

# SIXUENET

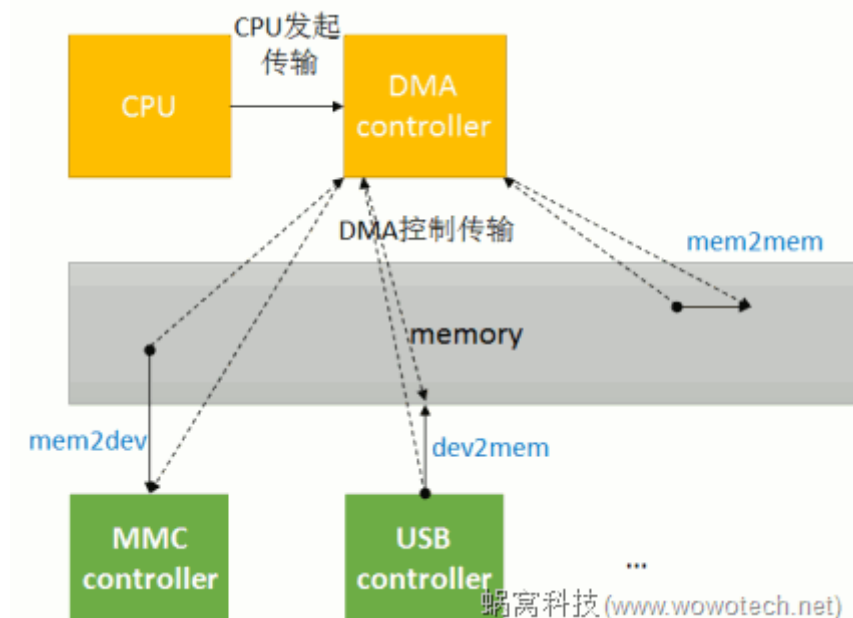
To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose  
(<http://www.mysixue.com/>)

## DMA SUBSYSTEM

📅 April 20, 2019 (<Http://Www.Mysixue.Com/?P=123>) 👤 JL  
(<Http://Www.Mysixue.Com/?Author=1>) 🔊

### 1 Introduction to DMA

DMA is the abbreviation of Direct Memory Access , as the name implies, it means to bypass the CPU and directly access memory . In a computer, compared to the CPU , the speed of memory and peripherals is very slow, so moving data between memory and memory (or memory and device) wastes CPU time, causing the CPU to be unable to process some real-time events in time. Therefore, engineers have designed a device specially used to transport data— DMA controller to assist the CPU in data transport, as shown in the following figure:



- DMA Controller, that is, DMA controller . There is such a peripheral on the general ARM CPU . It provides DMA services , so it can be considered as a Provider.
- There are some peripherals on the ARM CPU , which can use DMA to move data . For example, MMC, when it receives data , it will store the data in its own registers, and we can read the registers through the CPU to obtain these data. (which will consume CPU time) ; can also use DMA to move data register into memory ,

when moving a certain number , notifies CPU, then the CPU is read from memory one time ( DMA move process , CPU can Do other things) .

These peripherals use the services provided by the DMA controller , so they can be considered Consumers.

- Further , the DMA controller when moving data , for some memory requirements , for example, it requires continuous physical addresses and the like . Thus there is also directed to a DMA Buffer problem allocation .

Later will focus on the Provider , Consumer , DMA Buffer allocate these 3 aspects to introduce kernel mechanisms provided .

## 2 Introduction to DMA related concepts

### 2.1 DMA Controller & Channels

DMA Controller: refers to the DMA controller on the CPU .

DMA Channels: There may be multiple channels inside each controller , and each channel is used as a Channel.



Note that , in view of the conflict bus access , and memory coherency considerations , these Channels can not really work in parallel , but only time division multiplexing , DMA controller to act as an arbitration role .

Since it is time-multiplexed , then we can be based on certain physical Channel, abstract multiple virtual Channel, responsible for arbitration by the software , to determine which virtual at some point by the Channel to use physical Channel. Linux kernel does some abstract virt channel, we will introduce it in detail in the section " DMA Provider " .

### 2.2 DMA Slave Device

DMA Device: refers to those that can utilize DMA of on-chip peripherals moving data , for example the MMC , the USB the Controller and the like .

In some places, it is also called DMA Slave Device.

### 2.3 DMA DRQ

In the example of MMC using DMA to move data , when is it reasonable for the DMA controller to start moving data ? Obviously MMC receives the data and stores it in the register .

That DMA controllers know how to MMC if the data is received it ? The answer is DRQ. In the DMA Slave Device with DMA Controller has several physical connection between , Chen made DRQ. When the Device When you are ready send and receive data , it is through DRQ notifies the Controller to start moving .



### 2.4 transfer size

A simplest DMA controller requires only one parameter : transfer size.

Each clock cycle , the data controller transmits a byte , until all the data has been transmitted .

## 2.5 transfer width





However, this is often not enough in the real world, because some devices may need to transmit data of a specified bit in one clock cycle , for example:

- ✓ Memory transfer data between the time, hoping to the maximum width of the bus units ( 32-bit , 64-bit , etc.), to improve efficiency of data transmission;
- ✓   audio equipment, it is necessary to write accurate 16-bit or 24-bit data every time ;
- ✓ wait

Therefore, in order to meet these diverse needs, the DMA controller provides an additional parameter— transfer width .

## 2.6 burst size

When the source or destination of the transfer is memory , in order to improve efficiency, the DMA controller is not willing to access the memory every time it transfers , but opens a buffer internally and caches the data in its own buffer :

- ✓   memory is the source, read a batch of data from memory at a time , save it in its own buffer , and then transmit it to the destination a little bit (using the clock as the beat);
- ✓   memory is the destination, first transfer the source data to its own buffer , and after accumulating a certain amount of data, write it to memory once .

In this scenario, the amount of data that can be cached inside the DMA controller is called burst size

## 2.7 scatter-gather

The general DMA controller can only access the memory with continuous physical addresses . But in some scenarios , we only have some memory blocks with discontinuous physical addresses , and we need DMA to move the data of these memory blocks to other places . In this scenario, Chen Zuo scatter- gather .

Therefore, from the software level , the DMA core layer must provideThe ability of scatter-gather . If the DMA controller itself supportsFor scatter-gather operation , you can configure the controller directly ; if the controller does not support it , we can only simulate it with software .

For those DMA controllers that support scatter-gather operations , the software usually needs to prepare a table or link-list. Then you can inform the DMA controller of the address of the table and the number of elements , or the head element of the link-list to the control device . when the controller starts moving , it goes through each element , in order to know which address to get data from .

Different DMA hardware may use different forms , some use table, some use link-list, and some use others . But no matter which form , each element in the set needs to provide source address, destination address, transfer size , Transfer width , Burst size these information .

If software simulation is used , then we can only move the data of these memory blocks to a continuous address , and then let the DMA start to move from this new address . Of course, this will degrade performance .

### 3 DMA Consumer

Consumer refers to the kernel driver that uses the DMA function . For example , the driver of the MMC controller , part of the code will use DMA for data movement .

Core DMA subsystem consists dmaengine responsible for the management : provider to the engine to be registered ; consumer use engine provided APIs.

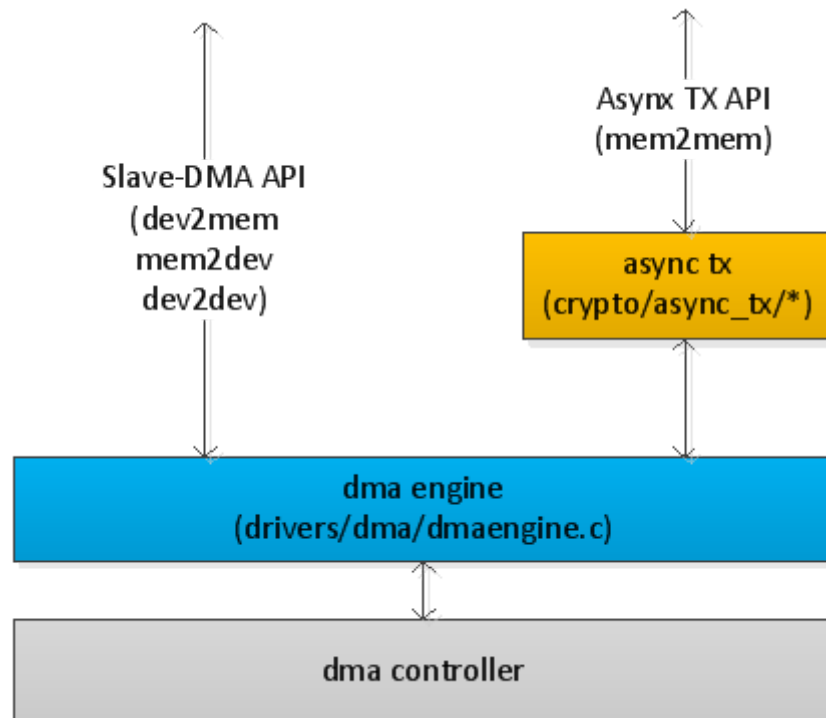
This section will discuss these APIs in detail .

#### 3.1 Slave-DMA API和Async TX API

In terms of direction, DMA transfer can be divided into 4 categories: memory to memory , memory to device , device to memory, and device to device .

As the agent of the CPU , Linux kernel , from its perspective, peripherals are all slaves , so these transfers ( MEM2DEV , DEV2MEM , DEV2DEV ) with device participation are called Slave-DMA transfers. And another memory- to- memory transmission is called Async TX .

