FMC Bus Abstraction for Linux

July 2012

Implementing a bus abstraction for fmc mezzanines

Alessandro Rubini for CERN (BE-CO-HT)

Table of Contents

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.

Currently, code and documentation for the FMC bus is part of the *spec-sw* project at `ohwr.org`.

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 FMC Device

Within this 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 drivers, 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).

This is the current structure defining a device. Please note that stuff is still being defined as I write this, so the structures are going to change (if in doubt, please check the header file in the repository rather than this document):

```
struct fmc_device {
        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;
};
```

3 FMC Driver

Within this framework the FMC driver 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 is needed for high-bandwidth peripherals like fast ADC cards. The `base` pointer is NULL for remote devices, and the driver will refuse to work with them if it needs direct access.

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.

4 Functions Exported by the Core

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

```
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);
```

They should be self-explicative, so nothing is added here.

5 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):

```
struct fmc_operations {
        uint32_t (*readl)(struct fmc_device *fmc, int offset);
        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 (*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, as most of the time they will just make a memory access to the host bus. If the carrier provided a *base* pointer, the driver may use direct access through it instead. For this reason the header offers the inline functions *fmc_readl* and *fmc_writel* that access *base* if respective method is NULL. 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 uses the *bus_id* module parameter (if provided) and returns the matching index in the array. If no match is found, `-ENOENT` is returned; if the argument has not been specified, all devices match the driver and 0 is returned. 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 currently documented in the *spec-sw* manual; this *validate* method is on show in `spec-fmc.c` and it is used by `fmc-trivial.c`.

`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 will 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.

`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, but it's only informative for the mezzanine driver. The handler will receive the *fmc* pointer as *dev_id*; the *flags* argument is still to be defined.

`read_ee`
`write_ee`

Read or write the EEPROM. The functions are expected to be only called before reprogramming and the carrier will refuse them with `ENODEV` after reprogramming. The offset is limited to 8kB 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 own I2C memory is seen at offset 1M and the internal SPI flash is seen at offset 16M. This is carrier-specific and should only be used by the driver after checking the `carrier_name` field.