Open Firmware

Recommended Practice:

Generic Names

Version 1.2

8/8/96

Unapproved Draft

Published by the Open Firmware Working Group

This document is a voluntary-use recommended practice of the Open Firmware Working Group. The Open Firmware Working Group is an ad hoc committee composed of individuals interested in Open Firmware as defined by IEEE 1275-1994, related standards, and their application to various computer systems.

The Open Firmware Working Group is involved both in IEEE sanctioned standards activities, whose final results are published by IEEE, and in informal recommendations such as this, which are published on the Internet at:

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

Membership in the Open Firmware Working Group is open to all interested parties. The working group meets at regular intervals at various locations. For more information send email to:

`p1275-wg@risc.sps.mot.com`

Revision History

Version 1.0	03/01/96	First version of document.
Version 1.1	04/11/96	Added an authorized-by section to 1.0, extended scope to add who the recommended practice applies to, modified Generic Names List(Guideline 1), and added Generic Names to table 1. Editorial changes were also made.
Version 1.2	08/07/96	Added definitions (Section 2.2). Edited Guile 1 name list (removed 'other', added 'interrupt-controller', 'fddi', 'fcc' & 'atm'). Removed material in Section 3.0., Generic *Names* per proposal #358. Made Section 6.0 an Information Section and removed PCI Class Codes Table 1 (Referenced the PCI binding). Added new Section 7.0, 'Compatibity Information'.

1. Introduction

This *Generic Names* Recommended Practice was authorized by the Open Firmware Working Group accepting Proposal Number 251.

1.1. Purpose

The rules for device naming described in Reference [1] attempt to simultaneously accomplish two objectives:

- To provide a human-readable identifier for use in device paths.
- To provide a unique identification of the device's detailed programming model to allow operating systems to determine which driver to use.

These objectives conflict with one another. For human use within a device path, a brief name that suggests the device's function is preferable. For use in determining an appropriate operating system device driver, an explicit, unique name that includes the manufacturer name and the detailed programming model information is preferable. Attempting to accomplish both objectives with a single name often results in a name that is unsatisfactory for either purpose.

Historically, implementations have often adopted a mixed approach, using brief names for built-in devices and verbose names for plug-in devices. This approach has the problem that the name of a particular device may be different depending on whether the device is built-in or plug-in. Also, because of the tension between the conflicting objectives, different manufacturers chose different trade-offs, some sacrificing uniqueness in favor of human readability and others making the opposite choice.

There is no inherent reason why a single property (**"name"**) must be used for both purposes. The **"compatible"** property already participates in the OS driver selection process, and if the first component of that property's value is an explicit, unique name, precise driver matching will result, even if the **"name"** property's value is a brief *generic* name (e.g. "disk").

Furthermore, the use of *generic* names does not defeat the purpose of unambiguously choosing a a particular device within the device tree through the use of a device path. The fundamental means for ensuring unambiguous node names is the unit-address component (the portion after the "@" character, which matches the first entry of the **"reg"** property's value). The driver-name component (which matches the **"name"** property's value) is inherently unreliable for the purpose of precisely choosing a particular device, due to the possibility of multiple identical devices at a given level of the device tree. Since, in the most general case, the unit-address component must be used to distinguish two devices of the same type, the increased probability of *name space collisions* that would result from the use of *generic* names does not cause loss of functionality. Indeed, it is not unreasonable to think that some users might prefer to distinguish two display devices by the device paths:

/pci/display@2 and /pci/display@4

as compared to:

/pci/IBM,v915 and /pci/number9-723

As already noted, OS driver selection software almost certainly prefers to have the explicit information contained in, for example, IBM,*v915*, but the software can get that information from the **"compatible"** property as easily as from the "name" property.

1.2. Scope

This recommended practice changes the naming conventions for device nodes, affecting plug-in cards, Open Firmware system implementations, and client programs.

Applies to all new recommended practices and system, bus and device bindings. Existing bindings may choose to accept this recommended practice for future revisions as appropriate.