Why emphasize this difference? Because the Linux for convenience based DMA of the memcpy , Memset other operations in dma engine on, encapsulation layer more concise API (as in FIG. Shown), such API is the Async the TX API (to async\_ beginning, e.g. async\_memcpy , async\_memset , async\_xor, etc.).



Finally, because the memory to the memory of the DMA transfer have a relatively simple API , no need to directly use the dma engine provided by API , finally resulting in dma engine provided by the API on especially for the Slave-DMA API (the mem2mem excluded).

This section mainly discusses the Slave-DMA API. There will be a chance to look at the Async TX API in the future.

## 3.2 Slave-DMA APIs

For Consumer driver writers, to perform DMA transfer based on the Slave-DMA API provided by dma engine , the following steps are required:

- ✓ `dma_request_channel()` for a DMA channel
- ✓ `dma_slave_config()` the parameters of DMA channel according to the characteristics of the device ( slave )
- ✓ `dmaengine_xfer()` DMA transfer is to be performed , obtain a descriptor ( descriptor ) used to identify the current transfer ( transaction )
- ✓ Submit this transfer ( transaction ) to the dma engine and start the transfer
- ✓ `dmaengine_free()` for the end of the transmission ( transaction )
- ✓ then repeated 3 to 5 to

The above 5 steps, except 3 is a bit hard to understand, the others are relatively intuitive and easy to understand. For details, please refer to the following introduction.

### 3.2.1 Apply / Release DMA channel

Any consumer must apply for a DMA channel before starting DMA transfer .

The DMA channel ( represented by the " struct dma\_chan " data structure in the kernel ) is provided by the provider and used by the consumer .

Consumers can apply for DMA channel through the following API :

```
struct dma_chan *dma_request_chan(struct device *dev, const char *name);
```

- ✓ The API was added in the linux-4.5 version, and the API used before that was `dma_request_channel(mask, x, y)`
- ✓ The API will return the dma channel named name bound to the specified device ( dev ) .
- ✓ DMA Engine the provider and the consumer can use Device Tree , the ACPI or struct dma\_slave\_map type match table provide the relationship between this binding

The dma channel obtained by the application can be released through the following API when it is not needed :

```
void dma_release_channel(struct dma_chan *chan);
```

### 3.2.2 Configure DMA channel parameters

After the consumer applies for a DMA channel for its own use , it needs to configure the channel according to its actual situation .

The configurable content is represented by the struct dma\_slave\_config data structure (for details, please refer to the latter part of this chapter ).

consumer the configuration information is filled into a struct dma\_slave\_config the variables, the following can be called API to tell the information to the DMA Controller :

```
int dmaengine_slave_config(struct dma_chan *chan, struct dma_slave_config *config)
```

### 3.2.3 Description obtaining a transport identifier ( TX descriptor )

DMA transfer is an asynchronous transfer, before starting the transmission, Consumer Driver needs some information for this transmission (e.g., src / dst of Buffer , direction of transmission, etc.) presented to the DMA Engine (are essentially DMA Controller Driver ), DMA Engine confirmation After okay , return a descriptor ( abstracted by struct dma\_async\_tx\_descriptor ). After that, the consumer driver can control and track this transmission in units of this descriptor.

For the data structure of struct dma\_async\_tx\_descriptor, please refer to the introduction later in this chapter .

Depending on the transmission mode, the consumer driver can use the following three APIs to obtain the transmission descriptor :

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(  
    struct dma_chan *chan, struct scatterlist *sgl,  
    unsigned int sg_len, enum dma_data_direction direction,  
    unsigned long flags);
```

This API is used for scatter - gather type transmission . The meaning of the parameter list is as follows :

- ✓ chan : dma channel used in this transmission
- ✓ sgl : a list allocated by dma\_map\_sg
- ✓ sg\_len: the number of elements in the list
- ✓ direction : the direction of data transmission , please refer to the definition of enum dma\_data\_direction (<https://elixir.bootlin.com/linux/v4.5-rc1/source/include/linux/dma-direction.h#L7>) for details (<https://elixir.bootlin.com/linux/v4.5-rc1/source/include/linux/dma-direction.h#L7>)
- ✓ the flags : for the dma controller driver transmitting some additional information, including (specific reference enum dma\_ctrl\_flags (<https://elixir.bootlin.com/linux/v4.5-rc1/source/include/linux/dmaengine.h#L190>) to DMA\_PREP\_ beginning defined): (<https://elixir.bootlin.com/linux/v4.5-rc1/source/include/linux/dmaengine.h#L190>)
  - ✧ DMA\_PREP\_INTERRUPT : Tell the DMA controller driver that after the transfer is completed, an interrupt will be generated and the callback function provided by the client will be called
  - ✧ DMA\_PREP\_FENCE : Tell the DMA controller driver that subsequent transfers depend on the result of this transfer (so that the controller driver will carefully organize the sequence of multiple DMA transfers)
  - ✧ ...

```
struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(  
    struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,  
    size_t period_len, enum dma_data_direction direction);
```

This API used in the scene audio, etc. , during a certain length dma transmission ( buf\_addr & buf\_len process) of , per transmission certain byte ( period\_len ) , it will call the callback function once the transfer is complete . Parameters comprises:

- ✓ chan : dma channel used in this transmission
- ✓ buf\_addr , buf\_len : buffer address and length of transmission
- ✓ period\_len : How often (in bytes ) to call the callback function . It should be noted that buf\_len should be an integer multiple of period\_len
- ✓ direction : the direction of data transmission

```
struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(  
    struct dma_chan *chan, struct dma_interleaved_template *xt,
```

```
unsigned long flags);
```

This API can be used for discontinuous and cross DMA transmission. It is usually used in image processing, display and other scenes. For details, please refer to the definition and explanation of struct dma\_interleaved\_template structure. Also okay ).

### 3.2.4 Set callback function

Once the descriptor is successfully obtained , we can add a callback function to the descriptor , and then submit the descriptor .

What can be done in the callback function ? The [official text](https://www.kernel.org/doc/html/latest/driver-api/dmaengine/client.html) (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/client.html) has the following instructions :

#### Note

Although the async\_tx API specifies that completion callback routines cannot submit any new operations, this is not the case for slave/cyclic DMA.

For slave DMA, the subsequent transaction may not be available for submission prior to callback function being invoked, so slave DMA callbacks are permitted to prepare and submit a new transaction.

For cyclic DMA, a callback function may wish to terminate the DMA via dmaengine\_terminate\_async().

Therefore, it is important that DMA engine drivers drop any locks before calling the callback function which may cause a deadlock.

Note that callbacks will always be invoked from the DMA engines tasklet, never from interrupt context.

### 3.2.5 Submit the transaction

Once the descriptor has been prepared and the callback information added, it must be placed on the DMA engine drivers pending queue.

Interface:

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
```

This returns a cookie can be used to check the progress of DMA engine activity via other DMA engine calls not covered in this document.

Note that dmaengine\_submit is only added to the transmission queue , and transmission is not turned on .

To open the transfer , need to Issue pending DMA requests.

### 3.2.6 Issue pending DMA requests

The transactions in the pending queue can be activated by calling the issue\_pending API. If channel is idle then the first transaction in queue is started and subsequent ones queued up.

On completion of each DMA operation, the next in queue is started and a tasklet triggered. The tasklet will then call the client driver completion callback routine for notification, if set.

Interface:

```
void dma_async_issue_pending(struct dma_chan *chan);
```

### 3.2.7 wait for callback notification

After the transmission request is submitted, the client driver can obtain the transmission completion message through the callback function .

Of course, can also `dma_async_is_tx_complete` like the API , the test transmission is completed. It will not be explained in detail.

Finally, if you can't wait, you can also use `dmaengine_pause` , `dmaengine_resume` , `dmaengine_terminate_XXX` and other APIs to suspend and terminate the transmission.

### 3.2.8 Other APIs

#### Terminate APIs

```
int dmaengine_terminate_sync(struct dma_chan *chan)
int dmaengine_terminate_async(struct dma_chan *chan)
int dmaengine_terminate_all(struct dma_chan *chan) /* DEPRECATED */
```

This causes all activity for the DMA channel to be stopped, and may discard data in the DMA FIFO which hasn't been fully transferred. No callback functions will be called for any incomplete transfers.

`dmaengine_terminate_async()` might not wait until the DMA has been fully stopped or until any running complete callbacks have finished. But it is possible to call `dmaengine_terminate_async()` from atomic context or from within a complete callback.

`dmaengine_terminate_sync()` will wait for the transfer and any running complete callbacks to finish before it returns. But the function must not be called from atomic context or from within a complete callback.

`dmaengine_terminate_all()` is deprecated and should not be used in new code.

#### Pause API

```
int dmaengine_pause(struct dma_chan *chan)
```

This pauses activity on the DMA channel without data loss.

#### Resume API

```
int dmaengine_resume(struct dma_chan *chan)
```

Resume a previously paused DMA channel. It is invalid to resume a channel which is not currently paused.

#### Check Txn complete



```
enum dma_status dma_async_is_tx_complete(struct dma_chan *chan,
                                         dma_cookie_t cookie, dma_cookie_t *last, dma_cookie_t *used)
```

This can be used to check the status of the channel. Please see the documentation in `include/linux/dmaengine.h` for a more complete description of this API.

This can be used in conjunction with `dma_async_is_complete()` and the cookie returned from `dmaengine_submit()` to check for completion of a specific DMA transaction.

