



Linux DMA in Device Drivers

Based on 3.14 Linux kernel

John Linn, Strategic Applications Engineer, 10/2014

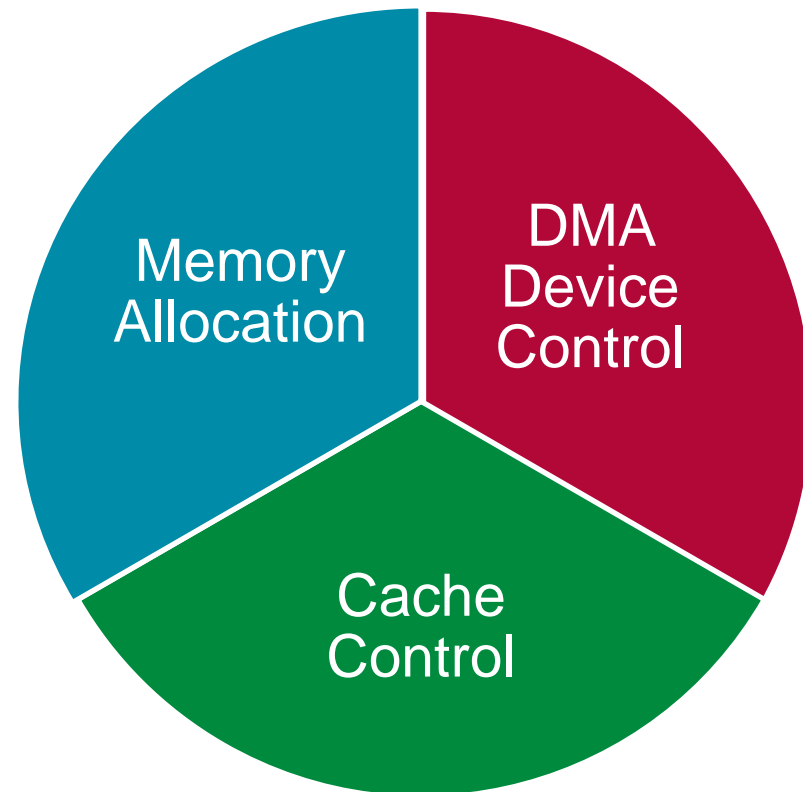
Agenda

- **Memory Allocation**
- **Kernel Configuration**
- **Cache Control**
- **DMA Engine**
- **DMA Engine Slave API**
- **DMA Kernel Driver Example**

- **Prerequisites**
 - **Knowledge of the Linux kernel in general such as building and configuring the kernel**
 - **Basic device driver experience in Linux**
 - **Experience with the C programming language**

