2. Device-Tree

NXKR_YoungSikYang

Exported on 03/06/2024

Table of Contents

Device tree

1. Understanding the device tree

Refer to the page [`u-boot/1. Device Driver/1`]

2. How the device tree is loaded by Bootloader

During the boot process, the secondary bootloader loads the DTB into memory. The specific method of loading can vary:

- **Directly from Storage**: The bootloader reads the DTB from its storage location into RAM.
- **Packaged with the Kernel**: In some configurations, the DTB may be appended to the kernel image. The bootloader loads the entire package into memory, and the kernel extracts the DTB.
- **User Selection or Automatic Detection**: In systems with multiple possible hardware configurations, the bootloader may present a selection to the user or automatically detect the hardware configuration to choose the correct DTB.

3. Platform device - Platform driver

- **Direct integration**: Devices managed by platform drivers are tightly integrated with the system hardware.
- **Static Configuration**: Information about the device (like memory addresses, IRQ numbers, etc) is specified within static data like the Device Tree.
- **No Hardware Enumeration**: These devices are typically directly integrated into the motherboard and do not have an enumeration mechanism unlike PCI or USB devices.

3.1 How to connect a device to a driver

3.1.1 Add a device(mydevice@1) in the device tree (`s5p6818.dtsi`)

```
{
        model = "nexell soc";
        compatible = "nexell,s5p6818";
        #address-cells = <0x1>;
        #size-cells = <0x1>;

        aliases {
                serial0 = &serial0;
                serial1 = &serial1;
                serial2 = &serial2;
                serial3 = &serial3;
                serial4 = &serial4;
                serial5 = &serial5;
                i2s0    = &i2s_0;
                i2s1    = &i2s_1;
                i2s2    = &i2s_2;
                spi0    = &spi_0;
                spi1    = &spi_1;
                spi2    = &spi_2;
                i2c0    = &i2c_0;
                i2c1    = &i2c_1;
                i2c2    = &i2c_2;

                pinctrl0 = &pinctrl_0;
        };

        mydevice@1 {
                compatible = "yang,mydevice";
                /* Other necessary properties */
        };
```

3.1.2 Source code

The driver code should match the device name(mydevice@1) and 'compatible'(yang,mydevice)

- `drivers/platform/my_platform_driver.c`

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>

// Device ID table
static struct of_device_id testdev_of_match[] = {
    { .compatible = "yang,mydevice", },
    {},
};
MODULE_DEVICE_TABLE(of, testdev_of_match);

// Probe function - called when device is detected
static int testdev_probe(struct platform_device *pdev)
{
    printk(KERN_INFO "yang's platform driver probed\n");
    // Device initialization and resource allocation go here
    // For example, mapping device memory:
    // void __iomem *regs = devm_platform_ioremap_resource(pdev, 0);
    return 0; // Return 0 if device is successfully initialized
}

// Remove function - called when device is removed
static int testdev_remove(struct platform_device *pdev)
{
    printk(KERN_INFO "yang's platform driver removed\n");
    // Cleanup code here, like freeing allocated resources
    return 0;
}

// Platform driver structure
static struct platform_driver testdev_driver = {
    .probe = testdev_probe,
    .remove = testdev_remove,
    .driver = {
        .name = "mydevice@1",
        .of_match_table = testdev_of_match,
        .owner = THIS_MODULE,
    },
};

// Module initialization
static int __init testdev_init(void)
{
    printk(KERN_INFO "yang's platform driver init\n");
```

```
    return platform_driver_register(&testdev_driver);
}

// Module exit
static void __exit testdev_exit(void)
{
    printk(KERN_INFO "yang's platform driver exit\n");
    platform_driver_unregister(&testdev_driver);
}

module_init(testdev_init);
module_exit(testdev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Test Platform Driver");
```

- Modify `drivers/platform/Makefile` to add the source to the built list

  obj-y indicates it will be a built-in driver that is built along with the kernel instead of a module driver.

```
obj-y += my_platform_driver.o
```

3.1.3 Probe

Check the probe log to see if the device has been connected to the driver.

