0x000 - 0x0ff */
00136   u_char           unused1[256];        /* 0x100 - 0x1ff */
00137   dma_status_t     DMA_DESC_STATUS[8];  /* 0x200 - 0x27f */
00138   u_char           unused2[128];        /* 0x280 - 0x2ff */
00139   volatile u_int   BAR0_VALUE;          /* 0x300 - 0x303 */
00140   u_char           unused3[12];         /* 0x304 - 0x30f */
00141   volatile u_int   BAR1_VALUE;          /* 0x310 - 0x313 */
00142   u_char           unused4[12];         /* 0x314 - 0x31f */
00143   volatile u_int   BAR2_VALUE;          /* 0x320 - 0x323 */
00144   u_char           unused5[220];        /* 0x324 - 0x3ff */
00145   volatile u_int   DMA_DESC_ENABLE;     /* 0x400 - 0x403 */
```

```
00146   u_char           unused7[28];         /* 0x404 - 0x41f */
00147   volatile u_int   DMA_RESET;           /* 0x420 - 0x423 */
00148   u_char           unused8[12];         /* 0x424 - 0x42f */
00149   volatile u_int   SOFT_RESET;          /* 0x430 - 0x433 */
00150   u_char           unused9[12];         /* 0x434 - 0x43f */
00151   volatile u_int   REGISTERS_RESET;     /* 0x440 - 0x443 */
00152 } flxcard_bar0_regs_t;
00153
00154 typedef struct
00155 {
00156   int_vec_t        INT_VEC[8];          /* 0x000 - 0x07f */
00157   u_char           unused1[128];        /* 0x080 - 0x0ff */
00158   volatile u_int   INT_TAB_ENABLE;      /* 0x100 - 0x103 */
00159 } flxcard_bar1_regs_t;
00160
00161 typedef struct
00162 {
00163   u_int n_devices;
00164   u_int cdmap[MAXCARDS][2];
00165 } device_list_t;
00166
00167 // Forward declaration (see I2CDevices.h)
00168 typedef struct i2c_device i2c_device_t;
00169
00170 // Device parameters typedefs
00171
00172 typedef struct
00173 {
00174   float  temperature;
00175   float  vccint;
00176   float  vccaux;
00177   float  vccbram;
00178   u_long fanspeed;
00179   u_long dna;
00180 } fpga_parameters_t;
00181
00182 typedef struct
00183 {
00184   bool  absent;
00185   char  name[10];
00186   int   temp;
00187   float v33;
00188   float v25;
00189   int   los;
00190   char  vname[16];
00191   char  voui[3];
00192   char  vpnum[16];
00193   char  vrev[2];
00194   char  vsernum[16];
00195   char  vdate[8];
00196   int   optical_power_uw[12];
00197 } minipod_parameters_t;
00198
00199 typedef struct
00200 {
00201   std::string name;
00202   float vccint_current;
00203   float vccint_voltage;
00204   float mgtavcc_current;
00205   float mgtavcc_voltage;
00206   float fpga_internal_diode_temperature;
00207   float mgtavtt_current;          // Only FLX-712
00208   float mgtavtt_voltage;          // Only FLX-712
00209   float mgtavttc_voltage;         // Only FLX-711
00210   float mgtvccaux_voltage;        // Only FLX-711
00211   float internal_temperature;
00212   float vcc;
00213 } ltc2991_1_parameters_t;
00214
00215 typedef struct
00216 {
00217   std::string name;
00218   float pex0p9v_current;
00219   float pex0p9v_voltage;
00220   float sys18_current;
00221   float sys18_voltage;
00222   float sys12_voltage;            // Only FLX-711
00223   float sys25_current;            // Only FLX-712
00224   float sys25_voltage;
00225   float pex8732_internal_diode_temperature;
00226   float internal_temperature;
00227   float vcc;
00228 } ltc2991_2_parameters_t;
00229
00230 typedef struct
00231 {
00232   std::string name;
```

```
00233   uint32_t t_local_hi;  // In degrees C
00234   uint32_t t_local_lo;  // In 1/10000ths C
00235   uint32_t t_remote_hi; // In degrees C
