PowerPC processor binding to:

IEEE Std 1275-1994

Standard for Boot

(Initialization, Configuration)

Firmware

Revision: 1.03 DRAFT

Date: July 3, 1995

1. Overview

This document specifies the application of *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements* to computer systems that use the PowerPC Instruction Set Architecture, including instruction-set-specific requirements and practices for debugging, client program interface and data formats. An implementation of Open Firmware for PowerPC *shall* implement the core requirements as defined in [1] and the PowerPC-specific extensions described in this binding.

While this document addresses the official PowerPC architecture [2], the name "PowerPC" only requires compliance to Book I. The descriptions that follow, and the relevant sections describing translation features for this binding, assume that the system's PowerPC processor(s) implement the entire set of Books I-III. Some "PowerPC" processors may implement different Book II-III features; such processors may need a variant of this binding describing the differences to the mapping functions, etc.

This document defines the binding to PowerPC processors that use 32-bit addressing. Since the minimum cell size of Open Firmware is 32 bits, only one cell is necessary to represent addresses of processor bus devices; hence, the value of `"#address-cells"` for `/` *shall* be 1.

> **Note: 64-bit processors can follow the specifications contained herein as long as all addresses relevant to Open Firmware are kept within the first 4 GB; i.e., the upper 32 bits of 64-bit addresses are assumed to contain 0s.**

1.1. Endian-ness Support

The implementation of the PowerPC Open Firmware must support both Big- and Little-Endian system implementation. This section describes added features to the core Open Firmware architecture features to support bi-endian booting.

1.2. Bi-Endian Booting

The Configuration Variable **`little-endian?`** must be implemented. The basic concept of bi-endian support is to keep in the **`little-endian?`** variable a "cached" indication of the desired endian-ness of client programs (for example operating systems or their loaders). This variable indicates the expected endian-mode of a boot target; **`false`** (0) indicates Big-Endian, **`true`** (-1) indicates Little-Endian; the default value of **`little-endian?`** is implementation dependent.

The client program must describe its endian-mode in the header section of its image as described in section 7.1.1.. When Open Firmware is started, Open Firmware must use the value of **`little-endian?`** to establish the endian-mode of a system. After Open Firmware locates and loads a client program, the correct endian-mode must be verified with the description in the header section of the client program image. If the cached value is correct, Open Firmware proceeds, with the system in its current endian-mode.

If, however, Open Firmware determines that the endian-mode of the client program is different from what it had assumed, it must set **`little-endian?`** appropriately and reconfigure the system (hardware and firmware) for the new endian-mode, possibly by resetting the system as with **`reset-all`**.

> **Note: the endian-mode applies to the entire hardware system, including the processor(s). Open Firmware *shall* perform whatever steps are required to enable the system to run in the specified mode.**

Client programs can use `setprop` to alter the value of **`little-endian?`**; users can alter it via the **`setenv`** command from the *user interface* (if present). The alteration of **`little-endian?`** *shall not* cause the system to be reconfigured until the system is re-booted.

> **Note: this mechanism introduces an extra configuration pass. However, this occurs only when switching the endian-mode from that which was last used. For most boots, Open Firmware will be appropriately configured, so that no additional overhead will occur.**

1.3. PowerPC address translation model

This section describes the model that is used for co-existence of Open Firmware and client programs (i.e., operating systems) with respect to address translation.

The following overview of translation is provided so that the issues relevant to Open Firmware for PowerPC can be discussed. Details that are not relevant to Open Firmware issues (e.g., protection) are not described in detail; the PowerPC architecture [2], particularly Book III, should be consulted for the details. For the scope of this section, terms will be used as defined in [2] and the 32-bit processor model is assumed.

1.3.1. Translation requirements

The default access mode of storage for load and stores (i.e., with translation disabled -- referred to as *Real-Mode*) within PowerPC assumes that caches are enabled (in copy-back mode). In order to perform access to I/O device registers, the access mode must be set to Cache-Inhibited, Guarded by establishing a translation with this mode and enabling translation. Thus, even though most of a client program and/or Open Firmware can run with translation disabled, it must be enabled when performing I/O.

1.3.2. HTAB translation

An Effective Address (EA) of the PowerPC processors is 32 bit wide. Each EA is then translated into a 52-bit Virtual Address (VA) by prepending a 24-bit Virtual Segment Id (VSID) to the 28 LSbs of the effective address; the VSID is obtained by indexing into a set of 16 Segment Registers (SRs) using the 4 MSbs of the EA. Finally, the virtual address is translated into a Real Address (RA). This is done by mapping the Virtual Page-Number (VPN) (bits 0-39 of the VA) into a Real Page Number (RPN) and concatenating this RPN with the byte offset (bits 40-51 of the VA). The mapping of VPN to RPN involves a hashing algorithm within the Hashed Page Table (HTAB) to locate a Page Table Entry (PTE) that matches the VPN and using that entry's RPN component. If a valid entry is not found, a Data Storage Interrupt (DSI) or Instruction Storage Interrupt (ISI) is signalled, depending upon the source of the access.

