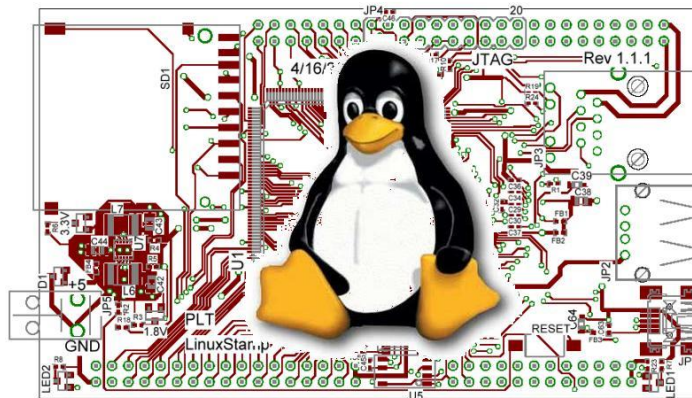
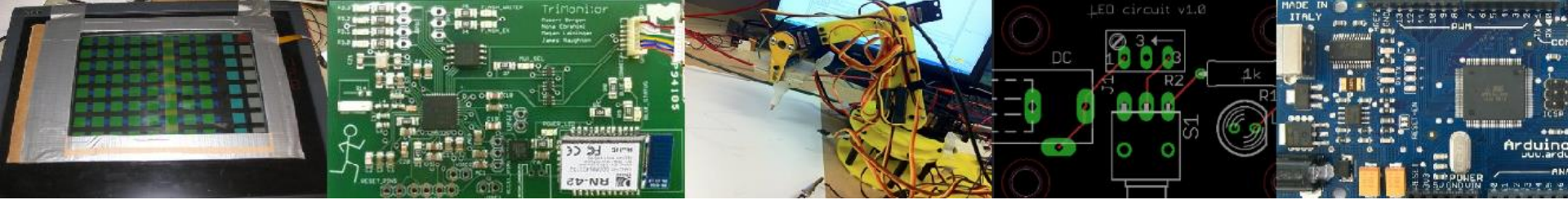


# Finish Licensing & Embedded Linux Lectures 6 and 7

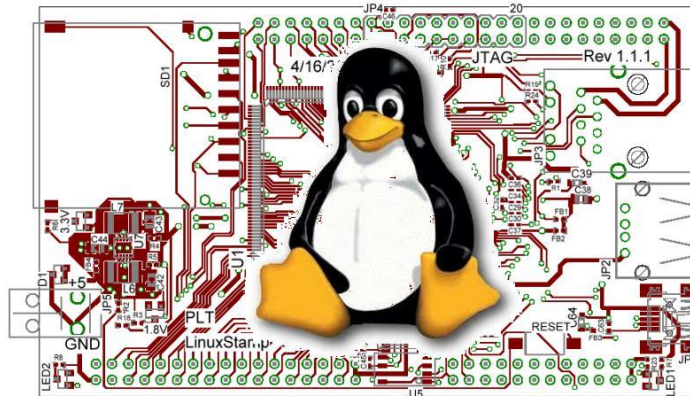


# NOTE:

- Be sure to register your RPi.
  - See Piazza post [@59](#) (pinned)
  - Basically, bring your Pi to lab this week and register it or follow directions for doing it from home.



# Copyright, Copyleft, and Legal Issues



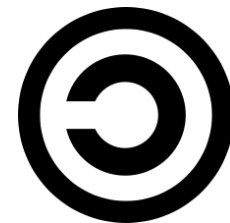
# The basic copyright options

- No license:
  - Without a license, the code is copyrighted by default. People can read the code, but they have no legal right to use it. To use the code, you must contact the author directly and ask permission.
- Public domain:
  - If your code is in the public domain, anyone may use your code for any purpose whatsoever. Nothing is in the public domain by default; you have to explicitly put your work in the public domain if you want it there. Otherwise, you must be dead a long time before your work reverts to the public domain.

# Linux (common usage of the word)

- A POSIX-compliant and widely deployed desktop/server operating system licensed under the GPL
  - POSIX
    - Unix-like environment (shell, standard programs like awk etc.)
  - Desktop OS
    - Designed for users and servers
    - Not designed for embedded systems
  - GPL
    - GNU Public License. May mean you need to make source code available to others.
      - First “copyleft” license.
    - Linux is licensed under GPL-2, not GPL-3.

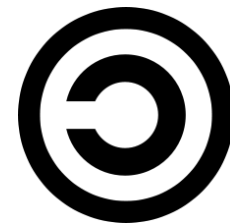




# GPL in three slides (1/3)

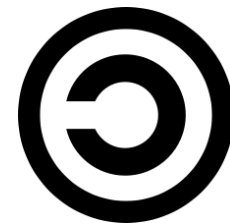
- A licensee of GPL v2-licensed software can:
  - copy and distribute the program's unmodified source code
  - modify the program's source code and distribute the modified source
  - distribute compiled versions of the program, both modified and unmodified
- Provided that:
  - all distributed copies (modified or not) carry a copyright notice and exclusion of warranty
  - all modified copies are distributed under the GPL v2
  - all compiled versions of the program are accompanied by the relevant source code, or a viable offer to make the relevant source code available





# GPL in three slides (2/3)

- Some points
  - If you don't redistribute the code, you don't need to share the source.
  - You can bundle software with GPL-ed software and not have to license the bundled software.
    - “Mere aggregations” aren't impacted.
  - Loadable Kernel Modules are tricky though
    - Often we need device drivers for our application (we'll be writing them later)
    - But they touch the Linux code in a non-trivial way.
      - There is some debate about if a LKM is an aggregation or a modification of the original kernel.
      - In general there are proprietary drivers out there and even open source groups that help support said drivers.
- General theme:
  - Be sure you understand the law before you use software licensed under the GPL on a proprietary project.
    - Using gcc to compile or ddd to debug is fine, but when you are modifying the code of software licensed under the GPL you might be obligated to release your code.
- Read (or at least scan): *The Cathedral and the Bazaar* before the midterm.