#### Note

Not all DMA engine drivers can return reliable information for a running DMA channel. It is recommended that DMA engine users pause or stop (via `dmaengine_terminate_all()`) the channel before using this API.

### Synchronize termination API

```
void dmaengine_synchronize(struct dma_chan *chan)
```

This function should be used after `dmaengine_terminate_async()` to synchronize the termination of the DMA channel to the current context. The function will wait for the transfer and any running complete callbacks to finish before it returns.

If `dmaengine_terminate_async()` is used to stop the DMA channel this function must be called before it is safe to free memory accessed by previously submitted descriptors or to free any resources accessed within the complete callback of previously submitted descriptors.

The behavior of this function is undefined if `dma_async_issue_pending()` has been called between `dmaengine_terminate_async()` and this function.

## 3.3 Description of important data structure

### 3.3.1 struct dma\_slave\_config

Which contains a complete DMA all possible parameters required for transmission, which is defined as follows:

Header file : `include/linux/dmaengine.h`

struct dma_slave_config	Comment
enum dma_transfer_direction direction	direction , indicates the direction of transmission, including : <ul style="list-style-type: none"><li>➤ DMA_MEM_TO_MEM , transfer from memory to memory n- OTE: Reference " . 3 .1 " , MEM -TO-MEM generally not with S Lave the DMA APIs-</li><li>➤ DMA_MEM_TO_DEV , transfer from memory to device</li><li>➤ DMA_DEV_TO_MEM , transfer from device to memory</li><li>➤ DMA_DEV_TO_DEV , device-to-device transfer</li></ul> Not necessarily support all DMA transmission directions, depending on the implementation of the provider

phys_addr_t src_addr	When the transmission direction is dev2mem or dev2dev , the location of the data to be read (usually a fixed FIFO address) . For mem2dev type channels , this parameter is not required (it will be specified during each transmission) .
phys_addr_t dst_addr	When the transmission direction is mem2dev or dev2dev , the location where the data is written (usually a fixed FIFO address) . For dev2mem type channels , this parameter does not need to be configured (it will be specified during each transmission) .
enum dma_slave_buswidth src_addr_width	The width of the src address, including 1 , 2 , 3 , 4 , 8 , 16 , 32 , 64 ( bytes ), etc. (For details, please refer to the definition of enum dma_slave_buswidth )
enum dma_slave_buswidth dst_addr_width	Class on
u32 src_maxburst	src maximum transmission Burst size ( Burst size introduction reference " 2.6 " ), the unit is src_addr_width (note, not a byte )
u32 dst_maxburst	Class on
bool device_fc	When the peripheral is a Flow Controller (flow controller), this field needs to be set to true . In the design of the connection between DMA and external devices in the CPU , the module that determines whether the DMA transfer ends, called flow controller , DMA controller or external device, can be used as a flow controller , depending on the design of the peripheral and DMA controller The principle, signal connection method, etc., will not be explained in detail
unsigned int slave_id	The external device tells the dma controller who it is through the slave_id (usually corresponding to a certain request line ). Many dma controllers do not distinguish between slaves , as long as they are given src , dst , len and other information, it can be transmitted, so slave_id can be ignored. And some controllers must clearly know which peripheral is the object of this transmission, and they must provide slave_id .

### 3.3.2 struct dma\_async\_tx\_descriptor

The transfer descriptor is used to identify a DMA transfer (similar to a file handle) .

" 3.2.3 " Section A the PI can obtain a transfer descriptor , Client after obtaining the descriptor, it may be a unit, subsequent operations (start transmission, wait for completion, etc.). You can also provide your own callback function to the dma engine through the descriptor .

Header file : include/linux/dmaengine.h

struct dma_async_tx_descriptor	Comment
dma_cookie_t cookie	cookie , an integer number, used to track this transmission. Under normal circumstances, the engine will maintain an incremental number internally . Whenever the client obtains the transmission description, it will assign the number to the cookie and then increase by one.

enum dma_ctrl_flags flags	The tags at the beginning of DMA_CTRL_ include : ➤ DMA_CTRL_REUSE , indicating that this descriptor can be reused until it is cleared or released ➤ DMA_CTRL_ACK , if the flag is 0 , it means that it cannot be reused temporarily
dma_addr_t phys	The physical address of the descriptor? ? I don't quite understand!
struct dma_chan *chan	Corresponding dma channel
dma_cookie_t (*tx_submit)(struct dma_async_tx_descriptor *tx)	The callback function provided by the controller driver is used to submit the descriptor to the list to be transferred. Usually called by the dma engine , the client driver will not directly deal with this interface
int (*desc_free)(struct dma_async_tx_descriptor *tx)	The callback function used to release the descriptor is provided by the controller driver and called by the dma engine . The client driver will not directly deal with this interface
dma_async_tx_callback callback	Transfer completion callback function (and its parameters), the client driver to provide
void *callback_param	
struct dmaengine_unmap_data *unmap	For the other parameters in the back, the client driver does not need to care about it, so I won't describe it for now.
struct dma_async_tx_descriptor *next	
struct dma_async_tx_descriptor *parent	
spinlock_t lock	

## 4 DMA Provider

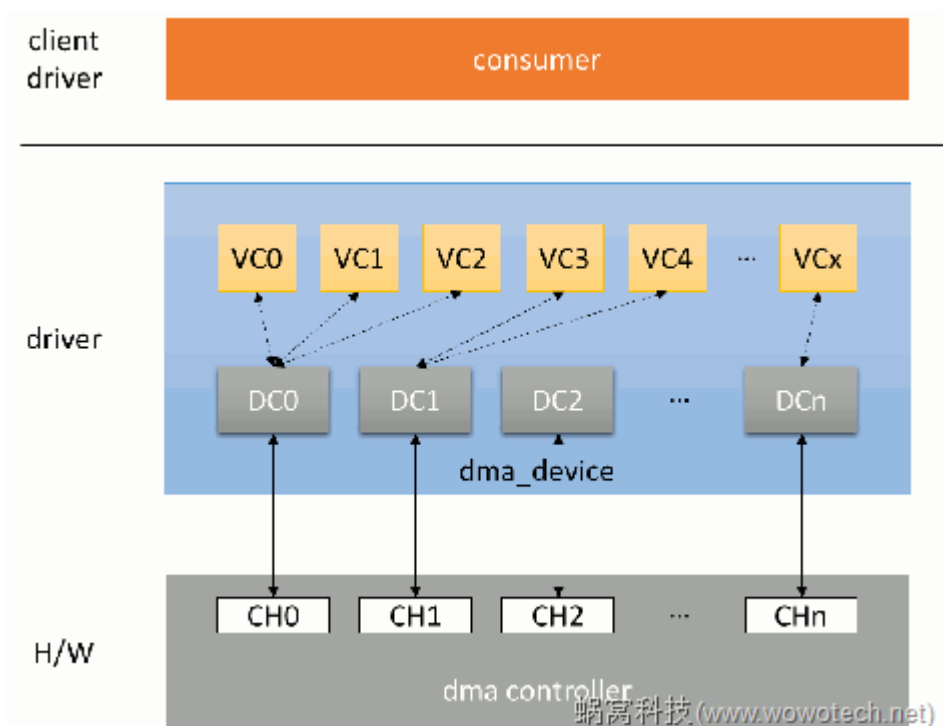
This section will introduce how to write dma controller driver under the framework of linux kernel dmaengine from the perspective of the provider .

### 4.1 Software framework driven by dma controller

The essence of the device driver is to describe and abstract the hardware, and then provide a friendly interface for the consumer to operate the hardware. The dma controller driver is no exception. What it does is nothing more than:

- Abstract and control the DMA controller
- Management DMA Channel (can be physical Channel , or virtual Channel ), and client driver to provide a friendly, easy to use interface
- Take the DMA channel as the operating object, respond to the client driver ( consumer ) transmission request, and control the DMA controller to perform the transmission

Of course, in accordance with convention, in order to unify the API provided to consumers and reduce the difficulty of DMA controller driver development (from essay questions to fill-in-the-blank questions), dmaengine framework provides a set of controller driver development framework. The main ideas are :



- using struct `dma_device` abstract the DMA Controller , Controller Driver as long as necessary to fill in the fields of the structure can be completed dma controller driver development
- using struct `dma_chan` (FIG. `DCn` ) physical abstraction the DMA channel (in FIG. `CHn` ), a physical channel and a controller number of channels can provide one-
- physics-based the DMA Channel , using struct `virt_dma_cha` abstracted virtual DMA Channel (figure `VCx` ). Multiple virtual channels can share a physical channel and perform time-sharing transmission on this physical channel
- 在这些数据结构上，提供一些 APIs 方便驱动开发使用

For the description of the above three data structures, please refer to the following introduction.

At the same time , we will introduce related API , controller driver development ideas and steps , and important processes related to controller driver in dmaengine later in this section .

## 4.2 Description of main data structure

### 4.2.1 struct `dma_device`

The struct `dma_device` used to abstract the dma controller is a complex data structure , but there are not many things that the dma controller driver cares about .

