# Training Agenda

- Understand how to use Telechip Dolphine Boards (803 series, 805 series etc..)
- Understand hardware details
- Understand How to port OS on Telechip based hardware
- Android porting
- Linux porting

# C2. Kernel Module Development and Cross Compilation (1/3)

**Duration: 40 hours**

- 5. Linux Kernel Module Development
- - Introduction to Kernel Modules
- - What are kernel modules? In-tree vs. out-of-tree modules.
- - Advantages of using kernel modules.
- - Creating, Compiling, and Loading Kernel Modules
- - Writing a simple kernel module.
- - Using `Makefile` for compiling modules.
- - Loading and unloading modules with `insmod` and `rmmod`.
- - Advanced Kernel Module Concepts
- - Parameter passing to modules.
- - Module dependencies and symbol exporting.
- Project 3: Kernel Module Creation
- - Develop a custom kernel module and load it on Telechips.

**Tools**

**Hardware: Telechips**
**Software: GCC, Make, Linux Kernel Source**

Maneeesh>> Need 2 Days to prepare slides for this topics

**Note:** Telechips as hardware, offered by Faurecia

Duration: 40 hours

# C2. Kernel Module Development and Cross Compilation (2/3)

- 6. Cross Compilation
- - Introduction to Cross-Compilation
- - Difference between native and cross-compilation.
- - Setting up a cross-compilation environment.
- - Setting Up a Cross-Compilation Toolchain
- - Installing and configuring cross-compilers (e.g., GCC for ARM).
- - Configuring toolchain paths and environment variables.
- - 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.

**Tools**

**Hardware: Host Machine, Telechips**
**Software: GCC Toolchain, Cross-compiler**

Maneesh>> Need 2-3 Days to prepare slides for this topics

**Note:** Telechips as hardware, offered by Faurecia

# C2. Kernel Module Development and Cross Compilation(3/3)

Duration: 40 hours

- 7. Device Tree and HW-SW Interface
- - Understanding the Hardware-Software Interface
- - Introduction to device trees and their purpose.
- - Device tree source (DTS) and device tree blob (DTB).
- - Device Tree Concepts, 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.
- Project 4: Device Tree Configuration
- - Modify the device tree to enable and configure specific peripherals on Telechips.

**Tools**

**Hardware: Telechips**
**Software: Device Tree Compiler, U-Boot**

Maneesh>> Need 2-3 Days to prepare slides for this topics

**Note:** Telechips as hardware, offered by Faurecia

# C2 : Kernel Module Development and Cross Compilations

# Kernel Modules

## 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** | Modules that are part of the **Linux kernel source tree** and built with the kernel. |
| **Out-of-Tree Modules** | 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 in 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.

**Practical =>**

- Writing a simple module.
- Passing parameters to a module.
- Exporting and using symbols.
- Using sysfs for module interaction.
- Creating a module with dependencies.

# Embedded Linux

# Cross Compilation

# Embedded Linux Hardware - Architecture

- The Linux kernel and most other architecture-dependent component support a wide range of 32 and 64 bits architectures
  - x86 and x86-64, as found on PC platforms, but also embedded
  - systems (multimedia, industrial)
  - ARM, with hundreds of different SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - Blackfin (DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

# Embedded Linux Hardware - MMU

- Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- Linux is not designed for small microcontrollers.
- Besides the toolchain, the bootloader and the kernel, all other components are generally architecture-independent

# Embedded Linux Hardware – Communication Protocols

- The Linux kernel has support for many common communication busses
- I2C
- SPI
- CAN
- 1-wire
- SDIO
- USB
- And also extensive networking support
- Ethernet, Wi-Fi, Bluetooth, CAN, etc.
- IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
- Firewalling, advanced routing, multicast.