This process is not performed for every translation! Processors will typically have a Translation Look-aside Buffer (TLB) that caches the most recent translations, thus exploiting the natural spatial locality of programs to reduce the overhead of address translation. On most PowerPC processors, the TLB updates are performed in hardware. However, the architecture allows an implementation to use a software-assisted mechanism to perform the TLB updates. Such schemes must not affect the architected state of the processor unless the translation fails; i.e., the HTAB does not contain a valid PTE for the VA and a DSI/ISI is signalled.

> **Note: one unusual feature of this translation mechanism is that valid translations might not be found in the HTAB; the HTAB might be too small to contain all of the currently valid translations. This introduces a level of complexity in the use of address translation by Open Firmware, as discussed below.**

1.3.3. BAT translation

To further reduce the translation overhead for contiguous regions of virtual and real address spaces (e.g., a frame buffer, or all of real memory), the Block Address Translation (BAT) mechanism is also supported by PowerPC. The Block Address Translation involves the use of BAT registers that contain a Block Effective Page Index (BEPI), a Block Length (BL) specifier and a Block Real Page Number (BRPN); the architecture defines 4 BAT registers for data (DBAT registers) and 4 BAT registers for instruction (IBAT registers)[1].

Block Address Translation is done my matching some number of upper bits of the EA (specified by the BL value) against each of the BAT registers. If a match is found, the corresponding BRPN bits replace the matched bits in the EA to produce the RA.

[1] The 601 has a single set of BAT registers that are shared by both instruction and data accesses.

BAT translation takes precedence over HTAB translation; i.e., if a mapping for a storage location is present in both a BAT register and the HTAB, the Block Address Translation will be used.

1.4. Open Firmware's use of memory

Open Firmware must use the memory resources within the space indicated by the **`real-base, real-size, virt-base`** and **`virt-size`** Configuration Variables defined for PowerPC. As described in section 7.1.1., a mechanism is defined to enable Open Firmware to determine if its current configuration is consistent with the requirements of the client.

When a client program is loaded and the header of the client has either the **`real-base`** or **virt-base** values equal to -1, firmware may choose its base locations. If the values in the header of the client conflict with the current values of the configuration variables then firmware shall set the configurations variables to those found in the client's header and restart itself.

If a client does not provide enough real or virtual memory for firmware then the firmware should not boot the client and the condition should be reported to the user

A PowerPC Open Firmware binding *shall* support two different addressing models, depending upon the setting of the **`real-mode?`** Configuration Variable. This variable indicates the Open Firmware addressing mode that a client program expects; **`false`** (0) indicates Virtual-Mode, **`true`** (-1) indicates Real-Mode; the default value of **`real-mode?`** is implementation dependent.

The management of **`real-mode?`** is analogous to **`little-endian?`**. Open Firmware determines its addressing mode using the value of **`real-mode?`**. If the current state of **`real-mode?`** (and hence, the current state of Open Firmware) is incorrect, it *shall* set **`real-mode?`** appropriately and reset itself, possibly by executing **`reset-all`**.

Memory that connot be allocated for general purpose use, for example physical memory on PowerPC systems used for interrupt vectors, shall not appear in the "available" property of the memory node. A Client Program that needs to use such memory for its architected purpose must not claim that area prior to use.

In the following two sections, some of conventions in Real-Mode and Virtual-Mode address translations are described. Remaining sections describe the assumptions that Open Firmware makes about the state and control of the system in regard to Open Firmware's use of system resources for three Open Firmware interfaces (e.g. Device, User and Client interfaces).

1.4.1. Real-Mode

When **`real-mode?`** is **`true`**, Open Firmware must configure its address translation to run in Real-Mode. In Real-Mode, the uses of address translations by Open Firmware and its clients are considered independentt; they do not share any translations. All interfaces between the two must pass addresses with the RA of the data. Any data structure shared by Open Firmware and its clients that refer to *virt* addresses in [1], or this binding, must be RAs (i.e., hardware addresses).

> **Note: in particular, that the address of the Client interface handler, that is passed to the client, has to be a RA.**

The Configuration Variables **`real-base`** and **`real-size`** should indicate the physical memory base and size in which Open Firmware must locate itself. The Configuration Variables **`virt-base`** and **`virt-size`** do not have meaning and should be set to -1.

1.4.2. Virtual-Mode

When **`real-mode?`** is **`false`**, Open Firmware *shall* configure itself to run in *Virtual-Mode*. In Virtual-Mode, Open Firmware and its client will share a single virtual address space. This binding provides interfaces to allow Open Firmware and its client to ensure that this single virtual address model can be maintained.

The Configuration Variables **virt-base** and **virt-size** should indicate the virtual address space base address and size that Open Firmware should use. The Configuration Variables **real-base** and **real-size** should indicate the physical memory base and size in which Open Firmware must locate itself.

