Linux DMA from User Space

Based on Linux kernel 3.14

John Linn, Strategic Applications Engineer 10/2014

Agenda

- Applications and examples of user space DMA
- Using the character device framework
- Implementing ioctl() functionality
- Implementing mmap() functionality
- Areas of caution
- Design for debug

- Prerequisites
  – Knowledge of the Linux kernel in general such as building and configuring the kernel
  – Character device driver experience in Linux
  – Experience with the C programming language
  – Linux DMA in Device Drivers session

Review From Linux DMA In Device Drivers

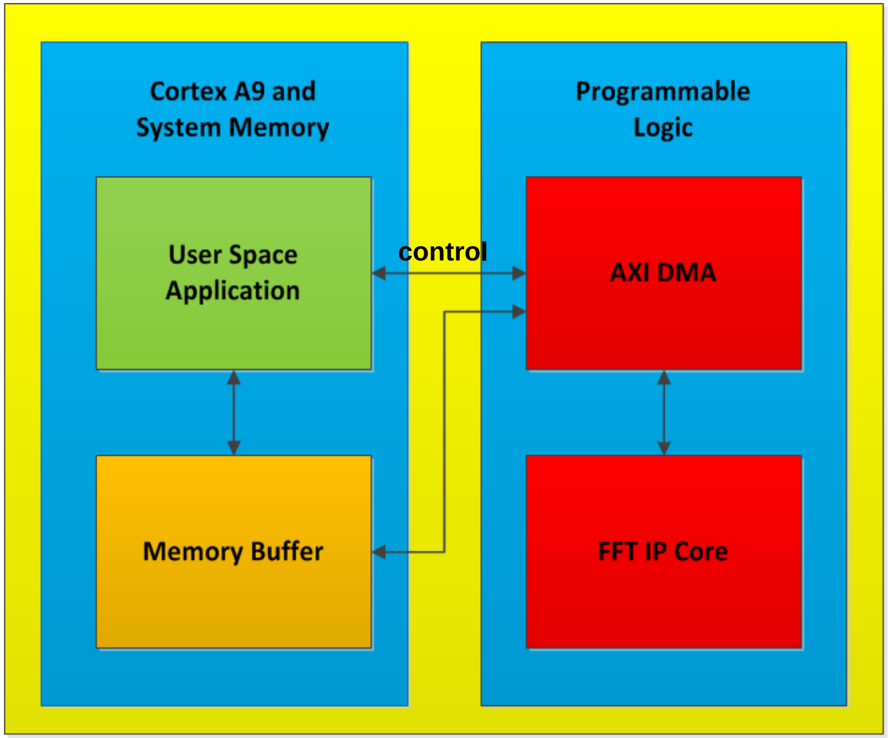
- **The primary components of DMA include the DMA device control, memory allocation and cache control**
- **DMA in Linux is designed to be used from kernel space by a higher layer device driver**
- **The DMA Engine in Linux is a framework which allows access to DMA controller drivers (such as AXI DMA) in a consistent and more abstract manner**
- **Xilinx provides device drivers which plug into the DMA Engine framework (AXI DMA, AXI CDMA, and AXI VDMA)**
- **Memory can be allocated using kmalloc() for cached memory or dma_alloc_coherent() for uncached memory**
- **DMA cache control functions such as dma_map_single() and dma_unmap_single() are used with cached memory buffers**

Introduction

- **A challenge in Linux is doing application processing in user space while moving data to and from devices in the PL**
- **Linux provides frameworks that allow user space to interface with kernel space for most types of devices (except DMA)**
- **User Space DMA is defined as the ability to access buffers for DMA transfers and control DMA transfers from a user space application**
  - This is not an industry standard and there are a number of possible methods
  - Similar methods have been used for years with display systems such as X11, as they needed direct access to video frame buffers
- **Xilinx SDIntegrator might be an easier solution for some applications and should be considered**
  - It uses similar principles without the user implementing any code

Applications of User Space DMA

- **A typical User Space DMA application creates data which needs to be transferred from the CPU memory to/from a custom IP core**