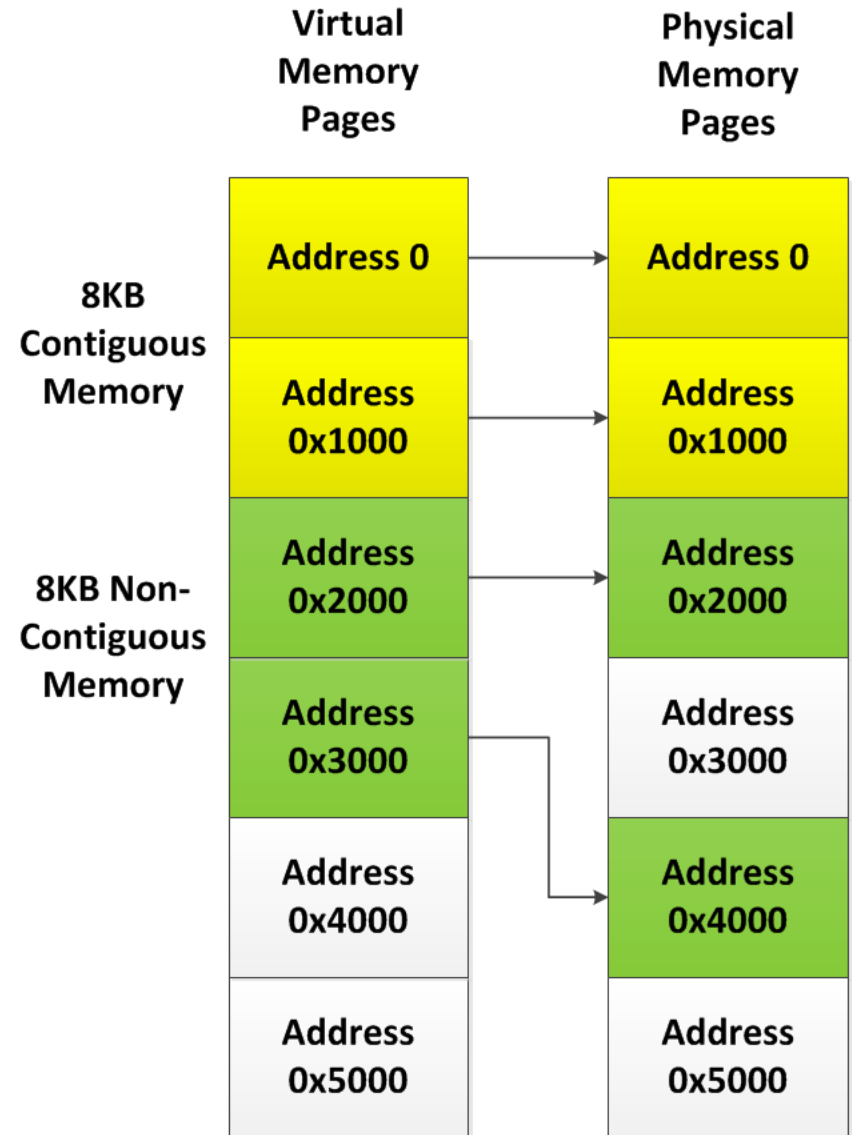
Introduction

- The goal of this session is to help users understand the Linux kernel DMA framework and how it can be used in a device driver
- DMA in Linux is designed to be used from a kernel space driver
- User space DMA is possible and is a more advanced topic that is not covered in this presentation
- The primary components of DMA include the DMA device control together with memory allocation and cache control



Memory Allocation For DMA – Part 1

- Linux provides memory allocation functions in the kernel
- The `vmalloc()` function allocates cached memory which is virtually contiguous but not physically contiguous
 - Not as useful for DMA without an I/O MMU
 - Zynq does not have an I/O MMU
- The `kmalloc()` function allocates cached memory which is physically contiguous
 - It is limited in the size of a single allocation
 - Testing showed 4 MB to be the limit, but it might vary with kernels



Memory Allocation For DMA – Part 2

- The `dma_alloc_coherent()` function allocates **non-cached** physically contiguous memory
 - The name coherent can be a confusing name (for me anyway)
 - The CPU and the I/O device see the same memory contents without any cache operations
 - Accesses to the memory by the CPU are the same as a cache miss when the cache is used
 - The CPU does not have to invalidate or flush the cache which can be time consuming
 - This function is the intended function for DMA memory allocation
 - There is another function, `dma_alloc_noncoherent()` but it's not really implemented so don't use it

Boot Time Memory Setup

- Memory can be reserved such that the kernel does not use it
 - MEM=512M on the kernel command line causes it to use only 512M of memory
 - The device tree memory can also be changed
- This is the oldest method allowing large amounts of memory to be allocated for DMA
- Drivers use `io_remap()` to map the physical memory address into the virtual address space
- There are multiple versions `io_remap()` which allow cached and non-cached
- These functions don't allocate any memory, they only map the memory into the address space in the page tables
- The Linux `io_remap()` function causes the memory to be setup as Device Memory in the MMU which should be slower than Normal Memory

Cortex A9 Memory Attributes – Device Memory

- **The Zynq TRM explains the details on pages 70 and 82**
- Each page of memory in Linux is setup with memory attributes based on its specific purpose
- The number and size of accesses are preserved, accesses are atomic, and will not be interrupted part way through
- Both read and write accesses can have side-effects on the system. Accesses are never cached
- Speculative accesses are never be performed
- Accesses cannot be unaligned
- The order of accesses arriving at Device memory is guaranteed to correspond to the program order of instructions which access device memory
- A write to Device memory is permitted to complete before it reaches the peripheral or memory component accessed by the write

