Introduction to Linux Device Drivers

John Linn
Based on 3.14 Linux kernel

Goals Of This Training

- **Make you aware of the architecture and frameworks of Linux**
- **Teach you how to read a simple device driver at a high level and understand its functionality**
- **Point you to good reference material where you can learn all the details**
  - The references are in the last slide
  - Linux Device Drivers is a book that is heavily used by all Linux kernel developers
- **The following are <u>not</u> goals of this training:**
  - Will not make you a device driver developer
  - Will not make you ready to submit a driver upstream to the kernel community
    - The APIs vary with kernel versions and it is hard to stay up to date on the coding guidelines for upstreaming unless you are actively doing it

Outline

- Concepts Review
- Kernel Modules
- Kernel Frameworks
- Device Tree
- Platform Device Driver
- Character Device Driver
- Debugging

Introduction

- **A lot of good documentation exists in the public domain if you know where to find it**
- **A lot of the information in this presentation is based on others' work including Free Electrons**
- **Free Electrons provides excellent training materials for free and licensed as Creative Commons CC-BY-SA**
  - http://free-electrons.com/docs

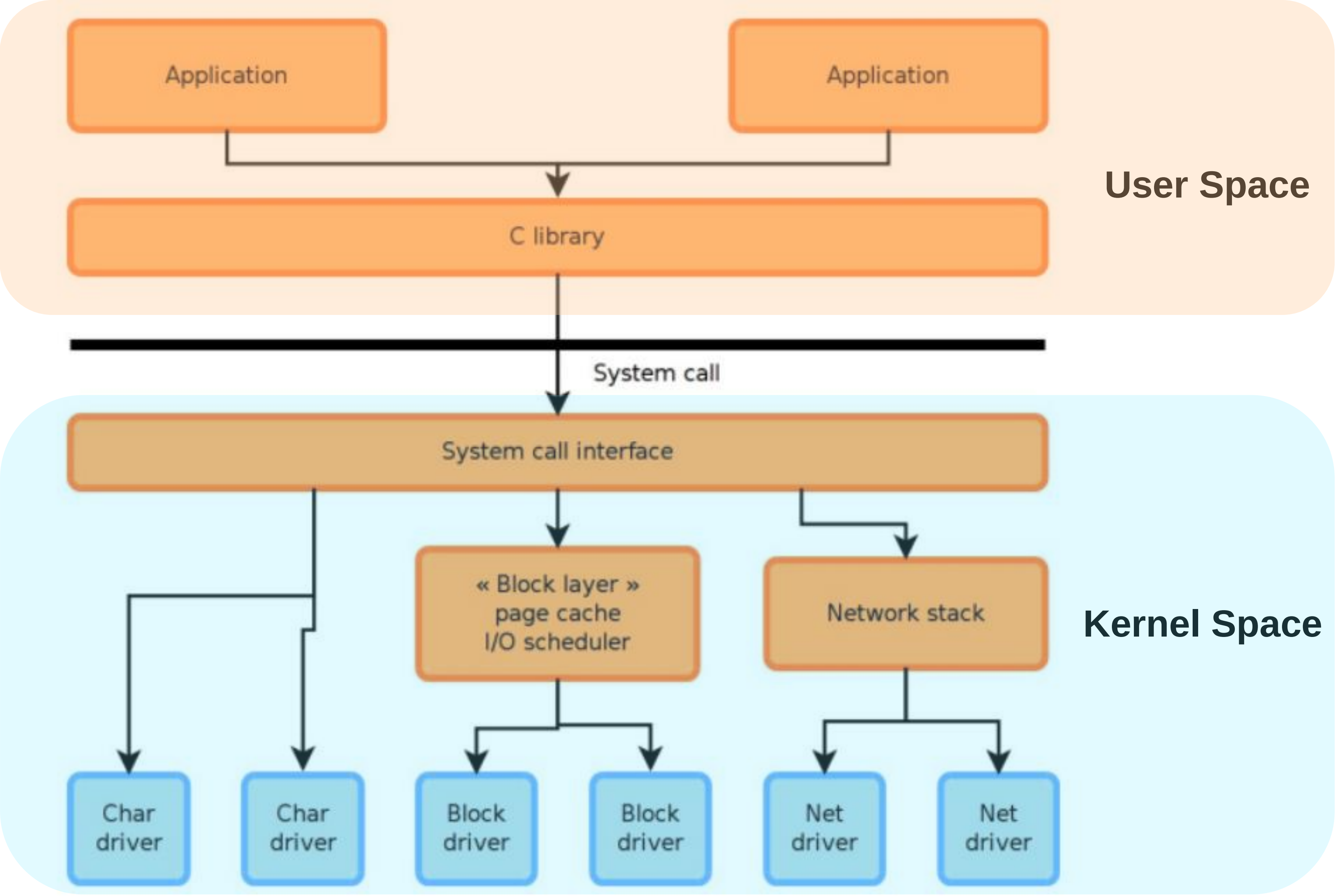
Linux Architecture 101

- **Virtual memory management is a key aspect of Linux**
  - The Memory Management Unit (MMU) of the processor translates virtual addresses to physical addresses
- **Linux divides virtual memory into kernel space and user space**
  - Kernel space is the memory area for the kernel and device drivers
    - Kernel space is the top 1 GB of memory, 0xC0000000 to 0xFFFFFFFF
  - User space is the memory area for user application software
    - User space is the bottom 3 GB of memory, 0 to 0xBFFFFFFF
  - Other kernel/user space memory configurations are configurable in the kernel such as 2 GB kernel and 2 GB user space
  - Kernel space virtual address 0xC0000000 maps to physical address zero
- **Linux uses the processor modes to create privilege levels**
  - The kernel executes at a higher privilege level than user space code such that it can access any resources in the system
  - Applications execute at a lower privilege level such that they must use the kernel to get to the restricted resources in the system

Linux Architecture 101 – Page 2

- **Library functions run in user space and provide a more convenient interface for the programmer**
  - Linux applications require a C library to build which is provided by the tools
  - The Xilinx Linux GNU tools are based on the GNU C Library (glibc)
  - The Xilinx standalone GNU tools are based on newlib library rather than glibc
- **System calls run in kernel mode on the user's behalf and are provided by the kernel itself**
- **A library function calls one or more system calls, and these system calls execute in supervisor mode since they are part of the kernel itself**
- **Once the system call completes its task, it returns and execution is transferred back to user mode**
- **The user space application is typically blocked until the library function and system call return (just like a function call)**
- **System calls may interact with the kernel proper, or with specific drivers and frameworks of the kernel**

