# Implementation of DMA driver development under Linux

Chao Xia[1] , Gang Yang[1]
1. School of Information Engineering, Communication University of China, Beijing, China
1372217460@qq.com, gangy @cuc.edu.com

**Abstract—The emergence of Direct Memory Access （DMA） aims to solve the problems of traditional central processing unit（CPU） processing transaction data width, transmission data bit limit, low transmission speed and so on. Using Xilinx Zynq-7000 in Field Programmable Gate Array （FPGA）to provide DMA soft core for data transmission cannot meet the specific application requirements of communication with the host computer. Because its Linux system has all the advantages of files, open source code, support for cross-platform hardware, and strong portability, the Linux system that transplants DMA driver on Zynq PS side can meet the specific requirements of communication with the host computer. And the DMA driver designed in this article serves as a bridge between Advanced RISC Machines （ARM）and FPGA to support any 4.x version of Xilinx kernel and any board using Zynq-7000 series processing system.**

***Keywords—CPU; ZYNQ; DMA driver; FPGA***

## I. Introduction

In the development process of today's embedded systems, in various fields such as data mining, computer vision, and real-time data collection, the frequency and total amount of data generated and processed by electronic devices are increasing rapidly. This large-scale and explosively growing data set puts forward higher requirements on the data transmission capacity of modern data transmission systems, especially the efficiency of the data transmission of the system has gradually become one of the key factors that determine the performance of the entire system.

Various transactions in the general system are carried out under the control of the CPU, and data transmission is no exception. However, due to the limited data width of the CPU, the number of data bits transmitted is limited. Compared with the rate of the system memory, this method of controlling data transmission by the CPU is slower, and the CPU cannot perform other instruction processing in time to solve the above problems. Introduce the concept of DMA (Direct Memory Access) [1]. DMA is the abbreviation of direct memory access, which refers to the transmission method of data transfer transactions directly between external devices and system memory or between memory and memory, without CPU control during data transfer. The DMA transmission mode does not require the CPU to directly control the transmission. It opens up a direct data transmission path for RAM and I/O devices through hardware, which can greatly improve the efficiency of the CPU.

Understand the basic principles and methods of DMA driver based on Linux. It is of great value to study its driving system, architecture and important data structure, and to improve the driving method of traditional embedded systems, and to compile efficient, complete and adaptable DMA drivers.

## II. Kernel module development

### A. Loading and unloading modules

There are two ways to run the Linux driver. One is to compile the driver into the Linux kernel so that the driver will run automatically when the Linux kernel is started. The second is to compile the driver into a module (the module extension is .ko under Linux), and use the "insmod" command to load the driver module after the Linux kernel is started. When debugging the driver, we usually choose to compile it into a module, so we only need to compile the driver code after modifying the driver, not the entire Linux code. And when debugging, only need to load or unload the driver module, without restarting the entire system. In short, the biggest advantage of compiling the driver into a module is to facilitate development. When the driver development is completed and there is no problem, you can compile the driver into the Linux kernel. To sum up the comparison of the two methods mentioned above, this article adopts the scheme of compiling the driver into a module.

```
module_init(axidma_init);
module_exit(axidma_exit);
```

The module_init function in the DMA character device driver in this article is used to register a module loading function to the Linux kernel. The parameter axidma_init is the specific function that needs to be registered. When the driver is loaded using the "insmod" command, the function axidma_init will be called[2]. The module_exit() function is used to register a module uninstall function to the Linux kernel. The parameter axxidma_exit is the specific function that needs to be registered. The axidma_exit function will be called when the specific driver is uninstalled using the "rmmod" command.

### B. *Module initialization and exit module*

The module initialization function registers any function provided by the module. The actual initialization function definition pattern is often as follows:

```
static int_init initfunc(void)
{
    //Initialization Handle
}
module_init(init_func);
```

The initialization function is declared as static_init as a mark, indicating that it is used in initialization. module_init (init_func) is mandatory to initialize the module. This macro definition adds a special section to the module object code to mark the location of the initialization function. Without this macro definition, the initialization function cannot be called.