Header file : `include/linux/dmaengine.h`

struct <code>dma_device</code>	Comment
<code>channels</code>	<p>A linked list header used to mount all dma channel s supported by the controller . When initializing, the dma controller driver first calls <code>INIT_LIST_HEAD</code> to initialize it .</p> <p>Add c hannel time , call <code>list_add_tail</code> all channel added to the list head er .</p>

src_addr_widths dst_addr_widths	A bitmap indicates which widths of src /dst types are supported by the controller , including 1 , 2 , 3 , 4 , 8 , 16 , 32 , 64 ( bytes ) , etc. (For details, please refer to the definition of enum dma_slave_buswidth )
directions	A bitmap indicates which transfer directions the controller supports, including DMA_MEM_TO_MEM , DMA_MEM_TO_DEV , DMA_DEV_TO_MEM , DMA_DEV_TO_DEV , for details, please refer to the definition and comments of enum dma_transfer_direction
max_burst	The largest burst transmission size supported
descriptor_reuse	Indicate whether the transmission description of the controller can be reused (the client driver can only obtain the transmission description once, and then perform multiple transmissions)
residue_granularity	granularity of the transfer residue reported to dma_set_residue. This can be either: <ul style="list-style-type: none"> <li>◦ Descriptor: your device doesn't support any kind of residue reporting. The framework will only know that a particular transaction descriptor is done.</li> <li>◦ Segment: your device is able to report which chunks have been transferred</li> <li>◦ Burst: your device is able to report which burst have been transferred</li> </ul>
cap_mask	<p>A bitmap , used to indicate the capabilities of the dma controller (what kind of DMA transfer can be performed ) .</p> <ul style="list-style-type: none"> <li>➤ DMA_MEMCPY , memory copy can be performed</li> <li>➤ DMA_XOR</li> <li>➤ Wait , please refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#supported-transaction-types">official website description for (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#supported-transaction-types)</a> details (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#supported-transaction-types)</li> </ul> <p>In addition, the definition of the bitmap needs to correspond to the callback function in the form of device_prep_dma_XXX (the bitmap supports a certain transmission type, and the callback function corresponding to that type must be provided)</p>
device_alloc_chan_resources device_free_chan_resources	<p>When the client driver applies for / releases the dma channel , dmaengine will call the corresponding alloc/free callback function of the dma controller driver to prepare the corresponding resources. The specific resources to be prepared need to be determined by the dma controller driver according to the actual situation of the hardware (this is the rogue of the dmaengine framework , haha ~ ) .</p> <p>In these two functions, you can sleep.</p>

device_prep_dma_xxx	<p>When the client driver obtains the transmission descriptor through the dmaengine_prep_xxx API , dmaengine will directly call back the corresponding device_prep_dma_xxx interface of the dma controller driver . As for what to do in these callback functions, please refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) . (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations)</p> <p>This function may be called in interrupt context , so it cannot sleep</p>
device_issue_pending	<p>When the client driver calls dma_async_issue_pending to start the transmission, it will call the callback function</p> <p>Refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) .</p>
device_tx_status	<p>Refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) .</p>
device_config	<p>The client driver calls dmaengine_slave_config[When configuring the dma channel , dmaengine will call the callback function .</p> <p>Refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) .</p>
device_pause device_resume device_terminate_all	<p>When the client driver calls APIs such as dmaengine_pause , dmaengine_resume , dmaengine_terminate_xxx , dmaengine will call the corresponding callback function .</p> <p>Refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) .</p>
device_synchronize	<p>Refer to the <a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations">official website description</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/provider.html#device-operations) .</p>

Summary : dmaengine 's abstraction and encapsulation of dma controller is only a thin layer , only some callback functions are encapsulated, which is implemented by dma controller driver .

dmaengine itself does not have much operation logic.

## 4.2.2 struct dma\_chan

dma\_chan is used to abstract dma channel , and its content is:

Header file : include/linux/dmaengine.h

struct dma_device	Comment
struct dma_device *device	Point to the dma controller where the channel is located
dma_cookie_t cookie	The last cookie returned by the dma controller driver to the client when the client driver uses the channel as the operating object to obtain the transmission descriptor

dma_cookie_t completed_cookie	The cookie of the last completed transfer on this channel . The dma controller driver can call the auxiliary function dma_cookie_complete to set its value when the transfer is complete .
int chan_id	
struct dma_chan_dev *dev	
struct list_head device_node	Used to add the channel to the channel list of dma_device
struct dma_chan_percpu __percpu *local	
int client_count	
int table_count	
/* DMA router */	
struct dma_router *router	
void *route_data	
void *private	

### 4.2.3 struct virt\_dma\_cha

struct virt\_dma\_chan to abstract a virtual DMA channel , a plurality of virtual channel may share a physical channel , the scheduling software by a plurality of transmission requests, a plurality of virtual channel transmission in serial physical channel is completed on. The definition of the data structure is as follows:

Header file : drivers/dma/virt-dma.h

struct virt_dma_desc	Comment
struct dma_async_tx_descriptor tx	
struct list_head node	

struct virt_dma_chan	Comment
struct dma_chan chan	Used to deal with client driver (shield the difference between physical channel and virtual channel )
struct tasklet_struct task	Used to wait for the completion of the transmission on the virtual channel (because it is a virtual channel , the completion of the transmission can only be determined by the software)
void (*desc_free)(struct virt_dma_desc *)	
spinlock_t lock	
struct list_head desc_allocated struct list_head desc_submitted struct list_head desc_issued struct list_head desc_completed	Four linked list headers, used to store virtual channel descriptors in different states

struct virt_dma_desc *cyclic	Virtual channel descriptor , defined above, a table shown in FIG . Only the struct dma_async_tx_descriptor made a simple package

## 4.3 APIs

Damengine directly provides not many APIs to the dma controller driver (most of the logical interactions are located in the callback function of the struct dma\_device structure), mainly including:

Header file : include/linux/dmaengine.h

APIs for DMA controller	Comment
int <b>dma_async_device_register</b> (struct dma_device *device)	After the dma controller driver prepares the struct dma_device variable, it can call dma_async_device_register to register it in the kernel . This interface will perform a series of checks on the device pointer, and then further initialize it, and finally put it on a global linked list named dma_device_list for later use
void <b>dma_async_device_unregister</b> (struct dma_device *device)	Logout interface
<b>/* Auxiliary interface related to cookie */</b>	Because the cookie- related operations have many commonalities, dma engine provides some common implementations
void dma_cookie_init(struct dma_chan *chan)	Initialize the cookie and completed_cookie fields in the dma channel
dma_cookie_t dma_cookie_assign(struct dma_async_tx_descriptor *tx)	Assign a cookie to the transmission description ( tx )
void dma_cookie_complete(struct dma_async_tx_descriptor *tx)	When any one of the transmission ( TX time) is completed, the interface can be called, to update the corresponding transmission channel to completed_cookie field
enum dma_status dma_cookie_status(...)	Get the transmission status of the specified cookie on the specified channel ( chan )
void dma_run_dependencies(struct dma_async_tx_descriptor *tx)	The client can submit multiple dma transmissions with dependencies at the same time . Therefore, when a certain transmission ends, the dma controller driver needs to check whether there is a transmission that depends on the transmission, and if so, it transmits it. The process of checking and transmitting can be carried out with the help of this interface (the dma controller driver only needs to be called, which saves a lot of things)
<b>/* Auxiliary interface related to device tree */</b>	
extern struct dma_chan *of_dma_simple_xlate(...)	These two interfaces can be used to client device node related dma parsed fields, and acquires the corresponding dma Channel
struct dma_chan *of_dma_xlate_by_chan_id(...)	



## 4.4 Methods and steps to write a dma controller driver

The basic steps to write a dma controller driver include (regardless of the virtual channel situation):

- 1) Define a struct `dma_device` variable, and fill in the key fields according to the actual hardware situation.
- 2) According to the number of channels supported by the controller , define a struct `dma_chan` variable for each channel , and after necessary initialization, add each channel to the channels linked list of the struct `dma_device` variable .
- 3) According to the hardware characteristics, implement the necessary callback functions in the struct `dma_device` variable ( `device_alloc_chan_resources/device_free_chan_resources` , `device_prep_dma_xxx` , `device_config` , `device_issue_pending`, etc.).
- 4) Call `dma_async_device_register` to register the struct `dma_device` variable in the kernel .
- 5) When the client driver application dma channel (for example, through the device tree in dma acquisition node), dmaengine Core calls dma controller driver of `device_alloc_chan_resources` function, the Controller Driver needs in this interface will be the channel ready resources.
- 6) When the client driver configures a certain dma channel , the dmaengine core will call the `device_config` function of the dma controller driver . The controller driver needs to prepare the content that the client wants to configure in this function for subsequent transmission.
- 7) Before the client driver starts a transmission, it will pass the transmitted information to the controller driver through the `dmaengine_prep_slave_xxx` interface . The controller driver needs to prepare the content to be transmitted in the corresponding `device_prep_dma_xxx` callback, and return to the client driver a transmission descriptor .
- 8) Then, the client driver will call `dmaengine_submit` to submit the transfer to the controller driver . At this time, dmaengine will call the `tx_submit` callback function provided by the controller driver for each transfer descriptor . The controller driver needs to attach the descriptor to this function in this function. In the transmission queue corresponding to the channel .
- 9) When the client driver starts to transmit, it will call `dma_async_issue_pending` . The controller driver needs to submit all the transmission requests on the queue to the hardware in the corresponding callback function ( `device_issue_pending` ).
- 10) Wait.

## 5 DMA Buffer mapping framework

In order to use DMA Provider, there is an important topic , is to DMA Provider assign " special " of Buffer. DMA the Controller due to a hardware problem , only move data to a certain Buffer, these Buffer termed It is " DMA Buffer".

