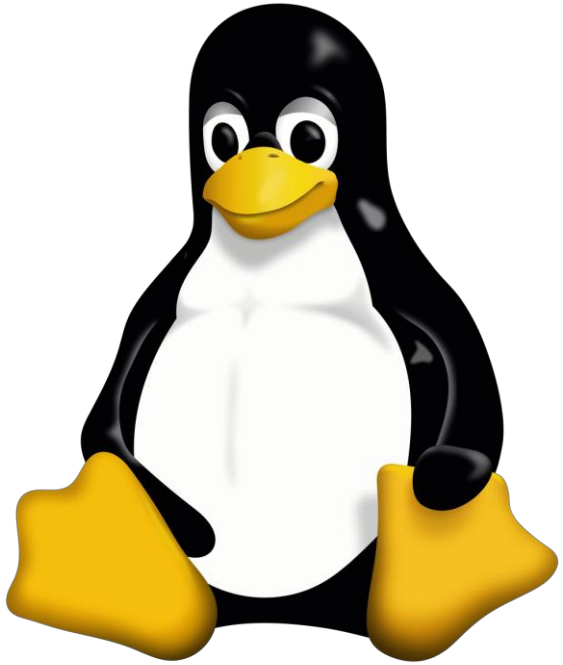
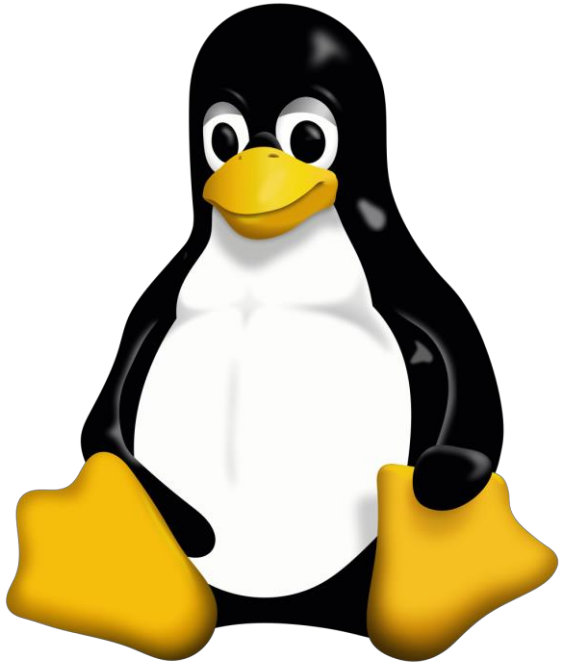


