

Training Agenda

- **Telechip Source code (uboot, Kernel) Download, Build and Flashing steps Hands-on with Telechip**
- **Kernel Modules**
- **Device Tree**
- **Device Driver (Platform Driver, I2C Driver)**
- **Partition Information for Telechip**
- **Linux Memory Management Overview**

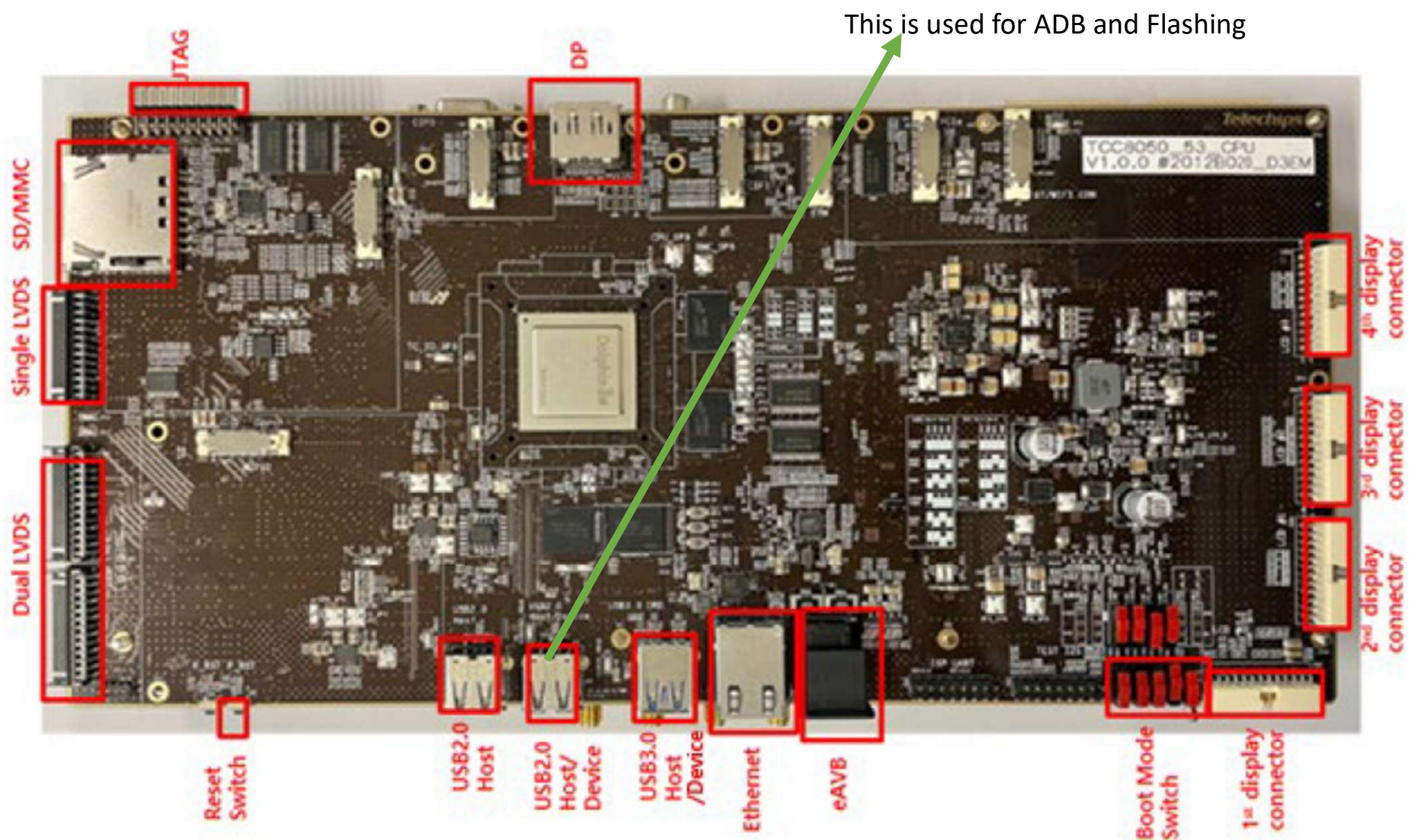
Topic 1 :

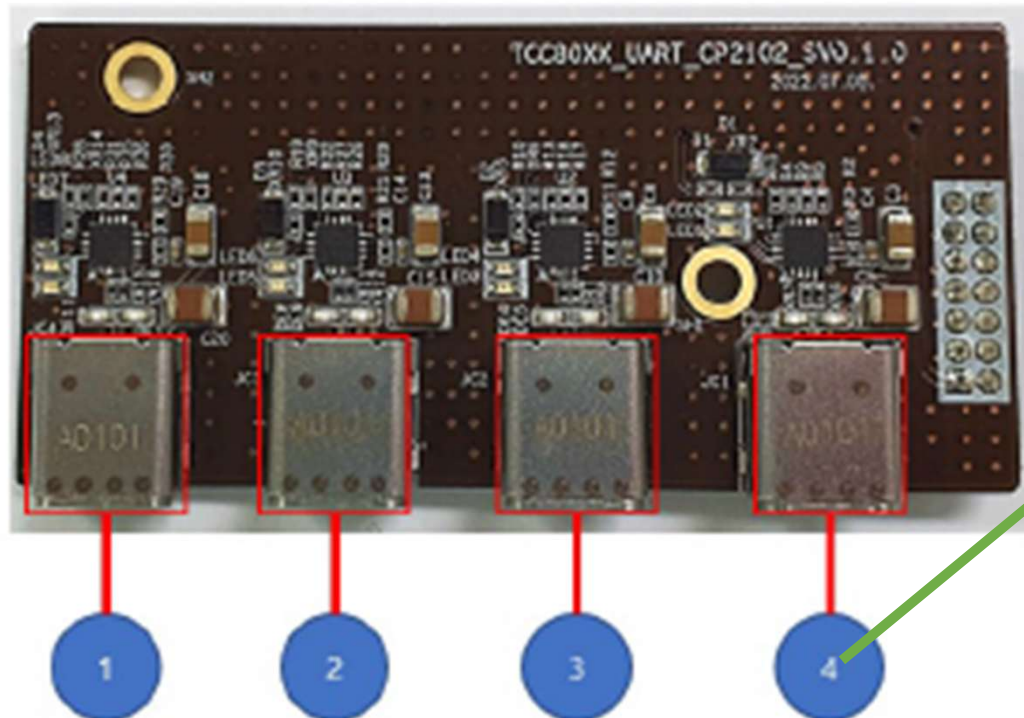
Telechip Source code (uboot, Kernel)

Download, Build and Flashing steps

Agenda :

- Compiling the Kernel and Modules for Telechips
- Cross-compiling the Linux kernel.
- Compiling and linking kernel modules for Telechips.
- Sub-Task: Set up a cross-compilation toolchain on a host machine and compile the kernel for Telechips

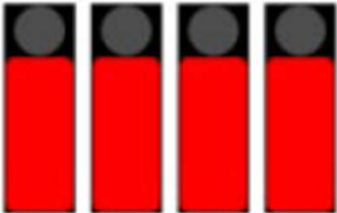
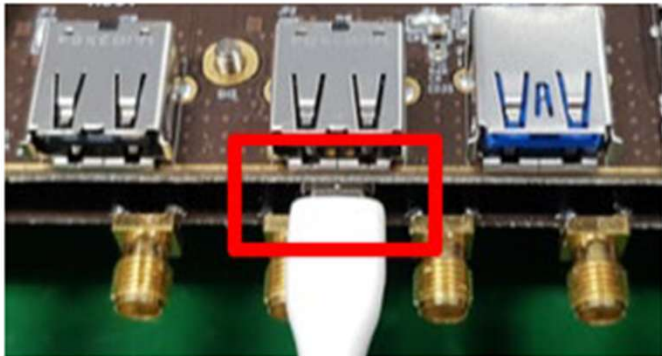




This is used for Main Core / Android Serial Console

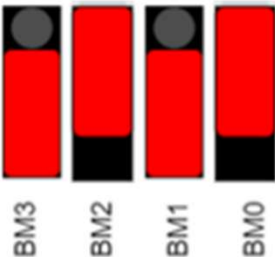
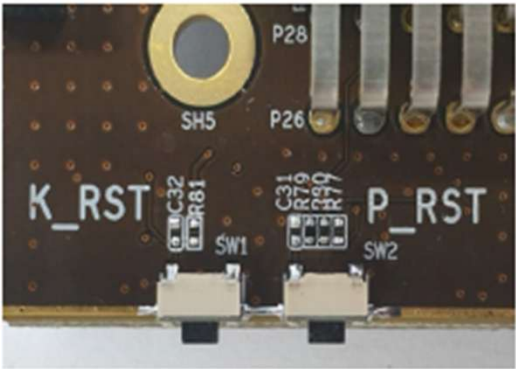
Number	Reference Number	Name	Description
1	JC4	USB Type-C Connector	Reserved
2	JC3	USB Type-C Connector	UART debug port for CR5
3	JC2	USB Type-C Connector	UART debug port for CA53
4	JC1	USB Type-C Connector	UART debug port for CA72

Table 2.3 Set to USB 2.0 Boot Mode

Category	USB Boot Mode Switch	FWDN Port
TCC805x/TCC803xPE	 <div data-bbox="617 938 951 1011"> BM3 BM2 BM1 BM0 </div>	

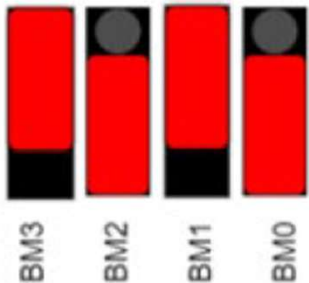
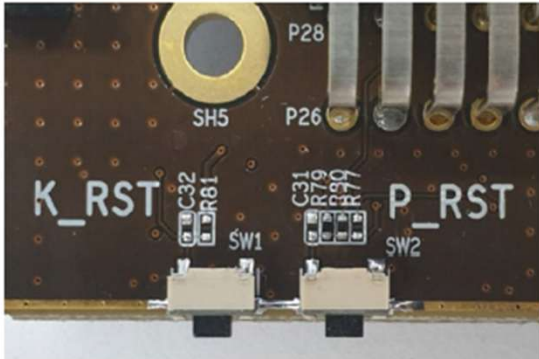
2.2.2 eMMC Boot Mode

Table 2.5 Set to eMMC Boot Mode

Category	eMMC Boot Mode Switch	Reset Switch	Note
TCC805x/TCC803xPE	 BM3 BM2 BM1 BM0		<p>Reset button on the bottom left corner of the board.</p> <p>K_RST resets power management I/O block. P_RST reboots a system.</p>

2.2.4 SNOR/UFS Boot Mode

Table 2.7 Set to SNOR/UFS Boot Mode

Category	SNOR/UFS Boot Mode Switch	Reset Switch	Note
TCC8050/TCC8053	 BM3 BM2 BM1 BM0		Reset button on the bottom left corner of the board.

Practice -

Telechip Serial Console Logs and ADB

Practice -

Telechip Source code Download
Steps

Practice -

Telechip Uboot Source code Build
Steps

Practice -

Telechip Kernel Source code Build
Steps

Practice -

Telechip code Flashing Steps

Topic 2 :

Kernel Modules

Agenda :

- Develop a custom kernel module
- Cross-compiling the Kernel Modules for Telechip HW
- Using ADB Push Load the kernel modules into Telechips HW.

Kernel Modules

Kernel modules are dynamically loadable pieces of code extending kernel functionality.

Example: **Device Drivers** for peripherals (USB, Wi-Fi, GPU).

Benefits:

- Modular and flexible
- Reduces kernel size
- Can be loaded/unloaded dynamically

Types of Kernel Modules

1.Built-in Modules: Compiled directly into the kernel.

2.Loadable Kernel Modules (LKMs): Can be loaded/unloaded at runtime.

In-Tree vs. Out-of-Tree Modules

In-Tree Modules	Out-of-Tree Modules
Modules that are part of the Linux kernel source tree and built with the kernel.	Third-party modules that are built independently and loaded separately.

Example:

- ext4 (In-Tree)
- NVIDIA GPU driver (Out-of-Tree)

Kernel symbol table: The names & addresses of all the kernel functions are present in their table.

Advantages of Kernel Modules:

- Modules make it easy to develop drivers without rebooting: **load, test, unload, rebuild, load...**
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.

Drawbacks of Kernel Modules:

- once loaded, have full access to the whole kernel address space. No particular protection.