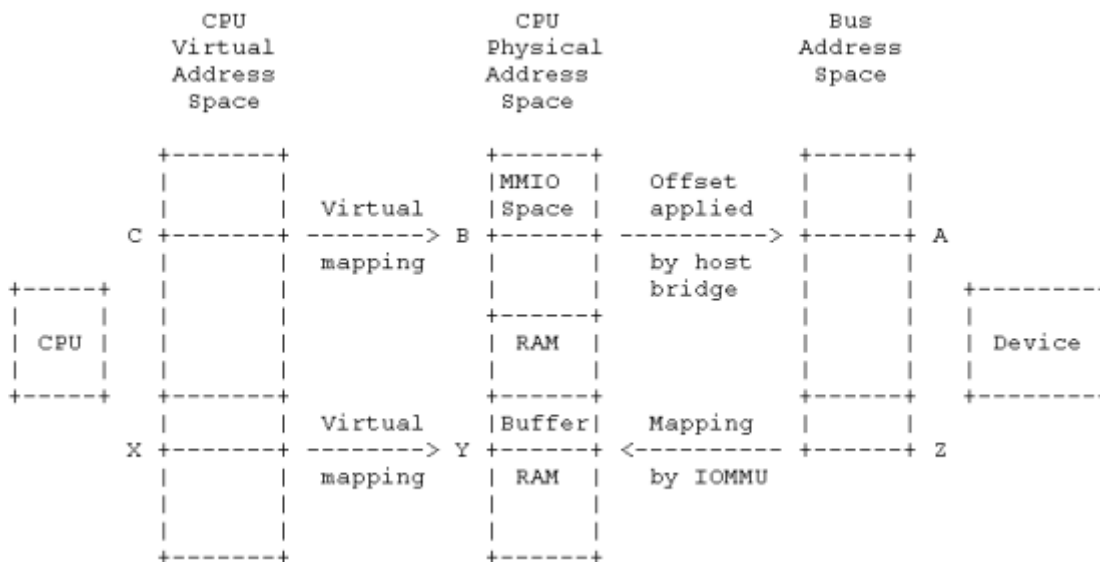
This section will be opened DMA Buffer mystery .

### 5.1 Physical address, virtual address , bus address and IOMMU

CPU as a controller , with its address bus to access the hardware memory. CPU addresses called "virtual address" , the address to access the address bus is called "physical address" , both by IOMMU mapping .

Within a processor , in addition to CPU, there are many on-chip peripherals , these peripherals some " semi-intelligent " is , they have their own address bus , and peripherals on the bus the bus access by mounting . Typical Such as PCI controller, NAND-Flash controller , and even I2C and SPI are also considered this type . This " semi-smart " address controller is called the " Bus address " . DMA the Controller is a typical " semi-intelligent " processor . Therefore DMA real scientific name should be called the address bus address .

We use an icon to further illustrate :






- The middle column represents the physical address space , of which MMIO Space is a subset of it , typically such as the "peripheral registers" in the ARM processor . RAM is another subset , such as the on-board DRAM.
- Left a virtual address space , the Linux kernel code used is the virtual address . Virtual addresses into physical addresses must be between MMU map: e.g. ioremap may be the MMIO space address B is mapped to C the CPU address space C , In this way, the CPU code can access the address C ; kmalloc() , vmalloc() and other similar interfaces can map the address Y to the address X , and the CPU code accesses the address X to access the DRAM.
- The right one is the bus address space , " semi-intelligent peripheral " via the address bus to access mounted thereon Device, the CPU can not directly access the address bus A , it can access " semi-intelligent peripherals " of the MMIO address , then These peripherals access address A instead of it .
- CPU needs to ask DRAM , " semi-intelligent peripherals " also need to access DRAM . Some of these " semi-intelligent peripherals " can directly access the DRAM address , such as the DMA controller whose register address is located in the physical address space . Others can not directly access DRAM, for example, mounted on the bus "Device", its internal DMA controller can not directly access the DRAM. The CPU access RAM & IOMMU need when MMU Mapping , "Device" access DRAM can also be mapped by IOMMU (for example, address Y is mapped to address Z ) . MMU can map non-contiguous physical addresses to continuous virtual addresses , and IOMMU can also map non-contiguous physical addresses to continuous bus addresses .

It should be noted , DMA controller just one of many " semi-intelligent peripherals ," one of them , therefore it uses DMA address can only be considered a subset of the bus address .

## 5.2 What kind of system memory can be accessed by the DMA controller

As mentioned above, the DMA address is only a subset of the bus address, so which memory can be used as the DMA address?

- by Interface of the partner system (e.g. `__get_free_page *()`) or the like `kmalloc()` or `kmem_cache_alloc()` interface to allocate such a common memory allocation of physical addresses successive memory.
- Use `vmalloc()` allocated DMA buffer can directly use it? It is best not to do this, although forced use may be no problem, but it is more troublesome after all.
  -  , the page frame allocated by `vmalloc` is not continuous. If the underlying hardware requires continuous physical memory, the memory allocated by `vmalloc` cannot meet the hardware requirements.
  - Even if the underlying DMA hardware supports scatter-gather, the memory allocated by `vmalloc` still has other problems. What we set to the DMA controller must be the hardware address. The virtual address allocated by `vmalloc` has no linear relationship with the corresponding physical address (interfaces such as `kmalloc` or `__get_free_page*`, and the returned virtual address has a fixed offset relationship with the physical address.), to get its physical address, we need to traverse the page table to find it.
-  ■ the global variables defined in the driver be used for DMA? If compiled to the kernel, then the global variables are located in the data section or bss section of the kernel. When the kernel is initialized, a kernel image mapping is established, so the memory occupied by global variables is continuous, and VA and PA have a linear relationship with a fixed offset, so they can be used for DMA operations. However, when defining the DMA buffer of these global variables, we must carefully align the cacheline and handle the synchronization of operations between the CPU and the DMA controller to avoid cache coherence problems.
-  happens if the driver is compiled into a module? At this time, the globally defined DMA buffer in the driver is not in the linear mapping area of the kernel. Its virtual address is allocated by `vmalloc` when the module is loaded. Therefore, if the DMA buffer is larger than a page frame at this time, then we actually cannot. The continuity of the underlying physical address is guaranteed, and the linear relationship between VA and PA cannot be guaranteed. This is different from compiling to the kernel.
- By `kmap` you can do memory interface returns the DMA buffer it? No, the principle is similar to `vmalloc`, so I won't go into details here.
- block devices using I/O buffer and the network device transmitting and receiving data buffer is how to ensure that the memory may be made DMA operation it? The block device I/O subsystem and the network subsystem will ensure this when allocating buffers.

## 5.3 DMA mapping APIs

DMA mapping actually consists of two actions:

- 1) Allocate a DMA buffer
- 2) The DMA Buffer Map for the DMA controller can access address bus

If we have allocated a buffer in our own way, then action 1 is not needed, just use action 2 to convert it to a bus address.

In the Linux system, since the DMA differences in hardware, a different address requires more different hardware, in order to shield the user of this difference, the Linux kernel provides a "DMA mapping framework", to provide a unified user to the API.

Later we will detail these API. Before that, let's look at why strongly recommend the use of these API to allocate DMA Buffer.

### 5.3.1 Why use DMA mapping API

Although on some hardware platforms, you can allocate the DMA Buffer in your own way (for example, some hardware can directly use the hardware address of the buffer allocated by the partner system as the DMA Buffer), but not all platforms can do this.

DMA Buffer, in the final analysis is the allocation of a Buffer then get its bus address. Seemingly simple, but if you want to be able to write a guarantee on all hardware platforms DMA operational safety, proper driver than you think it might be Difficult.

- different hardware platforms processing CACH consistency problems are not the same, if you can not take full account of the drive CACH consistency problems, that may occur memory is corrupt things.
- Some platforms have dedicated hardware to handle the bus address (e.g. the IOMMU), which makes the DMA operation becomes simpler, but there may be more complex.
- different DMA controllers have different addressing capabilities, such as the X-86 some platforms DMA controller can only access low 32M memory.
- Sometimes we need to "Bounce Buffer", than as we need DMA memory move data from a controller can not access it (e.g., in the example above 32M memory). At this point we need to first copy the data to the DMA access to the "Bounce Buffer" inside, and then open the move. While doing significant damage performance, but this time we have no choice.
- Wait

Fortunately, Kernel provides us with a BUS-Architecture and the DMA-Independent Mapping Framework. It shields the underlying hardware differences for us, the user only needs to use it provides the API to process DMA Buffer, to guarantee compatibility and Security.

It is worth mentioning that, on some platforms, the kernel code provides the API:

```
unsigned long virt_to_bus(volatile void *address);  
void *bus_to_virt(unsigned long address);
```

While talking two API can be carried out between the virtual address and the bus address translation, but we strongly recommend not to use them, but with the "DMA mapping API". If the hardware platform has the IOMMU, or involve similar to "Bounce Buffer" problems, both API are not working properly.

### 5.3.2 Overview of DMA mapping APIs

In using these APIs time, you need to refer to the header file: `Linux / DMA-mapping.h`

The kernel system defines a new data type "`dma_addr_t`", which is used to represent the DMA bus address. For users, it should be treated as a black box, and we do not need to modify its internal details. The only thing the user can do The thing is to get it, and then set it to the DMA controller.

The kernel provides two types of DMA mapping:

#### ➤ Coherent DMA mappings

We generally use it to allocate & map a DMA Buffer.