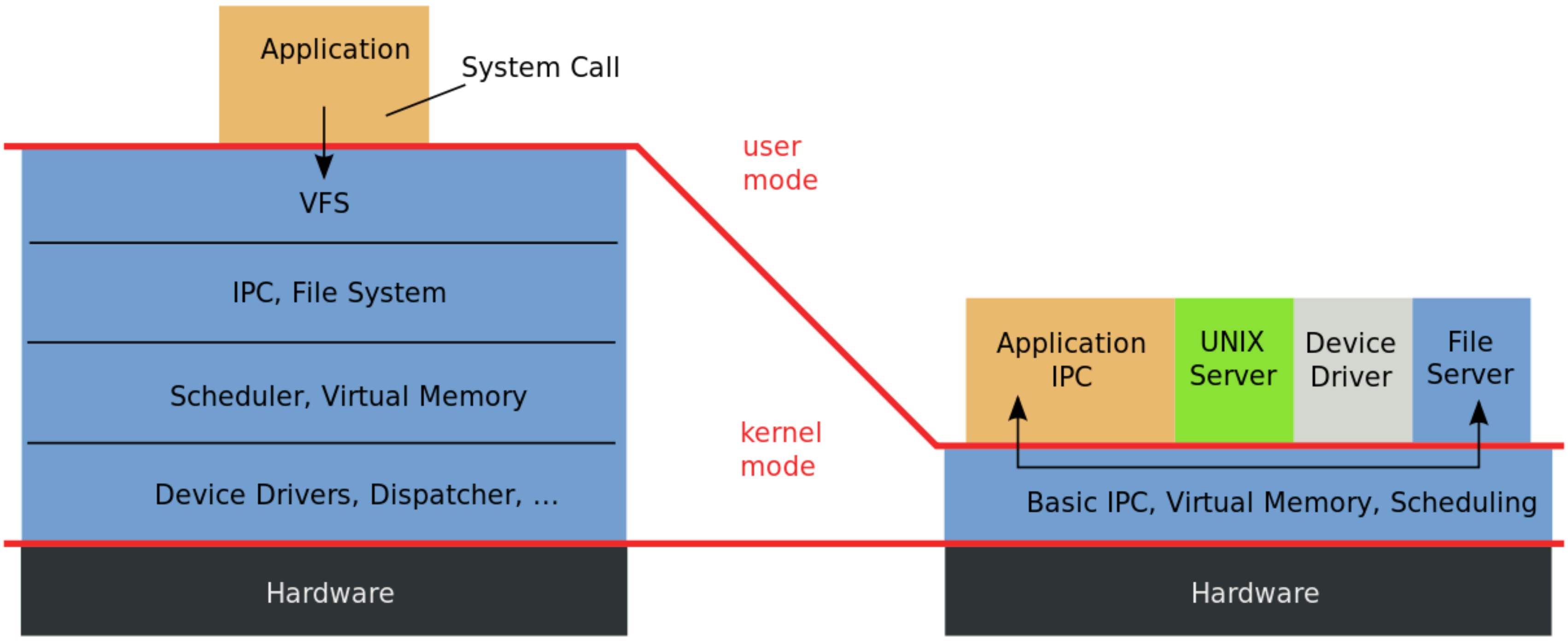
Linux Architecture 101 – Page 3



Linux Architecture 101 – Page 4

Linux, A Monolithic Kernel based Operating System

Microkernel based Operating System (such as FreeRTOS)



* Illustration taken from http://en.wikipedia.org/wiki/Microkernel

Linux Device Model (Chapter 14 LDD)

- **You don't have to be a kernel expert, but understanding some terms will help a lot**
- **The Linux Device model is built around the concept of busses, devices and drivers.**
- **All devices in the system are connected to a bus of some kind.**
- **A bus may be a software abstraction rather than a real bus.**
- **Busses primarily exist to gather similar devices together and coordinate initialization, shutdown and power management**
- **When a device in the system is found to match a driver, they are bound together. The specifics about how to match devices and drivers are bus-specific.**

