Booting the Linux kernel

Table of Contents
=================

I – Introduction

During the development of the Linux kernel, and more specifically, the addition of new platform types outside of the old IBM pSeries/iSeries pair, it was decided to enforce some strict rules regarding the kernel entry and bootloader <-> kernel interfaces, in order to avoid the degeneration that had become the ppc32 kernel entry point and the way a new platform should be added to the kernel. The legacy iSeries platform breaks those rules as it predates this scheme, but no new board support will be accepted in the main tree that doesn't follow them properly. In addition, since the advent of the arch/powerpc merged architecture for ppc32 and ppc64, new 32-bit platforms and 32-bit platforms which move into arch/powerpc will be required to use these rules as well.

The main requirement that will be defined in more detail below is the presence of a device-tree whose format is defined after Open Firmware specification. However, in order to make life easier to embedded board vendors, the kernel doesn't require the device-tree to represent every device in the system and only requires some nodes and properties to be present. This will be described in detail in section III, but, for example, the kernel does not require you to create a node for every PCI device in the system. It is a requirement to have a node for PCI host bridges in order to provide interrupt routing information and memory/IO ranges, among others. It is also recommended to define nodes for on chip devices and other buses that don't specifically fit in an existing OF specification. This creates a great flexibility in the way the kernel can then probe those and match drivers to device, without having to hard code all sorts of tables. It also makes it more flexible for board vendors to do minor hardware upgrades without significantly impacting the kernel code or cluttering it with special cases.

1) Entry point for arch/arm

There is one single entry point to the kernel, at the start of the kernel image. That entry point supports two calling conventions. A summary of the interface is described here. A full description of the boot requirements is documented in linux source code-->Documentation/arm/Booting

a) ATAGS interface. Minimal information is passed from firmware to the kernel with a tagged list of predefined parameters.

r0 : 0

r1 : Machine type number

r2 : Physical address of tagged list in system RAM

b) Entry with a flattened device-tree block. Firmware loads the physical address of the flattened device tree block (dtb) into r2, r1 is not used, but it is considered good practice to use a valid machine number as described in linux source code-->Documentation/arm/Booting.

r0 : 0

r1 : Valid machine type number. When using a device tree, a single machine type number will often be assigned to represent a class or family of SoCs.

r2 : physical pointer to the device-tree block (defined in chapter II) in RAM. Device tree can be located anywhere in system RAM, but it should be aligned on a 64 bit boundary.

The kernel will differentiate between ATAGS and device tree booting by reading the memory pointed to by r2 and looking for either the flattened device tree block magic value (0xd00dfeed) or the ATAG_CORE value at offset 0x4 from r2 (0x54410001).

II - The DT block format

This chapter defines the actual format of the flattened device-tree passed to the kernel. The actual content of it and kernel requirements are described later. You can find example of code manipulating that format in various places, including arch/powerpc/kernel/prom_init.c which will generate a flattened device-tree from the Open Firmware representation, or the fs2dt utility which is part of the kexec tools which will generate one from a filesystem representation. It is expected that a bootloader like uboot provides a bit more support, that will be discussed later as well.

Note: The block has to be in main memory. It has to be accessible in both real mode and virtual mode with no mapping other than main memory. If you are writing a simple flash bootloader, it should copy the block to RAM before passing it to the kernel.

1) Header

The kernel is passed the physical address pointing to an area of memory that is roughly described in include/linux/of_fdt.h by the structure boot_param_header:

```
struct boot_param_header {
        u32     magic;              /* magic word OF_DT_HEADER */
        u32     totalsize;          /* total size of DT block */
        u32     off_dt_struct;      /* offset to structure */
        u32     off_dt_strings;     /* offset to strings */
        u32     off_mem_rsvmap;     /* offset to memory reserve map */
        u32     version;            /* format version */
        u32     last_comp_version;  /* last compatible version */  /* version 2 fields below */
        u32     boot_cpuid_phys;    /* Which physical CPU id we're booting on */
                                    /* version 3 fields below */
        u32     size_dt_strings;    /* size of the strings block */
                                    /* version 17 fields below */
        u32     size_dt_struct;     /* size of the DT structure block */
};
```

Along with the constants:

```
/* Definitions used by the flattened device tree */
#define OF_DT_HEADER        0xd00dfeed    /* 4: version, 4: total size */
#define OF_DT_BEGIN_NODE    0x1           /* Start node: full name*/
#define OF_DT_END_NODE      0x2           /* End node */
#define OF_DT_PROP          0x3           /* Property: name off,size, content */
#define OF_DT_END           0x9
```

All values in this header are in big endian format, the various fields in this header are defined more precisely below. All "offset" values are in bytes from the start of the header; that is from the physical base address of the device tree block.

- magic

This is a magic value that "marks" the beginning of the device-tree block header. It contains the value 0xd00dfeed and is defined by the constant OF_DT_HEADER

- totalsize

This is the total size of the DT block including the header. The "DT" block should enclose all data structures defined in this chapter (who are pointed to by offsets in this header). That is, the device-tree structure, strings, and the memory reserve map.