2. References and Definitions

2.1. References

[1] *IEEE Std. 1275-1994, IEEE Standard for Boot(Initialization Configuration)Firmware:Core Requirements and Practices*, published by IEEE.

[2] *PCI Local Bus Specification*, Revision 2.1, June 1, 1995

[3] *PCI Bus binding to: IEEE Std. 1275-1994 (Initialization, Configuration) Firmware*.

2.2. Definitions

device: Normally, a particular hardware implementation of a function. Sometimes, a particular definition of an abstract implementation.

emulated device: A device, usually an older device, with which the device under questionclaims compatibility. The newer device claims to emulate the older.

emulating device: A device, usually a newer device, that claims compatibility with another,usually older, device.

3. *Generic* Names

Because of the difficulties that result from using *name* for two conflicting purposes, and since that dual use is unnecessary, the working group recommends the following guidelines for future devices:

Guideline 1: "name" property values should be *generic*, reflecting the device's function, but not necessarily its precise programming model. If appropriate, the value should be one of the following listed below:

- disk
- tape
- pci
- vme
- sbus
- scsi
- ide
- isa
- keyboard
- display
- mouse

- sound
- ethernet
- token-ring
- fddi
- fcs
- atm
- timer
- memory
- printer
- serial
- rtc
- nvram
- scanner
- interrupt-controllers

The Open Firmware Working Group may define additional *generic* names in future recommended practice documents or binding documents. For devices that do not fit in any existing generic category, the developer may request that the working group establish a new generic name, or may use an explicit name prefaced with a manufacturer name and a comma.

Guideline 2: The **"compatible"** property should be present. Its first component should be an explicit, unique name that identifies the device precisely enough to be able to infer the detailed programming model from that name. The format of that explicit name is *manufacturer,model*, as specified by the **"name"** properties of plug-in devices (see Reference [1], page 162).

4. Affect on OS Driver Selection Code

An operating system that implements the driver searching technique that Reference [1] recommends for the **"compatible"** property (see Reference [1], page 127) is likely to require no changes as a result of this recommended practice. If a **"name"** property has a generic value, the search for a driver matching that generic name is likely to fail, but the next step of the search (using the **"compatible"** property) will succeed, finding the same driver that would have been found had the explicit name been in the **"name"** property.

It is possible that the operating system will find a driver matching the generic name, and that said driver will not be the correct one. However, this is not a new problem, because generic names have already been used in the past for built-in devices. Consequently, an operating system that does not already have a mechanism for resolving or avoiding such *false matches* is likely to have problems eventually, with or without the proliferation of generic names.

The following suggests some possible techniques for dealing with such *false generic matches*.

a) In collections of OS drivers, avoid the use of generic names for the drivers themselves. For example, it is generally unwise to name a driver *ethernet*, since there are many different ethernet adapters with different programming models. Using the generic name *ethernet* to identify only one such driver is presumptuous.

b) Separate the OS's name spaces for drivers for *real* hardware devices and *pseudo-drivers* (collections of support routines that are used by *real* drivers). Some operating systems load such pseudo-drivers using a mechanism similar to the mechanism used for *real* drivers. Pseudo drivers often perform g*eneric* functions that apply equally well to, for example, all ethernet adapters, independent of their low-level programming models. Consequently, it is reasonable to use generic names (e.g. *ethernet*) for such pseudo-drivers. Separating the name spaces of *real* drivers and pseudo-drivers avoids false matches from generic device names to generic psuedo-driver names.

c) Make the OS's driver search mechanism depend upon the device's parent. In other words, separate the OS's driver name spaces so that drivers for devices that attach to, for example, PCI bus can be distinguished from those that attach to, for example, ISA bus. This reduces the range over which *false matches* can occur.

d) For cases where false matches are unavoidable (for example, if there is an existing driver with a generic name that must be retained for backwards compatibility) allow the drivers that can be incorrectly matched the possibility of rejecting the match. One technique for doing so is to for the driver to inspect the compatible property to ensure that it is appropriate. Another common technique is to have the generic driver *probe* the hardware to see if it behaves as expected (although this technique can cause problems).

