Android BSP

A **Board Support Package (BSP)** is a collection of **hardware-specific drivers, bootloader, kernel, and configuration files** that allow Android to run on a particular hardware platform.
In short, it **bridges Android software with the target hardware**.

It includes:

- **Bootloader** (U-Boot/aboot)
- **Kernel** (Linux kernel for target CPU/SoC)
- **Device Drivers** (I2C, SPI, GPIO, Camera, Display, Audio, Sensors)
- **Device Tree (DTS/DTB)**
- **HAL (Hardware Abstraction Layer)**
- **Board-specific configuration files**
- Sometimes **proprietary binaries** from SoC vendors

---

2. Android BSP Components

Component	Description
Bootloader	First code that runs on the board; initializes CPU, RAM, peripherals, and loads the kernel. Often U-Boot or Qualcomm aboot.
Linux Kernel	Core OS for Android; contains drivers for all hardware (I2C, SPI, USB, Ethernet, etc.)
Device Drivers	Specific drivers for SoC peripherals: camera, touchscreen, display, audio, sensors. Can be built-in or as modules (.ko).
Device Tree (DTS/DTB)	Describes hardware layout to the kernel; includes memory map, GPIO, I2C/SPI buses, clocks, regulators.
HAL (Hardware Abstraction	Android layer that lets applications access hardware in a standardized way.

Component	Description
Layer)	
Board Config Files	BoardConfig.mk, *.h in device/<vendor>/<board>/ that define memory, partitions, bootargs, etc.
Proprietary Libraries/Blobs	Vendor-provided prebuilt binaries (camera, GPU, modem, Wi-Fi).

---

3. Android BSP Directory Structure

Typical AOSP BSP files are under:

```
device/<vendor>/<board>/
   BoardConfig.mk       # Board-level configuration
   device.mk            # Board-specific Android build rules
   kernel/              # Kernel source or reference
   vendor/              # Proprietary binaries
   overlay/             # Device tree overlays if used
   rootdir/             # Root filesystem patches (optional)
```

Kernel is usually in:

kernel/<vendor>/<chipset>/

Bootloader in:

bootloader/<vendor>/<board>/

---

4. Key BSP Tasks

1. **Porting the Kernel**
    - Configure for target SoC (make menuconfig or defconfig)
    - Enable required drivers (I2C, SPI, GPIO, display, touchscreen)
    - Build .img or .ko modules for Android
2. **Device Tree Setup**
    - Create/modify DTS/DTSI files for SoC peripherals

- Set memory map, IRQ, clocks, GPIOs, regulators

3. **Bootloader Customization**
    - Set memory initialization, boot commands, fastboot/USB
    - Provide mechanisms to load kernel + ramdisk + DTB
4. **BoardConfig.mk & Android.mk**
    - Define:
        - BOARD_KERNEL_CMDLINE
        - BOARD_KERNEL_IMAGE
        - BOARD_RAMDISK_IMAGE
        - BOARD_VENDOR
        - BOARD_BOOTIMAGE_PARTITION_SIZE
5. **HAL & Driver Integration**
    - Provide HAL modules for sensors, camera, GPS, audio
    - Integrate vendor prebuilt libraries (.so or .a)
6. **Creating Flashable Images**
    - Combine bootloader + kernel + ramdisk + system
    - Generate images: boot.img, system.img, vendor.img, recovery.img

---

5. BSP Build Flow

1. **Set up environment**

source build/envsetup.sh
lunch <board>-userdebug

2. **Build kernel**

cd kernel/<vendor>/<soc>

```
make ARCH=arm CROSS_COMPILE=<toolchain> <defconfig>
make ARCH=arm CROSS_COMPILE=<toolchain> -j8
```

3. **Build Android image**

```
make -j8
```

4. **Flash images to board**

```
fastboot flash boot boot.img
fastboot flash system system.img
fastboot flash vendor vendor.img
```

---

6. Important Android BSP Terms

Term	Meaning
Bootimage	Contains kernel + ramdisk
Recovery	Special boot image for recovery mode
Ramdisk	Root filesystem for booting
Fastboot	Protocol to flash partitions
Partitions	boot, system, vendor, userdata, cache

---

7. Why BSP is Needed

- Android is written **hardware-agnostic**
- BSP **customizes Android to a board**
- Without BSP:
    - Kernel may not boot
    - Drivers won’t work

        - Hardware peripherals unusable
    - BSP enables:
        - Android OS boot
        - Proper driver operation
        - Vendor-specific hardware support