- off_dt_struct

This is an offset from the beginning of the header to the start of the "structure" part the device tree. (see 2) device tree)

- off_dt_strings

This is an offset from the beginning of the header to the start of the "strings" part of the device-tree

- off_mem_rsvmap

This is an offset from the beginning of the header to the start of the reserved memory map. This map is a list of pairs of 64-bit integers. Each pair is a physical address and a size. The list is terminated by an entry of size 0. This map provides the kernel with a list of physical memory areas that are "reserved" and thus not to be used for memory allocations, especially during early initialization. The kernel needs to allocate memory during boot for things like un-flattening the device-tree, allocating an MMU hash table, etc... Those allocations must be done in such a way to avoid overriding critical things like, on Open Firmware capable machines, the RTAS instance, or on some pSeries, the TCE tables used for the iommu. Typically, the reserve map should contain _at least_ this DT block itself (header,total_size). If you are passing an initrd to the kernel, you should reserve it as well. You do not need to reserve the kernel image itself. The map should be 64-bit aligned.

- version

This is the version of this structure. Version 1 stops here. Version 2 adds an additional field boot_cpuid_phys. Version 3 adds the size of the strings block, allowing the kernel to reallocate it easily at boot and free up the unused flattened structure after expansion. Version 16 introduces a new more "compact" format for the tree itself that is however not backward compatible. Version 17 adds an additional field, size_dt_struct, allowing it to be reallocated or moved more easily (this is particularly useful for bootloaders which need to make adjustments to a device tree based on probed information). You should always generate a structure of the highest version defined at the time of your implementation. Currently that is version 17, unless you explicitly aim at being backward compatible.

- last_comp_version

Last compatible version. This indicates down to what version of the DT block you are backward compatible. For example, version 2 is backward compatible with version 1 (that is, a kernel build for version 1 will be able to boot with a version 2 format). You should put a 1 in this field if you generate a device tree of version 1 to 3, or 16 if you generate a tree of version 16 or 17 using the new unit name format.

- boot_cpuid_phys

This field only exist on version 2 headers. It indicate which physical CPU ID is calling the kernel entry point. This is used, among others, by kexec. If you are on an SMP system, this value should match the content of the "reg" property of the CPU node in the device-tree corresponding to the CPU calling the kernel entry point (see further chapters for more information on the required device-tree contents)

- size_dt_strings

This field only exists on version 3 and later headers. It gives the size of the "strings" section of the device tree (which starts at the offset given by off_dt_strings).

- size_dt_struct

This field only exists on version 17 and later headers. It gives the size of the "structure" section of the device tree (which starts at the offset given by off_dt_struct).

So the typical layout of a DT block (though the various parts don't need to be in that order) looks like this (addresses go from top to bottom):

```
             ------------------------------
base -> |  struct boot_param_header  |
             ------------------------------
        |      (alignment gap) (*)       |
             ------------------------------
        |      memory reserve map       |
             ------------------------------
        |      (alignment gap)           |
             ------------------------------
        |                                |
        |    device-tree structure      |
        |                                |
             ------------------------------
        |      (alignment gap)           |
             ------------------------------
        |                                |
        |     device-tree strings       |
        |                                |
  -----> ------------------------------
  |
  |
  --- (base + totalsize)
```

(*) The alignment gaps are not necessarily present; their presence and size are dependent on the various alignment requirements of the individual data blocks.

2) Device tree generalities

This device-tree itself is separated in two different blocks, a structure block and a strings block. Both need to be aligned to a 4 byte boundary.

First, let's quickly describe the device-tree concept before detailing the storage format. This chapter does _not_ describe the detail of the required types of nodes & properties for the kernel, this is done later in chapter III.

The device-tree layout is strongly inherited from the definition of the Open Firmware IEEE 1275 device-tree. It's basically a tree of nodes, each node having two or more named properties. A property can have a value or not.

It is a tree, so each node has one and only one parent except for the root node who has no parent.

A node has 2 names. The actual node name is generally contained in a property of type "name" in the node property list whose value is a zero terminated string and is mandatory for version 1 to 3 of the format definition (as it is in Open Firmware). Version 16 makes it optional as it can generate it from the unit name defined below.

There is also a "unit name" that is used to differentiate nodes with the same name at the same level, it is usually made of the node names, the "@" sign, and a "unit address", which definition is specific to the bus type the node sits on.

The unit name doesn't exist as a property per-se but is included in the device-tree structure. It is typically used to represent "path" in the device-tree. More details about the actual format of these will be below.

The kernel generic code does not make any formal use of the unit address (though some board support code may do) so the only real requirement here for the unit address is to ensure uniqueness of the node unit name at a given level of the tree. Nodes with no notion of address and no possible sibling of the same name (like /memory or /cpus) may omit the unit address in the context of this specification, or use the "@0" default unit address. The unit name is used to define a node "full path", which is the concatenation of all parent node unit names separated with "/".

The root node doesn't have a defined name, and isn't required to have a name property either if you are using version 3 or earlier of the format. It also has no unit address (no @ symbol followed by a unit address). The root node unit name is thus an empty string. The full path to the root node is "/".