User Space Application /Module vs Kernel Modules

User Space - Application

1. Application starts with main() function and when main() function returns the application terminates
2. All the Standard C library functions are available to the application.
3. When we build an application, it will get compiled and linked to libraries so that executable file will be generated.
4. Currently running process info
/proc

Kernel Space - Module

1. In kernel Module, there will not be any main() function.
2. But kernel module can not call standard library functions. It can call only kernel functions, which are present, in side the kernel.
3. But when we build kernel module, it only will get compiled, it will not get linked to kernel functions, as kernel is not available as a library.
4. Currently running modules info
/sys/module/<module name>

Kernel Module Utilities

To dynamically load or unload a driver, use these commands, which reside in the [/sbin](#) directory, and must be executed with root privileges:

- [lsmod](#) – Lists currently loaded modules
- [modinfo <module_file>](#) - Gets information about a module
- [insmod <module_file>](#) – Inserts/loads the specified module file
- [modprobe <module>](#) – Inserts/loads the module, along with any dependencies
- [rmmod <module>](#) – Removes/unloads the module.

Kernel message Logging

printf

- Floating point used (%f,%lf)
- Dump the output to some console.

printk

- No floating point
- All printk calls put this output in to the log ring buffer of the kernel.

- All the printk output, by default **/var/log/messages** for all log values.
- This file is not readable by the normal user. Hence, user space utility **“dmesg”**.

Kernel message Logging

There are eight macros defined in [linux/kernel.h](#) in the kernel source, namely:

1. `#define KERN_EMERG "<0>" /* system is unusable */`
2. `#define KERN_ALERT "<1>" /* action must be taken immediately */`
3. `#define KERN_CRIT "<2>" /* critical conditions */`
4. `#define KERN_ERR "<3>" /* error conditions */`
5. `#define KERN_WARNING "<4>" /* warning conditions */`
6. `#define KERN_NOTICE "<5>" /* normal but significant condition */`
7. `#define KERN_INFO "<6>" /* informational */`
8. `#define KERN_DEBUG "<7>" /* debug-level messages */`

Kernel Functions return guidelines

- The kernel programming guideline for returning values from a function. Any kernel function needing error handling typically returns an integer-like type.
- For an **error**, we return a **negative** number: a minus sign appended with a macro that is available from a kernel header include [linux/errno.h](#)
- For **success**, **zero** is the most common return value.
- For **some additional information**, a **positive** value is returned, the value indicating the information, such as the number of bytes transferred by the function.

Driver vs Module

Feature	Kernel Module	Driver (Kernel Module)
Is it loadable?	Yes	Yes
Talks to hardware?	Not usually	Yes
Creates /dev/ file?	No	Yes (character/block drivers)
Registers with subsystem?	No	Yes
Needed for hardware?	Optional (demo only)	Required
Example	hello.ko	my_gpio_driver.ko

Practical =>

- Develop Hello world kernel module.
- Cross-compiling the Kernel Modules for Telechip HW
- Using ADB Push Load the kernel modules into Telechips HW

Practice -

Hello World Code Cross compile
and build on telechip HW

NOTE : A simple C program that prints 'Hello, Telechip!'

Practice -

Hello World kernel Module Code
Cross compile and build on telechip
HW

NOTE :- This kernel module that logs messages when loaded and unloaded.

Practice -

Simple USB notifier kernel Module
Code Cross compile and build on
telechip HW

NOTE :- This kernel module detects when USB keyboards or mice are connected or disconnected

Topic 3 :

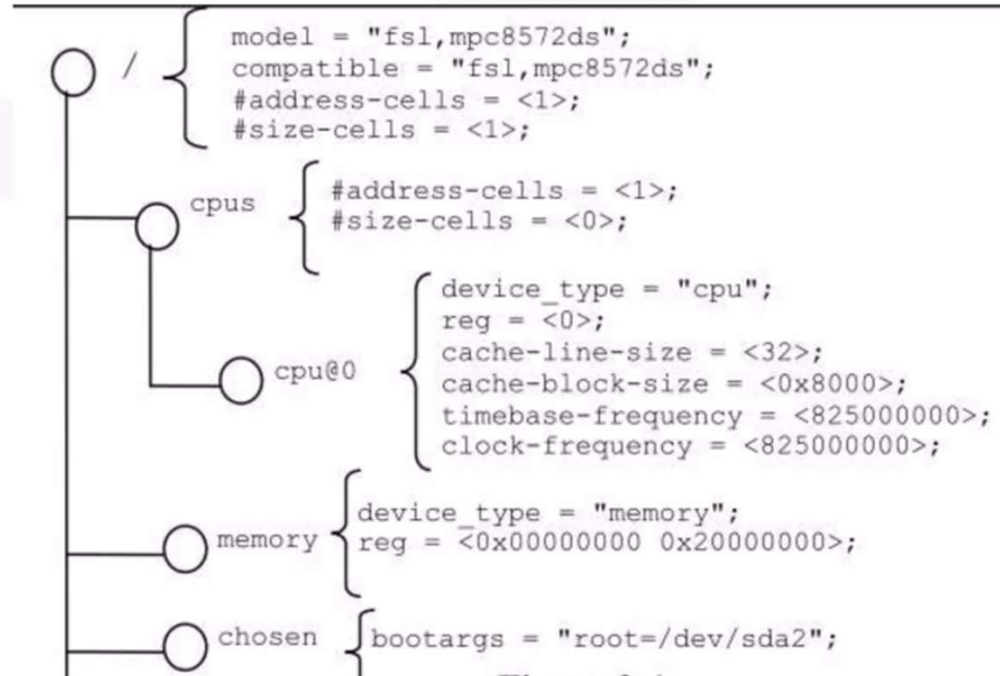
Device Tree

Agenda :

- Device tree, Syntax, and Structure
- Syntax of device tree source files.
- Nodes, properties, and overlays.
- Board Configuration Using the Device Tree
- Configuring board-specific peripherals.
- Device tree binding and its role in driver loading.

What is a Device Tree?

- The **Device Tree (DT)** is a data structure used to **describe hardware** to the Linux kernel.
- It is an alternative to **platform data** and is commonly used in **embedded systems**.
- Device Trees help in **hardware abstraction**, making it easier to support multiple boards with the same kernel.

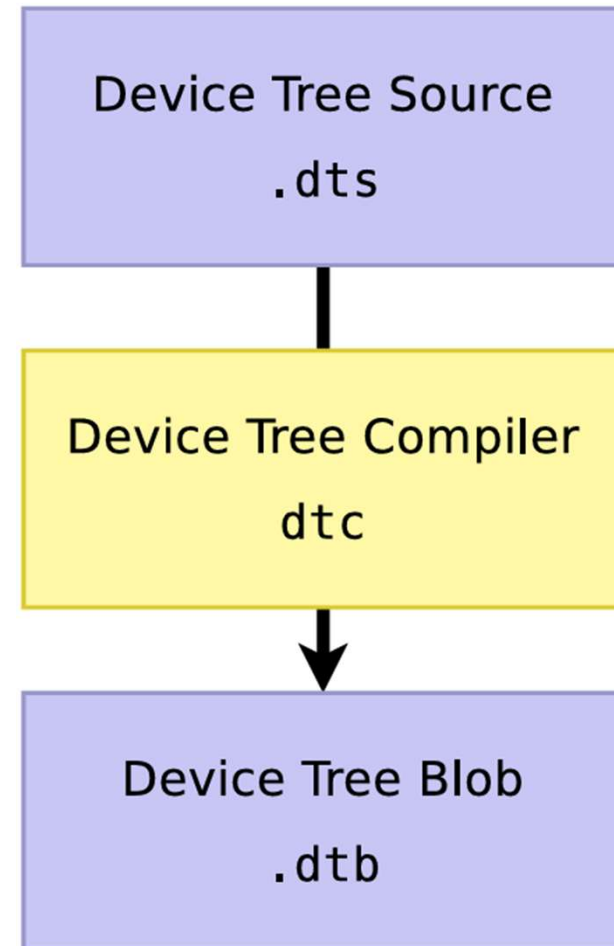


For each board unique device tree file needed

Boards	Device Tree File
Beaglebone Black	am335x-boneblack.dts
AM335x General Purpose EVM	am335x-evm.dts
AM335x Starter Kit	am335x-evmsk.dts
AM335x Industrial Communications Engine	am335x-icev2.dts
AM437x General Purpose EVM	am437x-gp-evm.dts, am437x-gp-evm-hdmi.dts (HDMI)
AM437x Starter Kit	am437x-sk-evm.dts
AM437x Industrial Development Kit	am437x-idk-evm.dts
AM57xx EVM	am57xx-evm.dts, am57xx-evm-reva3.dts (revA3 EVMs)
AM572x IDK	am572x-idk.dts
AM571x IDK	am571x-idk.dts
K2H/K2K EVM	keystone-k2hk-evm.dts
K2E EVM	keystone-k2e-evm.dts
K2L EVM	keystone-k2l-evm.dts
K2G EVM	keystone-k2g-evm.dts

DTS vs DTB

- The **Device Tree Source (DTS)** is a plain text file that describes hardware components.
- It is compiled into a **Device Tree Blob (DTB)**, which is loaded by the bootloader.
- The **kernel reads the DTB** to configure hardware drivers.



Device Tree File Structure

DTS file consists of –

Header	(/dts-v1/;)
Root node	(/)
Nodes	(describe hardware components)
Properties	(define attributes)

Device Tree Syntax Basics

- **Node:** Represents a device/peripheral.
- **Properties:** Key-value pairs inside a node.
- **Subnodes:** Child nodes inside a parent.
- **Labels:** Used for references.
- **Includes (.dtsi):** Allow reusing common definitions.
- **File types:**
 - .dts → board-specific
 - .dtsi → SoC/platform/common include

Telechip Device File mapping

```
tcc8050-android-lpd4x322_sv1.0.dts
|— tcc8050-android-ivi.dtsi
|   |— tcc-pmap-805x-android-customized.dtsi
|   |— tcc8050.dtsi
|   |   |— tcc805x.dtsi
|   |   |   |— tcc8050_53-pinctrl.dtsi
|   |   |— tcc805x-android.dtsi
|   |   |— tcc-pmap-common.dtsi
|— tcc8050-android-ivi-display.dtsi
|   |— tcc805x-display.dtsi
```

```
#include "tcc805x-android.dtsi"

/{
    model = "Telechips TCC8050 Evaluation Board";
    compatible = "telechips,tcc8050", "telechips,tcc805x";
};

&pcie {
    reset-gpio = <&gpmc 15 0>;
    //status = "okay";
};

&reserved_memory {
    #address-cells = <2>;
    #size-cells = <2>;
    /*-----
    * Secure Area 1 (CPU R/W, VPU X, GPU R/W, VIOC R)
    *-----
    */
    pmap_fb_video: fb_video {
        compatible = "telechips,pmap";
        telechips,pmap-name = "fb_video";
        alloc-ranges = <0x0 SECURE_AREA_1_BASE 0x0 SECURE_AREA_1_SIZE>;
        size = <0x0 PMAP_SIZE_FB_VIDEO>;
        telechips,pmap-secured = <1>;
        no-map;
    };
};
```

```
&rotary_encoder0 {  
    compatible = "rotary-encoder";  
    status = "disable";           //To use it, change the status from disable to okay  
    //pinctrl-names = "default";  
    //pinctrl-0 = <&mc_vol>;      //Set gpio to match device  
                                //Interruptable gpio should be used  
    //gpios = <&gpma 5 1>, <&gpma 6 1>; //Set gpio to match device  
                                //Interruptable gpio should be used  
    linux,axis = <0>;  
    rotary-encoder,relative-axis;  
};
```

```
&rotary_encoder1 {  
    compatible = "rotary-encoder";  
    status = "disable";           //To use it, change the status from disable to okay  
    //pinctrl-names = "default";  
    //pinctrl-0 = <&mc_ctr>;      //Set gpio to match device  
                                //Interruptable gpio should be used  
    //gpios = <&gpma 7 1>, <&gpma 8 1>; //Set gpio to match device  
                                //Interruptable gpio should be used  
    linux,axis = <0>;  
    rotary-encoder,relative-axis;  
};
```

```
/ {  
    model = "rockchip rockchip-Board";  
    compatible = "rockchip,rv1106";  
  
    memory@80000000 {  
        device_type = "memory";  
        reg = <0x80000000 0x40000000>; // 1GB RAM  
    };  
};
```

- / { ... } → root node.
- compatible → identifies hardware.
- memory@80000000 → node name (memory) + unit address (@address).
- reg → base address + size

```
display@0 {  
    compatible = "rockchip,rockchip-display";  
    reg = <0x0 0x16000000 0x10000>;  
    clocks = <&clk_disp>;  
    resets = <&rst_disp>;  
    status = "okay";  
  
    panel {  
        compatible = "panasonic,lvds-panel";  
        width-mm = <250>;  
        height-mm = <140>;  
    };  
};
```

Device Tree Overlays

- Overlays allow **modifying the base device tree** dynamically.
- Useful for **configurable hardware like expansion boards**.