---

8. Steps to Add/Port a Device Driver in BSP

1. Identify driver type:
    - In-kernel (built-in or module)
    - Out-of-tree (external)
2. Modify kernel .config or Kbuild
3. Add device tree node
4. Build kernel and module
5. Update Android HAL if needed
6. Test functionality

Android Boot Process – Full Flow

There are **6 main stages** in the Android boot process:

Power ON
↓
1️⃣ Boot ROM
↓
2️⃣ Bootloader (Primary & Secondary)
↓
3️⃣ Kernel Initialization
↓
4️⃣ Ramdisk & Init Process
↓

5️⃣ Zygote Process
↓
6️⃣ SystemServer & Android Framework
↓
7️⃣ Launcher (Home Screen)

Now let's understand **each stage in detail** 👇

1️⃣ Boot ROM (SoC internal ROM)

Location: Inside SoC (non-modifiable ROM)

What happens:

- When power is applied, the **CPU executes the Boot ROM code**.
- The Boot ROM:
  - Initializes minimal hardware (clock, stack, memory controller).
  - Loads **Primary Bootloader (PBL)** from a predefined location (e.g., eMMC, SD, NAND, SPI flash).
  - Validates bootloader integrity (secure boot chain).
- Each SoC vendor (Qualcomm, NXP, TI, etc.) implements this part.

🧠 **Key point:** Boot ROM is **hardware-specific and immutable**.

2️⃣ Bootloader Stage

Bootloader prepares the system environment and loads the Linux kernel.

It can have two stages:

- **Stage 1:** SPL (Secondary Program Loader)
- **Stage 2:** U-Boot / Aboot (main bootloader)

Functions:

- Initialize DRAM and peripherals.

- Setup UART (for debug), clocks, and PMIC.
- Load and verify:
    - Kernel (zImage or Image)
    - Ramdisk (initramfs)
    - Device Tree Blob (.dtb)
- Set kernel command line (bootargs)
- Pass control to kernel.

Bootloader in Android (Aboot or U-Boot)

In Android, bootloader also:

- Supports **fastboot** interface.
- Loads images from partitions:
  boot, recovery, system, vendor.
- Supports **verified boot** (AVB): checks hash and signature of partitions.

3️⃣ Kernel Initialization

File: zImage or Image.gz-dtb

Once the kernel is loaded:

- The **Linux kernel decompresses itself**.
- Initializes:
    - **MMU (Memory Management Unit)**
    - **Interrupts (IRQ)**
    - **Scheduler, File systems**
    - **Device drivers** (from BSP)
- Mounts **root filesystem (ramdisk)**.

- Starts the **init process** (PID 1).

🧠 **Device Tree (DTB)**:

- Kernel uses DTB to understand hardware layout: memory, GPIO, I2C, UART, etc.
- Helps kernel load appropriate drivers.

4️⃣ Init Process (Ramdisk Stage)

File: /init (binary inside ramdisk)

After kernel starts, it executes the **init** program.

Role of init:

- Parses configuration files:
  - /init.rc
  - /init.<board>.rc

What is an .rc file in Android?

.rc files are **Init configuration files** that tell the Android **init process** what to do during boot.

They are **plain text scripts** written in a specific **Init language (Android Init Language)**.
Each .rc file defines **services**, **commands**, **triggers**, and **actions** to initialize the system properly.

- Mounts essential file systems (/system, /vendor, /data)
- Starts daemons (e.g. ueventd, servicemanager)
- Sets up SELinux, permissions, properties.
- Starts **Zygote** process (Java world entry point)

Ramdisk = compressed root filesystem inside boot.img.

5️⃣ Zygote Process

Location: /system/bin/app_process

Zygote = "mother of all processes" in Android.

Functions:

- Starts **Dalvik/ART virtual machine**.
- Preloads common Java classes (framework, UI libs).
- Waits for commands from **SystemServer** or apps to fork new processes.

🧠 When you open an app → Zygote **forks a new process**, which becomes the app's runtime process.

6️⃣ SystemServer Process

Started by Zygote.

Responsibilities:

- Initializes core Android services:
    - ActivityManagerService (AMS)
    - WindowManagerService (WMS)
    - PackageManagerService (PMS)
    - PowerManagerService, InputManagerService, etc.
- Starts:
    - **SystemUI**
    - **Launcher**

🧠 SystemServer bridges **Android framework and native layers**.