Every node which actually represents an actual device (that is, a node which isn't only a virtual "container" for more nodes, like "/cpus" is) is also required to have a "compatible" property indicating the specific hardware and an optional list of devices it is fully backwards compatible with.

Finally, every node that can be referenced from a property in another node is required to have either a "phandle" or a "linux,phandle" property. Real Open Firmware implementations provide a unique "phandle" value for every node that the "prom_init()" trampoline code turns into "linux,phandle" properties. However, this is made optional if the flattened device tree is used directly. An example of a node referencing another node via "phandle" is when laying out the interrupt tree which will be described in a further version of this document.

The "phandle" property is a 32-bit value that uniquely identifies a node. You are free to use whatever values or system of values, internal pointers, or whatever to generate these, the only requirement is that every node for which you provide that property has a unique value for it.

Here is an example of a simple device-tree. In this example, an "o" designates a node followed by the node unit name. Properties are presented with their name followed by their content. "content" represents an ASCII string (zero terminated) value, while <content> represents a 32-bit value, specified in decimal or hexadecimal (the latter prefixed 0x). The various nodes in this example will be discussed in a later chapter. At this point, it is only meant to give you a idea of what a device-tree looks like. I have purposefully kept the "name" and "linux,phandle" properties which aren't necessary in order to give you a better idea of what the tree looks like in practice.

```
/ o device-tree
    |- name = "device-tree"
    |- model = "MyBoardName"
    |- compatible = "MyBoardFamilyName"
    |- #address-cells = <2>
    |- #size-cells = <2>
    |- linux,phandle = <0>
    |
    o cpus
    | | - name = "cpus"
    | | - linux,phandle = <1>
    | | - #address-cells = <1>
    | | - #size-cells = <0>
    | |
    | o PowerPC,970@0
    |   |- name = "PowerPC,970"
    |   |- device_type = "cpu"
    |   |- reg = <0>
    |   |- clock-frequency = <0x5f5e1000>
    |   |- 64-bit
    |   |- linux,phandle = <2>
    |
    o memory@0
    | |- name = "memory"
    | |- device_type = "memory"
    | |- reg = <0x00000000 0x00000000 0x00000000 0x20000000>
    | |- linux,phandle = <3>
    |
    o chosen
      |- name = "chosen"
      |- bootargs = "root=/dev/sda2"
      |- linux,phandle = <4>
```

This tree is almost a minimal tree. It pretty much contains the minimal set of required nodes and properties to boot a linux kernel; that is, some basic model information at the root, the CPUs, and the physical memory layout. It also includes misc information passed through /chosen, like in this example, the platform type (mandatory) and the kernel command line arguments (optional).

The /cpus/PowerPC,970@0/64-bit property is an example of a property without a value. All other properties have a value. The significance of the #address-cells and #size-cells properties will be explained in chapter IV which defines precisely the required nodes and properties and their content.

3) Device tree "structure" block

The structure of the device tree is a linearized tree structure. The "OF_DT_BEGIN_NODE" token starts a new node, and the "OF_DT_END_NODE" ends that node definition. Child nodes are simply defined before "OF_DT_END_NODE" (that is nodes within the node). A 'token' is a 32 bit value. The tree has to be "finished" with a OF_DT_END token

Here's the basic structure of a single node:

* token OF_DT_BEGIN_NODE (that is 0x00000001)
* for version 1 to 3, this is the node full path as a zero terminated string, starting with "/". For version 16 and later, this is the node unit name only (or an empty string for the root node)
* [align gap to next 4 bytes boundary]
* for each property:
    * token OF_DT_PROP (that is 0x00000003)
    * 32-bit value of property value size in bytes (or 0 if no value)
    * 32-bit value of offset in string block of property name
    * property value data if any
    * [align gap to next 4 bytes boundary]
* [child nodes if any]
* token OF_DT_END_NODE (that is 0x00000002)

So the node content can be summarized as a start token, a full path, list of properties, a list of child nodes, and an end token. Every child node is a full node structure itself as defined above.

NOTE: The above definition requires that all property definitions for a particular node MUST precede any subnode definitions for that node. Although the structure would not be ambiguous if properties and subnodes were intermingled, the kernel parser requires that the properties come first (up until at least 2.6.22). Any tools manipulating a flattened tree must take care to preserve this constraint.

4) Device tree "strings" block

In order to save space, property names, which are generally redundant, are stored separately in the "strings" block. This block is simply the whole bunch of zero terminated strings for all property names concatenated together. The device-tree property definitions in the structure block will contain offset values from the beginning of the strings block.

III - Required content of the device tree

WARNING: All "linux,*" properties defined in this document apply only to a flattened device-tree. If your platform uses a real implementation of Open Firmware or an implementation compatible with the Open Firmware client interface, those properties will be created by the trampoline code in the kernel's prom_init() file. For example, that's where you'll have to add code to detect your board model and set the platform number. However, when using the flattened device-tree entry point, there is no prom_init() pass, and thus you have to provide those properties yourself.