Linux Device Driver

Sunbeam Infotech



Linux Platform Bus

Sunbeam Infotech

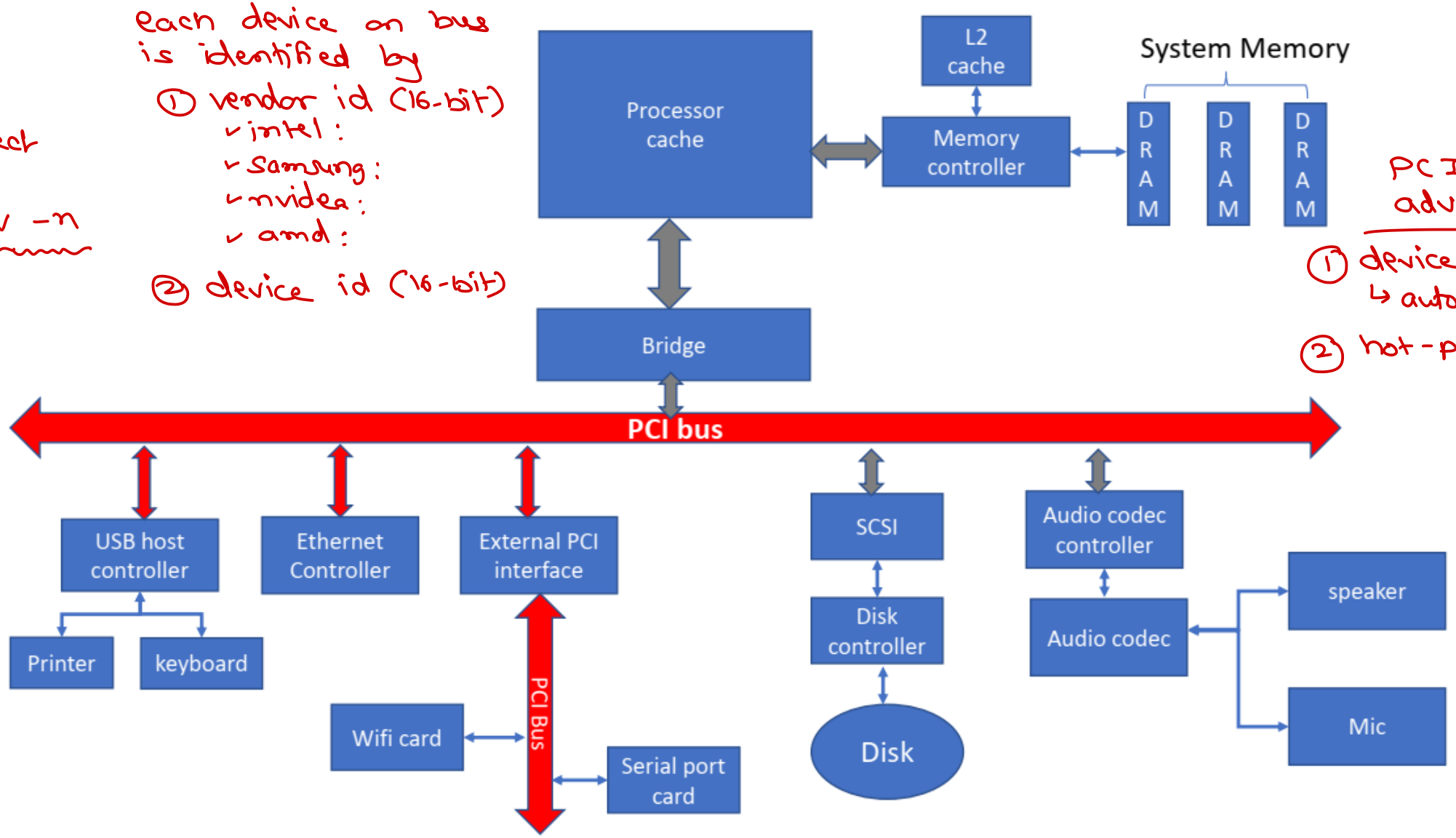
PCI bus – PC architecture

Peripheral Component Inter Connect

lspci -v -n

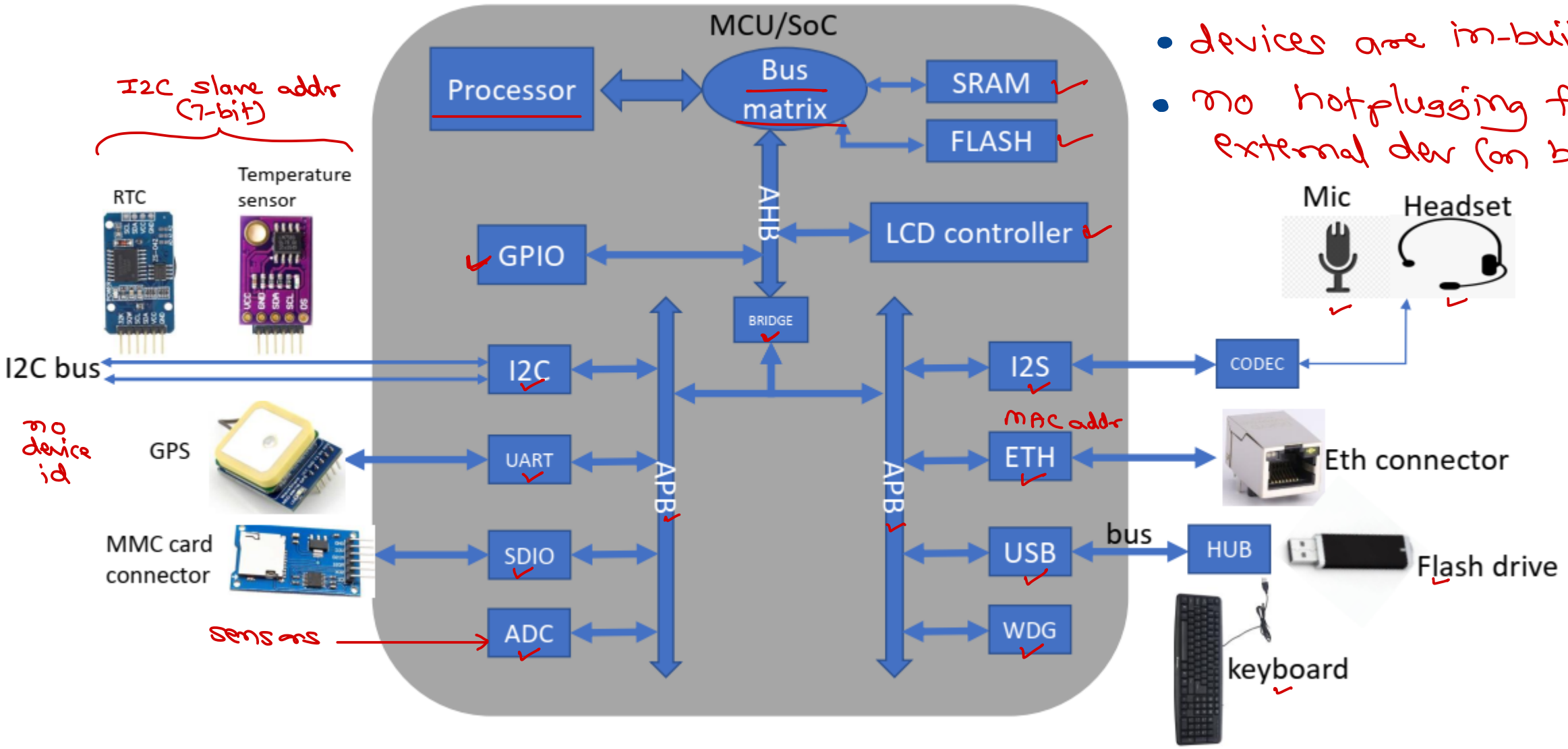
each device on bus is identified by
① vendor id (16-bit)
- intel :
- samsung :
- nvidea :
- amd :

② device id (16-bit)



PCI bus advantages
① device identification
↳ auto-discoverable
② hot-plugging

Embedded – SoC



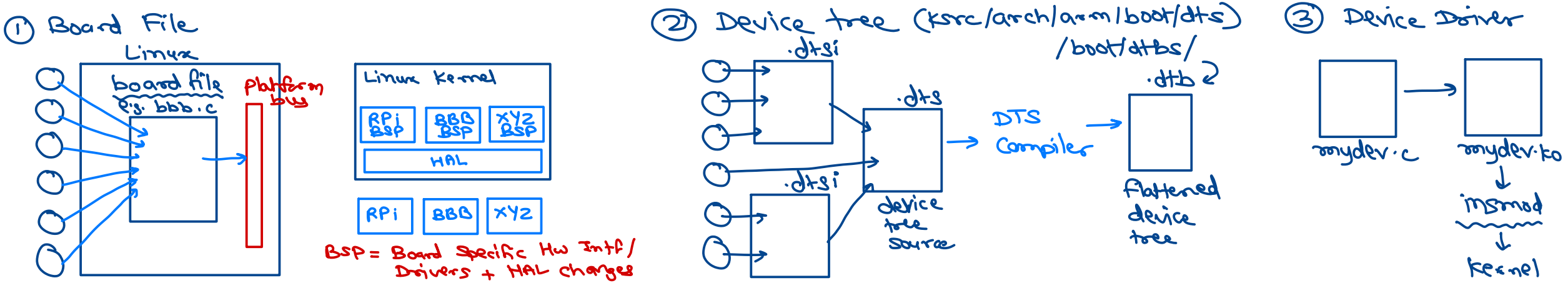
struct kobject

- Keeping track of various C struct objects is common need throughout the kernel.
- From Linux kernel 2.5 *struct kobject* is added for following functionalities.
- It provides following functionalities
  - Reference counting
  - Manage list of objects
  - Locking of sets
  - Exporting object properties to sysfs
- To avail these functionalities embed kobject into the desired struct.
- kobject functions: kobject_init(), kobject_get(), kobject_put(), kobject_add(), kobject_cleanup(), kobject_register(), kobject_unregister().

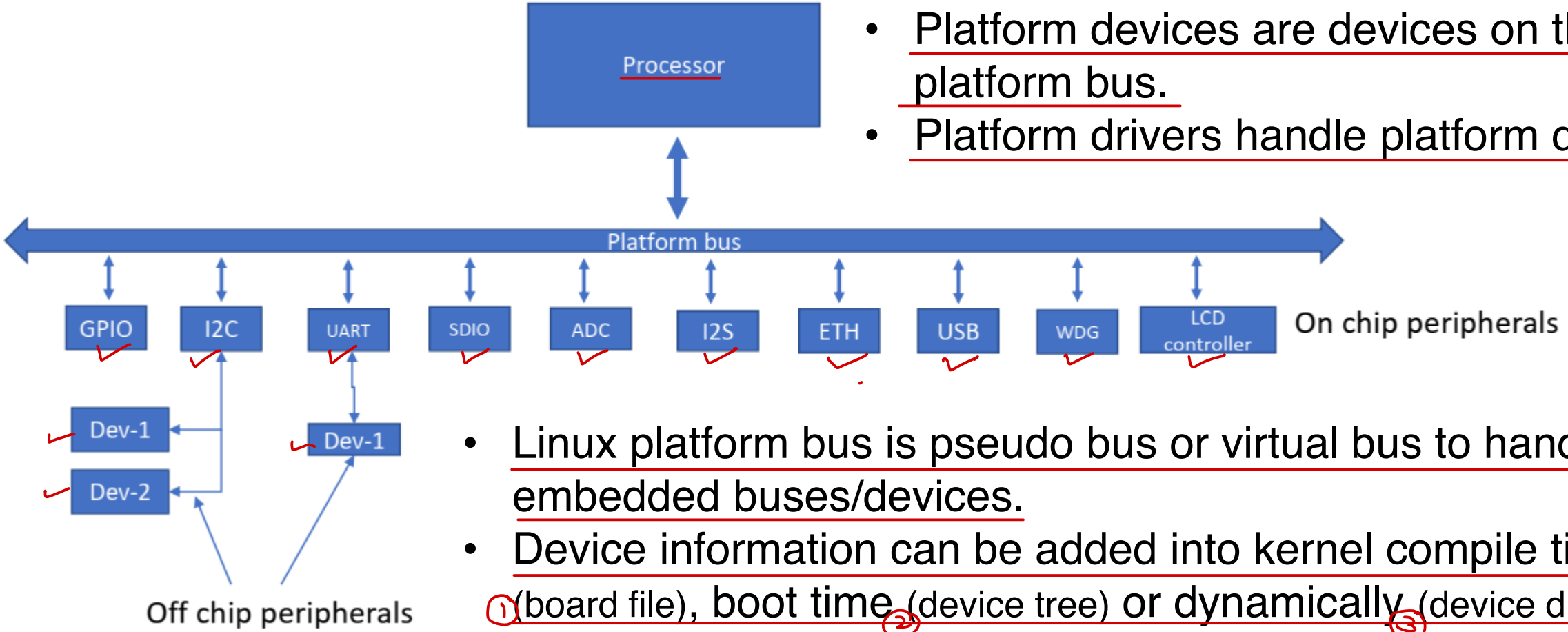
```
struct kobject {
    const char *k_name;
    struct kref kref;
    struct list_head entry;
    struct kobject *parent;
    struct kset *kset;
    struct kobj_type *ktype;
    struct sysfs_dirent *sd;
};
```

