Table of Contents

[Device Tree 详解 1](#_Toc133759727)

[1、DTS语法 2](#_Toc133759728)

[1.1、标准Property 3](#_Toc133759729)

[1.1.1、compatible 4](#_Toc133759730)

[1.1.2、model 4](#_Toc133759731)

[1.1.3、phandle 4](#_Toc133759732)

[1.1.4、#address-cells 、 #size-cells 6](#_Toc133759733)

[1.1.5、reg 7](#_Toc133759734)

[1.1.6、ranges 7](#_Toc133759735)

[1.1.7、interrupt property 8](#_Toc133759736)

[1.2、标准Node 12](#_Toc133759737)

[1.2.1、Root node 12](#_Toc133759738)

[1.2.2、/aliases node 13](#_Toc133759739)

[1.2.3、/memory node 13](#_Toc133759740)

[1.2.4、/chosen node 14](#_Toc133759741)

[1.2.5、/cpus node 14](#_Toc133759742)

[2、DTB 20](#_Toc133759743)

[2.1、DTB的编译 20](#_Toc133759744)

[2.2、DTB的文件结构 21](#_Toc133759745)

[2.3、Bootloader对DTB的传递 22](#_Toc133759746)

[3、Kernel解析 23](#_Toc133759747)

[3.1、DTB解析 23](#_Toc133759748)

[3.1.1 setup\_machine\_fdt() 23](#_Toc133759749)

[3.1.2 unflatten\_device\_tree() 24](#_Toc133759750)

[3.2、Device创建 31](#_Toc133759751)

[3.2.1 of\_platform\_populate() 31](#_Toc133759752)

[3.2.2 mt\_i2c\_driver 36](#_Toc133759753)

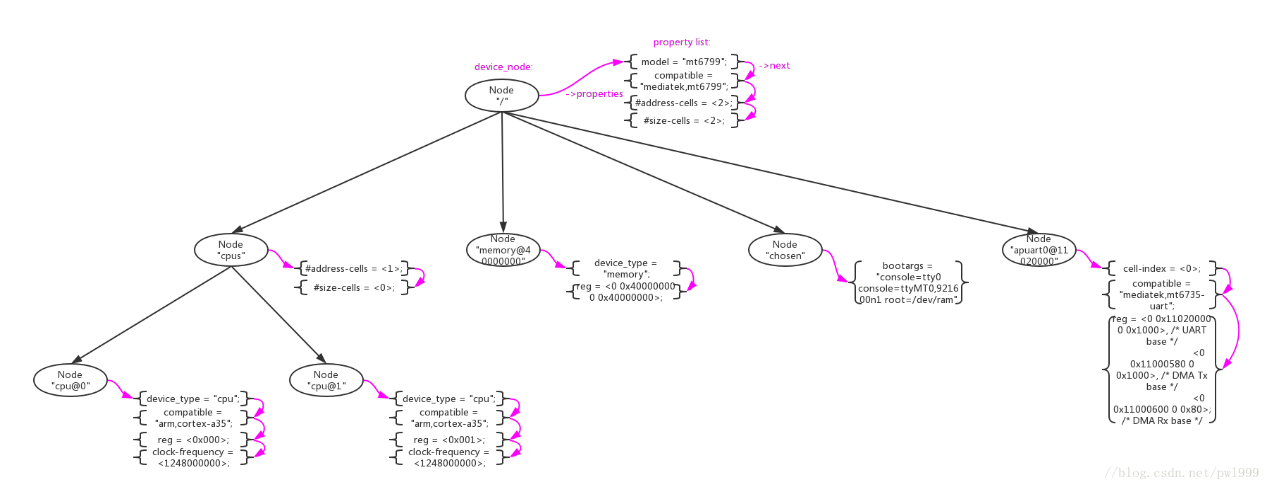
[3.2.3 mz\_mag\_driver 44](#_Toc133759754)

# Device Tree 详解

# 1、DTS语法

对于DeviceTree的来历和用处大部分人都已经非常了解了，DeviceTree发源于PowerPC架构，为了消除代码中冗余的各种device注册代码而产生的，现在已经成为了linux的通用机制。

DeviceTree的结构非常简单，由两种元素组成：Node(节点)、Property(属性)。下图是一个真实的简单的DeviceTree树形结构图。



* Node节点。在DTS中使用一对花括号"node-name{}"来定义;
* Property属性。在Node中使用"property-name=value"字符串来定义；

/ {

model = "mt6799";

compatible = "mediatek,mt6799";

interrupt-parent = <&gic>;

#address-cells = <2>;

#size-cells = <2>;

/\* chosen \*/

chosen {

bootargs = "console=tty0 console=ttyMT0,921600n1 root=/dev/ram";

};

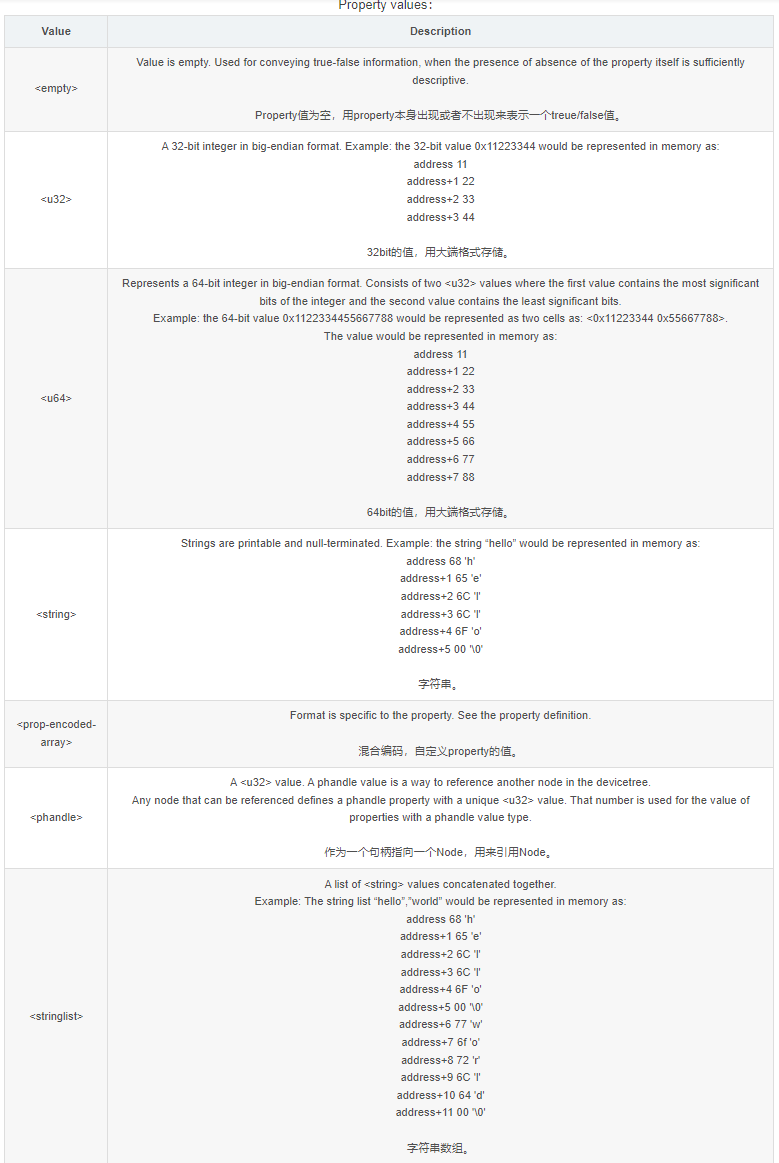
}

上述例子中定义了一个根节点"/"和一个子节点“chosen”，其他的字符串“model = “mt6799”;”、“compatible = “mediatek,mt6799”;”都是property。

Node、Property的名字和值都是可以自定义的，没有太大限制。但是DeviceTree的标准还是预定义了一些标准的Node和Property，在标准Node和Property之间还定义了一些约束规则。关于这些描述在 The DeviceTree Specification官方spec中有详细描述。这里为了方便大家，还是重复做一些搬运。

## 1.1、标准Property

Property的格式为"property-name=value"，其中value的取值类型如下：



### 1.1.1、compatible

“compatible"属性通常用来device和driver的适配，推荐的格式为"manufacturer,model”。

Graphical user interface, text, application, email

Description automatically generated

### 1.1.2、model

"model"属性只是简单的表示型号，root节点用其来传递值给machine\_desc\_str。

Graphical user interface, text, application

Description automatically generated

### 1.1.3、phandle

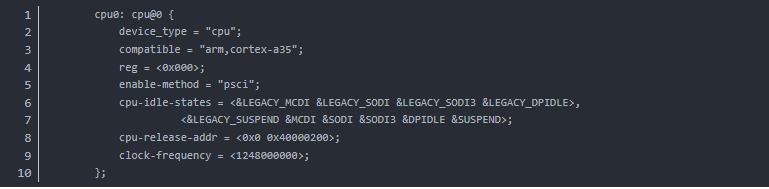
"phandle"属性通用一个唯一的id来标识一个Node，在property可以使用这个id来引用Node。

