四、Linux设备树

前言

视频教程

https://www.bilibili.com/video/BV1ax4y1C7x3/

设备树学习建议

- 因为设备树描述了整个芯片和开发板等所有硬件信息内容，所以他的信息量是非常庞大的，单泰山派linux的设备树算下来大概就有九千多行，大家不要被这个数字给吓到，这些内容都是原厂工程师写的，我们只需要掌握基本语法，能够看懂和对着模板修改就行。也不用想着全局掌握，设备树只是写配置的地方，真正使用他的地方还是在内核中，要弄懂整体框架还要结合后面的驱动来。
- 很多把stm32或者51过来的同学可能感觉莫名其妙学的很浅的感觉，吴工以前学的时候也是和大家一样想每个点甚至每个寄存器都弄的明明白白清清楚楚所以花了非常非常大的力气，Linux整体太庞大了集齐了多少开源大佬的智慧，随便一个小点拎出来都比我们以往写的main复杂，SOC也不同MCU，寄存器以及复杂度都不是一个量级的，如果强行自己全局把控面面俱细，那将会是一种痛苦，我们要做的是先玩起来，改设备树+一点简单的驱动基础+应用就是最快的玩起来方式。随着后面的深入大家会碰到很多问题，跟随着这些问题我们边玩边深入。坚持下去你将会快乐的成为大佬。

为什么学设备树?

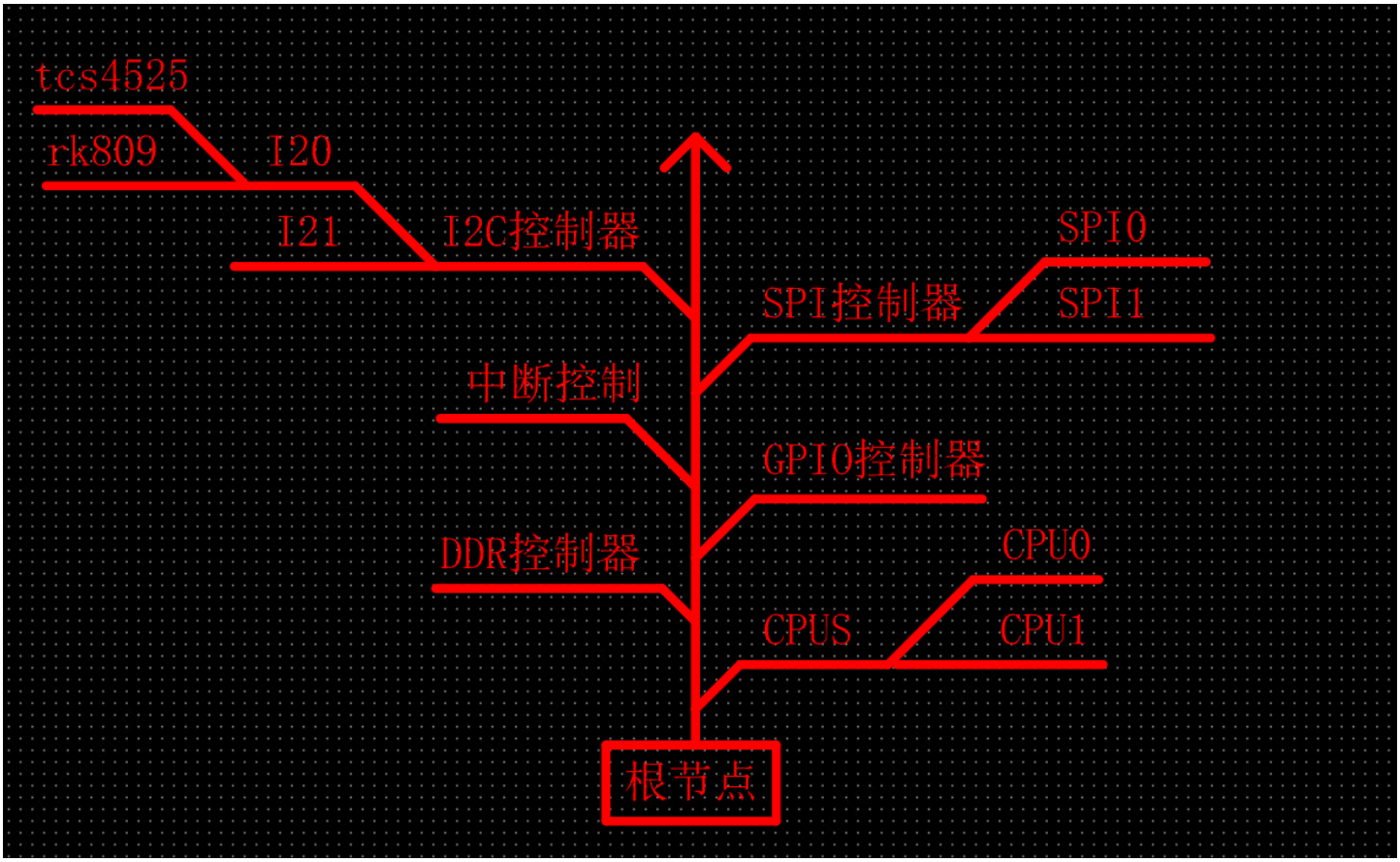
因为设备树非常重要，Linux驱动工程师大家都耳熟的，其实业界对Linux驱动工程师还有一个幽默的称呼:设备树开发工程师，为什么会有设备树开发工程师这个梗呢？因为现在基本都是基于原厂的SDK进行开发，SDK中把难的常见的驱动都帮你搞定了，大家拿到手以后修改的最多的就是设备树了，工作时间久了发现不是在修改设备树就是在修改设备树的途中，所以不言而喻设备树非常重要的。

什么时候学设备树?

吴工学习的时候是先学的裸机-->系统移植-->驱动基础-->掌握了驱动开发再去扩展设备树（当时设备树还没有这么火，很多硬件信息还是通过文件描述的），这个过程太过于漫长了如果毅力不够很容易放弃，这么多年过去了现在大家都是基于原厂SDK开发，原厂SDK把驱动什么的都写好了很多情况下我们只需要修改设备树就能实现很多有趣的定制，我们的目标本来就是先玩起来，玩比什么都重要，后面随着玩法的深入我们再去深入，所以我们直接从设备树开始，掌握泰山派常见外设的修改与调试比如：用户gpio、mipi屏、edp屏、hdmi屏、LED灯、红外遥控器wifi模块的匹配等。

什么是设备树?