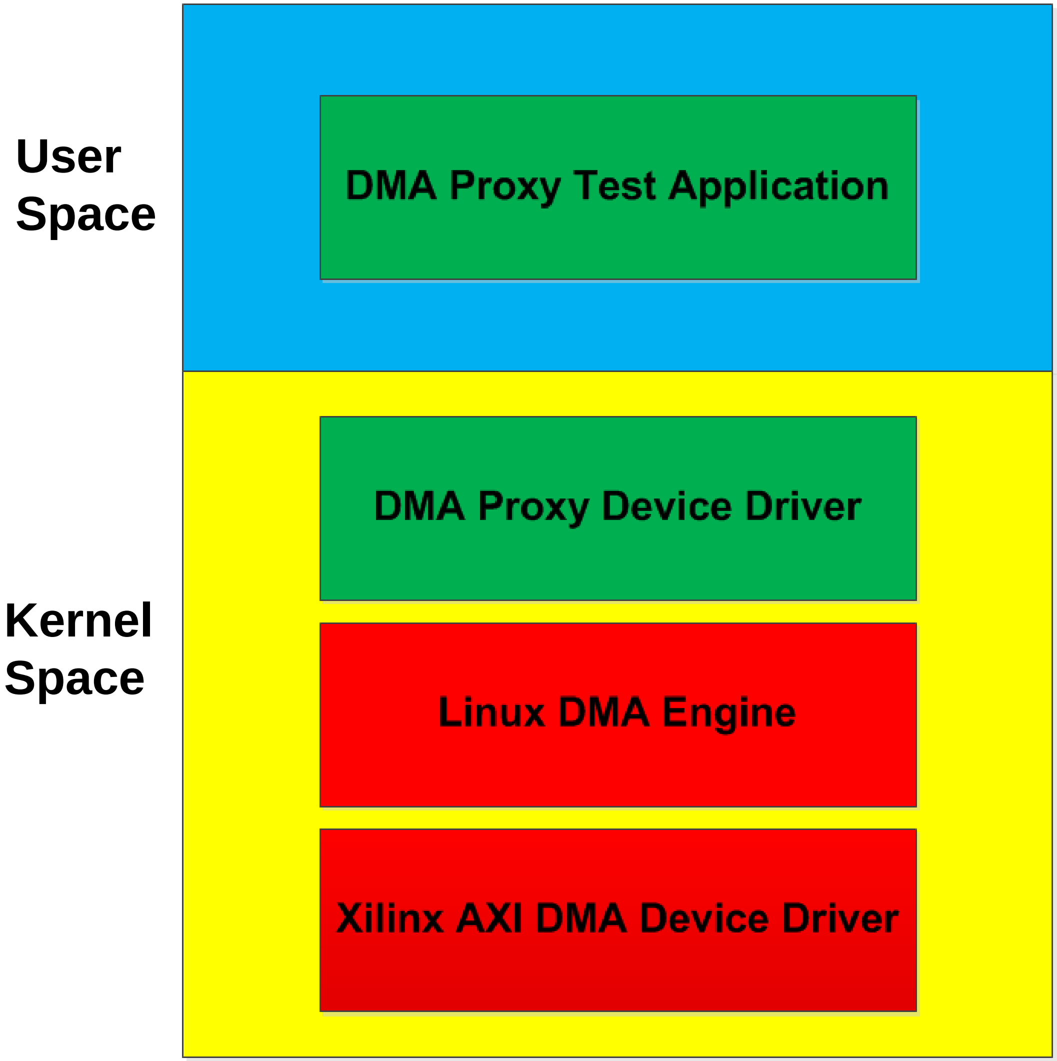


Examples

– FFT IP core processing a block of data
– Custom IP Core generating blocks of data
– See the Spectrum Analyzer Tech Tip

User Space DMA Software Example (High Level)

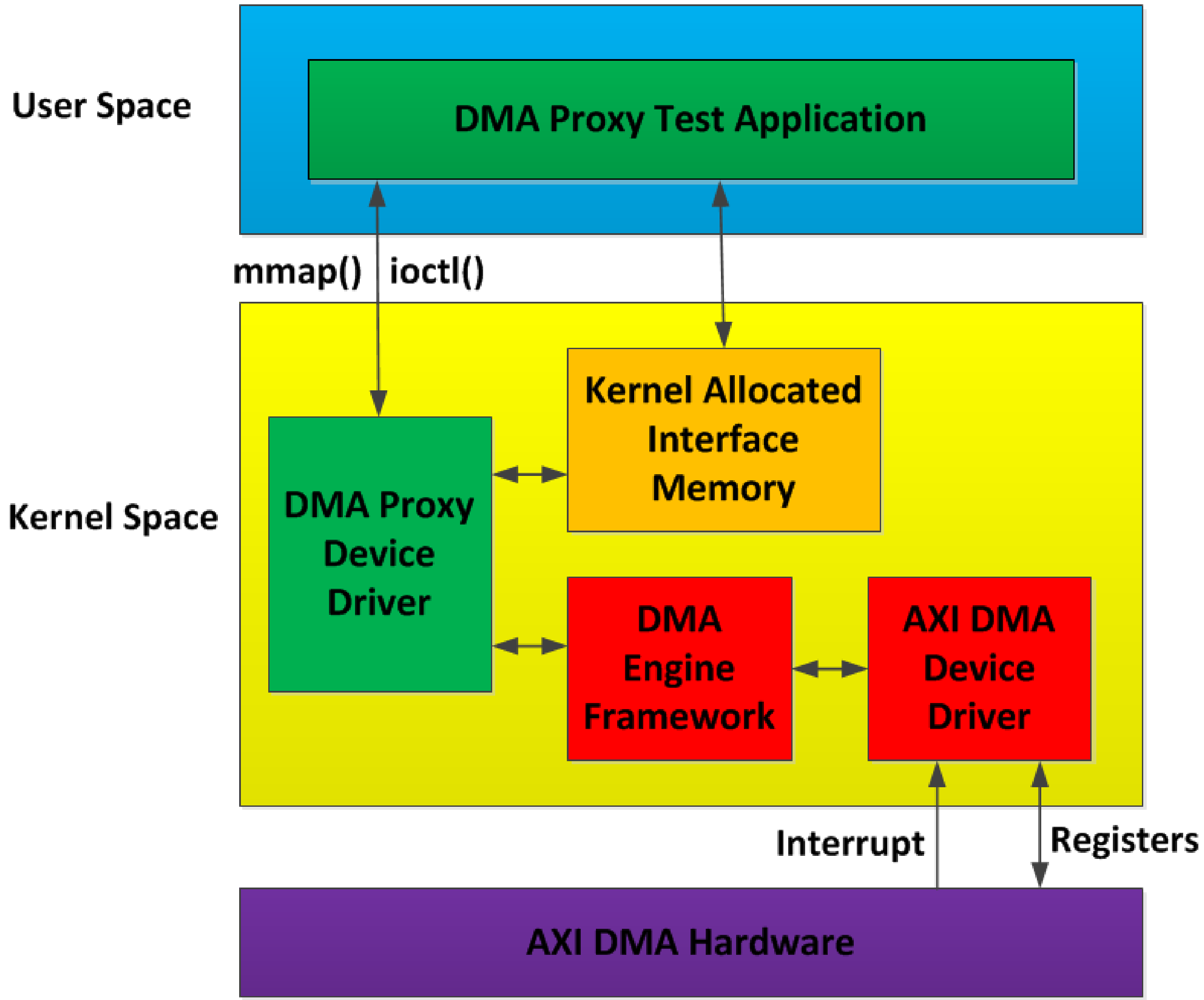
- The software design is made up of a kernel space device driver and a user space application
- The Xilinx AXI DMA Device Driver and Linux DMA Engine exist in the Linux kernel
- The DMA Proxy Device Driver is a character device driver that uses the Linux DMA Engine
- The DMA Proxy Test Application uses the DMA Proxy Device Driver to control DMA transfers



