DEVICETREE

April 8, 2019 (Http://Www.Mysixue.Com/?P=89) JL
(Http://Www.Mysixue.Com/?Author=1)

1. Introduction to document structure

First , introduce the background of Device Tree , describe why you want to use device-tree, and what it does .
Then introduce the grammatical format of Device Tree , so that you can understand a device tree source file, and modify this file.
Then it introduces how to compile a device tree source file that humans can understand into a device tree binary file that can be read by machines .
Finally, how the kernel is to parse device tree binary file of .

2. Introduction to Device Tree

In this chapter, we need to clarify two issues :
1. What is the function of Device Tree , why use device tree (why-device-tree)
2. How to use Device Tree (how-device-tree)

2.1 why-device-tree

In the article ``Unified Device Model'', we mentioned the concepts of bus, devic e, and driver .
device is used to describe a device , driver for driving a device , it can exhibit the respective functions .

An analogy : the SPI controller on an ARM CPU , we call it a device . Usually
With a platform_device structure to describe which : comprises a register address , using the GPIO pins , interrupt number and the like , platform_device generally placed inside the board file .
Use a platform_driver as a driver : The driver mainly makes the device show the corresponding function , such as how to send and receive data through the SPI controller , the driver code generally exists in the drivers/ directory .

There are often many boards for an ARM CPU on the market , such as FSL 's IMX.6 CPU, which has no less than 10 boards . Each board has only a small difference , such as the SPI controller used on this board. Pins are used as data lines , and there are several other pins used on another board . In this case, we often need to modify the platform_device in the board-level file . Or create a new board-level file for each board . .

Looks pretty good is not it , Linux kernel itself is designed for compatibility with a variety of hardware , each board a board-level file , we do not conflict with each other , very good ! Older version of the kernel has been doing , until one day Linux author Linus feel uncomfortable , he felt the same CPU different board-level documents are rubbish , but many connotations not any , should be removed from the kernel code , so there is now a device-tree mechanism .

This mechanism is equivalent to putting those platform_devices in the device tree , and then the bootloader passes this " tree " to the kernel , and the kernel is parsing the " tree " to obtain relevant information .
This has many advantages , such as : Imagine , if not the mechanism , then for the same CPU two different boards , you need to do two board-level files , compile the kernel twice , and then were run on a different board ; If Device Tree, you only need a board-level file , compile a kernel image , pass different " Tree " can be , easy to use the same kernel compatible with different hardware .

In addition to storing some " platform_device " information in the device tree , it can also contain many other contents . In summary , the main ones are as follows .
The main content of Device Tree :

- kernel which may be a board-level support for all file FSL IMX.6 board , the other board-level file supports all Atmel SAMA5D3 board ... ; Device Tree first to tell the kernel , which board-level file should be used . **Equivalent mach id** .
- Kernel runtime Parameters : The information in the old way , is uboot by bootargs passed to the kernel , including mem, console, filesystem_type the like , can now be placed in device tree in
- ❄♒♏ topology and characteristics of the device : What ARM CPU is used, what on-chip peripherals does this CPU have , the register address of each peripheral , the pins used , interrupts, etc. , equivalent to " platform_device "

2.2 how-device-tree

If you want to use Device Tree :
First, users must understand the device topology of their own hardware , kernel runtime parameters and which board-level file to use
Then organize this information into Device Tree source file (**DTS**)
Then the compiler Device Tree Compiler (**DTC**), to those suitable for human reading of DTS for processing machines become Device Tree binary File (**DTB**).

Then the bootloader (such as uboot) will first load this DTB into MEMORY1, then load the kernel into MEMORY2, start the kernel and tell the kernel to obtain the DTB from the location of MEMORY1 .

It's very simple . The logic is really simple . The trouble is how to write this DTS , which we will introduce soon .
Before that, think about a question :
Assume two boards have used the same IMX.6 the CPU, so that two boards of DTS file will not have a lot of duplicate place ? The answer is yes . In this case , we can extract the common parts , Becomes a dtsi file , and then DTS of different boards can #include this dtsi file .

The hierarchy of DTS follows the hierarchy of actual hardware :
For example ARM company designed a system architecture Cortex-A9; FSL company with this architecture design IMX6A, IMX6B, IMX6C 3 in CPU; Embest company with IMX6A made two boards IMX6A-BOARD1 , IMX6A-Board2.
Then the hierarchy of DTS is

A dtsi file describes the Cortex-A9 architecture , the general official name is skeleton. dtsi
A dtsi file description 3 models CPU common parts , imx6x. Dtsi
3 Ge unused dtsi describe 3 models CPU: imx6a.dtsi, imx6b.dtsi, imx6c.dtsi
2 different DTS descriptions 2 boards : imx6a-board1.dts , imx6a-board2.dts
In turn include a front dtsi.
Note: **The dts of the latter level can override the content of the previous level** .

