FMC ADC Programmer's Interface

September 2014 (fmc-adc-100m-sw-2014-05-44-g91e47c2-dirty)
A generic API for ADC devices

CERN BE-CO-HT / Alessandro Rubini and others

Table of Contents

Introduction

This is the documentation for the programmer's interface (API) that is expected to accompany most or all of the ADC cards being developed and used by the Controls grout at CERN.

This design is the outcome of email discussion between Michel Arruat, David Cobas, Federico Vaga and Alessandro Rubini in March 2013. Ideas have later been refined during real use. Alessandro wrote this document.

1 Bugs and Missing Features

To set the record straight, we'd better list the known issues right off at the start. We are aware of the issues, but they are not fixed at this point in time.

- The "parameters" functions are not documented, although they are here to stay.
- The library should automatically choose the buffering options: for zio, buffer type and size adjustment. This is not implemented and users must do that by themselves (see the driver manual about how to do it).
- Internals are not documented, while we think some guidance in understanding the code is useful, in case users need to trace unexpected behaviours that may be library bugs or application bugs. Also, verbose mode exists but it not documented.
- Verbose mode is not verbose enough: *sysfs* accesses are only reported in case of error, while a complete trace of them is often useful to understand what is going on. Similarly, "log to file" instead of "verbose to stderr" should be added as an option.
- Internally, a number of functions must be renamed, and a better split between api-generic, zio-generic and driver-specific code must be put in place.
- Some code parts, especially configuration, should be restructured to use data tables instead of open-coded conditionals. This is important for maintainance and easy addition of new boards.
- We need to add *reset_conf* and related functions, that are currently missing. Also, direct access to configuration structures inside the device structure should be provided. The current API will remain, though, we plan to just add easier use cases.

2 General Ideas and Rationale

The API is meant to be as generic, so the same code can be run mostly unchanged with a variety of boards and driver types. This is why the name used for headers and libraries is simply *fmc-adc*, without a card name in there – the *fmc* prefix reflects the current situation where all foreseen cards in the group are in the FMC form factor,

The user, however, is expected to select the card being worked on, by means of a name and a *device ID*, which generally reflects the geographic placement of the card.

2.1 Buffers

Each acquisition run is relying on a buffer. The buffer includes both *data* and *metadata*. The layout of data is depending on the card and its configuration; the metadata part depends on the type of driver currently running for the card. For example, if the driver is based on the ZIO framework, the *metadata* part of the buffer refers to a ZIO control block. The idea is that all information is available to the user, who can use it at will or ignore it and just use generic interfaces.

Buffers are generally allocated by the library, but the application can provide its own allocator if it really wants to. The reason for using library-driven allocation is that the library knows better. If, for example, the driver offers *mmap* support, the library can choose to map acquired data instead of calling *malloc*. However, for the rare case where the application knows better, it can override this. Also, the application should tell the number of buffers and expected data size at the beginning, so the library can properly configure the underlying driver, if this is needed.

For ZIO drivers, you can use two buffer types: *kmalloc* and *vmalloc*. Please check section *Buffers* in the *fmc-adc-driver* manual (discussion will be moved here as we finalize automatic setting of buffer features).

2.2 Configuration

Configuration is performed by passing parameters as 32-bit numbers. The library defines arrays of such parameters, one for each aspect of the overall problem (triggers, data, and so on). Each item in the array has a symbolic name, and each array is associated with a bitmask that specifies which parameters have been set. The choice of arrays is driven by the need for generic structures that can be used unchanged with different hardware cards.

While there is a little overhead in parsing the generic structures, configuration is not something that happens in hot paths, and we see no problems in that. Also, please note that setting the configuration and applying it are different steps, and the configuration data structures can be prepared and saved for later use.

2.3 Implementation Status

Currently the library is only implemented for the "FmcAdc100M14b4cha" card, driven by a ZIO driver. The API however will not change as we add new cards and drivers; we'll likely need a few new configuration items to match the specifics of the new cards, but this will work as long as you link the newer version of the library.

3 Error Reporting

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

`errno` values can be standard Posix items like `EINVAL`, or library-specific values, `FMCADC_ENOSET` ("Cannot set requested item"). All library-specific error values have a value greater than 1024, to prevent collision with standard values. To convert such values to a string please use the function:

```
char *fmcadc_strerror(int error);
```

The function returns static storage, so you can keep around the pointer it returns, if needed. Similarly, there is no concurrency between getting the string and using it, not even in multi-threaded environments (`errno` itself is already protected, in this respect).

