Device trees everywhere

David Gibson <dwg@au1.ibm.com>
Benjamin Herrenschmidt <benh@kernel.crashing.org>
OzLabs, IBM Linux Technology Center

October 4, 2019

Abstract

We present a method for booting a PowerPC® Linux® kernel on an embedded machine. To do this, we supply the kernel with a compact flattened-tree representation of the system's hardware based on the device tree supplied by Open Firmware on IBM® servers and Apple® Power Macintosh® machines.

The "blob" representing the device tree can be created using `dtc`— the Device Tree Compiler — that turns a simple text representation of the tree into the compact representation used by the kernel. The compiler can produce either a binary "blob" or an assembler file ready to be built into a firmware or bootwrapper image.

This flattened-tree approach is now the only supported method of booting a `ppc64` kernel without Open Firmware, and we plan to make it the only supported method for all `powerpc` kernels in the future.

1 Introduction

1.1 OF and the device tree

Historically, "everyday" PowerPC machines have booted with the help of Open Firmware (OF), a firmware environment defined by IEEE1275 [?]. Among other boot-time services, OF maintains a device tree that describes all of the system's hardware devices and how they're connected. During boot, before taking control of memory management, the Linux kernel uses OF calls to scan the device tree and transfer it to an internal representation that is used at run time to look up various device information.

The device tree consists of nodes representing devices or buses[1]. Each node contains *properties*, name–value pairs that give information about the device. The values are arbitrary byte strings, and for some properties, they contain tables or other structured information.

1.2 The bad old days

Embedded systems, by contrast, usually have a minimal firmware that might supply a few vital system parameters (size of RAM and the like), but nothing as detailed or complete as the OF device tree. This has meant that the various 32-bit PowerPC embedded ports have required a variety of hacks spread across the kernel to deal with the lack of device tree. These vary from specialised boot wrappers to parse parameters (which are at least reasonably localised) to CONFIG-dependent hacks in drivers to override normal probe logic with hard-coded addresses for a particular board. As well as being ugly of itself, such CONFIG-dependent hacks make it hard to build a single kernel image that supports multiple embedded machines.

Until relatively recently, the only 64-bit PowerPC machines without OF were legacy (pre-POWER5®) iSeries® machines. iSeries machines often only have virtual IO devices, which makes it quite simple to work around the lack of a device tree. Even so, the lack means the iSeries boot sequence must be quite different from the pSeries or Macintosh, which is not ideal.

The device tree also presents a problem for implementing `kexec()`. When the kernel boots, it takes over full control of the system from OF, even re-using OF's memory. So, when `kexec()` comes

[1] Well, mostly. There are a few special exceptions.

to boot another kernel, OF is no longer around for the second kernel to query.

2 The Flattened Tree

In May 2005 Ben Herrenschmidt implemented a new approach to handling the device tree that addresses all these problems. When booting on OF systems, the first thing the kernel runs is a small piece of code in `prom_init.c`, which executes in the context of OF. This code walks the device tree using OF calls, and transcribes it into a compact, flattened format. The resulting device tree "blob" is then passed to the kernel proper, which eventually unflattens the tree into its runtime form. This blob is the only data communicated between the `prom_init.c` bootstrap and the rest of the kernel.

When OF isn't available, either because the machine doesn't have it at all or because `kexec()` has been used, the kernel instead starts directly from the entry point taking a flattened device tree. The device tree blob must be passed in from outside, rather than generated by part of the kernel from OF. For `kexec()`, the userland `kexec` tools build the blob from the runtime device tree before invoking the new kernel. For embedded systems the blob can come either from the embedded bootloader, or from a specialised version of the `zImage` wrapper for the system in question.

2.1 Properties of the flattened tree

The flattened tree format should be easy to handle, both for the kernel that parses it and the bootloader that generates it. In particular, the following properties are desirable:

- *relocatable*: the bootloader or kernel should be able to move the blob around as a whole, without needing to parse or adjust its internals. In practice that means we must not use pointers within the blob.

