Power.org™ Standard for Embedded Power Architecture™ Platform Requirements (ePAPR)

Version 1.0 – 23 July 2008

The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. Implementation of certain elements of this document may require licenses under third party intellectual property rights, including without limitation, patent rights. Power.org and its Members are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS POWER.ORG SPECIFICATION PROVIDED "AS IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
IN NO EVENT SHALL POWER.ORG OR ANY MEMBER OF POWER.ORG BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, EXEMPLARY, PUNITIVE, OR CONSEQUENTIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Questions pertaining to this document, or the terms or conditions of its provision, should be addressed to:

IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08854
Attn: Power.org Board Secretary

LICENSE INFORMATION

Table of Contents

Revision History 8
1 Introduction 9
1.1 Purpose and Scope 9
1.2 Relationship to IEEE™ 1275 11
1.3 32-bit and 64-bit Support 11
1.4 References 12
1.5 Definition of Terms 14
1.6 Organizationally Unique Identifiers 15
2 The Device Tree 16
2.1 Overview 16
2.2 Device Tree Structure and Conventions 17
2.2.1 Node Names 17
2.2.1.1 Node Name Requirements 17
2.2.2 Generic Names Recommendation 19
2.2.3 Path Names 20
2.2.4 Properties 20
2.2.4.1 Property Names 20
2.2.4.2 Property Values 21
2.3 Standard Properties 23
2.3.1 compatible 23
2.3.2 model 23
2.3.3 phandle 24
2.3.4 status 25
2.3.5 #address-cells and #size-cells 26
2.3.6 reg 27
2.3.7 virtual-reg 27
2.3.8 ranges 28
2.3.9 dma-ranges 30
2.3.10 name 31
2.3.11 device_type 31
2.4 Interrupts and Interrupt Mapping 32
2.4.1 Properties for Interrupt Generating Devices 34
2.4.1.1 interrupts 34
2.4.1.2 interrupt-parent 34
2.4.2 Properties for Interrupt Controllers 35
2.4.2.1 #interrupt-cells 35
2.4.2.2 interrupt-controller 35
2.4.3 Interrupt Nexus Properties 35
2.4.3.1 interrupt-map 35

2.4.3.2 interrupt-map-mask ... 36
2.4.3.3 #interrupts-cells ... 36
2.4.4 Interrupt Mapping Example ... 37
3 Device Node Requirements ... 39
3.1 Base Device Node Types ... 39
3.2 Root node ... 39
3.3 aliases node ... 40
3.4 Memory node ... 41
3.5 Chosen ... 42
3.6 CPUS Node Properties ... 43
3.7 CPU Node Properties ... 43
3.7.1 General Properties of CPU nodes ... 43
3.7.2 TLB Properties ... 45
3.7.3 Internal (L1) Cache Properties ... 45
3.7.4 Example ... 46
3.8 Multi-level and Shared Caches ... 47
4 Client Program Image Format ... 49
4.1 Variable Address Image Format ... 49
4.1.1 ELF Basics ... 49
4.1.2 Boot Program Requirements ... 50
4.1.2.1 Processing of PT_LOAD segments ... 50
4.1.2.2 Entry point ... 50
4.1.3 Client Program Requirements ... 51
4.2 Fixed Address Image Format ... 51
5 Client Program Boot Requirements ... 52
5.1 Boot and Secondary CPUs ... 52
5.2 Device Tree ... 52
5.3 Initial Mapped Areas ... 53
5.4 CPU Entry Point Requirements ... 54
5.4.1 Boot CPU Initial Register Values ... 54
5.4.2 I/O Devices State ... 55
5.4.3 Initial I/O Mappings (IIO) ... 55
5.4.4 Boot CPU Entry Requirements: Real Mode ... 55
5.4.5 Boot CPU Entry Requirements for IMAs: Book IIII-E ... 56
5.4.6 Secondary CPU Entry Requirements; Real Mode ... 57
5.4.7 Secondary CPU Entry Requirements for IMAs -- Book III-E ... 57
5.5 Symmetric Multiprocessing (SMP) Boot Requirements ... 58
5.5.1 Overview ... 58
5.5.2 Spin Table ... 59
5.5.2.1 Overview ... 59
5.5.2.2 Boot Program Requirements ... 60
5.5.2.3 Client Program Requirements ... 61
5.5.3 Implementation-Specific Release from Reset ... 61

5.5.4 Timebase Synchronization 61
5.6 Asymmetric Configuration Considerations 61
6 Device Bindings 62
6.1 Binding Guidelines 62
6.1.1 General Principles 62
6.1.2 Miscellaneous Properties 63
6.1.2.1 clock-frequency 63
6.1.2.2 reg-shift 63
6.2 PCI and PCI Express 64
6.2.1 Overview 64
6.2.2 PCI Address Representation 64
6.2.3 Configuration Space 65
6.2.4 I/O Space 66
6.2.5 Memory Space 67
6.2.6 Hard-decoded Spaces 68
6.2.7 Bus nodes 69
6.2.8 Child nodes 70
6.3 ISA and Legacy Devices 71
6.3.1 ISA Interrupt Controllers 71
6.4 Serial devices 72
6.4.1 Serial Class Binding 72
6.4.1.1 clock-frequency 72
6.4.1.2 current-speed 72
6.4.2 National Semiconductor 16450/16550 Compatible UART Requirements 73
6.5 Network devices 73
6.5.1 Network Class Binding 73
6.5.1.1 address-bits 73
6.5.1.2 local-mac-address 74
6.5.1.3 mac-address 74
6.5.1.4 max-frame-size 74
6.5.2 Ethernet specific considerations 74
6.5.2.1 max-speed 75
6.5.2.2 phy-connection-type 75
6.5.2.3 phy-handle 75
6.6 Device Control Register (DCR) Devices 76
6.6.1 DCR Controller Requirements 76
6.6.1.1 dcr-reg 76
6.6.1.2 dcr-parent 76
6.6.2 DCR Programmed Device Requirements 77
6.6.2.1 dcr-controller 77
6.6.2.2 dcr-access-method 77
6.7 open PIC Interrupt Controllers 78
6.8 simple-bus 78

7 Flat Device Tree Physical Structure 79
7.1 Versioning 79
7.2 Header 80
7.3 Memory Reservation Block 81
7.3.1 Purpose 81
7.3.2 Format 82
7.4 Structure Block 82
7.4.1 Lexical structure 83
7.4.2 Tree structure 85
7.5 Strings Block 85
7.6 Alignment 86
Appendix A Device Tree Source Format (version 1) 87
Appendix B1 Ebony Device Tree 89
Appendix B2 – MPC8572DS Device Tree 95

Acknowledgements

The power.org Platform Architecture Technical Subcommittee would like thank the many individuals and companies that contributed to the development this specification through writing, technical discussions and reviews.

Individuals (in alphabetical order)

Hollis Blanchard
Dan Bouvier
Josh Boyer
Becky Bruce
Dale Farnsworth
Kumar Gala
David Gibson
Ben Herrenschmidt
Olof Johansson
Ashish Kalra
Grant Likely
Jon Loeliger
Hartmut Penner
Tim Radzykewycz
Heiko Schick
Timur Tabi
John Traill
John True
Matt Tyrlik
Dave Willoughby
Scott Wood
Jimi Xenidis
Stuart Yoder

Companies

Freescale Semiconductor
IBM
Montavista
P.A. Semi
Wind River

Other Acknowledgements

Significant aspects of the ePAPR device tree are based on work done by the Open Firmware Working Group which developed bindings for IEEE-1275. We would like to acknowledge their contributions.

We would also like to acknowledge the contribution of the PowerPC Linux community that initially developed and implemented the flattened device tree concept.

Revision History

Revision	Date	Description
0.1	1/11/2007	initial outline
0.2	4/24/2007	core device tree content
0.21	5/18/2007	update following review of 0.2
0.5	10/19/2007	single and multi-cpu boot, DTB format, PCI, interrupt mapping, unit-address is a string, some formatting, added <u64> data type
0.6	11/16/2007	new content: ELF loader, virtual-reg, serial, network, simple-bus, device class conventions, device bindings updated content: L1, L2/L3, PCI
0.61	11/29/2007	restructure and flatten—no content changes
0.62	12/20/2007	many minor changes following reviews—spelling, formatting, grammar, cross-references, some minor structural changes
0.63	1/28/2008	restructure chapters 6 & 7 – collapse into 1 chapter
0.70	2/22/2008	major content edit following December reviews
0.80	3/06/2008	updates from 0.7 review—IMA changes, note section changes, miscellaneous updates/cleanup
0.81	3/14/2008	minor IMA clarifications, add ET_EXEC to ELF section
0.90	3/31/2008	technical writer edits
0.91	4/1/2008	some edits to clarify some changes made in 0.90
0.92	4/2/2008	removed stdout alias, clarified size in r7 at boot
0.93	4/4/2008	clarified timbase sync, s/release-method/enable-method/ in a couple of places, fixed some minor formatting
0.94	4/11/2008	Added acknowledgements
1.0	7/23/2008	Convert draft to approved format

1 Introduction

1.1 Purpose and Scope

To initialize and boot a computer system, various software components interact—firmware might perform low-level initialization of the system hardware before passing control to software such as an operating system, bootloader, or hypervisor. Bootloaders and hypervisors can, in turn, load and transfer control to operating systems. Standard, consistent interfaces and conventions facilitate the interactions between these software components. In this document the term *boot program* is used to generically refer to a software component that initializes the system state and executes another software component referred to as a *client program.* Examples of a boot programs include: firmware, bootloaders, and hypervisors. Examples of a client program include: bootloaders, hypervisors, operating systems, and special purpose programs.

This specification, the Embedded Power Architecture Platform Requirements (ePAPR), provides a complete boot program to client program interface definition, combined with minimum system requirements that facilitate the development of a wide variety of embedded systems based on CPUs that implement the Power architecture as defined in the Power ISA™ [1].

This specification is targeted towards the requirements of embedded systems. An embedded system typically consists of system hardware, an operating system, and application software that are custom designed to perform a fixed, specific set of tasks. This is unlike general purpose computers, which are designed to be customized by a user with a variety of software and I/O devices. Other characteristics of embedded systems can include:

- a fixed set of I/O devices, possibly highly customized for the application
- a system board optimized for size and cost
- limited user interface
- resource constraints like limited memory and limited nonvolatile storage
- real-time constraints
- use of a wide variety of operating systems, including Linux, real-time operating systems, and custom or proprietary operating systems

Organization of this Document

- Chapter 1 introduces the architecture being specified by the ePAPR.
- Chapter 2 introduces the device tree concept and describes its logical structure and standard properties.
- Chapter 3 specifies the definition of a base set of device nodes required by ePAPR-compliant device trees.
  Chapter 4 specifies the ELF client program image format.
- Chapter 5 specifies the requirements for boot programs to start client programs on single and multiple CPU systems.
- Chapter 6 describes device bindings for certain classes of devices and specific device types.
- Chapter 7 specifies the physical structure of device trees.

Conventions Used in this Document

The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (shall equals is required to).

The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals is *permitted*).

Examples of device tree constructs are frequently shown in *Device Tree Syntax* form. See *Appendix A Device Tree Source Format (version 1)* for an overview of this syntax.

1.2 Relationship to IEEE™ 1275

The ePAPR is loosely related to the IEEE 1275 Open Firmware standard—*IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices* [2].

The original IEEE 1275 specification and its derivatives such as CHRP [10] and PAPR [16] address problems of general purpose computers, such as how a single version of an operating system can work on several different computers within the same family and the problem of loading an operating system from user-installed I/O devices.

Because of the nature of embedded systems, some of these problems faced by open, general purpose computers do not apply. Notable features of the IEEE 1275 specification that are omitted from the ePAPR include:

- Plug-in device drivers
- FCode
- The programmable Open Firmware user interface based on Forth
- FCode debugging
- Operating system debugging

What *is* retained from IEEE-1275 are concepts from the device tree architecture by which a boot program can describe and communicate system hardware information to client program, thus eliminating the need for the client program to have hard-coded descriptions of system hardware.

1.3 32-bit and 64-bit Support

The ePAPR supports CPUs with both 32-bit and 64-bit addressing capabilities. Where applicable, sections of the ePAPR describe any requirements or considerations for 32-bit and 64-bit addressing.

1.4 References

1. *Power ISA™*, Version 2.06, September 29, 2006. It is available from power.org (http://power.org).

2. Boot (Initialization Configuration) Firmware: Core Requirements and Practices, 1994. This is the core standard (also known as IEEE 1275) that defines the device tree concept adopted by the ePAPR. It is available from Global Engineering (http://global.ihs.com/).

3. *PowerPC Processor Binding to IEEE 1275-1994 Standard for Boot (Initialization, Configuration) Firmware*. Version 2.1 1996. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/bindings/ppc/release/ppc-2_1.html). This document specifies the PowerPC processor specific binding to the base standard.

4. *booting-without-of.txt* (Ben Herrenschmidt, Becky Bruce, et al.). From the Linux kernel source tree (http://www.kernel.org/). Describes the device tree as used by the Linux kernel.

5. Device Trees Everywhere. By .David Gibson and Ben Herrenschmidt (http://ozlabs.org/~dgibson/home/papers/dtc-paper.pdf). An overview of the concept of the device tree and device tree compiler.

6. PCI Bus Binding to: IEEE Std 1275-1994 Standard for Boot (Initialization Configuration) Firmware, Revision 2.1. 1998. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/bindings/pci/pci2_1.pdf)

7. *Open Firmware Recommended Practice: Interrupt Mapping*, Version 0.9. 1996. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/practice/imap/imap0_9d.pdf)

8. *Open Firmware Recommended Practice: Device Support Extensions*, Version 1.0, 1997. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/practice/devicex/dse1_0a.html) This document describes the binding for various device types such as network, RTC, keyboard, sound, etc.

9. *Open Firmware Recommended Practice: Universal Serial Bus Binding to IEEE 1275*, Version 1, 1998. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/bindings/usb/usb-1_0.ps)

10. *PowerPC Microprocessor Common Hardware Reference Platform (CHRP) Binding*, Version 1.8, 1998. Published by the Open Firmware Working Group. (http://playground.sun.com/1275/bindings/chrp/chrp1_8a.ps). This document specifies the properties for Open PIC-compatible interrupt controllers.

11. *CHRP ISA Interrupt Controller Device Binding*, Unapproved Draft version 1.1, Aug 19, 1996, Published by the Open Firmware Working Group. (http://playground.sun.com/1275/bindings/devices/postscript/isa-pic-1_1d.ps)

12. *The Open Programmable Interrupt Controller (PIC) Register Interface Specification* Revision 1.2, October 1995. Advanced Micro Devices and Cyrix Corporation.

13. *PCI Local Bus Specification*, Revision 2.2. Published by the PCI Special Interest Group.

14. *PCI Express Base Specification*, Revision 1.0a. Published by the PCI Special Interest Group.

15. *PCI-Express Binding to OF*. P1275 Openboot Working Group Proposal dated 18 August 2004.

16. *Power.org Standard for Power Architecture Platform Requirements*, Published by power.org.

17. *System V Application Binary Interface*, Edition 4.1, 1997, Published by The Santa Cruz Operation, Inc.

18. *The Open Programmable Interrupt Controller (PIC) Register Interface Specification Revision 1.2*, AMD and Cyrix. October 1995.

19. *RFC 2119, Key words for use in RFCs to Indicate Requirement Levels*
http://www.ietf.org/rfc/rfc2119.txt

20. *64-bit PowerPC ELF Application Binary Interface Supplement 1.9*. By Ian Lance Taylor. 2004.

1.5 Definition of Terms

- **AMP**. Asymmetric Multiprocessing. Computer architecture where two or more CPUs are executing different tasks. Typically, an AMP system executes different operating system images on separate CPUs.
- **boot CPU**. The first CPU which a boot program directs to a client program's entry point.
- **Book III-E**. PowerPC ISA Operating Environment Architecture - Embedded Environment. Section of the PowerPC ISA defining supervisor instructions and related facilities used in embedded PowerPC processor implementations.
- **boot program**. Used to generically refer to a software component that initializes the system state and executes another software component referred to as a client program. Examples of a boot programs include: firmware, bootloaders, and hypervisors. Examples of a client program include: bootloaders, hypervisors, operating systems, and special purpose programs.
- **client program**. Program that typically contains application or operating system software.
- **Cell**. A unit of information consisting of 32 bits.
- **DMA**. Direct memory access
- **DTB**. Device tree blob. Compact binary representation of the device tree.
- **DTC**. Device tree compiler. An open source tool used to create DTB files from DTS files.
- **DTS**. Device tree syntax. A textual representation of a device tree consumed by the DTC. See Appendix A Device Tree Source Format (version 1).
- **effective address**. Memory address as computed by processor storage access or branch instruction.
- **OUI**. Organizationally unique identifier
- **physical address**. Address used by the processor to access external device, typically a memory controller. PowerPC ISA uses real address terms when referring to a physical address.
- **PowerPC ISA**. PowerPC Instruction Set Architecture.
- **interrupt specifier**. A property value that describes an interrupt. Typically information that specifies an interrupt number and sensitivity and triggering mechanism is included.
- **secondary CPU**. In a multiprocessing system any CPU that is not executing boot program at the time control is transferred from boot program to client program.
- **SMP**. Symmetric multiprocessing. A computer architecture where two or more identical CPUs can execute the same task. Typically SMP system is executing a single operating system image.
- **SOC**. System on a chip. A single computer chip integrating a CPU core as well as number of other peripherals.
- **unit address**. Part of the node name specifying node's address in the address space of the parent node.
- **quiescent CPU**. A quiescent CPU is in a state where it cannot interfere with the normal operation of other CPUs, nor can its state be affected by the normal operation of other running CPUs, except by an explicit method for enabling or reenabling the quiescent CPU.

1.6 Organizationally Unique Identifiers

Nonstandard property names and some property values specify that the name or value should specify a unique string identifying the name of the manufacturer. The recommended form of a company's name is an Organizationally Unique Identifier (OUI), which is guaranteed to be unique worldwide.