1) Note about cells and address representation

The general rule is documented in the various Open Firmware documentations. If you choose to describe a bus with the device-tree and there exist an OF bus binding, then you should follow the specification. However, the kernel does not require every single device or bus to be described by the device tree.

In general, the format of an address for a device is defined by the parent bus type, based on the #address-cells and #size-cells properties. Note that the parent's parent definitions of #address-cells and #size-cells are not inherited so every node with children must specify them. The kernel requires the root node to have those properties defining addresses format for devices directly mapped on the processor bus.

Those 2 properties define 'cells' for representing an address and a size. A "cell" is a 32-bit number. For example, if both contain 2 like the example tree given above, then an address and a size are both composed of 2 cells, and each is a 64-bit number (cells are concatenated and expected to be in big endian format). Another example is the way Apple firmware defines them, with 2 cells for an address and one cell for a size. Most 32-bit implementations should define #address-cells and #size-cells to 1, which represents a 32-bit value. Some 32-bit processors allow for physical addresses greater than 32 bits; these processors should define #address-cells as 2.

"reg" properties are always a tuple of the type "address size" where the number of cells of address and size is specified by the bus #address-cells and #size-cells. When a bus supports various address spaces and other flags relative to a given address allocation (like prefetchable, etc...) those flags are usually added to the top level bits of the physical address. For example, a PCI physical address is made of 3 cells, the bottom two containing the actual address itself while the top cell contains address space indication, flags, and pci bus & device numbers.

For buses that support dynamic allocation, it's the accepted practice to then not provide the address in "reg" (keep it 0) though while providing a flag indicating the address is dynamically allocated, and then, to provide a separate "assigned-addresses" property that contains the fully allocated addresses. See the PCI OF bindings for details.

In general, a simple bus with no address space bits and no dynamic allocation is preferred if it reflects your hardware, as the existing kernel address parsing functions will work out of the box. If you define a bus type with a more complex address format, including things like address space bits, you'll have to add a bus translator to the prom_parse.c file of the recent kernels for your bus type.

The "reg" property only defines addresses and sizes (if #size-cells is non-0) within a given bus. In order to translate addresses upward (that is into parent bus addresses, and possibly into CPU physical addresses), all buses must contain a "ranges" property. If the "ranges" property is missing at a given level, it's assumed that translation isn't possible, i.e., the registers are not visible on the parent bus. The format of the "ranges" property for a bus is a listof:

bus address, parent bus address, size

"bus address" is in the format of the bus this bus node is defining, that is, for a PCI bridge, it would be a PCI address. Thus, (bus address, size) defines a range of addresses for child devices. "parent bus address" is in the format of the parent bus of this bus. For example, for a PCI host controller, that would be a CPU address. For a PCI<->ISA bridge, that would be a PCI address. It defines the base address in the parent bus where the beginning of that range is mapped.

For new 64-bit board support, I recommend either the 2/2 format or Apple's 2/1 format which is slightly more compact since sizes usually fit in a single 32-bit word. New 32-bit board support should use a 1/1 format, unless the processor supports physical addresses greater than 32-bits, in which case a 2/1 format is recommended.

Alternatively, the "ranges" property may be empty, indicating that the registers are visible on the parent bus using an identity mapping translation. In other words, the parent bus address space is the same as the child bus address space.

2) Note about "compatible" properties

These properties are optional, but recommended in devices and the root node. The format of a "compatible" property is a list of concatenated zero terminated strings. They allow a device to express its compatibility with a family of similar devices, in some cases, allowing a single driver to match against several devices regardless of their actual names.

3) Note about "name" properties

While earlier users of Open Firmware like OldWorld macintoshes tended to use the actual device name for the "name" property, it's nowadays considered a good practice to use a name that is closer to the device class (often equal to device_type). For example, nowadays, Ethernet controllers are named "ethernet", an additional "model" property defining precisely the chip type/model, and "compatible" property defining the family in case a single driver can driver more than one of these chips. However, the kernel doesn't generally put any restriction on the "name" property; it is simply considered good practice to follow the standard and its evolutions as closely as possible.

Note also that the new format version 16 makes the "name" property optional. If it's absent for a node, then the node's unit name is then used to reconstruct the name. That is, the part of the unit name before the "@" sign is used (or the entire unit name if no "@" sign is present).

4) Note about node and property names and character set

While Open Firmware provides more flexible usage of 8859-1, this specification enforces more strict rules. Nodes and properties should be comprised only of ASCII characters 'a' to 'z', '0' to '9', ',', '.', '_', '+', '#', '?', and '-'. Node names additionally allow uppercase characters 'A' to 'Z' (property names should be lowercase. The fact that vendors like Apple don't respect this rule is irrelevant here). Additionally, node and property names should always begin with a character in the range 'a' to 'z' (or 'A' to 'Z' for node names).

The maximum number of characters for both nodes and property names is 31. In the case of node names, this is only the leftmost part of a unit name (the pure "name" property), it doesn't include the unit address which can extend beyond that limit.