The cleanup function is also an essential part of the module. It is usually used to unregister the interface and is called when the module is removed. The purpose is to return the occupied resources to the system. The function is defined as follows:

```
static void_exit cleanup_function(void)
{
    //Cleanup
}
module_exit(cleanup_function);
```

Since the cleanup function does not need to return a value, it is declared as void type. Similarly, the exit mark is the same as the above init, which is only used to mark when the module is unloaded. module_exit is used by the kernel to find the cleanup function. If not, the kernel will return that the current module cannot be unloaded[3].

### C. *Error handling during initialization*

In the process of registering the initialization function with the kernel, the registration may fail. Even the simplest allocation of memory may cause the allocated memory to be unavailable. Therefore, the module code must continuously check whether the return value is successful. If an error occurs during the registration process, any operations registered before the failure must be undone[4].

Therefore, if any error occurs, first determine whether the module has the ability to continue to initialize itself anyway. If it turns out that your module fails to load by itself, you must cancel all registration actions before the failure. If you cannot unregister what you have obtained, due to the existence of some uncancelled pointers, the kernel will be placed in an unstable and unsafe state. In this case, the only way is to restart the system. Therefore, the handling of errors in initialization is particularly important.

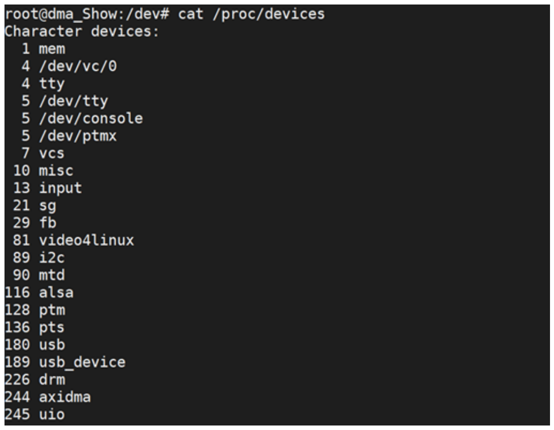
## III. Equipment number allocation

### D. *Equipment number indication*

In order to facilitate management, each device in Linux has a device number. The device number is composed of two parts: the major device number and the minor device number. The major device number represents a specific driver, and the minor device number represents each device that uses this driver. Linux provides a data type named dev_t to represent the device number. dev_t is defined in include/linux/types.h and is defined as follows:typedef __u32 __kernel_dev_t;

typedef __kernel_dev_t dev_t;

dev_t is actually an unsigned int type, which is a 32-bit data type. The 32-bit data constitutes two parts of the major device number and the minor device number. The upper 12 bits are the major device number and the lower 20 bits are the minor device number. Therefore, the range of the major device number in the Linux system is 0~4095. DMA character devices appear in the file system in the form of files. The device file is located in the /dev directory and can be viewed using the ls command. The device number of the axidma driver is shown in the figure below.



```
root@dma_Show:/dev# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
226 drm
244 axidma
245 uio
```

Figure 1. Axidma driver device number under linux

### E. *Assign and release device numbers*

There are two ways to allocate device numbers: static allocation and dynamic allocation. Statically assigning device numbers requires us to check all the used device numbers in the current system, and then select an unused one. Moreover, static allocation of device numbers can easily cause conflicts. The Linux community recommends using dynamic allocation of device numbers. Apply for a device number before registering a DMA character device. The system will automatically give you a device number that is not used, so as to avoid it. conflict. Just release the device number when uninstalling the driver. The statically allocated device number used by the DMA character device driver in this article. The application function of the device number, such as int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name) function alloc_chrdev_region is used to apply for the device number. This function has 4 parameters:

(1) dev: save the device number applied for;

(2) baseminor: the starting address of the minor device number, alloc_chrdev_region can apply for a segment of multiple consecutive device numbers. The major device numbers of these device numbers are the same, but the minor device numbers are different. The minor device numbers start with baseminor and increase gradually . Generally, baseminor is 0, which means that the minor device number starts from 0.