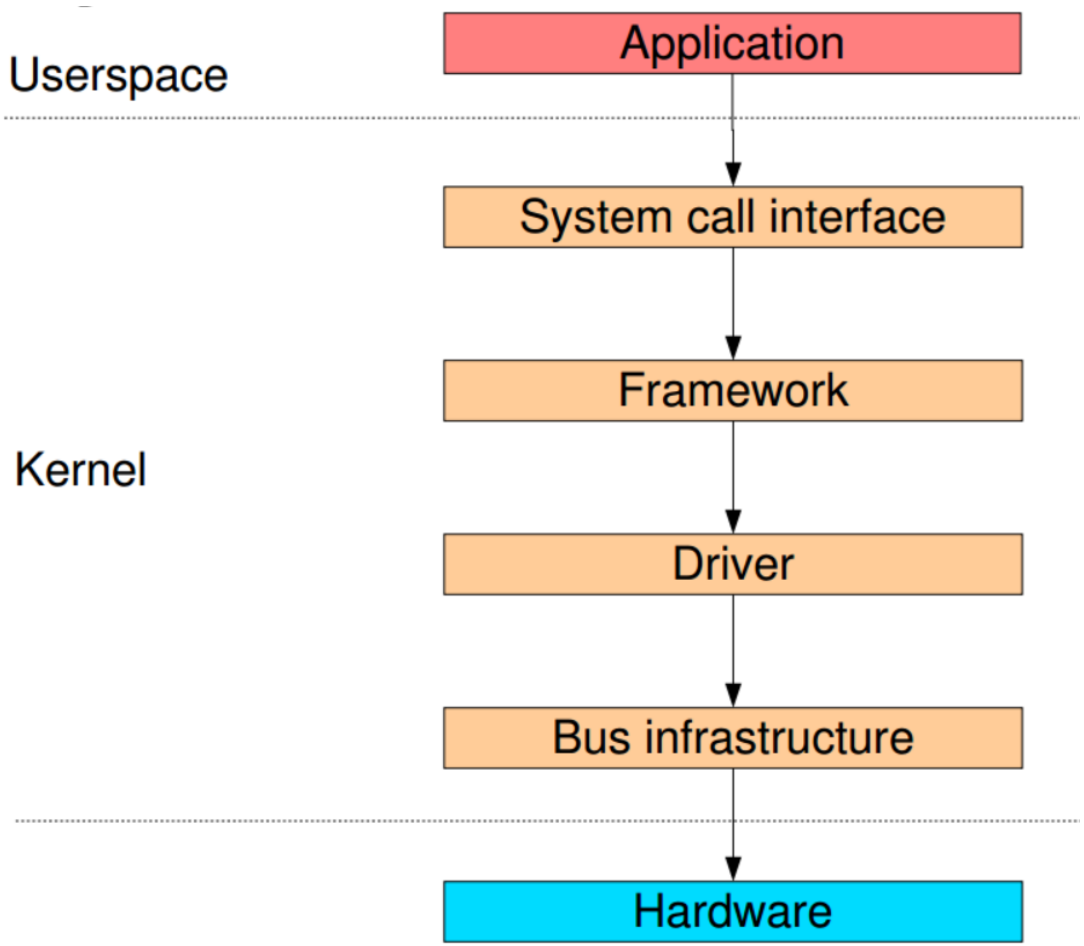
Linux Device Types

- ***Network devices***
  - These are represented as network interfaces, visible in userspace using the ifconfig utility
- ***Block devices***
  - These are used to provide userspace applications access to raw storage devices (hard disks, USB keys)
  - Visible to the applications as device files in /dev
- ***Character devices***
  - These are used to provide userspace applications access to all other types of devices (input, sound, graphics, serial, etc.)
  - They are also visible to the applications as device files in /dev
  - Many devices are character devices and a lot of user IP could be accessed as a character device
- **MTD devices**
  - Flash memory is a unique device type that has translations to allow them to be used as block and character devices

Linux Kernel Frameworks

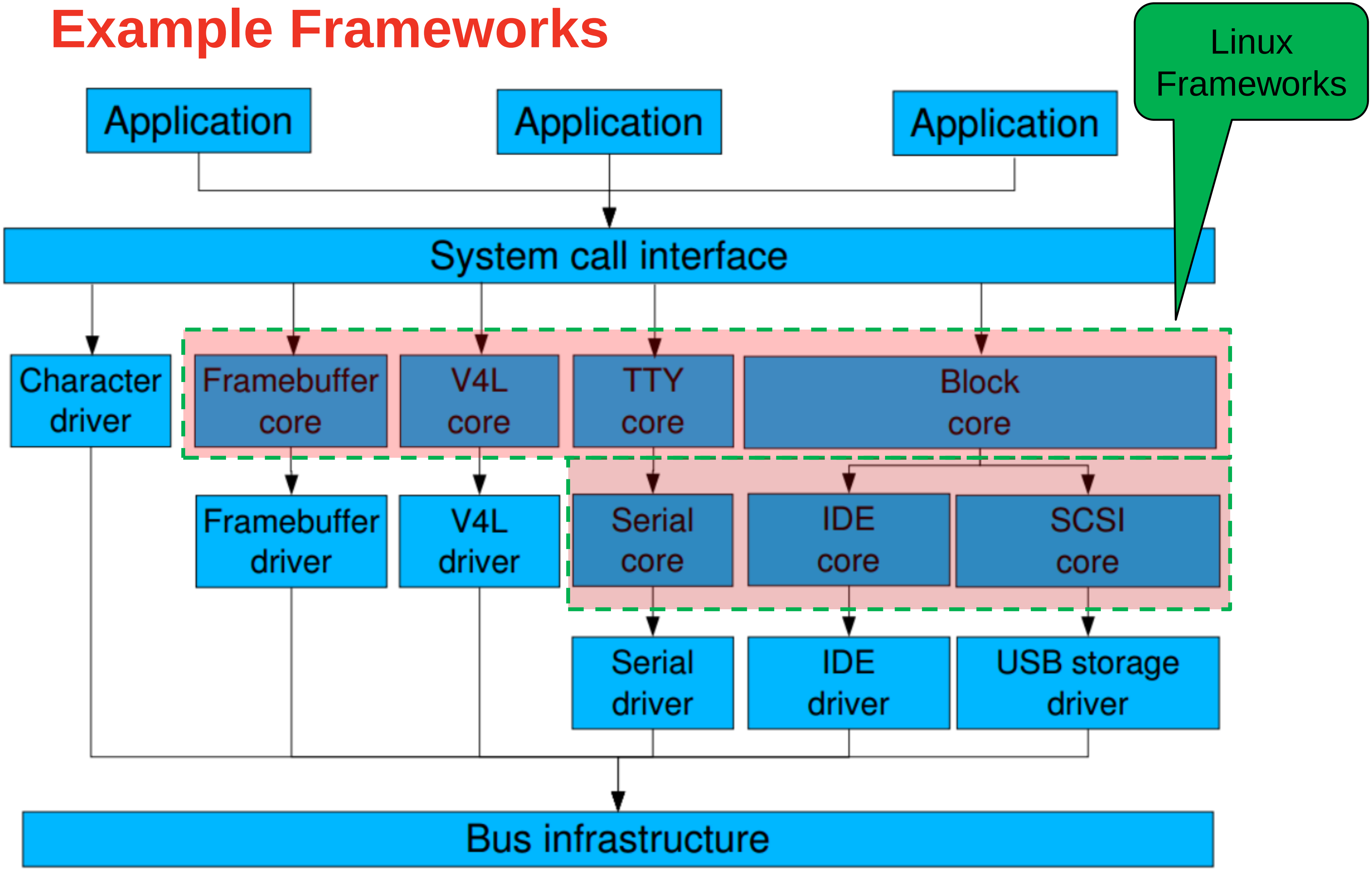
- **Many device drivers are not directly implemented as character devices or block devices. They are implemented under a framework, specific to a device type (framebuffer, V4L, serial, etc.).**
- **The framework factors out the common parts of drivers for the same type of devices to reduce code duplication**
- **From userspace, many are still seen as normal character devices**
- **The frameworks provide a coherent userspace interface (ioctl numbering and semantics, etc.) for every type of device, regardless of the driver**
  – The network framework of Linux provides a socket API such that an application can connect to a network using any network driver without knowing the details of the network driver
    - sockfd = socket(AF_INET, SOCK_STREAM, 0);

Linux Kernel Layers Focused on Frameworks



- A driver is always interfacing with:
  - A framework that allows the driver to expose the hardware features to userspace applications
  - A bus infrastructure (part of the device model), to detect/communicate with the hardware

