ARM Processor Binding to:

IEEE 1275-1994

Standard for Boot

(Initialization, Configuration)

Firmware

Revision: 0.3 DRAFT

Date: November 5, 1997

PRELIMINARY

Purpose of this ARM Processor Binding

This document specifies the application of Open Firmware to an ARM Processor, including requirements and practices to support unique firmware specific to an ARM Processor. The core requirements and practices specified by Open Firmware must be augmented by processor-specific requirements to form a complete specification for the firmware implementation for a ARM Processor. This document establishes such additional requirements pertaining to the processor and the support required by Open Firmware.

Task Group Members

The ARM Processor Binding team members were the following:

Mitch Bradley, FirmWorks

Greg Hill (editor), FirmWorks

Trademarks

The following terms, denoted by a registration symbol (®) or trademark symbol(™) on the first occurrence in this publication, are registered trademarks or trademarks of the companies as shown in the list below:

Trademark	**Company**

Revision History

Revision	**Date**	**Changes**
0.1	July 1, 1997	Initial, unapproved release.
0.2	August 19, 1997	Added the virtual address region 0x0000.0000 - 0x0000.1000 to the virtual space consumed by Open Firmware at the hand-off to a client program.
0.3	November 5, 1997	Clarified Open Firmware's unmapping and releasing of the unused portion of the client program load space after program preparation for execution has been completed.

Table of Contents

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 Advance RISC Machine (ARM) Instruction Set Architecture, including instruction-set-specific requirements and practices for debugging, client program interface and data formats. An implementation of Open Firmware for ARM *shall* implement the core requirements as defined in [1] and the ARM-specific extensions described in this binding.

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 When the following standards are superseded by an approved revision, the revision *shall* apply.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] *The ARM Architecture, 1/e,* Dave Jagger, Prentice Hall, March, 1997, (ISBN 0-13-736299-4).

[3] a.out file format as defined in the file `include/sys/exec_aout.h` of the NetBSD distribution. http://www.netbsd.org.

2.2. Terms

This standard uses technical terms as they are defined in the documents cited in “References” on page 1, plus the following terms:

core specification: Synonym for *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements* (i.e. the standard that specifies the system-independent and bus-independent requirements for Open Firmware).

Open Firmware: The firmware architecture defined by IEEE Std 1275-1994 and its applicable supplements or, when used as an adjective, a software component compliant with such an architecture.

virtual address: The address that a program uses to access a memory location or memory-mapped device register. Depending on the presence or absence of memory mapping hardware in the system, and whether or not that mapping hardware is enabled, a virtual address may or may not be the same as the physical address that appears on an external bus. Unless otherwise noted, all addresses mentioned in this document are virtual 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. Memory Management

4.1. Open Firmware's Use of Memory

Open Firmware *shall* use the memory resources within the virtual space beginning at 0xF700.000 whose size is 0x0100.0000.

4.1.1. Virtual-Mode

Open Firmware *shall* operate with the memory management unit enabled so that Open Firmware and its client 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.

4.1.2. Client Interface

Client interface services are invoked essentially as "subroutine" calls to Open Firmware. Hence, the client interface executes in the environment of its client, including any translations that the OS has established (i.e. addresses passed in to the client interface are assumed to be valid virtual addresses within the scope of the OS).

> **Note: If a client program takes control of memory management and address translation and wishes to continue using client interface services, the client program must establish a callback handler as described in "Client Callbacks" on page 12.**

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. page tables), Open Firmware cannot know how to perform these updates.

4.1.2.1. Open Firmware Rules

The following rules let clients (i.e. target operating systems) know where Open Firmware lives in the address space.

- Open Firmware *shall* maintain the value of the `"translations"` property of `/mmu` (see Section 5.4.1)
- 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.

4.1.2.2. Client Program Rules

- 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 6.4.1).
- A client program *shall* not directly manipulate any address translation hardware before it either:
  a) Ceases to invoke OF client services, or
  b) Issues a **`set-callback`** to install the `translate` callback.

  > **Note: The intended sequence is that a client program will first issue a `set-callback` and then take control of address translation hardware.**

5. Device Tree

This section describes the processor-related nodes of the device tree of an ARM Open Firmware implementation.

5.1. "/cpus" Node

There *shall* be a node named "cpus" which is a direct child of the root node.

The purpose of this node is to contain other "cpu" nodes and to define an address space by which individual "cpu" nodes can be distinguished from one another. This anticipates the possibility of multi-processor ARM systems in the future.

5.1.1. Physical Address Formats and Representations

5.1.1.1. Numerical Representation

The numerical representation of a child's address *shall* be a single binary integer in the range 0 … N-1, where N is the maximum number of CPUs supported by the system architecture.