A string representing an OUI should be expressed as the string "ONNNNNN" where NNNNNN is a sequence of uppercase hexadecimal digits representing the company's 24-bit OUI assigned by the IEEE Registration Authority.

To obtain an OUI, contact:

IEEE Registration Authority
IEEE Standards Department
445 Hoes Lane
Piscataway NJ 08854
Phone: (732) 465-6481
Fax: (732) 562-1571
Email: ieee-registration-authority@ieee.org

2 The Device Tree

2.1 Overview

The ePAPR specifies a concept called a *device tree* to describe system hardware. A boot program loads a device tree into a client program's memory and passes a pointer to the device tree to the client.

This chapter describes the logical structure of the device tree and specifies a base set of properties for use in describing device nodes. Chapter 3 specifies certain device nodes required by an ePAPR-compliant device tree. Chapter 6 describes the ePAPR defined device bindings— the requirements for representing certain device types classes of devices. Chapter 7 describes the physical structure of the device tree.

A device tree is a tree data structure with nodes that describe the physical devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.

An ePAPR-compliant device tree describes device information in a system that cannot be dynamically detected by a client program. For example, the architecture of PCI enables a client to probe and detect attached devices, and thus device tree nodes describing PCI devices might not be required. However, a device node is required to describe a PCI host bridge device in the system if it cannot be detected by probing.

Example

Figure 2-1 shows an example representation of a simple device tree that is nearly complete enough to boot a simple operating system, with the platform type, CPU, and memory described. Device nodes are shown with properties and values shown beside the node.

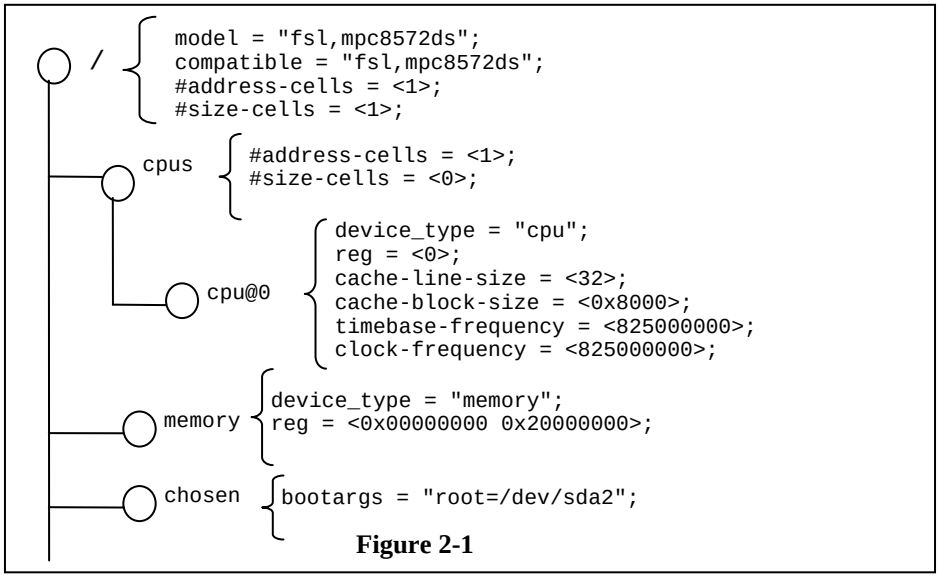


Figure 2-1

2.2 Device Tree Structure and Conventions

2.2.1 Node Names

2.2.1.1 Node Name Requirements

Each node in the device tree has a name according to the following convention:

```
name@unit-address
```

The *name* component specifies the name of the node. It shall be 1 to 31 characters in length and consist solely of characters from the set of characters in *Table 2-1*.

Table 2-1 Characters for node names

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

The *name* shall start with a lower or uppercase character. The name should describe the general class of device.

The *unit-address* component of the name is specific to the bus type on which the node sits. It consists of one or more ASCII characters from the set of characters in Table *2-1*. The fundamental requirement is that at any level of the device tree the unit-address be unique in order to differentiate nodes with the same name at the same level in the tree. The binding for a particular bus may specify additional, more specific requirements for the format of a unit-address.

The *unit-address* should match the first address specified in the `reg` property of the node. If the node has no *reg* property, the *unit-address* may be omitted if the node *name* alone differentiates the node from other nodes at the same level in the tree.

The root node does not have a *name* or *unit-address*. It is identified by a forward slash (/).

Example

See the node names examples in *Figure 2-2*.

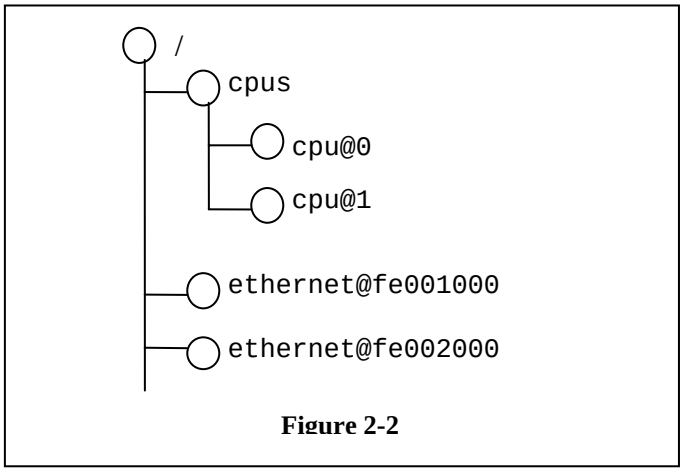


Figure 2-2

In the example:

- The nodes with the name `cpu` are distinguished by their *unit-address* values of 0 and 1.
- The nodes with the name `ethernet` are distinguished by their *unit-address* values of FE001000 and FE002000.

2.2.2 Generic Names Recommendation

The name of a node should be somewhat generic, reflecting the function of the device and not its precise programming model. If appropriate, the name should be one of the following choices:

- atm
- cache-controller
- compact-flash
- cpu
- disk
- display
- dma-controller
- ethernet
- ethernet-phy
- fdc
- flash
- gpio
- i2c
- ide
- interrupt-controller
- isa
- keyboard
- mdio
- memory
- mouse
- nvram
- pc-card
- pci
- rtc
- scsi
- serial
- sound
- spi
- timer
- usb
- vme

2.2.3 Path Names

A node in the device tree can be uniquely identified by specifying the full path from the root node, through all descendant nodes, to the desired node.

The convention for specifying a device path is:

```
/node-name-1/node-name-2/node-name-N
```

For example, in *Figure 2-2* the device path to `cpu #1` would be:

```
/cpus/cpu@1
```

The path to the root node is /.

The unit address may be omitted if the full path to the node is unambiguous.

If a client program encounters an ambiguous path, its behavior is undefined.

2.2.4 Properties

Each node in the device tree has properties that describe the characteristics of the node. Properties consist of a name and a value.

2.2.4.1 Property Names

Property names are strings of 1 to 31 characters from the following set of characters.

Table 2-2 Characters for property names

Character	Description
0-9	digit
a-z	lowercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash
?	question mark
#	hash

2.2.4.2 Property Values

A property value is an array of zero or more bytes that contain information associated with the property.

Properties might have an empty value if conveying true-false information. In this case, the presence or absence of the property is sufficiently descriptive.

Table 2-3 describes the set of basic value types defined by the ePAPR.

Table 2-3 Property values

Value	Description
<empty>	Value is empty—used for conveying true-false information, when the presence of absence of the property itself is sufficiently descriptive.

<u32>	A 32-bit integer in big-endian format. Example: the 32-bit value 0x11223344 would be represented in memory as:
	
	`address 11`
	`address+1 22`
	`address+2 33`
	`address+3 44`

<u64>	Represents a 64-bit integer in big-endian format. Consists of two <u32> values where the first value contains the most significant bits of the integer and the second value contains the least significant bits.
	
	Example: the 64-bit value 0x1122334455667788 would be represented as two cells as: <0x11223344 0x55667788>.
	
	The value would be represented in memory as:
	
	`address 11`
	`address+1 22`
	`address+2 33`
	`address+3 44`
	`address+4 55`
	`address+5 66`
	`address+6 77`
	`address+7 88`

1

<string>	Strings are printable and NULL-terminated. Example: the string “hello” would be represented in memory as: address 68 address+1 65 address+2 6C address+3 6C address+4 6F address+5 00
<prop-encoded-array>	Format is specific to the property. See the property definition.
<phandle>	A <u32> value. A *phandle* value is a way to reference another node in the device tree. Any node that can be referenced defines a *phandle* property with a unique <u32> value. That unique number is specified for the value of properties with a *phandle* value type.
<stringlist>	A list of <string> values concatenated together. Example: The string list “hello”,“world” would be represented in memory as: address 68 address+1 65 address+2 6C address+3 6C address+4 6F address+5 00 address+6 77 address+7 6F address+8 72 address+9 6C address+10 64 address+11 00

2
3

2.3 Standard Properties

The ePAPR specifies a set of standard properties for device nodes. These properties are described in detail in this section. Device nodes defined by the ePAPR (see *Chapter 3, Device Node Requirements*) may specify additional requirements or constraints regarding the use of the standard properties. Device bindings (*Chapter 6*) that describe the representation of specific devices may also specify additional requirements.

Note: All examples of device tree nodes in this document use the Device Tree Source (DTS) format for specifying nodes and properties.

2.3.1 compatible

Property: *compatible*
Value type: <*stringlist*>
Description:

The *compatible* property value consists of one or more strings that define the specific programming model for the device. This list of strings should be used by a client program for device driver selection. The property value consists of a concatenated list of null terminated strings, from most specific to most general. They allow a device to express its compatibility with a family of similar devices, potentially allowing a single device driver to match against several devices.

The recommended format is "`manufacturer,model`", where `manufacturer` is a string describing the name of the manufacturer (such as an OUI), and `model` specifies the model number.

Example:

```
compatible = "fsl,mpc8641-uart", "ns16550";
```

In this example, an operating system would first try to locate a device driver that supported fsl,mpc8641-uart. If a driver was not found, it would then try to locate a driver that supported the more general ns16550 device type.

2.3.2 model

Property: *model*
Value type: <*string*>
Description:

The model property value is a <string> that specifies the manufacturer's model number of the device.

The recommended format is: "`manufacturer,model`", where `manufacturer` is a string describing the name of the manufacturer (such as an OUI), and `model` specifies the model number.

Example:

```
model = “fsl,MPC8349EMITX”;
```

2.3.3 phandle

Property: *phandle*
Value type: <*u32*>
Description:

The *phandle* property specifies a numerical identifier for a node that is unique within the device tree. The *phandle* property value is used by other nodes that need to refer to the node associated with the property.

Example:

See the following device tree excerpt:

```
pic@10000000 {
      phandle = <1>;
      interrupt-controller;
};
```

A phandle value of 1 is defined. Another device node could reference the `pic` node with a phandle value of 1:

```
interrupt-parent = <1>;
```

Compatibility Note

Older versions of device trees may be encountered that contain a deprecated form of this property called `linux,phandle`. For compatibility, a client program might want to support `linux,phandle` if a `phandle` property is not present. The meaning and use of the two properties is identical.

Programming Note

Most device trees in *Device Tree Syntax (DTS)* (see Appendix A) will not contain explicit phandle properties. The DTC tool automatically inserts the *phandle* properties when the DTS is compiled into the binary DTB format.

2.3.4 status

Property: *status*
Value type: <*string*>
Description:

The status property indicates the operational status of a device. Valid values are listed and defined in the following table.

Table 2-4 Values for status property

Value	Description
"okay"	Indicates the device is operational
"disabled"	Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). Refer to the device binding for details on what *disabled* means for a given device.
"fail"	Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair.
"fail-sss"	Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The *sss* portion of the value is specific to the device and indicates the error condition detected.

2.3.5 #address-cells and #size-cells

Property: *#address-cells, #size-cells*
Value type: *<u32>*
Description:

The *#address-cells* and *#size-cells* properties may be used in any device node that has children in the device tree hierarchy and describe how child device nodes should be addressed. The *#address-cells* property defines the number of <u32> cells used to encode the address field in a child node's *reg* property. The *#size-cells* property defines the number of <u32> cells used to encode the size field in a child node's *reg* property.

The #address-cells and #size-cells properties are not inherited from ancestors in the device tree. They shall be explicitly defined.

An ePAPR-compliant boot program shall supply #address-cells and #size-cells on all nodes that have children.

If missing, a client program should assume a default value of 2 for *#address-cells*, and a value of 1 for *#size-cells*.

Example

See the device tree fragment shown in *Figure 2-3*.

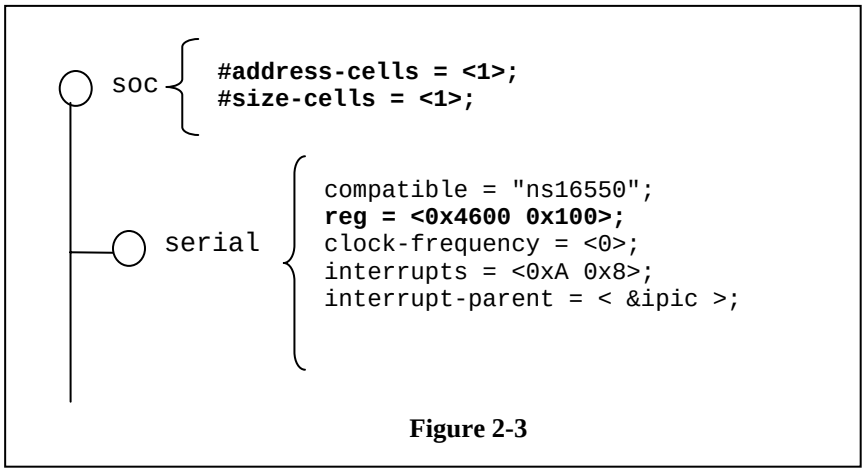


Figure 2-3

In *Figure 2-3* , the #address-cells and #size-cells properties of the `soc` node are both set to 1. This setting specifies that one cell is required to represent an address and one cell is required to represent the size of nodes that are children of this node.

The serial device *reg* property necessarily follows this specification set in the parent (`soc`) node—the address is represented by a single cell (0x4600), and the size is represented by a single cell (0x100).

2.3.6 reg

Property: *reg*
Value type: <*prop-encoded-array*> encoded as arbitrary number of (address,length) pairs.
Description:

The *reg* property describes the address and length of a device's memory mapped register space within its parent's address space. The value is a <*prop-encoded-array*>, composed of an arbitrary number of pairs of address and length, <*address*, *length*>.

The number of <u32> cells required to specify the address and length are bus-specific and are specified by the *#address-cells* and *#size-cells* properties in the parent of the device node. If the parent node specifies a value of 0 for #size-cells, the length field in the value of *reg* shall be omitted.

Example:

Suppose a device within a system-on-a-chip had two blocks of registers—a 32-byte block at offset 0x3000 in the SOC and a 256-byte block at offset 0xFE00. The *reg* property would be encoded as follows (assuming #address-cells and #size-cells values of 1):

```
reg = <0x3000 0x20 0xFE00 0x100>;
```

2.3.7 virtual-reg

Property: *virtual-reg*
Value type: <*u32*>
Description:

The *virtual-reg* property specifies an effective address that maps to the first physical address specified in the *reg* property of the device node. This property enables boot programs to provide client programs with virtual-to-physical mappings that have been set up.

2.3.8 ranges

Property: *ranges*
Value type: <*empty*> or <*prop-encoded-array*> encoded as arbitrary number of triplets of (*child-bus-address, parent-bus-address, length).*
Description:

The *ranges* property is for nodes describing the address mapping of a memory-mapped bus. It provides a means of defining a mapping or translation between the physical address space of the bus (the child address space) and the physical address space of the bus node's parent (the parent address space).

The format of the value of the ranges property is an arbitrary number of triplets of (*child-bus-address, parent-bus-address, length)*

- The *child-bus-address* is a physical address within the child bus' address space. The number of cells to represent the address is bus dependent and can be determined from the *#address-cells* of *this* node (the node in which the *ranges* property appears).

- The *parent-bus-address* is a physical address within the parent bus' address space. The number of cells to represent the parent address is bus dependent and can be determined from the *#address-cells* property of the node that defines the parent's address space.

- The *length* specifies the size of the range in the child's address space. The number of cells to represent the size can be determined from the *#size-cells* of *this* node (the node in which the *ranges* property appears).

If the property is defined with an <*empty*> value, it specifies that the parent and child address space is identical, and no address translation is required.

If the property is not present in a bus node, it is assumed that no mapping exists between children of the node and the parent address space.

See the example in Figure 2-4.

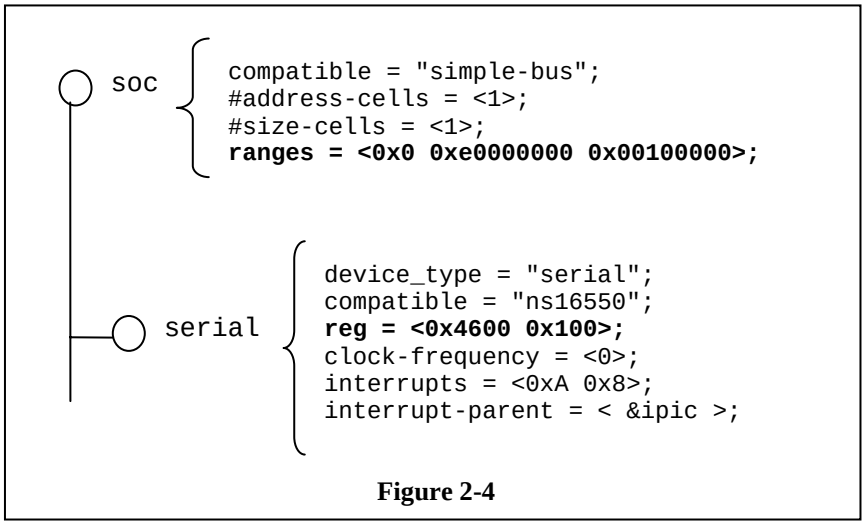


Figure 2-4

In *Figure 2-4* , the `soc` node specifies a *ranges* property of

```
<0x0 0xe0000000 0x00100000>;
```

This property value specifies that for an 1024KB range of address space, a child node addressed at physical 0x0 maps to a parent address of physical 0xe0000000. With this mapping, the `serial` device node can be addressed by a load or store at address 0xe0004600, an offset of 0x4600 (specified in *reg*) plus the 0xe0000000 mapping specified in *ranges*.

