PCI Bus Binding to:

IEEE Std 1275-1994
Standard for Boot
(Initialization Configuration)
Firmware

Revision 1.6

DRAFT

24 October, 1995

Foreword by the Chairman of the IEEE 1275 Working Group

(This foreword is not a part of the Specification.)

Introduction

Firmware is the ROM-based software that controls a computer between the time it is turned on and the time the primary operating system takes control of the machine. Firmware's responsibilities include testing and initializing the hardware, determining the hardware configuration, loading (or booting) the operating system, and providing interactive debugging facilities in case of faulty hardware or software.

Historical Perspective

Historically, firmware designs have been proprietary and often specific to a particular bus or instruction set architecture (ISA). This need not be the case. Firmware can be designed to be machine-independent and easily portable to different hardware. There is a strong analogy with operating systems in this respect. Prior to the advent of the portable UNIX operating system in the mid-seventies, the prevailing wisdom was that operating systems must be heavily tuned to a particular computer system design and thus effectively proprietary to the vendor of that system.

The *IEEE Std 1275-1994 Standard for Boot (Initialization Configuration), Core Requirements and Practices* (referred to in the remainder of this document as Open Firmware) specification is based on Sun Microsystem's OpenBoot firmware. The OpenBoot design effort began in 1988, when Sun was building computers based on three different processor families, thus OpenBoot was designed from the outset to be ISA-independent (independent of the Instruction Set Architecture). The first version of OpenBoot was introduced on Sun's SPARCstation 1 computers. Based on experience with those machines, OpenBoot version 2 was developed, and was first shipped on SPARCstation 2 computers. This standard is based on OpenBoot version 2.

Purpose and Features of the Open Firmware Specification

Open Firmware has the following features:

- Mechanism for loading and executing programs (such as operating systems) from disks, tapes, network interfaces, and other devices.
- ISA-independent method for identifying devices "plugged-in" to expansion buses, and for providing firmware and diagnostics drivers for these devices.
- An extensible and programmable command language based on the Forth programming language.
- Methods for managing user-configurable options stored in non-volatile memory.
- A "call back" interface allowing other programs to make use of Open Firmware services.
- Debugging tools for hardware, firmware, firmware drivers, and system software.

Purpose of this Bus Binding

This document specifies the application of Open Firmware to the PCI Local Bus, including PCI-specific requirements and practices for address format, interrupts, probing, and related properties and methods.

The core requirements and practices specified by Open Firmware must be augmented by system-specific requirements to form a complete specification for the firmware for a particular system. This document establishes such additional requirements pertaining to PCI.

Task Group Members

The following individuals were members of the Task Group that produced this document:

Ron Hochsprung, Apple Computer, Inc.

Mitch Bradley, FirmWorks (Chair, IEEE P1275 Working Group)

David Kahn, Sun Microsystems, Inc. (Vice Chair, IEEE P1275 Working Group)

The latest version of this binding may be found in the Bus Supplements area of the Web page at:

`http://playground.sun.com/pub/1275`

Revision History

Revision 0.1	**Oct. 7, 1993**	First revision distributed outside of task group (the number 0.1 did not appear on the cover).
Revision 0.2	**Oct. 28, 1993**	Changed the designator for 64-bit memory space from "M" to "x". Changed the parts of the specification related to PCI-to-PCI bridges to reflect the 0.4 bridge architecture spec.
Revision 1.0	**April 14, 1994**	Changed references from P1275 to Open Firmware. Changed size of fields for I/O address representations to reflect PCI architecture.
Revision 1.1	**June 28, 1994**	Added 't'-bit for aliasing, and discussion of "hard-decode" cases.
Revision 1.2	**August 7, 1994**	Added note about DD encoding. Added new standard properties for those of draft Rev 2.1 PCI spec. Deleted enabling of Memory space at post-probe. Added driver encapsulation description.
Revision 1.3	**September 27, 1994**	Changed generated name for Subsystem, if present. Added rule for I/O assignment. Added discussion of PCI-to-PCI bridge probing.
Revision 1.4	**December 16, 1994**	Added Expansion ROM address assignment, `t` bit for "below 1 MB".
Revision 1.5	**March 20, 1995**	Applied proposals 241, 242, 249, 264, 273. Added Legacy devices section, "clock-frequencey" property, clarified address assignment, added 't' bit for relocatable I/O space.
Revision 1.6	**October 12, 1995**	Applied proposals 274,286, 287 and 290.

1. Overview and References

This specification describes the application of Open Firmware to computer systems that use the PCI Local bus as defined in *PCI Local Bus Specification, Revision 2.1, June 1, 1995* and *PCI-to-PCI Bridge Architecture Specification, Revision 1.0 April 5, 1994* and is to be used in conjunction with those documents.

1.1. Definitions of Terms

bus controller: a hardware device that implements a PCI bus.

hard decode: a decoding which is not based upon a base register, but, rather, is fixed.

PCI device: a hardware device that connects to or "plugs in" to a PCI bus PCI function: one of a number of logically-independent parts of a PCI device. Many PCI devices have only one function per device; in such cases, the terms "PCI function" and "PCI device" can be used interchangeably.

PCI-to-PCI bridge: a hardware device that is, from an electrical standpoint, a single PCI function on one PCI bus (the "parent" bus) and the bus controller of a secondary PCI bus (the "child" bus).

PCI domain: a group of PCI buses connected together in a tree topology by PCI-to-PCI bridges.

relocatable region: a range of PCI address space whose base address is established by a single base address register.

Master PCI bus: within a PCI domain, the PCI bus that forms the root of the tree structure.

bus node: an Open Firmware device node that represents a bus controller. In cases where a node represents the interface, or "bridge", between one bus and another, the node is both a bus node relative to the bus it controls, and a child node of its parent bus. Note that an Open Firmware device node is not in itself a physical hardware device; rather, it is a software abstraction that describes a hardware device.

child node: an Open Firmware device node that represents a PCI function. Such a node can correspond to either a device that is "hardwired" to a planar PCI bus, or to an "add in" expansion card that is plugged into a standard PCI Expansion Connector.

2. Bus Characteristics

2.1. Address Spaces

PCI has several address spaces (Memory, I/O, Configuration), with different addressing characteristics.

2.1.1. Memory Space

Memory Space is the primary address space of PCI; it corresponds to traditional memory and "memory-mapped" I/O. PCI allows for a full 64-bit address range in Memory Space; however, most devices will not require a full 64-bit range. In order to provide compatibility between devices designed for 64-bit addressing and those for 32-bit addressing, the 32-bit address space appears as the first 4 GB region of the 64-bit space; i.e., 64-bit addresses with the 32 most-significant bits equal to 0 are used to access 32-bit devices. 64-bit initiators are required to use the 32-bit address protocol for any 64-bit address in which the upper 32 bits are all 0.

The PCI specification requires that all of a device's relocatable resources must be mappable in Memory Space, i.e. it is not permissible for a resource to be mappable only in I/O Space (described below).

The regions of Memory Space to which a PCI device responds are assigned dynamically during system initialization, by setting device base address registers in Configuration Space (see below). The size of each such region must be a power of two, and the assigned base address must be aligned on a boundary equal to the size of the region.

The encoding of the base address registers for Memory Space allows a resource to require address allocation within the first 1 MB. This requirement is reflected in the **`"reg"`** property entry for that base register by having the `'t'` bit set.

Memory Space addressing is "flat" across a PCI domain, in that addresses are not transformed as they cross PCI-to-PCI bridges. The flat address space is not necessarily limited to a single PCI domain; the PCI design attempts to make it possible to have a flat address across multiple PCI domains that are peers of one another on a higher-level host address bus.

An early revision of the PCI specification admitted the possibility that some devices might respond to fixed (non-relocatable) address ranges. The current revision permits this behavior for VGA and IDE devices, but it is possible that some other devices still behave that way. The current specification allows devices to respond to fixed addresses after a system reset, but provides a standard way to disable such response, which devices are required to implement.

2.1.2. I/O Space

I/O Space is similar to Memory Space, except that it is intended as to be used with the special "I/O access" instructions that some processors have. As with Memory Space, I/O Space addresses are assigned dynamically during system initialization, and the addressing is "flat" across a PCI domain.

