### Linux Kernel

Kernel is the heart of an operating system – controls disk, memory, task management etc. Has own space and not accessible by other programs.

### Porting

Referes to two things: adapting linux architecture to the CPU e.g. could be ARM, PC Power, MIPs. In addition, SOC porting can also be considered as part of the architecture porting.

Board porting – custom drivers and installation code for devices specific to a board.

### Why drivers?

Access GPIO pin, type and output. Can do this in kernel mode can access the address 0x3F20000. But in user mode cannot access this address which means you are using unprivileged

Hardware goes through a hardware abstraction layer to the kernel which is privileged memory and code which then gooes through a memory boundary to the user space (unprivileged memory and code). So crossing the memory boundary and the hardware abstraction layer, code may work for your processor but transferring to new kernel or OS it may not work – which you need drivers to do. These make tunnels – an interface to legally cross the boundaries.

Set up build environment and write code like its ran in kernel mode. Which means can add features to it to access GPIO interface

### Components needed for when writing drivers

Raspberry-pi-kernel-headers – when build code for the kernel need the headers for the kernel you are working on. Provides proper interface into the kernel that will work on version of the OS. Code will access kernel in inproper way if that isn’t the case. When installs this you run make and it will build the code for you properly

To check it works:

uname -r

5.10.63-v7l+

**pi@raspberrypi**:**~/gpio-driver $** ls /lib/modules/$(uname -r)/headers

ls: cannot access '/lib/modules/5.10.63-v7l+/headers': No such file or directory

**pi@raspberrypi**:**~/gpio-driver $** ls /lib/modules/$(uname -r)/build

**arch** **Documentation** **init** **lib** **net** **sound**

**block** **drivers** **ipc** Makefile **samples** **tools**

**certs** **fs** Kconfig **mm** **scripts** **usr**

**crypto** **include** **kernel** Module.symvers **security** **virt**

Check the build package

Create a make file that describes how you create the kernel driver.

So after making all those we just type Make and then we go from there

Then it needs installing – insmod, lsmond to check, then install using rmod

Do lsmod to check it

This guy is the goat- do some more bare metal and driver stuff for fun?

<https://www.youtube.com/c/LowLevelLearning/videos>

## The ProcFS File System

Make a driver for RPi that has access to peripherals of GPIO interface – provide custom functionality in the kernel.

Sp previous kernel could be used in kernel space but not in user space

We were trying to access 0x3F200000 address – address of GPIO interface in physical memory. Trying to access this from user space we illegally cross the memory boundary (user space to kernel space) and memory boundary from kernel space tp physical hardware. So to cross these boundaries we develop these drivers.

So looking at our previous file – which just switches on and off when a driver is installed – this is the kernel space to hardware file. Now to cross the boundary in user space we need to do the following.

So how does the user talk to the driver:

Dev files – live in /dev – look like a file but they are not – they are a special file that maps into the kernel and then provides read write etc functionality into the kernel that can be used by the user space.

They are meant to map to a single physical device – map to the device drivers.

Procfs File – more generically used to provide info about the kernel to the user and then act as a more generic control interface for generic softwares.

So we are going to make a procfs file in /proc that we can read and write from that will provide output and input to the kernel driver and control the hardware interface.

So e.g. cat /proc/net/tcp – call goes into the kernel and then parses all the connections that are running to TCP – feels like a file

So we are going to make a ProcFS entry that will take data and parse it – eventually will control GPIO interface via that.

So we need to edit lll-gpio low level driver file to give it a procfs entry point and allow kernel to read/write from it.

Global object defined called proc\_dir\_entry that is used to track the dir entry that comes out when we approach the kernel.

It is a pointer and is placed in memory.

So we have made the gpio message buffer and interface to the driver

**pi@raspberrypi**:**~/gpio-driver $** nano lll-gpio-driver.c

**pi@raspberrypi**:**~/gpio-driver $** sudo lsmod | grep lll

**pi@raspberrypi**:**~/gpio-driver $** sudo insmod lll-gpio-driver.ko

**pi@raspberrypi**:**~/gpio-driver $** sudo lsmod | grep lll

**lll**\_gpio\_driver 16384 0

**pi@raspberrypi**:**~/gpio-driver $** ls /proc

So as you can see we can instatiate it and when you go to look at the procfs you can see it will load up the driver – you can write to this and then read from it

### Stage 3 – flipping bits on and off – i.e. controlling the GPIO

Page is a block of stored memory

So here we have to first get the address from the virtual to the physical and map it using ioremap()

A page size is one block of memory.

Registers - number of bits a register holds that are called a bit – used to hold and information written to it based on the next instruction to be executed or decoded. So the function select register is the reigsters that stores the next instruction or function for a particular GPIO

So now you can make regular old c code that compiles and can read to the driver as a file and then write to it.

So we do the following:

**pi@raspberrypi**:**~ $** cd userProgramTest/

**pi@raspberrypi**:**~/userProgramTest $** nano onOff21.c

**pi@raspberrypi**:**~/userProgramTest $** gcc onOff21.c -o onOff21

**pi@raspberrypi**:**~/userProgramTest $** ls

**onOff21** onOff21.c

**pi@raspberrypi**:**~/userProgramTest $** ./onOff21

^C

**pi@raspberrypi**:**~/userProgramTest $**

gcc is the compiler – same as visual studio used to compile c code