5) Required nodes and properties

These are all that are currently required. However, it is strongly ecommended that you expose PCI host bridges as documented in the PCI binding to Open Firmware, and your interrupt tree as documented in OF interrupt tree specification.

a) The root node

The root node requires some properties to be present:

- model : this is your board name/model
- #address-cells : address representation for "root" devices
- #size-cells: the size representation for "root" devices
- compatible : the board "family" generally finds its way here, for example, if you have 2 board models with a similar layout, that typically get driven by the same platform code in the kernel, you would specify the exact board model in the compatible property followed by an ntry that represents the SoC model.

The root node is also generally where you add additional properties specific to your board like the serial number if any, that sort of thing. It is recommended that if you add any "custom" property whose name may clash with standard defined ones, you prefix them with your vendor name and a comma.

Additional properties for the root node:

- serial-number : a string representing the device's serial number

b) The /cpus node

This node is the parent of all individual CPU nodes. It doesn't have any specific requirements, though it's generally good practice to have at least:

```
#address-cells = <00000001>
#size-cells    = <00000000>
```

This defines that the "address" for a CPU is a single cell, and has no meaningful size. This is not necessary but the kernel will assume that format when reading the "reg" properties of a CPU node, see below

c) The /cpus/* nodes

So under /cpus, you are supposed to create a node for every CPU on the machine. There is no specific restriction on the name of the CPU, though it's common to call it <architecture>,<core>. For example, Apple uses PowerPC,G5 while IBM uses PowerPC,970FX.However, the Generic Names convention suggests that it would be better to simply use 'cpu' for each cpu node and use the compatible property to identify the specific cpu core.

Required properties:

- device_type : has to be "cpu"
- reg : This is the physical CPU number, it's a single 32-bit cell and is also used as-is as the unit number for constructing the unit name in the full path. For example, with 2 CPUs, you would have the full path:

      /cpus/PowerPC,970FX@0
      /cpus/PowerPC,970FX@1
      (unit addresses do not require leading zeroes)
- d-cache-block-size : one cell, L1 data cache block size in bytes (*)
- i-cache-block-size : one cell, L1 instruction cache block size in bytes
- d-cache-size : one cell, size of L1 data cache in bytes
- i-cache-size : one cell, size of L1 instruction cache in bytes

(*) The cache "block" size is the size on which the cache management instructions operate. Historically, this document used the cache "line" size here which is incorrect. The kernel will prefer the cache block size and will fallback to cache line size for backward compatibility.

Recommended properties:

- timebase-frequency : a cell indicating the frequency of the timebase in Hz. This is not directly used by the generic code, but you are welcome to copy/paste the pSeries code for setting the kernel timebase/decrementer calibration based on this value.
- clock-frequency : a cell indicating the CPU core clock frequency in Hz. A new property will be defined for 64-bit values, but if your frequency is < 4Ghz, one cell is enough. Here as well as for the above, the common code doesn't use that property, but you are welcome to re-use t he pSeries or Maple one. A future kernel version might provide a common function for this.
- d-cache-line-size : one cell, L1 data cache line size in bytes if different from the block size
- i-cache-line-size : one cell, L1 instruction cache line size in bytes if different from the block size

You are welcome to add any property you find relevant to your board, like some information about the mechanism used to soft-reset the CPUs. For example, Apple puts the GPIO number for CPU soft reset lines in there as a "soft-reset" property since they start secondary CPUs by soft-resetting them.

d) the /memory node(s)

To define the physical memory layout of your board, you should create one or more memory node(s). You can either create a single node with all memory ranges in its reg property, or you can create several nodes, as you wish. The unit address (@ part) used for the full path is the address of the first range of memory defined by a given node. If you use a single memory node, this will typically be @0.

Required properties:

- device_type : has to be "memory"
- reg : This property contains all the physical memory ranges of your board. It's a list of addresses/sizes concatenated together, with the number of cells of each defined by the #address-cells and #size-cells of the root node. For example, with both of these properties being 2 like in the example given earlier, a 970 based machine with 6Gb of RAM could typically have a "reg" property here that looks like:

```
00000000 00000000 00000000 80000000
00000001 00000000 00000001 00000000
```

That is a range starting at 0 of 0x80000000 bytes and a range starting at 0x100000000 and of 0x100000000 bytes. You can see that there is no memory covering the IO hole between 2Gb and 4Gb. Some vendors prefer splitting those ranges into smaller segments, but the kernel doesn't care.

Additional properties:

- hotpluggable : The presence of this property provides an explicit hint to the operating system that this memory may potentially be removed later. The kernel can take this into consideration when doing nonmovable allocations and when laying out memory zones.

e) The /chosen node

This node is a bit "special". Normally, that's where Open Firmware puts some variable environment information, like the arguments, or the default input/output devices.

This specification makes a few of these mandatory, but also defines some linux-specific properties that would be normally constructed by the prom_init() trampoline when booting with an OF client interface, but that you have to provide yourself when using the flattened format.