# GPL in three slides (3/3)

- GPL v3
  - Prevents using GPL on hardware that won't run other code (“Tivoization”)
    - Though only for consumer hardware (IBM has a business model here?)
  - Addresses patents
    - Can't sue for (software?) patent on code you release.
- Lesser GPL
  - Mainly for libraries/APIs.
  - Makes it clear can use libraries in proprietary code without having to release proprietary code.



# OK, one more

- gcc is GPL v3
  - But has a runtime exception.
  - When you use GCC to compile a program, GCC may combine portions of certain GCC header files and runtime libraries with the compiled program. The purpose of this Exception is to allow compilation of non-GPL (including proprietary) programs to use, in this way, the header files and runtime libraries covered by this Exception.
- Linux is GPL v2
  - Fear of limiting DRM and private keys keeps them away from v3.
- There is a short preamble:
  - This copyright does *\*not\** cover user programs that use kernel services by normal system calls - this is merely considered normal use of the kernel, and does *\*not\** fall under the heading of "derived work".

# Another common License:

## Creative Commons

- These are basically a free *configurable* license.
  - Attribution (**by**)
    - Must give author credit
  - ShareAlike (**sa**)
    - Like GPL, they must distribute any changes.
  - NonCommercial (**nc**)
    - Only for non-commercial
  - NoDerivatives (**nd**)
    - Can't make changes.
- Idea is, someone smart wrote the words to do what you probably want to do.
  - You still hold ownership, so if they want a different license, they can come to you to negotiate.
- Wikipedia is CC BY-SA

Creative Commons is generally a poor license for code but still used for that.

See <https://creativecommons.org/faq/>

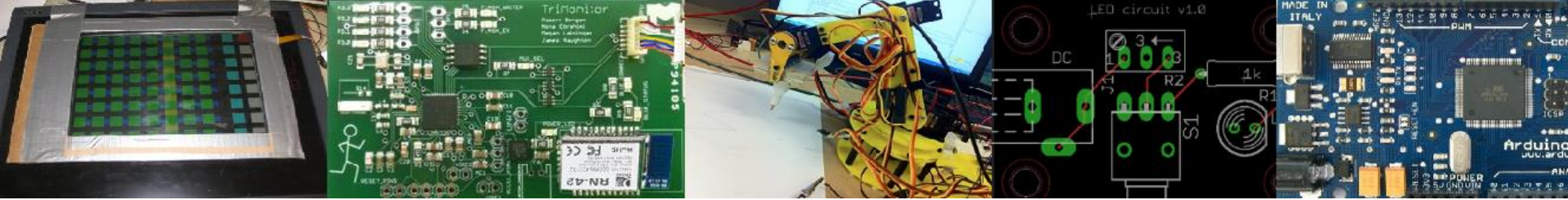
But quite useful for documentation and databases

# MIT license

- Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
- The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
- (And then a disclaimer of liability)
- Basically, just grants permission.
  - Can add to a closed-source product, etc.
  - More-or-less public domain with the disclaimer of liability being required in any place the code is used.

# Fair use

- Fair use lets you use copyrighted works
  - But in general there are guidelines for what you can use.
- Four factors are:
  - the purpose and character of your use
    - Transformative or not (parody falls here).
  - the nature of the copyrighted work
    - Published/not published, factual works.
  - the amount and substantiality of the portion taken
    - The amount taken and the % taken both play a role.
  - the effect of the use upon the potential market
    - If you reduce the market for the original work, that's a problem.
- Fair use in code is tricky at best.
  - APIs sometimes fit here (see Oracle America vs. Google Inc.)



# Start on Linux



Many figures and text in this section taken from *Embedded Linux Primer*, second edition  
We (kind of) have on-line access to the book.

Also [http://freesoftwaremagazine.com/articles/drivers\\_linux/](http://freesoftwaremagazine.com/articles/drivers_linux/) is used a lot!

# Two main tasks

- Introduce Linux and in particular embedded Linux.
  - Pretty high-level.
  - Lots of material coming pretty quickly.
- How to write device drivers for Linux.
  - Much more detailed
  - Pretty complex coding
  - Will do in lab 4.



# Linux Introduction

- Background
  - Kernel overview, legal issues, versioning, building, user basics, FHS, booting
- Embedded Linux (common but not required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

Warning: We are going to do this section very quickly indeed. There is a lot of stuff I want you to see. There is enough stuff in this *section* of this lecture for 20 hours of discussion. We don't have that kind of time. Just going to lightly touch on some stuff.