00236   uint32_t t_remote_lo; // In 1/10000ths C
00237   uint32_t status;
00238   uint32_t device_id;
00239   uint32_t manufacturer_id;
00240 } tmp435_parameters_t;
00241
00242 typedef struct
00243 {
00244   std::string name;
00245   uint32_t    config_reg;
00246   uint32_t    shunt_volt_reg;
00247   uint32_t    bus_volt_reg;
00248   uint32_t    power_reg;
00249   uint32_t    current_reg;
00250   uint32_t    calib_reg;
00251   uint32_t    manufacturer_id;
00252   float       r_shunt;
00253 } ina226_parameters_t;
00254
00255 typedef struct
00256 {
00257   std::string name;
00258   uint32_t    mfr_special_id;
00259   uint32_t    vin;
00260   uint32_t    iin;
00261   uint32_t    pin;
00262   uint32_t    vout[2];
00263   uint32_t    iout[2];
00264   uint32_t    pout[2];
00265   uint32_t    temp_die;
00266   uint32_t    temp_ext[2];
00267 } ltm4700_parameters_t;
00268
00269 typedef struct
00270 {
00271   std::string name;
00272   bool        valid;
00273   uint32_t    manufacturer_id;
00274   uint32_t    silicon_rev;
00275   uint8_t     input_cfg[10][8];
00276   uint32_t    fault_stat[6];
00277   uint32_t    adc[12];
00278   float       adc_fullscale[12];
00279   std::string adc_name[12];
00280 } adm1066_parameters_t;
00281
00282 typedef struct
00283 {
00284   std::string name;
00285   bool        valid;
00286   uint32_t    ic_device_id;
00287   uint64_t    ic_device_rev;
00288   uint32_t    vout[17];
00289   uint32_t    vout_mode[17];
00290   uint32_t    status_vout[17];
00291   std::string vout_name[17];
00292   uint32_t    vout_uv_warn[17];
00293   uint32_t    vout_uv_fault[17];
00294   uint32_t    vout_ov_warn[17];
00295   uint32_t    vout_ov_fault[17];
00296 } adm1266_parameters_t;
00297
00298 typedef struct
00299 {
00300   std::string name;
00301   bool absent;
00302
00303   // FireFly TX lower memory map
00304   uint32_t status;
00305   uint32_t status_summ;
00306   uint32_t latched_alarms_tx_los;
00307   uint32_t latched_alarms_tx_fault;
00308   uint32_t latched_alarms_temp;
00309   uint32_t latched_alarms_vcc33;
00310   uint32_t latched_alarms_cdr_lol;
00311   int32_t  case_temp;
00312   uint32_t vcc;
00313   uint32_t elapsed_optime;
00314   uint32_t chan_disable;
00315   uint32_t squelch_disable;
00316   uint32_t polarity_invert;
00317   uint64_t input_equalization;
00318   uint32_t cdr_enable;
00319   uint32_t mask_tx_los_alarms;
```

```
00320   uint32_t mask_fault_flags;
00321   uint32_t mask_temp_alarms;
00322   uint32_t mask_vcc33_alarms;
00323   uint32_t mask_cdr_lol_alarms;
00324   uint32_t firmware_version;
00325
00326   // Upper page 0;
00327   std::string vendor_part;
00328   std::string vendor_serial;
00329 } ffly_tx_parameters_t;
00330
00331 typedef struct
00332 {
00333   std::string name;
00334   bool absent;
00335
00336   // FireFly RX lower memory map
00337   uint32_t status;
00338   uint32_t status_summ;
00339   uint32_t latched_alarms_rx_los;
00340   uint32_t latched_alarms_rx_power;
00341   uint32_t latched_alarms_temp;
00342   uint32_t latched_alarms_vcc33;
00343   uint32_t latched_alarms_cdr_lol;
00344   int32_t  case_temp;
00345   uint32_t vcc;
00346   uint32_t elapsed_optime;
00347   uint32_t chan_disable;
00348   uint32_t output_disable;
00349   uint32_t polarity_invert;
00350   uint64_t output_swing;
00351   uint64_t output_preemphasis;
00352   uint32_t cdr_enable;
00353   // ...etc...