7️⃣ Launcher (Home Screen)

- After all services start, ActivityManagerService launches the **Home (Launcher) app**.
- The Home screen is the first visible UI to the user.

🎉 At this point, Android is **fully booted**.

🧩 Simplified Layer View

[User Apps]
↑
[Android Framework]
↑
[System Services / SystemServer]
↑
[Zygote & ART]
↑
[Init Process / Daemons]
↑
[Linux Kernel + Drivers (BSP)]
↑
[Bootloader]
↑
[Boot ROM (SoC)]

🔐 Optional: Secure Boot Flow

Some devices use **Verified Boot / AVB**:

1. Boot ROM validates Bootloader (signature).
2. Bootloader validates Kernel + Ramdisk.
3. Kernel validates /system, /vendor, etc.
4. If validation fails → device enters **Recovery Mode**.

📂 Important Boot Files

File	Description
boot.img	Contains kernel + ramdisk
system.img	Android system (framework, libs)
vendor.img	Vendor-specific binaries and HALs
recovery.img	Recovery environment
dtbo.img	Device tree overlay (Android 8+)
vbmeta.img	Verified boot metadata

Excellent — this is a core Android BSP topic 🔧

🚀 Adding a Device Driver in Android BSP — Full Guide

Android BSP =
Bootloader + Kernel + Device Drivers + HAL + Android Framework

When you "add a driver," you're mostly working in the **kernel + device tree layer**, but to make it functional in Android userspace, you might also need **HAL integration**.

🧩 Overview: Where a Driver Fits in Android Stack

[Android Apps]
↓
[Android Framework]
↓
[HAL (Hardware Abstraction Layer)]
↓

[Linux Kernel Device Driver]
↓
[Hardware (I2C, SPI, GPIO, UART, etc.)]

So, to **add a new driver**, you integrate it at the **kernel** level and optionally connect it to the **HAL** if user-space interaction is needed.

⚙️ STEP 1: Identify the Driver Type

There are two categories:

Type	Description	Example
In-Kernel (Built-in)	Comes with kernel source (under drivers/)	I2C, SPI, GPIO, USB, MMC
Out-of-Tree (External)	Custom or vendor-supplied driver (you add it manually)	Custom sensor, camera, display, etc.

📦 STEP 2: Add Driver Source Files

📂 Typical kernel structure:

```
kernel/
└── drivers/
    ├── i2c/
    ├── spi/
    ├── input/
    ├── misc/
    └── mydriver/
        ├── mydriver.c
        └── Kconfig
        └── Makefile
```

If it's **out-of-tree**, create your own folder (e.g., mydriver/).

📜 STEP 3: Modify Makefile and Kconfig

Example Kconfig:

```
config MY_DRIVER
   tristate "My Custom Driver"
   depends on I2C
   help
     Enable support for My Custom I2C device
```

Example Makefile:

```
obj-$(CONFIG_MY_DRIVER) += mydriver.o
```

Then link it in the parent drivers/Makefile and drivers/Kconfig:

```
source "drivers/mydriver/Kconfig"
```

⚡ STEP 4: Enable the Driver in Kernel Config

Run:

```
make menuconfig
```

Then enable your driver:

Device Drivers → My Custom Driver (M)

“M” means module (.ko file), “*” means built-in.

📦 STEP 5: Device Tree (DTS/DTB) Entry

In Android (and Linux), hardware is described via **Device Tree**.

Example:

```
my_sensor@40 {
   compatible = "mycompany,my-sensor";
   reg = <0x40>;
   interrupt-parent = <&gpio1>;
```

```
    interrupts = <5 IRQ_TYPE_LEVEL_LOW>;
    status = "okay";
};
```

📘 This tells the kernel:

- Device is I2C device at address 0x40
- Compatible string to match driver
- Interrupt and GPIO mapping

Then in your driver code (mydriver.c):

```
static const struct of_device_id my_sensor_dt_ids[] = {
    { .compatible = "mycompany,my-sensor", },
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, my_sensor_dt_ids);
```

This allows kernel to **auto-bind** the driver to the hardware.