- *insert and delete*: sometimes the bootloader might want to make tweaks to the flattened tree, such as deleting or inserting a node (or whole subtree). It should be possible to do this without having to effectively regenerate the whole flattened tree. In practice this means limiting the use of internal offsets in the blob that need recalculation if a section is inserted or removed with `memmove()`.

- *compact*: embedded systems are frequently short of resources, particularly RAM and flash memory space. Thus, the tree representation should be kept as small as conveniently possible.

2.2 Format of the device tree blob

Offset	**Contents**	
0x00	0xd00dfeed	magic number
0x04	*totalsize*	
0x08	*off_struct*	
0x0C	*off_strs*	
0x10	*off_rsvmap*	
0x14	*version*	
0x18	*last_comp_ver*	
0x1C	*boot_cpu_id*	$\geqslant$v2 only
0x20	*size_strs*	$\geqslant$v3 only
⋮	⋮	
off_rsvmap	*address0*	memory reserve table
+ 0x04	...	
+ 0x08	*len0*	
+ 0x0C	...	
⋮	⋮	
	0x00000000-00000000	end marker
	0x00000000-00000000	
⋮	⋮	
off_strs	'n' 'a' 'm' 'e'	strings block
+ 0x04	0 'm' 'o' 'd'	
+ 0x08	'e' 'l' 0 ...	
⋮	⋮	
+ *size_strs*		
⋮	⋮	
off_struct	OF_DT_BEGIN_NODE	structure block
+ 0x04	'/' 0 0 0	root node
+ 0x08	OF_DT_PROP	
+ 0x0C	0x00000005	"`model`"
+ 0x10	0x00000008	
+ 0x14	'M' 'y' 'B' 'o'	
+ 0x18	'a' 'r' 'd' 0	
⋮	⋮	
	OF_DT_END_NODE	
	OF_DT_END	
⋮	⋮	
totalsize		

Figure 1: Device tree blob layout

The format for the blob we devised, was first described on the `linuxppc64-dev` mailing list in [?]. The format has since evolved through various revisions, and the current version is included as part of the `dtc` (see §??) git tree, [?].

Figure ?? shows the layout of the blob of data containing the device tree. It has three sections of variable size: the *memory reserve table*, the *structure block* and the *strings block*. A small header gives the blob's size and version and the locations of the three sections, plus a handful of vital parameters used during early boot.

The memory reserve map section gives a list of regions of memory that the kernel must not use[2]. The list is represented as a simple array of (address, size) pairs of 64 bit values, terminated by a zero size entry. The strings block is similarly simple, consisting of a number of null-terminated strings appended together, which are referenced from the structure block as described below.

The structure block contains the device tree proper. Each node is introduced with a 32-bit `OF_DT_BEGIN_NODE`tag, followed by the node's name as a null-terminated string, padded to a 32-bit boundary. Then follows all of the properties of the node, each introduced with a `OF_DT_PROP`tag, then all of the node's subnodes, each introduced with their own `OF_DT_BEGIN_NODE`tag. The node ends with an `OF_DT_END_NODE`tag, and after the `OF_DT_END_NODE`for the root node is an `OF_DT_END`tag, indicating the end of the whole tree[3]. The structure block starts with the `OF_DT_BEGIN_NODE`introducing the description of the root node (named `/`).

Each property, after the `OF_DT_PROP`, has a 32-bit value giving an offset from the beginning of the strings block at which the property name is stored. Because it's common for many nodes to have properties with the same name, this approach can substantially reduce the total size of the blob. The name offset is followed by the length of the property value (as a 32-bit value) and then the data itself padded to a 32-bit boundary.

[2]Usually such ranges contain some data structure initialised by the firmware that must be preserved by the kernel.

[3]This is redundant, but included for ease of parsing.