3. Device Tree Source syntax format

DTS basic unit is a node, node which contains the Property; then these node a tree fashion tissue . Follows :

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a -cell-property = <1> ;
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;
        child-node1 {
            first-child-property ;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property ;
        n-cell-property = <1 2 3>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

In the following subsections :
First, we will introduce the grammatical format of node and property
Then introduce several special nodes
Then introduce several special properties
Then introduce how to customize node and property

3.1 Node & Property

node by node-name plus one pair {} composition . {} may contain a description of the node 's property and child-node.

```
[ label :]node-name [@unit-address] {
     [ properties definitions]
     [ child nodes]
}
```

[] Indicates optional content

- node-name : **a must , which consists of a simple ascii characters , up to 31 characters** node-name. **General on behalf of the node is used to describe any device** : for example, a 3com509 of Ethernet controllers , node-name should be ethernet, and Not 3com509 .
  node-name at the same level should UNIQUE , different levels can be of the same name , similar to the folder .
- @ unit-address: Optional , if a node below reg this property, then @ unit-address is a must ; otherwise , it must not have @ unit-address
- label: optional , if we want to refer to a particular node, in the absence of label in the case , must be given full path for the job . Label action is what you refer to a node time , the direct use of " & label " manner on the line , you do not need to knock it long string full path.
  The label must be unique in the entire DTS
- properties definitions : on behalf of the node 's properties , we will introduce the writing rules of the property below.
- child nodes : child nodes, child nodes follow the same rules . Except for the root node, each node parent is its parent node

Properties defined using key = value ; form , different key representative of the type of Property, and the value of the form are the following :

- can be empty
  an-empty-property ;
- Can be a text string or string list
  a-string-property = "A string";
  a-string-list-property = "first string", "second string";
- may be . 1 or more 32 'bit unsigned integers , each 32bit integer is referred to . 1 th cell
  1 cell: a-cell-property = <1>;
  3 cells: n-cell-property = <1 2 3>;
- Can be binary data
  a-byte-data-property = [0x01 0x23 0x34 0x56];
- It can also be the above type of mix
  mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;

3.2 root node

Each DTS are **must** have **a** root node.

```
/ {
    compatible = "acme,coyotes-revenge";
};
```

- The node-name of the root node must be /
- ❄♒♏ root node has no parent, and all other nodes are its child-node
- the root Node contains the necessary property, of these property details , refer to the corresponding section
  - compatible : used to match board-level files , refer to the " compatible property " section

- #address-cells and #size-cells, refer to the section " reg&cell property " for details of them . These two properties are optional

3.3 CPUs node

Each DTS both **must** have a cpus node described chip CPU, for example a Cortex-A9 dual-core CPU:

```
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <1>;
        };
    };
```

- The node-name of the cpus node must be cpus
- Its parent is the root node, and each child-node describes a CPU
- It generally contains two attributes # address-cells , #-size cells , is used to describe its child-node of reg properties , reg property details with reference to the corresponding section
- Each child-node must contain DEVICE_TYPE , its attribute value is " CPU " .

3.4 memory node

Each DTS must have a memory node to describe the memory status .

```
    memory {
        device_type = "memory";
        reg = <0x80000000 0x10000000>; /* 256 MB */
    };
```

- If there is a device_type attribute under the memory node , and the attribute value is " memory " . There is no special restriction on the node-name .
  Without this attribute , the node-name must be a memory @ 0
- Although its parent is not mandatory to be the root node, it is strongly recommended that its parent-node be the root node.

3.5 device node

The device node is used to describe those " platform-device " , for example :

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    serial@101F0000 {
        compatible = "arm,pl011";
        reg = <0x101F0000 0x8000>;
    };

    spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x8000>;
    };

};
```

- ☞❑❒ the meaning of reg&cell property , please refer to the corresponding chapter
- compatible is used to match the device driver

3.6 interrupt node & related property

Each CPU has an interrupt controller , which handles interrupts from DMA, SPI, UART, GPIO and other peripherals ; among them, GPIO has many pins , and each pin can generate interrupts , but all GPIO pins The interrupt shares an interrupt number on the CPU side .
In this case :
How to describe the interrupt controller of the CPU ;
If you specify the interrupt number of each peripheral ;
How to describe the interrupt of each pin of GPIO ?

Let's use an example to illustrate :

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    INTC : @ 48.2 million interrupt-Controller {
        compatible = "ti,am33xx-intc";
        interrupt-controller ;
        #interrupt-cells = <1>;
        reg = <0x48200000 0x1000>;
    };

    spi0: spi@48030000 {
        compatible = "ti,omap4-mcspi";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x48030000 0x400>;
        interrupts = <65>;
    }

    gpio0: gpio@44e07000 {
        compatible = "ti,omap4-gpio";
        ti, hwmods = "gpio1";
        gpio-controller ;
        #gpio-cells = <2>;
        interrupt-controller ;
        #interrupt-cells = <2>;
        reg = <0x44e07000 0x1000>;
        interrupts = <96>;
    };

    wlcore : wlcore {
        interrupt-parent = <&gpio0 >;
        interrupts = <31 IRQ_TYPE_LEVEL_HIGH>; /* gpio 31 */
    };


};
```

intc This node describes an interrupt controller , let's look at **how to describe an interrupt controller** :

