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).

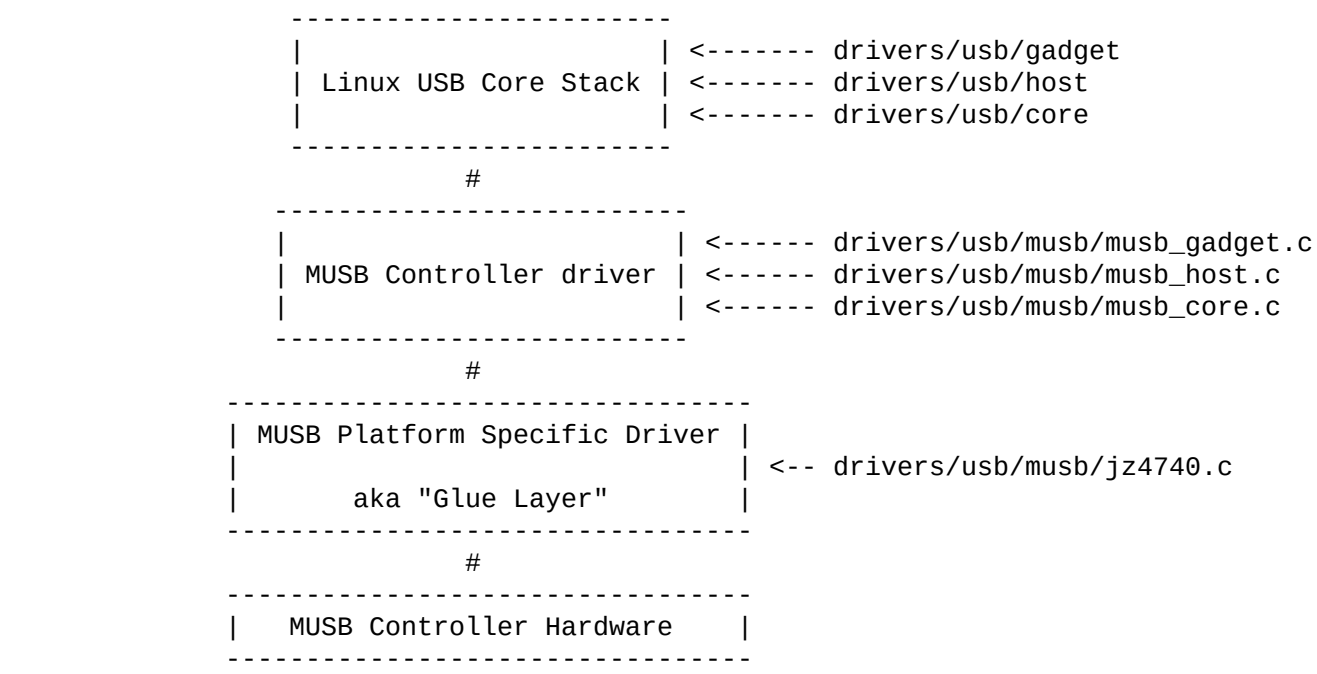
Instead, these embedded UDC rely on the USB On-the-Go (OTG) specification which they implement at least partially. The silicon reference design used in most cases is the Multipoint USB Highspeed Dual-Role Controller (MUSB HDRC) found in the Mentor Graphics Inventra™ design.

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.

Chapter 2. Linux MUSB Basics

To get started on the topic, please read USB On-the-Go Basics (see Resources) which provides an introduction of USB OTG operation at the hardware level. A couple of wiki pages by Texas Instruments and Analog Devices also provide an overview of the Linux kernel MUSB configuration, albeit focused on some specific devices provided by these companies. Finally, getting acquainted with the USB specification at USB home page may come in handy, with practical instance provided through the Writing USB Device Drivers documentation (again, see Resources).

Linux USB stack is a layered architecture in which the MUSB controller hardware sits at the lowest. The MUSB controller driver abstract the MUSB controller hardware to the Linux USB stack.



As outlined above, the glue layer is actually the platform specific code sitting in between the controller driver and the controller hardware.

Just like a Linux USB driver needs to register itself with the Linux USB subsystem, the MUSB glue layer needs first to register itself with the MUSB controller driver. This will allow the controller driver to know about which device the glue layer supports and which functions to call when a supported device is detected or released; remember we are talking about an embedded controller chip here, so no insertion or removal at run-time.

All of this information is passed to the MUSB controller driver through a platform_driver structure defined in the glue layer as:

```
static struct platform_driver jz4740_driver = {
 .probe  = jz4740_probe,
 .remove  = jz4740_remove,
 .driver  = {
  .name = "musb-jz4740",
```

```
 },
};
```

The probe and remove function pointers are called when a matching device is detected and, respectively, released. The name string describes the device supported by this glue layer. In the current case it matches a platform_device structure declared in arch/mips/jz4740/platform.c. Note that we are not using device tree bindings here.