5. Existing Devices

Existing devices with explicit names need not change. The recommended search techniques continue to work correctly with such devices.

6. PCI Name Properties Information

Instead of creating their children’s device’s name properties from the vendor and device ID’s, PCI bus device nodes should create them from the class codes contained in the *PCI Bus binding* [3] and *PCI Local Bus Specification* [2], Appendix D, Class Codes.

The **"vendor-id"** and **"device-id"** properties should continue to reflect the same explicit information as before (Refer to [3] about these properties).

For PCI devices without FCode drivers, the PCI bus node should automatically create a **"compatible"** property with a single entry whose value reflects the vendor and device ID’s, as previously specified for the name property.

For PCI devices with FCode drivers, that FCode device driver *shall* create a **"compatible"** property whose first entry uniquely identifies the functions programming model, as previously defined for FCode-generated **"name"** properties. That **"compatible"** property may have additional entries as appropriate. If the device's programming model is not entirely defined by the characteristics of the chip that is directly connected to the PCI bus (for example, if there are additional programmable chips *behind* that directly-connected chip), then that first entry should identify the complete programming model for the device (i.e. the vendor and model number for the entire card), not just the programming model of the directly-connected chip. In any case, the **"compatible"** property value *shall* also contain a string of the form pci*VVVV,DDDD*, identifying the directly-connected chip. *VVVV* and *DDDD* are lower-case ASCII hexadecimal numbers, without leading zeroes, representing the Vendor ID and Device ID of the chip that is directly connected to the PCI bus(This can be the sub-system Vendor ID and Device ID, if present; otherwise, the Vendor ID and Device ID if the sub-system Vendor ID and Device ID are not present) . The **"compatible"** property may have additional entries as appropriate.

Examples:

Preferred: IBM,*v915 pci5333,88b0 vga*
Acceptable: *pci5333,88b0 vga*

Note: As defined by IEEE 1275, the value of the "compatible" property is a list of strings encoded as with "encode-string", so the property value in the *Preferred* example above could be generated with:

```
"IBM,v915"         encode-string
"pci5333,88b0"     encode-string encode+
 "vga"             encode-string encode+
 "compatible"      property
```

The resulting prop-encoded array consists of a series of null-terminated strings concatenated one after another.

Note: It is better to identify the function as a whole, instead of identifying only the *chip* that is directly connected to the PCI bus (e.g.: use IBM,*v915,* instead of *S3,928)*, because, there may be numerous other display adapters that use the same graphics chip, but which are not entirely compatible due to the presence of different support chips like DAC's and programmable clock generators.

7. Compatibility Information

This section discusses compatiblity concerns and implications associated with the strings identfying device compatibility in the Open Firmware **"compatible"** property.

7.1. Historical Perspective

Historically, one of the problems with devices that identify themselves is that they have a choice: they can identify themselves accurately, or they can identify a well-known product they claim to emulate. If they identify themselves accurately, then there is a backwards compatibility problem; existing software will not recognize new hardware, even if the new hardware is a pure superset of the old, supported, hardware. If, on the other hand, they identify themselves as an older product that they emulate, the identification is incorrect or misleads humans and makes it difficult for software to take advantage of added features or work around *imperfections* in the implementation of the emulation.

7.2. Compatibility Intent

Open Firmware's **"compatible"** property attempts to address these concerns by allowing both precise identification of the device and explicit identification of other devices with which this device is compatible. The intent is that software does not have explicit support for this particular device but does have support for one of the devices with which it is *compatible*, the software will be able to function and correctly operate the device. Because the device may have features in addition to those supported by the *compatible* device, the software may not take full advantage of the hardware.

This recommended practice takes this one step further, by noting that the attributes or characteristics that a particular device is MOST compatible with it is itself, and requiring that the device itself be the first entry in the **"compatible"** property. This *idenity compatibility* is an important concept because an implementation that uses a specified name within the **"compatible"** property has to ensure that the implementation must be compatible with any stated specification for a particular device or classs of devices.