2.3.9 dma-ranges

Property: *dma-ranges*
Value type: <*empty*> or <*prop-encoded-array*> encoded as arbitrary number of triplets of (*child-bus-address*, *parent-bus-address*, *length)*.
Description:

The *dma-ranges* property is used to describe the direct memory access (DMA) structure of a memory-mapped bus whose device tree parent can be accessed from DMA operations originating from the bus. It provides a means of defining a mapping or translation between the physical address space of the bus and the physical address space of the parent of the bus.

The format of the value of the *dma-ranges* property is an arbitrary number of triplets of (*child-bus-address*, *parent-bus-address*, *length)*. Each triplet specified describes a contiguous DMA address range.

- The *child-bus-address* is a physical address within the child bus' address space. The number of cells to represent the address depends on the bus and can be determined from the *#address-cells* of *this* node (the node in which the *dma-ranges* property appears).

- The *parent-bus-address* is a physical address within the parent bus' address space. The number of cells to represent the parent address is bus dependent and can be determined from the *#address-cells* property of the node that defines the parent's address space.

- The *length* specifies the size of the range in the child's address space. The number of cells to represent the size can be determined from the *#size-cells* of *this* node (the node in which the *dma-ranges* property appears).

As a special case, a *dma-ranges* property may be present at the root of the device tree. The property describes the range of addresses that the system bus can accept as targets of a DMA operation. In this case there is no parent bus. Thus the format of the value reduces to a pair: (*child-bus-address*, *length)*.

2.3.10 name

Compatibility Note

Property: *name*
Value type: <*string*>
Description:

The *name* property is a string specifying the name of the node. This property is deprecated, and its use is not recommended. However, it might be used in older non-ePAPR-compliant device trees.

Operating system should determine a node's name based on the *name* component of the node name (see section 2.2.1).

2.3.11 device_type

Property: *device_type*
Value type: <*string*>
Description:

The *device_type* property was used in IEEE 1275 to describe the device's FCode programming model. Because ePAPR does not have FCode, new use of the property is deprecated, and it should be included only on cpu and memory nodes for compatibility with IEEE 1275–derived device trees.

2.4 Interrupts and Interrupt Mapping

The ePAPR adopts the interrupt tree model of representing interrupts specified in *Open Firmware Recommended Practice: Interrupt Mapping, Version 0.9* [**7**]. Within the device tree a logical interrupt tree exists that represents the hierarchy and routing of interrupts in the platform hardware. While generically referred to as an *interrupt tree* it is more technically a directed acyclic graph.

The physical wiring of an interrupt source to an interrupt controller is represented in the device tree with the *interrupt-parent* property. Nodes that represent interrupt-generating devices contain an *interrupt-parent* property which has a *phandle* value that points to the device to which the device's interrupts are routed, typically an interrupt controller. If an interrupt-generating device does not have an interrupt-parent property, its interrupt parent is assumed to be its device tree parent.

Each interrupt generating device contains an *interrupts* property with a value describing one or more interrupt sources for that device—each source represented with information called an *interrupt specifier*. The format and meaning of an *interrupt-specifier* is interrupt domain specific, i.e., it is dependent on the specific type of interrupt parent the interrupt is routed to. The *#interrupt-cells* property is used by an interrupt parent to define the number of <u32> values needed to encode an interrupt specifier. For example, for an Open PIC interrupt controller, an interrupt-specifer takes two 32-bit values and consists of an interrupt number and level/sense information for the interrupt.

An interrupt domain is the context in which an interrupt specifier is interpreted. The root of the domain is either (1) an interrupt controller or (2) an interrupt *nexus*.

1. An *interrupt controller* is physical device and will need a driver to handle interrupts routed through it. It may also cascade into another interrupt domain. An interrupt controller is specified by the presence of an *interrupt-controller* property on that node in the device tree.

2. An *interrupt nexus* defines a translation between one interrupt domain and another. The translation is based on both domain-specific and bus-specific information. This translation between domains is performed with the *interrupt-map* property. For example, a PCI controller device node could be an interrupt nexus that defines a translation from the PCI interrupt *namespace* (INTA, INTB, etc.) to an interrupt controller with Interrupt Request (IRQ) numbers.

The root of the interrupt tree is determined when traversal of the interrupt tree reaches an interrupt controller node without an *interrupts* property and thus no explicit interrupt parent.

See Figure 2-5 for an example of a graphical representation of a device tree with interrupt parent relationships shown. Figure 2-6 shows the corresponding interrupt tree.

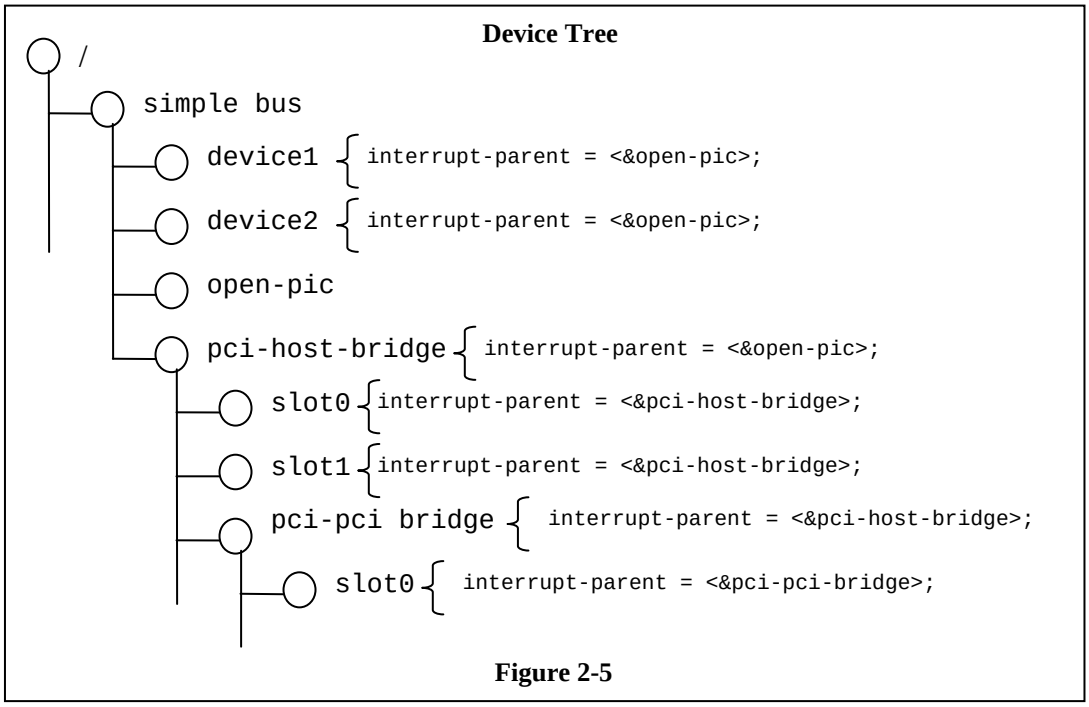


Figure 2-5

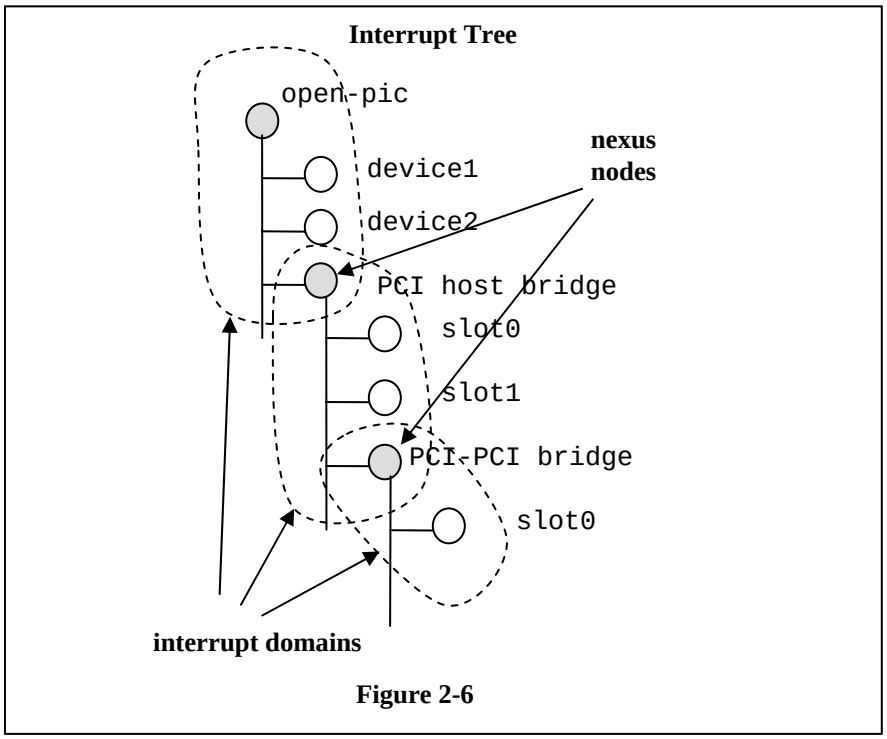


Figure 2-6

In the example shown in Figure 2-5 and Figure 2-6 :

- The `open-pic` interrupt controller is the root of the interrupt tree.
- The interrupt tree root has three children—devices that route their interrupts directly to the `open-pic`
  - `device1`
  - `device2`
  - `PCI bus controller`
- Three interrupt domains exist—one rooted at the `open-pic` node, one at the `PCI host bridge` node, and one at the `PCI-PCI bridge` node.
- There are two nexus nodes— one at the `PCI host bridge` and one at the `PCI-PCI bridge`

2.4.1 Properties for Interrupt Generating Devices

2.4.1.1 interrupts

Property: *interrupts*
Value type: <*prop-encoded-array*> encoded as arbitrary number of *interrupt specifiers*
Description:

The *interrupts* property of a device node defines the interrupt or interrupts that are generated by the device. The value of the interrupts property consists of an arbitrary number of interrupt specifiers. The format of an interrupt specifier is defined by the binding describing the node's interrupt parent.

Example:

A common definition of an interrupt specifier in an open PIC–compatible interrupt domain consists of two cells—an interrupt number and level/sense information. See the following example, which defines a single interrupt specifier, with an interrupt number of 0xA and level/sense encoding of 8.

```
interrupts = <0xA 8>;
```

2.4.1.2 interrupt-parent

Property: *interrupt-parent*
Value type: <*phandle*>
Description:

Because the hierarchy of the nodes in the interrupt tree might not match the device tree, the *interrupt-parent* property is available to make the definition of an interrupt parent explicit. The value is the *phandle* to the interrupt parent. If this property is missing from a device, its interrupt parent is assumed to be its device tree parent.

2.4.2 Properties for Interrupt Controllers

2.4.2.1 #interrupt-cells

Property: *#interrupt-cells*
Value type: <*u32*>
Description:
The *#interrupt-cells* property defines the number of cells required to encode an *interrupt specifier* for an interrupt domain.

2.4.2.2 interrupt-controller

Property: *interrupt-controller*
Value type: <*empty*>
Description:
The presence of an *interrupt-controller* property defines a node as an interrupt controller node.

2.4.3 Interrupt Nexus Properties

An interrupt nexus node shall have an *#interrupt-cells* property.

2.4.3.1 interrupt-map

Property: *interrupt-map*
Value type: <*prop-encoded-array*> encoded as an arbitrary number of interrupt mapping entries.
Description:
An *interrupt-map* is a property on a nexus node that bridges two interrupt domains and specifies how interrupt specifiers in the child domain are mapped to the parent domain.

The interrupt map is a table where each row is a mapping entry consisting of five components: *child unit address, child interrupt specifier, interrupt-parent, parent unit address, parent interrupt specifier.*

- **child unit address**. The unit address of the child node being mapped. The number of 32-bit cells required to specify this is described by the *#address-cells* property of the bus node on which the child is located.

- **child interrupt specifier**. The interrupt specifier of the child node being mapped. The number of 32-bit cells required to specify this component is described by the *#interrupt-cells* property of this node—the nexus node containing the *interrupt-map* property.

- **interrupt-parent** . A single <*phandle*> value that points to the interrupt parent to which the child domain is being mapped.

- **parent unit address**. The unit address in the domain of the interrupt parent. The number of 32-bit cells required to specify this address is described by the *#address-cells* property of the node pointed to by the interrupt-parent field.

- **parent interrupt specifier**. The interrupt specifier in the parent domain. The number of 32-bit cells required to specify this component is described by the *#interrupt-cells* property of this node—the nexus node containing the *interrupt-map* property.

Lookups are performed on the interrupt mapping table by matching a unit-address/interrupt specifier pair against the child components in the interrupt-map. Because some fields in the unit interrupt specifier may not be relevant, a mask is applied before the lookup is done. This mask is defined in the *interrupt-map-mask* property (see *section 2.4.3.2*).

Note: Both the child node and the interrupt parent node are required to have *#address-cells* and *#interrupt-cells* properties defined. If a unit address component is not required, *#address-cells* shall be explicitly defined to be zero.

2.4.3.2 interrupt-map-mask

Property: *interrupt-map-mask*
Value type: <*prop-encoded-array*> encoded as a bit mask
Description:
An *interrupt-map-mask* property is specified for a nexus node in the interrupt tree. This property specifies a mask that is applied to the incoming unit interrupt specifier being looked up in the table specified in the *interrupt-map* property.

2.4.3.3 #interrupts-cells

Property: *#interrupts-cells*
Value type: <*u32*>
Description:
The *#interrupt-cells* property defines the number of cells required to encode an *interrupt specifier* for an interrupt domain.

2.4.4 Interrupt Mapping Example

Figure 2-7 shows the representation of a fragment of a device tree with a PCI bus controller and a sample interrupt map for describing the interrupt routing for two PCI slots (IDSEL 0x11,0x12). The INTA, INTB, INTC, and INTD pins for slots 1 and 2 are wired to the Open PIC interrupt controller.

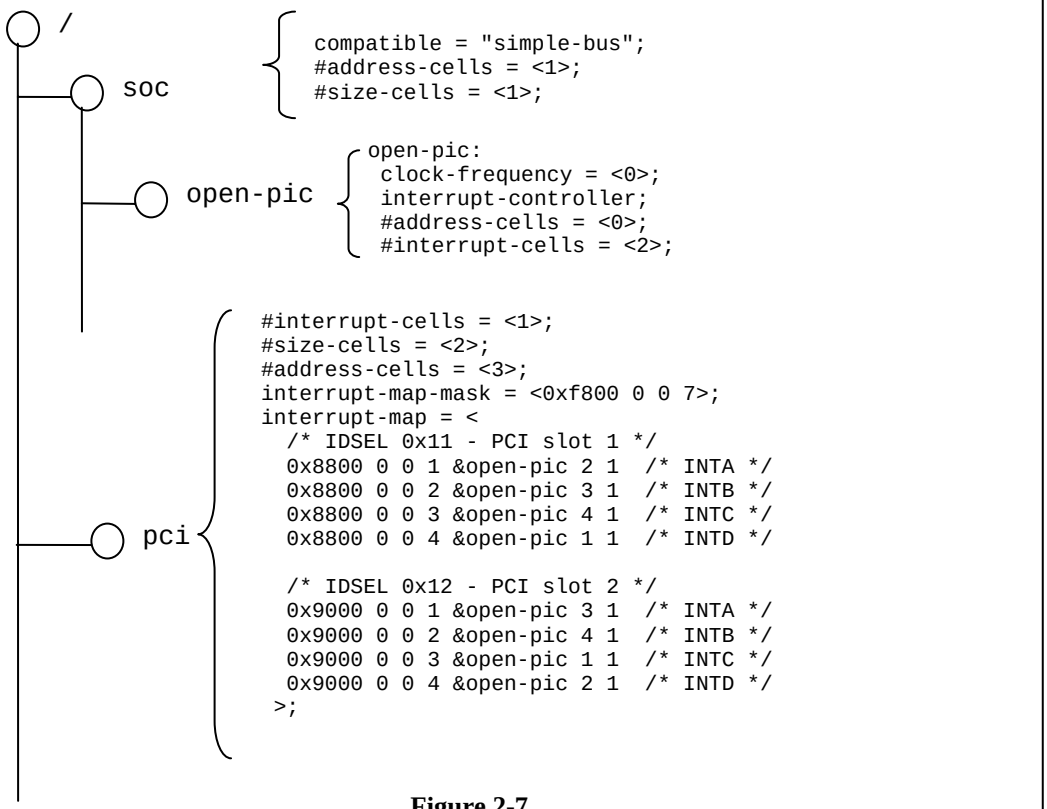
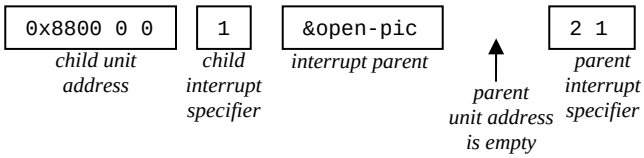


Figure 2-7

- One Open PIC interrupt controller is represented and is identified as an interrupt controller with an *interrupt-controller* property.

- Each row in the interrupt-map table consists of five parts—a child unit address and interrupt specifier, which is mapped to an *interrupt-parent* node with a specified parent unit address and interrupt specifier.

- For example, the first row of the interrupt-map table specifies the mapping for INTA of slot 1. The components of that row are shown in the following diagram.



