EMBEDDED DEVICE DRIVERS

Linux Device Drivers on Beaglebone Black

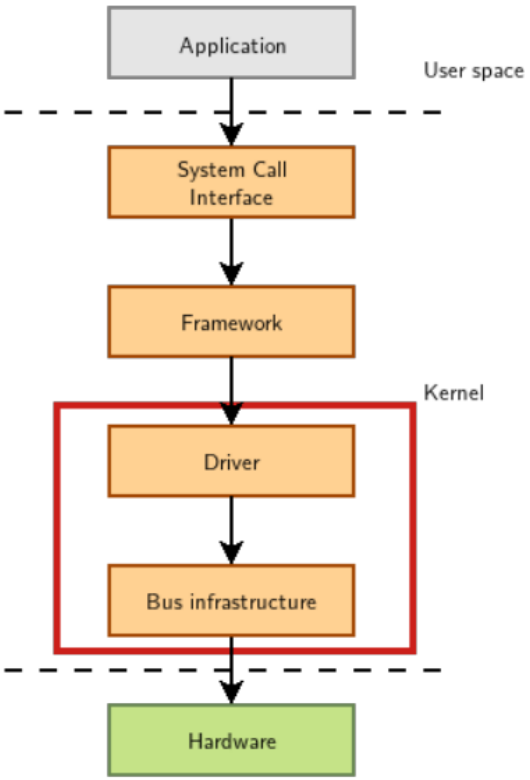
Linux device model

- Linux kernel needs to handle various device connect scenarios
  - The same device needs to use the same driver
  - On multiple CPU architectures
  - But the controllers may be different

  - A single driver
  - Needs to support multiple devices
  - Of the same kind
- This necessitates a separation
  - Controller drivers
    - Control behavior of controllers for complex protocol busses
  - Device drivers
    - Control behavior of specific device instantiations on the bus

Linux device model illustrated

- In Linux, a (device) driver talks to
  - A framework
    - That allows the driver to expose
    - Hardware in a generic manner
  - A bus infrastructure
    - To detect and communicate
    - With the actual hardware



Device model data structures

- The device model is organized around:
  - ***struct bus_type***
    - Represents a bus (USB, I2C, SPI, etc.)
  - ***struct device_driver***
    - Represents a driver capable of handling
      - Certain devices on a certain bus
  - ***struct device***
    - Represents one device connected to a bus

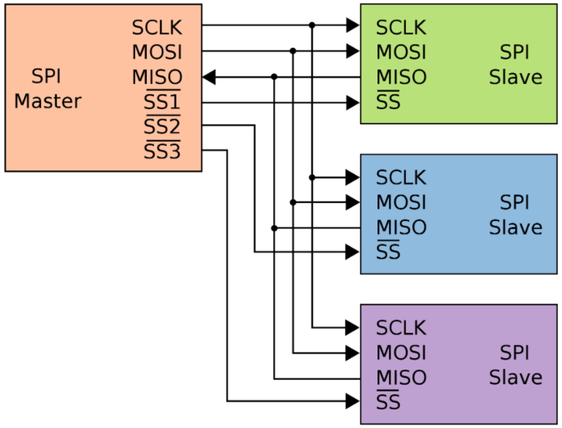
- The kernel uses inheritance
  - To create more specialized versions
    - For each bus subsystem

SPI: What?

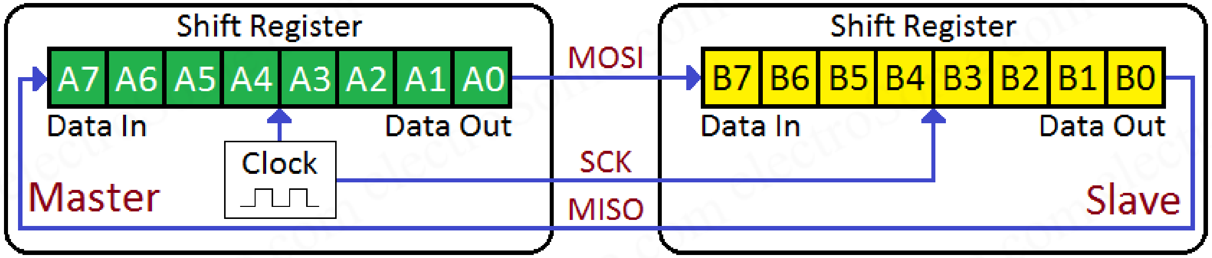
- Serial protocol
  - Roles: Master, Slave
  - 4-wires – SCLK (clock), SS (slave select), MOSI, MISO
    - MOSI: Master Out, Slave In
    - MISO: Master In, Slave Out
    - Synchronous, master-driven clock
  - Multi-drop, single-master
  - Full-duplex protocol
    - Involves shift-register data movement
  - Addressing
    - Masters are address-less
    - Slave address through **active-low** chip select (CS)
      - 1 per slave – leads to complex designs for many slaves
  - Speeds
    - Range from 500kHz to several MHz
  - Connects mid-speed devices like LCD displays, DACs, etc.

