

IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices

Sponsor

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

Approved March 17, 1994

IEEE Standards Board

Approved August 23, 1994

American National Standards Institute

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. Open Firmware provides the following: a mechanism for loading and executing programs (such as operating systems) from disks, tapes, network interfaces, and other devices; an ISA-independent method for identifying devices “plugged in” to expansion buses and for providing firmware and diagnostics drivers for these devices; an extensible and programmable command language based on the Forth programming language; methods for managing user-configurable options stored in nonvolatile memory; a “call back” interface allowing other programs to make use of Open Firmware services; and debugging tools for hardware, firmware, firmware drivers, and system software.

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

Copyright © 1994 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1994. Printed in the United States of America.

ISBN 1-55937-426-8

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE Standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This introduction is not a part of IEEE Std 1275-1994, IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices.)

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.

Historically, firmware design has been proprietary and often specific to a particular bus or instruction set architecture (ISA). This need not be the case. Firmware can be designed to be machine-dependent and easily portable to different hardware. There is a strong analogy with operating systems in this respect. Prior to the advent of the portable UNIX® operating system in the mid-seventies, the prevailing wisdom was that operating systems must be heavily tuned to a particular computer system design and thus effectively proprietary to the vendor of that system.

IEEE Std 1275-1994 (Open Firmware) is based on Sun Microsystems'® OpenBoot™ firmware. The OpenBoot design effort began in 1988, when Sun was building computers based on three different processor families. Thus, OpenBoot was designed from the outset to be ISA-independent. The first version of OpenBoot was introduced on Sun's SPARCstation™ 1 computers. Based on experience with those machines, OpenBoot version 2 was developed and was first shipped on SPARCstation 2 computers. This standard is based on OpenBoot version 2.

Open Firmware has the following features:

- A mechanism for loading and executing programs (such as operating systems) from disks, tapes, network interfaces, and other devices
- An ISA-independent method for identifying devices “plugged in” to expansion buses and for providing firmware and diagnostics drivers for these devices
- An extensible and programmable command language based on the Forth programming language
- Methods for managing user-configurable options stored in non-volatile memory
- A “call back” interface allowing other programs to make use of Open Firmware services
- Debugging tools for hardware, firmware, firmware drivers, and system software

The following individuals were members of the P1275 Working Group at the time this document was produced:

William M. (Mitch) Bradley, *Chair*

David M. Kahn, *Vice Chair*

John Rible, *Draft Editor*

Mike Williams, *Secretary*

Paul Fischer
Ron Hochsprung

Thanos Mentzelopoulos
David L. Paktor
Michael Saari

Paul Thomas
Mike Tuciarone

® UNIX is a registered trademark of ?, Inc.

® Sun Microsystems is a registered trademark of Sun Microsystems, Inc., in the United States and other countries.

™ OpenBoot is a trademark of Sun Microsystems, Inc.

® SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc.

™ SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

The following persons were on the balloting committee:

Ray S. Alderman
Keith D. Anthony
Norman R. Bartek
Chris Bezirtzoglou
John Black
Paul L. Borrill
William M. Bradley
Charles Brill
Andy Cheese
Alan Chin
David Cohen
Sourav K. Dutta
Wilhelm P. Evertz
Jurgen Feg
Wayne Fischer
Paul Fisher
Gordon Force
William A. Fox
Paul Fulton
Julio Gonzalez-Sanz

Mark Hassel
Michael C. Hayward
Ron Hochsprung
David V. James
Horace Jones, Jr.
David M. Kahn
Steve Kleiman
Ralph Lachenmaier
Erik Lode
Larry McMahan
William E. Molyneaux
James D. Mooney
Klaus Dieter Mueller
Gregory C. Novak
Michael Orlovsky
Kamiar Khani Oskouee
David L. Paktor
Chandresh J. Patel
Mira Pauker

Ludwig Pregernig
William Ramsay
Frederick E. Sauer
Rudolf Schubert
Donald Senzig
Ravi Shankar
Kerry Shore
Robert Snively
Paul Thomas
Michael G. Thompson
Robert Tripi
Yoshiaki Wakimura
Martin Walsh
Eike Waltz
Michael Wenzel
Martin J. Whittaker
Michael G. Williams
Yoshio Yamaguchi
Oren Yuen
Janusz Zalewski