  - The child unit address is <0x8800 0 0>. This value is encoded with three 32-bit cells, which is determined by the value of the *#address-cells* property (value of 3) of the PCI controller. The three cells represent the PCI address as described by the ePAPR binding for the PCI bus (see *section 6.2*).
    - The encoding includes the bus number (0x0 << 16), device number (0x11 << 11), and function number (0x0 << 8).
  - The child interrupt specifier is <1>, which specifies INTA as described by the ePAPR PCI binding. This takes one 32-bit cell as specified by the *#interrupt-cells* property (value of 1) of the PCI controller, which is the child interrupt domain.
  - The interrupt parent is specified by a phandle which points to the interrupt parent of the slot, the Open PIC interrupt controller.
  - The parent has no unit address because the parent interrupt domain (the open-pic node) has an *#address-cells* value of 0.
  - The parent *interrupt specifier* is <2 1>. The number of cells to represent the interrupt specifier (two cells) is determined by the *#interrupt-cells* property on the interrupt parent, the open-pic node.
    - The value <2 1> is a value specified by the device binding for the Open PIC interrupt controller (see *section 6.7*). The value <2> specifies the physical interrupt source number on the interrupt controller to which INTA is wired. The value <1> specifies the level/sense encoding.
- In this example, the *interrupt-map-mask* property has a value of <0xf800 0 0 7>. This mask is applied to a child unit interrupt specifier before performing a lookup in the interrupt-map table.
- **Example:** To perform a lookup of the open-pic interrupt source number for INTB for IDSEL 0x12 (slot 2), function 0x3, the following steps would be performed:
  - The child unit address and interrupt specifier form the value <0x9300 0 0 2>.
    - The encoding of the address includes the bus number (0x0 << 16), device number (0x12 << 11), and function number (0x3 << 8).
    - The interrupt specifier is 2, which is the encoding for INTB as per the PCI binding.
  - The *interrupt-map-mask* value <0xf800 0 0 7> is applied, giving a result of <0x9000 0 0 2>.
  - That result is looked up in the *interrupt-map* table, which maps to the parent interrupt specifier <4 1>.

3 Device Node Requirements

3.1 Base Device Node Types

The sections that follow specify the requirements for the base set of device nodes required in an ePAPR-compliant device tree.

All device trees shall have a root node and the following nodes shall be present at the root of all device trees:

- One *cpus* node
- At least one *memory* node

3.2 Root node

The device tree has a single root node of which all other device nodes are descendants. The full path to the root node is /.

Properties

Table 3-1 Root node properties

Property Name	Usage	Value Type	Definition
#address-cells	R	<u32>	Specifies the number of <u32> cells to represent the address in the *reg* property in *children* of root.
#size-cells	R	<u32>	Specifies the number of <u32> cells to represent the size in the *reg* property in *children* of root.
model	R	<string>	Specifies a string that uniquely identifies the model of the system board. The recommended format is "`manufacturer,model-number`".
compatible	R	<stringlist>	Specifies a list of platform architectures with which this platform is compatible. This property can be used by operating systems in selecting platform specific code. The recommended form of the property value is: "<Manufacturer>,<Model-number>" For example: `compatible = "fsl,mpc8572ds"`
epapr-version	R	<string>	This property shall contain the string: "ePAPR-<ePAPR version>" where: • <ePAPR version> is the text (without blanks) after the word *Version* on the cover page of the PAPR spec that the platform adheres to For example: `epapr-version = "ePAPR-1.0"`
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

3.3 aliases node

A device tree may have an aliases node (`/aliases`) that defines one or more alias properties. The alias node shall be at the root of the device tree and have the node name `aliases`.

Each property of the `/aliases` node defines an *alias*. The property name specifies the alias name. The property value specifies the full path to a node in the device tree. For example, the property `serial0 = /simple-bus@fe000000/serial@llc500` defines the alias *serial0*.

Alias names shall be a lowercase text strings of 1 to 31 characters from the following set of characters.

Table 3-2 Characters for alias names

Character	Description
`0-9`	digit
`a-z`	lowercase character
`-`	dash

An alias value is a device path and is encoded as a string. The value represents the full path to a node, but the path does not need to refer to a leaf node.

A client program or any device tree property, may use an alias property name to refer to a full device path as all or part of its string value. A client program, when considering a string as a device path, shall detect and use the alias.

Example:

```
aliases {
 serial0 = "/simple-bus@fe000000/serial@llc500";
 ethernet0 = "/simple-bus@fe000000/ethernet@31c000";
}
```

Given the alias serial0, a client program can look at the `/aliases` node and determine the alias refers to the device path /simple-bus@fe000000/serial@llc500.

3.4 Memory node

A memory device node is required for all device trees. The memory node describes the physical memory layout for the system. If a system has multiple ranges of memory, multiple memory nodes can be created, or the ranges can be specified in the *reg* property of a single memory node.

The name component of the node name (see 2.2.1) shall be `memory`.

Properties

Table 3-3 Memory node properties

Property Name	Usage	Value Type	Definition
device_type	R	<string>	Value shall be "memory".
reg	R	<prop-encoded-array>	Consists of an arbitrary number of address and size pairs that specify the physical address and size of the memory ranges.
initial-mapped-area	O	<prop-encoded-array>	Specifies the address and size of the Initial Mapped Area (see section 0) . Is a prop-encoded-array consisting of a triplet of (*effective address, physical address, size*). The effective and physical address shall each be 64-bit (<u64> value), and the size shall be 32-bits (<u32> value).
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

Example

Given a 64-bit PowerPC system with the following physical memory layout:

- RAM: starting address 0x0, length 0x80000000 (2GB)
- RAM: starting address 0x100000000, length 0x100000000 (4GB)

The memory node would be defined as follows, assuming an #address-cells value of 2 and a #size-cells value of 2:

```
memory@0 {
    device_type = "memory";
    reg = <0x000000000 0x00000000 0x00000000 0x80000000
        0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

The *reg* property is used to define the address and size of the two memory ranges. The 2 GB I/O region is skipped. Note that the *#address-cells* and *#size-cells* properties of the root node specify a value of 2, which means that two 32-bit cells are required to define the address and length for the *reg* property of the memory node.

3.5 Chosen

The *chosen* node does not represent a real device in the system but describes parameters chosen or specified by the system firmware at run time. It shall be a child of the root node.

The node name (see 2.2.1) shall be `chosen`.

Properties

Table 3-4 Chosen node properties

Property Name	Usage	Value Type	Definition
bootargs	O	<string>	A string that specifies the boot arguments for the client program. The value could potentially be a null string if no boot arguments are required.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

Example

```
chosen {
  bootargs = “root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200”;
};
```

3.6 CPUS Node Properties

A cpus device node is required for all device trees. It does not represent a real device in the system, but acts as a container for child cpu nodes which represent the systems CPUs.

The node name (see 2.2.1) shall be cpus.

Properties

Table 3-5 cpus node properties

Property Name	Usage	Value Type	Definition
#address-cells	R	<u32>	Value shall be 1. Specifies that one 32-bit address cell is required in the *reg* property in *children* of this node.
#size-cells	R	<u32>	Value shall be 0. Specifies that no size is required in the *reg* property in *children* of this node.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

For an example, see section 3.7.4.

3.7 CPU Node Properties

The cpu device node describes the CPUs or processor cores in a system. For multiple-core CPUs, a cpu node is defined for each core.

The node name (see 2.2.1) should be cpu.

3.7.1 General Properties of CPU nodes

The following table describes the general properties of CPU nodes. Some of the properties described in *Table 3-6* are select standard properties with specific applicable detail.

Table 3-6 cpu node general properties

Property Name	Usage	Value Type	Definition
device_type	R	<string>	Value shall be “cpu”.
reg	R	<u32>	A standard property. For a uniprocessor system, the value shall be zero. For a multiprocessor system, the value shall specify the CPU’s unit address.
clock-frequency	R	<prop-encoded-array>	Specifies the current clock speed of the CPU in Hertz. The value is a <prop-encoded-array> in one of two forms: 1. a 32-bit integer consisting of one <u32> specifying the frequency. 2. a 64-bit integer represented as a <u64> specifying the frequency.
timebase-frequency	R	<prop-encoded-array>	Specifies the current frequency at which the timebase and decrementer registers are updated (in Hertz). The value is a <prop-encoded-array> in one of two forms: 1. A 32-bit integer consisting of one <u32> specifying the frequency. 2. a 64-bit integer represented as a <u64>.

1

cache-op-block-size	SD	<u32>	Specifies the block size in bytes upon which cache block instructions operate (e.g., dcbz). Required if different than the L1 cache block size.
reservation-granule-size	SD	<u32>	Specifies the reservation granule size supported by this processor in bytes.
status	SD	<string>	A standard property describing the state of a CPU. This property shall be present for nodes representing CPUs in a symmetric multiprocessing (SMP) configuration. For a CPU node the meaning of the “okay” and “disabled” values are as follows: • **okay**. The CPU is running. • **disabled**. The CPU is in a quiescent state. A quiescent CPU is in a state where it cannot interfere with the normal operation of other CPUs, nor can its state be affected by the normal operation of other running CPUs, except by an explicit method for enabling or reenabling the quiescent CPU. (see the *enable-method* property). In particular ,a running CPU shall be able to issue broadcast TLB invalidates without affecting a quiescent CPU. Examples: A quiescent CPU could be in a spin loop, held in reset, and electrically isolated from the system bus or in another implementation dependent state. **Note**: See section 5.5 *(Symmetric Multiprocessing (SMP) Boot Requirements)* for a description of how these values are used for booting multi-CPU SMP systems.
enable-method	SD	<stringlist>	Describes the method by which a CPU in a disabled state is enabled. This property is required for CPUs with a status property with a value of “disabled”. The value consists of one or more strings that define the method to release this CPU. If a client program recognizes any of the methods, it may use it. The value shall be one of the following: • "spin-table" The CPU is enabled with the spin table method defined in the ePAPR. • "[vendor],[method]" An implementation-dependent string that describes the method by which a CPU is released from a "disabled" state. The required format is: vendor,method,. where vendor is a string describing the name of the manufacturer and method is a string describing the vendor-specific mechanism. Example: "fsl,MPC8572DS" **Note**: Other methods may be added to later revisions of the ePAPR specification.
cpu-release-addr	SD	<u64>	The *cpu-release-addr* property is required for cpu nodes that have an enable-method property value of "spin-table". The value specifies the *physical* address of a spin table entry that releases a secondary CPU from its spin loop. See section 5.5.2, *Spin Table* or details on the structure of a spin table.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

2

Compatibility Note

Older versions of device trees may be encountered that contain a *bus-frequency* property on CPU nodes. For compatibility, a client-program might want to support *bus-frequency*. The format of the value is identical to that of *clock-frequency*. The recommended practice is to represent the frequency of a bus on the `bus` node using a *clock-frequency* property.

3.7.2 TLB Properties

The following properties of a `cpu` node describe the translate look-aside buffer in the processor's MMU.

Table 3-7, cpu node TLB properties

Property Name	Usage	Value Type	Definition
tlb-split	SD	<empty>	If present specifies that the TLB has a split configuration, with separate TLBs for instructions and data. If absent, specifies that the TLB has a unified configuration. Required for a CPU with a TLB in a split configuration.
tlb-size	SD	<u32>	Specifies the number of entries in the TLB. Required for a CPU with a unified TLB for instruction and data addresses.
tlb-sets	SD	<u32>	Specifies the number of associativity sets in the TLB. Required for a CPU with a unified TLB for instruction and data addresses.
d-tlb-size	SD	<u32>	Specifies the number of entries in the data TLB. Required for a CPU with a split TLB configuration.
d-tlb-sets	SD	<u32>	Specifies the number of associativity sets in the data TLB. Required for a CPU with a split TLB configuration.
i-tlb-size	SD	<u32>	Specifies the number of entries in the instruction TLB. Required for a CPU with a split TLB configuration.
i-tlb-sets	SD	<u32>	Specifies the number of associativity sets in the instruction TLB. Required for a CPU with a split TLB configuration.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

3.7.3 Internal (L1) Cache Properties

The following properties of a `cpu` node describe the processor's internal (L1) cache.

Table 3-8 Cache properties

Property Name	Usage	Value Type	Definition
cache-unified	SD	<empty>	If present, specifies the cache has a unified organization. If not present, specifies that the cache has a Harvard architecture with separate caches for instructions and data.
cache-size	SD	<u32>	Specifies the size in bytes of a unified cache. Required if the cache is unified (combined instructions and data).
cache-sets	SD	<u32>	Specifies the number of associativity sets in a unified cache. Required if the cache is unified (combined instructions and data)

cache-block-size	SD	<u32>	Specifies the block size in bytes of a unified cache. Required if the processor has a unified cache (combined instructions and data)
cache-line-size	SD	<u32>	Specifies the line size in bytes of a unified cache, if different than the cache block size Required if the processor has a unified cache (combined instructions and data).
i-cache-size	SD	<u32>	Specifies the size in bytes of the instruction cache. Required if the cpu has a separate cache for instructions.
i-cache-sets	SD	<u32>	Specifies the number of associativity sets in the instruction cache. Required if the cpu has a separate cache for instructions.
i-cache-block-size	SD	<u32>	Specifies the block size in bytes of the instruction cache. Required if the cpu has a separate cache for instructions.
i-cache-line-size	SD	<u32>	Specifies the line size in bytes of the instruction cache, if different than the cache block size. Required if the cpu has a separate cache for instructions.
d-cache-size	SD	<u32>	Specifies the size in bytes of the data cache. Required if the cpu has a separate cache for data.
d-cache-sets	SD	<u32>	Specifies the number of associativity sets in the data cache. Required if the cpu has a separate cache for data.
d-cache-block-size	SD	<u32>	Specifies the block size in bytes of the data cache. Required if the cpu has a separate cache for data.
d-cache-line-size	SD	<u32>	Specifies the line size in bytes of the data cache, if different than the cache block size. Required if the cpu has a separate cache for data.
next-level-cache	SD	<phandle>	If present, indicates that another level of cache exists. The value is the phandle of the next level of cache. The phandle value type is fully described in section 2.3.3.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

Compatibility Note

Older versions of device trees may be encountered that contain a deprecated form of the *next-level-cache* property called *l2-cache*. For compatibility, a client-program may wish to support *l2-cache* if *a next-level-cache* property is not present. The meaning and use of the two properties is identical.

3.7.4 Example

Here is an example of a `cpus` node with one child `cpu` node:

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
        device_type = "cpu";
        reg = <0>;
        d-cache-block-size = <32>;     // L1 - 32 bytes
        i-cache-block-size = <32>;     // L1 - 32 bytes
        d-cache-size = <0x8000>;      // L1, 32K
        i-cache-size = <0x8000>;      // L1, 32K
        timebase-frequency = <82500000>;  // 82.5 MHz
        clock-frequency = <825000000>;   // 825 MHz
    };
};
```

3.8 Multi-level and Shared Caches

Processors and systems may implement additional levels of cache hierarchy—for example, second-level (L2) or third-level (L3) caches. These caches can potentially be tightly integrated to the CPU or possibly shared between multiple CPUs.

A device node with a device type of *cache* describes these types of caches and shall be a child of an associated "cpu" node.

For caches shared between CPUs, the cache device shall be a child of a single cpu node. The cache device node shall define a *phandle* property, and all cpu nodes that share the cache each shall contain a *next-level-cache* property that specifies a *phandle* to the cache.

Multiple-level and shared caches are represented with the properties in Table 3-9. The L1 cache properties described in Table 3-8.

Table 3-9 Multiple-level and shared cache properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	A standard property. The value shall include the string "cache"
cache-level	R	<u32>	Specifies the level in the cache hierarchy. For example, a level 2 cache has a value of <2>.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

See the following example of a device tree representation of two CPUs, each with their own on-chip L2 and a shared L3.

```
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu@0 {
            device_type = "cpu";
            reg = <0>;
            cache-unified;
            cache-size = <0x8000>;       // L1, 32KB
            cache-block-size = <32>;
            timebase-frequency = <82500000>; // 82.5 MHz
            next-level-cache = <&L2_0>;          // phandle to L2

            L2_0:l2-cache {
              compatible = “cache”;
              cache-unified;
              cache-size = <0x40000>;     // 256 KB
              cache-sets = <1024>;
              cache-block-size = <32>;
              cache-level = <2>;
              next-level-cache = <&L3>;   // phandle to L3

              L3:l3-cache {
                compatible = “cache”;
                cache-unified;
                cache-size = <0x40000>;    // 256 KB
                cache-sets = <0x400>;     // 1024
                cache-block-size = <32>
                cache-level = <3>;
              };
            };
        };

        cpu@1 {
            device_type = "cpu";
            reg = <0>;
            cache-unified;
            cache-block-size = <32>;
            cache-size = <0x8000>;       // L1, 32KB
            timebase-frequency = <82500000>; // 82.5 MHz
            clock-frequency = <825000000>;  // 825 MHz
            cache-level = <2>;
            next-level-cache = <&L2_1>;         // phandle to L2

            L2_1:l2-cache {
              compatible = “cache”;
              cache-unified;
              cache-size = <0x40000>;   // 256 KB
              cache-sets = <0x400>;    // 1024
              cache-line-size = <32>    // 32 bytes
              next-level-cache = <&L3>;  // phandle to L3
            };
        };
    };
```

4 Client Program Image Format

This section describes the image format in which an ePAPR client is encoded in order to boot it from an ePAPR-compliant boot program. Two variants on the image format are described: variable-address images and fixed-address images. ePAPR-compliant boot programs shall support client images in the variable-address format, should support images in the fixed-address format, and may also support other formats not described in this document.

4.1 Variable Address Image Format

This ePAPR image format is a constrained form of ELF (*Executable and Linking Format*, see [**17**]) executable. That is, an ePAPR client image shall be a valid ELF file, but also has additional requirements described in the next sections.

4.1.1 ELF Basics

A variable-address client image is a 32-bit ELF client image with the following ELF header field values:

```
e_ident[EI_CLASS] ELFCLASS32(0x1)
e_ident[EI_DATA]  ELFDATA2MSB(0x2)
e_type            ET_DYN(0x3)
e_machine         EM_PPC(0x14)
```

That is, it is a 32-bit PowerPC shared-object image in 2's complement, big-endian format.

Every ePAPR image shall have at least one program header of type PT_LOAD. It may also have other valid ELF program headers. The client image shall be arranged so that all its ELF program headers lie within the first 1024 bytes of the image.

4.1.2 Boot Program Requirements

When loading a client image, the boot program need only consider ELF segments of type PT_LOAD. Other segments may be present, but should be ignored by the boot program. In particular, the boot program should not process any ELF relocations found in the client image.