Practice -

- Walkthrough Telechip Device Tree File
- Walkthrough Telechip Device Tree Overlay File
- Understand each parameters of DTS files

Topic 4 :

Device Driver (Platform Driver, I2C Driver)

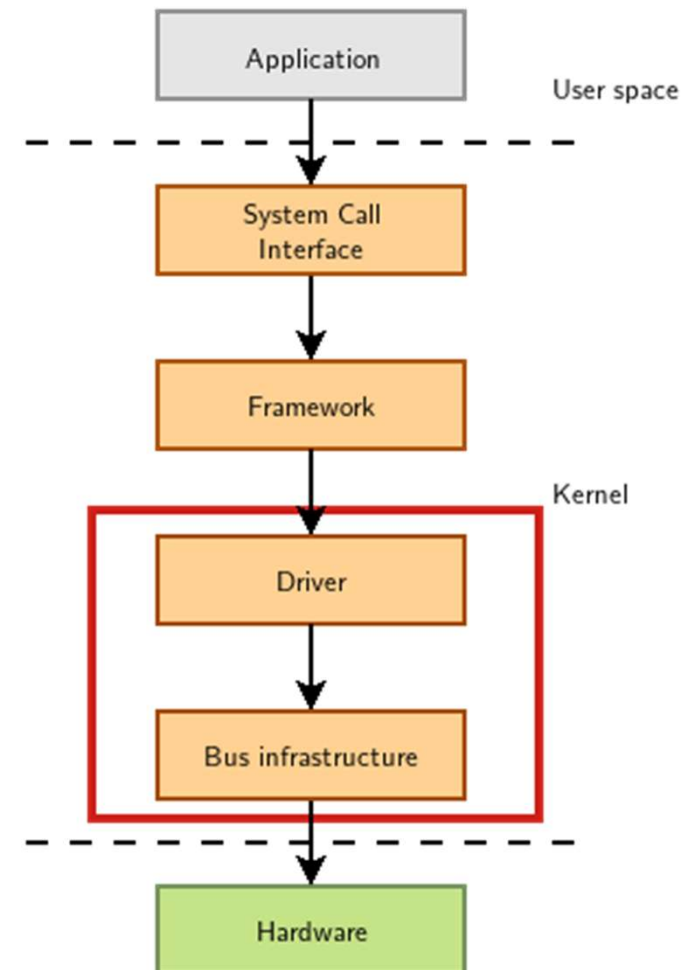
Agenda :

- Understand Device Driver
- Understand Platform devices and drivers
- Understand I2C platform driver along with registration and API's usage

Linux Device Drivers

What is device driver

- Software layer between application & hardware
- Used to control the device control & access the data from the device
- A driver is always Interfacing with –
 - a framework that allows the driver to expose the hardware features in a generic way.
 - a bus infrastructure, part of the device model, to detect/communicate with the hardware
- Linux Device Driver
 - Present in built with kernel
 - Loadable as module in run time

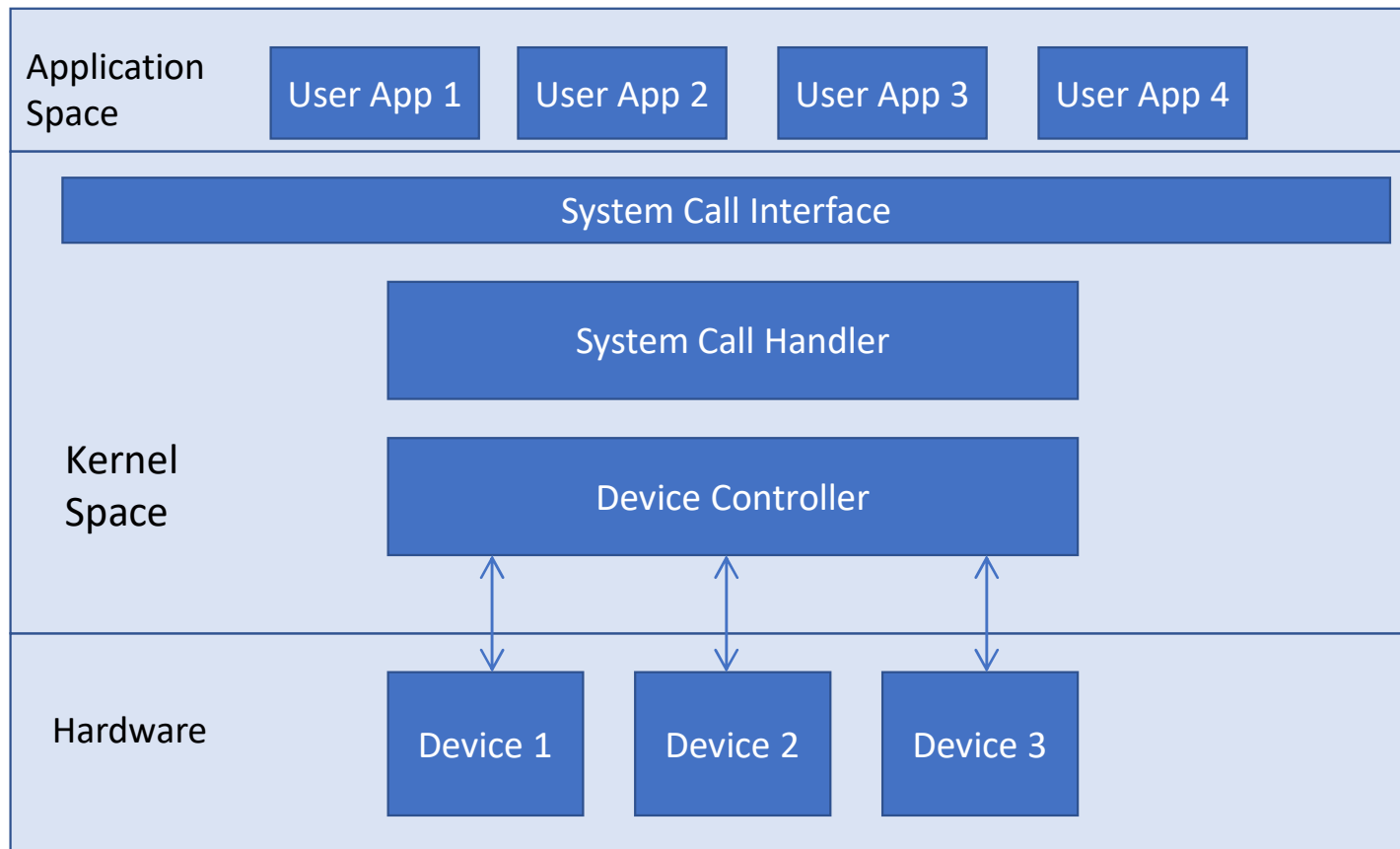


Why Drivers Are Needed

Kernel cannot directly control hardware or interact with user programs.

- A driver provides the glue between:
 - User space: Apps, commands (cat, echo, etc.)
 - Kernel space: Hardware, memory, buses

Linux Device Driver Architecture



Types of Device Driver

- Character Driver
 - Can be accessed as a stream of bytes like a file
 - Examples: UART, GPIO etc.
- Block Driver
 - Device that can host a file systems like a disk
 - Handles I/O operation with one or more blocks
 - Examples: HDD, SSD etc.
- Network Driver
 - Device that any network transaction through an interface
 - Interface is in charge of sending & receiving data packets
 - Examples: Ethernet, WiFi etc.

Comparison Table

Feature	Character Drivers	Block Drivers	Network Drivers
Data Unit	Stream of bytes	Fixed-size blocks	Packets
Access Pattern	Sequential	Random	Packet-oriented
Examples	Keyboards, serial ports	Hard drives, SSDs	Ethernet cards, Wi-Fi
Device Files	/dev/ttyS0, /dev/input	/dev/sda, /dev/sdb	No device files (e.g., eth0)
System Calls	read(), write()	read(), write()	socket(), send(), recv()
Buffering	Unbuffered	Buffered	Packet-based
Use Case	Low-latency devices	Storage devices	Network communication

Driver vs Module

Kernel Module

A kernel module is any piece of code that can be loaded into or removed from the Linux kernel at runtime (using insmod/rmmod).

Device Driver

- A driver is a specific type of kernel module that knows how to:
- Communicate with hardware (like I2C, UART, GPIO, etc.)
- Or represent virtual devices (like loopback, /dev/null)
- Register with subsystems like:
 - Character device (/dev/xyz)
 - Block device (/dev/sda)
 - Platform bus / I2C bus / USB stack

“All drivers are modules, but not all modules are drivers.”

Driver vs Module

Feature	Kernel Module	Driver (Kernel Module)
Is it loadable?	Yes	Yes
Talks to hardware?	Not usually	Yes
Creates /dev/ file?	No	Yes (character/block drivers)
Registers with subsystem?	No	Yes
Needed for hardware?	Optional (demo only)	Required
Example	hello.ko	my_gpio_driver.ko

Major & Minor Numbers

- Device Number represented in 32 bit
 - 12 Bits => Major Number
 - 20 Bits => Minor Number
- Device identification
 - c => character driver
 - b => block driver
- dev_t => device number representation
- MAJOR(dev_t dev);
- MINOR(dev_t dev);
- MKDEV(int major, int minor);

Allocating Device Numbers

- `register_chrdev_region`
 - `dev_t` first => beginning device number
 - `count` => number of contiguous device numbers
 - `name` => name of the device
 - 0 on success, negative error code on failure
- `alloc_chrdev_region`
 - `dev_t *dev` => output parameter on completion
 - `firstminor` => requested first minor
 - `count` => number of contiguous device numbers
 - `name` => name of the device
 - 0 on success, negative error code on failure

Freeing device number

- `unregister_chrdev_region`
 - `dev_t` first => beginning of device number range to be freed
 - `count` => number of contiguous device numbers
 - Returns void
 - The usual place to call would be in module's cleanup

mknod

- Creates block or character special device files
- `mknod [OPTIONS] NAME TYPE [MAJOR MINOR]`
 - NAME => special device file name
 - TYPE
 - c, u => creates a character (unbuffered) special file
 - b => creates a block special file
 - p => creates a FIFO
 - MAJOR => major number of device file
 - MINOR => minor number of device file

File Operations

- owner => pointer to module that owns structure
- open
- release
- read
- write
- poll
- ioctl
- mmap
- llseek
- readdir
- flush
- fsync

File Ops - Example

```
struct file_operations scull_fops = {  
    .owner = THIS_MODULE,  
    .llseek = hello_llseek,  
    .read = hello_read,  
    .write = hello_write,  
    .ioctl = hello_ioctl,  
    .open = hello_open,  
    .release = hello_release,  
};
```

Char Device Registration

- struct cdev
- Allocating dynamically
 - cdev_alloc
- cdev_init
 - cdev => cdev structure pointer
 - fops => file operations structure pointer
- cdev_add
 - cdev => cdev structure pointer
 - dev_t num => first device number
 - count => number of device numbers to be associated

Char device removal

- `cdev_del`
 - `cdev` => `cdev` structure pointer
- `cdev` structure pointer should not be accessed after passing it to `cdev_del`

FOPS - open & release

- inode => inode structure pointer
- filp => file structure pointer
- returns 0 or error code based on open

FOPS – read & write

- filp => file structure pointer
- buff => character buffer to be written by read / read by write function
- count => number of bytes
- offp => offset
- Returns 0 or negative error code accordingly

FOPS - ioctl

- inode => inode structure pointer
- filp => file structure pointer
- cmd => ioctl command
- args => ioctl command arguments
- returns 0 or error code

copy_to_user

- to => user pointer where data to be copied
- from => kernel pointer is data source
- count => number of bytes to be copied
- Used during driver read operations

copy_from_user

- to => kernel pointer where data to be copied
- from => user pointer is data source
- count => number of bytes to be copied
- Used during driver write operation

Practice -

- Show Char driver example

Linux Platform Device & Platform Driver

Platform Devices?

- Platform devices are hardware blocks integrated in SoCs.
- They cannot be discovered dynamically (unlike PCI/USB).
- Examples: UART, I²C, SPI, GPIO, timers, watchdog.
- Their existence is described in the Device Tree (DTS).