4 Initialization and Cleanup

The library may keep internal information, so the application should call its initialization function. After use, it should call the exit function to release internal data, but it is not mandatory to do that at program termination, because the operating system releases anything in any case – the library doesn't leave unexpected files in persistent storage.

```
int fmcadc_init(void);
void fmcadc_exit(void);
```

The functions don't do anything at this point, but they may be implemented in later releases. For example, the library may scan the system and cache the list of peripheral cards found, to make later *open* calls faster.

5 Opening and closing

Each device must be opened before use, and it should be closed after use. It is not mandatory to close if the process is going to terminate, as the library has no persistent storage to clean up – but there may be persistent buffer storage allocated, and *fmcadc_close* may release it in future versions.

```
fmcadc_dev *fmcadc_open(char *name, unsigned int dev_id,
                        unsigned long totalsamples,
                        unsigned int nbuffer,
                        unsigned long flags)
struct fmcadc_dev *fmcadc_open_by_lun(char *name, int lun,
                                      unsigned long totalsamples,
                                      unsigned int nbuffer,
                                      unsigned long flags);
extern int fmcadc_close(struct fmcadc_dev *dev);
```

This is the meaning of the various arguments:

`fmcadc_dev`
: The device is an opaque object for the user. It should be passed around but not be looked into.

`name`
: Devices are opened by name, and the name for the only supported card at the moment is "`fmc-adc-100m14b4cha`".

`dev_id`
: The device identifier is used to enumerate several cards in the same system. The number is usually dependent on the geographic placement of the card (bus number, slot number).

`lun`
: Logical Unit Number. This number comes from a database description of the system (*open_by_lun* is not currently implemented).

`totalsamples`
: This is a a hint about how big a buffer the application will use. Managing big acquisitions (hundreds of megabytes, or gigabytes) requires some pre-allocation of the data, and sometimes this configuration happens at device level, so it's good to have the information at open time. This is a number of samples, i.e. `nshots * (presamples + postsamples)`, not a number of bytes.

`nbuffer`
: This is a hint about how many buffers are being used at the same time by the application. For example, a multi-shot acquisition requires all buffers to be available at the same time. Again, this information may require device-level configuration at open time.

`flags`

This argument is used to pass user flags. The library currently supports `FMCADC_F_FLUSH` (that reads and discards any input samples possibly left over by a previous acquisition) and `FMCADC_F_VERBOSE` (that enables diagnostic messages to *stderr*).

currently unused, but some driver may need to have some more information, or flags, at open time.

6 Time Stamps

The timestamp structrure is defined as follows:

```
struct fmcadc_timestamp {
        uint64_t secs;
        uint64_t ticks;
        uint64_t bins;
};
```

This is the same structure as used by the ZIO framework, but it is not specific to ZIO – the choice made there was just the best of breed, agreed upon in a discussion held within the HT section.

Currently, timestamps are only used in association with buffers: after an acquisition is over and saved to a buffer, the user can ask for the timestamp of the acquired buffers. See Chapter 9 [Buffers], page 6

7 Configuration

Configuration is the most intensive part of the library, because there are a number of parameters that can be set or retrieved.

Unfortunately, for lack of time on my side, it is not properly documented here.

Briefly, configuration is described by a few data structures. Each structure includes a "type" identifier, some internal fields and an array of configuration values. A bitmask (in the structure itself) states which configuration values are active. All parameters are 32 bits wide; each structure includes 64 values. The position of each parameter in the array is fixed, and as boards are added to this library we may need to add new values for unforeseen requirements. We don't expect to change the API, nor replace the meaning of the already-defined array members.

Please check the header file to know the currently-defined parameters.

The library offsers the following functions related to configuration:

```
void fmcadc_set_conf(struct fmcadc_conf *conf,
                     unsigned int conf_index, uint32_t val);
int fmcadc_get_conf(struct fmcadc_conf *conf,
                    unsigned int conf_index,
                    uint32_t *val);
int fmcadc_reset_conf(struct fmcadc_dev *dev, unsigned int flags,
                      struct fmcadc_conf *conf);
int fmcadc_apply_config(struct fmcadc_dev *dev, unsigned int flags,
                        struct fmcadc_conf *conf);
int fmcadc_retrieve_config(struct fmcadc_dev *dev,
                           struct fmcadc_conf *conf);
```

The *set* and *get* functions access a single configuration element of the configuration structure. Please remember that there are several configuration structures, but indexes start from 0 for each of them – future revisions of this library may include a non-ambiguous naming, while still keeping the current API for compatibility.