Cortex A9 Memory Attributes – Normal Memory

- The processor can repeat read and some write accesses
- The processor can pre-fetch or speculatively access additional memory locations, with no side-effects (if permitted by MMU access permission settings)
- The processor does not perform speculative writes
- Unaligned accesses can be performed
- Multiple accesses can be merged by processor hardware into a smaller number of accesses of a larger size

Contiguous Memory Allocator (CMA)

- This is a newer feature of the kernel that some people may not know about
- There had been a lot of demand for larger memory buffers needed for many applications including multimedia
- CMA came into the kernel at version 3.5, about 2 years ago
- Is only accessible in the DMA framework via `dma_alloc_coherent()`
- Allows very large amounts of physically contiguous memory to be allocated
- Defaults to small amounts
 - Can be increased on the kernel command line (`CMA=`) which doesn't require a kernel rebuild
 - Can be increased in the kernel configuration

CMA Kernel Configuration

- The Xilinx kernel has CMA turned on by default, but this may vary with kernel versions
- Note that the Contiguous Memory Allocator must be turned on to see the configuration options in the device drivers configuration for DMA CMA (next slide)

```
/home/linnj/xilinx2/702-axi-dma-loopback/subsystems/linux/configs/kernel/config - Lin
u> Kernel Features
Kernel Features
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes,
<N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module <>
[*] Symmetric Multi-Processing
[*] Allow booting SMP kernel on uniprocessor systems (EXPERIMENTAL)
[ ] Support cpu topology definition
[ ] Architected timer support
[ ] Multi-Cluster Power Management
[ ] big.LITTLE support (Experimental)
Memory split (3G/1G user/kernel split) --->
(2) Maximum number of CPUs (2-32)
[*] Support for hot-pluggable CPUs
[ ] Support for the ARM Power State Coordination Interface (PSCI)
Preemption Model (Preemptible Kernel (Low-Latency Desktop)) --->
Timer frequency (100 Hz) --->
[ ] Compile the kernel in Thumb-2 mode
[*] Use the ARM EABI to compile the kernel
[ ] Allow old ABI binaries to run with this kernel (EXPERIMENTAL)
[*] High Memory Support
[ ] Allocate 2nd-level pagetables from highmem
[ ] Allow for memory compaction
[*] Enable bounce buffers
[*] Enable KSM for page merging
(4096) Low address space to protect from user allocation
[*] Cross Memory Support
[ ] Enable cleancache driver to cache clean pages if tmem is present
[ ] Enable frontswap to cache swap pages if tmem is present
[ ] Contiguous Memory Allocator
[ ] Memory allocator for compressed pages
[ ] Use kernel mem{cpy,set}() for {copy_to,clear}_user()
[ ] Enable seccomp to safely compute untrusted bytecode
[ ] Xen guest support on ARM (EXPERIMENTAL)
<Select> < Exit > < Help > < Save > < Load >
```

DMA CMA Kernel Configuration

```
/home/linnj/xilinx2/702-axi-dma-loopback/subsystems/linux/configs/kernel/config - L
u> Device Drivers > Generic Driver Options

Generic Driver Options
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes,
<N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < >

(/sbin/hotplug) path to uevent helper
[*] Maintain a devtmpfs filesystem to mount at /dev
[*] Automount devtmpfs at /dev, after the kernel mounted the rootfs
[*] Select only drivers that don't need compile-time external firmware
[*] Prevent firmware from being built
<*> Userspace firmware loading support
[ ] Include in-kernel firmware blobs in kernel binary
( ) External firmware blobs to build into the kernel binary
[*] Fallback user-helper invocation for firmware loading
[ ] Driver Core verbose debug messages
[ ] Managed device resources verbose debug messages
[*] DMA Contiguous Memory Allocator
    *** Default contiguous memory area size: ***
(16) Size in Mega Bytes (NEW)
    Selected region size (Use mega bytes value only) --->
(8) Maximum PAGE_SIZE order of alignment for contiguous buffers (NEW)
(7) Maximum count of the CMA device-private areas (NEW)

<Select> < Exit > < Help > < Save > < Load >
```

