國立臺灣大學電子工程學研究所

碩士論文

Graduate Institute of Electronic Engineering

National Taiwan University

Master Thesis

支援 Xilinx AXI DMA 的 Linux UIO 驅動程式

Linux UIO driver for Xilinx AXI DMA

劉宇唐

Yu-Tang Liu

指導教授：鄭振牟 博士

Advisor: Chen-Mou Cheng, Ph.D.

中華民國 107 年 7 月

July 2018

摘要

近年來，由於 AI、VR 產業的崛起，FPGA 產業越來越受到重視。為了簡化 FPGA 的開發流程，使用嵌入式 Linux 會是一個不錯的方法。透過 Linux Kernel 提供的 UIO 驅動程式，我們可以把我們在硬體端設計出來的 IP 視為一個外部裝置，然後在 Linux 使用者空間裡的程式中，輕鬆地開發軟體端的應用。然而，有些硬體端的設計，卻無法透過同樣的方法，利用 UIO 驅動程式，建立裝置節點，而帶有直接記憶體存取 IP 的設計就是其中之一。由於 UIO 驅動程式並無法支援此種設計，我們必須擁有"root" 權限，才能使用我們的設計，但是提供"root" 給一般使用者並不是一個好方法。在此論文中，我們修改了 Linux 內建的 UIO 驅動程式，使得一般用戶也能在使用者空間中使用帶有 DMA 的硬體設計。

關鍵字: 賽靈思，直接記憶體存取，*AXI*，*Linux UIO* 驅動程式

Abstract

In recent year, increasingly importance has been attached to FPGAs with the development of AI,VR. To simplify the development process on FPGAs, embedded Linux on FPGAs will be a good way. With UIO driver provided in Linux Kernel, we can mount our block design, that is, custom IP(Intellectual Property) core in Vivado as a device node, and program it in Linux user space. However, there are some designs that UIO driver cannot recognizes. The design with DMAs(Direct Memory Access) is the one of them. With this kind of design, because UIO driver is not work, we need ”root” to control our IP, and providing root privileges to users is never a good solution. In this thesis, we modifiy UIO driver so that users can easily use designs with DMA in user-space.

Keywords: *Xilinx, DMA, AXI, Linux UIO driver*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

FPGA(Field Programmable Gate Array) is

1.1 Motivation

1.2 Contribution

We propose a develop flow to use AXI-Stream IP, with a little modified of UIO driver and little trick setting in device tree file. We also provide functions called **read()/write()** which send DMA transactions to DMA controller. Whole scenario is very simple and intuitive, and is not large different from normal flow. The efficiency is also nice,

Chapter 2

Preliminaries

In this chapter, we introduce the background technology for our work. Embedded Linux, UIO Driver, AXI Bus, and DMA.

2.1 Embedded Linux

Embedded Linux is a kernel and set of libraries and utilitied designed to run on an embedded system(for example:router). Figure 2.1 shows the stages of booting Linux on the target platform. For example, when turns on the FPGA, the board will boot ROM and find the boot mode setting, then load FSBL(First Stage Bootloader), which will load bitstream to initialize the PL side on FPGA. Then the FGPA will load the SSBL(Second Stage Bootloader), here we use u-boot for demonstration. The main purpose of u-boot is to load Linux Kernel, it loads kernel image with ***devicetree file*** of the target platform. With well prepared file system(e.g ramdiskfs), the OS should run up successfully.

2.1.1 Device Tree

Device Tree is a mechanism to describe all hardware and devices of a system. In early Linux kernel, hardware description is hardcode in kernel file, so porting kernel to different ARM-CPU based system is painful. To solve this problem, Device Tree is introduced. Like x86 based system, we should consider Linux kernel image is a black box, and give the hardware informations of system to kernel.

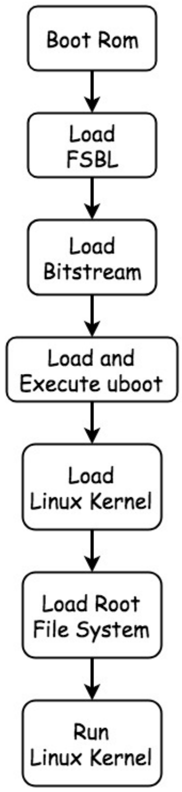

Figure 2.1: Linux Boot Stage on Target Platform

In FPGA development flow, the whole system almost keeps the same, the only thing that might change is our design in PL(Programmable Logic) side. To boot Linux with different PL design, only a little modification of devicetree file is needed.

2.1.2 Linux Kernel Driver

2.2 UIO

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. To address this situation, the userspace I/O system (UIO) was designed.Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped.
- The device can be controlled completely by writing to this memory.

