FMC Bus Abstraction for Linux

Version 1.0, October 2012
A kernel bus to support FMC mezzanines and carriers

Alessandro Rubini for CERN (BE-CO-HT)
# Table of Contents

**Introduction** ................................................................. 1

1 What is a Linux Bus ......................................................... 1

2 Functions Exported by fmc.ko .......................................... 1

3 FMC Device ................................................................. 2
   3.1 The API Offered by Carriers ...................................... 2
   3.2 The GPIO Abstraction .............................................. 4

4 FMC Driver ................................................................. 5
   4.1 Module Parameters .................................................. 5
   4.2 fmc-trivial .......................................................... 6
   4.3 fmc-write-eeprom .................................................. 6
Chapter 2: Functions Exported by fmc.ko

Introduction

This document describes the implementation of the fmc bus for Linux. FMC (FPGA Mezzanine Carrier) is the standard we use for our I/O devices, in the context of White Rabbit and related hardware.

In our I/O environments we need to write drivers for each mezzanine card, and such drivers must work independent of the carrier being used. To achieve this, we abstract the FMC interface.

We have a carrier for PCI-E called SPEC and one for VME called SVEC, but more are planned. Also, we support stand-alone devices (usually plugged on a SPEC card), controlled through Etherbone, developed by GSI.

Code and documentation for the FMC bus was born as part of the spec-sw project, but now it lives in its own project. Other projects, i.e. software support for the various carriers, should include this as a submodule.

The most up to date version of code and documentation is always available from the repository you can clone from:

- `git://ohwr.org/fmc-projects/fmc-bus.git` (read-only)
- `git@ohwr.org:fmc-projects/fmc-bus.git` (read-write for developers)

Selected versions of the documentation, as well as complete tar archives for selected revisions are placed to the Files section of the project: [http://www.ohwr.org/projects/fmc-bus/files](http://www.ohwr.org/projects/fmc-bus/files)

1 What is a Linux Bus

Within the Linux kernel, a bus is a data structure with a few methods. It’s main role is registering a list of devices and a list of drivers, offering a match function that compares the respective identifiers (in a bus-specific way) to assign drivers to devices.

Activation and deactivation of devices happens through the probe and remove functions of the respective driver; an advanced user can also use sysfs to change the binding of drivers to devices (for example, if more than one driver can drive the same device you may want to force the choice).

2 Functions Exported by fmc.ko

The FMC core exports the usual 4 functions that are needed for a bus to work:

```c
int fmc_driver_register(struct fmc_driver *drv);
void fmc_driver_unregister(struct fmc_driver *drv);
int fmc_device_register(struct fmc_device *tdev);
void fmc_device_unregister(struct fmc_device *tdev);

uint32_t fmc_readl(struct fmc_device *fmc, int offset);
void fmc_writel(struct fmc_device *fmc, uint32_t val, int off);
void *fmc_get_drvdata(struct fmc_device *fmc);
void fmc_set_drvdata(struct fmc_device *fmc, void *data);
```

They should be self-explanative, so nothing is added here. The data structure that describe a device is detailed in Chapter 3 [FMC Device], page 2, the one that describes a driver is detailed in Chapter 4 [FMC Driver], page 5.
3 FMC Device

Within the Linux bus framework, the FMC device is created and registered by the carrier driver. For example, the PCI driver for the SPEC card fills a data structure for each SPEC that it drives, and registers an associated FMC device. The SVEC driver can do exactly the same for the VME carrier (actually, it should do it twice, because the SVEC carries two FMC mezzanines). Similarly, an Etherbone driver will be able to register its own FMC devices, offering communication primitives through frame exchange.

The contents of the EEPROM within the FMC will be used for identification purposes, i.e. for matching the device with its own driver. For this reason the device structure includes a complete copy of the EEPROM (actually, the carrier driver may choose to only return the leading part of it).

The following listing shows the current structure defining a device. Please note that all the machinery is in place but some details may still change in the future. For this reason, there is a version field at the beginning of the structure. As usual, the minor number will change for compatible changes (like a new flag) and the minor number will increase when an incompatible change happens (for example, a change in layout of some fmc data structures). Device writers should just set it to the value FMC_VERSION, and be ready to get back -EINVAL at registration time.

```c
struct fmc_device {
    unsigned long version; /* to be set to FMC_VERSION */
    struct fmc_device_id id; /* for the match function */
    struct fmc_operations *op; /* carrier-provided */
    int irq; /* according to host bus. 0 == none */
    int eeprom_len; /* Usually 8kB, may be less */
    uint8_t *eeprom; /* Full contents or leading part */
    char *carrier_name; /* "SPEC" or similar, for special use */
    void *carrier_data; /* "struct spec *" or equivalent */
    __iomem void *base; /* May be NULL (Etherbone) */
    struct device dev; /* For Linux use */
    struct device *hwdev; /* The underlying hardware device */
    struct sdb_array *sdb;
    void *mezzanine_data;
};
```

The meaning of each field is summarized in its own line above. All of the fields must be filled by the carrier driver before registration, with a few exceptions; please note that hwdev is used for messages, using dev_err() or similar functions, so it must be properly set or the system will Oops with a NULL pointer pretty soon. Similarly, the carrier must read its own EEPROM memory before registering the driver.