Key Learning For The Session

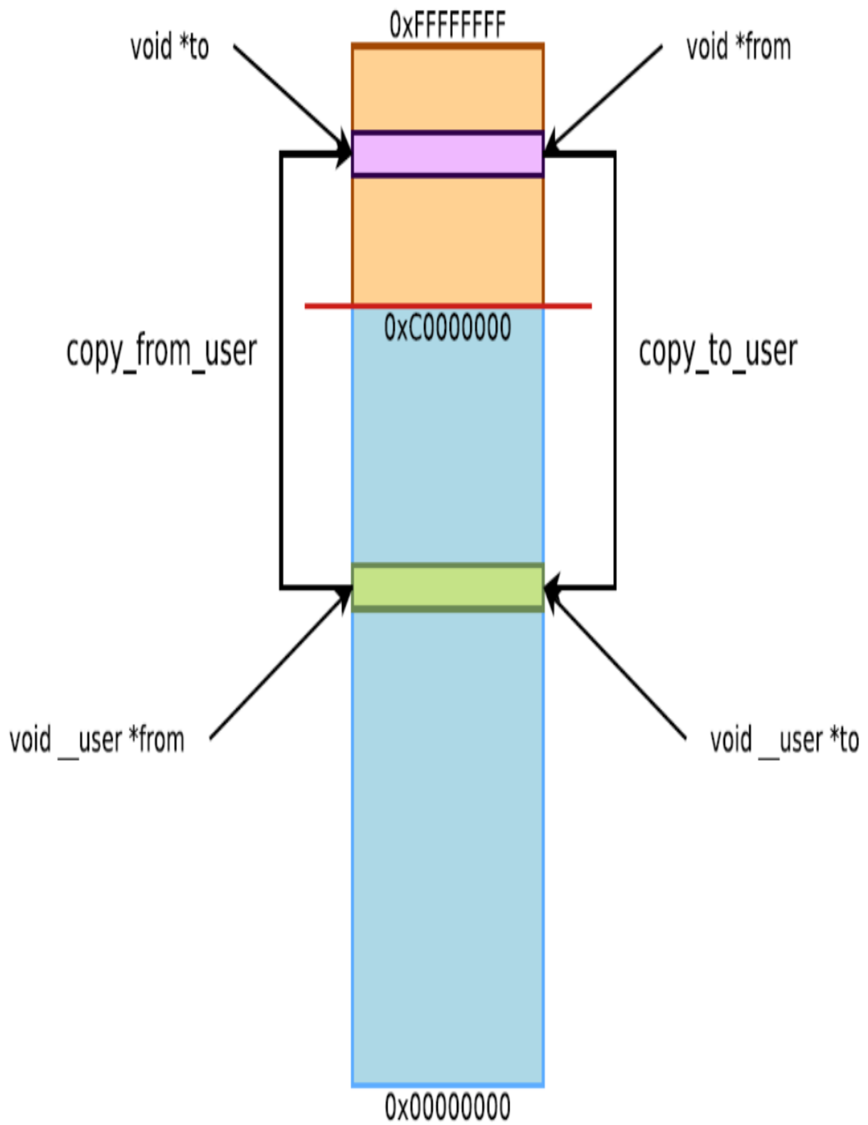
- **Creation of a character device driver that extends the functionality of the DMA kernel driver from the Linux DMA in Device Drivers session**
- **Creation of a user space application that uses the character device driver to perform DMA transfers**
- **Implementation of ioctl() in the device driver and in the user space application to cause the DMA Engine to perform DMA transfers**
- **Implementation of mmap() in the device driver and in the user space application to map kernel allocated memory into user space process address space**
- **These principles should work across any DMA device that is supported by the Linux DMA Engine**