Relocatable I/O Space *shall* be allocated at addresses of the form `aaaa.aaaa.aaaa.aaaa.aaaa.aa00.aaaa.aaaa`. This guarantees that relocatable I/O addresses will not conflict with hard-decoded address that have non-zero bits in AD[9…8]. Because PCI-to-PCI bridges restrict I/O address space to 16 bits, relocatable I/O Space across PCI-to-PCI bridges *shall* be of the form `0000.0000.0000.0000.aaaa.aa00.aaaa.aaaa.`

Note: although the PCI specification allows 32-bit I/O Space addresses, many of the processors that have special I/O access instructions present only a 16-bit I/O address. However, Open Firmware allows for the specification of the full 32-bit range.

PCI also allows devices to have I/O base address registers that implement only the low-order 16 bits. I.e., the upper 16 bits are assumed to be 0. When probing, after writing all 1s, the data read back will have the high-order 16 bits equal to 0, while the low-order 16 bits will reflect the address space requirement. Address space for such a base register must be allocated within the first 64 KB of I/O Space. This requirement is reflected in the **"reg"** property for that base register by having the `'t'`-bit set. This is interpretation of the `'t'`-bit for I/O Space is distinquished from the "alias" case by having the `'n'`-bit equal to 0 in its **"reg"** entry; the corresponding **"assigned-addresses"** entry *shall* have the `'t'`-bit equal to 0.

2.1.3. Hard-decoded Spaces

PCI allows devices to "hard-decode" Memory and I/O addresses; i.e., the addresses are not subject to relocation via a base register. These address ranges are represented by having the non-relocatable bit (`'n'`) set in their corresponding **"reg"** and **"assigned-addresses"** properties, where the base-register field of the *phys.hi* is 0.

Furthermore, such devices are allowed to "alias" their hard-decoded I/O addresses by ignoring all but the lower 10 bits of an I/O address. To conserve **"reg"** property space, a bit (the `'t'`-bit, for ten-bit) is included in the encoding of hard-decoded (non-relocatable, `'n'`-bit = 1) I/O address **"reg"** and **"assigned-addresses"** entries to indicated that the address range includes all such aliases.

2.1.4. Configuration Space

Configuration Space is used primarily during device initialization. Each device contains a set of Configuration Registers which are used to identify and configure the device. Configuration Cycles access a device's Configuration Registers, including the "address base registers" which must be initialized before the device will respond to Memory and I/O Space accesses.

In contrast to Memory and I/O Space addressing, Configuration Space addressing is effectively "geographical", in that the Configuration Space address of a particular device is determined by its physical location on a PCI bus (i.e. the slot in which it is installed), or more generally, its physical location within a "tree" of interconnected PCI-to-PCI bridges.

The method for generating Configuration Cycles is system-dependent. In some systems, special registers are used to generate Configuration Space cycles. In other systems, Configuration Space might be memory-mapped as a region within a large physical address space. In particular, the hardware method for specifying the Bus Number and Device Number is system-dependent. Bus Number and Device Number are described below as though they are binary-encoded fields within an address; in practice, that is not necessarily true at the hardware level. However, the representation described below is adequate as an internal software representation, because it is capable of representing the entire possible space of PCI Configuration Space addresses.

A Configuration Space address consists of several fields:

2.1.4.1. Bus Number: 8 bits.

Each PCI bus within a PCI domain is assigned a unique identifying number, known as the "bus number". The assignments occur during system initialization, when the bus controllers for the PCI buses within the PCI domain are located. The bus number for a particular bus is written into a register in that bus's bus controller.

During a Configuration Cycle, each bus controller compares the bus number field of the address to its assigned bus number. If they match, the bus controller selects one of the devices on its PCI bus, according to the value of the Device Number field. Otherwise, the bus controller either forwards the configuration cycle to its subordinate PCI-to-PCI bridges (if the bus number is for one of its subordinate bridges) or ignores the cycle.

2.1.4.2. Device Number: 5 bits

During a Configuration Cycle, the bus controller selected by the bus number field decodes the Device Number field, activating the single corresponding "IDSEL" device select line to enable one of the PCI devices on that bus. For PCI buses with plug-in slots, the Device Number field effectively selects a particular slot. Electrical limitations restrict the number of devices on an individual PCI bus to fewer than the 32 that could otherwise be selected by this 5-bit field.

Some PCI bus controllers use the same physical wires for the IDSEL lines and higher-numbered address lines, thus, on the bus that is selected by the bus number field, the Device Number does not appear on the address bus in its 5-bit binary-encoded form. Rather, the 5-bit field is decoded to a "one of n" select that asserts exactly one upper address line. This fact does not affect the logical representation of the Device Number as a 5-bit binary-encoded field.

Note: the decoding mechanism (e.g., the address bit selected) from the Device Number is system dependent. Furthermore, the implementation of the Open Firmware **`config-xx`** *words can "hide" this detail. However, it is recommended that an Open Firmware implementation choose a numbering which is meaningful to the user.*

2.1.4.3. Function Number: 3 bits

Each PCI device can have from one to eight logically-independent functions, each with its own independent set of configuration registers. A PCI device that is selected during a Configuration Cycle decodes the Function Number field to select the appropriate set of configuration registers for that function. The assignment of Function Numbers to particular functions is a hard-wired characteristic of the individual PCI device. For a PCI device with only one function, the Function Number must be zero.

2.1.4.4. Register Number: 8 bits

The register number field, decoded by the PCI device, selects a particular register within the set of configuration registers corresponding to the selected function. The layout (locations and meanings of particular bits) of the first few configuration registers (i.e. those with small register numbers) is specified by the PCI standard; other configuration registers are device-specific. The standard configuration registers perform such functions as assigning Memory Space and I/O Space base addresses for the device's addressable regions.

In many PCI hardware implementations, Configuration Space does not appear as a direct subset of the system's physical address space; instead, Configuration Space accesses are performed by a sequence of reads or writes to special system registers.

2.1.4.5. "Address-less" Cycles

In addition to these address spaces, PCI has two types of transactions in which the address bus is not used. Special

Cycles (writes) are "broadcast" cycles in which the data conveys all of the information. Interrupt Acknowledge Cycles (reads) are intended to support interrupt control hardware associated with PCI devices. The PCI specification does not specify the details of such interrupt control hardware.

2.1.4.6. Low-order Address Bits

The address characteristics described above do not take into account the way that the PCI bus uses the least-significant two address bits. In general, at the hardware level, the PCI bus uses the two low address bits (AD[1::0]) not to identify the particular byte to be accessed, but instead to convey additional information about the data transfer, such as the type of address incrementing for burst transfers. The bytes are selected with "byte enable" signals.

That hardware subtlety is irrelevant for the purposes of this specification; within the Open Firmware domain, addresses identify individual 8-bit, 16-bit, and 32-bit registers or memory locations in the usual way. Within this document, "address" refers to that software view of an address, which in the case of the two lower address bits is not necessarily the same as what is on the PCI address wires.

2.2. Address Formats and Representations

2.2.1. Physical Address Formats

2.2.1.1. Numerical Representation

(The Numerical Representation of an address is the format that Open Firmware uses for storing an address within a property value and on the stack, as an argument to a package method.) The numerical representation of a PCI address consists of three cells, encoded as follows. For this purpose, the least-significant 32 bits of a cell is used; if the cell size is larger than 32 bits, any additional high-order bits are zero. Bit# 0 refers to the least-significant bit.

```
               Bit#     33222222 22221111 11111100 00000000
                        10987654 32109876 54321098 76543210

phys.hi cell:   npt000ss bbbbbbbb dddddfff rrrrrrrr
phys.mid cell:  hhhhhhhh hhhhhhhh hhhhhhhh hhhhhhhh
phys.lo cell:   llllllll llllllll llllllll llllllll
```

where:

`n`	is 0 if the address is relocatable, 1 otherwise
`p`	is 1 if the addressable region is "prefetchable", 0 otherwise
`t`	is 1 if the address is aliased (for non-relocatable I/O), below 1 MB (for Memory), or below 64 KB (for relocatable I/O).
`ss`	is the space code, denoting the address space
`bbbbbbbb`	is the 8-bit Bus Number
`ddddd`	is the 5-bit Device Number
`fff`	is the 3-bit Function Number
`rrrrrrrr`	is the 8-bit Register Number
`hh...hh`	is a 32-bit unsigned number
`ll...ll`	is a 32-bit unsigned number

When the `hh...hh` and `ll...ll` fields are concatenated to form a larger number, `hh...hh` is the most significant portion and `ll...ll` is the least significant portion.