SPI: Topology and data transfer

- 1 master, n slave configuration ***(here, n=3)***
  - Master has to
    - Activate each ***SS$_n$*** line
    - To communicate to each slave



- Data transfers always duplex
  - Involves shift registers at each end



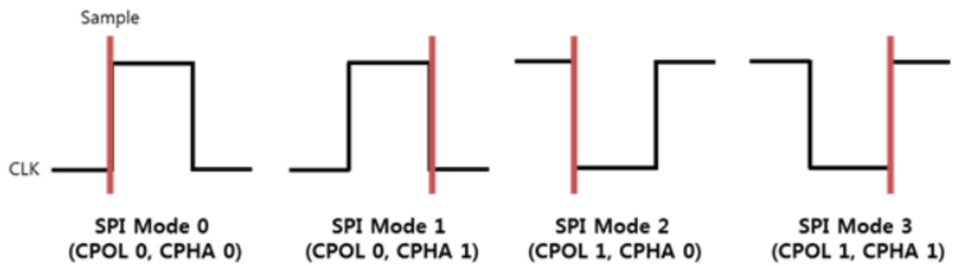
SPI: Modes of data transfer

- Multiple possibilities of data bit sampling
  - w.r.t. SCLK polarity and edge (rising/falling)

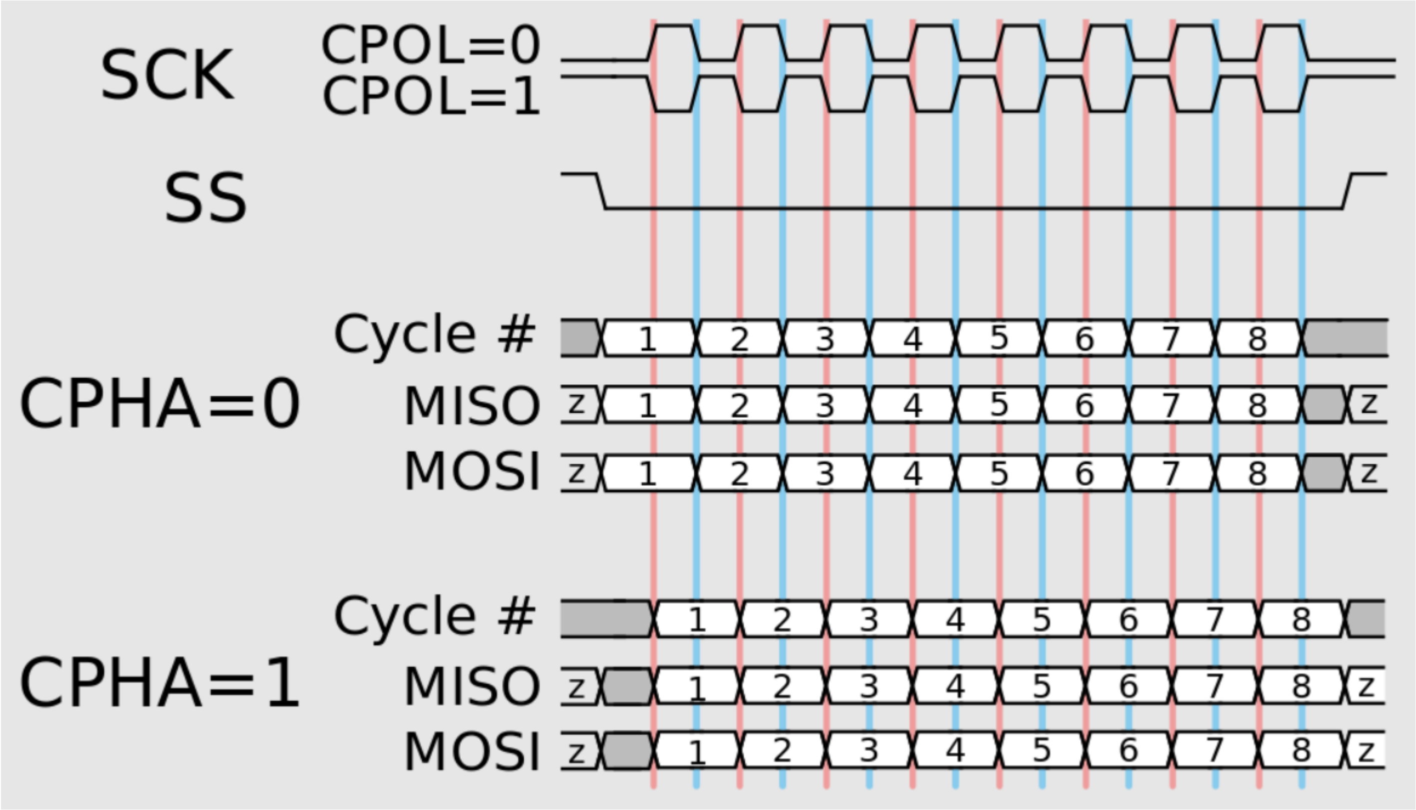
SPI Modes

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

  - Clock polarity denoted by CPOL
  - Sampling edge denoted by CPHA



SPI: Modal waveforms



LKM: SPI subsystem

- SPI subsystem on Linux
  - SPI Core
    - Provides APIs for core data structures
    - Registration, cancellation and management
    - Hardware-platform independent layer
    - In ***<kernel>/drivers/spi/spi.c***

