# IEEE Standard for Boot (Initialization Configuration) Firmware: Instruction Set Architecture (ISA) Supplement for IEEE 1754

*Circuits and Devices*

*Communications Technology*

**IEEE Computer Society**

Sponsored by the
Bus Architecture Standards Committee

*Electromagnetics and Radiation*

*Energy and Power*

*Industrial Applications*

*Signals and Applications*

*Standards Coordinating Committees*

IEEE Std 1275.1-1994

IEEE

***Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.***

*November 18, 1994* *SH94234*

# IEEE Standard for Boot (Initialization Configuration) Firmware: Instruction Set Architecture (ISA) Supplement for IEEE 1754

Sponsor

**Bus Architecture Standards Committee of the IEEE Computer Society**

Approved September 22, 1994

**IEEE Standards Board**

**Abstract:** Firmware is the read-only-memory (ROM)-based software that controls a computer between the time it is turned on and the time the primary operating system takes control of the machine. Firmware's responsibilities include testing and initializing the hardware, determining the hardware configuration, loading (or booting) the operating system, and providing interactive debugging facilities in case of faulty hardware or software. The core requirements and practices specified by IEEE Std 1275-1994 must be supplemented by system-specific requirements to form a complete specification for the firmware for a particular system. This standard establishes such additional requirements pertaining to the instruction set architecture (ISA) defined by IEEE Std 1754-1994, IEEE Standard for a 32-bit Microprocessor Architecture.

**Keywords:** boot, configuration, debug, FCode, firmware, Forth, initialization, plug-in device, ROM

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-462-4

## Contents

# IEEE Standard for Boot (Initialization Configuration) Firmware: Instruction Set Architecture (ISA) Supplement for IEEE 1754

## 1. Overview

This standard specifies the application of IEEE Std 1275-1994, IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices, to computer systems that use the instruction set architecture (ISA) defined by IEEE Std 1754-1994,[1] including instruction-set-specific requirements and practices for debugging, client program interface, and data formats. These requirements are imposed upon firmware compliant with IEEE Std 1275-1994 when such firmware is used on a computer system that uses the above ISA. The requirements are *not* imposed on the ISA itself.

## 2. References

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

IEEE Std 1754-1994, IEEE Standard for a 32-bit Microprocessor Architecture.[2]

NOTE—Where the notation style differs between IEEE Std 1275-1994 and IEEE Std 1754-1994, the notation of this document follows that of IEEE Std 1275-1994.

## 3. Definitions of terms

In addition to the following terms, this standard uses technical terms as they are defined in IEEE Std 1275-1994 and in IEEE Std 1754-1994 (see clause 2):

**3.1 core specification:** Synonym for IEEE Std 1275-1994, i.e., the standard that specifies the system-independent and bus-independent requirements for Open Firmware.

**3.2 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.

## 4. 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 2-byte alignment (i.e., any even address is an acceptable *a-addr*). (This ISA requires 4-byte alignment at the hardware level. However, the Forth implementation can hide this restriction; doing so can result in worthwhile reductions in ROM size in some cases.) An implementation may allow 1-byte alignment of *a-addr* addresses, but shall not require alignment more strict than that of 2 bytes.

[1] Information on references can be found in clause 2.

[2] As this standard goes to press, IEEE Std 1754-1994 is approved but not yet published. The draft standard is, however, available from the IEEE. Anticipated publication date is December 1994. Contact the IEEE Standards Department at 1 (908) 562-3800 for status information.

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, in conjunction with the alignment requirements imposed by the instruction set architecture, implies 4-byte alignment of *qaddrs* and 2-byte alignment of *waddrs*.)

## 5. Client interface requirements

An Open Firmware client interface implementation for an IEEE 1754 compliant processor shall behave as described in this clause.

### 5.1 Client program loading

#### 5.1.1 Default load address

The default load address is the virtual address 0x4000. At least 0x80000 bytes of memory shall be available at that address. It is strongly recommended that as much memory as is practical for the particular system be available there, thus allowing the loading of large client programs.