(3) count: the number of device numbers to be applied for;

(4)name: the name of the device;

After unregistering the character device, the device number should be released. The device number release function is void unregister_chrdev_region(dev_t from, unsigned count). This function has two parameters:

(1) from: the device number to be released;

(2) count: Indicates the number of device numbers to be released starting from from;

## IV. DMA DRIVER

### *DMA drives the Platform mechanism*

Compared with the traditional device_driver mechanism, the Linux Platform_driver mechanism has its advantages mainly in that the Platform mechanism registers its own resources into the kernel and is managed uniformly by the kernel. When these resources are used in the driver, they are applied for and used through the standard interface provided by Platform_device. This improves the independence of drive and resource management, and has better portability and security. The following is a schematic diagram driven by the Platform mechanism.

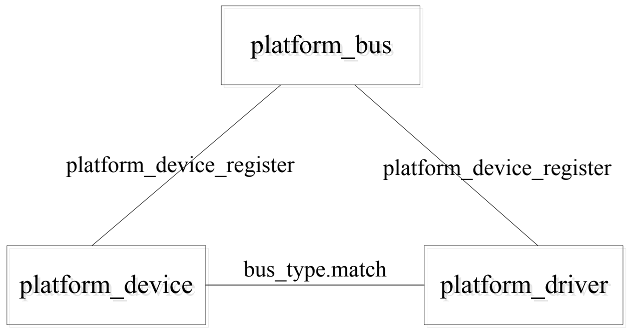


Figure 2. Schematic diagram of platform mechanism

Like traditional drivers, the platform mechanism is also divided into three steps:

1) Bus registration stage:

Kernel_init() → do_basic_setup() → driver_init() → platform_bus_init()→bus_register in the main.c file when the kernel is initialized, registers a platform bus (virtual bus platform_bus).

2) Add equipment phase:

When the device is registered, platform_device_register() →platform_device_add()→(pdev→dev.bus→device_add() just hang the device to the virtual bus.

3) Driver registration stage:

platform_driver_register() → driver_register() → bus_add_driver() → driver_attach() → bus_for_each_dev() Perform driver_attach()→driver_probe_device() on each device connected to the virtual platform bus to determine whether drv→bus→match() is executed If it succeeds, execute platform_match→strncmp(pdev→name, drv→name, BUS_ID_SIZE) through the pointer, and call really_probe if they match. Start the real probe. If the probe succeeds, bind the device to the driver.

### *DMA driver introduction*

The major and minor device numbers indicate the driver and the device. After registering the device number, the next step is to consider associating the device number with the operation and management of the device. Defining the file_operation structure is the link that establishes this connection. The definition of this structure contains a set of function pointers, which are the operation methods of the device object. It is used to provide the interface between the DMA character device driver and the virtual file system. Several main functions are defined as follows: static struct file_operations axidma_fops = {

```
.owner = THIS_MODULE,
.open = axidma_open,
.release = axidma_release,
.unlocked_ioctl = axidma_ioctl,
.mmap = axidma_mmap};
```

The meaning of each function is as follows：

TABLE I. FILE_OPERATIONS FUNCTION FUNCTION

| Main function | Function |
|---|---|
| axidma_open | Turn on the DMA device |
| axidma_release | The release function is used to release the space allocated by the DMA device during initialization |
| axidma_ioctl | The driver performs corresponding operations through different interactive protocols, including but not limited to DMA reading, DMA writing, setting DMA transfer completion asynchronous signal, etc. |
| axidma_mmap | In the mmap function, the driver allocates a continuous memory address on the physical address for the DMA buffer through kmalloc |

*API interface function design*

The driver and user-space application programming interface (API) functions introduced in this section are used as a common layer between the processor and FPGA so that a DMA port in the Zynq processing system can be used as the interface between the processing system and the FPGA. Communication bridge, so that the upper-level application does not need to care about the working process of the hardware, but only needs to care about how to call the API function to complete the control of the hardware IP. The driver framework is shown in Figure 3