- **interrupt-controller** ; This empty attribute means that this node is an interrupt controller
- **#interrupt-cells** = <1>; This attribute represents how many cells the interrupt controller will use to describe the interrupt

The interrupt controller is described , so how to use it , there is an attribute under root-node :

- **interrupt-parent** = <&intc>; This attribute represents which interrupt controller the node uses . If there is no interrupt-parent attribute under a node , it will integrate the interrupt controller of the parent-node
- For example spi0 this node does not interrupt-parent attribute , use the root-node interrupt controller , which is & intc
  - **interrupts** = <65>; Representative spi0 in intc the interrupt number is 65, because intc requirements for . 1 th cell described interrupt , so here only . 1 th cell can

The node gpio0 is special :
First , it uses intc as the interrupt controller , and the interrupt number is 96;
Secondly , it is itself an interrupt controller , and requires a 2 th cell described interrupt .

And wlcore this node is used gpio0 as an interrupt controller , and should gpio0 requirements , with two cell is described interrupt : The first cell represents the interrupt number , the second cell represents the interrupt trigger

3.7 aliases node

This is a special node with some aliases defined . Its form is generally as follows :

```
    aliases {
        ethernet0 = ð0;
        serial0 = &serial0;
    };
```

➢ node-name must be aliases

➢ The value of the attribute can be either a label or a full path.

- ■🖐♐ ♓♦ is a label, it is equivalent to giving the label an alias
- If the full path, the path of the equivalent of abbreviations , allow other node references . This is the role with " Lable " similar
- Note that , the form ethernet0 = & eth0; instead `Ethernet0 = < & eth0 > ;` ! ! The former is to take an alias , `the` latter is to refer to a certain `node`

3.8 chosen node

The chosen node is mainly used to describe the runtime parameters specified by the system bootloader .

```
    chosen {
        bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
    };
```

➢ If there is chosen this node, its parent-node must be root-node

➢ node-name must be chosen or chosen@0

➢ bootargs: previously passed to the kernel through u-boot

3.9 compatible property

It is a text string or String List ; generally comprises the root node and the device node in .
In the root node in , it is used to match documents which use board .
In the device node in , it is used to match the device driver.

```
compatible = "manufacturer, model1 ‶[, ‶manufacturer, model2″] ;
```

➢ compatible typically made . 1 th string composed , shaped like a " Manufacturer, MODEL1 " . [] Is optional content

- manufacturer: represents the manufacturer , such as freescale
- mode1: represents a certain device of the manufacturer , such as imx6
- Together is : compatible = " freescale , imx6 " ;

3.10 reg & cell property

When do I need reg property it ? Suppose a node below contains address information , you need to use reg.
For example :
When we use a node to describe the SPI controller , the node must contain the register address of the SPI controller ;
In the example , when we use a node description DDRAM controller, when , needed given DDRAM access address , the address with the hardware connected to the relationship (which address lines with a few , which use chip-select pin, etc.).
The former case is relatively simple , the latter requires the use of memory map, that is, the ranges property.

Let's take a look at the grammatical format of reg property . To describe a reg, two other properties are needed : #address-cells , #size-cells .

```
    parent-node {
        #address-cells = <1>;
        #size-cells = <1>;
        child-noe@fc069000 {
            reg = <0xfc0690000x800 [address2 length2] ….>;
        };
    };
```

- ☝ a child-node contains address information , its parent-node has two important attributes
  - # address-cells : the representative child-node of reg properties , with a few uint32 integer representing the base address , if a 32 bit the CPU, generally . 1 th cell on it ; if 64 bit the CPU, it is necessary 2 th cell
  - #size-cells : In the reg attribute representing the child-node , use several uint32 integers to represent the address length
- reg = <0xfc069000 0x800 [address2 length2] ….>
  - conjunction with the above , a . 1 th uint32 represents a group address , the base address is 0xfc069000
  - with . 1 th uint32 representative length , length is 0x800 , i.e. 2K
  - [] is optional, and some regs may need to describe multiple addresses , their base addresses are different , and their lengths are different . However, multiple addresses follow the same rules
  - If the node contains under reg information , the node-name must contain @ unit-address , the value of unit-address is the first address in reg .
- where the address from the parent-node 's perspective , and is limited to the parent-node .
  If an address is a CPU needs to access , for example, SPI register address control is fc069000 , is required from the root-node to see the angle , ensure the address is seen fc069000
  So how to convert the direct address domain between node and node ? You need to use the rangs property, which is specifically introduced in the next section .