1.4.3. Device Interface (Real-Mode)

While Open Firmware is performing system initialization and probing functions, it establishes and maintains its own translations. In particular, it maintains its own HTAB (and/or BAT registers) and handles any DSI/ISI interrupts itself.

> **Note: in Real-Mode, all translations will be *virt=real*; the primary reason for translation is to allow appropriate I/O accesses.**

1.4.4. Device Interface (Virtual-Mode)

Open Firmware will establish its own translation environment, handling DSI/ISI interrupts as in the Real-Mode case. However, this environment will, in general, contain *virt≠real* translations. The virtual address space used by Open Firmware must be compatible with its client.

> **Note: Since these virtual addresses will be used by the Client and/or User Interfaces (e.g., for pointers to its code, device-tree, etc.), their translations must be preserved until the target OS decides that it no longer requires the services of Open Firmware.**

1.4.5. Client Interface (Real-Mode)

In Real-Mode, addresses of client data are real.; the client must ensure that all data areas referred to across the Client Interface are valid real addresses. This may require moving data to meet any requirements for contiguous storage areas (e.g., for **read/write** calls). Translation *shall* be disabled before the client interface call is made.

Open Firmware will typically have to maintain its translations in order to perform I/O. Since the client may be running with translation enabled (except for the Client interface call), Open Firmware *shall* save the state of all relevant translation resources (e.g., SDR1, BATs) and restore them before returning to the client. Likewise, it *may* take over the DSI/ISI interrupts for its own use (e.g., for doing "lazy" allocation of BATs); it *shall* preserve the state of any interrupt vectors for its client.

Since the state of the address translation system is not predictable to any interrupts, the client *shall* ensure that interrupts are disabled before calling the Client Interface handler.

Once client program starts execution, physical memory shall be managed by client program. Since Open Firmware does not know which physical memory the client program is using, Open Firmware shall request physical memory, if needed, to the client program (using **alloc-real-mem**).

1.4.6. Client Interface (Virtual-Mode)

Client interface calls are essentially "subroutine" calls to Open Firmware. Hence, the client interface executes in the environment of its client, including any translations that the OS has established. E.g., addresses passed in to the client interface are assumed to be valid virtual addresses within the scope of the OS. Any DSI/ISI interrupts are either invalid addresses or caused by HTAB "spills". In either case, the OS has the responsibility for the handling of such exceptions.

> **Note: addresses that the Open Firmware internal use will be those that were established by the Device interface (or, by subsequent actions of the Client or User interface). Thus, the client must preserve these Open Firmware translations if it takes over the virtual memory management function.**

In addition to using existing translations, the Client Interface might require the establishment of new translations (e.g., due to **map-in** calls during **open** time), or the removal of old translations (e.g., during **map-out** calls during **close** time). Since this requires altering the Client's translation resources (e.g., HTAB), possibly handling spill conditions, Open Firmware can not know how to perform these updates to the HTAB.

Hence, there *shall* be `callback` services provided by the client for use by Open Firmware for such actions; see section 7.6.2..

1.4.7. User Interface (Real-Mode)

In Real-Mode, Open Firmware regains total control of the system. As with the Client interface in Real-Mode, it *shall* save the state of the translation resources (including interrupt vectors) upon entry and *shall* restore them upon exit.

1.4.8. User Interface (Virtual-Mode)

When the User interface is invoked, Open Firmware is responsible for managing the machine. Therefore, it will take over control of any relevant interrupt vectors for its own handling. In particular, it will take over DSI/ISI handling in order to report errors to the user for bad addresses, protection violations, etc. However, as described above, one source of DSI/ISI may simply be HTAB spills. As with the case of **`map-in`** and **`map-out`** calls, the User interface can not know how to handle such spill conditions, itself, or even if this is, in fact, a spill versus a bad address.

Hence, there *shall* be a `callback` service provided by the client for use by Open Firmware to resolve such translations; see section 7.6.2..

1.5. Real Mode rules

Claim methods shall not allocate more pages than are necessary to satisfy the request.

1.6. Virtual-Mode rules

In order to let clients (i.e., target operating systems) know where Open Firmware lives in the address space, the following rules *shall* be followed by an Open Firmware implementation for PowerPC and by client programs.

Open Firmware:

- Open Firmware *shall* maintain its "translations' "mmu"-node property (see section 4.2.3.)
- *shall not* use BATS during Client or User interface operations.
- Open Firmware's **claim** methods *shall* not allocate more pages than are necessary to satisfy the request.
- When a client executes **set-callback**, Open Firmware *shall* attempt to invoke the "translate" callback. If the translate callback is implemented, Open Firmware *shall* cease use of address translation hardware, instead using the client callbacks for changes to address translation.

  The **exit** service must continue to work after a **set-callback** that takes over address translation. This implies that Open Firmware takes responsibililty for address translation hardware upon **exit** and must maintain internal information about translations that it requests of the client.

Client Programs:

- Client programs that take control of the management of address translation hardware and expect to be able to subsequently invoke Open Firmware client services must provide callbacks to assist Open Firmware in address translation (see section 7.6.2.).