DMA Cache Control

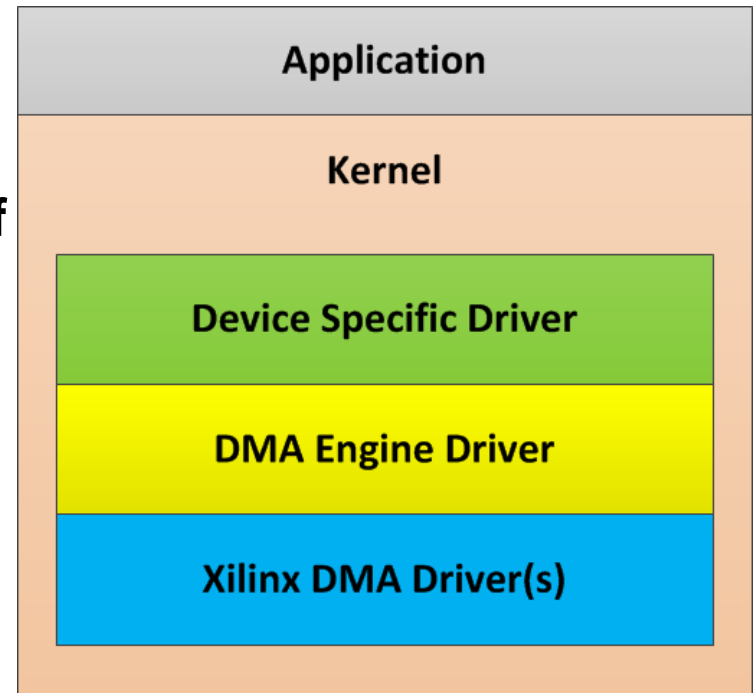
- Linux provides DMA functions for cache control of DMA buffers
- Cache control is based on the direction of DMA transfer, from memory to a device, from device to memory, or bidirectional
- DMA controllers in the PL are cache coherent in Zynq with ACP port
 - The HP ports are not cache coherent such that cache control is required
- For transfers from memory to a device, the memory must be flushed from the cache to memory before a DMA transfer is started
- For transfers from a device to memory, the cache must be invalidated after the transfer and before the CPU accesses memory
- `dma_map_single()` is provided to transfer ownership of a buffer from the CPU to the DMA hardware
 - It can cause a cache flush for the buffer in the memory to device direction
- `dma_unmap_single()` is provided to transfer ownership of a buffer from the DMA hardware back to the CPU
 - It can cause a cache invalidate for the buffer in the device to memory direction

Linux Kernel Details For DMA

- A **descriptor** is used to describe a DMA transaction such that a single data structure can be passed in an API.
 - A descriptor can also describe a DMA transaction to a DMA core such as the AXI DMA when it is built to use scatter gather
- A **completion** is a lightweight mechanism which allows one thread to tell another thread that a task is done
- A **tasklet** implements deferrable functionality and replaces older bottom half mechanisms for drivers
 - A function can be scheduled to run at a later time with a **tasklet**
- A **cookie** is an piece of opaque data which is returned from a function, then passed to yet a different function communicating information which only those functions understand
 - A DMA cookie is returned from **dmaengine_submit()** and is passed to **dma_async_is_tx_complete()** to check for completion of a specific DMA transaction
 - DMA cookies may also contain a status of a DMA transaction

