Introducción al Device Tree

Mg. Ing. Pablo Slavkin
Mg. Ing. Hanes N. Sciarrone
MSE - 2024

Implementación de Manejadores de Dispositivos

Introducción al Device Tree

- La descripción de hardware mediante un device tree se conoce en linux desde hace muchos años.
- Originalmente utilizada para arquitecturas PowerPC y SPARC.
- En 2005 se comenzó un gran clean-up y merge de arquitecturas de 32 y 64 bits sobre arquitecturas PowerPC.
- Dentro de las decisiones tomadas, se implementó una representación de Device Tree que fuera plana (FDT).
- Esto era necesario para poder pasar un blob binario al kernel al bootear.

Introducción al Device Tree

- Luego de algunos años, esta descripción del hardware se generalizó para varias arquitecturas en el mainline.

- Al día de hoy, arquitecturas como ARM, microblaze, mips, powerpc, sparc, y x86 poseen Device Tree.

- Para ARM, estos archivos de descripción pueden verse en **arch/arm/boot/dts/**

Introducción al Device Tree

- El Device Tree no es más que un archivo de descripción de hardware.
- Su nombre proviene de la estructura de árbol simple que presenta.
- En ella, cada dispositivo se representa como un nodo, con sus propiedades.
- Todos los subsistemas del kernel tienen su enlace con el Device Tree.

Introducción al Device Tree

```
uart0: serial@44e09000 {
    compatible = "ti,omap3-uart";
    ti,hwmods = "uart1";
    clock-frequency = <48000000>;
    reg = <0x44e09000 0x2000>;
    interrupts = <72>;
    status = "disabled";
};
```

Introducción al Device Tree

- **serial@44e09000** es el nombre del **nodo**.
- **uart0** es una etiqueta que puede ser referenciada como &uart0.
- Las demas lineas son **propiedades**.
- Las propiedades pueden ser cadenas de caracteres, enteros o ser referencias a otros nodos.

Introducción al Device Tree

- Cada plataforma de hardware tiene su propio device tree.
- Asimismo, muchas plataformas utilizan el mismo procesador.
- Muchas veces varios procesadores de la misma familia comparten similitudes.
- Para permitir esto, un device tree puede contener otro.
- Esto puede hacerse mediante dos acciones:
  - Declaración **/include/** dada por el lenguaje del DT.
  - Utilizando **#include** como en un archivo C.

Introducción al Device Tree

- La utilización del método de inclusión de archivos es indistinta, no existe preferencia.

```
/ {
    compatible = "ti,am33xx";
    [...]
    ocp {
    [...]
        uart0: serial@44e09000 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x44e09000 0x2000>;
            interrupts = <72>;
            status = "disabled";
            [...]
        };
    };
};
```

am33xx.dtsi

Definition of the AM33xx SoC

+

```
[...]
&uart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins>;
        status = "okay";
};
```

am335x-bone-common.dtsi

Common definitions for BeagleBone boards

+

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
/ {
        model = "TI AM335x BeagleBone Black";
        compatible = "ti,am335x-bone-black",
                     "ti,am335x-bone",
                     "ti,am33xx";
};

[...]
```

am335x-boneblack.dts

Definition for BeagleBone Black

Introducción al Device Tree

Compiled DTB

```
/ {
     compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
     model = "TI AM335x BeagleBone Black";
     [...]
     ocp {
          uart0: serial@44e09000 {
              compatible = "ti,am3352-uart", "ti,omap3-uart";
              reg = <0x44e09000 0x2000>;
              interrupts = <72>;
              pinctrl-names = "default";
              pinctrl-0 = <&uart0_pins>;
              status = "okay";
          };
     };
};
```

am335x-boneblack.dtb

Introducción al Device Tree

- Los archivos que componen un device tree son:
  - Archivos ***.dtsi** los cuales son device tree source includes
    - Describen hardware común a varias plataformas.
    - Son los archivos que se incluyen como se mencionó antes.
  - Archivos ***.dts** que son los device tree source:
    - Describen una plataforma en específico
    - Hacen uso (incluyen) los archivos **dtsi**.

Introducción al Device Tree

- El Device Tree tiene tres propósitos principales:
- **Identificación de plataformas**:
  - El kernel utiliza la información en el DT para identificar la máquina específica (hardware).
  - A veces se utiliza para ejecutar algunos fixes necesarios para un hardware específico.
  - La mayoría de las veces la identidad del hardware es irrelevante, y el kernel selecciona código de configuración basado en el SoC o el CPU.

Introducción al Device Tree

- Se determina coincidencia según la propiedad **compatible** en el nodo raíz del device tree.
- Comparación con lista **dt_compat** contenida en **struct machine_desc**. (**arch/arm/include/asm/mach/arch.h**).

- **Configuración en tiempo de ejecución**:
  - En la mayoría de los casos el DT es el unico metodo de comunicar información del firmware al kernel.
  - Por ello se utiliza para pasar parámetros al kernel (**bootargs**) y la eventual posición de una imagen **initrd**.

Introducción al Device Tree

- La mayoría de esta información se coloca en el nodo **chosen**.