3.11 rang e s property

The rang e s attribute is used to switch the address domain between node and node . Let's use an example to understand it .

Assuming we have an external-bus , two controllers , DDRAM and Norflash , are mounted below :
The chip selection of DDRAM is 0, and the address space is 128M;
Norfalsh 's chip select is 1, and the address space is 64M.
Then the node describing external-bus should look like this :

```
    external-bus {
        #address-cells = <2>
        #size-cells = <1> ;

        ddram@0,0 {
            compatible = "ddram-manufacture, ddram-model";
            reg = <0 0 0x8000000>;
        };

        norflash@1,0 {
            compatible = "norflash-manufacture, norflash-model";
            reg = <1 0 0x4000000>;
        };
    };
```

- # = address-cells <2> : represented by a 2 two cell described external-bus of the child-node of reg, the first cell representative of chip select , the second cell denotes an offset from the start of the chip select
  # size-cells = <1> represented by a . 1 th cell described address length

Perfect is not it , but above reg address description , CPU is not accessible . CPU only know to control its address lines , you have to tell the CPU to send any address access to DDRAM, send what addresses have access to Norflash. These addresses It is the address seen from the perspective of root-node .
At this time, you need to use the rang e s attribute , we add the rang e s attribute , and the code becomes like this :

```
/ {
    compatible = "cpu-manufacture, cpu-model";
    #address-cells = <1>;
    #size-cells = <1>;

    external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0 0x10100000 0x8000000 //Chipselect 1, DDRAM
                  1 0 0x18100000   0x4000000>; //Chipselect 2, Norflash

        ddram@0,0 {
            compatible = "ddram-manufacture, ddram-model";
            reg = <0 0 0x8000000>;
        };

        norflash@1,0 {
            compatible = "norflash-manufacture, norflash-model";
            reg = <1 0 0x4000000>;
        };
    };
};
```

The red part above is the added code , rang es converts the address domain under the external-bus node to roo-node.

- ranges = < child – address parent – address size – of – the – region – in – the – child – address – space >
  - Child - address : it is the use of several cell say then ? External-Bus the Node at # address-cells is how much , we must use the number of cell.
    The value of child-address corresponds to the value of child-node under the external-bus node one -to-one , for example, 0 0; 1 0
  - parent-address : it is the use of several cell say then ? External-Bus the Node 's parent-node of # address-cells is how much , we must use the number of cell.
    parent-address value is parent-node access to address , such as 0x10100000 , it means CPU access DDRAM time , use this address is

- size-of-at The-Region at The-Child-in-Space-address- : it is that with a few cell say then ? External-Bus the Node 's parent-node of # size-cells is how much , we must use the number of cell .
  0x8000000 on behalf of the CPU angle , can access the base address is 0x10100000 , address range 128M region .
  This value can be larger than the address range of the node ddram@0,0 . In this case, it means that from the perspective of the CPU , a large address range (for example, 512M) can be accessed , but only 128M is mounted under external-bus . the DDRAM.

The above basically explains the grammatical format of the ranges attribute . In addition to the above , **ranges can also be empty** , as follows :

```
/ {
    compatible = "cpu-manufacture, cpu-model";

    platform-node {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges ;

        spi@fc069000 {
            reg = <0xfc069000 0x800>;
        };

        i2c@44e0b000 {
            #address-cells = <1>;
            #size-cells = <0>;

            reg = <0x44e0b000 0x1000>;

            tps@24 {
                reg = <0x24>;
            };
        };
    };
};
```

The above example , platform-node is a virtual concept , it's not a real bus , to create such a node only to the following child-node are classified , so that logic a little clearer .
For SPI, the CPU directly access the register address fc069000 on it , in this case , Ranges will be empty , **on behalf of 1: 1 address field switching** .

In addition Did you notice , the I2C @ 44e0b000 this node under and no ranges this property , **this case shows that the node in the address field does not need to convert to the parent node under** .
For example, tps@24 describes a touch screen controlled by i2c , and the i2c device address of the screen is 0x24, the address only i2c sense controller , for CPU does not make sense , does not require ranges to root-node under .

3.12 Custom node and property

In addition to the above-mentioned special nodes and properties, we can also customize some nodes or properties. Customized nodes or pro perty should also follow the grammatical format we listed in section 3.1 .

Custom node and properties are generally used for certain specific drivers . For the meaning represented by these nodes and properties , we need to make a txt document and store it inDocumentation / devicetree / bindings directory .

For the DTS that already exists in the kernel code , if there is a custom node or property in it, you can also find an explanation in the bindings directory .