Example Frameworks



Virtual File Systems - Overview

- **System and kernel information**
  - Presented to user space application as virtual file systems
  - Created dynamically and only exist in memory
- **Two virtual filesystems most known to users**
  - proc, mounted on /proc, contains operating system related information (processes, memory management parameters...)
    - This is an older mechanism that became somewhat chaotic
  - sysfs, mounted on /sys, contains a representation of the system as a set of devices and buses together with information about these devices
    - This is the newer mechanism and is the preferred place to add system information

Virtual File Systems - sysfs

- **The *sysfs* virtual filesystem is a mechanism for the kernel to export operating details to user space**
- **The kernel exports the following items to user space**
  - The bus, device, drivers, etc. structures internal to the kernel
  - */sys/bus/* contains the list of buses
  - */sys/devices/* contains the list of devices
  - */sys/class* enumerates devices by class (net, input, block...), whatever the bus they are connected to
- **Used for example by udev to provide automatic module loading, firmware loading, device file creation, etc. (more details on udev later)**
- **Take your time to explore /sys on your workstation**

Kernel Modules

- **The Linux kernel by design is a monolithic kernel, but is also modular**
- **The kernel can dynamically load and unload parts of the kernel code which are referred to kernel modules**
- **Modules allow the kernel capabilities to be extended without modifying the rest of the code or rebooting the kernel**
- **A kernel module can be inserted or removed while the kernel is running**
  - It can be inserted manually by a root user or from a user space script at startup
- **Kernel modules help to keep the kernel size to a minimum and makes the kernel very flexible**
- **Kernel modules are useful to reduce boot time since time is not spent initializing devices and kernel features that are only needed later**
- **Once loaded, kernel modules have full control and privileges in the system such that only the root user can load and unload modules**

Kernel Modules Details

- **Naming Convention: <file name>.ko**
- **Location: /lib/modules/<kernel_version> on the root filesystem**
- **Device drivers can be kernel modules or statically built into the kernel image**
  - The default kernel build from Xilinx generally builds most drivers into the kernel statically so they are started automatically
- **A kernel module is not necessarily a device driver; it is an extension of the kernel**
- **Kernel modules are loaded into virtual memory of the kernel**
  - Kernel virtual space is limited, but can be adjusted on the command line
- **Building a device driver as a module makes the development easier since it can be loaded, tested, and unloaded without rebooting the kernel**
  - FTP and NFS work well to transfer the module to the target file system

Should Your Functionality Be an Application or Kernel Module?

- **Consider the following comparison with an application being the default**

Application	Kernel Module
Runs in user space	Runs in kernel space
Perform a task from beginning to end	Registers itself in order to serve future requests
Linked to the appropriate library such as *glibc*	Linked only to the kernel
	The only functions it can call are those exported by the kernel

A First Simple Module – Hello World

```
#include <linux/init.h>
#include <linux/module.h>

static int __init simple_init(void)
{
    printk(KERN_ALERT "Hello World\n");
    return 0;
}

static void __exit simple_exit(void)
{
    printk(KERN_ALERT "Goodbye\n");
}

module_init(simple_init);
module_exit(simple_exit);
```

- **Some basic include files are needed for it to compile**
- **The initialization function simple_init() can register different types of facilities, including different kinds of devices, file systems, and more**
- **The exit function simple_exit() can unregister interfaces and returns all resources to the system**
- ***module_init* and *module_exit* Adds a special section to the module's object code stating where the module's initialization and exit functions are to be found**

Petalinux and Kernel Modules

- **Petalinux will create the makefile and a module skeleton for a kernel module using the petalinux-create command**
  - `petalinux-create -t modules -n simple --enable`
  - The module is created in the components/modules/simple directory
  - The module can only be disabled from building thru the petalinux configuration process
- **Petalinux will build the kernel module when the software system is built**
  - Or it can build it only by specifying the module in the root filesystem
- **There is documentation in the kernel tree describing the build process (Documentation/kbuild/modules.txt)**
- **It is possible to build a module without a makefile**
  - `make ARCH=arm -C <kernel directory> M=$PWD`
  - The kernel tree needs to have been configured and prepared to allow a module to be built against it

Testing A Kernel Module

- **An easy way to test a module is to FTP the module to the embedded target assuming the target has an FTP server running**
  - The Petalinux root file system includes FTP so that it is ready to use
  - FTP under Petalinux defaults to the /var/ftp directory
  - It is easy to insert the module from the /var/ftp directory
- **The module is loaded using the *insmod* or *modprobe* command**
  - The **modprobe** command loads modules from a standard path in the root file system (/lib/modules/*) and also loads dependent modules
  - The **insmod** command only loads the specified module
- **The module is unloaded using the *rmmod* command, then a new version of the module can be inserted**
- **A buggy module can hang the kernel such that a reboot is needed**
- **Character device drivers are easy to test from the command line shell with cat, echo, and dd**

Device Tree In A Nutshell – Page 1

- **The principle of the Device Tree is to separate a large part of the hardware description from the kernel sources**
- **Device Tree allows a single kernel image to run on different boards with the differences being described in the device tree**
- **This mechanism takes its roots from OpenFirmware (OF) used on PowerPC platforms. This is why the "of" is part of some kernel functions.**
- **Device Tree is a tree of nodes that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board**
- **Each node can have a number of properties describing various properties of the devices: addresses, interrupts, clocks, etc.**
- **Written in a specialized language, the Device Tree source code is compiled into a Device Tree Blob by the Device Tree Compiler (DTC)**

Device Tree In A Nutshell – Page 2

- **The DTC checks the device tree syntax but the semantics of the device tree are checked at runtime by the kernel and drivers**
- **At boot time, the kernel is given a compiled device tree, referred to as a Device Tree Blob, which is parsed to instantiate all the devices described in the device tree**
- **Device trees are located in the kernel tree at arch/<arm or microblaze>/boot/dts**
- **The device tree compiler is part of the Linux kernel tree**
- **Some key properties in a device tree node, referred to as bindings**
  - The *compatible* property is used to bind a device with a device driver
  - The *interrupts* property contains the interrupt number used by the device
  - The *reg* property contains the memory range used by the device
- **There is limited documentation for the device tree bindings for each device such that driver code inspection may be necessary**
  - The docs are in the kernel tree at Documentation/devicetree/bindings