Under normal circumstances, we will drive the probe call upon it to obtain Buffer, releasing them only until the driver uninstall, so these Buffer life cycle will be longer, so also known such Buffer static Buffer (of course, this is you need the API distribution Buffer, immediately release the No Buffer).

Further, this is the PI distribution buffer while the CPU and DMA visible control (corresponding, later we will see other mapping embodiment acquired buffer, at the same time, can only be seen either), thus these buffers generally located cache-coherent area, relatively speaking, they are relatively expensive.

### ➤ Streaming DMA mappings

As we already have a buffer time, usable type PI which maps to DMA buffer.

Such API mapping buffer at a time, only by CPU or DMA party access controller which. If you need access to role swap, we need to call some API.

On some hardware platforms, the use of streaming mappings can significantly improve performance (perhaps because of cache consistency, streaming mappings a way that the CPU to continue using cache). And in this way to buffer holding period is shorter (if needed mapping, is not required when unmapping), allows multiple drivers have the opportunity to use the same buffer (eg multiple drives may all want the same peripheral register is mapped to DMA buffer), and the kernel developers recommend priority Consider using the streaming mappings method. However, this method is a bit more complicated in code writing and requires strict compliance with some rules.

Well, let's take a closer look at these two types of API.

## 5.3.3 Coherent DMA mappings API

Consistent DMA mapping has the following two characteristics:

- continued use of the DMA buffer (not disposable), therefore Consistent DMA always (but not absolute) in the initialization of the Map, the shutdown time unmap.
- CPU and DMA controller initiate parallel access to the DMA buffer, there is no need to consider the impact of the cache. That is to say, no software is required for cache operation. Both the CPU and the DMA controller can see each other's updates to the DMA buffer. In fact, the Consistent in the consistent DMA mapping can actually be called coherent, that is, cache coherent.

By default, the coherent mask is set to the lower 32 bit (0xFFFFFFFF). Even if the default value is OK, we also recommend that you set the coherent mask in the driver through the interface (see "DMA Addressing Restrictions" for details).

Scenarios where Consistent DMA mapping is generally used include:

- 1) The network card driver and the network card DMA controller often interact through some descriptors in the memory (which form a ring or chain). The memory that stores the descriptors generally uses Consistent DMA mapping.
- 2) the SCSI on the adapter hardware DMA may be some data structures (main memory mailbox command interaction), these saved mailbox command of memory generally used the Consistent DMA Mapping.
- 3) Some external devices have the ability to execute the firmware code (microcode) on the main memory. The main memory for storing the microcode generally uses Consistent DMA mapping.

The above examples have the same characteristics: CPU modifications to the memory can be immediately perceived by the device, and vice versa. Consistency mapping can guarantee this.

It should be noted that consistent DMA mapping does not mean that tools such as memory barriers are not needed to ensure memory order. The CPU may rearrange memory access instructions on consistent memory for performance. For example: If DMA consistent memory there are two on the Word, are word0 and word1, for device side, must ensure word0 to update, and then have to word1 update, then you need to write code like this:

```
desc->word0 = address;
wmb();
desc->word1 = DESC_VALID;
```

Only in this way can it be ensured that the device drivers can work normally on all platforms.

In addition, on some platforms, after modifying the DMA Consistent buffer , your driver may need to flush the write buffer so that the device side can perceive the memory change. This action is similar to the flush write buffer action in the PCI bridge .

Let's take a look at the details of A PI in detail .

## **dma\_alloc\_coherent**

A driver can set up a coherent mapping with a call to dma\_alloc\_coherent:


```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
*dma_handle, int flag) ;
```

- This function handles both the allocation and the mapping of the buffer.
- The first two arguments are the device structure and the size of the buffer needed.
- The function returns the result of the DMA mapping in two places:
  - The return value from the function is a kernel virtual address for the buffer, which may be used by the driver;
  - the associated bus address, meanwhile, is returned in dma\_handle.
- Allocation is handled in this function so that the buffer is placed in a location that works with DMA; usually the memory is just allocated with get\_free\_pages (but note that the size is in bytes, rather than an order value).  
The flag argument is the usual GFP\_ value describing how the memory is to be allocated; it should usually be GFP\_KERNEL (usually) or GFP\_ATOMIC (when running in atomic context).
- The A the PI distribution B uffer is P AGE\_SIZE aligned , if you do not need a big driving the DMA Buffer , you can select the description below dma\_pool Interface

## **dma\_free\_coherent**

When the driver needs UMAP and releases the dma buffer , it needs to call the following interface:

```
void dma_free_coherent(struct device *dev, size_t size, void *vaddr, dma_addr_t
dma_handle) ;
```

-  dev and size parameters of this interface function have been described above, and the two parameters cpu\_addr and dma\_handle are the two address return values of the dma\_alloc\_coherent() interface.
- needs to be emphasized is this: and dma\_alloc\_coherent different, dma\_free\_coherent not be called in interrupt context. (Because on some platforms, the operation of free DMA will trigger the operation of TLB maintenance (thus triggering the communication between CPU cores ). If the IRQ is turned off, it will be locked in the code of SMP IPI ).

## **DMA pools**

If your driver requires a lot of small dma buffers , then dma pool is the most suitable mechanism for you. This concept is similar to kmem\_cache , \_\_get\_free\_pages often obtains a continuous page frame , while kmem\_cache wholesales a large number of page frames and then "retails" it by itself. The dma pool is to obtain a large block of

consistent DMA memory through the `dma_alloc_coherent` interface , and then the driver can call `dma_pool_alloc` to divide a small block of dma buffer from that large block of DMA memory for its own use.

Related A the PI as follows :

Header file : `linux/dmapool.h`

APIs for DMA pool	Comment
<pre>struct dma_pool *dma_pool_create(const char *name, struct device *dev, size_t size, size_t align, size_t allocation);</pre>	<p>A DMA pool must be created before use</p> <ul style="list-style-type: none"> <li>➤ name is a name for the pool</li> <li>➤ dev is your device structure</li> <li>➤ size is the size of the buffers to be allocated from this pool</li> <li>➤ align is the required hardware alignment for allocations from the pool (expressed in bytes)</li> <li>➤ allocation is, if nonzero, a memory boundary that allocations should not exceed</li> </ul> <p>If allocation is passed as 4096, for example, the buffers allocated from this pool do not cross 4-KB boundaries</p>
<pre>void dma_pool_destroy(struct dma_pool *pool);</pre>	<p>You should return all allocations to the pool before destroying it</p>
<pre>void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t *handle);</pre>	<p>Allocate a DMA buffer:</p> <ul style="list-style-type: none"> <li>➤ mem_flags is the usual set of GFP_ allocation flags</li> <li>➤ If all goes well, a region of memory (of the size specified when the pool was created) is allocated and returned</li> <li>➤ As with <code>dma_alloc_coherent</code>, the address of the resulting DMA buffer is returned as a kernel virtual address and stored in handle as a bus address</li> </ul>
<pre>void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);</pre>	<p>Returned unneeded buffers to the pool</p>

### 5.3.4 Streaming DMA mappings API

Streaming DMA mapping is a one-time, generally require DMA when the transfer was carried Mapping , once the DMA transfer is complete, immediately `ummap` (unless you use `dma_sync_*` interface, which will be described below). And the hardware can be optimized for sequential access.

The streaming here can be considered asynchronous , or it does not belong to the scope of coherent memory .

Scenarios that generally use streaming DMA mapping include:

- DMA buffer used by the network card for data transmission
- Various data buffers in the file system. The data in these buffers will eventually be read and written to the SCSI device. Generally speaking, the driver will accept these buffers , then perform streaming DMA mapping , and then interact with the DMA on the SCSI device .

Let's look at the details next .

#### DMA transfer direction

When setting up a streaming mapping, you must tell the kernel in which direction the data is moving.

It is defined as follows :

enum dma_data_direction	Comment
DMA_TO_DEVICE	From memory ( dma buffer ) to device
DMA_FROM_DEVICE	From device to memory ( dma buffer )
DMA_BIDIRECTIONAL	Two-way transmission
DMA_NONE	Related to debugging . Before the driver knows the precise DMA direction , the transfer direction can be set to this state .  If finally this type of data is transmitted , it will trigger "Kernel Panic"

We strongly require the driver to know the appropriateness of the DMA transfer direction, and specify precisely DMA\_TO\_DEVICE or DMA\_FROM\_DEVICE .

If you really do not know the specific operation direction, it is also possible to set it to DMA\_BIDIRECTIONAL , which means that the DMA operation can perform data movement in any direction. Your platform needs to ensure that this can make DMA work properly, of course, this may introduce some performance overhead.

In addition to potential platform-related performance optimizations, precisely specifying the DMA operation direction has another advantage that is convenient for debugging. Some platforms actually have a write permission Boolean value in the page table (referring to the page table that maps the bus address to the physical address) when creating the DMA mapping . This value is very similar to the page protection in the user program address space. When the DMA controller hardware detects a violation of permission settings (at this time, the DMA buffer is set to the DMA\_TO\_DEVICE type, in fact, the DMA controller can only read the DMA buffer ), such a platform can write errors to the kernel log, which facilitates debugging .

Only streaming mappings will indicate the DMA operation direction. The DMA operation direction implied by consistent DMA mapping is DMA\_BIDIRECTIONAL . Let's give an example of streaming mappings : In the network card driver, if you want to send data, you need to specify the operation direction of DMA\_TO\_DEVICE when map/umap , and when receiving data packets, map/umap needs to specify the DMA operation direction is DMA\_FROM\_DEVICE .