5.1.1.2. Text Representation

The text representation of a child's address *shall* be the ASCII hexadecimal number corresponding to the numerical representation of the address.

Conversion of the hexadecimal number 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 zeros.

5.1.1.3. Unit Address Representation

A processor's "unit-address" (i.e. the first component of its "reg" value) is the interprocessor interrupt destination identifier used by the platform. For a uni-processor platform, the "unit-address" *shall* be zero.

5.1.2. "/cpus" Node Properties

The following properties *shall* be created within the "cpus" node.

"name"

Standard *prop-name* to define the name of the node.

prop-encoded-array: Text string, encoded as with **encode-string**.

The value of this property *shall* be "cpus".

"#address-cells"

Standard *prop-name* to define the number of cells required to represent the physical addresses for the "cpu" nodes (i.e. the children of the "cpus" node).

prop-encoded-array: Integer constant 1, encoded as with **encode-int**.

The value of "#address-cells" for the "cpus" node *shall* be 1.

"#size-cells"

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

prop-encoded-array: Integer constant 0, encoded as with **encode-int.**

The value of "#size-cells" for the "cpus" node is 0 because the processors that are represented by the "cpu" nodes do not consume any physical address space.

5.2. "/cpus/cpu" Node

Each CPU in the system *shall* have a node describing it as follows.

5.2.1. "/cpus/cpu" node properties

"name"

Standard *prop-name* to define the name of the node.

prop-encoded-array: Text string, encoded as with **encode-string**.

The value of this property *shall* be "cpu".

"device_type"

Standard *prop-name* to specify the implemented interface.

prop-encoded-array: Text string, encoded as with **encode-string**.

The value of this property *shall* be "cpu".

"reg"

Standard *prop-name* to specify the node's addressable resources.

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

The value of this property *shall* be a one-cell integer representing the interrupt dispatch number that is used to direct inter-processor interrupts to this CPU.

For a uniprocessor system, the value *shall* be 0.

"model"

Standard *prop-name* to define a cpu node's model.

prop-encoded-array: Text string, encoded as with **encode-string**.

The value of this property *shall* be a string consisting of the CPU type and revision (e.g. "Strongarm SA-110-3").

"clock-frequency"

Standard property specifying the internal processor speed of this node.

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

The value of this property *shall* be an integer specifying the CPU internal clock frequency in Hertz (e.g. 235,500,000).

"bus-frequency"

Standard property specifying the speed of this processor's bus.

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

The value of this property *shall* be an integer specifying the speed in Hertz of this processor's bus (e.g. 66,666,666).

5.2.1.1. TLB Properties

Since the ARM 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 in decimal (e.g. 32).

"tlb-sets"

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

Note: The number of sets is related to, but not the same as, the number of "ways of associativity". Specifically:

tlb-sets * number-of-ways = tlb-size

For a fully-associative TLB, tlb-sets = 1.

For a direct-mapped TLB, tlb-sets = tlb-size.

5.2.1.2. Internal (L1) Cache Properties

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

`"write-buffer-size"`

Standard property, encoded as with **`encode-int`**, that represents the maximum number of bytes in the write buffer (e.g. 16).

If there is no write buffer, the value *shall* be 0.

`"d-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the size of the internal data cache in bytes (e.g. 16384).

`"d-cache-block-size"`

Standard property, encoded as with **`encode-int`**, that represents the size of a data cache block size, in bytes (e.g. 32).

`"d-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents the number of data cache sets.

Note: The number of sets is related to, but not the same as, the number of "ways of associativity". Specifically:

d-cache-sets * number-of-ways * d-cache-block-size = d-cache-size

For a fully-associative data cache, d-cache-sets = 1.

For a direct-mapped data cache, d-cache-sets = d-cache-size / d-cache-block-size

`"i-cache-size"`

Standard property, encoded as with **`encode-int`**, that represents the size of the instruction cache in bytes (e.g. 16384).

`"i-cache-block-size"`

Standard property, encoded as with **`encode-int`**, that represents the size of an instruction cache block in bytes (e.g. 32).

`"i-cache-sets"`

Standard property, encoded as with **`encode-int`**, that represents number of instruction cache sets.

Note: The number of sets is related to, but not the same as, the number of "ways of associativity". Specifically:

i-cache-sets * number-of-ways * i-cache-block-size = i-cache-size

For a fully-associative instruction cache, i-cache-sets = 1.

For a direct-mapped data cache, i-cache-sets = i-cache-size / i-cache-block-size

TBD: Do we need to report which kinds of flush and clean instructions are supported?

5.2.2. `"/cpus/cpu"` Node Methods

`open`

Standard method that prepares this device for subsequent use.

`close`

Standard method that restores a previously-opened device to its “not in use” state.