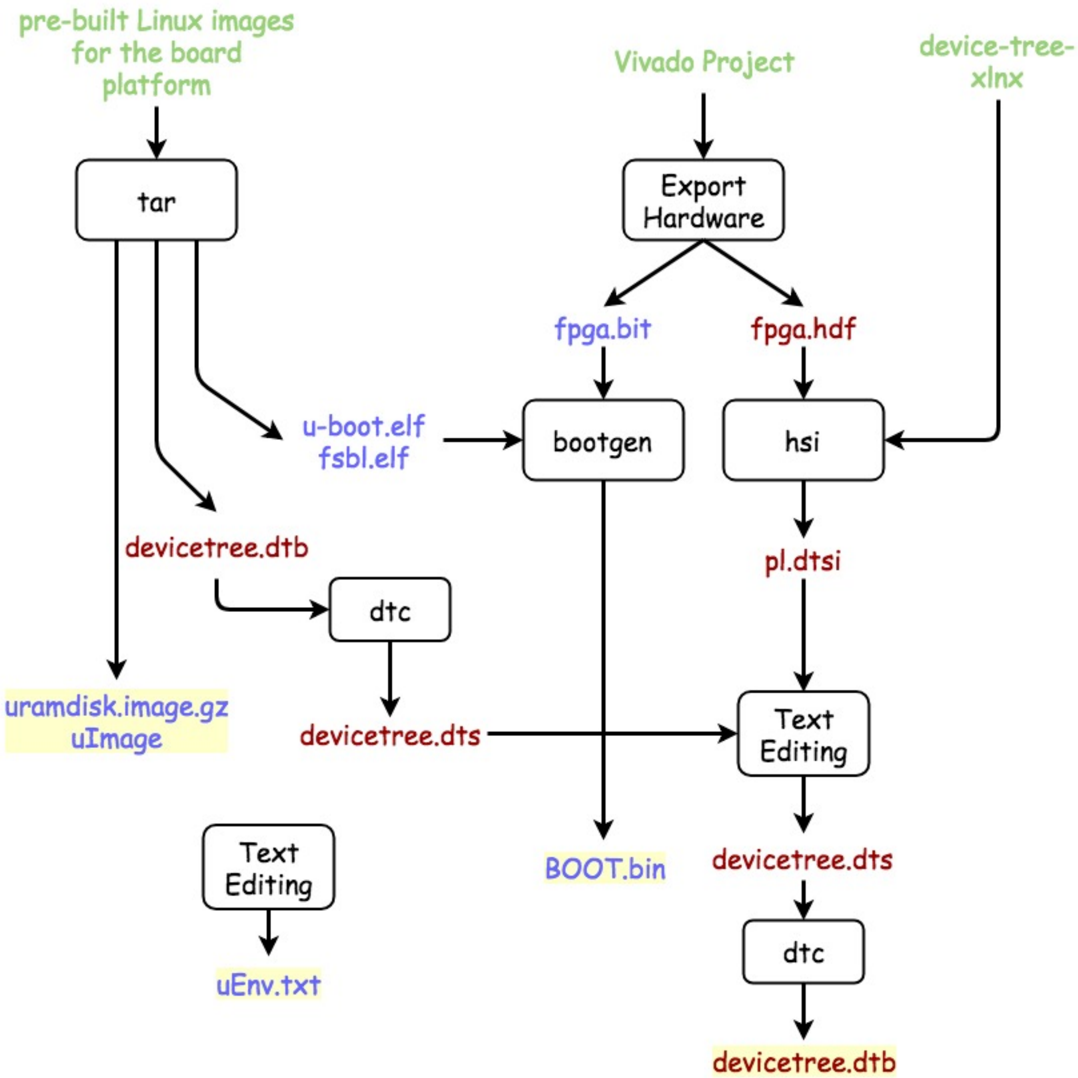

Figure 2.2: Embedded Linux on FPGA

- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

Figure 2.3 shows how the UIO system works, in software-side of FPGA development, we only care about the value in the hardware register and when we can get the correct value, so memeory -mapping to user-spcae application and interrupt handler is realy enough in our design flow.

