Application Notes

Amlogic Bootup and Nand EMMC Partition Config Guide

Revision 0.1

AMLOGIC, Inc.
2518 Mission College Blvd
Santa Clara, CA 95054
U.S.A.
www.amlogic.com

Content

Revision History

Revision	Revised Date	By	Changes
0.1	Aug. 21, 2014	Long Yu	Initial release draft

1. Bootup Config

Boot up config information are obtained through Device Tree.

The Flattened Device Tree (FDT) is a data structure for describing the hardware in a system. It is a derived from the device tree format used by Open Firmware to encapsulate platform information and convey it to the operating system. The operating system uses the FDT data to find and register the devices in the system.

Currently the Linux kernel can read device tree information in the ARM, x86, Microblaze, PowerPC, and Sparc architectures. There is interest in extending support for device trees to other platforms, to unify the handling of platform description across kernel architectures.

For example: we open meson_xxxx.dtd document of meson_xx platform, then we can revise some parameters of devices.

The device tree is a simple tree structure of nodes and properties. Properties are key-value pairs, and node may contain both properties and child nodes. For example, the following is a simple tree in the .dts format:

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

This tree is obviously pretty useless because it doesn't describe anything, but it does show the structure of nodes an properties. There is:

- a single root node: "/"
- a couple of child nodes: "node1" and "node2"
- a couple of children for node1: "child-node1" and "child-node2"
- a bunch of properties scattered through the tree.

Properties are simple key-value pairs where the value can either be empty or contain an arbitrary byte stream. While data types are not encoded into the data structure, there are a few fundamental data representations that can be expressed in a device tree source file.

- Text strings (null terminated) are represented with double quotes:

```
string-property = "a string"
```

- 'Cells' are 32 bit unsigned integers delimited by angle brackets:

```
cell-property = <0xbeef 123 0xabcd1234>
```

- binary data is delimited with square brackets:

```
binary-property = [0x01 0x23 0x45 0x67];
```

- Data of differing representations can be concatenated together using a comma:

```
mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;
```

CPUs

Next step is to describe for each of the CPUs. A container node named "cpus" is added with a child node for each CPU. In this case the system is a dual-core Cortex A9 system from ARM.

```
/ {
    compatible = "acme,coyotes-revenge";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

The compatible property in each cpu node is a string that specifies the exact cpu model in the form <manufacturer>,<model>, just like the compatible property at the top level.

Node Names

It is worth taking a moment to talk about naming conventions. Every node must have a name in the form <name>[@<unit-address>].

<name> is a simple ascii string and can be up to 31 characters in length. In general, nodes are named according to what kind of device it represents. ie. A node for a 3com Ethernet adapter would be use the name ethernet, not 3com509.

The unit-address is included if the node describes a device with an address. In general, the unit address is the primary address used to access the device, and is listed in the node's reg property. We'll cover the reg property later in this document.

Sibling nodes must be uniquely named, but it is normal for more than one node to use the same generic name so long as the address is different (ie, serial@101f1000 & serial@101f2000).

Devices

Every device in the system is represented by a device tree node. The next step is to populate the tree with a node for each of the devices. For now, the new nodes will be left empty until we can talk about how address ranges and irqs are handled.

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

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };

    serial@101F0000 {
        compatible = "arm,pl011";
    };

    serial@101F2000 {
        compatible = "arm,pl011";
    };

    gpio@101F3000 {
        compatible = "arm,pl061";
    };

    interrupt-controller@10140000 {
        compatible = "arm,pl190";
    };
```

```
        spi@10115000 {
            compatible = "arm,pl022";
        };

        external-bus {
            ethernet@0,0 {
                compatible = "smc,smc91c111";
            };

            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                rtc@58 {
                    compatible = "maxim,ds1338";
                };
            };

            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";
            };
        };
    };
```

In this tree, a node has been added for each device in the system, and the hierarchy reflects the how devices are connected to the system. ie. devices on the extern bus are children of the external bus node, and i2c devices are children of the i2c bus controller node. In general, the hierarchy represents the view of the system from the perspective of the CPU.

This tree isn't valid at this point. It is missing information about connections between devices. That data will be added later.

Some things to notice in this tree:

- Every device node has a compatible property.
- The flash node has 2 strings in the compatible property. Read on to the next section to learn why.

Devices that are addressable use the following properties to encode address information into the device tree:

- reg
- #address-cells
- #size-cells

Each addressable device gets a reg which is a list of tuples in the form reg = <address1 length1 [address2 length2] [address3 length3] ... >. Each tuple represents an address range used by the device. Each address value is a list of one or more 32 bit integers called cells. Similarly, the length value can either be a list of cells, or empty.

Since both the address and length fields are variable of variable size, the #address-cells and #size-cells properties in the parent node are used to state how many cells are in each field. Or in other words, interpreting a reg property correctly requires the parent node's #address-cells and #size-cells values. To see how this all works, lets add the addressing properties to the sample device tree, starting with the CPUs.

CPU addressing

The CPU nodes represent the simplest case when talking about addressing. Each CPU is assigned a single unique ID, and there is no size associated with CPU ids.

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

In the cpus node, #address-cells is set to 1, and #size-cells is set to 0. This means that child reg values are a single uint32 that represent the address with no size field. In this case, the two cpus are assigned addresses 0 and 1. #size-cells is 0 for cpu nodes because each cpu is only assigned a single address.

You'll also notice that the reg value matches the value in the node name. By convention, if a node has a reg property, then the node name must include the unit-address, which is the first address value in the reg property.

Memory Mapped Devices

Instead of single address values like found in the cpu nodes, a memory mapped device is assigned a range of addresses that it will respond to. #size-cells is used to state how large the length field is in each child reg tuple. In the following example, each address value is 1 cell (32 bits), and each length value is also 1 cell, which is typical on 32 bit systems. 64 bit machines may use a value of 2 for #address-cells and #size-cells to get

```
    / {
        #address-cells = <1>;
        #size-cells = <1>;

        ...

        serial@101f0000 {
            compatible = "arm,pl011";
            reg = <0x101f0000 0x1000 >;
        };

        serial@101f2000 {
            compatible = "arm,pl011";
            reg = <0x101f2000 0x1000 >;
        };

        gpio@101f3000 {
            compatible = "arm,pl061";
            reg = <0x101f3000 0x1000
                   0x101f4000 0x0010>;
        };

        interrupt-controller@10140000 {
            compatible = "arm,pl190";
            reg = <0x10140000 0x1000 >;
        };

        spi@10115000 {
            compatible = "arm,pl022";
            reg = <0x10115000 0x1000 >;
        };

        ...
    };
```

Each device is assigned a base address, and the size of the region it is assigned. The GPIO device address in this example is assigned two address ranges; 0x101f3000...0x101f3fff and 0x101f4000..0x101f400f.

Some devices live on a bus with a different addressing scheme. For example, a device can be attached to an external bus with discrete chip select lines. Since each parent node defines the addressing domain for its children, the address mapping can be chosen to best describe the system. The code below show address assignment for devices attached to the external bus with the chip select number encoded into the address.

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

            ethernet@0,0 {
                compatible = "smc,smc91c111";
                reg = <0 0 0x1000>;
            };

            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                reg = <1 0 0x1000>;
                rtc@58 {
                    compatible = "maxim,ds1338";
                };
            };

            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";
                reg = <2 0 0x4000000>;
            };
        };
```

The external-bus uses 2 cells for the address value; one for the chip select number, and one for the offset from the base of the chip select. The length field remains as a single cell since only the offset portion of the address needs to have a range. So, in this example, each reg entry contains 3 cells; the chipselect number, the offset, and the length.

Since the address domains are contained to a node and its children, parent nodes are free to define whatever addressing scheme makes sense for the bus. Nodes outside of the immediate parent and child nodes do not normally have to care about the local addressing domain, and addresses have to be mapped to get from one domain to another.

Non Memory Mapped Devices

Other devices are not memory mapped on the processor bus. They can have address ranges, but they are not directly accessible by the CPU. Instead the parent device's driver would perform indirect access on behalf of the CPU.

To take the example of i2c devices, each device is assigned an address, but there is no length or range associated with it. This looks much the same as CPU address assignments.

```
            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <1 0 0x1000>;
                rtc@58 {
                    compatible = "maxim,ds1338";
                    reg = <58>;
                };
            };
```

2. Nand/EMMC partition config

The nand is arranged as an array of pages. A page consists of 256 / 512 Byte data and 8 / 16 Byte spare (out of band) area. Newer chips have 2048 Bytes data and and 64 Bytes spare area sizes. The spare area is used to store ECC (error correction code), bad block information and filesystem dependend data. n pages build one block. The read / write access to data is on a per page basis. Erase is done on a per block basis. The commands to read / write / erase the chip is given by writing to the chip with the Command Latch Enable pin high. Address is given by writing with the Address Latch Enable pin high.

The emmc is a managed memory capable of storing code and data. It is specifically designed for mobile devices. The e•MMC is intended to offer the performance and features required by mobile devices while maintaining low power consumption. The e•MMC device contains features that support high throughput for large data transfers and performance for small random data more commonly found in code usage. It also contains many security features

Nand/emmc partition config is specified in storage.c document.

For example:

struct partitions partition_table[MAX_PART_NUM]={

/*each of partitions includes name, size and mask capacities. The name is the name of patition. The size is the capacity of patition. The mask_flags indicates the type of patition code. It include code,data,cache type.

/boot

This is the partition that enables the phone to boot, as the name suggests. It includes the kernel and the ramdisk. Without this partition, the device will simply not be able to boot. Wiping this partition from recovery should only be done if absolutely required and once done, the device must NOT be rebooted before installing a new one, which can be done by installing a ROM that includes a /boot partition.

/system

This partition basically contains the entire operating system, other than the kernel and the ramdisk. This includes the Android user interface as well as all the system applications that come pre-installed on the device. Wiping this partition will remove Android from the device without rendering it unbootable, and you will still be able to put the phone into recovery or bootloader mode to install a new ROM.

/recovery

The recovery partition can be considered as an alternative boot partition that lets you boot the device into a recovery console for performing advanced recovery and maintenance operations on it. To learn more about this partition and its contents

/data

Also called userdata, the data partition contains the user's data – this is where your contacts, messages, settings and apps that you have installed go. Wiping this partition essentially performs a factory reset on your device, restoring it to the way it was when you first booted it, or the way it was after the last official or custom ROM installation. When you perform a wipe data/factory reset from recovery, it is this partition that you are wiping.

/cache

This is the partition where Android stores frequently accessed data and app components. Wiping the cache doesn't effect your personal data but simply gets rid of the existing data there, which gets automatically rebuilt as you continue using the device.

/misc

This partition contains miscellaneous system settings in form of on/off switches. These settings may include CID (Carrier or Region ID), USB configuration and certain hardware settings etc. This is an important partition and if it is corrupt or missing, several of the device's features will will not function normally.

```
struct partitions partition_table[MAX_PART_NUM]={
{
        .name = "logo",
        .size = 32*SZ_1M,
        .mask_flags = STORE_CODE,
},
{
        .name = "recovery",
                .size = 32*SZ_1M,
                .mask_flags = STORE_CODE,
        },
        {
                .name = "misc",
                .size = 32*SZ_1M,
                .mask_flags = STORE_CODE,
        },
        {
                .name = "boot",
                .size = 32*SZ_1M,
                .mask_flags = STORE_CODE,
        },
        {
                .name = "system",
                .size = 1024*SZ_1M,
                .mask_flags = STORE_CODE,
        },
        {
                .name = "cache",
                .size = 512*SZ_1M,
                .mask_flags = STORE_CACHE,
        },
        {
                .name = "data",
                .size = NAND_PART_SIZE_FULL,
                .mask_flags = STORE_DATA,
        },
        /*{
                .name = "media",
                .size = NAND_PART_SIZE_FULL,
        },*/
};
```

Above these partitions, the partition size can be changed, but should pay attention to address alignment. At the same time there are a lot of free space these partitions, convenient and subsequent development.