uboot 实例分析

Table of Contents

1. 硬件概述

1.1. 硬件

tiny210(s5pv210)

2. 上电启动流程（BL0-BL2）

2.1. 说明

本文主要以友善之臂的 tiny210 板子作说明，使用的是 s5pv210 核心。
主要集中于以下几个问题:
* 支持哪些存储介质?
* 上电之后的启动流程? 分成了几个阶段? 具体负责什么功能?
* 各个阶段使用的存储介质是什么? 原因?
* 各个阶段的运行地址是什么?
* 镜像存放可能存放在几种存储介质中，如何判断要从哪种存储介质中获取镜像? （也就是启动模式）

本文只介绍 non secure boot 的情况。也就是说不是在 secure 环境下，无需做签名处理。

2.2. 和启动相关的硬件介绍

2.2.1. 可运行存储介质

要先理解一点，运行介质都会在 CPU 的地址空间上，占用地址空间的一部分。CPU 可以根据寻址地址从运行介质从读取一条指令或者一条数据，而并不是说直接在运行介质上执行。

IROM
ROM（Read Only Memory），唯读记忆体。ROM 数据不能随意更新，但是在任何时候都可以读取。主要用于存放一些固定的不需要修改的代码或者数据。掉电之后，数据还可以保存。
IROM 则是指集成于芯片内部的 ROM.
SRAM
SRAM（Static Random Access Memory），即静态随机存取存储器。它是一种具有静止存取功能的内存，不需要刷新电路即能保存它内部存储的数据。
优点，速度快，不必配合内存刷新电路，可提高整体的工作效率。初始化简单。
缺点，集成度低，掉电不能保存数据，功耗较大，相同的容量体积较大，而且价格较高，少量用于关键性系统以提高效率。
SDRAM
由 General SDRAM and Controller 进行控制。
SDRAM（Synchronous Dynamic Random Access Memory），同步动态随机存储器，同步是指内存工作需要同步时钟，内部的命令的发送与数据的传输都以它为基准；动态是指存储阵列需要不断的刷新来保证数据不丢失；随机是指数据不是线性依次存储，而是自由指定地址进行数据读写。相对于 SRAM 来说，体积较小且价格偏移。

三者差异简单归纳如下表:

可运行存储介质	掉电数据保存情况	数据修改	访问速度	动态刷新电路	价格
ROM	掉电数据保存	不易修改	最慢	不需要	最便宜
SRAM	掉电数据丢失	可以修改	最快	不需要	最贵
SDRAM	掉电数据丢失	可以修改	快	需要	便宜

补充说明
综上，因为 BL0 是上电启动代码，固定不变的，所以将其固化在 IROM 中运行。

因为 SRAM 是集成在平台内部，BL0 阶段之后可以直接使用，但是 size 比较小。
而 SDRAM 需要根据外围 DDR 进行初始化，BL0 之后还无法使用，需要在 BL1 中进行初始化之后才能使用。
故 BL1 放在 SRAM 中运行，BL2 放在 SDRAM 中运行。
后续会继续说明。

2.2.2. 存储介质

SD/MMC/eMMC
nand flash / nor flash / OneNand flash
eSSD

2.2.3. tiny210 运行介质和存储介质说明

首先看一下对应 s5pv210 平台和存储相关的硬件支持

64KB 的 IROM
94KB 的内部 SRAM
通用 SDRAM 控制器，用于控制 SDRAM
需要外部实现 SDRAM.
4/8 位高速 SD/MMC 控制器，用于控制 4-bit SD / 4-bit MMC / 4 or 8-bit eMMC
需要外部实现 SD/MMC/eMMC
Nand Flash 控制器，用于控制 nand flash
需要外部实现 nand flash
OneNand 控制器，用于控制 OneNand flash。
需要外部实现 OneNand flash
eSSD 控制器，用于控制 eSSD
需要外部实现 eSSD
UART/USB 控制器

在 s5pv210 的基础上，对应 tiny210 板子上有如下外围硬件：
* 512MB 的 DDR 支持
* 1GB 的 nand flash 支持（K9K8G08U0B）
* SD card 接口支持，可以直接支持 SD card

综上，tiny210 板子上有如下存储介质：

64KB 的 IROM
94KB 的内部 SRAM
512MB 的 DDR
512MB 的 nand flash 支持
SD card（自己插了一个 4GB 的 SD 卡）

2.3. s5pv210 上电启动的几个阶段

注意：以下介绍的启动流程和官方给的启动流程《S5PV210-iROM-ApplicationNote-Preliminary》有所差异，是实际上我在项目中使用的启动流程，也是比较通用的启动流程。和官方文档中比较大的差异在于，BL2 在下面介绍是放在 SDRAM 中运行，而官方文档《S5PV210-iROM-ApplicationNote-Preliminary》中则是放在 IRAM 中运行，因为编出来的 uboot 过大，超过了限制，所以官方文档是不可行的。

2.3.1. BL0

运行在 IROM 上！！！

加载说明
上电后启动的第一个阶段。其代码固化在 s5pv210 的 IROM 中，也就是出厂后已经完成，无法进行修改。上电后 CPU 会直接从 IROM 上的 BL0 的代码上开始执行。

主要工作有：
初始化系统时钟、特殊设备的控制器、启动设备、看门狗、堆栈、SRAM 等等
验证 BL1 镜像
从存储介质上（比如 SD\eMMC\nand flash）或者通过 USB 加载 BL1 镜像到对应内部 SRAM 上
跳转到 BL1 镜像所在的地址上

2.3.2. BL1

运行在内部 SRAM 上！！！

- 加载说明

上电后启动的第二个阶段。BL0 阶段会将其镜像或者代码从存储介质上（比如 SD\eMMC\nand flash）上加载到内部 SRAM 上.

- 主要工作有：

初始化部分时钟（和 SDRAM 相关）
初始化 DDR（外部 SDRAM）
从存储介质上（比如 SD\eMMC\nand flash）将 BL2 镜像加载到 SDRAM 上
验证 BL2 镜像的合法性
跳转到 BL2 镜像所在的地址上

2.3.3. BL2

运行在外部 SDRAM 上！！！

- 加载说明

上电后启动的第三个阶段，BL1 阶段会将其镜像或者代码从存储介质上（比如 SD\eMMC\nand flash）上加载到外部 SDRAM 上.

- 主要功能

BL2 就是指传统意义上的 bootloader，也就是我们这里的 uboot 的主体，负责 flash 操作、uboot 命令操作等等，并且最终目标是加载 OS 和启动 OS，如 linux。关于 uboot 的实现，会在后面具体分析，这里不详细说明。
上电到 BL2 的流程图如下（略过 BL2 的流程部分）

1. 上电，直接执行 BL0
2. BL0(IROM)
3. BL0(IROM)：初始化系列操作
4. BL0(IROM)：加载 BL1 镜像到 SRAM 上
5. BL0(IROM)：跳转到 BL1 的地址上
6. BL1(SRAM)
7. BL1(SRAM)：初始化 SDRAM
8. BL1(SRAM)：加载 BL2 镜像到 SDRAM 上
9. BL1(SRAM)：跳转到 BL2 的地址上
10. BL2(SDRAM)
11. jump into OS

2.4. 各个阶段在地址空间上的运行位置

2.4.1. s5pv210 地址空间

参考《S5PV210_UM_REV1.1》 2.1 MEMORY ADDRESS MAP
s5pv210 整体地址空间映射表如下：

起始地址	结束地址	长度	映射区域描述
0x0000_0000	0x1FFF_FFFF	512MB	Boot area（取决于启动模式）
0x2000_0000	0x3FFF_FFFF	512MB	DRAM 0
0x4000_0000	0x7FFF_FFFF	1024MB	DRAM 1
0x8000_0000	0x87FF_FFFF	128MB	SROM Bank 0
0x8800_0000	0x8FFF_FFFF	128MB	SROM Bank 1
0x9000_0000	0x97FF_FFFF	128MB	SROM Bank 2
0x9800_0000	0x9FFF_FFFF	128MB	SROM Bank 3
0xA000_0000	0xA7FF_FFFF	128MB	SROM Bank 4
0xA800_0000	0xAFFF_FFFF	128MB	SROM Bank 5
0xB000_0000	0xBFFF_FFFF	256MB	OneNAND/NAND Controller and SFR
0xC000_0000	0xCFFF_FFFF	256MB	MP3_SRAM output buffer
0xD000_0000	0xD000_FFFF	64KB	IROM
0xD001_0000	0xD001_FFFF	64KB	Reserved
0xD002_0000	0xD003_7FFF	96KB	IRAM
0xD800_0000	0xDFFF_FFFF	128MB	DMZ ROM
0xE000_0000	0xFFFF_FFFF	512MB	SFR region

重点关注如下几个地址空间：

起始地址	结束地址	长度	映射区域描述
0x0000_0000	0x1FFF_FFFF	512MB	Boot area（取决于启动模式）
0x2000_0000	0x3FFF_FFFF	512MB	DRAM 0
0x4000_0000	0x7FFF_FFFF	1024MB	DRAM 1
0xD000_0000	0xD000_FFFF	64KB	IROM
0xD002_0000	0xD003_7FFF	96KB	IRAM

2.4.2. BL0 的运行地址

通过文档《S5PV210-iROM-ApplicationNote-Preliminary》位于如下区域

起始地址	结束地址	长度	映射区域描述
0x0000_0000	0x0000_FFFF	64KB	Internal ROM

s5pv210 上电之后，CPU 会直接从 0x0 地址取指令，也就是直接执行 BL0。

2.4.3. BL1 的运行地址

上述已经说明 BL1 在 IRAM 中运行，IRAM 空间如下：

起始地址	结束地址	长度	映射区域描述
0xD002_0000	0xD003_7FFF	96KB	IRAM

但是并不是说整个 IRAM 都是给 BL1 使用的。
BL1 使用的部分如下：

起始地址	结束地址	长度	映射区域描述
0xD002_0000	0xD003_5400	-	BL1 代码空间

BL1 镜像最大为 16KB。
这里有一个注意事项:
BL1 是加载在 0xD002_0000 的位置上（不包括 USB 启动的方法），并不是说 BL1 就是从 0xD002_0000 开始运行的。0xD002_0000 开始的 16B 是用于作为 BL1 的 header。BL1 代码的实际运行地址是 0xD002_0010
这部分 header 是用来给 BL0 验证 BL1 的镜像的有效性使用的。16B 的 header 格式如下：

地址	数据
0xD002_0000	BL1 镜像包括 header 的长度
0xD002_0004	保留，设置为 0
0xD002_0008	BL1 镜像除去 header 的校验和
0xD002_000c	保留，设置为 0

当从存储介质上加载 BL1 镜像的时候（非 UART/USB 模式），需要校验这个 header 和 BL1 的镜像是否匹配，后续启动模式中会继续说明。

2.4.4. BL2 的运行地址

上述已经知道 BL2 在 SDRAM 的地址空间中运行,SDRAM 空间如下:

起始地址	结束地址	长度	映射区域描述
0x2000_0000	0x3FFF_FFFF	512MB	DRAM 0
0x4000_0000	0x7FFF_FFFF	1024MB	DRAM 1

tiny210 的 ddr 是 512M 并且使用的是 DRAM0 的地址空间，故 BL2 的运行空间是

起始地址	结束地址	长度	映射区域描述
0x2000_0000	0x3FFF_FFFF	512MB	DRAM 0

但是在实际使用中，我把 uboot 加载到了 ddr 上 0x23E0_0000（具体参考 CONFIG_SYS_TEXT_BASE 的定义）的地址上。

==========
综上所述,各个阶段的运行地址如下:

阶段	运行地址
BL0	0x0000_0000（固定）
BL1	0xD002_0010（固定）
BL2	0x3000_0000（可在代码中进行修改）

2.5. 启动模式（如何将对应 BL1 加载到对应 RAM 上面）

在上述中，我们知道 BL0 负责将 BL1 的镜像加载到对应的地址上。
那么，有一个问题，BL1 的镜像可能存放在 SD 卡，eMMC，Nand flash 等存储介质上，也有可能是直接通过 USB 上写入到对应 RAM 的地址上。那么，BL0 如何判断要从哪里去加载 BL1 的镜像呢，这就取决于启动模式了。

2.5.1. s5pv210 支持的启动模式

OneNand Boot 模式
在这种模式下，BL0 会从 OneNAND 的 page0 开始加载，先加载 16B 的 header 到 0xD002_0000，通过 header 中的 size 判断需要读取多少 page，然后再加载剩余的 page 到 0xD002_0010 上。
最后计算除 header 外的数据的校验和，和 header 中的校验和比较，一旦匹配，则跳转到 0xD002_0010。
因此，当我们选择这种方式的时候，需要给 BL1 的镜像加上 16B 的 header，然后以 OneNand page0 位置开始存放整个镜像。

Nand Boot (with H/W 8/16-Bit ECC)

在这种模式下，BL0 会从 NAND 的 page0 开始加载，先加载 16B 的 header 到 0xD002_0000，通过 header 中的 size 判断需要读取多少 page，然后再加载剩余的 page 到 0xD002_0010 上。

最后计算除 header 外的数据的校验和，和 header 中的校验和比较，一旦匹配，则跳转到 0xD002_0010。

因此，当我们选择这种方式的时候，需要给 BL1 的镜像加上 16B 的 header，然后以 Nand page0 位置开始存放整个镜像。

SD / MMC Boot

在这种模式下，BL0 会跳过 SD 的 block0，从 SD 的 block1 加载，先加载 block1 到 0xD002_0000，通过 header 中的 size 判断需要读取多少 block，然后再加载剩余的 block。

最后计算除 header 外的数据的校验和，和 header 中的校验和比较，一旦匹配，则跳转到 0xD002_0010。

因此，当我们选择这种方式的时候，需要给 BL1 的镜像加上 16B 的 header，然后以 block1 位置开始存放整个镜像。

eSSD Boot

在这种模式下，BL0 会跳过 eSSD 的 block0，从 eSSD 的 block1 加载，先加载 block1 到 0xD002_0000，通过 header 中的 size 判断需要读取多少 block，然后再加载剩余的 block。

最后计算除 header 外的数据的校验和，和 header 中的校验和比较，一旦匹配，则跳转到 0xD002_0010。

因此，当我们选择这种方式的时候，需要给 BL1 的镜像加上 16B 的 header，然后以 block1 位置开始存放整个镜像。

eMMC Boot

在这种模式下，BL0 会从 eMMC 的 block0 加载，先加载 block0 到 0xD002_0000，通过 header 中的 size 判断需要读取多少 block，然后再加载剩余的 block。

最后计算除 header 外的数据的校验和，和 header 中的校验和比较，一旦匹配，则跳转到 0xD002_0010。

因此，当我们选择这种方式的时候，需要给 BL1 的镜像加上 16B 的 header，然后以 block0 位置开始存放整个镜像。

UART Boot

在这种模式下，BL0 会检测串口数据，并且将其复制到 0xD002_0000 开始的位置。在 PC 侧通过 DWN 软件将镜像通过串口进行写入。

因为这种模式下不需要验证 header，一旦下载完成之后，会直接跳转到 0xD002_0000 进入 BL1。

USB Boot

在这种模式下，BL0 会检测 USB 数据，并且将其复制到 0xD002_0000 开始的位置。在 PC 侧通过 DWN 软件将镜像通过 USB 进行写入。

因为这种模式下不需要验证 header，一旦下载完成之后，会直接跳转到 0xD002_0000 进入 BL1。

几个模式的差异如下表格

模式	硬件支持	BL1 镜像存放起始位置	BL1 镜像是否需要 header	加载位置	跳转位置

OneNand Boot	OneNand flash	page0	是	0xD002_0000	0xD002_0010
Nand Boot	Nand flash	page0	是	0xD002_0000	0xD002_0010
SD / MMC Boot	SD / MMC	block1	是	0xD002_0000	0xD002_0010
eSSD Boot	eSSD	block1	是	0xD002_0000	0xD002_0010
eMMC Boot	eMMC	block0	是	0xD002_0000	0xD002_0010
UART Boot	UART	无	否	0xD002_0000	0xD002_0000
USB Boot	USB	无	否	0xD002_0000	0xD002_0000

2.5.2. s5pv210 选择启动模式的方法

s5pv210 通过 OM[5:0]这 6 个引脚的组合来选择对应的启动模式。
具体查看《S5PV210-iROM-ApplicationNote-Preliminary》表格 Table4. iROM OM pin description
以下举几个常见的例子

启动方式	OM5	OM4	OM3	OM2	OM1	OM0
SD BOOT	0	0	1	1	0	0
Nand 2KB 5cycle (Nand 8bit ECC)	0	0	0	0	1	0
Nand 4KB 5cycle (Nand 8bit ECC)	0	0	0	1	0	0
Nand 4KB 5cycle (default)(Nand 16bit ECC)	0	0	0	1	1	0
优先检测 UART\USB 模式	1	-	-	-	-	-

注意，当 OM5 用于决定是否先进入 UART\USB 模式。如果 OM5 为 0，则直接进入 OM[4:0]对应的模式中。
如果 OM5 为 1，则先判断串口是否有响应（因此要进入该模式需要现在 DNW 软件上打开对应 UART 下载功能），有响应则进入 UART 模式，否则，超时，检测是否进入 USB 模式。
然后判断 USB 是否有相应（因此要进入该模式需要现在 DNW 软件上打开对应 USB 下载功能），有响应则进入 USB 模式，否则，超时，跳转到 OM[4:0]对应的模式中。

2.5.3. tiny210 支持的启动模式

通过 tiny210 的原理图《Tiny210-1305-Schematic》上看 OM5\OM4\OM0 均被拉低为 0.而 OM[3:1]则有拨码开关 S2 和反相器来控制其组合。
最终得到 tiny210 只支持如下两种启动方式

启动方式	OM5	OM4	OM3	OM2	OM1	OM0
SD BOOT	0	0	1	1	0	0
Nand 4KB 5cycle (default)(Nand 16bit ECC)	0	0	0	1	1	0

3. 从存储设备加载代码到 DDR

3.1. 说明

3.1.1. 疑问

前面文章中《[uboot] （第三章）uboot 流程一一uboot-spl 代码流程》中，最后 uboot-spl 的操作是调用 board_init_f 了，在 board_init_f 中实现了加载 BL2 镜像（uboot.bin）到 ddr 上，然后跳转到相应位置。
那么前面有个坑，就是 tiny210 的 board_init_f 中，是 如何实现加载 BL2 镜像（uboot.bin）到 ddr 上，然后跳转到 BL2 的相应地址上?
这个也就是本文里面的主要介绍的:
s5pv210 是如何实现把镜像从存储设备上加载到 ddr 上的?

3.1.2. 原理

通过《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》， s5pv210 的 BL0 会根据启动模式从相应存储设备上获取 BL1 的镜像到 DDR 上。
例如，当启动模式是 SD boot 的时候，BL0 会从 SD 卡上获取 BL1 镜像，加载到对应位置上。当启动模式是 nand flash boot 的时候，BL0 会从 nand flash 上获取 BL1 镜像，加载到对应位置上。
如下表格（从《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》截取的一部分）

模式	硬件支持	BL1 镜像存放起始位置	BL1 镜像是否需要 header	加载位置	跳转位置
Nand Boot	Nand flash	page0	是	0xD002_0000	0xD002_0010
SD / MMC Boot	SD / MMC	block1	是	0xD002_0000	0xD002_0010

同时，BL0 是固化在 s5pv210 的 IROM 中的，是芯片在出厂的时候已经实现好了的。相应的，其从存储设备（SD/nand/eMMC）上获取镜像的代码就也就必须是已经实现好的。
既然，从存储设备（SD/nand/eMMC）上获取镜像的代码已经实现好了，也就是存储设备代码拷贝函数，并且在 BL1 阶段和 BL2 阶段可能也要需要使用到相同的功能，于是 s5pv210 将这部分代码的也是固化在 IROM 中，并且将这些函数接口的函数指针存放到某一个固定的地方。
当 BL1 或者 BL2 需要使用到从存储设备（SD/nand/eMMC）上获取镜像的函数接口时，就可以从对应的地方获取函数指针。
而剩下的，也就是最重要的是，这些函数指针被存放在上什么地方？这也是我们后续重点关注的地方。
为什么是函数指针而不是函数地址呢，是因为可以直接使用函数指针来调用函数。

3.2. s5pv210 代码拷贝函数介绍

现在，我们要关心的是 s5pv210 的存储设备代码拷贝函数的函数原型以及函数指针的存放位置。
主要参考文档: 《S5PV210-iROM-ApplicationNote-Preliminary-20091126》
先一段文档里面的说明:
_The S5PV210 internally has a ROM code of block copy function for boot-u device. Therefore, developer may not needs to implements device copy functions. These internal functions can copy any data from memory devices to SDRAM. User can use these function after ending up the internal ROM

boot process completely._
简单翻译为如下：
s5pv210 的 IROM 中集成了一些启动存储设备的块拷贝函数。因此，开发者不需要实现对应的存储设备的拷贝函数了。这些拷贝函数可以实现从存储设备到 memory 的拷贝。当 IROM 启动完成之后，开发者也可以直接使用这些拷贝函数了。
这和我们第一节中，说明的是一致的。

3.2.1. 存储设备代码拷贝函数（Device Copy Function）地址存储位置

这里要特别注意，这里既不是指函数指针，也不是指函数地址，而是指函数指针被存放到的位置。
可以通过函数指针被存放到的位置找到函数指针，调用函数指针就可以调用到相应的函数了。

函数指针的地址	函数名	功能
0xD0037F90	NF8_ReadPage	is advanced NF8_ReadPage function.
0xD0037F94	NF16_ReadPage_Adv	is advanced NF16_ReadPage function
0xD0037F98	CopySDMMCtoMem	can copy any data from SD/MMC device to SDRAM
0xD0037F9C	CopyMMC4_3toMem	can copy any data from eMMC device to SDRAM.
0xD0037FA0	CopyOND_ReadMultiPages	can copy any data from OneNand device to SDRAM.
0xD0037FA4v	CopyOND_ReadMultiPages_Adv	can copy any data from OneNand device to SDRAM.
0xD0037FA8	Copy_eSSDtoMem	can copy any data from eSSD device to SDRAM.
0xD0037FAC	Copy_eSSDtoMem_Adv	can copy any data from eSSD device to SDRAM.
0xD0037FB0	NF8_ReadPage_Adv128p	is advanced NF8_ReadPage function.

更详细的功能介绍建议参考文档《S5PV210-iROM-ApplicationNote-Preliminary-20091126》。
这些地址是递归的，位于 0xD003_7F90-0xD003_7FB0
通过文档《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》，我们知道了 0xD002_0000-0xD003_7FFF 是属于 IRAM 的区域，所以说’函数指针存放地址’是放在 IRAM 上的，按照我猜，应该是在启动过程中，BL0 把这些 IROM 上这些固化的函数的地址写到 IRAM 对应的位置上，例如 0xD0037F90，对应位置就可以当作一个函数指针来使用。

3.2.2. 函数原型说明

文档中的函数原型和使用过程中的函数原型有所差异。个人感觉可能是文档有问题，但是这里还是以文档为准。使用介绍中，再以实际的使用为准。
因为 tiny210 只支持 SD boot 和 nand flash 的方式，下面我就以 SD 和 nand flash 作为存储设备的拷贝函数做介绍。

（1）CopySDMMCtoMem
从 SD/MMC 拷贝（加载）块到内存中的函数。

函数结构如下（也就是一个简单的使用范例）
```
/**
* This Function copy MMC(MoviNAND/iNand) Card Data to memory.
* Always use EPLL source clock.
* This function works at 20Mhz.
* @param u32 StartBlkAddress : Source card(MoviNAND/iNand MMC)) Address.(It must block address.)
* @param u16 blockSize : Number of blocks to copy.
* @param u32* memoryPtr : Buffer to copy from.
* @param bool with_init : determined card initialization.
* @return bool(u8) - Success or failure.
*/
#define CopySDMMCtoMem(z,a,b,c,e) (((bool(*)(int, unsigned int, unsigned short, unsigned int*, bool))(*((unsigned int *)0xD0037F98)))(z,a,b,c,e))
```

可以简单推测出函数原型为 bool CopySDMMCtoMem(int, unsigned int, unsigned short, unsigned int*, bool)。

argv0=起始块号
argv1=块数量
argv2=要拷贝到内存的什么位置上，也就是目标地址指针。
argv3=是否需要检测初始化
argv4=返回参数，用于决定是成功还是失败。

注意：以上是文档的说明，
但是在实际使用中，实际的函数原型如下：
boot copy_sd_to_ddr(u32 channel, u32 start_block, u16 block_size, u32 *trg, u32 init);
可见，文档是有问题的。后续会说明。

（2）NF8_ReadPage_Adv
从 nand flash 拷贝（加载）页到内存中的函数。
8bit ECC 校验的函数结构如下（也就是一个简单的使用范例）
```
/**
* This Function copies a block of page to destination memory.( 8-Bit ECC only )
* @param uint32 block : Source block address number to copy.
* @param uint32 page : Source page address number to copy.
* @param uint8 *buffer : Target Buffer pointer.
* @return int32 - Success or failure.
*/
#define NF8_ReadPage_Adv (a,b,c) (((int(*)(uint32, uint32, uint8*))(*((uint32 *) 0xD0037F90)))(a,b,c))
```

其参数意义参考注释，这里不加以说明了。

3.3. s5pv210 代码拷贝函数使用示例

s5pv210 代码拷贝函数是以函数指针的方式进行使用的，因为，在这里先介绍一个简单的函数指针的使用示例。

3.3.1. 函数指针简单示例说明

假设有一个函数原型是 int function(int a, int b)，其函数指针被存放在了 0x10 的位置上。
那么我们调用该函数的方法如下：
（1）首先需要获取到函数指针
因为函数指针存放在了 0x10 的位置上，所以先对 0x10 进行强制类型转换成一个指针(unsigned int *)0x10，
然后*((unsigned int *)0x10)就可以获取到函数指针变量。
暂时表示如下
f_ptr=(*((unsigned int *)0x10))
（2）根据函数原型可以得到对应的函数指针类型是 int (*)(int, int),所以需要对（1）的函数指针进行强制类型转换
(int (*)(int, int))f_ptr，
暂时表示如下
ok_f_ptr=((int (*)(int, int))f_ptr)
（3）调用该函数的方法为 ok_f_ptr(a, b)
等价于*ok_f_ptr(a, b)，一般都是使用 ok_f_ptr(a, b)
综上，展开则为((int (*)(int, int))(*((unsigned int *)0x10)))(a,b)

3.3.2. tiny210 从 SD 上加载代码实现

目前 project X 的设计中，tiny210 的 BL1(uboot-spl)是从 SD 上加载 BL2 的镜像(uboot.bin)到 ddr 上的。
并且 BL2(uboot)也是从 SD 卡上加载 kenrel、rootfs、dtb 到 ddr 上的。原理都是一样的。
以下，我们以“BL1(uboot-spl)是从 SD 上加载 BL2 的镜像(uboot.bin)到 ddr 上的”的代码 copy_bl2_to_ddr 为例。
理解了上述 1 之后，以下代码就很好理解了。
代码 copy_bl2_to_ddr 如下：
board/samsung/tiny210/board.c 中

```
#define CopySDMMCtoMem 0xD0037F98
// 上述我们已经说明了，CopySDMMCtoMem 的函数指针是存放在 0xD0037F98 的。

typedef u32(*copy_sd_mmc_to_mem)
    (u32 channel, u32 start_block, u16 block_size, u32 *trg, u32 init);
// 这样就定义了 copy_sd_mmc_to_mem 为 u32 (*)(u32, u32, u16, u32 *, u32)类型，后续可以直接用于函数指针变量的定义和类型强制转换
// argv0=通道号
// argv1=起始块号
// argv2=块数量
// argv3=要拷贝到的目标地址
// argv4 不解

// 以下就是加载 BL2 镜像到 DDR 中的主体了，是在 BL1 中执行的。
// 主要关心一下函数指针的获取、转化和调用过程
void copy_bl2_to_ddr(void)
{
  u32 sdmmc_base_addr;
  copy_sd_mmc_to_mem copy_bl2 = (copy_sd_mmc_to_mem)(*(u32*)CopySDMMCtoMem);
// 这个是重点函数，也是要重点理解的地方
```

// (u32*)CopySDMMCtoMem 先将 0xD003_7F98 强制类型转换成一个指针
// (*(u32*)CopySDMMCtoMem)从 0xD003_7F98 中获取函数指针
// (copy_sd_mmc_to_mem)对得到的函数指针进行强制类型转换，赋值给 copy_bl2，后续直接调用 copy_bl2 即可。

```
    sdmmc_base_addr = *(u32 *)SDMMC_BASE;
// 获取通道，SD 有两个通道，这部分在文档上没有说明，这里我们也不关心。

    if(sdmmc_base_addr == SDMMC_CH0_BASE_ADDR)
        copy_bl2(0, MOVI_BL2_SDCARD_POS, MOVI_BL2_BLKCNT, (u32
*)CONFIG_SYS_TEXT_BASE, 0);
// 直接调用 copy_bl2，就可以调用到具体的函数指针了
// MOVI_BL2_SDCARD_POS 表示我们把 BL2 的镜像放到了哪个块上
// MOVI_BL2_BLKCNT 表示 BL2 占用的块长度
// CONFIG_SYS_TEXT_BASE=0x23E0_0000，定义在 include/configs/tiny210.h 中，表示我们要把 BL2 的镜像（uboot.bin）放到什么位置上。

    if(sdmmc_base_addr == SDMMC_CH2_BASE_ADDR)
        copy_bl2(2, MOVI_BL2_SDCARD_POS, MOVI_BL2_BLKCNT, (u32
*)CONFIG_SYS_TEXT_BASE, 0);
}
```

通过上述代码 uboot-spl 调用 copy_bl2_to_ddr 就可以实现把 uboot.bin 从 SD 中拷贝到 0x23E0_0000，后面跳转到 0x23E0_0000 后，就可以直接进入 uboot 了。
uboot 中把 kernel、rootfs、dtb 从 SD 中拷贝到对应位置上也是通过一样的方法，具体请自行参考代码 copy_kernel_to_ddr 的实现。

综上，就实现了利用 IROM 的代码从 SD 上拷贝镜像、代码到 ddr 中。
nand flash 后续实现之后再进行补充。

4. uboot 流程一概述

4.1. bootloader & uboot

4.1.1. bootloader 的概念

Bootloader 是在操作系统运行之前执行的一段小程序。而这段小程序的最终目的，正确地设置好软硬件环境，使之能够成功地引导操作系统。

4.1.2. bootloader 的核心功能

bootloader 的核心功能就是引导操作系统，部分工作如下

初始化部分硬件，包括时钟、内存等等

加载内核到内存上

加载文件系统、atags 或者 dtb 到内存上

根据操作系统启动要求正确配置好一些硬件

启动操作系统

4.1.3. bootloader 的 monitor 功能

上述 2 是 bootloader 的核心功能，也就是引导操作系统的功能。

但是部分 bootloader 还支持 monitor 功能，提供了更多的命令行接口，具体部分功能如下：

进行调试

读写内存

烧写 Flash

配置环境变量

命令引导操作系统

4.1.4. 嵌入式几种常见的 bootloader

uboot

这也是最常见的 bootloader，开源，常用于 ARM,MIPS 等平台。

支持 monitor 功能，也是在项目 project X 中，使用的 bootloader

所以后续两节会针对这个 bootloader 进行说明

superboot

不开源，友善之臂的 tiny210 代码中默认使用这个 bootloader

LK(Little Kernel)

常用于高通平台，支持 monitor 功能。

4.2. uboot-spl & uboot

4.2.1. uboot-spl

由 uboot 编译生成，对应于 BL1 阶段，也就是 BL1 的镜像，uboot-spl.bin。
根据《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》，其代码运行于 IRAM 中

- 主要工作有：

初始化部分时钟（和 SDRAM 相关）
初始化 DDR（外部 SDRAM）
从存储介质上（比如 SD\eMMC\nand flash）将 BL2 镜像加载到 SDRAM 上
验证 BL2 镜像的合法性
跳转到 BL2 镜像所在的地址上

后续会从编译和代码流程两方面来介绍 uboot-spl。
对应文章：
《[uboot] （第二章）uboot 流程——uboot-spl 编译流程》
《[uboot] （第三章）uboot 流程——uboot-spl 代码流程》

4.2.2. uboot

由 uboot 编译生成，对应于 BL2 阶段，也就是 BL2 的镜像,uboot.bin。
根据《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》，其代码运行于 SDRAM 中.