00354   uint32_t firmware_version;
00355
00356   // Upper page 0;
00357   std::string vendor_part;
00358   std::string vendor_serial;
00359 } ffly_rx_parameters_t;
00360
00361 typedef struct
00362 {
00363   std::string name;
00364   bool absent;
00365
00366   // FireFly TX/RX lower memory map
00367   uint32_t status;
00368   uint32_t latched_alarms_tx_los;
00369   uint32_t latched_alarms_tx_fault;
00370   uint32_t latched_alarms_cdr_lol;
00371   uint32_t latched_alarms_temp;
00372   uint32_t latched_alarms_vcc;
00373   uint32_t latched_alarms_rx_power;
00374   uint32_t elapsed_optime;
00375   int32_t  case_temp;
00376   uint32_t vcc_3v3;
00377   uint32_t vcc_1v8;
00378   uint32_t rx_optical_power[4];
00379   uint32_t firmware_revision;
00380   uint32_t tx_chan_disable;
00381   uint32_t cdr_enable;
00382   uint32_t cdr_rate_select;
00383   uint32_t mask_los_alarms;
00384   uint32_t mask_tx_fault_flags;
00385   uint32_t mask_cdr_lol_alarms;
00386   uint32_t mask_temp_alarms;
00387   uint32_t mask_vcc_alarms;
00388
00389   // Upper page 0;
00390   std::string vendor_part;
00391   std::string vendor_serial;
00392 } ffly_tr_parameters_t;
00393
00394 // Device monitoring data collections typedefs
00395
00396 typedef fpga_parameters_t fpga_monitoring_t;
00397
00398 typedef struct
00399 {
00400   ltc2991_1_parameters_t  ltc2991_1;
00401   ltc2991_2_parameters_t  ltc2991_2;
00402 } ltc_monitoring_t;
00403
00404 typedef struct
00405 {
00406   ina226_parameters_t  ina226[10+1]; // +1 for FLX-155
```

```
00407 } ina_monitoring_t;
00408
00409 typedef struct
00410 {
00411   tmp435_parameters_t  tmp435[7+1];  // +1 for FLX-155
00412 } tmp_monitoring_t;
00413
00414 typedef ltm4700_parameters_t ltm_monitoring_t;
00415
00416 typedef struct
00417 {
00418   adm1066_parameters_t  adm1066[2];
00419   adm1266_parameters_t  adm1266;
00420 } adm_monitoring_t;
00421
00422 typedef struct
00423 {
00424   ffly_tx_parameters_t  ffly_tx[2+2]; // +2 for FLX-155
00425   ffly_rx_parameters_t  ffly_rx[2+2]; // +2 for FLX-155
00426   ffly_tr_parameters_t  ffly_tr[1+1]; // +1 for FLX-155
00427 } firefly_monitoring_t;
00428
00429 typedef struct
00430 {
00431   minipod_parameters_t  minipod[8];
00432 } minipod_monitoring_t;
00433
00434 typedef struct
00435 {
00436   fpga_monitoring_t     fpga;
00437   ltc_monitoring_t      ltc;
00438   ina_monitoring_t      ina;
00439   ltm_monitoring_t      ltm;
00440   tmp_monitoring_t      tmp;
00441   adm_monitoring_t      adm;
00442   minipod_monitoring_t  pod;
00443   firefly_monitoring_t  firefly;
00444 } monitoring_data_t;
00445
00446 // Macros
00447 #define TOHEX(n) std::hex << "0x" << ((uint64_t)n) << std::dec
00448
00449 class FlxCard
00450 {
00451 public:
00453     FlxCard();
00454
00455     ~FlxCard();
00456
00508     void card_open( int device_nr, u_int lock_mask,
00509                     bool read_config = false,
00510                     bool ignore_version = false );
00511
00518     void card_close();
00519
00526     static u_int number_of_cards();
00527
00539     static u_int number_of_devices();
00540
00557     static device_list_t device_list();
00558
00569     static int card_to_device_number( int card_number );
00570
00592     u_int lock_mask( int device_nr );
00593     u_int get_lock_mask( int device_nr ) { return lock_mask( device_nr ); } // Backwards-compatibility
00594
00600     u_long openBackDoor( int bar );