Platform bus, device and driver

- In Linux on PC architecture, most of the IO devices are connected over PCI and USB buses.
- PCI and USB buses are auto-discoverable (lspci, lsusb) and hot-pluggable (plug n play).
- Typical embedded Linux on ARM or other architecture do not have PCI bus.
- In embedded hardware (SoC) most of devices/buses are available on chip itself and are directly connected to CPU.
- Embedded buses like SPI, I2C, CAN, I2S are not discoverable/hot-pluggable.



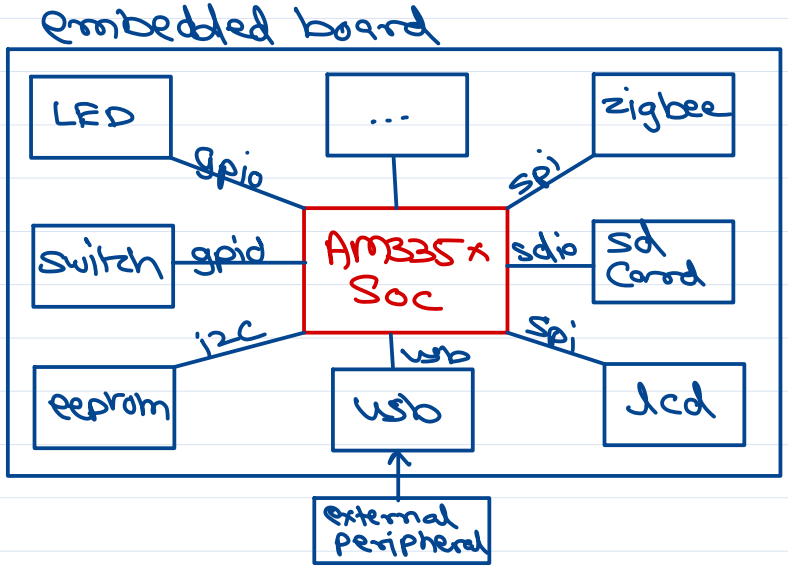
Platform bus



- Platform devices are devices on the platform bus.
- Platform drivers handle platform devices.

- Linux platform bus is pseudo bus or virtual bus to handle embedded buses/devices.
- Device information can be added into kernel compile time (board file), boot time (device tree) or dynamically (device driver).
  - Memory/IO address, IRQ number, Device Id, Device address, Pin configuration, Power/voltage information, etc.
- Device Tree: https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt

Device tree



Embedded Buses/Connectivity

1. USB
2. RS232
3. SPI
4. I2C
5. CAN

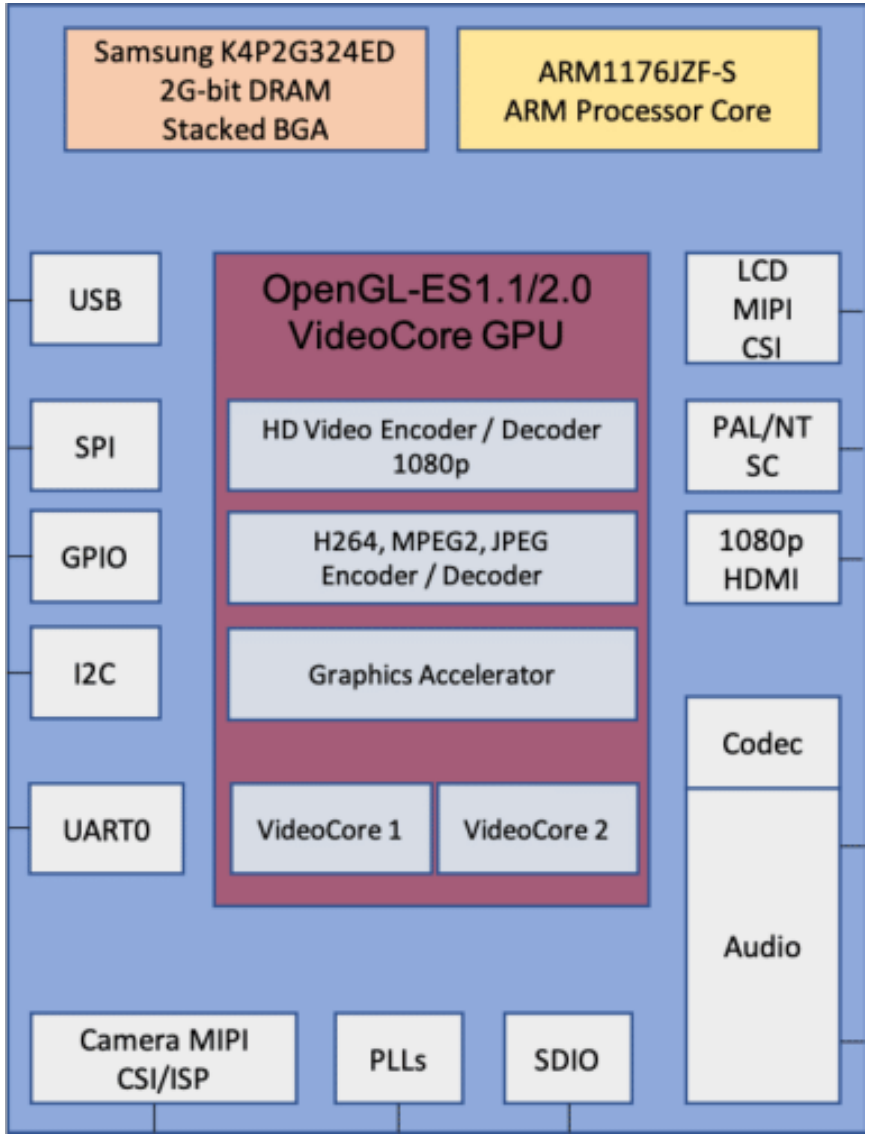
→ platform devices

- board file
  - ↳ board specific + probe drivers (recompile kernel for each board).
  - my_board_init()
    - ✓ add_device_serial()
    - ✓ add_device_spi()
    - ✓ add_device_eth()
    - ✓ add_device_i2c()
    - ✓ add_device_gpio()
  - + device drivers (.ko)
- device tree source
  - ↳ board specific describe data struct.
  - .dts → DTS Compiler → .dtb

Same kernel can be used to init diff boards with diff dtb files.