```
chosen {
    bootargs = "console=ttyS0,115200 loglevel=8";
    initrd-start = <0xc8000000>;
    initrd-end = <0xc8200000>;
};
```

Introducción al Device Tree

- **Población de dispositivos**:
  - Luego de que se identifica la plataforma y que la información inicial es parseada, se inicia el kernel de manera normal.
  - La lista de dispositivos se puede obtener parseando el DT y asignando las estructuras de dispositivos dinámicamente.

Introducción al Device Tree

- El DT soporta algunos tipos de datos, los cuales se muestran en el ejemplo de nodo siguiente

```
/* This is a comment */
// This is another comment
node_label: nodename@reg{
    string-property = "a string";
    string-list = "red fish", "blue fish";
    one-int-property = <197>; /* One cell in this property */
};
```

Introducción al Device Tree

```
node_label: nodename@reg{
    int-list-property = <0xbeef 123 0xabcd4>;
    /*each number(cell)is a 32 bit integer(uint32). There are 3 cells in
    this property */
    mixed-list-property = "a string", <0xadbcd45>, <35>, [0x01 0x23 0x45];
    byte-array-property = [0x01 0x23 0x45 0x67];
    boolean-property;
};
```

Introducción al Device Tree

- Las cadenas de caracteres se representan con doble comillas.
- Utilizar comas entre dos cadenas de caracteres genera una lista.
- Las celdas son del tipo 32-bit sin signo delimitados por angle brackets.
- Los tipos de datos booleanos son una propiedad vacía.
- El valor true o false depende de si la propiedad está presente o no.

Introducción al Device Tree

- Existe una convención de nombres para los nodos.
- Cada nodo debe ser llamado de la forma **<name>[@<address>]**.
- El campo **<name>** puede ser tener hasta 31 caracteres.
- El campo **[@<address>]** es opcional dependiendo si el nodo representa un dispositivo que puede asignarse a una dirección.

Introducción al Device Tree

```
i2c@021a0000 {
    compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
    reg = <0x021a0000 0x4000>;
    [...]
};
```

- En este ejemplo vemos un nodo representando un dispositivo i2c.
- La etiqueta está ausente, porque la misma es opcional.

Introducción al Device Tree

- En general las etiquetas sólo son útiles si el nodo debe ser referenciado desde una propiedad de otro nodo.
- Las etiquetas pueden verse como “punteros” a los nodos.
- Es importante entender cómo son utilizadas porque su uso es extensivo en cualquier DT.
- Una etiqueta es una manera de identificar unívocamente un nodo.
- El compilador de DT transforma ese nombre en un valor único de 32-bits

Introducción al Device Tree

```
aliases {
    ethernet0 = &fec;
    gpio0 = &gpio1;
    gpio1 = &gpio2;
    mmc0 = &usdhc1;
    [...]
};
```