# Software components

- **Cross-compilation toolchain**
  - Compiler that runs on the development machine, but generates code for the target
- **Bootloader**
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- **Linux Kernel**
  - Contains the process and memory management, network stack, device drivers and provides services to userspace applications
- **C library**
  - The interface between the kernel and the userspace applications
- **Libraries and applications**
  - Third-party or in-house

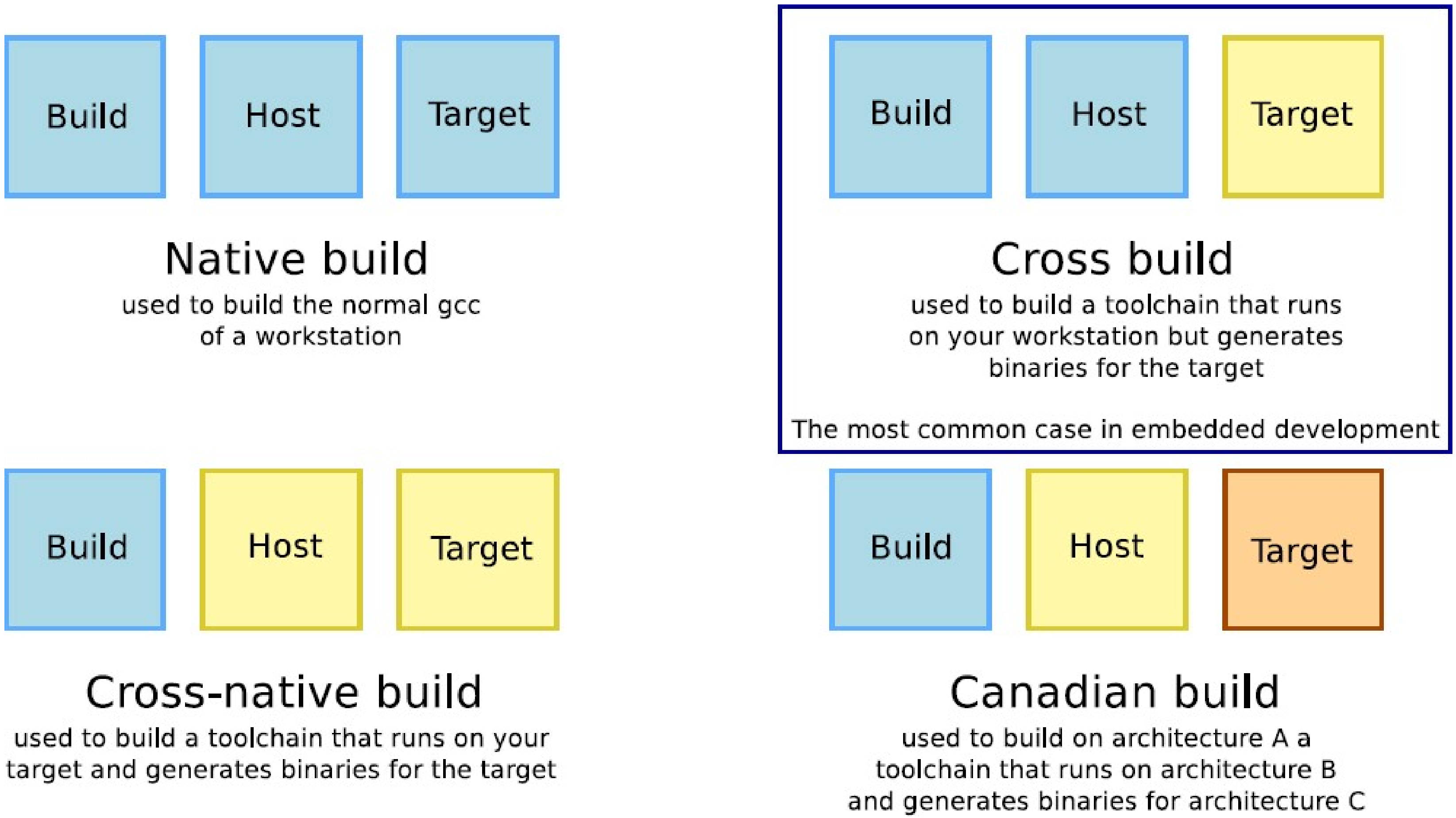
# Cross-compiling toolchains

- The usual development tools available on a GNU/Linux workstation is a native toolchain
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
- The target is too restricted in terms of storage and/or memory
- The target is very slow compared to your workstation
- You may not want to install all development tools on your target.
- Therefore, cross-compiling toolchains are generally used. They run on your workstation but generate code for your target.