#### 5.1.2 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 `bf_magic` and `bf_format` quadlets contain the specified values. If either quadlet does not contain the specified value, the behavior of the Open Firmware `load` command is implementation-dependent. The offsets given below are from the beginning of the loaded image. Each of the quadlets described below is in big-endian byte order.

| Offset | Name | Contents |
|---|---|---|
| 0 | `bf_magic` | 0x0103.0107 |
| 4 | `bf_text` | Size of the client program's code |
| 8 | `bf_data` | Size of the client program's initialized data |
| 12 | `bf_bss` | Size of the client program's uninitialized data area |
| 16 | `bf_pad1` | Undefined |
| 20 | `bf_origin` | Client program entry address |
| 24 | `bf_pad2` | Undefined |
| 28 | `bf_format` | 0xffff.ffff |

The program image immediately follows the header. After recognizing this header, `load` allocates and maps `bf_text` + `bf_data` + `bf_bss` bytes of memory beginning at the address given by `bf_origin`, moves the program image, of size `bf_text` + `bf_data`, to that address, and zeroes `bf_bss` bytes of memory beginning at `bf_origin` + `bf_text` + `bf_data`.

NOTE—Some existing client programs use other values in `bf_format`. An Open Firmware implementation may implement compatibility modes to handle such client programs. The details of such compatibility modes are outside the scope of this standard.

## 5.2 Initial program state

This subclause 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.

### 5.2.1 Register values

The CPU registers shall contain the following values:

| Register(s) | Value |
|---|---|
| `%psr` | S=1<br>ICC=0<br>EF=1 if floating-point coprocessor present<br>EC=1 if second coprocessor present<br>EE=0<br>ET=1<br>PIL: implementation-dependent value sufficiently low to allow the Open Firmware timer interrupt to occur<br>CWP: 0 |
| `%wim` | 2 |
| `%tbr` | See 5.2.4. |
| `%y` | 0 |
| `%i7` | 0 |
| `%o6,%i6` | See 5.2.2. |
| `%o1,%o2` | See 5.2.3. |
| `%o3` | Address of *client interface* handler. See 5.3. |
| Other registers | Other global (`%g0-%g7`), local (`%l0-%l7`), in (`%i0-%i5`), and out (`%o0,%o4,%o5,%o7`) registers may be used for conveying information required by other client interfaces that are outside the scope of this standard. Any registers that are not used for such purposes shall contain zero. |

NOTE—The stipulation that unused other registers contain zero makes it possible for a firmware system to support multiple different client interfaces simultaneously. For example, a firmware system might present both an Open Firmware client interface and also a different interface for compatibility with some existing client program. A client program can determine whether or not a particular client interface is present by testing for a nonzero value in one of the registers that that client interface uses. The presence of the Open Firmware client interface is denoted by a nonzero value in `%o3`. An earlier firmware system that was an ancestor of Open Firmware uses `%o0` to pass the (nonzero) address of its client interface data structure to the client program.

### 5.2.2 Initial stack

When the first machine instruction of a *client program* begins execution, there shall be a valid stack and the processor state shall have the following characteristics:

— `%i6` shall contain zero.

— `%o6` shall contain an 8-byte-aligned address referring to a location within an area of memory that the Open Firmware implementation has allocated for use as the client program's stack. That address shall be at least 96 bytes below the top address of the stack memory area (providing space for saving the window registers of the current window), and at least 8000 bytes above the bottom address of stack memory area (providing room for stack growth).

An Open Firmware implementation shall handle window overflow traps by saving the "local" and "in" registers of the window below the trap window to the address specified by the `%o6` register of that window.

An Open Firmware implementation shall handle window underflow traps by restoring the "local" and "in" registers of the window above the trap window from the address specified by the `%o6` register of that window.