# Linux

- The software commonly referred to as “Linux” is really a number of things glued together
  - The Linux Kernel
  - GNU tools (bash, ls, emacs, etc.)
  - Usually also
    - A package manager (dpkg, RPM, etc.)
    - A desktop environment (GNOME, KDE, etc.)
- Linux distributions (“distros”)
  - Collections of the above tools (Debian, Red Hat, etc.)
  - These distros are often labeled as a “Linux-based OS”
- The Kernel isn’t an OS by itself
  - The distros are

# What is a kernel? (1/2)

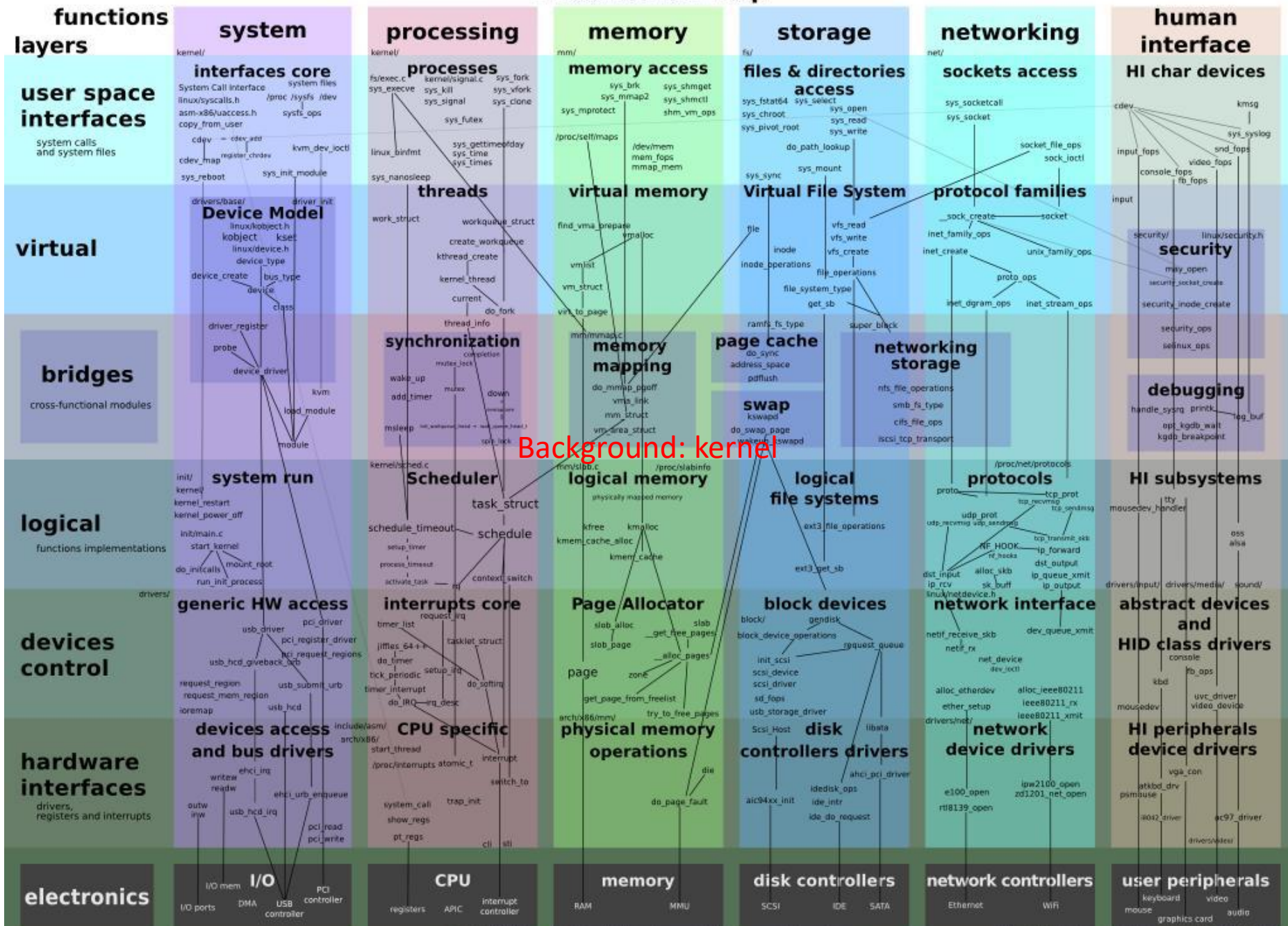
- The kernel's job is to talk to the hardware and software, and to manage the system's resources as best as possible.
  - It talks to the hardware via the drivers that are included in the kernel (or additionally installed later on in the form of a “kernel module”).
    - This way, when an application wants to do something (say change the volume setting of the speakers), it can just submit that request to the kernel, and the kernel can use the driver it has for the speakers to actually change the volume.