4.1.2.1 Processing of PT_LOAD segments

The boot program shall load the contents of any PT_LOAD segments into RAM, and then pass control to the entry point specified in the ELF header in the manner specified in section 5.4.

Each PT_LOAD segments shall be loaded at an address decided by the boot program, subject to the following constraints.

- The load address shall be congruent with the program header's `p_paddr` value; modulo, with the program header's `p_align` value.
- If there is more than one PT_LOAD segment, then the difference between the loaded address and the address specified in the `p_paddr` field shall be the same for all segments. That is, the boot program shall preserve the relative offsets between PT_LOAD segments by physical address.

The `p_vaddr` field is reserved to represent the effective address at which the segments will appear after the client program has performed MMU setup. The boot program should not use the program header's `p_vaddr` field for determining the load address of segments.

4.1.2.2 Entry point

The program entry point is the address of the first instruction that is to be executed in a program image. The ELF header `e_entry` field gives the effective address of the program entry point. However, as described in section 5.4, *CPU Entry Point Requirements*, the client program shall be entered either in real mode or with an initial MMU mapping which resembles real mode. (Effective address is equal to physical over the relevant range.)

Therefore, the boot program shall compute the physical address of the entry point before entering the client program. To perform this calculation, it shall locate the program segment containing the entry point, determine the difference between `e_entry` and the `p_vaddr` of that segment, and add this difference to the physical address where the segment was loaded.

This adjusted address will be the physical address of the first client program instruction executed after the boot program jumps to the client program.

4.1.3 Client Program Requirements

The client program is entered with MMU state as described in section 5.4, *CPU Entry Point Requirements* (i.e., with an identity mapping between effective and physical addresses). Therefore, the code at the client program's entry point shall be prepared to execute in this environment, which may be different than the MMU environment in which most of the client program executes. The `p_vaddr` fields of the client's ELF program headers will reflect this final environment, not the environment in which the entry point is executed.

The code at the entry point shall be written so that it can be executed at any address. It shall establish a suitable environment in which the remainder of the client program executes. The ePAPR does not specify its method, but the task could involve:

- Processing ELF relocations to relocate the client's own image to its loaded address. Note that in this case the client image shall be specially linked so that the ELF relocation information, plus any data required to find that information is contained in both the loaded segments and the segments and sections set aside for relocation information.
- Processing other tables of relocation information in some format specific to the client program.
- Physically copying the client image to the address at which it prefers to execute.
- Configuring the MMU so that the client image can execute at its preferred effective address, regardless of the physical address at which it is loaded.

4.2 Fixed Address Image Format

Fixed-address client images are identical to variable-address client images except for the following changes:

- The `e_type` ELF header field shall have the value ET_EXEC (0x1).
- The boot program, instead of loading each PT_LOAD segment at an address of its choosing shall load each PT_LOAD segment at the physical address given in the program header's `p_paddr` field. If it cannot load the segment at this address (because memory does not exist at that address or is already in use by the boot program itself), then it shall refuse to load the image and report an error condition.

The fixed-address image format is intended for use by very simple clients (such as diagnostic programs), avoiding the need for such clients to physically relocate themselves to a suitable address. Clients should in general avoid using the fixed-address format, because creating a usable fixed-address image requires knowing which physical areas will be available for client use on the platform in question.

5 Client Program Boot Requirements

5.1 Boot and Secondary CPUs

A *boot cpu* is the CPU on which control is transferred from the boot program to a client program. Other CPUs that belong to the client program are considered *secondary* CPUs.

For a partition with multiple CPUs in an SMP configuration, one CPU shall be designated as the boot cpu. The unit address of the CPU node for the boot cpu is set in the `boot_cpuid_phys` field of the flattened device tree header (see section 7.2, *Header*).

5.2 Device Tree

A boot program shall load a device tree image into the client program's memory before transferring control to the client on the boot cpu. The logical structure of the device tree shall comply with the requirements specified in section 3.1 (*Base Device Node Types*). The physical structure of the device tree image shall comply with the requirements specified in chapter 7 (*Flat Device Tree Physical Structure*).

The loaded device tree image shall be aligned on an 8-byte boundary in the client's memory.

5.3 Initial Mapped Areas

CPUs that implement the Power ISA Book III-E embedded environment, which run with address translation always enabled, have some unique boot requirements related to initial memory mappings. This section introduces the concept of an *Initial Mapped Area* (or *IMA*), which is applicable to Book III-E CPUs.

A client program's IMA is a region of memory that contains the entry points for a client program. Both boot CPUs and secondary CPUs begin client program execution in an IMA. The terms *Boot IMA* (BIMA) and *Secondary IMA* (SIMA) are used to distinguish the IMAs for boot CPUs and secondary CPUs where necessary.

All IMAs have the following requirements:

1. An IMA shall be virtually and physically contiguous

2. An IMA shall start at effective address zero (0) which shall be mapped to a naturally aligned physical address

3. The mapping shall not be invalidated except by a client program's explicit action (i.e., not subject to broadcast invalidates from other CPUs)

4. The Translation ID (TID) field in the TLB entry shall be zero.

5. The memory and cache access attributes (WIMGE) have the following requirements:
   - WIMG unspecified
   - E=0 (i.e., big-endian)

6. An IMA may be mapped by a TLB entry larger than the IMA size, provided the MMU guarded attribute is set (G=1)

7. An IMA may span multiple TLB entries.

Programming Note

Those CPUs with an IPROT capable TLB should use the IPROT facility to ensure requirement #3.

5.4 CPU Entry Point Requirements

This section describes the state of the processor and system when a boot program passes control to a client program.

5.4.1 Boot CPU Initial Register Values

A boot CPU shall have its initial register values set as described in the following table.

Table 5-1 Boot CPU initial register values

Register	Value
MSR	PR=0 supervisor state EE=0 interrupts disabled ME=0 machine check interrupt disabled IP=0 interrupt prefix-- low memory IR=0,DR=0 real mode (see note 1) IS=0,DS=0 address space 0 (see note 1) SF=0, CM=0, ICM=0 32-bit mode The state of any additional MSR bits are defined in the applicable processor supplement specification.
R3	Effective address of the device tree image. **Note**: This address shall be 8 bytes aligned in memory.
R4	0
R5	0
R6	ePAPR magic value—to distinguish from non-ePAPR-compliant firmware • For Book III-E CPUs shall be 0x45504150 • For non-Book III-E CPUs shall be 0x65504150
R7	shall be the size of the boot or secondary IMA in bytes
R8	0
R9	0
TCR	WRC=0, no watchdog timer reset will occur (see note 2)
other registers	implementation dependent

Note 1: Applicable only to CPUs that define these bits
Note 2: Applicable to Book III-E CPUs only

5.4.2 I/O Devices State

The boot program shall leave all devices with the following conditions true:

- All devices: no DMA and not interrupting
- Host bridges: responding to config cycles and passing through config cycles to children

5.4.3 Initial I/O Mappings (IIO)

A boot program might pass a client program a device tree containing device nodes with a *virtual-reg* property (see 2.3.7, *virtual-reg*). The *virtual-reg* property describes an *Initial I/O* (or *IIO*) mapping set up by firmware, and the value is the effective address of a device's registers.

For Book III-E CPUs, effective to physical address mappings shall be present in the CPU's MMU to map any IIO. An IIO has the following requirements on Book III-E CPUs:

1. An IIO shall be virtually and physically contiguous.

2. An IIO shall map the effective address in *virtual-reg* to the physical address at which the device appears at the point of entry.

3. An IIO shall not be invalidated except by client's explicit action (i.e., not subject to broadcast invalidates from other partitions).

4. The Translation ID (TID) field in the TLB entry shall be zero.

5. The memory and cache access attributes (WIMGE) have the following requirements:
   - WIMG shall be suitable for accessing the device in question. Typically I=1, G=1.
   - E=0 (i.e., big-endian)

6. An IIO shall be large enough to cover all of device's registers.

7. Multiple devices may share an IIO.

5.4.4 Boot CPU Entry Requirements: Real Mode

For real mode (i.e., non-Book III-E) CPUs, the following requirements apply at client entry for boot CPUs:

1. If a CPU has a real mode, the CPU shall have address translation disabled at client entry (i.e., `MSR[IR] =0, MSR[DR]=0`).

2. All PT_LOAD segments shall be loaded into BIMA.

3. The device tree shall be loaded into the BIMA (with the address in r3).

4. r7 shall contain the size of the BIMA.

5.4.5 Boot CPU Entry Requirements for IMAs: Book IIII-E

For Book III-E CPUs the following requirements apply at client entry for boot CPUs:

1. The Boot IMA (BIMA) mapping in the MMU shall map effective address 0 to the lowest physical memory address available.
2. All PT_LOAD segments shall be loaded into BIMA.
3. The device tree shall be loaded into the BIMA (with the address in r3).
4. IIOs shall be present for all devices with a *virtual-reg* property
5. Other mappings may be present in Address Space (AS) 0.
6. No mappings shall be present in Address Space (AS) 1.
7. r7 shall contain the size of the BIMA.
8. The MMU mappings for the BIMA and all IIOs shall be such that the TLBs can accommodate a reasonable number of additional mappings.

Programming Notes

- A boot program might wish to select BIMA size based on client image layout in order to satisfy requirement #2
- Client can determine physical address of IMA by either of two methods:
    1. tlbsx on EA 0, then read and parse TLB entry
    2. parse device tree to find lowest address `reg` range in any memory node

5.4.6 Secondary CPU Entry Requirements; Real Mode

For real mode (i.e., non-Book III-E) CPUs, the following requirements apply at client entry for secondary CPUs:

1. If a CPU has a real mode, the CPU shall have address translation disabled at client entry (i.e., (MSR[IR] =0, MSR[DR]=0).

2. r7 shall contain the size of the BIMA.

5.4.7 Secondary CPU Entry Requirements for IMAs -- Book III-E

For Book III-E CPUs, the following requirements apply at client entry for secondary CPUs:

1. The Secondary IMA (SIMA) mapping in the MMU shall map effective address 0 to the `entry_addr` field in the spin table, aligned down to the SIMA size.

2. The SIMA shall have a minimum size of 4KB.

3. Other mappings may be present in Address Space (AS) 0.

4. No mappings shall be present in Address Space (AS) 1.

5. r7 shall contain the size of the SIMA.

6. The MMU mapping for the SIMA shall be such that the TLBs can accommodate a reasonable number of additional mappings.

Note: Spin table entries do not need to lie in either the BIMA or SIMA.

Programming Notes

- A client program should physically align its secondary entry points so that the 4 KB SIMA size requirement (see requirement #2) is sufficient to ensure that enough code is in the SIMA to transfer the secondary CPU to the client's MMU domain (which will typically involve a temporary mapping in AS1)

- Boot programs will typically need to establish the SIMA mapping after leaving the spin loop and reading the `entry_addr` spin table field. However, this mapping might not be necessary if, for example, the boot program always uses a SIMA that covers all RAM.

5.5 Symmetric Multiprocessing (SMP) Boot Requirements

5.5.1 Overview

For CPUs in an SMP configuration, one CPU shall be designated the boot CPU and initialized as described in section 5.4, *CPU Entry Point Requirements*. All other CPUs are considered *secondary*.

A boot program passes control to a client program on the boot CPU only. At the time the client program is started, all secondary CPUs shall in a quiescent state. A quiescent CPU is in a state where it cannot interfere with the normal operation of other CPUs, nor can its state be affected by the normal operation of other running CPUs, except by an explicit method for enabling or re-enabling the quiescent CPU. The *status* property of the quiescent CPU's `cpu` node in the device tree shall have a value of "disabled" (see 3.7.1, *General Properties of CPU nodes*).

Secondary CPUs may be started using the *spin table* or *implementation-specific* mechanisms described in the following sections.

5.5.2 Spin Table

5.5.2.1 Overview

The ePAPR defines a spin table mechanism for starting secondary CPUs. The boot program places all secondary CPUs into a loop where each CPU spins until the `branch_address` field in the spin table is updated specifying that the core is released.

A spin table is a table data structure consisting of 1 entry per CPU where each entry is defined as follows:

```
struct {
      uint64_t entry_addr;
      uint64_t r3;
      uint32_t rsvd1;
      uint32_t pir;
};
```

The spin table fields are defined as follows:

- **entry_addr.** Specifies the physical address of the client entry point for the spin table code to branch to.

- **r3**. Contains the value to put in the r3 register at secondary cpu entry. The high 32-bits are ignored on 32-bit chip implementations. 64-bit chip implementations however shall load all 64-bits

- **pir**. Contains a value to load into the PIR (processor identification) register for those CPUs with writable PIR.

Before a secondary CPU enters a spin loop, the spin table fields shall be set with these initial values.

Field	Initial Value
`entry_addr`	0x1
`r3`	Value of the *reg* property from the CPU node in the device tree that corresponds to this CPU.
`pir`	A valid PIR value, different on each CPU within the same partition.

The spin table shall be cache-line size aligned in memory.

5.5.2.2 Boot Program Requirements

The boot program shall place a spin loop and spin table into an area of memory that is appropriate for the platform. If the spin loop and table reside in a memory region belonging to a client program, the memory occupied by the loop and table shall be marked *reserved* in the device tree's DTB memory reservation block (see *section 7.3, Memory Reservation Block*).

Before starting a client program on the boot cpu, the boot program shall set certain properties in the device tree passed to the client as follows:

- Each secondary CPU's cpu node shall have a *status* property with a value of "disabled".
- Each secondary CPU's cpu node shall have an *enable-method* property.
- For each secondary cpu node with an *enable-method* value of "spin-table", the `cpu` node shall have a *cpu-release-addr* property that describes the address of the applicable spin table entry to release the CPU.

For secondary CPUs with address translation always enabled (e.g., Book III-E), the boot program shall set up an address mapping in the secondary CPU's MMU for the spin loop and table.

The boot program shall place a spinning CPU in a *quiescent* state where it cannot interfere with the normal operation of other CPUs, nor can its state be affected by the normal operation of other running CPUs, except by an explicit method for enabling or reenabling the quiescent CPU. (see the *enable-method* property).

Note in particular that a running CPU shall be able to issue broadcast TLB invalidations without affecting a quiescent CPU.

When a secondary CPU is released from its spin loop, its state shall be identical to the state of boot CPUs (see 5.4.1, *Boot CPU Initial Register Values*) *except* as noted here:

- R3 contains the value of the `r3` field from the spin table.
- R6 shall be 0.
- If the CPU has a programmable PIR register, the PIR shall contain the value of the `pir` field from the spin table.
- No I/O device mappings (see 5.4.3, *Initial I/O Mappings* (IIO)) are required.
- For CPUs with address translation always enabled, the initial memory mapping (described in 1.1.4) shall not be affected by any actions taken by any other CPU.

5.5.2.3 Client Program Requirements

When a client program is started on its boot CPU, it is passed a device tree that specifies all secondary CPUs that belong to the client, the state of those CPUs, and the address of the spin table entry to release each CPU.

For each secondary CPU, the physical address of the spin table entry for the CPU is specified in the device tree in the cpu node's *cpu-release-addr* property. To activate a secondary CPU, the client program (running on the boot cpu) may write the `pir` field value, may write the `r3` value, and shall write the `entry_addr` value. After the client has written the `entry_addr` field, the `entry_addr` field might subsequently be altered by the boot program.

5.5.3 Implementation-Specific Release from Reset

Some CPUs have implementation-specific mechanisms to hold CPUs in reset (or otherwise inhibit them from executing instructions) and can also direct CPUs to arbitrary reset vectors.

The use of implementation-specific mechanisms is permitted by the ePAPR. CPUs with this capability are indicated by an implementation-specific value in the *enable-method* property of a CPU node. A client program can release these types of CPUs using implementation-specific means not specified by the ePAPR.

When this method is used to release secondary cores from reset, the timebase synchronization requirement (5.5.4, *Timebase* Synchronization) for boot programs does not apply.

5.5.4 Timebase Synchronization

For configurations that use the spin table method of booting secondary cores (i.e.CPU's *enable-method* = "spin-table"), the boot program shall enable and synchronize the time base (TBU and TBL) across the boot and secondary CPUs.

For configurations that use implementation specific methods (see section 5.5.3) to release secondary cores, the methods must provide some means of synchronizing the time base across CPUs. The precise means to accomplish this, which steps are the responsibility of the boot program, and which are the responsibility of the client program is specified by the implementation specific method.

5.6 Asymmetric Configuration Considerations

For multiple CPUs in a partitioned or asymmetric (AMP) configuration, the ePAPR boot requirements apply independently to each *domain* or partition. For example, a four-CPU system could be partitioned into three domains: one SMP domain with two CPUs and two UP domains each with one CPU. Each domain could have distinct client image, device tree, boot cpu, etc.

6 Device Bindings

This chapter contains requirements, known as *bindings*, for how specific types and classes of devices are represented in the device tree. The *compatible* property of a device node describes the specific binding (or bindings) to which the node complies.

Bindings may be defined as extensions of other each. For example a new bus type could be defined as an extension of the *simple-bus* binding. In this case, the *compatible* property would contain several strings identifying each binding—from the most specific to the most general (see section 2.3.1, *compatible*).

6.1 Binding Guidelines

6.1.1 General Principles

When creating a new device tree representation for a device, a binding should be created that fully describes the required properties and value of the device. This set of properties shall be sufficiently descriptive to provide device drivers with needed attributes of the device.

Some recommended practices include:

1. Define a compatible string using the conventions described in section 2.3.1.

2. Use the standard properties (defined in sections 2.3 and 2.4) as applicable for the new device. This usage typically includes the *reg* and *interrupts* properties at a minimum.

3. Use the conventions specified in section 6 (*Device Bindings*) if the new device fits into one the ePAPR defined device classes.

4. Use the miscellaneous property conventions specified in section 6.1.2, if applicable.

5. If new properties are needed by the binding, the recommended format for property names is: "`<company>,<property-name>`", where `<company>` is an OUI or short unique string like a stock ticker that identifies the creator of the binding.
   Example: `ibm,ppc-interrupt-server#s`

6.1.2 Miscellaneous Properties

This section defines a list of helpful properties that might be applicable to many types of devices and device classes. They are defined here to facilitate standardization of names and usage.