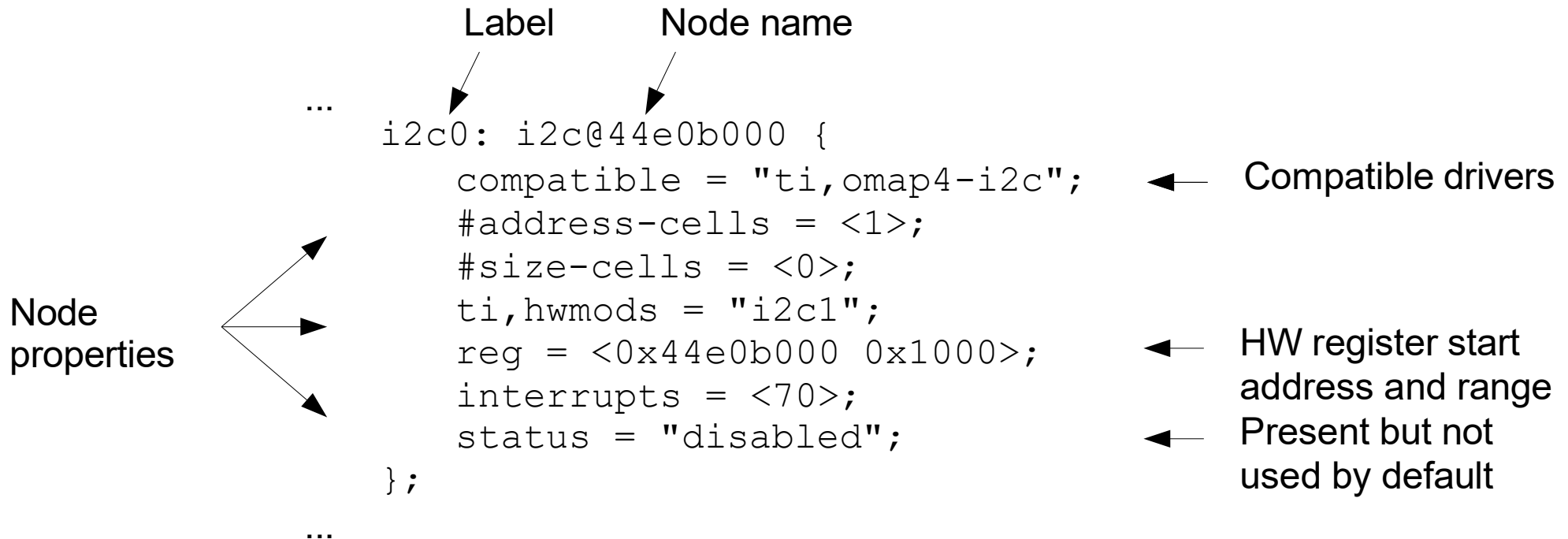
Platform Drivers?

- Kernel drivers that manage platform devices.
- Rely on DTS/board files to know about devices.
- No bus enumeration → matching is done via 'compatible' strings.

Platform Device vs Platform Driver

- Platform Device:
 - Defined in DTS node with 'compatible', 'reg', 'interrupts'.
 - Represents the HW block.
- Platform Driver:
 - Written in Driver code, registered via platform_driver struct.
 - Binds to platform devices using 'compatible' property.

Declaring a platform device: .dtsi example



From [arch/arm/boot/dts/am33xx.dtsi](#)

Instantiating a platform device: .dts example

Phandle
(reference
to label)

```
&i2c0 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&i2c0_pins>;  
  
    status = "okay";  
    clock-frequency = <400000>;  
  
    tps: tps@24 {  
        reg = <0x24>;  
    };  
  
    baseboard_eeprom: baseboard_eeprom@50 {  
        compatible = "at,24c256";  
        reg = <0x50>;  
  
        #address-cells = <1>;  
        #size-cells = <1>;  
        baseboard_data: baseboard_data@0 {  
            reg = <0 0x100>;  
        };  
    };  
};
```

Pin muxing configuration
(routing to external package pins)

Enabling this device, otherwise ignored
Node property: frequency

I2C bus identifier

List of devices on
i2c0

From [arch/arm/boot/dts/am335x-boneblue.dts](https://github.com/torvalds/linux/blob/master/arch/arm/boot/dts/am335x-boneblue.dts)

DT: matching devices and drivers

Platform drivers are matched with platform devices that have the same `compatible` property

```
static const struct of_device_id omap_i2c_of_match[] = {
    {
        .compatible = "ti,omap4-i2c",
        .data = &omap4_pdata,
    },
    {
        ...
    };
    ...
static struct platform_driver omap_i2c_driver = {
    .probe      = omap_i2c_probe,
    .remove     = omap_i2c_remove,
    .driver     = {
        .name    = "omap_i2c",
        .pm      = OMAP_I2C_PM_OPS,
        .of_match_table = of_match_ptr(omap_i2c_of_match),
    },
};
```

← Compatible string

From [drivers/i2c/busses/i2c-omap.c](https://github.com/torvalds/linux/blob/master/drivers/i2c/busses/i2c-omap.c)

Usage of the platform bus

Just like physical buses, the platform bus is used by the driver to retrieve information about each device

```
static int omap_i2c_probe(struct platform_device *pdev)
{
    ...

    struct device_node      *node = pdev->dev.of_node;
    struct omap_i2c_dev      *omap;
    ...
    irq = platform_get_irq(pdev, 0);
    ...
    omap = devm_kzalloc(&pdev->dev, sizeof(struct omap_i2c_dev), GFP_KERNEL);
    ...
    mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    omap->base = devm_ioremap_resource(&pdev->dev, mem);
    u32 freq = 100000; /* default to 100000 Hz */
    ...
    of_property_read_u32(node, "clock-frequency", &freq);
    ...
    return 0;
}
```

From [drivers/i2c/busses/i2c-omap.c](#)

Workflow: Device \leftrightarrow Driver

1. DTS describes HW device with 'compatible'.
2. Kernel creates a platform_device.
3. Driver registers a platform_driver.
4. Core matches 'compatible' \rightarrow calls probe().
5. Driver init HW and register subsystems (I²C, input, etc.).

Why Use Platform Drivers?

- Standardized framework in Linux.
- Separates HW description (DTS) from code.
- Easy to add/change devices without touching driver code.
- Supports suspend/resume and power mgmt.
- Works across different SoCs.

Example: Platform Device in Telechip DTS

```
i2c0: i2c@7600000 {  
    compatible = "telechips,tcc8050-i2c";  
    reg = <0x0 0x07600000 0x0 0x1000>;  
    interrupts = <0 101 4>;  
    clocks = <&clk_peri 12>;  
    status = "okay";  
};
```

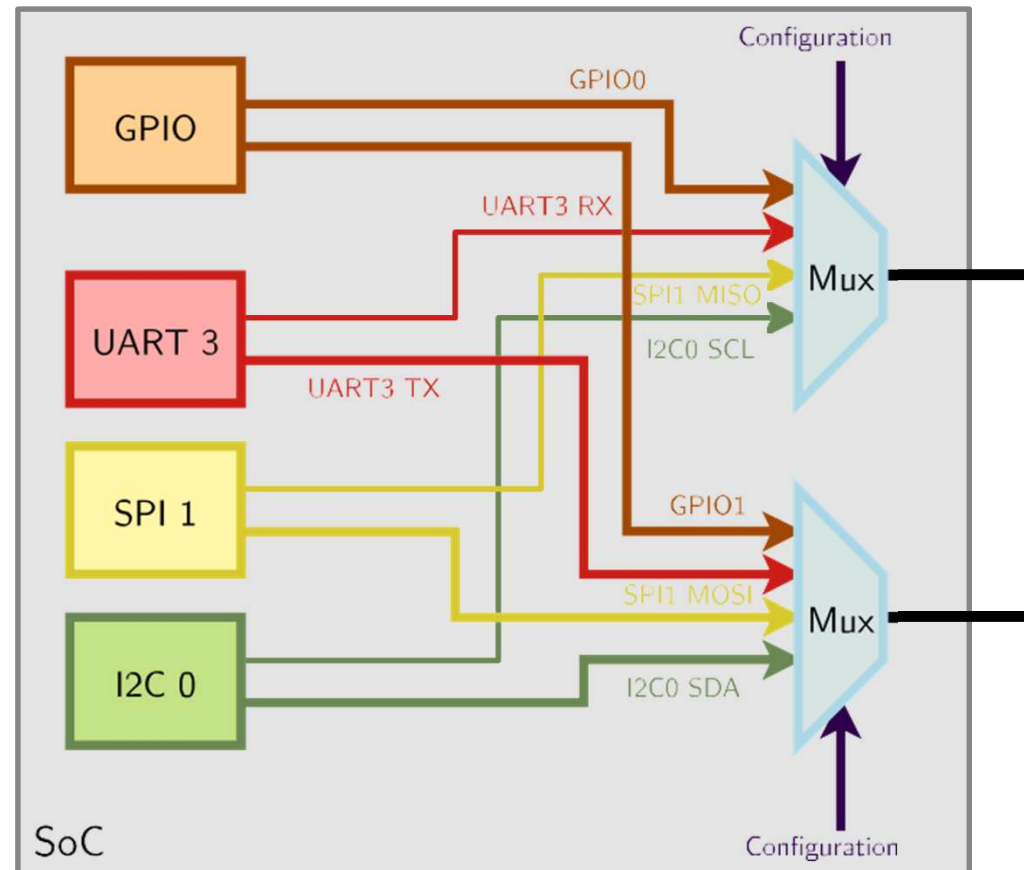
Example: Platform Driver in C

```
static const struct of_device_id tcc_i2c_dt_ids[] = {  
    { .compatible = "telechips,tcc8050-i2c" },  
    {}  
};
```

```
static struct platform_driver tcc_i2c_driver = {  
    .probe = tcc_i2c_probe,  
    .remove = tcc_i2c_remove,  
    .driver = {  
        .name = "tcc-i2c",  
        .of_match_table = tcc_i2c_dt_ids,  
    },  
};
```

Pin multiplexing

- Modern SoCs have too many hardware blocks compared to physical pins exposed on the chip package.
- Therefore, pins have to be multiplexed
- Pin configurations are defined in the Device Tree
- Correct pin multiplexing is mandatory to make a device work from an electronic point of view.



Linux I2C Driver

I2C Features

- Philips Semiconductors (now NXP Semiconductors, Qualcomm) developed a simple
 - Serial
 - 8-bit oriented,
 - bidirectional
 - 2-wire
 - synchronous communication protocol.
- Only 2 lines:
 - SDA (Serial Data Line)
 - SCL (Serial Clock Line)
- I2C protocol derived from System Management BUS (SMBUS – developed by INTEL)
- **I²C is an acronym for the “Inter-IC” bus, a simple bus protocol which is widely used where low data rate communications suffice.**

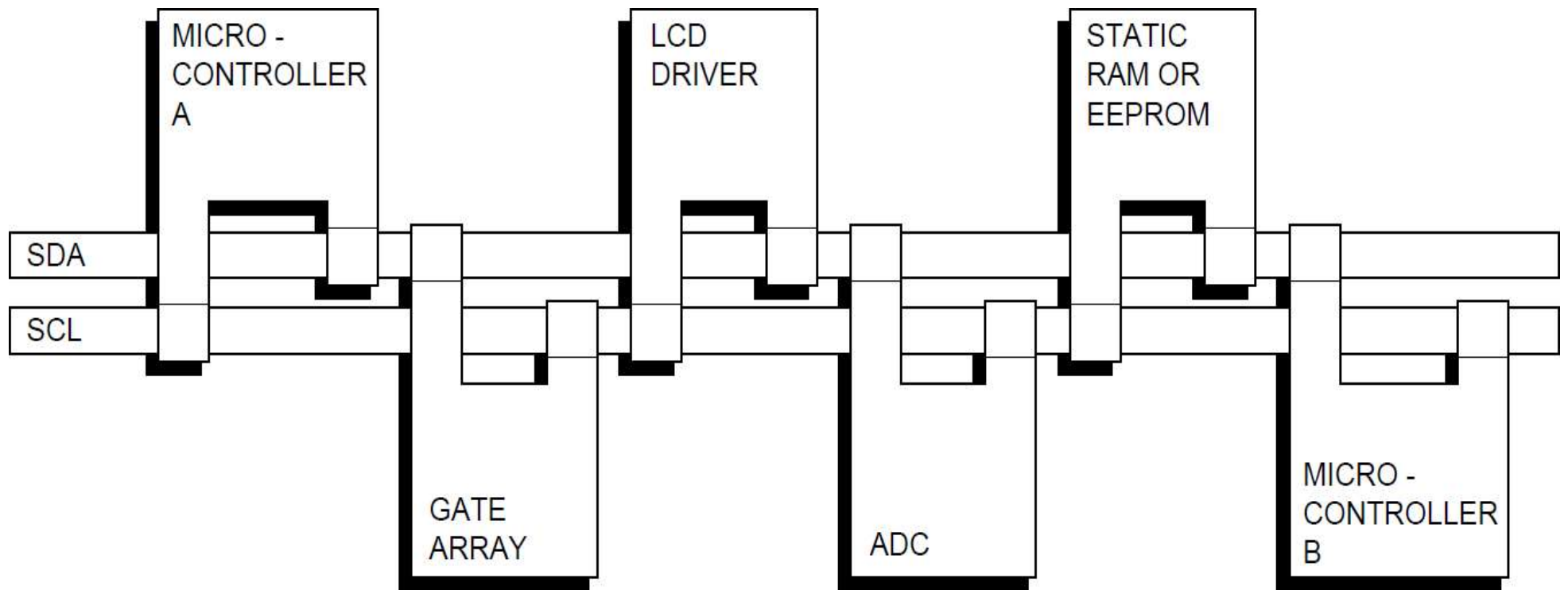
I2C Features

- Inter-Integrated Circuit (I²C) is a 2-wire bus. SDA (data) and SCL (clock).
- Master generates clock; slaves respond at 7/10-bit addresses.
- I2C various Data transfer modes:
 - Standard-mode: 100kbit/s
 - Fast-mode: 400kbit/s
 - Fast-mode Plus (Fm+): 1Mbit/s
 - High-speed mode: 3.4 Mbit/s
 - Ultra Fast-mode: 5 Mbit/s (Unidirectional mode)
- Each device connected to the bus is software addressable by a unique 7/10-bit address.
- simple master/slave relationships.
- masters can operate as master-transmitters or as master-receivers
- Common for sensors, PMICs, touch controllers, EEPROMs. Etc.