5.3. `"/chosen"` Node

In addition to the standard properties defined for this node by [1], this binding defines the following property.

`"cpu"`

Standard property, encoded as with **`encode-int`**, that represents the ihandle of an instance of the `"cpu"` node corresponding to the CPU on which the firmware is executing.

5.4. Memory Management Unit

5.4.1. Memory Management Unit Properties

To aid a client in “taking over” the translation mechanism while still enabling interaction with Open Firmware (via the client interface), the client needs to know the granularity of the virtual address space and what translations have been established by Open Firmware. In addition to the standard properties listed in Section 3.6.5 of [1], the following standard properties *shall* exist within the package to which the `"mmu"` property of the `/chosen` package refers.

`"page-size"`

Standard property, encoded as with **`encode-int`**, that specifies the number of bytes in the smallest mappable region of virtual address space.

The value of this property *shall* be 4096 (decimal).

`"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`**.

5.4.2. Memory Management Unit Methods

There are no additional methods required beyond those specified in Section 3.6.5 of [1].

5.5. Ancillary (L2, L3 ….) Cache Node Properties

Some systems might include secondary (L2) or tertiary (L3), etc. cache(s). They can be implemented as either Harvard-style or unified. Unlike the L1 properties, that are contained within the `"cpu"` nodes, the properties of ancillary caches are contained within other device tree nodes.

The following properties define the characteristics of such ancillary 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 ancillary cache (including the cpu node under which the ancillary cache node is contained) *shall* contain an `"l2-cache"` property whose value is the *phandle* of that ancillary cache node.

> **Note: 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 ancillary cache nodes *shall* be `"cache"`.

"cache-unified"

This property, if present, indicates that the cache at this node has a unified organization. Absence of this property indicates that the caches at this node 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 instruction cache at this node.

"i-cache-sets"

Standard property, encoded as with **encode-int**, that represents number of associativity sets of the instruction cache at this node. 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 data cache at this node.

"d-cache-sets"

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

"l2-cache"

Standard property, encoded as with **encode-int**, that represents the next 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.

"i-cache-line-size"

Standard property, encoded as with encode-int, that represents the internal instruction cache's line size, in bytes, if different than its block size.

"d-cache-line-size"

Standard property, encoded as with encode-int, that represents the internal data cache's line size, in bytes, if different than its block size.

Note: If this is a unified cache, the corresponding i- and d- sizes must be equal.

6. Client Interface Requirements

An ARM 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. Client Program Loading

6.1.1. Load Address

The default load address is 0xF0000000, the value of **load-base**. Client programs are assumed to be designed to be loaded at 0xF0000000.

Prior to the first execution of **load**, the firmware *shall* allocate and map at least 6 MB of physical memory at this address, unless the hardware configuration of the system makes this impossible. In that case, the firmware *shall* map as much memory as practical.

Note: As described in Section 6.1.2., for most load formats, once a loaded program has been prepared for execution, any memory in the load area that is not actually consumed by the loaded image is then unmapped and released to the available list.

6.1.2. Client Program Header

An Open Firmware implementation *shall* recognize the sequence of eight quadlets described below as a valid client program header (as used by the **load** User Interface command in the core specification) if the a_midmag quadlet contains the specified values. The offsets given below are from the beginning of the loaded image.

Offset	Name	Endianess	Contents
0	a_midmag	Big	0x008F010B
4	a_text	Little	The length in bytes of the header plus the text segment in both the file and the execution image.
8	a_data	Little	The length in bytes of the data segment in both the file and the execution image
12	a_bss	Little	The length in bytes of the bss segment in the execution image. The bss segment is not stored in the file, because its initial contents are always zero.
16	a_sym	Little	The length in bytes of the symbols "section" of the file. (The symbol table in the execution image consists of two "sections", one for the symbols and a second for the strings.[1])
20	a_entry	Little	The virtual address at which program execution is to begin.
24	a_trsize	Little	The size of the text relocation table. Used only for object files. Contains 0 for executable files.
28	a_drsize	Little	The size of the data relocation table. Used only for object files. Contains 0 for executable files.

The program image immediately follows the header. After recognizing this header, **load** *shall*:

- Synchronize the instruction and data caches from **load-base** to **load-base** + a_text + a_data + 0x20,
- Move the symbol "section" and string "section" of the symbol table from **load-base** + a_text + a_data + 0x20 to **load-base** + a_text + a_data + a_bss + 0x20.
- Zero a_bss bytes of memory beginning at **load-base** + a_text + a_data + 0x20,
- Release and unmap the physical memory from **load-base** + a_text + a_data + a_bss + a_sym + string_size[1] + 0x20 (i.e. from the end of the prepared client program memory image) to the end of the load area. (The goal of this step is to have the "available" properties in the /memory and /mmu nodes accurately reflect the memory actually consumed by the client program prepared image.)
- Set the **pc** in the *saved-program-state* to a_entry.
- Set the remaining elements of the *saved-program-state* to their initial values.