Linux DMA Engine

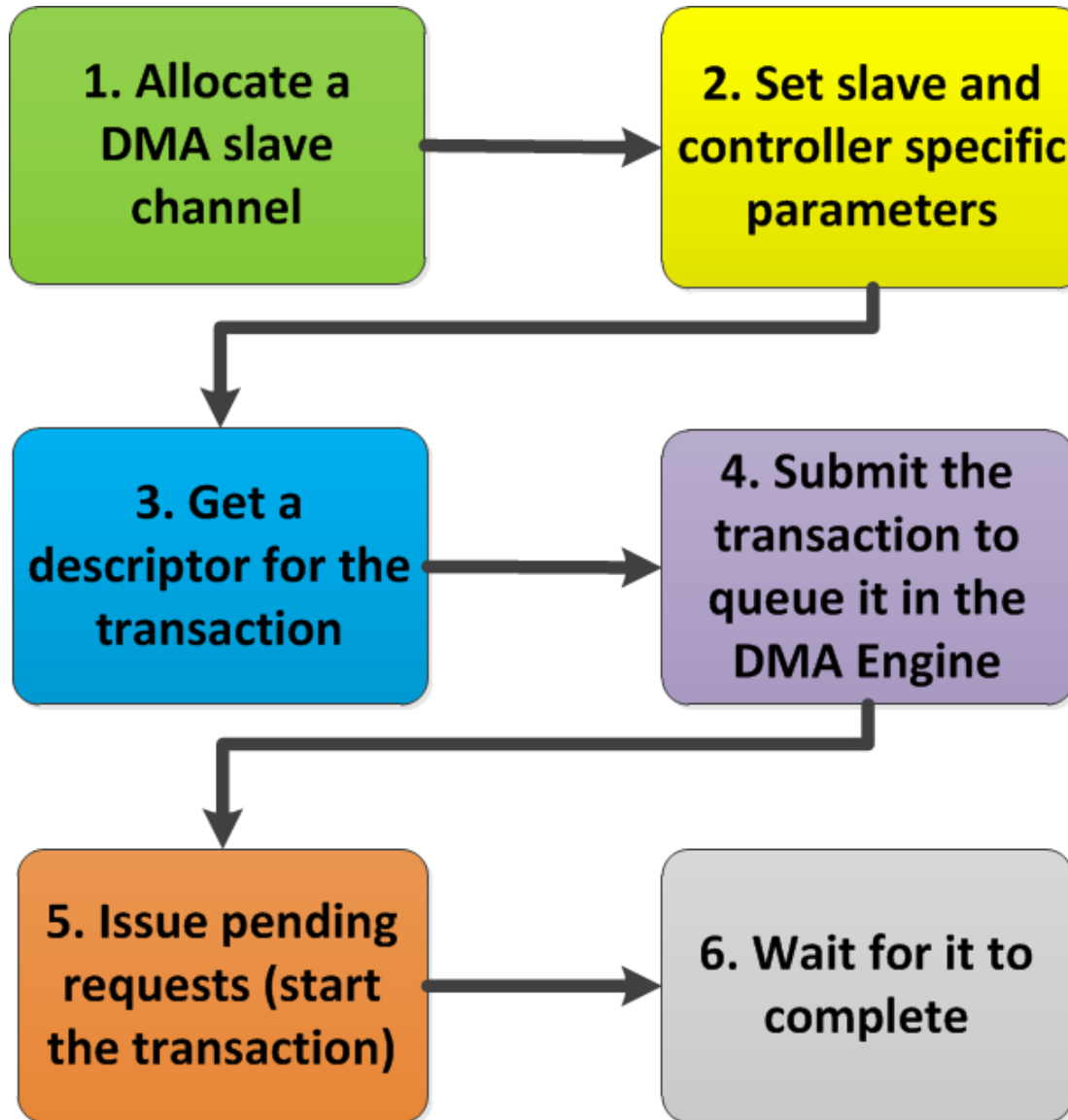
- A driver, `dmaengine.c`, along with Xilinx DMA drivers, is located in `drivers/dma` of the kernel
- Documentation about this seems to be limited
 - In kernel: `Documentation/dmaengine.txt`
 - No other good information on the web
- The Xilinx kernel has the DMA engine driver turned on by default
 - The Xilinx DMA core drivers are only visible in the kernel configuration when it is enabled
- The DMA test for the AXI DMA cores in the Xilinx kernel uses the DMA engine slave API
 - This test code is pretty complex with multiple threads such that it's not easy to get down to the basics
 - The tests are also located in `drivers/dma` (`axidmatest.c`)



Linux DMA Engine Slave API – Page 1

- **The DMA Engine driver works as a layer on top of the Xilinx DMA drivers using the **slave** DMA API**
 - It appears that **slave** may refer to the fact that the software initiates the DMA transactions to the DMA controller hardware rather than a hardware device with integrated DMA initiating a transaction
- **Drivers which use the DMA Engine driver are referred to as a client**
- **The API designed to handle complex DMA with scatter gather**

Linux DMA Engine Slave API – Page 2



➤ The slave DMA usage consists of following these steps.