通俗的讲设备树就是用于描述硬件信息的一个配件文件，因为他描述的时候的拓扑结构很像树，所以就叫做设备树。详细的讲设备树是一种树状的结构，由节点（Node）和属性（Property）组成。每个节点描述一个硬件设备或资源，节点通过父子关系和兄弟关系进行连接。如下所示以一个根节点开始。根节点可以包含一些全局属性和设备节点。每个设备节点以一个路径标识符（例如/cpu@0）和多个属性（键值对）组成。设备节点可以包含子节点，形成嵌套的层次结构。属性描述了设备的特性，例如设备的名称、类型、寄存器地址、中断信息等。设备节点的路径和属性组合在一起提供了设备树的完整描述，以描绘硬件系统中各个设备的层次结构和配置信息。



常见的设备树术语:

1. **设备树（Device Tree）**：一种用于描述硬件设备和资源连接的数据结构，是一种中立、可移植的设备描述方法。
2. **DTS（设备树源文件 Device Tree Source）**：设备树的源码文件，可以理解成c语言的.c文件。
3. **DTSI （设备树包含文件 Device Tree Include）**：包含了设备树源文件中的可重复使用的部分，可以通过 `#include` 指令在其他设备树源文件中引用。通常用于共享公共的设备树定义和配置，可以理解成c语言的.h文件。
4. **DTC（设备树编译器 Device Tree Compiler）**：用于将设备树源文件（DTS）编译成设备树二进制文件（DTB）的工具，可以理解成c语言的gcc编译器。

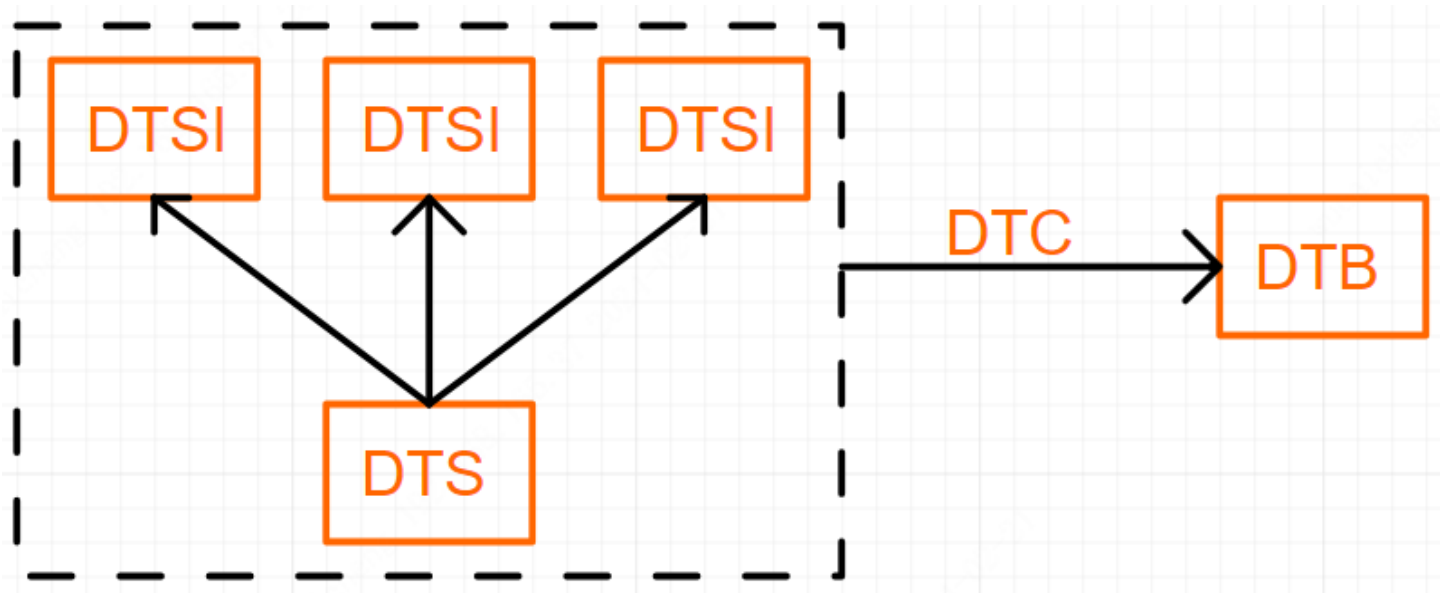
5. **DTB（设备树二进制文件 Device Tree Blob）**：设备树源文件编译生成的二进制文件，可以理解成c语言的.hex或者bin。
6. **节点（Node）**：在设备树中用来描述硬件设备或资源的一个独立部分。每个节点都有一个唯一的路径和一组属性。
7. **属性（Property）**：用于描述节点的特征和配置信息，包括设备的名称、地址、中断号、寄存器配置等。
8. **属性值（Property Value）**：属性中的具体数据，可以是整数、字符串、布尔值等各种类型。
9. **父节点和子节点（Parent Node and Child Node）**：在设备树中，每个节点都可以有一个父节点和多个子节点，用于描述设备之间的连接关系。

学习目标

- 了解设备树的基本结构和语法；
- 大致看懂泰山派设备树文件；
- 知道后面屏幕相关设备树参数如何修改；
- 能基本懂瑞芯微官方设备树相关指导文档，并按照文档就行修改设备树，实现特定功能。

设备树文件结构

我们通常使用`.dts`（设备树源文件）或`.dtsi`（设备树源文件包含文件）来写设备树。编写完成以后通过DTC工具编译生成`.dtb`（设备树二进制）文件，内核在启动时加载这个二进制文件来获得必要的硬件信息。DTS、DTSI、DTC、DTB之间的关系如下图：



泰山派使用的是瑞芯微的主控，我们可以在下面目录找到瑞芯微64位的所有处理器与所有板子相关的设备树其中也包括了泰山派的：

```
1 泰山派SDK/kernel/arch/arm64/boot/dts/rockchip$
```

