

Writing an MUSB Glue Layer

Apelete Seketeli <apelete at seketeli.net>

Writing an MUSB Glue Layer

by Apelete Seketeli

Copyright © 2014 Apelete Seketeli

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this documentation; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the Linux kernel source tree.

Table of Contents

1. Introduction	1
2. Linux MUSB Basics	2
3. Handling IRQs	8
4. Device Platform Data	10
5. Device Quirks	13
6. Conclusion	15
7. Acknowledgements	16
8. Resources	17

Chapter 1. Introduction

The Linux MUSB subsystem is part of the larger Linux USB subsystem. It provides support for embedded USB Device Controllers (UDC) that do not use Universal Host Controller Interface (UHCI) or Open Host Controller Interface (OHCI).

Instead, these embedded UDC rely on the USB On-the-Go (OTG) specification which they implement at least partially. The silicon reference design used in most cases is the Multipoint USB Highspeed Dual-Role Controller (MUSB HDRC) found in the Mentor Graphics Inventra™ design.

As a self-taught exercise I have written an MUSB glue layer for the Ingenic JZ4740 SoC, modelled after the many MUSB glue layers in the kernel source tree. This layer can be found at `drivers/usb/musb/jz4740.c`. In this documentation I will walk through the basics of the `jz4740.c` glue layer, explaining the different pieces and what needs to be done in order to write your own device glue layer.


```
    }

    ret = clk_prepare_enable(clk);
    if (ret) {
        dev_err(&pdev->dev, "failed to enable clock\n");
        goto err_platform_device_put;
    }

    musb->dev.parent = &pdev->dev;

    glue->dev = &pdev->dev;
    glue->musb = musb;
    glue->clk = clk;

    return 0;

err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

The first few lines of the probe function allocate and assign the glue, musb and clk variables. The GFP_KERNEL flag (line 8) allows the allocation process to sleep and wait for memory, thus being usable in a blocking situation. The PLATFORM_DEVID_AUTO flag (line 12) allows automatic allocation and management of device IDs in order to avoid device namespace collisions with explicit IDs. With devm_clk_get() (line 18) the glue layer allocates the clock -- the devm_ prefix indicates that clk_get() is managed: it automatically frees the allocated clock resource data when the device is released -- and enable it.

Then comes the registration steps:

```
static int jz4740_probe(struct platform_device *pdev)
{
    struct musb_hdrc_platform_data *pdata = &jz4740_musb_platform_data;

    pdata->platform_ops = &jz4740_musb_ops;

    platform_set_drvdata(pdev, glue);

    ret = platform_device_add_resources(musb, pdev->resource,
                                       pdev->num_resources);
    if (ret) {
        dev_err(&pdev->dev, "failed to add resources\n");
        goto err_clk_disable;
    }

    ret = platform_device_add_data(musb, pdata, sizeof(*pdata));
    if (ret) {
        dev_err(&pdev->dev, "failed to add platform_data\n");
        goto err_clk_disable;
    }
}
```


controller hardware responsible for sending/receiving the USB data. Since it is an implementation of the physical layer of the OSI model, the transceiver is also referred to as PHY.

Getting hold of the MUSB PHY driver data is done with `usb_get_phy()` which returns a pointer to the structure containing the driver instance data. The next couple of instructions (line 12 and 14) are used as a quirk and to setup IRQ handling respectively. Quirks and IRQ handling will be discussed later in Chapter 5 and Chapter 3.

```
static int jz4740_musb_exit(struct musb *musb)
{
    usb_put_phy(musb->xceiv);

    return 0;
}
```

Acting as the counterpart of `init`, the `exit` function releases the MUSB PHY driver when the controller hardware itself is about to be released.

Again, note that `init` and `exit` are fairly simple in this case due to the basic set of features of the JZ4740 controller hardware. When writing an `musb` glue layer for a more complex controller hardware, you might need to take care of more processing in those two functions.

Returning from the `init` function, the MUSB controller driver jumps back into the probe function:

```
static int jz4740_probe(struct platform_device *pdev)
{
    ret = platform_device_add(musb);
    if (ret) {
        dev_err(&pdev->dev, "failed to register musb device\n");
        goto err_clk_disable;
    }

    return 0;

err_clk_disable:
    clk_disable_unprepare(clk);
err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

This is the last part of the device registration process where the glue layer adds the controller hardware device to Linux kernel device hierarchy: at this stage, all known information about the device is passed on to the Linux USB core stack.

```
static int jz4740_remove(struct platform_device *pdev)
{
    struct jz4740_glue *glue = platform_get_drvdata(pdev);

    platform_device_unregister(glue->musb);
}
```

```
    clk_disable_unprepare(glue->clk);  
  
    return 0;  
}
```

Acting as the counterpart of probe, the remove function unregister the MUSB controller hardware (line 5) and disable the clock (line 6), allowing it to be gated.

Instruction on line 18 is another quirk specific to the JZ4740 USB device controller, which will be discussed later in Chapter 5.

The glue layer still needs to register the IRQ handler though. Remember the instruction on line 14 of the init function:

```
static int jz4740_musb_init(struct musb *musb)
{
    musb->isr = jz4740_musb_interrupt;

    return 0;
}
```

This instruction sets a pointer to the glue layer IRQ handler function, in order for the controller hardware to call the handler back when an IRQ comes from the controller hardware. The interrupt handler is now implemented and registered.

by reading the relevant controller hardware registers. This issue will be discussed when we get to device quirks in Chapter 5. Last two fields (line 8 and 9) are also about device quirks: `fifo_cfg` points to the USB endpoints configuration table and `fifo_cfg_size` keeps track of the size of the number of entries in that configuration table. More on that later in Chapter 5.

Then this configuration is embedded inside `jz4740_musb_platform_data` `musb_hdrc_platform_data` structure (line 11): `config` is a pointer to the configuration structure itself, and `mode` tells the controller driver if the controller hardware may be used as `MUSB_HOST` only, `MUSB_PERIPHERAL` only or `MUSB_OTG` which is a dual mode.

Remember that `jz4740_musb_platform_data` is then used to convey platform data information as we have seen in the probe function in Chapter 2

Chapter 6. Conclusion

Writing a Linux MUSB glue layer should be a more accessible task, as this documentation tries to show the ins and outs of this exercise.

The JZ4740 USB device controller being fairly simple, I hope its glue layer serves as a good example for the curious mind. Used with the current MUSB glue layers, this documentation should provide enough guidance to get started; should anything gets out of hand, the linux-usb mailing list archive is another helpful resource to browse through.

Chapter 7. Acknowledgements

Many thanks to Lars-Peter Clausen and Maarten ter Huurne for answering my questions while I was writing the JZ4740 glue layer and for helping me out getting the code in good shape.

I would also like to thank the Qi-Hardware community at large for its cheerful guidance and support.

Chapter 8. Resources

USB Home Page: <http://www.usb.org>

linux-usb Mailing List Archives: <http://marc.info/?l=linux-usb>

USB On-the-Go Basics: <http://www.maximintegrated.com/app-notes/index.mvp/id/1822>

Writing USB Device Drivers: https://www.kernel.org/doc/html/docs/writing_usb_driver/index.html

Texas Instruments USB Configuration Wiki Page: <http://processors.wiki.ti.com/index.php/Usbgeneralpage>

Analog Devices Blackfin MUSB Configuration: <http://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:drivers:musb>