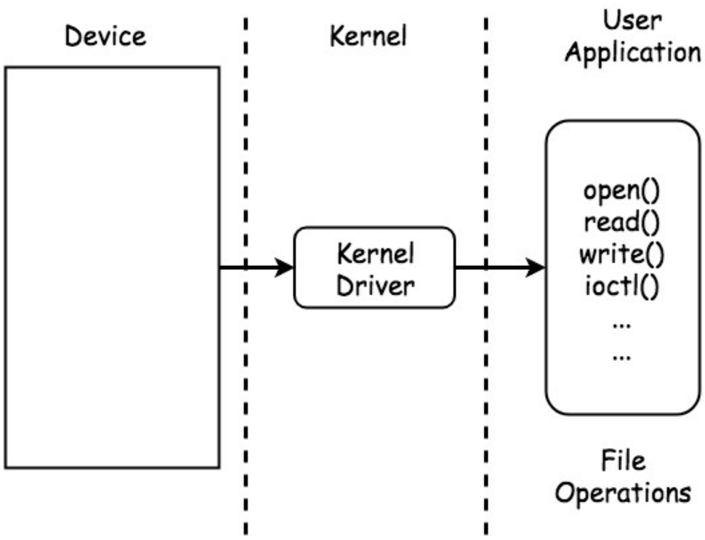

Figure 2.3: Linux Kernel Driver

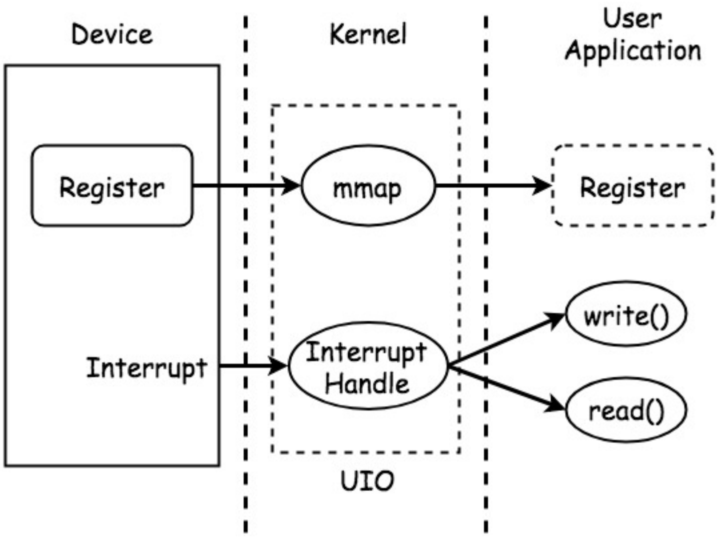

Figure 2.4: The UIO way.

2.3 AXI Bus

Advanced eXtensible Interface (AXI) protocol is part of ARM AMBA(Advanced Micro-controller Bus Architecture). It is a on-chip bus interface that is targeted at high performance, high clock frequency system designs and includes features that make it suitable for high speed sub-micrometer interconnect:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes

- burst based transactions with only start address issued
- issuing of multiple outstanding addresses with out of order responses
- easy addition of register stages to provide timing closure.

2.3.1 AMBA

2.3.2 AXI4

AXI4

- AXI4:
- AXI4-Lite:
- AXI4-Stream:

2.4 DMA

DMA(Direct Memory Access) is a feature that allows hardware subsystems to access main system memory independent of CPU. For example, when CPU want to submit a DMA transaction, it needs to give DMA controller where the data is (memory address), and size of the data(data length). After submit, CPU can back to work for other task, once the transaction is done, CPU will receive a interrupt from DMA controller.

2.4.1 DMA Engine

Chapter 3

Proposed solution

Now we can finally conclude why UIO driver doesn't work on some IPs with DMA, because AXI-Stream is a special bus protocol which is not compatible with AXI4 bus protocol. Figure 3.1 and Figure 3.2 show the difference of two designs. So, if we want to use UIO driver to control our custom AXI-Stream IP, we need to adapt UIO driver to control the DMA controller so that we can use it to submit DMA transaction to our IP.

We have modeled the high level problem and proposed a possible solution, but there is still some concerns. How do we submit DMA transaction through UIO? Is there any problem when doing the DMA transaction?

3.1 Problems

3.1.1 File Operations.

Recall the normal usage of custom AXI4-Full/Lite IP with UIO,

```
int main(void)
{
    int fd = open("/dev/uio0", O_RDWR);
    void *ptr = mmap(0, 0x10000, PROT_READ|PROT_WRITE,
                     MAP_SHARED, fd, 0);
    volatile uint32_t *ctrl = (uint32_t *)ptr;
    *ctrl = 0x00000000;
```