- 主要工作有：

初始化部分硬件，包括时钟、内存等等
加载内核到内存上
加载文件系统、atags 或者 dtb 到内存上
根据操作系统启动要求正确配置好一些硬件
启动操作系统

monitor 工作，主要是处理命令行的命令，以下是部分操作：

flash 操作

环境变量操作

启动操作

后续会从编译、代码整体流程以及部分功能的具体流程来介绍 uboot-spl。

对应文章：

《[uboot] （第四章）uboot 流程一一uboot 编译流程》

《[uboot] （第五章）uboot 流程一一uboot 整体代码流程》

5. uboot 流程一uboot-spl 编译流程

5.1. uboot-spl 编译和生成文件

spl 的编译是编译 uboot 的一部分，和 uboot.bin 走的是两条编译流程，这个要重点注意。

正常来说，会先编译主体 uboot，也就是 uboot.bin.再编译 uboot-spl，也就是 uboot-spl.bin,虽然编译命令是一起的，但是编译流程是分开的。

5.1.1. 编译方法

在 project X 项目中，所有镜像，包括 uboot、kernel、rootfs 都是放在 build 目录下进行编译的。具体去参考该项目 build 的 Makefile 的实现。

假设 config 已经配置完成，在 build 编译命令如下：

```
make uboot
```

Makefile 中对应的命令如下：

```
BUILD_DIR=$(shell pwd)
OUT_DIR=$(BUILD_DIR)/out
UBOOT_OUT_DIR=$(OUT_DIR)/u-boot
UBOOT_DIR=$(BUILD_DIR)/../u-boot
uboot:
    mkdir -p $(UBOOT_OUT_DIR)
```

make -C $(UBOOT_DIR) CROSS_COMPILE=$(CROSS_COMPILE) KBUILD_OUTPUT=$(UBOOT_OUT_DIR) $(BOARD_NAME)_defconfig

make -C $(UBOOT_DIR) CROSS_COMPILE=$(CROSS_COMPILE) KBUILD_OUTPUT=$(UBOOT_OUT_DIR)

-C $(UBOOT_DIR) 指定了要在../uboot，也就是 uboot 的代码根目录下执行 make

CROSS_COMPILE=$(CROSS_COMPILE) 指定了交叉编译器

KBUILD_OUTPUT=$(UBOOT_OUT_DIR) 指定了最终编译的输出目录是 build/out/u-boot.

最终，相当于进入了 uboot 目录执行了 make 动作。

也就是说 spl 的编译是编译 uboot 的一部分，和 uboot.bin 走的是两条编译流程，这个要重点注意。

正常来说，会先编译主体 uboot，也就是 uboot.bin.再编译 uboot-spl，也就是 uboot-spl.bin,虽然编译命令是一起的，但是编译流程是分开的。

5.1.2. 生成文件

最终编译完成之后，会在 project-x/build/out/u-boot/spl 下生成如下文件：

arch common dts include u-boot-spl u-boot-spl.cfg u-boot-spl.map

board drivers fs tiny210-spl.bin u-boot-spl.bin u-boot-spl.lds u-boot-spl-nodtb.bin

其中，arch、common、dts、include、board、drivers、fs 是对应代码的编译目录，各个目录下都会生成相应的 built.o，是由同目录下的目标文件连接而成。

重点说一下以下几个文件：

文件	说明
u-boot-spl	初步链接后得到的 spl 文件
u-boot-spl-nodtb.bin	在 u-boot-spl 的基础上，经过 objcopy 去除符号表信息之后的可执行程序
u-boot-spl.bin	在不需要 dtb 的情况下，直接由 u-boot-spl-nodtb.bin 复制而来，也就是编译 spl 的最终目标
tiny210-spl.bin	由 s5pv210 平台决定，需要在 u-boot-spl.bin 的基础上加上 16B 的 header 用作校验
u-boot-spl.lds	spl 的连接脚本

u-boot-spl.map	连接之后的符号表文件
u-boot-spl.cfg	由 spl 配置生成的文件

5.2. uboot-spl 编译流程

5.2.1. 编译整体流程

根据零、2 生成的文件说明可知简单流程如下：

（1）各目录下 built-in.o 的生成

源文件、代码文件 >编译、汇编->目标文件>同目录目标文件连接> built-in 目标文件

对应二、2（5）的实现

（2）由所有 built-in.o 以 u-boot-spl.lds 为连接脚本通过连接来生成 u-boot-spl

built-in->目标文件 ->以 u-boot-spl.lds 为连接脚本进行统一连接-> u-boot-spl

对应二、2（4）的实现

（3）由 u-boot-spl 生成 u-boot-spl-nodtb.bin

u-boot-spl ->objcopy 动作去掉符号信息表 ->u-boot-spl-nodtb.bin

对应二、2（3）的实现

（4）由 u-boot-spl-nodtb.bin 生成 u-boot-spl.bin，也就是 spl 的 bin 文件

u-boot-spl-nodtb.bin ->在不需要 dtb 的情况下,复制-> u-boot-spl.bin

对应二、2（2）的实现

后续的编译的核心过程就是按照上述的四个编译流程就是按照上述四个步骤来的。

5.2.2. 具体编译流程分析

我们直接从 make uboot 命令分析，也就是从 uboot 下的 Makefile 的依赖关系来分析整个编译流程。

注意，这个分析顺序和上述的整体编译流程的顺序是反着的。

（1）入口分析

在 project-x/u-boot/Makefile 中

```
all:          $(ALL-y)

ALL-$(CONFIG_SPL) += spl/u-boot-spl.bin
## 当配置了 CONFIG_SPL，make 的时候就会执行 spl/u-boot-spl.bin 这个目标

spl/u-boot-spl.bin: spl/u-boot-spl
      @:
spl/u-boot-spl: tools prepare $(if $(CONFIG_OF_SEPARATE),dts/dt.dtb)
      $(Q)$(MAKE) obj=spl -f $(srctree)/scripts/Makefile.spl all
## obj=spl 会在 out/u-boot 目录下生成 spl 目录
## -f $(srctree)/scripts/Makefile.spl 说明执行的 Makefile 文件是 scripts/Makefile.spl
## $(MAKE) all 相当于 make 的目标是 all
```

综上，由 CONFIG_SPL 来决定是否需要编译出 spl 文件，也就是 BL1。

后续相当于执行了 “make -f u-boot/scripts/Makefile.spl obj=spl all” 命令。

在 project-x/u-boot/scripts/Makefile.spl 中，

```
SPL_BIN := u-boot-spl
ALL-y   += $(obj)/$(SPL_BIN).bin $(obj)/$(SPL_BIN).cfg
## 所以最终目标是 spl/u-boot-spl.bin 和 spl/u-boot-spl.cfg
```

在 project-x/u-boot/scripts/Makefile.spl 中建立了 spl/u-boot-spl.bin 的依赖关系，后续 make 过程的主体都是在 Makefile.spl 中。

（2）spl/u-boot-spl.bin 的依赖关系

在 project-x/u-boot/scripts/Makefile.spl 中

$(obj)/$(SPL_BIN).bin: $(obj)/$(SPL_BIN)-nodtb.bin FORCE

$(call if_changed,copy)

$(obj)/$(SPL_BIN).bin 依赖于$(obj)/$(SPL_BIN)-nodtb.bin。

$(call if_changed,copy)表示当依赖文件发生变化时，直接把依赖文件复制为目标文件，即直接把$(obj)/$(SPL_BIN)-nodtb.bin 复制为$(obj)/$(SPL_BIN).bin

如上述 Makefile 代码 spl/u-boot-spl.bin 依赖于 spl/u-boot-spl-nodtb.bin，并且由 spl/u-boot-spl-nodtb.bin 复制而成。

对应于上述二、1（4）流程。

（3）spl/u-boot-spl-nodtb.bin 的依赖关系

在 project-x/u-boot/scripts/Makefile.spl 中

$(obj)/$(SPL_BIN)-nodtb.bin: $(obj)/$(SPL_BIN) FORCE

$(call if_changed,objcopy)

$(obj)/$(SPL_BIN)-nodtb.bin 依赖于$(obj)/$(SPL_BIN)。也就是 spl/u-boot-spl-nodtb.bin 依赖于 spl/u-boot-spl.

$(call if_changed,objcopy)表示当依赖文件发生变化时，将依赖文件经过 objcopy 处理之后得到目标文件。

也就是通过 objcopy 把 spl/u-boot-spl 的符号信息以及一些无用信息去掉之后，得到了 spl/u-boot-spl-nodtb.bin。

如上述 Makefile 代码 spl/u-boot-spl-nodtb.bin 依赖于 spl/u-boot-spl，并且由 spl/u-boot-spl 经过 objcopy 操作之后得到。

对应于上述二、1（3）流程。

（4）spl/u-boot-spl 的依赖关系

在 project-x/u-boot/scripts/Makefile.spl 中

$(obj)/$(SPL_BIN): $(u-boot-spl-init) $(u-boot-spl-main) $(obj)/u-boot-spl.lds FORCE

$(call if_changed,u-boot-spl)

$(call if_changed,u-boot-spl)来生成目标

$(call if_changed,u-boot-spl)对应 cmd_u-boot-spl 命令

如上，spl/u-boot-spl 依赖于$(u-boot-spl-init) 、$(u-boot-spl-main)和 spl/u-boot-spl.ld，并且最终会调用 cmd_u-boot-spl 来生成 spl/u-boot-spl。

cmd_u-boot-spl 实现如下：

quiet_cmd_u-boot-spl = LD $@

cmd_u-boot-spl = (cd $(obj) && $(LD) $(LDFLAGS) $(LDFLAGS_$(@F)) \

$(patsubst $(obj)/%,%,$(u-boot-spl-init)) --start-group \

$(patsubst $(obj)/%,%,$(u-boot-spl-main)) --end-group \

$(PLATFORM_LIBS) -Map $(SPL_BIN).map -o $(SPL_BIN))

将 cmd_u-boot-spl 通过 echo 命令打印出来之后得到如下（拆分出来看的）：

cmd_u-boot-spl=(

cd spl &&

/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-ld

-T u-boot-spl.lds --gc-sections -Bstatic --gc-sections

arch/arm/cpu/armv7/start.o

--start-group

arch/arm/mach-s5pc1xx/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/cpu/built-in.o arch/arm/lib/built-in.o board/samsung/tiny210/built-in.o board/samsung/common/built-in.o common/init/built-in.o drivers/built-in.o dts/built-in.o fs/built-in.o

--end-group

arch/arm/lib/eabi_compat.o

-L /home/disk3/xys/temp/project-x/build/arm-none-linux-gnueabi-4.8/bin/../lib/gcc/arm-none-linux-gnueabi/4.8.3 -lgcc

-Map u-boot-spl.map

-o u-boot-spl)

可以看出上述是一条连接命令，以 spl/u-boot-spl.ld 为链接脚本，把$(u-boot-spl-init) 、$(u-boot-spl-main)的指定的目标文件连接到 u-boot-spl 中。

连接很重要的东西就是连接标识，也就是 $(LD) $(LDFLAGS) $(LDFLAGS_$(@F)的定义。

尝试把$(LD) \$(LDFLAGS) \$(LDFLAGS_\$(@F)) 打印出来，结果如下：

LD=/home/disk3/xys/temp/project-x/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-ld

LDFLAGS=

LDFLAGS_u-boot-spl=-T u-boot-spl.lds --gc-sections -Bstatic --gc-sections

$(LDFLAGS_$(@F)对应于 LDFLAGS_u-boot-spl

也就是说在 LDFLAGS_u-boot-spl 中指定了链接脚本。

重点关注$(LDFLAGS_$(@F))的由来

@F 是一个自动化变量，提取目标的文件名，比如目标是$(obj)/$(SPL_BIN)，也就是 spl/u-boot-spl,那么@F 就是 u-boot-spl。

所以 LDFLAGS_$(@F)就是 LDFLAGS_u-boot-spl

定义如下

LDFLAGS_$(SPL_BIN) += -T u-boot-spl.lds $(LDFLAGS_FINAL)

ifneq ($(CONFIG_SPL_TEXT_BASE),)

LDFLAGS_$(SPL_BIN) += -Ttext $(CONFIG_SPL_TEXT_BASE)

endif

当指定 CONFIG_SPL_TEXT_BASE 时，会配置连接地址。在 tiny210 项目中，因为 spl 是地址无关代码设计，故没有设置连接地址。

$(LDFLAGS_FINAL)在如下几个地方定义了

./config.mk:19:LDFLAGS_FINAL :=

./config.mk:80:LDFLAGS_FINAL += -Bstatic

./arch/arm/config.mk:16:LDFLAGS_FINAL += --gc-sections

./scripts/Makefile.spl:43:LDFLAGS_FINAL += --gc-sections

综上：最后 LDFLAGS_u-boot-spl=-T u-boot-spl.lds --gc-sections -Bstatic --gc-sections 就可以理解了。

对应于上述二、1（2）流程。

（5）u-boot-spl-init & u-boot-spl-main 依赖关系（代码是如何被编译的）

先看一下这两个值打印出来的

u-boot-spl-init=spl/arch/arm/cpu/armv7/start.o

u-boot-spl-main= spl/arch/arm/mach-s5pc1xx/built-in.o spl/arch/arm/cpu/armv7/built-in.o spl/arch/arm/cpu/built-in.o spl/arch/arm/lib/built-in.o spl/board/samsung/tiny210/built-in.o spl/board/samsung/common/built-in.o spl/common/init/built-in.o spl/drivers/built-in.o spl/dts/built-in.o spl/fs/built-in.o

可以观察到是一堆目标文件的路径。这些目标文件最终都要被连接到 u-boot-spl 中。

u-boot-spl-init & u-boot-spl-main 的定义如下代码：

project-x/u-boot/scripts/Makefile.spl

```
u-boot-spl-init := $(head-y)
head-y          := $(addprefix $(obj)/,$(head-y))
## 加 spl 路径
## ./arch/arm/Makefile 定义了如下：
## head-y := arch/arm/cpu/$(CPU)/start.o

u-boot-spl-main := $(libs-y)
libs-y += $(if $(BOARDDIR),board/$(BOARDDIR)/)
libs-$(HAVE_VENDOR_COMMON_LIB) += board/$(VENDOR)/common/
libs-$(CONFIG_SPL_FRAMEWORK) += common/spl/
libs-y += common/init/
libs-$(CONFIG_SPL_LIBCOMMON_SUPPORT) += common/ cmd/
libs-$(CONFIG_SPL_LIBDISK_SUPPORT) += disk/
libs-y += drivers/
libs-y += dts/
libs-y += fs/
libs-$(CONFIG_SPL_LIBGENERIC_SUPPORT) += lib/
libs-$(CONFIG_SPL_POST_MEM_SUPPORT) += post/drivers/
libs-$(CONFIG_SPL_NET_SUPPORT) += net/
libs-y          := $(addprefix $(obj)/,$(libs-y))
## 加 spl 路径
u-boot-spl-dirs := $(patsubst %/,%,$(filter %/, $(libs-y)))
## 这里注意一下，u-boot-spl-dir 是 libs-y 没有加 built-in.o 后缀的时候被定义的。
libs-y := $(patsubst %/, %/built-in.o, $(libs-y))
## 加 built-in.o 文件后缀
```

那么 u-boot-spl-init & u-boot-spl-main 是如何生成的呢?

需要看一下对应的依赖如下：

$(sort $(u-boot-spl-init) $(u-boot-spl-main)): $(u-boot-spl-dirs) ;

也就是说$(u-boot-spl-init) $(u-boot-spl-main)依赖于$(u-boot-spl-dirs)

sort 函数根据首字母进行排序并去除掉重复的。

u-boot-spl-dirs := $(patsubst %/,%,$(filter %/, $(libs-y)))

$(filter %/, $(libs-y)过滤出'/'结尾的字符串，注意，此时$(libs-y)的内容还没有加上 built-in.o 文件后缀

patsubst 去掉字符串中最后的'/'的字符。

最后 u-boot-spl-dirs 打印出来如下：

u-boot-spl-dirs=spl/arch/arm/mach-s5pc1xx spl/arch/arm/cpu/armv7 spl/arch/arm/cpu spl/arch/arm/lib spl/board/samsung/tiny210 spl/board/samsung/common spl/common/init spl/drivers spl/dts spl/fs

也就是从 libs-y 改造而来的。

$(u-boot-spl-dirs) 的依赖规则如下：

$(u-boot-spl-dirs):

	(Q)(MAKE) $(build)=$@

也就是会对每一个目标文件依次执行 make $(build)=目标文件

$(build)定义如下：

project-x/u-boot/scripts/Kbuild.include

build := -f $(srctree)/scripts/Makefile.build obj

以 arch/arm/mach-s5pc1xx 为例

“$(MAKE) $(build)=$@”展开后格式如下

make -f ~/code/temp/project-x/u-boot/scripts/Makefile.build obj=spl/arch/arm/mach-s5pc1xx。

Makefile.build 定义 built-in.o、.lib 以及目标文件.o 的生成规则。这个 Makefile 文件生成了子目录的.lib、built-in.o 以及目标文件.o。

Makefile.build 第一个编译目标是__build，如下

PHONY := __build

__build:

所以会直接编译执行__build 这个目标，其依赖如下

__build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \

$(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \

$(subdir-ym) $(always)

@:

和 built-in.o 相关的是依赖 builtin-target。下面来看这个依赖。

builtin-target := $(obj)/built-in.o

以 obj=spl/arch/arm/mach-s5pc1xx 为例，那么 builtin-target 就是 spl/arch/arm/mach-s5pc1xx/built-in.o.

依赖关系如下：

$(builtin-target): $(obj-y) FORCE

$(call if_changed,link_o_target)

$(call if_changed,link_o_target)将所有依赖连接到$(builtin-target)，也就是相应的 built-in.o 中了。

具体实现可以查看 cmd_link_o_target 的实现，这里不详细说明了。

那么$(obj-y)是从哪里来的呢？是从相应目录下的 Makefile 中 include 得到的。

The filename Kbuild has precedence over Makefile

kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))

kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)

include $(kbuild-file)

当 obj=spl/arch/arm/mach-s5pc1xx 时,得到对应的 kbuild-file=u-boot/arch/arm/mach-s5pc1xx/Makefile

而在 u-boot/arch/arm/mach-s5pc1xx/Makefile 中定义了 obj-y 如下：

obj-y = cache.o

obj-y += reset.o

obj-y += clock.o

对应 obj-y 对应一些目标文件，由 C 文件编译而来，这里就不说明了。

对应于上述二、1（1）流程。

（6）spl/u-boot-spl.lds 依赖关系

这里主要是为了找到一个匹配的连接文件。

```
$(obj)/u-boot-spl.lds

$(obj)/u-boot-spl.lds: $(LDSCRIPT) FORCE

	$(call if_changed_dep,cpp_lds)

## 依赖于$(LDSCRIPT)，$(LDSCRIPT)定义了连接脚本所在的位置，

## 然后把链接脚本经过 cpp_lds 处理之后复制到$(obj)/u-boot-spl.lds 中，也就是 spl/u-boot-spl.lds 中。

## cpp_lds 处理具体实现看 cmd_cpu_lds 定义，具体是对应连接脚本里面的宏定义进行展开。

## $(LDSCRIPT)定义如下

ifeq ($(wildcard $(LDSCRIPT)),)

	LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot-spl.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

	LDSCRIPT := $(srctree)/$(CPUDIR)/u-boot-spl.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

	LDSCRIPT := $(srctree)/arch/$(ARCH)/cpu/u-boot-spl.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

$(error could not find linker script)

endif

## 也就是说依次从 board/板级目录、cpudir 目录、arch/架构/cpu/目录下去搜索 u-boot-spl.lds 文件。

## 例如，tiny210(s5vp210 armv7)最终会在./arch/arm/cpu/下搜索到 u-boot-spl.lds
```

综上，最终指定了 project-X/u-boot/arch/arm/cpu/u-boot-spl.lds 作为连接脚本。

5.3. 一些重点定义

1、CONFIG_SPL

在二、2（1）中说明。

用于指定是否需要编译 SPL，也就是是否需要编译出 uboot-spl.bin 文件

2、连接脚本的位置

在二、2（6）中说明。

对于 tiny210(s5pv210 armv7)来说，连接脚本的位置在

project-x/u-boot/arch/arm/cpu/u-boot-spl.lds

3、CONFIG_SPL_TEXT_BASE

在二、2（4）中说明。

用于指定 SPL 的连接地址，可以定义在板子对应的 config 文件中。

4、CONFIG_SPL_BUILD

在编译 spl 过程中，会配置

project-x/scripts/Makefile.spl 中定义了如下

KBUILD_CPPFLAGS += -DCONFIG_SPL_BUILD

也就是说在编译 uboot-spl.bin 的过程中，CONFIG_SPL_BUILD 这个宏是被定义的。

5.4. uboot-spl 链接脚本说明

1、连接脚本整体分析

相对比较简单，直接看连接脚本的内容 project-x/u-boot/arch/arm/cpu/u-boot-spl.lds

前面有一篇分析连接脚本的文章了《[kernel 启动流程] 前篇——vmlinux.lds 分析》，可以参考一下。

```
ENTRY(_start)
//定义了地址为_start 的地址，所以我们分析代码就是从这个函数开始分析的！！！
SECTIONS
{
   . = 0x00000000;

//以下定义文本段
   . = ALIGN(4);
   .text :
   {
      __image_copy_start = .;
//定义__image_copy_start 这个标号地址为当前地址
      *(.vectors)
//所有目标文件的 vectors 段，也就是中断向量表连接到这里来
      CPUDIR/start.o (.text*)
//start.o 文件的.text 段链接到这里来
      *(.text*)
//所有目标文件的.text 段链接到这里来
   }

//以下定义只读数据段
   . = ALIGN(4);
   .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }

//以下定义数据段
   . = ALIGN(4);
   .data : {
      *(.data*)
//所有目标文件的.data 段链接到这里来
   }
```

```
//以下定义 u_boot_list 段，具体功能未知
    . = ALIGN(4);
    .u_boot_list : {
        KEEP(*(SORT(.u_boot_list*)));
    }

    . = ALIGN(4);

    __image_copy_end = .;
//定义__image_copy_end 符号的地址为当前地址
//从__image_copy_start 到__image_copy_end 的区间，包含了代码段和数据段。

//以下定义 rel.dyn 段，这个段用于 uboot 资源重定向的时候使用，后面会说明
    .rel.dyn : {
        __rel_dyn_start = .;
//定义__rel_dyn_start 符号的地址为当前地址，后续在代码中会使用到
        *(.rel*)
        __rel_dyn_end = .;
//定义__rel_dyn_end 符号的地址为当前地址，后续在代码中会使用到
//从__rel_dyn_start 到__rel_dyn_end 的区间，应该是在代码重定向的过程中会使用到，后续遇到再说明。
    }

    .end :
    {
        *(.__end)
    }

    _image_binary_end = .;
```

//定义_image_binary_end 符号的地址为当前地址

//以下定义 bss 段

```
    .bss __rel_dyn_start (OVERLAY) : {
        __bss_start = .;
        *(.bss*)
         . = ALIGN(4);
        __bss_end = .;
    }
    __bss_size = __bss_end - __bss_start;
    .dynsym _image_binary_end : { *(.dynsym) }
    .dynbss : { *(.dynbss) }
    .dynstr : { *(.dynstr*) }
    .dynamic : { *(.dynamic*) }
    .hash : { *(.hash*) }
    .plt : { *(.plt*) }
    .interp : { *(.interp*) }
    .gnu : { *(.gnu*) }
    .ARM.exidx : { *(.ARM.exidx*) }
}
```

2、符号表中需要注意的符号

project-x/build/out/u-boot/spl/u-boot-spl.map

Linker script and memory map

```
.text           0x00000000      0xd10
                0x00000000                __image_copy_start = .
 *(.vectors)
                0x00000000                _start
                0x00000020                _undefined_instruction
                0x00000024                _software_interrupt
```

```
                0x00000028              _prefetch_abort
                0x0000002c              _data_abort
                0x00000030              _not_used
                0x00000034              _irq
                0x00000038              _fiq
...
                0x00000d10              __image_copy_end = .
.rel.dyn        0x00000d10      0x0
                0x00000d10              __rel_dyn_start = .
 *(.rel*)
 .rel.iplt      0x00000000      0x0 arch/arm/cpu/armv7/start.o
                0x00000d10              __rel_dyn_end = .
.end
 *(.__end)
                0x00000d10              _image_binary_end = .

.bss            0x00000d10      0x0
                0x00000d10              __bss_start = .
 *(.bss*)
                0x00000d10              . = ALIGN (0x4)
                0x00000d10              __bss_end = .
```

重点关注

* __image_copy_start & __image_copy_end

界定了代码的位置，用于重定向代码的时候使用，后面碰到了再分析。

* _start

在 u-boot-spl.lds 中 ENTRY(_start)，也就规定了代码的入口函数是_start。所以后续分析代码的时候就是从这里开始分析。

* __rel_dyn_start & __rel_dyn_end

* _image_binary_end

5.5. tiny210（s5pv210）的额外操作

1、为什么 tiny210 的 spl 需要额外操作？需要什么额外操作?