DMA Proxy Software Detailed Design



Copying Data Between Kernel and User Space Review

- Moving data between userspace and kernel space is the primary method for I/O since the application is in userspace and the device drivers are in kernel space
- The *copy_to_user()* function copies a buffer of bytes from kernel space to userspace
- The *copy_from_user()* function copies a buffer of bytes from userspace to kernel space
- Functions also exist for copying a single datum or null terminated string



XILINX ALL PROGRAMMABLE.

Zero Copy Buffer Design

- **Many software designs copy data from user space to kernel space and from kernel space to user space**
- **For larger buffers copying data is inefficient and in the case of DMA it defeats the purpose of using DMA to move the data**
- **A zero copy design avoids copying memory and is required for user space DMA applications**
- **Some network stacks (not Linux) provide a zero copy design and achieve higher performance**
- **Mapping a kernel space allocated memory buffer into user space removes the need to copy data**
- **Mapping user space allocated buffers into kernel space so that a driver can access them is another method**
  - This is more complex and not covered in this session

Character Device Framework Review

- The character device framework of Linux provides functionality such as open(), read(), write() and close() which allows a device driver to be accessed using the file I/O operations from user space
- It also provides the ioctl() interface which is used to control the device in non standard ways
- The function prototype in a driver:
  - int (*ioctl) (struct file *filp, unsigned int cmd, unsigned long arg);
- The cmd and arg arguments are passed from user space to the driver unchanged such that they are easily used for control
- The ioctl() function of the device driver can perform any functionality including blocking until the functionality is complete

Controlling The Kernel Space Driver

- **The user space application needs to control the kernel space driver to allow DMA transactions to be managed**
- **The read() and write() file operations could easily be used**
  - These do offer the ability to do asynchronous (non-blocking) I/O using poll() and select() functions
- **The ioctl() file operation is designed for device control and is used to control the DMA Proxy device driver for simplicity**
- **The mmap() file operation allows memory of the device driver to be mapped into the address space of the caller in a user space process**
- **The UIO driver framework provides another alternative for this design which is simpler but limited and less flexible**
  - mmap() can be overridden with your own implementation for non-cached memory
  - It's not as flexible as the character device framework

The Character Device Driver Simplified Example

```
int dma_proxy_open() { };
int dma_proxy_ioctl() { };
int dma_proxy_mmap() { };
int dma_proxy_release() { };

static struct file_operations dma_proxy_fops =
{
        .owner          = THIS_MODULE,
        .open           = dma_proxy_open,
        .unlocked_ioctl = dma_proxy_ioctl,
        .mmap           = dma_proxy_mmap,
        .release        = dma_proxy_release,
};
int dma_proxy_init()
{
        struct cdev cdev;
        cdev_init(&cdev, &dma_proxy_fops);
        cdev_add(&cdev, ….);
}
```

- Create empty file operation functions dma_proxy_open(), dma_proxy_ioctl(), dma_proxy_mmap(), & dma_proxy_release()
- Create the file_operations data structure dma_proxy_fops
- The driver dma_proxy_init() function calls the character device functions to create the character device
- The cdev_init() function initializes the character device including setting up the file functions such as dma_proxy_ioctl()
- The cdev_add() function connects the character device to the kernel

Cached Buffers Considerations

- **Cache control from user space is challenging and less obvious**
  - Cache control is done in the DMA Proxy device driver from kernel space
- **Many people would assume that using caches makes everything faster**
  - It depends on how the application uses the data and the data size
  - Caching large buffers can pollute the CPU cache, causing other system impacts
- **The cache operations required for a DMA driver do take time for the CPU**
- **An application which only controls a DMA transfer without touching any of the data can use uncached memory**
- **The amount of memory that can be allocated varies for cached and uncached memory**
  - 4 MB cached memory using kmalloc() or get_free_pages()
  - Configurable (much larger) with uncached memory using dma_alloc_coherent() and the contiguous memory allocator in Linux