Recommended properties:

- bootargs : This zero-terminated string is passed as the kernel command line
- linux,stdout-path : This is the full path to your standard console device if any. Typically, if you have serial devices on your board, you may want to put the full path to the one set as the default console in the firmware here, for the kernel to pick it up as its own default console.

Note that u-boot creates and fills in the chosen node for platforms that use it.

(Note: a practice that is now obsolete was to include a property under /chosen called interrupt-controller which had a phandle value that pointed to the main interrupt controller)

f) the /soc<SOCname> node

This node is used to represent a system-on-a-chip (SoC) and must be present if the processor is a SoC. The top-level soc node contains information that is global to all devices on the SoC. The node name should contain a unit address for the SoC, which is the base address of the memory-mapped register set for the SoC. The name of an SoC node should start with "soc", and the remainder of the name should represent the part number for the soc. For example, the MPC8540's soc node would be called "soc8540".

Required properties:

- ranges : Should be defined as specified in 1) to describe the translation of SoC addresses for memory mapped SoC registers.
- bus-frequency: Contains the bus frequency for the SoC node. Typically, the value of this field is filled in by the boot loader.
- compatible : Exact model of the SoC

Recommended properties:

- reg : This property defines the address and size of the memory-mapped registers that are used for the SOC node itself. It does not include the child device registers - these will be defined inside each child node. The address specified in the "reg" property should match the unit address of the SOC node.
- #address-cells : Address representation for "soc" devices. The format of this field may vary depending on whether or not the device registers are memory mapped. For memory mapped registers, this field represents the number of cells needed to represent the address of the registers. For SOCs that do not use MMIO, a special address format should be defined that contains enough cells to represent the required information. See 1) above for more details on defining #address-cells.
- #size-cells : Size representation for "soc" devices
- #interrupt-cells : Defines the width of cells used to represent interrupts. Typically this value is <2>, which includes a 32-bit number that represents the interrupt number, and a 32-bit number that represents the interrupt sense and level. This field is only needed if the SOC contains an interrupt controller.

The SOC node may contain child nodes for each SOC device that the platform uses. Nodes should not be created for devices which exist on the SOC but are not used by a particular platform. See chapter VI for more information on how to specify devices that are part of a SOC.

Example SOC node for the MPC8540:

```
soc8540@e0000000 {
        #address-cells = <1>;
        #size-cells = <1>;
        #interrupt-cells = <2>;
        device_type = "soc";
        ranges = <0x00000000 0xe0000000 0x00100000>
        reg = <0xe0000000 0x00003000>;
        bus-frequency = <0>;
}
```

IV - "dtc", the device tree compiler

dtc source code can be found at
<http://git.jdl.com/gitweb/?p=dtc.git>

WARNING: This version is still in early development stage; the resulting device-tree "blobs" have not yet been validated with the kernel. The current generated block lacks a useful reserve map (it will be fixed to generate an empty one, it's up to the bootloader to fill it up) among others. The error handling needs work, bugs are lurking, etc...

dtc basically takes a device-tree in a given format and outputs a device-tree in another format. The currently supported formats are:

Input formats:

- "dtb": "blob" format, that is a flattened device-tree block with header all in a binary blob.
- "dts": "source" format. This is a text file containing a "source" for a device-tree. The format is defined later in this chapter.
- "fs" format. This is a representation equivalent to the output of /proc/device-tree, that is nodes are directories and properties are files

Output formats:

- "dtb": "blob" format
- "dts": "source" format
- "asm": assembly language file. This is a file that can be sourced by gas to generate a device-tree "blob". That file can then simply be added to your Makefile. Additionally, the assembly file exports some symbols that can be used.

The syntax of the dtc tool is

```
dtc [-I <input-format>] [-O <output-format>]
    [-o output-filename] [-V output_version] input_filename
```

The "output_version" defines what version of the "blob" format will be generated. Supported versions are 1,2,3 and 16. The default is currently version 3 but that may change in the future to version 16.

Additionally, dtc performs various sanity checks on the tree, like the uniqueness of linux, phandle properties, validity of strings, etc...

The format of the .dts "source" file is "C" like, supports C and C++ style comments.

```
/ {
}
```

The above is the "device-tree" definition. It's the only statement supported currently at the toplevel.

```
/ {
  property1 = "string_value";   /* define a property containing a 0
                                 * terminated string
                                 */

  property2 = <0x1234abcd>;     /* define a property containing a
                                 * numerical 32-bit value (hexadecimal)
                                 */

  property3 = <0x12345678 0x12345678 0xdeadbeef>;
                                /* define a property containing 3
                                 * numerical 32-bit values (cells) in
                                 * hexadecimal
                                 */
  property4 = [0x0a 0x0b 0x0c 0x0d 0xde 0xea 0xad 0xbe 0xef];
                                /* define a property whose content is
                                 * an arbitrary array of bytes
                                 */

  childnode@address {   /* define a child node named "childnode"
                                 * whose unit name is "childnode at
                                 * address"
                                 */

    childprop = "hello\n";      /* define a property "childprop" of
                                 * childnode (in this case, a string)
                                 */
  };
};
```

Nodes can contain other nodes etc... thus defining the hierarchical structure of the tree.