I2C Terminology

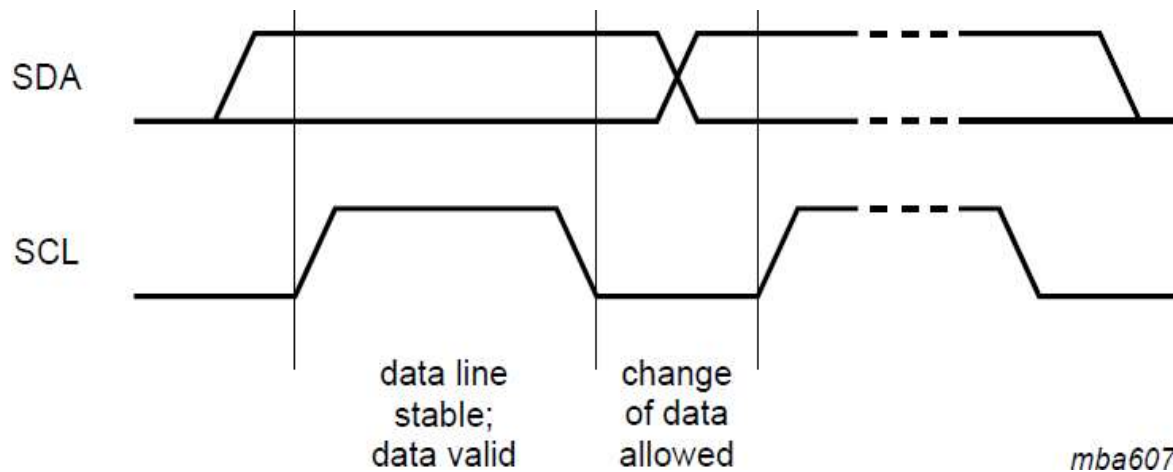
Term	Description
Transmitter	The device which sends data to the bus
Receiver	The device which receives data from the bus
Master	the device which initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master
Multi-Master	more than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted

I2C Bus Configuration



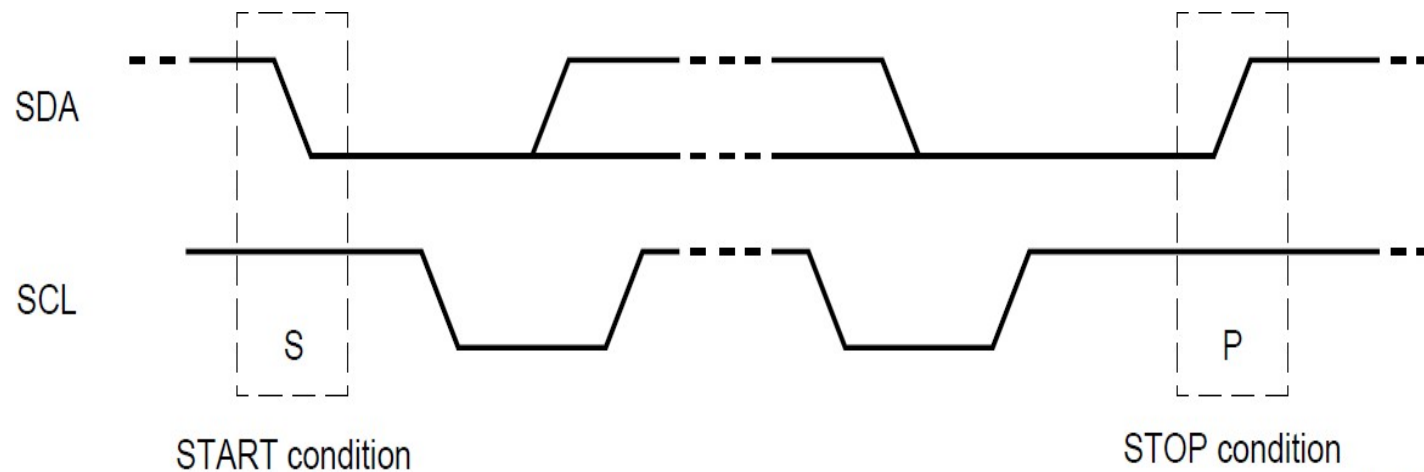
Data Validity

- The data on the SDA line must be stable during the HIGH period of the clock.
- The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure).
- One clock pulse is generated for each data bit transferred



Start and Stop Conditions

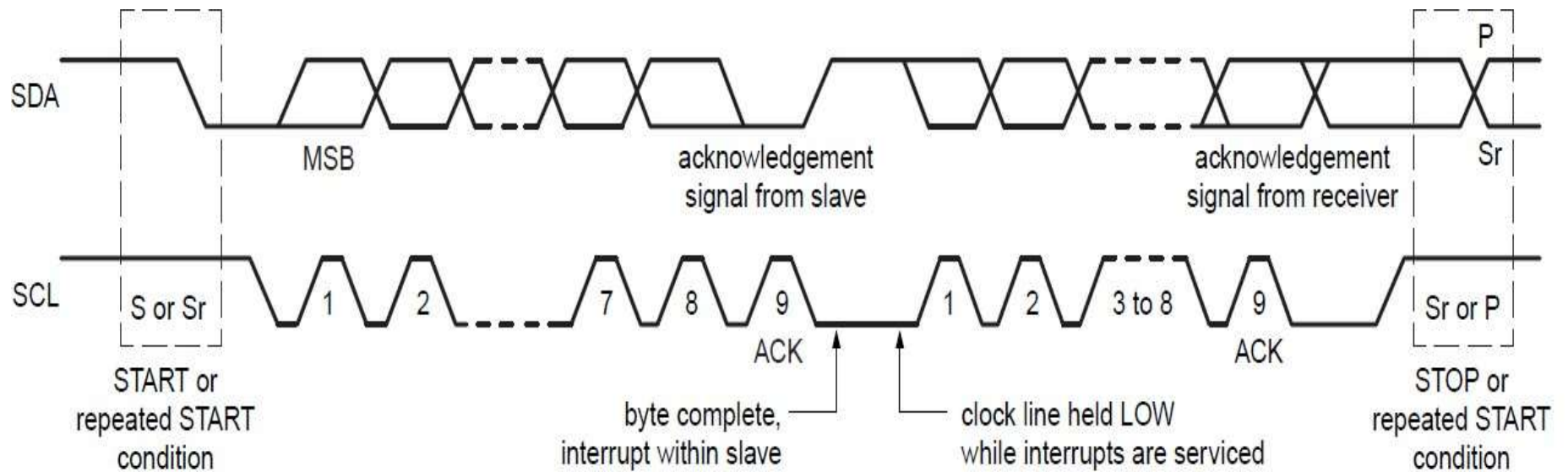
- A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.
- A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.
- START and STOP conditions are always generated by the master.



mba608

Byte Format

- Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted.
- Each byte must be followed by an Acknowledge bit.
- Data is transferred with the Most Significant Bit (MSB) first.



Acknowledge (ACK) and Not Acknowledge (NACK)

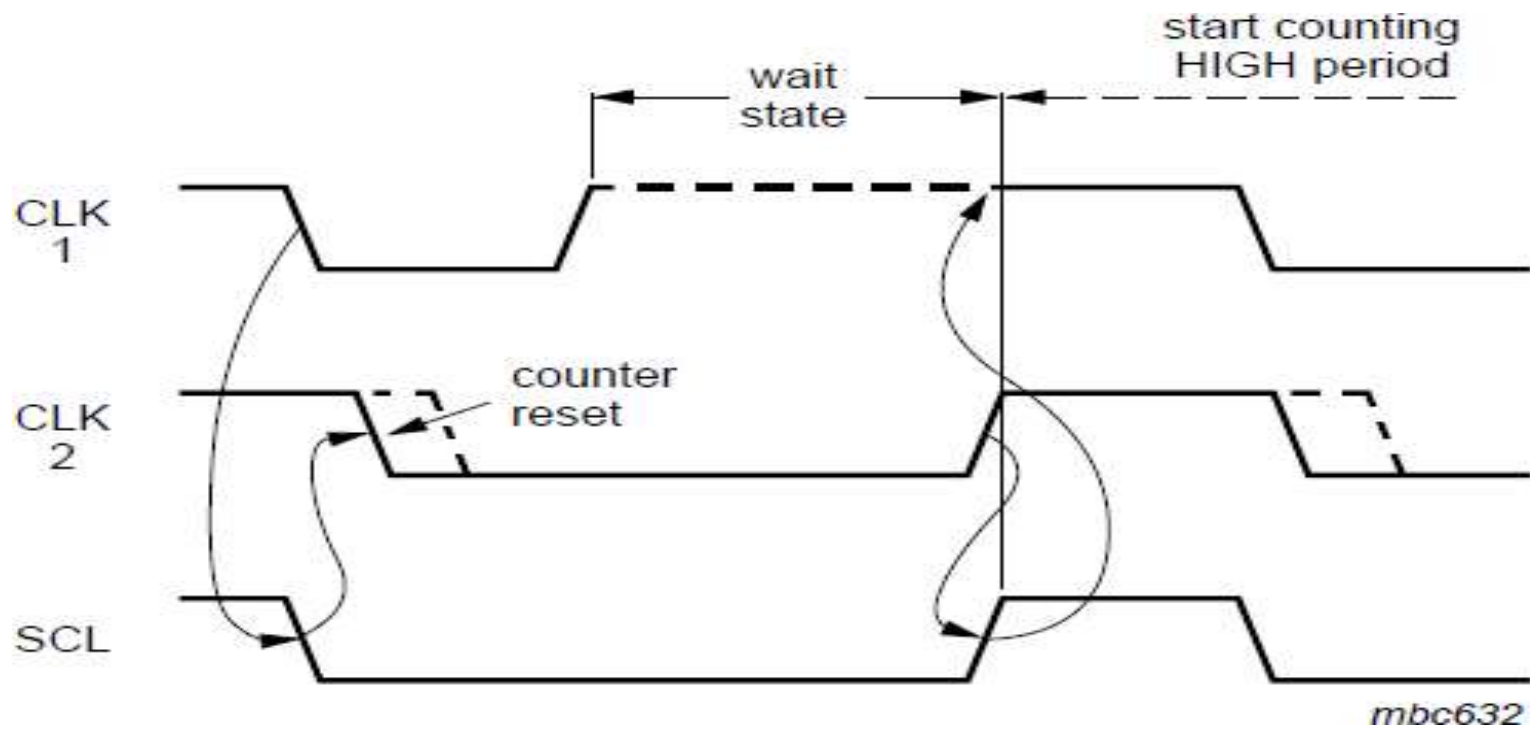
- ACK Defines: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse.
- NACK Defines: When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal.