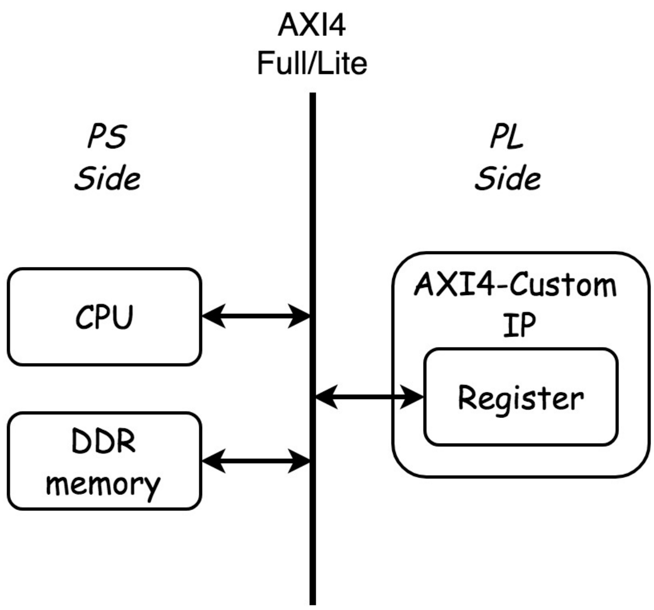

Figure 3.1: Custom IP with AXI4-Lite/Full Register.

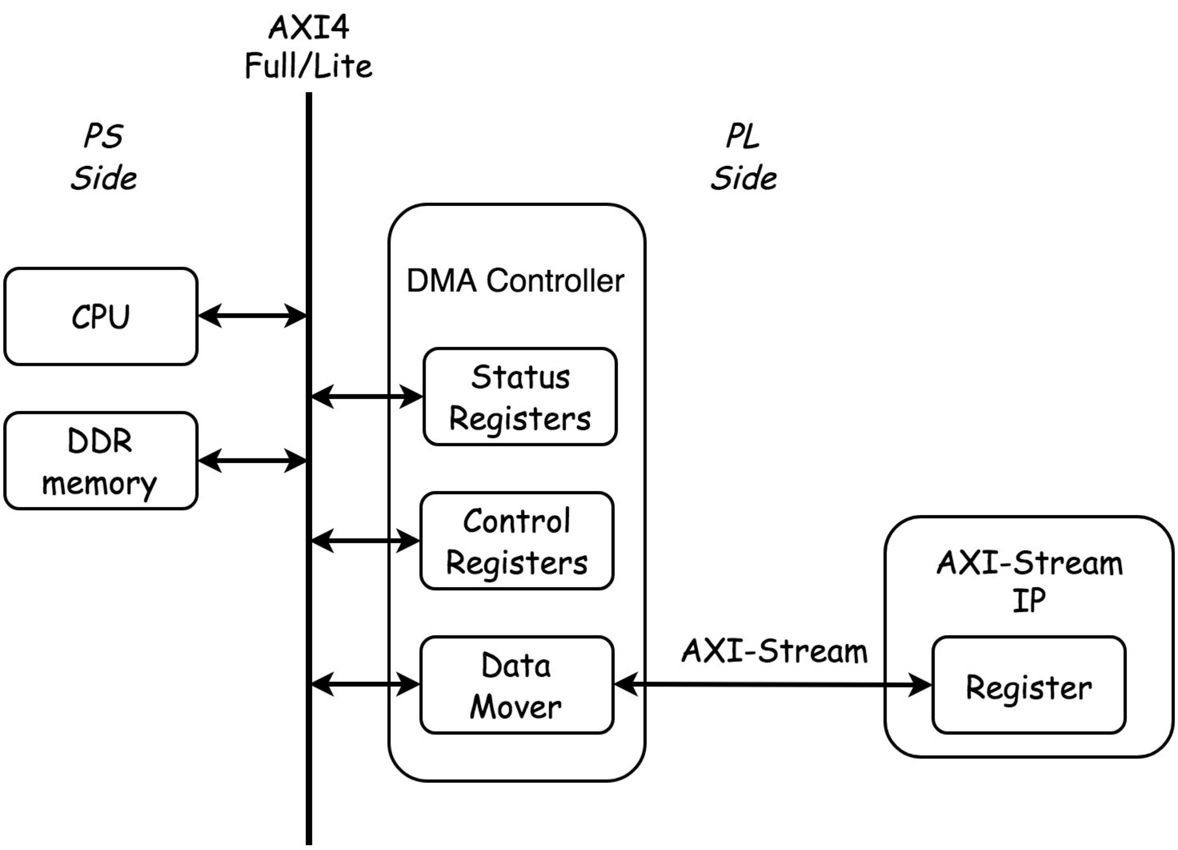

Figure 3.2: Custom IP with DMA and AXI-Stream Register.

```
    ...
}
```

typically, we open the device node to get device register pointer and memory-map to user memory, then we can manupulate the register like it is ..... The point is, we need file operations to communicate with DMA in UIO driver,

Let's take a look at two file operations **write(), read()**. Figure 3.5 shows what these two functions doing, basicly, these two functions in UIO driver is the handle about interrupt control. However, in our design, UIO node is actually a virtual device node, so interrupt control (and memory map) is no more needed. That means we can use **write(), read()** to do the ***real*** read/write work.