2.3 Contents of the tree

Having seen how to represent the device tree structure as a flattened blob, what actually goes into the tree? The short answer is "the same as an OF tree". On OF systems, the flattened tree is transcribed directly from the OF device tree, so for simplicity we also use OF conventions for the tree on other systems.

In many cases a flat tree can be simpler than a typical OF provided device tree. The flattened tree need only provide those nodes and properties that the kernel actually requires; the flattened tree generally need not include devices that the kernel can probe itself. For example, an OF device tree would normally include nodes for each PCI device on the system. A flattened tree need only include nodes for the PCI host bridges; the kernel will scan the buses thus described to find the subsidiary devices. The device tree can include nodes for devices where the kernel needs extra information, though: for example, for ISA devices on a subsidiary PCI/ISA bridge, or for devices with unusual interrupt routing.

Where they exist, we follow the IEEE1275 bindings that specify how to describe various buses in the device tree (for example, [?] describe how to represent PCI devices). The standard has not been updated for a long time, however, and lacks bindings for many modern buses and devices. In particular, embedded specific devices such as the various System-on-Chip buses are not covered. We intend to create new bindings for such buses, in keeping with the general conventions of IEEE1275 (a simple such binding for a System-on-Chip bus was included in [?] a revision of [?]).

One complication arises for representing "phandles" in the flattened tree. In OF, each node in the tree has an associated phandle, a 32-bit integer that uniquely identifies the node[4]. This handle is used by the various OF calls to query and traverse the tree. Sometimes phandles are also used within the tree to refer to other nodes in the tree. For example, devices that produce interrupts generally have an `interrupt-parent` property giving the phandle of the interrupt controller that handles interrupts from this device. Parsing these and other interrupt related properties allows the kernel to build a com-

[4]In practice usually implemented as a pointer or offset within OF memory.

plete representation of the system's interrupt tree, which can be quite different from the tree of bus connections.

In the flattened tree, a node's phandle is represented by a special `linux,phandle` property. When the kernel generates a flattened tree from OF, it adds a `linux,phandle` property to each node, containing the phandle retrieved from OF. When the tree is generated without OF, however, only nodes that are actually referred to by phandle need to have this property.

Another complication arises because nodes in an OF tree have two names. First they have the "unit name", which is how the node is referred to in an OF path. The unit name generally consists of a device type followed by an `@` followed by a *unit address*. For example `/memory@0` is the full path of a memory node at address 0, `/ht@0,f2000000/pci@1` is the path of a PCI bus node, which is under a HyperTransport™ bus node. The form of the unit address is bus dependent, but is generally derived from the node's `reg` property. In addition, nodes have a property, `name`, whose value is usually equal to the first path of the unit name. For example, the nodes in the previous example would have `name` properties equal to `memory` and `pci`, respectively. To save space in the blob, the current version of the flattened tree format only requires the unit names to be present. When the kernel unflattens the tree, it automatically generates a `name` property from the node's path name.

3 The Device Tree Compiler

As we've seen, the flattened device tree format provides a convenient way of communicating device tree information to the kernel. It's simple for the kernel to parse, and simple for bootloaders to manipulate. On OF systems, it's easy to generate the flattened tree by walking the OF maintained tree. However, for embedded systems, the flattened tree must be generated from scratch.

Embedded bootloaders are generally built for a particular board. So, it's usually possible to build the device tree blob at compile time and include it in the bootloader image. For minor revisions of the board, the bootloader can contain code to make the necessary tweaks to the tree before passing it to the booted kernel.

```
/memreserve/ 0x20000000-0x21FFFFFF;

/ {
    model = "MyBoard";
    compatible = "MyBoardFamily";
    #address-cells = <2>;
    #size-cells = <2>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        PowerPC,970@0 {
            device_type = "cpu";
            reg = <0>;
            clock-frequency = <5f5e1000>;
            timebase-frequency = <1FCA055>;
            linux,boot-cpu;
            i-cache-size = <10000>;
            d-cache-size = <8000>;
        };
    };

    memory@0 {
        device_type = "memory";
        memreg: reg = <00000000 00000000
                       00000000 20000000>;
    };

    mpic@0x3fffdd08400 {
        /* Interrupt controller */
        /* ... */
    };

    pci@40000000000000 {
        /* PCI host bridge */
        /* ... */
    };

    chosen {
        bootargs = "root=/dev/sda2";
        linux,platform = <00000600>;
        interrupt-controller =
            < &/mpic@0x3fffdd08400 >;
    };
};
```