# What is a kernel? (2/2)

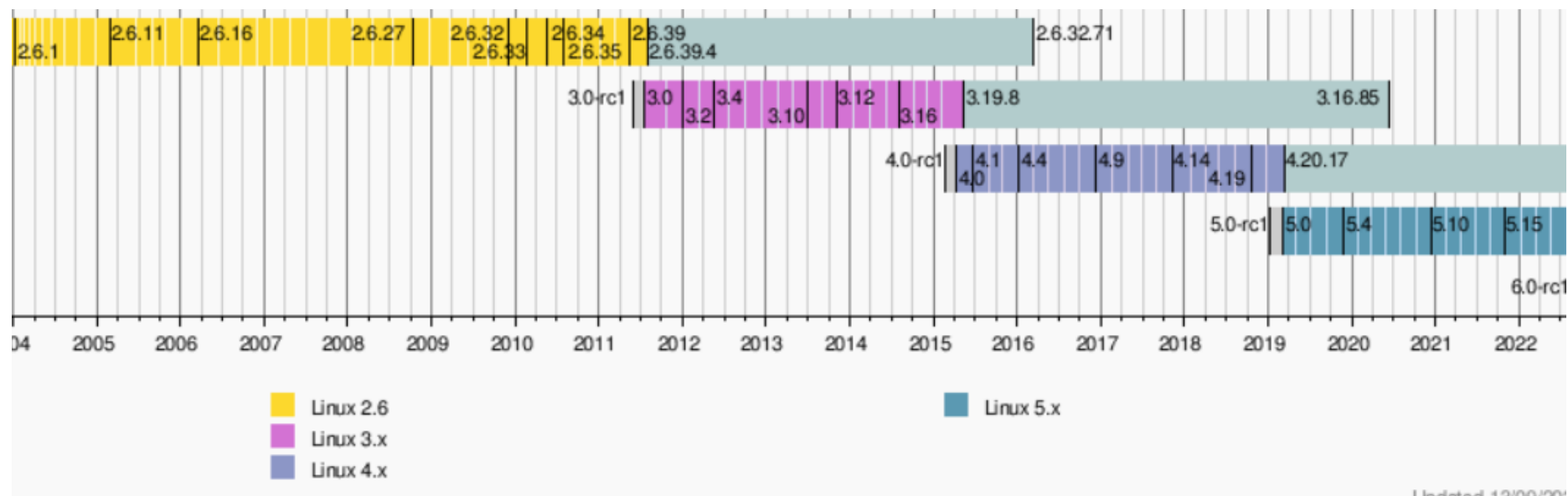
- The kernel is highly involved in resource management.
  - It has to make sure that there is enough memory available for an application to run, as well as to place an application in the right location in memory.
  - It tries to optimize the usage of the processor so that it can complete tasks as quickly as possible.
    - It also aims to avoid deadlocks.
  - It's a fairly complicated circus act to coordinate all of those things, but it needs to be done and that's what the kernel is for.

## Background: kernel

## Linux kernel map



# Linux kernel history



- 3.0 isn't a radical change from 2.6, instead a 2.6 upgrade was move to 3.0 for Linux's 20<sup>th</sup> anniversary.

- “So what are the big changes? NOTHING. Absolutely nothing. Sure, we have the usual two thirds driver changes, and a lot of random fixes, but the point is that 3.0 is \*just\* about renumbering...”

[<https://lkml.org/lkml/2011/5/29/204>]

- And 4.0 is more of the same.
- <http://www.itworld.com/article/2887558/linux-torvalds-bumps-linux-kernel-to-version-4x.html>

I'd like to point out (yet again) that we don't do feature-based releases, and that “5.0” doesn't mean anything more than that the 4.x numbers started getting big enough that I ran out of fingers and toes.

Linus Torvalds



# What version am I working with?

- If running, use “uname” command
  - “uname -a” for all information
- If looking at source
  - First few lines of the top-level Makefile will tell you.

# How do I download and build the kernel?

- Use git.
- Typing “make” with no target at the top-level should build the kernel.
  - Need gcc installed (no other compiler will do).
  - Should generate an ELF file called “vmlinux”
    - But lots of configuration stuff

# Linux user basics--shell

- You have a *shell* which handles user commands
  - May just search for an executable file (application) in certain locations
  - Allows for moving data between those applications.
    - Pipes etc.
  - Is itself a programming language.
  - There are many of these (bash, sh, tcsh, csh, ksh, zsh)
    - Most have very similar interfaces (type application name, it runs), but the programming language part varies quite a bit.
    - Geek humor:
      - **sh** is called the Bourne shell, written by Stephen Bourne
      - **bash**, often treated as an upgrade to **sh**, is the “Bourne again shell”

# Linux user basics—file systems

- Linux supports a huge variety of file systems.
  - But they have some commonalities.
- Pretty much a standard directory structure with each directory holding either other directories or files.
  - Each file and directory has a set of permissions.
    - One owner (a single user)
    - One group (a list of users who may have special access)
    - There are three permissions, read, write and execute
      - Specified for owner, group, and world.
- There are also links (hard and soft)
  - So rather than copying files I can point to them.

# FHS:

## File System Hierarchy Standard

- There is a standard for laying out file systems
  - Part of this is the standard top-level directories

TABLE 6-1 Top-Level Directories

Directory	Contents
bin	Binary executables, usable by all users on the system <sup>1</sup>
dev	Device nodes (see Chapter 8, “Device Driver Basics”)
etc	Local system configuration files
home	User account files
lib	System libraries, such as the standard C library and many others
sbin	Binary executables usually reserved for superuser accounts on the system
tmp	Temporary files
usr	A secondary file system hierarchy for application programs, usually read-only
var	Contains variable files, such as system logs and temporary configuration files

# A “minimal” file system

LISTING 6-1 Contents of a Minimal Root File System

```
.
|-- bin
|   |-- busybox
|   '-- sh -> busybox
|-- dev
|   '-- console
|-- etc
|   '-- init.d
|       '-- rcS
'-- lib
    |-- ld-2.3.2.so
    |-- ld-linux.so.2 -> ld-2.3.2.so
    |-- libc-2.3.2.so
    '-- libc.so.6 -> libc-2.3.2.so
```

5 directories, 8 files

- Busybox is covered later



# Background: booting

1. Some circuit magic happens
  - Get clock running, reset registers etc.
2. Bootloader starts
  - Initialize devices such as I2C, serial, DRAM, cache, etc.
  - Starts the OS
3. Kernel starts
  - Might set up other things needed
4. Init gets called
  - Lots of stuff...