Note: The above header is that used by NetBSD [3].

If the a_midmag quadlet does not contain the specified value, the behavior of the Open Firmware **load** command with respect to client program recognition is as follows:

1. string_size is the 32-bit, little-endian value at **load-base** + a_text + a_data + a_sym which describes the length in bytes of the string "section" of the symbol table.

- If the file is in Forth source format [i.e. the file begins with the characters "\ " (0x5C, 0x20)], the file *shall* be interpreted as with "**`load-base file-size @ evaluate`**", or
- If the file is FCode [i.e. the file begins with the start1 FCode token (0xF1)], the file *shall* be evaluated as with "**`load-base 1 byte-load`**", or
- If the file contains another format that an implementation chooses to support, the file should be processed in an appropriate implementation-dependent manner, or
- If the file format is still not recognized, the image *shall* be treated as a raw binary image whose entry point is **`load-base`**.

6.2. Initial Program State

This section defines the "initial program state", the execution environment that exists when the first machine instruction of a *client program* 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.

6.2.1. Initial Register Values

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

Register(s)	Value
`pc`	Entry point of loaded program.
`psr`	Condition code values unspecified
	I = 0 = interrupts enabled
	F = enabled (if and only if the firmware is using fast interrupts)
	T = 0, if present
	M0 - M4 = SVC32 mode = 0x13
`sp`	See Section 6.2.2
`r0`	See Section 6.2.3
`r1,r2`	See Section 6.2.4
`r2`	0
`r3`	Reserved for platform binding
`r4`	Reserved for platform binding
Other user mode registers	0

6.2.2. Initial Stack

Client programs *shall* be invoked with a valid stack pointer (**`sp`**) with at least 4K bytes of memory available for stack growth. The stack pointer *shall* be 4-byte aligned.

6.2.3. Client Interface Calling Convention

To invoke a client interface service, a *client program:*

- Constructs a client interface *argument array* as specified in [1],
- Places the array's address in **`r0`**, and
- Transfers control to the *client interface handler*, with the return address in **`r14`**.

  A typical way of accomplishing this is:

```
\ First set r1 to Client Interface Handler entry point address, then:
mov   r14, pc  \ Establish return address pointer
mov   pc,  r1  \ Load pc with CIF Handler entry point
```

The client interface handler *shall* use various CPU registers as described in the following table. The term "preserved" below means that the register *shall* have the same value when returning as it did when the client interface service was invoked.

Register(s)	Value
`r0`	Argument array address on client interface entry.
	Result value (**`true`** or **`false`**) on client interface return.
`r1-r3`	Scratch registers; potentially destroyed.
`r4-r12`	Preserved.
`r13`	Stack pointer; preserved. Need not point to a valid stack upon entry. Consequently, a client program need not create a valid stack prior to calling the client interface handler.
`r14`	Contains return address and is potentially destroyed.
`psr`	Condition codes potentially destroyed; other fields preserved.

6.2.4. 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 **`r1`** and **`r2`**, 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.**

6.2.5. Trap table

In this section, *save-state-and-interact* means to save the CPU state to the extent possible, display (if possible) a message indicating that a trap occurred, and return control to the Open Firmware user interface if it is present.

A client program that installs its own trap table entries but wishes to continue using Open Firmware debugging services should preserve the Open Firmware trap table entries for any traps that the client program does not explicitly need to handle.

Open Firmware *shall* use the following format for its trap table.

Trap Table

0	`ldr pc, [pc, #56]`
4	`ldr pc, [pc, #56]`
8	
12	
…	
64	& handler for Exception 0
68	& handler for Exception 1

Note: The specification of a definite trap table format makes it easy for client programs to determine the addresses of the firmware's individual trap handlers. This is useful for client programs that wish to share the responsibility for handling traps with Open Firmware.

Typical Open Firmware trap responses are as follows:

Trap Type	Response
Reset (in ROM)	Restart Open Firmware.
External timer interrupt	Implement **`alarm`**, **`get-msecs`** and perhaps **`ms`**.
Other external interrupts	*save-state-and-interact* (Typically Open Firmware runs with these other interrupts disabled.)
Undefined instruction	Breakpoints.
Fast interrupt	*save-state-and-interact* (but may perform other functions needed for a specific hardware design).
Data access exception	*save-state-and-interact*
Address exception	*save-state-and-interact*

6.2.6. Virtual address space and memory allocation

When a client program begins execution, an Open Firmware implementation's use of any virtual address space outside of the ranges 0x0000.0000-0x0000.1000 and 0xF700.0000-0xF7FF.FFFF *shall* have ceased, except for the virtual address space and associated memory where the client program is loaded (see Section 6.1.2). Subsequently, the Open Firmware implementation *shall* not allocate virtual address space outside those ranges, except as explicitly requested by a client program.