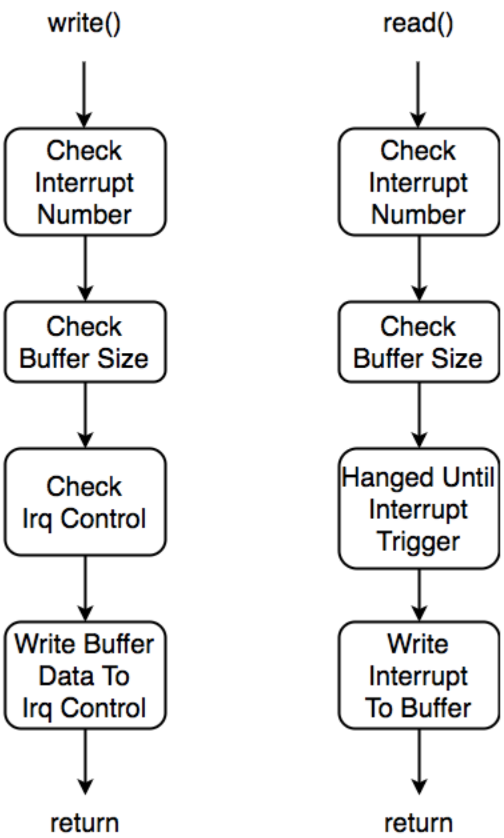

Figure 3.3: UIO write/read functions

3.1.2 Scatter Gather

In tranditional DMA transaction, it can only accept a contiguous (nonsegmented) block of physical memory, so, if we want to use DMA in userspace, and we can not get a contiguous memory space(like CMA), then we need to use DMA Scatter/Gather mode. This mode allows non- contiguous (nonsegmented) block of physical memory and this mode need to be turn on in Vivado design first. In this mode, DMA controller automatically give the start address of the segmented of memory after the previous transaction of segmented memory is completed. To apply this mode, we need to construct a special data structure, Scatterlist, which collects start address and lengths of segmented block of user buffer memory. DMA engine will do the transaction according to this list.

3.1.3 Cache Coherency

While using DMA to do the data transfering, it may lead cache coherency problems. If we want to receive data to the buffer through the DMA, but the buffer is in cache now, to apply transaction, we give controller the buffer address and length. Once the transaction is done, we read the buffer and the value is same as old value. CPU think the value in memory is not changed because whole data transfering is through DMA controller, so CPU keeps the old buffer data in cache, that makes the difference between cache data and real data. Figure 3.4 shows the cache coherency problem, both read and write may lead this problem, so if we want to transfer correct data, we must solve this problem.

3.2 Linux UIO driver for AXI DMA

3.3 Implementation

3.3.1 Device Tree

In first, we need to set our virtual device in our device tree file, in AXI4 IP, we have device block looks like:

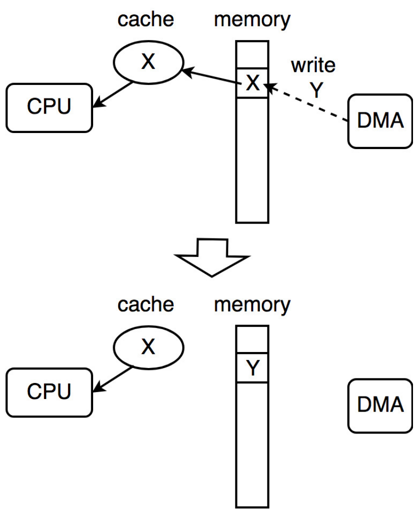

Figure 3.4: Cache Coherency Problems.

