The Device Tree: Plug and play for Embedded Linux

Eli Billauer

December 3rd, 2012

Introduction

Embedded processors

- System on Chip (SoC)
- FPGAs
- May have the same instruction set, but...
- ... number of cores, memory size, frequency may vary
- ... different peripherals on the bus
- ... different boards

No BIOS to convey this information.

Sources for hardware information

- The kernel command line
- The kernel configuration
- Hardcoded in the boot sources
- Header files
- The device tree

The result: See arch/arm/

Device tree basics

Device tree forms

The device tree comes in three forms:

- A text file (*.dts) – “source”
- A binary blob (*.dtb) – “binary blob”
- A file system: /proc/device-tree – “runtime”

The blob is loaded into RAM before the kernel kicks off.

Names and acronyms

- Device Tree
- Flattened Device Tree (FDT)
- Open Firmware (OF)

Syntax

- Curly brackets: Hierarchy (directory)
- The hierarchy's (node) name just before curly bracket
- Assignments: Content (file)
- Strings and integers: C-style (0x hex notation)
- Arrays of integers ("cells") within <and >
- Lists of values separated by commas
- C-style comments
- C-style labels

A sample device tree .dts listing

```
/dts-v1/;
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "xlnx,zynq-zed";
  interrupt-parent = <&gic>;
  model = "Xillinux for Zedboard";
  aliases {
    serial0 = &ps7_uart_1;
  } ;
  chosen {
    bootargs = "consoleblank=0 root=/dev/mmcblk0p2 rw rootwait earlyprintk";
    linux,stdout-path = "/axi@0/uart@E0001000";
  };

  cpus {

      [ ... CPU definitions ... ]

   } ;
  ps7_ddr_0: memory@0 {
    device_type = "memory";
    reg = < 0x0 0x20000000 >;
  } ;
```

A sample device tree .dts listing (cont.)

```
  ps7_axi_interconnect_0: axi@0 {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
    ranges ;
    gic: interrupt-controller@f8f01000 {
      #interrupt-cells = < 3 >;
      compatible = "arm,cortex-a9-gic";
      interrupt-controller ;
      reg = < 0xf8f01000 0x1000  >,< 0xf8f00100 0x100  >;
    } ;
    pl310: pl310-controller@f8f02000 {
      arm,data-latency = < 3 2 2 >;
      arm,tag-latency = < 2 2 2 >;
      cache-level = < 2 >;
      cache-unified ;
      compatible = "arm,pl310-cache";
      interrupts = < 0 34 4 >;
      reg = < 0xf8f02000 0x1000 >;
    } ;

      [ ... more peripheral definitions ... ]
  } ;
} ;
```

Accessing /proc/device-tree

```
# hexdump -C '/proc/device-tree/#size-cells'
00000000  00 00 00 01                                       |....|
00000004
# hexdump -C '/proc/device-tree/axi@0/compatible'
00000000  78 6c 6e 78 2c 70 73 37  2d 61 78 69 2d 69 6e 74  |xlnx,ps7-axi-int|
00000010  65 72 63 6f 6e 6e 65 63  74 2d 31 2e 30 30 2e 61  |erconnect-1.00.a|
00000020  00 73 69 6d 70 6c 65 2d  62 75 73 00              |.simple-bus.|
0000002c

# cat '/proc/device-tree/axi@0/compatible'
xlnx,ps7-axi-interconnect-1.00.asimple-bus
```

Note the Big Endian representation of the integer!

Compilation and reverse compilation

- The compiler is part of the Linux kernel tree.
- Compiles from any to any format
- Source to blob:

```
$ scripts/dtc/dtc -I dts -O dtb -o /path/to/my-tree.dtb /path/to/my-tree.dts
```

- Blob to source:

```
$ scripts/dtc/dtc -I dtb -O dts -o /path/to/fromdtb.dts /path/to/found_this.dtb
```

- The /proc/device pseudo filesystem can be converted to source as well

Walking through a DTS file

A starter

```
/dts-v1/;
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "xlnx,zynq-zed";
  interrupt-parent = <&gic>;
  model = "Xillinux for Zedboard";
  aliases {
    serial0 = &ps7_uart_1;
  } ;

  chosen {
    bootargs = "consoleblank=0 root=/dev/mmcblk0p2 rw rootwait earlyprintk";
    linux,stdout-path = "/axi@0/uart@E0001000";
  };
```