The '`p`' bit reflects the state of the "P" bit in the corresponding hardware Base Address register.

Encoding of type code "ss":

`00` denotes Configuration Space, in which case:

	`n`	must be zero
	`p`	must be zero
	`t`	must be zero
		`bbbbbbbb,ddddd,fff,rrrrrrrr`is the Configuration Space address
		`hh...hh,ll...ll`must be zero

`01` denotes I/O Space, in which case:

	`p`	must be zero
	`t`	is set if 10-bit aliasing is present (for non-relocatable), or below 64 KB required (for relocatable).
	`bbbbbbbb,ddddd,fff,rrrrrrrr`	identifies the region's Base Address Register `rrrrrrrr` can be 0x10, 0x14, 0x18, 0x1c, 0x20 or 0x24 (for relocatable). `rrrrrrrr`is 0x00 for non-relocatable
	`hh...hh`	must be zero
If `n` is 0:	`ll...ll`	is the 32-bit offset from the start of the relocatable region of I/O Space
If `n` is 1:	`ll...ll`	is the 32-bit I/O Space address

`10` denotes 32-bit-address Memory Space, in which case:

	`p`	may be either 0 or 1
	`t`	is set if an address below 1 MB is required
	`bbbbbbbb,ddddd,fff,rrrrrrrr`	identifies the relocatable region's Base Address Register `rrrrrrrr` can be 0x10, 0x14, 0x18, 0x1c, 0x20, 0x24, 0x30 (relocatable) `rrrrrrrr` is 0x00 for non-relocatable
	`hh...hh`	must be zero
If `n` is 0:	`ll...ll`	is the 32-bit offset from the start of the relocatable region of 32-bit address Memory Space
If `n` is 1:	`ll...ll`	is the 32-bit Memory Space address

`11` denotes 64-bit-address Memory Space, in which case:

	`p`	may be either 0 or 1
	`t`	must be 0
	`bbbbbbbb,ddddd,fff,rrrrrrrr`	identifies the first register of the relocatable region's Base Address Register pair. `rrrrrrrr` can be 0x10, 0x14, 0x18, 0x1c, or 0x20
If `n` is 0:	`hh...hh,ll...ll`	is the 64-bit offset from the start of the relocatable region of 64-bit address Memory Space to the start of the subregion
If `n` is 1:	`hh...hh,ll...ll`	is the 64-bit Memory Space address

Note: Although the bit format of the phys.hi cell is generally consistent with the bit format of a particular kind of hardware mechanism for Configuration Space access that is recommended by the PCI standard, the use of that format does not imply that the hardware must use the same format. The numerical representation specified herein contains the information needed to select a particular hardware device, specifying the format by which that information is communicated among elements of Open Firmware firmware and client programs. A driver for a particular PCI bus hardware implementation is free to extract that information and reformat as necessary for the hardware.

Note: Although the PCI Local Bus Specification *defines both prefetchable and non-prefetchable 64-bit-address Memory Space, the* PCI-to-PCI Bridge Architecture Specification *does not specify a standard means of supporting non-prefetchable 64-bit-address Memory Space across PCI-to-PCI bridges.*

2.2.1.2. Text Representation

The text representation of a PCI address is one of the following forms:

```
DD
DD,F
[n]i[t]DD,F,RR,NNNNNNNN
[n]m[t][p]DD,F,RR,NNNNNNNN
[n]x[p]DD,F,RR,NNNNNNNNNNNNNNNN
```

where:

DD	is an ASCII hexadecimal number in the range 0…1F
F	is an ASCII numeral in the range 0…7
RR	is an ASCII hexadecimal number in the range 0…FF
NNNNNNNN	is an ASCII hexadecimal number in the range 0…FFFFFFFF
NNNNNNNNNNNNNNNN	is an ASCII hexadecimal number in the range 0…FFFFFFFFFFFFFFFF
[n]	is the letter 'n', whose presence is optional
[t]	is the letter 't', whose presence is optional
[p]	is the letter 'p', whose presence is optional
i	is the letter 'i'
m	is the letter 'm'
x	is the letter 'x'
,	is the character ',' (comma)

The correspondence between the text representations and numerical representation is as follows:

DD

corresponds to a Configuration Space address with the numerical value:
ss is 00
bbbbbbbb is the parent's bus number
ddddd is the binary encoding of DD
fffis zero
rrrrrrrr is zero
hh…hh is zero
ll…ll is zero

DD,F

corresponds to a Configuration Space address with the numerical value:
ss is 00
bbbbbbbb is the parent's bus number
ddddd is the binary encoding of DD
fffis the binary encoding of F
rrrrrrrris zero
hh…hh is zero
ll…ll is zero

[n]i[t]DD,F,RR,NNNNNNNN

corresponds to a relocatable (if 'n' is not present) or a non-relocatable (if 'n' is present) 32-bit I/O Space address with the numerical value. If 't' is present, only the low-order 10 bits of an I/O address range is indicated; aliases are assumed to for all high-order bits. The numerical value is:
ss is 01
bbbbbbbb is the parent's bus number
ddddd is the binary encoding of DD
fffis the binary encoding of F
rrrrrrrr is the binary encoding of RR
hh…hh is zero
ll…ll is the binary encoding of NNNNNNNN

`[n]m[t][p]DD,F,RR,NNNNNNNN`

corresponds to a relocatable (if 'n' is not present) or a non-relocatable (if 'n' is present) 32-bit Memory Space address. If 't' is present, the address is within the first 1 MB of memory address space. The the numerical value is:

`ss` is `10`
`bbbbbbbb` is the parent's bus number
`ddddd` is the binary encoding of `DD`
`fff`is the binary encoding of `F`
`rrrrrrrr`is the binary encoding of `RR`
`hh...hh` is zero
`ll...ll` is the binary encoding of `NNNNNNNN`

`[n]x[p]DD,F,RR,NNNNNNNNNNNNNNNN`

corresponds to a relocatable (if 'n' is not present) or a non-relocatable (if 'n' is present) 64-bit Memory Space address with the numerical value:

`ss` is `10`
`bbbbbbbb` is the parent's bus number
`ddddd` is the binary encoding of `DD`
`fff`is the binary encoding of `F`
`rrrrrrrr` is the binary encoding of `RR`
`hh...hh,ll...ll` is the binary encoding of `NNNNNNNNNNNNNNNN`

Conversion of hexadecimal numbers from text representation to numeric representation shall be case-insensitive, and leading zeros shall be permitted but not required.

Conversion from numeric representation to text representation shall use the lower case forms of the hexadecimal digits in the range `a...f`, suppressing leading zeroes, and the `DD` form shall be used for Configuration Space addresses where `fff` is zero.

2.2.1.3. Unit Address Representation

As required by this specification's definition of the **"reg"** property, a function's "unit-number" (i.e. the first component of its "**reg**" value) is the Configuration Space address of the function's configuration registers. Since the "unit-number" is the address that appears in an Open Firmware 'device path', it follows that only the `DD` and `DD,FF` forms of the text representation can appear in a 'device path'.

Note: Since the predominant use of the text representation is within 'device paths', text representations of I/O and Memory Space addresses are rarely seen by casual users.

Note: The bus number does not appear in the text representation. If the bus number were present, then the pathname of a particular device would depend on the particular assignment of bus numbers to bus controllers, so the pathname could change if PCI-to-PCI bridges were added or removed from other slots. (It is generally undesirable for the pathname of a particular device to depend on the presence or absence of other devices that are not its ancestors in the device tree.) The combination of a device node's position in the device tree, its Device Number and its Function Number uniquely select an individual function based on physical characteristics of the system, so the function's pathname does not change unless the device is physically moved.

Note: The bus number appears in the numerical representation because that makes it easier to implement Configuration Space access methods. The ***decode-unit*** *method automatically inserts the bus number in the numerical representation; it can do so because each bus node knows the bus number of the bus it represents.*

Open Firmware implications:

Since some processors cannot generate I/O cycles directly, I/O Space accesses must be done with the register access words (e.g., **rb@**, **rw!**).