列出此目录下所有的文件（有删减）：

```
泰山派SDK/kernel/arch/arm64/boot/dts/rockchip$ ls -a
Makefile
rk3326-863-lp3-v10.dts                          rk3566-box-demo-v10-android9.dts
rk3326-863-lp3-v10.dtsi                         rk3566-box-demo-v10.dts
rk3326-863-lp3-v10-rkisp1.dts                   rk3566-box-demo-v10.dtsi
rk3326-86v-v10.dts                              rk3566-box.dtsi
rk3326.dtsi                                     rk3566.dtsi
rk3326-evb-ai-va-v10.dts                        rk3566-eink.dtsi
rk3326-evb-ai-va-v11.dts                        rk3566-evb1-ddr4-v10.dts
rk3326-evb-ai-va-v11-i2s-dmic.dts               rk3566-evb1-ddr4-v10.dtsi
rk3326-evb-ai-va-v12.dts                        rk3566-evb1-ddr4-v10-linux.dts
rk3326-evb-ext-sii902x-rgb-to-hdmi-v10.dtsi     rk3566-evb1-ddr4-v10-lvds.dts
rk3326-evb-lp3-v10-avb.dts                      rk3566-evb2-lp4x-v10.dts
rk3326-evb-lp3-v10.dts                          rk3566-evb2-lp4x-v10.dtsi
rk3326-evb-lp3-v10.dtsi                         rk3566-evb2-lp4x-v10-edp.dts
rk3326-evb-lp3-v10-linux.dts                    rk3566-evb2-lp4x-v10-eink.dts
rk3326-evb-lp3-v10-robot-linux.dts              rk3566-evb2-lp4x-v10-i2s-mic-
array.dts
rk3326-evb-lp3-v10-robot-no-gpu-linux.dts       rk3566-evb2-lp4x-v10-linux.dts
rk3326-evb-lp3-v11-avb.dts                      rk3566-evb2-lp4x-v10-pdm-mic-
array.dts
rk3326-evb-lp3-v11.dts                          rk3566-evb3-ddr3-v10.dts
rk3326-evb-lp3-v12-linux.dts                    rk3566-evb3-ddr3-v10.dtsi
rk3326-linux.dtsi                               rk3566-evb3-ddr3-v10-linux.dts
rk3328-android.dtsi                             rk3566-evb5-lp4x-v10.dts
rk3328-box-liantong-avb.dts                     rk3566-evb5-lp4x-v10.dtsi
rk3328-box-liantong.dts                         rk3566-evb.dtsi
rk3328-box-liantong.dtsi                        rk3566-evb-mipitest-v10.dts
rk3328-box-plus-dram-timing.dtsi                rk3566-evb-mipitest-v10.dtsi
rk3328-dram-2layer-timing.dtsi                  rk3566-rk817-eink.dts
rk3328-dram-default-timing.dtsi                 rk3566-rk817-eink-w103.dts
rk3328.dtsi                                     rk3566-rk817-eink-w6.dts
rk3328-evb-android-avb.dts                      rk3566-rk817-tablet.dts
rk3328-evb-android.dts                          rk3566-rk817-tablet-k108.dts
rk3328-evb-android.dtsi                         rk3566-rk817-tablet-rkg11.dts
rk3328-evb.dts                                  rk3566-rk817-tablet-v10.dts
rk3328-roc-cc.dts                               rk3568-amp.dtsi
rk3328-rock64-android-avb.dts                   rk3568-android9.dtsi
rk3328-rock64-android.dts                       rk3568-android.dtsi
rk3328-rock64-android.dtsi                      rk3568-dram-default-timing.dtsi
rk3328-rock64.dts                               rk3568.dtsi
rk3358.dtsi                                     rk3568-evb1-ddr4-v10-android9.dts
```

```
rk3358-evb-ddr3.dtsi                          rk3568-evb1-ddr4-v10.dts
rk3358-evb-ddr3-v10-linux.dts                 rk3568-evb1-ddr4-v10.dtsi
rk3358-linux.dtsi                             rk3568-evb1-ddr4-v10.dtsi3
rk3358m-vehicle-ddr3.dtsi                     rk3568-evb1-ddr4-v10-linux.dts
rk3358m-vehicle-v10.dts                       rk3568-evb1-ddr4-v10-linux-spi-
nor.dts
rk3368-808.dtsi                               rk3568-evb2-lp4x-v10-bt1120-to-
hdmi.dts
rk3368-808-evb.dts                            rk3568-evb2-lp4x-v10.dts
rk3368a-817-tablet-bnd.dts                    rk3568-evb2-lp4x-v10.dtsi
rk3368a-817-tablet.dts                        rk3568-evb4-lp3-v10.dts
rk3368-android.dtsi                           rk3568-evb5-ddr4-v10.dts
rk3368-cif-sensor.dtsi                        rk3568-evb5-ddr4-v10.dtsi
rk3368-dram-default-timing.dtsi               rk3568-evb6-ddr3-v10.dts
rk3368.dtsi                                   rk3568-evb6-ddr3-v10.dtsi
rk3368-evb-act8846.dts                        rk3568-evb6-ddr3-v10-linux.dts
rk3368-evb.dtsi                               rk3568-evb6-ddr3-v10-rk628-bt1120-
to-hdmi.dts
rk3368-geekbox.dts                            rk3568-evb6-ddr3-v10-rk628-
rgb2hdmi.dts
rk3368-lion.dtsi                              rk3568-evb6-ddr3-v10-rk630-bt656-
to-cvbs.dts
rk3368-lion-haikou.dts                        rk3568-evb7-ddr4-v10.dts
rk3368-orion-r68-meta.dts                     rk3568-evb.dtsi
rk3368-p9-avb.dts                             rk3568-iotest-ddr3-v10.dts
rk3368-p9.dts                                 rk3568-iotest-ddr3-v10-linux.dts
rk3368-p9.dtsi                                rk3568-linux.dtsi
rk3368-px5-evb-android.dts                    rk3568-nvr-demo-v10.dts
rk3368-px5-evb.dts                            rk3568-nvr-demo-v10.dtsi
rk3368-r88-dcdc.dts                           rk3568-nvr-demo-v10-linux.dts
rk3368-r88.dts                                rk3568-nvr-demo-v10-linux-spi-
nand.dts
rk3368-sheep.dts                              rk3568-nvr-demo-v12.dtsi
rk3368-sheep-lvds.dts                         rk3568-nvr-demo-v12-linux.dts
rk3368-sziauto-rk618.dts                      rk3568-nvr-demo-v12-linux-spi-
nand.dts
rk3368-tablet.dts                             rk3568-nvr.dtsi
rk3368-xikp-avb.dts                           rk3568-nvr-linux.dtsi
rk3368-xikp.dts                               rk3568-pinctrl.dtsi
rk3368-xikp.dtsi                              rk630.dtsi
rk3399-android.dtsi                           rk630-rk3568-ddr3-v10.dts
rk3399-box.dtsi                               rockchip-pinconf.dtsi
rk3399-box-rev1.dts                           tspi-rk3566-core-v10.dtsi
rk3399-box-rev2.dts                           tspi-rk3566-csi-v10.dtsi
rk3399-dram-default-timing.dtsi               tspi-rk3566-dsi-v10.dtsi
rk3399.dtsi                                   tspi-rk3566-edp-v10.dtsi
rk3399-early-opp.dtsi                         tspi-rk3566-gmac1-v10.dtsi
```

```
rk3399-evb-cros.dtsi                          tspi-rk3566-hdmi-v10.dtsi
rk3399-evb.dts                                tspi-rk3566-user-v10-linux.dtb
rk3399-evb.dtsi                               .tspi-rk3566-user-v10-
linux.dtb.cmd
rk3399-evb-ind.dtsi                           .tspi-rk3566-user-v10-
linux.dtb.d.dtc.tmp
rk3399-evb-ind-lpddr4-android-avb.dts         .tspi-rk3566-user-v10-
linux.dtb.d.pre.tmp
rk3399-evb-ind-lpddr4-android.dts             .tspi-rk3566-user-v10-
linux.dtb.dts.tmp
rk3399-evb-ind-lpddr4-linux.dts               .tspi-rk3566-user-v10-
linux.dtb.dts.tmp.domain
rk3399-evb-ind-lpddr4-v13-android-avb.dts     tspi-rk3566-user-v10-linux.dts
rk3399-evb-rev1-android.dts
```

大家不要被上面的文件数量吓到了，上面涵盖了所有瑞芯微处理器和开发板的设备树，按照后续归类下来也就 `dts、dtsi、dtb、Makefile` 四类文件下面我们一一解读

DTS、DTSI、DTB和Makefile。它们各有不同的作用，并且之间存在一定的关系。

- DTS文件是设备树的源文件，以".dts"为后缀。它使用一种特定的语法，用于描述硬件设备及其相关信息，比如设备名称、类型、地址、中断等。DTS文件是人可读的文本文件，可以被处理器编译成DTB文件，它相当于C语言中的.c文件。
- DTSI文件是设备树源文件的包含文件，以".dtsi"为后缀。它用于存储一些常用的设备树片段，这些片段可以在多个DTS文件中重复使用，避免了代码重复。DTSI文件可以在DTS文件中进行包含（类似C语言的#include），以便重用其中定义的设备树片段。
- DTB文件是设备树的二进制文件，以".dtb"为后缀。它是通过将DTS文件编译而来的，使用了特定的编译器工具，比如 `dtc` （Device Tree Compiler）。DTB文件是机器可读的二进制格式，可以被操作系统加载和解析，以用于设备驱动的配置和初始化。它相当于C语言中的.bin文件。
- Makefile：Makefile是一个用于构建和编译项目的脚本文件。在设备树的上下文中，Makefile用于自动化编译和构建DTS/DTSI文件，生成对应的DTB文件。