Strings support common escape sequences from C: "\n", "\t", "\r", "\(octal value)", "\x(hex value)".

It is also suggested that you pipe your source file through cpp (gcc preprocessor) so you can use #include's, #define for constants, etc...

Finally, various options are planned but not yet implemented, like automatic generation of phandles, labels (exported to the asm file so you can point to a property content and change it easily from whatever you link the device-tree with), label or path instead of numeric value in some cells to "point" to a node (replaced by a phandle at compile time), export of reserve map address to the asm file, ability to specify reserve map content at compile time, etc...

We may provide a .h include file with common definitions of that proves useful for some properties (like building PCI properties or interrupt maps) though it may be better to add a notion of struct definitions to the compiler...

V - Recommendations for a bootloader

Here are some various ideas/recommendations that have been proposed while all this has been defined and implemented.

- The bootloader may want to be able to use the device-tree itself and may want to manipulate it (to add/edit some properties, like physical memory size or kernel arguments). At this point, 2 choices can be made. Either the bootloader works directly on the flattened format, or the bootloader has its own internal tree representation with pointers (similar to the kernel one) and re-flattens the tree when booting the kernel. The former is a bit more difficult to edit/modify, the later requires probably a bit more code to handle the tree structure. Note that the structure format has been designed so it's relatively easy to "insert" properties or nodes or delete them by just memmoving things around. It contains no internal offsets or pointers for this purpose.

- An example of code for iterating nodes & retrieving properties directly from the flattened tree format can be found in the kernel file drivers/of/fdt.c. Look at the of_scan_flat_dt() function, its usage in early_init_devtree(), and the corresponding various early_init_dt_scan_*() callbacks. That code can be re-used in a GPL bootloader, and as the author of that code, I would be happy to discuss possible free licensing to any vendor who wishes to integrate all or part of this code into a non-GPL bootloader.

VI - System-on-a-chip devices and nodes

Many companies are now starting to develop system-on-a-chip processors, where the processor core (CPU) and many peripheral devices exist on a single piece of silicon. For these SOCs, an SOC node should be used that defines child nodes for the devices that make up the SOC. While platforms are not required to use this model in order to boot the kernel, it is highly encouraged that all SOC implementations define as complete a flat-device-tree as possible to describe the devices on the SOC. This will allow for the genericization of much of the kernel code.

1) Defining child nodes of an SOC

Each device that is part of an SOC may have its own node entry inside the SOC node. For each device that is included in the SOC, the unit address property represents the address offset for this device's memory-mapped registers in the parent's address space. The parent's address space is defined by the "ranges" property in the top-level soc node. The "reg" property for each node that exists directly under the SOC node should contain the address mapping from the child address space to the parent SOC address space and the size of the device's memory-mapped register file.

For many devices that may exist inside an SOC, there are predefined specifications for the format of the device tree node. All SOC child nodes should follow these specifications.

2) Representing devices without a current OF specification

Currently, there are many devices on SoCs that do not have a standard representation defined as part of the Open Firmware specifications, mainly because the boards that contain these SoCs are not currently booted using Open Firmware. Binding documentation for new devices should be added to the Documentation/devicetree/bindings directory. That directory will expand as device tree support is added to more and more SoCs.

VII - Specifying interrupt information for devices

The device tree represents the buses and devices of a hardware system in a form similar to the physical bus topology of the hardware.

In addition, a logical 'interrupt tree' exists which represents the hierarchy and routing of interrupts in the hardware.

The interrupt tree model is fully described in the document "Open Firmware Recommended Practice: Interrupt Mapping Version 0.9". The document is available at:
<http://www.devicetree.org/open-firmware/practice/>

1) interrupts property

Devices that generate interrupts to a single interrupt controller should use the conventional OF representation described in the OF interrupt mapping documentation.

Each device which generates interrupts must have an 'interrupt' property. The interrupt property value is an arbitrary number of of 'interrupt specifier' values which describe the interrupt or interrupts for the device.

The encoding of an interrupt specifier is determined by the interrupt domain in which the device is located in the interrupt tree. The root of an interrupt domain specifies in its #interrupt-cells property the number of 32-bit cells required to encode an interrupt specifier. See the OF interrupt mapping documentation for a detailed description of domains.

For example, the binding for the OpenPIC interrupt controller specifies an #interrupt-cells value of 2 to encode the interrupt number and level/sense information. All interrupt children in an OpenPIC interrupt domain use 2 cells per interrupt in their interrupts property.

The PCI bus binding specifies a #interrupt-cell value of 1 to encode which interrupt pin (INTA,INTB,INTC,INTD) is used.

2) interrupt-parent property

The interrupt-parent property is specified to define an explicit link between a device node and its interrupt parent in the interrupt tree. The value of interrupt-parent is the phandle of the parent node.

If the interrupt-parent property is not defined for a node, its interrupt parent is assumed to be an ancestor in the node's _device tree_ hierarchy.

3) OpenPIC Interrupt Controllers

OpenPIC interrupt controllers require 2 cells to encode interrupt information. The first cell defines the interrupt number. The second cell defines the sense and level information.