It is recommended that a range of virtual addresses be set aside for use by **map-in** to I/O Space devices so that the register access words can determine when an I/O cycle needs to be generated.

Since Configuration Space often does not appear as a subset of the system's physical address space, this firmware specification provides bus-specific methods to access Configuration Space. Likewise, it provides methods for

Special Cycles and Interrupt Acknowledge Cycles.

2.3. Bus-specific Configuration Variables

An Open Firmware-compliant User Interface on a system with a single built-in PCI bus *may* implement the following PCI-specific Configuration Variable.

`pci-probe-list` (-- list-str list-len) N

Holds list of slots to probe with **`probe-pci`** .

A configuration variable containing a string, formatted as described in the following section, indicating the set of slots that will be probed when **`probe-pci`** is executed. The maximum length shall be sufficient to contain a string listing all of the PCI bus's implemented slots.

Configuration Variable Type: string

Default value: a system-dependent value that includes all available slots, in numerically-ascending order.

Note: **`pci-probe-list`** *is intended for the common case of a system with a single built-in PCI bus. On systems with multiple PCI buses, fine-grained control over the probe order can be achieved by repeated execution of the* **`probe-self`** *method within individual bus nodes. In any case, the ability to control the probe order is primarily intended as a convenience when debugging faulty expansion cards. Normally, the default probe order (all available slots) is used.*

2.4. Format of a Probe List

A PCI probe list is a text string consisting of a series of lower-case hexadecimal numbers separated by commas. Each number is in the range `0…1F`, corresponding to the slot with the same Device Number. For a given PCI bus implementation, only the numbers corresponding to existing slots are valid.

The first number in the series specifies the first slot to be probed, and so on.

2.5. FCode Evaluation Semantics

See the description of **`probe-pci`** for the precise specification of the FCode evaluation semantics.

3. Bus Nodes

Note: A PCI-to-PCI bridge is a parent of one PCI bus and a child of another. Consequently, a device node representing a PCI bridge is both a Bus Node and a Child Node, with both sets of properties and methods.

3.1. Properties

3.1.1. Open Firmware-defined Properties for Bus Nodes

The following standard properties, as defined in Open Firmware, have special meanings or interpretations for PCI.

`"device_type"` S

Standard *prop-name* to specify the implemented interface. prop-encoded-array: a string encoded with **`encode-string`**.

The meaning of this property is as defined in Open Firmware. A Standard Package conforming to this specification and corresponding to a device that implements a PCI bus shall implement this property with the string value "`pci`".

`"#address-cells"` S

Standard *prop-name* to define the number of cells necessary to represent a physical address.

prop-encoded-array: Integer constant 3, encoded with **`encode-int`**.

The value of **`"#address-cells"`** for PCI Bus Nodes is 3.

`"#size-cells"` S

Standard *prop-name* to define the number of cells necessary to represent the length of a physical address range.

prop-encoded-array: Integer constant 2, encoded as with **`encode-int`**.

The value of **`"#size-cells"`** for PCI Bus Nodes is 2, reflecting PCI's 64-bit address space.

"reg" S

Standard *prop-name* to define the package's unit-address.

For nodes representing PCI-to-PCI bridges, the **"reg"** property is as defined for PCI Child Nodes. The value denotes the Configuration Space address of the bridge's configuration registers.

For bridges from some other bus to PCI bus, the **"reg"** property is as defined for that other bus.

"ranges" S

Standard *prop-name* to define the mapping of parent address to child address spaces.

This property *shall* be present for all PCI bus bridges. In particular, for PCI-to-PCI bridges, this property *shall* indicate the settings of the mapping registers, thus representing the addresses to which the bridge will respond. For PCI-to-PCI bridges, there shall be an entry in the **"ranges"** property for each of the Memory, Prefetchable Memory and/or I/O spaces if that address space is mapped through the bridge.

3.1.2. Bus-specific Properties for Bus Nodes

"clock-frequency" S

prop-name, denotes frequency of PCI clock.

prop-encoded-array: An integer, encoded as with **encode-int**, that represents the clock frequency, in hertz, of the PCI bus for which this node is the parent.

"bus-range" S

prop-name, denotes range of bus numbers controlled by this PCI bus.

prop-encoded-array: Two integers, each encoded as with **encode-int**, the first representing the bus number of the PCI bus implemented by the bus controller represented by this node (the *secondary bus* number in PCI-to-PCI bridge nomenclature), and the second representing the largest bus number of any PCI bus in the portion of the PCI domain that is subordinate to this node (the *subordinate bus* number in PCI-to-PCI bridge nomenclature).

"slot-names" S

prop-name, describes external labeling of add-in slots.

prop-encoded-array: An integer, encoded as with **encode-int**, followed by a list of strings, each encoded as with **encode-string**.

The integer portion of the property value is a bitmask of available slots; for each add-in slot on the bus, the bit corresponding to that slot's Device Number is set. The least-significant bit corresponds to Device Number 0, the next bit corresponds to Device Number 1, etc. The number of following strings is the same as the number of slots; the first string gives the label that is printed on the chassis for the slot with the smallest Device Number, and so on.

3.2. Methods

3.2.1. Open Firmware-defined Methods for Bus Nodes

A Standard Package implementing the `"pci"` device type shall implement the following standard methods as defined in Open Firmware, with the physical address representations as specified in section 2.1 of this standard, and with additional PCI-specific semantics:

open (-- okay?) Prepare this device for subsequent use

close (--) Close this previously-open device

map-in (phys.low phys.mid phys.hi size -- virt) Map the specified subregion.

PCI-to-PCI bridges pass through addresses unchanged. Consequently, a PCI-to-PCI bridge node's implementation of **map-in** typically just forwards the request to its parent.

For a master PCI bus node in "probe state", if the physical address is relocatable, the **map-in** method shall assign a base address and set the appropriate base address register to that address. Such "probe state" assignments are temporary and are not necessarily valid after the corresponding **map-out**.

map-out (virt size --) Destroy mapping from previous map-in

PCI-to-PCI bridges pass through addresses unchanged. Consequently, a PCI-to-PCI bridge node's implementation of **map-out** typically just forwards the request to its parent.

For a master PCI bus node in "probe state", if the physical address is relocatable and there are no other active mappings within the relocatable region containing that address, the **map-out** method shall unassign the base address of the region, freeing the corresponding range of PCI address space for later re-use. A Standard FCode program shall unmap (as with **map-out**) all base addresses that it mapped and shall disable memory or I/O space access in the Command Register.

dma-alloc (size -- virt) Allocate a memory region for later use

dma-free (virt size --) Free memory allocated with dma-alloc

dma-map-in (.. virt size cacheable? -- devaddr) Convert virtual address to device bus DMA address.

dma-map-out (virt devaddr size --) Free DMA mapping set up with dma-map-in

dma-sync (virt devaddr size --) Synchronize (flush) DMA memory caches

probe-self	(arg-str arg-len reg-str reg-len fcode-str fcode-len --)	Interpret FCode, as a child of this node
decode-unit	(addr len -- phys.lo phys.mid phys.hi)	Convert text representation of address to numerical representation
encode-unit	(phys.lo phys.mid phys.hi -- addr len)	Convert numerical representation of address to text representation

Note: The PCI bus is little-endian; i.e. a byte address whose least-significant two bits are both zero uses the bus byte lane containing the least-significant portion of a 32-bit quantity. Typically, a bridge from a big-endian bus to a PCI bus will swap the byte lanes so that the order of a sequence of bytes is preserved when that sequence is transferred across the bridge. As a result, the hardware changes the position of bytes within a 32-bit quantity that is viewed as a 32-bit unit, rather than as a sequence of individually-addressed bytes. In order to properly implement the semantics of the Open Firmware register access words (e.g. ***rl!****), the device node for such byte-swapping bridges must substitute versions of those words that "undo" the hardware byte-swapping.*

3.2.2. Bus-specific Methods for Bus Nodes

3.2.3. Configuration Access Words

The methods described below have execution semantics similar (especially with respect to write-buffer flushing, atomicity, etc.) to those of the register access words (e.g., **rb@**, **rw!**); in most implementations, these methods will be implemented via the register access words.