建议先参考《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》一文。

tiny210 只支持 SD 启动的方式和 NAND FLASH 启动的方式。

从《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》一文中，我们已经得知了当使用 SD 启动的方式和 NAND FLASH 启动的方式，也就是 BL1 镜像存放在 SD 上或者 nand flash 上时，s5pv210 中固化的 BL0，都需要对 BL1 的前 16B 的 header 做校验。BL1 就是我们所说的 uboot-spl.bin，但是默认编译出来的 uboot-spl.bin 就是一个纯粹的可执行文件，并没有加上特别的 header。

因此，我们需要在生成 uboot-spl.bin 之后，再为其加上 16B 的 header 后生成 tiny210-spl.bin。

16B 的 header 格式如下：

地址	数据
0xD002_0000	BL1 镜像包括 header 的长度
0xD002_0004	保留，设置为 0
0xD002_0008	BL1 镜像除去 header 的校验和
0xD002_000c	保留，设置为 0

2、如何生成 header？（如何生成 tiny210-spl.bin）

project-x/u-boot/scripts/Makefile.spl

```
ifdef CONFIG_SAMSUNG
ALL-y   += $(obj)/$(BOARD)-spl.bin
endif
```

当平台是 SAMSUNG 平台的时候，也就是 CONFIG_SAMSUNG 被定义的时候，就需要生成对应的板级 spl.bin 文件，例如 tiny210 的话，就应该生成对应的 spl/tiny210-spl.bin 文件。

```
ifdef CONFIG_S5PC110
$(obj)/$(BOARD)-spl.bin: $(obj)/u-boot-spl.bin
        $(objtree)/tools/mks5pc1xxspl $< $@
```

如果是 S5PC110 系列的 cpu 的话，则使用如上方法打上 header。tiny210 的 cpu 是 s5pv210 的，属于 S5PC110 系列，所以走的是这路。

$(objtree)/tools/mks5pc1xxspl 对应于编译 uboot 时生成的 build/out/u-boot/tools/mks5pc1xxspl

其代码路径位于 u-boot/tools/mks5pc1xxspl.c，会根据 s5pc1xx 系列的 header 规则为输入 bin 文件加上 16B 的 header，具体参考代码。

这里就构成了 u-boot-spl.bin 到 tiny210-spl.bin 的过程了。

else

$(obj)/$(BOARD)-spl.bin: $(obj)/u-boot-spl.bin

$(if $(wildcard $(objtree)/spl/board/samsung/$(BOARD)/tools/mk$(BOARD)spl),\

$(objtree)/spl/board/samsung/$(BOARD)/tools/mk$(BOARD)spl,\

$(objtree)/tools/mkexynosspl) $(VAR_SIZE_PARAM) $< $@

endif

endif

这里就构成了 u-boot-spl.bin 到 tiny210-spl.bin 的过程了。

综上，spl 的编译就完成了。

6. uboot 流程—uboot-spl 代码流程

6.1. 说明

6.1.1. uboot-spl 入口说明

通过 uboot-spl 编译脚本 project-X/u-boot/arch/arm/cpu/u-boot-spl.lds

ENTRY(_start)

所以 uboot-spl 的代码入口函数是_start
对应于路径 project-X/u-boot/arch/arm/lib/vector.S 的_start，后续就是从这个函数开始分析。

6.1.2. CONFIG_SPL_BUILD 说明

前面说过，在编译 SPL 的时候，编译参数会有如下语句：
project-X/u-boot/scripts/Makefile.spl

KBUILD_CPPFLAGS += -DCONFIG_SPL_BUILD

所以说在编译 SPL 的代码的过程中，CONFIG_SPL_BUILD 这个宏是打开的。
uboot-spl 和 uboot 的代码是通用的，其区别就是通过 CONFIG_SPL_BUILD 宏来进行区分的。

6.2. uboot-spl 需要做的事情

CPU 初始刚上电的状态。需要小心的设置好很多状态，包括 cpu 状态、中断状态、MMU 状态等等。
在 armv7 架构的 uboot-spl，主要需要做如下事情

- 关闭中断，svc 模式
- 禁用 MMU、TLB
- 芯片级、板级的一些初始化操作
    - IO 初始化
    - 时钟
    - 内存
    - 选项，串口初始化
    - 选项，nand flash 初始化
    - 其他额外的操作
- 加载 BL2，跳转到 BL2

上述工作，也就是 uboot-spl 代码流程的核心。

6.3. 代码流程

6.3.1. 代码整体流程

代码整体流程如下，以下列出来的就是 spl 核心函数。
_start————>reset————>关闭中断
……|
……————->cpu_init_cp15————>关闭 MMU,TLB
……|
……————->cpu_init_crit————->lowlevel_init————->平台级和板级的初始化
……|
……————->_main————>board_init_f_alloc_reserve & board_init_f_init_reserve & board_init_f————->加载 BL2,跳转到 BL2
board_init_f 执行时已经是 C 语言环境了。在这里需要结束掉 SPL 的工作，跳转到 BL2 中。

6.3.2. _start

上述已经说明了_start 是整个 spl 的入口，其代码如下：
arch/arm/lib/vector.S

```
_start:

#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG

   .word   CONFIG_SYS_DV_NOR_BOOT_CFG

#endif

   b   reset
```

会跳转到 reset 中。
注意，spl 的流程在 reset 中就应该被结束，也就是说在 reset 中，就应该转到到 BL2，也就是 uboot 中了。
后面看 reset 的实现。

6.3.3. reset

建议先参考[kernel 启动流程] （第二章）第一阶段之一一设置 SVC、关闭中断，了解一下为什么要设置 SVC、关闭中断以及如何操作。

代码如下：
arch/arm/cpu/armv7/start.S

```
    .globl  reset
    .globl  save_boot_params_ret


reset:
    /* Allow the board to save important registers */
    b   save_boot_params
save_boot_params_ret:
    /*
     * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
     * except if in HYP mode already
     */
    mrs r0, cpsr
    and r1, r0, #0x1f       @ mask mode bits
    teq r1, #0x1a       @ test for HYP mode
    bicne   r0, r0, #0x1f       @ clear all mode bits
    orrne   r0, r0, #0x13       @ set SVC mode
    orr r0, r0, #0xc0       @ disable FIQ and IRQ
    msr cpsr,r0
```

@@ 以上通过设置 CPSR 寄存器里设置 CPU 为 SVC 模式，禁止中断

@@ 具体操作可以参考《[kernel 启动流程] （第二章）第一阶段之一一设置 SVC、关闭中断》的分析

```
    /* the mask ROM code should have PLL and others stable */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl  cpu_init_cp15
```

@@ 调用 cpu_init_cp15，初始化协处理器 CP15,从而禁用 MMU 和 TLB。

@@ 后面会有一小节进行分析

```
    bl  cpu_init_crit
```

@@ 调用 cpu_init_crit，进行一些关键的初始化动作，也就是平台级和板级的初始化

@@ 后面会有一小节进行分析

```
#endif

    bl  _main
```

@@ 跳转到主函数，也就是要加载 BL2 以及跳转到 BL2 的主体部分

6.3.4. cpu_init_cp15

建议先参考[kernel 启动流程]（第六章）第一阶段之一一打开 MMU 两篇文章的分析。
cpu_init_cp15 主要用于对 cp15 协处理器进行初始化，其主要目的就是关闭其 MMU 和 TLB。
代码如下(去掉无关部分的代码):
arch/arm/cpu/armv7/start.S

```
ENTRY(cpu_init_cp15)
    /*
     * Invalidate L1 I/D
     */
    mov r0, #0          @ set up for MCR
    mcr p15, 0, r0, c8, c7, 0   @ invalidate TLBs
    mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
    mcr p15, 0, r0, c7, c5, 6   @ invalidate BP array
    mcr     p15, 0, r0, c7, c10, 4  @ DSB
    mcr     p15, 0, r0, c7, c5, 4   @ ISB
```

@@ 这里只需要知道是对 CP15 处理器的部分寄存器清零即可。

@@ 将协处理器的 c7\c8 清零等等，各个寄存器的含义请参考《ARM 的 CP15 协处理器的寄存器》

```
    /*
     * disable MMU stuff and caches
     */
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
    bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
    orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
    orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
    bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
#else
    orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
#endif
    mcr p15, 0, r0, c1, c0, 0
```

@@ 通过上述的文章的介绍，我们可以知道 cp15 的 c1 寄存器就是 MMU 控制器

@@ 上述对 MMU 的一些位进行清零和置位，达到关闭 MMU 和 cache 的目的，具体的话去看一下上述文章吧。

```
ENDPROC(cpu_init_cp15)
```

6.3.5. cpu_init_crit

cpu_init_crit，进行一些关键的初始化动作，也就是平台级和板级的初始化。其代码核心就是 lowlevel_init，如下
arch/arm/cpu/armv7/start.S

```
ENTRY(cpu_init_crit)
    /*
     * Jump to board specific initialization...
     * The Mask ROM will have already initialized
     * basic memory. Go here to bump up clock rate and handle
     * wake up conditions.
```

```
    */
    b   lowlevel_init       @ go setup pll,mux,memory
ENDPROC(cpu_init_crit)
```

所以说 **lowlevel_init** 就是这个函数的核心。
lowlevel_init 一般是由板级代码自己实现的。但是对于某些平台来说，也可以使用通用的 lowlevel_init，其定义在 arch/arm/cpu/lowlevel_init.S 中
以 tiny210 为例，在移植 tiny210 的过程中，就需要在 board/samsung/tiny210 下，也就是板级目录下面创建 lowlevel_init.S，在内部实现 lowlevel_init。（其实只要实现了 lowlevel_init 了就好，没必要说在哪里是实现，但是通常规范都是创建了 lowlevel_init.S 来专门实现 lowlevel_init 函数）。

在 lowlevel_init 中，我们要实现如下：
* 检查一些复位状态
* 关闭看门狗
* 系统时钟的初始化
* 内存、DDR 的初始化
* 串口初始化（可选）
* Nand flash 的初始化

下面以 tiny210 的 lowlevel_init 为例（这里说明一下，当时移植 tiny210 的时候，是直接把 kangear 的这个 lowlevel_init.S 文件拿过来用的）
这部分代码和平台相关性很强，简单介绍一下即可
board/samsung/tiny210/lowlevel_init.S

```
lowlevel_init:
    push    {lr}


    /* check reset status  */


    ldr r0, =(ELFIN_CLOCK_POWER_BASE+RST_STAT_OFFSET)
    ldr r1, [r0]
    bic r1, r1, #0xfff6ffff
    cmp r1, #0x10000
    beq wakeup_reset_pre
    cmp r1, #0x80000
    beq wakeup_reset_from_didle
```

@@ 读取复位状态寄存器 0xE010_a000 的值，判断复位状态。

```
    /* IO Retention release */
    ldr r0, =(ELFIN_CLOCK_POWER_BASE + OTHERS_OFFSET)
    ldr r1, [r0]
    ldr r2, =IO_RET_REL
    orr r1, r1, r2
    str r1, [r0]
```

@@ 读取混合状态寄存器 E010_e000 的值，对其中的某些位进行置位，复位后需要对某些 wakeup 位置 1，具体我也没搞懂。

```
    /* Disable Watchdog */
    ldr r0, =ELFIN_WATCHDOG_BASE    /* 0xE2700000 */
    mov r1, #0
    str r1, [r0]
```

@@ 关闭看门狗

@@ 这里忽略掉一部分对外部 SROM 操作的代码

```
    /* when we already run in ram, we don't need to relocate U-Boot.
     * and actually, memory controller must be configured before U-Boot
     * is running in ram.
     */
    ldr r0, =0x00ffffff
    bic r1, pc, r0      /* r0 <- current base addr of code */
    ldr r2, _TEXT_BASE      /* r1 <- original base addr in ram */
    bic r2, r2, r0      /* r0 <- current base addr of code */
    cmp     r1, r2                  /* compare r0, r1                */
    beq     1f          /* r0 == r1 then skip sdram init   */
```

@@ 判断是否已经在 SDRAM 上运行了，如果是的话，就跳过以下两个对 ddr 初始化的步骤

@@ 判断方法如下：

@@ 1、获取当前 pc 指针的地址，屏蔽其低 24bit，存放与 r1 中

@@ 2、获取_TEXT_BASE（CONFIG_SYS_TEXT_BASE）地址，也就是 uboot 代码段的链接地址，后续在 uboot 篇的时候会说明，并屏蔽其低 24bit

@@ 3、如果相等的话，就跳过 DDR 初始化的部分

```
    /* init system clock */
    bl system_clock_init
```

@@ 初始化系统时钟，后续有时间再研究一下具体怎么配置的

```
    /* Memory initialize */
    bl mem_ctrl_asm_init
```

@@ 重点注意：在这里初始化 DDR 的！！！后续会写一篇文章说明一下 s5pv210 平台如何初始化 DDR.

```
1:
    /* for UART */
    bl uart_asm_init
```

@@ 串口初始化，到这里串口会打印出一个'O'字符，后续通过写字符到 UTXH_OFFSET 寄存器中，就可以在串口上输出相应的字符。

```
    bl tzpc_init

#if defined(CONFIG_NAND)
    /* simple init for NAND */
    bl nand_asm_init
```

@@ 简单地初始化一下 NAND flash，有可能 BL2 的镜像是在 nand flash 上面的。

```
#endif

    /* Print 'K' */
    ldr r0, =ELFIN_UART_CONSOLE_BASE
    ldr r1, =0x4b4b4b4b
```

str r1, [r0, #UTXH_OFFSET]

@@ 再串口上打印‘K’字符，表示 lowlevel_init 已经完成

pop {pc}

@@ 弹出 PC 指针，即返回。

当串口中打印出‘OK’的字符的时候，说明 lowlevel_init 已经执行完成。
system_clock_init 是初始化时钟的地方。 mem_ctrl_asm_init 这个函数是初始化 DDR 的地方。后续应该有研究一下这两个函数。这里先有个印象。

6.3.6. _main

spl 的 main 的主要目标是调用 board_init_f 进行先前的板级初始化动作，在 tiny210 中，主要设计为，加载 BL2 到 DDR 上并且跳转到 BL2 中。DDR 在上述 lowlevel_init 中已经初始化好了。由于 board_init_f 是以 C 语言的方式实现，所以需要先构造 C 语言环境。
注意：**uboot-spl 和 uboot 的代码是通用的，其区别就是通过 CONFIG_SPL_BUILD 宏来进行区分的。**
所以以下代码中，我们只列出 spl 相关的部分，也就是被 CONFIG_SPL_BUILD 包含的部分。
arch/arm/lib/crt0.S

ENTRY(_main)

/*

* Set up initial C runtime environment and call board_init_f(0).

*/

@ 注意看这里的注释，也说明了以下代码的主要目的是，初始化 C 运行环境，调用 board_init_f。

ldr sp, =(CONFIG_SPL_STACK)

bic sp, sp, #7 /* 8-byte alignment for ABI compliance */

mov r0, sp

bl board_init_f_alloc_reserve

mov sp, r0

/* set up gd here, outside any C code */

mov r9, r0

bl board_init_f_init_reserve

```
    mov r0, #0
    bl  board_init_f

ENDPROC(_main)
```

代码拆分如下：

（1）因为后面是 C 语言环境，首先是设置堆栈

```
    ldr sp, =(CONFIG_SPL_STACK)
```

@@ 设置堆栈为 CONFIG_SPL_STACK

```
    bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
```

@@ 堆栈是 8 字节对齐，2^7bit=2^3byte=8byte

```
    mov r0, sp
```

@@ 把堆栈地址存放到 r0 寄存器中

关于 CONFIG_SPL_STACK，我们通过前面的文章《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》

我们已经知道 s5pv210 的 BL1(spl)是运行在 IRAM 的，并且 IRAM 的地址空间是 0xD002_0000-0xD003_7FFF，IRAM 前面的部分放的是 BL1 的代码部分，所以把 IRAM 最后的空间用来当作堆栈。

所以 CONFIG_SPL_STACK 定义如下：

include/configs/tiny210.h

```
#define CONFIG_SPL_STACK    0xD0037FFF
```

注意：上述还不是最终的堆栈地址，只是暂时的堆栈地址！！！

（2）为 GD 分配空间

```
    bl  board_init_f_alloc_reserve
```

@@ 把堆栈的前面一部分空间分配给 GD 使用

```
    mov sp, r0
```

@@ 重新设置堆栈指针 SP

```
    /* set up gd here, outside any C code */
    mov r9, r0
```

@@ 保存 GD 的地址到 r9 寄存器中

注意：虽然 sp 的地址和 GD 的地址是一样的，但是堆栈是向下增长的，而 GD 则是占用该地址后面的部分，所以不会有冲突的问题。
关于 GD，也就是 struct global_data，可以简单的理解为 uboot 的全局变量都放在了这里，比较重要，所以后续有会写篇文章说明一下 global_data。这里只需要知道在开始 C 语言环境的时候需要先为这个结构体分配空间。
board_init_f_alloc_reserve 实现如下
common/init/board_init.c

```
ulong board_init_f_alloc_reserve(ulong top)

{

   /* Reserve early malloc arena */

   /* LAST : reserve GD (rounded up to a multiple of 16 bytes) */

   top = rounddown(top-sizeof(struct global_data), 16);
```

// 现将 top（也就是 r0 寄存器，前面说过存放了暂时的指针地址），减去 sizeof(struct global_data)，也就是预留出一部分空间给 sizeof(struct global_data)使用。

// rounddown 表示向下 16 个字节对其

```
   return top;
```

// 到这里，top 就存放了 GD 的地址，也是 SP 的地址

//把 top 返回，注意，返回后，其实还是存放在了 r0 寄存器中。

}

还有一点，其实 GD 在 spl 中没什么使用，主要是用在 uboot 中，但在 uboot 中的时候还需要另外分配空间，在讲述 uboot 流程的时候会说明。

（3）初始化 GD 空间
前面说了，此时 r0 寄存器存放了 GD 的地址。

```
   bl  board_init_f_init_reserve
```

board_init_f_init_reserve 实现如下
common/init/board_init.c
编译 SPL 的时候_USE_MEMCPY 宏没有打开，所以我们去掉了_USE_MEMCPY 的无关部分。

```
void board_init_f_init_reserve(ulong base)

{

   struct global_data *gd_ptr;

   int *ptr;
```

```
    /*
     * clear GD entirely and set it up.
     * Use gd_ptr, as gd may not be properly set yet.
     */

    gd_ptr = (struct global_data *)base;
// 从 r0 获取 GD 的地址
    /* zero the area */
    for (ptr = (int *)gd_ptr; ptr < (int *)(gd_ptr + 1); )
        *ptr++ = 0;
// 对 GD 的空间进行清零
}
```

（4）跳转到板级前期的初始化函数中
如下代码

```
    bl  board_init_f
```

board_init_f 需要由板级代码自己实现。
在这个函数中，tiny210 主要是实现了从 SD 卡上加载了 BL2 到 ddr 上，然后跳转到 BL2 的相应位置上
tiny210 的实现如下：
board/samsung/tiny210/board.c

```
#ifdef CONFIG_SPL_BUILD
void board_init_f(ulong bootflag)
{
    __attribute__((noreturn)) void (*uboot)(void);
    int val;
#define DDR_TEST_ADDR 0x30000000
#define DDR_TEST_CODE 0xaa
    tiny210_early_debug(0x1);
    writel(DDR_TEST_CODE, DDR_TEST_ADDR);
    val = readl(DDR_TEST_ADDR);
    if(val == DDR_TEST_CODE)
```

```
        tiny210_early_debug(0x3);
    else
    {
        tiny210_early_debug(0x2);
        while(1);
    }
// 先测试 DDR 是否完成


    copy_bl2_to_ddr();
// 加载 BL2 的代码到 ddr 上


    uboot = (void *)CONFIG_SYS_TEXT_BASE;
// uboot 函数设置为 BL2 的加载地址上
    (*uboot)();
// 调用 uboot 函数，也就跳转到 BL2 的代码中
}
#endif
```

关于 copy_bl2_to_ddr 的实现，也就是如何从 SD 卡或者 nand flash 上加载 BL2 到 DDR 上的问题，请参考后续文章《[project X] tiny210(s5pv210)代码加载说明》。

到此，SPL 的任务就完成了，也已经跳到了 BL2 也就是 uboot 里面去了。

7. uboot 流程—uboot 编译流程

7.1. uboot 编译和生成文件

7.1.1. 说明

现在的 uboot 已经做得和 kernel 很像，最主要的一点是，uboot 也使用了 dtb 的方法，将设备树和代码分离开来（当然可以通过宏来控制）。
project-x/u-boot/configs/tiny210_defconfig

```
CONFIG_OF_CONTROL=y
// 用于表示是否使用了 dtb 的方式
```

CONFIG_OF_SEPARATE=y

// 是否将 dtb 和 uboot 分离表一

所以在 uboot 的编译中，和 spl 的最大区别是还要编译 dtb。（前面我们将的 spl 是没有使用 dtb 的，当然好像也可以使用 dtb，只是我没有试过）。

7.1.2. 编译方法

在 project X 项目中，所有镜像，包括 uboot、kernel、rootfs 都是放在 build 目录下进行编译的。具体去参考该项目 build 的 Makefile 的实现。
假设 config 已经配置完成，在 build 编译命令如下：

make uboot

Makefile 中对应的命令如下：
project-x/build/Makefile

BUILD_DIR=$(shell pwd)

OUT_DIR=$(BUILD_DIR)/out

UBOOT_OUT_DIR=$(OUT_DIR)/u-boot

UBOOT_DIR=$(BUILD_DIR)/../u-boot

uboot:

mkdir -p $(UBOOT_OUT_DIR)

make -C $(UBOOT_DIR) CROSS_COMPILE=$(CROSS_COMPILE) KBUILD_OUTPUT=$(UBOOT_OUT_DIR) $(BOARD_NAME)_defconfig

make -C $(UBOOT_DIR) CROSS_COMPILE=$(CROSS_COMPILE) KBUILD_OUTPUT=$(UBOOT_OUT_DIR)

-C $(UBOOT_DIR) 指定了要在../uboot，也就是 uboot 的代码根目录下执行 make

CROSS_COMPILE=$(CROSS_COMPILE) 指定了交叉编译器

KBUILD_OUTPUT=$(UBOOT_OUT_DIR) 指定了最终编译的输出目录是 build/out/u-boot.

最终，相当于进入了 uboot 目录执行了 make 动作。

7.1.3. 生成文件

最终编译完成之后，会在 project-x/build/out/u-boot 下生成如下文件：

arch common dts include net tools u-boot.cfg u-boot.lds u-boot.srec

board disk examples lib scripts System.map u-boot u-boot.dtb u-boot.map u-boot.sym

cmd drivers fs Makefile source test u-boot.bin u-boot-dtb.bin u-boot-nodtb.bin

其中，arch、common、dts、include、board、drivers、fs 等等目录是对应代码的编译目录，**各个目录下都会生成相应的 built.o，是由同目录下的目标文件连接而成。**
重点说一下以下几个文件：

文件	说明
u-boot	初步链接后得到的 uboot 文件
u-boot-nodtb.bin	在 u-boot 的基础上，经过 objcopy 去除符号表信息之后的可执行程序
u-boot.dtb	dtb 文件
u-boot-dtb.bin	将 u-boot-nodtb.bin 和 u-boot.dtb 打包在一起的文件
u-boot.bin	在需要 dtb 的情况下，直接由 u-boot-dtb.bin 复制而来，也就是编译 u-boot 的最终目标
u-boot.lds	uboot 的连接脚本
System.map	连接之后的符号表文件
u-boot.cfg	由 uboot 配置生成的文件

7.2. uboot 编译流程

7.2.1. 编译整体流程

根据一、2 生成的文件说明可知简单流程如下：

（1）各目录下 built-in.o 的生成

源文件、代码文件->编译、汇编->目标文件->同目录目标文件连接->built-in 目标文件

（2）由所有 built-in.o 以 u-boot.lds 为连接脚本通过连接来生成 u-boot

built-in 目标文件->以 u-boot.lds 为连接脚本进行统一连接->u-boot

（3）由 u-boot 生成 u-boot-nodtb.bin

u-boot→objcopy 动作去掉符号信息表→>u-boot-nodtb.bin

（4）由生成 uboot 的 dtb 文件

dts 文件→dtc 编译打包→dtb 文件 u-boot.dtb

（5）由 u-boot-nodtb.bin 和 u-boot.dtb 生成 u-boot-dtb.bin

u-boot-nodtb.bin 和 u-boot.dtb->追加整合两个文件->u-boot-dtb.bin

（6）由 u-boot-dtb.bin 复制生成 u-boot.bin

u-boot-dtb.bin->复制->u-boot.bin

7.2.2. 具体编译流程分析

我们直接从 make uboot 命令分析，也就是从 uboot 下的 Makefile 的依赖关系来分析整个编译流程。
注意，这个分析顺序和上述的整体编译流程的顺序是反着的。

- **（1）入口分析**
  在 project-x/u-boot/Makefile 中

all: $(ALL-y)

- ALL-y += u-boot.srec u-boot.bin u-boot.sym System.map u-boot.cfg binary_size_check

u-boot.bin 就是我们的目标，所以后需要主要研究 u-boot.bin 的依赖关系。

- **（2）u-boot.bin 的依赖关系**
  在 project-x/u-boot/Makefile 中

ifeq ($(CONFIG_OF_SEPARATE),y)

CONFIG_OF_SEPARATE 用于定义是否有 DTB 并且是否是和 uboot 分开编译的。

tiny210 是有定义这个宏的，所以走的是上面这路

u-boot-dtb.bin: u-boot-nodtb.bin dts/dt.dtb FORCE

$(call if_changed,cat)

由 u-boot-nodtb.bin 和 dts/dt.dtb 连接在一起，先生成 u-boot-dtb.bin

$(call if_changed,cat)会调用到 cmd_cat 函数，具体实现我们不分析了

u-boot.bin: u-boot-dtb.bin FORCE

$(call if_changed,copy)

直接将 u-boot-dtb.bin 复制为 u-boot.bin

$(call if_changed,copy)会调用到 cmd_copy 函数，具体实现我们不分析了

else

u-boot.bin: u-boot-nodtb.bin FORCE

$(call if_changed,copy)

endif

对应于上述二、1（5）流程和上述二、1（6）流程。
后续有两个依赖关系要分析，分别是 u-boot-nodtb.bin 和 dts/dt.dtb。
u-boot-nodtb.bin 依赖关系参考下述二、2（3）-2（6）.
dts/dt.dtb 依赖关系参考下述二、2（7）
其中 u-boot-nodtb.bin 的依赖关系和 SPL 的相当类似，可以先参考一下《[uboot] （第二章）uboot 流程一一uboot-spl 编译流程》。

- **（3）u-boot-nodtb.bin 的依赖关系**
  在 project-x/u-boot/Makefile 中

u-boot-nodtb.bin: u-boot FORCE

$(call if_changed,objcopy)

$(call DO_STATIC_RELA,$<,$@,$(CONFIG_SYS_TEXT_BASE))

$(BOARD_SIZE_CHECK)

$(call if_changed,objcopy)表示当依赖文件发生变化时，将依赖文件经过 objcopy 处理之后得到目标文件。

也就是通过 objcopy 把 u-boot 的符号信息以及一些无用信息去掉之后，得到了 u-boot-nodtb.bin。

如上述 Makefile 代码 u-boot-nodtb.bin 依赖于 u-boot，并且由 u-boot 经过 objcopy 操作之后得到。

对应于上述二、1（3）流程.