Linux DMA Engine Slave API – Page 3

- **Client drivers typically need a channel from a particular DMA controller only**
 - In some cases a specific channel is desired
 - For AXI DMA, the 1st channel is the transmit channel and the 2nd channel is the receive channel
- **The function `dma_request_channel()` is used to request a channel**
 - A channel allocated is exclusive to the caller
- **The function `dma_release_channel()` is used to release a channel**
- **The `dmaengine_prep_slave_single()` function gets a descriptor for a DMA transaction**
 - This is really converting a single buffer without a descriptor to use a descriptor
 - Other functions are provided which allow other DMA modes including cyclic and interleaved modes

Linux DMA Engine Slave API – Page 4

- The `dmaengine_submit()` function submits the descriptor to the DMA engine to be put into the pending queue
 - The returned cookie can be used to check the progress
- The `dma_async_issue_pending()` function is used to start the DMA transaction that was previously put in the pending queue
 - If channel is idle then the first transaction in queue is started and subsequent transactions are 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.

Allocating a Channel Example

A private channel is not affected by processing for other channels

```
dma_cap_mask_t mask;  
dma_cap_zero(mask);  
dma_cap_set(DMA_SLAVE | DMA_PRIVATE, mask);  
  
chan = dma_request_channel(mask, NULL, NULL);  
  
// application specific processing  
// with the channel  
  
dma_release_channel(chan);
```

A more specific channel can be requested with a filter

- **Set up the capabilities for the channel that will be requested**
- **Request the DMA channel from the DMA engine**
- **Release the channel after the application is done with it**

Starting A DMA Transfer Example

```
completion cmp;
enum dma_ctrl_flags flags = DMA_CTRL_ACK | DMA_PREP_INTERRUPT;

char *buf = kmalloc(1024, GFP_KERNEL);
dma_map_single(device, dma_buffer, 1024, DMA_TO_DEVICE);

chan_desc = dmaengine_prep_slave_single(chan, buf, 1024,
                                         DMA_MEM_TO_DEV , flags);

chan_desc->callback = <call back function when the transfer completes>;
chan_desc->callback_param = cmp;

dma_cookie_t cookie = dmaengine_submit(chan_desc);
```

DMA_CTRL_ACK initializes the descriptor indicating the client owns it

DMA_PREP_INTERRUPT is used to cause an interrupt on completion

1. **Allocate a 1KB buffer of cached contiguous memory**
2. **Cause the buffer to be ready to use by the DMA including any cache operations required**
3. **Create a descriptor for the DMA transaction**
4. **Setup the callback function for the descriptor**
5. **Queue the descriptor in the DMA engine**

Linux Asynchronous Transfer API

- The `async_tx` API provides methods for describing a chain of asynchronous bulk memory transfers/transforms with support for inter-transactional dependencies
- It is implemented as a `dmaengine` client that smooths over the details of different hardware offload engine implementations
- Code that is written to the API can optimize for asynchronous operation and the API will fit the chain of operations to the available offload resources
- The `dma_async_issue_pending()` function starts the DMA transaction
 - The DMA engine calls the callback function that was supplied with the `submit` function when the transfer is complete
- The `dma_async_is_tx_complete()` function checks to see if the DMA transaction completed

Waiting For DMA Completion Example

Called by the DMA Engine when the transfer completes

```
void transfer_complete(void * completion) {  
    complete(completion);  
}  
unsigned long timeout = msecs_to_jiffies(3000);  
enum dma_status status;  
struct completion cmp;  
  
init_completion(&cmp);  
dma_async_issue_pending(chan);  
  
timeout = wait_for_completion_timeout(&cmp,  
                                     timeout);
```