Device Tree Details and A Simple Example

- **A simple example below illustrates a node of a device tree**
  - An AMBA bus with a GPIO that has registers mapped to 0x4120000 and is using interrupt 91
    - 91 – 32 = 59, where 32 is the first Shared Peripheral Interrupt
  - The device is compatible with a driver containing a matching compatible string of “xlnx,simple”
  - The device driver source code may be the only way to really understand what properties it is expecting from the device tree

```
ps7_axi_interconnect_0: amba@0 {
          #address-cells = <1>;
          #size-cells = <1>;
          compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
          ranges ;
          axi_gpio_0: gpio@41200000 {
                    #gpio-cells = <2>;
                    compatible = "xlnx,simple";
                    gpio-controller ;
                    interrupt-parent = <&ps7_scugic_0>;
                    interrupts = <0 59 4>;
                    reg = <0x41200000 0x10000>;
                    xlnx,is-dual = <0x1>;
          } ;
};
```

Device Tree – Breaking News for 2014.2

- It's no longer just *.dts files, now there are *.dtsi files
- The dtsi files are included files while the dts file is the final device tree
- This is a nice feature the Linux kernel has had for several years that Xilinx was not using (yes it is a change that you need to deal with)
- A dts file includes dtsi files and the inclusion process works by overlaying the tree of the including file over the tree of the included file
- When properties are repeated in dtsi files the last one is the final
- The PL and PS are separate DTSI files while there is top level dts file that includes them
- The device tree compiler can be used to create the final device tree which is handy for debug (by specifying DTS input and output)

Device Tree – Inclusion Example

ps.dtsi (included file)

```
ps7_ttc_1: ps7-ttc@0xf8002000 {
  clocks = <&clkc 6>;
  compatible = "xlnx,ps7-ttc-1.00.a";
  interrupt-parent = <&ps7_scugic_0>;
  interrupts = <0 37 4>,<0 38 4>,<0 39 4>;
  reg = <0xF8002000 0x1000>;
  status = "disabled";
} ;
```

\+

system-top.dts (including file)

```
/include/ "ps.dtsi"

&ps7_ttc_1 {
  compatible = "xlnx,psttc", "generic-uio";
  status = "okay";
};
```

Note the "&" used to reference an existing node (rather than creating a new node

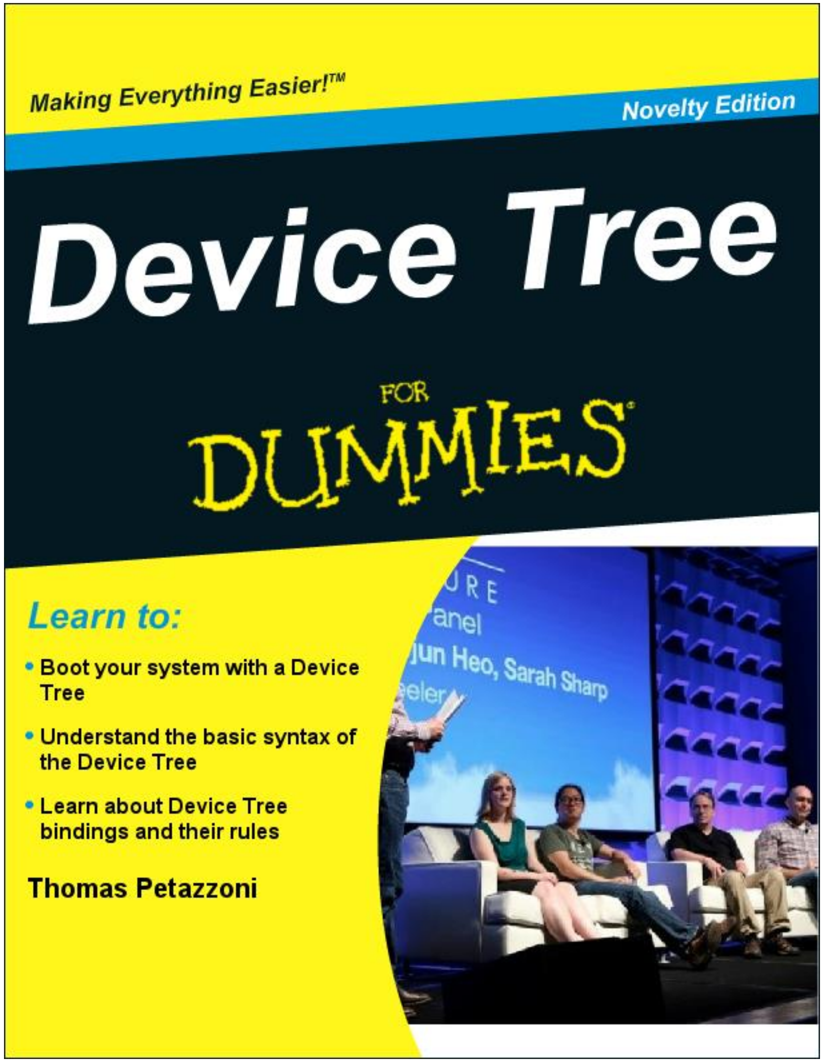
=

```
ps7_ttc_1: ps7-ttc@0xf8002000 {
  clocks = <&clkc 6>;
  compatible = "xlnx,psttc", "generic-uio";
  interrupt-parent = <&ps7_scugic_0>;
  interrupts = <0 37 4>, <0 38 4>, <0 39 4>;
  reg = <0xF8002000 0x1000>;
  status = "okay";
} ;
```

The result for the duplicated (red) properties is the same as the including file.

Device Tree – Where To Find More Details

- There are a lot of technical details not covered in this presentation, as Device Tree could be a complete presentation by itself
- Now there are some good references such as a “Device Tree For Dummies” PDF by Thomas Petazzoni
- You need to know the basics for device driver operation, but don’t have to be an expert



Lab 1

- **Stop here and do Lab 1**
  - Get board setup, copy images to the SD card, boot Linux
  - Verify network connectivity with host
  - Verify FTP working from host to board
  - Create a basic kernel module using Petalinux
  - Build it and test it on the board