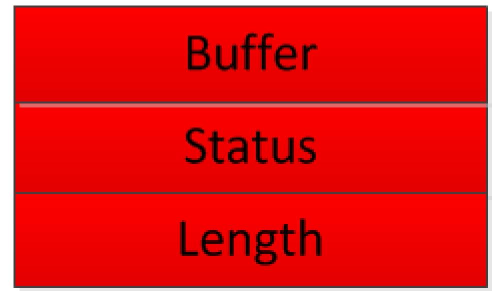
Details of Controlling DMA From User Space

- **Shared memory between user space and kernel space can be used for more than data buffers**
- **Control and status in addition to data is needed from user space**
- **Control of the DMA includes the ability to:**
  - start/stop a transaction
  - a source address for the data buffer
  - a length specifying how many bytes of data are in the data buffer
- **Status of the DMA includes the ability to see that the transfer completed and any errors that might have occurred**
- **The DMA Proxy example uses kernel allocated memory referred to as interface memory**

Interface Memory Details

- The interface memory is allocated by the DMA proxy driver and mapped to user space using mmap()
- The dma_proxy_channel_interface contains the data, control and status for a channel
- The user space application controls the DMA proxy driver using the data in the interface memory
- The DMA proxy device driver controls the DMA Engine using the data in the interface memory

DMA Proxy Channel Interface



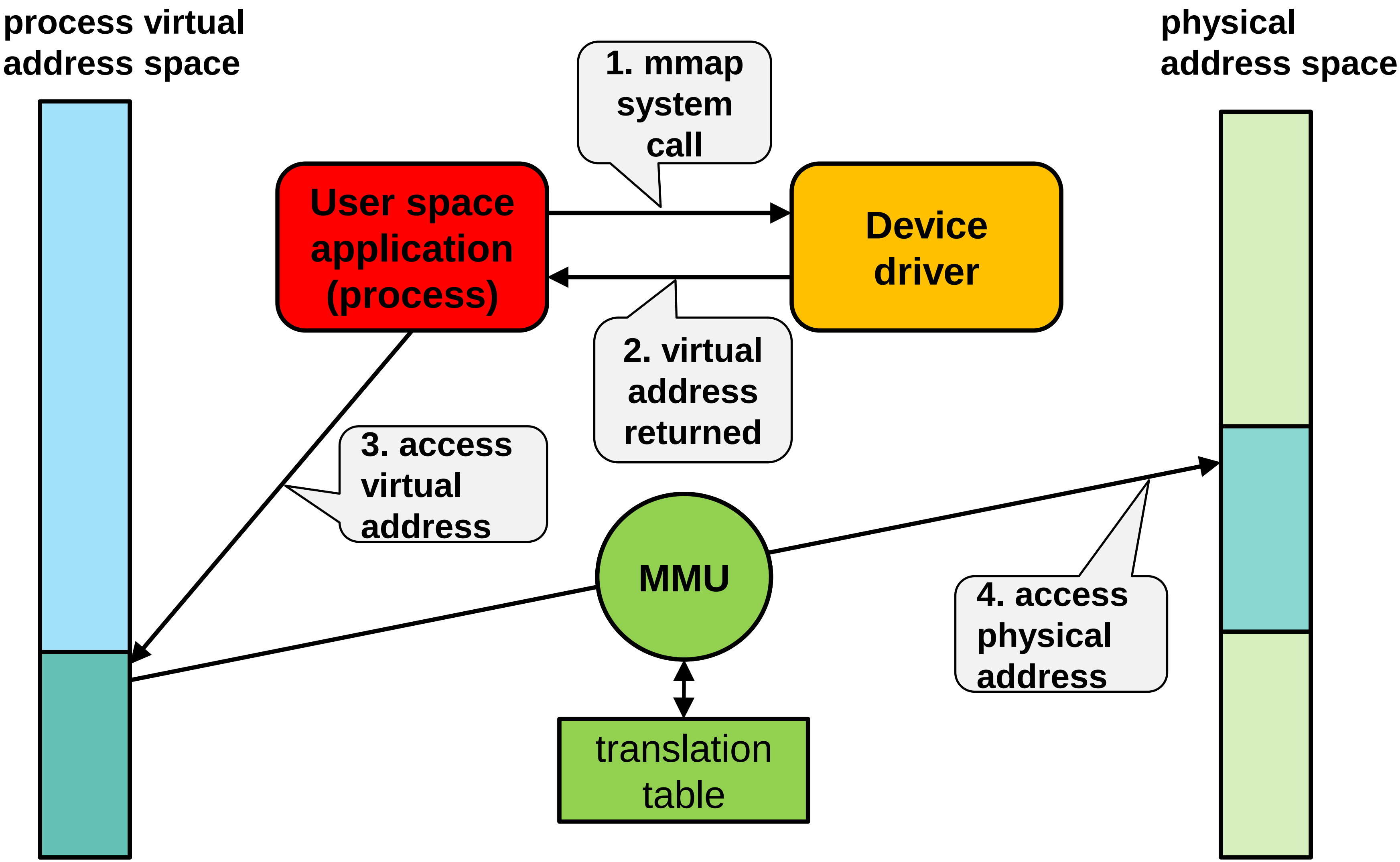

```
struct dma_proxy_channel_interface {
   unsigned char buffer[32 * 1024 * 1024];
   enum proxy_status {
      PROXY_NO_ERROR = 0, PROXY_BUSY = 1,
      PROXY_TIMEOUT = 2, PROXY_ERROR = 3
   } status;
   unsigned int length;
};
```

Note the buffer is the first member of the struct to ensure it is cache line aligned.