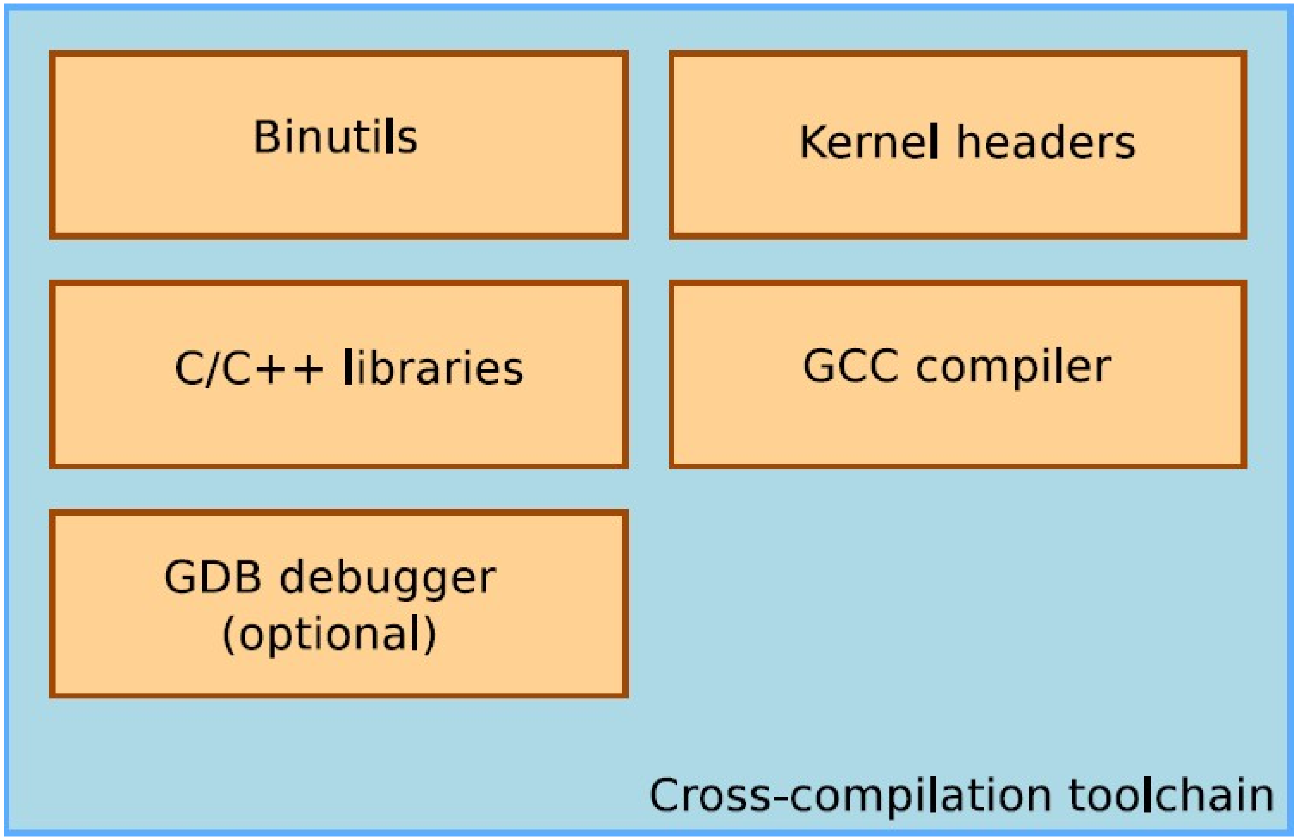
# Machines in build procedures

- Three machines must be distinguished when discussing toolchain creation
- The **build machine**, where the toolchain is built.
- The **host machine**, where the toolchain will be executed.
- The **target machine**, where the binaries created by the toolchain are executed.
- Four common build types are possible for toolchains