Acknowledge (ACK) and Not Acknowledge (NACK)

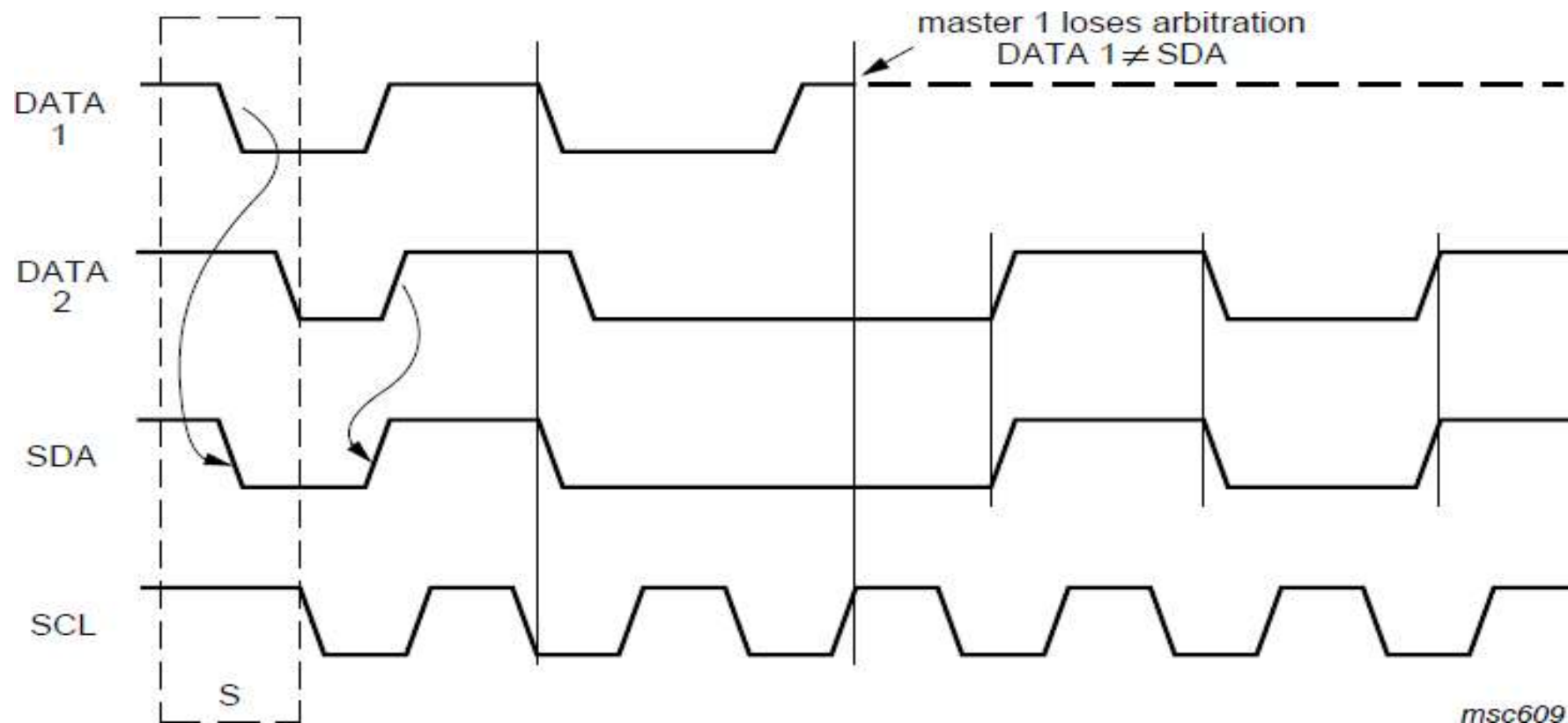
There are five conditions to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

Clock Synchronization



Arbitration



I2C FAQ

- **Q. What happens if we omit the pullup resistors on I2C lines?**

A. There will be no communication on the I2C bus. At all. The MCU will not be able to generate the I2C start condition. The MCU will not be able to transmit the I2C address.

- **Q. The lack of pullups is likely to damage any of those two ICs in my board?**

A. Even without the internal pull-ups, a lack of any pull-ups will not damage either IC. The internal build of i2c device SCL and SDA lines are like NPN transistors. They are Open Collectors, essentially current controlled/switched diodes.

I2C FAQ

Why need pull up resistors in I2C?

- Generally you will need to have the pullup resistors for an I2C interface circuit. If the interface is truly a full spec I2C on both ends of the wires then the signal lines without the resistors will never be able to go to the high level.
- They may remain low or go to some intermediate level determined by the leakage current in the parts at each end.
- The reason for this is because true I2C is an open drain bus.
- I2C is a TTL-logic protocol; so your data and clock lines are open-drain. In other words, the I2C hardware can only drive these lines low; they are left floating when not a zero. That's where the pull-up resistors come in.
- Some devices may actually have on-chip pullup resistors in the 20K to 100K ohm range just to hold the interface pins at a high inactive level when the I2C interface on the part is not in use. For simple and short interfaces these pullup resistors may be just enough to provide the current needed to pull the lines high while clocks and/or data is being signaled.
- i2c pull up resistors allows for features like concurrent operation of more than one I2C master (if they are multi-master capable) or stretching (slaves can slow down communication by holding down SCL).

I2C Interrupts

Bus Free interrupt (BF) is generated to inform the Local Host that the I2C bus became free (when a Stop Condition is detected on the bus) and the module can initiate his own I2C transaction.

Start Condition interrupt (STC) is generated after the module being in idle mode have detected (synchronously or asynchronously) a possible Start Condition on the bus (signalized with WakeUp).

Arbitration lost interrupt (AL) is generated when the I2C arbitration procedure is lost.

No-acknowledge interrupt (NACK) is generated when the master I2C does not receive acknowledge from the receiver.

Registers-ready-for-access interrupt (ARDY) is generated by the I2C when the previously programmed address, data, and command have been performed and the status bits have been updated.

This interrupt is used to let the CPU know that the I2C registers are ready for access.

I2C Interrupts

Receive interrupt/status (RRDY) is generated when there is received data ready to be read by the CPU from the I2C_DATA register (see the FIFO Management subsection for a complete description of required conditions for interrupt generation). The CPU can alternatively poll this bit to read the received data from the I2C_DATA register.

Transmit interrupt/status (XRDY) is generated when the CPU needs to put more data in the I2C_DATA register after the transmitted data has been shifted out on the SDA pin (see the FIFO Management subsection for a complete description of required conditions for interrupt generation). The CPU can alternatively poll this bit to write the next transmitted data into the I2C_DATA register.

Receive draining interrupt (RDR) is generated when the transfer length is not a multiple of threshold value, to inform the CPU that it can read the amount of data left to be transferred and to enable the draining mechanism.

Transmit draining interrupt (XDR) is generated when the transfer length is not a multiple of threshold value, to inform the CPU that it can read the amount of data left to be written and to enable the draining mechanism.

How to Program I2C

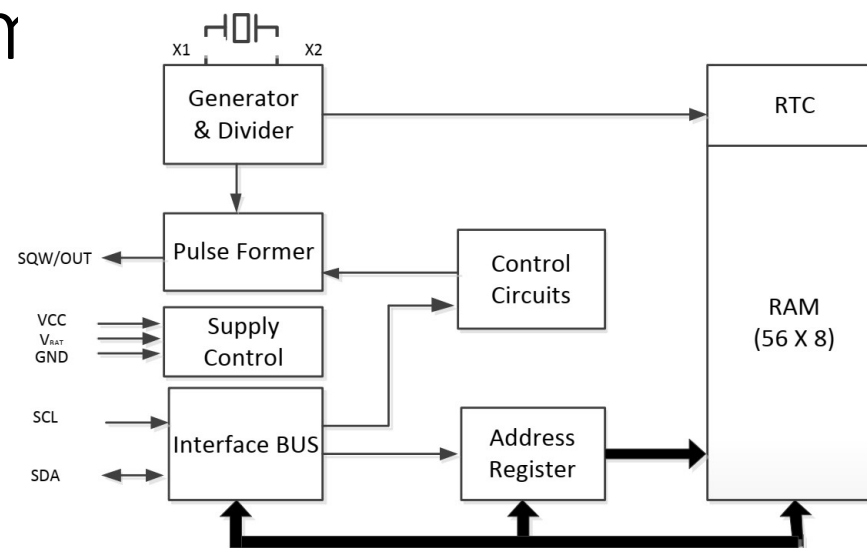
- Module Configuration Before Enabling the Module
- Initialization Procedure
- Configure Slave Address and DATA Counter Registers
- Initiate a Transfer
- Receive Data
- Transmit Data

I2C Slave Device
DS1307 RTC

DS1307 Features/Specifications

- Count of seconds, minutes, hours, week days, date, months and years with consideration of the leap years (before 2100)
- 56 bytes of the power self-sufficient RAM for the data storage;
- Two-wire consecutive interface;
- Programmable rectangular output signal;
- Automatic determination of the supply voltage drop and the switching diagram;
- Consumption of less than 500 nA in the back-up supply mode with the operating generator;
- Temperature range of the industrial application: -40degreeCent to – +85degreecent
- Accuracy is better than ± 1 minute per month

RTC Functional Block Diagram



RTC Pin Description

PIN DISCRIPTION

Pin	Symbol	I/O	Pin Description
1	X1	In	Pin for connection of the quartz resonator
2	X2	In	Pin for connection of the quartz resonator
3	VBAT	In	Pin for battery
4	GND	In	Ground pin
5	SDA	Bi	Input / output of serial data
6	SCL	In	Input of the consecutive cycle signal
7	SQW/OUT	Out	Output of rectangular signal
8	VCC	In	Power supply pin

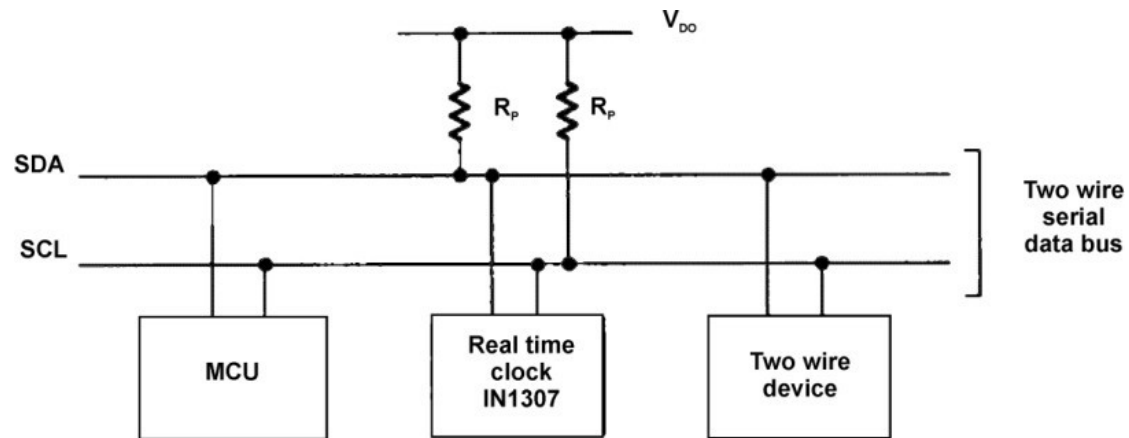
RTC Register Programming Model

REGISTERS RTC IN1307

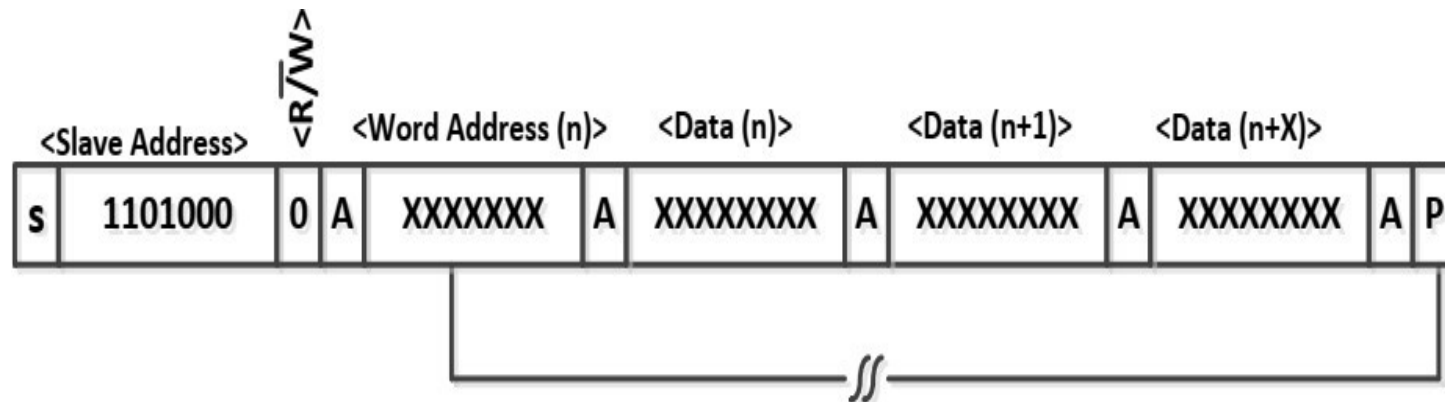
	BIT7							BIT0	
00H	CH	2nd DIGIT of SECONDS			1st DIGIT of SECONDS				00-59
	X	2nd DIGIT of MINUTES			1st DIGIT of MINUTES				00-59
	X	12 / 24	2nd DIGIT of HOURS A/P	2nd DIGIT of HOURS	1st DIGIT of HOURS				01-12 00-23
	X	X	X	X	X	DAY of WEEK			1-7
	X	X	2nd DIGIT of DATE		1st DIGIT of DATE				01-28/29 01-30 01-31
	X	X	X	2nd DIGIT of MONTH	1st DIGIT of MONTH				01-12
	2nd DIGIT of YEARS				1st DIGIT of YEARS				00-99
07H	OUT	X	X	SQWE	X	X	RS1	RS0	