Device Nodes

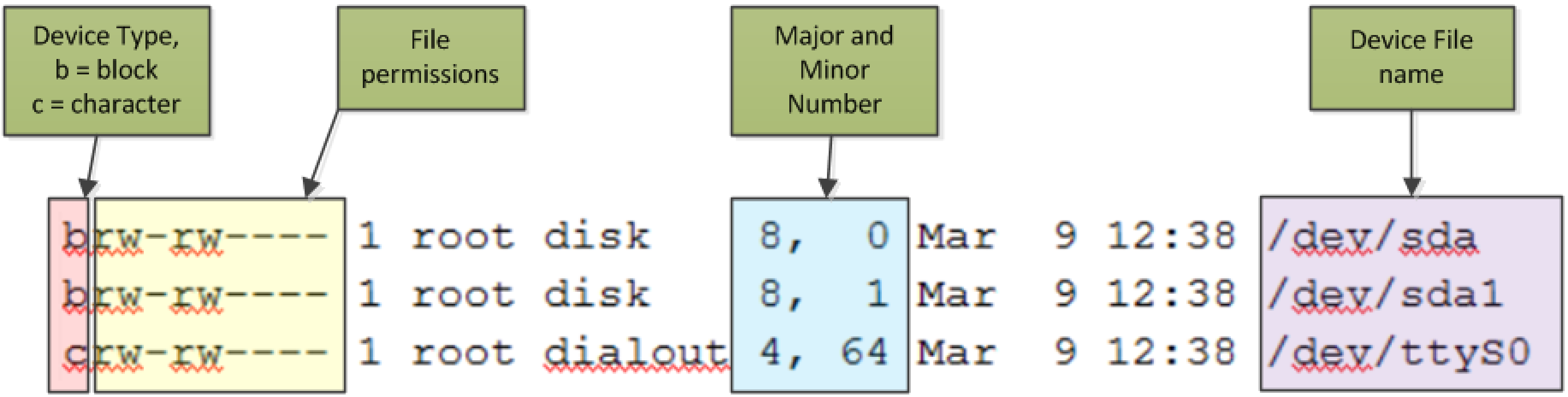
- **Devices in the kernel are block or character devices and are identified using a major and a minor number**
- **The *major number* indicates the family of the device**
- **The *minor number* indicates the number of the device to allow multiple instances of a major device type**
- **Major and minor numbers can be statically or dynamically allocated**
  - The statically allocated numbers are typically identical across Linux systems
- **Device Nodes are documented in the kernel tree at Documentation/devices.txt**

Device Files

- Most system objects in UNIX are represented as files
- This allows applications to manipulate system objects with the normal file operations (open, read, write, close, etc.)
- Devices are represented as files to the applications through device files
- A device file is a special type of file that associates a file name visible to userspace applications to the triplet (type, major, minor) that the kernel understands
- Device files are stored in the /dev directory of the root file system

Device File Examples

- Device files in the file system are illustrated below (ls /dev –al)



- Example C code that uses the file API to write data to a serial port

```
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Device File Creation

- **Device files can be created manually using the mknod command**
  - `mknod /dev/<device> [c|b] major minor`
  - Needs root privileges
- **Components can be added to create/remove device files automatically when devices appear and disappear**
  - devtmpfs virtual filesystem (built into the Xilinx kernel by default)
  - udev, solution used by desktop and server Linux systems
    - Udev runs as a daemon and listens for uevents the kernel sends out when a new device is initialized or removed from the system
  - mdev, a lighter solution than udev, provided in BusyBox
    - BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc.

Platform Devices

- **Hardware devices may be connected through a bus allowing enumeration, hotplugging, or providing unique identifiers for devices (such as PCI, PCIe and USB)**
- **On embedded systems, devices are often <u>not</u> connected through a bus which allows the devices to be uniquely identified.**
  - Many devices are directly part of a system-on-chip: UARTs, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- **In this case devices, instead of being dynamically detected, must be statically described in either the kernel source code or the device tree**
- **The platform bus, a software abstraction, was created to handle such devices. It supports platform drivers that handle platform devices.**
- **The platform bus works like other buses (USB, PCI), except that devices are enumerated statically rather than being discovered dynamically**

Platform Devices And Device Tree

- **As platform devices cannot be detected dynamically, they are defined statically using a device tree**
- **Platform devices can also be defined in source code (as was done before device tree in the ARM kernel)**
  - This is not typically done anymore as device tree operation is encouraged
- **Each device managed by a particular driver typically uses different hardware resources such as interrupts and I/O addresses**
- **Device tree processing in the kernel is responsible for adding platform devices to the platform bus**

Platform Driver

- **A platform driver is a device driver for a specific platform device on the platform bus**
  - Most Xilinx Linux device drivers used by customers are platform drivers
- **A platform driver does not inherently have any interface to user space without hooking into a kernel framework, such as the character device framework**
  - The name *platform* only specifies the bus (the platform bus) that the device is located on
  - Character, block, and network device drivers can all be platform device drivers if the device they support is located on the platform bus
- **Platform drivers follow the standard driver model convention except discovery/enumeration is handled outside the drivers**
  - In ARM and Microblaze Linux architectures, device tree processing does the discovery/enumeration of the platform bus
- **Platform devices and drivers are described in the kernel tree at Documentation/driver-model/platform.txt**

Platform Driver Initialization

- **A Platform driver is connected to the kernel by the *platform_driver_register()* function**
- **The kernel calls the *probe()* function of the driver when it discovers the corresponding platform device**
- **The probe() function is responsible for initializing the device, mapping I/O memory, and registering the interrupt handlers**
  - The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information
- **The probe() function also registers the device to the kernel framework**
  - An example framework is the character device processing for a character driver

Platform Driver Exit

- The kernel calls the *remove()* function of the driver when the corresponding platform device is no longer used by the kernel
- The remove() function is responsible for unregistering the device from the kernel framework and shuting it down
- A platform driver is disconnected from the kernel by the *platform_driver_unregister()* function

Platform Driver Resources – Page 1

- **The platform driver has access to the I/O resources (memory address and interrupt) in the device tree through a kernel API**
- **This is an area of the API that has been changing such that you can see other methods which may require more effort**
- **platform_get_resource() gets the memory range for the device from the device tree**
- **platform_get_irq() gets the interrupt for the device from the device tree**
- **These kernel functions automatically read standard platform device parameters from the platform device in the device tree**
- **Other non-standard or user defined parameters can be read from the device tree using other kernel functions named of_***

Platform Driver Resources – Page 2

- **devm_ioremap_resource() maps the physical memory range of the device into the virtual memory map**
  - The memory attributes for this memory range default to non-cached
- **devm_request_irq() connects the interrupt handler to the interrupt processing of the kernel**
- **The devm*() functions of the kernel framework are kernel managed resources which the kernel tracks and then automatically handles them when the device goes away**

Platform Device Driver In A Kernel Module