Graphical user interface, text, application, email

Description automatically generated

在DeviceTree中通过另一种方式进行phandle的定义和引用更加常见：

定义一个“label：”来引用Node，在编译是系统会自动为node生成一个phandle属性。“cpu0"是一个label，用来引用node"cpu@0”：



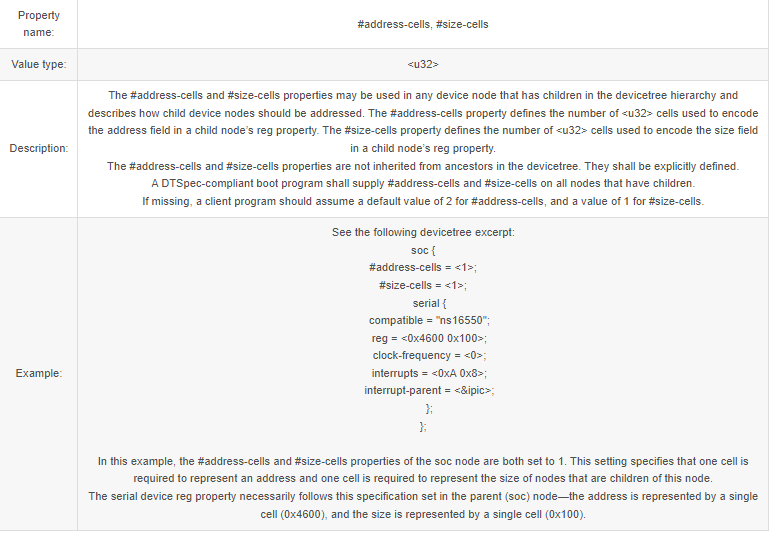
使用"&“来引用“label”，即是引用phandle。property"cpu"通过”&cpu0"来对node"cpu@0"：

A screenshot of a computer

Description automatically generated with medium confidence

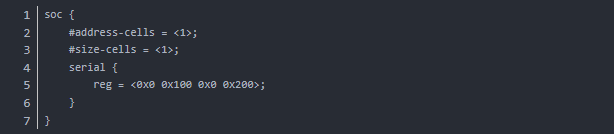
### 1.1.4、#address-cells 、 #size-cells

"#address-cells, #size-cells"属性用来定义当前node的子node中"reg"属性的解析格式。



举例说明：

1、如果node"soc"中"#address-cells=<1>"、"#size-cells=<1>"，那么子node"serial"中"reg"属性的解析为“addr1 = 0x0, size1 = 0x100, addr2 = 0x0, size2 = 0x200”：



2、如果node"soc"中"#address-cells=<2>"、"#size-cells=<2>"，那么子node"serial"中"reg"属性的解析为“addr1 = 0x100, size1 = 0x200”：

Text

Description automatically generated with medium confidence3、如果node"soc"中"#address-cells=<2>"、"#size-cells=<0>"，那么子node"serial"中"reg"属性的解析为“addr1 = 0x100, addr2 = 0x200”：

Text

Description automatically generated

### 1.1.5、reg

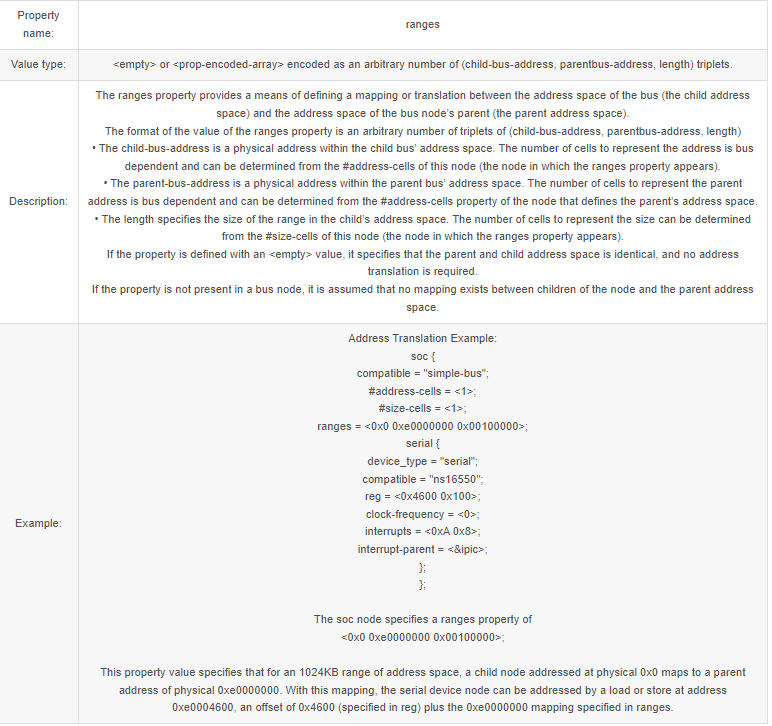
“reg"属性解析出"address,length"数字，解析格式依据父节点的”#address-cells、#size-cells"定义。

Graphical user interface, text, application, email

Description automatically generated

### 1.1.6、ranges

"ranges"属性用来做当前node和父node之间的地址映射，格式为(child-bus-address, parentbus-address, length)。其中child-bus-address的解析长度受当前node的#address-cells属性控制，parentbus-address的解析长度受父node的#address-cells属性控制length的解析长度受当前node的#size-cells属性控制。



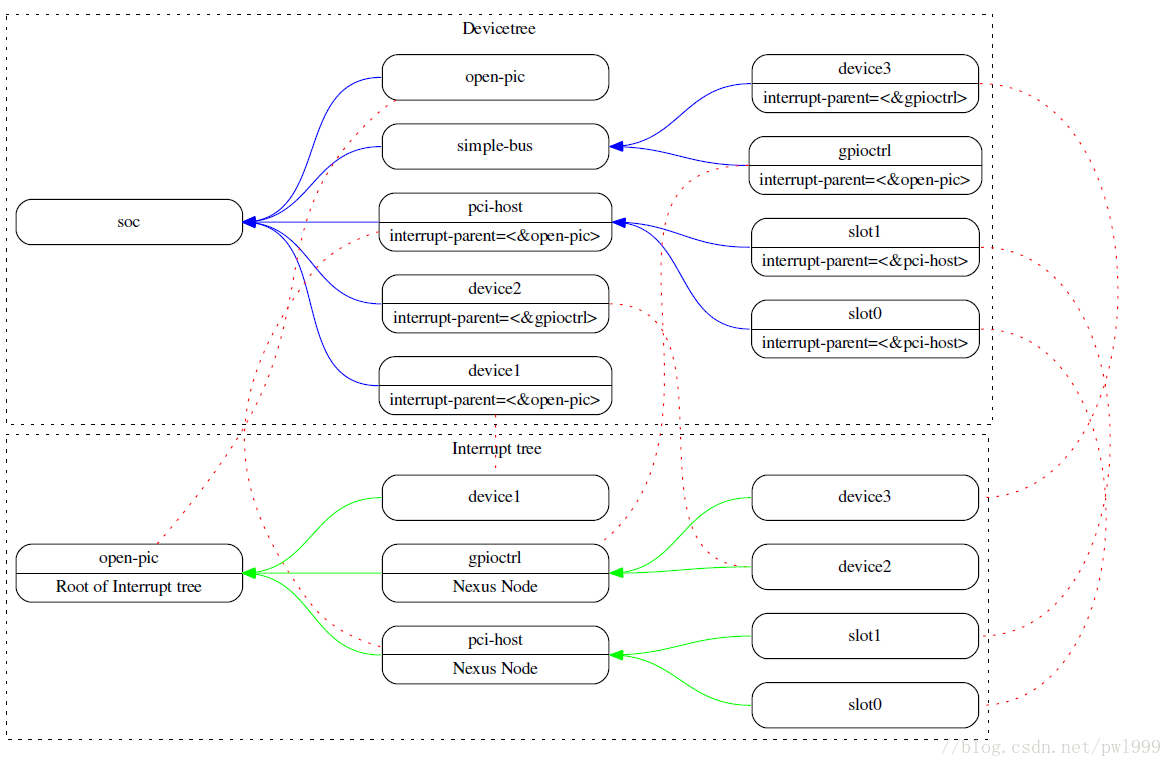
### 1.1.7、interrupt property

和中断相关的node可以分成3种：

“Interrupt Generating Devices”，中断发生设备，这种设备可以发生中断。

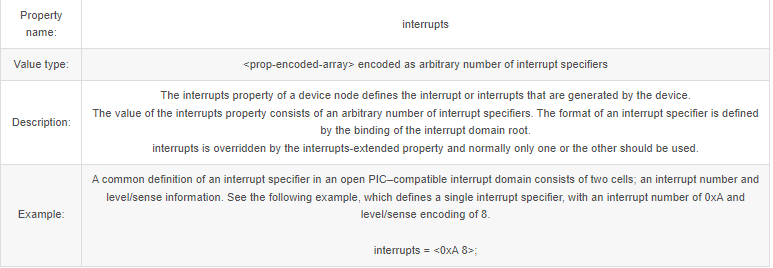
“Interrupt Controllers”，中断控制器，处理中断。

“Interrupt Nexus”，中断联结，路由中断给中断控制器。



#### 1.1.7.1、Interrupt Generating Devices Property

"interrupts"属性用来定义设备的中断解析，根据其"interrupt-parent"node中定义的“#interrupt-cells”来解析。比如#interrupt-cells=2，那根据2个cells为单位来解析"interrupts"属性。