In order to register itself to the controller driver, the glue layer goes through a few steps, basically allocating the controller hardware resources and initialising a couple of circuits. To do so, it needs to keep track of the information used throughout these steps. This is done by defining a private jz4740_glue structure:

```
struct jz4740_glue {
 struct device           *dev;
 struct platform_device  *musb;
 struct clk  *clk;
};
```

The dev and musb members are both device structure variables. The first one holds generic information about the device, since it's the basic device structure, and the latter holds information more closely related to the subsystem the device is registered to. The clk variable keeps information related to the device clock operation.

Let's go through the steps of the probe function that leads the glue layer to register itself to the controller driver.

N.B.: For the sake of readability each function will be split in logical parts, each part being shown as if it was independent from the others.

```
static int jz4740_probe(struct platform_device *pdev)
{
 struct platform_device  *musb;
 struct jz4740_glue  *glue;
 struct clk                     *clk;
 int    ret;

 glue = devm_kzalloc(&pdev->dev, sizeof(*glue), GFP_KERNEL);
 if (!glue)
  return -ENOMEM;

 musb = platform_device_alloc("musb-hdrc", PLATFORM_DEVID_AUTO);
 if (!musb) {
  dev_err(&pdev->dev, "failed to allocate musb device\n");
  return -ENOMEM;
 }

 clk = devm_clk_get(&pdev->dev, "udc");
 if (IS_ERR(clk)) {
  dev_err(&pdev->dev, "failed to get clock\n");
  ret = PTR_ERR(clk);
  goto err_platform_device_put;
```

```
 }

 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;
 }
```

```
 return 0;

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

The first step is to pass the device data privately held by the glue layer on to the controller driver through platform_set_drvdata() (line 7). Next is passing on the device resources information, also privately held at that point, through platform_device_add_resources() (line 9).

Finally comes passing on the platform specific data to the controller driver (line 16). Platform data will be discussed in Chapter 4, but here we are looking at the platform_ops function pointer (line 5) in musb_hdrc_platform_data structure (line 3). This function pointer allows the MUSB controller driver to know which function to call for device operation:

```
static const struct musb_platform_ops jz4740_musb_ops = {
 .init  = jz4740_musb_init,
 .exit  = jz4740_musb_exit,
};
```

Here we have the minimal case where only init and exit functions are called by the controller driver when needed. Fact is the JZ4740 MUSB controller is a basic controller, lacking some features found in other controllers, otherwise we may also have pointers to a few other functions like a power management function or a function to switch between OTG and non-OTG modes, for instance.

At that point of the registration process, the controller driver actually calls the init function:

```
static int jz4740_musb_init(struct musb *musb)
{
 musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
 if (!musb->xceiv) {
  pr_err("HS UDC: no transceiver configured\n");
  return -ENODEV;
 }

 /* Silicon does not implement ConfigData register.
  * Set dyn_fifo to avoid reading EP config from hardware.
  */
 musb->dyn_fifo = true;

 musb->isr = jz4740_musb_interrupt;

 return 0;
}
```

The goal of jz4740_musb_init() is to get hold of the transceiver driver data of the MUSB controller hardware and pass it on to the MUSB controller driver, as usual. The transceiver is the circuitry inside the

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.

Chapter 3. Handling IRQs

Additionally to the MUSB controller hardware basic setup and registration, the glue layer is also responsible for handling the IRQs:

```
static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
{
 unsigned long   flags;
 irqreturn_t     retval = IRQ_NONE;
 struct musb     *musb = __hci;

 spin_lock_irqsave(&musb->lock, flags);

 musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
 musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
 musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

 /*
  * The controller is gadget only, the state of the host mode IRQ bits is
  * undefined. Mask them to make sure that the musb driver core will
  * never see them set
  */
 musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
     MUSB_INTR_RESET | MUSB_INTR_SOF;

 if (musb->int_usb || musb->int_tx || musb->int_rx)
  retval = musb_interrupt(musb);

 spin_unlock_irqrestore(&musb->lock, flags);