RTC Interface

- I2C Comm. Protocol



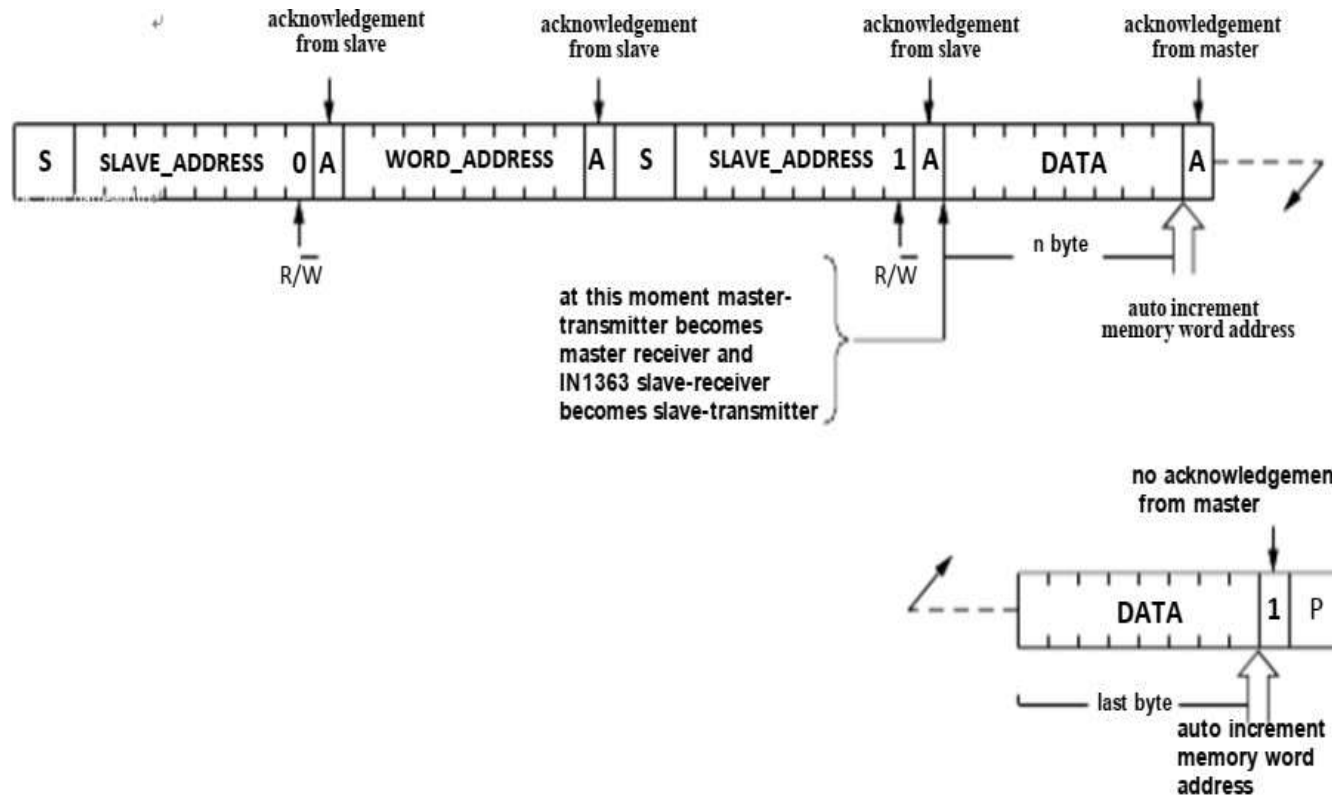
RTC Write communication protocol



S - START
A - ACKNOWLEDGE
P - STOP

* $\overline{R/W}$ - READ/WRITE OR DIRECTION BIT ADDRESS = D0h

RTC Read communication protocol



Master reads after setting word address (write word address ; read data)

I²C in Linux

- Linux I²C framework has 3 layers:
 - I²C core
 - I²C bus (adapter) drivers
 - I²C client (device) drivers
- Optionally exposes /dev/i2c-X devices to userspace.

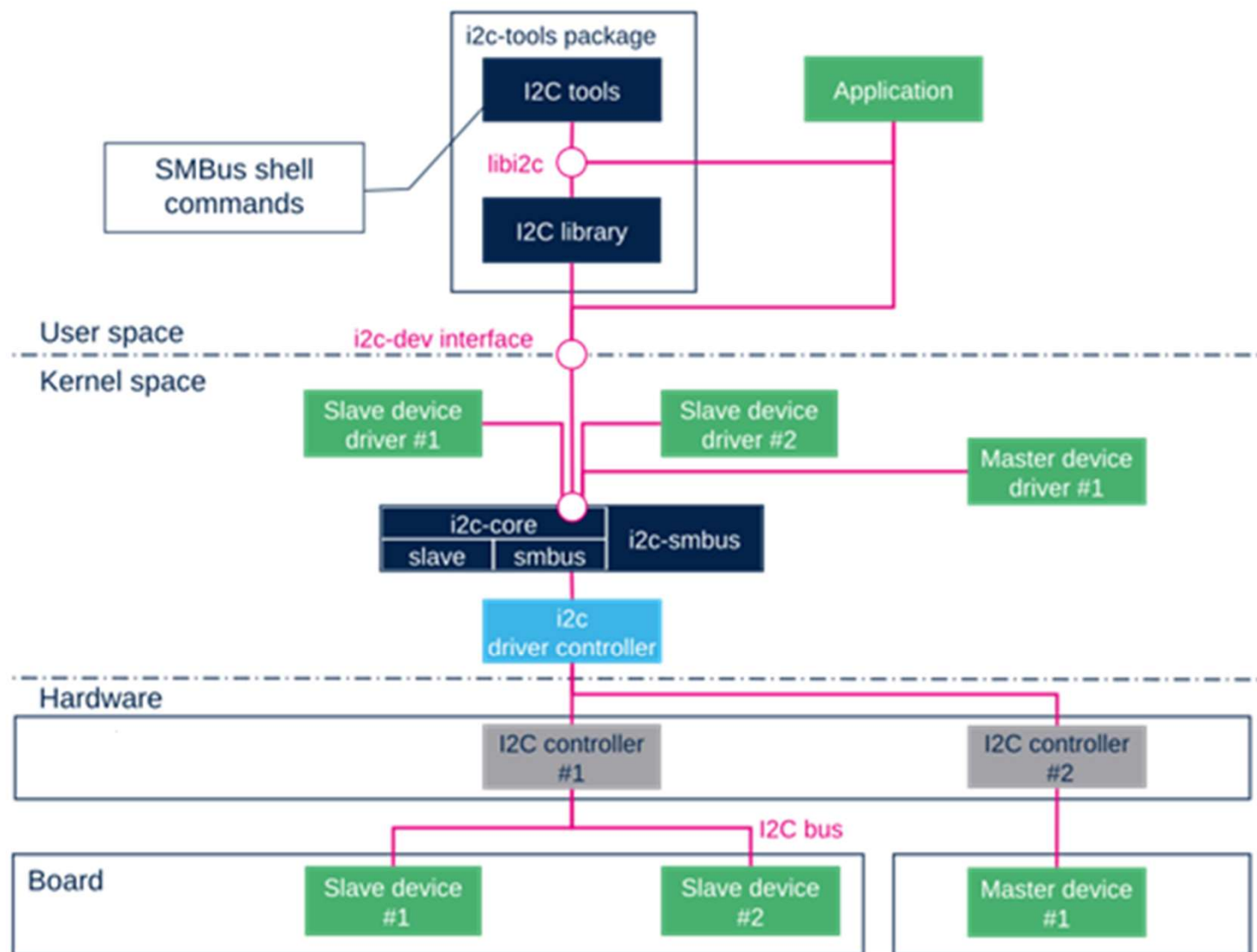
I²C Bus (Adapter) Drivers

- Implement SoC I²C controller logic.
- Register 'i2c_adapter' with core.
- Provide algorithm callbacks:
 - master_xfer()
 - smbus_xfer()
 - functionality()
- Example: Telechips i2c-tcc-v3.c.

I²C Client (Device) Drivers

- Represent I²C slave devices on the bus.
- Declared in DTS under a bus node.
- Driver matches via 'compatible' property.
- Use APIs: `i2c_transfer()`, `i2c_smbus_read/write()`.
- Example: Atmel maXTouch touchscreen driver.

I2C Drivers Architecture



I2C Device Tree

```

&i2c0 {
    #address-cells = <1>;
    #size-cells = <0>;
    status = "okay";
    port-mux = <6>;
    pinctrl-names = "default";
    pinctrl-0 = <&i2c6_bus>;

    /* pmic da9131-D0 */
    da9131_D0: pmic@68 {
        compatible = "dlg,da9131";
        reg = <0x68>;

        ....

        ....

mxt_touch@4b {
    compatible = "atmel,maxtouch";
    /*status = "okay";*/
    status = "okay";
    reg = <0x4b>;
    pinctrl-names = "default";
    pinctrl-0 = <&tsc0_default>;

    interrupt-parent = <&gpsd1>;
    interrupts = <9 IRQ_TYPE_EDGE_FALLING>;
};
};

```

```

i2c6_bus: i2c6_bus {
    telechips,pins = "gpa-22", "gpa-23"; // DISPO_TSC_SCL (A22), SDA (A23)
    telechips,pin-function = <8>;      // mux = I2C function
    telechips,input-buffer-enable;
    telechips,no-pull;
};

tsc0_default: tsc0_default {
    telechips,pins = "gpisd1-9", "gpb-16"; // IRQ (SD1_09), Reset (B16)
    telechips,pin-function = <0>;      // function = GPIO
    bias-pull-up;
    input-enable;
};

```

13	GPIO_A23	DISPO_TSC_SDA	◄►	Serial Data for Touch IC
14	GPIO_SD1_09	DISPO_TSC_IRQ	◄	Touch screen Interrupt Request
15	GPIO_A22	DISPO_TSC_SCL	►	Serial Clock for Touch IC
16	GPIO_B16	DISPO_TSC_RST#	►	Touch screen Reset

I2C Driver Summary

- Linux I²C = core + bus drivers + client drivers.
- **DTS** - describes i2c controllers and devices.
- **Core** - creates clients from DTS nodes.
- **Bus drivers** - handle HW registers.
- **Client drivers** - focus on device-specific logic.

Practice -

- Walkthrough Telechip I2C Driver
- Understand each parameters of I2C DTS files

Topic 5 :

Partition Information for Telechip

Agenda :

- Understand Partition Information of SD Data for Telechip EVB
- Partition List File
- UFS Flashing Steps
- eMMC Flashing Steps

NOTE :

- SD card support is not available on Telechip EVK.

Partition Information of SD Data for TCC805x EVK

FWDN Partition	Size (KB)	Label Name	Block Device	Description	File Name
Partition 1	2048	bl3_ca72_a	mmcblk0p1	Main Core Boot Loader A	ca72_bl3.rom
Partition 2	2048	bl3_ca72_b	mmcblk0p2	Main Core Boot Loader B	ca72_bl3.rom
Partition 3	2048	bl3_ca53_a	mmcblk0p3	Sub Core Boot Loader A	ca53_bl3.rom
Partition 4	2048	bl3_ca53_b	mmcblk0p4	Sub Core Boot Loader B	ca53_bl3.rom
Partition 5	36864	boot_a	mmcblk0p5	Kernel image A	boot.img
Partition 6	36864	boot_b	mmcblk0p6	Kernel image B	boot.img
Partition 7	4124672	super	mmcblk0p7	Resizable image (system.img + vendor.img)	super.img
Partition 8	8192	dtbo_a	mmcblk0p8	Device tree Overlay A	dtbo.img
Partition 9	8192	dtbo_b	mmcblk0p9	Device tree Overlay B	dtbo.img
Partition 10	153600	cache	mmcblk0p10	Cache partition	cache.img
Partition 11	1024	env	mmcblk0p11	Environment	-
Partition 12	5120	splash	mmcblk0p12	splash partition	-
Partition 13	1024	misc	mmcblk0p13	misc partition	-
Partition 14	1024	subcore_misc	mmcblk0p14	Sub-core misc partition	-
Partition 15	1024	tcc	mmcblk0p15	tcc partition	-
Partition 16	8192	sest	mmcblk0p16	sest partition	-
Partition 17	40960	subcore_boot_a	mmcblk0p17	Sub-core Kernel Image	tc-boot-tcc8050-sub.img
Partition 18	40960	subcore_boot_b	mmcblk0p18	Sub-core Kernel Image	tc-boot-tcc8050-sub.img
Partition 19	409600	subcore_root_a	mmcblk0p19	Sub-core Device tree	telechips-subcore-image-tcc8050-sub.ext4
Partition 20	409600	subcore_root_b	mmcblk0p20	Sub-core Device tree	telechips-subcore-image-tcc8050-sub.ext4
Partition 21	1024	subcore_dtb_a	mmcblk0p21	Sub-core root file system	tcc8050-linux-subcore_sv1.0-tcc8050-sub.dtb
Partition 22	1024	subcore_dtb_b	mmcblk0p22	Sub-core root file system	tcc8050-linux-subcore_sv1.0-tcc8050-sub.dtb
Partition 23	1024	vbmeta_a	mmcblk0p23	Hashtree used for Android Verified Boot (AVB)	vbmeta.img
Partition 24	1024	vbmeta_b	mmcblk0p24	Hashtree used for Android Verified Boot (AVB)	vbmeta.img
Partition 25	8192	tamper_evidence	mmcblk0p25	tamper_evidence partition	-
Partition 26	16384	metadata	mmcblk0p26	metadata partition	-
Partition 27	1000	subcore_env	mmcblk0p27	Sub-core Environment	-
Partition 28	40000	subcore_splash	mmcblk0p28	Sub-core splash partition	splash_1920x720x32.img
Partition 29	0	userdata	mmcblk0p29	User storage	-