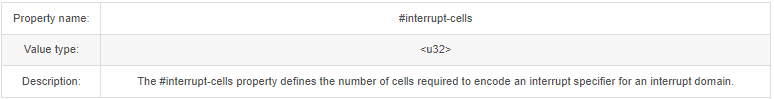


"interrupt-parent"属性用来制定当前设备的Interrupt Controllers/Interrupt Nexus，phandle指向对应的node。

Property name: interrupt-parent Value type: <phandle> Description: Because the hierarchy of the nodes in the interrupt tree might not match the devicetree, the interrupt-parent property is available to make the definition of an interrupt parent explicit. The value is the phandle to the interrupt parent. If this property is missing from a device, its interrupt parent is assumed to be its devicetree parent.

#### 1.1.7.2、Interrupt Controllers Property

"#interrupt-cells"属性用来规定连接到该中断控制器上的设备的"interrupts"属性的解析长度。



"interrupt-controller"属性用来声明当前node为中断控制器。



#### 1.1.7.3、Interrupt Nexus Property

"#interrupt-cells"属性用来规定连接到该中断控制器上的设备的"interrupts"属性的解析长度。

Graphical user interface, text, application

Description automatically generated

"interrupt-map"属性用来描述interrupt nexus设备对中断的路由。解析格式为5元素序列“child unit address, child interrupt specifier, interrupt-parent, parent unit address, parent interrupt specifier”。

其中：

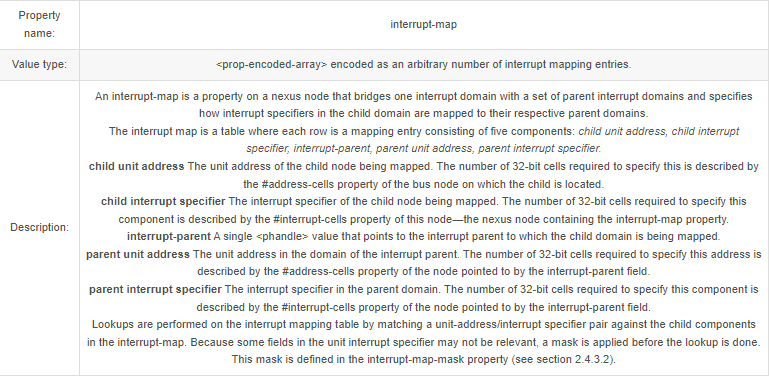
“child unit address”的cells长度由子节点的“#address-cells”指定；

“child interrupt specifier”的cells长度由子节点的“#interrupt-cells”指定；

“interrupt-parent”phandle指向interrupt controller的引用；

“parent unit address”的cells长度由父节点的“#address-cells”指定；

“parent interrupt specifier”的cells长度由父节点的“#interrupt-cells”指定；



举例：

soc {

compatible = "simple-bus";

#address-cells = <1>;

#size-cells = <1>;

open-pic {

clock-frequency = <0>;

interrupt-controller;

#address-cells = <0>;

#interrupt-cells = <2>;

};

pci {

#interrupt-cells = <1>;

#size-cells = <2>;

#address-cells = <3>;

interrupt-map-mask = <0xf800 0 0 7>;

interrupt-map = <

/\* IDSEL 0x11 - PCI slot 1 \*/

0x8800 0 0 1 &open-pic 2 1 /\* INTA \*/

0x8800 0 0 2 &open-pic 3 1 /\* INTB \*/

0x8800 0 0 3 &open-pic 4 1 /\* INTC \*/

0x8800 0 0 4 &open-pic 1 1 /\* INTD \*/

/\* IDSEL 0x12 - PCI slot 2 \*/

0x9000 0 0 1 &open-pic 3 1 /\* INTA \*/

0x9000 0 0 2 &open-pic 4 1 /\* INTB \*/

0x9000 0 0 3 &open-pic 1 1 /\* INTC \*/

0x9000 0 0 4 &open-pic 2 1 /\* INTD \*/

>;

};

};

• For example, the first row of the interrupt-map table specifies the mapping for INTA of slot . The components of that row are shown here

child unit address: 0x8800 0 0

child interrupt specifier: 1

interrupt parent: &open-pic

parent unit address: (empty because #address-cells = <0> in the open-pic node)

parent interrupt specifier: 2 1

## 1.2、标准Node

Node Name常常由两部分组成“node-name@unit-address”，主要是为了防止Node Name重复冲突：

"node-name"是node的名字；

"unit-address"是node中“reg”属性描述的开始地址；

例如："msdc@11240000"中node-name=“msdc”，unit-address=“11240000”。

/ {

model = "mt6799";

compatible = "mediatek,mt6799";

interrupt-parent = <&gic>;

#address-cells = <2>;

#size-cells = <2>;

msdc0:msdc@11240000 {

compatible = "mediatek,msdc";

reg = <0x0 0x11240000 0x0 0x10000

0x0 0x10000e84 0x0 0x2>; /\* FPGA PWR\_GPIO, PWR\_GPIO\_EO \*/

interrupts = <GIC\_SPI 79 IRQ\_TYPE\_LEVEL\_LOW>;

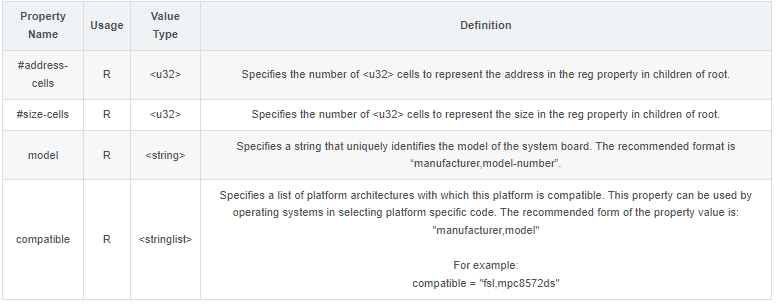
};

下面主要介绍一下一些预先定义的标准Node。

### 1.2.1、Root node

每个DeviceTree只有一个根节点。根节点需要有以下必备属性：

Root Node Properties



### 1.2.2、/aliases node

用来给一些绝对路径定义别名：

aliases {

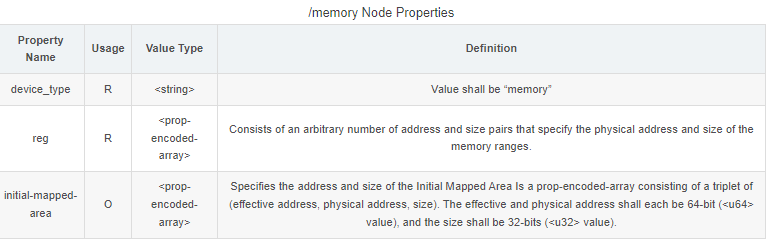
serial0 = "/simple-bus@fe000000/serial@llc500";

ethernet0 = "/simple-bus@fe000000/ethernet@31c000";

};

### 1.2.3、/memory node

用来传递内存布局：



举例：

• RAM: starting address 0x0, length 0x80000000 (2GB)

• RAM: starting address 0x100000000, length 0x100000000 (4GB)

\ {

#address-cells = <2>;

#size-cells = <2>;

memory@0 {

device\_type = "memory";

reg = <0x000000000 0x00000000 0x00000000 0x80000000

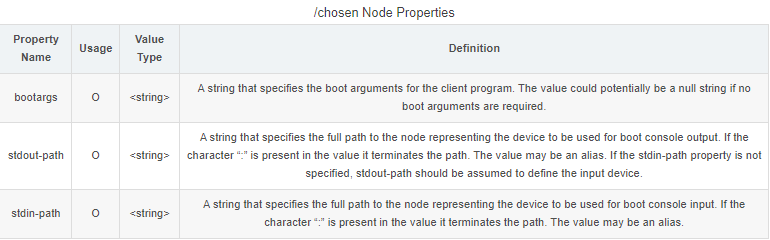
0x000000001 0x00000000 0x00000001 0x00000000>;