### 5.2.3 Client program arguments

Registers `%o1` and `%o2` may be used to pass to the client program an array of bytes of arbitrary content, with `%o1` containing the base address of the array and `%o2` the length. If no such array is passed, `%o1` and `%o2` shall contain zero.

NOTES

1—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.

2—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.

### 5.2.4 Trap table

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

`%tbr` shall refer to a trap table that handles traps as described in this subclause.

Open Firmware trap table entries shall not contain PC-relative branch offsets, in order that client programs can copy trap table entries without modifying them.

| **Hardware Trap #** | **Name** | **Behavior** |
|---|---|---|
| 5 | Window overflow | See 5.2.2. |
| 6 | Window underflow | See 5.2.2. |
| 0x11–0x1f | Interrupt vectors | An Open Firmware implementation may use certain interrupt vectors for internal functions, for example, managing the timer used to implement the ALARM mechanism. Vectors that are not used for such purposes shall *save-state-and-interact*. |
| 0xff | Software trap 127 | This trap vector is used for Open Firmware breakpoints. The Open Firmware handler for this trap vector shall *save-state-and-interact*. |

A client program that installs its own trap table but wishes to continue using Open Firmware services should preserve the Open Firmware trap table entries for any traps that the client program does not explicitly need to handle. A good technique for doing this is to copy the contents of the Open Firmware trap table into the client program's trap table, and then to replace only those entries to be serviced directly by the client program.

A client program shall not alter entries within the Open Firmware trap table.

When an Open Firmware command interpreter is entered after a client program has begun execution (e.g., via a user abort sequence, the `enter` client interface service, or a trap), the Open Firmware implementation shall restore the trap table register to point to its own trap table. If execution of the client program is subsequently resumed (e.g., with the `go` command), the Open Firmware implementation shall restore the trap table register to the value previously established by the client program.

### 5.2.5 MMU

The *memory management unit (MMU)*, if present, shall be enabled.

NOTE—Many client programs require no knowledge of the details, or even the existence, of the MMU.

### 5.2.6 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 0xffd0.0000–0xffef.ffff and 0xfe00.0000–0xfeff.ffff shall have ceased, except for the virtual address space and associated memory that is allocated for the client program's code and data, as specified in the client program header. Subsequently, the Open Firmware implementation shall not allocate virtual address space outside those ranges, except as needed for the execution of subsequent client programs or as explicitly requested by a client program.

An Open Firmware implementation should use the virtual address space range 0xffd0.0000–0xffef.ffff in preference to the range 0xfe00.0000–0xfeff.ffff, to the extent that is possible. Furthermore, allocation within the range 0xfe00.0000–0xfeff.ffff should allocate higher addresses before lower addresses.

Client programs shall not depend on the ability to be loaded (as specified by its client program header) within either of those address ranges.

NOTE—By inspecting the value of "`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.

### 5.2.7 Memory cache(s)

IEEE Std 1754-1994 does not specify the cache organization, thus cache details depend on the system architecture. As a general guideline, it is recommended that the *initial program state* should have caches enabled.

## 5.3 Client interface handler

The *client interface handler* shall perform the following sequence of operations:

— Invoke the Open Firmware client interface service specified by the argument array whose address was in `%o0` when the code sequence was invoked.
— Place the return value (indicating success or failure of the attempt to invoke the service) back in `%o0`.
— Return control to the client program at the address equal to eight plus the value that was in `%o7` when the code sequence was invoked.

The execution of the client interface handler, including the invocation of the *client interface service*, shall preserve the contents of all CPU registers other than `%o0`, `%o1`, `%o2`, `%o3`, `%o4`, and `%o5`.

NOTE—This implies that in order to invoke a client interface service, a *client program* first constructs a client interface argument array and puts its address in `%o0`, puts the return address minus eight in `%o7`, and then jumps to the client interface handler (typically this is done by using a JMPL instruction with `%o7` as the destination register).