Introduction to Mapping Memory with mmap()

- The character device driver framework of Linux provides the ability to map memory into a user space process virtual address space
- A character driver must implement the mmap() function which a user space application can call
- The mmap() function has several ways it is used and feels a bit confusing with overloaded arguments
- In this application it is used to map a physical memory address range into the virtual memory address space
- A virtual address, corresponding to the physical address, is returned from mmap()
- Whenever the user space program reads or writes in the virtual address range it is accessing the physical address range
- This provides improved performance as no system calls are required

Mapping Device Memory Flow



Details of Mapping Memory with mmap()

- **Calling mmap() from the user space application**
  - The call to mmap() requires an address and size for the memory being mapped into user space
  - The application passes zero for the address to map as it does not know the address of the buffer allocated in the kernel driver
  - The size cannot be zero as mmap() will return an error
  - The application knows the size using a shared data definition in a header file
- **Implementing mmap() in the kernel space device driver**
  - The mmap() function in the driver must alter the caching attributes to match the kernel buffer being mapped <u>if the buffer is not cached</u>
    - The kernel has a mapping of the memory in the MMU and another is going to be created for the user space application process and they must match
    - Memory allocated with kmalloc() is cached
  - The DMA framework provides a mmap() function which can be called from the driver mmap() function to perform the memory mapping for buffers allocated from the DMA framework
    - Memory allocated with dma_alloc_coherent() is uncached

Simple User Space Application Example

```
struct dma_proxy_channel_interface { }

void main() {
   struct dma_proxy_channel_interface *proxy_interface_p;
   int proxy_fd;

   proxy_fd  =  open("/dev/dma_proxy", O_RDWR);
   proxy_interface_p = mmap(0, sizeof(dma_proxy_channel_interface),
                            PROT_READ | PROT_WRITE, MAP_SHARED, proxy_fd, 0);
}
```

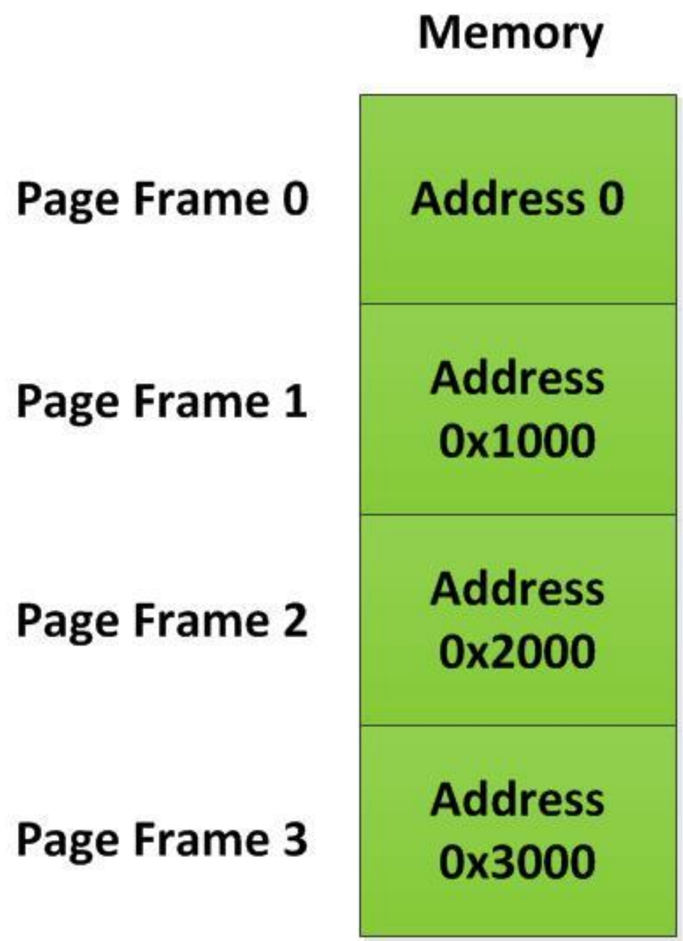
The device file causes the mmap() function to run in the driver

- **Start with an empty main() function and a defined channel interface data type**
- **Open the device file for the DMA proxy**
- **Call the mmap() function to map the kernel allocated buffer into the process address space**
- **The first argument with a value of 0 lets the kernel choose the virtual address which the physical address will be mapped to**
- **The second argument is the size of the memory range to map**