};

}

### 1.2.4、/chosen node

其中“bootargs”属性用来传递cmdline参数，“stdout-path”属性用来指定标准输出设备，“stdin-path”属性用来指定标准输入设备。

举例：

/\* chosen \*/

chosen {

bootargs = "console=tty0 console=ttyMT0,921600n1 root=/dev/ram";

};

### 1.2.5、/cpus node

/cpus节点也是必须的，下面举个具体例子：

cpus {

#address-cells = <1>;

#size-cells = <0>;

cpu0: cpu@0 {

device\_type = "cpu";

compatible = "arm,cortex-a35";

reg = <0x000>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1248000000>;

};

cpu1: cpu@001 {

device\_type = "cpu";

compatible = "arm,cortex-a35";

reg = <0x001>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1248000000>;

};

cpu2: cpu@002 {

device\_type = "cpu";

compatible = "arm,cortex-a35";

reg = <0x002>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1248000000>;

};

cpu3: cpu@003 {

device\_type = "cpu";

compatible = "arm,cortex-a35";

reg = <0x003>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1248000000>;

};

cpu4: cpu@100 {

device\_type = "cpu";

compatible = "arm,cortex-a53";

reg = <0x100>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1378000000>;

};

cpu5: cpu@101 {

device\_type = "cpu";

compatible = "arm,cortex-a53";

reg = <0x101>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1378000000>;

};

cpu6: cpu@102 {

device\_type = "cpu";

compatible = "arm,cortex-a53";

reg = <0x102>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1378000000>;

};

cpu7: cpu@103 {

device\_type = "cpu";

compatible = "arm,cortex-a53";

reg = <0x103>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1378000000>;

};

cpu8: cpu@200 {

device\_type = "cpu";

compatible = "arm,cortex-a73";

reg = <0x200>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1638000000>;

};

cpu9: cpu@201 {

device\_type = "cpu";

compatible = "arm,cortex-a73";

reg = <0x201>;

enable-method = "psci";

cpu-idle-states = <&LEGACY\_MCDI &LEGACY\_SODI &LEGACY\_SODI3 &LEGACY\_DPIDLE>,

<&LEGACY\_SUSPEND &MCDI &SODI &SODI3 &DPIDLE &SUSPEND>;

cpu-release-addr = <0x0 0x40000200>;

clock-frequency = <1638000000>;

};

cpu-map {

cluster0 {

core0 {

cpu = <&cpu0>;

};

core1 {

cpu = <&cpu1>;

};

core2 {

cpu = <&cpu2>;

};

core3 {

cpu = <&cpu3>;

};

};

cluster1 {

core0 {

cpu = <&cpu4>;

};

core1 {

cpu = <&cpu5>;

};

core2 {

cpu = <&cpu6>;

};

core3 {

cpu = <&cpu7>;

};

};

cluster2 {

core0 {

cpu = <&cpu8>;

};

core1 {

cpu = <&cpu9>;

};

};

};

idle-states {

entry-method = "arm,psci";

LEGACY\_MCDI: legacy-mcdi {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0000001>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

LEGACY\_SODI: legacy-sodi {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0000002>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

LEGACY\_SODI3: legacy-sodi3 {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0000003>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

LEGACY\_DPIDLE: legacy-dpidle {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0000004>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

LEGACY\_SUSPEND: legacy-suspend {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0000005>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

MCDI: mcdi {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x0010001>;

entry-latency-us = <600>;

exit-latency-us = <600>;

min-residency-us = <1200>;

};

SODI: sodi {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x1010002>;

entry-latency-us = <800>;

exit-latency-us = <1000>;

min-residency-us = <2000>;

};

SODI3: sodi3 {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x1010003>;

entry-latency-us = <800>;

exit-latency-us = <1000>;

min-residency-us = <2000>;

};

DPIDLE: dpidle {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x1010004>;

entry-latency-us = <800>;

exit-latency-us = <1000>;

min-residency-us = <2000>;

};

SUSPEND: suspend {

compatible = "arm,idle-state";

arm,psci-suspend-param = <0x1010005>;

entry-latency-us = <800>;

exit-latency-us = <1000>;

min-residency-us = <2000>;

};

};

};

# 2、DTB

## 2.1、DTB的编译

DTB(Devicetree Blob)是DTS的二进制文件格式，Kernel使用DTC工具将DTS源文件编译成DTB，bootloader再将DTB文件传递给Kernel解析。

不遵守标准书写的DTS文件在编译的时候会报错。

## 2.2、DTB的文件结构

Table

Description automatically generated

DTB文件的结构如上图所示，主要在3部分：

struct ftd\_header。文件头结构；

structure block。存放含Node和Property的Value；

strings block。存放Property的Name；把Property Name单独分为一个区域的原因是，有很多Property Name是重复的，单独一个区域可以使用指针引用，节约空间。

dtb中的fdt\_header的数据结构：

struct fdt\_header {

uint32\_t magic;

uint32\_t totalsize;

uint32\_t off\_dt\_struct;

uint32\_t off\_dt\_strings;

uint32\_t off\_mem\_rsvmap;

uint32\_t version;

uint32\_t last\_comp\_version;

uint32\_t boot\_cpuid\_phys;

uint32\_t size\_dt\_strings;

uint32\_t size\_dt\_struct;

};

dtb中node header的数据结构：

struct fdt\_node\_header {

fdt32\_t tag;

char name[0]; // node name 存放在structure block

};

dtb中property header的数据结构：

struct fdt\_property {

fdt32\_t tag;

fdt32\_t len;

fdt32\_t nameoff; // perperty name存放在strings block

char data[0];

};

整个文件使用5种token来分割出node和property：

FDT\_BEGIN\_NODE (0x00000001)

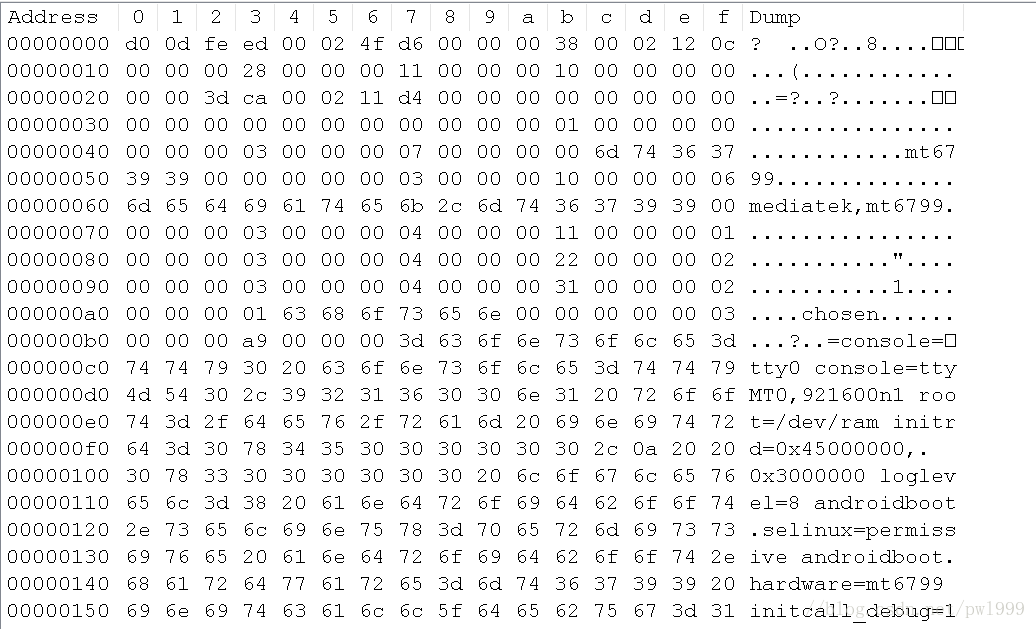
FDT\_END\_NODE (0x00000002)

FDT\_PROP (0x00000003)

FDT\_NOP (0x00000004)

FDT\_END (0x00000009)

可以使用hex编辑器来查看DTB文件的结构：



## 2.3、Bootloader对DTB的传递

没有仔细去看

# 3、Kernel解析

## 3.1、DTB解析

### 3.1.1 setup\_machine\_fdt()

直接在dtb中解析根节点的一些属性和子节点给系统早期使用。

解析"/"节点的model"属性给machine\_desc赋值；

解析"/chosen"node中的"bootargs"属性给boot\_command\_line；

解析"/“节点的”#size-cells"、"#address-cells"属性；

解析"/memory"node中的"reg"属性，并将memory区域加入到系统；

start\_kernel() -> setup\_arch() -> setup\_machine\_fdt():

↓

static void \_\_init setup\_machine\_fdt(phys\_addr\_t dt\_phys)