The client interface handler may assume that it is invoked with a valid stack with enough space free to store 24 quadlets (enough for the active register window and the global registers), and that operational window overflow and window underflow trap handlers are installed in the current trap table.

A *client program* calling a client interface service must ensure that when the client interface handler is invoked, the stack is valid and has enough space free to store 24 quadlets (enough for the active register window and the global registers), and that operational window overflow and window underflow trap handlers are installed in the current trap table.

NOTE—These conditions are met when a client program first begins execution under the control of an Open Firmware implementation. In order to call client interface services, the client program must not destroy the integrity of the stack.

## 6. User interface extensions

An Open Firmware user interface implementation for an IEEE 1754 compliant processor should implement the following additional commands.

**`catch-interrupt`** ( level -- )
Install simple interrupt hander for indicated interrupt priority level.

Establish a handler for interrupt *level* (1–15). If an interrupt occurs on that *level*, the handler sets the value of `interrupt-occurred?` to `true` (–1) and sets the value of `vector-used` to the interrupt level.

**`interrupt-occurred?`** ( -- a-addr )
`variable` contains `true` if an interrupt occurred.

A `variable` that will be set to `true` (–1) when an interrupt occurs on a level guarded by `catch-interrupt`.

**`pil!`** ( level -- )
Set the current CPU interrupt priority level (0 .. 15).

The other (noninterrupt-priority) bits within the Processor Status Register are not changed.

**`pil@`** ( -- level )
Return the current CPU interrupt priority level (0 .. 15).

**`spacec!`** ( byte addr asi -- )
Store *byte* at *addr* in space *asi.*

**`spacec@`** ( addr asi -- byte )
Fetch *byte* from *addr* in space *asi.*

**`spaced!`** ( qdata.lo qdata.hi qaddr asi -- )
Store 2 quadlets at *qaddr* in space *asi.*

*qdata.hi* is stored at *qaddr,* and *qdata.lo* at *qaddr*+4, using an STDA instruction. A trap may result if *qaddr* is not a multiple of eight.

**`spaced@`** ( qaddr asi -- qdata.lo qdata.hi )

Fetch 2 quadlets from *qaddr* in space *asi*.

*qdata.hi* is the contents of the location at *qaddr*, and *qdata.lo* is the contents of the location at *qaddr*+4. The operation is performed with an LDDA instruction. A trap may result if *qaddr* is not a multiple of eight.

**`spacel!`** ( quad qaddr asi -- )

Store quadlet *quad* at *qaddr* in space *asi*.

A trap may result if *qaddr* is not a multiple of 4.

**`spacel@`** ( qaddr asi -- quad )

Fetch quadlet *quad* from *qaddr* in space *asi*.

A trap may result if *qaddr* is not a multiple of 4.

**`spacew!`** ( w waddr asi -- )

Store doublet *w* at *waddr* in space *asi*.

A trap may result if *waddr* is not a multiple of 2.

**`spacew@`** ( waddr asi -- w )

Fetch doublet *w* from *waddr* in space *asi*.

A trap may result if *waddr* is not a multiple of 2.

**`vector-used`** ( -- a-addr )

`variable` contains the level of the last interrupt.

A `variable` that will be set to the interrupt level when an interrupt occurs on a level guarded by `catch-interrupt`.

## 6.1 Machine register access

Processors compliant with IEEE Std 1754-1994 contain multiple register "windows." The processor hardware physically implements a single set of the "global" registers (registers 0–7) and multiple sets (typically, seven or eight) of the "out," "local," and "in" register windows. From the point of view of a typical program, the number of logical windows is limited only by the amount of memory available for the program stack. The hardware, through "window overflow" and "window underflow" traps, manages the hardware register windows transparently to the program, saving them to and restoring them from memory as necessary.

The *saved-program-state* shall contain the values of the set of 16 windowed registers that was active when the state was saved. The process of saving the program state shall include flushing the other hardware register window sets to the locations reserved for them on the program's stack. If the registers that specify those locations are invalid, an Open Firmware implementation may omit the flushing of the corresponding register window sets.