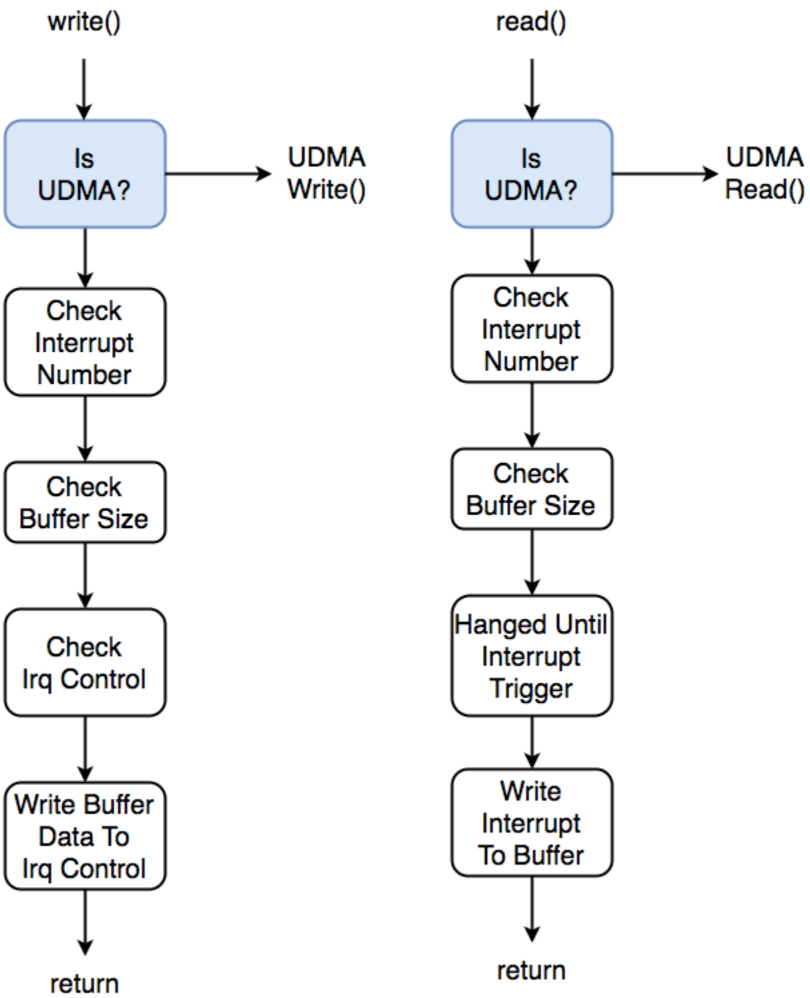

Figure 3.5: UIO write/read functions

```
my_customIP@43c00000 {
```

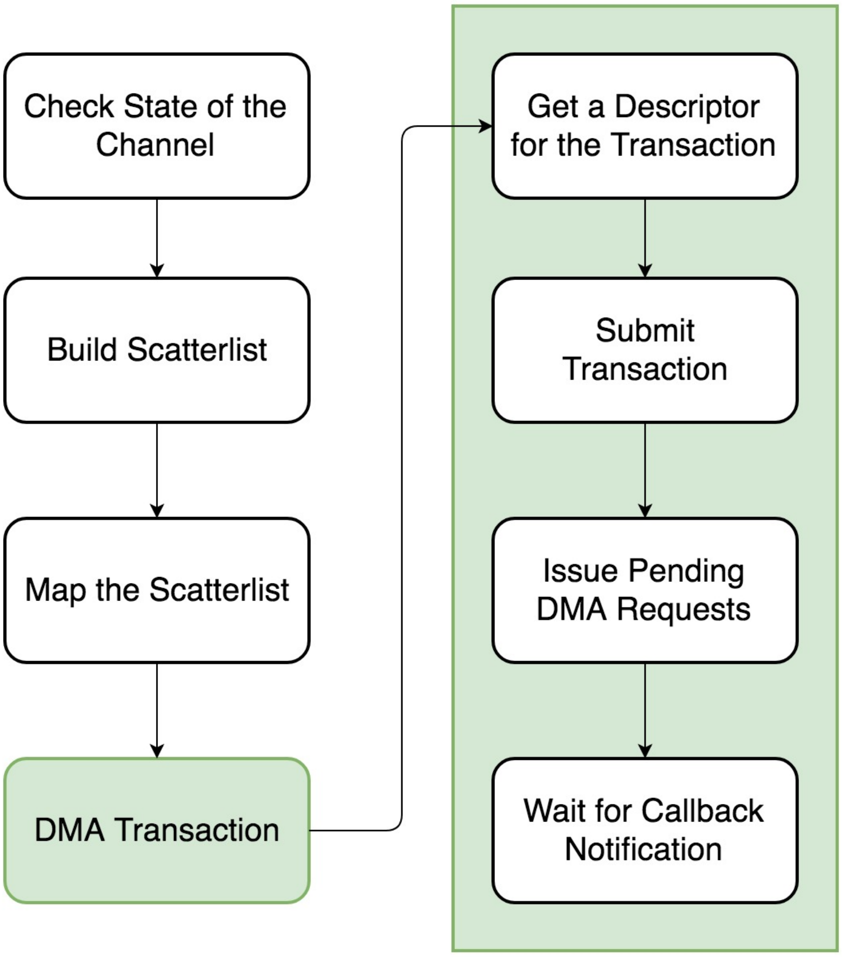

Figure 3.6: udma prepare for dma.

```
  compatible = "generic-uio";
  reg = <0x43c00000 0x10000>;
  interrupts = <0 29 1>;
  interrupt-parent = <0x3>;
  xlnx,s00-axi-addr-width = <0x6>;
  xlnx,s00-axi-data-width = <0x20>;
};
```