2. References and Terms

2.1. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision *shall* apply.

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

[2] *PowerPC Architecture*, published by IBM (Customer Reorder Number 52G7487).

[3] *System V Application Binary Interface*, published by UNIX System Laboratories. This document describes the generic architecture of the ELF (Executable and Linking Format) object file format.

[4] *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*, found in the March, 1994 issue of *Microsoft Systems Journal*.

[5] *System V Application Binary Interface, PowerPC Processor Supplement*, Sunsoft. This document defines the PowerPC specific ABI for System V and also gives details on the PowerPC ELF format.

2.2. Terms

This standard uses technical terms as they are defined in the documents cited in "References", plus the following terms:

core, core specification: refers to *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements*

ELF: Executable and Linking Format. A binary object file format defined by [3][5] that is used to represent *client programs* in Open Firmware for PowerPC.

linkage area: An area within the stack that is reserved for saving certain registers across procedure calls in PowerPC run-time models. This area is reserved by the caller and is allocated above the current stack pointer (**`%r1`**).

Open Firmware: The firmware architecture defined by the core specification or, when used as an adjective, a software component compliant with the core specification.

procedure descriptor: a data structure used by some PowerPC run-time models to represent a C "pointer to procedure". The first word of this structure contains the actual address of the procedure.

Real-Mode: The mode in which Open Firmware and its client are running with translation disabled; all addresses passed between the client and Open Firmware are real (i.e., hardware) addresses.

Table of Contents (TOC): A data structure used by some PowerPC run-time models that is used for access to global variables and for inter-module linkage. When a TOC is used, **%r2** contains its base address.

Virtual-Mode: The mode in which Open Firmware and its client share a single virtual address space, and address translation is enabled; all addresses passed between the client and Open Firmware are virtual (translated) addresses.

3. Data Formats and Representations

The cell size *shall* be 32 bits. Number ranges for *n*, *u*, and other cell-sized items are consistent with 32-bit, two's-complement number representation.

The required alignment for items accessed with *a-addr* addresses *shall* be four-byte aligned (i.e., a multiple of 4).

Each operation involving a *qaddr* address *shall* be performed with a single 32-bit access to the addressed location; similarly, each *waddr* access *shall* be performed with a single 16-bit access. This implies four-byte alignment for *qaddrs* and two-byte alignment for *waddrs*.

4. Properties

This section describes the standard properties of PowerPC Open Firmware implementation.

4.1. Root node properties

The following properties of the root node ("/") shall be created by an Open Firmware implementation.

> **Note: Note that the root node typically corresponds to the common processor bus in a PowerPC system.**

"#address-cells"

Standard property, encoded as with **encode-int**, that specifies the number of cells required to represent physical addresses on the processor bus; the value of "#address-cells" for the processor bus *shall* be 1.

"clock-frequency"

Standard property, encoded as with **encode-int**, that represents the primary system bus speed (in hertz).

"system-id"

Standard property, encoded as with encode-string, that contains the identification of the computer system. This string shall be unique across all systems and all manufacturers. If the system-id begins with "0", it shall be of the form "0nnnnnnmmmmmm" where nnnnnn is a sequence of 6 uppercase hexadecimal digits representing a 24-bit Organizationally Unique Identifier (OUI) assigned by the IEEE Registration Authority Committee, and mmmmmm is a sequence of 6 uppercase hexadecimal digits representing a 24-bit binary number assigned by the manufacturer to assure uniqueness.

Note: For systems with built-in ethernet or other IEEE 802-style interfaces, the 6-byte MAC address assigned to that interface meets the requirements and should be used as the system-id.

4.2. CPU node properties

For each CPU in the system, a cpu-node *shall* be defined as a child of "/" (the root). The following properties apply to each of these nodes.

"name"

Open Firmware standard propetry. The value of the is property shall be of the form: "PowerPC,XXX", where XXX is 601, 603, 604 etc.

"device_type"

Open Firmware standard property. The value of this property for CPU nodes *shall* be "cpu".

"cpu-version"

Standard property, encoded as with **encode-int**, that represents the processor type. This value is obtained by reading the Processor Version Register of the CPU.

"clock-frequency"

Standard property, encoded as with **encode-int**, that represents the internal processor speed (in hertz) of this node.

"timebase-frequency"

Standard property, encoded as with **encode-int**, that represents the rate (in hertz) at which the PowerPC TimeBase and Decrementer registers increment.

Note: The 601 PowerPC does not have a timebase frequency, therefore on a 601 PowerPC the value reported in this property shall be 1 billiion which represents the logical rate of the real time clock.

4.2.1. TLB properties

Since the PowerPC architecture defines the MMU as being part of the processor, the properties defined by Section 3.6.5 of [1] and the following MMU-related properties shall be presented under "cpu" nodes.

"tlb-size"

Standard property, encoded as with **encode-int**, that represents the total number of TLB entries.

"tlb-sets"

Standard property, encoded as with encode-int, that represents the number of associativity sets of the TLB. A value of 1 indicates that the TLB is fully-associative.