When the IEEE Standards Board approved this standard on March 17, 1994, it had the following membership:

Wallace S. Read, Vice Chair

Andrew G. Salem, Secretary

Donald C. Loughry, Chair

Gilles A. Baril
Bruce B. Barrow
José A. Barrios de la Paz
Clyde R. Camp
James Constantino
Stephen L. Diamond
Donald C. Fleckenstein
Jay Forster*
Ramiro Garcia

Donald N. Heirman
Richard J. Holleman
Jim Isaak
Ben C. Johnson
Sonny Kasturi
Lorraine C. Kevra
E. G. "Al" Kiener
Ivor N. Knight

Joseph L. Koepfinger*
D. N. "Jim" Logothetis
L. Bruce McClung
Marco W. Migliaro
Mary Lou Padgett
Arthur K. Reilly
Ronald H. Reimer
Gary S. Robinson
Leonard L. Tripp

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall
Richard B. Engelman
David E. Soffrin
Stanley I. Warshaw

Paula M. Kelty
IEEE Standards Project Editor

Contents

CLAUSE	PAGE
1. Overview	1
1.1 Purpose and scope	1
1.2 Firmware problems	1
1.3 Solutions	2
1.4 Document organization	2
1.5 Compliance	3
2. References, definitions, and assumptions.....	4
2.1 References	4
2.2 Special word usage	4
2.3 Definitions of terms	5
2.4 Forth language assumptions and conventions	10
2.5 Hardware assumptions.....	12
3. Internal structure	13
3.1 Forth dictionary	13
3.2 Device tree.....	13
3.3 Configuration memory.....	19
3.4 Standard property names.....	20
3.5 Standard system nodes.....	21
3.6 Standard packages.....	22
3.7 Standard device types	25
3.8 Standard support packages.....	28
4. Internal procedures	33
4.1 Forth language environment.....	33
4.2 Start-up sequence	33
4.3 Path resolution.....	37
5. Device interface.....	44
5.1 General	44
5.2 FCode evaluator.....	46
5.3 FCode functions	48
5.4 Standard FCode program.....	61
6. Client interface	62
6.1 General	62
6.2 Client program environment	62
6.3 Client interface services.....	63
7. User interface.....	70
7.1 General	70
CLAUSE	PAGE

7.2	Standard command interpreter	71
7.3	Forth language command group.....	73
7.4	Administration command group.....	84
7.5	Firmware Debugging command group.....	91
7.6	Client Program Debugging command group.....	93
7.7	FCode Debugging command group.....	96
ANNEXES		
Annex A	Open Firmware glossary.....	97
Annex B	Open Firmware terminal emulator control sequences.....	195
Annex C	The tokenizer	197
Annex D	Sun4c bus specifics.....	199
Annex E	SCSI host adapter package class.....	207
Annex F	Answers to common questions	229
Annex G	Summary lists.....	239
Annex H	Historical notes	243
Annex I	Index of Open Firmware glossary terms.....	251
Annex J	Bibliography	262

IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices

1. Overview

1.1 Purpose and scope

This document describes a software architecture for the *firmware* that controls a computer before the operating system has begun execution. Typically, firmware is stored in read-only memory (ROM) or programmable read-only memory (PROM), so that it may be executed immediately after the computer is turned on.

The main jobs of the firmware are to test the machine hardware and to boot the operating system, usually from a mass storage device or a network. The operating system may also require other services from the firmware. Finally, firmware often provides some support for interactive hardware and software debugging. In addition to the main operating system, other programs, such as diagnostic operating systems, may utilize firmware services.

This standard uses *OpenBoot PROM Architecture Specification* [B6]¹ as a starting point, and is bus, vendor, operating system (OS), and instruction-set-architecture (ISA)-independent. Supplements (numbered 1275. x) include specifications for this standard's application to particular ISAs and buses.

This document specifies firmware that controls the operation of a computer system before the primary operating system has taken control of the machine. The material specified includes facilities for determining the hardware configuration; testing, identification, and use of *plug-in devices* prior to primary OS control; reporting the hardware configuration to the operating system; the user interface for controlling these operations; and debugging facilities for hardware and system software.

Additional introductory material can be found in annex F.