It blocks waiting for the completion or a timeout

- A DMA transfer was previously submitted to the DMA Engine
- A callback function was connected to the descriptor when it was submitted (queued)
- Initialize the completion so the DMA engine can indicate when it's done
- Cause the DMA engine to start on any pending (queued) work
- Wait for the DMA transfer to complete

Processing the Transfer Status Example

```
timeout = wait_for_completion_timeout(&cmp, timeout);
status = dma_async_is_tx_complete(chan, cookie, NULL, NULL);
if (timeout == 0) {
    // timeout processing
} else if (status != DMA_COMPLETE) {
    if (status == DMA_ERROR) {
        // error processing
    }
}
```

- Wait for the transfer to complete
- Get the status of the DMA transfer using the cookie which was the result of submitting it to the DMA Engine
- The transfer could have timed out or completed, with an error or OK

Requesting A Specific DMA Channel

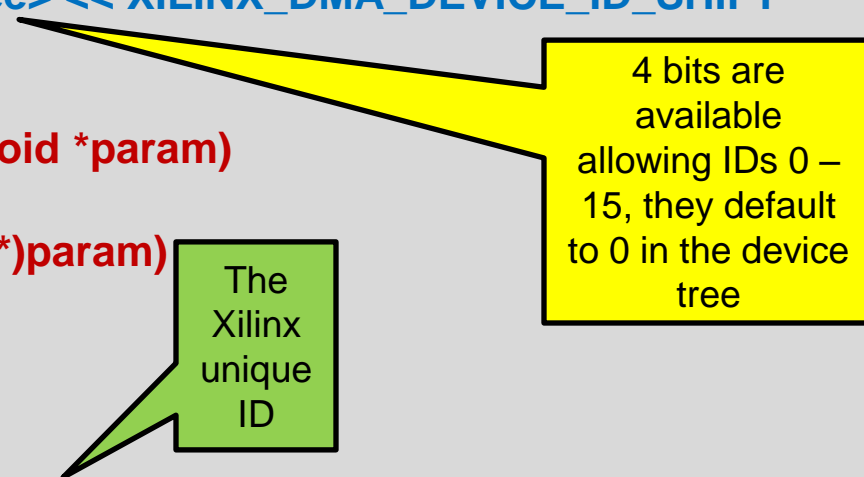
- The `dma_request_channel()` function provides parameters to allow a specific channel to be requested when there are multiple channels
 - `struct dma_chan *dma_request_channel(dma_cap_mask_t mask, dma_filter_fn filter_fn, void *filter_param)`
- `dma_filter_fn` is defined as:
 - `typedef bool (*dma_filter_fn)(struct dma_chan *chan, void *filter_param)`
 - the `filter_fn` routine will be called once for each free channel which has a capability matching those specified in the mask input
 - `filter_fn` is expected to return 'true' when the desired DMA channel is found
- The DMA channel unique ID is defined by the DMA driver using the DMA Engine
 - For Xilinx, the AXI DMA, AXI CDMA, and AXI VDMA drivers
 - They use a 32 bit word which is made up of the device id from the device tree for the channel together with the channel direction and a Xilinx ID

Requesting A Specific Channel Example

```
#include <linux/amba/xilinx_dma.h>
u32 device_id = <device-id from device tree> << XILINX_DMA_DEVICE_ID_SHIFT
u32 match;

static bool filter(struct dma_chan *chan, void *param)
{
    if (((int *)chan->private) == *(int *)param)
        return true;
    return false;
}

direction = DMA_MEM_TO_DEV;
match = (direction & 0xFF) | XILINX_DMA_IP_DMA | device_id;
chan = dma_request_channel(mask, filter, (void *)&match);
```



4 bits are available allowing IDs 0 – 15, they default to 0 in the device tree

The Xilinx unique ID

- A filter function determines if the channel matches the desired channel
- Set up the criteria for the channel being requested
- Request the channel specifying the filter function and the match criteria