The fields that are not set by the carrier are: fmc_device_id, which is set by the bus controller according to EEPROM contents; sdb, which is set by the bus controller when scanning an SDB bus; mezzanine_data which is a pointer used by the mezzanine driver.

**Note:** mezzanine_data may be redundant, because Linux offers the drvdata approach, to the field may be removed in later versions of this bus implementation.

As I write this, she SPEC carrier is already completely functional in the fmc-bus environment, and is a good reference to look at.

3.1 The API Offered by Carriers

The carrier provides a number of methods by means of the fmc_operations structure, which currently is defined like this (again, it is a moving target, please refer to the header rather than this document):

```c
struct fmc_operations {
    uint32_t (*readl)(struct fmc_device *fmc, int offset);
};
```
Chapter 3: FMC Device

```c
void (*writel)(struct fmc_device *fmc, uint32_t value, int offset);
int (*reprogram)(struct fmc_device *f, struct fmc_driver *d, char *gw);
int (*validate)(struct fmc_device *fmc, struct fmc_driver *drv);
int (*irq_request)(struct fmc_device *fmc, irq_handler_t h,
                      char *name, int flags);
void (*irq_ack)(struct fmc_device *fmc);
int (*irq_free)(struct fmc_device *fmc);
int (*gpio_config)(struct fmc_device *fmc, struct fmc_gpio *gpio,
                      int ngpio);
int (*read_ee)(struct fmc_device *fmc, int pos, void *d, int l);
int (*write_ee)(struct fmc_device *fmc, int pos, const void *d, int l);
```

The individual methods perform the following tasks:

**readl**

**writel**

These functions access FPGA registers by whatever means the carrier offers. They are not expected to fail, and most of the time they will just make a memory access to the host bus. If the carrier provides a `base` pointer, the driver may use direct access through that pointer. For this reason the header offers the inline functions `fmc_readl` and `fmc_writel` that access `base` if the respective method is NULL. A driver that wants to be portable and efficient should use `fmc_readl` and `fmc_writel`. For Etherbone, or other non-local carriers, error-management is still to be defined.

**validate**

Module parameters are used to manage different applications for two or more boards of the same kind. Validation is based on the `busid` module parameter, if provided, and returns the matching index in the associated array. See Section 4.1 [Module Parameters], page 5 in doubt. If no match is found, `-ENOTENT` is returned; if the user didn’t pass `busid=`, all devices will pass validation. The value returned by the validate method can be used as index into other parameters (for example, some drivers use the `lm32=` parameter in this way). Such "generic parameters" are documented in Section 4.1 [Module Parameters], page 5, below. The `validate` method is used by `fmc-trivial.ko`, described in Section 4.2 [fmc-trivial], page 6.

**reprogram**

The carrier enumerates FMC devices by loading a standard (or golden) FPGA binary that allows EEPROM access. Each driver, then, will need to reprogram the FPGA by calling this function. If the name argument is NULL, the carrier should reprogram the golden binary. If the gateware name has been overridden through module parameters (in a carrier-specific way) the file loaded will match the parameters. Per-device gateware names can be specified using the `gateware=` parameter, see Section 4.1 [Module Parameters], page 5.

**irq_request**

**irq_ack**

**irq_free**

Interrupt management is carrier-specific, so it is abstracted as operations. The interrupt number is listed in the device structure, and for the mezzanine driver the number is only informative. The handler will receive the `fmc` pointer as `dev_id`; the `flags` argument is passed to the Linux `request_irq` function, but fmc-specific flags may be added in the future. You’ll most likely want to pass the `IRQF_SHARED` flag.

**gpio_config**

The method allows to configure a GPIO pin in the carrier, and read its current value if it is configured as input. See Section 3.2 [The GPIO Abstraction], page 4 for details.
Chapter 3: FMC Device

read_ee
write_ee

Read or write the EEPROM. The functions are expected to be only called before reprogramming and the carrier should refuse them with \texttt{ENODEV} after reprogramming. The offset is expected to be within 8kB (the current size), but addresses up to 1MB are reserved to fit bigger I2C devices in the future. Carriers may offer access to other internal flash memories using these same methods: for example the SPEC driver may define that its carrier I2C memory is seen at offset 1M and the internal SPI flash is seen at offset 16M. This multiplexing of several flash memories in the same address space is is carrier-specific and should only be used by a driver that has verified the \texttt{carrier\_name} field.