# Machines in build procedures



# Components


Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger (optional)
Cross-compilation toolchain

# Binutils

- Binutils is a set of tools to generate and manipulate binaries for a given CPU architecture
  - as, the assembler, that generates binary code from assembler source code
  - ld, the linker
  - ar, ranlib, to generate .a archives, used for libraries
  - objdump, readelf, size, nm, strings, to inspect binaries. Very useful analysis tools!
  - strip, to strip useless parts of binaries in order to reduce their size
- [http://www.gnu.org/software/binutils/](http://www.gnu.org/software/binutils/)
- GPL license

# Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant defnitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in <linux/...> and <asm/...> and a few other directories corresponding to the ones visible in include/ in the kernel sources

# Kernel headers (2)

- System call numbers, in <asm/unistd.h>

```
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
```

- Constant denitions, here in <asm-generic/fcntl.h>, included from <asm/fcntl.h>, included from <linux/fcntl.h>

```
#define O_RDWR 00000002
```

- Data structures, here in <asm/stat.h>

```
struct stat {
unsigned long st_dev;
unsigned  long  st_ino;
[...]
};
```

# Building a toolchain manually

- Building a cross-compiling toolchain by yourself is a difficult and painful task! Can take days or weeks!
  - Lots of details to learn: many components to build, complicated configuration.
  - Lots of decisions to make (such as C library version, ABI, floating point mechanisms, component versions).
  - Need kernel headers and C library sources
  - Need to be familiar with current gcc issues and patches on your platform
  - Useful to be familiar with building and confguring tools
  - See the Crosstool-NG docs/ directory for details on how toolchains are built.

# Installing and using a pre-compiled toolchain

- Method 1:
  - Usually, it is simply a matter of extracting a tarball wherever you want.
  - Then, add the path to toolchain binaries in your PATH:
  - Export  PATH=/path/to/toolchain/bin/:$PATH
- Method 2:
- Install package
  - $ sudo apt-get install gcc-arm-linux-gnueabi

# SoC and Board

# Practical : Discuss Telechip HW, Toolchain, Bootloader and kernel

# Environment variables commands

- U-Boot can be configured through environment variables, which affect the behaviour of the different commands.
- Environment variables are loaded from ash to RAM at U-Boot startup, can be modified and saved back to ash for persistence
- There is a dedicated location in ash to store U-Boot environment, defined in the board configuration file.
- Commands to manipulate environment variables:
  - **printenv**, shows all variables
  - **printenv <variable-name>,** shows the value of one variable
  - **setenv <variable-name> <variable-value>,** changes the value of a variable, only in RAM
  - **saveenv**, saves the current state of the environment to flash

# Important U-Boot env variables

- **bootcmd**, contains the command that U-Boot will automatically execute at boot time after a configurable delay, if the process is not interrupted
- **bootargs**, contains the arguments passed to the Linux kernel, covered later
- **serverip**, the IP address of the server that U-Boot will contact for network related commands
- **ipaddr**, the IP address that U-Boot will use
- **netmask**, the network mask to contact the server
- **ethaddr**, the MAC address, can only be set once
- **bootdelay**, the delay in seconds before which U-Boot runs bootcmd
- **autostart**, if yes, U-Boot starts automatically an image that has been loaded into memory