OCM and DMA

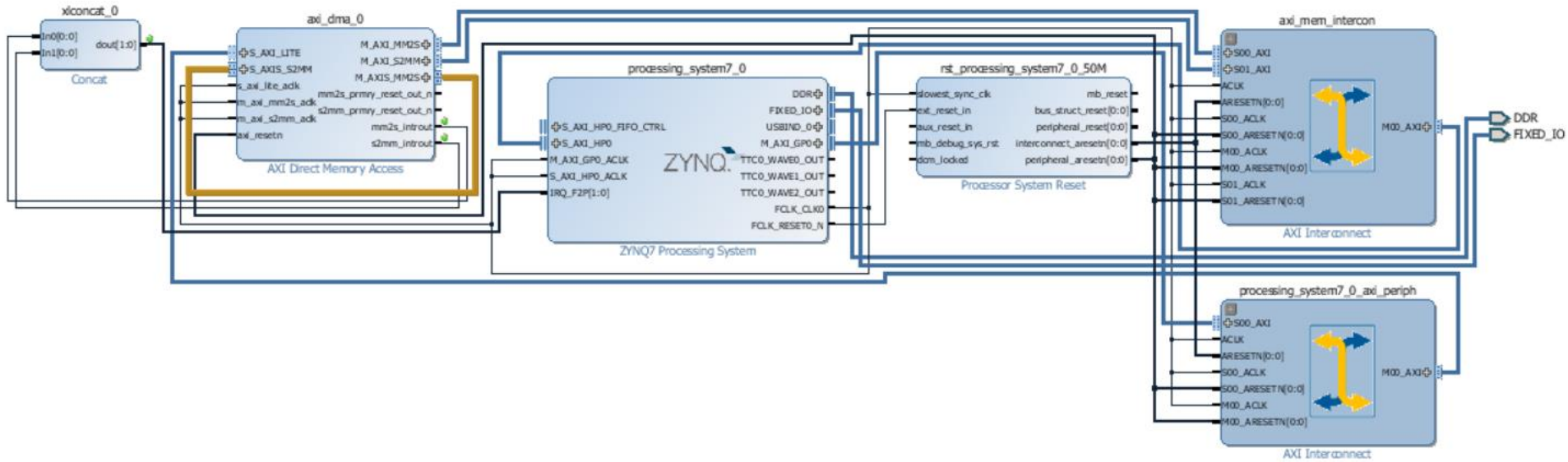
- The zynq BSP includes a general purpose allocator for OCM
 - `arch/arm/mach-zynq/zynq_ocm.c`
- It maps the memory in the MMU as device memory rather than normal memory which is typically slower
- The API is different, but simple, and there's minimal documentation
 - `include/linux/genalloc.h`
- Getting a handle to the pool is the toughest part as you need to look it up thru the device tree node
- The function `gen_pool_dma_alloc()` is used to allocate a block of memory from the pool
- The driver works for OCM mapped low or high in memory as it reads the SLCR to determine where it's located

DMA With Accelerator Coherency Port (ACP)

- When DMA is connected to the ACP port of Zynq the DMA transactions can be cache coherent such that software does not need to worry about the caches
- Cache operations in software can be a significant amount of processing for large buffers
- There are tradeoffs to be made as the DMA transactions can also disrupt the CPU caches such that there could be performance impacts to the software

Hardware System For Testing

- Using AXI DMA without scatter gather, with the transmit stream looped back to the receive stream



References

- http://infocenter.arm.com/help/topic/com.arm.doc.dai0228a/DAI228A_DMA_on_SMP_systems.pdf
- <https://www.kernel.org/doc/Documentation/crypto/async-tx-api.txt>
- <https://www.kernel.org/doc/Documentation/dmaengine.txt>
- <https://www.kernel.org/doc/Documentation/DMA-API.txt>
- <https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt>
- `include/linux/async_tx.h`
- `include/linux/dmaengine.h`
- <http://lwn.net/Articles/450286/>
- <http://lwn.net/Articles/267134/>