6.1.2.1 clock-frequency

Property: *clock-frequency*
Value type: <*prop-encoded-array*>
Description:

Specifies the frequency of a clock in Hz. The value is a <prop-encoded-array> in one of two forms:

1. a 32-bit integer consisting of one <u32> specifying the frequency
2. a 64-bit integer represented as a <u64> specifying the frequency

6.1.2.2 reg-shift

Property: *reg-shift*
Value type: <*u32*>
Description:

The *reg-shift* property provides a mechanism to represent devices that are identical in most respects except for the number of bytes between registers. The *reg-shift* property specifies in bytes how far the discrete device registers are separated from each other. The individual register location is calculated by using following formula: “registers address” << reg-shift. If unspecified, the default value is 0.

For example, in a system where 16540 UART registers are located at addresses 0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, and 0x1C, a `reg-shift = <2>` property would be used to specify register locations.

6.2 PCI and PCI Express

6.2.1 Overview

The Peripheral Component Interconnect or PCI specifies a parallel, single duplex computer bus for connecting PCI devices. PCI Express or PCIe is a point-to-point, full duplex, switch-based interconnect that maintains software compatibility with PCI devices. PCI Express has an extended configuration space compared to PCI that needs to be represented accordingly. Also some PCI properties are not applicable to PCI Express.

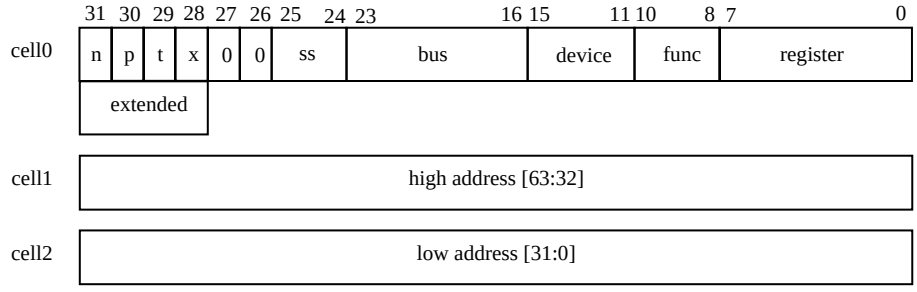
See references [13] [14] [15] listed on the ePAPR references page for technical details on PCI and PCI Express.

The sections that follow describe how PCI buses and device are represented in the device tree.

6.2.2 PCI Address Representation

PCI consists of three separate address spaces: *configuration*, *I/O*, and *memory* space (32-bit and 64-bit).

A PCI address is represented using three 32-bit cells with bit 0 referring to the least significant bit. For cell sizes larger than 32 bits the least significant 32 bits are used with additional high order bits set to zero.



where:

n	0 if the address is relocatable, 1 otherwise
p	1 if the addressable region is prefetchable, 0 otherwise
t	1 if the address is aliased (for nonrelocatable I/O), below 1 MB for memory , or below 64 KB (for relocatable I/O)
x	shall be 0 for ss != 00
extended	is the extended configuration space offset bits 8:11 for PCI Express. This field shall be 0 for conventional configuration space.
ss	denotes the type of address space
bus	the 8-bit bus number
device	the 5-bit device number
func	the 3-bit function number

Figure 6-1, PCI Address Representation

6.2.3 Configuration Space

Configuration cycles access a device's configuration registers, which must be initialized before any memory or I/O space access.

The configuration space address consists of the fields described in Table 6-1.

Table 6-1 Config space fields

Field	Size	Description
Bus Number	8 bits	Each bus within the PCI domain is assigned a unique bus number. The assignment occurs during initialization where all PCI buses and devices are enumerated
Device Number	5 bits	The device number field is used to select individual devices on a PCI bus. Device number assignment is hard-wired dependent on the physical device select line the device is connected to.
Function Number	3 bits	The function number is used to select one of eight independent functional blocks within a PCI device. The assignment of function number is hard-wired within each PCI device. For a single function PCI device, the function number shall be zero.
Register Number	8 or 12 bits	The register number specifies a register within the configuration registers for the selected function. For PCI the register number is an 8-bit value. For PCI Express the register number is a 12-bit value.

Configuration Space Representation

An `ss` field value of **00** specifies configuration space, in which case the fields comprising the address shall be as described in Table 6-2.

Table 6-2 Config space representation

field	value
`extended`	Shall be zero for conventional configuration space. For extended configuration space, is bits 8:11of the offset
`bus,device,func,register`	the configuration space address
`high address`	Shall be zero
`low address`	Shall be zero

6.2.4 I/O Space

I/O Space is intended to be accessed by special *I/O access* instructions available on some CPUs. The I/O addresses are dynamically allocated during the PCI enumeration phase.

The encoding of I/O addresses allow for an address allocation in the first 64KB of I/O space. This is indicated in the *reg* property entry having the *t* bit set and *n* bit cleared (to distinguish between the hard-decoded alias case) .

I/O Space Representation

An `ss` field value of **01** specifies I/O space in which case the fields comprising an address shall be as specified in Table 6-3.

Table 6-3 I/O space representation

field	value
`n`	0 if the address is relocatable, 1 otherwise
`p`	shall be 0
`t`	1 if the address if 10-bit aliasing is present (for nonrelocatable I/O) or below 64 KB (for relocatable I/O)
`x`	shall be 0
`bus,device,func,register`	specifies the base address register of the region • for nonrelocatable `register` shall be zero • for relocatable `register` can be 0x10, 0x14, 0x18, 0x1C, 0x20, or 0x24
`high address`	shall be zero
`low address`	For relocatable is the 32-bit offset from the start of the relocatable region. For nonrelocatable is the 32-bit I/O space address

6.2.5 Memory Space

PCI allows both 32-bit and 64-bit address ranges in Memory space. For compatibility a 32-bit address appears as the first 4 GB within the 64-bit space. The memory addresses are dynamically allocated during the PCI enumeration phase.

The encoding of the base address registers allows an address allocation within the first 1 MB of memory space. This allocation is indicated by the *reg* property entry having the *t* bit set.

32-bit Memory Space Representation

An `ss` field value of `10` denotes 32-bit memory space, in which case the fields comprising the address shall be as specified in Table 6-4.

Table 6-4 32-bit memory space representation

field	value
`n`	0 if the address is relocatable, 1 otherwise
`p`	may be 0 or 1
`t`	1 if the address is below 1 MB
`bus,device,func,register`	specifies the base address register of the region • for nonrelocatable `register` shall be zero • for relocatable `register` can be 0x10, 0x14, 0x18, 0x1C, 0x20, 0x24, 0x30
`high address`	shall be zero
`low address`	For relocatable is the 32-bit offset from the start of the relocatable region. For nonrelocatable is the 32-bit memory space address

64-bit Memory Space Representation

An `ss` field value of **11** denotes 64-bit memory space, in which case the fields comprising the address shall be as specified in Table 6-5.

Table 6-5 64-bit memory space representation

field	value
`n`	0 if the address is relocatable, 1 otherwise
`p`	may be 0 or 1
`t`	shall be 0
`bus,device,func,register`	specifies the first register of the relocatable region's base address register pair. `register` can be 0x10, 0x14, 0x18, 0x1C 0x20
`high address,low address`	For relocatable, is the 64-bit offset from the start of the relocatable region of 64-bit address memory space to the start of the subregion. For nonrelocatable, is the 64-bit memory space address.

6.2.6 Hard-decoded Spaces

PCI allows devices to *hard-decode* memory and I/O addresses, which means they cannot be relocated from a base register. Their corresponding *reg* and *assigned-addresses* properties have the nonrelocatable bit *n* set to show this capability.

Also such devices can make an alias of their hard-decoded I/O addresses by ignoring all but the lower 10 bits of an I/O address. Their corresponding *reg* and *assigned-addresses* property entries have the *t* bit set to show this capability.

6.2.7 Bus nodes

Node Name: The name of the node should be `pci`.

Description

The pci bus describes the node that represents a PCI bus controller. In the case where `pci` node represents a bridge between one bus and another, then this bus node is also a child node of the parent bus.

Properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	A standard property. The value shall include the string "pci"
#address-cells	R		A standard property. Specifies the number of <u32> cells to represent a physical address. For PCI Bus nodes the value is 3.
#size-cells	R		A standard property. Specifies the number of <u32> cells to represent a physical address range. For PCI Bus nodes the value is 2 (64-bit address range).
reg	SD	<prop-encoded-array>	A standard property. For PCI to PCI bridges *reg* is defined as for PCI child nodes (see section 6.2.8) and specifies the Configuration Space address of the bridge's configuration registers. For bridges from some other type of bus, *reg* is as defined for that bus.
ranges	SD	<prop-encoded-array>	A standard property. Shall be present for all PCI bus bridges. This property represents the mapping between parent address to child address spaces. For PCI-PCI bus bridges, there shall be an entry for each of the Configuration, I/O, and Memory spaces if that address space is mapped through the bridge. If no address spaces are mapped through the bridge, then there shall be no `ranges` property.
clock-frequency	O	<prop-encoded-array>	Specifies the frequency of the PCI clock. See section 6.1.2.1, *clock-frequency* for details on the encoding.
bus-range	O	<prop-encoded-array	Specifies the range of bus numbers controlled by this PCI bus. Consists of two <u32> cells. The first value represents the bus number implemented by the bus controller represented by *this* node. The second value represents the largest bus number of any PCI bus in the PCI domain that is subordinate to this node.
slot-names	O	<prop-encoded-array	Describes the labeling of add-in slots. Consists of a <u32> value followed by a <stringlist>. The <u32> value describes the available slots in a system as a bitmask—for each slot the bit corresponding to that slot's Device Number is set. The least significant bit of the integer corresponds to device number 0, the next bit to device number 1, etc. The number of following strings is the same number as the selected slots, the first string is the name of slot with the lowest device number etc.
bus-master-capable	O	<prop-encoded-array	Specifies whether a device is bus master capable. Consists of a <u32> value encoded as a bitmask. A bit that is set specifies that the device is wired to be bus master capable. A bit that is clear specifies that the device is not bus master capable. The least significant bit corresponds to device number 0, the next bit to device number 1, etc.
Usage legend: R=Required, **shall** be present; O=Optional, **may** be present;, OR=Optional but recommended– **should** be present , SD=See Definition.			

6.2.8 Child nodes

Description

The pci child node describes the node that represents a PCI function within a PCI device.

Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Specifies one or more strings that identify the compatibility of the system board. This property can be used by operating systems in selecting platform specific code. The recommended format is `"manufacturer,model-number"`.
reg	R	<prop-encoded-array>	A standard property that is required for PCI Child nodes. Defines the device's addressable regions. Encoded as an arbitrary number (*physical-addr*, *size*) pairs. The *physical-addr* component consists of three cells as defined in section 6.2.2. The *size* is a pair of <u32> integers. The first integer specifies the most significant 32-bit of the 64-bit region size. The second integer specifies the least significant 32-bits. The first pair shall be the configuration space address of the beginning of the set of the function's configuration registers (i.e., the register field is zero) and the size shall be zero. Each additional pair specifies an addressable region of memory or I/O space.
interrupts	O	<prop-encoded-array>	A standard property. The presence of this property specifies that the function represented by this node is connected to a PCI interrupt line. The value is a <u32> value encoded as: INTA=1 INTB=2 INTC=3 INTD=4
assigned-addresses	O	<prop-encoded-array>	Specifies a fixed physical address corresponding to a functions Configuration space base register.
Usage legend: R=Required, **shall** be present; O=Optional, **may** be present;, OR=Optional but recommended– **should** be present , SD=See Definition			

6.3 ISA and Legacy Devices

6.3.1 ISA Interrupt Controllers

This section specifies the requirements for representing an interrupt controller compatible with the PowerPC Common Hardware Reference Platform (CHRP™) ISA interrupt controller. The ISA interrupt controller is basically a set of two cascaded 8259 interrupt controllers. Their interrupt lines may be set as edge or level sensitive.

ISA PIC interrupt controllers require two 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 shall be encoded as follows in interrupt specifiers:

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

Properties

Table 6-6, ISA interrupt controller properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Value shall include “iic”.
reg	R	<prop-encoded-array>	Specifies the physical address of the registers device within the address space of the parent bus.
interrupt-controller	R	<empty>	Specifies that this node is an interrupt controller.
#interrupt-cells	R	<u32>	Shall be 2.
#address-cells	R	<u32>	Shall be 0.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

Compatibility Note

Older versions of device trees may be encountered that contain a deprecated from of an ISA interrupt controller node with a device type value of `chrp,iic`. For compatibility, a client program might want to support `chrp,iic`, which is identical in meaning, and use to `iic`.

6.4 Serial devices

6.4.1 Serial Class Binding

The class of serial devices consists of various types of point to point serial line devices. Examples of serial line devices include the 8250 UART, 16550 UART, HDLC device, and BISYNC device. In most cases hardware compatible with the RS-232 standard fit into the serial device class.

I^2C and SPI (Serial Peripheral Interface) devices shall not be represented as serial port devices because they have their own specific representation.

6.4.1.1 clock-frequency

Property: *clock-frequency*
Value type: *<u32>*
Description:
Specifies the frequency in Hertz of the baud rate generator's input clock.
Example:

```
clock-frequency = <100000000>;
```

6.4.1.2 current-speed

Property: *current-speed*
Value type: *<u32>*
Description:
Specifies the current speed of a serial device in bits per second. A boot program should set this property if it has initialized the serial device.
Example:

```
current-speed = <115200>; # 115200 baud
```

6.4.2 National Semiconductor 16450/16550 Compatible UART Requirements

Serial devices compatible to the National Semiconductor 16450/16550 UART (Universal Asynchronous Receiver Transmitter) should be represented in the device tree using following properties.

Properties

Table 6-7 ns16550 properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Value shall include "ns16550".
clock-frequency	R	<u32>	Specifies the frequency (in Hz) of the baud rate generator's input clock
current-speed	OR	<u32>	Specifies current serial device speed in bits per second
reg	R	<prop-encoded-array>	Specifies the physical address of the registers device within the address space of the parent bus
interrupts	OR	<prop-encoded-array>	Specifies the interrupts generated by this device. The value of the *interrupts* property consists of one or more interrupt specifiers. The format of an interrupt specifier is defined by the binding document describing the node's interrupt parent.
reg-shift	O	<u32>	Specifies in bytes how far the discrete device registers are separated from each other. The individual register location is calculated by using following formula: "registers address" << reg-shift. If unspecified, the default value is 0.
virtual-reg	SD	<u32> or <u64>	See section 2.3.7. Specifies an effective address that maps to the first physical address specified in the *reg* property This property is required if this device node is the system's console.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

6.5 Network devices

Network devices are packet oriented communication devices. Devices in this class are assumed to implement the data link layer (layer 2) of the seven-layer OSI model and use Media Access Control (MAC) addresses. Examples of network devices include Ethernet, FDDI, 802.11, and Token-Ring.

6.5.1 Network Class Binding

6.5.1.1 address-bits

Property: *address-bits*
Value type: *<u32>*
Description:
Specifies number of address bits required to address the device described by this node. This property specifies number of bits in MAC address. If unspecified, the default value is 48.
Example:

```
address-bits = <48>;
```

6.5.1.2 local-mac-address

Property: *local-mac-address*
Value type: <*prop-encoded-array*> encoded as array of hex numbers
Description:
Specifies IEEE 802.3 MAC address that was assigned to the network device described by the node containing this property.
Example:

```
local-mac-address = [ 00 00 12 34 56 78];
```

6.5.1.3 mac-address

Property: *mac-address*
Value type: <*prop-encoded-array*> encoded as array of hex numbers
Description:
Specifies the IEEE 802.3 MAC address that was last used by the boot program. This property should be used in cases where the MAC address assigned to the device by the boot program is different from the *local-mac-address* property. This property shall be used only if the value differs from *local-mac-address* property value.
Example:

```
mac-address = [ 0x01 0x02 0x03 0x04 0x05 0x06 ];
```

6.5.1.4 max-frame-size

Property: *max-frame-size*
Value type: <*u32*>
Description:
Specifies maximum packet length in bytes that the physical interface can send and receive.
Example:

```
max-frame-size = <1518>;
```

6.5.2 Ethernet specific considerations

Network devices based on the IEEE 802.3 collections of LAN standards (collectively referred to as Ethernet) may be represented in the device tree using following properties, in addition to properties specified of the *network* device class.

The properties listed in this section augment the properties listed in the *network* device class.

6.5.2.1 max-speed

Property: *max-speed*
Value type: <*u32*>
Description:
Specifies maximum speed (specified in megabits per second) supported the device.
Example:

```
max-speed = <1000>;
```

6.5.2.2 phy-connection-type

Property: *phy-connection-type*
Value type: <*string*>
Description:
Specifies interface type between the Ethernet device and a physical layer (PHY) device. The value of this property is specific to the implementation.

Recommended values are shown in the following table.

Connection type	Value
Media Independent Interface	“mii”
Reduced Media Independent Interface	“rmii”
Gigabit Media Independent Interface	“gmii”
Reduced Gigabit Media Independent Interface	“rgmii”
rgmii with internal delay	“rgmii-id”
rgmii with internal delay on TX only	“rgmii-txid”
rgmii with internal delay on RX only	“rgmii-rxid”
Ten Bit Interface	“tbi”
Reduced Ten Bit Interface	“rtbi”
Serial Media Independent Interface	“smii”

Example:

```
phy-connection-type = “mii”;
```

6.5.2.3 phy-handle

Property: *phy-handle*
Value type: <*phandle*>
Description:
Specifies a reference to a node representing a physical layer (PHY) device connected to this Ethernet device. This property is required in case where the Ethernet device is connected a physical layer device.
Example:

```
phy-handle = <&PHY0>;
```

6.6 Device Control Register (DCR) Devices

The DCR properties specify a set of standard properties that can be used to represent devices that contain DCR registers. Device Control Registers are on-chip registers that architecturally exist outside the processor core. The existence of the DCR registers and instructions used to access DCR registers is defined in PowerPC ISA. The DCR registers exist in a separate address space (separate from address space accessed using load and store instructions) PowerPC ISA defines number of instructions for accessing the DCR registers, for example `mfdcr`, `mtdcr`. Typically DCR registers control on-chip peripherals found in an SOC.