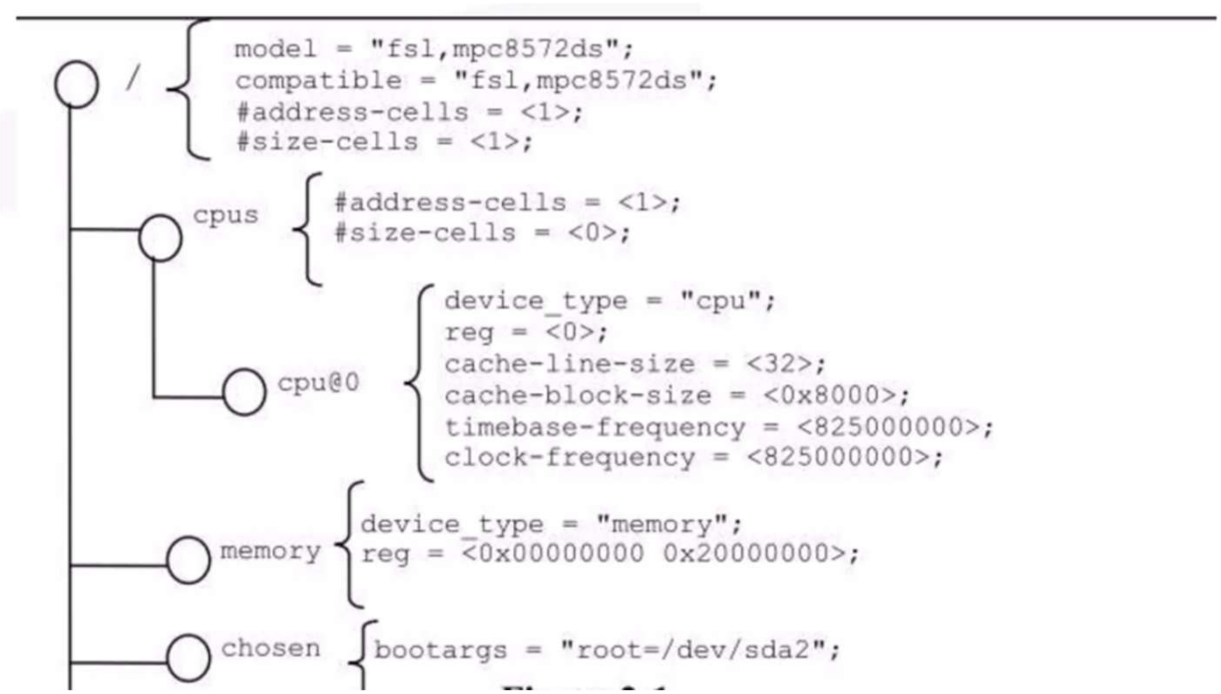
**Practical =>**

- Write kernel Module and flash it on Telechip HW and test
- Compile Bootloader and kernel for Telechip HW

# Device Tree

## 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.

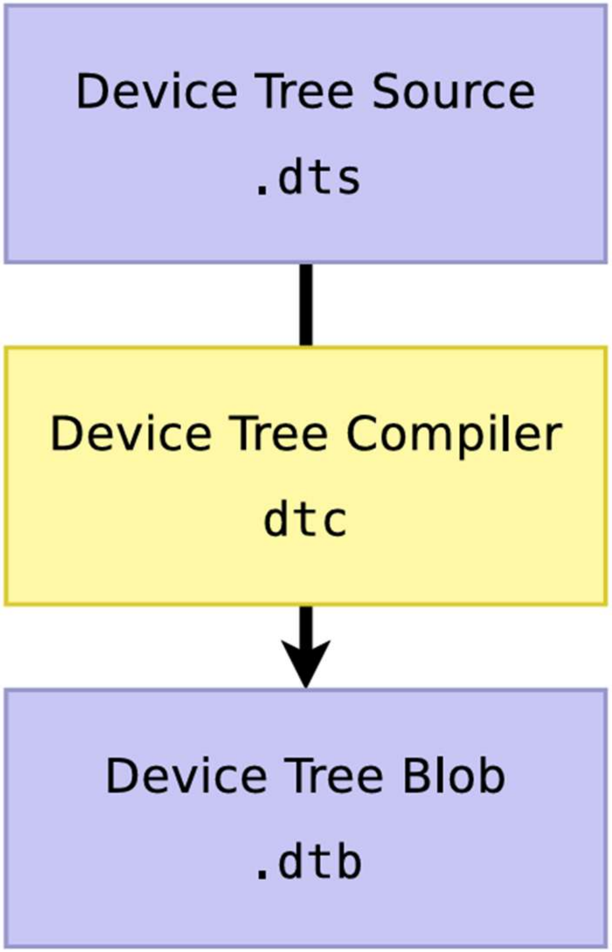


## For each board unique device tree file needed

| 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 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
- Make some customization