通过我们之前的单片机C语言编程经验来看我们写代码只用关心 `.c` 和 `.h` 文件，那我们上面说到DTS和DTSI分别类似于C语言的 `.c` 和 `.h` 文件所以我们后面实战也只需要关注这两个后缀的文件

下面列出泰山派相关的所有设备树：

```
tspi-rk3566-core-v10.dtsi  tspi-rk3566-dsi-v10.dtsi  tspi-rk3566-gmac1-
v10.dtsi  tspi-rk3566-user-v10-linux.dtb
tspi-rk3566-csi-v10.dtsi   tspi-rk3566-edp-v10.dtsi  tspi-rk3566-hdmi-v10.dtsi
  tspi-rk3566-user-v10-linux.dts
rk3568-dram-default-timing.dtsi  rk3566.dtsi
rk3568.dtsi
```

rk3568.dtsi：主控相关配置

rk3566.dtsi：包含了rk3568.dtsi头文件

tspi-rk3566-core-v10.dtsi：tspi核心配置层，这里是几乎后期不需要怎么改动

tspi-rk3566-edp-v10.dtsi：edp显示屏幕相关的配置

tspi-rk3566-dsi-v10.dtsi：mipi显示屏幕相关的配置

tspi-rk3566-hdmi-v10.dtsi: hdmi显示屏相关的配置

tspi-rk3566-csi-v10.dtsi：mipi摄像头相关配置

tspi-rk3566-gmac1-v10.dtsi： 网口相关配置

tspi-rk3566-user-v10.dts：用户自定义相关配置他会去包含前面的所有

设备树之间的调用关系：

前面我们讲了dts和dtsi就类比c语言的.c和.h的关系，大家都知道c语言是有调用关系的所以对于我们的设备树也有调用关系，下先从 `tspi-rk3566-user-v10.dts` 文件开始下手可以得出调用关系，这里牵扯到include的用法我们后面深入讲解目前大家就理解成c语言include就行：

```
tspi-rk3566-user-v10.dts
    rk3566.dtsi
        rk3568.dtsi
            rk3568-dram-default-timing.dtsi
    rk3568-android.dtsi
    tspi-rk3566-core-v10.dtsi
    tspi-rk3566-edp-v10.dtsi
    tspi-rk3566-dsi-v10.dtsi
    tspi-rk3566-hdmi-v10.dtsi
    tspi-rk3566-csi-v10.dtsi
    tspi-rk3566-gmac1-v10.dtsi
```

设备树编译器DTC

如果大家全编译过泰山派开发板的SDK固件的话就会发现 `SDK/kernel/arch/arm64/boot/dts/rockchip` 路径下生成了一个 `tspi-rk3566-user-v10-linux.dtb` 文件，这就是我们设备树编译器的功劳，我们全编译的时候首先去编译生成了DTC编译器，然后在用这个编译器去编译我们的设备树，我们可以在 `SDK/kernel/scripts/dtc` 目录下找到我们的dtc编译器源码和源码生产的dtc编译器执行文件。以通过pwd记录一下这个目录的绝对路径，后面我们通过绝对路径使用dtc来编译测试。

当然如果目前还没有下载泰山派的sdk，也可以可以单独安装dtc编译器进行练习

- 首先，更新你的包列表以确保你得到最新版本的软件包：

```
1 sudo apt update
```

- 然后，使用 `apt` 安装 `device-tree-compiler` 软件包：

```
1 sudo apt install device-tree-compiler
```

- 安装完成后，可以用以下命令检查 DTC 的版本来确认它是否正确安装：

```
1 dtc --version
```

编译（从 .dts 和 .dtsi到 .dtb）

将 Device Tree 源文件（.dts）编译为 Device Tree Blob（.dtb）编译命令格式如下：

```
1 dtc -I dts -O dtb -o output_file.dtb input_file.dts
```

- `-I dts` 指定输入文件格式是 Device Tree Source。
- `-O dtb` 指定输出文件格式是 Device Tree Blob。
- `-o output_file.dtb` 设定输出文件的名称。
- `input_file.dts` 是你想编译的 .dts 文件。

反编译（从 .dtb 到 .dts）

将 Device Tree Blob（.dtb）反编译为 Device Tree 源文件（.dts）：

```
dtc -I dtb -O dts -o output_file.dts input_file.dtb
```

- `-I dtb` 指定输入文件格式是 Device Tree Blob。
- `-O dts` 指定输出文件格式是 Device Tree Source。
- `-o output_file.dts` 设定输出文件的名称。
- `input_file.dtb` 是你想反编译的 .dtb 文件。

让我们来实操一下

编译测试

创建一个input_file.dts文件填入内容

```
/dts-v1/;

/ {
};
```

编译命令（注意dtc编译工具目录要改成你自己的）：

```
dtc -I dts -O dtb -o output_file.dtb input_file.dts
```

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ /home/wucaicheng/tspi/linux/kernel/scripts/dtc/dtc -I dts -O dtb -o devicetree.dt
s
```

编译成功：

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ ls
input_file.dts  output_file.dtb
```

反编译

反编译命令（注意dtc编译工具目录要改成你自己的，后面演示的命令我就把路径去掉了，但是大家要知道实用sdk中的工具要带路径访问，自己安装的不用）：

```
dtc -I dtb -O dts -o backoutput_file.dts output_file.dtb
```

反编译成功：

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ /home/wucaicheng/tspi/linux/kernel/scripts/dtc/dtc -I dtb -O dts -o backoutput_fi
le.dts output_file.dtb
```

通过ls查看一下是否生成了backoutput_file.dts文件

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ ls
backoutput_file.dts  input_file.dts  output_file.dtb
```

通过cat查看反编译后的内容，可以发现和之前创建的内容是一样的

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ cat backoutput_file.dts
/dts-v1/;

/ {
};
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/dtc_test$ cat input_file.dts
/dts-v1/;

/ {
};
```

设备树基本语法

前面我们已经知道设备树主要是由节点和属性构成，这里为方便理解可以把设备树类比成旅游景区的姻缘树。

1. **节点（Node）**：
   - 在设备树中，节点代表了硬件中的一个逻辑单元或部件，它可以是一个复杂的组件，比如一个CPU，也可以是简单的对象，如一个I/O端口。
   - 比喻姻缘树，节点可以对应到姻缘树的树根、主干和支干。树根可以看作是整个姻缘树的起点，对应到设备树中的根节点，它代表了整个硬件系统的起始点。主干和支干则分别对应到设备树中的各级子节点，它们代表了各个不同的设备和子系统。
2. **属性（Property）**：
   - 在设备树中，属性是节点的一部分，它们为节点提供了额外的描述信息。
   - 比喻姻缘树，属性可以对应到姻缘绳（即红色祈福牌）。这些祈福牌上写有的信息（如名字、年龄等）就如同设备树中的属性，为游客（即设备和子系统）提供了额外的信息或配置。
   - 祈福牌（属性）可以绑定在姻缘树（设备树）的任何一个树干（节点）上，这意味着不同的设备和子系统可以有不同的属性集，这些属性集描述了它们各自的特性和配置需求。

版本

在设备树文件中通过 `/dts-v1/;` 来指定版本，一般写在dts的第一行，这个声明非常重要的，因为他描述了了设备树文件所使用的语法和规范版本。如果没有这个声明，设备树编译器可能会无法正确解释文件中的语法，导致编译错误或者设备树在加载时出现问题。

