Device Support Extensions to:

IEEE Std 1275-1994

Standard for Boot

(Initialization, Configuration)

Firmware

Revision: 0.2 DRAFT

Date: July 5, 1995

1. Overview

This document specifies the application of *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements*[1] to a variety of common peripheral types. It is intended that Open Firmware implementations supporting these device types will supply the methods and properties described herein.

2. References

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

[2] *ISO-639*, *Code for the representation of names of languages*, published by International Organization for Standardization.

[3] *ANSI/IEEE X3.215-1994, Programming Languages -- Forth.*

3. Keyboard devices

Open Firmware does not have a specific device class for keyboards; instead, keyboards are instances of the `"serial"` device class. However, certain features of keyboards (i.e., the ability to re-map keys, etc.) make it desirable to implement them as a separate device class.

In general, keyboard devices produce a hardware *scan-code* that is specific to the type of keyboard. These scan-codes are then mapped via software to produce the character code for the key, taking into account the state of "modifier" keys (e.g., Shift, Control) and the keyboard layout. The mapping of scan-codes to character value depends upon the keyboard layout; this is dependent upon the language that is being supported by the keyboard. E.g., the layout of keys for a French keyboard is different than that for an English keyboard.

For purposes of localization, it is necessary for the scan-code conversion to be controlled by the software, including Open Firmware (via the Client Interface). This section adds mechanisms to allow client programs to alter the scan-code conversion, based upon the keyboard (language) layout.

The language-specific layout is specified by means of a 2-character abbreviation, as described by [2]. The following languages may be supported:

CS	Czech
DA	Danish
NL	Dutch
EN	English (default)
FI	Finnish
FR	French
DE	German
HU	Hungarian
IT	Italian
NO	Norwegian
PL	Polish
PT	Portuguese
SK	Slovak
ES	Spanish
SV	Swedish

For testing of keyboard devices, additional low-level methods are defined for `"keyboard"` devices. **`get-scancode`** will read the next available scan-code from the keyboard; **`scancode->char`** will perform character mapping, using the same conversion as would normal **`read`**s. This enables test software to test for a specific character to terminate the test without having to know the scan-code to character translations.

3.1. "keyboard" device standard properties

"device_type"

Standard Open Firmware property; the value of this property for keyboard devices *shall* be `"keyboard"`.

"language"

Standard property, encoded as with **encode-string**, that indicates the current scan-code to character conversion, based upon the language's keyboard layout. This value indicates the language (i.e., keyboard layout scan-code conversion) to which the keyboard driver is currently set.

3.2. "keyboard" device methods

In addition to the Open Firmware standard **open**, **close** and **read** methods, the following methods shall be supported by an Open Firmware implementation of a `"keyboard"` device. Note that the **open** routine can take an optional argument which specifies the language (i.e., **scancode->char** mapping) to be used.

get-scancode (msecs -- scancode true | false)

This method returns the "raw" scan-code value of the next key alteration. `msecs` is the number of milliseconds to wait for a keystroke before reporting failure; a value of 0 implies wait until keystroke. If this timeout expires before a keystroke is read, **false** is returned. Otherwise, the `scancode` and **true** are returned.

scancode->char (scancode -- char true | false)

Using the `scancode`, perform translation to a character. If the `scancode` represents a modifier key (e.g., Shift), no translation will be available; in this case, **false** is returned. If the `scancode` represents a translated character, `char` and **true** are returned.

set-language (str len -- true | false)

If the keyboard driver can support the requested character-set, it **shall** set its `"language"` property to the value specified by `str,len`, and return a value of **true**. If it can not support the requested language, no change of `"language"` is made and a value of **false** is returned.

The reason for adding a special call, instead of just using the **property** (`setprop`) call, is to allow the device to "filter" the request. I.e., a scan-code conversion may not be available for the requested language. This call allows the device driver to change the property only if it can support the requested mapping.

3.3. Abort key sequence for keyboard devices

For PC-101 keyboards, the abort sequence (see *install-abort* in the IEEE 1275 glossary) shall be Ctrl-Brk.

4. Pointing devices

This class of device covers a broad category of "pointing" devices, the most common embodiment of which is the mouse. These devices typically can generate X,Y coordinates and button-press information on some periodic basis. The following properties and methods are defined for such devices.

4.1. "mouse" device standard properties

"device_type"

Standard Open Firmware property; the value of this property for pointing devices shall be `"mouse"`.

"#buttons"

Standard property, encoded as with **encode-int**, that indicates the number of physical buttons supported by the device. This property can be used to interpret the *`buttons`* value returned by the **get-event** method.

"absolute-position"

Standard property, whose presence indicates that this device supplies absolute X,Y coordinates (e.g., a graphics tablet). Absence of this property indicates that the device supplies relative X,Y position (e.g., a mouse).

4.2. "mouse" device methods

In addition to the Open Firmware standard **open** and **close** methods, the following methods shall be supported by an Open Firmware implementation of a "mouse" device.

Pointing devices typically supply data only when an "event" occurs (e.g., the mouse moves or a button is pressed). The following method attempts to obtain an event from the device, reporting whether an event occurred.

get-event **(msecs -- pos.x pos.y buttons true | false)**

Standard method for obtaining the next "event" of pointing devices. `msecs` is the number of milliseconds to wait for an event before reporting failure; a value of 0 implies wait until event. `pos.x`, `pos.y` return the positioning information; they are interpreted as unsigned or signed, depending upon the presence or absence of the "absolute-position" property. `buttons` returns a bit-mask (in the low-order bits) representing any buttons that are pressed; the number of significant bits to examine is defined by the "#buttons" property. The least significant bit corresponds to the primary (usually the left-most) button of the device. The top stack result indicates whether an event was detected within the time-out period.