It contains IP name, IP register address, register length, interrupt control...etc. These are essential properties if you want to apply a driver to control the device. But in our design, we have no real device, so the device tree will looks like:

```
udma0 {
    compatible = "generic-uio";
```

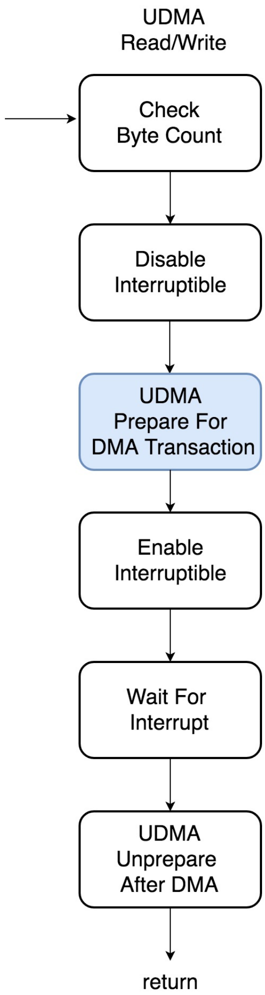

Figure 3.7: UDMA write/read functions

```
    dmas = <dma-channel1 dma-channel2 >;
    dma-names = "loop_tx", "loop_rx";
    ezdma,dirs = <2 1>;
};
```

where dmas property refers to the DMA channel under “axidma” in device tree, for example, if “axidma” looks like:

```
        loopback_dma: axidma@40410000 {
            #dma-cells = <1>;
            compatible = "xlnx,axi-dma";
            reg = < 0x40410000 0x10000 >;

            xlnx,include-sg;
            loopback_dma_mm2s_chan: dma-channel@40410000 {
```

```
            compatible = "xlnx,axi-dma-mm2s-channel";
            interrupt-parent = <&gic>;
            interrupts = <0 31 4>;
            xlnx,datawidth = <0x20>;
            xlnx,sg-length-width = <14>;
            xlnx,device-id = <0x1>;
        };

        loopback_dma_s2mm_chan: dma-channel@40410030 {
            compatible = "xlnx,axi-dma-s2mm-channel";
            interrupt-parent = <&gic>;
            interrupts = <0 32 4>;
            xlnx,datawidth = <0x20>;
            xlnx,sg-length-width = <14>;
            xlnx,device-id = <0x1>;
        };
    };
```

then “dmas” should be “<&loopback_dma 0 &loopback_dma >”, dma-names is fixed in driver, please make sure the names are same as the setting. “dirs” tells driver the direction of DMA channel. If the direction is not same as declare in “axidma”, it will fail when UIO is probing.After all these settings, our UIO driver should catch the device correctly and probe a device node under /dev.

3.3.2 Compile New Kernel

Because we modified the UIO driver and add some new library, we need to compile a new kernel. First, replace the old “uio.c” and “uio_pdrv_genirq.c” with new files. Then put “udma.c” under “drivers/uio” folder, and “udma.h” under “include/linux” folder. We need to add “obj-y += udma.o” in Makefile under “drivers/uio” . After, we can compile our new kernel with new UIO driver which can support DMA functions.

3.3.3 Linux On FPGA

Same as the we mentioned in former chapter, we boot Linux on FPGA in SD card with two partitions. The boot files in first partition have much changed in our scenario, "devicetree.dtb" and "uImage" we have discussed in former subsection. The modification in "uEnv.txt" is quite easy, this file provides additional environment variables for the bootloader,U-boot. It will looks like:

```
bootargs=console=ttyPS0,115200 root=/dev/mmcblk0p2 rootwait rw
    earlyprintk uio_pdrv_genirq.of_id=genric-uio
sdboot=if mmcinfo; then run uenvboot; echo Copying Linux from SD to RAM
    ... && load mmc 0 ${kernel_load_address} ${kernel_image} && load
    mmc 0 ${devicetree_load_address} ${devicetree_image} && load mmc 0
    ${ramdisk_load_address} ${ramdisk_image} && bootm ${
    kernel_load_address} - ${devicetree_load_address}; fi
```

To combine driver and device, please make sure the string behind "uio_pdrv_genirq.of_id="(in this case, is "generic-uio") is same as the property "compatible" in device tree file.

Figure 3.8 gives a simple illustration of our scenario.Unlike lots changes in first partition of SD card, we keep second partition as usual, this partition provides the file system when we boot our Linux, just keeps it as the same.