The binding for a specific device may specify additional requirements regarding the use of the standard DCR properties, including which are required or optional.

The DCR device tree hierarchy is independent of the main tree structure (similar to the way interrupt tree structure and MMIO tree structure are independent). Devices can have standard *reg* property and *dcr-reg* property, if required.

6.6.1 DCR Controller Requirements

6.6.1.1 dcr-reg

Property: *dcr-reg*
Value type: <*prop-encoded-array*>
Description:

Specifies the DCR address and number of DCR registers allocated to the specific device. The value is a <*prop-encoded-array*> and is composed of an arbitrary number of *pairs* of DCR address and number of DCR registers:

```
<DCR address, number of DCR registers>
```

One <u32> cell is required specify the DCR address. One <u32> cell is required to specify the number of DCR registers.

Example:

```
dcr-reg = <180 62>;
```

6.6.1.2 dcr-parent

Property: *dcr-parent*
Value type: <*phandle*>
Description:

Specifies a reference to a node that initiates cycles on the DCR bus. Typically (but not exclusively) the reference is made to a CPU node.

Example:

```
dcr-parent = < &/cpus/PowerPC,440GP@0 >;
```

6.6.2 DCR Programmed Device Requirements

6.6.2.1 dcr-controller

Property: *dcr-controller*
Value type: <*empty*>
Description:
Specifies that the node is a DCR controller that initiates cycles on the DCR bus.

6.6.2.2 dcr-access-method

Property: *dcr-access-method*
Value type: <*string*>
Description:
Specifies the access method for accessing the DCR controller. Two possible values are permitted for this property `native`, and `mmio`. The `native` value indicates that DCR instructions should be used to access DCR bus. The `mmio` value indicates that memory mapped accesses should be used to access DCR bus.
Example:

```
dcr-access-method = “mmio”;
```

6.7 open PIC Interrupt Controllers

This section specifies the requirements for representing open PIC compatible interrupt controllers. An open PIC interrupt controller implements the open PIC architecture (developed jointly by AMD and Cyrix) and specified in *The Open Programmable Interrupt Controller (PIC) Register Interface Specification Revision 1.2* [**18**].

Device tree nodes whose interrupt-parent is an open PIC interrupt controller require two cells to encode interrupt-specifiers. The first cell defines the interrupt number. The second cell defines the sense and level information.

Sense and level information shall be encoded as follows in interrupt specifiers:

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

Properties

Table 6-8 Open-pic properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Value shall include "open-pic".
reg	R	<prop-encoded-array>	Specifies the physical address of the registers device within the address space of the parent bus
interrupt-controller	R	<empty>	Specifies that this node is an interrupt controller
#interrupt-cells	R	<u32>	Shall be 2.
#address-cells	R	<u32>	Shall be 0.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition **Note**: All other standard properties (section 2.3) are allowed but are optional.			

6.8 simple-bus

System-on-a-chip processors may have an internal I/O bus that cannot be probed for devices. The devices on the bus can be accessed directly without additional configuration required. This type of bus is represented as a node with a compatible value of `"simple-bus"`.

Properties

Table 6-9 Simple-bus properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Value shall include `simple-bus`.
ranges	R	<prop-encoded-array>	This property represents the mapping between parent address to child address spaces (see section 2.3.8, *ranges*).
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition **Note**: All other standard properties (section 2.3) are allowed but are optional.			

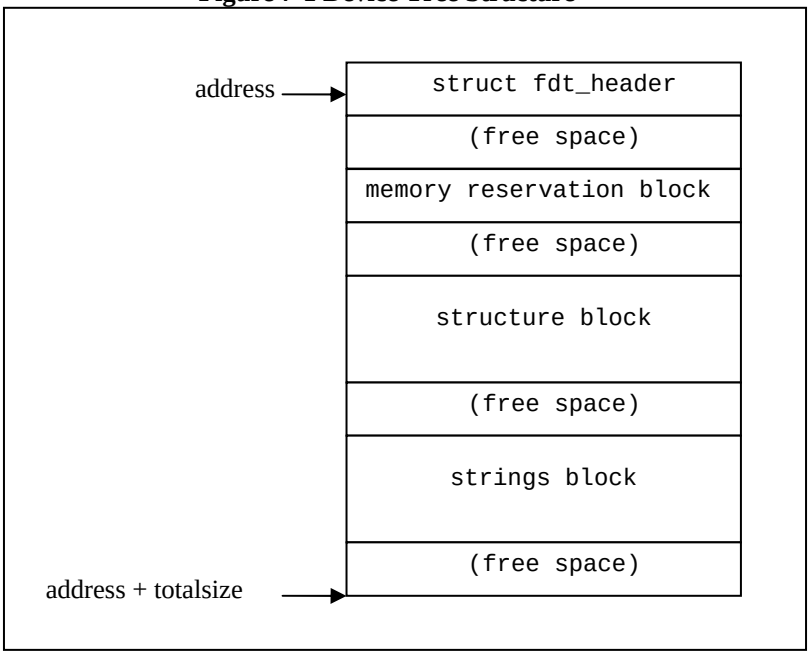
7 Flat Device Tree Physical Structure

An ePAPR boot program communicates the entire device tree to the client program as a single, linear, pointerless data structure known as the *flattened device tree* or *device tree blob*.

This data structure consists of a small header (see 7.2), followed by three variable sized sections: the memory reservation block (see 7.3), the structure block (see 7.4) and the strings block (see 7.5). These should be present in the flattened device tree in that order.

Thus, the device tree structure as a whole, when loaded into memory at address, will resemble the diagram in Figure 7-1 (lower addresses are at the top of the diagram).

Figure 7-1 Device Tree Structure



The (free space) sections may not be present, though in some cases they might be required to satisfy the alignment constraints of the individual blocks (see 7.6).

7.1 Versioning

Several versions of the flattened device tree structure have been defined since the original definition of the format. Fields in the header give the version, so that the client program can determine if the device tree is encoded in a compatible format.

This document describes only version 17 of the format. ePAPR-compliant boot programs shall provide a device tree of version 17 or later, and should provide a device tree of a version that is backwards compatible with version 16. ePAPR-compliant client programs shall accept device trees of any version backwards compatible with version 17 and may accept other versions as well.

Note: The version is with respect to the binary structure of the device tree, not its content.

7.2 Header

The layout of the header for the device tree is defined by the following C structure. All the header fields are 32-bit integers, stored in big-endian format.

```
struct fdt_header {
      uint32_t magic;
      uint32_t totalsize;
      uint32_t off_dt_struct;
      uint32_t off_dt_strings;
      uint32_t off_mem_rsvmap;
      uint32_t version;
      uint32_t last_comp_version;
      uint32_t boot_cpuid_phys;
      uint32_t size_dt_strings;
      uint32_t size_dt_struct;
};
```

- `magic`

  This field shall contain the value 0xd00dfeed (big-endian).

- `totalsize`

  This field shall contain the total size of the device tree data structure. This size shall encompass all sections of the structure: the header, the memory reservation block, structure block and strings block, as well as any free space gaps between the blocks or after the final block.

- `off_dt_struct`

  This field shall contain the offset in bytes of the structure block (see 7.4) from the beginning of the header.

- `off_dt_strings`

  This field shall contain the offset in bytes of the strings block (see 7.5) from the beginning of the header.

- `off_mem_rsvmap`

  This field shall contain the offset in bytes of the memory reservation block (see 7.3) from the beginning of the header.

- `version`

  This field shall contain the version of the device tree data structure. The version is 17 if using the structure as defined in this document. An ePAPR boot program may provide the device tree of a later version, in which case this field shall contain the version number defined in whichever later document gives the details of that version.

- `last_comp_version`

  This field shall contain the lowest version of the device tree data structure with which the version used is backwards compatible. So, for the structure as defined in this document (version 17), this field shall contain 16 because version 17 is backwards compatible with version 16, but not earlier versions. As per 7.1, an ePAPR boot program should provide a device tree in a format which is backwards compatible with version 16, and thus this field shall always contain 16.

- *`boot_cpuid_phys`*

  This field shall contain the physical ID of the system's boot CPU. It shall be identical to the physical ID given in the *reg* property of that CPU node within the device tree.

- `size_dt_strings`

  This field shall contain the length in bytes of the strings block section of the device tree blob.

- `size_dt_struct`

  This field shall contain the length in bytes of the structure block section of the device tree blob.

7.3 Memory Reservation Block

7.3.1 Purpose

The *memory reservation block* provides the client program with a list of areas in physical memory which are *reserved*; that is, which shall not be used for general memory allocations. It is used to protect vital data structures from being overwritten by the client program. For example, on some systems with an IOMMU, the TCE (translation control entry) tables initialized by an ePAPR boot program would need to be protected in this manner. Likewise, any boot program code or data used during the client program's runtime would need to be reserved (e.g., RTAS on Open Firmware platforms). The ePAPR does not require the boot program to provide any such runtime components, but it does not prohibit implementations from doing so as an extension.

More specifically, a client program shall not access memory in a reserved region unless other information provided by the boot program explicitly indicates that it shall do so. The client program may then access the indicated section of the reserved memory in the indicated manner. Methods by

which the boot program can indicate to the client program specific uses for reserved memory may appear in this document, in optional extensions to it, or in platform-specific documentation.

The reserved regions supplied by a boot program may, but are not required to, encompass the device tree blob itself. The client program shall ensure that it does not overwrite this data structure before it is used, whether or not it is in the reserved areas.

7.3.2 Format

The memory reservation block consists of a list of pairs of 64-bit big-endian integers, each pair being represented by the following C structure.

```
struct fdt_reserve_entry {
      uint64_t address;
      uint64_t size;
};
```

Each pair gives the physical address and size of a reserved memory region. These given regions shall not overlap each other. The list of reserved blocks shall be terminated with an entry where both address and size are equal to 0. Note that the address and size values are always 64-bit. On 32-bit CPUs the upper 32-bits of the value are ignored.

Each `uint64_t` in the memory reservation block, and thus the memory reservation block as a whole, shall be located at an 8-byte aligned offset from the beginning of the device tree blob (see 7.6)

7.4 Structure Block

The structure block describes the structure and contents of the device tree itself. It is composed of a sequence of tokens with data, as described in 7.4.1. These are organized into a linear tree structure, as described in 7.4.2.

Each token in the structure block, and thus the structure block itself, shall be located at a 4-byte aligned offset from the beginning of the device tree blob (see 7.6).

7.4.1 Lexical structure

The structure block is composed of a sequence of pieces, each beginning with a *token*, that is, a big-endian 32-bit integer. Some tokens are followed by extra data, the format of which is determined by the token value. All tokens shall be aligned on a 32-bit boundary, which may require padding bytes (with a value of 0x0) to be inserted after the previous token's data.

The five token types are as follows:

- FDT_BEGIN_NODE (0x00000001)

  The FDT_BEGIN_NODE token marks the beginning of a node's representation. It shall be followed by the node's unit name as extra data. The name is stored as a NULL-terminated string, and shall include the unit address (see 2.2.1, *Node Names*), if any.

  The node name is followed by zeroed padding bytes, if necessary for alignment, and then the next token, which may be any token except FDT_END.

- FDT_END_NODE (0x00000002)

  The FDT_END_NODE token marks the end of a node's representation. This token has no extra data; so it is followed immediately by the next token, which may be any token except FDT_PROP.

- FDT_PROP (0x00000003)

  The FDT_PROP token marks the beginning of the representation of one property in the device tree. It shall be followed by extra data describing the property. This data consists first of the property's length and name represented as the following C structure:

```
struct {
      uint32_t len;
      uint32_t nameoff;
}
```

  Both the fields in this structure are 32-bit big-endian integers.

  - `len` gives the length of the property's value in bytes (which may be zero, indicating an empty property, see 2.2.4.2, *Property Values*).
  - `nameoff` gives an offset into the strings block (see 7.5) at which the property's name is stored as a NULL-terminated string.

  After this structure, the property's value is given as a byte string of length `len`. This value is followed by zeroed padding bytes (if necessary) to align to the next 32-bit boundary and then the next token, which may be any token except FDT_END.

- FDT_NOP (0x00000004)

  The FDT_NOP token will be ignored by any program parsing the device tree. This token has no extra data; so it is followed immediately by the next token, which can be any valid token.

  A property or node definition in the tree can be overwritten with FDT_NOP tokens to remove it from the tree without needing to move other sections of the tree's representation in the device tree blob.

- FDT_END (0x00000009)

  The FDT_END token marks the end of the structure block. There shall be only one FDT_END token, and it shall be the last token in the structure block. It has no extra data; so the byte immediately after the FDT_END token has offset from the beginning of the structure block equal to the value of the `size_dt_struct` field in the device tree blob header.

7.4.2 Tree structure

The device tree structure is represented as a linear tree: the representation of each node begins with an FDT_BEGIN_NODE token and ends with an FDT_END_NODE token. The node's properties and subnodes (if any) are represented before the FDT_END_NODE, so that the FDT_BEGIN_NODE and FDT_END_NODE tokens for those subnodes are nested within those of the parent.

The structure block as a whole consists of the root node's representation (which contains the representations for all other nodes), followed by an FDT_END token to mark the end of the structure block as a whole.

More precisely, each node's representation consists of the following components:

- (optionally) any number of FDT_NOP tokens
- FDT_BEGIN_NODE token
  - The node's name as a NULL-terminated string
  - [zeroed padding bytes to align to a 4-byte boundary]
- For each property of the node:
  - (optionally) any number of FDT_NOP tokens
  - FDT_PROP token
    - property information as given in 7.4.1
    - [zeroed padding bytes to align to a 4-byte boundary]
- Representations of all child nodes in this format
- (optionally) any number of FDT_NOP tokens
- FDT_END_NODE token

Note that this process requires that all property definitions for a particular node precede any subnode definitions for that node. Although the structure would not be ambiguous if properties and subnodes were intermingled, the code needed to process a flat tree is simplified by this requirement.

7.5 Strings Block

The strings block contains strings representing all the property names used in the tree. These NULL-terminated strings are simply concatenated together in this section, and referred to from the structure block by an offset into the strings block.

The strings block has no alignment constraints and may appear at any offset from the beginning of the device tree blob.

7.6 Alignment

For the data in the memory reservation and structure blocks to be used without unaligned memory accesses, they shall lie at suitably aligned memory addresses. Specifically, the memory reservation block shall be aligned to an 8-byte boundary and the structure block to a 4-byte boundary.

Furthermore, the device tree blob as a whole can be relocated without destroying the alignment of the subblocks.

As described in the previous sections, the structure and strings blocks shall have aligned offsets from the beginning of the device tree blob. To ensure the in-memory alignment of the blocks, it is sufficient to ensure that the device tree as a whole is loaded at an address aligned to the largest alignment of any of the subblocks, that is, to an 8-byte boundary. As described in 5.2 (*Device Tree)* an ePAPR-compliant boot program shall load the device tree blob at such an aligned address before passing it to the client program. If an ePAPR client program relocates the device tree blob in memory, it should only do so to another 8-byte aligned address.

Appendix A Device Tree Source Format (version 1)

The Device Tree Source (DTS) format is a textual representation of a device tree in a form that can be processed by dtc into a binary device tree in the form expected by the kernel. The following description is not a formal syntax definition of DTS, but describes the basic constructs used to represent device trees.

Node and property definitions

Device tree nodes are defined with a node name and unit address with braces marking the start and end of the node definition. They may be preceded by a label.

```
[label:] node-name[@unit-address] {
      [properties definitions]
      [child nodes]
}
```

Nodes may contain property definitions and/or child node definitions. If both are present, properties shall come before child nodes.

Property definitions are name value pairs in the form:

```
[label:] property-name = value;
```

except for properties with empty (zero length) value which have the form:

```
[label:] property-name;
```

Property values may be defined as an array of 32-bit integer cells, as NULL-terminated strings, as bytestrings or a combination of these.

- Arrays of cells are represented by angle brackets surrounding a space separated list of C-style integers. Example:

  ```
  interrupts = <17 0xc>;
  ```

- A 64-bit value is represented with two 32-bit cells. Example:

  ```
  clock-frequency = <0x00000001 0x00000000>;
  ```

- A NULL-terminated string value is represented using double quotes (the property value is considered to include the terminating NULL character). Example:

  ```
  compatible = "simple-bus";
  ```

- A bytestring is enclosed in square brackets [ ] with each byte represented by two hexadecimal digits. Spaces between each byte are optional. Example:

  ```
  local-mac-address = [00 00 12 34 56 78];
  ```

  or equivalently:

  ```
  local-mac-address = [000012345678];
  ```

- Values may have several comma-separated components, which are concatenated together. Example:

```
compatible = "ns16550", "ns8250";
example = <0xf00f0000 19>, "a strange property format";
```

- In a cell array a reference to another node will be expanded to that node's phandle. References may be & followed by a node's label. Example:

```
interrupt-parent = < &mpic >;
```

  or they may be & followed by a node's full path in braces. Example:

```
interrupt-parent = < &{/soc/interrupt-controller@40000} >;
```

- Outside a cell array, a reference to another node will be expanded to that node's full path. Example:

```
ethernet0 = &EMAC0;
```

- Labels may also appear before or after any component of a property value, or between cells of a cell array, or between bytes of a bytestring. Examples:

```
reg = reglabel: <0 sizelabel: 0x1000000>;
prop = [ab cd ef byte4: 00 ff fe];
str = start: "string value" end: ;
```

File layout

Version 1 DTS files have the overall layout:

```
/dts-v1/;

[memory reservations]

/ {
      [property definitions]
      [child nodes]
};
```

- The /dts-v1/; shall be present to identify the file as a version 1 DTS (dts files without this tag will be treated by dtc as being in the obsolete version 0, which uses a different format for integers in addition to other small but incompatible changes).