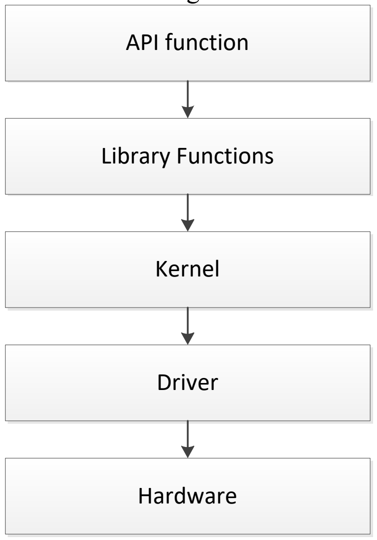


Figure 3 Driver framework

The axidma_init function is used to initialize the DMA device. The initialization process is as follows: First, find the DMA device through the base address of the DMA device, and turn on the DMA device. Then detect all available channels in the DMA driver, allocate space for channel metadata by querying the number of all DMA channels in the module, obtain the metadata of the available channels through the ioctl function, allocate space for each data channel and assign ID, and finally Assign the data in the channel metadata to the DMA channel.

TABLE II. DMA DRIVER API INTERFACE FUNCTION

| API interface function | Parameter |
|---|---|
| axidma_init | / |
| axidma_malloc | dev, size |
| axidma_destroy | dev |
| axidma_free | dev, *addr, size |
| axidma_transfer | dev, channel, *buf, len, bool wait |

The axidma_malloc function allocates a specified size of memory space for the device through the mmap function. This function allocates a DMA buffer of contiguous physical memory that can be shared between the processor and FPGA. The axidma_destroy function is used to destroy the DMA device. It first releases the channel number and signal metadata of the DMA device, then closes the DMA device, and finally releases the device structure space. The axidma_free function is used to release the device memory allocated by the axidma_malloc function. axidma _transfer is used to open a specified channel and DMA transfer with a specified length. If it is a DMA read, the DMA will send the data in the transmit buffer to the PL; if it is a DMA write, the DMA will store the data uploaded by the PL into the receive buffer. The function respectively transmits the transmission control information (channel number, buffer area, transmission length), DMA transmission instruction (transmission direction) to the driver layer through ioctl, and informs the driver layer to drive the DMA transmission. After the transmission is over, the status of whether the transmission is successful will be returned. Since the DMA channel number here is consistent with the configuration of the device tree, in the device tree, different channels will be set and assigned channel numbers. Therefore, the transmission direction and other information required by the streaming DMA will be read from the device tree via the driver, and then called directly in the driver. When receiving the driver's DMA transfer completion interrupt, it indicates that the single DMA transfer has been completed.

## V. CONCLUDING REMARKS

This article mainly introduces the driver's realization process in detail. Loading and unloading modules, module initialization and exit modules in the kernel module, and initialization error handling. Specific research is also done on the allocation of the device number of the DMA drive device. The driver mainly includes three aspects, the Platform mechanism of the DMA driver, the DMA driver and the API interface function, and the realization process of the specific driver is studied. Because the structure of the driver and the application program is quite different, the correlation is weak and the workload is large, so this article has a strong universality, that is, the DMA driver is fully applicable to the 4.x version of the Xilinx kernel and the use of Zynq-7000 The board of the series processing system.

## REFERENCES

[1] J Mitola. Software radios: Survey, critical evaluation and future directions[J]. IEEE Aerospace and Electronic Systems Magazine, 1993, 8(4): 25-36.
[2] Xilinx Inc. Zynq-7000 All Programmable SoC Technical Reference Manual [EB/OL], 2016
[3] Xilinx Inc. AXI DMA v7.1 LogiCORE IP Product Guide [EB/OL], 2016.
[4] Xilinx Inc. AXI DMA v7.1 LogiCORE IP Product Guide[EB/OL],2016.10.
[5] Xilinx Inc. Vivado Design Suite User Guide Design with IP[EB/OL],2016.10.
[6] Xilinx Inc. Zynq-7000 All Programmable SoC Technical Reference Manual[EB/OL],2016.9.