3.2 The GPIO Abstraction

Support for GPIO pins in the \textit{fmc\textunderscore bus} environment is not very straightforward and deserves special discussion.

While the general idea of a carrier-independent driver seems to fly, configuration of specific signals within the carrier needs at least some knowledge of the carrier itself. For this reason, the specific driver can request to configure carrier-specific GPIO pins, numbered from 0 to at most 4095. Configuration is performed by passing a pointer to an array of \texttt{struct fmc\_gpio} items, as well as the length of the array. This is the data structure:

```c
struct fmc_gpio {
  char *carrier_name;
  int gpio;
  int _gpio; /* internal use by the carrier */
  int mode; /* GPIOF\_DIR\_OUT etc, from \textlt{\textless}linux/gpio.h\textgt \ */
  int irqmode; /* IRQF\_TRIGGER\_LOW and so on */
};
```

By specifying a \texttt{carrier\_name} for each pin, the driver may access different pins in different carriers. The \texttt{gpio\_config} method is expected to return the number of pins successfully configured, ignoring requests for other carriers. However, if no pin is configured (because no structure at all refers to the current \texttt{carrier\_name}), the operation returns an error so the caller will know that it is running under a yet-unsupported carrier.

So, for example, a driver that has been developed and tested on both the SPEC and the SVEC may request configuration of two different GPIO pins, and expect one such configuration to succeed – if none succeeds it most likely means that the current carrier is a still-unknown one.

If, however, your GPIO pin has a specific known role, you can pass a special number in the \texttt{gpio} field, using one of the following macros:

```c
#define FMC\_GPIO\_RAW(x) \(x\) /* 4096 of them */
#define FMC\_GPIO\_IRQ(x) \((x) + 0x1000\) /* 256 of them */
#define FMC\_GPIO\_LED(x) \((x) + 0x1100\) /* 256 of them */
#define FMC\_GPIO\_KEY(x) \((x) + 0x1200\) /* 256 of them */
#define FMC\_GPIO\_TP(x) \((x) + 0x1300\) /* 256 of them */
#define FMC\_GPIO\_USER(x) \((x) + 0x1400\) /* 256 of them */
```

Use of virtual GPIO numbers (anything but \texttt{FMC\_GPIO\_RAW}) is allowed provided the \texttt{carrier\_name} field in the data structure is left unspecified (NULL). Each carrier is responsible for providing a mapping between virtual and physical GPIO numbers. The carrier may then use the \texttt{_gpio} field to cache the result of this mapping.

All carriers must map their I/O lines to the sets above starting from zero. The SPEC, for example, maps interrupt pins 0 and 1, and test points 0 through 3 (even if the test points on the PCB are called 5,6,7,8).
If, for example, a driver requires a free LED and a test point (for a scope probe to be plugged at some point during development) it may ask for `FMC_GPIO_LED(0)` and `FMC_GPIO_TP(0)`. Each carrier will provide suitable GPIO pins. Clearly, the person running the drivers will know the order used by the specific carrier driver in assigning leds and testpoints, so to make a carrier-dependent use of the diagnostic tools.

In theory, some form of autodetection should be possible: a driver like the `wr-nic` (which uses `IRQ(1)` on the SPEC card) should configure `IRQ(0)`, make a test with software-generated interrupts and configure `IRQ(1)` if the test fails. This probing step should be used because even if the `wr-nic` gateware is known to use `IRQ1` on the SPEC, the driver should be carrier-independent and thus use `IRQ(0)` as a first bet – actually, the knowledge that `IRQ0` may fail is carrier-dependent information, but using it doesn’t make the driver unsuitable for other carriers.

The return value of `gpio_config` is defined as follows:

- If no pin in the array can be used by the carrier, `-ENODEV`.
- If at least one virtual GPIO number cannot be mapped, `-ENOENT`.
- On success, 0 or positive. The value returned is the number of high input bits (if no input is configured, the value for success is 0).

While I admit the procedure is not completely straightforward, it allows configuration, input and output with a single carrier operation. Given the typical use case of FMC devices, GPIO operations are not expected to ever be in hot paths, and GPIO access so far has only been used to configure the interrupt pin, mode and polarity. Especially reading inputs is not expected to be common. If your device has GPIO capabilities in the hot path, you should consider using the kernel’s GPIO mechanisms.

## 4 FMC Driver

An FMC driver is concerned with the specific mezzanine and associated gateware. As such, it is expected to be independent of the carrier being used. The matching between device and driver is only based on the content of the EEPROM (as mandated by the FMC standard) and the driver will perform I/O accesses only by means of carrier-provided functions.