```
gpio1: gpio@0209c000 {
    compatible = "fsl,imx6q-gpio",
    "fsl,imx35-gpio";
    [...]
};
node_label: nodename@reg {
    [...];
    gpios = <&gpio1 7 GPIO_ACTIVE_HIGH>;
};
```

Introducción al Device Tree

- Se denomina *pointer handle* (**phandle**) al valor de 32-bit que se asocia a un nodo para poder ser referenciado desde otro.
- Al utilizar la sintaxis **<&mylabel>** se está apuntando al nodo que tiene la etiqueta **mylabel**.
- Observar que la utilización del operador **&** es idéntica que en C.
- Se observa también un nodo “alias”.
- Se introdujeron para no tener que recorrer un DT completo para encontrar un nodo específico.

Introducción al Device Tree

- Los nodos alias actúan como un índice.
- Pueden verse como tablas look-up rápidas.
- Existe una función para localizar un nodo por su alias **find_node_by_alias()**
- Los alias no son utilizados directamente en los archivos fuentes de los DT.
- Son frecuentemente desreferenciados por el Kernel.

Introducción al Device Tree

- Además de los archivos DTS y DTB antes mencionados, existe una tercera forma de ver un DT.
- Es posible ver una representación en tiempo real del DT en **/proc/device-tree**.
- Para poder ver esto se debe configurar la variable **CONFIG_PROC_DEVICETREE** al momento de compilar el kernel.
- Esto es muy útil para efectuar debugging.

Introducción al Device Tree

HANDS ON

1. Explorar las fuentes para ver el DT.
2. Compilar el DT.
3. Configurar el kernel para ver el DT.
4. Navegar a través de /proc/device-tree

Addressing de puertos SPI e I2C

Addressing de SPI e I2C

- Los dispositivos SPI e I2C pertenecen al grupo de dispositivos no mapeados en memoria (no accesibles por el CPU).

- El driver de dispositivo padre (en este caso driver controlador de bus) debe efectuar accesos indirectos.

- Cada dispositivo SPI/I2C se representa como un subnodo del bus correspondiente donde está asentado.

Addressing de SPI e I2C

- Existen propiedades comunes para muchas clases de dispositivos.
- Esto se observa especialmente para los que se desprenden de buses ya conocidos por el kernel (SPI, I2C, USB, etc).
- Son las propiedades **reg**, **#address-cells** y **#size-cells**
- Tienen el propósito de efectuar direccionamiento dentro de los buses.
- La propiedad de direccionamiento principal es **reg**. Las otras dos indican como debe ser interpretado **reg**.

Addressing de SPI e I2C

- La propiedad **reg** en definitiva es una lista de tuples.

```
reg = <address0 size0 [address1size1] [address2size2] ... >
```

- Para el caso de los dispositivos no mapeados en memoria, **#size-cells = 0** y por lo tanto **reg** queda de la forma:

```
reg = <address>
```

Addressing de SPI e I2C

- En el caso especial de los dispositivos I2C, la propiedad **reg** determina la dirección que utilizan en el bus.
- NOTA: Para los dispositivos SPI el valor en **reg** representa el índice de la línea Chip Select asignada al dispositivo.
- Recordar que I2C posee direccionamiento en su protocolo, pero SPI sólo se habilita por líneas CS.

OF Match Style

OF Match Style

- El match tipo OF es el primero ejecutado por el core platform para hacer coincidir dispositivos con sus drivers.
- Utiliza la propiedad **compatible** para linkear la entrada del dispositivo almacenada en **of_match_table**.
- **of_match_table** es un campo de **struct driver**.
- Cada nodo de dispositivo en un DT tiene una propiedad **compatible** que es un string o una lista de strings.

OF Match Style

- Cualquier platform driver que declare una de estas strings listadas en **compatible** provocará un match.
- A partir de este match, la función **probe()** de este driver será ejecutada.
- Cada entrada de DT susceptible de hacer un match es definida en el kernel como una instancia de **struct of_device_id**.
- Esta estructura esta definida en **linux/mod_devicetable.h**

OF Match Style

```
/* we are only interested in the two last elements of the
structure */
struct of_device_id {
    [...]
    char compatible[128];
    const void *data;
};
```

OF Match Style

- Campo **compatible**: Es el string que se utiliza para hacer el match con el dispositivo descrito en el DT.

- Campo ***data**: Este puntero puede apuntar a cualquier estructura. Se utiliza para datos de configuración del device.

OF Match Style

- Como la entrada **of_match_table** en la estructura **driver** es un puntero, puede pasarse un array de estructuras.
- Esto hace que un solo driver pueda ser compatible para más de un dispositivo.

```
static const struct of_device_id imx_uart_dt_ids[] = {
    { .compatible = "fsl,imx6q-uart", },
    { .compatible = "fsl,imx1-uart", },
    { .compatible = "fsl,imx21-uart", },
    { /* sentinel */ }
};
```

OF Match Style

- Como ya se llenó el array de IDs, se deben pasar al campo **of_match_table** de la estructura **driver** correspondiente.

```
static struct platform_driver serial_imx_driver = {
    [...]
    .driver = {
        .name = "imx-uart",
        .of_match_table = imx_uart_dt_ids,
        [...]
    },
};
```

OF Match Style

- En este punto, solo el driver conoce el array **of_device_id**.
- Para informar al kernel, así la lista de IDs se carga en el platform core), debe utilizarse **MODULE_DEVICE_TABLE.**

```
MODULE_DEVICE_TABLE(of, imx_uart_dt_ids);
```

- El driver ya es compatible con DT. Solo resta declarar un nodo en el DT que sea compatible con este driver

OF Match Style

```
uart1: serial@02020000 {
    compatible = "fsl,imx6q-uart", "fsl,imx21-uart";
    reg = <0x02020000 0x4000>;
    interrupts = <0 26 IRQ_TYPE_LEVEL_HIGH>;
    [...]
};
```

Gracias.