1.2 Firmware problems

In an open-systems environment, the job of loading the operating system is greatly complicated by the possibility of user-installed I/O devices. If the *firmware* developer knows in advance the complete list of I/O devices from which the operating system may be loaded, then the software drivers for those devices may easily be included in the firmware. If, however, new bootable devices (devices from which the operating system may be loaded) may be added to the system later, then the firmware must have a way to acquire boot drivers for those devices. The obvious solution of shipping a complete set of system firmware with each new device quickly becomes impractical as the number of devices and systems grows.

A similar situation applies to the devices used for displaying messages showing the progress of the testing and booting processes. The firmware must have a driver for each device on which it wishes to display messages. One solution is to require every display device to emulate some baseline device. This solution works, but the constraints that it imposes on hardware can increase costs and stifle innovation.

Hardware innovation can proceed more rapidly when a single generic version of the operating system can work on several different computers within the same family. If the different computers look exactly the same to the software, then this is easy, but it is rare to find two different computers that look *exactly* the same at the operating system level. However, since the firmware can be considered in some sense to be part of the hardware, then the firmware can sometimes make the operating system's task of *autoconfiguration* (adapting to minor hardware

¹ The numbers in brackets preceded by the letter B correspond to those of the bibliography in annex J.

differences) easier, either by hiding the differences or by reporting the hardware characteristics so the operating system does not have to guess.

1.3 Solutions

The Open Firmware architecture solves those problems, and in addition, provides extensive interactive features for hardware and software debugging.

The design of Open Firmware is processor-independent, and every effort was made to eliminate knowledge of machine details from the specification of its interfaces.

The following Open Firmware features are notable:

- **Plug-in device drivers.** New devices may be added to an Open Firmware system and used for booting or message display without modification to the main Open Firmware system ROM. Each such device has its own *plug-in driver*, usually located in a ROM on the device itself. Thus, the set of I/O devices supported by a particular system may evolve without requiring changes or upgrades to the system ROM.
- **FCode.** Plug-in drivers are written in a byte-coded machine-independent interpreted language called *FCode*. FCode is based on Forth semantics. Since FCode is machine-independent, the same device and driver can be used on machines with different CPU instruction sets. Each Open Firmware system ROM contains an FCode interpreter.
- **Device tree.** The set of devices attached to the system, including permanently installed devices and *plug-in devices*, is described by an Open Firmware data structure known as the *device tree*. The operating system may inspect the device tree to determine the hardware configuration of the system. Each device in the device tree is described by a *property list*. The set of *properties* describing a device is arbitrarily extensible so that any type of device and any kind of information that needs to be reported about the device can be accommodated.
- **Modularity.** Some Open Firmware features (such as booting) are required, and others are optional. The set of Open Firmware features supported on a particular system may be chosen to meet the goals and constraints of that system.
- **Programmable user interface.** The Open Firmware user interface is based on the industry-standard interactive programming language Forth so that sequences of user commands can be combined to form complete programs. This provides a powerful capability for debugging hardware and software; Open Firmware is a very good tool for the initial “bring-up” of new hardware and software. In addition, the Open Firmware programming features can often be used to implement “work-arounds” for many kinds of system bugs.
- **FCode debugging.** The Open Firmware user interface language (Forth) and the FCode language share a common interpretation mechanism, so it is easy to develop and debug *FCode programs* with built-in Open Firmware tools.
- **Operating system debugging.** Open Firmware has commands for debugging operating system code, often making the use of a kernel debugger unnecessary.

1.4 Document organization

In this document, Open Firmware is described in terms of its external interfaces and the internal structures and procedures on which those interfaces depend. See figure 1. The content of the clauses and annexes is as follows:

- Clause 1 provides the background for understanding the goals and objectives of this document.
- Clause 2 provides a list of documents used as references, defines the terminology used, and describes the assumptions made by this standard.
- Clauses 3 and 4 define the internal structures and procedures upon which the Open Firmware model is based.
- Clause 5 defines the *device interface* for identification and use of *plug-in devices*.
- Clause 6 defines the *client interface*, which provides services to booted programs.

- Clause 7 defines the *user interface*, a *command interpreter* for human use.
- Annex A gives detailed definitions of all the individual *commands*, *methods*, *properties*, *configuration variables*, and strings mentioned in this document.
- Annex B defines the escape sequences of the terminal emulator *package*.
- Informative annexes C through I give additional useful information, including *device driver* source code examples, suggested behavior of development tools, historical and compatibility notes, and an index of glossary terms.