7.3. Hardware Programming Interface

Ideally, the hardware programming interface of a new device that is *compatible* with an old device will be exactly the same as, or a pure superset of, the interface supported by the old device. This means not only that the device is *capable* of operating in a compatible fashion, but that the firmware will set it up to operate compatibly.

In particular:

- All defined registers must have their defined semantics.
- All defined commands (if applicable) must have their defined semantics.
- There may be additional registers and/or commands which provide additional functionality, but manipulating them must not be required to operate the device in a compatible fashion.
- Device responses must conform to the defined semantics.
- There may be additional device responses possible, but they must be disabled so that the software receives only compatible responses.

7.4. Open Firmware Properties

As with the hardware interface, Open Firmware Properties must be the same as, or a pure superset of, the defined Open Firmware Properties of the emulated device.

In particular:

- All properties defined by the emulated device must have their defined semantics.
- There may be additional properties, but their use must not be required for compatible operation.
- All defined entries in **"reg"**, **"interrupts"**, and similar properties must have their defined semantics. Such entries must be at the beginning of the property, in the order defined by the emulated device.
- There may be additional entries in **"reg"**, **"interrupts"**, and similar properties, but their use must not be required for compatible operation. Such entries may be added only at the *end* of the property, after all entries defined by the emulated device.
- No property defined by the emulated device may have a value other than those defined by the emulated device.

7.5. Open Firmware Methods

Again, the methods supplied must be the same as, or a pure superset of, the defined properties of the emulated device.

In particular:

- All methods defined by the emulated device must have their defined semantics.
- There may be additional methods, but their use must not be required for compatible operation.
- All defined arguments to methods, in particular **open** arguments, must have the semantics defined by the emulated device.
- There may be additional values for arguments to methods, but their use must not be required for compatible operation.
- Results returned by methods defined by the emulated device must have values as defined by the emulated device.

7.6. What does *compatible* NOT imply?

Compatibilty does not imply the following:

- Anything explicitly specified as *undefined* by the emulated device may differ, and should be expected to differ, in the emulating device.
- Anything not defined by the emulated device may differ in the emulating device.
- Registers, portions of registers, commands, methods, properties, and behaviors marked in the emulated device as *reserved*, *undefined*, *obsolete*, *do not use*, et cetera may have different semantics in the emulating device.
- The contents of the **"reg"**, **"interrupts"**, or similar properties after the last entry defined by the emulated device are undefined and may be used in emulating devices.
- The absolute register addresses, interrupt routing, and so on may differ between the two devices. Software must use the properties supplied to determine these values and must not make assumptions based on the device identification.
- Electrical interfaces may differ between the two devices.
- Physical appearance and packaging may differ between the two devices.
- Labelling, jumpers, switches, installation procedures, and so on may differ between the two devices.

7.7. Real World Perspective

Two devices are seldom perfectly compatible. New functionality often requires a tradeoff with the older and seldomly used functionality. Subtle undocumented behaviors are often different. While the comments above imply absolute compatibility, in reality, placing a device into a *compatible* list is subjective and results in a judgement call by a vendor. If the newer device is compatible with all but a few never-used interfaces of the old, it may be appropriate to call it *compatible* even though it is not strictly a superset. The vendor must decide whether the customer's best interests lie with the ability to, most likely, use the new hardware with existing software, or with an absolute guarantee of compatibility. No simple set of rules can say how to make this decision; it will depend on the exact device and its market. This gray area is why it is absolutely essential that all devices identify themselves precisely as possible so that, in the event that there is an unexpected incompatibility, the software can take appropriate device-specific corrective action. The preferred compatibility policy should be when any functionally doubt exists between two devices, be precise in specifying device compatibility. Compatibility requirements can always specified in a less-precise manner or identification by using the **"compatible"** property.

-- END OF DOCUMENT --