```
1 /dts-v1/; // 指定这是一个设备树文件的版本（v1）
```

注释

和C语言一样，有两种方法分别如下：

```
/* 这是一个注释 */
// 这是一个注释
```

头文件

前面我们已经讲了dts是源文件，头文件是dtsi，在设备树中主要有两种方法去包含头文件

方法一：

通过设备树的语法包含头文件

```
/include/ "xxxx.dtsi" //xxxx是你要包含的文件名称
```

试一试:

创建一个名为device_tree.dtsi的头文件，并写入以下内容

```
/dts-v1/;
/{

};
```

再创建一个device_tree.dts名字的源文件，并写入以下内容

```
/dts-v1/;
/include/ "device_tree.dtsi"  //这里去包含上面的头文件
```

编译

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

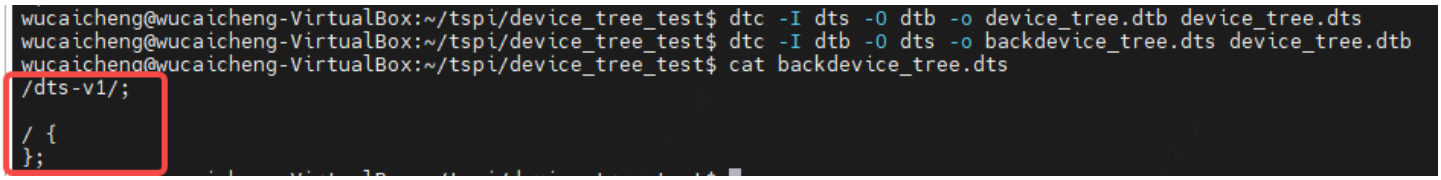
在反编译回来

```
dtc -I dtb -O dts -o backdevice_tree.dts device_tree.dtb
```

查看反编译回来的文件发现内容就是我们dtsi里的证明包含成功

```
cat backdevice_tree.dts
```

结果



方法二：

通过c语言语法包含头文件，他不止能包含dtsi文件还可以包含.h文件

```
#include "xxxx.dtsi" //xxxx是你要包含的dtsi文件名称
或者
#include "xxxx.h" //xxxx是你要包含的.h文件名称
```

不过需要注意的是`#include`是非设备树语法他是c语言语法，所以直接用dts编译是会报错的，我们需要先用cpp编译生成一个预编译文件，然后在用dtc编译这个预编译文件生成dtb，瑞芯微的设备树就是这么干的。

预编译命令

使用`cpp`工具将xxxx`.dts`文件中的头文件展开，生成一个临时文件xxxx`.dtb.dts.tmp`

```
cpp -nostdinc -I[dir_with_dts_includes] -x assembler-with-cpp [src_dts_file] > [tmp_dts_file]
```

- `-nostdinc`：不使用标准的系统头文件目录，避免不必要的报错。
- `-I[dir_with_dts_includes]`：这里是头文件的目录，如果就是在当前目录就用`I.`。
- `[src_dts_file]`是你的源设备树文件（`.dts`）。
- `[tmp_dts_file]`是预处理后的输出文件，为了和瑞芯微保持统一我们到时候命名后最写成`xxxxxxxxxxxx.dtb.dts.tmp`

试一试:

创建一个名为device_tree.h的头文件，并写入以下内容

```
#define LCKFB "www.lckfb.com"
```

创建一个名为device_tree.dtsi的头文件，并写入以下内容

```
#include "device_tree.h" //这里去包含上面的头文件
/{
        model = LCKFB;
};
```

再创建一个device_tree.dts名字的源文件，并写入一下内容

```
/dts-v1/;
#include "device_tree.dtsi"  //这里去包含上面的头文件
```

预编译

```
cpp -nostdinc -I. -x assembler-with-cpp device_tree.dts > device_tree.dtb.dts.tmp
```

查看一下预编译的内容，经过预编译以后，头文件都已经被提前处理了，比如`model = LCKFB;`就被代替成了`model = "www.lckfb.com";`

```
cat device_tree.dtb.dts.tmp
```

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ cat device_tree.temp.dts
# 1 "device_tree.dts"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "device_tree.dts"
/dts-v1/;
# 1 "device_tree.dtsi" 1
# 1 "device_tree.h" 1
# 2 "device_tree.dtsi" 2
/{
 model = "www.lckfb.com";
};
# 2 "device_tree.dts" 2
```

编译

```
dtc -I dts -O dtb -o device_tree.dtb device_tree.dtb.dts.tmp
```

在反编译回来

```
dtc -I dtb -O dts -o backdevice_tree.dts device_tree.dtb
```

查看反编译回来的文件发现内容就是我们dtsi里的证明包含成功

```
cat backdevice_tree.dts
```

设备树节点

根节点

设备树的根节点是设备树结构中的顶层节点，也是整个设备树的入口点，类比成前面姻缘树的根，根节点的名字是 `/`

```
/dts-v1/; // 指定这是一个设备树文件的版本（v1）

/ {// 根节点定义开始
    /* 这里是根节点，它代表整个设备树的基础。
       在这里可以定义一些全局属性或者包含更多的子节点。 */
}; // 根节点定义结束
```

子节点

子节点格式通常由以下几个基本元素组成：

- **节点名和可选的单元地址**：节点名通常是相关硬件设备或功能的名称，可选的单元地址表示设备的特定实例或资源，如内存地址、设备ID等。
- **一对花括号** `{}`：花括号用于封装节点的属性和子节点内容，包括开始花括号和结束花括号。
- **属性定义**：节点中定义了一系列属性，属性有名称和值，具体的值可以是数字、字符串或者数据数组等。属性和值之间使用等号 `=` 相连。

- **子节点定义**：一个节点可以包含多个子节点，这些子节点又可以进一步定义更为详细的属性或包含它们自己的子节点，从而创建一个层次结构。

```
标签: 节点名[@单元地址] { // 标签: 和 @单元地址不是必须
    子节点名1 {
        子节点1的子节点 {
        };
    };

    子节点名2 {
    };
};
```

节点示例：

```
/dts-v1/;  // 设备树编译器版本信息

/ {  // 根节点
    node1 {  // 第一个子节点
        child_node {  // node1 的子节点
            // 空节点，没有定义任何属性或子节点
        };
    };

    node2 {  // 第二个子节点
        // 空节点，没有定义任何属性或子节点
    };
};
```

节点名的命名规则(适合根节点以外的节点，根节点的命名就只有 `/` 符)

命名易懂（习惯）：

节点的名称应该描述节点所代表的硬件设备或者功能，让人能够容易理解节点的作用，见面知意。

```
/dts-v1/;
/ {
    // 串口设备
    serial {
    };

    // I2C 控制器及设备
```

```
    i2c {
    };

    // USB 控制器
    usb {
    };
}
```