Sense and level information should be encoded as follows:

    0 = low to high edge sensitive type enabled

1 = active low level sensitive type enabled
2 = active high level sensitive type enabled
3 = high to low edge sensitive type enabled

4) ISA Interrupt Controllers

ISA PIC interrupt controllers require 2 cells to encode interrupt information. The first cell defines the interrupt number. The second cell defines the sense and level information.

ISA PIC interrupt controllers should adhere to the ISA PIC encodings listed below:

0 = active low level sensitive type enabled
1 = active high level sensitive type enabled
2 = high to low edge sensitive type enabled
3 = low to high edge sensitive type enabled

VIII Changes done in Linux source code

- First identified all mentioned drivers in the scope of the work as per the reference Ips. Analyzed how drivers are integreted in the linux source code with the reference spear board.
- As per our linux requirement modified the following files in the linux source code.

  - **drivers/amba/Kconfig**
    added **default y** line under the **ARM_AMBA**
  - **drivers/irqchip/Kconfig**
    added **default y** line under the **ARM_GIC**

The first point will makes us to enable the all ARM related drivers and to enable the AMBA bus also at make menuconfig time. Those enabled drivers at make menuconfig time are reflects in the .config file.

The second point will enables the GIC interrupt controller what mentioned in dts file. It will reflect in the .config file.

For make menuconfig needs to run the following command in linux source code top directory.

sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig

The above process is the configuration process all drivers needs to be select at the make menuconfig time.

1. To enable UART PL011 in kernel configuration at make menuconfig time:

```
Device Drivers >
        Character devices >
                Serial drivers >
                        <*> ARM AMBA PL011 serial port support
                         [*]  Support for console on AMBA serial port
```

2. To enable GPIO PL061 in kernel configuration at make menuconfig time:

```
Device Drivers >
        GPIO Support >
                Memory mapped GPIO drivers >
                        [*] PrimeCell PL061 GPIO support
```

3. To enable TIMER SP804 in kernel configuration at make menuconfig time:

```
Device Drivers >
        Clock Source drivers >
                [*] Support for Dual Timer SP804 module
```

4. To enable RTC PL031 in kernel configuration at make menuconfig time:

```
Device Drivers >
        Real Time Clock >
                <*> ARM AMBA PL031 RTC
```

5. To enable WATCHDOG SP805 in kernel configuration at make menuconfig time:

```
Device Drivers >
```

```
                    Watchdog Timer Support >
                            <*> ARM SP805 Watchdog
```

6. To enable SPI PL022 in kernel configuration at make menuconfig time:

```
    Device Drivers >
            SPI support >
                    <*> ARM AMBA PL022 SSP controller
```

7. To enable DMAC PL330 in kernel configuration at make menuconfig time:

```
    Device Drivers >
            DMA Engine support >
                    <*> DMA API Driver for PL330
```

8. To enable SDMMC in kernel configuration at make menuconfig time:

```
    Device Drivers >
            MMC/SD/SDIO card support >
            <*> Synopsys Designware Memory Card Interface
            <*>   Synopsys Designware MCI Support as platform device
            <*>    Synopsys Designware MCI support on PCI bus
```

9. To enable Ethernet through USB in kernel configuration at make menuconfig time:

```
    [*] Networking support
    Device Drivers >
        Network device support >
            <*> Multi-purpose USB Networking Framework
            <*>    ASIX AX88xxx Based USB 2.0 Ethernet Adapters
            <*>    ASIX AX88179/178A USB 3.0/2.0 to Gigabit Ethernet
            -*-     CDC Ethernet support (smart devices such as cable modems)
            <*>    CDC  NCM support
            <*>    NetChip 1080 based cables (Laplink, ...)
            <*>  Simple USB Network Links (CDC Ethernet subset)
             [*]      eTEK based host-to-host cables (Advance, Belkin, ...)
             [*]      Embedded ARM Linux links (iPaq, :..)
            <*> Sharp Zaurus (stock ROMs) and compatible
```

10. To enable GIC(PL390)

In Linux source code:

- **drivers/irqchip/Kconfig**
  added **default y** line under the **ARM_GIC**

This will enables the GIC interrupt controller It will reflect in the .config file by default.

11. To enable Corelink NIC 301

In Linux source code:

- **drivers/amba/Kconfig**
  added **default y** line under the **ARM_AMBA**

This will enables the ARM AMBA bus by default and it will reflect in the .config file by default.

After complete configuration needs to generate the uImage with the following commands.

To compile linux source code run the following command

```
sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

To generate uImage run the below command.

```
sudo make ARCH=arm uImage LOADADDR=00008000 CROSS_COMPILE=arm-linux-gnueabi-
```

- Next step is the writing the dts file with recpect to the given Ips by refering the shikhara_soc document. In that documents needs to follow the address mapping, interrupt mapping and needs to read the all IP related documents in order to map the registers related to shikhara.
- Address mapping and Interrupt mapping was done in the spear1340-evb.dts file

    For complete dts file please refer the spear1340-evb.dts file