# Outline

- Background
  - Legal issues, versioning, building, user basics, FHS, booting
- **Embedded Linux** (common but not required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

# What makes a Linux install “embedded”?

- It’s one of those poorly defined terms, but in general it will have one or more of the following
  - small footprint
  - flash files system
  - real-time extensions of some sort

# Small footprint--Busybox

- A single executable that implements the functionality of a massive number of standard Linux utilities
- `ls, gzip, ln, vi`. Pretty much everything you normally need.
- Some have limited features
  - `Gzip` only does the basics for example.
  - Pick which utilities you want it to do
    - Can either drop support altogether or install real version if needed
  - Highly configurable (similar to Linux itself), easy to cross-compile.
- Example given is around 2MB statically compiled!

# Using busybox

- Two ways to play
  - Invoke from the command line as busybox
    - **busybox ls /**
  - Or create a softlink to busybox and it will run as the name of that link.
    - So if you have a softlink to busybox named “ls” it will run as ls.

# Links done for you

- Normally speaking, you'll use the softlink option.
  - You can get it to put in all the links for you with “make install”
    - Be darn careful you don't overwrite things locally if you are doing this on the host machine.
      - That would be bad.

```
| -- bin
|   |-- busybox
|   |-- cat -> busybox
|   |-- dmesg -> busybox
|   |-- echo -> busybox
|   |-- hostname -> busybox
|   |-- ls -> busybox
|   |-- ps -> busybox
|   |-- pwd -> busybox
```

```
$ make CONFIG_PREFIX=/mnt/remote install
/mnt/remote/bin/ash -> busybox
/mnt/remote/bin/cat -> busybox
/mnt/remote/bin/chgrp -> busybox
/mnt/remote/bin/chmod -> busybox
/mnt/remote/bin/chown -> busybox
...
/mnt/remote/usr/bin/xargs -> ../../bin/busybox
/mnt/remote/usr/bin/yes -> ../../bin/busybox
/mnt/remote/usr/sbin/chroot -> ../../bin/busybox
```



# System initialization

- Busybot can also be “init”
  - But it’s a simpler/different version than the init material covered above.
  - More “bash” like

```
#!/bin/sh
echo "Mounting proc"
mount -t proc /proc /proc

echo "Starting system loggers"
syslogd
klogd

echo "Configuring loopback
      interface"
ifconfig lo 127.0.0.1

echo "Starting inetd"
xinetd

# start a shell
busybox sh
```

# BusyBox Summary

- BusyBox is a powerful tool for embedded systems that replaces many common Linux utilities in a single multical binary.
- BusyBox can significantly reduce the size of your root file system image.
- Configuring BusyBox is straightforward, using an interface similar to that used for Linux configuration.
- System initialization is possible but somewhat different with BusyBox

# Outline

- Background
  - Legal issues, versioning, building, user basics, FHS, booting
- Embedded Linux (common but not required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support

# Flash storage devices

- Significant restrictions on writing
  - data can be changed from a 1 to a 0 with writes to the cell's address
  - 0 to 1 requires an entire block be erased.
- Therefore, to modify data stored in a Flash memory, the block in which the modified data resides must be completely erased.
  - Write times for updating data in Flash memory can be many times that of a hard drive.
- Also very limited write cycles (100 to 1,000,000 or so) before wear out.
  - Wear leveling, conservative specifications generally make things okay.

# Outline

- Background
  - Legal issues, versioning, building, user basics, FHS, booting
- Embedded Linux (common but not required attributes)
  - Small footprint (BusyBox)
  - Flash file system
  - Real time support
    - RT Linux patch
    - Other solutions

# Real-time Kernel Patch

- The patch had many contributors, and it is currently maintained by Ingo Molnar; you can find it at:
  - [www.kernel.org/pub/linux/kernel/projects/rt/](http://www.kernel.org/pub/linux/kernel/projects/rt/)
  - <https://ubuntu.com/blog/real-time-kernel-technical>
- Since about Linux 2.6.12, soft real-time performance in the single-digit milliseconds on a reasonably fast x86 processor is readily achieved
- Some claim “nearly-worst-case” latency of 30us!

# Features

- The real-time patch adds a fourth preemption mode called `PREEMPT_RT`, or Preempt Real Time.
  - Features from the real-time patch are added, including replacing spinlocks with preemptable mutexes.
    - This enables involuntary preemption everywhere within the kernel except for areas protected by `preempt_disable()`.
    - This mode significantly smoothes out the variation in latency (jitter) and allows a low and predictable latency for time-critical real-time applications.
- The problem is...
  - Not all devices can handle being interrupted.
  - This means that with the RT patch in play you might get random crashes.