Linux Pages and Page Frame Numbers

- Virtual and physical memory are divided into handy sized units called *pages*
- These pages are all the same size, 4KB for ARM and MicroBlaze
- A page frame number is simply an index within physical memory that is counted in page-sized units
- The page frame number for a physical address can be created using the constant PAGE_SHIFT

  page_frame_number = physical_address >> PAGE_SHIFT



Simple Memory Mapping Driver Example

```
static int dma_proxy_mmap(struct file *filp, struct vm_area_struct *vma)
{
        if (remap_pfn_range(vma, vma->vm_start,
                            virt_to_physical(buffer_pointer) >> PAGE_SHIFT,
                            vma->vm_end - vma->vm_start,
                            vma->vm_page_prot))
                return -EAGAIN;
        return 0;
}
```

Convert the physical address to the page frame number

Note: This is for memory allocated with kmalloc()

- **Start with an empty mmap() function with the expected Linux interface**
- **The remap_pfn_range() function is an easy way to implement the mmap() function for memory including allocated buffers or a device**
- **Only one argument has to be created as all others come in the vma structure**
- **The 3rd argument is the page frame number which is based on the physical address**
- **Note: mmap() defaults to cached memory such that the cache attributes of the vma match the buffer allocated from kmalloc()**
- **The cache attributes are in vma->vm_page_prot and could be altered**

DMA Memory Mapping Driver Example

```
static int dma_proxy_mmap(struct file *filp, struct vm_area_struct *vma)
{
    return dma_common_mmap(dma_device_pointer,
                           vma,
                           buffer_pointer,
                           physical_buffer_pointer,
                           vma->vm_end - vma->vm_start);
}
```

Pointers are virtual addresses by default

Note: This is for memory allocated with dma_alloc_coherent()

- **Start with an empty mmap() function with the expected Linux interface**
- **The dma_common_mmap() function is the easy way to implement the mmap() function**
- **The buffer_pointer and physical_buffer_pointer are both returned from dma_alloc_coherent()**

XILINX ALL PROGRAMMABLE.

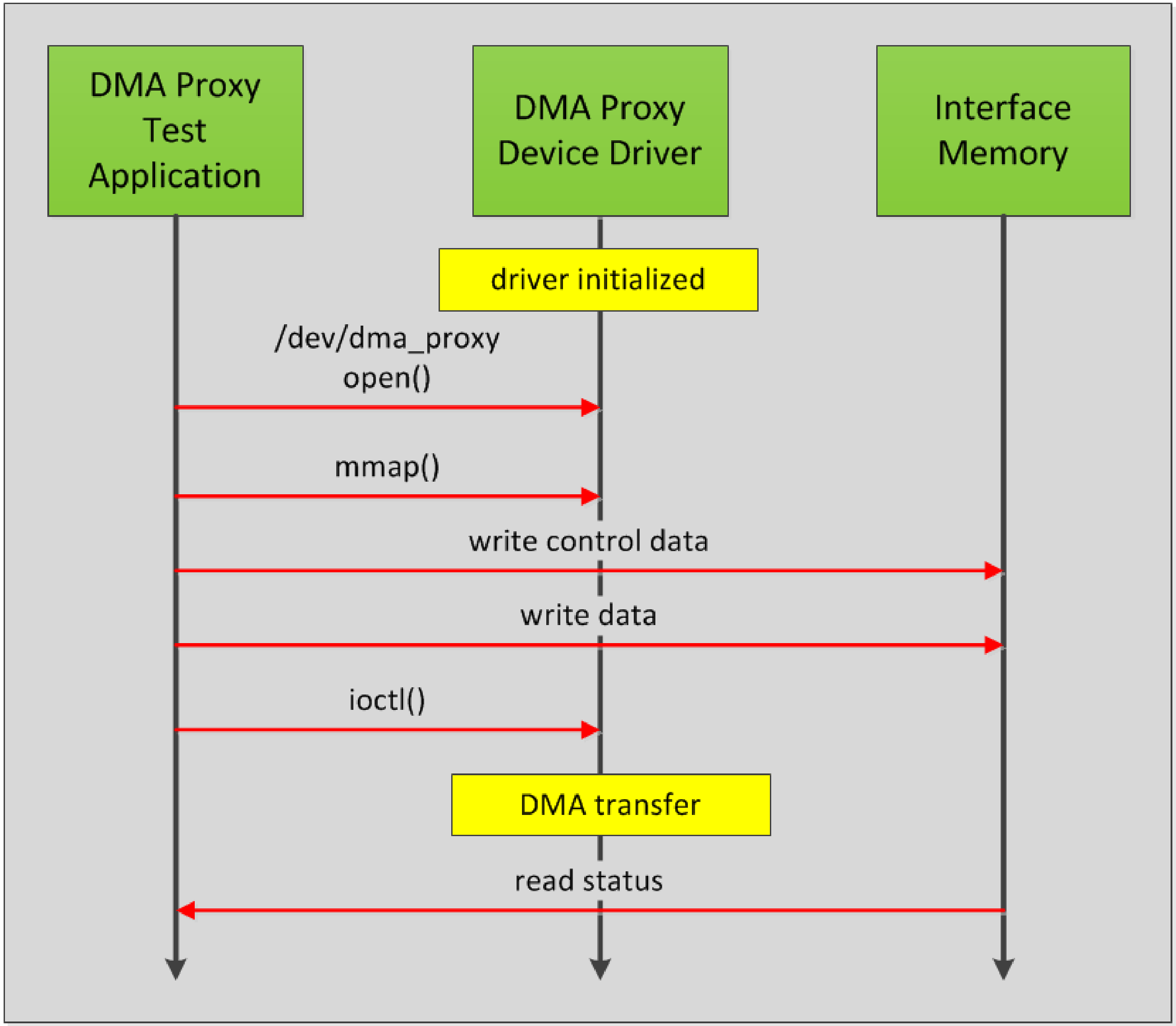
A Simple ioctl() Example Controlling DMA

```
static void transfer(struct dma_proxy_channel *pchannel_p) { };
static int open(struct inode *ino, struct file *file)
{
        file->private_data = container_of(ino->i_cdev, struct dma_proxy_channel, cdev);
        return 0;
}
static long ioctl(struct file *file, unsigned int unused1, unsigned long unused2)
{
        struct dma_proxy_channel *pchannel_p = (struct dma_proxy_channel *)file->private_data;
        transfer(pchannel_p);
        return 0;
}
```