{

/\* (1) 映射dtb内存，到使之可以访问 \*/

void \*dt\_virt = fixmap\_remap\_fdt(dt\_phys);

/\* (2) 早期扫描device tree中的一些node和property \*/

if (!dt\_virt || !early\_init\_dt\_scan(dt\_virt)) {

pr\_crit("\n"

"Error: invalid device tree blob at physical address %pa (virtual address 0x%p)\n"

"The dtb must be 8-byte aligned and must not exceed 2 MB in size\n"

"\nPlease check your bootloader.",

&dt\_phys, dt\_virt);

while (true)

cpu\_relax();

}

/\* (3) 使用device tree中root node的"model/compatible"属性给machine\_desc赋值 \*/

machine\_desc\_set(of\_flat\_dt\_get\_machine\_name());

dump\_stack\_set\_arch\_desc("%s (DT)", of\_flat\_dt\_get\_machine\_name());

}

↓

bool \_\_init early\_init\_dt\_scan(void \*params)

{

bool status;

/\* (2.1)校验dtb数据 \*/

status = early\_init\_dt\_verify(params);

if (!status)

return false;

/\* (2.2) \*/

early\_init\_dt\_scan\_nodes();

return true;

}

↓

void \_\_init early\_init\_dt\_scan\_nodes(void)

{

/\* (2.2.1) 解析"/chosen"node中的"bootargs"属性 \*/

/\* Retrieve various information from the /chosen node \*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_chosen, boot\_command\_line);

/\* (2.2.2) 解析"/"node中的"#size-cells"、"#address-cells"属性 \*/

/\* Initialize {size,address}-cells info \*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_root, NULL);

/\* (2.2.3) 解析"/memory"node中的"reg"属性，并将memory区域加入到系统 \*/

/\* Setup memory, calling early\_init\_dt\_add\_memory\_arch \*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_memory, NULL);

}

↓

early\_init\_dt\_scan\_memory() -> early\_init\_dt\_add\_memory\_arch() -> memblock\_add()

### 3.1.2 unflatten\_device\_tree()

将DTB完全解析为内核使用的的device\_node、property结构：

start\_kernel() -> setup\_arch() -> unflatten\_device\_tree():

↓

void \_\_init unflatten\_device\_tree(void)

{

/\* (1) 解析dtb数据到kernel中 \*/

\_\_unflatten\_device\_tree(initial\_boot\_params, &of\_root,

early\_init\_dt\_alloc\_memory\_arch);

/\* (2) 扫描"/aliases"、"/chosen"节点来进行一些预制值的配置 \*/

/\* Get pointer to "/chosen" and "/aliases" nodes for use everywhere \*/

of\_alias\_scan(early\_init\_dt\_alloc\_memory\_arch);

}

↓

static void \_\_unflatten\_device\_tree(const void \*blob,

struct device\_node \*\*mynodes,

void \* (\*dt\_alloc)(u64 size, u64 align))

{

unsigned long size;

int start;

void \*mem;

pr\_debug(" -> unflatten\_device\_tree()\n");

if (!blob) {

pr\_debug("No device tree pointer\n");

return;

}

pr\_debug("Unflattening device tree:\n");

pr\_debug("magic: %08x\n", fdt\_magic(blob));

pr\_debug("size: %08x\n", fdt\_totalsize(blob));

pr\_debug("version: %08x\n", fdt\_version(blob));

if (fdt\_check\_header(blob)) {

pr\_err("Invalid device tree blob header\n");

return;

}

/\* (1.1) 第一遍扫描，计算dtb解析需要的内存空间 \*/

/\* First pass, scan for size \*/

start = 0;

size = (unsigned long)unflatten\_dt\_node(blob, NULL, &start, NULL, NULL, 0, true);

size = ALIGN(size, 4);

pr\_debug(" size is %lx, allocating...\n", size);

/\* (1.2) 分配所需内存 \*/

/\* Allocate memory for the expanded device tree \*/

mem = dt\_alloc(size + 4, \_\_alignof\_\_(struct device\_node));

memset(mem, 0, size);

\*(\_\_be32 \*)(mem + size) = cpu\_to\_be32(0xdeadbeef);

pr\_debug(" unflattening %p...\n", mem);

/\* (1.3)第二遍扫描，在分配的内存中创建device\_node、property树形结构来存储dtb的解析 \*/

/\* Second pass, do actual unflattening \*/

start = 0;

unflatten\_dt\_node(blob, mem, &start, NULL, mynodes, 0, false);

if (be32\_to\_cpup(mem + size) != 0xdeadbeef)

pr\_warning("End of tree marker overwritten: %08x\n",

be32\_to\_cpup(mem + size));

pr\_debug(" <- unflatten\_device\_tree()\n");

}

↓

static void \* unflatten\_dt\_node(const void \*blob,

void \*mem,

int \*poffset,

struct device\_node \*dad,

struct device\_node \*\*nodepp,

unsigned long fpsize,

bool dryrun)

{

const \_\_be32 \*p;

struct device\_node \*np;

struct property \*pp, \*\*prev\_pp = NULL;

const char \*pathp;

unsigned int l, allocl;

static int depth;

int old\_depth;

int offset;

int has\_name = 0;

int new\_format = 0;

/\* (1.1.1) 解析node，解析node中的name值 \*/

pathp = fdt\_get\_name(blob, \*poffset, &l);

if (!pathp)

return mem;

allocl = ++l; /\* l 为当前路径的长度 \*/

/\* version 0x10 has a more compact unit name here instead of the full

\* path. we accumulate the full path size using "fpsize", we'll rebuild

\* it later. We detect this because the first character of the name is

\* not '/'.

\*/

if ((\*pathp) != '/') {

new\_format = 1;

if (fpsize == 0) {

/\* root node: special case. fpsize accounts for path

\* plus terminating zero. root node only has '/', so

\* fpsize should be 2, but we want to avoid the first

\* level nodes to have two '/' so we use fpsize 1 here

\*/

fpsize = 1;

allocl = 2;

l = 1;

pathp = "";

} else {

/\* account for '/' and path size minus terminal 0

\* already in 'l'

\*/

fpsize += l; /\* 当前full path的长度 = 上一次full path的长度 + 当前node nam的长度 l \*/

allocl = fpsize;

}

}

/\* 计算解析当前node节点需要的内存大小 = device\_node + full path \*/

np = unflatten\_dt\_alloc(&mem, sizeof(struct device\_node) + allocl,

\_\_alignof\_\_(struct device\_node));

/\* dryrun = true，只进行长度计算

dryrun = fasle，进行实际的赋值 \*/

if (!dryrun) {

char \*fn;

of\_node\_init(np);

np->full\_name = fn = ((char \*)np) + sizeof(\*np); /\* device\_node->full\_name，指向device\_node结构体的结尾 \*/

if (new\_format) {

/\* rebuild full path for new format \*/

if (dad && dad->parent) {

/\* 先拷入上次的full name \*/

strcpy(fn, dad->full\_name);

#ifdef DEBUG

if ((strlen(fn) + l + 1) != allocl) {

pr\_debug("%s: p: %d, l: %d, a: %d\n",

pathp, (int)strlen(fn),

l, allocl);

}

#endif

fn += strlen(fn);

}

/\* 再加上 '/' \*/

\*(fn++) = '/';

}

/\* 最后加上当前node的name \*/

memcpy(fn, pathp, l);

prev\_pp = &np->properties;

/\* node和node之间树形结构的创建 \*/

if (dad != NULL) {

np->parent = dad;

np->sibling = dad->child;

dad->child = np;

}

}

/\* (1.1.2) 解析node中的property \*/

/\* process properties \*/

for (offset = fdt\_first\_property\_offset(blob, \*poffset);

(offset >= 0);

(offset = fdt\_next\_property\_offset(blob, offset))) {

const char \*pname;

u32 sz;

/\* 解析一个property：

p：property中的data

pname：property的name指针，实际存储位置在dt\_strings区域中

sz：property data的长度

\*/

if (!(p = fdt\_getprop\_by\_offset(blob, offset, &pname, &sz))) {

offset = -FDT\_ERR\_INTERNAL;

break;

}

if (pname == NULL) {

pr\_info("Can't find property name in list !\n");

break;

}

if (strcmp(pname, "name") == 0)

has\_name = 1;

/\* 计算解析当前property的内存大小 = property \*/

pp = unflatten\_dt\_alloc(&mem, sizeof(struct property),

\_\_alignof\_\_(struct property));

/\* 实际的property赋值 \*/

if (!dryrun) {

/\* We accept flattened tree phandles either in

\* ePAPR-style "phandle" properties, or the

\* legacy "linux,phandle" properties. If both

\* appear and have different values, things

\* will get weird. Don't do that. \*/

/\* 如果property为"phandle"，设置父node的device\_node->phandle为当前属性的值 \*/

if ((strcmp(pname, "phandle") == 0) ||

(strcmp(pname, "linux,phandle") == 0)) {

if (np->phandle == 0)

np->phandle = be32\_to\_cpup(p);