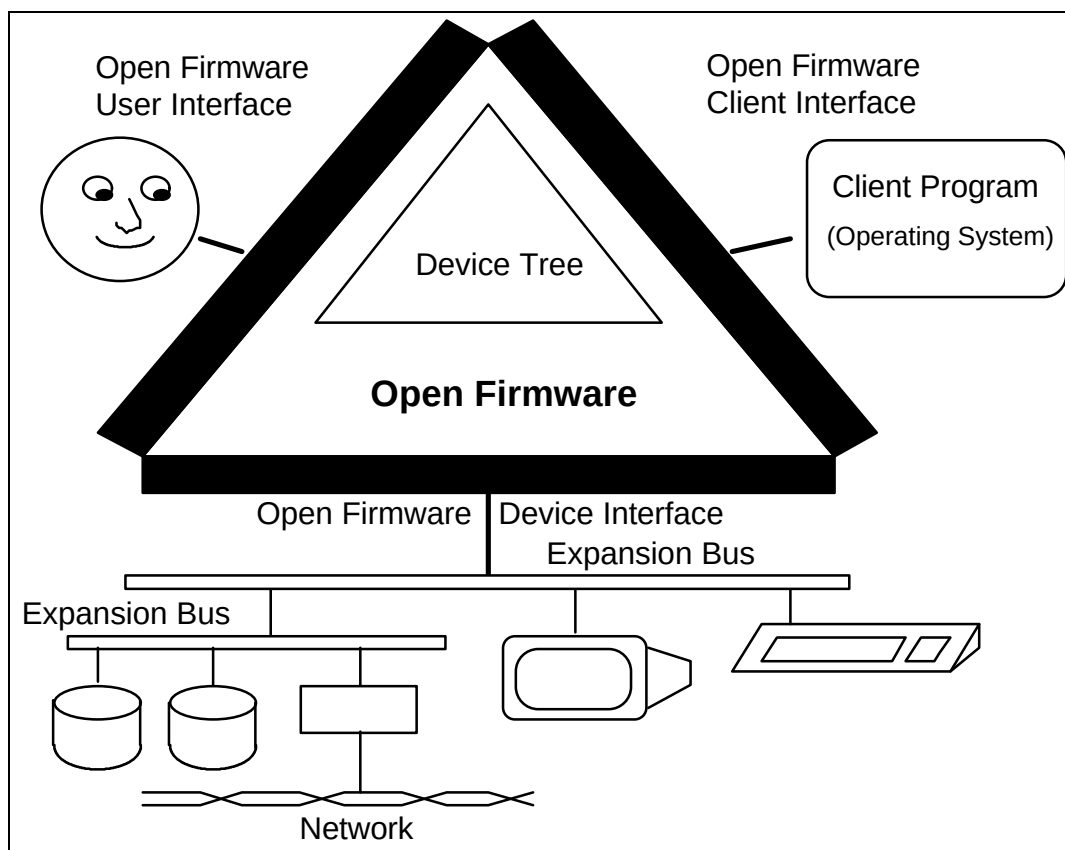


Figure 1—Typical Open Firmware system diagram

1.5 Compliance

In order to comply with this standard, a system shall implement at least one of the following interfaces:

Interface	Clause	Requirements given in subclause
Device interface	5	5.1.2
Client interface	6	6.1.2
User interface	7	7.1.2

A system claiming compliance with this standard shall clearly specify which of the above interfaces are claimed to be compliant.

2. References, definitions, and assumptions

2.1 References

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

ANSI X3.64-1979 (Reaff 1990), Additional Controls for Use with the American National Standard Code for Information Interchange.²

ANSI X3.215-1994, American National Standard for Information Systems—Programming Languages—Forth.

IEEE Std 100-1992, The New IEEE Standard Dictionary of Electrical and Electronics Terms (ANSI).³

IEEE Std 1275.1-1994, IEEE Standard for Boot (Initialization Configuration) Firmware: Supplement for IEEE 1754 ISA.

IEEE Std 1275.2-1994, IEEE Standard for Boot (Initialization Configuration) Firmware: Supplement for IEEE 1496 Bus (SBus).