Figure 2: Example `dtc` source

The device trees for embedded boards are usually quite simple, and it's possible to hand construct the necessary blob by hand, but doing so is tedious. The "device tree compiler", `dtc`[5], is designed to make creating device tree blobs easier by converting a text representation of the tree into the necessary blob.

[5] `dtc` can be obtained from [?].

3.1 Input and output formats

As well as the normal mode of compiling a device tree blob from text source, `dtc` can convert a device tree between a number of representations. It can take its input in one of three different formats:

- source, the normal case. The device tree is described in a text form, described in §??.
- blob (`dtb`), the flattened tree format described in §??. This mode is useful for checking a pre-existing device tree blob.
- filesystem (`fs`), input is a directory tree in the layout of `/proc/device-tree` (roughly, a directory for each node in the device tree, a file for each property). This is useful for building a blob for the device tree in use by the currently running kernel.

In addition, `dtc` can output the tree in one of three different formats:

- blob (`dtb`), as in §??. The most straightforward use of `dtc` is to compile from "source" to "blob" format.
- source (`dts`), as in §??. If used with blob input, this allows `dtc` to act as a "decompiler".
- assembler source (`asm`). `dtc` can produce an assembler file, which will assemble into a `.o` file containing the device tree blob, with symbols giving the beginning of the blob and its various subsections. This can then be linked directly into a bootloader or firmware image.

For maximum applicability, `dtc` can both read and write any of the existing revisions of the blob format. When reading, `dtc` takes the version from the blob header, and when writing it takes a command line option specifying the desired version. It automatically makes any necessary adjustments to the tree that are necessary for the specified version. For example, formats before 0x10 require each node to have an explicit `name` property. When `dtc` creates such a blob, it will automatically generate `name` properties from the unit names.

3.2 Source format

The "source" format for `dtc` is a text description of the device tree in a vaguely C-like form. Figure ?? shows an example. The file starts with `/memreserve/` directives, which gives address ranges to add to the output blob's memory reserve table, then the device tree proper is described.

Nodes of the tree are introduced with the node name, followed by a `{` ... `};` block containing the node's properties and subnodes. Properties are given as just *name* `=` *value*`;`. The property values can be given in any of three forms:

- *string* (for example, `"MyBoard"`). The property value is the given string, including terminating NULL. C-style escapes (`\t`, `\n`, `\0` and so forth) are allowed.
- *cells* (for example, `<0 8000 f0000000>`). The property value is made up of a list of 32-bit "cells", each given as a hex value.
- *bytestring* (for example, `[1234abcdef]`). The property value is given as a hex bytestring.

Cell properties can also contain *references*. Instead of a hex number, the source can give an ampersand (`&`) followed by the full path to some node in the tree. For example, in Figure ??, the `/chosen` node has an `interrupt-controller` property referring to the interrupt controller described by the node `/mpic@0x3fffdd08400`. In the output tree, the value of the referenced node's phandle is included in the property. If that node doesn't have an explicit phandle property, `dtc` will automatically create a unique phandle for it. This approach makes it easy to create interrupt trees without having to explicitly assign and remember phandles for the various interrupt controller nodes.