00604     u_long bar0Address() { return openBackDoor( 0 ); }
00608     u_long bar1Address() { return openBackDoor( 1 ); }
00612     u_long bar2Address() { return openBackDoor( 2 ); }
00626     int  dma_max_tlp_bytes() { return m_maxTlpBytes; }
00627
00646     void dma_to_host( u_int dma_id, u_long dst, size_t size, u_int flags );
00647
00680     void dma_from_host( u_int dma_id, u_long src, size_t size, u_int flags );
00681
00686     bool dma_enabled( u_int dma_id );
00687
00696     void dma_wait( u_int dma_id );
00697
00703     void dma_stop( u_int dma_id );
00704
00716     void dma_advance_ptr( u_int dma_id, u_long dst, size_t size, size_t bytes );
00717
00723     void dma_set_ptr( u_int dma_id, u_long dst );
00724
00732     u_long dma_get_ptr( u_int dma_id );
```

```
00733     /* For backwards compatibility: */
00734     u_long dma_get_read_ptr( u_int dma_id ) { return dma_get_ptr( dma_id ); }
00735
00736     // Sets the DMA_FIFO_FLUSH register to "1".
00737     //void dma_fifo_flush();  //MJ: disabled. See FlxCard.cpp
00738
00742     void dma_reset();
00743
00750     u_long dma_get_fw_ptr( u_int dma_id );
00751     /* For backwards compatibility: */
00752     u_long dma_get_current_address( u_int dma_id ) { return dma_get_fw_ptr( dma_id ); }
00753
00759     bool dma_cmp_even_bits( u_int dma_id );
00760
00795     void i2c_read( const char *device_str, u_char reg_addr, u_char *value, int nbytes = 1 );
00796
00800     void i2c_write( const char *device_str, u_char reg_addr, u_char data );
00801     void i2c_write( const char *device_str, u_char reg_addr, u_char *data, int nbytes );
00802
00814     void i2c_write_byte( u_char dev_addr, u_char byte );
00815
00821     void i2c_write_byte( u_char dev_addr, u_char reg_addr, u_char byte );
00822
00827     u_char i2c_read_byte( u_char dev_addr, u_char reg_addr );
00828
00836     bool i2c_flush( u_int *count = 0 );
00837
00842     void i2c_write_bytes( uint8_t dev_addr, uint8_t reg_addr,
00843                           int nbytes, uint8_t *bytes );
00844     void i2c_read_bytes ( uint8_t dev_addr, uint8_t reg_addr,
00845                           int nbytes, uint8_t *bytes );
00846
00866     u_int gbt_setup( int alignment, int channel_mode );
00867
00878     long int gbt_version_delay( u_long svn_version, char *filename );
00879
00895     u_long rxusrclk_freq( u_int channel );
00896
00901     void gth_rx_reset( int quad = -1 );
00902
00925     void irq_enable( u_int interrupt = ALL_IRQS );
00926
00940     void irq_disable( u_int interrupt = ALL_IRQS );
00941
00953     void irq_wait( u_int interrupt );
00954
00972     void irq_clear( u_int interrupt = ALL_IRQS );
00973
00987     void irq_cancel( u_int interrupt = ALL_IRQS );
00988
00999     void irq_reset_counters( u_int interrupt = ALL_IRQS );
01000
01024     u_long cfg_get_option( const char *key, bool show_options = false );
01025
01039     void cfg_set_option( const char *key, u_long value, bool show_options = false );
01040
01051     u_long cfg_get_reg( const char *key );
01052
01064     void cfg_set_reg( const char *key, u_long value );
01065
01071     void registers_reset();
01072
01079     static regmap_register_t *cfg_register( const char *name );
01086     static regmap_bitfield_t *cfg_bitfield( const char *name );
01098     static std::string cfg_bitfield_options( const char *name, bool include_all_substr = true );