IEEE P1275.3, Standard for Boot (Initialization Configuration) Firmware—Supplement for VMEbus, D1, August 1994.⁴

IEEE P1275.4, Standard for Boot (Initialization Configuration) Firmware—Supplement for IEEE 896 (Futurebus+) Bus, D13, August 1994.

ISO 8859-1 : 1987, Information processing—8-bit single-byte coded graph character sets—Part 1: Latin alphabet No. 1.⁵

RFC783, Trivial File Transfer Protocol (TFTP) Protocol Definition, NIC, June 1981.⁶

RFC906 Bootstrap Loading using TFTP, NIC, June 1984.

2.2 Special word usage

The following words have very specific meanings and are used in this standard to differentiate between required and optional features of the Open Firmware specification:

- The word *shall* is used to indicate *mandatory* requirements.
- The word *should* is used to indicate *advisory* requirements (that which is strongly recommended).
- The word *may* is used to indicate *optional* requirements.

² ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

³ IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331.

⁴ Numbers preceded by the letter P are authorized standards projects that were not approved by the IEEE Standards Board at the time of this document's publication. For information about obtaining drafts, contact the IEEE.

⁵ ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

⁶ Internet RFCs are retrievable by FTP at ds.internic.net/rfcnnn.txt (where nnn is a standard's publication number, such as 783 or 906), or call InterNIC at 1-800-444-4345 for information about receiving copies through the mail.

2.3 Definitions of terms

The following definitions give the meanings of the technical terms as they are used in this standard. Terms defined herein are italicized upon their first occurrence in each subclause throughout the rest of the document. Terms related to the Forth programming language are defined in ANSI X3.215-1994.⁷

2.3.1 active package: The package, if any, whose methods are accessible by name to the command interpreter, and to which newly created methods and properties are added.

2.3.2 alignment: The suitability of particular addresses for accessing particular types of data. For example, some processors require even addresses for accessing 16-bit data items.

2.3.3 big endian: A representation of multibyte numerical values in which bytes with greater numerical significance appear at lower memory addresses.

2.3.4 boot: To load and execute a client program.

2.3.5 built-in device: A device that is either permanently attached to the computer system, not easily removable, or present in all system configurations (i.e., not optional).

2.3.6 byte: A unit of computer data consisting of 8 bits.

2.3.7 cell: The primary unit of information in the architecture of a Forth System. See 2.3.2.

2.3.8 child node: A node that “descends” from another node, i.e., all nodes except the root node. *See also:* **parent node**.

2.3.9 client execution environment: The machine state that exists when a client program begins execution.

2.3.10 client interface: A set of data and procedures giving a client program access to client interface services.

2.3.11 client interface handler: A mechanism by which control and data are transferred from a client program to the firmware, and subsequently returned, for the purpose of providing client interface services.

2.3.12 client interface services: Those services that Open Firmware provides to client programs, including device tree access, memory allocation, mapping, console I/O, mass storage, and network I/O.

2.3.13 client program: A software program that is loaded and executed by Open Firmware (or a secondary boot program). (The client program may use services provided by the Open Firmware client interface.)

2.3.14 close: To destroy a package instance.

2.3.15 colon definition: A command defined as a sequence of previously existing commands.

2.3.16 command: As applied to this standard, a procedure in the Forth programming language. The execution of a command performs some operation, usually affecting the state of one or more system resources in a predefined way. (New commands may be defined as sequences of previously defined commands. Most commands have human-readable names expressed as a sequence of textual characters. *See also:* **Forth word; word name**.)

2.3.17 command group: A set of commands with defined behaviors, the group as a whole providing some particular capability (for example, one command group is concerned with client program debugging).

⁷Information on references can be found in 2.1.

2.3.18 command interpreter: The portion of a Forth system that processes user input and Forth language source code by accepting a sequence of textual characters representing Forth word names and executing the corresponding Forth words.

2.3.19 configuration variable: A named parameter, whose value is stored in nonvolatile memory, that controls some aspect of the firmware's behavior.

2.3.20 console: A device used as the primary means of communication with a human being, consisting of an input device, used for receiving information supplied by the human, and an output device, used for sending information to the human. (Typically, a console is either an ASCII terminal connected to a serial port or the combination of a text/graphics display device and a keyboard.)