In some special cases it is possible for a driver to directly access FPGA registers, by means of the `base` field of the device structure. This may be needed for high-bandwidth peripherals like fast ADC cards. If the `device` module registered a remote device (for example by means of Etherbone), the `base` pointer will be NULL. Therefore, drivers must be ready to deal with NULL base pointers, and fail gracefully. Most driver, however, are not expected to access the pointer directly but run `fmc_readl` and `fmc_writel` instead, which will work in any case.

In even more special cases, the driver may access carrier-specific functionality: the `carrier_name` string allows the driver to check which is the current carrier and make use of the `carrier_data` pointer. We chose to use carrier names rather than numeric identifiers for greater flexibility, but also to avoid a central registry within the `fmc.h` file – we hope other users will exploit our framework with their own carriers. An example use of carrier names is in GPIO setup (see Section 3.2 [The GPIO Abstraction], page 4), although the name match is not expected to be performed by the driver. If you depend on specific carriers, please check the carrier name and fail gracefully if your driver finds it is running in an unknown environment.

### 4.1 Module Parameters

Most of the FMC drivers need the same set of kernel parameters. This package includes support to implement common parameters by means of fields in the `fmc_driver` structure and simple macro definitions.
The parameters are carrier-specific, in that they rely on the busid concept, that varies among carriers. For the SPEC, the identifier is a PCI bus and devfn number, 16 bits wide in total; drivers for other carriers will most likely offer something similar but not identical, and some code duplication is unavoidable.

This is the list of parameters that are common to several modules to see how they are actually used, please look at spec-trivial.c.

busid=

This is an array of integers, listing carrier-specific identification numbers. For PIC, for example, 0x0400 represents bus 4, slot 0. If any such ID is specified, the driver will only accept to drive cards that appear in the list (even if the FMC ID matches). This is accomplished by the validate carrier method.

gateware=

The argument is an array of strings. If no busid= is specified, the first string of gateware= is used for all cards; otherwise the identifiers and gateware names are paired one by one, in the order specified.

show_sdb=

For modules supporting it, this parameter asks to show the SDB internal structure by means of kernel messages. It is disabled by default because those lines tend to hide more important messages, if you look at the system console while loading the drivers.

For example, if you are using the trivial driver to load two different gateware files to two different cards, you can use the following parameters to load different binaries to the cards, after looking up the PCI identifiers. This has been tested with a SPEC carrier.

```
inmod fmc-trivial.ko \ 
  busid=0x0200,0x0400 \ 
  gateware=fmc/fine-delay.bin,fmc/simple-dio.bin
```

Please note that not all sub-modules support all of those parameters. You can use modinfo to check what is supported by each module.

4.2 fmc-trivial

The simple module fmc-trivial is just a simple client that registers an interrupt handler. I used it to verify the basic mechanism of the FMC bus and how interrupts worked.

The module implements the generic FMC parameters, so it can program a different gateware file in each card. The whole list of parameters it accepts are:

busid=  
gateware=  

Generic parameters. See Section 4.1 [Module Parameters], page 5.

This driver is worth reading, but it is not worth describing here.

4.3 fmc-write-eeprom

This module is designed to load a binary file from /lib/firmware and to write it to the internal EEPROM of the mezzanine card. This driver uses the busid generic parameter.

Overwriting the EEPROM is not something you should do daily, and it is expected to only happen during manufacturing. For this reason, the module makes it unlikely for the random user to change a working EEPROM.

The module takes the following measures:
- It accepts a `file` argument (within `/lib/firmware`) and if no such argument is received, it doesn’t write anything to EEPROM (i.e. there is no default file name).
- If the file name ends with `.bin` it is written verbatim starting at offset 0.
- If the file name ends with `.tlv` it is interpreted as type-length-value (i.e., it allows `writev(2)`-like operation).
- If the file name doesn’t match any of the patterns above, it is ignored and no write is performed.
- Only cards listed with `busid` are written to. If no `busid` is specified, no programming is done (and the probe function of the driver will fail).

Each TLV tuple is formatted in this way: the header is 5 bytes, followed by data. The first byte is `w` for `write`, the next two bytes represent the address, in little-endian byte order, and the next two represent the data length, in little-endian order. The length does not include the header (it is the actual number of bytes to be written).

This is a real example: that writes 5 bytes at position 0x110:

```
spusa.root# od -t x1 -Ax /lib/firmware/try.tlv
000000 77 10 01 05 00 30 31 32 33 34 00000a
spusa.root# insmod /tmp/fmc-write-eeprom.ko busid=0x0200 file=try.tlv
[19983.391498] spec 0000:03:00.0: write 5 bytes at 0x0110
[19983.414615] spec 0000:03:00.0: write_eeprom: success
```

Please note that you’ll most likely want to use SDBFS to build your EEPROM image, at least if your mezzanines are being used in the White Rabbit environment. For this reason the TLV format is not expected to be used much and is not expected to be developed further.