01099
01115     u_int card_model() { return m_cardType; }
01116     u_int card_type() { return card_model(); }
01117
01121     u_int firmware_type() { return m_firmwareType; }
01122
01126     std::string firmware_type_string()
01127       { return( m_firmwareType<FIRMW_STR.size() ? FIRMW_STR[m_firmwareType] : std::string("????") ); }
01128
01132     std::string firmware_string() { return m_firmwareString; }
01133
01137     bool lpgbt_type() { return m_lpgbtType; }
01138
01142     bool lpgbt_fec12() { return m_lpgbtFec12; }
01143
01147     bool lpgbt_5gbps() { return m_lpgbt5Gbps; }
01148
01152     bool fullmode_type() { return m_fullmodeType; }
01153
01157     void soft_reset();
01158
01162     void cr_fromhost_reset();
```

```
01163
01167     u_int number_of_channels() { return m_numberOfChans; }
01168
01172     std::pair<int,int> ec_elink_indices();
01173
01177     std::pair<int,int> ic_elink_indices();
01178
01189     fpga_monitoring_t     fpga_monitoring();
01190     ltc_monitoring_t      power_monitoring_ltc2991();
01191     ina_monitoring_t      power_monitoring_ina226( int *ina_count = 0 );
01192     tmp_monitoring_t      temperature_monitoring_tmp435( int *tmp_count = 0 );
01193     ltm_monitoring_t      power_monitoring_ltm4700();
01194     adm_monitoring_t      powersequencer_monitoring_adm1x66( int *adm_count = 0 );
01195     minipod_monitoring_t  minipod_monitoring( u_int selection, int *number_of_pods = 0 );
01196     firefly_monitoring_t  firefly_monitoring( int *tx_count = 0, int *rx_count = 0, int *tr_count = 0
      );
01197
01221     monitoring_data_t get_monitoring_data( u_int mon_mask );
01222
01226     float fpga_temperature();
01227
01231     int fpga_fanspeed();
01232
01237     std::vector<int> minipods_temperature();
01238
01244     std::vector<int> minipods_optical_power();
01245
01252     bool firefly_detect( const std::string &device_name,
01253                          std::string &vendor_part,
01254                          std::string &vendor_sn );
01255
01272     void configure( std::string filename, bool do_links = true, bool do_registers = true );
01273
01278     void readConfiguration( std::string filename );
01279
01285     void readConfiguration();
01286
01291     u_int number_of_elinks_tohost( int dma_index = -1 );
01292
01297     u_int number_of_elinks_toflx();
01298     u_int number_of_elinks_fromhost() { return number_of_elinks_toflx(); }
01299
01304     std::vector<flxcard::elink_descr_t> elinks_tohost( int dma_index = -1 );
01305
01310     std::vector<flxcard::elink_descr_t> elinks_toflx();
01311     std::vector<flxcard::elink_descr_t> elinks_fromhost() { return elinks_toflx(); }
01312
01313     std::vector<flxcard::elink_descr_t> elinks_toflx_broadcast();
01314     std::vector<flxcard::elink_descr_t> elinks_fromhost_broadcast() { return elinks_toflx_broadcast();
      }
01315
01319     u_int number_of_dma_tohost() { return m_numberOfDma; }
01320
01325     bool uses_dma_index_mask() { return m_useDmaIndexMask; }
01326
01327     void set_hdlc_instant_timeout( bool b ) { m_hdlcInstantTimeout = b; }
01328     bool hdlc_instant_timeout( ) { return m_hdlcInstantTimeout; }
01329
01333     unsigned int link_mode();
01334
01338     bool is_elink_enabled( u_int channel, u_int egroup, u_int epath, bool is_to_flx );
01339