4. Compile Device Tree Binary

No need to know the details for now

5. How to use Device Tree in kernel code

5.1 How UBOOT passes dtb to the kernel

During the kernel startup process , you will enter the file linux/arch/arm/kernel/head.S (we will introduce the detailed process of kernel startup in a special article).
head.S there is a comment , the definition of the bootloader and kernel parameter passing requirements :

```
/*
* Kernel startup entry point.
* ―――――――――――
*
* This is normally called from the decompressor code. The requirements
* are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
* r1 = machine nr , r2 = atags or dtb pointer .
*
```

The current kernel supports the old tag list method , and also supports the device tree method .
r2 may be a pointer to DTB (the bootloader must be copied to memory before being passed to the kernel) , or it may be a pointer to tag list .
In ARM boot code compilation portion (primarily head.S and head-common.S) , Machine NR and pointing DTB or atags pointer is stored in the variable __machine_arch_type and __atags_pointer in , is done to follow c codes are processed .

5.2 The process of kernel parsing dtb

NOTE: the kernel code **logic** mainly described in this section ; about dtb code related details , such as the kernel data structures which describe what a Node, in what manner to scan device-tree, in the next section .

The kernel boot process , after the completion of the implementation of the above assembly code , do some other initialization action , then runs to init / main.c in , perform start_kernel function , start_kernel will call arch / arm / kernel / setup.c in the setup_arc h function .
setup_arch will call the corresponding functions to parse dtb. These functions are as follows :

```
void __init setup_arch ( char ** cmdline_p )
{
    const struct machine_desc * mdesc ;

    setup_ processor ( );

    /* Match board-level files ,  parse runtime parameters, and parse memory node information ,
     * __atags_pointer is the dtb  assigned in the assembly code the pointer to
  */
    mdesc = setup_machine_fdt ( __atags_pointer );

     /* If dtb  is not used ,  the old method is used to match the board-level files .  The  old method mainly
relies on the machine id  passed by u-boot */
    if (! mdesc )
        mdesc = setup_machine_tags ( __atags_pointer , __machine_arch_type );

    ...

    /*For  each node, generate a struct device_node
     * And mount all device_nodes  to the global linked list of_allnodes in
  */
    unflatten_device_ tree ( );

    / * Parse DTB  in CPUs node in */
    arm_dt_init_cpu_ maps ( );

    ...
}
```

How to match board-level files

The old method relies on the machine id passed by u-boot to match the board-level files , and the new method uses dtb to match . The kernel system defaults to the dtb method .

In dtb mode , the **setup_machine_fdt** function is used to match board-level files . Let's take a look at his implementation details .

Implementation file : arch/arm/kernel/devtree.c

```
const struct machine_desc * __init setup_machine_fdt ( unsigned int dt_phys )
{
    const struct machine_desc * mdesc , * mdesc_best = NULL ;
```

```
#ifdef CONFIG_ARCH_MULTIPLATFORM
    DT_MACHINE_ START ( GENERIC_DT , "Generic DT based system" )
    MACHINE_END

    mdesc_best = & __mach_desc_GENERIC_DT ;
#endif

     / * 1th: verification pass over dtb is effective , upon verification , will convert the virtual address
dtb pointer stored in initial_boot_params in , for subsequent use Code * /
    if (! dt_phys || ! early_init_dt_verify ( phys_to_virt ( dt_phys )))
        return NULL ;

    /* 2th: Find the most suitable machine_desc */
    mdesc = of_flat_dt_match_machine ( mdesc_best , arch_get_next_mach );

     /* 3th: If the most suitable machine_desc is not found , use early_print to print out the
relevant message .
      * early_print bottom layer of is to use the assembly operation serial port to print messages
  */
    if (! mdesc ) {
        const char * prop ;
        int size ;
        unsigned long dt_root ;

        early_ print ( "\nError: unrecognized/unsupported "
                "Device tree compatible list :\ n[ " );

        dt_root = of_get_flat_dt_ root ( );
        prop = of_get_flat_dt_prop ( dt_root , "compatible" , & size );
        while ( size > 0 ) {
            early_ print ( "'%s' " , prop );
            size -= strlen ( prop ) + 1 ;
            prop += strlen ( prop ) + 1 ;
        }
        early_ print ( "]\n\n" );

        dump_machine_ table ( ); /* does not return */
    }

     /* We really don't want to do this, but sometimes firmware provides buggy
data */
    if ( mdesc -> dt_fixup )
        mdesc -> dt_fixup ();
    /* 4th: If machine_desc is found , perform some other processing , this function will be described
separately later */
    early_init_dt_scan_ nodes ( );

    /* Change machine number to match the mdesc we're using */
    __machine_arch_type = mdesc -> nr ;

    return mdesc ;
}
```

The general process of setup_machine_fdt has been annotated in Chinese in the above code .
This function returns a machine_desc, we can machine_desc simply understood as a description for board-level file . You should be able to roughly guess that function does it : according to dtb from a bunch of kernel machine_desc in , select the most The appropriate machine_desc . (In the above codeof_flat_dt_match_machine is used to implement this logic).