2.3.21 current instance: The package instance whose private data is currently accessible.

2.3.22 data stack: A stack that may be used for passing parameters between Forth definitions.

2.3.23 decompiler: A software component that takes one or more compiled Forth commands and generates the equivalent text representation for those commands.

2.3.24 defer word: A Forth word whose name has been entered into the dictionary (by the defining word **defer**), but whose action was left unresolved and may be resolved at a later time.

2.3.25 device: A hardware unit that is capable of performing some specific function.

2.3.26 device alias: A shorthand representation for a device path.

2.3.27 device arguments: The component of a node name that is provided to a package's **open** method to provide additional device-specific information.

2.3.28 device driver: The software responsible for managing low-level I/O operations for a particular hardware device or set of devices. Contains all the device-specific code necessary to communicate with a device and provides a standard interface to the rest of the system. *See also:* **firmware device driver; operating system device driver.**

2.3.29 device interface: One of the interfaces specified in this standard that allows devices to be identified, characterized, and used to assist other Open Firmware functions such as booting.

2.3.30 device node: A particular entry in the device tree, usually describing a single device or bus, consisting of properties, methods, and private data. (A device node may have multiple child nodes and has exactly one parent node. The root node has no parent node.)

2.3.31 device path: A textual name identifying a device node by showing its position in the device tree.

2.3.32 device specifier: Either a device path, a device alias, or a hybrid path that begins with a device alias and ends with a device path.

2.3.33 device tree: A hierarchical data structure representing the physical configuration of the system. (The device tree describes the properties of the system's devices and the devices' relationships to one another. Most Open Firmware elements [devices, buses, libraries of software procedures, etc.] are named and located by the device tree.)

2.3.34 device type: Identifies the set of properties and package classes that a node is expected to implement. Specified by the "**device_type**" property.

2.3.35 disassembler: A program that translates machine code into an equivalent human-readable assembly-language representation.

2.3.36 disk label: Contains descriptive information, usually in a well-known location such as physical block zero, about the device and the media and may include logical partitioning information.

2.3.37 doublet: A unit of computer data consisting of 16 bits.

2.3.38 driver name: The component of a node name that corresponds to the value of the device's "**name**" property.

2.3.39 FCode: A computer programming language defined by this standard, which is semantically similar to the Forth programming language but is encoded as a sequence of binary byte codes representing a defined set of Forth definitions.

2.3.40 FCode driver: A device driver, written in FCode, intended for use by Open Firmware and its client programs.

2.3.41 FCode function: A self-contained procedural unit of the FCode programming language to which an FCode number may be assigned.

2.3.42 FCode evaluator: The portion of Open Firmware that processes FCode programs by reading a sequence of bytes representing FCode numbers and executing or compiling the associated FCode functions.

2.3.43 FCode number: A number from 0 to 4095 (conventionally written in hexadecimal as 0x00 to 0x0FFF) that denotes a particular FCode function.

2.3.44 FCode probing: The process of locating and evaluating an FCode program.

2.3.45 FCode program: A program encoded as a sequence of byte codes according to the rules of the FCode programming language.

2.3.46 FCode source: An FCode program in text form. *See also:* **tokenizer**.

2.3.47 firmware: A program, typically stored in read-only memory, that controls a computer from the time that it is turned on until the time that the primary operating system assumes control of the computer.

2.3.48 firmware device driver: A device driver intended for use by firmware. *Contrast with:* **operating system device driver**. *See also:* **device driver**.

2.3.49 Forth word: *See:* **command**.

2.3.50 frame buffer: A hardware device that is used as an interface between a computer and a video monitor, generally containing an array of memory that is written by the computing system (software) and is read by special hardware for the purpose of display.

2.3.51 ihandle: A cell-sized datum identifying a particular package instance.

2.3.52 instance: *See:* **package instance**.

2.3.53 leaf node: A device node that has no children.

2.3.54 least significant: Within a group of data items (e.g., bits or bytes) that, taken as a whole, represents a numerical value, the item within the group with the smallest numerical weighting.

2.3.55 little endian: A representation of multibyte numerical values in which bytes with lesser numerical significance appear at lower memory addresses.

2.3.56 load: To move the image of a client program from a long-term storage medium (such as a disk) into memory where it may be executed.