 return retval;
}
```

Here the glue layer mostly has to read the relevant hardware registers and pass their values on to the controller driver which will handle the actual event that triggered the IRQ.

The interrupt handler critical section is protected by the spin_lock_irqsave() and counterpart spin_unlock_irqrestore() functions (line 7 and 24 respectively), which prevent the interrupt handler code to be run by two different threads at the same time.

Then the relevant interrupt registers are read (line 9 to 11):

- MUSB_INTRUSB: indicates which USB interrupts are currently active,
- MUSB_INTRTX: indicates which of the interrupts for TX endpoints are currently active,
- MUSB_INTRRX: indicates which of the interrupts for TX endpoints are currently active.

Note that musb_readb() is used to read 8-bit registers at most, while musb_readw() allows us to read at most 16-bit registers. There are other functions that can be used depending on the size of your device registers. See musb_io.h for more information.

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.

Chapter 4. Device Platform Data

In order to write an MUSB glue layer, you need to have some data describing the hardware capabilities of your controller hardware, which is called the platform data.

Platform data is specific to your hardware, though it may cover a broad range of devices, and is generally found somewhere in the arch/ directory, depending on your device architecture.

For instance, platform data for the JZ4740 SoC is found in arch/mips/jz4740/platform.c. In the platform.c file each device of the JZ4740 SoC is described through a set of structures.

Here is the part of arch/mips/jz4740/platform.c that covers the USB Device Controller (UDC):

```
/* USB Device Controller */
struct platform_device jz4740_udc_xceiv_device = {
 .name = "usb_phy_gen_xceiv",
 .id   = 0,
};

static struct resource jz4740_udc_resources[] = {
 [0] = {
  .start = JZ4740_UDC_BASE_ADDR,
  .end   = JZ4740_UDC_BASE_ADDR + 0x10000 - 1,
  .flags = IORESOURCE_MEM,
 },
 [1] = {
  .start = JZ4740_IRQ_UDC,
  .end   = JZ4740_IRQ_UDC,
  .flags = IORESOURCE_IRQ,
  .name  = "mc",
 },
};

struct platform_device jz4740_udc_device = {
 .name = "musb-jz4740",
 .id   = -1,
 .dev  = {
  .dma_mask          = &jz4740_udc_device.dev.coherent_dma_mask,
  .coherent_dma_mask = DMA_BIT_MASK(32),
 },
 .num_resources = ARRAY_SIZE(jz4740_udc_resources),
 .resource      = jz4740_udc_resources,
};
```

The jz4740_udc_xceiv_device platform device structure (line 2) describes the UDC transceiver with a name and id number.

At the time of this writing, note that "usb_phy_gen_xceiv" is the specific name to be used for all transceivers that are either built-in with reference USB IP or autonomous and doesn't require any PHY programming. You will need to set CONFIG_NOP_USB_XCEIV=y in the kernel configuration to make use of the corresponding transceiver driver. The id field could be set to -1 (equivalent to

PLATFORM_DEVID_NONE), -2 (equivalent to PLATFORM_DEVID_AUTO) or start with 0 for the first device of this kind if we want a specific id number.

The jz4740_udc_resources resource structure (line 7) defines the UDC registers base addresses.

The first array (line 9 to 11) defines the UDC registers base memory addresses: start points to the first register memory address, end points to the last register memory address and the flags member defines the type of resource we are dealing with. So IORESOURCE_MEM is used to define the registers memory addresses. The second array (line 14 to 17) defines the UDC IRQ registers addresses. Since there is only one IRQ register available for the JZ4740 UDC, start and end point at the same address. The IORESOURCE_IRQ flag tells that we are dealing with IRQ resources, and the name "mc" is in fact hard-coded in the MUSB core in order for the controller driver to retrieve this IRQ resource by querying it by its name.

Finally, the jz4740_udc_device platform device structure (line 21) describes the UDC itself.

The "musb-jz4740" name (line 22) defines the MUSB driver that is used for this device; remember this is in fact the name that we used in the jz4740_driver platform driver structure in Chapter 2. The id field (line 23) is set to -1 (equivalent to PLATFORM_DEVID_NONE) since we do not need an id for the device: the MUSB controller driver was already set to allocate an automatic id in Chapter 2. In the dev field we care for DMA related information here. The dma_mask field (line 25) defines the width of the DMA mask that is going to be used, and coherent_dma_mask (line 26) has the same purpose but for the alloc_coherent DMA mappings: in both cases we are using a 32 bits mask. Then the resource field (line 29) is simply a pointer to the resource structure defined before, while the num_resources field (line 28) keeps track of the number of arrays defined in the resource structure (in this case there were two resource arrays defined before).

With this quick overview of the UDC platform data at the arch/ level now done, let's get back to the MUSB glue layer specific platform data in drivers/usb/musb/jz4740.c:

