MEN Chameleon Bus

Introduction

This document describes the architecture and implementation of the MEN Chameleon Bus (called MCB throughout this document).

Scope of this Document

This document is intended to be a short overview of the current implementation and does by no means describe the complete possibilities of MCB based devices.

Limitations of the current implementation

The current implementation is limited to PCI and PCIe based carrier devices that only use a single memory resource and share the PCI legacy IRQ. Not implemented are:

- Multi-resource MCB devices like the VME Controller or M-Module carrier.
- MCB devices that need another MCB device, like SRAM for a DMA Controller's buffer descriptors or a video controller's video memory.
- A per-carrier IRQ domain for carrier devices that have one (or more) IRQs per MCB device like PCIe based carriers with MSI or MSI-X support.

Architecture

MCB is divided into 3 functional blocks:

- The MEN Chameleon Bus itself,
- drivers for MCB Carrier Devices and
- the parser for the Chameleon table.

MEN Chameleon Bus

The MEN Chameleon Bus is an artificial bus system that attaches to a so called Chameleon FPGA device found on some hardware produced my MEN Mikro Elektronik GmbH. These devices are multi-function devices implemented in a single FPGA and usually attached via some sort of PCI or PCIe link. Each FPGA contains a header section describing the content of the FPGA. The header lists the device id, PCI BAR, offset from the beginning of the PCI BAR, size in the FPGA, interrupt number and some other properties currently not handled by the MCB implementation.

Carrier Devices

A carrier device is just an abstraction for the real world physical bus the Chameleon FPGA is attached to. Some IP Core drivers may need to interact with properties of the carrier device (like querying the IRQ number of a PCI device). To provide abstraction from the real hardware bus, an MCB carrier device provides callback methods to translate the driver's MCB function calls to hardware related function calls. For example a carrier device may implement the get_irq() method which can be translated into a hardware bus query for the IRQ number the device should use.

Parser

The parser reads the first 512 bytes of a Chameleon device and parses the Chameleon table. Currently the parser only supports the Chameleon v2 variant of the Chameleon table but can easily be adopted to support an older or possible future variant. While parsing the table's entries new MCB devices are allocated and their resources are assigned according to the resource assignment in the Chameleon table. After resource assignment is finished, the MCB devices are registered at the MCB and thus at the driver core of the Linux kernel.

Resource handling

The current implementation assigns exactly one memory and one IRQ resource per MCB device. But this is likely going to change in the future.

Memory Resources

Each MCB device has exactly one memory resource, which can be requested from the MCB bus. This memory resource is the physical address of the MCB device inside the carrier and is intended to be passed to ioremap() and friends. It is already requested from the kernel by calling request_mem_region().

IRQs

Each MCB device has exactly one IRQ resource, which can be requested from the MCB bus. If a carrier device driver implements the ->get_irq() callback method, the IRQ number assigned by the carrier device will be returned, otherwise the IRQ number inside the Chameleon table will be returned. This number is suitable to be passed to request_irq().

Writing an MCB driver

The driver structure

Each MCB driver has a structure to identify the device driver as well as device ids which identify the IP Core inside the FPGA. The driver structure also contains callback methods which get executed on driver probe and removal from the system:

```
static const struct mcb_device_id foo_ids[] = {
        { .device = 0x123 },
        { }
};
MODULE_DEVICE_TABLE(mcb, foo_ids);

static struct mcb_driver foo_driver = {
driver = {
        .name = "foo-bar",
        .owner = THIS_MODULE,
},
        .probe = foo_probe,
        .remove = foo_remove,
        .id_table = foo_ids,
};
```

Probing and attaching

When a driver is loaded and the MCB devices it services are found, the MCB core will call the driver's probe callback method. When the driver is removed from the system, the MCB core will call the driver's remove callback method:

```
static init foo_probe(struct mcb_device *mdev, const struct mcb_device_id *id);
static void foo_remove(struct mcb_device *mdev);
```

Initializing the driver

When the kernel is booted or your foo driver module is inserted, you have to perform driver initialization. Usually it is enough to register your driver module at the MCB core:

```
static int __init foo_init(void)
{
        return mcb_register_driver(&foo_driver);
}
module_init(foo_init);

static void __exit foo_exit(void)
{
        mcb_unregister_driver(&foo_driver);
}
module_exit(foo_exit);
```

The module_mcb_driver() macro can be used to reduce the above code:

```
module_mcb_driver(foo_driver);
```

Using DMA

To make use of the kernel's DMA-API's function, you will need to use the carrier device's 'struct device'. Fortunately 'struct mcb_device' embeds a pointer (->dma_dev) to the carrier's device for DMA purposes:

```
ret = dma_set_mask_and_coherent(&mdev->dma_dev, DMA_BIT_MASK(dma_bits));
if (rc)
        /* Handle errors */
```