The *reset* function (which is not yet implemented) sets all items in the structure to the advertised default for the specific device (it can only be called after opening the device). Like the previous two, it is only concerned with data structures, and it makes no hardware access.

The *apply* and *retrieve* functions talk with the device driver, to transfer active items of the configuration to the hardware. As mnemonic, you can keep in mind that the functions with the longer `config` name do more (talk with hardware), while those with the shorter name `conf` do less.

8 Acquisition

The library offers three functions related to acquisition:

```
int fmcadc_acq_start(struct fmcadc_dev *dev, unsigned int flags,
                     struct timeval *timeout);
int fmcadc_acq_poll(struct fmcadc_dev *dev, unsigned int flags,
                     struct timeval *timeout);
int fmcadc_acq_stop(struct fmcadc_dev *dev, unsigned int flags);
```

The *start* function tells hardware to start acquisition according to the current configuration (but starting in itself is usually a fast operation, because it doesn't rewrite configuration to the hardware, at least with currently supported boards). It can return immediately or wait for completion, with a timeout. The function supports `FMCADC_F_FLUSH`, to discard previous leftover data before activating the acquisition, if any is there.

The *poll* function waits for acquisition to complete. Again, it can return immediately or wait. A return value of 0 means acquisition is over. For multi-shot we would like to return the number of still-missing shots, but this is not yet supported.

The *stop* function stops an already-begun acquisition. It is not expected to be of frequent use.

According to the low-level implementation, functions waiting for data (i.e. *start* when passed a timeout and *poll*) may fail with `FMCADC_EDISABLED` if the acquisition is externally disabled by other entities. ZIO supports that (since Nov 2013) by setting `POLLERR` when a user disables the trigger through a sysfs write. Other backends may offer their own notification means.

This is the meaning of the arguments:

`dev`

The device returned by a previous `fmcadc_open` call.

`flags`

Flags to request special actions. 0 always selects the default behavior. Currently, the *start* function supports `FMCADC_F_FLUSH` – this is not supported by the *poll* function, because after telling hardware to start I/O we cannot safely discard "old" data. Flags can be used for other things: for example, we may soon define a flags to automatically enable a new acquisition when the previous one is over, for stream-like applications. (In that case after each successful *poll* the next call would wait for another acquisition to be over).

`timeout`

The timeout is used like the argument for *select(2)*: if NULL, the functions wait forever, otherwise they specify the maximum allowed waiting time – and passing zero values asks to not wait at all.

9 Buffers

The buffer is a data structure. It includes data and metadata (but the format of metadata depends on the specific driver in charge of the hardware), as well as other informative fields:

```
struct fmcadc_buffer {
        void *data;
        void *metadata;
        int samplesize;
        int nsamples;
        struct fmcadc_dev *dev;
        unsigned long flags; /* internal to the library */
        void *mapaddr;
        unsigned long maplen;
};
```

The buffer is allocated and released by the library. The rationale is described in Section 2.1 [Buffers Overview], page 1. the application is not allowed to pass its own buffer. It is possible, however, for the application to override the allocation for data by passing a pointer to its own preferred allocator.

The following functions are related to buffers:

```
struct fmcadc_buffer *fmcadc_request_buffer(struct fmcadc_dev *dev,
                                            int nsamples,
                                            void *(*alloc_fn)(size_t),
                                            unsigned int flags);
int fmcadc_fill_buffer(struct fmcadc_dev *dev,
                       struct fmcadc_buffer *buf,
                       unsigned int flags,
                       struct timeval *timeout);
struct fmcadc_timestamp *fmcadc_tstamp_buffer(struct fmcadc_buffer *buf,
                                              struct fmcadc_timestamp *ts);
int fmcadc_release_buffer(struct fmcadc_dev *dev,
                          struct fmcadc_buffer *buf,
                          void (*free_fn)(void *));
```

The `request_buffer` function allocates an empty buffer and returns it to the user. It returns `NULL` on failure. The buffer is empty, and neither the *data* nor thr *metadata* can be accessed until the buffer is filled.