Next, look at the details of API .

### Map single buffer

APIs for single buffer mapping	Comment
<code>dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);</code>	When you want a certain BUF ( parameter * Buffer point this BUF, it is CPU virtual address accessible ) is converted to DMA when accessible address , can use the API.  The return value is hardware available DMA address ; but if an error occurs , the return value is NULL.
<code>void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size, enum dma_data_direction direction);</code>	When the transfer is completed (case I usually receive a DMA transfer complete interrupt) , use this API released mapping. Parameters Size and Direction have the MAP coincide with the beginning .  NOTE : When a BUF is MAP after , only the DMA hardware may use it , the CPU can not read it again . Only when this BUF is Umap later , the CPU can access it .



void <b>dma_sync_single_for_cpu</b> (struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);	Upon receiving DMA transmission complete interrupt, if CPU need access to buf, but to continue to access the complete DMA transfer, this time we can not Unmap, but the use of this API. Called API after, the CPU can access this buf up. But this time DMA hardware can no longer access it.
void <b>dma_sync_single_for_device</b> (struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);	When the CPU After the visit, this API to exchange buf role. In this case only the DMA hardware can access it, the CPU can no longer access it.

In the use of streaming DMA mappings time, there are some rules to follow:

- The use of buf must be consistent with the given transmission direction.
- described in the API description, the CPU and DMA hardware access to buf must comply with relevant conventions.
- DMA hardware moving buf process of, not Unmap buf.

You might wonder why buf is in the API after, the CPU can no longer access it? The reason is two-fold:

- Consider first CACH problem. Suppose the CPU wants to transfer a block data DMA, it will first piece of data is written to mem buf. However, due to CACH exists, this data may be temporarily stored in the CACH in, further No written in MEM. when the call `Dma_map_single` time, the underlying implementation code force Flush, the CACH data brush into MEM. If after this the CPU directly updated BUF content, then the "update" may be present only in CACH in, Mem is not actually written.
- Secondly, consider the bounce buffer problem. Suppose a buf which address beyond DMA hardware addressability, this time in the API piece buf what scene appear? On some platforms will simply fail, but some in platforms DMA create an addressable hardware within the range of Bounce Buffer, then BUF contents inside COPY to Bounce Buffer. obviously, in this case, if the CPU in API after the visit buf, only the content of "original buf" is updated.

Speaking of bounce buffer, by the way, it is also the reason why the "transmission direction" should be used correctly:

- For DMA\_TO\_DEVICE, will only `dma_map_single` do when COPY action.
- For DMA\_FROM\_DEVICE, will only `dma_unmap_single` do when COPY action.
- For DMA\_BIDIRECTIONAL, made the above both cases COPY action.

Incorrect use of "transmission direction" will not only affect performance, but may also cause errors.

## Map single page

The `dma_map_single` function uses the CPU pointer (virtual address) when performing DMA mapping, which leads to a drawback of this function: it cannot use HIGHMEM memory for mapping. In view of this, the map/unmap interface provides another similar interface. This interface does not use CPU pointers, but uses page and page offset for DMA mapping:

APIs for single page mapping	Comment
dma_addr_t <b>dma_map_page</b> (struct device *dev, struct page *page, unsigned long offset, size_t size, enum dma_data_direction direction);	Map a page. Although Offset and size allows you only in the API in the page part of, but you should avoid doing so. If the map only contains part of the cache line, this may cause "memory corruption"

void <b>dma_unmap_page</b> (struct device *dev, dma_addr_t dma_address, size_t size, enum dma_data_direction direction);	Unmmap a page
--	---------------

## Scatter/gather mappings

Scatter/gather mappings are a special type of streaming DMA mapping.

Suppose you have several buffers, all of which need to be transferred to or from the device. This situation can come about in several ways:

- from a readv or writev system call
- a clustered disk I/O request
- a list of pages in a mapped kernel I/O buffer.

You could simply map each buffer, in turn, and perform the required operation, but there are advantages to mapping the whole list at once.

The advantages are:

- Many devices can accept a scatterlist of array pointers and lengths, and transfer them all in one DMA operation; for example, “zero-copy” networking is easier if packets can be built in multiple pieces.
- Another reason to map scatterlists as a whole is to take advantage of systems that have IOMMU. On such systems, physically discontinuous pages can be assembled into a single, contiguous array from the device’s point of view.  
This technique works only when the entries in the scatterlist are equal to the page size in length (except the first and last), but when it does work, it can turn multiple operations into a single DMA, and speed things up accordingly.
- Finally, if a bounce buffer must be used, it makes sense to coalesce the entire list into a single buffer (since it is being copied anyway).

So now you’re convinced that mapping of scatterlists is worthwhile in some situations.

Next, let us see some details.

### *scatterlist*

struct scatterlist [] is a data structure associated with architecture, defined in <asm / scatterlist.h> in .

Its main function is used to describe a plurality of buffers. You can think of it as an array or linked list, each entry representative of a buffer.

Different architectures have different implementations, but generally speaking, every entry has several elements as follows:

- struct page \*page : the buffer to be used in the scatter/gather operation.
- unsigned int length;
- unsigned int offset : The length of that buffer and its offset within the page.

In the following API before, we need to prepare a struct scatterlist [] array, and fill it for each entry.

### *APIs*

APIs for multi-buffer mapping	Comment
-------------------------------	---------

<pre>int <b>dma_map_sg</b>(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction)</pre>	<p>To map a scatter/gather DMA operation, your driver should set the page, offset, and length fields in a struct scatterlist entry for each buffer to be transferred. Then call this API.</p> <p>where nents is the number of scatterlist entries passed in.</p> <p>The return value is the number of DMA buffers to transfer; it may be less than nents.</p> <p>The bus address and length of each buffer are stored in the struct scatterlist entries, you can obtain them with following Macros.</p>
<pre>dma_addr_t <b>sg_dma_address</b>(struct scatterlist *sg);</pre>	<p>Returns the bus (DMA) address from this scatterlist entry</p>
<pre>unsigned int <b>sg_dma_len</b>(struct scatterlist *sg);</pre>	<p>Returns the length of this buffer.</p> <p>Again, remember that the length of the buffers to transfer may be different from what was passed in to dma_map_sg</p>
<pre>void <b>dma_unmap_sg</b>(struct device *dev, struct scatterlist *list, int nents, enum dma_data_direction direction);</pre>	<p>Once the transfer is complete, a scatter/gather mapping is unmapped with a call to dma_unmap_sg.</p> <p>Note that nents must be the number of entries that you originally passed to dma_map_sg and not the number of DMA buffers the function returned to you.</p>
<pre>void <b>dma_sync_sg_for_cpu</b>(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);</pre> <pre>void <b>dma_sync_sg_for_device</b>(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);</pre>	<p>The function is consistent with the description in "map single buffer".</p>

## Error handling

DMA address space is limited on some CPU architectures, so allocation and mapping may cause errors. We can use the following methods to determine whether an error has occurred:

- Check if dma\_alloc\_coherent() returns NULL or dma\_map\_sg returns 0 .
  - Check that dma\_map\_single and dma\_map\_page return dma address (via dma\_mapping\_error function)
- Although not all DMA mapping implementations support the dma\_mapping\_error interface ( calling the dma\_mapping\_error function will actually call the mapping\_error member function in the underlying dma\_map\_ops operation function set ), it is still a good practice to call it for error handling. The advantage of this is to ensure that the DMA mapping code can work normally in all DMA implementations without relying on the details of the underlying implementation.

The sample code is as follows :

```
dma_addr_t dma_handle;

dma_handle = dma_map_single(dev, addr, size, direction);
if (dma_mapping_error(dev, dma_handle)) {
    goto map_error_handling;
```

```
}
```

- when mapping a plurality of B UF / Page time, if the intermediate occurs mapping error , then the need for those already mapped the B UF / Page performed unmap operation.

### 5.3.5 DMA addressing restriction API

Does your device have DMA addressing restrictions? Different hardware platforms have different configuration methods. Some platforms have no restrictions. Peripherals can access every Byte of system memory , while others cannot. For example: The system bus has a 32 th 'bit , and your device through DMA can only be driven at a low 24 -bit address, in this case, a peripheral initiates the DMA when in operation, can only access 16M less system memory. If the device has a restriction on DMA addressing, the driver needs to notify the kernel of this restriction. If the driver does not notify the kernel, the kernel considers that the peripheral DMA can access all 32-bit address lines of the system bus by default . For the 64 bit platform, the situation is similar and will not be repeated here.

Whether there is a DMA addressing restriction is related to the hardware design, and sometimes the standard bus protocol also stipulates this. For example: The PCI-X specification stipulates that all PCI-X devices must support 64-bit addressing.

If there are addressing restrictions, then in the probe function of the peripheral driver , you need to ask the kernel to see if there is a DMA controller that can support the addressing restrictions of this peripheral. Although there is a default addressing restriction setting, it is better to perform related processing in the probe function. At least this shows that you have considered addressing restrictions for your peripherals.

Once the device DMA addressing limit is determined , we can set it through the following interface:

APIs for dma mask	Comment
int dma_set_mask_and_coherent(struct device *dev, u64 mask);	D MA Map has C oherent and S treaming both . Assumed that in both c ases , the DMA addressing limits consistent , then we can use this A the PI is set . If the limit address is inconsistent in both cases , then we are going to use the following A PI.
int dma_set_mask(struct device *dev, u64 mask);	Set streaming type DMA address mask
int dma_set_coherent_mask(struct device *dev, u64 mask);	Set coherent type DMA address mask

If calling these interfaces returns 0 , it means everything is OK , and the DMA operation from the device to the memory of the specified mask can be supported by the system (including DMA controller , bus layer, etc.).