- The transfer() function manages the DMA engine to cause the DMA transfer to occur
- The transfer() function uses the interface memory to determine the details of the DMA transaction including the length of the transfer
- The open() function is called when the application opens the device file
- The ioctl() function receives a notification requesting a DMA transfer to be performed for the device channel

Software Design Sequencing

- **The diagram illustrates the interaction between the user space application, the device driver, and the interface memory with time flowing from top to bottom**



XILINX ALL PROGRAMMABLE.

Design Alternatives

- **A design which only blocks is much simpler than one that does not block**
  - Non-blocking requires asynchronous processing to complete the transaction; this is more complex
- **The DMA Buffer Sharing framework in Linux could be helpful**
  - This session is focused on the simplest example while this adds more complexity
- **It is also possible for a kernel module to get access to user space allocated memory through the get_user_pages() function**

Performance Reviewed

- **Testing was done with both standalone (bare metal) and with Linux to compare the performance**
- **The performance of an unloaded Linux system was very similar to standalone**
- **The performance was only reviewed with respect to the time for the receive channel ioctl() call from the application to the driver**
- **Cached buffers can appear to be lower performance due to cache processing by the CPU**
  - **The additional performance of faster application processing of the cached buffers must be factored in**
- **Larger buffers should definitely not be cached in Linux as the system performance is greatly impacted**
  - The exact size where to stop caching was not determined
- **There appeared to be very little performance impact due to the transmit channel running while the receive channel was being measured**

Areas Of Caution for DMA

- **Memory mappings (cached, noncached, etc.) should always match for a buffer across kernel and user space**
- **Buffer alignment with respect to cache lines is needed for DMA**
- **The driver could exit and free the memory while the application is still trying to use it**
  - This is not typically an issue when the driver is built into the kernel
- **These methods have only been tested in a prototype system**
  - Not used by any customers yet

Designing For Debug

- **Using interface memory to pass control to the driver rather than passing the data as arguments in ioctl() is more flexible**
- **The kernel space device driver can also alter the memory to control itself**
  - This is a good way to test the driver before the user space application is written
  - It also can help discern a working device driver from an issue with mapping memory into the user space application

Dumping Kernel Page Tables

- **This feature is new to the 3.14 kernel**
- **The kernel page tables will show DMA allocated memory and verify it is not cached and is bufferable/write combined memory**
- **It can also help verify buffers are released**
- **Configure the kernel with CONFIG_ARM_PTDUMP**
  - From the Kernel Hacking menu, select Export kernel pagetable
- **cat /sys/kernel/debug/kernel_page_tables**

```
---[ Kernel Mapping ]---
0xc0000000-0xc0a00000          10M      RW x  SHD
0xc0a00000-0xdf800000         494M      RW NX SHD
0xdf800000-0xdf844000         272K      RW NX SHD MEM/BUFFERABLE/WC
0xdf844000-0xdf900000         752K      RW NX SHD MEM/CACHED/WBWA
0xdf900000-0xdfc01000        3076K      RW NX SHD MEM/BUFFERABLE/WC
0xdfc01000-0xdfd00000        1020K      RW NX SHD MEM/CACHED/WBWA
0xdfd00000-0xe0001000        3076K      RW NX SHD MEM/BUFFERABLE/WC
0xe0001000-0xef800000      253948K      RW NX SHD MEM/CACHED/WBWA
```

A 3 MB DMA buffer

XILINX ALL PROGRAMMABLE.

Systems With AXI DMA

- The AXI DMA IP core can be used for DMA to and from a custom IP core
- A system using AXI DMA without scatter gather, with the transmit stream looped back to the receive stream, can be used for testing
- The length of transfers is configured at build time with a max of 23 bits which limits the transfer length to be 8MB – 1 bytes (0 is a valid length)