}

/\* And we process the "ibm,phandle" property

\* used in pSeries dynamic device tree

\* stuff \*/

if (strcmp(pname, "ibm,phandle") == 0)

np->phandle = be32\_to\_cpup(p);

/\* 给property的其他字段赋值：（DTB的空间没有释放，被property成员指针引用）

property->name：指针指向dtb strings blcok区域中的属性name

property->length：属性data的长度

property->value：指针指向dtb stucture block区域中的属性data

\*/

pp->name = (char \*)pname;

pp->length = sz;

pp->value = (\_\_be32 \*)p;

\*prev\_pp = pp;

prev\_pp = &pp->next;

}

}

/\* with version 0x10 we may not have the name property, recreate

\* it here from the unit name if absent

\*/

if (!has\_name) {

const char \*p1 = pathp, \*ps = pathp, \*pa = NULL;

int sz;

while (\*p1) {

if ((\*p1) == '@')

pa = p1;

if ((\*p1) == '/')

ps = p1 + 1;

p1++;

}

if (pa < ps)

pa = p1;

sz = (pa - ps) + 1;

pp = unflatten\_dt\_alloc(&mem, sizeof(struct property) + sz,

\_\_alignof\_\_(struct property));

if (!dryrun) {

pp->name = "name";

pp->length = sz;

pp->value = pp + 1;

\*prev\_pp = pp;

prev\_pp = &pp->next;

memcpy(pp->value, ps, sz - 1);

((char \*)pp->value)[sz - 1] = 0;

pr\_debug("fixed up name for %s -> %s\n", pathp,

(char \*)pp->value);

}

}

/\* 根据"name"、 "device\_type"属性，来给device\_node结构中的name、type成员赋值 \*/

if (!dryrun) {

\*prev\_pp = NULL;

np->name = of\_get\_property(np, "name", NULL);

np->type = of\_get\_property(np, "device\_type", NULL);

if (!np->name)

np->name = "<NULL>";

if (!np->type)

np->type = "<NULL>";

}

/\* (1.1.3)如果还有子node的存在，递归解析 \*/

old\_depth = depth;

\*poffset = fdt\_next\_node(blob, \*poffset, &depth);

if (depth < 0)

depth = 0;

while (\*poffset > 0 && depth > old\_depth)

mem = unflatten\_dt\_node(blob, mem, poffset, np, NULL,

fpsize, dryrun);

if (\*poffset < 0 && \*poffset != -FDT\_ERR\_NOTFOUND)

pr\_err("unflatten: error %d processing FDT\n", \*poffset);

/\*

\* Reverse the child list. Some drivers assumes node order matches .dts

\* node order

\*/

if (!dryrun && np->child) {

struct device\_node \*child = np->child;

np->child = NULL;

while (child) {

struct device\_node \*next = child->sibling;

child->sibling = np->child;

np->child = child;

child = next;

}

}

if (nodepp)

\*nodepp = np;

return mem;

}

## 3.2、Device创建

### 3.2.1 of\_platform\_populate()

首先root节点下的第1级子节点创建成platform device。

对root节点下的第1级子节点，如果有"compatible"属性创建对应platform device；

如果"compatible"属性等于of\_default\_bus\_match\_table(“simple-bus”/“simple-mfd”/“arm,amba-bus”)中任意一种，继续对其子节点进行platform device创建。

start\_kernel() -> ... ->do\_initcalls() -> arm64\_device\_init():

↓

const struct of\_device\_id of\_default\_bus\_match\_table[] = {

{ .compatible = "simple-bus", },

{ .compatible = "simple-mfd", },

#ifdef CONFIG\_ARM\_AMBA

{ .compatible = "arm,amba-bus", },

#endif /\* CONFIG\_ARM\_AMBA \*/

{} /\* Empty terminated list \*/

};

static int \_\_init arm64\_device\_init(void)

{

if (of\_have\_populated\_dt()) {

of\_iommu\_init();

of\_platform\_populate(NULL, of\_default\_bus\_match\_table,

NULL, NULL);

} else if (acpi\_disabled) {

pr\_crit("Device tree not populated\n");

}

return 0;

}

↓

int of\_platform\_populate(struct device\_node \*root,

const struct of\_device\_id \*matches,

const struct of\_dev\_auxdata \*lookup,

struct device \*parent)

{

struct device\_node \*child;

int rc = 0;

/\* (1) 获取dts中的root node \*/

root = root ? of\_node\_get(root) : of\_find\_node\_by\_path("/");

if (!root)

return -EINVAL;

/\* (2) 对root node的child node进行platform device创建 \*/

for\_each\_child\_of\_node(root, child) {

rc = of\_platform\_bus\_create(child, matches, lookup, parent, true);

if (rc) {

of\_node\_put(child);

break;

}

}

of\_node\_set\_flag(root, OF\_POPULATED\_BUS);

of\_node\_put(root);

return rc;

}

↓

static int of\_platform\_bus\_create(struct device\_node \*bus,

const struct of\_device\_id \*matches,

const struct of\_dev\_auxdata \*lookup,

struct device \*parent, bool strict)

{

const struct of\_dev\_auxdata \*auxdata;

struct device\_node \*child;

struct platform\_device \*dev;

const char \*bus\_id = NULL;

void \*platform\_data = NULL;

int rc = 0;

/\* (2.1) 确保要创建为platform device的node，拥有"compatible"属性 \*/

/\* Make sure it has a compatible property \*/

if (strict && (!of\_get\_property(bus, "compatible", NULL))) {

pr\_debug("%s() - skipping %s, no compatible prop\n",

\_\_func\_\_, bus->full\_name);

return 0;

}

auxdata = of\_dev\_lookup(lookup, bus);

if (auxdata) {

bus\_id = auxdata->name;

platform\_data = auxdata->platform\_data;

}

if (of\_device\_is\_compatible(bus, "arm,primecell")) {

/\*

\* Don't return an error here to keep compatibility with older

\* device tree files.

\*/

of\_amba\_device\_create(bus, bus\_id, platform\_data, parent);

return 0;

}

/\* (2.2) 对当前的node创建platform device \*/

dev = of\_platform\_device\_create\_pdata(bus, bus\_id, platform\_data, parent);

/\* (2.3) 根据of\_default\_bus\_match\_table，如果node中含有以下属性：

compatible = "simple-bus"

compatible = "simple-mfd"

compatible = "arm,amba-bus"

则继续对node的子node进行platform device创建

\*/

if (!dev || !of\_match\_node(matches, bus))

return 0;

/\* (2.4) 递归对本node的child node进行platform device创建 \*/

for\_each\_child\_of\_node(bus, child) {

pr\_debug(" create child: %s\n", child->full\_name);

rc = of\_platform\_bus\_create(child, matches, lookup, &dev->dev, strict);

if (rc) {

of\_node\_put(child);

break;

}

}

of\_node\_set\_flag(bus, OF\_POPULATED\_BUS);

return rc;

}

↓

static struct platform\_device \*of\_platform\_device\_create\_pdata(

struct device\_node \*np,

const char \*bus\_id,

void \*platform\_data,

struct device \*parent)

{

struct platform\_device \*dev;

if (!of\_device\_is\_available(np) ||

of\_node\_test\_and\_set\_flag(np, OF\_POPULATED))

return NULL;

/\* (2.2.1) 分配node对应的platform\_device结构，

并且解析node中的"reg"、"interrupts"属性，

作为platform\_device->resource \*/

dev = of\_device\_alloc(np, bus\_id, parent);

if (!dev)

goto err\_clear\_flag;

/\* (2.2.2) device对应的bus为platform\_bus\_type \*/

dev->dev.bus = &platform\_bus\_type;

dev->dev.platform\_data = platform\_data;

of\_dma\_configure(&dev->dev, dev->dev.of\_node);

of\_msi\_configure(&dev->dev, dev->dev.of\_node);

/\* (2.2.3) 注册platform\_device->dev为标准的device \*/

if (of\_device\_add(dev) != 0) {

of\_dma\_deconfigure(&dev->dev);

platform\_device\_put(dev);

goto err\_clear\_flag;

}

return dev;

err\_clear\_flag:

of\_node\_clear\_flag(np, OF\_POPULATED);

return NULL;

}

↓

struct platform\_device \*of\_device\_alloc(struct device\_node \*np,

const char \*bus\_id,

struct device \*parent)

{

struct platform\_device \*dev;

int rc, i, num\_reg = 0, num\_irq;

struct resource \*res, temp\_res;

/\* (2.2.1.1) 分配platform\_device空间 \*/

dev = platform\_device\_alloc("", -1);

if (!dev)

return NULL;