BCM2835 & AM335x

RPi



BBB →

ARM®
Cortex®-A8
Up to 1 Ghz

32KB and 32KB L1 + SED
256KB L2 + ECC
176KB ROM | 64KB RAM

Graphics
PowerVR
SGX
3D GFX

Crypto
64KB
shared
RAM

Display
24-bit LCD controller
Touch screen controller

PRU-ICSS
EtherCAT, PROFINET,
EtherNet/IP,
and more

L3 and L4 interconnect

Serial
UART x6
SPI x2
I²C x3
McASP x2
(4 channel)
CAN x2
(Ver. 2 A and B)
USB 2.0 HS
DRD + PHY x2

System
eDMA
Timers x8
WDT
RTC
eHRPWM x3
eQEP x3
PRCM

eCAP x3
ADC (8 channel)
12-bit SAR
JTAG
Crystal
Oscillator x2

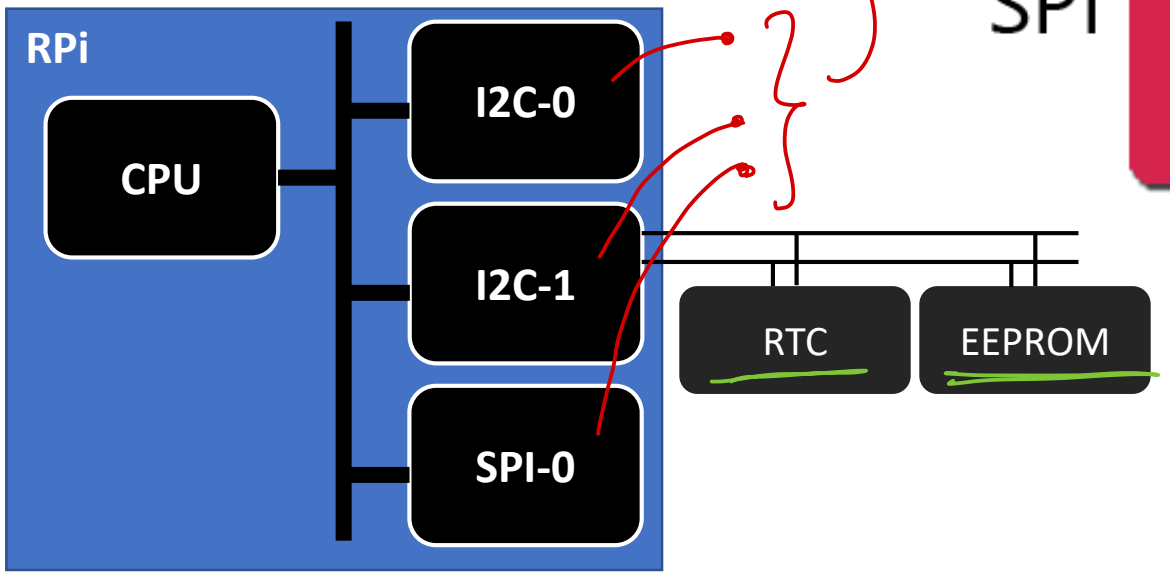
Parallel
MMC, SD and
SDIO x3
GPIO

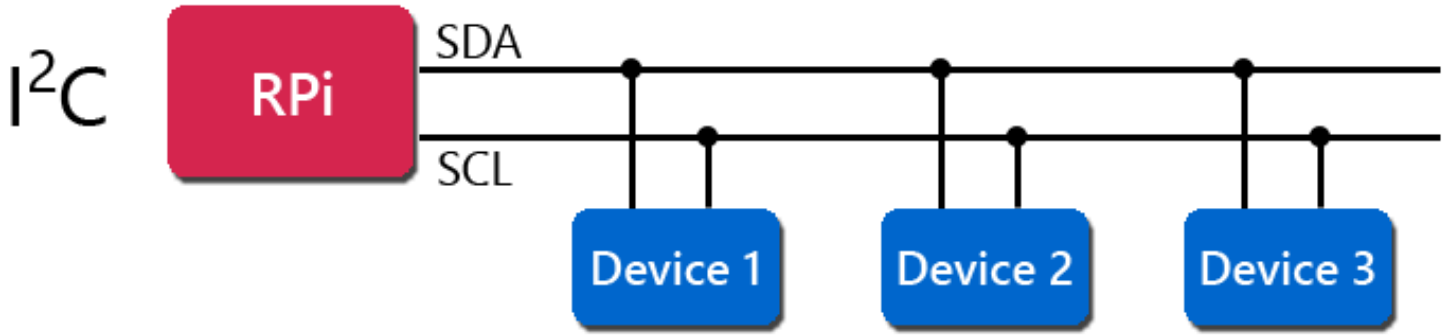
EMAC (2-port) 10M, 100M, 1G
IEEE 1588v2, and switch
(MII, RMII, RGMII)

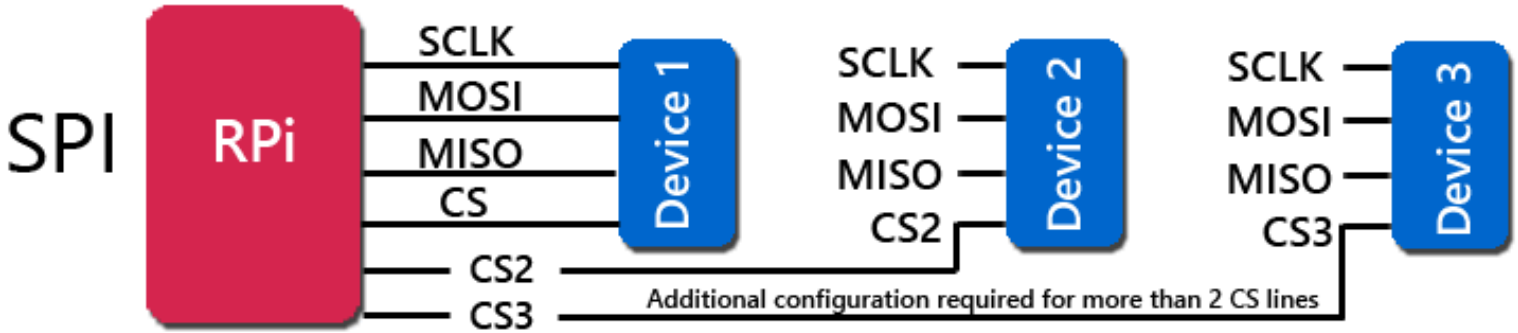
Memory interface
mDDR(LPDDR), DDR2,
DDR3, DDR3L
(16-bit; 200, 266, 400, 400 MHz)
NAND and NOR (16-bit ECC)

Platform devices and drivers

- Embedded devices (on-board and off-board) are platform devices.
- Respective vendors usually provide platform drivers host/bus controllers called as "Controller drivers".







- Devices on I2C bus are I2C client devices.
- RPi provides i2c host controller driver and client drivers implemented in user or kernel space.