Note: By inspecting the value of the `"available"` and `"existing"` properties in an MMU package, if such a package exists, a client program can determine precisely which ranges of virtual address space the firmware is using. For maximum portability, a client program ought not depend on the availability of any particular "hardcoded" virtual address.

6.2.7. Memory Cache(s)

The caches of the processor *shall* be enabled when the client program is invoked. (As specified by Section 6.1.2, the I-cache must be consistent with the D-cache for all memory areas occupied by the client program.)

6.2.8. Interrupt Sharing

This section describes techniques for handling interrupts when client programs are making active use of Open Firmware client services and device drivers.

Typically, Open Firmware implementations attempt to minimize their use of interrupts, for simplicity and robustness. On many systems, the only Open Firmware feature that demands the use of interrupts is **`alarm`**. On some systems, the implementation of **`get-msecs`** and sometimes **`ms`** also depend upon interrupts. (For the Digital Network Architecture, interrupts are required for all three of those functions, although **`ms`** can be implemented reasonably well without interrupts, at least for short durations.)

The only interrupt that is needed for these functions is a periodic timer tick. Relatively few high-level firmware functions depend upon those low-level interrupt-dependent functions. Typically, **`alarm`** is used to poll the console input device (usually a keyboard or a serial port) periodically to check for a "break" sequence indicating the user's desire to interrupt the current firmware activity. If the firmware is not receiving a periodic timer tick interrupt, alarm handlers will not be called. The typical result is that the user will be unable to interrupt arbitrary firmware activity from the console input device, but the firmware will be otherwise functional.

The most common uses for **`get-msecs`** are network protocol time-outs and device driver time-outs to prevent indefinite "hangs" when waiting for a device to respond. In many cases, when the network is responsive and devices are working correctly, a **`get-msecs`** failure (usually caused by the firmware not receiving a periodic

timer tick interrupt such that **get-msecs** always returns the same value) may not have any noticeable effect on overall firmware operation.

Nevertheless, even though most device drivers make scarce use of **alarm** and **get-msecs**, it cannot be guaranteed that a particular driver will not require them for a critical function. Therefore, it is prudent for client programs that use Open Firmware services, particularly those that involve I/O, to preserve the delivery of timer tick interrupts to the firmware.

6.3. Additional Client Interface Services

In addition to the list of client interface methods defined in Section 6.3.2 of [1], the `/openprom/client-services` node *shall* contain the following methods.

`restart`

```
IN: [string] command
OUT: <doesn't return>
```

Resets the system (as with the command **reset-all**) in such a way that the firmware, during its subsequent startup sequence, will execute *command* instead of performing the automatic default boot process. *command* is a string containing a sequence of user interface commands.

The permissible length of *command* may depend upon the availability of system resources such as free space in the NVRAM that the firmware uses for storing configuration variables. In the absense of the exhaustion of such resources for other uses, the firmware shall be able to accept *command* strings of at least 80 characters.

`call-static-method`

```
IN: [string] method, phandle, stack-arg1, …, stack-argP
OUT: catch-result, stack-result1, …, stack-resultQ
```

call-method invokes the static device method named *method* in the package identified by *phandle*. The *N_args*-2 arguments associated with *method*, *stack-arg1*, …, *stack-argP*, are pushed onto the Forth data stack, with *stack-arg1* on top of the stack, and *method* is executed as with the User Interface method `$call-method`, guarded by `catch`.

The result returned by `catch` is returned in *catch-result*. If *catch-result* is non-zero (meaning that an error occurred during the execution of *method*), the depth of the Forth data stack is restored to its depth prior to the execution of **call-method**. The values of the elements of the returned values portion of the argument array are undefined.

If *catch-result* is zero, **call-method** pops up to *K_returns*-1 items from the Forth data stack into the returned values portion of the argument array, with *stack_result1* corresponding to the top of the stack.

N_args and *K_returns* are stored in the argument array, and may be different for different calls to **call-method**. If the number of items *J* left on the Forth data stack as a result of the execution of *method* is less than *K_returns*-1, only *stack_result1* … *stack_resultJ* are modified; other elements of the returned values portion of the argument array are unaffected. If *J* is greater than *K_returns*-1, (*J* - *Q*) additional items are popped from the Forth data stack after setting *stack_result1* … *stack_resultQ*, so that, in all cases, the execution of **call-method** results in no net change to the depth of the Forth data stack.