The data type 'config-addr' refers to the 'phys.hi' cell of the numerical representation of a Configuration Space address. The 'config-addr' shall be aligned to the data type of the access.

config-l@ (config-addr -- data)
Performs a 32-bit Configuration Read.

config-l! (data config-addr --)
Performs a 32-bit Configuration Write.

config-w@ (config-addr -- data)
Performs a 16-bit Configuration Read.

config-w! (data config-addr --)
Performs a 16-bit Configuration Write.

config-b@ (config-addr -- data)
Performs an 8-bit Configuration Read.

config-b! (data config-addr --)
Performs an 8-bit Configuration Write.

assign-package-addresses (phandle --)
Assigns addresses (i.e., creates `"assigned-addresses"` property) for the child node denoted by *phandle*.

3.2.3.1. Address-less Access Words

intr-ack (--)
Performs a PCI Interrupt Acknowledge Cycle.

special-! (data bus# --)
Performs a PCI Special Cycle on the indicated `bus#`.

Note: Standard PCI-to-PCI bridges provide a mechanism for converting Configuration Cycles with particular addresses to Special Cycles. Consequently, for a PCI-to-PCI bridge, the likely implementation of **special-!** *involves invoking the parent node's* **config-l!** *method.*

4. Child Nodes

Note: A PCI-to-PCI bridge is a parent of one PCI bus and a child of another. Consequently, a device node representing a PCI bridge is both a Bus Node and a Child Node, with both sets of properties and methods.

4.1. Properties

4.1.1. Open Firmware-defined Properties for Child Nodes

The following properties, as defined in Open Firmware, have special meanings or interpretations for PCI.

"reg" S

Standard *prop-name*, defines device's addressable regions.

prop-encoded-array: Arbitrary number of (*phys-addr size*) pairs.

phys-addr is (*phys.lo phys.mid phys.hi*), encoded as with **encode-phys**. *size* is a pair of integers, each encoded as with **encode-int** .

The first integer denotes the most-significant 32 bits of the 64-bit region size, and the second integer denotes the least significant 32 bits thereof.

This property is mandatory for PCI Child Nodes, as defined by Open Firmware. The property value consists of a sequence of (*phys-addr, size*) pairs. In the first such pair, the *phys-addr* component shall be the Configuration Space address of the beginning of the function's set of configuration registers (i.e. the `rrrrrrrr` field is zero) and the *size* component shall be zero. Each additional (*phys-addr, size*) pair shall specify the address of an addressable region of Memory Space or I/O Space associated with the function. In these pairs, if the "n" bit of phys.hi is 0, reflecting a relocatable address, then *phys.mid* and *phys.lo* specify an address relative to the value of the associated base register. In general this value will be zero, specifying an address range corresponding directly to the hardware's. If the "n" bit of phys.hi is 1, reflecting a non-relocatable address, then *phys.mid* and *phys.hi* specify an absolute PCI address.

In the event that a function has an addressable region that can be accessed relative to more than one Base Address Register (for example, in Memory Space relative to one Base Register, and in I/O Space relative to another), only the primary access path (typically, the one in Memory Space) shall be listed in the **"reg"** property, and the secondary access path shall be listed in the **"alternate-reg"** property.

Note: The device FCode is free to construct the second and later pairs in any order, including or omitting references to base address registers, hard-decoded registers, and so on. However, for compatibility between FCode and earlier non-FCode versions of a device and between Open Firmware and non-Open Firmware systems, it is recommended that the device FCode construct the `"reg"` property exactly *as the platform firmware would have in the absence of device FCode, as described under **probe-pci**.*

"interrupts" S

prop-name, the presence of which indicates that the function represented by this node is connected to a PCI expansion connector's interrupt line.

prop-encoded-array: Integer, encoded as with **encode-int**. The integer represents the interrupt line to which this function's interrupt is connected; INTA=1, INTB=2, INTC=3, INTD=4. This value is determined from the contents of the device's Configuration Interrupt Pin Register.

4.1.2. Bus-specific Properties for Child Nodes

Standard Packages corresponding to devices that are children of a PCI bus shall implement the following properties, if applicable.

"alternate-reg" S

prop-name, defines alternate access paths for addressable regions.

prop-encoded-array: Arbitrary number of (*phys-addr size*) pairs.

phys-addr is (*phys.lo phys.mid phys.hi*), encoded as with **encode-phys.** *size* is a pair of integers, each encoded as with **encode-int** .

The first integer denotes the most-significant 32 bits of the 64-bit region size, and the second integer denotes the least significant 32 bits thereof.

This property describes alternative access paths for the addressable regions described by the **"reg"** property. Typically, an alternative access path exists when a particular part of a device can be accessed either in Memory Space or in I/O Space, with a separate Base Address register for each of the two access paths. The primary access paths are described by the **"reg"** property, and the secondary access paths, if any, are described by the **"alternate-reg"** property.

If the function has no alternative access paths, the device node shall have no **"alternate-reg"** property. If the device has alternative access paths, each entry (i.e. each *phys-addr size* pair) of its value represents the secondary access path for the addressable region whose primary access path is in the corresponding entry of the **"reg"** property value. If the number of **"alternate-reg"** entries exceeds the number of **"reg"** property entries, the additional entries denote addressable regions that are not represented by **"reg"** property entries, and are thus not intended to be used in normal operation; such regions might, for example, be used for diagnostic functions. If the number of **"alternate-reg"** entries is less than the number of **"reg"** entries, the regions described by the extra **"reg"** entries do not have alternative access paths. An **"alternate-reg"** entry whose *phys.hi* component is zero indicates that the corresponding region does not have an alternative access path; such an entry can be used as a "place holder" to preserve the positions of later entries relative to the corresponding **"reg"** entries. The first **"alternate-reg"** entry, corresponding to the **"reg"** entry describing the function's Configuration Space registers, shall have a *phys.hi* component of zero.

"fcode-rom-offset" S

prop-name, denotes offset of FCode image within the device's Expansion ROM.

prop-encode-array: one integer, encoded as with **encode-int**.

This property indicates the offset of the PCI Expansion ROM image within the device's Expansion ROM in which the FCode image was found; i.e., the offset of the 0xAA55 signature of that image's PCI Expansion ROM Header. This value *shall* be generated before the FCode is evaluated. Note that the absence of this property indicates that no FCode exists for this device node.

"assigned-addresses" S

prop-name, denotes assigned physical addresses

prop-encoded-array: Zero to six (*phys-addr size*) pairs.

phys-addr is (*phys.lo phys.mid phys.hi*), encoded as with **encode-phys**. *size* is a pair of integers, each encoded as with **encode-int** .

The first integer denotes the most-significant 32 bits of the 64-bit region size, and the second integer denotes the least significant 32 bits thereof.

Each entry (i.e. each *phys-addr size* pair) in this property value corresponds to either one or two (in the case 64-bit-address Memory Space) of the function's Configuration Space base address registers. The entry indicates the physical PCI domain address that has been assigned to that base address register (or register pair), and the size in bytes of the assigned region. The size shall be a power of two (since the structure of PCI Base Address registers forces the decoding granularity to powers of two). The 'n' bit of each *phys-addr* shall be set to 1, indicating that the address is absolute (within the PCI domain's address space), not relative to the start of a relocatable region. The type code shall not be '00' (Configuration Space). The 'bbbbbbbb,ddddd,fff,rrrrrrrr' field indicates the base register to which the entry applies, and the 'hh...hh, ll...ll' field contains the assigned address.

If addresses have not yet been assigned to the function's relocatable regions, this property shall be absent.

The values reported in **"assigned-addresses"** represent the physical addresses that have been assigned. If Open Firmware can not assign address space for a resource (e.g., the address space has been exhausted), that resources will not have an entry in the **"assigned-addresses"** property. If no resources were assigned address space, the **"assigned-addresses"** property *shall* have a *prop-encoded-array* of zero length.