CPUs

```
cpus {
  #address-cells = <1>;
  #cpus = <0x2>;
  #size-cells = <0>;
  ps7_cortexa9_0: cpu@0 {
    clock-frequency = <666666688>;
    compatible = "xlnx,ps7-cortexa9-1.00.a";
    d-cache-line-size = <0x20>;
    d-cache-size = <0x8000>;
    device_type = "cpu";
    i-cache-line-size = <0x20>;
    i-cache-size = <0x8000>;
    model = "ps7_cortexa9,1.00.a";
    reg = <0>;
    timebase-frequency = <333333344>;
    xlnx,cpu-1x-clk-freq-hz = <0x69f6bcb>;
    xlnx,cpu-clk-freq-hz = <0x27bc86c0>;
  } ;
  ps7_cortexa9_1: cpu@1 {

    [ ... repeated, of course ... ]
  } ;
} ;
```

Memory

```
ps7_ddr_0: memory@0 {
  device_type = "memory";
  reg = < 0x0 0x20000000 >;
} ;
```

0x20000000 = 512M

Peripherals

```
  ps7_axi_interconnect_0: axi@0 {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
    ranges ;
    gic: interrupt-controller@f8f01000 {
      #interrupt-cells = < 3 >;
      compatible = "arm,cortex-a9-gic";
      interrupt-controller ;
      reg = < 0xf8f01000 0x1000  >,< 0xf8f00100 0x100  >;
    } ;
    pl310: pl310-controller@f8f02000 {
      arm,data-latency = < 3 2 2 >;
      arm,tag-latency = < 2 2 2 >;
      cache-level = < 2 >;
      cache-unified ;
      compatible = "arm,pl310-cache";
      interrupts = < 0 34 4 >;
      reg = < 0xf8f02000 0x1000 >;
    } ;

     [ ... ]

  } ;
} ;
```

Defining a peripheral

Its entry in the device tree

```
xillybus_0: xillybus@50000000 {
  compatible = "xlnx,xillybus-1.00.a";
  reg = < 0x50000000 0x1000 >;
  interrupts = < 0 59 1 >;
  interrupt-parent = <&gic>;
  xlnx,max-burst-len = <0x10>;
  xlnx,native-data-width = <0x20>;
  xlnx,slv-awidth = <0x20>;
  xlnx,slv-dwidth = <0x20>;
  xlnx,use-wstrb = <0x1>;
} ;
```

Kernel code: Load me!

```
static struct of_device_id xillybus_of_match[] __devinitdata = {
  { .compatible = "xlnx,xillybus-1.00.a", },
  {}
};

MODULE_DEVICE_TABLE(of, xillybus_of_match);

[ ... ]

static struct platform_driver xillybus_platform_driver = {
  .probe = xilly_drv_probe,
  .remove = xilly_drv_remove,
  .driver = {
    .name = "xillybus",
    .owner = THIS_MODULE,
    .of_match_table = xillybus_of_match,
  },
};
```

platform_driver_register(&xillybus_platform_driver) must be called in the modules initialization function.

The probe method

A sanity check. Not clear if it's really needed:

```
static int __devinit xilly_drv_probe(struct platform_device *op)
{
  const struct of_device_id *match;

  match = of_match_device(xillybus_of_match, &op->dev);

  if (!match)
    return -EINVAL;
```

The probe method (cont.)

Accessing registers:

```
int rc = 0;
struct resource res;
void *registers;

rc = of_address_to_resource(&op->dev.of_node, 0, &res);
if (rc) {
  /* Fail */
}

if  (!request_mem_region(res.start, resource_size(&res), "xillybus")) {
  /* Fail */
}

registers = of_iomap(op->dev.of_node, 0);

if (!registers) {
  /* Fail */
}
```

The probe method (cont.)

Register the interrupt handler:

```
irq = irq_of_parse_and_map(op->dev.of_node, 0);

rc = request_irq(irq, xillybus_isr, 0, "xillybus", op->dev);
```

This relates to:

```
interrupts = < 0 59 1 >;
interrupt-parent = <&gic>;
```

in the device tree. The numbers' meaning is driver dependent (and sometimes completely off-beat even if it works).

The probe method (cont.)

Grab this piece of data from the device tree:

```
xlnx,slv-awidth = <0x20>;
```

Kernel code:

```
void *ptr;
int value;

  ptr = of_get_property(op->dev.of_node, "xlnx,slv-awidth", NULL);

  if (!ptr) {
    /* Couldn't find the entry */
  }

  value = be32_to_cpup(ptr);
```

Summary

Conclusions

- It's simple!
- It makes sense
- It's useful
- It's a winner
- but...
- Nobody has cared to explain how it works
- Therefore it's messy
- A lot of useless stuff nobody dares to delete

Sources

http://xillybus.com/tutorials/device-tree-zynq-1
http://devicetree.org/Device_Tree_Usage

include/linux/of.h (and of_*.h) in the kernel tree

Thank you!

Questions?