01343     unsigned int tohost_elink_dmaid( u_int channel, u_int egroup, u_int epath );
01344
01348     unsigned int tohost_elink_dmaid( uint16_t elinknr );
01349
01353     bool has_tohost_elink_streams( u_int channel, u_int egroup, u_int epath );
01354
01359     uint8_t detector_id();
01360
01367     uint16_t connector_id( uint16_t channel );
01368
01373 private:
01374     static int  m_cardsOpen;
01375     int         m_fd;
01376     int         m_deviceNumber;
01377     int         m_maxTlpBytes;
01378     u_int       m_cardType;
01379     u_int       m_firmwareType;
01380     bool        m_configRead;
01381     bool        m_lpgbtType;
01382     bool        m_lpgbtFec12;
01383     bool        m_lpgbt5Gbps;
01384     bool        m_fullmodeType;
01385     bool        m_useDmaIndexMask;
01386     bool        m_hdlcInstantTimeout;
01387     u_int       m_numberOfChans;
```

```
01388     u_int       m_numberOfDma;
01389     u_int       m_fromHostDataFormat;
01390     std::string m_firmwareString;
01391     u_int       m_myLocks;
01392     u_int       m_myLockTag;  //MJ: do we still need the lock tag??
01393     u_long      m_bar0Base;
01394     u_long      m_bar1Base;
01395     u_long      m_bar2Base;
01396
01397     // Space to store a device's e-link configuration, for all links + Emulator 'link'
01398     flxcard::LinkConfig m_linkConfig[flxcard::FLX_LINKS + 1];
01399
01400     // Space to store additional register settings
01401     std::vector<flxcard::regsetting_t> m_regSettings;
01402
01403 public:
01409     volatile flxcard_bar0_regs_t *m_bar0;
01410     volatile flxcard_bar1_regs_t *m_bar1;
01411     volatile flxcard_bar2_regs_t *m_bar2;
01412
01413     // Pointer to a list of this FLX device's I2C devices
01414     i2c_device_t const *m_i2cDevices;
01415
01420 private:
01421     u_long map_memory_bar( u_long pci_addr, size_t size );
01422     void   unmap_memory_bar( u_long vaddr, size_t size );
01423
01424     void   i2c_wait_not_full( );
01425     void   i2c_wait_not_empty( );
01426     int    i2c_parse_address_string( const char *str,
01427                                      u_char *port1, u_char *port2, u_char *port3,
01428                                      u_char *dev_addr );
01429     void   i2c_set_switches( u_char switch1_val, u_char switch2_val, u_char switch3_val );
01430
01431     void   gbt_tx_configuration( int channel_tx_mode, int alignment );
01432     int    gbt_rx_configuration( int channel_rx_mode );
01433     int    gbt_software_alignment( int number_channels );
01434     bool   gbt_channel_alignment( u_int channel );
01435     void   gbt_topbot_oddeven_set( u_int channel, u_int topbot, u_int oddeven );
01436     bool   gbt_topbot_alignment( u_int channel, u_int topbot,
01437                                  u_long *phase_found, u_int *oddeven );
01438     u_long gbt_shift_phase( u_int channel );
01439
01440     int    check_digic_value2( const char *str, u_long *version, u_long *delay );
01441
01442     std::string lockbits2string( u_int locks );
01443
01444     void configureLinks( flxcard::LinkConfig *link_config,
01445                          int *dma_index_invalid_count,
01446                          int *unsupported_tohost_elinks_count,
01447                          int *unsupported_fromhost_elinks_count );
01448
01449     void lockForConfiguration( FlxCard &flx );
01450 };
01451
01452 #endif // FLXCARD_H
```

Index