- **（4）u-boot 的依赖关系**
  在 project-x/u-boot/Makefile 中

u-boot: $(u-boot-init) $(u-boot-main) u-boot.lds FORCE

$(call if_changed,u-boot__)

$(call if_changed,u-boot__)来生成目标

$(call if_changed,u-boot__)对应 cmd_u-boot__命令

如上，u-boot 依赖于$(u-boot-init) 、$(u-boot-main)和 u-boot.lds，并且最终会调用 cmd_u-boot__来生成 u-boot。
cmd_u-boot__实现如下
project-x/u-boot/Makefile

cmd_u-boot__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_u-boot) -o $@ \

-T u-boot.lds $(u-boot-init) \

--start-group $(u-boot-main) --end-group \

$(PLATFORM_LIBS) -Map u-boot.map

将 cmd_u-boot__通过 echo 命令打印出来之后得到如下（拆分出来看的）：
project-x/u-boot/Makefile

/project-x/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-ld

-pie --gc-sections -Bstatic -Ttext 0x23E00000

-o u-boot

-T u-boot.lds

arch/arm/cpu/armv7/start.o

--start-group

arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/lib/built-in.o arch/arm/mach-s5pc1xx/built-in.o board/samsung/common/built-in.o board/samsung/tiny210/built-in.o cmd/built-in.o common/built-in.o disk/built-in.o drivers/built-in.o drivers/dma/built-in.o drivers/gpio/built-in.o drivers/i2c/built-in.o drivers/mmc/built-in.o drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o drivers/mtd/spi/built-in.o drivers/net/built-in.o drivers/net/phy/built-in.o drivers/pci/built-in.o drivers/power/built-in.o drivers/power/battery/built-in.o drivers/power/fuel_gauge/built-in.o drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o drivers/power/regulator/built-in.o drivers/serial/built-in.o drivers/spi/built-in.o drivers/usb/common/built-in.o drivers/usb/dwc3/built-in.o drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o drivers/usb/ulpi/built-in.o fs/built-in.o lib/built-in.o net/built-in.o test/built-in.o test/dm/built-in.o

--end-group

arch/arm/lib/eabi_compat.o

-L /project-x/build/arm-none-linux-gnueabi-4.8/bin/../lib/gcc/arm-none-linux-gnueabi/4.8.3 -lgcc

-Map u-boot.map

可以看出上述是一条连接命令，以 u-boot.lds 为链接脚本，把$(u-boot-init) 、$(u-boot-main)的指定的目标文件连接到 u-boot 中。
并且已经指定输出文件为 u-boot，连接脚本为 u-boot.lds。
连接很重要的东西就是连接标识，也就是 $(LD) $(LDFLAGS) $(LDFLAGS_u-boot)的定义。
尝试把$(LD) \$(LDFLAGS) \$(LDFLAGS_u-boot)) 打印出来，结果如下：

LD=~/project-x/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-ld

LDFLAGS=

LDFLAGS_u-boot=-pie --gc-sections -Bstatic -Ttext 0x23E00000

LDFLAGS_u-boot 定义如下

LDFLAGS_u-boot += -pie

LDFLAGS_u-boot += $(LDFLAGS_FINAL)

ifneq ($(CONFIG_SYS_TEXT_BASE),)

LDFLAGS_u-boot += -Ttext $(CONFIG_SYS_TEXT_BASE)

endif

当指定 CONFIG_SYS_TEXT_BASE 时，会配置连接地址。在 tiny210 项目中，定义如下：

./include/configs/tiny210.h:52:#define CONFIG_SYS_TEXT_BASE 0x23E00000

$(LDFLAGS_FINAL)在如下几个地方定义了

./config.mk:19:LDFLAGS_FINAL :=

./config.mk:80:LDFLAGS_FINAL += -Bstatic

./arch/arm/config.mk:16:LDFLAGS_FINAL += --gc-sections

通过上述 LDFLAGS_u-boot=-pie --gc-sections -Bstatic -Ttext 0x23E00000 也就可以理解了

对应于上述二、1（2）流程。

对应于上述二、1（2）流程。
关于 u-boot 依赖的说明在（5）、（6）中继续介绍

- **（5）u-boot-init & u-boot-main 依赖关系（代码是如何被编译的）**
  先看一下这两个值打印出来的

  u-boot-init=arch/arm/cpu/armv7/start.o

  u-boot-main= arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/lib/built-in.o arch/arm/mach-s5pc1xx/built-in.o board/samsung/common/built-in.o board/samsung/tiny210/built-in.o cmd/built-in.o common/built-in.o disk/built-in.o drivers/built-in.o drivers/dma/built-in.o drivers/gpio/built-in.o drivers/i2c/built-in.o drivers/mmc/built-in.o drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o drivers/mtd/spi/built-in.o drivers/net/built-in.o drivers/net/phy/built-in.o drivers/pci/built-in.o drivers/power/built-in.o drivers/power/battery/built-in.o drivers/power/fuel_gauge/built-in.o drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o drivers/power/regulator/built-in.o drivers/serial/built-in.o drivers/spi/built-in.o drivers/usb/common/built-in.o drivers/usb/dwc3/built-in.o drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o drivers/usb/ulpi/built-in.o fs/built-in.o lib/built-in.o net/built-in.o test/built-in.o test/dm/built-in.o

可以观察到是一堆目标文件的路径。这些目标文件最终都要被连接到 u-boot 中。
u-boot-init & u-boot-main 的定义如下代码：
project-x/u-boot/Makefile

u-boot-init := $(head-y)

head-y 定义在如下位置

./arch/arm/Makefile:73:head-y := arch/arm/cpu/$(CPU)/start.o

libs-y += lib/

libs-y += fs/

libs-y += net/

libs-y += disk/

libs-y += drivers/

libs-y += drivers/dma/

libs-y += drivers/gpio/

libs-y += drivers/i2c/

...

u-boot-dirs	:= $(patsubst %/,%,$(filter %/, $(libs-y))) tools examples

过滤出路径之后，加上 tools 目录和 example 目录

libs-y		:= $(patsubst %/, %/built-in.o, $(libs-y))

先加上后缀 built-in.o

u-boot-main := $(libs-y)

那么 u-boot-init & u-boot-main 是如何生成的呢?
需要看一下对应的依赖如下：

$(sort $(u-boot-init) $(u-boot-main)): $(u-boot-dirs) ;

也就是说$(u-boot-init) $(u-boot--main)依赖于$(u-boot-dirs)

sort 函数根据首字母进行排序并去除掉重复的。

##u-boot-dirs	:= $(patsubst %/,%,$(filter %/, $(libs-y))) tools examples

$(filter %/, $(libs-y)过滤出'/'结尾的字符串，注意，此时$(libs-y)的内容还没有加上 built-in.o 文件后缀

patsubst 去掉字符串中最后的'/'的字符。

最后 u-boot-dirs 打印出来如下：

u-boot-dirs=arch/arm/cpu arch/arm/cpu/armv7 arch/arm/lib arch/arm/mach-s5pc1xx board/samsung/common board/samsung/tiny210 cmd common disk drivers drivers/dma drivers/gpio drivers/i2c drivers/mmc drivers/mtd drivers/mtd/onenand drivers/mtd/spi drivers/net drivers/net/phy drivers/pci drivers/power drivers/power/battery drivers/power/fuel_gauge drivers/power/mfd drivers/power/pmic drivers/power/regulator drivers/serial drivers/spi drivers/usb/common drivers/usb/dwc3 drivers/usb/emul drivers/usb/eth drivers/usb/gadget drivers/usb/gadget/udc

drivers/usb/host drivers/usb/musb-new drivers/usb/musb drivers/usb/phy drivers/usb/ulpi fs lib net test test/dm tools examples

u-boot-dirs 依赖规则如下：

PHONY += $(u-boot-dirs)

$(u-boot-dirs): prepare scripts

(Q)(MAKE) $(build)=$@

依赖于 prepare scripts

prepare 会导致 prepare0、prepare1、prepare2、prepare3 目标被执行，最终编译了 tools 目录下的东西，生成了一些工具

然后执行(Q)(MAKE) $(build)=$@

也就是会对每一个目标文件依次执行 make \$(build)=目标文件

对每一个目标文件依次执行 make $(build)=目标文件
$(build)定义如下：
project-x/u-boot/scripts/Kbuild.include

build := -f $(srctree)/scripts/Makefile.build obj

以 arch/arm/mach-s5pc1xx 为例
“$(MAKE) $(build)=$@”展开后格式如下
make -f project-x/u-boot/scripts/Makefile.build obj=arch/arm/mach-s5pc1xx。

Makefile.build 定义 built-in.o、.lib 以及目标文件.o 的生成规则。这个 Makefile 文件生成了子目录的.lib、built-in.o 以及目标文件.o。
Makefile.build 第一个编译目标是__build，如下

PHONY := __build

__build:

所以会直接编译执行__build 这个目标，其依赖如下

__build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \

$(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \

$(subdir-ym) $(always)

@:

和 built-in.o 相关的是依赖 builtin-target。下面来看这个依赖。

builtin-target := $(obj)/built-in.o

以 obj=arch/arm/mach-s5pc1xx 为例，那么 builtin-target 就是 arch/arm/mach-s5pc1xx/built-in.o.

依赖关系如下：

$(builtin-target): $(obj-y) FORCE

    $(call if_changed,link_o_target)

$(call if_changed,link_o_target)将所有依赖连接到$(builtin-target)，也就是相应的 built-in.o 中了。

具体实现可以查看 cmd_link_o_target 的实现，这里不详细说明了。

那么$(obj-y)是从哪里来的呢？是从相应目录下的 Makefile 中 include 得到的。

The filename Kbuild has precedence over Makefile

kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))

kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)

include $(kbuild-file)

当 obj=arch/arm/mach-s5pc1xx 时,得到对应的 kbuild-file=u-boot/arch/arm/mach-s5pc1xx/Makefile

而在 u-boot/arch/arm/mach-s5pc1xx/Makefile 中定义了 obj-y 如下：

obj-y = cache.o

obj-y += reset.o

obj-y += clock.o

对应 obj-y 对应一些目标文件，由 C 文件编译而来，这里就不说明了。

后面来看目标文件的编译流程
./scripts/Makefile.build/scripts/Makefile.build

Built-in and composite module parts

$(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE

    $(call cmd,force_checksrc)

    $(call if_changed_rule,cc_o_c)

调用 cmd_cc_o_c 对.c 文件进行编译

cmd_cc_o_c 格式如下：

cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<

$(CC) $(c_flags)打印出来如下：

CC=/home/disk3/xys/temp/project-x/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-gcc

c_flags=-Wp,-MD,arch/arm/mach-s5pc1xx/.clock.o.d -nostdinc -isystem /home/disk3/xys/temp/project-x/build/arm-none-linux-gnueabi-4.8/bin/../lib/gcc/arm-none-linux-gnueabi/4.8.3/include -Iinclude -I/home/disk3/xys/temp/project-x/u-boot/include -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/include -include /home/disk3/xys/temp/project-x/u-boot/include/linux/kconfig.h -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/mach-s5pc1xx -Iarch/arm/mach-s5pc1xx -D__KERNEL__ -D__UBOOT__ -Wall -Wstrict-prototypes -Wno-format-security -fno-builtin -ffreestanding -Os -fno-stack-protector -fno-delete-null-pointer-checks -g -fstack-usage -Wno-format-nonliteral -D__ARM__ -marm -mno-thumb-interwork -mabi=aapcs-linux -mword-relocations -fno-pic -mno-unaligned-access -ffunction-sections -fdata-sections -fno-common -ffixed-r9 -msoft-float -pipe -march=armv7-a -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/mach-s5pc1xx/include -DKBUILD_STR(s)=#s -DKBUILD_BASENAME=KBUILD_STR(clock) -DKBUILD_MODNAME=KBUILD_STR(clock)

对应于上述二、1（1）流程。

- **（6）u-boot.lds 依赖关系**
  这里主要是为了找到一个匹配的连接文件。

```
u-boot.lds: $(LDSCRIPT) prepare FORCE
    $(call if_changed_dep,cpp_lds)
ifndef LDSCRIPT
    ifeq ($(wildcard $(LDSCRIPT)),)
        LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot.lds
    endif
    ifeq ($(wildcard $(LDSCRIPT)),)
        LDSCRIPT := $(srctree)/$(CPUDIR)/u-boot.lds
    endif
    ifeq ($(wildcard $(LDSCRIPT)),)
        LDSCRIPT := $(srctree)/arch/$(ARCH)/cpu/u-boot.lds
    endif
endif
```

也就是说依次从 board/板级目录、cpudir 目录、arch/架构/cpu/目录下去搜索 u-boot.lds 文件。

例如，tiny210(s5vp210 armv7)最终会在./arch/arm/cpu/下搜索到 u-boot.lds

综上，最终指定了 project-X/u-boot/arch/arm/cpu/u-boot.lds 作为连接脚本。

- **（7）dts/dt.dtb 依赖关系**
  该依赖关系的主要目的是生成 dtb 文件。
  首先了解 dts 文件被放在了 arch/arm/dts 里面，并通过 dts 下的 Makefile 进行选择。
  Makefile 如下（剪切出一部分）
  project-X/u-boot/arch/arm/dts/Makefile

```
dtb-$(CONFIG_S5PC110) += s5pc1xx-goni.dtb
dtb-$(CONFIG_EXYNOS5) += exynos5250-arndale.dtb \
    exynos5250-snow.dtb \
    exynos5250-spring.dtb \
    exynos5250-smdk5250.dtb \
    exynos5420-smdk5420.dtb \
    exynos5420-peach-pit.dtb \
    exynos5800-peach-pi.dtb \
    exynos5422-odroidxu3.dtb
dtb-$(CONFIG_TARGET_TINY210) += \
    s5pv210-tiny210.dtb
## 填充选择 dtb-y

targets += $(dtb-y)

# Add any required device tree compiler flags here
DTC_FLAGS +=
## 用于添加 DTC 编译选项

PHONY += dtbs
dtbs: $(addprefix $(obj)/, $(dtb-y))
    @:
## 伪目标，其依赖为$(dtb-y)加上了源路径，如下
## arch/arm/dts/s5pc1xx-goni.dtb
```

arch/arm/dts/s5pv210-tiny210.dtb

后续会使用到这个伪目标

接下来看一下 dts/dt.dtb 的依赖关系

dtbs dts/dt.dtb: checkdtc u-boot

(Q)(MAKE) $(build)=dts dtbs

checkdtc 依赖用于检查 dtc 的版本

u-boot 一旦发生变化那么就重新编译一遍 dtb

重点关注命令 (Q)(MAKE) $(build)=dts dtbs

展开来就是 make -f ~/project-x/u-boot/scripts/Makefile.build obj=dts dtbs

我们相当于值在/scripts/Makefile.build 下执行了目标 dtbs

在 scripts/Makefile.build 中 dtbs 的目标定义在哪里呢
project-X/u-boot/scripts/Makefile.build

kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)

include $(kbuild-file)

把对应的 Makefile 路径包含了进去，也就是 arch/arm/dts/Makefile

如前面所说，arch/arm/dts/Makefile 中定义了 dtbs 的目标

dtbs: $(addprefix $(obj)/, $(dtb-y))

@:

这里我们就找到对应的依赖关系了，依赖就是$(obj)/, $(dtb-y)，举个例子就是 arch/arm/dts/s5pv210-tiny210.dtb

include scripts/Makefile.lib

包含了 scripts/Makefile.lib，在编译 dts 的时候会用到

接下来就是$(obj)/, $(dtb-y)的依赖关系了
project-X/u-boot/scripts/Makefile.lib

$(obj)/%.dtb: $(src)/%.dts FORCE

$(call if_changed_dep,dtc)

使用了通配符的方式

这样就通过 dtc 对 dts 编译生成了 dtb 文件

对应于上述二、1（4）流程。

7.3. 一些重点定义

- 1、**连接标志**
  在二、2（4）中说明。
  连接命令在 cmd_u-boot__中，如下

  cmd_u-boot__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_u-boot) -o $@ \

  -T u-boot.lds $(u-boot-init) \

  --start-group $(u-boot-main) --end-group \

  $(PLATFORM_LIBS) -Map u-boot.map

连接标识如下:

LD=~/project-x/build/arm-none-linux-gnueabi-4.8/bin/arm-none-linux-gnueabi-ld

LDFLAGS=

LDFLAGS_u-boot=-pie --gc-sections -Bstatic -Ttext 0x23E00000

LDFLAGS_u-boot 定义如下

LDFLAGS_u-boot += -pie

LDFLAGS_u-boot += $(LDFLAGS_FINAL)

ifneq ($(CONFIG_SYS_TEXT_BASE),)

LDFLAGS_u-boot += -Ttext $(CONFIG_SYS_TEXT_BASE)

endif

‘-o’指定了输出文件是 u-boot，’ -T’是指定了连接脚本是当前目录下的 u-boot.lds, -Ttext 指定了连接地址是 CONFIG_SYS_TEXT_BASE。

- 2、**连接地址**
  在二、2（4）中说明。
  CONFIG_SYS_TEXT_BASE 指定了 u-boot.bin 的连接地址。这个地址也就是 uboot 的起始运行地址。
  对于 tiny210，其定义如下（可以进行修改）
  /include/configs/tiny210.h

  #define CONFIG_SYS_TEXT_BASE 0x23E00000

- 3、**连接脚本**
  在二、2（6）中说明。
  u-boot/arch/arm/cpu/u-boot.lds

  u-boot.lds: $(LDSCRIPT) prepare FORCE

  $(call if_changed_dep,cpp_lds)

ifndef LDSCRIPT

```
        ifeq ($(wildcard $(LDSCRIPT)),)
            LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot.lds
        endif
        ifeq ($(wildcard $(LDSCRIPT)),)
            LDSCRIPT := $(srctree)/$(CPUDIR)/u-boot.lds
        endif
        ifeq ($(wildcard $(LDSCRIPT)),)
            LDSCRIPT := $(srctree)/arch/$(ARCH)/cpu/u-boot.lds
        endif
    endif
```

综上，最终指定了 project-X/u-boot/arch/arm/cpu/u-boot.lds 作为连接脚本。

7.4. uboot 链接脚本说明

7.4.1. 连接脚本整体分析

相对比较简单，直接看连接脚本的内容 project-x/u-boot/arch/arm/cpu/u-boot.lds
前面有一篇分析连接脚本的文章了《[kernel 启动流程] 前篇——vmlinux.lds 分析》，可以参考一下。
参考如下，只提取了一部分：

```
ENTRY(_start)
//定义了地址为_start 的地址，所以我们分析代码就是从这个函数开始分析的！！！
   . = 0x00000000;


//以下定义文本段
   . = ALIGN(4);
   .text :
   {
      __image_copy_start = .;
//定义__image_copy_start 这个标号地址为当前地址
      *(.vectors)
//所有目标文件的 vectors 段，也就是中断向量表连接到这里来
      CPUDIR/start.o (.text*)
```

```
//start.o 文件的.text 段链接到这里来
        *(.text*)
//所有目标文件的.text 段链接到这里来
    }

//以下定义只读数据段
    . = ALIGN(4);
    .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }

//以下定义数据段
    . = ALIGN(4);
    .data : {
        *(.data*)
//所有目标文件的.data 段链接到这里来
    }

    . = ALIGN(4);

//以下定义 u_boot_list 段，具体功能未知
    . = ALIGN(4);
    .u_boot_list : {
        KEEP(*(SORT(.u_boot_list*)));
    }

    . = ALIGN(4);

    .image_copy_end :
    {
        *(.__image_copy_end)
    }
```

//定义__image_copy_end 符号的地址为当前地址

//从__image_copy_start 到__image_copy_end 的区间，包含了代码段和数据段。

```
    .rel_dyn_start :
    {
        *(.__rel_dyn_start)
    }
```

//定义__rel_dyn_start 符号的地址为当前地址，后续在代码中会使用到

```
    .rel.dyn : {
        *(.rel*)
    }

    .rel_dyn_end :
    {
        *(.__rel_dyn_end)
    }
```

//定义__rel_dyn_end 符号的地址为当前地址，后续在代码中会使用到

//从__rel_dyn_start 到__rel_dyn_end 的区间，应该是在代码重定向的过程中会使用到，后续遇到再说明。

```
    .end :
    {
        *(.__end)
    }

    _image_binary_end = .;
```

//定义_image_binary_end 符号的地址为当前地址

// 以下定义堆栈段

```
    .bss_start __rel_dyn_start (OVERLAY) : {
        KEEP(*(.__bss_start));
        __bss_base = .;
    }

    .bss __bss_base (OVERLAY) : {
        *(.bss*)
         . = ALIGN(4);
         __bss_limit = .;
    }

    .bss_end __bss_limit (OVERLAY) : {
        KEEP(*(.__bss_end));
    }
}
```

7.4.2. 以下以.vectors 段做说明，

.vectors 是 uboot 链接脚本第一个链接的段，也就是_start 被链接进来的部分，也负责链接异常中断向量表
先看一下代码 project-x/u-boot/arch/arm/lib/vectors.S

```
.globl _start
    .section ".vectors", "ax"
@@ 定义在.vectors 段中


_start:


    b   reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
```

```
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq


    .globl  _undefined_instruction
    .globl  _software_interrupt
    .globl  _prefetch_abort
    .globl  _data_abort
    .globl  _not_used
    .globl  _irq
    .globl  _fiq
```

@@ 定义了异常中断向量表

通过“arm-none-linux-gnueabi-objdump -D u-boot > uboot_objdump.txt”进行反编译之后，得到了如下指令

```
23e00000 <__image_copy_start>:
23e00000:   ea0000be    b   23e00300 <reset>
23e00004:   e59ff014    ldr pc, [pc, #20]   ; 23e00020 <_undefined_instruction>
23e00008:   e59ff014    ldr pc, [pc, #20]   ; 23e00024 <_software_interrupt>
23e0000c:   e59ff014    ldr pc, [pc, #20]   ; 23e00028 <_prefetch_abort>
23e00010:   e59ff014    ldr pc, [pc, #20]   ; 23e0002c <_data_abort>
23e00014:   e59ff014    ldr pc, [pc, #20]   ; 23e00030 <_not_used>
23e00018:   e59ff014    ldr pc, [pc, #20]   ; 23e00034 <_irq>
23e0001c:   e59ff014    ldr pc, [pc, #20]   ; 23e00038 <_fiq>
```

// 可以看出以下是异常终端向量表

```
23e00020 <_undefined_instruction>:
23e00020:   23e00060    mvncs   r0, #96 ; 0x60
```

// 其中，23e00020 存放的是未定义指令处理函数的地址，也就是 23e00060

// 以下以此类推

```
23e00024 <_software_interrupt>:
23e00024:   23e000c0    mvncs   r0, #192    ; 0xc0

23e00028 <_prefetch_abort>:
23e00028:   23e00120    mvncs   r0, #8

23e0002c <_data_abort>:
23e0002c:   23e00180    mvncs   r0, #32

23e00030 <_not_used>:
23e00030:   23e001e0    mvncs   r0, #56 ; 0x38

23e00034 <_irq>:
23e00034:   23e00240    mvncs   r0, #4

23e00038 <_fiq>:
23e00038:   23e002a0    mvncs   r0, #10
23e0003c:   deadbeef    cdple   14, 10, cr11, cr13, cr15, {7}
```

7.4.3. 符号表中需要注意的符号

前面我们说过了在 tiny210 中把连接地址设置为 0x23e00000。
project-x/build/out/u-boot/spl/u-boot.map

```
Linker script and memory map


Address of section .text set to 0x23e00000
.text           0x23e00000    0x29b28
 *(.__image_copy_start)
 .__image_copy_start
                0x23e00000        0x0 arch/arm/lib/built-in.o
                0x23e00000                __image_copy_start
 *(.vectors)
```

```
.vectors       0x23e00000      0x300 arch/arm/lib/built-in.o
               0x23e00000              _start
               0x23e00020              _undefined_instruction
               0x23e00024              _software_interrupt
               0x23e00028              _prefetch_abort
               0x23e0002c              _data_abort
               0x23e00030              _not_used
               0x23e00034              _irq
               0x23e00038              _fiq
               0x23e00040              IRQ_STACK_START_IN
*(.__image_copy_end)
.__image_copy_end
               0x23e36b78      0x0 arch/arm/lib/built-in.o
*(.__rel_dyn_start)
.__rel_dyn_start
               0x23e36b78      0x0 arch/arm/lib/built-in.o
*(.__rel_dyn_end)
.__rel_dyn_end
               0x23e3cbb8      0x0 arch/arm/lib/built-in.o
               0x23e3cbb8              _image_binary_end = .
*(.__bss_start)
.__bss_start   0x23e36b78      0x0 arch/arm/lib/built-in.o
               0x23e36b78              __bss_start
.__bss_end     0x23e6b514      0x0 arch/arm/lib/built-in.o
               0x23e6b514              __bss_end
```

重点关注

* __image_copy_start & __image_copy_end

界定了代码空间的位置，用于重定向代码的时候使用，在 uboot relocate 的过程中，需要把这部分拷贝到 uboot 的新的地址空间中，后续在新地址空间中运行。

具体可以参考《[uboot] （番外篇）uboot relocation 介绍》。

* _start

在 u-boot-spl.lds 中 ENTRY(_start)，也就规定了代码的入口函数是_start。所以后续分析代码的

时候就是从这里开始分析。
* __rel_dyn_start & __rel_dyn_end
由链接器生成，存放了绝对地址符号的 label 的地址，用于修改 uboot relocate 过程中修改绝对地址符号的 label 的值。
具体可以参考《[uboot] （番外篇）uboot relocation 介绍》。
* _image_binary_end

综上，u-boot 的编译就完成了。

8. 番外篇-global_data 介绍

8.1. global_data 功能

8.1.1. global_data 存在的意义

在某些情况下，uboot 是在某些只读存储器上运行，比如 ROM、nor flash 等等。

在 uboot 被重定向到 RAM（可读可写）之前，我们都无法写入数据，更无法通过全局变量来传递数据。

而 global_data 则是为了解决这个问题。

这里顺便一下，后续的 uboot 的 relocation 操作，也就是 uboot 的重定向操作，最主要的目的也是为了解决这个问题，后续会专门说明。

8.1.2. global_data 简单介绍

global_data 又称之为 GD.

简单地说，uboot 把 global_data 放在 RAM 区，并且使用 global_data 来存储全局数据。由此来解决上述场景中无法使用全局变量的问题。

8.2. global_data 数据结构

8.2.1. 数据结构说明

global_data 数据结构结构体定义为 struct global_data，被 typedef 为 gd_t。

也就是说可以直接通过 struct global_data 或者 gd_t 来进行声明。

struct global_data 定义如下（过滤掉一些被宏定义包含的部分）：

include/asm-generic/global_data.h

typedef struct global_data {

```
    bd_t *bd;
    unsigned long flags;
    unsigned int baudrate;
    unsigned long cpu_clk;  /* CPU clock in Hz!     */
    unsigned long bus_clk;
    /* We cannot bracket this with CONFIG_PCI due to mpc5xxx */
    unsigned long pci_clk;
    unsigned long mem_clk;
    unsigned long have_console; /* serial_init() was called */
    unsigned long env_addr; /* Address  of Environment struct */
    unsigned long env_valid;    /* Checksum of Environment valid? */
    unsigned long ram_top;  /* Top address of RAM used by U-Boot */
    unsigned long relocaddr;    /* Start address of U-Boot in RAM */
    phys_size_t ram_size;   /* RAM size */
    unsigned long mon_len;  /* monitor len */
    unsigned long irq_sp;       /* irq stack pointer */
    unsigned long start_addr_sp;    /* start_addr_stackpointer */
    unsigned long reloc_off;
    struct global_data *new_gd; /* relocated global data */
    const void *fdt_blob;   /* Our device tree, NULL if none */
    void *new_fdt;      /* Relocated FDT */
    unsigned long fdt_size; /* Space reserved for relocated FDT */
    struct jt_funcs *jt;        /* jump table */
    char env_buf[32];   /* buffer for getenv() before reloc. */
    unsigned long timebase_h;
    unsigned long timebase_l;
    struct udevice *cur_serial_dev; /* current serial device */
    struct arch_global_data arch;   /* architecture-specific data */

} gd_t;
```

8.2.2. 成员说明

重点说明

bd_t *bd：board info 数据结构定义,位于文件 include/asm-arm/u-boot.h 定义,主要是保存开发板的相关参数。

unsigned long env_addr：环境变量的地址。

unsigned long ram_top：RAM 空间的顶端地址

unsigned long relocaddr：UBOOT 重定向后地址

phys_size_t ram_size：物理 ram 的 size

unsigned long irq_sp：中断的堆栈地址

unsigned long start_addr_sp：堆栈地址

unsigned long reloc_off：uboot 的 relocation 的偏移

struct global_data *new_gd：重定向后的 struct global_data 结构体

const void *fdt_blob：我们设备的 dtb 地址

void *new_fdt：relocation 之后的 dtb 地址

unsigned long fdt_size：dtb 的长度

struct udevice *cur_serial_dev：当前使用的串口设备。

其他成员在后续时候到的时候在进行说明。

8.3. global_data 存放位置以及如何获取其地址

8.3.1. global_data 区域设置代码

（1）首先参考一下分配 global_data 的代码。

common/init/board_init.c

```
// 这个函数用于给 global_data 分配空间，在 relocation 之前调用
// 传入的参数是顶部地址，但是不一定是要内存顶部的地址，可以自己进行规划，后面_main 函数会说明
ulong board_init_f_alloc_reserve(ulong top)
{
   /* Reserve early malloc arena */
```

```
#if defined(CONFIG_SYS_MALLOC_F)
    top -= CONFIG_SYS_MALLOC_F_LEN;
// 先从顶部向下分配一块 CONFIG_SYS_MALLOC_F_LEN 大小的空间给 early malloc 使用
// 关于 CONFIG_SYS_MALLOC_F_LEN 可以参考 README
// 这块内存是用于在 relocation 前用于给 malloc 函数提供内存池。
#endif

    /* LAST : reserve GD (rounded up to a multiple of 16 bytes) */
    top = rounddown(top-sizeof(struct global_data), 16);
// 继续向下分配 sizeof(struct global_data)大小的内存给 global_data 使用，向下 16byte 对齐
// 这时候得到的地址就是 global_data 的地址。

    return top;
// 将 top，也就是 global_data 的地址返回
}
```

（2）然后看一下初始化 global_data 区域的代码。

common/init/board_init.c

去除无关代码的部分

```
// 这个函数用于对 global_data 区域进行初始化，也就是清空 global_data 区域
// 传入的参数就是 global_data 的基地址
void board_init_f_init_reserve(ulong base)
{
    struct global_data *gd_ptr;

    /*
     * clear GD entirely and set it up.
     * Use gd_ptr, as gd may not be properly set yet.
     */
```

```
    gd_ptr = (struct global_data *)base;
    /* zero the area */
    memset(gd_ptr, '\0', sizeof(*gd));
// 先通过 memset 函数对 global_data 数据结构进行清零


    /* next alloc will be higher by one GD plus 16-byte alignment */
    base += roundup(sizeof(struct global_data), 16);
// 因为 global_data 区域是 16Byte 对齐的，对齐后，后面的地址就是 early malloc 的内存池的地址，具体参考上述 board_init_f_alloc_reserve
// 所以这里就获取了 early malloc 的内存池的地址


    /*
     * record early malloc arena start.
     * Use gd as it is now properly set for all architectures.
     */
#if defined(CONFIG_SYS_MALLOC_F)
    /* go down one 'early malloc arena' */
    gd->malloc_base = base;
// 将内存池的地址写入到 gd->malloc_base 中
    /* next alloc will be higher by one 'early malloc arena' size */
    base += CONFIG_SYS_MALLOC_F_LEN;
//加上 CONFIG_SYS_MALLOC_F_LEN，获取 early malloc 的内存池的末尾地址，这里并没有什么作用，是为了以后在 early malloc 的内存池后面多加一个区域时的修改方便。
#endif
}
```