小写字母，下划线或连字符（习惯）：

节点名通常全使用小写字母，反正我看官方的案例里面全部用的都是小写所以我们也保持风格统一。如果节点名包含多个单词，通常使用下划线(_)或连字符(-)来分隔这些单词。

```
/dts-v1/;
/ {
    serial_port {//比如我们看这个就知道是串口端口的意思
    };
    gpio-controller {//比如我们看这个就知道是gpio控制器的意思
    };
}
```

遵循已有的约定（习惯）：

如果你要描述的硬件信息是已经有现有描述过的，就尽量不要自己命名，可以去设备树文件里面复制参考官方的。

```
//绑定文档
SDK/kernel/Documentation/devicetree/bindings
//64位设备树
SDK/linux/kernel/arch/arm64/boot/dts
//32位设备树
SDK/linux/kernel/arch/arm/boot/dts
```

避免特殊字符（规则）：

名称中应避免使用空格、点（`.`）、斜杠（`/`）、反斜杠（`\`）等特殊字符。前面习惯你不注意还是可以用的但是，这个如果你不注意可能会直接报错

错误演示：

```
/dts-v1/;

```

```
/{
        node1\1{ //这里有一个特殊字符
                child_node{
                };
        };
        node2{

        };
};
```

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ cat device_tree.dts
/dts-v1/;

/{
        node1\1{
                child_node{
                };
        };
        node2{

        };
};
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ dtc -I dts -O dtb -o output_file.dtb device_tree.dts
Error: device_tree.dts:4.13-14 syntax error
FATAL ERROR: Unable to parse input tree
```

唯一性（规则）：

在设备树的同一级别层次内，节点名称应唯一。如果相同类型的节点有多个实例，通常在节点名称后附加一个索引号或实例特有的信息来区分。

错误演示

```
/dts-v1/;

/{
        node1{ //节点命相同
                child_node{
                };
        };
        node1{//节点命相同
        };
};

```

但是在同一层级外节点是可以名字相同的，下面不会报错

```
/dts-v1/;

/{
        node1{
```

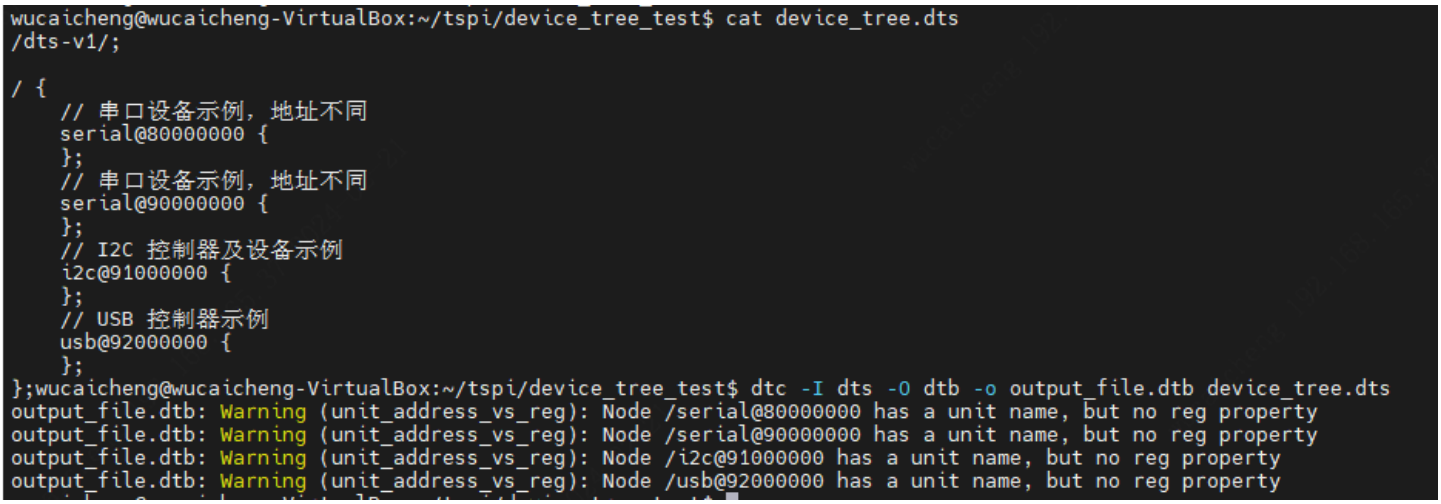
```
5                   child_node{//节点命相同
6                   };
7           };
8           node2{
9                   child_node{//节点命相同
10                  };
11          };
12  };
```

地址和类型（可选）：

节点名称中可以包含节点所代表的硬件的地址信息和类型。例如，`i2c@1c2c0000`指的是位于`1c2c0000`位置的I2C控制器。注意注意：这个并不是实际寄存器只是拿来看的增加可读性和避免命名冲突的，实际的地址我们后面属性会讲reg属性才是实际描述的寄存器地址。

```
1  /dts-v1/;
2
3  / {
4      // 串口设备示例，地址不同
5      serial@80000000 {
6      };
7      // 串口设备示例，地址不同
8      serial@90000000 {
9      };
10     // I2C 控制器及设备示例
11     i2c@91000000 {
12     };
13     // USB 控制器示例
14     usb@92000000 {
15     };
16 };
```

成功，警告没有关系是提示我们没有reg等属性



标签（重要）

上面子节点格式中我们还提到了标签，标签在节点名中不是必须的，但是我们可以通过他来更方便的操作节点，在设备树文件中有大量使用到。下面例子中定义了标签，并通过引用 `uart1` 标签方式往 `serial@80000000` 中追加一个 `node_add2` 节点。

创建一个名为 `device_tree.dts` 的文件并填入以下内容

```
/dts-v1/;

/ {
    // 串口设备示例，地址不同，uart1是标签
    uart1: serial@80000000 {
        node_add1{
        };
    };
    // 串口设备示例，地址不同，uart2是标签
    uart2: serial@90000000 {
    };
    // I2C 控制器及设备示例，i2c1是标签
    i2c1: i2c@91000000 {
    };
    // USB 控制器示例，USB是标签
    usb1: usb@92000000 {
    };
};

&uart1{ // 通过引用标签的方式往 serial@80000000 中追加一个节点非覆盖。
    node_add2{
    };
};
```

编译

```
1 dtc -I dts -O dtb -o device_tree.dtb device_tree.dts
```

忽略警告

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ dtc -I dts -O dtb -o device_tree.dtb device_tree.dts
device_tree.dtb: Warning (unit_address_vs_reg): Node /serial@80000000 has a unit name, but no reg property
device_tree.dtb: Warning (unit_address_vs_reg): Node /serial@90000000 has a unit name, but no reg property
device_tree.dtb: Warning (unit_address_vs_reg): Node /i2c@91000000 has a unit name, but no reg property
device_tree.dtb: Warning (unit_address_vs_reg): Node /usb@92000000 has a unit name, but no reg property
```

反编译

```
1 dtc -I dtb -O dts -o backdevice_tree.dts device_tree.dtb
```

忽略警告

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ dtc -I dtb -O dts -o backdevice_tree.dts device_tree.dtb
backdevice_tree.dts: Warning (unit_address_vs_reg): Node /serial@80000000 has a unit name, but no reg property
backdevice_tree.dts: Warning (unit_address_vs_reg): Node /serial@90000000 has a unit name, but no reg property
backdevice_tree.dts: Warning (unit_address_vs_reg): Node /i2c@91000000 has a unit name, but no reg property
backdevice_tree.dts: Warning (unit_address_vs_reg): Node /usb@92000000 has a unit name, but no reg property
```

查看