Platform Driver

- https://www.kernel.org/doc/Documentation/driver-model/platform.txt

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
};
```

called when device attached or driver loaded (matching).

called when device detached or driver unloaded (matching).

to shutdown/power off the device.

when device is suspended (idle) and resume. During suspension few dev can be kept in low power state.

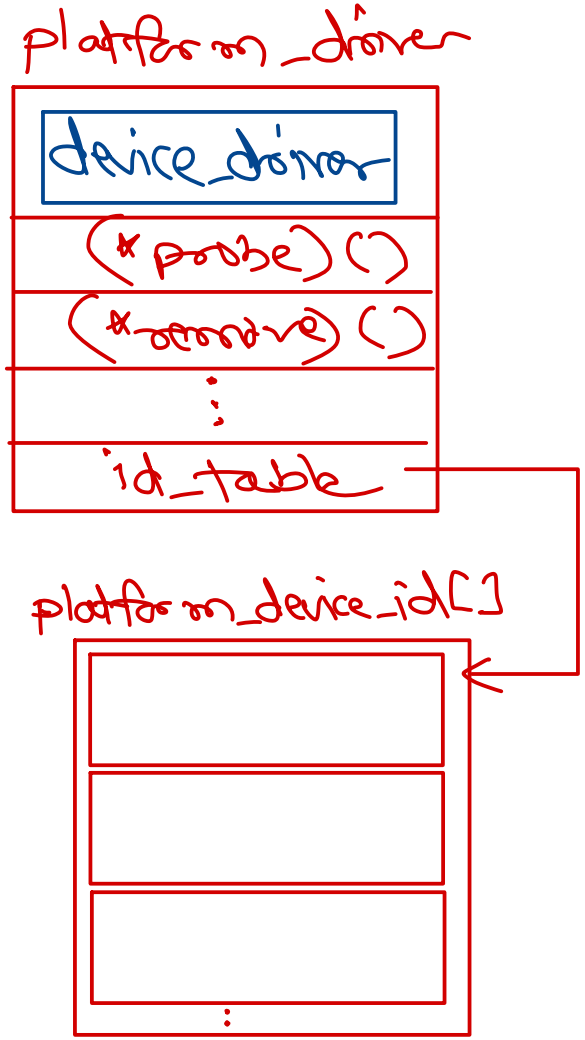
- To register platform driver
  - platform_driver_register(drv);

Platform Driver

- https://www.kernel.org/doc/Documentation/driver-model/platform.txt

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
};
```

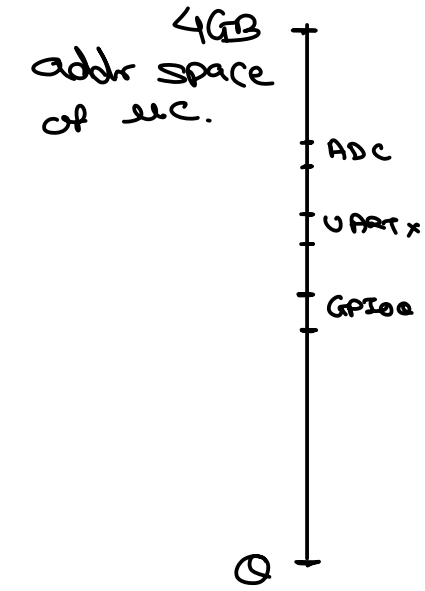
- To register platform driver
  - platform_driver_register(drv);

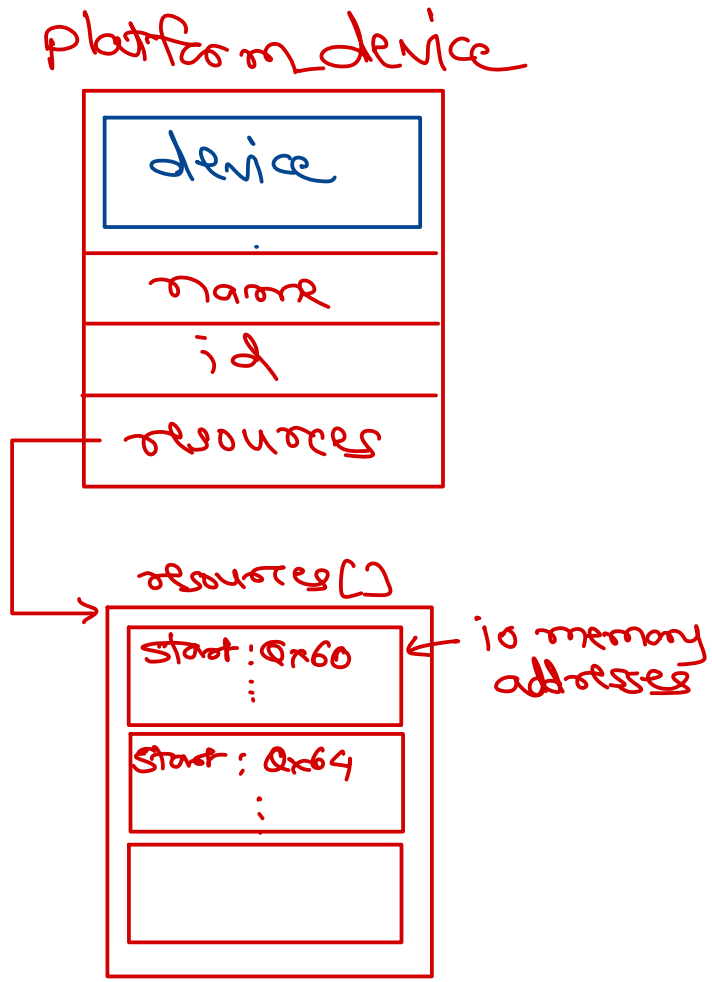


Platform Device

- Platform device is represented by struct platform_device.

```
struct platform_device {
        const char      *name;
        u32             id;
        struct device   dev;
        u32             num_resources;
        struct resource *resource;
};
```