4.2.2. Internal (L1) cache properties

The PowerPC architecture defines a Harvard-style cache architecture; however, unified caches are an implementation option. All of the PowerPC cache instructions act upon a cache "block" (also referred to as a cache "line"). The internal (also referred to as "L1") caches of PowerPC processors are represented in the Open Firmware device tree by the following properties contained under `"cpu"` nodes.

`"cache-unified"`

This property, if present, indicates that the internal cache has a unified organization. Absence of this property indicates that the internal caches are implemented as separate instruction and data caches.

`"i-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the total size (in bytes) of the internal instruction cache.

`"i-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents number of associativity sets of the internal instruction cache. A value of 1 signifies that the instruction cache is fully associative.

`"i-cache-block-size"`

Standard property, encoded as with **`encode-int`**, that represents the internal instruction cache's block size, in bytes.

`"d-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the total size (in bytes) of the internal data cache.

`"d-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents number of associativity sets of the internal data cache. A value of 1 signifies that the data cache is fully associative.

`"d-cache-block-size"`

Standard property, encoded as with **`encode-int`**, that represents the internal (L1) data cache's block size, in bytes.

`"l2-cache"`

Standard property, encoded as with **`encode-int`**, that represents another level of cache in the memory hierarchy.

Absence of this property indicates that no further levels of cache are present. If present, its value is the *phandle* of the device node that represents the L2 cache.

4.2.3. Memory Management Unit properties

To aid a client in "taking over" the translation mechanism and still interact with Open Firmware (via the *client interface*), the client needs to know what translations have been established by Open Firmware. The following standard property shall exist within the package to which the "mmu" property of the /chosen package refers.

`"translations"`

This property, consisting of sets of translations, defines the currently active translations that have been established by Open Firmware (e.g., using map). Each set has the following format:

```
( virt size phys mode )
```

Each value is encoded as with **`encode-int`**.

4.3. External (L2,L3...) cache node properties

Some systems might include external (L2) cache(s). As with the internal caches, they can be implemented as either Harvard-style or unified. Unlike the L1 properties, that are contained within the `"cpu"` nodes, L2 caches are contained within other device tree nodes.

The following properties define the characteristics of such L2 caches. These properties *shall* be contained as a child node of one of the CPU nodes; this is to allow path-name access to the node. All `"cpu"` nodes that share the same L2 cache (including the cpu node under which the L2 cache node is contained) *shall* contain an `"l2-cache"` property whose value is the *`phandle`* of that L2 cache node.

> **Note: it is possible to extend this scheme to arbitrary levels of secondary, tertiary, etc. caches. The `"l2-cache"` property shall be used in one level of the cache hierarchy to represent the next level. The device node for a subsequent level shall appear as a child of one of the caches in the hierarchy to allow path-name traversal.**

`"device-type"`

Open Firmware Standard property; the device-type of L2-cache nodes *shall* be `"cache"`.

`"cache-unified"`

This property, if present, indicates that the L2 cache has a unified organization. Absence of this property indicates that the L2 caches are implemented as separate instruction and data caches.

`"i-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the total size (in bytes) of the L2 instruction cache.

`"i-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents number of associativity sets of the L2 instruction cache. A value of 1 signifies that the instruction cache is fully associative.

`"d-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the total size (in bytes) of the L2 data cache.

`"d-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents number of associativity sets of the L2 instruction cache. A value of 1 signifies that the instruction cache is fully associative.

`"l2-cache"`

Standard property, encoded as with **`encode-int`**, that represents another level of cache in the memory hierarchy.

Absence of this property indicates that no further levels of cache are present. If present, its value is the *phandle* of the device node that represents the cache at the next level.

4.4. "/openprom" node property

5. Methods

This section describes the additional standard methods required of a PowerPC Open Firmware implementation.

5.1. MMU related methods

The mmu methods defined by section 3.6.5. of [1] *shall* be implemented by CPU nodes. The value of the **`mode`** parameter for the relevant methods (e.g., **`map`**) *shall* be the value that is contained within PTEs that control Write-through, Cache-Inhibit, Memory-coherent, Guarded and the 2 protection bits; thus, its format is: **`WIMGxPP`**, where x is a reserved bit that *shall* be 0. In order for I/O accesses to be properly performed in a PowerPC system, address ranges that are mapped by **`map-in`** *shall* be marked as Cache-Inhibited, Guarded.

6. Client Interface Requirements

A PowerPC Open Firmware implementation *shall* implement a client interface (as defined in chapter 6 of [1]) according to the specifications contained within this section.