```
1 cat backdevice_tree.dts
```

通过引用标签的方式我们成功的往 `serial@80000000` 中追加了一个 `node_add2` 节点，这种方式在泰山派的设备树中有大量使用到。

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$ cat backdevice_tree.dts
/dts-v1/;

/ {

        serial@80000000 {

                node_add1 {
                };

                node_add2 {
                };
        };

        serial@90000000 {
        };

        i2c@91000000 {
        };

        usb@92000000 {
        };
};
wucaicheng@wucaicheng-VirtualBox:~/tspi/device_tree_test$
```

别名（特殊节点，了解即可）

aliases是一种在设备树中提供简化标识符的方式。它们主要用来为复杂的节点提供一个简单的别名，目的是为了方便引用，和标签有异曲同工之妙，但他们的作用是用途都不同，标签是针对特定设备节点的命名，而别名是针对设备节点路径的命名。

```
//通过aliases来定义别名
aliases{
    //这里面描述的都是别名
    [别名]=[标签]
    [别名]=[节点路径]
};
```

```
/dts-v1/;

/ {
    // 别名定义
    aliases {
        uart1 = &uart1; // uart1 别名指向名为 uart1 的设备节点
        uart2 = &uart2; // uart2 别名指向名为 uart2 的设备节点
        uart3 = "/serial@10000000"; // uart3 别名指向路径为 /serial@10000000 的设备节点,这里的/表示根目录
    };

    // 串口设备示例，地址不同，uart1 是标签
    uart1: serial@80000000 {
        // 可在此处添加串口设备的配置信息
    };

    // 串口设备示例，地址不同，uart2 是标签
    uart2: serial@90000000 {
        // 可在此处添加串口设备的配置信息
    };

    // 串口设备示例，地址不同，uart3 是标签，通过路径方式定义
    serial@10000000 {
        // 可在此处添加串口设备的配置信息
    };
};
```

标签和别名的区别（了解即可）

完了吧是不是标签和别名傻傻分不清了，个人的理解标签是为了在设备树中使用舒服的，有了标签我们访问就不需要访问全名了，别名是为了在内核源码中使用舒服有了别名在内核中查找设备树就不必写完整路径直接写别名就行。

全路径查找设备树：

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/kernel/drivers$ grep -r -n "of_find_node_by_path" ./
./leds/leds-powernv.c:289:      led_node = of_find_node_by_path("/ibm,opal/leds");
./ata/pata_macio.c:483:         struct device_node *root = of_find_node_by_path("/");
./macintosh/windfarm_mpu.h:88:  np = of_find_node_by_path(nodename);
./macintosh/windfarm_pm72.c:512:        u3 = of_find_node_by_path("/u3@0,f8000000");
./macintosh/via-pmu-led.c:91:   dt = of_find_node_by_path("/");
./watchdog/cpwd.c:556:  options = of_find_node_by_path("/options");
./of/overlay.c:224:     fragment_node = __of_find_node_by_path(ovcs->overlay_tree, path + 1);
./of/overlay.c:225:     overlay_node = __of_find_node_by_path(fragment_node, "__overlay__/");
./of/overlay.c:624:             node =  of_find_node_by_path(path);
./of/overlay.c:731:             fragment->target = of_find_node_by_path("/__symbols__");
./of/base.c:588:        root = of_find_node_by_path("/");
./of/base.c:838:struct device_node *__of_find_node_by_path(struct device_node *parent,
./of/base.c:865:                node = __of_find_node_by_path(node, path);
./of/base.c:920:                                np = of_find_node_by_path(pp->value);
./of/base.c:1913:       of_aliases = of_find_node_by_path("/aliases");
./of/base.c:1914:       of_chosen = of_find_node_by_path("/chosen");
./of/base.c:1916:               of_chosen = of_find_node_by_path("/chosen@0");
./of/base.c:1947:               np = of_find_node_by_path(pp->value);
./of/of_numa.c:30:      cpus = of_find_node_by_path("/cpus");
./of/of_private.h:116:struct device_node *__of_find_node_by_path(struct device_node *parent,
./of/platform.c:432:    root = root ? of_node_get(root) : of_find_node_by_path("/");
./of/platform.c:484:    root = root ? of_node_get(root) : of_find_node_by_path("/");
./of/platform.c:542:    node = of_find_node_by_path("/firmware");
./of/unittest.c:53:     np = of_find_node_by_path("/testcase-data");
./of/unittest.c:61:     np = of_find_node_by_path("/testcase-data/");
./of/unittest.c:64:     np = of_find_node_by_path("/testcase-data/phandle-tests/consumer-a");
./of/unittest.c:71:     np = of_find_node_by_path("testcase-alias");
./of/unittest.c:79:     np = of_find_node_by_path("testcase-alias/");
```

别名查找设备树：

```
wucaicheng@wucaicheng-VirtualBox:~/tspi/linux/kernel/drivers$ grep -r -n "of_alias_get_id" ./
./mmc/core/host.c:402:  alias_id = of_alias_get_id(dev->of_node, "mmc");
Binary file ./mmc/core/host.o matches
./mmc/host/dw_mmc-k3.c:134:     priv->ctrl_id = of_alias_get_id(host->dev->of_node, "mshc");
Binary file ./mmc/host/dw_mmc.o matches
./mmc/host/dw_mmc.c:2865:               ctrl_id = of_alias_get_id(host->dev->of_node, "mshc");
Binary file ./watchdog/watchdog_core.o matches
./watchdog/watchdog_core.c:216:         ret = of_alias_get_id(wdd->parent->of_node, "watchdog");
./pinctrl/pinctrl-st.c:1473:    int bank_num = of_alias_get_id(np, "gpio");
./pinctrl/pinctrl-at91.c:1712:  int alias_idx = of_alias_get_id(np, "gpio");
./pinctrl/samsung/pinctrl-samsung.c:990:        id = of_alias_get_id(node, "pinctrl");
Binary file ./rtc/class.o matches
./rtc/class.c:197:              of_id = of_alias_get_id(dev->of_node, "rtc");
./rtc/class.c:199:              of_id = of_alias_get_id(dev->parent->of_node, "rtc");
./of/base.c:1971: * of_alias_get_id - Get alias id for the given device_node
./of/base.c:1978:int of_alias_get_id(struct device_node *np, const char *stem)
./of/base.c:1997:EXPORT_SYMBOL_GPL(of_alias_get_id);
Binary file ./of/base.o matches
```

设备树属性

属性是键值对，定义了节点的硬件相关参数，属性有很多种我们下面只讲常用的标准属性，其他属性大家用到的时候再查。属性有对应的值，其中值的类型也有好几种，各种属性我们等会一一列举，我们先把属性能填哪些值搞明白。在设备树中，属性的值类型可以有多种，这些类型通常用于描述设备或子系统的各种特性和配置需求。以下是一些常见的属性值类型：

1. **字符串（String）**：
    - 属性名称：`compatible`
    - 示例值：`compatible = "lckfb,tspi-v10", "rockchip,rk3566";`

- 描述：指定该设备或节点与哪些设备或驱动兼容。

2. **整数（Integer）**：
   - 属性名称：`reg`
   - 示例值：`reg = <0x1000>;`。
   - 描述：定义设备的物理地址和大小，通常用于描述内存映射的I/O资源。
3. **数组（Array）**：
   - 属性名称：`reg`
   - 示例值：`reg = <0x1000,0x10>;`。
   - 描述：定义设备的物理地址和大小，通常用于描述内存映射的I/O资源。
4. **列表（List）**：
   - 属性名称：`interrupts`
   - 示例值：`interrupts = <0 39 4>, <0 41 4>,<0 40 4>;`。
   - 描述：用于定义例如中断列表，其中每个元组可以表示不同的中断属性（如编号和触发类型）。
5. **空值（Empty）**：
   - 属性名称：`regulator-always-on;`
   - 示例值：`regulator-always-on;`
   - 描述：表示该节点下的regulator是永久开启的，不需要动态控制。
6. **引用（Reference）**：
   - 属性名称：`gpios`
   - 示例值：`gpios = <&gpio1 RK_PB0 GPIO_ACTIVE_LOW>;`
   - 描述：提供一个句柄（通常是一个节点的路径或标识符），用于在其他节点中引用该节点。

model属性（字符串）

model的值是字符串，主要是用于描述开发板型号，有助于用户和开发人员识别硬件。