- Memory reservations define an entry for the device tree blob's memory reservation table. They have the form:

  e.g., `/memreserve/ <address> <length>;`

  Where <address> and <length> are 64-bit C-style integers.

- The / { ... }; section defines the root node of the device tree.

- C style (/* ... */) and C++ style (// ...) comments are supported.

Appendix B1 Ebony Device Tree

This appendix shows a complete device tree for the IBM 440-based Ebony system.

```
/*
 * Device Tree Source for IBM Ebony
 *
 * Copyright (c) 2006, 2007 IBM Corp.
 * Josh Boyer <jwboyer@linux.vnet.ibm.com>, David Gibson <dwg@au1.ibm.com>
 *
 *
 *
 * This file is licensed under the terms of the GNU General Public
 * License version 2. This program is licensed "as is" without
 * any warranty of any kind, whether express or implied.
 */
/dts-v1/;
/ {
  #address-cells = <0x2>;
  #size-cells = <0x1>;
  model = "ibm,ebony";
  compatible = "ibm,ebony";
  dcr-parent = <&/cpus/cpu@0>;

  aliases {
    ethernet0 = &EMAC0;
    ethernet1 = &EMAC1;
    serial0 = &UART0;
    serial1 = &UART1;
  };

  cpus {
    #address-cells = <0x1>;
    #size-cells = <0x0>;

    cpu@0 {
      device_type = "cpu";
      model = "PowerPC,440GP";
      reg = <0x0>;
      clock-frequency = <0x179a7b00>;
      timebase-frequency = <0x179a7b00>;
      i-cache-line-size = <0x20>;
      d-cache-line-size = <0x20>;
      i-cache-size = <0x8000>;
      d-cache-size = <0x8000>;
      dcr-controller;
      dcr-access-method = "native";
    };
  };

  memory {
    device_type = "memory";
    reg = <0x0 0x0 0x8000000>;
  };

```

```
  UIC0: interrupt-controller0 {
    compatible = "ibm,uic-440gp", "ibm,uic";
    interrupt-controller;
    cell-index = <0x0>;
    dcr-reg = <0xc0 0x9>;
    #address-cells = <0x0>;
    #size-cells = <0x0>;
    #interrupt-cells = <0x2>;
  };

  UIC1: interrupt-controller1 {
    compatible = "ibm,uic-440gp", "ibm,uic";
    interrupt-controller;
    cell-index = <0x1>;
    dcr-reg = <0xd0 0x9>;
    #address-cells = <0x0>;
    #size-cells = <0x0>;
    #interrupt-cells = <0x2>;
    interrupt-parent = <&UIC1>;
    interrupts = <0x1e 0x4 0x1f 0x4>;
  };

  cpc {
    compatible = "ibm,cpc-440gp";
    dcr-reg = <0xb0 0x3 0xe0 0x10>;
  };

  plb {
    compatible = "ibm,plb-440gp", "ibm,plb4";
    #address-cells = <0x2>;
    #size-cells = <0x1>;
    ranges;
    clock-frequency = <0x7de2900>;

    sram {
      compatible = "ibm,sram-440gp";
      dcr-reg = <0x20 0x8 0xa 0x1>;
    };

    dma {
      compatible = "ibm,dma-440gp";
      dcr-reg = <0x100 0x27>;
    };

    MAL0: mcmal {
      compatible = "ibm,mcmal-440gp", "ibm,mcmal";
      dcr-reg = <0x180 0x62>;
      num-rx-chans = <0x4>;
      num-tx-chans = <0x4>;
      interrupt-parent = <&MAL0>;
      interrupts = <0x0 0x1 0x2 0x3 0x4>;
      #interrupt-cells = <0x1>;
      #address-cells = <0x0>;
      #size-cells = <0x0>;
      interrupt-map = <
        0 &UIC0 a 4
        1 &UCI0 b 4
        2 &UCI1 0 4
        3 &UCI1 1 4
```

```
        4 &UIC1 2 4>;
      interrupt-map-mask = <0xffffffff>;
    };

    POB0: opb {
      compatible = "ibm,opb-440gp", "ibm,opb";
      #address-cells = <0x1>;
      #size-cells = <0x1>;
      ranges = <0x00000000 0x1 0x00000000 0x80000000
        0x80000000 0x1 0x80000000 0x80000000>;
      dcr-reg = <0x90 0xb>;
      interrupt-parent = <&UIC1>;
      interrupts = <0x7 0x4>;
      clock-frequency = <0x3ef1480>;

      ebc {
        compatible = "ibm,ebc-440gp", "ibm,ebc";
        dcr-reg = <0x12 0x2>;
        #address-cells = <0x2>;
        #size-cells = <0x1>;
        clock-frequency = <0x3ef1480>;
        ranges = <
          0x0 0x0 0xfff00000 0x100000
          0x1 0x0 0x48000000 0x100000
          0x2 0x0 0xff800000 0x400000
          0x3 0x0 0x48200000 0x100000
          0x7 0x0 0x48300000 0x100000>;
        interrupts = <0x5 0x4>;
        interrupt-parent = <&UIC1>;

        fpga@7,0 {
          compatible = "Ebony-FPGA";
          reg = <0x7 0x0 0x10>;
          virtual-reg = <0xe8300000>;
        };

        ir@3,0 {
          reg = <0x3 0x0 0x10>;
        };

        large-flash@2,0 {
          compatible = "jedec-flash";
          bank-width = <0x1>;
          reg = <0x2 0x0 0x400000>;
          #address-cells = <0x1>;
          #size-cells = <0x1>;

          partition@380000 {
            reg = <0x380000 0x80000>;
            label = "firmware";
          };

          partition@0 {
            reg = <0x0 0x380000>;
            label = "fs";
          };
        };
```

```
        nvram@1,0 {
          compatible = "ds1743-nvram";
          reg = <0x1 0x0 0x2000>;
          #bytes = <0x2000>;
        };

        small-flash@0,80000 {
          compatible = "jedec-flash";
          bank-width = <0x1>;
          reg = <0x0 0x80000 0x80000>;
          #address-cells = <0x1>;
          #size-cells = <0x1>;

          partition@0 {
            read-only;
            reg = <0x0 0x80000>;
            label = "OpenBIOS";
          };
        };
      };

      UART0: serial@40000200 {
        device_type = "serial";
        compatible = "ns16550";
        reg = <0x40000200 0x8>;
        virtual-reg = <0xe0000200>;
        clock-frequency = <0xa8c000>;
        current-speed = <0x2580>;
        interrupts = <0x0 0x4>;
        interrupt-parent = <&UIC0>;
      };

      UART1: serial@40000300 {
        device_type = "serial";
        compatible = "ns16550";
        reg = <0x40000300 0x8>;
        virtual-reg = <0xe0000300>;
        clock-frequency = <0xa8c000>;
        current-speed = <0x2580>;
        interrupts = <0x1 0x4>;
        interrupt-parent = <&UIC0>;
      };

      i2c@40000400 {
        compatible = "ibm,iic-440gp", "ibm,iic";
        reg = <0x40000400 0x14>;
        interrupts = <0x2 0x4>;
        interrupt-parent = <&UIC0>;
      };

      i2c@40000500 {
        compatible = "ibm,iic-440gp", "ibm,iic";
        reg = <0x40000500 0x14>;
        interrupts = <0x3 0x4>;
        interrupt-parent = <&UIC0>;
      };
```

```
1
2         gpio@40000700 {
3           compatible = "ibm,gpio-440gp";
4           reg = <0x40000700 0x20>;
5         };
6
7         ZMII0: emac-zmii@40000780 {
8           compatible = "ibm,zmii-440gp", "ibm,zmii";
9           reg = <0x40000780 0xc>;
10        };
11
12        EMAC0: ethernet@40000800 {
13          linux,network-index = <0x0>;
14          device_type = "network";
15          compatible = "ibm,emac-440gp", "ibm,emac";
16          interrupts = <0x1c 0x4 0x1d 0x4>;
17          interrupt-parent = <&UIC1>;
18          reg = <0x40000800 0x70>;
19          local-mac-address = [00 04 ac e3 1b 0b];
20          mal-device = <&MAL0>;
21          mal-tx-channel = <0x0 0x1>;
22          mal-rx-channel = <0x0>;
23          cell-index = <0x0>;
24          max-frame-size = <0x5dc>;
25          rx-fifo-size = <0x1000>;
26          tx-fifo-size = <0x800>;
27          phy-mode = "rmii";
28          phy-map = <0x1>;
29          zmii-device = <&ZMII0>;
30          zmii-channel = <0x0>;
31        };
32
33        EMAC1: ethernet@40000900 {
34          linux,network-index = <0x1>;
35          device_type = "network";
36          compatible = "ibm,emac-440gp", "ibm,emac";
37          interrupts = <0x1e 0x4 0x1f 0x4>;
38          interrupt-parent = <&UIC1>;
39          reg = <0x40000900 0x70>;
40          local-mac-address = [00 04 ac e3 1b 0c];
41          mal-device = <&MAL0>;
42          mal-tx-channel = <0x2 0x3>;
43          mal-rx-channel = <0x1>;
44          cell-index = <0x1>;
45          max-frame-size = <0x5dc>;
46          rx-fifo-size = <0x1000>;
47          tx-fifo-size = <0x800>;
48          phy-mode = "rmii";
49          phy-map = <0x1>;
50          zmii-device = <&ZMII0>;
51          zmii-channel = <0x1>;
52        };
53
54        gpt@40000a00 {
55          reg = <0x40000a00 0xd4>;
56          interrupts = <0x12 0x4 0x13 0x4 0x14 0x4 0x15 0x4 0x16 0x4>;
57          interrupt-parent = <&UIC0>;
58        };
59      };
```

```
    PCIX0: pci@20ec00000 {
      device_type = "pci";
      #interrupt-cells = <0x1>;
      #size-cells = <0x2>;
      #address-cells = <0x3>;
      compatible = "ibm,plb440gp-pcix", "ibm,plb-pcix";
      primary;
      reg = <0x2 0xec00000 0x8
        0x0 0x0 0x0
        0x2 0xed00000 0x4
        0x2 0xec80000 0xf0
        0x2 0xec80100 0xfc>;

      ranges = <0x2000000 0x0 0x80000000 0x3 0x80000000 0x0 0x80000000
        0x1000000 0x0 0x0 0x2 0x8000000 0x0 0x10000>;

      dma-ranges = <0x42000000 0x0 0x0 0x0 0x0 0x0 0x80000000>;

      interrupt-map-mask = <0xf800 0x0 0x0 0x0>;
      interrupt-map = <
      0x800 0x0 0x0 0x0 &UIC0 0x17 0x8
      0x1000 0x0 0x0 0x0 &UIC0 0x18 0x8
      0x1800 0x0 0x0 0x0 &UIC0 0x19 0x8
      0x2000 0x0 0x0 0x0 &UIC0 0x1a 0x8>;
    };
  };
};
```

Appendix B2 – MPC8572DS Device Tree

This appendix shows a device tree for the Freescale MPC8572DS system. **Note**: to simplify the example, some portions of the device tree have been removed.

```
/*
 * MPC8572 DS Device Tree Source
 *
 * Copyright 2007 Freescale Semiconductor Inc.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation; either version 2 of the License, or (at your
 * option) any later version.
 */
/dts-v1/;
/ {
  model = "fsl,MPC8572DS";
  compatible = "fsl,MPC8572DS";
  #address-cells = <1>;
  #size-cells = <1>;

  aliases {
    ethernet0 = &enet0;
    ethernet1 = &enet1;
    serial0 = &serial0;
    serial1 = &serial1;
    pci1 = &pci1;
  };

  cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
      device_type = "cpu";
      reg = <0>;
      d-cache-line-size = <32>;  // 32 bytes
      i-cache-line-size = <32>;  // 32 bytes
      d-cache-size = <0x8000>;   // L1, 32K
      i-cache-size = <0x8000>;   // L1, 32K
      timebase-frequency = <0>;  // filled in by u-boot
      bus-frequency = <0>;     // filled in by u-boot
      clock-frequency = <0>;    // filled in by u-boot
    };

    cpu@1 {
      device_type = "cpu";
      reg = <1>;
      d-cache-line-size = <32>;  // 32 bytes
      i-cache-line-size = <32>;  // 32 bytes
      d-cache-size = <0x8000>;   // L1, 32K
      i-cache-size = <0x8000>;   // L1, 32K
      timebase-frequency = <0>;  // filled in by u-boot
      bus-frequency = <0>;     // filled in by u-boot
      clock-frequency = <0>;    // filled in by u-boot
    };
```

```
  };

  memory {
    device_type = "memory";
    reg = <00000000 00000000>;  // Filled by U-Boot
  };

  soc8572@ffe00000 {
    #address-cells = <1>;
    #size-cells = <1>;
    device_type = "simple-bus";
    ranges = <00000000 0xffe00000 00100000>;
    reg = <0xffe00000 0x00001000>; // CCSRBAR
    bus-frequency = <0>;    // Filled in by uboot.

    memory-controller@2000 {
      compatible = "fsl,mpc8572-memory-controller";
      reg = <0x2000 0x1000>;
      interrupt-parent = <&mpic>;
      interrupts = <0x12 0x2>;
    };

    memory-controller@6000 {
      compatible = "fsl,mpc8572-memory-controller";
      reg = <0x6000 0x1000>;
      interrupt-parent = <&mpic>;
      interrupts = <0x12 0x2>;
    };

    l2-cache-controller@20000 {
      compatible = "fsl,mpc8572-l2-cache-controller";
      reg = <0x20000 0x1000>;
      cache-line-size = <32>; // 32 bytes
      cache-size = <0x80000>;  // L2, 512K
      interrupt-parent = <&mpic>;
      interrupts = <0x10 0x2>;
    };

    i2c@3000 {
      #address-cells = <1>;
      #size-cells = <0>;
      cell-index = <0>;
      compatible = "fsl-i2c";
      reg = <0x3000 0x100>;
      interrupts = <0x2b 0x2>;
      interrupt-parent = <&mpic>;
      dfsrr;
    };
```

```
1
2       i2c@3100 {
3         #address-cells = <1>;
4         #size-cells = <0>;
5         cell-index = <1>;
6         compatible = "fsl-i2c";
7         reg = <0x3100 0x100>;
8         interrupts = <2b 2>;
9         interrupt-parent = <&mpic>;
10        dfsrr;
11      };
12
13      mdio@24520 {
14        #address-cells = <1>;
15        #size-cells = <0>;
16        compatible = "fsl,gianfar-mdio";
17        reg = <0x24520 0x20>;
18
19        phy0: ethernet-phy@0 {
20          interrupt-parent = <&mpic>;
21          interrupts = <0xA 1>;
22          reg = <0>;
23        };
24        phy1: ethernet-phy@1 {
25          interrupt-parent = <&mpic>;
26          interrupts = <0xA 1>;
27          reg = <1>;
28        };
29      };
30
31      enet0: ethernet@24000 {
32        cell-index = <0>;
33        device_type = "network";
34        model = "eTSEC";
35        compatible = "gianfar";
36        reg = <0x24000 0x1000>;
37        local-mac-address = [ 00 00 00 00 00 00 ];
38        interrupts = <0x1d 0x2 0x1e 0x2 0x22 0x2>;
39        interrupt-parent = <&mpic>;
40        phy-handle = <&phy0>;
41        phy-connection-type = "rgmii-id";
42      };
```

```
1
2       enet1: ethernet@25000 {
3         cell-index = <1>;
4         device_type = "network";
5         model = "eTSEC";
6         compatible = "gianfar";
7         reg = <25000 1000>;
8         local-mac-address = [ 00 00 00 00 00 00 ];
9         interrupts = <23 2 24 2 28 2>;
10        interrupt-parent = <&mpic>;
11        phy-handle = <&phy1>;
12        phy-connection-type = "rgmii-id";
13      };
14
15      serial0: serial@4500 {
16        cell-index = <0>;
17        device_type = "serial";
18        compatible = "ns16550";
19        reg = <0x4500 0x100>;
20        clock-frequency = <0>;
21        interrupts = <0x2a 0x2>;
22        interrupt-parent = <&mpic>;
23      };
24
25      serial1: serial@4600 {
26        cell-index = <1>;
27        device_type = "serial";
28        compatible = "ns16550";
29        reg = <0x4600 0x100>;
30        clock-frequency = <0>;
31        interrupts = <0x2a 0x2>;
32        interrupt-parent = <&mpic>;
33      };
34
35      global-utilities@e0000 {  //global utilities block
36        compatible = "fsl,mpc8572-guts";
37        reg = <0xe0000 0x1000>;
38        fsl,has-rstcr;
39      };
40
41      mpic: pic@40000 {
42        clock-frequency = <0>;
43        interrupt-controller;
44        #address-cells = <0>;
45        #interrupt-cells = <2>;
46        reg = <0x40000 0x40000>;
47        compatible = "chrp,open-pic";
48        device_type = "open-pic";
49        big-endian;
50      };
51    };
52
53    pci1: pcie@ffe09000 {
54      cell-index = <1>;
55      compatible = "fsl,mpc8548-pcie";
56      device_type = "pci";
57      #interrupt-cells = <1>;
58      #size-cells = <2>;
59      #address-cells = <3>;
```

```
    reg = <ffe09000 1000>;
    bus-range = <0 ff>;
    ranges = <02000000 0 a0000000 a0000000 0 20000000
       01000000 0 00000000 ffc10000 0 00010000>;
    clock-frequency = <1fca055>;
    interrupt-parent = <&mpic>;
    interrupts = <1a 2>;
    interrupt-map-mask = <f800 0 0 7>;
    interrupt-map = <
      /* IDSEL 0x0 */
      0000 0 0 1 &mpic 4 1
      0000 0 0 2 &mpic 5 1
      0000 0 0 3 &mpic 6 1
      0000 0 0 4 &mpic 7 1
      >;
    pcie@0 {
      reg = <0 0 0 0 0>;
      #size-cells = <2>;
      #address-cells = <3>;
      device_type = "pci";
      ranges = <02000000 0 a0000000
         02000000 0 a0000000
         0 20000000

         01000000 0 00000000
         01000000 0 00000000
         0 00100000>;
    };
  };

};
```