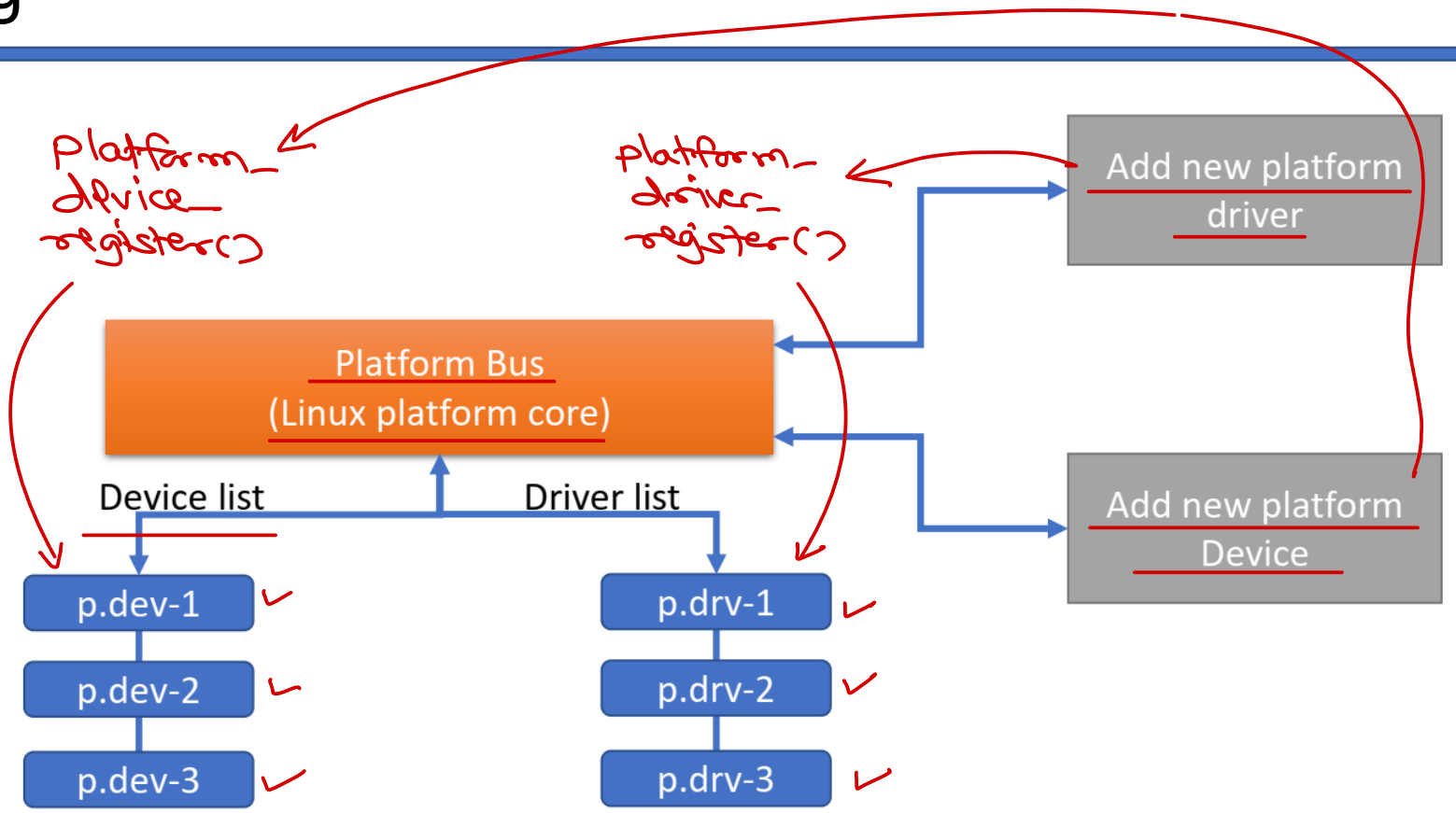




- The device is registered using: platform_device_register(dev);
  - From board file (compile time) or device driver (dynamically) – deprecated.
  - Devices are now registered using device tree.

Device and Driver matching

- Platform device and Platform driver are matched by the bus core matching mechanism.
  - Driver can detect the matching device added into the system.
  - Correct driver is auto-loaded when new device is added into the system.
- Each bus type has its match function that scans device and driver list.
- Linux platform core maintains platform device and driver list. It is auto updated when device or driver is added. e.g. /sys/bus/i2c – devices/drivers

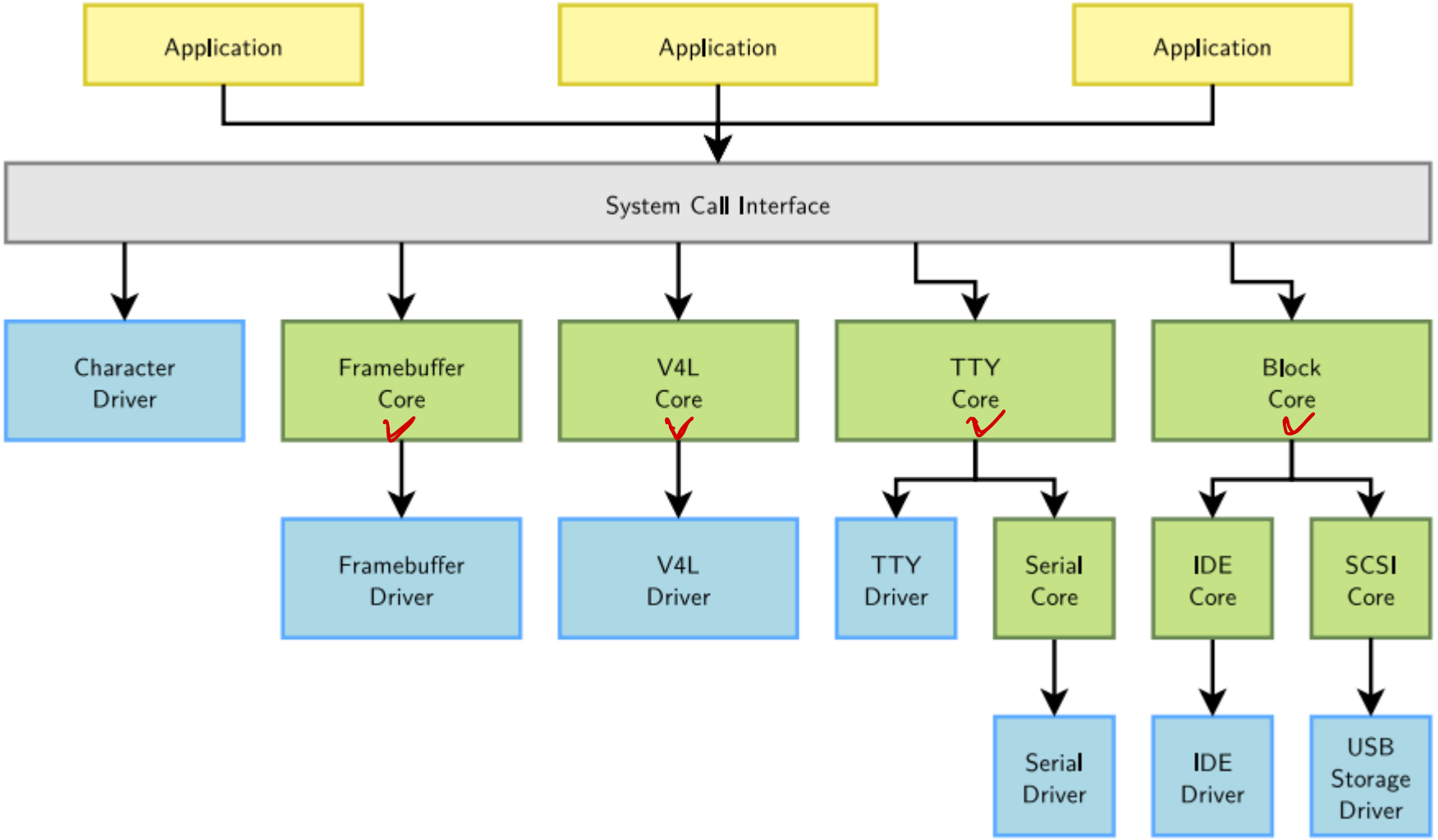


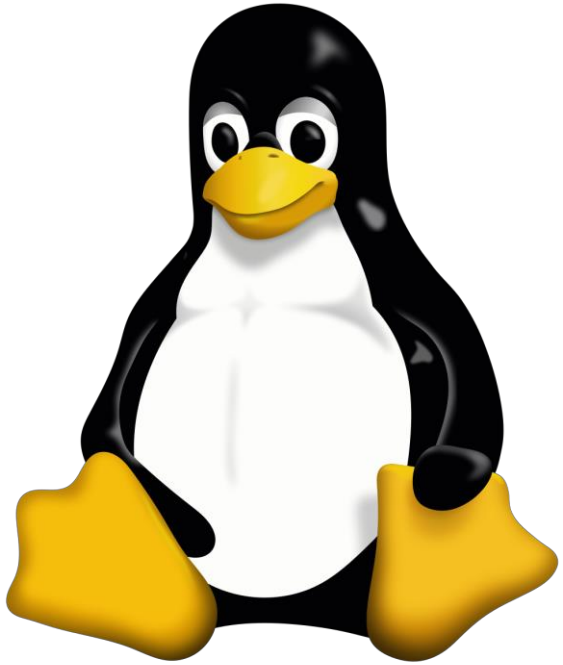
- Match is done by name or ids.
- Upon match, probe() of driver is called by the core.
- When device/driver is removed, remove() is called.

platform_driver operations

- Platform driver must implement and register these method while platform_driver_register().
- When matching is done by the core, probe() will be called with platform_device as argument.
- probe() is responsible for
  - Device detection (verify) and initialization
  - Mapping IO memory and Register ISRs
  - Create user space access points (/dev or /sys)
  - Register device to the kernel framework
- When device or driver is removed, remove() will be called by the core.
- remove() is responsible for
  - Free memory and ISRs.
  - Shut-down or de-initialize the device.
  - Unregister device from the kernel framework
- suspend() is called to put device is pause/sleep (low power) state.
- resume() is called to set device in normal state (from sleep state).
- shutdown() is called to stop the device during system shutdown.