A compliant Open Firmware implementation must allow at least six *stack_arg* and six *stack_result* items.

The behavior of `call-static-method` is undefined if *method* is not a static method.

6.4. Client Callbacks

If a client program takes control of timer interrupts or memory management, but needs to continue using Open Firmware client services thereafter, the client program must register a callback routine that supplies services that the firmware can use to perform the functions that the client program has subsumed.

The following subsections define those callback services.

The callback mechanism must operate in accordance with the specifications for `callback`, `$callback`, and `set-callback` in [1]. In particular, the callback mechanism operates in a fashion very similar to the client interface handler, except in the other direction (the firmware calls the client, instead of the client calling the firmware). The client program first invokes the `set-callback` client service to inform the firmware of the address of the callback handler routine. Subsequently, the firmware constructs an "argument array" and calls the callback handler routine, passing the address of the argument array as an argument (in **`r0`** for the ARM processor).

Several of the callback services defined below refer to the system-dependent MMU page size. For the ARM processor, that page size is 4096 bytes. Under some circumstances, the ARM MMU can deal with finer granularity than one page; specifically, it can apply a different protection to each of the four 1K sub-pages of a page. The MMU-related callback services defined below do not support sub-page granularity; all operations are performed in units of one page.

6.4.1. Virtual Address Translation Assist Callbacks

`map`

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

This callback service creates an address translation associating the region of virtual address space of size *size* beginning at the virtual address *virt* with the region of physical address of the same size beginning at the physical address *phys*.

mode specifies the values for the "AP" fields and the "C" and "B" bits of a page table entry. A page table entry is encoded as follows:

0000.0000.0000.0000.0000.AP0d.ddd0.CB00

where:

AP	Encodes the access permissions
dddd	Is the domain number
C	Is the cacheable bit
B	Is the bufferable bit
0	Is a bit whose value is 0
.	Is punctuation used to delimit groups of 4 bits

The fields and bits shown above are in accordance with the ARM Memory Management Unit Architecture definition in [2].

The indicated access permissions apply to the entire range specified by the arguments to **`map`**, which implies that the implementation of **`map`** must propagate the AP bits into each of the AP[0-3] sub-page access permission fields of any second-level descriptors that are used to accomplish the translation.

The firmware must specify domain 0 for any *mode* arguments that it generates internally, and must preserve the domain field unchanged for any *mode* value that it receives as the return value from an invocation of the **`translate`** callback and subsequently passes as an argument to the **`map`** callback.

The implementation of **`map`** may use any combination of first-level and second-level descriptors to accomplish its function, subject to the restriction that it must faithfully establish the requested translation without changing any other extant translations that are outside the requested range.

The *virt*, *phys*, and *size* arguments that the caller passes to this service must be multiples of the system-dependent MMU page size.

The return value *error shall* be zero if the operation succeeded, or a system-specific non-zero error code otherwise.

```
unmap
        IN: [address] virt, size
        OUT: throw-code
```

This callback service removes any address translation currently associated with the region of virtual address space of size *size* beginning at the virtual address *virt*. Typically, this involves setting the address translation for that virtual region to a system-specific “invalid” or “not mapped” state.

The *virt* and *size* arguments that the caller passes to this service must be multiples of the system-dependent MMU page size.

```
translate
        IN: [address] virt
        OUT: throw-code, error, [address] phys, mode
```

This callback service returns information about the address translation currently associated with the virtual address *virt*. If there is currently no valid translation for that virtual address, the *error* return value *shall* be a system-specific non-zero error code and the number of return values (as indicated by the N_returns cell of the argument array) *shall* be two. Otherwise, *error shall* be zero, the number of return values *shall* be four, *phys shall* be the physical address to which *virt* is translated, and *mode shall* specify the values for the “AP” fields and the “C” and “B” bits of a page table entry as described above under **`map`**.

The value returned in the *mode* result *shall* reflect the domain signified by the first-level descriptor and the access permissions signified by either the section descriptor (if the virtual address is mapped by a section descriptor) or the first sub-page (AP0) (if the virtual address is mapped by a second-level descriptor).

The firmware must specify domain 0 for any *mode* arguments that it generates internally, and must preserve the domain field unchanged for any *mode* value that it receives as the return value from an invocation of the **`translate`** callback and subsequently passes as an argument to the **`map`** callback.

The *virt* argument that the caller passes to this service must be a multiple of the system-dependent MMU page size.