🔨 STEP 6: Build the Kernel

```
cd kernel/
make ARCH=arm CROSS_COMPILE=<toolchain> <defconfig>
make ARCH=arm CROSS_COMPILE=<toolchain> -j8
```

If built as a module:

```
make modules
```

Output:

```
drivers/mydriver/mydriver.ko
```

🧰 STEP 7: Integrate into Android Build

If you want Android to package the kernel driver:

- Add .ko to:

  device/<vendor>/<board>/BoardConfig.mk

  Example:

  ```
  BOARD_VENDOR_KERNEL_MODULES += \
    $(KERNEL_OUT)/drivers/mydriver/mydriver.ko
  ```

- Or install manually using init.rc:

  ```
  on boot
    insmod /vendor/lib/modules/mydriver.ko
  ```

🧩 STEP 8: HAL / Userspace Integration (If Needed)

If your driver needs to be accessed by Android apps (like camera, sensor, GPS),
you must add a **HAL layer**.

Steps:

1. Create HAL module in:

   hardware/<vendor>/<module>/

2. Provide interface file:

   int my_sensor_read_data(...);

3. Add Android.mk / Android.bp for HAL.
4. Register with Android framework or system service.

For simple drivers (like GPIO, LED), HAL might not be needed — you can access via **sysfs** or **/dev**.

🧠 STEP 9: Verify Device Node and Driver Probe

Boot the board and run:

dmesg | grep mydriver

You should see:

mydriver: probe successful

Check device node:

ls /dev

If your driver creates /dev/mydriver, it's successfully loaded.

🧪 STEP 10: Test Your Driver

You can write test apps or use adb shell:

```
cat /sys/class/mydriver/value
echo 1 > /sys/class/mydriver/enable
```

Or Android app with JNI to access /dev/mydriver.

📂 Example Directory Summary

```
kernel/
 └── drivers/mydriver/
      ├── mydriver.c
      ├── Kconfig
      ├── Makefile
device/vendor/board/
 ├── BoardConfig.mk
 ├── init.board.rc
 ├── fstab.board
hardware/vendor/mydriver/
 ├── Android.bp
 ├── mydriver_hal.cpp
vendor/etc/init/
 ├── mydriver.rc
```

📘 Quick Summary Table

Step	Task	Description
1	Identify Driver Type	In-kernel or external
2	Add Source	Add .c, Kconfig, Makefile
3	Edit Kconfig/Makefile	Enable in build
4	menuconfig	Enable symbol
5	Device Tree	Add DTS node
6	Build	Compile kernel/modules
7	Integrate in Android	Add to BSP build
8	HAL (optional)	Add user-space access
9	Test	Check dmesg, /dev nodes
10	Verify in Android	Access via HAL or sysfs

Advantages:
✔ **Fast bring-up, stable base, hardware abstraction, vendor-tested code.**
Disadvantages:
❌ **Vendor lock-in, closed-source parts, outdated kernels, and upgrade dependency.**

handle BSP portability issues — especially when your BSP (Board Support Package) is for one board, and you want it to work on another **board using the same SoC** but different **PMICs, GPIOs, or peripherals**.

⚙️ Problem Context

You already have:

✅ SoC (same)
❌ Hardware (different: PMIC, GPIOs, sensors)

That means the **kernel and drivers** mostly work, but **Device Tree, power setup, and driver bindings** must be modified.

🧩 1️⃣ Understand the Hardware Difference

Before editing anything, collect this information for the **new board**:

Component	Example
PMIC (Power IC)	PM8998 → PM6150
GPIOs	LED moved from GPIO45 → GPIO23
I2C Bus Address	Sensor from I2C0 @0x40 → I2C1 @0x44
SPI Devices	Different chip select
Clocks/ Regulators	Some rails renamed or voltage different
Interrupts	Different GPIO pins for IRQ
Peripherals	New UART, missing camera, etc.

You can get this from the **board schematics** or **hardware manual**.

🧱 2️⃣ Modify the Device Tree (DTS / DTSI)

The **Device Tree (DT)** describes your hardware to the Linux kernel.

If your BSP was for boardA, and you are now using boardB,
create a new device tree file:

```
arch/arm64/boot/dts/vendor/
  ├── socname-boardA.dts
  ├── socname-boardB.dts   ← create new
```

Step 2.1: Duplicate and Rename

Copy the old DTS:

cp socname-boardA.dts socname-boardB.dts

Step 2.2: Update Compatible String

At the top:

```
/ {
    model = "MyCompany BoardB";
    compatible = "mycompany,boardB", "vendor,soc";
};
```

This identifies your board.