If the return value is non- zero , it means that such DMA addressing cannot be completed correctly. If you force it to do so, it will have unpredictable consequences.

Another common scenario is for devices with 64 -bit addressing capabilities. Generally speaking, we will first try to set a 64 -bit address mask, but it may fail at this time, so we will try to reduce the mask to 32 bits. The reason why the kernel fails when setting the 64 -bit mask is not because the platform cannot perform 64 -bit addressing, but simply because 32 -bit addressing is more efficient than 64 -bit addressing. For example, on the SPARC64 platform, PCI SAC addressing has better performance than DAC addressing.

The driver must check the return value. If the return value is not 0 , it is recommended to modify the mask or not use DMA .

The following code describes how to determine the address mask of streaming type DMA :

```

int using_dac;

if (!dma_set_mask(dev, DMA_BIT_MASK(64))) {
    using_dac = 1;
} else if (!dma_set_mask(dev, DMA_BIT_MASK(32))) {
    using_dac = 0;
} else {
    dev_warn(dev, "mydev: No suitable DMA available\n");
    goto ignore_this_device;
}

```

Setting the coherent type DMA address mask is similar, so I won't go into details here.

It should be noted that: the coherent address mask is always equal to or less than the streaming address mask. Therefore, generally speaking, we only need to set the streaming address mask successfully, then use the same mask or a smaller mask. Setting the coherent address mask will always succeed .

### 5.3.6 PCI double-address cycle mappings API

Normally, the DMA support layer works with 32-bit bus addresses, possibly restricted by a specific device's DMA mask.

The PCI bus, however, also supports a 64-bit addressing mode, the double-address cycle (DAC).

The generic DMA layer does not support this mode for a couple of reasons:

- ✓ the first of which being that it is a PCI-specific feature.
- ✓ Also, many implementations of DAC are buggy at best, and, because DAC is slower than a regular, 32-bit DMA.

Even so, there are applications where using DAC can be the right thing to do: if you have a device that is likely to be working with very large buffers placed in high memory, you may want to consider implementing DAC support.

This support is available only for the PCI bus, so PCI-specific routines must be used. To use DAC, your driver must include **<linux/pci.h>**.

APIs for DAC support	Comment
int <b>pci_dac_set_dma_mask</b> (struct pci_dev *pdev, u64 mask);	Set addressing restrictions . You can use DAC addressing only if this call returns 0.
dma64_addr_t <b>pci_dac_page_to_dma</b> (struct pci_dev *pdev, struct page *page, unsigned long offset, int direction);	A special type (dma64_addr_t) is used for DAC mappings. To establish one of these mappings, call this API.  DAC mappings, you will notice, can be made only from struct page pointers (they should live in high memory, after all, or there is no point in using them). They must be created a single page at a time.  The direction argument is the PCI equivalent of the enum dma_data_direction used in the generic DMA layer; it should be PCI_DMA_TODEVICE, PCI_DMA_FROMDEVICE, or PCI_DMA_BIDIRECTIONAL.

No release api	DAC mappings require no external resources, so there is no need to explicitly release them after use.
void <b>pci_dac_dma_sync_single_for_cpu</b> (struct pci_dev *pdev, dma64_addr_t dma_addr, size_t len, int direction);	The function is consistent with the description in `` m ap single buffer " .
void <b>pci_dac_dma_sync_single_for_device</b> (struct pci_dev *pdev, dma64_addr_t dma_addr, size_t len, int direction);	The function is consistent with the description in `` m ap single buffer " .

## 5.4 Implementation of DMA mapping

Above A PI just on providing a unified interface . Next it also needs someone implement these interfaces . DMA Mappings achieved is a big topic . Herein do not intend to expand , have the opportunity to elaborate

### 5.4.1 cma

C ma is one of the implementations . The code path is : drivers/base/dma-contiguous.c

If you need to use c ma, you need to turn on CONFIG\_DMA\_CMA = y when compiling .

In B ootargs in , we can "cma = xxxM" is specified in C mA reserved memory size . Parse the tag parameters are as follows :

```
static int __init early_cma(char *p)
{
    pr_debug("%s(%s)\n", __func__, p);
    size_cmdline = memparse(p, &p);
    if (*p != '@')
        return 0;
    base_cmdline = memparse(p + 1, &p);
    if (*p != '-') {
        limit_cmdline = base_cmdline + size_cmdline;
        return 0;
    }
    limit_cmdline = memparse(p + 1, &p);

    return 0;
}
early_param("cma", early_cma);
```

## 5.5 Other matters needing attention

### 5.5.1 Optimize data structure

On many platforms, `dma_unmap_{single,page}()` actually does nothing and is an empty function. Therefore, tracking the mapped dma address and its length is basically a waste of memory space. For the convenience of driver engineers to write code, we provide several practical tools (macro definitions). Without them, the driver will fully use `ifdef` or similar "work around". Below we do not introduce these macro definitions one by one, but give some sample codes, and the driver engineer can draw a picture according to the gourd.

**DEFINE\_DMA\_UNMAP\_{ADDR,LEN}** . Use this macro definition in the DMA buffer data structure, specific examples are as follows:

before:

```
struct ring_state {
    struct sk_buff *skb;
    dma_addr_t mapping;
    __u32 len;
};
```

after:

```
struct ring_state {
    struct sk_buff *skb;
    DEFINE_DMA_UNMAP_ADDR(mapping);
    DEFINE_DMA_UNMAP_LEN(len);
};
```

- The `CONFIG_NEED_DMA_MAP_STATE` configured differently, `DEFINE_DMA_UNMAP_{ADDR, LEN}` may be defined associated dma address members and length, it may be empty.

**dma\_unmap\_{addr,len}\_set()** . Use this macro definition to assign values, specific examples are as follows:

before:

```
ringp->mapping = FOO;
ringp->len = BAR;
```

after:

```
dma_unmap_addr_set(ringp, mapping, FOO);
dma_unmap_len_set(ringp, len, BAR);
```

**dma\_unmap\_{addr,len}()** , use this macro to access variables.

before:

```
dma_unmap_single(dev, ringp->mapping, ringp->len,
    DMA_FROM_DEVICE);
```

after:

```
dma_unmap_single(dev,
    dma_unmap_addr(ringp, mapping),
    dma_unmap_len(ringp, len),
    DMA_FROM_DEVICE);
```

The above code basically does not need to explain you will understand. In addition, we deal with dma address and len separately, because in some implementations, unmaping operations only require dma address information.

## 5.5.2 Issues that need attention in platform migration

If you are only a driver engineer and are not responsible for migrating Linux to a certain cpu arch , then you can actually ignore the following content.

### 1. Struct scatterlist requirements

If the cpu arch supports IOMMU (including software emulated IOMMU ), then you need to turn on the CONFIG\_NEED\_SG\_DMA\_LENGTH kernel option.

### 2. ARCH\_DMA\_MINALIGN

The code related to the CPU architecture must ensure that the buffer allocated by kmalloc is DMA-safe ( the buffer allocated by kmalloc may also be used for DMA buffer ). The correct operation of the driver and the kernel subsystem depends on this condition. If a cpu arch does not fully support DMA-coherent (for example, the hardware does not guarantee that the data in the cpu cache is equal to the data in the main memory ), then ARCH\_DMA\_MINALIGN must be defined . Through this macro definition, the buffer allocated by kmalloc can be guaranteed to be aligned on ARCH\_DMA\_MINALIGN , thereby ensuring that the DMA buffer allocated by kmalloc will not share a cacheline with other buffers . For specific examples, please refer to arch/arm/include/asm/cache.h .

In addition, please note: ARCH\_DMA\_MINALIGN is the alignment constraint of DMA buffer , you don't need to worry about the data alignment constraint of CPU ARCH (for example, some CPU arch requires some data objects to need 64-bit alignment).

## 6 DMA Buffer Share

Application of a drive module D MA Buffer, may be S Hare to other modules, or other drive subsystem . Kernel D MA-buf subsystem (<https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html>) provides related management framework .

## 7 DMA Test Guide

The DMA Test Guide (<https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html>) describes how to use the test code that comes with the kernel to test whether the D MA Controller driver we wrote is perfect .

## 8 RefLink

Link	Comment
<a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/index.html">https://www.kernel.org/doc/html/latest/driver-api/dmaengine/index.html</a>	Kernel DMA Engine Doc



<a href="http://www.wowotech.net/linux_kernel/dma_engine/overview.html">http://www.wowotech.net/linux_kernel/dma_engine/overview.html</a> (http://www.wowotech.net/linux_kernel/dma_engine/overview.html) <a href="http://www.wowotech.net/linux_kernel/dma_engine/api.html">http://www.wowotech.net/linux_kernel/dma_engine/api.html</a> (http://www.wowotech.net/linux_kernel/dma_engine/api.html) <a href="http://www.wowotech.net/linux_kernel/dma_controller_driver.html">http://www.wowotech.net/linux_kernel/dma_controller_driver.html</a> (http://www.wowotech.net/linux_kernel/dma_controller_driver.html)	anywhere DMA Engine Doc
<a href="https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html">https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html</a> (https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html)	Kernel DMA Test Guidance
<a href="https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html">https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html</a> (https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html)	Kernel dma-buf share Doc