Then the kernel pile machine_desc is how to generate it ? Is based on what rules to select the most appropriate machine_desc it ?

First answer the first question : how did the bunch of machine_desc in the kernel come into being?
In the board-level file of the kernel , DT_MACHINE_START , MACHINE_END will be used to define a machine_desc. For example, the following code (arch\arm\mach-omap2\board-generic.c) :

```
#ifdef CONFIG_SOC_OMAP2420
static const char * const omap242x_boards_compat [] __initconst = {
    " Ti, omap2420" ,
    NULL ,
};

DT_MACHINE_ START ( OMAP242X_DT , "Generic OMAP2420 (Flattened Device Tree)" )
    . reserve    = omap_reserve ,
    . map_io     = omap242x_map_io ,
    . init_early = omap2420_init_early ,
    . init_machine   = omap_generic_init ,
    . init_ time  = omap2_sync32k_timer_init ,
    .dt_ compat = omap242x_boards_compat ,
    . restart    = omap2xxx_restart ,
MACHINE_END
#endif
```

DT_MACHINE_START is a macro , it defines a machine_desc, and put the machine_desc stored in the ".arch.info.init" This section (Section) in .
If DT_MACHINE_START is used in multiple board-level files , there will be a bunch of machine_desc in the section ".arch.info.init" .

In answer to the second question : According to what rules to choose the most suitable machine_desc
of_flat_dt_match_machine is used to select machine_desc of , its logic is : from dtb of the root-node acquires the compatible value of an attribute , then the ".arch.info.init" in this section sequentially removed machine_desc; Comparative compatible -value and machine_desc - > dt_compat , find the most suitable machine_desc. The so-

called most suitable is the one with the most matching string .
After you find it , you know which board-level file to use .

How to parse runtime parameters

existIn the setup_machine_fdt function , if a suitable machine_desc is found , it will call early_init_dt_scan_nodes for subsequent processing .

The processing done in early_init_dt_scan_nodes is as follows :

- resolve runtime parameters.
- save the root-node in the # address-cells and # size-cells values
- Analyze the memory- node and save the memory information for use by the kernel system .

This section only analyzes the first two issues , and the last issue is described in the next section .

```
void __init early_init_dt_scan_nodes ( void )
{
    /* Retrieve various information from the /chosen node */
    of_scan_flat_ dt ( early_init_dt_scan_chosen ,boot_command_line );

    /* Initialize {size ,address }-cells info */
    of_scan_flat_ dt ( early_init_dt_scan_root , NULL );

    /* Setup memory, calling early_init_dt_add_memory_arch */
    of_scan_flat_ dt ( early_init_dt_scan_memory , NULL );
}
```

The of_scan_flat_dt function means to scan each node in dtb , and for each node, call the function pointer passed by the first parameter .

How to parse runtime parameters : call early_init_dt_scan_chosen for each node , and save the result in the global variable boot_command_line .

early_init_dt_scan_chosen

- find root-node below . 1 stage (depth ==. 1) node Child- in , there is no node-name is the "chosen" or "chosen @ 0" of the node
- If there is this node, the acquired node under bootargs value of the property . And this value is stored in boot_command_line this global variable .
  If the chosen node is not defined in dtb or there is no bootargs attribute under the node , the kernel will use the default CONFIG_CMDLINE

How to save the values of #address-cells and #size-cells under root-node : call early_init_dt_scan_root for each node .

early_init_dt_scan_root

- First judge whether it is root-node (whether the depth is equal to 0)
- ✋♐ it is root-node, get the two attribute values under the node
  - get # size-cells value of this property , and the results stored in dt_root_size_cells
  - get # address-cells value of this property , and the results stored in dt_root_addr_cells

If these two attributes are not defined under root-node , the kernel system will use the default values (OF_ROOT_NODE_SIZE_CELLS_DEFAULT , OF_ROOT_NODE_ADDR_CELLS_DEFAULT)

How to deal with memory mode

Following the content of the previous section , **how to parse memory-node** : call the early_init_dt_scan_memory function for each node .

early_init_dt_scan_memory

- Check if there is an attribute named device_type under node :
  If yes , judge whether the attribute value is " memory " , if yes , then this node is a memory node
  Without device_type this property , it is determined that the node whether the root-node of . 1 stage child-node, and the node-name whether the " Memory @ 0 " , such as the conditions are met , then this node is a Memory node
- for all memory node (most DTB only in . 1 th memory-node), acquired node under reg attribute , then combined dt_root_addr_cells , dt_root_size_cells , the reg parse out the memory starting address and size , the last of this memory is Information is added to the kernel system