```
static struct musb_hdrc_config jz4740_musb_config = {
 /* Silicon does not implement USB OTG. */
 .multipoint = 0,
 /* Max EPs scanned, driver will decide which EP can be used. */
 .num_eps    = 4,
 /* RAMbits needed to configure EPs from table */
 .ram_bits   = 9,
 .fifo_cfg = jz4740_musb_fifo_cfg,
 .fifo_cfg_size = ARRAY_SIZE(jz4740_musb_fifo_cfg),
};

static struct musb_hdrc_platform_data jz4740_musb_platform_data = {
 .mode   = MUSB_PERIPHERAL,
 .config = &jz4740_musb_config,
};
```

First the glue layer configures some aspects of the controller driver operation related to the controller hardware specifics. This is done through the jz4740_musb_config musb_hdrc_config structure.

Defining the OTG capability of the controller hardware, the multipoint member (line 3) is set to 0 (equivalent to false) since the JZ4740 UDC is not OTG compatible. Then num_eps (line 5) defines the number of USB endpoints of the controller hardware, including endpoint 0: here we have 3 endpoints + endpoint 0. Next is ram_bits (line 7) which is the width of the RAM address bus for the MUSB controller hardware. This information is needed when the controller driver cannot automatically configure endpoints

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 5. Device Quirks

Completing the platform data specific to your device, you may also need to write some code in the glue layer to work around some device specific limitations. These quirks may be due to some hardware bugs, or simply be the result of an incomplete implementation of the USB On-the-Go specification.

The JZ4740 UDC exhibits such quirks, some of which we will discuss here for the sake of insight even though these might not be found in the controller hardware you are working on.

Let's get back to the init function first:

```
static int jz4740_musb_init(struct musb *musb)
{
 musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
 if (!musb->xceiv) {
  pr_err("HS UDC: no transceiver configured\n");
  return -ENODEV;
 }

 /* Silicon does not implement ConfigData register.
  * Set dyn_fifo to avoid reading EP config from hardware.
  */
 musb->dyn_fifo = true;

 musb->isr = jz4740_musb_interrupt;

 return 0;
}
```

Instruction on line 12 helps the MUSB controller driver to work around the fact that the controller hardware is missing registers that are used for USB endpoints configuration.

Without these registers, the controller driver is unable to read the endpoints configuration from the hardware, so we use line 12 instruction to bypass reading the configuration from silicon, and rely on a hard-coded table that describes the endpoints configuration instead:

```
static struct musb_fifo_cfg jz4740_musb_fifo_cfg[] = {
{ .hw_ep_num = 1, .style = FIFO_TX, .maxpacket = 512, },
{ .hw_ep_num = 1, .style = FIFO_RX, .maxpacket = 512, },
{ .hw_ep_num = 2, .style = FIFO_TX, .maxpacket = 64, },
};
```

Looking at the configuration table above, we see that each endpoints is described by three fields: hw_ep_num is the endpoint number, style is its direction (either FIFO_TX for the controller driver to send packets in the controller hardware, or FIFO_RX to receive packets from hardware), and maxpacket defines the maximum size of each data packet that can be transmitted over that endpoint. Reading from the table, the controller driver knows that endpoint 1 can be used to send and receive USB data packets of 512 bytes at once (this is in fact a bulk in/out endpoint), and endpoint 2 can be used to send data packets of 64 bytes at once (this is in fact an interrupt endpoint).

Note that there is no information about endpoint 0 here: that one is implemented by default in every silicon design, with a predefined configuration according to the USB specification. For more examples of endpoint configuration tables, see musb_core.c.

Let's now get back to the interrupt handler function:

```
static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
{
 unsigned long   flags;
 irqreturn_t     retval = IRQ_NONE;
 struct musb     *musb = __hci;

 spin_lock_irqsave(&musb->lock, flags);

 musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
 musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
 musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

 /*
  * The controller is gadget only, the state of the host mode IRQ bits is
  * undefined. Mask them to make sure that the musb driver core will
  * never see them set
  */
 musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
     MUSB_INTR_RESET | MUSB_INTR_SOF;

 if (musb->int_usb || musb->int_tx || musb->int_rx)
  retval = musb_interrupt(musb);

 spin_unlock_irqrestore(&musb->lock, flags);

 return retval;
}
```

Instruction on line 18 above is a way for the controller driver to work around the fact that some interrupt bits used for USB host mode operation are missing in the MUSB_INTRUSB register, thus left in an undefined hardware state, since this MUSB controller hardware is used in peripheral mode only. As a consequence, the glue layer masks these missing bits out to avoid parasite interrupts by doing a logical AND operation between the value read from MUSB_INTRUSB and the bits that are actually implemented in the register.

These are only a couple of the quirks found in the JZ4740 USB device controller. Some others were directly addressed in the MUSB core since the fixes were generic enough to provide a better handling of the issues for others controller hardware eventually.

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/htmldocs/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