  - SPI Controller Driver
    - Platform specific driver, controlling several slave devices
    - Supports both master and slave roles
    - Control the hardware registers
    - May use DMA / GPIO for data transfers

  - SPI Protocol / Device Driver
    - Used to communicate to specific slave devices over the SPI bus

LKM: SPI device driver dev (1/2)

- Get the SPI controller driver

  *struct spi_controller **__*spi_busnum_to_master*__*(u16 spi_busnum);*

- Add the slave device to the SPI controller
  - Create ***spi_board_info***

  *struct spi_device **__*spi_new_device*__*(struct spi_controller *ctrlr, struct spi_board_info *info);*

- Configure the SPI device

  int ***spi_setup****(struct spi_device *dev);*

LKM: SPI device driver dev (2/2)

- Start data transfers between master and slave

  int ***spi_sync_transfer****(struct spi_device *dev, struct spi_transfer *xfer, unsigned int num_xfers);*
    - Blocking

  int ***spi_async****(struct spi_device *dev, struct spi_message *mesg);*
    - Async transfer

  int ***spi_write_then_read****(struct spi_device *dev, const void *txbuf, unsigned n_tx, void *rxbuf, unsigned n_rx);*
    - Write then read, synchronous

- Remove the slave device on exit

  void ***spi_unregister_device****(struct spi_device *dev);*

LKM: SPI driver exercise (1/2)

- Refer ***mod15*** directory
  - ***spidev_test.c*** contains user-space driver code for testing an SPI loopback
    - Linux mounts the SPI1 Bus as ***/dev/spidev1.x***
    - We open this device, and set its properties via IOCTLs
      - Write/read modes, write/read bits per word
      - Write-read max speeds
      - SPI mode for data transfer
    - We short / connect SPI1 out and in pins
      - **P9_29 (MOSI) / D0** and **P9_30 (MISO) / D1**
    - We then transfer ***tx_buf*** and receive ***rx_buf***

  - Compile ***spidev_test.c*** and transfer ***a.out*** to BBB
    - Run the executable

```
root@BeagleBone:/home/debian# ./a.out -D /dev/spidev1.0
spi mode: 0
bits per word: 8
max speed: 500000 Hz (500 KHz)

FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
root@BeagleBone:/home/debian#
```

LKM: SPI driver exercise (2/2)

- On BBB, HDMI pins muxed with SPI1
  - So HDMI has to be disabled by disabling the dtb0
    - In **/boot/uEnv.txt**

```
disable_uboot_overlay_video=1
disable_uboot_overlay_audio=1
```

- Also, SPI1 needs to be enabled specifically
  - Via calling out its dtb0 and u-boot cmdline
    - In **/boot/uEnv.txt**

```
###Custom Cape
dtb_overlay=BB-SPIDEV1-00A0.dtbo

console=ttyS0,115200n8
cmdline=coherent_pool=1M net.ifnames=0 lpj=1990656 rng_core.default_quality=100
capemgr.enable_partno=BB-SPIDEV1-01
```

```
uboot_overlays: [fdt_buffer=0x60000] ...
uboot_overlays: loading /boot/dtbs/5.10.168-ti-r72/overlays/BB-ADC-00A0.dtbo ...
645 bytes read in 20 ms (31.3 KiB/s)
uboot_overlays: loading /boot/dtbs/5.10.168-ti-r72/overlays/BB-BONE-eMMC1-01-00A0.dtbo ...
1605 bytes read in 20 ms (78.1 KiB/s)
uboot_overlays: uboot loading of [BB-HDMI-TDA998x-00A0.dtbo] disabled by /boot/uEnv.txt [disable_uboot
_overlay_video=1]...
uboot_overlays: [dtb_overlay=BB-SPIDEV1-00A0.dtbo] ...
uboot_overlays: loading /boot/dtbs/5.10.168-ti-r72/overlays/BB-SPIDEV1-00A0.dtbo ...
1522 bytes read in 18 ms (82 KiB/s)
```

SPI: Pros and Cons

- Pros:
  - Faster than UART and I2C due to full-duplex and hi-speed nature
  - No packetization, no overhead
  - No slave addressing mechanism in transaction
  - Not limited to 8-bit data transfers
  - No pull-up / pull-down resistors

- Cons
  - Expensive due to 4-wire requirement
  - No acknowledgement in protocol, software has to take care
  - No error detection/correction in base protocol
  - Master-slave should be tuned to each other
    - No clock stretching
  - Single master
  - Distances supported are shorter than I2C and UART

THANK YOU!