```
static struct platform_driver simple_driver = {
};

static int __init simple_init(void)
{
	return platform_driver_register(&simple_driver);
}

static void __exit simple_exit(void)
{
	platform_driver_unregister(&simple_driver);
}

module_init(simple_init);
module_exit(simple_exit);
```

- Starting with a simple kernel module
  1. Make it a simple empty platform driver
  2. Platform driver *simple_driver* is connected to the kernel by the *platform_driver_register()* function
  3. Platform driver *simple_driver* is disconnected from the kernel by the *platform_driver_unregister()* function
- The module initialization function *simple_init()* is called when the module is inserted
- The module exit function *simple_exit()* is called when the module is removed

Platform Devices Driver Basics

```
static int simple_probe(struct platform_device *pdev)
{
}
static int simple_remove(struct platform_device *pdev)
{
}

static struct of_device_id simple_of_match[] = {
        { .compatible = "xilinx,simple", },
        { /* end of list */ },
};

static struct platform_driver simple_driver = {
        .driver = {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = simple_of_match,
        },
        .probe          = simple_probe,
        .remove         = simple_remove,
};
```

1. Create the simple_probe() and simple_remove() functions which will be called by the kernel when the driver is bound to a device
2. Create the simple_of_match data structure which is used to bind the driver to the device and matches the device tree
3. Create the platform driver data structure describing the platform driver simple_driver
4. The compatible member of simple_of_match data is connected to the kernel in the structure simple_driver
5. The simple_probe() and simple_remove() functions are connected to the kernel in the structure simple_driver

Platform Device Driver Memory/Interrupt Details

```
int simple_irq() { };

int simple_probe()
{
        resource = platform_get_resource(pdev, IORESOURCE_MEM, 0);

        base_address = devm_ioremap_resource(dev, resource);

        irq = platform_get_irq(pdev, 0);
        devm_request_irq(dev, irq, &simple_irq, 0, DRIVER_NAME, lp);
}
```

A simple example without error processing

- **Get the device memory range from the device tree by calling platform_get_resource()**
- **The devm_ioremap_resource() function is called to map the device physical memory into the virtual address space**
- **Get the interrupt number from the device tree by calling platform_get_irq()**
- **The interrupt function simple_irq() is connected to the kernel by calling devm_request_irq() function**

Character Device Driver

- Character drivers can be useful for many customer IP blocks
- From the point of view of an application, a character device is essentially a file
- The driver of a character device implements operations that let applications access the device as a file: open, close, read, write, etc.
- A character driver implements the operations in the struct file_operations structure and registers them
- The Linux virtual filesystem layer calls the driver's operations when a userspace application makes the corresponding system call
- A platform device driver can also be a character device driver if it implements the interface and registers with the kernel framework

Character Device Driver File Operations

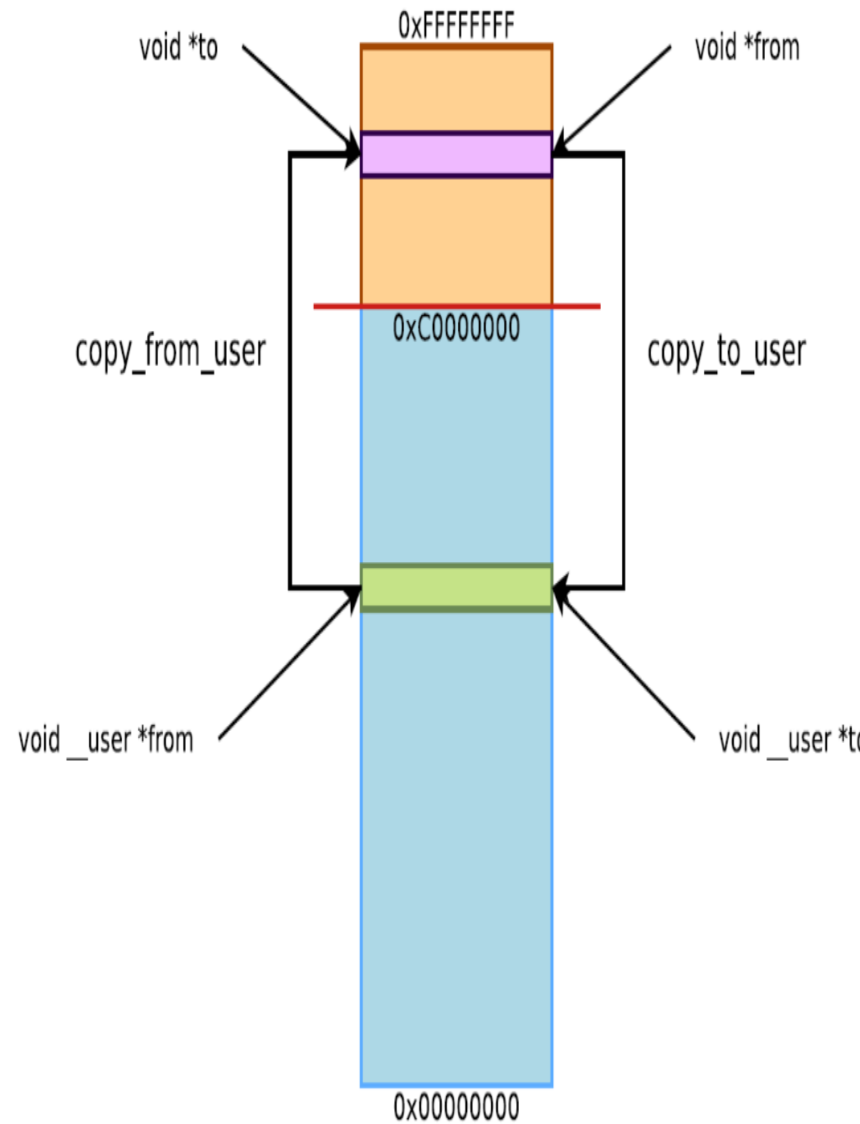
- **There are a number of operations that a character device can optionally support**
  - The open(), read(), write() and release() functions are typically implemented as a minimum
- ***open()* function**
  - Called when a userspace application opens a device file
  - Contains details such as the current position, the opening mode, etc.
  - Has a void *private_data pointer that one can freely use
- ***release()* function**
  - Called when userspace application closes the file
- ***ioctl()* function**
  - Called by a userspace application to perform some special I/O operation which does not fit neatly into the read/write interface of a character device
  - Examples might be to control the baud rate of the serial port such that no data is sent through the serial port, but its configuration is altered

File Operations, Read and Write Details

- ***read()*** **function**
  - Called when a userspace application calls the read() library function for the device
  - Reads data from the device, writes a specified maximum number of bytes in the user-space buffer, and updates the file status
  - Returns the number of bytes read
  - Can block when there isn't enough data to read from the device
- ***write()*** **function**
  - Called when a userspace application calls the write() library function for the device
  - Reads a specified number of bytes from a userspace buffer, writes the data to the device, updates the file status
  - Returns the number of bytes written
  - Can block when the device is not ready to accept the data

Copying Data Between Kernel and User Space

- 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
- Zero copy methods exist but are more complex and less typical



Character Device Driver Details

- **A character device framework is provided by the kernel. This framework allows the device to be accessed using the file I/O operations.**

- ***alloc_chrdev_region()*** **allocates a character device number**
- ***unregister_chrdev_region()*** **frees a previously allocated character device number**

- ***cdev_init()*** **initializes the character device structure**
- ***cdev_add()*** **adds the character device to the kernel**
- ***cdev_del()*** **removes the character device from the kernel**

Creating A Character Device Simplified Example