（3）arm 平台如何分配 global_data 区域，并保存其地址。

代码如下，去除掉被宏定义包含的无关代码部分

arch/arm/lib/crt0.S

```
ENTRY(_main)
/*
```

```
 * Set up initial C runtime environment and call board_init_f(0).
 */
   ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
```

@@ 预设堆栈指针为 CONFIG_SYS_INIT_SP_ADDR

@@ 在 tiny210 中初步设置为如下（include/configs/tiny210.h）：

```
@@ #define CONFIG_SYS_SDRAM_BASE          0x20000000
@@ #define MEMORY_BASE_ADDRESS  CONFIG_SYS_SDRAM_BASE
@@ #define PHYS_SDRAM_1     MEMORY_BASE_ADDRESS
@@ #define CONFIG_SYS_LOAD_ADDR     (PHYS_SDRAM_1 + 0x1000000)  /* default load address */
@@ #define CONFIG_SYS_INIT_SP_ADDR CONFIG_SYS_LOAD_ADDR
```

@@ 最终可以得到 CONFIG_SYS_INIT_SP_ADDR 是 0x3000_0000，也就是 uboot relocation 的起始地址

@@ 补充一下，DDR 的空间是 0x2000_0000-0x4000_0000

@@ 注意！！！这里只是预设的堆栈地址，而不是最终的堆栈地址！！！

```
   bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
```

@@ 8byte 对齐

```
   mov r0, sp
   bl  board_init_f_alloc_reserve
```

@@ 将 sp 的值放到 r0 中，也就是作为 board_init_f_alloc_reserve 的参数

@@ 返回之后，r0 里面存放的是 global_data 的地址

@@ 注意，同时也是堆栈地址，因为堆栈是向下增长的，所以不必担心和 global_data 冲突的问题

@@ 综上，此时 r0 存放的，既是 global_data 的地址，也是堆栈的地址

```
   mov sp, r0
```

@@ 把堆栈地址 r0 存放到 sp 中

```
    /* set up gd here, outside any C code */
    mov r9, r0
@@ 把 global_data 的地址存放在 r9 中
@@ 此时 r0 存放的还是 global_data 的地址

    bl  board_init_f_init_reserve
@@ 调用 board_init_f_init_reserve 对 global_data 进行初始化，r0 也就是其参数。
```

注意：最终 global_data 的地址存放在 r9 中了。

8.3.2. global_data 内存分布

内存分布如下：

————————-CONFIG_SYS_LOAD_ADDR ————————————高地址

…….. early malloc 内存池

————————————-early malloc 内存池基地址 ————————————

…… global_data 区域

————————-global_data 基地址（r9）, 也是堆栈的起始地址————-

……堆栈空间

————————————————堆栈结束————————————————-低地址

注意：最终 global_data 的地址存放在 r9 中了。

8.4. global_data 使用方式

8.4.1. 原理说明

前面我们一直强调了 global_data 的地址存放在 r9 中了。

所以当我们需要 global_data 的时候，直接从 r9 寄存器中获取其地址即可。

uboot 中定义了一个宏 DECLARE_GLOBAL_DATA_PTR，使我们可以更加简单地获取 global_data。

定义如下：

arch/arm/include/asm/global_data.h

#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r9")

DECLARE_GLOBAL_DATA_PTR 定义了 gd_t *gd，并且其地址是 r9 中的值。

一旦使用了 DECLARE_GLOBAL_DATA_PTR 声明之后，后续就可以直接使用 gd 变量，也就是 global_data 了。

8.4.2. 使用示例

DECLARE_GLOBAL_DATA_PTR 定义了 gd_t *gd，并且其地址是 r9 中的值。

一旦使用了 DECLARE_GLOBAL_DATA_PTR 声明之后，后续就可以直接使用 gd 变量，也就是 global_data 了。

简单例子如下：

common/board_r.c

```
DECLARE_GLOBAL_DATA_PTR
// 通过 DECLARE_GLOBAL_DATA_PTR 定义了 gd_t *gd
// 相当于如下：
// #define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r9")

static int initr_reloc(void)
{
   /* tell others: relocation done */
   gd->flags |= GD_FLG_RELOC | GD_FLG_FULL_MALLOC_INIT;
// 直接使用 gd 变量，也就是 uboot 的 global_data。

   return 0;
```

}

global_data 相对比较简单，也就不多说了。

9. 番外篇-uboot relocation 介绍

9.1. relocate 介绍

9.1.1. uboot 的 relocate

uboot 的 relocate 动作就是指 uboot 的重定向动作，也就是将 uboot 自身镜像拷贝到 ddr 上的另外一个位置的动作。

9.1.2. uboot 为什么要进行 relocate

考虑以下问题

* 在某些情况下，uboot 是在某些只读存储器上运行，比如 ROM、nor flash 等等。需要将这部分代码拷贝到 DDR 上才能完整运行 uboot。

（当然，如果我们在 spl 阶段就把 uboot 拷贝到 ddr 上，就不会有这种情况。但是 uboot 本身就是要考虑各种可能性）

* 一般会把 kernel 放在 ddr 的低端地址上。

考虑到以上情况，uboot 的 relocation 动作会把自己本身 relocate 到 ddr 上（前提是在 SPL 的过程中或者在 dram_init 中已经对 ddr 进行初始化了），并且会 relocate 到 ddr 的顶端地址使之不会和 kernel 的冲突。

3、uboot 的一些注意事项

既然 uboot 会把自身 relocate 到 ddr 的其他位置上，那么相当于执行地址也会发生变化。也就是要求 uboot 既要能在 relocate 正常执行，也要能在 relocate 之后正常执行。这就涉及到 uboot 需要使用“位置无关代码”技术，也就是 Position independent code 技术。

9.2. “位置无关代码”介绍及其原理

9.2.1. 什么是“位置无关代码”

“位置无关代码”是指无论代码加载到内存上的什么地址上，都可以被正常运行。也就是当加载地址和连接地址不一样时，CPU 也可以通过相对寻址获得到正确的指令地址。

9.2.2. 如何生成“位置无关代码”

（1）生成位置无关代码分成两部分

* 首先是编译源文件的时候，需要将其编译成位置无关代码，主要通过 gcc 的-fpic 选项（也有可能是 fPIC,fPIE, mword-relocations 选项）

* 其次是连接时要将其连接成一个完整的位置无关的可执行文件，主要通过 ld 的-fpie 选项

（2）ARM 在如何生成“位置无关代码”

* 编译 PIC 代码

在《[uboot] （第四章）uboot 流程一一uboot 编译流程》中，我们知道 gcc 的编译选项如下：

c_flags=-Wp,-MD,arch/arm/mach-s5pc1xx/.clock.o.d -nostdinc -isystem /home/disk3/xys/temp/project-x/build/arm-none-linux-gnueabi-4.8/bin/../lib/gcc/arm-none-linux-gnueabi/4.8.3/include -Iinclude -I/home/disk3/xys/temp/project-x/u-boot/include -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/include -include /home/disk3/xys/temp/project-x/u-boot/include/linux/kconfig.h -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/mach-s5pc1xx -Iarch/arm/mach-s5pc1xx -D__KERNEL__ -D__UBOOT__ -Wall -Wstrict-prototypes -Wno-format-security -fno-builtin -ffreestanding -Os -fno-stack-protector -fno-delete-null-pointer-checks -g -fstack-usage -Wno-format-nonliteral -D__ARM__ -marm -mno-thumb-interwork -mabi=aapcs-linux -mword-relocations -fno-pic -mno-unaligned-access -ffunction-sections -fdata-sections -fno-common -ffixed-r9 -msoft-float -pipe -march=armv7-a -I/home/disk3/xys/temp/project-x/u-boot/arch/arm/mach-s5pc1xx/include -DKBUILD_STR(s)=#s -DKBUILD_BASENAME=KBUILD_STR(clock) -DKBUILD_MODNAME=KBUILD_STR(clock)

重点关注“-mword-relocations -fno-pic”。

由于使用 pic 时 movt / movw 指令会硬编码 16bit 的地址域，而 uboot 的 relocation 并不支持这个，

所以 arm 平台使用 mword-relocations 来生成位置无关代码。-fno-pic 则表示不使用 pic。

如下./arch/arm/config.mk

The movt / movw can hardcode 16 bit parts of the addresses in the

instruction. Relocation is not supported for that case, so disable

such usage by requiring word relocations.

PLATFORM_CPPFLAGS += $(call cc-option, -mword-relocations)

PLATFORM_CPPFLAGS += $(call cc-option, -fno-pic)

生成 PIE 可执行文件

在《[uboot] （第四章）uboot 流程一一uboot 编译流程》中，我们知道 ld 的连接选项如下：

LDFLAGS_u-boot=-pie --gc-sections -Bstatic -Ttext 0x23E00000

-pie 选项用于生成 PIE 位置无关可执行文件。

9.2.3. “位置无关代码”原理

这里只是个人根据实验的一些看法。

“位置无关代码”主要是通过使用一些只会使用相对地址的指令实现，比如“b”、“bl”、“ldr”、“adr”等等。

对于一些绝对地址符号（例如已经初始化的全局变量），会将其以 label 的形式放在每个函数的代码实现的末端。

同时，在链接的过程中，会把这些 label 的地址统一维护在.rel.dyn 段中，当 relocation 的时候，方便对这些地址的 fix。

综上，个人觉得，既然使用绝对地址，那么就是说并不是完全的代码无关，而是说可以通过调整绝对地址符号的 label 表来实现代码的搬移。如果不做 relocate 或者在 relocate 之前还是需要加载到连接地址的位置上，这里只是个人看法！！！

个人也挺迷惑的，不知道对不对，这里希望有知道答案的大神给个意见。

9.2.4. .rel.dyn 段介绍和使用

前面也说了：

对于一些绝对地址符号（例如已经初始化的全局变量），会将其以 label 的形式放在每个函数的代码实现的末端。

同时，在链接的过程中，会把这些 label 的地址统一维护在.rel.dyn 段中，当 relocation 的时候，方便对这些地址的 fix。

这边简单的给个例子：

u-boot/common/board_f.c 中

static init_fnc_t init_sequence_f[] = {

// 这里定义了全局变量 init_sequence_f

}

void board_init_f(ulong boot_flags)

{

if (initcall_run_list(init_sequence_f))

// 这里使用了全局变量 init_sequence_f

hang();

}

通过如下命令对编译生成的 u-boot

arm-none-linux-gnueabi-objdump -D u-boot > uboot_objdump.txt

board_init_f 和 init_sequence_f 相关的连接地址如下：

Disassembly of section .text:

23e08428 <board_init_f>:

23e08438: e59f000c ldr r0, [pc, #12] ; 23e0844c <board_init_f+0x24>

// 通过 ldr r0, [pc, #12]，相当于是 ldr r0,[23e0844c] ,

// 也就是通过后面的 label 项，获得了 init_sequence_f 的地址。

23e0844c: 23e35dcc mvncs r5, #204, 26 ; 0x3300

// 23e0844c: 23e35dcc 是一个 label 项，23e0844c 表示这个 label 的地址，23e35dcc 表示这个 label 里面的值，也就是全局变量 23e35dcc 的地址。

Disassembly of section .data:

23e35dcc <init_sequence_f>:

// 全局变量 init_sequence_f 的地址在 23e35dcc

Disassembly of section .rel.dyn:

23e37b88: 23e0844c mvncs r8, #76, 8 ; 0x4c000000

23e37b8c: 00000017 andeq r0, r0, r7, lsl r0

// 把 init_sequence_f 的 label 的地址存在.rel.dyn 段中，方便后续 relocation 的时候，对 label 中的绝对变量地址进行整理修改。

各个符号的地址意义

23e08428，是 board_init_f 的地址

23e35dcc，是 init_sequence_f 的地址

23e0844c，是 board_init_f 为 init_sequence_f 做的 label 的地址，所以其值是 init_sequence_f 的地址，也就是 23e35dcc

23e37b88，把 init_sequence_f 的 label 的地址存放在.rel.dyn 段中的这个位置

根据上述对全局变量的寻址进行简单的说明

当 board_init_f 读取 init_sequence_f 时，会通过相对偏移获取 init_sequence_f 的 label 的地址（23e0844c），再从 23e0844c 中获取到 init_sequence_f 的地址（23e35dcc）。

综上，当 uboot 对自身进行 relocate 之后，此时全局变量的绝对地址已经发生变化，如果函数按照原来的 label 去获取全局变量的地址的时候，这个地址其实是 relocate 之前的地址。因此，在 relocate 的过程中需要对全局变量的 label 中的地址值进行修改，所以 uboot 将这些 label 的地址全部维护在.rel.dyn 段中，然后再统一对.rel.dyn 段指向的 label 进行修改。后续代码可以看出来。

9.3. uboot relocate 代码介绍

9.3.1. uboot relocate 地址和布局。

前面已经说明，uboot 的 relocation 动作会把自己本身 relocate 到 ddr 上（前提是在 SPL 的过程中或者在 dram_init 中已经对 ddr 进行初始化了），并且会 relocate 到 ddr 的顶端地址使之不会和 kernel 的冲突。

但是 relocate 过程中，并不是直接把 uboot 直接放到 ddr 的顶端位置，而是会有一定的布局，预留一些空间给其他一些需要固定空间的功能使用。

uboot relocate 从高地址到低地址布局如下（并不是所有的区域都是需要的，可以根据宏定义来确定），注意，对应区域的 size 在这个时候都是确定的，不会发生变化了。

relocate 区域	size
prom 页表区域	8192byte
logbuffer	LOGBUFF_RESERVE

pram 区域	CONFIG_PRAM<<10
round_4k	用于 4kb 对齐
mmu 页表区域	PGTABLE_SIZE
video buffer	不关心。但是是确定的。不会随着代码变化
lcd buffer	不关心。但是是确定的。不会随着代码变化
trace buffer	CONFIG_TRACE_BUFFER_SIZE
uboot 代码区域	gd->mon_len，并且对齐 4KB 对齐
malloc 内存池	TOTAL_MALLOC_LEN
Board Info 区域	sizeof(bd_t)
新 global_data 区域	sizeof(gd_t)
fdt 区域	gd->fdt_size
对齐	16b 对齐
堆栈区域	无限制

9.3.2. relocate 代码流程

主要是分成如下流程

* 对 relocate 进行空间规划

* 计算 uboot 代码空间到 relocation 的位置的偏移

* relocate 旧的 global_data 到新的 global_data 的空间上

* relocate 旧的 uboot 代码空间到新的空间上去

* 修改 relocate 之后全局变量的 label。（不懂的话参考第二节）

* relocate 中断向量表

（1）首先看一下 relocate 的整体代码

去掉无关代码的代码如下：

arch/arm/lib/crt0.S

ENTRY(_main)

bl board_init_f

@@ 在 board_init_f 里面实现了

@@ （1）对 relocate 进行空间规划

@@ （2）计算 uboot 代码空间到 relocation 的位置的偏移

@@ （3）relocate 旧的 global_data 到新的 global_data 的空间上

ldr sp, [r9, #GD_START_ADDR_SP] /* sp = gd->start_addr_sp */

bic sp, sp, #7 /* 8-byte alignment for ABI compliance */

ldr r9, [r9, #GD_BD] /* r9 = gd->bd */

sub r9, r9, #GD_SIZE /* new GD is below bd */

@@ 把新的 global_data 地址放在 r9 寄存器中

adr lr, here

ldr r0, [r9, #GD_RELOC_OFF] /* r0 = gd->reloc_off */

add lr, lr, r0

@@ 计算返回地址在新的 uboot 空间中的地址。b 调用函数返回之后，就跳到了新的 uboot 代码空间中。

ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */

@@ 把 uboot 的新的地址空间放到 r0 寄存器中，作为 relocate_code 的参数

b relocate_code

@@ 跳转到 relocate_code 中，在这里面实现了

@@ （1）relocate 旧的 uboot 代码空间到新的空间上去

@@ （2）修改 relocate 之后全局变量的 label

@@ 注意，由于上述已经把 lr 寄存器重定义到 uboot 新的代码空间中了，所以返回之后，就已经跳到了新的代码空间了！！！！！！

bl relocate_vectors

@@ relocate 中断向量表

注意上面的注释，从 relocate_code 返回之后就已经在新的 uboot 代码空间中运行了。

这里简单地说明一下 board_init_f：

```
static init_fnc_t init_sequence_f[] = {
#ifdef CONFIG_SANDBOX
    setup_ram_buf,
#endif
    setup_mon_len,
#ifdef CONFIG_OF_CONTROL
    fdtdec_setup,
#endif
#ifdef CONFIG_TRACE
    trace_early_init,
...
}
// 可以看出 init_sequence_f 是一个函数指针数组

void board_init_f(ulong boot_flags)
{
    if (initcall_run_list(init_sequence_f))
// 在这里会 init_sequence_f 里面的函数
        hang();
}
```

（2）对 relocate 进行空间规划

布局已经在上面说过了。

其规划只要体现在 gd 一些指针的设置，如下面所示

```
————————————————————————————— <——(gd->ram_top)
| 最高的区域
————————————————————————————
```

| ……

——————————————————————

| uboot 代码区域

—————————————————————— <——(gd->relocaddr)

| ……

——————————————————————

| Board Info 区域

—————————————————————— <——(gd->bd)

| 新 global_data 区域

—————————————————————— <——(gd->new_gd)

| fdt 区域

—————————————————————— <——(gd->new_fdt)

| …..

—————————————————————— <——(gd->start_addr_sp)

| 堆栈区域

——————————————————————

在 board_init_f 中，会依次执行 init_sequence_f 数组里面函数。其中，和 relocate 空间规划的函数如下：

```
static init_fnc_t init_sequence_f[] = {
    setup_dest_addr,
#if defined(CONFIG_SPARC)
    reserve_prom,
#endif
#if defined(CONFIG_LOGBUFFER) && !defined(CONFIG_ALT_LB_ADDR)
    reserve_logbuffer,
#endif
#ifdef CONFIG_PRAM
    reserve_pram,
#endif
```

```
    reserve_round_4k,
#if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF)) && \
        defined(CONFIG_ARM)
    reserve_mmu,
#endif
#ifdef CONFIG_DM_VIDEO
    reserve_video,
#else
# ifdef CONFIG_LCD
    reserve_lcd,
# endif
    /* TODO: Why the dependency on CONFIG_8xx? */
# if defined(CONFIG_VIDEO) && (!defined(CONFIG_PPC) || defined(CONFIG_8xx)) && \
        !defined(CONFIG_ARM) && !defined(CONFIG_X86) && \
        !defined(CONFIG_BLACKFIN) && !defined(CONFIG_M68K)
    reserve_legacy_video,
# endif
#endif /* CONFIG_DM_VIDEO */
    reserve_trace,
#if !defined(CONFIG_BLACKFIN)
    reserve_uboot,
#endif
#ifndef CONFIG_SPL_BUILD
    reserve_malloc,
    reserve_board,
#endif
    setup_machine,
    reserve_global_data,
    reserve_fdt,
    reserve_arch,
```

```
    reserve_stacks,
```

代码里面都是一些简单的减法以及指针的设置。可以参考上述“区域布局”和指针设置自己看一下代码，这里不详细说明。

这里说明一下 setup_dest_addr，也就是一些指针的初始化。

```
static int setup_dest_addr(void)
{
    debug("Monitor len: %08lX\n", gd->mon_len);
// gd->mon_len 表示了整个 uboot 代码空间的大小，如下
// gd->mon_len = (ulong)&__bss_end - (ulong)_start;
// 在 uboot 代码空间 relocate 的时候，relocate 的 size 就是由这里决定

    debug("Ram size: %08lX\n", (ulong)gd->ram_size);
// gd->ram_size 表示了 ram 的 size，也就是可使用的 ddr 的 size，在 board.c 中定义如下
// int dram_init(void)
// {
//  gd->ram_size = PHYS_SDRAM_1_SIZE;也就是 0x2000_0000
//  return 0;
// }


#ifdef CONFIG_SYS_SDRAM_BASE
    gd->ram_top = CONFIG_SYS_SDRAM_BASE;
#endif
    gd->ram_top += get_effective_memsize();
    gd->ram_top = board_get_usable_ram_top(gd->mon_len);
// gd->ram_top 计算 ddr 的顶端地址
// CONFIG_SYS_SDRAM_BASE(0x2000_0000+0x2000_0000=0x4000_0000)

    gd->relocaddr = gd->ram_top;
// 从 gd->ram_top 的位置开始分配
```

```
    debug("Ram top: %08lX\n", (ulong)gd->ram_top);
    return 0;
}
```

（3）计算 uboot 代码空间到 relocation 的位置的偏移

同样在 board_init_f 中，调用 init_sequence_f 数组里面的 setup_reloc 实现。

```
static int setup_reloc(void)
{
#ifdef CONFIG_SYS_TEXT_BASE
    gd->reloc_off = gd->relocaddr - CONFIG_SYS_TEXT_BASE;
// gd->relocaddr 表示新的 uboot 代码空间的起始地址，CONFIG_SYS_TEXT_BASE 表示旧的 uboot 代码空间的起始地址，二者算起来就是偏移了。
#endif
}
```

（4）relocate 旧的 global_data 到新的 global_data 的空间上

同样在 board_init_f 中，调用 init_sequence_f 数组里面的 setup_reloc 实现。

```
static int setup_reloc(void)
{
    memcpy(gd->new_gd, (char *)gd, sizeof(gd_t));
// 直接把 gd 的地址空间拷贝到 gd->new_gd 中
}
```

（5）relocate 旧的 uboot 代码空间到新的空间上去

代码在 relocate_code 中，上述（1）中可以知道此时的 r0 是 uboot 的新的地址空间。

主要目的是把__image_copy_start 到__image_copy_end 的代码空间拷贝到新的 uboot 地址空间中。

关于__image_copy_start 和__image_copy_end 可以看《[uboot] （第四章）uboot 流程一一uboot 编译流程》

```
ENTRY(relocate_code)
    ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
// 获取 uboot 代码空间的首地址
    subs    r4, r0, r1      /* r4 <- relocation offset */
// 计算新旧 uboot 代码空间的偏移
```

```
    beq relocate_done       /* skip relocation */
    ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
```

// 获取 uboot 代码空间的尾地址

```
copy_loop:
    ldmia   r1!, {r10-r11}      /* copy from source address [r1]    */
    stmia   r0!, {r10-r11}      /* copy to   target address [r0]    */
    cmp r1, r2          /* until source end address [r2]    */
    blo copy_loop
```

// 把旧代码空间复制到新代码空间中。

（6）修改 relocate 之后全局变量的 label

需要先完全理解第二节 “ “位置无关代码” 介绍及其原理”

主要目的是修改 label 中的地址。

这里复习一下:

* 绝对地址符号的地址会放在 label 中提供位置无关代码使用

* label 的地址会放在.rel.dyn 段中

综上，当 uboot 对自身进行 relocate 之后，此时全局变量的绝对地址已经发生变化，如果函数按照原来的 label 去获取全局变量的地址的时候，这个地址其实是 relocate 之前的地址。因此，在 relocate 的过程中需要对全局变量的 label 中的地址值进行修改，所以 uboot 将这些 label 的地址全部维护在.rel.dyn 段中，然后再统一对.rel.dyn 段指向的 label 进行修改。后续代码可以看出来。

.rel.dyn 段部分示例如下:

```
23e37b88:       23e0844c        mvncs   r8, #76, 8      ; 0x4c000000
23e37b8c:       00000017        andeq   r0, r0, r7, lsl r0
23e37b90:       23e084b4        mvncs   r8, #180, 8     ; 0xb4000000
23e37b94:       00000017        andeq   r0, r0, r7, lsl r0
23e37b98:       23e084d4        mvncs   r8, #212, 8     ; 0xd4000000
23e37b9c:       00000017        andeq   r0, r0, r7, lsl r0
23e37ba0:       23e0854c        mvncs   r8, #76, 10     ; 0x13000000
23e37ba4:       00000017        andeq   r0, r0, r7, lsl r0
```

可以看出.rel.dyn 段用了 8 个字节来描述一个 label，其中，高 4 字节是 label 地址标识 0x17，低 4 字节就是 label 的地址。

所以需要先判断 label 地址标识是否正确，然后再根据第四字节获取 label，对 label 中的符号地址进行修改。

代码如下：

```
ENTRY(relocate_code)
    /*
     * fix .rel.dyn relocations
     */
    ldr r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
    ldr r3, =__rel_dyn_end  /* r3 <- SRC &__rel_dyn_end */
// __rel_dyn 段是由链接器生成的。
// 把__rel_dyn_start 放到 r2 中，把__rel_dyn_end 放到 r3 中

fixloop:
    ldmia   r2!, {r0-r1}        /* (r0,r1) <- (SRC location,fixup) */
// 从__rel_dyn_start 开始，加载两个字节到 r0 和 r1 中，高字节存在 r1 中表示标志，低字节存在 r0 中，表示 label 地址。
    and r1, r1, #0xff
    cmp r1, #23         /* relative fixup? */
// 比较高 4 字节是否等于 0x17
    bne fixnext
// 不等于的话，说明不是描述 label 地址，进行下一次循环

// label 在 relocate uboot 的时候也已经复制到了新的 uboot 地址空间了！！！
// 这里要注意，是对新的 uboot 地址空间 label 进行修改！！！
    /* relative fix: increase location by offset */
    add r0, r0, r4
// 获取新的 uboot 地址空间的 label 地址，
// 因为 r0 存的是旧地址空间的 label 地址，而新地址空间的 label 地址就是在旧地址空间的 label 地址加上偏移得到
// r4 就是 relocate offset，也就是新旧地址空间的偏移
```

```
    ldr r1, [r0]
// 从 label 中获取绝对地址符号的地址，存放在 r1 中

    add r1, r1, r4
    str r1, [r0]
// 根据前面的描述，我们的目的就是要 fix label 中绝对地址符号的地址，也就是将其修改为新地址空间的地址
// 所以为 r1 加上偏移之后，重新存储到 label 中。
// 后面 CPU 就可以根据 LABEL 在新 uboot 的地址空间中寻址到正确的符号。

fixnext:
    cmp r2, r3
    blo fixloop
```

（7）relocate 中断向量表

前面在《[uboot] （第四章）uboot 流程一一uboot 编译流程》中已经分析了，异常中断向量表的定义如下

arch/arm/lib/vectors.S

```
    .globl  _undefined_instruction
    .globl  _software_interrupt
    .globl  _prefetch_abort
    .globl  _data_abort
    .globl  _not_used
    .globl  _irq
    .globl  _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:    .word prefetch_abort
_data_abort:        .word data_abort
```

_not_used: .word not_used

_irq: .word irq

_fiq: .word fiq

我们知道 arm 的异常中断向量表需要复制到 0x00000000 处或者 0xFFFF0000 处（不知道的建议网上度娘一下）。

当 uboot 进行 relocate 之后，其异常处理函数的地址也发生了变化，因此，我们需要把新的异常中断向量表复制到 0x00000000 处或者 0xFFFF0000 处。

这部分操作就是在 relocate_vectors 中进行。

异常中断向量表在 uboot 代码空间中的地址如下：

23e00000 <__image_copy_start>:

23e00000: ea0000be b 23e00300 <reset>

23e00004: e59ff014 ldr pc, [pc, #20] ; 23e00020 <_undefined_instruction>

23e00008: e59ff014 ldr pc, [pc, #20] ; 23e00024 <_software_interrupt>

23e0000c: e59ff014 ldr pc, [pc, #20] ; 23e00028 <_prefetch_abort>

23e00010: e59ff014 ldr pc, [pc, #20] ; 23e0002c <_data_abort>

23e00014: e59ff014 ldr pc, [pc, #20] ; 23e00030 <_not_used>

23e00018: e59ff014 ldr pc, [pc, #20] ; 23e00034 <_irq>

23e0001c: e59ff014 ldr pc, [pc, #20] ; 23e00038 <_fiq>

// 可以看出以下是异常终端向量表

23e00020 <_undefined_instruction>:

23e00020: 23e00060 mvncs r0, #96 ; 0x60

// 其中，23e00020 存放的是未定义指令处理函数的地址，也就是 23e00060

// 以下以此类推

23e00024 <_software_interrupt>:

23e00024: 23e000c0 mvncs r0, #192 ; 0xc0

23e00028 <_prefetch_abort>:

23e00028:   23e00120   mvncs   r0, #8

23e0002c <_data_abort>:

23e0002c: 23e00180   mvncs   r0, #32

23e00030 <_not_used>:

23e00030:   23e001e0   mvncs   r0, #56 ; 0x38

23e00034 <_irq>:

23e00034:   23e00240   mvncs   r0, #4

23e00038 <_fiq>:

23e00038:   23e002a0   mvncs   r0, #10

23e0003c: deadbeef   cdple   14, 10, cr11, cr13, cr15, {7}

23e00040 <IRQ_STACK_START_IN>:

所以异常中断向量表就是从偏移 0x20 开始的 32 个字节。

代码如下（去除掉无关代码部分）：

```
ENTRY(relocate_vectors)
    /*
     * Copy the relocated exception vectors to the
     * correct address
     * CP15 c1 V bit gives us the location of the vectors:
     * 0x00000000 or 0xFFFF0000.
     */
```

@@ 注意看注释，通过 cp15 协处理器的 c1 寄存器的 V 标志来判断 cpu 从什么位置获取中断向量表，

@@ 换句话说，就是中断向量表应该被复制到什么地方！！！

ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */

@@ 获取 uboot 新地址空间的起始地址，存放到 r0 寄存器中

mrc p15, 0, r2, c1, c0, 0 /* V bit (bit[13]) in CP15 c1 */

ands r2, r2, #(1 << 13)

ldreq r1, =0x00000000 /* If V=0 */

ldrne r1, =0xFFFF0000 /* If V=1 */

@@ 获取 cp15 协处理器的 c1 寄存器的 V 标志，当 V=0 时，cpu 从 0x00000000 获取中断向量表，当 V=1 时，cpu 从 0xFFFF0000 获取中断向量表

@@ 将该地址存在 r1 中

ldmia r0!, {r2-r8,r10}

stmia r1!, {r2-r8,r10}

@@ 前面说了异常中断向量表就是从偏移 0x20 开始的 32 个字节。

@@ 所以这里是过滤掉前面的 0x20 个字节（32 个字节，8*4）

@@ 但是不明白为什么还要 stmia r1!, {r2-r8,r10}，理论上只需要让 r0 的值产生 0x20 的偏移就可以了才对？？？不明白。

@@ 经过上述两行代码之后，此时 r0 的值已经偏移了 0x20 了

ldmia r0!, {r2-r8,r10}

stmia r1!, {r2-r8,r10}

@@ 继续从 0x20 开始，获取 32 个字节，存储到 r1 指向的地址，也就是 cpu 获取中断向量表的地址

@@ r2-r8,r10 表示从 r2 到 r8 寄存器和 r10 寄存器，一个 8 个寄存器，每个寄存器有 4 个字节，所以就从 r0 指向的地址处获取到了 32 个字节

@@ 再把 {r2-r8,r10}的值存放到 r1 指向的地址，也就是 cpu 获取中断向量表的地址

bx lr

@@ 返回

ENDPROC(relocate_vectors)

经过上述，uboot relocate 就完成了。

10. 番外-uboot 驱动模型

10.1. 说明

10.1.1. uboot 的驱动模型简单介绍

uboot 引入了驱动模型（driver model），这种驱动模型为驱动的定义和访问接口提供了统一的方法。提高了驱动之间的兼容性以及访问的标准型。

uboot 驱动模型和 kernel 中的设备驱动模型类似，但是又有所区别。

在后续我们将驱动模型（driver model）简称为 DM，其实在 uboot 里面也是这样简称的。

具体细节建议参考./doc/driver-model/README.txt

10.1.2. 如何使能 uboot 的 DM 功能

（1）配置 CONFIG_DM

在 configs/tiny210_defconfig 中定义了如下：

CONFIG_DM=y

（2）使能相应的 uclass driver 的 config。

DM 和 uclass 是息息相关的，如果我们希望在某个模块引入 DM，那么就需要使用相应模块的 uclass driver 来代替旧版的通用 driver。

关于 uclass 我们会在后续继续说明。

以 serial 为例，为了在 serial 中引入 DM，我在 configs/tiny210_defconfig0 中打开了 CONFIG_DM_SERIAL 宏，如下

CONFIG_DM_SERIAL=y

看 driver/serial/Makefile

ifdef CONFIG_DM_SERIAL

obj-y += serial-uclass.o ## 引入 dm 的 serial core 驱动

else

obj-y += serial.o ## 通用的 serial core 驱动

endif

可以发现编译出来的 serial core 的驱动代码是不一样的。

（3）对应设备驱动也要引入 dm 的功能

其设备驱动主要是实现和底层交互，为 uclass 层提供接口。后续再具体说明。

__后续都以 serial-uclass 进行说明

10.2. uboot DM 整体架构

10.2.1. DM 的四个组成部分

uboot 的 DM 主要有四个组成部分

udevice

简单就是指设备对象，可以理解为 kernel 中的 device。

driver

udevice 的驱动，可以理解为 kernel 中的 device_driver。和底层硬件设备通信，并且为设备提供面向上层的接口。

uclass

先看一下 README.txt 中关于 uclass 的说明：

Uclass - a group of devices which operate in the same way. A uclass provides

a way of accessing individual devices within the group, but always

using the same interface. For example a GPIO uclass provides

operations for get/set value. An I2C uclass may have 10 I2C ports,

4 with one driver, and 6 with another.