6.1. Calling Conventions

Register(s)	Value	Notes
%msr	**preserved by client interface**	
%cr	**preserved by client interface**	**1**
%r1–%r2	**preserved by client interface**	
%r3	**argument array address on client interface entry**	**2**
	result value (true or false) on client interface return	**2**
%r13–%r31	**preserved by client interface**	
%sprg0–%sprg3	**preserved by client interface**	
%fpscr	**preserved by client interface**	
%f0–%f31	**preserved by client interface**	
%lr, **%ctr**, **%xer**	**undefined**	
Other SPRs	**preserved by client interface**	3

Table 1. Register usage conventions

Notes

1. Only the non-volatile fields (cr2–cr4) need to be preserved.

2. As defined by section 6.3.1. of [1].

3. Other special purpose registers

To invoke a client interface service, a *client program* constructs a client interface *argument array* as specified in the core Open Firmware document, places its address in **%r3** and transfers to the *client interface handler*, with the return address in **%lr**. (A typical way of accomplishing this is to copy the *client interface handler*'s address into **%ctr** and executing a **bctrl**.)

The *client interface handler shall* perform the service specified by the contents of the argument array that begins at the address in **%r3**, place the return value (indicating success or failure of the attempt to invoke the client interface service) back into **%r3**, and return to the *client program*. This is typically done by a Branch to Link Register (**blr**).

The *client interface handler shall* preserve the contents of the Stack Pointer (**%r1**), TOC Pointer (**%r2**), Condition Register (**%cr**) all non-volatile registers (**%r13–%r31**) and all special purpose registers except **%lr**, **%ctr** and **%xer**.

The preservation of **%r2** allows TOC-based client programs to function correctly. Open Firmware *shall not* depend upon whether its client is TOC-based or not. If the client interface handler, itself, is TOC-based, it must provide for the appropriate initialization of its **%r2**.

7. Client Program Requirements

7.1. Client Program Format

The data format of a client program compliant with this specification *shall* be either ELF (Executable and Linkage Format) as defined by [3][5], and extended by section 7.1.1., or PE (Portable Executable) as defined by [4]. The standard ELF format contains explicit indication as to the program's execution modes (e.g., 32- or 64-bit, Big- or Little-Endian); this version of the specification only deals with the 32-bit version (i.e., `ELFCLASS32`).

> **Note: other client program formats *may* be supported, in an implementation specific manner, by an Open Firmware implementation.**

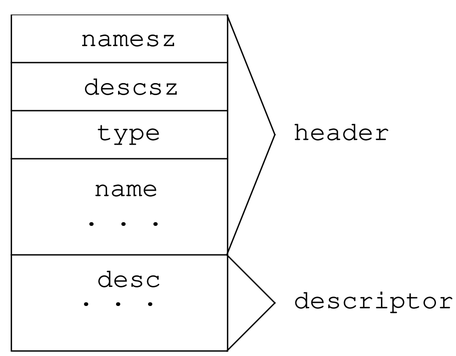
A standard *client program* shall be statically linked, requiring no relocation of the image. The program's entry point (`e_entry`) *shall* contain the address of the first PowerPC instruction of the client program. It is the responsibility of the client program to establish the appropriate value of the TOC (**`%r2`**), if necessary.

> **Note: the entry point is the address of the first instruction of the client program, not that of a procedure descriptor.**

7.1.1. ELF Note Section

Part of the process of loading a client program involves verifying its environmental requirements (e.g., endian-ness and address translation mode) against the current firmware configuration. The client's endian-ness can be directly determined by examining the ELF EI-DATA value; ELFDATA2LSB (1) implies Little-Endian while ELFDATA2MSB (2) implies Big-Endian. However, the other client requirements (e.g., address translation mode) are defined by means of an ELF Note Section (SHT_NOTE). The following describes the format of the Note Section for a client program file.

As defined by [3], an ELF file can be "annotated" by means of Note Sections within the executable file. A Note Section contains a "header" followed by a (possibly null) "descriptor", as follows:



> **Note: the endian format of the values corresponds to the endian-ness specified by the `EI-DATA` field of the file.**

The format of a Note Section header can be described by an Open Firmware `struct` as:

```
struct   \ Note Section header for PR*P Open Firmware
/L  field   ns.namesz    \ length of ns.name, including NULL
/L  field   ns.descrsz
/L  field   ns.type
0   field   ns.name      \ NULL-terminated, /L padded
```

The `ns.name` field of the PowerPC Open Firmware Note Section *shall* be `"PowerPC"`; the `ns.type` field *shall* be `0x1275`.

Following the Note Section header is a descriptor (`desc`); the length (in bytes) of the descriptor is specified by a word in the Note Section's header (`descsz`). The interpretation of the descriptor depends upon the kind of Note Section in which it is contained. For Open Firmware, the format of the Note Section's descriptor can be described by an Open Firmware `struct`, as follows:

```
struct   \ Note Section descriptor for PR*P Open Firmware
/L   field    ns.real-mode
/L   field    ns.real-base
/L   field    ns.real-size
/L   field    ns.virt-base
/L   field    ns.virt-size
```

7.2. Load Address

The load address is specified by the value of the **`load-base`** Configuration Variable. The value of **load-base** shall be a real address in real mode or a virtual address in virtual mode. At least 4 MB of memory *shall* be available for loading at that address. It is strongly recommended that as much memory as is practical for the particular system be available there, thus allow the loading of large client programs. Note that this address represents the area into which the client program file will be read by **load**; it does not correspond to the addresses at which the program will be executed.