```
/{
    model = "lckfb tspi V10 Board";
}
```

compatible属性（字符串或字符串列表）

`compatible`：这是最最最最最关键的属性之一，它用于标识设备的兼容性字符串。操作系统使用这个属性来匹配设备与相应的驱动程序。

```
rk_headset: rk-headset {
    compatible = "rockchip_headset", "rockchip_headset2";
};
```

耳机检测驱动中会通过`"rockchip_headset"`来匹配驱动

`kernel/drivers/headset_observe/rockchip_headset_core.c`

```
.......
static const struct of_device_id rockchip_headset_of_match[] = {
    { .compatible = "rockchip_headset", },  // 定义设备树匹配项，指定兼容性字符串，与上面的设备树匹配
    {},                                     // 结束符号
};
MODULE_DEVICE_TABLE(of, rockchip_headset_of_match);  // 定义设备树匹配表供内核使用

static struct platform_driver rockchip_headset_driver = {
    .probe  = rockchip_headset_probe,   // 注册设备探测函数
    .remove = rockchip_headset_remove,  // 注册设备移除函数
    .resume = rockchip_headset_resume,  // 注册设备恢复函数
    .suspend =  rockchip_headset_suspend, // 注册设备挂起函数
    .driver = {
        .name   = "rockchip_headset",   // 设备名称
        .owner  = THIS_MODULE,          // 持有模块的指针
        .of_match_table = of_match_ptr(rockchip_headset_of_match),  // 设备树匹配表指针
    },
};
.........
```

reg属性（地址，长度对）

描述了设备的物理地址范围，包括基址与大小，与`address-cells`和`size-cells`结合使用。

```
gmac1: ethernet@fe010000 {
    reg = <0x0 0xfe010000 0x0 0x10000>;
}
```

#address-cells属性（整数）和#size-cells属性（整数）

用于说明父节点如何解释它的子节点中的 `reg` 属性。

`reg` 属性的一般格式：

```
reg = <[address1] [length1] [address2] [length2] ...>;
```

- `[addressN]`：表示区域的起始物理地址。用多少个无符号整数来表示这个地址取决于父节点定义的 `#address-cells` 的值。例如，如果 `#address-cells` 为1，则使用一个32位整数表示地址；如果 `#address-cells` 为2，则使用两个32位整数表示一个64位地址。
- `[lengthN]`：表示区域的长度（或大小）。用多少个无符号整数来表示这个长度同样取决于父节点定义的 `#size-cells` 的值。

根据 `#address-cells` 和 `#size-cells` 的定义，单个[address, length]对可能会占用2、3、4个或更多的元素。

例如，如果一个设备的寄存器空间位于地址 `0x03F02000` 上，并且占用 `0x1000` 字节的大小，假设其父节点定义了 `#address-cells = <1>` 和 `#size-cells = <1>`，`reg` 属性的表达方式如下：

```
reg = <0x03F02000 0x1000>;
```

如果地址是64位的，父节点 `#address-cells = <2>` 和 `#size-cells = <1>`，那么 `reg` 属性可能会这样写，以表示地址 `0x00000001 0x03F02000` 和大小 `0x1000`：

```
reg = <0x00000001 0x03F02000 0x1000>;
```

案例

```
/ {
    #address-cells = <2>;
    #size-cells = <2>;
    cpus {
        #address-cells = <2>;
        #size-cells = <0>;
        cpu0: cpu@0 {
            受cpus节点的影响
```

```
            #address-cells = <2>;
            #size-cells = <0>;
            所以地址就是0x0，大小就是 0x0
            */
            reg = <0x0 0x0>;
        };
    };
    gmac1: ethernet@fe010000 {
        /*
        受根节点的影响
        #address-cells = <2>;
        #size-cells = <2>;
        所以地址就是0xfe010000 ，大小就是 0x10000
        */
        reg = <0x0 0xfe010000 0x0 0x10000>;
    };
};
```

status属性（字符串）

这个属性非常重要，我们设备树中其实修改的最大的就是打开某个节点或者关闭某个节点，status属性的值是字符串类型的，他可以有以下几个值，最常用的是okay和disabled。

`status` 属性值包括：

- `"okay"`：表示设备是可操作的，即设备当前处于正常状态，可以被系统正常使用。
- `"disabled"`：表示设备当前是不可操作的，但在未来可能变得可操作。这通常用于表示某些设备（如热插拔设备）在插入后暂时不可用，但在驱动程序加载或系统配置更改后可能会变得可用。
- `"fail"`：表示设备不可操作，且设备检测到了一系列错误，且设备不太可能变得可操作。这通常表示设备硬件故障或严重错误。
- `"fail-sss"`：与 `"fail"` 含义相同，但后面的 `sss` 部分提供了检测到的错误内容的详细信息。

```
//用户三色灯
&leds {
    status = "okay";
};
//耳机插入检测，不使用扩展板情况需关闭，否则默认会检测到耳机插入
&rk_headset {
    status = "disabled";
};
```

device_type属性（字符串）

`device_type` 属性通常只用于 `cpu` 节点或 `memory` 节点。例如，在描述一个CPU节点时，`device_type` 可能会被设置为 `"cpu"` ，而在描述内存节点时，它可能会被设置为 `"memory"` 。

```
device_type = "cpu";
```

自定义属性

自定义属性需要注意不要和标准属性冲突，而且尽量做到见名知意

```
/ {
    my_custom_node { /* 自定节点 */
        compatible = "myvendor,my-custom-device"; /* 兼容性属性 */
        my_custom_property = <1>; /* 自定义属性，假设为整数类型 */
        my_custom_string_property = "My custom value"; /* 自定义字符串属性 */
    };
};
```

实战讲解

课后作业

- [ ] 修改设备树实现泰山派上的红灯1秒钟亮灭一次，蓝灯两秒钟亮灭，绿灯三秒亮灭一次
- [ ] 修改设备树失能泰山派上的三色用户LED灯
- [ ] 修改设备树失能和使能HDMI输出
- [ ] 关闭网口解决串口因寻不到网口频繁打印日志问题
- [ ] 调试串口的波特率目前是1500000，尝试把波特率改成115200
- [ ] 修改红外遥控器值