3.3.4 Example

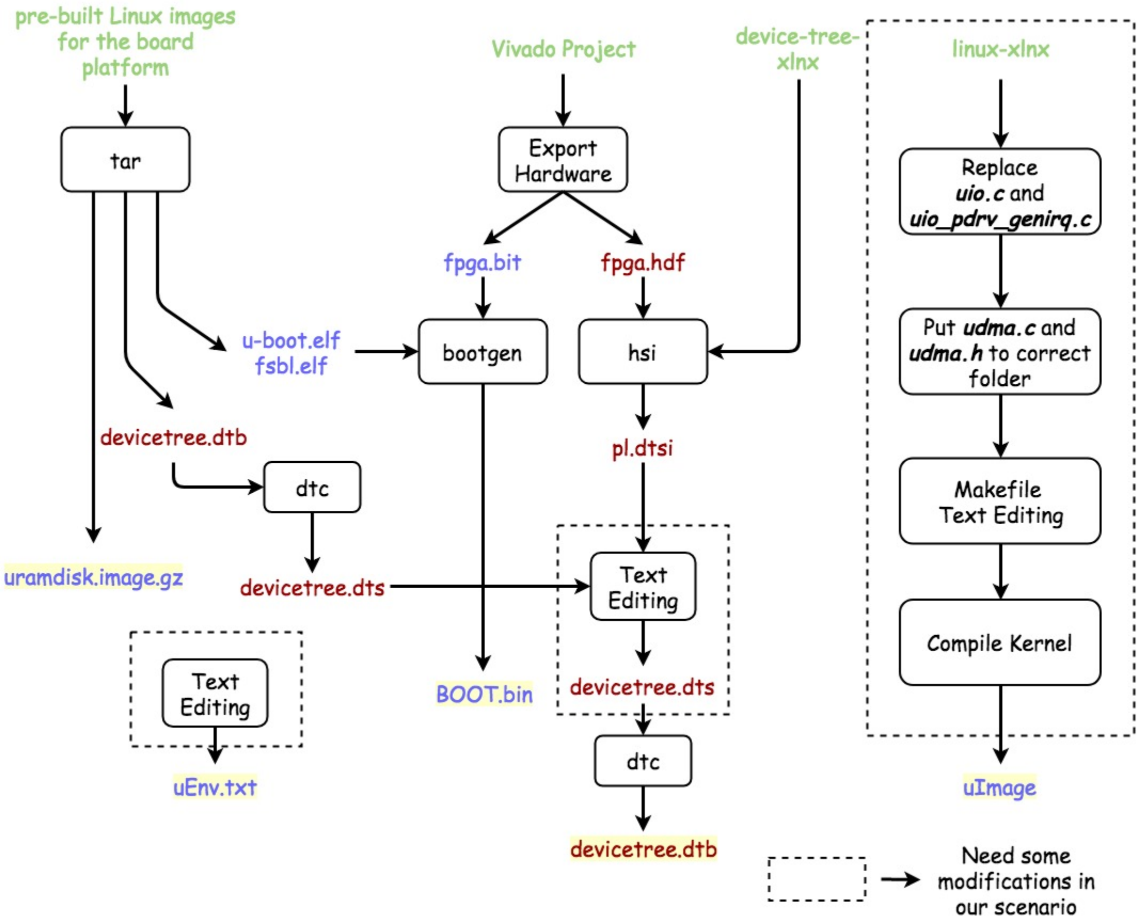

Figure 3.8: Embedded Linux on FPGA(UDMA ver.)

Chapter 4

Environment Framework

In our work,

- ZedBoard() as our target FGPA platform.
- Linux Kernel is compiled from the github repository provided by Xilinx.
- Linaro as our file system.
- Custom IP is designed(and provided) in Vivado 2016.04.
- Device Tree file is generated by SDK 2016.04, and need some little but significant changes.

Chapter 5

Analysis

This chapter, we use three different IP in our hardware designs. DMA with AXI Stream FIFO, DMA with OpenCores tinyAES and DMA with ECDSA(Curve secp256k1).

5.1 AXI FIFO

5.2 Custom Stream IP

5.2.1 DMA with OpenCores tinyAES

5.2.2 DMA with ECDSA(Curve secp256k1)

5.3 Comparison

Id	Family	#	Id	Family	#
A	DroidKungFu	473	K	FakePlayer	74
B	DorDrae	420	L	Wroba	74
C	Meds	221	M	Plankton	63
D	Fakeguard	203	N	DroidDreamLight	52
E	Boxer	202	O	Cawitt	51
F	Kmin	183	P	Badao	46
G	Rooter	117	Q	Fake10086	46
H	Boqx	114	R	Cupi	39
I	DroidAp	106	S	Coogos	39
J	DroidKungFu3	93	T	DroidDream	39

Table 5.1: Top 20 malware families in the dataset.

Chapter 6

Conclusion

References