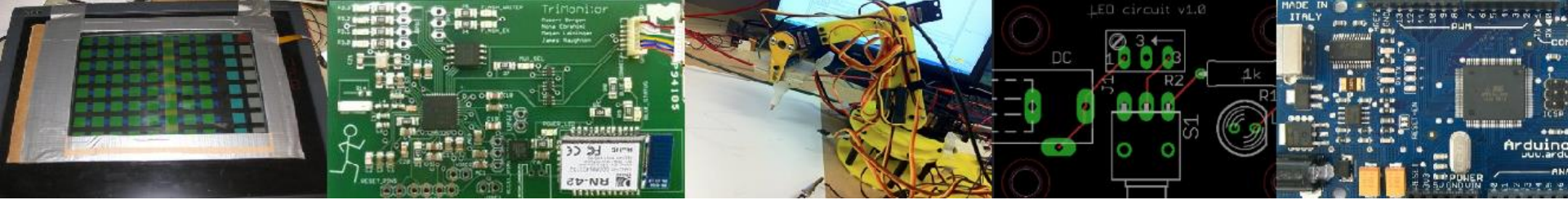
# There are lots of other attempts and discussions about a real-time Linux

- RTLinux
  - Wind River had something up and running for years.
    - Ended support in 2011.
    - Seems fairly restrictive.
- Real Time Linux Foundation, Inc.
  - Holding workshops on this for 13 years.
- <http://lwn.net/Articles/397422/>
  - Nice overview of some of the issues
- Zephyr is an RTOS that is developed as part of the Linux Foundation.
  - Looks like a traditional RTOS.
  - Much more stuff (closer to a traditional OS)



# Windows for IoT

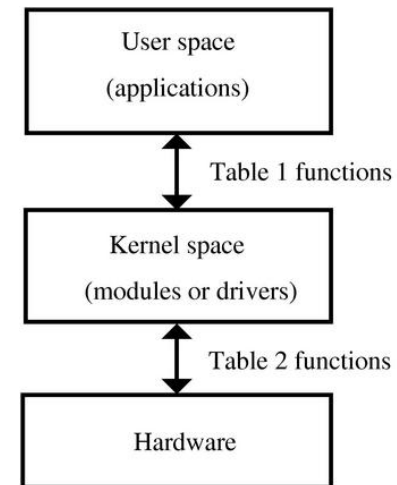
- A stripped down version of Windows with a focus on IoT issues.
  - Appears to be fairly popular.
  - Latest version appears to require 4 GB of RAM and 64 GB of storage.
    - Stripped-down version *may* get to 256MB RAM? It's a bit unclear what you can really get to, but 4 GB is the official number.
- Multiple versions all supported by Visual Studio.
  - Reasonable choice for things that are plugged in.
  - Remember, the best engineering solution isn't always the best solution
    - What I mean is that the “best” solutions require a lot of engineering time and therefore \$\$\$\$.
    - Might be best to use this (programmers familiar with the environment, lots of stuff done for you) to reduce total cost even if the parts are more expensive.



# EECS 473

## Advanced Embedded Systems

Linux device drivers and  
loadable kernel modules



# Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

A fair bit of this presentation, including some figures, comes from

[http://www.freesoftwaremagazine.com/articles/drivers\\_linux#](http://www.freesoftwaremagazine.com/articles/drivers_linux#)

Other sources noted at the end of the presentation.

# Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

# Device driver

(Thanks Wikipedia!)

- A device driver is a computer program allowing higher-level computer programs to interact with a hardware device.
  - A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects.
  - When a calling program invokes a routine in the driver, the driver issues commands to the device.
  - Drivers are hardware-dependent and operating-system-specific.

# Devices in Linux (1/2)

- There are special files called “device files” in Linux.
  - A user can interact with it much like a normal file.
  - But they *generally* provide access to a physical device.
  - They are generally found in `/dev` and `/sys`
    - `/dev/fb` is the frame buffer
    - `/dev/ttyS0` is one of the serial ports
- Not all device files correspond to physical devices.
  - Pseudo-devices.
    - Provide various functions to the programmer
    - `/dev/null`
      - Accepts and discards all input; produces no output.
    - `/dev/zero`
      - Produces a continuous stream of NULL (zero value) bytes.
    - etc.

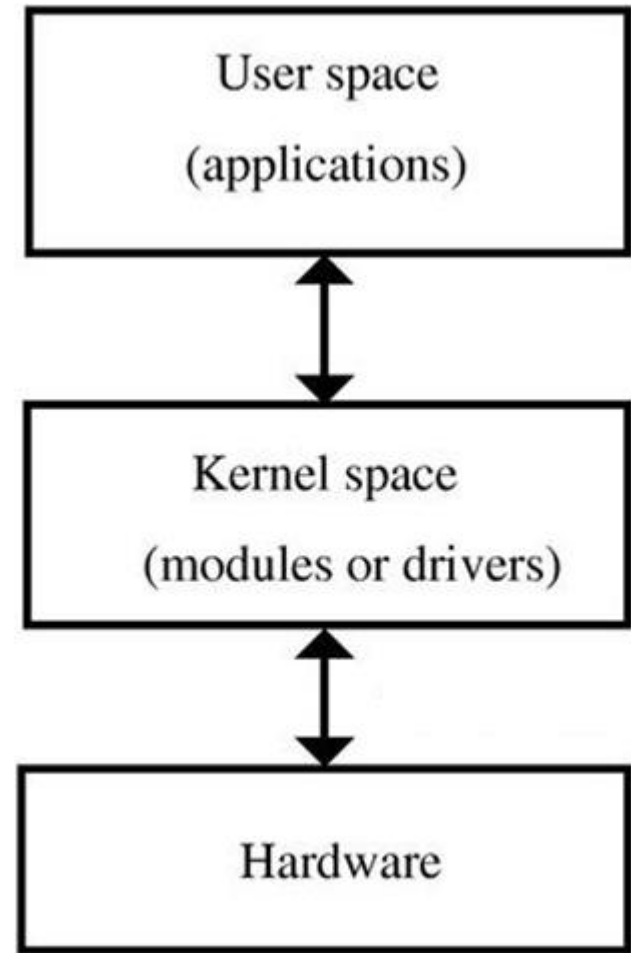
```
crw-rw---- 1 root dialout 4, 64 Jun 20 13:01 ttyS0
```

# Devices in Linux (2/2)

- Pretty clearly you need a way to connect the device file to the actual device
  - Or pseudo device for that matter
- We want to be able to “fake” this by writing functions that handle the file I/O.
  - So we need to associate functions with all the things we can do with a file.
    - Open, close.
    - Read, write.
- Today we’ll talk about all that...