The `dtc` source can also include "labels", which are placed on a particular node or property. For example, Figure ?? has a label "`memreg`" on the `reg` property of the node `/memory@0`. When using assembler output, corresponding labels in the output are generated, which will assemble into symbols addressing the part of the blob with the node or property in question. This is useful for the common case where an embedded board has an essentially fixed device tree with a few variable properties, such as the size of memory. The bootloader for such a board can have a device tree linked in, including a symbol referring to the right place in the blob to update the parameter with the correct value determined at runtime.

3.3 Tree checking

Between reading in the device tree and writing it out in the new format, `dtc` performs a number of checks on the tree:

- *syntactic structure*: `dtc` checks that node and property names contain only allowed characters and meet length restrictions. It checks that a node does not have multiple properties or subnodes with the same name.
- *semantic structure*: In some cases, `dtc` checks that properties whose contents are defined by convention have appropriate values. For example, it checks that `reg` properties have a length that makes sense given the address forms specified by the `#address-cells` and `#size-cells` properties. It checks that properties such as `interrupt-parent` contain a valid phandle.
- *Linux requirements*: `dtc` checks that the device tree contains those nodes and properties that are required by the Linux kernel to boot correctly.

These checks are useful to catch simple problems with the device tree, rather than having to debug the results on an embedded kernel. With the blob input mode, it can also be used for diagnosing problems with an existing blob.

4 Future Work

4.1 Board ports

The flattened device tree has always been the only supported way to boot a `ppc64` kernel on an embedded system. With the merge of `ppc32` and `ppc64` code it has also become the only supported way to boot any merged `powerpc` kernel, 32-bit or 64-bit. In fact, the old `ppc` architecture exists mainly just to support the old ppc32 embedded ports that have not been migrated to the flattened device tree approach. We plan to remove the `ppc` architecture eventually, which will mean porting all the various embedded boards to use the flattened device tree.

4.2 `dtc` features

While it is already quite usable, there are a number of extra features that `dtc` could include to make creating device trees more convenient:

- *better tree checking*: Although `dtc` already performs a number of checks on the device tree, they are rather haphazard. In many cases `dtc` will give up after detecting a minor error early and won't pick up more interesting errors later on. There is a `-f` parameter that forces `dtc` to generate an output tree even if there are errors. At present, this needs to be used more often than one might hope, because `dtc` is bad at deciding which errors should really be fatal, and which rate mere warnings.
- *binary include*: Occasionally, it is useful for the device tree to incorporate as a property a block of binary data for some board-specific purpose. For example, many of Apple's device trees incorporate bytecode drivers for certain platform devices. `dtc`'s source format ought to allow this by letting a property's value be read directly from a binary file.
- *macros*: it might be useful for `dtc` to implement some sort of macros so that a tree containing a number of similar devices (for example, multiple identical ethernet controllers or PCI buses) can be written more quickly. At present, this can be accomplished in part by running the source file through CPP before compiling with `dtc`. It's not clear whether "native" support for macros would be more useful.

About the authors

David Gibson has been a member of the IBM Linux Technology Center, working from Canberra, Australia, since 2001. Recently he has worked on Linux hugepage support and performance counter support for ppc64, as well as the device tree compiler. In the past, he has worked on bringup for various ppc and ppc64 embedded systems, the orinoco wireless driver, ramfs, and a userspace checkpointing system (`esky`).

Benjamin Herrenschmidt was a MacOS developer for about 10 years, but ultimately saw the light and installed Linux on his Apple PowerPC machine. After writing a bootloader, BootX, for it in 1998, he started contributing to the PowerPC

Linux port in various areas, mostly around the support for Apple machines. He became official PowerMac maintainer in 2001. In 2003, he joined the IBM Linux Technology Center in Canberra, Australia, where he ported the 64 bit PowerPC kernel to Apple G5 machines and the Maple embedded board, among others things. He's a member of the ppc64 development "team" and one of his current goals is to make the integration of embedded platforms smoother and more maintainable than in the 32-bit PowerPC kernel.

Legal Statement