The window register access commands (`%o0–%o7`, `%l0–%l7`, `%i0–%i7`) refer to one set of window registers at any given time; that set is known as the *displayed register set*. The various displayed register sets are denoted by small integers. Set zero is the set that was active when the program state was saved. Set one is the set that would be active if a RESTORE instruction were executed from set zero, and so on. The maximum set number is determined not by the number of hardware register window sets, but instead by the number of "logical" register window sets in use by the program at the time the state was saved. That maximum number can be less than, equal to, or greater than the number of register windows implemented by the hardware.

When the Forth interpreter (Open Firmware user interface) is invoked, after the program state is saved, the displayed register set shall be set to zero, and may be changed with the **w** command. If the displayed register set is zero, the register values shall be accessed from the *saved-program-state*. Otherwise, the values shall be accessed from the appropriate save area in the program stack.

The following commands represent registers within the *saved-program-state*. Executing the command returns the saved value of the corresponding register. The saved value can be changed by preceding the command with the new value and `to`. The actual registers are restored to the saved values when `go` is executed.

`%g0` through `%g7` ( -- n )
Access saved copies of "global" registers.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

`%i0` through `%i7` ( -- n )
Access saved copies of "in" registers.

Return (or set, if preceded by `to`) the value, within the window register save area for the current display window, corresponding to the contents of the register with the same name as the command.

`%l0` through `%l7` ( -- n )
Access saved copies of "local" registers.

Return (or set, if preceded by `to`) the value, within the window register save area for the current display window, corresponding to the contents of the register with the same name as the command.

`%o0` through `%o7` ( -- n )
Access saved copies of "out" registers.

Return (or set, if preceded by `to`) the value, within the window register save area for the current display window, corresponding to the contents of the register with the same name as the command.

`%pc` and `%npc` ( -- n )
Saved program counter and next program counter.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

`%psr` ( -- n )
Saved processor state register.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

`%tbr` ( -- n )
Saved trap base register.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

`%wim` ( -- n )
Window invalid mask register.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

`%y` ( -- n )
y register.

Return (or set, if preceded by `to`) the value, within the *saved-program-state*, corresponding to the contents of the register with the same name as the command.

**`%f0` through `%f31`** ( -- n )

Access floating-point registers.

Return (or set, if preceded by `to`) the value corresponding to the contents of the register with the same name as the command.

**`%fsr`** ( -- n )

Access floating-point state register.

Return (or set, if preceded by `to`) the value corresponding to the contents of the register with the same name as the command.

The following commands display the *saved-program-state*:

**`.locals`** ( -- )

Display all the integer registers in the current window.

**`w`** ( window# -- )

Set the current window for display of the `%i?`, `%o?`, and `%l?` registers.

**See also:** `.window`.

**`.window`** ( window# -- )

Display all the integer registers in the window specified by *window#*.

**Equivalent to:** `w .locals`

NOTE—the current window will be changed to the value given by *window#*.

**`.psr`** ( -- )

Formatted display of the saved processor state register.

This command sets the values in both program counter registers:

**`set-pc`** ( a-addr -- )

Set `%pc` to *a-addr* and `%npc` to *a-addr*+4.

## 6.2 Debugger extensions

The commands `dis`, `+dis`, `.instruction`, and `.adr` shall display addresses and symbol name offsets in hexadecimal.

**`return`** ( -- )

Execute until a return from subroutine is reached.

Set a breakpoint at the address given by register `%i7` + 8 and then execute `go`.

**`returnl`** ( -- )

Execute until a return from subroutine is reached.

Same as `return` except uses `%o7` instead of `%i7`.

## 6.3 Configuration variables

`watchdog-reboot?` ( -- reboot? ) N

If `true`, reboot automatically after watchdog reset.

Configuration variable type: *Boolean*. Suggested default value: `false`.

## 6.4 Restrictions

None.