# Kernel vs. User space

- User Space
  - End-user programs. They use the kernel to interface to the hardware.
- Kernel Space
  - Provides a standard (and hopefully multi-user secure) method of using and sharing the hardware.
    - Private function member might be a good analogy.
  - A lot of things are different here.
    - Many calls to the kernel can't be made from the kernel.
      - E.g. malloc.





# Linux Device Drivers...

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

# Kernel and Kernel Modules

- Often, if you want to add something to the kernel you need to rebuild the kernel and reboot.
  - A “loadable kernel module” (LKM) is an object file that extends the base kernel.
  - Exist in most OSes
    - Including Windows, FreeBSD, Mac OS X, etc.
  - Modules get added and removed as needed
    - To save memory, add functionality, etc.

# Linux Kernel Modules

- In general must be licensed under a free license.
  - Doing otherwise will taint the whole kernel.
    - A tainted kernel sees little support.
    - Might be a copyright problem if you redistribute.
- The Linux kernel changes pretty rapidly, including APIs etc.
  - This can make it a real chore to keep LKMs up to date.
  - Also makes a tutorial a bit of a pain.
    - Though honestly it seems fairly stable over the last 5 years.

# Creating a module

- All modules need to define functions that are to be run when:
  - The module is loaded into the kernel
  - The module is removed from the kernel
- We just write C code (see next slide)
- We need to compile it as a kernel module.
  - We invoke the kernel's makefile.
  - `sudo make -C /lib/modules/xxx/build M=$PWD modules`
    - This makes (as root) using the makefile in the path specified.
    - I think it makes all C files in the directory you started in
    - Creates .ko (rather than .o) file
    - Xxx is some kernel version/directory

# Simple module

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- `MODULE_LICENSE`
  - Required.
  - Short list of allowed licenses.
- `Printk()`
  - Kernel print.
    - Prints message to console and to log.
    - <1> indicates high priority message, so it gets logged.
- `Module_init()`
  - Tells system what module to call when we first load the module.
- `Module_exit()`
  - Same but called when module released.

# Modules:

## Listing, loading and removing

- From the command line:
  - lsmod
    - List modules.
  - insmod
    - Insert module into kernel
      - Adds to list of available modules
    - Causes function specified by `module_init()` to be called.
  - rmmod
    - Removes module from kernel

# lsmod

Module	Size	Used by
memory	10888	0
hello	9600	0
binfmt_misc	18572	1
bridge	63776	0
stp	11140	1 bridge
bnep	22912	2
video	29844	0

# insmod

- Very (very) simple
  - **insmod xxxxx.ko**
    - Says to insert the module into the kernel



# Other (better) way to load a module

- **Modprobe** is a smarter version of insmod.
  - Actually it's a smarter version of insmod, lsmod and rmmod...
    - It can use short names/aliases for modules
    - It will first install any dependent modules
- We'll use insmod for the most part
  - But be aware of modprobe

# So?

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- When insmod, log file gets a “Hello World!”
- When rmmod, that message prints to log (and console...)
- It’s not the name, it’s the module\_init().

# Modules?

- There are a number of different reasons one might have a module
  - But the main one is to create a device driver
  - It's not realistic for Linux to have a device driver for all possible hardware in memory all at once.
    - Would be too much code, requiring too much memory.
  - So we have devices as modules
    - Loaded as needed.

# What is a “device”?

- As mentioned in the overview, Linux devices are accessed from user space in exactly the same way files are accessed.
  - They are generally found in /dev and /sys
- To link normal files with a kernel module, each device has a “major number”
  - Each device also has a “minor number” which can be used by the device to distinguish what job it is doing.

```
% ls -l /dev/fd0 /dev/fd0u1680  
brwxrwxrwx    1 root   floppy    2,   0 Jul  5  2000 /dev/fd0  
brw-rw----    1 root   floppy    2,  44 Jul  5  2000 /dev/fd0u1680
```

Two floppy devices. They are actually both the same bit of hardware using the same driver (major number is 2), but one is 1.68MB the other 1.44.

# Creating a device

- **`mknod /dev/memory c 60 0`**
  - Creates a character device named `/dev/memory`
  - Major number 60
  - Minor number 0
- Minor numbers are passed to the driver to distinguish different hardware with the same driver.
  - Or, potentially, the same hardware with different parameters (as the floppy example)

# Linux Device Drivers

- Overview
  - What is a device driver?
    - Linux devices
  - User space vs. Kernel space
- Modules and talking to the kernel
  - Background
  - Example
  - Some thinky stuff

# A complete pseudo-device

- We are going to create a pseudo-device that is just a single byte of memory.
  - Whatever the last thing you wrote to it, is what will be read.
- For example
  - **echo -n abcdef >/dev/memory**
  - Followed by **cat /dev/memory**
    - Prints an “f”.
- Silly, but not unreasonable.
  - It’s also printing some stuff to the log.
    - Not a great idea in a real device, but handy here.