How to deal with CPUs node

The setup_arch function will call arm_dt_init_cpu_maps to parse the CPUs node. The details are as follows :

```
void __init arm_dt_init_cpu_maps ( void )
```

```
{
    /* Find the device node whose full path is "/cpus" .
      * The device node cpus is just a container , which includes the definition of each child cpu
node and the properties shared by all cpu nodes .
     */
    cpus = of_find_node_by_path ( "/cpus" );

      for_each_child_of_node ( cpus , cpu ) { // traverse all child nodes of
cpus
        U32 HWID ;
         IF ( of_node_cmp ( cpu -> of the type , "cpu" )) // we only care about those
device_type is cpu 's node
             continue ;

         if ( of_property_read_u32 ( cpu , "reg" , & hwid )) { // Read the value of
the reg attribute and assign it to hdid
             return ;
        }
         /* The 8 MSBs of the attribute value of reg must be set to 0, which is defined by the ARM CPU
binding .*/
        if ( hwid & ~ MPIDR_HWID_BITMASK )
             return ;

        /* Duplicate CPU id is not allowed , that is a catastrophic setting */
        for ( j = 0 ; j < cpuidx ; j ++)
```

```
            if ( WARN ( tmp_map [ j ] == hwid , "Duplicate /cpu reg "
                                     " Properties in the DT\n" ))
                return ;

        /*The  array tmp_map  saves the MPIDR  value ( CPU  ID  value) of all CPUs  in the system ,
         *The  specific index  coding rules are :
         * Tmp_map [0] holds the booting CPU 's id  value ,
         *   The ID  values of the rest of the CPUs  are stored in the positions 1  to NR_CPUS  .
         */
        if ( hwid == mpidr ) {
            i = 0 ;
            bootcpu_valid = true ;
        } else {
            i = cpuidx ++;
        }
        tmp_map [ i ] = hwid ;
    }

    /*  Set the cpu  logical  map  array according to the information in the DTB  */
    for ( i = 0 ; i < cpuidx ; i ++) {
        set_cpu_ possible ( i , true );
        cpu_logical_ map ( i ) = tmp_map [ i ];
    }
}
```

To understand the content of this part , you need to understand the concept of ARM CUPs binding , you can refer to the description of the CPU.txt file in the Documentation/devicetree/bindings/arm directory

How to deal with interrupt node

Initialization is throughstart_kernel ->init_IRQ->machine_desc->implemented by init_irq() .
machine_desc is defined in the board-level file that we found that matches the DTB best .
The init_irq() function is generally called directlyof_irq_init

of_irq_init is defined in drivers/of/irq.c , it will scan each node to determine whether there is an "interrupt-controller" attribute in the node . If there is , it will be regarded as an interrupt control node .

of_irq_init actual details of the code more complex than the above , understand its need to meet the interrupt subsystem of background knowledge , here no details .

How to deal with device node

Article mentioned at the beginning , device-tree that corresponds to the original definition in the board file platform_device, into device-tree in coming .
The basic unit of device-tree is node. You should be able to guess that each device-node represents a platform_device. But how does the kernel code deal with it ? In the device model chapter, we mentioned platform_device/bus/driver, the use of device-tree after the mechanism , device model did not change , the node

must be converted into a number of " struct platform_device " , and register to platfrom_bus next , to the driver with the work .

OK, let's take a look at how the kernel code by node generated platfrom_device of .
The code execution path is as follows : start_kernel -> rest_init -> kernel_init -> kernel_init_freeable -> do_basic_setup -> do_initcalls .
In the do_initcalls function , the kernel will execute each initcall function in turn . In this process , customize_machine will be called , as follows :

```
static int __init customize_machine ( void )
{

    if ( machine_desc -> init_machine )
        machine_desc -> init_ machine ( );
    else
        of_platform_ populate ( NULL , of_default_bus_match_table , NULL , NULL
);

    return 0 ;
}
arch_ initcall ( customize_machine );
```

customize_machine first calls machine_desc-> init_machine , if machine_desc not defined init_machine, will be called of_platform_populate .
`of_platform_populate` is used to `node` converted into `platfrom_device` is . Most board-level defined in the file `init_machine` inside , but also directly call`of_platform_populate`