7.3. Initial Program State

This section defines the "initial program state", the execution environment that exists when the first machine instruction of a *client program* of the format specified above begins execution. Many aspects of the "initial program state" are established by **`init-program`**, which sets the *saved program state* so that subsequent execution of **go** will begin execution of the *client program* with the specified environment.

7.3.1. Initial Register Values

Upon entry to the client program, the following registers shall contain the following values:

Register(s)	Value	Notes
%msr	EE=0, interrupts disabled	1
	PR=0, supervisor state	
	FP=1, floating point enabled	
	ME=1, machine checks enabled	
	FE0,FE1=0, floating point exceptions disabled	
	IP, see section 7.5.	
	IR,DR, see section 1.4.1.	
	SF=0, 32-bit mode	
	ILE,LE, see section 1.2.	2
%r1	see section 7.3.2.	
%r2	0	3
%r3	reserved	4
%r4	reserved	4
%r5	see section 7.3.3.	
%r6,%r7	see section 7.3.4.	
Others	0	

Table 2. Initial Register Values

Notes:

1. Open Firmware will typically require the use of external interrupts for its *user interface*. However, when a *client program* is invoked, external interrupts *shall* be disabled. If a *client program* causes the invocation of the user interface, external interrupts *may* be re-enabled.

2. The 601 processor uses a different mechanism for controlling the endian-mode of the processor. On the 601, the LE bit is contained in the HID0 register; this bit controls the endian-mode of both program and privileged states.

3. Open Firmware does not make any assumptions about whether a client program is TOC-based or not. It is the responsibility of the client program to set **`%r2`** to its TOC, if necessary.

4. As defined in the Section 5.6 of the PR*P specification.

7.3.2. Initial Stack

Client programs *shall* be invoked with a valid stack pointer (**`%r1`**) with at least 32K bytes of memory available for stack growth. The stack pointer *shall* be 16-byte aligned, reserving sufficient room for a linkage area (32 bytes above the address in **`%r1`**). If the system is executing in Real-Mode, the value in **`%r1`** is a real (hardware) address; if in Virtual-Mode, the address in **`%r1`** is a mapped virtual address.

7.3.3. Client Interface Handler Address

When client programs are invoked, **`%r5`** *shall* contain the address of the entry point of the *client interface handler*. If the system is executing in Real-Mode, the value in **`%r5`** is a real (hardware) address; if in Virtual-Mode, the address in **`%r5`** is a mapped virtual address.

Note: this address points to the first instruction of the *client interface handler*, not to a procedure descriptor.

7.3.4. Client Program Arguments

The calling program *may* pass to the client an array of bytes of arbitrary content; if this array is present, its address and length *shall* be passed in registers **`%r6`** and **`%r7`**, respectively. For programs booted directly by Open Firmware, the length of this array is zero. Secondary boot programs may use this argument array to pass information to the programs that they boot.

Note: The Open Firmware standard makes no provision for specifying such an array or its contents. Therefore, in the absence of implementation-dependent extensions, a client program executed directly from an Open Firmware implementation will not be passed such an array. However, intermediate boot programs that simulate or propagate the Open Firmware client interface to the programs that they load can provide such an array for their clients.

Note: `boot` command line arguments, typically consisting of the name of a file to be loaded by a secondary boot program followed by flags selecting various secondary boot and operating system options, are provided to client programs via the "`bootargs`" and "`bootpath`" properties of the "`/chosen`" node.

7.4. Caching

The caches of the processor *shall* be enabled when the client program is called. Memory areas allocated on behalf of the client program *shall* be marked as cacheable. Accesses to "I/O" devices (especially, to devices across "bridges") *shall* be made with the register access words (e.g., **`rl@`**).

7.5. Interrupts

Open Firmware requires that interrupts be "vectored" to its handlers when it is in control of the processor; this will occur when the User Interface is running. Client Interface calls are considered to execute in the environment of the client, and hence, do not assume ownership of interrupts.

In order for Open Firmware to process interrupts in an efficient manner, an area of memory for the exclusive use by Open Firmware *shall* be reserved by the client program at (real) memory locations 0x1E0…0x1FF.

Open Firmware *shall* save and restore the first location of each interrupt that it wants to "take over". I.e., whenever Open Firmware needs the use of an interrupt, it *shall* save the current contents of the corresponding entry point and replace that location with a branch to its entry point. When Open Firmware returns control, it *shall* restore the RAM location to its original contents.

7.6. Client callbacks

This section defines the callback mechanism that allows Open Firmware to access services exported to it by the client program. As described in section 6.3.2.6 (and the glossary entries for `callback` and `$callback`) in [1], the callback mechanism follows the same rules as those of Client interface calls. I.e., an *argument array* is constructed by Open Firmware and the address of that array is passed (via **`%r3`**) to the client's callback routine; the address of the callback routine is supplied to Open Firmware by means of the `set-callback` client call.