6.4.2. Claim and Release Callbacks

```
claim-phys
        IN: [address] min_addr, [address] max_addr, size, align
        OUT: throw-code, error, [address] phys_addr
```

This callback service allocates consecutive pages of physical RAM subject to the following constraints:

- The beginning address of the first page *shall* be (using unsigned comparison) greater than or equal to *min_addr* and less than or equal to *max_addr.*
- The beginning address of the first page *shall* be a multiple of the value of *align*. The value of *align* passed to this callback must be a multiple of the system-dependent MMU page size.
- The size of the region *shall* be *size* bytes. The value of *size* passed to this callback must be a multiple of the system-dependent MMU page size.
- The pages *shall* be at consecutive addresses.
- The region *shall* not span the boundary between the maximum unsigned number and zero.

If the allocation fails, the *error* return value *shall* be a system-dependent non-zero error code and the number of return values (as indicated by the N_returns cell of the argument array) *shall* be two. Otherwise, *error shall* be zero, the number of return values *shall* be three, and *phys_addr shall* be the physical address of the beginning of the first allocated page.

There are three general possibilities for the *min_addr* and *max_addr* arguments:

- *min_addr* = 0, *max_addr* = <maximum unsigned integer>

  In this case, the firmware places no constraints on the address range of the return value; it is willing to accept any physical memory that meets the alignment constraints. (This is the usual case.)

- *min_addr = max_addr*

  In this case, the firmware is requesting the allocation of a specific range of physical memory. This case is rarely used; typically, when the firmware needs to use a specific page or pages of RAM, it claims that RAM prior to the time that the client program takes control of memory allocation. This callback is used to implement the firmware `"/memory"` node **`claim`** method, and it is possible that a user or program might attempt to claim a specific physical page, so this case is defined. It is of course possible that it may not be possible to satisfy such a request, in which case the callback would return an error code.

  If *min_addr=max_addr*, they must be multiples of *align*, otherwise it would not be possible to simultaneously satisfy both the alignment and range constraints.

- <otherwise>

  If *min_addr* and *max_addr* differ, but *min_addr* is not zero or *max_addr* is not the maximum unsigned integer, then the firmware is requesting the allocation of physical memory somewhere within a particular address range. The most common use of this case is for DMA-based I/O devices that cannot access arbitrary addresses. For example, in systems with bus-mastering ISA devices, there is often no way for those devices to supply DMA addresses outside the range 0 to 16 MBytes. Typically, the firmware uses this form of constrained physical allocation only for such cases.

  Since the definition of this callback service specifies that the constraint applies only to the beginning address of the first page of allocated memory, if the entire allocated range must fall within a certain region, the firmware must be careful to supply a value for *max_addr* such that the value of *max_addr + size* does not extend beyond the end of the desired region.

`release-phys`

```
IN: [address] phys_addr, size
OUT: throw-code
```

Free consecutive pages of physical RAM, making it available for later use. The size of the region to be freed is given by *size*. The *phys_addr* and *size* arguments passed to this callback must be multiples of the system-dependent MMU page size. The physical RAM within the region to be freed must either have been previously allocated by `claim-phys` or have already been “owned” by the firmware at the time that the client program took control of physical memory allocation.

`claim-virt`

```
IN: [address] min_addr, [address] max_addr, size, align
OUT: throw-code, error, [address] virt_addr
```

This callback service allocates consecutive page frames of virtual address space subject to the following constraints:

- The beginning address of the first page frame *shall* be (using unsigned comparison) greater than or equal to *min_addr* and less than or equal to *max_addr.*
- The beginning address of the first page frame *shall* be a multiple of the value of *align*. The value of *align* passed to this callback must be a multiple of the system-dependent MMU page size.
- The size of the region *shall* be *size* bytes. The value of *size* passed to this callback must be a multiple of the system-dependent MMU page size.
- The pages frames *shall* be at consecutive addresses.
- The region *shall* not span the boundary between the maximum unsigned number and zero.

If the allocation fails, the *error* return value *shall* be a system-dependent non-zero error code and the number of return values (as indicated by the N_returns cell of the argument array) *shall* be two. Otherwise, *error shall* be zero, the number of return values *shall* be three, and *virt_addr shall* be the virtual address of the beginning of the first allocated page frame.

There are three general possibilities for the *min_addr* and *max_addr* arguments:

- *min_addr* = 0, *max_addr* = <maximum unsigned integer>

  In this case, the firmware places no constraints on the address range of the return value; it is willing to accept any physical address that meets the alignment constraints. (This is the usual case.)

- *min_addr = max_addr*

  In this case, the firmware is requesting the allocation of a specific virtual address. This case is rarely used; typically, when the firmware needs to use to a specific virtual address, it claims that virtual address prior to the time that the client program takes control of address allocation. This callback is used to implement the firmware `"/mmu"` node **`claim`** method, and it is possible that a user or program might attempt to claim a specific virtual address, so this case is defined. It is of course possible that it may not be possible to satisfy such a request, in which case the callback would return an error code.