Let's take a look at the specific implementation of `of_platform_populate (drivers/of/platform.c)`

```
static int of_platform_bus_create ( struct device_node * bus , - - the device
node to be created
                    const struct of_device_id * matches ,—— list to match
                    const struct of_dev_auxdata * lookup , -auxiliary data
            struct device * parent , bool strict )——— parent points to the parent node.
Does strict require an exact match
{
    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 ;

    /* Make sure it has a compatible property */
     /* This code shows that if a node represents platform_device, it must have a compatible
attribute */
    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 ); /* Find additional data matching the device
node in the incoming lookup table */
    if ( auxdata ) {
         bus_id = auxdata -> name ; /* If found, use the statically defined content in the additional
data */
        platform_data = auxdata -> platform_data ;
    }

     /* ARM company provides the CPU core , in addition , it designed the AMBA bus to connect each
block in the SOC .
        * The peripherals on the SOC that comply with this bus standard are called ARM Primecell
Peripherals.
      * If a node device is compatible attribute value is arm, primecell words ,
      * You can call of_amba_device_create to add an amba device to the amba bus
      */
    if ( of_device_is_compatible ( bus , "arm,primecell" )) {
        of_amba_device_ create ( bus , bus_id , platform_data , parent );
        return 0 ;
    }

    /* If it is not ARM Primecell Peripherals,
      * Then we need to platform bus increase on a platform device the
  */
     dev = of_platform_device_create_pdata ( bus , bus_id , platform_data ,
parent );
    if (! dev || ! of_match_node ( matches , bus ))
        return 0 ;

    /* A device node may have child-node,
      * So it is necessary to call of_platform_bus_create repeatedly to process all device nodes
  */
    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 ;
        }
    }
    return rc ;
```

```
}
```

OK, of_platform_device_create_pdata in the above code is used to create platform_device.

```
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 ;

    / * Check status property , ensure that enable or OK in
   * enable/OK means the device is enabled
    * In general , the CPU corresponding dtsi peripherals on each piece of document which will node creates good , but the status is disable
      * In the board-level file , if we want to use an on-chip peripheral , set its status override to enable
      */
    if (! of_device_is_available ( np ))
        return NULL ;

   /* of_device_alloc in addition to allocating the memory of struct platform_device ,
     * The memory of the resource required by the platform device is also allocated ( refer to the resource member in struct platform_device ) .
    * Of course , this needs to resolve the device node 's interrupt attributes and reg property
  */
    dev = of_device_alloc ( np , bus_id , parent );
    if (! dev )
        return NULL ;

    /* Set other members in platform_device */
    dev -> dev . coherent_dma_mask = DMA_BIT_MASK ( 32 );
    if (! dev -> dev . dma_mask )
        dev -> dev . dma_mask = & dev -> dev . coherent_dma_mask ;
    dev -> dev . bus = & platform_bus_type ;
    dev -> dev . platform_data = platform_data ;

    /* Register this platform device to the platform_bus bus */
    if ( of_device_add ( dev ) != 0 ) {
        platform_device_ put ( dev );
        return NULL ;
    }
    return dev ;
}
```

After completing the above process , a `node` containing the `compatible` attribute is converted into a `platform_device` and registered on the `platfrom_bus` bus

5.3 kernel parsing related code of dtb

This section describes dtb related code details , data structures, and so on .

data structure

System kernel , with struct device_node to abstract the device tree one of the Node, are interpreted as follows :
Header file : include/linux/of.h

struct device_node	Comment
const char *name	node-name
const char *ty pe	The attribute value of the " device_type " attribute
phandle phandle	The phandle of the node can be understood as a pointer to the location of the node
const char *full_name	node 's full path
struct property *properties	The list of attributes of the node
struct property *deadprops	If you need to delete some properties , the kernel is not really deleted , but hangs into th e list of deadprops
struct device_node *parent	Point to the parent node
struct device_node *child	Point to child node
struct device_node *sibling	Point to sibling node
struct device_node *next	Through this pointer, you can get the next node of the same type (I don't really understa nd its specific meaning yet)
struct device_node *allnext	Used to mount yourself in the node global list linked list node global list is a global chained list , DTB all node are mounted on this list the
struct kobject kobj	Reference count of this node
unsigned long _flags	
void *data	

Code analysis

Header file : include/linux/of.h
Implementation file : drivers / of /fdt.c

Macro Logically , when the u-boot the DTB then passed to the kernel , the kernel parses the DTB : for each Node, generates a struct device_node , and initializes the device_node relevant content ; then this device_node mount system kernel a global variables defined **of_allnodes** in .
The implementation of the above logic is defined in fdt.cunflatten_device_tree function , as follows

```
void __init unflatten_device_tree ( void )
{
        __unflatten_device_ tree ( initial_boot_params , & of_allnodes ,
                                   early_init_dt_alloc_memory_arch );
        /* Get pointer to "/chosen" and "/aliases" nodes for use everywhere */
        of_alias_ scan ( early_init_dt_alloc_memory_arch );
}
```

When unflatten_device_tree is executed , the global list **of_allnodes** in on the preservation of all device_node information , other modules can fdt.c series provided API to query node related information .
These APIs include (only some of them are listed):

API device_node	**Comment**
struct device_node *of_find_node_by_path(const char *path)	According to the given full path, get a device_node
…	Not one by one , I will add it later

Linux (Http://Www.Mysixue.Com/?Cat=5) , Device Model (Http://Www.Mysixue.Com/?Cat=8)