GPT Partition List File Example -

```
bl3_ca72_a:2048k@../../../../ca72_bl3.rom
bl3_ca72_b:2048k@../../../../ca72_bl3.rom
bl3_ca53_a:2048k@../../../../ca53_bl3.rom
bl3_ca53_b:2048k@../../../../ca53_bl3.rom
boot_a:36864k@../../../../out/target/product/car_tcc8050_arm64/boot.img
boot_b:36864k@../../../../out/target/product/car_tcc8050_arm64/boot.img
super:4124672k@../../../../out/target/product/car_tcc8050_arm64/super.img
dtbo_a:8192k@../../../../out/target/product/car_tcc8050_arm64/dtbo.img
dtbo_b:8192k@../../../../out/target/product/car_tcc8050_arm64/dtbo.img
cache:153600k@../../../../out/target/product/car_tcc8050_arm64/cache.img
env:1024k@
splash:5120k@
misc:1024k@
subcore_misc:1024k@
tcc:1024k@
sest:8192k@
subcore_boot_a:40960k@../../../../out/target/product/car_tcc8050_arm64/tc-boot-tcc8050-sub.img
subcore_boot_b:40960k@../../../../out/target/product/car_tcc8050_arm64/tc-boot-tcc8050-sub.img
subcore_root_a:409600k@../../../../out/target/product/car_tcc8050_arm64/telechips-subcore-image-tcc8050-sub.ext4
subcore_root_b:409600k@../../../../out/target/product/car_tcc8050_arm64/telechips-subcore-image-tcc8050-sub.ext4
subcore_dtb_a:1024k@../../../../out/target/product/car_tcc8050_arm64/tcc8050-linux-subcore_sv1.0-tcc8050-sub.dtb
subcore_dtb_b:1024k@../../../../out/target/product/car_tcc8050_arm64/tcc8050-linux-subcore_sv1.0-tcc8050-sub.dtb
vbmeta_a:1024k@../../../../out/target/product/car_tcc8050_arm64/vbmeta.img
vbmeta_b:1024k@../../../../out/target/product/car_tcc8050_arm64/vbmeta.img
tamper_evidence:8192k@
metadata:16384k@
subcore_env:1M@
subcore_splash:40M@../../../../drivers/camera/splash/splash_1920x720x32.img
userdata:0k@
```


Topic 6 : [Optional]

Linux Memory Management Overview

Agenda :

- Understand Linux Memory Management Overview
- Buddy Allocator Concepts & Structure
- Slab Allocator Concepts & Structure
- Real-Time Linux and PREEMPT_RT Patch

NOTE :

- If time Permits I will try to cover these topics. .

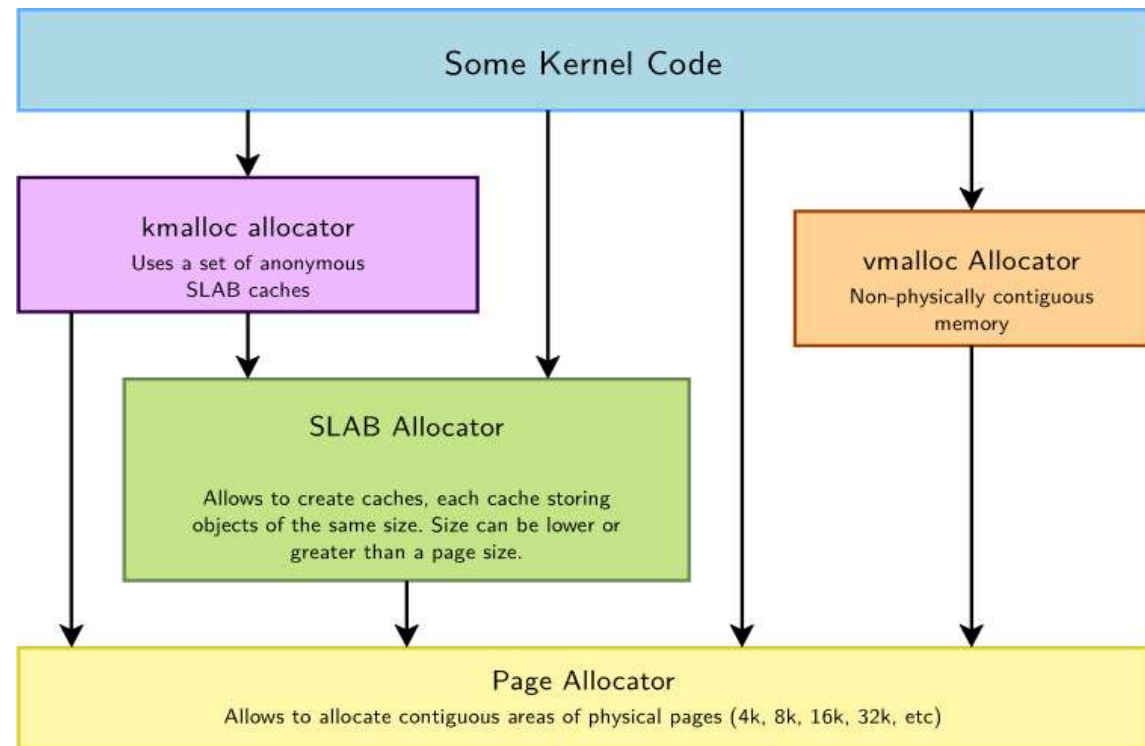
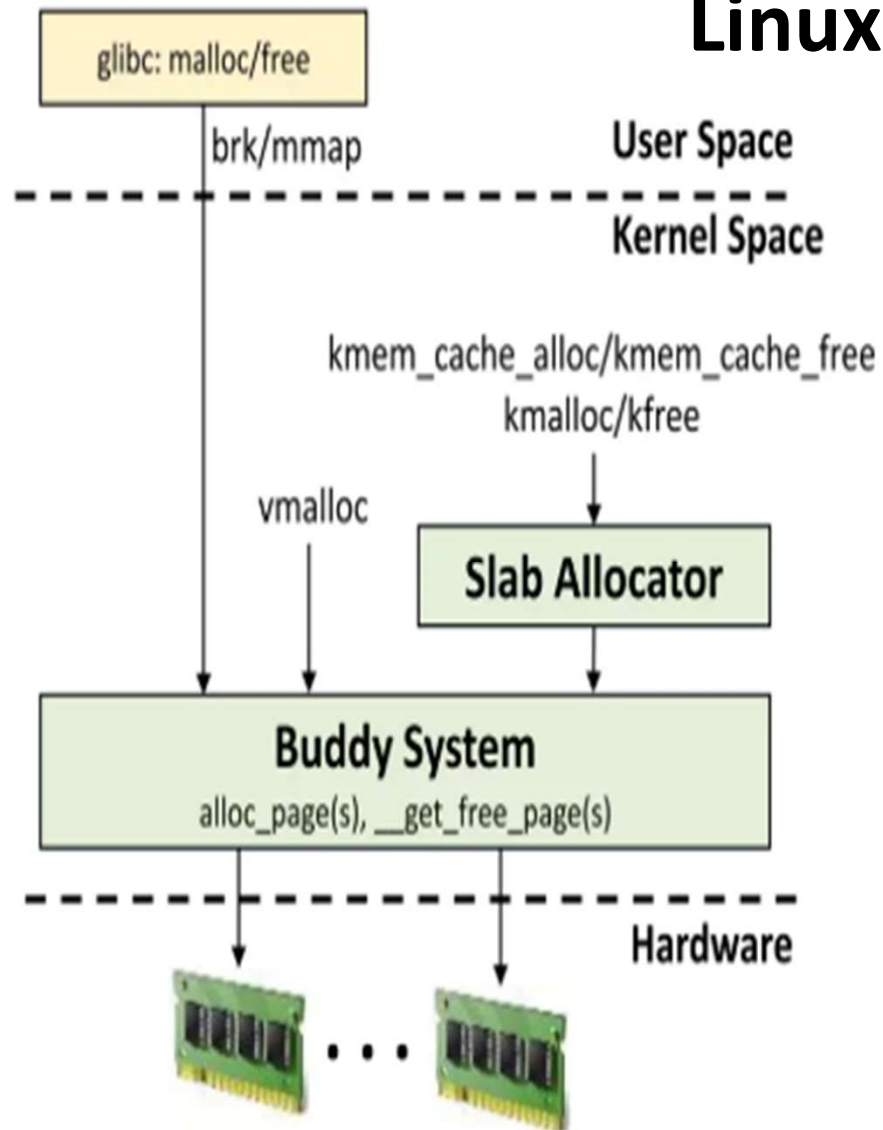
Linux Memory Management ?

- Computer's memory is like a huge, empty field.
- When programs need memory, they ask the operating system (OS) for a "chunk."
- The OS's job is to hand out these chunks quickly and without wasting space.
- The Buddy and Slab allocators are two managers that work together to do this job efficiently.

The kernel offers two mechanisms for allocating memory, both of which are built on top of the kernel's page allocator (i.e. buddy allocator):

- **slab allocator** obtains physically contiguous memory in the kernel's own address space; this allocator is typically accessed via `kmalloc()`,
- **vmalloc()** returns memory in a separate address space; that memory will be virtually contiguous but may be physically scattered.

Linux Memory Management



The Buddy Allocator

Consider Buddy Allocator as a manager who only deals with chunks of memory that are powers of two (e.g., 4KB, 8KB, 16KB, ...). Its entire "field" of memory is one big block.

How it works:

1. It starts with one large, contiguous block of memory (e.g., 512KB).
2. When a request comes in (e.g., for 20KB), it finds the smallest power-of-two block that is big enough (in this case, 32KB).
3. If a 32KB block isn't free, it takes a larger block (e.g., 64KB) and **splits it in half**, creating two "buddies."
4. It gives one buddy to the program and keeps the other for future requests.
5. When the program is done, the buddy manager **merges** the freed block with its "buddy" (if it's also free) to form a larger block again. This prevents fragmentation.

The Buddy Allocator

[1] Request for 20KB comes in.

[2] Start with a big free block:

[_____512KB_____]

[3] Split, until we get a 32KB block:

[_____256KB_____][_____256KB_____]

[__128KB__][__128KB__][_____256KB_____]

[64KB][64KB][__128KB__][_____256KB_____]

[32KB][32KB][64KB][__128KB__][_____256KB_____]

^

|

[4] |-- Allocate this 32KB block for the 20KB request.

Problem with Buddy Allocator

- It can cause **internal fragmentation**.
- Because if Our example itself our program asked for 20KB but got 32KB.
- That extra 12KB is wasted.
- For large requests, this is acceptable, but for thousands of small requests (like for individual data structures), this waste adds up

The Slab Allocator

Slab Allocator as a special memory allocator which can allocate smaller size of memory for very common, small kernel objects such as the dentry, mm_struct, inode, files_struct structures etc

How it works:

- It asks the Buddy Allocator for a few large pages (slabs) of memory.
- It then divides each slab into dozens or hundreds of identical, small chunks (e.g., each exactly the size of a task_struct or an inode kernel object).
- These chunks are pre-initialized (or "constructed") so they are ready to use.
- When the kernel needs a new task_struct, the slab allocator just grabs a free one from its pre initialised pool and give. When it's done, it returns it to the slab, marked as free but still pre-initialized.

The Slab Allocator

- [1] The Slab Allocator gets a 4KB page from the Buddy System.
- [2] It creates a "slab" for "task_struct" objects (each 256 bytes):
- [3] [task_struct | task_struct | task_struct | (16 total)]
- [4] Final Status: [Free | Used | Free | Free | Used |]

Why slab allocator is good :

Zero Waste (No Fragmentation): Every chunk is perfectly sized for the object it holds.

Blazing Fast: Memory is pre-allocated and pre-initialized. Handing out an object is just updating a pointer.

Cache-Friendly: Objects in the same slab are often in the same CPU cache line, making access super fast.

Why We Need Both ?

- The Buddy Allocator is excellent at managing the entire memory space. It efficiently handles large, uncommon requests and provides big, contiguous pages to the Slab Allocator. It fights external fragmentation.
- The Slab Allocator sits on top of the Buddy system. It takes those big pages and optimizes them for the millions of small, frequent requests that the kernel makes every second. It eliminates internal fragmentation for these small objects.

The Final Workflow:

- Suppose kernel needs a new inode object.
- The Slab Allocator checks its "inode slab." If it has a free one, it returns it instantly.
- If the slab is empty, it goes to the Buddy Allocator and ask for a memory for e.g it says, "Give me a new 4KB page."
- It then turns this new page into a fresh slab full of new inode objects.
- It give one inode slab and keeps the rest for future requests.

Thank you