The `fill_buffer` function is run after an acquisition, and fills the buffer with the next available shot. The size of the buffer must be at least equal to the size of the shot; otherwise the result is undefined. The function may fail with `FMCADC_EDISABLED` if the acquisition is externally disabled/aborted by other entities.

`tstamp_buffer` extracts the acquisition timestamp from the buffer, in a driver-specific way (most likely looking in the metadata structure). If the caller passes a `ts` pointer, the timestamp is copied into that pointer and the same pointer is returned. If `ts` is NULL, the function returns a pointer of its own choice, that is only valid until the buffer is released.

The `release_buffer` function releases any resources associated with the buffer.

This is the meaning of the various arguments, in the order in which they appear:

`dev`

The device must be the result of an `fmcadc_open` call (or *open_by_lun*).

`nsamples`

The number of samples associated with this buffer (the size of each sample is known by the device type). This is the total number, so a 4-multiplexed acquisition of 10 samples requires 40 samples here. If the application plans to run several acquisitions (even of different sizes), it can allocate the buffers beforehand. Thus, the driver can't know the `nsamples` value in advance. The number of samples in the buffer may be bigger than the number of samples that will actually be acquired later.

`alloc_fn`

The pointer is usually expected to be NULL, as the driver knows how to allocate the buffer. However, if the pointer is not null, the library will use it to allocate the data area of the buffer (the buffer structure and the metadata are not allocated with this function).

`flags`

Currently unused, available for future special cases.

`buf`

The buffer pointer, as returned by the allocation function.

`timeout`

The timeout for filling the buffer is used like the timeout in *select(2)*. If NULL, there is no timeout, if set, it is used as a maximum waiting time. If allocation times out, `NULL` is returned, with `-ETIMEDOUT` as *errno*.

`ts`

A pointer to timestamp, so the library can copy the timestamp to user-provided storage.

`free_fn`

If the data section of the buffer was allocated by a custom allocator, this is the pointer to associated *free* function. The two function pointers match the prototypes of *malloc* and *free*.

By accepting a user-defined allocator, we allow customized management of the data area in the most flexible way. The application may have special needs that are unforeseeable by the driver. In the simplest case, the application needs to send out or save the data with its own ancillary information: the custom allocator in this case can return the pointer inside the pre-built structure. If the driver retrieves data using the *read* system call, this trick can save a data copy overall – if the driver would see data through *mmap*, there is no saving in using the custom allocator but no additional cost, either.

10 Internals

To be documented in a later version.

11 Incompatibilities

During June 2013 we defined the final API as documented here. However, code written before June 28th may use the initial conventions, that are no more active. The current API is designed to be persistent over the years, and work with any ADC card belonging to the same family as our first board, the “fmc-adc-100m14b4cha” one.

This is the list of incompatibilities. I list the commit, so you can see the actual change in *git* history if you are affected.

- Some configuration names changed. Now naming is consistent across all configuration macros. Change applied in commit 78a2ac0b.
- The data structures have 64 configuration items, not 32. This requires to use `long long` for the mask, but users are strongly encouraged to use the helpers *conf_set* and *conf_get*. Change applied in commit aa1eae16.

- Configuration functions use `conf` in their name, not `attr`. Change applied in commit 34c93097.
- The *fmcadc_strerror* function takes only the `errnum` argument, not `dev` any more. Applied in commit e83791de.
- Applications are asked to call *fmcadc_init* and *fmcadc_exit* (but the latter is optional at program termination).
- The open function states the number of shots and the total data size, so the library can prepare for the best allocation strategy. Applied in commit 6f3e4435.
- Buffer functions are different: the buffer is allocated by the library (not by the caller any more) and there is a new function to fill the buffer. Commit 3e83241e implements the prototype, but keeps the previous semantic; code and examples have been fixed later.
- A new *reset_conf* function exists, to force default values on all configuration variables. This can be used to prevent configuration leakage from the previous acquisition to the next. Resetting configuration is optional, so you can use a process to configure the card and another to acquire (I personally set range and gain before I run acquisition programs, for example).
- There is a new function to poll for acquisition. This is useful since the process may want to do something else while waiting for the trigger.

Index