Note: There is no implied correspondence between the order of entries in the **"reg"** *property value and order of entries in the* **"assigned-addresses"** *property value. The correspondence between the* **"reg"** *entries and* **"assigned-addresses"** *entries is determined by matching the fields denoting the Base Address register.*

"power-consumption" S

prop-name, describes function's power requirements

prop-encoded-array: list of integers, encoded as with **encode-int**, describing the device's maximum power consumption in microwatts, categorized by the various power rails and the device's power-management state (standby or fully-on). The ints are encoded in the following order:

> unspecified standby, unspecified full-on, +5V standby, +5V full-on, +3.3V standby, +3.3V full-on, I/O pwr standby, I/O pwr full-on, reserved standby, reserved full-on

The "unspecified" entries indicate that the power division among the various rails is unknown. The "unspecified" entries shall be zero if any of the other entries are non-zero. The "unspecified" entries are provided so that the **"power-consumption"** property can be created for devices without FCode, from the information on the PRSNT1# and PRSNT2# connector pins.

If the number of ints in the encoded property value is less than ten, the power consumption is zero for the cases corresponding to the missing entries. For example, if there are four ints, they correspond to the two "unspecified" and the two "+5" numbers, and the others are zero.

4.1.2.1. Standard PCI Configuration Properties

The following properties are created during the probing process, after the device node has been created, but before evaluating the device's FCode (if any). They represent the values of standard PCI configuration registers. This information is likely to be useful for Client and User interfaces.

Unless specified otherwise, each of the following properties has a *prop-encoded-array* whose value is an integer taken directly from the corresponding hardware register, encoded as with **encode-int**.

"vendor-id" S

"device-id" S

"revision-id" S

"class-code" S

"interrupts" S

This property shall be present if the Interrupt Pin register is non-zero, and shall be absent otherwise.

"min-grant" S

"max-latency" S

"devsel-speed" S

"fast-back-to-back" S

prop-encoded-array: <none>

This property shall be present if the "Fast Back-to-Back" bit (bit 7) in the function's Status Register is set, and shall be absent otherwise.

"subsystem-id" S

This property shall be present if the "Subsystem ID" register is non-zero, and shall be absent otherwise.

"subsystem-vendor-id" S

This property shall be present if the "Subsystem Vendor ID" register is non-zero, and shall be absent otherwise.

"66mhz-capable" S

prop-encoded-array: <none>

This property shall be present if the "66 MHz Capable" bit (bit 6) in the function's Status Register is set, and shall be absent otherwise.

"udf-supported" S

prop-encoded-array: <none>

This property shall be present if the "UDF Supported" bit (bit 5) in the function's Status Register is set, and shall be absent otherwise.

4.2. Methods

5. Bus-specific User Interface Commands

An Open Firmware-compliant User Interface on a system with PCI *should* implement the following PCI-specific user interface commands.

probe-pci (--)

Interprets FCode for all built-in PCI slots in numerical order.

Enter "probe state", thus affecting subsequent behavior of the **map-in** and **map-out** methods.

Scan all slots in numerical order. For each slot, read the header type field in the configuration register set for function number 0. If the header type field indicates a PCI-to-PCI bridge, perform the function described in the Probing PCI-to-PCI bridges section. If the header type field indicates a multi-function device, perform the following sequence for each of the functions that are present (as determined by the presence of a non-FFFFh value in the Vendor ID field of the function's Configuration Space header). Otherwise, perform the following sequence for the card's function number 0. The first attempted access to each function *shall* use **lpeek**, because in some systems an attempted access to a non-existent device might result in a processor exception (e.g. a "bus error").

*Note: Although some PCI implementations will not generate processor exceptions for aborted cycles, that is not an inherent limitation of PCI itself, but instead an implementation choice that is appropriate for some system architectures. A PCI host bridge knows if it terminated a cycle with a master abort. Since the PC system architecture lacks the notion of a bus error, PC to PCI host bridges cannot report a bus error to the PC, so they have to complete the cycle and return all ones to the x86 processor. However, in a non-PC system, the PCI host bridge could terminate the processor cycle with a bus error. Open Firmware **peek** and **poke** can behave in their normal way; if the processor can get a bus error, peek and poke can report it. If not, **peek** and **poke** will never say they got a bus error, they will just return whatever data the cycle yielded. This is not a problem, because the probing process involves doing a **peek** and also looking at the data to see if it is right.*

Create the following properties from the information given in the configuration space header.

"vendor-id"

"device-id"

"revision-id"

"class-code"

"interrupts"

"min-grant" (Unless Header Type is 01h)

"max-latency" (Unless Header Type is 01h)

"devsel-speed"

"fast-back-to-back"

"subsystem-id"

"subsystem-vendor-id"

"66mhz-capable"

"udf-supported"

Note: The feasibility of automatically creating the above properties depends on the ability to recognize the configuration header format. At present, there are two such formats - the base format defined by the PCI Local Bus Specification *and the PCI-to-PCI bridge format defined by the* PCI-to-PCI Bridge Architecture Specification. *Those two* *formats are almost, but not entirely, consistent with respect to the fields defined above (in particular, the max-latency and min-grant fields have a different meaning in the bridge header format). If additional formats are defined in the future, then it is possible that firmware written before those formats are defined will not be able to recognize them. The question arises: Should the firmware assume that, with respect to the above fields, new formats are consistent with the existing ones, creating the properties without regard to header type, or should the firmware do nothing if it sees an unrecognized header type. The latter is, in some sense, safer, but it also precludes forwards compatibility, which is a serious deficiency.*

Then determine whether or not the function has a expansion ROM image containing an FCode Program.

Note: The location of the Expansion ROM Base Address Register differs between the two currently-defined header types. Where will it be in future header types? Furthermore, the details of Expansion ROMs on PCI-to-PCI bridges are not specified by the current revision of the PCI-to-PCI bridge spec.

If the function has an FCode Program, evaluate the FCode Program as follows:

Copy the FCode program from expansion ROM into a temporary buffer in RAM and evaluate it as with **byte-load**. (The temporary RAM buffer may be deallocated afterwards.) Set the **"fcode-rom-offset"** property to the offset of the ROM image at which the FCode was found.

When the FCode Program begins execution, **my-address** and **my-space** together return the Configuration Space address of the device's configuration register set.

Note: Since the phys.mid and phys.lo components of Configuration Spaces addresses must be zero, **my-address** *returns a pair of zeros; the interesting phys.hi information, which is necessary for accessing the configuration registers via the* **config-xx** *methods, is returned by* **my-space**.

The FCode Program is responsible for creating the **"name"** and **"reg"** properties.

If the function does not have an FCode Program:

Create the following properties from information in the device's Configuration Space registers:

"reg"Create the following entries, in the order shown:

- An entry describing the Configuration Space for the device.
- For each active base address register, in Configuration Space order, an entry describing the entire space mapped by that base address register (or, register pair). The phys-mid and phys-lo components of these entries are to be zero, and the size components are to be derived by probing the base address register (or register pair).
- If the device supports an Expansion ROM, an entry describing the Expansion ROM base address register, constructed as for normal base address registers.
- If applicable, "legacy" entries described in section 7., in the order shown.

Note: Without FCode, it is not possible to determine whether or not there are multiple base address registers mapping the same resource, so it is not possible to create an "alternate-reg" *property.*

Note: the number of active base address registers depends in part on the header type configuration field; in particular, header types 0x01and 0x81, denoting the standard PCI-to-PCI bridge header format, have at most two base address registers, whereas header types 0x00 and 0x80have up to seven base address registers (including the Expansion ROM's).

"name"Construct a name of the form **pciVVVV,DDDD**. If the Subsystem ID field in the configuration registers for this device is non-zero, VVVV,DDDD *shall* be the Subsystem Vendor ID and Subsystem ID respectively; otherwise VVVV,DDDD *shall* be the value of the Vendor ID and Device ID fields. VVVV and DDDD are ASCII hexadecimal numbers, lower case, without leading zeros.

Create the **"power-consumption"** property from the state of the PRSNT1# and PRSNT2# connector, if possible.

Disable fixed-address response by clearing the Command Register.

After all slots have been so probed, exit "probe state" and assign base addresses (by allocating the address space and setting the base address register) for each distinct base address register (or register pair). For each base address register (or register pair) listed in any child's **"reg"** property, create an **"assigned-addresses"** properties describing those assignments in the corresponding child device nodes.

On each PCI bus within the domain, set the Latency Timer registers for each master to values appropriate for the other devices on that bus, according to the values of the other device's MIN_GNT and MAX_LAT registers.