/\* (2.2.1.2) 计算node中"reg"属性中"address、size"resource的个数 \*/

/\* count the io and irq resources \*/

while (of\_address\_to\_resource(np, num\_reg, &temp\_res) == 0)

num\_reg++;

num\_irq = of\_irq\_count(np); /\* 计算node中"interrupts"属性中irq的个数 \*/

/\* (2.2.1.3) 给resource分配空间并解析值 \*/

/\* Populate the resource table \*/

if (num\_irq || num\_reg) {

res = kzalloc(sizeof(\*res) \* (num\_irq + num\_reg), GFP\_KERNEL);

if (!res) {

platform\_device\_put(dev);

return NULL;

}

dev->num\_resources = num\_reg + num\_irq;

dev->resource = res;

for (i = 0; i < num\_reg; i++, res++) {

rc = of\_address\_to\_resource(np, i, res);

WARN\_ON(rc);

}

if (of\_irq\_to\_resource\_table(np, res, num\_irq) != num\_irq)

pr\_debug("not all legacy IRQ resources mapped for %s\n",

np->name);

}

/\* (2.2.1.4) 根据device能够找到of node \*/

dev->dev.of\_node = of\_node\_get(np);

dev->dev.parent = parent ? : &platform\_bus;

/\* (2.2.1.5) 配置device的那么，基本命名规则为：

dev\_set\_name(dev, dev\_name(dev) ? "%llx.%s:%s" : "%llx.%s",

(unsigned long long)addr, node->name,

dev\_name(dev))

\*/

if (bus\_id)

dev\_set\_name(&dev->dev, "%s", bus\_id);

else

of\_device\_make\_bus\_id(&dev->dev);

return dev;

}

### 3.2.2 mt\_i2c\_driver

因为第1级子节点会被注册成platform device，例如i2c/spi控制器，那么对应也需要注册platform driver。已i2c控制器驱动为例：

控制器首先会创建对应platform driver，把adapter注册成i2c device；

在adapter的probe过程中，会调用of\_i2c\_register\_devices()函数遍历控制器下挂的i2c设备的DTS节点，并将其注册成i2c\_client；

drivers\i2c\busses\i2c-mtk.c:

static const struct of\_device\_id mtk\_i2c\_of\_match[] = {

{ .compatible = "mediatek,mt6735-i2c", .data = &mt6735\_compat },

{ .compatible = "mediatek,mt6797-i2c", .data = &mt6797\_compat },

{ .compatible = "mediatek,mt6757-i2c", .data = &mt6757\_compat },

{ .compatible = "mediatek,mt6799-i2c", .data = &mt6799\_compat },

{ .compatible = "mediatek,elbrus-i2c", .data = &elbrus\_compat },

{},

};

static struct platform\_driver mt\_i2c\_driver = {

.probe = mt\_i2c\_probe,

.remove = mt\_i2c\_remove,

.driver = {

.name = I2C\_DRV\_NAME,

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr(mtk\_i2c\_of\_match),

},

};

static int mt\_i2c\_probe(struct platform\_device \*pdev)

{

int ret = 0;

struct mt\_i2c \*i2c;

unsigned int clk\_src\_in\_hz;

struct resource \*res;

const struct of\_device\_id \*of\_id;

i2c = devm\_kzalloc(&pdev->dev, sizeof(struct mt\_i2c), GFP\_KERNEL);

if (i2c == NULL)

return -ENOMEM;

ret = mt\_i2c\_parse\_dt(pdev->dev.of\_node, i2c);

if (ret)

return -EINVAL;

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

i2c->base = devm\_ioremap\_resource(&pdev->dev, res);

if (IS\_ERR(i2c->base))

return PTR\_ERR(i2c->base);

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 1);

i2c->pdmabase = devm\_ioremap\_resource(&pdev->dev, res);

if (IS\_ERR(i2c->pdmabase))

return PTR\_ERR(i2c->pdmabase);

i2c->irqnr = platform\_get\_irq(pdev, 0);

if (i2c->irqnr <= 0)

return -EINVAL;

init\_waitqueue\_head(&i2c->wait);

ret = devm\_request\_irq(&pdev->dev, i2c->irqnr, mt\_i2c\_irq,

#ifdef CONFIG\_MEIZU\_BSP

IRQF\_NO\_SUSPEND | IRQF\_TRIGGER\_NONE, I2C\_DRV\_NAME, i2c);

#else

IRQF\_TRIGGER\_NONE, I2C\_DRV\_NAME, i2c);

#endif /\*CONFIG\_MEIZU\_BSP\*/

if (ret < 0) {

dev\_err(&pdev->dev,

"Request I2C IRQ %d fail\n", i2c->irqnr);

return ret;

}

of\_id = of\_match\_node(mtk\_i2c\_of\_match, pdev->dev.of\_node);

if (!of\_id)

return -EINVAL;

i2c->dev\_comp = of\_id->data;

i2c->adap.dev.of\_node = pdev->dev.of\_node;

i2c->dev = &i2c->adap.dev;

i2c->adap.dev.parent = &pdev->dev;

i2c->adap.owner = THIS\_MODULE;

i2c->adap.algo = &mt\_i2c\_algorithm;

i2c->adap.algo\_data = NULL;

i2c->adap.timeout = 2 \* HZ;

i2c->adap.retries = 1;

i2c->adap.nr = i2c->id;

spin\_lock\_init(&i2c->cg\_lock);

if (i2c->dev\_comp->dma\_support == 2) {

if (dma\_set\_mask(&pdev->dev, DMA\_BIT\_MASK(33))) {

dev\_err(&pdev->dev, "dma\_set\_mask return error.\n");

return -EINVAL;

}

} else if (i2c->dev\_comp->dma\_support == 3) {

if (dma\_set\_mask(&pdev->dev, DMA\_BIT\_MASK(36))) {

dev\_err(&pdev->dev, "dma\_set\_mask return error.\n");

return -EINVAL;

}

}

#if !defined(CONFIG\_MT\_I2C\_FPGA\_ENABLE)

i2c->clk\_main = devm\_clk\_get(&pdev->dev, "main");

if (IS\_ERR(i2c->clk\_main)) {

dev\_err(&pdev->dev, "cannot get main clock\n");

return PTR\_ERR(i2c->clk\_main);

}

i2c->clk\_dma = devm\_clk\_get(&pdev->dev, "dma");

if (IS\_ERR(i2c->clk\_dma)) {

dev\_err(&pdev->dev, "cannot get dma clock\n");

return PTR\_ERR(i2c->clk\_dma);

}

i2c->clk\_arb = devm\_clk\_get(&pdev->dev, "arb");

if (IS\_ERR(i2c->clk\_arb))

i2c->clk\_arb = NULL;

else

dev\_dbg(&pdev->dev, "i2c%d has the relevant arbitrator clk.\n", i2c->id);

#endif

if (i2c->have\_pmic) {

i2c->clk\_pmic = devm\_clk\_get(&pdev->dev, "pmic");

if (IS\_ERR(i2c->clk\_pmic)) {

dev\_err(&pdev->dev, "cannot get pmic clock\n");

return PTR\_ERR(i2c->clk\_pmic);

}

clk\_src\_in\_hz = clk\_get\_rate(i2c->clk\_pmic) / i2c->clk\_src\_div;

} else {

clk\_src\_in\_hz = clk\_get\_rate(i2c->clk\_main) / i2c->clk\_src\_div;

}

dev\_dbg(&pdev->dev, "clock source %p,clock src frequency %d\n",

i2c->clk\_main, clk\_src\_in\_hz);

strlcpy(i2c->adap.name, I2C\_DRV\_NAME, sizeof(i2c->adap.name));

mutex\_init(&i2c->i2c\_mutex);

ret = i2c\_set\_speed(i2c, clk\_src\_in\_hz);

if (ret) {

dev\_err(&pdev->dev, "Failed to set the speed\n");

return -EINVAL;

}

ret = mt\_i2c\_clock\_enable(i2c);

if (ret) {

dev\_err(&pdev->dev, "clock enable failed!\n");

return ret;

}

mt\_i2c\_init\_hw(i2c);

mt\_i2c\_clock\_disable(i2c);

i2c->dma\_buf.vaddr = dma\_alloc\_coherent(&pdev->dev,

PAGE\_SIZE, &i2c->dma\_buf.paddr, GFP\_KERNEL);

if (i2c->dma\_buf.vaddr == NULL) {

dev\_err(&pdev->dev, "dma\_alloc\_coherent fail\n");

return -ENOMEM;

}

i2c\_set\_adapdata(&i2c->adap, i2c);

/\* ret = i2c\_add\_adapter(&i2c->adap); \*/

ret = i2c\_add\_numbered\_adapter(&i2c->adap);

if (ret) {

dev\_err(&pdev->dev, "Failed to add i2c bus to i2c core\n");

free\_i2c\_dma\_bufs(i2c);

return ret;