If the system is running in Real-Mode, the address of the client callback routine *shall* be a real (hardware) address; if it is running in Virtual-Mode, the client callback routine address *shall* be a mapped virtual address.

7.6.1. Real-Mode physical memory management assist callback

Once the control of physical memory is transferred to the client program, Open Firmware which is running in real-mode must use the callback service provided by the client program to allocate physical memory. Client programs which expect Open Firmware to operate in read-mode must implement the following physical memory management routine for Open Firmware:

alloc_real_mem

```
IN:   [address] min_addr, [address] max_addr, size, mode
OUT:  error,[address] real_addr
```

This routine allocates a contiguous physical memory of *size* bytes within the address range between *min_addr* and *max_addr*. The *mode* parameter contains the WIMGxPP bits as defined in section 5. A non-zero error code *shall* be returned if the mapping can not be performed. If erro code is zero (i.e. allocation is succededed) the routine returns the base address of the physical memory allocated for Open Firmware.

7.6.2. Virtual address translation assist callbacks

As mentioned in section 1.6., when Open Firmware is running Virtual-Mode, client programs that take over control of the system's memory management must provide a set of callbacks that implement MMU functions. This section defines the input arguments and return values for these callbacks. The notation follows the style used in chapter 6 of the Open Firmware specification [1].

`map`

```
IN:   [address] phys, [address] virt, size, mode
OUT:  error
```

This routine creates system-specific translation information; this will typically include the addition of PTEs to the HTAB. If the mapping is successfully performed, a value of zero *shall* be placed in the *`ret1`* cell of the argument array; a non-zero error code *shall* be returned (in *`ret1`*) if the mapping can not be performed.

`unmap`

```
IN:   [address] virt, size
OUT:  none
```

The system removes any data structures (e.g., PTEs) for the virtual address range.

`translate`

```
IN: [address] virt
OUT: error, [address] real, mode
```

The system attempts to compute the real address (`real`) to which the virtual address (`virt`) is mapped. If the translation is successful, a PTE *shall* be placed into the HTAB for this translation, the resulting real address *shall* be returned in `real.hi,real.lo` and *`ret1`* shall be set to **`false`** (0). If the translation is not successful, *`ret1`* *shall* be set to a non-zero error code.

This call *shall* be made when Open Firmware handles a DSI/ISI within the User interface. A successful result of the translate call indicates that Open Firmware can complete the interrupted access; a failure indicates that an access was made to an invalid address.

8. User Interface Requirements

An implementation of Open Firmware for PowerPC *shall* conform to the core requirements as specified in [1] and the following PowerPC-specific extensions.

8.1. Machine Register Access

The following *user interface* commands represent PowerPC registers within the *saved program state*. Executing the command returns the saved value of the corresponding register. The saved value may be set by preceding the command with **to**; the actual registers are restored to the saved values when **go** is executed.

8.1.1. Branch Unit Registers

`%cr`

Access saved copy of Condition Register.

`%ctr`

Access saved copy of Count Register.

`%lr`

Access saved copy of Link Register.

`%msr`

Access saved copy of the low order 16 bits of SRR1 register.

`%srr0` and **`%srr1`**

Access saved copy of Save/Restore Registers.

`%pc`

An alias of "**%srr0**"

8.1.2. Fixed-Point Registers

`%r0` through **`%r31`**

Access saved copies of fixed-point registers.

`%xer`

Access saved copy of XER register.

`%sprg0` through **`%sprg3`**

Access saved copies of SPRG registers.

8.1.3. Floating-Point Registers

Unlike the other registers, the floating point unit registers are not normally saved, since they are not used by Open Firmware. The following access words, therefore, access the registers directly.

`%f0` through **`%f31`**

Access floating point registers.

`%fpscr`

Access Floating Point Status and Control Register.

The following command displays the PowerPC CPU's *saved program state*.

`.registers`

9. Configuration Variables

In addition to the standard Configuration Variables defined by the core Open Firmware document [1], the following Configuration Variables *shall* be implemented for PowerPC:

`"little-endian?"`

This boolean variable controls the endian-mode of Open Firmware. If **`true`** (-1), the endian-mode is Big-Endian; if **`false`** (0), the endian-mode is Little-Endian. The default value of is implementation dependent.

`"real-mode?"`

This boolean variable controls the address translation mode of Open Firmware. If **`true`** (-1), the addressing mode is Real-Mode; if **`false`** (0), the addressing mode is Virtual-Mode. The default value is implementation dependent.

`"real-base"`

This integer variable defines the starting physical address to be used by Open Firmware.

`"real-size"`

This integer variable defines the size of the physical address space which can be used by Open Firmware.

`"virt-base"`

This integer variable defines the starting virtual memory address which can be used by Open Firmware.

`"virt-size"`

This integer variable defines the size of the virtual address space which can be used by Open Firmware.

`"load-base"`

This integer variable defines the default load address for *client program*s when using the **`load`** method. The default value is implementation dependent.