On each PCI bus within the domain, if all target devices on that bus have "fast back-to-back" capability, set the "fast back-to-back" enable bits in the Command registers of master devices.

make-properties (--)

Create the default PCI properties (as described above for **probe-pci**) for the *current instance*.

This user interface word is intended to be used for debugging FCode within the context of **begin-package** … **end-package**. This word *should* be executed before evaluating the FCode for the node.

assign-addresses (--)

Assign addresses for the *current instance*.

This user interface word is intended to be used for debugging Fcode within the context of **begin-package** … **end-package**. Executing this word causes addresses to be assigned to this node (based on the current **"reg"** property value), creating an **"assigned-addresses"** property reflecting those addresses. This word should be executed after evaluating the FCode for the node.

6. Probing PCI-to-PCI bridges

The recursive algorithm described in this section allows bus number and address assignment to be done in a single depth-first manner. Bus numbers are assigned on the way down the PCI bus hierarchy and addresses are assigned to on the way back up. Another algorithm may be used.

Note: while this is a simple algorithm (e.g., it does not require a "global" address allocation pass), it does not provide the most optimal system-wide address assignment.

If the function is a standard PCI-to-PCI bridge (as indicated by the class code and the header type fields), set the bridge's Primary Bus # register to the bus number of the parent bus, assign the next bus number to that bridge, setting its Secondary Bus # register to that number, set the bridge's Subordinate Bus # register to `0xFF`, and recursively scan the slots of that bridge's subordinate bus.

When that recursive scanning process returns, set the bridge's Subordinate Bus # register to the largest bus number assigned during the recursive scan. At this point, bus numbers have been assigned to all subordinate buses, and addresses have been assigned for all devices on the subordinate buses, for this bridge within the PCI domain. Then, assign addresses to all devices on the Secondary Bus and set the Memory Base, Memory Limit, I/O Base and I/O Limit registers of the bridge to their appropriate values and enable Memory and I/O transactions. Due to the mapping characteristics of PCI-to-PCI bridges, the ranges of Memory addresses for subordinate devices *shall* be aligned to 1 MB boundaries, and the ranges of I/O addresses *shall* be aligned to 4 KB boundaries.

After setting the Memory Base, Memory Limit, I/O Base and I/O Limit registers, create a **`"ranges"`** property that represents the Memory and I/O mappings that have been established.

7. Legacy devices

"Legacy" VGA and IDE devices that are implemented on PCI will typically have the same "hard-decoded" addresses as they did on ISA. Such devices that have FCode can use explicit indication of their address requirements as described above. However, for cards that have no FCode image, Open Firmware *shall* assume the standard address ranges and *shall* create the **`"reg"`** property with these ranges, in addition to any relocatable regions that have base registers.

For VGA devices (class-code = `0x000100` or `0x030000`), the address ranges are:

`0x3B0-0x3BB`	(I/O, aliased; `t`=1)
`0x3C0-0x3DF`	(I/O, aliased; `t`=1)
`0xA0000-0xBFFFF`	(Memory, below 1MB, `t`=1)

For IDE devices (class-code = `0x010100`), the address ranges are:

`0x1F0-0x1F7`	Primary Command Block (I/O)
`0x3F6`	Primary Control Block (I/O)
`0x170-0x17F`	Secondary Command Block (I/O)
`0x376`	Secondary Control Block (I/O)

8. Relationship between PCI Base Registers and Open Firmware Properties

Especially in the case of a PCI device with onboard FCode support, there is no particular relationship between PCI base registers and the **`"reg"`** and **`"assigned-addresses"`** properties. A particular base register may or may not be represented in **`"reg"`** and **`"assigned-addresses"`**, and those properties may contain entries referring to addresses not controlled by any base register. No particular ordering is required in either the **`"reg"`** property or the **`"assigned-addresses"`** property. A client wishing to make use of the addressing information provided by Open Firmware must scan the **`"assigned-addresses"`** property looking for an entry specifying the desired base-register field in its *phys.hi*.

9. ROM Image Format for FCode:

Offset from start of ROM Image	Data
`00h - 01h`	ROM signature field of ROM Header (PCI spec 6.3.1.1)

`02h - 03h`	Pointer to FCode program. This is a 16-bit field that is the offset from the start of the ROM image and points to the FCode Program. The field is in little-endian format. (This field is within the "Reserved for processor-unique data" field of the ROM Header.)
`04h - 17h`	Reserved (remainder of "Reserved for processor-unique data" field of the ROM Header).
`18h - 19h`	"Pointer to PCI Data Structure" field of ROM Header.
`1Ah - FFFFh`	PCI Data Structure (PCI spec 6.3.1.2) with "Code Type" = 1
`38h - FFFFh`	The PCI Data Structure (PCI spec 6.3.1.2), Vital Product Data, and FCode Program can each begin anywhere within this range, in any order. The "Code Type" field of the PCI Data Structure shall have the value "`1`". The FCode Program is as described in Open Firmware; its size is given by the standard Open Firmware FCode Program header. FCode bytes shall appear at consecutive byte addresses.

10. Encapsulated Drivers

This section describes a mechanism which allows the encapsulation of run-time drivers within the standard Open Firmware expansion ROM.

The FCode contained within a PCI card's expansion ROM provides for Open Firmware drivers for the device. To enhance the "plug-and-play" of cards in common system platforms, it is desirable to be able to include run-time drivers within this expansion ROM, thus eliminating the extra step of installing drivers onto the OS boot device.

The information about run-time drivers is encoded as additional standard properties within the device tree. These properties are created by the FCode probe code of the plug-in card, and are used by the OS to locate and load the appropriate driver. Two new properties are defined; they differ as to how the location of the run-time driver is defined.

`"driver,…"` format

This property, encoded as with **`encode-bytes`**, contains the run-time driver.

This format is used when the run-time driver is contained within the FCode image, itself. The value of the property <u>is</u> the encapsulated driver; the `prop-addr,prop-len` reported by the various "get-property" FCodes and/or `getprop` Client interface call represent the location and size of the driver within the device tree's data space. I.e., **`decode-bytes`** could be used to copy the driver into the desired run-time location.

`"driver-reg,…"` format

This property, encoded as with the **`"reg"`** standard property , contains a relative pointer to the run-time driver.

This format is used when the driver is not directly contained within the FCode image, but rather, is located in some other portion of the Expansion ROM. The value is encoded in a **`"reg"`** format, where the address is relative to the expansion ROM's base address. This format conserves device tree (and, FCode) space, but requires the OS to perform the actions of mapping in the Expansion ROM, using the information supplied by this property and the **`"assigned-addresses"`** for the Expansion ROM, and copying the driver, itself.

*Note: the **`"fcode-rom-offset"`** property facilitates the generation of this property within the context of the FCode's image. The driver can be located relative to the ROM image that contains the FCode (but, does not have to be within the FCode, itself) without regard to the location of that ROM image relative to others within the same Expansion ROM. I.e., "self-relocating" images containing encapsulated drivers can be created that can be concatenated with other images without altering any data within an image (except, of course, for the Indicator to properly indicate the last image).*

10.1. Naming conventions

The complete property name for these encapsulated drivers is chosen to allow multiple drivers to co-exist within the expansion ROM. An OS will locate its desired driver by an exact match of its property name among any such **`"driver,"`** (**`"driver-reg,"`**) properties contained within the device tree for this device. The formats of the complete names are:

"driver,*OS-vendor*,*OS-type*,*Instruction-set*"
"driver-reg,*OS-vendor*,*OS-type*,*Instruction-set*"

The OS-vendor component is as defined for device-names; i.e., organizational unique identifier (e.g., stock symbol). The OS-type & Instruction-set components are defined by the OS-vendor. An example would be:

"driver-reg,AAPL,MacOS,PowerPC"

11. Examples of **"reg"** and **"assigned-addresses"** properties

The examples in these sections demonstrate the generation and use of the **"reg"** and **"assigned-addresses"** properties for PCI devices. The first sections demonstrate how **"reg"** and **"assigned-addresses"** properties get created while the last sections show how a Client can use the information to determine PCI addresses of device resources.