```
int simple_open() { };
int simple_write() { };
int simple_read() { };
int simple_release() { };

static struct file_operations simple_fops = {
        .owner    = THIS_MODULE,
        .open     = simple_open,
        .write    = simple_write,
        .read   = simple_read,
        .release  = simple_release,
};
int simple_probe()
{
        struct cdev cdev;
        cdev_init(&cdev, &simple_fops);
        cdev_add(&cdev, ….);
}
```

- Create empty file operation functions simple_open(), simple_write(), simple_read(), simple_release()
- Create the file_operations data structure simple_fops
- The platform driver simple_probe() 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 simple_read() and simple_write()
- The cdev_add() function connects the character device to the kernel

Creating The Device Node Details

- **A device node, as reviewed earlier, is needed to allow user space to communicate with kernel space**
- **Many people create device nodes manually as they were done in the past, but using the API takes care of this**
- **A class for the device in /sys is needed to allow a device node in /dev to be automatically created**
  - The class for the device in /sys is seen as a directory
- **The driver creates the class using the kernel API.**
  - ***class_create()*** creates a class in the /sys/class directory
  - ***class_destroy()*** destroys the class
- **The driver creates the device node using the kernel API.**
  - ***device_create()*** creates a device node in the /dev directory
  - ***device_destroy()*** removes a device node in the /dev directory

Sys FileSystem Attributes Rather Than Ioctl

- **The ioctl interface of device drivers is an older interface can be less preferred in the kernel community**
  - It is hard to document the interface for each driver which is typically unique
- **File attributes in the sys filesystem is the preferred method rather than ioctl**
  - They are more self documenting
  - They are easier to use as they can be accessed from a command line using utilities like cat, echo and dd
  - For example, cat /sys/devices/amba.0/41200000.gpio/irqreg displays the contents of the interrupt register for the GPIO device
  - Slower to access due to open and close
- ***device_create_file()*** **creates a file attribute under the current device in /sys**
- ***device_remove_file()*** **removes a file attribute under the current device in /sys**

Creating A Sys File Attribute Details

- **A file attribute is created in the directory of the device in filesystem at /sys/devices/<bus>/<device>**
  - The path is dependent on the node of the device in the device tree
- **Before calling *device_create_file()* to create the attribute, the attribute data structure must be created**
- **The macro DEVICE_ATTR() is used to create the attribute and requires the following inputs.**
  - A name for the attribute in the filesystem
  - Permissions which determine if the attribute can be read and/or written
  - A function to read the data from the driver
  - A function to write the data into the driver

Creating A File Attribute Simplified Example

- A platform driver creates a file attribute /sys/devices/<bus>/<device>/irqreg which can be read and written from user space

```
simple_show_reg() { }
simple_store_reg() { }

DEVICE_ATTR(irqreg, S_IWUSR | S_IRUGO,
            simple_show_reg, simple_store_reg);

simple_probe()
{
    device_create_file(dev, &dev_attr_irqreg);
}
```

The permissions to allow read/write

- Macros in Linux can be less obvious as details are hidden. DEVICE_ATTR() causes the data structure dev_attr_irqreg to be created.

- Create the empty access functions simple_show_reg() and simple_store_reg() which will be called to read/write the data
- Create the attribute data structure for attribute irqreq which has read and write access and uses the access functions just created
- The platform driver simple_probe() function creates the file attribute named irqreg in the sys filesystem under the device by calling device_create_file()
- The access functions, simple_show_reg() and simple_store_reg() are connected to the kernel

Debugging With Printk

- **printk()**
  - Kernel version of printf()
  - Priority of kernel messages (log level) can be specified with the following symbols defined in <linux/kernel.h>
    - KERN_EMERG: Emergency message
    - KERN_ALERT: Alert message
    - KERN_CRIT: Critical situation
    - KERN_ERR: Error report
    - KERN_WARNING: Warning message
    - KERN_NOTICE: Noticeable message
    - KERN_INFO: Information
    - KERN_DEBUG: Debug message
  - Does not support floating point numbers
  - Example:

    printk(KERN_DEBUG "line %s:%i\n", __FILE__, __LINE__);
- **The log level can be altered from the command line in the proc file system**
  - "echo 7 > /proc/sys/kernel/printk" changes the current level so all messages are printed

Other Debug Tools

- **The sys and proc file systems contain a lot of good information**
- **/proc/interrupts shows the interrupt number assigned to a device and the number of interrupts that have occurred**
- **/proc/device-tree has the nodes of the device tree**
  - Some nodes are binary such that hexdump must be used to view them
- **/proc/cmdline has the kernel command line, which can be handy**
- **/proc/iomem shows the I/O memory claimed by device drivers**

References

- **Linux Device Drivers Version 3**
  - https://lwn.net/Kernel/LDD3/
- **Free Electrons**
  - http://free-electrons.com/
  - http://lxr.free-electrons.com/
- **Linux Foundation**
  - http://training.linuxfoundation.org/free-linux-training/linux-training-videos/how-to-build-character-drivers-for-the-linux-kernel
  - http://training.linuxfoundation.org/free-linux-training/linux-training-videos/interrupt-handling-in-linux-device-drivers

Lab 2

- **Complete a platform character device driver**
  - Get a platform driver working
  - Add character device functionality
  - Build the driver
  - Test the driver on the board