Step 2.3: Fix GPIO and I2C Configurations

If LED GPIO or sensor address changed:

Old:

```
led1 {
    gpios = <&gpio3 12 GPIO_ACTIVE_HIGH>;
};
```

New:

```
led1 {
    gpios = <&gpio2 5 GPIO_ACTIVE_HIGH>;
};
```

Old sensor node:

```
sensor@40 {
    compatible = "vendor,my-sensor";
    reg = <0x40>;
```

```
    interrupt-parent = <&gpio3>;
    interrupts = <12 IRQ_TYPE_LEVEL_LOW>;
};
```

New (different bus & IRQ):

```
sensor@44 {
    compatible = "vendor,my-sensor";
    reg = <0x44>;
    interrupt-parent = <&gpio2>;
    interrupts = <7 IRQ_TYPE_LEVEL_LOW>;
};
```

Step 2.4: Update PMIC or Regulator Nodes

If PMIC changes:

```
pmic: pmic@0 {
    compatible = "qcom,pm6150";
    regulators {
        vddio-supply = <&pm6150_l12>;
    };
};
```

Update the **compatible string** to match the correct PMIC driver.

Step 2.5: Edit Clocks or Power Domains

If clock IDs or names differ:

```
clocks = <&gcc 12>;   // Old
```

change to the new clock line as per datasheet.

🧠 3️⃣ Check “compatible” Strings in Drivers

Each driver in kernel has:

```
static const struct of_device_id mydriver_of_match[] = {
    { .compatible = "vendor,my-sensor", },
    {}
};
```

If your new DTS uses a **different compatible**, update the driver or DTS to match.

🧩 4️⃣ Update BoardConfig / Bootargs if Needed

For Android BSP:

device/vendor/boardB/BoardConfig.mk

Modify:

```
BOARD_KERNEL_CMDLINE := console=ttyMSM0,115200n8 androidboot.hardware=boardB
TARGET_BOARD_DTS := socname-boardB.dts
```

Ensure your Android build points to the new DTB.

⚡ 5️⃣ Build the Kernel and DTB

```
make ARCH=arm64 CROSS_COMPILE=<toolchain> socname-boardB.img -j8
```

Then check:

arch/arm64/boot/dts/vendor/socname-boardB.dtb

Use dtc to inspect:

```
dtc -I dtb -O dts -o boardB.dts boardB.dtb
```

🔧 6️⃣ Modify Init and HAL (If Needed)

Sometimes new hardware → new service in init.rc or new HAL binary.

Example (new sensor):

```
on boot
    insmod /vendor/lib/modules/my_sensor.ko
```

If the HAL path changes, modify:

/vendor/etc/init/<halservice>.rc

or the Android makefile:

```
PRODUCT_COPY_FILES += \
    device/vendor/boardB/init.boardB.rc:root/init.boardB.rc
```

🧩 7️⃣ Test the Ported BSP

1. Boot the new board.
2. Check dmesg:

   ```
   dmesg | grep mydriver
   ```

   ✅ Look for “probe successful”
   ❌ Look for errors like “no such regulator” or “invalid GPIO”.
3. Verify device nodes:

   ```
   ls /dev/
   cat /sys/class/
   ```
4. Test I2C/SPI:

   ```
   i2cdetect -y 1
   ```

🧠 8️⃣ Debug Common Portability Issues

Error	Cause	Fix
probe failed	Wrong compatible or missing DT node	Match compatible strings
no such regulator	PMIC node missing	Update PMIC DTS entries
invalid GPIO	Wrong GPIO bank/index	Check GPIO mapping in TRM
no irq resource	Wrong interrupts	Correct DTS interrupt values
device not found	I2C address or bus mismatch	Fix reg and bus references

---

📘 Summary Flow

Old BSP (BoardA)
↓
Duplicate DTS → boardB.dts
↓
Edit GPIO / I2C / PMIC / compatible
↓
Update BoardConfig.mk and bootargs
↓
Rebuild Kernel + DTB
↓
Flash and test
↓
Fix errors (dmesg / sysfs)

Quick Validation Checklist

Step	What to Check	Command / Location
1	Kernel driver loaded	`adb shell dmesg

Step	What to Check	Command / Location
2	Device node exists	adb shell ls /dev/
3	Sysfs interface	adb shell ls /sys/class/
4	Sensor HAL available	adb shell dumpsys sensorservice
5	Sensor data read	cat /sys/class/... or use test app
6	Event generation	adb shell getevent
7	HAL logging	`adb logcat

Make sure your **driver probe is successful** (dmesg first).

- Confirm **device tree mapping** (compatible, reg, interrupt) matches hardware.
- If HAL is used, check **sensors.list** and logcat messages.