In the following examples:

xxxx	represents the bus, device, function numbers as appropriate
hhhhhhhh	represents the high 32 bits of a PCI domain address
llllllll	represents the low 32 bits of a PCI domain address

11.1. Creation of **"reg"** and **"assigned-address"** properties

The following sections describe several scenarios of how **"reg"** and **"assigned-addresses"** properties would get created in various situations.

11.1.1. A single 256-byte address base register, without FCode.

This example device has a single a single 256-byte, non-prefetchable memory range and no Expansion ROM:

Base address register 0x10:

Discovered to be present, and requiring 0x100 bytes of address space, by reading 0xFFFFFF00 after writing 0xFFFFFFFF.

Base address register 0x14:

Not present, as discovered by reading back 0x00000000 after writing 0xFFFFFFFF.

(same for 0x18, 0x1C, 0x20, 0x24, and 0x30)

"reg" property (created during probing)

```
phys_hi  phys_mid phys_lo  size_hi  size_lo
00xxxx00 00000000 00000000 00000000 00000000
02xxxx10 00000000 00000000 00000000 00000100
```

"assigned-addresses" property (created after probing, during address assignment)

```
82xxxx10 00000000 lllll100 00000000 00000100
```

with the resulting base address registers contents:

```
0x10:     lllll100
```

11.1.2. A simple VGA device, without FCode.

This example consists of a simple VGA device, with no relocatable regions and a 4K non-FCode Expansion ROM.

Base address register 0x10:

Discovered to be not implemented by reading back a 0x00000000 after writing 0xFFFFFFFF.

(same for 0x14, 0x18, 0x1C, 0x20, 0x24)

Expansion ROM base address register (0x30):

Discovered to be present, and requiring 0x1000 bytes by reading back a 0xFFFFF000 after writing 0xFFFFFFFF.

"reg" property (created during probing)

```
phys_hi  phys_mid phys_lo  size_hi  size_lo
00xxxx00 00000000 00000000 00000000 00000000
02xxxx30 00000000 00000000 00000000 00001000
81xxxx00 00000000 000003B0 00000000 0000000C
81xxxx00 00000000 000003C0 00000000 00000020
82xxxx00 00000000 000A0000 00000000 00020000
```

"assigned-addresses" property (created after probing, during address assignment)

```
82xxxx30 00000000 lllll000 00000000 00001000
```

with the resulting base address registers:

```
0x30:       lllll000
```

11.1.3. A single 256 resource, Memory and I/O accessible, without FCode

This example consists of a device that has a single resource, requiring 256 bytes of address space, but which is accessible as either I/O or Memory mapping; i.e., it contains two address base registers, one for I/O Space and one for Memory Space. For this example, no Expansion ROM base address register is implemented; hence, there is no FCode.

Base address register 0x10:

Discovered to be a Memory base address register, requiring 256 bytes of address space, by reading back a `0xFFFFFF00` after writing `0xFFFFFFFF`.

Base address register 0x14:

Discovered to be a I/O base address register, requiring 256 bytes of address space, by reading back a `0xFFFFFF01` after writing `0xFFFFFFFF`.

Base address registers 0x18, 0x1C 0x20, 0x24 and 0x30:

Discovered to be not implemented by reading back a `0x00000000` after writing `0xFFFFFFFF`.

"reg" property (created during probing)

```
phys_hi  phys_mid phys_lo  size_hi  size_lo
00xxxx00 00000000 00000000 00000000 00000000
02xxxx10 00000000 00000000 00000000 00000100
01xxxx14 00000000 00000000 00000000 00000100
```

"assigned-addresses" property (created after probing, during address assignment)

```
81xxxx14 00000000 llllll00 00000000 00000100
82xxxx10 00000000 llllll00 00000000 00000100
```

with the resulting base registers:

```
0x10:       llllll00    (a memory address)
0x14:       llllll01    (an I/O address)
```

Note that this platform appears to allocate I/O space first, yielding an assigned-addresses property in a different order from the reg property.

11.1.4. A single 256 resource, Memory and I/O accessible, with FCode.

The same function as in 11.4, but with 4K of FCode that reveals that the first 32 bytes of the registers are unused and the second 32 bytes are used only for diagnostic purposes:

Base address register 0x10:

Discovered to be a Memory base address register, requiring 256 bytes of address space, by reading back a `0xFFFFFF00` after writing `0xFFFFFFFF`.

Base address register 0x14:

Discovered to be a I/O base address register, requiring 256 bytes of address space, by reading back a `0xFFFFFF01` after writing `0xFFFFFFFF`.

Base address registers 0x18, 0x1C 0x20, 0x24:

Discovered to be not implemented by reading back a `0x00000000` after writing `0xFFFFFFFF`.

Expansion ROM base address register (0x30):

Discovered to be present, and requiring `0x1000` bytes of address space, by reading `0xFFFFF000` after writing `0xFFFFFFFF`.

The Expansion ROM is discovered to contain a valid FCode image which is evaluated. This FCode creates its own **"reg"** property, reflecting knowledge of the intended usage of the addressable resource.

"reg" property (created during probing by the FCode program)

```
phys_hi  phys_mid phys_lo  size_hi  size_lo
00xxxx00 00000000 00000000 00000000 00000000
02xxxx10 00000000 00000040 00000000 000000C0
```

"alternate-reg" property (created during probing by the FCode program)

```
00000000 00000000 00000000 00000000 00000000
01xxxx14 00000000 00000040 00000000 000000c0 *1
02xxxx10 00000000 00000020 00000000 00000020 *2
01xxxx14 00000000 00000020 00000000 00000020 *3
```

Notes:

*1 Secondary access to operational registers
*2 Primary access to diagnostic registers
*3 Secondary access to diagnostic registers

"assigned-addresses" property (created after probing, during address assignment)

```
81xxxx14 00000000 llllll00 00000000 00000100
82xxxx10 00000000 llllll00 00000000 00000100
```

with the resulting base registers:

```
0x10:        llllll00    (a memory address)
0x14:        llllll01    (an I/O address)
```

11.2. Computing PCI addresses from "reg" and "assigned-addresses"

The following sections show how a device driver or Client program could use the information provided in the **"reg"** and **"assigned-addresses"** properties.

The phrases "PCI address space" or "PCI domain" indicate an address is the PCI physical address space as used on the PCI bus in question. This PCI physical address space is not necessarily the same as the physical address space in which a processor would access the PCI resource. Especially in the case of PCI I/O space, host PCI bridges (i.e., PCI bridges that are attached to the processor bus) may perform an address translation function. In the following examples, another sequence of steps is, in general, necessary to determine the appropriate processor address to use. The information necessary for these steps is contained in the **"ranges"** properties of bus nodes.

11.2.1. Determining the address of a register of example in Section 11.1.4.

Problem: Given the last device described in Section 11.1.4., determine the physical address associated with the 3rd byte of its operational registers.

1. Extract the second *phys-addr,size* pair from the **"reg"** property. (We know to use the second pair because the first pair is the Configuration Space entry and the device documentation tells us that the second pair is the operational registers.)

2. Note that the 'n' bit is zero, indicating a relocatable region. Note that the `rrrrrrrr` field is `0x10`.

3. Search the **"assigned-addresses"** property for an entry with an `rrrrrrrr` field of `0x10`. Find the second entry.

4. Add the *phys_mid,phys_lo* value from the **"assigned-addresses"** entry to the *phys_mid,phys_lo* value (0x40) from the **"reg"** entry. This yields the physical base (in PCI address space) of the device's operational registers.

5. Add this value to the desired register offset, 3. This yields the (PCI domain's) physical address of the desired register.

11.2.2. Determining the address of a register of example in Section 11.1.2.

Problem: Given the VGA device described above (in Section 11.1.2.), determine the PCI address associated with the sequencer index register (I/O address `0x3C4` in the documentation).

(Of course, we could "just know" that the answer is `0x3C4`, but let's do it according to the book.)

1. Extract the fourth *phys-addr,size* pair from the **"reg"** property. (We know to use the fourth pair because the first pair is the Configuration Space entry, the second pair is the ROM, the third pair is the monochrome I/O range, the fourth pair is the color I/O range, and the fifth pair is the video memory.)

2. Note that the '`n`' bit is one, indicating a non-relocatable region.

3. Add the *phys_mid,phys_lo* values from this **"reg"** entry to the desired register offset, 4. This yields the physical address (in the PCI domain) of the sequencer index register.