# includes

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>    /* printk() */
#include <linux/slab.h>      /* kmalloc() */
#include <linux/fs.h>        /* everything... */
#include <linux/errno.h>     /* error codes */
#include <linux/types.h>     /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h>     /* O_ACCMODE */
#include <asm/system.h>      /* cli(), *_flags */
#include <asm/uaccess.h>     /* copy_from/to_user */
```



# License and function prototypes

```
MODULE_LICENSE("Dual BSD/GPL");
```

```
int memory_open      (struct inode *inode, struct file *filp);
```

```
int memory_release (struct inode *inode, struct file *filp);
```

```
ssize_t memory_read (struct file *filp, char *buf, size_t count,  
                      loff_t *f_pos);
```

```
ssize_t memory_write (struct file *filp, char *buf,  
                      size_t count , loff_t *f_pos);
```

```
void memory_exit (void);
```

```
int memory_init (void);
```

# Setting up the standard interface

```
struct file_operations
memory_fops = {
    read:      memory_read,
    write:     memory_write,
    open:      memory_open,
    release:   memory_release
};

struct file_operations
memory_fops = {
    .read      = memory_read,
    .write     = memory_write,
    .open      = memory_open,
    .release   = memory_release
};
```

- This is a weird bit of C syntax.
  - Initializes struct elements.
    - So “read” member is now “memory\_read”
  - Technically unsupported these days?
    - gcc supports it though
  - dot notation is in the C99 standard.
    - But some kernel code still uses colon.

# file\_operations struct

```
struct file_operations {
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t(*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t(*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
    ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);

    unsigned long (*get_unmapped_area) (struct file *, unsigned long,
    unsigned long, unsigned long, unsigned long);
};
```

# file\_operations:

## A few members

```
struct file_operations {  
    ssize_t(*read)    (struct file *, char __user *,  
                        size_t, loff_t *);  
    ssize_t(*write)   (struct file *, const char __user *,  
                        size_t, loff_t *);  
    int (*ioctl)       (struct inode *, struct file *,  
                        unsigned int, unsigned long);  
    int (*open)        (struct inode *, struct file *);  
    int (*release)     (struct inode *, struct file *);  
};
```

# Set up init and exit

## Some globals

```
module_init(memory_init);  
module_exit(memory_exit);
```

```
int memory_major = 60;  
char *memory_buffer;
```

# memory\_init

```
int memory_init(void) {
    int result;
    result = register_chrdev(memory_major, "memory", &memory_fops);
    if (result < 0) {
        printk("<1>memory: cannot obtain major number %d\n",
               memory_major);
        return result;
    }

    /* Allocating memory for the buffer */
    memory_buffer = kmalloc (1, GFP_KERNEL);
    if (!memory_buffer) {
        result = -ENOMEM;
        goto fail;
    }

    memset(memory_buffer, 0, 1); // initialize 1 byte with 0s.
    printk("<1> Inserting memory module\n");
    return 0;

fail:
    memory_exit();
    return result;
}
```

60, via global

Device name, need not be the same as in /dev

Name of file\_operations structure.

Kmalloc does what you'd expect. The flag provides rules about where and how to get the memory. See [makelinux.com/ldd3/chp-8-sect-1](http://makelinux.com/ldd3/chp-8-sect-1)

# memory\_exit

```
void memory_exit(void) {  
  
    unregister_chrdev(memory_major, "memory");  
    if (memory_buffer) {  
        kfree(memory_buffer);  
    }  
}
```

# Open and release (close)

```
int memory_open (struct inode *inode,  
                  struct file *filp) {  
    printk("<1> Minor: %d\n",  
           MINOR(inode->i_rdev));  
    return 0;  
}
```

```
int memory_release (struct inode *inode,  
                     struct file *filp) {  
    return 0;  
}
```



## Modules: single character memory example

```
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos) {

    /* Transferring data to user space */
    copy_to_user (buf, memory_buffer, 1);

    /* Changing reading position as best suits */
    if (*f_pos == 0) {
        *f_pos += 1;
        return 1;
    } else {
        return 0;
    }
}
```

**f\_pos** is the file position.  
What do you think happens  
if you don't change \*f\_pos?

**copy\_to\_user** copies to a location in userspace (the first argument) from kernel space (the second argument), a specific number of bytes. Recall virtual memory...

# memory\_write

```
ssize_t memory_write( struct file *filp,  
    char *buf, size_t count, loff_t *f_pos)  
{  
    char *tmp;  
  
    tmp=buf+count-1;  
    copy_from_user(memory_buffer,tmp,1);  
    return 1;  
}
```

# How do you set it up?

- Make the module

```
make -C /lib/modules/2.6.28-16-  
generic/build M=$PWD modules
```

- Insert the module

```
insmod memory.ko
```

- Create the device

```
mknod /dev/memory c 60 0
```

- Make the device read/write

```
chmod 666 /dev/memory
```

# What did all that do?

- We created a device that is just a single byte of memory.
  - Whatever the last thing you wrote to it, is what will be read.
- For example
  - `$ echo -n abcdef >/dev/memory`
  - Followed by `$ cat /dev/memory`
    - Prints an “f”.

# See also

- <https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os> looks well done.
  - For 5.0
  - I've not double-checked it all