Kernel frameworks

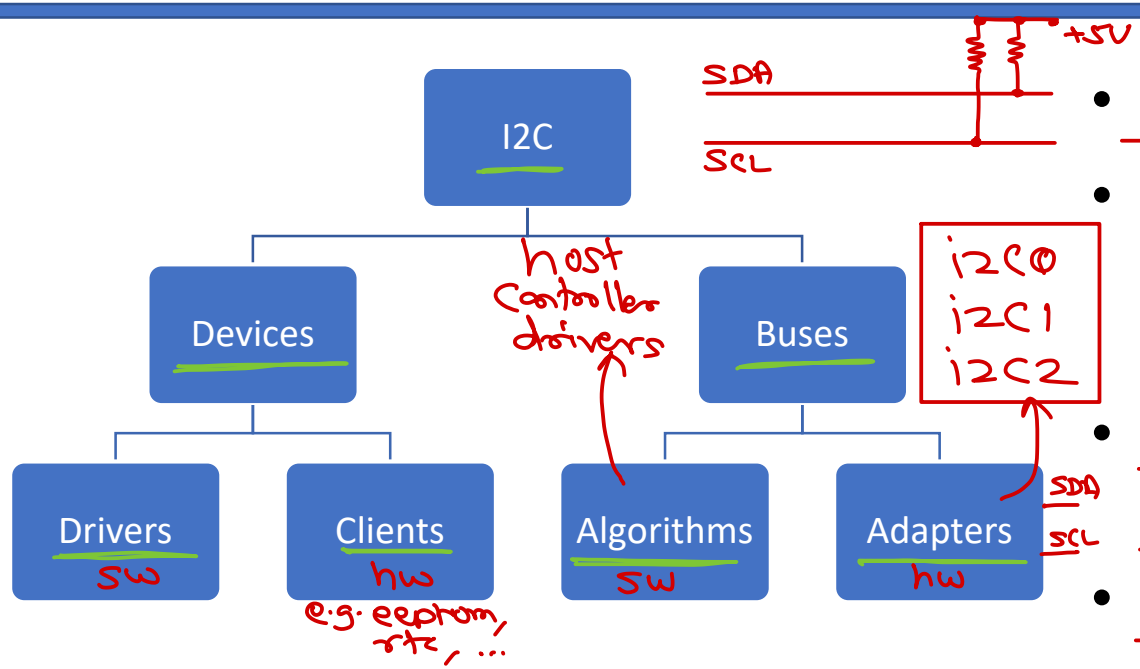




Linux I2C Bus

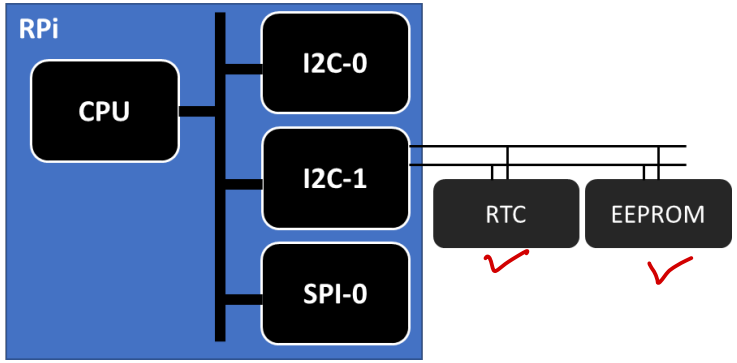
Sunbeam Infotech

Linux I2C sub-system



- I2C client is a slave device (chip) on bus.
- I2C driver handle/operate I2C client device.
  - I2C drivers and clients coupled with each other.
  - I2C driver is invokes APIs from I2C sub-system. (core)
- I2C host controller/adapter drivers usually provided by the vendor.
- I2C buses and devices are populated in Linux.
  - /dev/i2c*
  - /sys/bus/i2c/devices/

- Adapter represent a bus. Tie up algorithm and bus number. Each adapter based on an algorithm driver or own implementation.
- Algorithm driver contains general code for class of I2C adapters.



I2C device driver

eeprom AT24C256 – addr packet = 1010 0000 / 7bit addr = 0 → R|W

0x50

0x50 for AT24C256

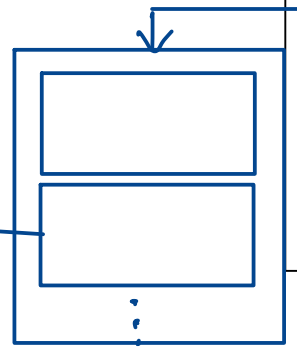
- Get the I2C adapter.
  - *struct i2c_adapter *i2c_get_adapter(int bus_number);* (0 or 1 or 2)
- Create the i2c_board_info structure and create a device using that.
  - *struct i2c_board_info my_board_info = { I2C_BOARD_INFO("my_dev", i2c_addr_7bits) };*
  - *struct i2c_client * i2c_new_client_device (struct i2c_adapter * adap, struct i2c_board_info const * info);*
- Create the i2c_device_id for the slave device and register that.
  - *struct i2c_device_id my_dev_id[] = { { "my_dev", 0 }, { } };*
- Create the i2c_driver structure and add that to the I2C subsystem.
  - *struct i2c_driver my_driver =*
    { .driver = { .name="my_dev", .owner=THIS_MODULE }, .probe=my_dev_probe, .remove=my_dev_remove, .id_table=my_dev_id };
  - *i2c_add_driver(struct i2c_driver *i2c_drive);*
- Now transfer the data between master and slave (in Linux framework, char/block device driver).
  - *i2c_master_send(), i2c_smbus_write_byte(), i2c_smbus_write_byte_data(), i2c_smbus_write_word_data(), i2c_smbus_write_block_data(), i2c_master_recv(), i2c_smbus_read_byte(), i2c_smbus_read_byte_data(), i2c_smbus_read_word_data(), i2c_smbus_read_block_data(), i2c_transfer().*
- At the end, remove the device and the driver.
  - *void i2c_unregister_device(struct i2c_client *client);*
  - *void i2c_del_driver(struct i2c_driver *i2c_drive);*

- SMBus protocol is compatible & subset of I2C.
- SMBus speed is 10 KHz to 100 KHz with timeout of 35ms (clock stretching < 35ms).