```
dmesg | grep "yang"
```

```
# dmesg | grep "yang"
yang's test device driver init
yang's test device driver probed
```

4. How to view the device tree in linux

4.1 /proc/device-tree/

```
root@s5p6818:~# ls /proc/device-tree/
#address-cells   chosen          gpio_keys       memory          name            nx-v4l2         psci
#size-cells      compatible      i2c@10          model           nexell-ion@0    oscillator      reserved-memory
aliases          cpus            leds            mydevice@1      nx-devfreq      pmu             soc
```

This directory contains directories and files corresponding to nodes and properties defined in the Device Tree

4.2 /sys/firmware/devicetree/base/

```
root@s5p6818:~# ls /sys/firmware/devicetree/base/
#address-cells   chosen          gpio_keys       memory          name            nx-v4l2         psci
#size-cells      compatible      i2c@10          model           nexell-ion@0    oscillator      reserved-memory
aliases          cpus            leds            mydevice@1      nx-devfreq      pmu             soc
```

This path is like **/proc/device-tree/ but** is often preferred because it's more structured and designed to be easier to navigate programmatically.

4.3 Reading these directories and files

Reading them allows for obtaining information about the system's hardware configuration, such as peripheral addresses, interrupt numbers, and device parameters.

They can be read as described below.

4.3.1 Checking the child nodes of a device

```
ls /sys/firmware/devicetree/base/mydevice@1
```

4.3.2 Reading the properties

```
cat /sys/firmware/devicetree/base/mydevice@1/name
cat /sys/firmware/devicetree/base/mydevice@1/compatible
```

Memory set-up

1. Disable the default DRAM setup

First, modify common/fdt_support.c so that CONFIG_SYS_SDRAM_SIZE in u-boot `configs/s5p6818_bitminer.h does not automatically set up the memory` .

`fdt_fixup_memory_banks()` overrides the device tree and sets up the memory.

```
int fdt_fixup_memory_banks(void *blob, u64 start[], u64 size[], int banks)
{
    return 0;
```

2. Memory size

This section shows how to change the memory size by modifying the device tree.

2.1 Before changing the memory size

2.1.1 Device tree(s5p6818-bitminer-common.dtsi)

```
memory {
    device_type = "memory";
    reg = <0x40000000 0x0db00000>;
```

format: reg = <start_address size>

2.1.2 Check the memory size

```
dmesg | grep "memory"
```

```
Memory: 177728K/224256K available (6292K kernel code, 896K rwdata, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

2.2 After changing the memory size

2.2.1 Device tree(s5p6818-bitminer-common.dtsi)

```
memory {
    device_type = "memory";
    reg = <0x40000000 0xab00000>;
```

format: reg = <start_address size>

2.2.2 Check the memory size

```
dmesg | grep "memory"
```

```
Memory: 128576K/175104K available (6292K kernel code, 896K rwdata, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

3. Reserved memory

Reserved memory within the Linux kernel refers to portions of memory that are allocated for specific uses or devices and are not available for general-purpose use by the operating system's memory manager.

Below is about how to handle reserved memory using the device tree.

3.1 Before creating reserved memory

3.1.1 Device tree(s5p6818.dtsi)

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
};

mydevice@1 {
        compatible = "yang,mydevice";
        /* Other necessary properties */
};
```

3.1.2 Check the memory

```
cat /proc/meminfo
```

```
root@s5p6818:~# cat /proc/meminfo
MemTotal:         194604 kB
```

```
dmesg | grep "memory"
```

```
Memory: 177728K/224256K available (6292K kernel code, 896K rwdata, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

3.2 After creating reserved memory

3.2.1 Device tree(s5p6818.dtsi)

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    my_reserved: buffer@0 {
        reg = <0x40000000 0x2000000>;
    };
};

mydevice@1 {
        compatible = "yang,mydevice";
        #memory-region = <&my_reserved>;
        /* Other necessary properties */
};
```

`memory-region` property can be used to allocate the reserved memory to a device.

3.2.2 Check if the reserved memory was successfully added

```
cat /proc/meminfo
```

```
root@s5p6818:~# cat /proc/meminfo
MemTotal:         174144 kB
```

```
dmesg | grep "memory"
```

```
Memory: 157268K/224256K available (6292K kernel code, 896K rwdata, 3948K rodata, 492K init, 634K bss, 50604K reserved,
```

4. CMA(Contiguous Memory Allocator)

CMA allows for the allocation of large contiguous blocks of memory before or after the system has booted. It's particularly useful for devices that need to perform DMA operations requiring contiguous physical memory, such as video frame buffers or hardware that performs direct I/O. CMA helps mitigate the issue of memory fragmentation, making it easier to allocate large contiguous memory regions.

Below is about how to add a CMA memory pool using the device tree.

4.1 Device tree(s5p6818.dtsi)

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    my_reserved: buffer@0 {
        #reg = <0x40000000 0x2000000>;
    };
    linux,cma {
        compatible = "shared-dma-pool";
        reusable;
        size = <0x4000000>;
        linux,cma-default;
    };
};
```

4.2 Check the added CMA memory

```
dmesg | grep "memory"
```

```
Reserved memory: created CMA memory pool at 0x0000000049800000, size 64 MiB
```

Reference

- Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt