hardware interfacing with embedded Linux

ESE2025

Flattened Device Tree on Embedded Linux Systems

- according to Derek Molloy, “The flattened device tree (FDT) is a human-readable data structure that describes the hardware on a particular [platform]. The FDT is described using device-tree source (DTS) files, where a .dts file contains a board-level definition and a .dtsi file typically includes SoC-level definitions. The .dtsi files are typically included into a .dts file.” (page 272, *Exploring Beaglebone*, 2nd Ed.)

flattened device tree (DTS) format

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        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 {
        };
    };
};
```

FDT key-value pair data representations

- 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>;`
- Commas are also used to create lists of strings:
    - `string-list = "red fish", "blue fish";`

FDT example (from elinux.org)

Consider the following imaginary machine (loosely based on ARM Versatile), manufactured by "Acme" and named "Coyote's Revenge":

- One 32-bit ARM CPU
- processor local bus attached to memory mapped serial port, spi bus controller, i2c controller, interrupt controller, and external bus bridge
- 256MB of SDRAM based at 0
- 2 Serial ports based at 0x101F1000 and 0x101F2000
- GPIO controller based at 0x101F3000
- SPI controller based at 0x10170000 with following devices
    - MMC slot with SS pin attached to GPIO #1
- External bus bridge with following devices
    - SMC SMC91111 Ethernet device attached to external bus based at 0x10100000
    - i2c controller based at 0x10160000 with following devices
        - Maxim DS1338 real time clock. Responds to slave address 1101000 (0x58)
    - 64MB of NOR flash based at 0x30000000

Initial structure

- The first step is to lay down a skeleton structure for the machine. This is the bare minimum structure required for a valid device tree. At this stage you want to uniquely identify the machine:

```
/dts-v1/;

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

Describe 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.

```
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
 };
```

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

Devices

```
/dts-v1/;
/ {
    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";
        };
    };
};
```

Devices Cont’d

Some things to notice in this tree:

- Every device node has a `compatible` property.
- The flash node has 2 strings in the compatible property.
- As mentioned earlier, node names reflect the type of device, not the particular model.

Understanding the `compatible` Property

Every node in the tree that represents a device is required to have the `compatible` property. `compatible` is the key an operating system uses to decide which device driver to bind to a device.

`compatible` is a list of strings. The first string in the list specifies the exact device that the node represents in the form `"<manufacturer>,<model>"`. The following strings represent other devices that the device is *compatible* with.

For example, the Freescale MPC8349 System on Chip (SoC) has a serial device which implements the National Semiconductor ns16550 register interface. The compatible property for the MPC8349 serial device should therefore be: `compatible = "fsl,mpc8349-uart", "ns16550"`. In this case, `fsl,mpc8349-uart` specifies the exact device, and `ns16550` states that it is register-level compatible with a National Semiconductor 16550 UART.

How Addressing Works

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.
- 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.

```
/dts-v1/;

/ {
    #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 >;
    };
    ...

};
```