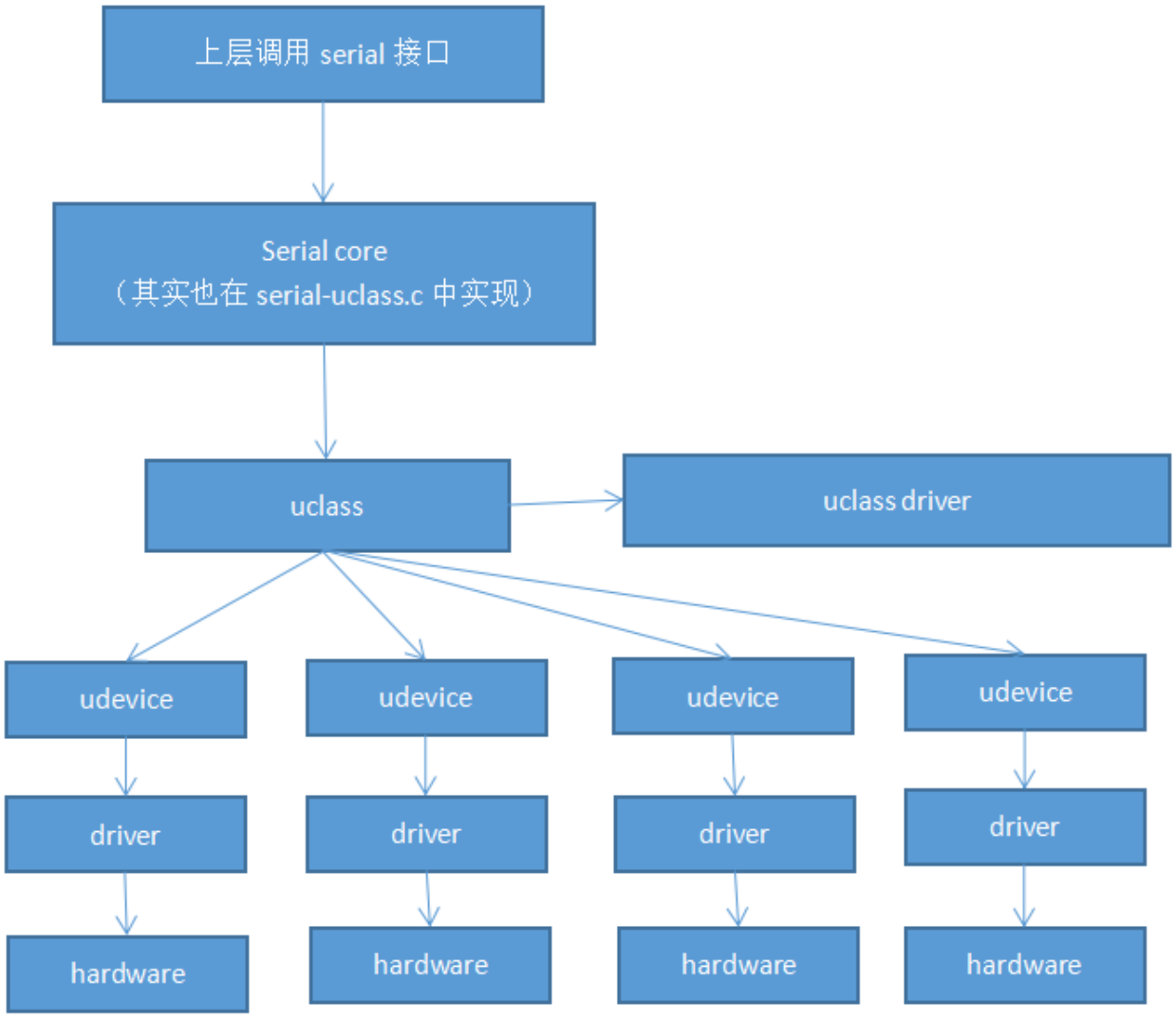
uclass，使用相同方式的操作集的 device 的组。相当于是一种抽象。uclass 为那些使用相同接口的设备提供了统一的接口。

例如，GPIO uclass 提供了 get/set 接口。再例如，一个 I2C uclass 下可能有 10 个 I2C 端口，4 个使用一个驱动，另外 6 个使用另外一个驱动。

uclass_driver

对应 uclass 的驱动程序。主要提供 uclass 操作时，如绑定 udevice 时的一些操作。

10.2.2. 调用关系框架图



DM 下的接口调用流程

10.2.3. 相互之间的关系

结合上图来看:

上层接口都是和 uclass 的接口直接通讯。

uclass 可以理解为一些具有相同属性的 udevice 对外操作的接口，uclass 的驱动是 uclass_driver，主要为上层提供接口。

udevice 的是指具体设备的抽象，对应驱动是 driver，driver 主要负责和硬件通信，为 uclass 提供实际的操作集。

udevice 找到对应的 uclass 的方式主要是通过：udevice 对应的 driver 的 id 和 uclass 对应的 uclass_driver 的 id 是否匹配。

udevice 会和 uclass 绑定。driver 会和 udevice 绑定。uclass_driver 会和 uclass 绑定。

这里先简单介绍一下：uclass 和 udevice 都是动态生成的。在解析 fdt 中的设备的时候，会动态生成 udevice。

然后找到 udevice 对应的 driver，通过 driver 中的 uclass id 得到 uclass_driver id。从 uclass 链表中查找对应的 uclass 是否已经生成，没有生成的话则动态生成 uclass。

10.2.4. GD 中和 DM 相关的部分

```
 typedef struct global_data {
#ifdef CONFIG_DM
    struct udevice  *dm_root;   /* Root instance for Driver Model */
// DM 中的根设备，也是 uboot 中第一个创建的 udevice，也就对应了 dts 里的根节点。
    struct udevice  *dm_root_f; /* Pre-relocation root instance */
// 在 relocation 之前 DM 中的根设备
    struct list_head uclass_root;   /* Head of core tree */
// uclass 链表，所有被 udevice 匹配的 uclass 都会被挂载到这个链表上
#endif
} gd_t;
```

10.3. DM 四个主要组成部分详细介绍

后续以数据结构、如何定义、存放位置、如何获取四个部分进行说明进行说明

10.3.1. uclass id

每一种 uclass 都有自己对应的 ID 号。定义于其 uclass_driver 中。其附属的 udevice 的 driver 中的 uclass id 必须与其一致。

所有 uclass id 定义于 include/dm/uclass-id.h 中

列出部分 id 如下

```
enum uclass_id {
    /* These are used internally by driver model */
    UCLASS_ROOT = 0,
    UCLASS_DEMO,
    UCLASS_CLK,     /* Clock source, e.g. used by peripherals */
    UCLASS_PINCTRL,     /* Pinctrl (pin muxing/configuration) device */
    UCLASS_SERIAL,      /* Serial UART */
}
```

10.3.2. uclass

（1）数据结构

```
struct uclass {
    void *priv;  // uclass 的私有数据指针
    struct uclass_driver *uc_drv; // 对应的 uclass driver
    struct list_head dev_head; // 链表头，连接所属的所有 udevice
    struct list_head sibling_node; // 链表节点，用于把 uclass 连接到 uclass_root 链表上
};
```

（2）如何定义

uclass 是 uboot 自动生成。并且不是所有 uclass 都会生成，有对应 uclass driver 并且有被 udevice 匹配到的 uclass 才会生成。

具体参考后面的 uboot DM 初始化一节。或者参考 uclass_add 实现。

（3）存放位置

所有生成的 uclass 都会被挂载 gd->uclass_root 链表上。

（4）如何获取、API

直接遍历链表 gd->uclass_root 链表并且根据 uclass id 来获取到相应的 uclass。

具体 uclass_get-》uclass_find 实现了这个功能。

有如下 API:

```
int uclass_get(enum uclass_id key, struct uclass **ucp);
// 从 gd->uclass_root 链表获取对应的 uclass
```

10.3.3. uclass_driver

（1）数据结构

include/dm/uclass.h

```
struct uclass_driver {
    const char *name; // 该 uclass_driver 的命令
    enum uclass_id id; // 对应的 uclass id
/* 以下函数指针主要是调用时机的区别  */
    int (*post_bind)(struct udevice *dev); // 在 udevice 被绑定到该 uclass 之后调用
    int (*pre_unbind)(struct udevice *dev); // 在 udevice 被解绑出该 uclass 之前调用
    int (*pre_probe)(struct udevice *dev); // 在该 uclass 的一个 udevice 进行 probe 之前调用
    int (*post_probe)(struct udevice *dev); // 在该 uclass 的一个 udevice 进行 probe 之后调用
    int (*pre_remove)(struct udevice *dev);// 在该 uclass 的一个 udevice 进行 remove 之前调用
    int (*child_post_bind)(struct udevice *dev); // 在该 uclass 的一个 udevice 的一个子设备被绑定到该 udevice 之后调用
    int (*child_pre_probe)(struct udevice *dev); // 在该 uclass 的一个 udevice 的一个子设备进行 probe 之前调用
    int (*init)(struct uclass *class); // 安装该 uclass 的时候调用
    int (*destroy)(struct uclass *class); // 销毁该 uclass 的时候调用
```

```
    int priv_auto_alloc_size; // 需要为对应的 uclass 分配多少私有数据
    int per_device_auto_alloc_size; //
    int per_device_platdata_auto_alloc_size; //
    int per_child_auto_alloc_size; //
    int per_child_platdata_auto_alloc_size;  //
    const void *ops; //操作集合
    uint32_t flags;   // 标识为
};
```

（2）如何定义

通过 UCLASS_DRIVER 来定义 uclass_driver.

以 serial-uclass 为例

```
UCLASS_DRIVER(serial) = {
    .id         = UCLASS_SERIAL,
    .name       = "serial",
    .flags      = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe    = serial_post_probe,
    .pre_remove    = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
};
```

UCLASS_DRIVER 实现如下：

```
#define UCLASS_DRIVER(__name)                    \
    ll_entry_declare(struct uclass_driver, __name, uclass)

#define ll_entry_declare(_type, _name, _list)              \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)      \
            __attribute__((unused,              \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

关于 ll_entry_declare 我们在《[uboot] （第六章）uboot 流程——命令行模式以及命令处理介绍》已经介绍过了

最终得到一个如下结构体

```
struct uclass_driver  _u_boot_list_2_uclass_2_serial = {
    .id        = UCLASS_SERIAL,   // 设置对应的 uclass id
    .name        = "serial",
    .flags        = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe     = serial_post_probe,
    .pre_remove     = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
}
```

并且存放在.u_boot_list_2_uclass_2_serial 段中。

（3）存放位置

通过上述，我们知道 serial 的 uclass_driver 结构体_u_boot_list_2_uclass_2_serial 被存放到.u_boot_list_2_uclass_2_serial 段中。

通过查看 u-boot.map 得到如下

```
.u_boot_list_2_uclass_1
                0x23e368e0        0x0 drivers/built-in.o
.u_boot_list_2_uclass_2_gpio
                0x23e368e0       0x48 drivers/gpio/built-in.o
                0x23e368e0                _u_boot_list_2_uclass_2_gpio  // gpio uclass driver 的符号
.u_boot_list_2_uclass_2_root
                0x23e36928       0x48 drivers/built-in.o
                0x23e36928                _u_boot_list_2_uclass_2_root // root uclass drvier 的符号
.u_boot_list_2_uclass_2_serial
                0x23e36970       0x48 drivers/serial/built-in.o
```

0x23e36970 _u_boot_list_2_uclass_2_serial // serial uclass driver 的符号

.u_boot_list_2_uclass_2_simple_bus

0x23e369b8 0x48 drivers/built-in.o

0x23e369b8 _u_boot_list_2_uclass_2_simple_bus

.u_boot_list_2_uclass_3

0x23e36a00 0x0 drivers/built-in.o

0x23e36a00 . = ALIGN (0x4)

最终，所有 uclass driver 结构体以列表的形式被放在.u_boot_list_2_uclass_1 和.u_boot_list_2_uclass_3 的区间中。

这个列表简称 uclass_driver table。

（4）如何获取、API

想要获取 uclass_driver 需要先获取 uclass_driver table。

可以通过以下宏来获取 uclass_driver table

```
struct uclass_driver *uclass =
    ll_entry_start(struct uclass_driver, uclass);
// 会根据.u_boot_list_2_uclass_1 的段地址来得到 uclass_driver table 的地址

const int n_ents = ll_entry_count(struct uclass_driver, uclass);
// 获得 uclass_driver table 的长度
```

接着通过遍历这个 uclass_driver table，得到相应的 uclass_driver。

有如下 API

```
struct uclass_driver *lists_uclass_lookup(enum uclass_id id)
// 从 uclass_driver table 中获取 uclass id 为 id 的 uclass_driver。
```

10.3.4. udevice

（1）数据结构

include/dm/device.h

```
struct udevice {
    const struct driver *driver; // 该 udevice 对应的 driver
    const char *name; // 设备名
    void *platdata; // 该 udevice 的平台数据
    void *parent_platdata; // 提供给父设备使用的平台数据
    void *uclass_platdata; // 提供给所属 uclass 使用的平台数据
    int of_offset; // 该 udevice 的 dtb 节点偏移，代表了 dtb 里面的这个节点 node
    ulong driver_data; // 驱动数据
    struct udevice *parent; // 父设备
    void *priv; // 私有数据的指针
    struct uclass *uclass; // 所属 uclass
    void *uclass_priv; // 提供给所属 uclass 使用的私有数据指针
    void *parent_priv; // 提供给其父设备使用的私有数据指针
    struct list_head uclass_node; // 用于连接到其所属 uclass 的链表上
    struct list_head child_head; // 链表头，连接其子设备
    struct list_head sibling_node; // 用于连接到其父设备的链表上
    uint32_t flags; // 标识
    int req_seq;
    int seq;
#ifdef CONFIG_DEVRES
    struct list_head devres_head;
#endif
};
```

（2）如何定义

在 dtb 存在的情况下，由 uboot 解析 dtb 后动态生成，后续在“uboot DM 的初始化”一节中具体说明。

（3）存放位置

连接到对应 uclass 中

也就是会连接到 uclass->dev_head 中

连接到父设备的子设备链表中

也就是会连接到 udevice->child_head 中，并且最终的根设备是 gd->dm_root 这个根设备。

（4）如何获取、API

从 uclass 中获取 udevice

遍历 uclass->dev_head，获取对应的 udevice。有如下 API

```
#define uclass_foreach_dev(pos, uc) \
    list_for_each_entry(pos, &uc->dev_head, uclass_node)

#define uclass_foreach_dev_safe(pos, next, uc)  \
    list_for_each_entry_safe(pos, next, &uc->dev_head, uclass_node)

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp); // 通过索引从 uclass 中获取 udevice
int uclass_get_device_by_name(enum uclass_id id, const char *name, // 通过设备名从 uclass 中获取 udevice
                struct udevice **devp);
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp);
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
                 struct udevice **devp);
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
                const char *name, struct udevice **devp);
int uclass_first_device(enum uclass_id id, struct udevice **devp);
int uclass_first_device_err(enum uclass_id id, struct udevice **devp);
int uclass_next_device(struct udevice **devp);
int uclass_resolve_seq(struct udevice *dev);
```

10.3.5. driver

和 uclass_driver 方式是相似的。

（1）数据结构

include/dm/device.h

```
struct driver {
    char *name;    // 驱动名
    enum uclass_id id;  // 对应的 uclass id
    const struct udevice_id *of_match;    // compatible 字符串的匹配表，用于和 device tree 里面的设备节点匹配
    int (*bind)(struct udevice *dev);   // 用于绑定目标设备到该 driver 中
    int (*probe)(struct udevice *dev);   // 用于 probe 目标设备,激活
    int (*remove)(struct udevice *dev); // 用于 remove 目标设备。禁用
    int (*unbind)(struct udevice *dev); // 用于解绑目标设备到该 driver 中
    int (*ofdata_to_platdata)(struct udevice *dev); // 在 probe 之前，解析对应 udevice 的 dts 节点，转化成 udevice 的平台数据
    int (*child_post_bind)(struct udevice *dev); // 如果目标设备的一个子设备被绑定之后，调用
    int (*child_pre_probe)(struct udevice *dev); // 在目标设备的一个子设备被 probe 之前，调用
    int (*child_post_remove)(struct udevice *dev); // 在目标设备的一个子设备被 remove 之后，调用
    int priv_auto_alloc_size; //需要分配多少空间作为其 udevice 的私有数据
    int platdata_auto_alloc_size; //需要分配多少空间作为其 udevice 的平台数据
    int per_child_auto_alloc_size;  // 对于目标设备的每个子设备需要分配多少空间作为父设备的私有数据
    int per_child_platdata_auto_alloc_size; // 对于目标设备的每个子设备需要分配多少空间作为父设备的平台数据
    const void *ops;    /* driver-specific operations */ // 操作集合的指针，提供给 uclass 使用，没有规定操作集的格式，由具体 uclass 决定
    uint32_t flags; // 一些标志位
};
```

（2）如何定义

通过 U_BOOT_DRIVER 来定义一个 driver

以 s5pv210 为例：

driver/serial/serial_s5p.c

```
U_BOOT_DRIVER(serial_s5p) = {
    .name    = "serial_s5p",
    .id    = UCLASS_SERIAL,
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

U_BOOT_DRIVER 实现如下：

```
#define U_BOOT_DRIVER(__name)                    \
    ll_entry_declare(struct driver, __name, driver)

#define ll_entry_declare(_type, _name, _list)              \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)      \
            __attribute__((unused,              \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

关于 ll_entry_declare 我们在《[uboot] （第六章）uboot 流程——命令行模式以及命令处理介绍》已经介绍过了

最终得到如下一个结构体

```
struct driver _u_boot_list_2_driver_2_serial_s5p= {
    .name    = "serial_s5p",
```

```
    .id    = UCLASS_SERIAL,
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

并且存放在.u_boot_list_2_driver_2_serial_s5p 段中

（3）存放位置

通过上述，我们知道 serial_s5p 的 driver 结构体_u_boot_list_2_driver_2_serial_s5p 被存放在.u_boot_list_2_driver_2_serial_s5p 段中。

通过查看 u-boot.map 得到如下

```
 .u_boot_list_2_driver_1
                0x23e36754        0x0 drivers/built-in.o
.u_boot_list_2_driver_2_gpio_exynos
                0x23e36754       0x44 drivers/gpio/built-in.o
                0x23e36754                _u_boot_list_2_driver_2_gpio_exynos
.u_boot_list_2_driver_2_root_driver
                0x23e36798       0x44 drivers/built-in.o
                0x23e36798                _u_boot_list_2_driver_2_root_driver
.u_boot_list_2_driver_2_serial_s5p
                0x23e367dc       0x44 drivers/serial/built-in.o
                0x23e367dc                _u_boot_list_2_driver_2_serial_s5p
.u_boot_list_2_driver_2_simple_bus_drv
                0x23e36820       0x44 drivers/built-in.o
                0x23e36820                _u_boot_list_2_driver_2_simple_bus_drv
```

.u_boot_list_2_driver_3

0x23e36864 0x0 drivers/built-in.o

最终，所有 driver 结构体以列表的形式被放在.u_boot_list_2_driver_1 和.u_boot_list_2_driver_3 的区间中。

这个列表简称 driver table。

（4）如何获取、API

想要获取 driver 需要先获取 driver table。

可以通过以下宏来获取 driver table

```
struct driver *drv =
    ll_entry_start(struct driver, driver);
// 会根据.u_boot_list_2_driver_1 的段地址来得到 uclass_driver table 的地址

const int n_ents = ll_entry_count(struct driver, driver);
// 获得 driver table 的长度
```

接着通过遍历这个 driver table，得到相应的 driver。

```
struct driver *lists_driver_lookup_name(const char *name)
// 从 driver table 中获取名字为 name 的 driver。
```

10.4. DM 的一些 API 整理

先看一下前面一节理解一下。

10.4.1. uclass 相关 API

```
int uclass_get(enum uclass_id key, struct uclass **ucp);
```

// 从 gd->uclass_root 链表获取对应的 uclass

10.4.2. uclass_driver 相关 API

struct uclass_driver *lists_uclass_lookup(enum uclass_id id)

// 从 uclass_driver table 中获取 uclass id 为 id 的 uclass_driver。

10.4.3. udevice 相关 API

```
#define uclass_foreach_dev(pos, uc) \
    list_for_each_entry(pos, &uc->dev_head, uclass_node)

#define uclass_foreach_dev_safe(pos, next, uc)  \
    list_for_each_entry_safe(pos, next, &uc->dev_head, uclass_node)
```

int device_bind(struct udevice *parent, const struct driver *drv,

const char *name, void *platdata, int of_offset,

struct udevice **devp)

// 初始化一个 udevice，并将其与其 uclass、driver 绑定。

int device_bind_by_name(struct udevice *parent, bool pre_reloc_only,

const struct driver_info *info, struct udevice **devp)

// 通过 name 获取 driver 并且调用 device_bind 对 udevice 初始化，并将其与其 uclass、driver 绑定。

int uclass_bind_device(struct udevice *dev)

// 绑定 udevice 到其对应的 uclass 的设备链表中

```
{
    uc = dev->uclass;
    list_add_tail(&dev->uclass_node, &uc->dev_head);
```

}

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp); // 通过索引从 uclass 中获取 udevice，注意，在获取的过程中就会对设备进行 probe

int uclass_get_device_by_name(enum uclass_id id, const char *name, // 通过设备名从 uclass 中获取 udevice

struct udevice **devp);

int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp);

int uclass_get_device_by_of_offset(enum uclass_id id, int node,

struct udevice **devp);

int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,

const char *name, struct udevice **devp);

int uclass_first_device(enum uclass_id id, struct udevice **devp);

int uclass_first_device_err(enum uclass_id id, struct udevice **devp);

int uclass_next_device(struct udevice **devp);

int uclass_resolve_seq(struct udevice *dev);

10.4.4. driver 相关 API

struct driver *lists_driver_lookup_name(const char *name)

// 从 driver table 中获取名字为 name 的 driver。

10.5. uboot 设备的表达

10.5.1. 说明

uboot 中可以通过两种方法来添加设备

通过直接定义平台设备（这种方式基本上不使用）

通过在设备树添加设备信息

注意：这里只是设备的定义，最终还是会被 uboot 解析成 udevice 结构体的。

10.5.2. 直接定义平台设备

（这种方式除了根设备外基本上不使用）

（1）通过 U_BOOT_DEVICE 宏来进行定义或者直接定义 struct driver_info 结构体

（2）U_BOOT_DEVICE 宏以 ns16550_serial 为例

```
U_BOOT_DEVICE(overo_uart) = {
   "ns16550_serial",
   &overo_serial
};
```

U_BOOT_DEVICE 实现如下：

和上述的 U_BOOT_DRIVER 类似，这里不详细说明了

```
#define U_BOOT_DEVICE(__name)                   \
   ll_entry_declare(struct driver_info, __name, driver_info)

/* Declare a list of devices. The argument is a driver_info[] array */
#define U_BOOT_DEVICES(__name)                  \
   ll_entry_declare_list(struct driver_info, __name, driver_info)
```

（3）直接定义 struct driver_info 结构体，以根设备为例

uboot 会创建一个根设备 root，作为所有设备的祖设备

root 的定义如下：

```
static const struct driver_info root_info = {
   .name       = "root_driver",
};
```

10.5.3. 在设备树添加设备信息

在对应的 dts 文件中添加相应的设备节点和信息，以 tiny210 的 serial 为例：

arch/arm/dts/s5pv210-tiny210.dts

```
/dts-v1/;
#include "skeleton.dtsi"
/{
	aliases {
		console = "/serial@e2900000";
	};

	serial@e2900000 {
		compatible = "samsung,exynos4210-uart";
		reg = <0xe2900000 0x100>;
		interrupts = <0 51 0>;
		id = <0>;
	};
};
```

dts 的内容这里不多说了。

10.6. uboot DM 的初始化

关于下面可能使用到的一些 FDT 的 API 可以参考一下《[uboot] （番外篇）uboot 之 fdt 介绍》。

关于下面可能使用到一些 DM 的 API 可以参考一下上述第四节。

10.6.1. 主要工作

DM 的初始化

创建根设备 root 的 udevice，存放在 gd->dm_root 中。

根设备其实是一个虚拟设备，主要是为 uboot 的其他设备提供一个挂载点。

初始化 uclass 链表 gd->uclass_root

DM 中 udevice 和 uclass 的解析

udevice 的创建和 uclass 的创建

udevice 和 uclass 的绑定

uclass_driver 和 uclass 的绑定

driver 和 udevice 的绑定

部分 driver 函数的调用

10.6.2. 入口说明

dm 初始化的接口在 dm_init_and_scan 中。

可以发现在 uboot relocate 之前的 initf_dm 和之后的 initr_dm 都调用了这个函数。

```
static int initf_dm(void)
{
#if defined(CONFIG_DM) && defined(CONFIG_SYS_MALLOC_F_LEN)
    int ret;
    ret = dm_init_and_scan(true); // 调用 dm_init_and_scan 对 DM 进行初始化和设备的解析
    if (ret)
        return ret;
#endif
    return 0;
}

#ifdef CONFIG_DM
static int initr_dm(void)
{
```

```
    int ret;
    /* Save the pre-reloc driver model and start a new one */
    gd->dm_root_f = gd->dm_root; // 存储 relocate 之前的根设备
    gd->dm_root = NULL;
    ret = dm_init_and_scan(false); // 调用 dm_init_and_scan 对 DM 进行初始化和设备的解析
    if (ret)
        return ret;
    return 0;
}
#endif
```