  If *min_addr=max_addr*, they must be multiples of *align*, otherwise it would not be possible to simultaneously satisfy both the alignment and range constraints.

- <otherwise>

  If *min_addr* and *max_addr* differ, but *min_addr* is not zero or *max_addr* is not the maximum unsigned integer, then the firmware is requesting the allocation of virtual address space somewhere within a particular address range. The most common use of this case is for DMA-based I/O devices that cannot access arbitrary addresses when used on a system with virtually-addressed DMA. For example, some DMA devices drive some number of high-order address lines to fixed values, so on a virtual-DMA system, the DMA addresses for those devices must be taken from a specific region. Typically, the firmware uses this form of constrained physical allocation only for such cases.

  Since the definition of this callback service specifies that the constraint applies only to the beginning address of the first page of allocated memory, if the entire allocated range must fall within a certain region, the firmware must be careful to supply a value for *max_addr* such that the value of *max_addr* + *size* does not extend beyond the end of the desired region.

`release-virt`

```
IN: [address] virt_addr, size
OUT: throw-code
```

Free consecutive page frames of virtual address space, making it available for later use. The size of the region to be freed is given by *size*. The values *virt_addr* and *size* passed to this callback must be multiples of the system-dependent MMU page size.

The virtual address space within the region to be freed must either have been previously allocated by `claim-virt` or have already been “owned” by the firmware at the time that the client program took control of virtual address space allocation.

6.4.3. Interrupt Callback

`tick`

```
IN: [address] intsave
OUT: throw-code
```

This callback service notifies the client program each time that the firmware handles a timer tick interrupt, giving the client program a chance to schedule or perform periodic operations based upon that tick.

If the client program has registered a callback handler, the firmware *shall* attempt to invoke the `tick` callback as part of the process of handling a timer tick interrupt. (Open Firmware typically uses timer tick interrupts to implement the `alarm` and `get-msecs` features). If the client program does not implement the `tick` callback, the *throw-code* result will be non-zero (according to the usual semantics of client callbacks), in which case the firmware *shall* proceed with the rest of its timer tick handling process as though the callback had not been attempted.

The firmware *shall* invoke the `tick` callback in IRQ mode with interrupts disabled.

intsave is the address of a 260-byte array of memory that is used to pass to and from the client program the values of certain processor registers as they existed just prior to the occurrence of the tick interrupt. The firmware *shall* set the contents of the array as follows, prior to invoking the `tick` callback. Each array entry

is a 32-bit little-endian integer representing the saved value of an ARM CPU register.

Offset (decimal)	Contents
0	PSR
4	r0
8	r1
12	r2
16	r3
…	…
56	r13 (SP)
60	r14 (LR)
64	r15 (PC)

For “banked” registers (registers for which the ARM processor has multiple copies for different modes), the saved register values *shall* be for the register set corresponding to the mode indicated in the saved PSR value.

After the `tick` callback returns, the firmware *shall* proceed with the rest of its timer tick handling process, and upon completion *shall* restore the state of the CPU registers to the values contained in the *intsave* array. The client program's `tick` callback handler may alter the contents of *intsave* in order to cause the firmware to restore the register state to a state that differs from the state that existed when the timer tick occurred.

The client program's `tick` callback handler must not enable interrupts during its execution, but may, by modifying the PSR value in the *intsave* array, cause a delayed change in the interrupt enabled/disabled state that will take effect upon completion of the firmware's timer tick handler.

The client program's `tick` callback handler must not invoke any Open Firmware client services during its execution. If Open Firmware client services need to be invoked as a result of timer ticks, the appropriate way to do so is for the client program to use the `tick` callback handler to schedule those activities for execution after the firmware's tick handler completes (typically by modifying the *intsave* array).

7. User Interface Requirements

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

7.1. Machine Register Access

The following *user interface* commands represent ARM 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.

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

`.registers`

Display **`r0`** through **`r15`** and **`psr`**.

7.1.1. Integer Registers

`psr`

Access saved copy of Program Status Register.

`r0` through **`r15`**
Access saved copies of integer registers.

`up`
Synonym for **`r9`**.

`tos`
Synonym for **`r10`**.

`rp`
Synonym for **`r11`**.

`ip`
Synonym for **`r12`**.

`sp`
Synonym for **`r13`**.

`lr`
Synonym for **`r14`**.

`pc`
Synonym for **`r15`**.

7.1.2. Floating-Point Registers

Implementation of floating point register access is optional.

`f0` through **`f7`**
Access saved copies of floating point registers.

7.1.3. SCC Registers

TBD

7.2. ROM Upgrade Method

The following command is optional.

7.2.1. `net-flash ( -- )`

Reprograms the firmware from the network.

7.3. Configuration Variables

There are no ARM-specific configuration variables.