}

platform\_set\_drvdata(pdev, i2c);

if (!map\_cg\_regs(i2c))

pr\_warn("Map cg regs successfully.\n");

return 0;

}

↓

i2c\_add\_numbered\_adapter() -> \_\_i2c\_add\_numbered\_adapter() ->

↓

static int i2c\_register\_adapter(struct i2c\_adapter \*adap)

{

int res = 0;

/\* Can't register until after driver model init \*/

if (unlikely(WARN\_ON(!i2c\_bus\_type.p))) {

res = -EAGAIN;

goto out\_list;

}

/\* Sanity checks \*/

if (unlikely(adap->name[0] == '\0')) {

pr\_err("i2c-core: Attempt to register an adapter with "

"no name!\n");

return -EINVAL;

}

if (unlikely(!adap->algo)) {

pr\_err("i2c-core: Attempt to register adapter '%s' with "

"no algo!\n", adap->name);

return -EINVAL;

}

rt\_mutex\_init(&adap->bus\_lock);

mutex\_init(&adap->userspace\_clients\_lock);

INIT\_LIST\_HEAD(&adap->userspace\_clients);

/\* Set default timeout to 1 second if not already set \*/

if (adap->timeout == 0)

adap->timeout = HZ;

/\* 注册adapter为i2c\_bus上的device \*/

dev\_set\_name(&adap->dev, "i2c-%d", adap->nr);

adap->dev.bus = &i2c\_bus\_type;

adap->dev.type = &i2c\_adapter\_type;

res = device\_register(&adap->dev);

if (res)

goto out\_list;

dev\_dbg(&adap->dev, "adapter [%s] registered\n", adap->name);

pm\_runtime\_no\_callbacks(&adap->dev);

#ifdef CONFIG\_I2C\_COMPAT

res = class\_compat\_create\_link(i2c\_adapter\_compat\_class, &adap->dev,

adap->dev.parent);

if (res)

dev\_warn(&adap->dev,

"Failed to create compatibility class link\n");

#endif

/\* bus recovery specific initialization \*/

if (adap->bus\_recovery\_info) {

struct i2c\_bus\_recovery\_info \*bri = adap->bus\_recovery\_info;

if (!bri->recover\_bus) {

dev\_err(&adap->dev, "No recover\_bus() found, not using recovery\n");

adap->bus\_recovery\_info = NULL;

goto exit\_recovery;

}

/\* Generic GPIO recovery \*/

if (bri->recover\_bus == i2c\_generic\_gpio\_recovery) {

if (!gpio\_is\_valid(bri->scl\_gpio)) {

dev\_err(&adap->dev, "Invalid SCL gpio, not using recovery\n");

adap->bus\_recovery\_info = NULL;

goto exit\_recovery;

}

if (gpio\_is\_valid(bri->sda\_gpio))

bri->get\_sda = get\_sda\_gpio\_value;

else

bri->get\_sda = NULL;

bri->get\_scl = get\_scl\_gpio\_value;

bri->set\_scl = set\_scl\_gpio\_value;

} else if (!bri->set\_scl || !bri->get\_scl) {

/\* Generic SCL recovery \*/

dev\_err(&adap->dev, "No {get|set}\_gpio() found, not using recovery\n");

adap->bus\_recovery\_info = NULL;

}

}

exit\_recovery:

/\* create pre-declared device nodes \*/

/\* 循环遍历adapter node下挂载的其他子node，注册成为i2c bus的device \*/

of\_i2c\_register\_devices(adap);

acpi\_i2c\_register\_devices(adap);

acpi\_i2c\_install\_space\_handler(adap);

if (adap->nr < \_\_i2c\_first\_dynamic\_bus\_num)

i2c\_scan\_static\_board\_info(adap);

/\* Notify drivers \*/

mutex\_lock(&core\_lock);

bus\_for\_each\_drv(&i2c\_bus\_type, NULL, adap, \_\_process\_new\_adapter);

mutex\_unlock(&core\_lock);

return 0;

out\_list:

mutex\_lock(&core\_lock);

idr\_remove(&i2c\_adapter\_idr, adap->nr);

mutex\_unlock(&core\_lock);

return res;

}

↓

static void of\_i2c\_register\_devices(struct i2c\_adapter \*adap)

{

struct device\_node \*node;

/\* Only register child devices if the adapter has a node pointer set \*/

if (!adap->dev.of\_node)

return;

dev\_dbg(&adap->dev, "of\_i2c: walking child nodes\n");

/\* 遍历adapter node下的子node，并创建标准的i2c bus的device \*/

for\_each\_available\_child\_of\_node(adap->dev.of\_node, node) {

if (of\_node\_test\_and\_set\_flag(node, OF\_POPULATED))

continue;

of\_i2c\_register\_device(adap, node);

}

}

↓

static struct i2c\_client \*of\_i2c\_register\_device(struct i2c\_adapter \*adap,

struct device\_node \*node)

{

struct i2c\_client \*result;

struct i2c\_board\_info info = {};

struct dev\_archdata dev\_ad = {};

const \_\_be32 \*addr\_be;

u32 addr;

int len;

dev\_dbg(&adap->dev, "of\_i2c: register %s\n", node->full\_name);

if (of\_modalias\_node(node, info.type, sizeof(info.type)) < 0) {

dev\_err(&adap->dev, "of\_i2c: modalias failure on %s\n",

node->full\_name);

return ERR\_PTR(-EINVAL);

}

addr\_be = of\_get\_property(node, "reg", &len);

if (!addr\_be || (len < sizeof(\*addr\_be))) {

dev\_err(&adap->dev, "of\_i2c: invalid reg on %s\n",

node->full\_name);

return ERR\_PTR(-EINVAL);

}

addr = be32\_to\_cpup(addr\_be);

if (addr & I2C\_TEN\_BIT\_ADDRESS) {

addr &= ~I2C\_TEN\_BIT\_ADDRESS;

info.flags |= I2C\_CLIENT\_TEN;

}

if (addr & I2C\_OWN\_SLAVE\_ADDRESS) {

addr &= ~I2C\_OWN\_SLAVE\_ADDRESS;

info.flags |= I2C\_CLIENT\_SLAVE;

}

if (i2c\_check\_addr\_validity(addr, info.flags)) {

dev\_err(&adap->dev, "of\_i2c: invalid addr=%x on %s\n",

info.addr, node->full\_name);

return ERR\_PTR(-EINVAL);

}

info.addr = addr;

info.of\_node = of\_node\_get(node);

info.archdata = &dev\_ad;

if (of\_get\_property(node, "wakeup-source", NULL))

info.flags |= I2C\_CLIENT\_WAKE;

result = i2c\_new\_device(adap, &info);

if (result == NULL) {

dev\_err(&adap->dev, "of\_i2c: Failure registering %s\n",

node->full\_name);

of\_node\_put(node);

return ERR\_PTR(-EINVAL);

}

return result;

}

### 3.2.3 mz\_mag\_driver

具体的I2c设备驱动，在总线驱动使用of\_i2c\_register\_devices()创建设备以后，就可以适配工作了。

dts：

arch\arm64\boot\dts\mediatek\mt6799.dtsi:

i2c1: i2c@11090000 {

compatible = "mediatek,mt6799-i2c";

id = <1>;

reg = <0 0x11090000 0 0x1000>,

<0 0x11000100 0 0x80>;

interrupts = <GIC\_SPI 85 IRQ\_TYPE\_LEVEL\_LOW>;

clocks = <&pericfg CLK\_PERICFG\_RG\_I2C1\_BCLK>, <&pericfg CLK\_PERICFG\_RG\_AP\_DM>;

clock-names = "main", "dma";

clock-div = <5>;

};

arch\arm64\boot\dts\mediatek\mz6799\_6m\_v2\_2k\_n.dtsi:

&i2c1 {

apds9922:apds9922@53 {

compatible = "mediatek,apds9922";

interrupt-parent = <&eintc>;

interrupts = < 8 IRQ\_TYPE\_EDGE\_FALLING>;

debounce = <8 0>;

gpio = < 8 >;

reg = <0x53>;

status = "okay";

};

}

driver：

drivers\iio\magnetometer\mz\_mag.c:

static const struct i2c\_device\_id mz\_mag\_id[] = {

{"mediatek,mmc3530", 0 },

{"mediatek,akm09911", 1 },

{ }

};

static struct i2c\_driver mz\_mag\_driver = {

.probe = mz\_mag\_probe,

.id\_table = mz\_mag\_id,

.driver = {

.name = MZ\_MAG\_DEV\_NAME,

.owner = THIS\_MODULE,

#ifdef CONFIG\_OF

.of\_match\_table = of\_match\_ptr(msensor\_of\_match),

#endif

},

};

————————————————

版权声明：本文为CSDN博主「pwl999」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/pwl999/article/details/79631434