主要区别在于参数。

首先说明一下 dts 节点中的“u-boot,dm-pre-reloc”属性，当设置了这个属性时，则表示这个设备在 relocate 之前就需要使用。

当 dm_init_and_scan 的参数为 true 时，只会对带有“u-boot,dm-pre-reloc”属性的节点进行解析。而当参数为 false 的时候，则会对所有节点都进行解析。

由于“u-boot,dm-pre-reloc”的情况比较少，所以这里只学习参数为 false 的情况。也就是 initr_dm 里面的 dm_init_and_scan(false);。

10.6.3. dm_init_and_scan 说明

driver/core/root.c

```
int dm_init_and_scan(bool pre_reloc_only)
{
    int ret;

    ret = dm_init();    // DM 的初始化
    if (ret) {
        debug("dm_init() failed: %d\n", ret);
        return ret;
    }
```

```
    ret = dm_scan_platdata(pre_reloc_only); //  从平台设备中解析 udevice 和 uclass
    if (ret) {
        debug("dm_scan_platdata() failed: %d\n", ret);
        return ret;
    }

    if (CONFIG_IS_ENABLED(OF_CONTROL)) {
        ret = dm_scan_fdt(gd->fdt_blob, pre_reloc_only); // 从 dtb 中解析 udevice 和 uclass
        if (ret) {
            debug("dm_scan_fdt() failed: %d\n", ret);
            return ret;
        }
    }

    ret = dm_scan_other(pre_reloc_only);
    if (ret)
        return ret;

    return 0;
}
```

10.6.4. DM 的初始化——dm_init

对应代码如下：

driver/core/root.c

```
#define DM_ROOT_NON_CONST       (((gd_t *)gd)->dm_root) // 宏定义根设备指针 gd->dm_root
#define DM_UCLASS_ROOT_NON_CONST    (((gd_t *)gd)->uclass_root) // 宏定义 gd->uclass_root，uclass 的链表

int dm_init(void)
```

```
{
    int ret;

    if (gd->dm_root) {
    // 根设备已经存在，说明 DM 已经初始化过了
        dm_warn("Virtual root driver already exists!\n");
        return -EINVAL;
    }

    INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);
        // 初始化 uclass 链表

    ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);
        // DM_ROOT_NON_CONST 是指根设备 udevice，root_info 是表示根设备的设备信息
        // device_bind_by_name 会查找和设备信息匹配的 driver，然后创建对应的 udevice 和 uclass 并进行绑定，最后放在 DM_ROOT_NON_CONST 中。
        // device_bind_by_name 后续我们会进行说明，这里我们暂时只需要了解 root 根设备的 udevice 以及对应的 uclass 都已经创建完成。

    if (ret)
        return ret;
#if CONFIG_IS_ENABLED(OF_CONTROL)
    DM_ROOT_NON_CONST->of_offset = 0;
#endif
    ret = device_probe(DM_ROOT_NON_CONST);
        // 对根设备执行 probe 操作，
        // device_probe 后续再进行说明

    if (ret)
        return ret;
```

```
    return 0;
}
```

这里就完成的 DM 的初始化了

（1）创建根设备 root 的 udevice，存放在 gd->dm_root 中。

（2）初始化 uclass 链表 gd->uclass_root

10.6.5. 从平台设备中解析 udevice 和 uclass——dm_scan_platdata

跳过。

10.6.6. 从 dtb 中解析 udevice 和 uclass——dm_scan_fdt

关于 fdt 以及一些对应 API 请参考《[uboot] （番外篇）uboot 之 fdt 介绍》。

对应代码如下（后续我们忽略 pre_reloc_only=true 的情况）：

driver/core/root.c

```
int dm_scan_fdt(const void *blob, bool pre_reloc_only)
// 此时传进来的参数 blob=gd->fdt_blob，  pre_reloc_only=0
{
    return dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);
// 直接调用 dm_scan_fdt_node
}

int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
            bool pre_reloc_only)
// 此时传进来的参数
// parent=gd->dm_root，表示以 root 设备作为父设备开始解析
// blob=gd->fdt_blob，指定了对应的 dtb
// offset=0，从偏移 0 的节点开始扫描
```

```
// pre_reloc_only=0，不只是解析 relotion 之前的设备
{
    int ret = 0, err;

        /*  以下步骤相当于是遍历每一个 dts 节点并且调用 lists_bind_fdt 对其进行解析  */

    for (offset = fdt_first_subnode(blob, offset);
        // 获得 blob 设备树的 offset 偏移下的节点的第一个子节点
         offset > 0;
         offset = fdt_next_subnode(blob, offset)) {
              // 循环查找下一个子节点
        if (!fdtdec_get_is_enabled(blob, offset)) {
                    // 判断节点状态是否是 disable，如果是的话直接忽略
            dm_dbg("   - ignoring disabled device\n");
            continue;
        }
        err = lists_bind_fdt(parent, blob, offset, NULL);
              // 解析绑定这个节点，dm_scan_fdt 的核心，下面具体分析
        if (err && !ret) {
            ret = err;
            debug("%s: ret=%d\n", fdt_get_name(blob, offset, NULL),
                  ret);
        }
    }
    return ret;
}
```

lists_bind_fdt 是从 dtb 中解析 udevice 和 uclass 的核心。

其具体实现如下：

driver/core/lists.c

```
int lists_bind_fdt(struct udevice *parent, const void *blob, int offset,
        struct udevice **devp)
// parent 指定了父设备，通过 blob 和 offset 可以获得对应的设备的 dts 节点，对应 udevice 结构通过 devp 返回
{
    struct driver *driver = ll_entry_start(struct driver, driver);
// 获取 driver table 地址
    const int n_ents = ll_entry_count(struct driver, driver);
// 获取 driver table 长度
    const struct udevice_id *id;
    struct driver *entry;
    struct udevice *dev;
    bool found = false;
    const char *name;
    int result = 0;
    int ret = 0;

    dm_dbg("bind node %s\n", fdt_get_name(blob, offset, NULL));
// 打印当前解析的节点的名称
    if (devp)
        *devp = NULL;
    for (entry = driver; entry != driver + n_ents; entry++) {
// 遍历 driver table 中的所有 driver，具体参考三、4 一节
        ret = driver_check_compatible(blob, offset, entry->of_match,
                      &id);
// 判断 driver 中的 compatibile 字段和 dts 节点是否匹配
        name = fdt_get_name(blob, offset, NULL);
// 获取节点名称
        if (ret == -ENOENT) {
            continue;
```

```
        } else if (ret == -ENODEV) {
            dm_dbg("Device '%s' has no compatible string\n", name);
            break;
        } else if (ret) {
            dm_warn("Device tree error at offset %d\n", offset);
            result = ret;
            break;
        }

        dm_dbg("   - found match at '%s'\n", entry->name);
        ret = device_bind(parent, entry, name, NULL, offset, &dev);
// 找到对应的 driver，调用 device_bind 进行绑定，会在这个函数中创建对应 udevice 和 uclass 并切进行绑定，后面继续说明
        if (ret) {
            dm_warn("Error binding driver '%s': %d\n", entry->name,
                ret);
            return ret;
        } else {
            dev->driver_data = id->data;
            found = true;
            if (devp)
                *devp = dev;
// 将 udevice 设置到 devp 指向的地方中，进行返回
        }
        break;
    }

    if (!found && !result && ret != -ENODEV) {
        dm_dbg("No match for node '%s'\n",
            fdt_get_name(blob, offset, NULL));
```

```
    }


    return result;
}
```

在 device_bind 中实现了 udevice 和 uclass 的创建和绑定以及一些初始化操作，这里专门学习一下 device_bind。

device_bind 的实现如下(去除部分代码)

driver/core/device.c

```
int device_bind(struct udevice *parent, const struct driver *drv,
        const char *name, void *platdata, int of_offset,
        struct udevice **devp)
// parent:父设备
// drv：设备对应的 driver
// name：设备名称
// platdata：设备的平台数据指针
// of_offset：在 dtb 中的偏移，即代表了其 dts 节点
// devp：所创建的 udevice 的指针，用于返回
{
    struct udevice *dev;
    struct uclass *uc;
    int size, ret = 0;


    ret = uclass_get(drv->id, &uc);
        // 获取 driver id 对应的 uclass，如果 uclass 原先并不存在，那么会在这里创建 uclass 并其 uclass_driver 进行绑定


    dev = calloc(1, sizeof(struct udevice));
        // 分配一个 udevice
```

```
dev->platdata = platdata; // 设置 udevice 的平台数据指针
dev->name = name; // 设置 udevice 的 name
dev->of_offset = of_offset; // 设置 udevice 的 dts 节点偏移
dev->parent = parent; // 设置 udevice 的父设备
dev->driver = drv;    // 设置 udevice 的对应的 driver，相当于 driver 和 udevice 的绑定
dev->uclass = uc;    // 设置 udevice 的所属 uclass

dev->seq = -1;
dev->req_seq = -1;
if (CONFIG_IS_ENABLED(OF_CONTROL) && CONFIG_IS_ENABLED(DM_SEQ_ALIAS)) {
    /*
     * Some devices, such as a SPI bus, I2C bus and serial ports
     * are numbered using aliases.
     *
     * This is just a 'requested' sequence, and will be
     * resolved (and ->seq updated) when the device is probed.
     */
    if (uc->uc_drv->flags & DM_UC_FLAG_SEQ_ALIAS) {
        if (uc->uc_drv->name && of_offset != -1) {
            fdtdec_get_alias_seq(gd->fdt_blob,
                    uc->uc_drv->name, of_offset,
                    &dev->req_seq);
        }
            // 设置 udevice 的 alias 请求序号
    }
}

if (!dev->platdata && drv->platdata_auto_alloc_size) {
    dev->flags |= DM_FLAG_ALLOC_PDATA;
    dev->platdata = calloc(1, drv->platdata_auto_alloc_size);
```

```
        // 为 udevice 分配平台数据的空间，由 driver 中的 platdata_auto_alloc_size 决定
    }

    size = uc->uc_drv->per_device_platdata_auto_alloc_size;
    if (size) {
        dev->flags |= DM_FLAG_ALLOC_UCLASS_PDATA;
        dev->uclass_platdata = calloc(1, size);
            // 为 udevice 分配给其所属 uclass 使用的平台数据的空间，由所属 uclass 的 driver 中的 per_device_platdata_auto_alloc_size 决定
    }

    /* put dev into parent's successor list */
    if (parent)
        list_add_tail(&dev->sibling_node, &parent->child_head);
        // 添加到父设备的子设备链表中

    ret = uclass_bind_device(dev);
        // uclass 和 udevice 进行绑定，主要是实现了将 udevice 链接到 uclass 的设备链表中

    /* if we fail to bind we remove device from successors and free it */
    if (drv->bind) {
        ret = drv->bind(dev);
        // 执行 udevice 对应 driver 的 bind 函数
    }

    if (parent && parent->driver->child_post_bind) {
        ret = parent->driver->child_post_bind(dev);
        // 执行父设备的 driver 的 child_post_bind 函数
    }
    if (uc->uc_drv->post_bind) {
```

```
    ret = uc->uc_drv->post_bind(dev);
    if (ret)
        goto fail_uclass_post_bind;
    // 执行所属 uclass 的 post_bind 函数
}

if (devp)
    *devp = dev;
    // 将 udevice 进行返回

dev->flags |= DM_FLAG_BOUND;
    // 设置已经绑定的标志
    // 后续可以通过 dev->flags & DM_FLAG_ACTIVATED 或者 device_active 宏来判断设备是否已经被激活

return 0;
```

上述就完成了 dtb 的解析，udevice 和 uclass 的创建，以及各个组成部分的绑定关系。

注意，这里只是绑定，即调用了 driver 的 bind 函数，但是设备还没有真正激活，也就是还没有执行设备的 probe 函数。

10.7. DM 工作流程

经过前面的 DM 初始化以及设备解析之后，我们只是建立了 udevice 和 uclass 之间的绑定关系。但是此时 udevice 还没有被 probe，其对应设备还没有被激活。

激活一个设备主要是通过 device_probe 函数，所以在介绍 DM 的工作流程前，先说明 device_probe 函数。

10.7.1. device_probe

driver/core/device.c

```
int device_probe(struct udevice *dev)
{
    const struct driver *drv;
    int size = 0;
    int ret;
    int seq;

    if (dev->flags & DM_FLAG_ACTIVATED)
        return 0;
// 表示这个设备已经被激活了

    drv = dev->driver;
    assert(drv);
// 获取这个设备对应的 driver

    /* Allocate private data if requested and not reentered */
    if (drv->priv_auto_alloc_size && !dev->priv) {
        dev->priv = alloc_priv(drv->priv_auto_alloc_size, drv->flags);
// 为设备分配私有数据
    }

    /* Allocate private data if requested and not reentered */
    size = dev->uclass->uc_drv->per_device_auto_alloc_size;
    if (size && !dev->uclass_priv) {
        dev->uclass_priv = calloc(1, size);
// 为设备所属 uclass 分配私有数据
    }

// 这里过滤父设备的 probe
```

```
    seq = uclass_resolve_seq(dev);
    if (seq < 0) {
        ret = seq;
        goto fail;
    }
    dev->seq = seq;

    dev->flags |= DM_FLAG_ACTIVATED;
// 设置 udevice 的激活标志

    ret = uclass_pre_probe_device(dev);
// uclass 在 probe device 之前的一些函数的调用

    if (drv->ofdata_to_platdata && dev->of_offset >= 0) {
        ret = drv->ofdata_to_platdata(dev);
// 调用 driver 中的 ofdata_to_platdata 将 dts 信息转化为设备的平台数据
    }

    if (drv->probe) {
        ret = drv->probe(dev);
// 调用 driver 的 probe 函数，到这里设备才真正激活了
    }

    ret = uclass_post_probe_device(dev);

    return ret;
}
```

主要工作归纳如下：

分配设备的私有数据

对父设备进行 probe

执行 probe device 之前 uclass 需要调用的一些函数

调用 driver 的 ofdata_to_platdata，将 dts 信息转化为设备的平台数据

调用 driver 的 probe 函数

执行 probe device 之后 uclass 需要调用的一些函数

10.7.2. 通过 uclass 来获取一个 udevice 并且进行 probe

通过 uclass 来获取一个 udevice 并且进行 probe 有如下接口

driver/core/uclass.c

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp) //通过索引从 uclass 的设备链表中获取 udevice，并且进行 probe

int uclass_get_device_by_name(enum uclass_id id, const char *name,

struct udevice **devp) //通过设备名从 uclass 的设备链表中获取 udevice，并且进行 probe

int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp) //通过序号从 uclass 的设备链表中获取 udevice，并且进行 probe

int uclass_get_device_by_of_offset(enum uclass_id id, int node,

struct udevice **devp) //通过 dts 节点的偏移从 uclass 的设备链表中获取 udevice，并且进行 probe

int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,

const char *name, struct udevice **devp) //通过设备的“phandle”属性从 uclass 的设备链表中获取 udevice，并且进行 probe

int uclass_first_device(enum uclass_id id, struct udevice **devp) //从 uclass 的设备链表中获取第一个 udevice，并且进行 probe

int uclass_next_device(struct udevice **devp) //从 uclass 的设备链表中获取下一个 udevice，并且进行 probe

这些接口主要是获取设备的方法上有所区别，但是 probe 设备的方法都是一样的，都是通过调用 uclass_get_device_tail->device_probe 来 probe 设备的。

以 uclass_get_device 为例

```
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)
{
    struct udevice *dev;
    int ret;

    *devp = NULL;
    ret = uclass_find_device(id, index, &dev); //通过索引从 uclass 的设备链表中获取对应的 udevice
    return uclass_get_device_tail(dev, ret, devp); // 调用 uclass_get_device_tail 进行设备的 get，最终会调用 device_probe 来对设备进行 probe
}

int uclass_get_device_tail(struct udevice *dev, int ret,
                struct udevice **devp)
{
    ret = device_probe(dev);
// 调用 device_probe 对设备进行 probe，这个函数在前面说明过了
    if (ret)
        return ret;

    *devp = dev;

    return 0;
}
```

10.7.3. 工作流程简单说明

serial-uclass 较为简单，我们以 serial-uclass 为例

（0）代码支持

< 1 > serial-uclass.c 中定义一个 uclass_driver

```
UCLASS_DRIVER(serial) = {
    .id         = UCLASS_SERIAL, //注意这里的 uclass id
    .name       = "serial",
    .flags      = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe    = serial_post_probe,
    .pre_remove    = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
};
```

< 2 > 定义 s5pv210 的 serial 的 dts 节点

```
        serial@e2900000 {
                compatible = "samsung,exynos4210-uart"; //注意这里的 compatible
                reg = <0xe2900000 0x100>;
                interrupts = <0 51 0>;
                id = <0>;
        };
```

< 3 > 定义设备驱动

```
U_BOOT_DRIVER(serial_s5p) = {
    .name    = "serial_s5p",
    .id    = UCLASS_SERIAL, //注意这里的 uclass id
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

```
static const struct udevice_id s5p_serial_ids[] = {
    { .compatible = "samsung,exynos4210-uart" }, //注意这里的 compatible
    { }
};
```

（1）udevice 和对应 uclass 的创建

在 DM 初始化的过程中 uboot 自己创建对应的 udevice 和 uclass。

具体参考“六、uboot DM 的初始化”

（2）udevice 和对应 uclass 的绑定

在 DM 初始化的过程中 uboot 自己实现将 udevice 绑定到对应的 uclass 中。

具体参考“六、uboot DM 的初始化”

（3）对应 udevice 的 probe

由模块自己实现。例如 serial 则需要在 serial 的初始化过程中，选择需要的 udevice 进行 probe。

serial-uclass 只是操作作为 console 的 serial，并不具有通用性，这里简单的了解下。

代码如下，过滤掉无关代码

driver/serial/serial-uclass.c

```
int serial_init(void)
{
    serial_find_console_or_panic(); // 调用 serial_find_console_or_panic 进行作为 console 的 serial 的初始化
    gd->flags |= GD_FLG_SERIAL_READY;

    return 0;
}

static void serial_find_console_or_panic(void)
```

```
{
    const void *blob = gd->fdt_blob;
    struct udevice *dev;
    int node;

    if (CONFIG_IS_ENABLED(OF_CONTROL) && blob) {
        /* Check for a chosen console */

// 这里过滤掉获取指定的 serial 的 dts 节点的代码

        if (!uclass_get_device_by_of_offset(UCLASS_SERIAL, node,
                        &dev)) {
// 这里调用 uclass_get_device_by_of_offset，通过 dts 节点的偏移从 uclass 的设备链表中获取 udevice，并且进行 probe。
// 注意，是在这里完成设备的 probe 的！！！
            gd->cur_serial_dev = dev;
// 将 udevice 存储在 gd->cur_serial_dev，后续 uclass 中可以直接通过 gd->cur_serial_dev 获取到对应的设备并且进行操作
// 但是注意，这种并不是通用做法！！！
            return;
        }
    }
}
```

（4）uclass 的接口调用

可以通过先从 root_uclass 链表中提取对应的 uclass，然后通过 uclass->uclass_driver->ops 来进行接口调用，这种方法比较具有通用性。

可以通过调用 uclass 直接 expert 的接口，不推荐，但是 serial-uclass 使用的是这种方式。

这部分应该属于 serial core，但是也放在了 serial-uclass.c 中实现。

以 serial_putc 调用为例，serial-uclass 使用如下：

```
void serial_putc(char ch)
```

```
{
    if (gd->cur_serial_dev)
        _serial_putc(gd->cur_serial_dev, ch);// 将 console 对应的 serial 的 udevice 作为参数传入
}

static void _serial_putc(struct udevice *dev, char ch)
{
    struct dm_serial_ops *ops = serial_get_ops(dev);// 获取设备对应的 driver 函数的 ops 操作集
    int err;

    do {
        err = ops->putc(dev, ch); // 以 udevice 为参数，调用 ops 中对应的操作函数
    } while (err == -EAGAIN);
}
```

到此整个流程简单介绍到这。

这里几乎都是纸上谈兵，后续会来一篇 gpio-uclass 的使用实战。

11. 番外-uboot 之 fdt 介绍

11.1. 介绍

FDT，flatted device tree，扁平设备树。熟悉 linux 的人对这个概念应该不陌生。

简单理解为将部分设备信息结构存放到 device tree 文件中。

uboot 最终将其 device tree 编译成 dtb 文件，使用过程中通过解析该 dtb 来获取板级设备信息。

uboot 的 dtb 和 kernel 中的 dtb 是一致的。这部分建议直接参考 wowo 的 dtb 的文章

Device Tree（一）：背景介绍

Device Tree（二）：基本概念

Device Tree（三）：代码分析

关于 uboot 的 fdt，可以参考 doc/README.fdt-control。

11.2. dtb 介绍

11.2.1. dtb 结构介绍

结构体如下
DTB header
alignment gap
memory reserve map
alignment gap
device-tree structure
alignment gap
device-tree string

dtb header 结构如下：

结构体如下
magic
totalsize
off_dt_struct
off_dt_strings
off_mem_rsvmap
version
……

其中，magic 是一个固定的值，0xd00dfeed（大端）或者 0xedfe0dd0（小端）。

以 s5pv210-tiny210.dtb 为例：

执行”hexdump -C s5pv210-tiny210.dtb | more”命令

@:dts$ hexdump -C s5pv210-tiny210.dtb | more

00000000 d0 0d fe ed 00 00 5a cc 00 00 00 38 00 00 58 14 |......Z....8..X.|

00000010 00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00 |...(............|

可以看到 dtb 的前面 4 个字节就是 0xd00dfeed，也就是 magic。

综上，我们只要提取待验证 dtb 的地址上的数据的前四个字节，与 0xd00dfeed（大端）或者 0xedfe0dd0（小端）进行比较，如果匹配的话，就说明对应待验证 dtb 就是一个合法的 dtb。

11.2.2. dtb 在 uboot 中的位置

dtb 可以以两种形式编译到 uboot 的镜像中。

dtb 和 uboot 的 bin 文件分离

如何使能

需要打开 CONFIG_OF_SEPARATE 宏来使能。

编译说明

在这种方式下，uboot 的编译和 dtb 的编译是分开的，先生成 uboot 的 bin 文件，然后再另外生成 dtb 文件。

具体参考《[uboot] （第四章）uboot 流程一一uboot 编译流程》。

最终位置

dtb 最终会追加到 uboot 的 bin 文件的最后面。也就是 uboot.img 的最后一部分。

因此，可以通过 uboot 的结束地址符号，也就是_end 符号来获取 dtb 的地址。

具体参考《[uboot] （第四章）uboot 流程一一uboot 编译流程》。

dtb 集成到 uboot 的 bin 文件内部

如何使能

需要打开 CONFIG_OF_EMBED 宏来使能。

编译说明

在这种方式下，在编译 uboot 的过程中，也会编译 dtb。

最终位置

注意：最终 dtb 是包含到了 uboot 的 bin 文件内部的。

dtb 会位于 uboot 的.dtb.init.rodata 段中，并且在代码中可以通过__dtb_dt_begin 符号获取其符号。

因为这种方式不够灵活，文档上也不推荐，所以后续也不具体研究，简单了解一下即可。

另外，也可以通过 fdtcontroladdr 环境变量来指定 dtb 的地址

可以通过直接把 dtb 加载到内存的某个位置，并在环境变量中设置 fdtcontroladdr 为这个地址，达到动态指定 dtb 的目的。

在调试中使用。

11.3. uboot 中如何支持 fdt

11.3.1. 相关的宏

CONFIG_OF_CONTROL

用于配置是否使能 FDT。

./source/configs/tiny210_defconfig:312:CONFIG_OF_CONTROL=y

CONFIG_OF_SEPARATE、CONFIG_OF_EMBED

配置 dtb 是否集成到 uboot 的 bin 文件中。具体参考上述。一般都是使用分离的方式。

11.3.2. 如何添加一个 dtb

以 tiny210 为例，具体可以参考 project X 项目中 uboot 的 git 记录:
8a371676710cc0572a0a863255e25c35c82bb928

（1）在 Makefile 中添加对应的目标 dtb

arch/arm/dts/Makefile

```
dtb-$(CONFIG_TARGET_TINY210) += \
    s5pv210-tiny210.dtb
```

（2）创建对应的 dts 文件

arch/arm/dts/s5pv210-tiny210.dts,注意文件名要和 Makefile 中的 dtb 名一致

```
/dts-v1/;
```

/{

};

（3）打开对应的宏

configs/tiny210_defconfig

CONFIG_OF_CONTROL=y

CONFIG_OF_SEPARATE=y ##其实这里不用配，arm 默认就是指定这种方式

（4）因为最终的编译出来的 dtb 可能会多个，这里需要为 tiny210 指定一个 dtb

configs/tiny210_defconfig

CONFIG_DEFAULT_DEVICE_TREE="s5pv210-tiny210"

编译，解决一些编译错误，就可以发现最终生成了 u-boot.dtb 文件。

通过如下“hexdump -C u-boot.dtb | more”命令可以查看我们的 dtb 文件，得到部分内容如下：

```
hlos@node4:u-boot$ hexdump -C u-boot.dtb | more
00000000  d0 0d fe ed 00 00 01 a4  00 00 00 38 00 00 01 58  |...........8...X|
00000010  00 00 00 28 00 00 00 11  00 00 00 10 00 00 00 00  |...(............|
00000020  00 00 00 4c 00 00 01 20  00 00 00 00 00 00 00 00  |...L... ........|
00000030  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00  |................|
```

11.4. uboot 中如何获取 dtb

11.4.1. 整体说明

在 uboot 初始化过程中，需要对 dtb 做两个操作：

获取 dtb 的地址，并且验证 dtb 的合法性

因为我们使用的 dtb 并没有集成到 uboot 的 bin 文件中，也就是使用的 CONFIG_OF_SEPARATE 方式。因此，在 relocate uboot 的过程中并不会去 relocate dtb。因此，这里我们还需要自行为 dtb 预留内存空间并进行 relocate。关于 uboot relocate 的内容请参考《[uboot] （番外篇）uboot relocation 介绍》。

relocate 之后，还需要重新获取一次 dtb 的地址。

这部分过程是在 init_board_f 中实现，参考《[uboot] （第五章）uboot 流程一一uboot 启动流程》。

对应代码 common/board_f.c

```
static init_fnc_t init_sequence_f[] = {
...
#ifdef CONFIG_OF_CONTROL
    fdtdec_setup, // 获取 dtb 的地址，并且验证 dtb 的合法性
#endif
...
    reserve_fdt, // 为 dtb 分配新的内存地址空间
...
    reloc_fdt, // relocate dtb
...
}
```

后面进行具体函数的分析。

11.4.2. 获取 dtb 的地址，并且验证 dtb 的合法性（fdtdec_setup）

对应代码如下：

lib/fdtdec.c

```
int fdtdec_setup(void)
{
#if CONFIG_IS_ENABLED(OF_CONTROL) // 确保 CONFIG_OF_CONTROL 宏是打开的

# ifdef CONFIG_OF_EMBED
    /* Get a pointer to the FDT */
    gd->fdt_blob = __dtb_dt_begin;
// 当使用 CONFIG_OF_EMBED 的方式时，也就是 dtb 集成到 uboot 的 bin 文件中时，通过 __dtb_dt_begin 符号来获取 dtb 地址。

# elif defined CONFIG_OF_SEPARATE
    /* FDT is at end of image */
    gd->fdt_blob = (ulong *)&_end;
```

//当使用 CONFIG_OF_SEPARATE 的方式时，也就是 dtb 追加到 uboot 的 bin 文件后面时，通过 _end 符号来获取 dtb 地址。

endif

```
    /* Allow the early environment to override the fdt address */
    gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
                        (uintptr_t)gd->fdt_blob);
```

// 可以通过环境变量 fdtcontroladdr 来指定 gd->fdt_blob，也就是指定 fdt 的地址。

#endif

```
// 最终都把 dtb 的地址存储在 gd->fdt_blob 中
    return fdtdec_prepare_fdt();
// 在 fdtdec_prepare_fdt 中检查 fdt 的合法性
}

/* fdtdec_prepare_fdt 实现如下  */
int fdtdec_prepare_fdt(void)
{
    if (!gd->fdt_blob || ((uintptr_t)gd->fdt_blob & 3) ||
        fdt_check_header(gd->fdt_blob)) {
        puts("No valid device tree binary found - please append one to U-Boot binary, use u-boot-dtb.bin or define CONFIG_OF_EMBED. For sandbox, use -d <file.dtb>\n");
        return -1;
// 判断 dtb 是否存在，以及是否有四个字节对齐。
// 然后再调用 fdt_check_header 看看头部是否正常。fdt_check_header 主要是检查 dtb 的 magic 是否正确。
    }
    return 0;
}
```

验证 dtb 的部分可以参考《[kernel 启动流程]（第四章）第一阶段之一一dtb 的验证》。

11.4.3. 为 dtb 分配新的内存地址空间（reserve_fdt）

relocate 的内容请参考《[uboot] （番外篇）uboot relocation 介绍》。

common/board_f.c 中