I2C device driver

```
struct i2c_board_info {
  char type[I2C_NAME_SIZE];
  unsigned short flags;
  unsigned short addr;
  void * platform_data;
  struct dev_archdata * archdata;
  struct device_node * of_node;
  struct fwnode_handle * fwnode;
  int irq;
};
```

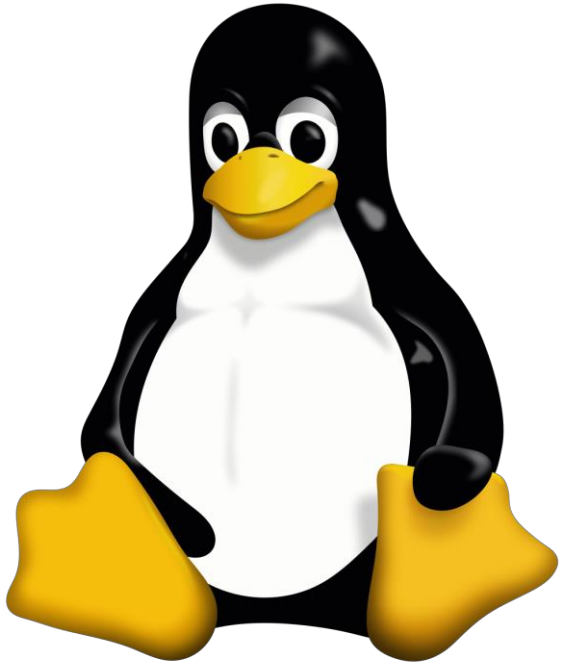
```
struct i2c_driver {
  unsigned int class;
  int (* attach_adapter) (struct i2c_adapter *);
  int (* probe) (struct i2c_client *, const struct i2c_device_id *);
  int (* remove) (struct i2c_client *);
  void (* shutdown) (struct i2c_client *);
  void (* alert) (struct i2c_client *, unsigned int data);
  int (* command) (struct i2c_client *client, unsigned int cmd, void *arg);
  struct device_driver driver;
  const struct i2c_device_id * id_table;
  int (* detect) (struct i2c_client *, struct i2c_board_info *);
  const unsigned short * address_list;
  struct list_head clients;
};
```



```
struct i2c_device_id {
  char name[I2C_NAME_SIZE];
  kernel_ulong_t driver_data;
};
```

I2C device driver – data transfer

- I2C client driver initiates transfer using a function like i2c_transfer(), i2c_master_send(), etc.
- This internally invokes master_xfer() in the bus driver (drivers/i2c/busses/*).
- The bus driver splits the entire transaction into START, STOP, ADDRESS, READ with ACK, READ with NACK, etc. The bus driver writes to the I2C hardware adaptor to generate these conditions on the I2C bus one by one, sleeping on a wait queue in between.
- Once the hardware has finished a transaction on the bus (for eg a START condition), an interrupt will be generated and the ISR will wake up the sleeping master_xfer().
- Once master_xfer() wakes up, it will advise the hardware adaptor to send the next condition (for eg ADDRESS of the chip).
- This continues till the whole transaction is over and return back to the client driver.
- *Since transfer functions sleeps, I2C transactions cannot be used in ISRs.*

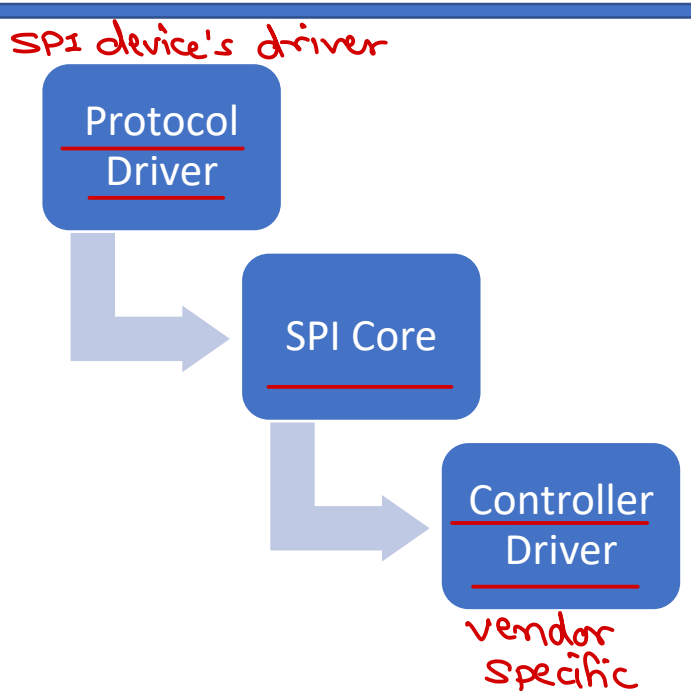


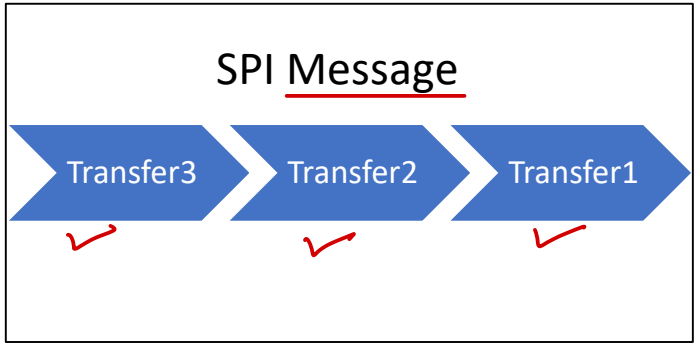
Linux SPI Bus

Sunbeam Infotech

Linux SPI Sub-system

- SPI sub-system has 3 parts
  - SPI core – provides core data structures, registration, cancellation and unified interface for SPI drivers. It is platform independent. (kernel/drivers/spi/spi.c).
  - SPI controller driver – low-level (hardware register level) platform specific driver usually implemented by vendor. Loaded while system booting & provides appropriate read(), write().
  - SPI protocol driver – handle/interact with SPI device. The interaction is in terms of messages and transfers.
- SPI Transfers and Messages
  - Transfer – defines a single operation between master and slave. Use tx/rx buffer pointers and optional delay/chip select behaviour after op.
  - Message – atomic sequence of transfer. Argument to all SPI read/write functions.





SPI device driver

SPI mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

- Get the SPI Controller driver.
  - *struct spi_controller * spi_busnum_to_master(u16 bus_num);*
- Add the slave device to the SPI Controller.
  - *struct spi_board_info my_dev_info = { .modalias = "my_spi_driver", .max_speed_hz = 4000000, .bus_num = 1, .chip_select = 0, .mode = SPI_MODE_0 };*
  - *struct spi_device * spi_new_device(struct spi_controller *ctlr, struct spi_board_info *chip); - spi_alloc_device() + spi_add_device();*
- Configure the SPI
  - *int spi_setup(struct spi_device *spi); // call after any change in spi_device.*
- Transfer the data between master and slave.
  - *int spi_sync_transfer(struct spi_device *spi, struct spi_transfer *xfers, unsigned int num_xfers);*
  - *int spi_async(struct spi_device *spi, struct spi_message *message);*
  - *int spi_write_then_read(struct spi_device * spi, const void * txbuf, unsigned n_tx, void * rxbuf, unsigned n_rx);*
- At the end remove the device & driver.
  - *void spi_unregister_device(struct spi_device *spi);*

```
struct spi_board_info {
  char modalias[SPI_NAME_SIZE];
  const void      *platform_data;
  const struct property_entry *properties;
  void *controller_data;
  int irq;
  u32 max_speed_hz, mode;
  u16 bus_num, chip_select;
};
```

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>