5. Real Time Clock (RTC) Device

Open Firmware for Real Time Clocks defines the following properties and methods. The representation of time is defined by the TIME&DATE method of the ANS Forth standard [3].

5.1. "rtc" device standard properties

"device_type"

Standard Open Firmware property; the value of this property *shall* be "rtc".

5.2. "rtc" device methods

In addition to the Open Firmware standard **open** and **close** methods, the following methods shall be supported by an Open Firmware implementation of a "rtc" device.

get-time **(-- n1 n2 n3 n4 n5 n6)**

Return the current time as the integers `n1`…`n6`, where `n1` is the second {0…59}, `n2` is the minute {0…59}, `n3` is the hour {0…23}, `n4` is the day {1…31}, `n5` is the month {1…12}, and `n6` is the year (e.g., 1994).

set-time **(n1 n2 n3 n4 n5 n6 --)**

Set the current time from the integers `n1`…`n6`, where `n1` is the second {0…59}, `n2` is the minute {0…59}, `n3` is the hour {0…23}, `n4` is the day {1…31}, `n5` is the month {1…12}, and `n6` is the year (e.g., 1994).

6. Sound Device

In order to use a sound device within the context of Open Firmware (e.g., "boot beeps"), the following properties and methods *shall* be implemented.

6.1. "sound" device standard properties

"device-type"

Standard Open Firmware property; the value of this property *shall* be "sound".

"#channels"

Standard property, encoded as with **encode-int**, that represents the number of channels supported by the device.

"sample-resolution"

Standard property, encoded as with **encode-int**, that represents the number of bits of resolution for each sound sample.

"sample-width"

Standard property, encoded as with **encode-int**, that represents the number of bytes required for storing a sample.

"sample-rates"

Standard property, consisting of an array of integers, each encoded as with **encode-int**, that represents the rates (in hertz) at which this device can be sampled.

6.2. "sound" device standard methods

The following methods *shall* be implemented by a `"sound"` device.

open **(-- true | false)**

This Standard method prepares the `"sound"` device for subsequent **read**s or **write**s. An argument can be supplied (i.e., following a '**:**' in the path-name component, available via **my-args**) to specify sampling parameters. The argument is a string consisting of the external representation of the sample-rate to be used; if absent, an implementation-defined sample rate is used. An error is signalled by **open** (i.e., it returns **false**) if the requested sample rate can not be supported by the device.

close **(--)**

Standard Open Firmware behavior.

read **(addr size -- actual)**

Acquire sound data, storing the samples at `addr`. The sample rate is established by **open**.

write **(addr size -- actual)**

Output sound samples, stored at `addr`. The sample rate is established by **open**.

7. NVRAM Device

Access toNVRAM is supported by this device-type. The NVRAM is treated as a device that can be read and written using the standard Open Firmware **read** and **write** methods; the starting position within the NVRAM can be specified by the **seek** method.

The platform binding may define CRCs that are used to validate the integrity of the data within the NVRAM. To provide flexibility in using NVRAM, options are provided on the **open** method for NVRAM that determine whether the CRCs have to be valid at **open** time, and/or whether they are written at **close** time. These options appear as the *argument* component of the device specifier used to **open** the device (as with **open-dev**).

7.1. NVRAM properties

"device-type"

Standard Open Firmware property; the value of this property *shall* be `"nvram"`.

7.2. NVRAM methods

The following methods have the semantics of the Open Firmware methods:

read **(addr len -- actual)**

write **(addr len -- actual)**

seek (**pos.lo pos.hi -- status**)

The following methods have additional behavior depending upon the *argument* used to open the device.

open (**-- true | false**)

Standard method used to initiate access to the device and control how CRCs are used and/or generated. The **open** method *shall* use **my-args** to determine special handling, as follows:

	Empty. The **open** method will verify the CRCs within the NVRAM area. If correct, the **open** will succeed, returning **true**; subsequent calls to **write** will cause the CRCs to be calculated and stored when the device is **close**'d. If not correct, **open** will report failure by returning **false**.
`raw`	The **open** will succeed, without verifying the CRCs. Calls to **write** will not be reflected in the CRCs at **close** time.
`repair`	The **open** will succeed, without verifying the CRCs. However, the CRCs will be regenerated at **close** time.

close (**--**)

Standard method, whose behavior depends upon what the *argument* value was at the time the device was **open**'d. If *argument* was empty, or `repair`, the CRCs will be computed and stored.

8. Parallel port device

Access to the parallel port is supported by this device-type. The parallel port is treated as a byte-stream device.

8.1. Parallel port properties

"device-type"

Standard Open Firmware property; the value of this property *shall* be `"parallel"`.

8.2. Parallel port methods

The following methods have the semantics of the corresponding Open Firmware standard methods:

open (**-- true | false**)

close (**--**)

write (**addr len -- actual**)

9. /aliases Node Properties

The following aliases shall be provided under the "/aliases" node if an applicable device node exists:

keyboard

mouse

rtc

sound

nvram

parallel