```
static int reserve_fdt(void)
{
#ifndef CONFIG_OF_EMBED
// 当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
// 当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate
    if (gd->fdt_blob) {
        gd->fdt_size = ALIGN(fdt_totalsize(gd->fdt_blob) + 0x1000, 32);
// 获取 dtb 的 size

        gd->start_addr_sp -= gd->fdt_size;
        gd->new_fdt = map_sysmem(gd->start_addr_sp, gd->fdt_size);
// 为 dtb 分配新的内存空间
        debug("Reserving %lu Bytes for FDT at: %08lx\n",
              gd->fdt_size, gd->start_addr_sp);
    }
#endif

    return 0;
}
```

11.4.4. relocate dtb（reloc_fdt）

relocate 的内容请参考《[uboot] （番外篇）uboot relocation 介绍》。

common/board_f.c 中

```
static int reloc_fdt(void)
{
```

```
#ifndef CONFIG_OF_EMBED
// 当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
// 当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate
    if (gd->flags & GD_FLG_SKIP_RELOC)
// 检查 GD_FLG_SKIP_RELOC 标识
        return 0;
    if (gd->new_fdt) {
        memcpy(gd->new_fdt, gd->fdt_blob, gd->fdt_size);
// relocate dtb 空间
        gd->fdt_blob = gd->new_fdt;
// 切换 gd->fdt_blob 到 dtb 的新的地址空间上
    }
#endif

    return 0;
}
```

11.5. uboot 中 dtb 解析的常用接口

gd->fdt_blob 已经设置成了 dtb 的地址了。

注意，fdt 提供的接口都是以 gd->fdt_blob（dtb 的地址）为参数的。

11.5.1. 接口功能

以下只简单说明几个接口的功能，没有深究到实现原理。先说明几个，后续继续补充。

另外，用节点在 dtb 中的偏移地址来表示一个节点。也就是节点变量 node 中，存放的是节点的偏移地址

lib/fdtdec.c 中

fdt_path_offset

int fdt_path_offset(const void *fdt, const char *path)

eg：node = fdt_path_offset(gd->fdt_blob, “/aliases”);

功能：获得 dtb 下某个节点的路径 path 的偏移。这个偏移就代表了这个节点。

fdt_getprop

const void *fdt_getprop(const void *fdt, int nodeoffset, const char *name, int *lenp)

eg： mac = fdt_getprop(gd->fdt_blob, node, “mac-address”, &len);

功能：获得节点 node 的某个字符串属性值。

fdtdec_get_int_array、fdtdec_get_byte_array

int fdtdec_get_int_array(const void *blob, int node, const char *prop_name, u32 *array, int count)

eg： ret = fdtdec_get_int_array(blob, node, “interrupts”, cell, ARRAY_SIZE(cell));

功能：获得节点 node 的某个整形数组属性值。

fdtdec_get_addr

fdt_addr_t fdtdec_get_addr(const void *blob, int node, const char *prop_name)

eg：fdtdec_get_addr(blob, node, “reg”);

功能：获得节点 node 的地址属性值。

fdtdec_get_config_int、fdtdec_get_config_bool、fdtdec_get_config_string

功能：获得 config 节点下的整形属性、bool 属性、字符串等等。

fdtdec_get_chosen_node

int fdtdec_get_chosen_node(const void *blob, const char *name)

功能：获得 chosen 下的 name 节点的偏移

fdtdec_get_chosen_prop

const char *fdtdec_get_chosen_prop(const void *blob, const char *name)

功能：获得 chosen 下 name 属性的值

lib/fdtdec_common.c 中

fdtdec_get_int

int fdtdec_get_int(const void *blob, int node, const char *prop_name, int default_val)

eg： bus->udelay = fdtdec_get_int(blob, node, “i2c-gpio,delay-us”, DEFAULT_UDELAY);

功能：获得节点 node 的某个整形属性值。

fdtdec_get_uint

功能：获得节点 node 的某个无符号整形属性值。

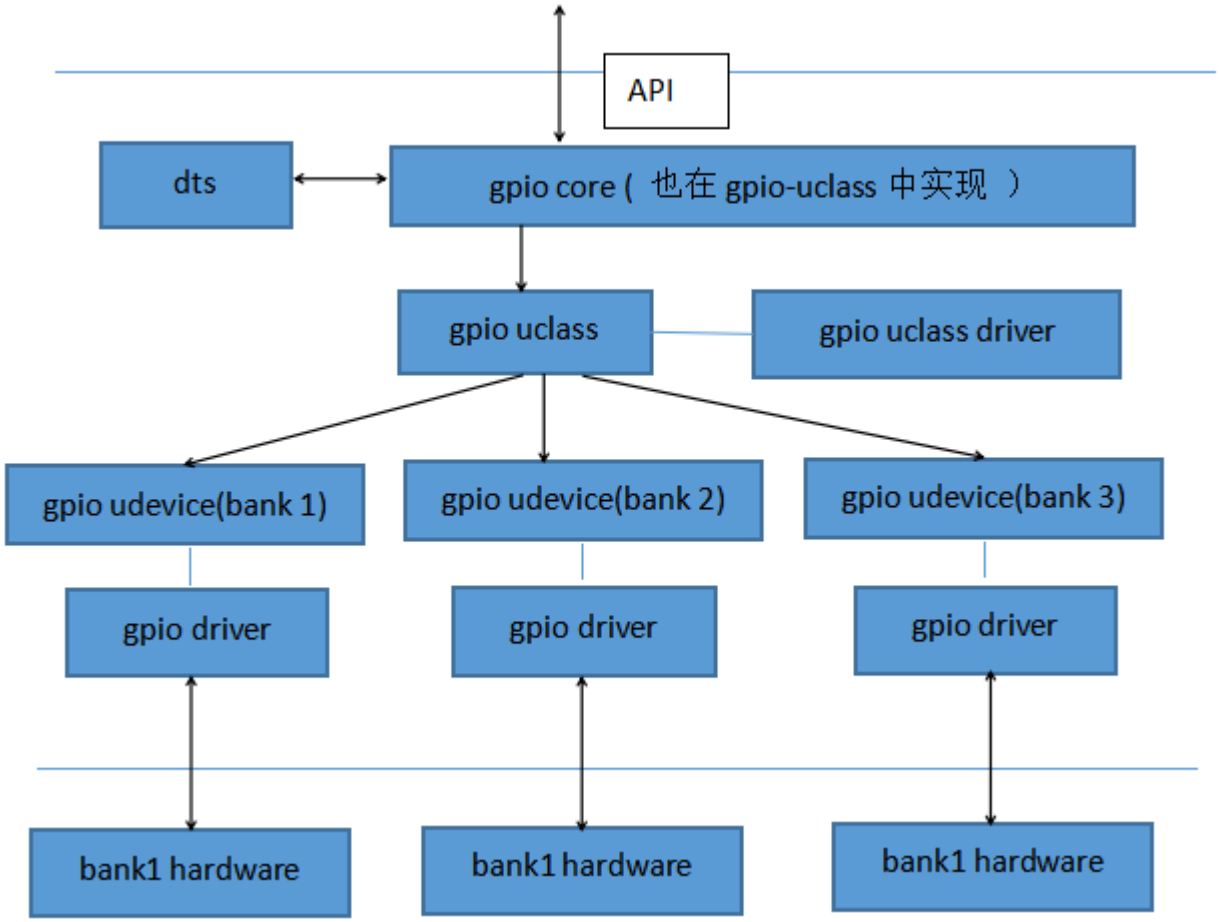
12. 番外 dm-gpio 使用方法以及工作流程

12.1. dm-gpio 架构

建议先参考《[uboot] （番外篇）uboot 驱动模型》搞懂 uboot 下的驱动模型。

12.1.1. dm-gpio 架构图

如下：



12.1.2. 说明

gpio core

也在 gpio uclass 中实现，

主要是为上层提供接口

从 dts 中获取 GPIO 属性

从 gpio uclass 的设备链表中获取到相应的 udevice 设备，并使用其操作集

gpio uclass

链接属于该 uclass 的所有 gpio udevice 设备

为 gpio udevice 的 driver 提供统一的操作集接口

bank 和 gpio

有些平台上，将某些使用同一组寄存器的 gpio 构成一个 bank，例如三星的 s5pv210

当然，并不是所有平台都有 bank 的概念，例如高通，高通的 GPIO 都有自己独立的寄存器，因此，可以将高通当成只有一个 bank

gpio udevice

一个 bank 对应一个 gpio udevice，用 bank 中的偏移来表示具体的 GPIO 号

gpio udevice 的 driver 就会根据 bank 以及 offset 对相应的寄存器上的相应的 bit 进行操作。

12.1.3. 原理介绍

这里先简单介绍一下 dm 下 gpio 的工作原理

一个 bank 对应一个 udevice，udevice 中私有数据中存放着该 bank 的信息，比如相应寄存器地址等等

上层用 gpio_desc 描述符来描述一个 GPIO，其中包括该 GPIO 所属的 udevice、在 bank 内的偏移、以及标志位。

上层通过调用 gpio core 的接口从 dtsi 获取到 GPIO 属性对应的 gpio_desc 描述符

上层使用 gpio_desc 描述符来作为调用 gpio core 的操作接口的参数

gpio core 从 gpio_desc 描述符提取 udevice，并调用其 driver 中对应的操作集，以 bank 内的偏移作为其参数（这样 driver 就能判断出是哪个 GPIO 了）

driver 中提取 udevice 的私有数据中的 bank 信息，并进行相应的操作

12.2. dm-gpio 的使用示例

这里简单的以 tiny210 的 led_1 的闪烁为例子来介绍一下 GPIO 的使用。

虽然 uboot 中已经实现了 led 的 uclass，但是我们这里先不使用这个 uclass，只是简单的实现通过 dm-gpio 来控制一个 led。

12.2.1. 硬件简单说明（具体去参考原理图吧）

s5pv210 的 GPIO 的地址空间

参考《S5PV210_UM_REV1.1》2.1.2 SPECIAL FUNCTION REGISTER MAP，

s5pv210 的 GPIO 的地址空间为 0xE020_0000 开始的 1000B 的区域

在 tiny210 上，有四个 led，低电平为亮，高电平为灭

GPJ2_0 -> led_1

GPJ2_1 -> led_2

GPJ2_2 -> led_3

GPJ2_3 -> led_4

GPJ2 这个 bank 的寄存器

对应控制寄存器 0xE020_0280

对应数据寄存器 0xE020_0284

12.2.2. dtsi 中的定义

这里只说明 GPIO 引用的节点，不说明 gpio 控制器的节点，gpio 控制器的节点在后面 driver 中进行说明

arch/arm/dts/s5pv210-tiny210.dts

```
led_test {
    compatible = "dm-gpio,test";
    led1-gpio = <&gpj2 0 0>;     // 属性"led1-gpio"，表示 gpj2 bank 的第 0 个 gpio，0 是值标志
    led2-gpio = <&gpj2 1 0>;    // gpj2 bank 的第 1 个 gpio，0 是值标志
    led3-gpio = <&gpj2 2 0>;    // gpj2 bank 的第 2 个 gpio，0 是值标志
    led4-gpio = <&gpj2 3 0>;    // gpj2 bank 的第 3 个 gpio，0 是值标志
};
```

注意，这里<&gpj2 0 0>取决于 driver 是如何对这个属性进行转化的。并不是所有平台都是一样的。

12.2.3. 驱动中从节点中获取一个 GPIO

我们在 board_late_init 中来实现 GPIO 闪烁的功能

board/samsung/tiny210/board.c

```
int board_late_init(void)
{
    struct gpio_desc led1_gpio;
    int node;
```

```
node = fdt_node_offset_by_compatible(gd->fdt_blob, 0, "dm-gpio,test");
```

// 注意，我们并没有使用 dm 模型的 driver 来匹配 compatible = "dm-gpio,test"的节点，所以这里直接获取 compatible = "dm-gpio,test"的节点，存放在 node 中

```
if (node < 0)
{
    printf("============================Don't find dm-gpio,test node\n");
    return -1;
}

gpio_request_by_name_nodev(gd->fdt_blob, node, "led1-gpio", 0, &led1_gpio, GPIOD_IS_OUT);
```

// 调用 gpio_request_by_name_nodev 来获取 node 节点中的"led1-gpio"属性，并转化为 gpio_desc 描述符，标志为输出

```
if (dm_gpio_is_valid(&led1_gpio)）// 判断对应 gpio_desc 是否可用
{
    printf("============================Get led1-gpio\n");
    while(1)
    {
        dm_gpio_set_value(&led1_gpio, 0);  // 调用 dm_gpio_set_value 将 led1_gpio 设置输出低电平
        mdelay(1000);
        dm_gpio_set_value(&led1_gpio, 1);  // 调用 dm_gpio_set_value 将 led1_gpio 设置输出高电平
        mdelay(1000);
    }
}
else
{
    printf("============================Can't get led1-gpio\n");
}
```

12.2.4. 简单说明

在 dtsi 中将需要的 GPIO 以一定的格式在属性中描述，如下：

led1-gpio = <&gpj2 0 0>;

在驱动中调用 gpio_request_by_name_nodev 接口或者其他接口从 dtsi 节点中获取对应 GPIO 属性的 gpio_desc 描述符

调用 dm_gpio_is_valid 判断该 gpio_desc 是否可用

调用 gpio core 的 dm 操作接口，以 gpio_desc 为参数，实现控制 GPIO 的目的

后面介绍 dm-gpio 的实现

12.3. gpio uclass

建议先参考《[uboot] （番外篇）uboot 驱动模型》搞懂 uboot 下的驱动模型。

在 gpio uclass 中实现了两部分，

gpio uclass driver：gpio uclass 驱动，提供 gpio udevice 绑定到 uclass 的设备链表之前和之后的一些操作，以及 udevice 的 driver 的操作集规范。

gpio core：为上层提供 GPIO 的操作接口

12.3.1. 需要打开的宏

（1）CONFIG_DM

dm-gpio 是基于 uboot 的 DM 实现的，关于 uboot 的 DM 可以参考《[uboot] （番外篇）uboot 驱动模型》。

所以需要先使能 DM，也就是打开 CONFIG_DM 宏。

在 configs/tiny210_defconfig 中定义了如下：

CONFIG_DM=y

（2）打开 CONFIG_DM_GPIO 宏

在 configs/tiny210_defconfig 中定义了如下：

CONFIG_DM_GPIO=y

12.3.2. 结构体说明

（1）gpio_desc

前面也说过了，上层是通过 gpio_desc 来和 gpio core 进行交互的。

```
struct gpio_desc {
    struct udevice *dev;    /* Device, NULL for invalid GPIO */  //对应 bank 的 udevice 设备
    unsigned long flags;
    uint offset;        /* GPIO offset within the device */ // 该 GPIO 在 bank 内的偏移
};
```

offset 对应 bank 内的偏移，通过 offset+bank 的 gpio_base 可以得到具体的 GPIO 号（用于兼容老版本的方式）。

调用 uclass 的接口可以从 dts 直接获得对应 GPIO 的描述符。DM 框架下对 GPIO 的使用推荐使用这种方法。

（2）gpio_dev_priv

每个 gpio uclass 设备链表下的 udevice 都有一部分属于 uclass 的私有数据。udevice->uclass_priv。

都有一个这样的数据结构，提供给 gpio uclass 使用，让 gpio uclass 知道这个 udevice 中的 gpio 信息和情况。

```
struct gpio_dev_priv {
    const char *bank_name;  // udevice 的名字，也是 bank 名称
    unsigned gpio_count; // 这个 bank 中的 GPIO 数量
    unsigned gpio_base; // 每个 bank 的 gpio 都占有 gpio uclass 的一部分连续的 gpio 空间，gpio_base 则表示该 bank 的第一个 GPIO 号。
    char **name; //  阵列，每个 gpio 被 request 之后，都会有自己的 label 名称，存储在这里，可以防止 GPIO request 冲突
};
```

（3）标准 dm-gpio 的操作集合

```
struct dm_gpio_ops {
    int (*request)(struct udevice *dev, unsigned offset, const char *label);
    int (*free)(struct udevice *dev, unsigned offset);
    int (*direction_input)(struct udevice *dev, unsigned offset);
    int (*direction_output)(struct udevice *dev, unsigned offset,
            int value);
```

```
    int (*get_value)(struct udevice *dev, unsigned offset);
    int (*set_value)(struct udevice *dev, unsigned offset, int value);
    int (*get_function)(struct udevice *dev, unsigned offset);
        /* 这里具体看注释 */
    int (*xlate)(struct udevice *dev, struct gpio_desc *desc,
            struct fdtdec_phandle_args *args);
};
```

注意都是以 udevice 和 bank 内的偏移为参数，驱动中可以根据 udevice 获取到对应 bank 的信息以及私有数据，然后根据偏移就可以确认出到底是哪个 GPIO 了。

12.3.3. gpio uclass driver

DM 模型中 gpio 模块的 uclass。定义模块加载前和加载后的操作函数等等。

```
UCLASS_DRIVER(gpio) = {
    .id     = UCLASS_GPIO,
    .name       = "gpio",
    .flags      = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe = gpio_post_probe,
    .pre_remove = gpio_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct gpio_dev_priv),
};
```

（1）gpio_post_probe

gpio_post_probe 会在 gpio udevice probe 之后被调用，实现如下：

```
static int gpio_post_probe(struct udevice *dev)
{
    struct gpio_dev_priv *uc_priv = dev_get_uclass_priv(dev); // 获取刚 probe 的 udevice 的提供给 uclass 使用私有数据


    uc_priv->name = calloc(uc_priv->gpio_count, sizeof(char *)); // 为私有数据中的 GPIO label 分配空间
    if (!uc_priv->name)
```

return -ENOMEM;

return gpio_renumber(NULL); // 重新分配 GPIO 空间

}

12.3.4. gpio core

主要负责通过 gpio uclass 为上层提供接口，

（1）接口说明

其中既提供了兼容老版本的接口，也提供了 DM 框架下的接口

DM 框架下的接口

注意，外部通过 gpio_desc 来描述一个 GPIO，所以这些接口都是以 gpio_desc 作为参数

gpio_request_by_name

int gpio_request_by_name(struct udevice *dev, const char *list_name, int index, struct gpio_desc *desc, int flags)

通过对应的 udevice 找到其 dtsi 节点中属性名为 list_name 的 GPIO 属性并转化为 gpio_desc，并且 request。

gpio_request_by_name_nodev

int gpio_request_by_name_nodev(const void *blob, int node, const char *list_name, int index, struct gpio_desc *desc, int flags)

通过对应的 dtsi 节点中属性名为 list_name 的 GPIO 属性并转化为 gpio_desc，并且 request。

dm_gpio_request

int dm_gpio_request(struct gpio_desc *desc, const char *label)

申请 gpio_desc 描述的 GPIO

dm_gpio_get_value

int dm_gpio_get_value(const struct gpio_desc *desc)

获取 gpio_desc 描述的 GPIO 的值

dm_gpio_set_value

int dm_gpio_set_value(const struct gpio_desc *desc, int value)

设置 gpio_desc 描述的 GPIO 的值

dm_gpio_set_dir_flags

int dm_gpio_set_dir_flags(struct gpio_desc *desc, ulong flags)

设置 gpio_desc 描述的 GPIO 的输入输出方向，带标志

dm_gpio_set_dir

int dm_gpio_set_dir(struct gpio_desc *desc)

设置 gpio_desc 描述的 GPIO 的输入输出方向

dm_gpio_is_valid

static inline bool dm_gpio_is_valid(const struct gpio_desc *desc)

判断 gpio_desc 是否可用

老接口：

这些接口是为了兼容老版本的接口，注意，但是最终还是会调用 DM 框架下的接口

gpio_request

int gpio_request(unsigned gpio, const char *label)

申请一个 GPIO

gpio_direction_input

int gpio_direction_input(unsigned gpio)

设置某个 GPIO 为输入

gpio_direction_output

int gpio_direction_output(unsigned gpio, int value)

设置某个 GPIO 为输出

gpio_get_value

int gpio_get_value(unsigned gpio)

获取某个 GPIO 上的值

gpio_set_value

int gpio_set_value(unsigned gpio, int value)

设置 GPIO 的值

（2）举例：dm_gpio_set_value

```
int dm_gpio_set_value(const struct gpio_desc *desc, int value)
{
    int ret;

    ret = check_reserved(desc, "set_value"); // 判断这个 GPIO 是否已经被 request 了，必须是 request 之后才能被使用
    if (ret)
        return ret;

    if (desc->flags & GPIOD_ACTIVE_LOW)
        value = !value;
    gpio_get_ops(desc->dev)->set_value(desc->dev, desc->offset, value);
        // 先获取对应 udevice 的 driver，也就是对应 bank 的 drvier 的操作集中的 set_value 方法，
        // 然后以 udevice、bank 内的偏移，电平值为参数传入到 set_value 方法
        // 剩下的就是 driver 中的事了，driver 中需要能根据 udevice 和 bank 内偏移操作到对应的 GPIO 上去
    return 0;
}
```

（3）举例：gpio_set_value

```
int gpio_set_value(unsigned gpio, int value)
{
    struct gpio_desc desc;
    int ret;

    ret = gpio_to_device(gpio, &desc); // 将 GPIO 转化为对应的 gpio_desc 描述符
        // 前面我们也说了，所有 bank 都占用 uclass 的 gpio 空间的一部分、并且是连续的不冲突的，因此，uclass core 可以将对应的 gpio 转化为 gpio_desc 描述符
        // 具体实现自己看代码
    if (ret)
        return ret;
    return dm_gpio_set_value(&desc, value);
```

}

12.4. gpio driver

所有驱动都有自己的方式来解析 gpio_desc，这里我们以 s5pv210 为例：

不同 driver 可能差异较大。这里只作为一个参考。

drivers/gpios5p_gpio.c

12.4.1. 重点说明

所有 bank 都有对应的 udevice，因此在绑定的过程都需要创建 udevice 中关于这个 bank 的私有数据。

s5pv210 下这个私有数据是：

```
struct s5p_gpio_bank {
    unsigned int    con;
    unsigned int    dat;
    unsigned int    pull;
    unsigned int    drv;
    unsigned int    pdn_con;
    unsigned int    pdn_pull;
    unsigned char   res1[8];
};
```

其 size 刚好是一个 bank 的占有的寄存器的地址空间长度。并且意义也和寄存器中的对应。

另外，s5pv210 并不会为每个 bank 都创建一个匹配节点（注意是匹配节点而不是节点，节点还是会用的，匹配节点是指带有 compitable 属性的节点），而是将所有 bank 的节点都放在一个匹配节点下，而 driver 会在 bind 这个匹配节点的时候，去 probe 这个节点下面的 bank 节点。而匹配节点是不做 probe 操作的。

12.4.2. dtsi 中的定义

```
/{
    pinctrl: pinctrl@e0200000 {
        compatible = "samsung,s5pc110-pinctrl";  // 只有这个节点被匹配到，但是在这个节点 bind 的过程中，会去为子节点的 bank 创建对应 udevice 并且绑定到 gpio uclass 上，然后 probe 子节点
```

```
        reg = <0xe0200000 0x1000>;
        #address-cells = <1>;
        #size-cells = <1>;
        gpa0: gpa0 {
            gpio-controller;
            #gpio-cells = <2>;
        };

        gpa1: gpa1 {
            gpio-controller;
            #gpio-cells = <2>;
        };

        gpb: gpb {
            gpio-controller;
            #gpio-cells = <2>;
        };

        gpc0: gpc0 {
            gpio-controller;
            #gpio-cells = <2>;
        };
// 后面的 bank 这里不写了。。。
    };
};
```

注意这里的 bank 是有顺序的，必须和硬件上 bank 的顺序完全一致。

12.4.3. 定义一个 driver

具体成员的意义参考《[uboot] （番外篇）uboot 驱动模型》

```
U_BOOT_DRIVER(gpio_exynos) = {
    .name   = "gpio_exynos",
    .id = UCLASS_GPIO, // 定义所属 uclass 的 id
    .of_match = exynos_gpio_ids,
    .bind   = gpio_exynos_bind,
    .probe = gpio_exynos_probe,
    .priv_auto_alloc_size = sizeof(struct exynos_bank_info),
    .ops    = &gpio_exynos_ops,
};
```

12.4.4. 实现 exynos_gpio_ids

```
static const struct udevice_id exynos_gpio_ids[] = {
    { .compatible = "samsung,s5pc100-pinctrl" },
    { .compatible = "samsung,s5pc110-pinctrl" },
    { .compatible = "samsung,exynos4210-pinctrl" },
    { .compatible = "samsung,exynos4x12-pinctrl" },
    { .compatible = "samsung,exynos5250-pinctrl" },
    { .compatible = "samsung,exynos5420-pinctrl" },
    { }
};
```

12.4.5. 实现 gpio_exynos_bind

对应 udevice 绑定到 uclass 上所做的操作

```
static int gpio_exynos_bind(struct udevice *parent)
{
    struct exynos_gpio_platdata *plat = parent->platdata;
    struct s5p_gpio_bank *bank, *base;
    const void *blob = gd->fdt_blob;
    int node;
```

```
/* If this is a child device, there is nothing to do here */
if (plat)
    return 0;
    // 只有匹配的节点，也就是所有 bank 节点的父节点会做这个操作

base = (struct s5p_gpio_bank *)dev_get_addr(parent); // 获取 gpio 寄存器的基地址
for (node = fdt_first_subnode(blob, parent->of_offset), bank = base;
     node > 0;
     node = fdt_next_subnode(blob, node), bank++) { // 遍历所有子节点
    struct exynos_gpio_platdata *plat;
    struct udevice *dev;
    fdt_addr_t reg;
    int ret;

    if (!fdtdec_get_bool(blob, node, "gpio-controller")) // 判断是否有"gpio-controller"属性，也就是是否一个 bank
        continue;
    plat = calloc(1, sizeof(*plat));    // 分配平台数据
    if (!plat)
        return -ENOMEM;

    plat->bank_name = fdt_get_name(blob, node, NULL);  // 设置平台数据的部分值
    ret = device_bind(parent, parent->driver, // 绑定子设备，这里会为 bank 创建 udevice，并且绑定到 gpio uclass 中
            plat->bank_name, plat, -1, &dev);
    if (ret)
        return ret;

    dev->of_offset = node; // 设备 bank 对应的 udevice 的 node
```

```
        reg = dev_get_addr(dev);
        if (reg != FDT_ADDR_T_NONE)
            bank = (struct s5p_gpio_bank *)((ulong)base + reg); // 所有 bank 的物理空间是以此排列的，并且长度都是一样的。

        plat->bank = bank; // 设置 bank 的物理地址

        debug("dev at %p: %s\n", bank, plat->bank_name);
    }

    return 0;
}
```

12.4.6. 实现 gpio_exynos_probe

```
static int gpio_exynos_probe(struct udevice *dev)
{
    struct gpio_dev_priv *uc_priv = dev_get_uclass_priv(dev);
    struct exynos_bank_info *priv = dev->priv;
    struct exynos_gpio_platdata *plat = dev->platdata;

    /* Only child devices have ports */
    if (!plat)
        return 0;
        // 只有 bank 节点会做 probe 操作，而 bank 的父节点则不会做 probe 操作

    priv->bank = plat->bank; // 设置 bank 的私有数据

    uc_priv->gpio_count = GPIO_PER_BANK;  // 设置提供给 uclass 使用的私有数据 gpio_dev_priv
    uc_priv->bank_name = plat->bank_name; // 设置提供给 uclass 使用的私有数据 gpio_dev_priv

    return 0;
```

}

12.4.7. 实现操作集

用于给 gpio uclass 提供 udevice 对应的 bank 操作集，必须使用 dm_gpio_ops 来进行定义

```
static const struct dm_gpio_ops gpio_exynos_ops = {
    .direction_input    = exynos_gpio_direction_input,
    .direction_output   = exynos_gpio_direction_output,
    .get_value      = exynos_gpio_get_value,
    .set_value      = exynos_gpio_set_value,
    .get_function       = exynos_gpio_get_function,
    .xlate          = exynos_gpio_xlate,
};
```

在这些操作方法中，最终会直接操作到 GPIO 对应的寄存器上去。

以 exynos_gpio_set_value 为例：

```
static int exynos_gpio_set_value(struct udevice *dev, unsigned offset,
                int value)
// 以 bank 对应的 udevice，以及 bank 内的偏移为参数
{
    struct exynos_bank_info *state = dev_get_priv(dev); // 获取平台私有数据，state->bank 则是对应 bank 的寄存器地址
    s5p_gpio_set_value(state->bank, offset, value);
    return 0;
}


static void s5p_gpio_set_value(struct s5p_gpio_bank *bank, int gpio, int en)
{
    unsigned int value;


    value = readl(&bank->dat); // 读 bank 中的 data 寄存器
```

```
    value &= ~DAT_MASK(gpio);
    if (en)
        value |= DAT_SET(gpio);
    writel(value, &bank->dat); // 写入
}
```

注意，上述还提供了转化方式 exynos_gpio_xlate，负责解析 dtsi 的 GPIO 属性。所以，dtsi 的 GPIO 的属性的格式就是取决于这里是怎么解析的。