2.3.57 memory management unit (MMU): A device that performs address translation between a CPU's virtual addresses and the physical addresses of some bus; typically, the bus represented by the root node.

2.3.58 method: A software procedure associated with a package.

2.3.59 MMU: *See:* **memory management unit**.

2.3.60 most significant: Within a group of data items (e.g., bits or bytes) that, taken as a whole, represents a numerical value, the item within the group with the greatest numerical weighting.

2.3.61 node: In the context of Open Firmware, node is a synonym for device node. *See also:* **device node**.

2.3.62 node name: A text string of the form “driver-name@unit-address:device-arguments”, which identifies a device node within the address space of its parent.

2.3.63 nonvolatile memory: Computer memory whose contents are preserved when the system power is off.

2.3.64 open: To create a package instance.

2.3.65 Open Firmware: Firmware conforming to IEEE Std 1275-1994, IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices.

2.3.66 operating system device driver: A device driver intended for use by a primary operating system. *Contrast with:* **firmware device driver**. *See also:* **device driver**.

2.3.67 package: The combination of a node's properties, methods, and private data.

2.3.68 package instance: A data structure resulting from the opening of a particular package, consisting of a set of values for the package's private data.

2.3.69 parent node: The node to which a device node is attached. Each device node has exactly one parent node, except the root node, which has none. (A device node descends from its parent node. Traveling “up” the device tree takes one through parent nodes to the root node. Traveling “down” the device tree takes one through child nodes to the leaf nodes.)

2.3.70 phandle: A cell-sized datum identifying a particular package.

2.3.71 physical address: A unique identifier that selects a particular device from the set of all devices connected to a particular bus.

2.3.72 physical address space: The set of possible physical addresses for a particular bus.

2.3.73 plug-in device: A device that may be installed and removed at will, especially a device that is attached to a bus intended for system expansion.

2.3.74 plug-in driver: A package, usually associated with a plug-in device and serving as the interface to that device, that is created by evaluating an FCode program resident on that device.

2.3.75 printable character: A character in the range 0x21 through 0x7E or the range 0xA1 through 0xFE (see 2.3.3).

2.3.76 private data: Data, associated with a package, that is used by the methods of that package but is not intended for use by other software.

2.3.77 probe address: The address of a device that is known when the associated FCode program begins execution.

2.3.78 prop-encoded-array: The primitive data type, consisting of a sequence of bytes, used to represent a property value.

2.3.79 property: A descriptive item, consisting of an identifying property name and an associated property value, that represents some characteristic of a device node or of its associated device.

2.3.80 property encoding: A specific data format, defined by this standard, that is used to represent various types of information within a prop-encoded-array.

2.3.81 property name: A text string used to specify, or name, a particular property.

2.3.82 property value: The data portion of a property, stored in property encoding format.

2.3.83 quadlet: A unit of computer data consisting of 32 bits.

2.3.84 root node: The device node that is the root of the device tree.

2.3.85 saved program state: The set of information, necessary to begin or resume the execution of a client program, describing the machine state (including CPU registers) that will be established upon resumption or initiation of client program execution.

2.3.86 script: An area of nonvolatile memory reserved for user interface commands to be evaluated at particular times during the Open Firmware start-up sequence.

2.3.87 secondary boot program: A client program whose purpose is to load and execute another client program.

2.3.88 select: Context determines which of the following applies:

- To establish a particular device node as the active package.
- To establish a particular device as either the console input device or console output device.
- To establish a particular instance as the current instance.

2.3.89 stack: A last-in, first-out (LIFO) data structure. This document sometimes uses the phrase “the stack” to mean “the Forth data stack”.

2.3.90 stack diagram: A notational convention used to show the effect of a Forth word on the data stack and, where applicable, the input buffer and return stack. See ANSI X3.215-1994 for syntactic details.

2.3.91 static method: A method that can be executed without an instance of its package.

2.3.92 support package: A package, residing in the **/packages** node, that provides a service to assist in the implementation of a particular device type.

2.3.93 terminal emulator: Software that makes a frame-buffer appear to have the characteristics of a cursor-addressable text terminal.

2.3.94 text window: That portion of a display screen that is being used to display text (human-readable characters and words).

2.3.95 tokenizer: A development tool that converts FCode source code into a (binary) FCode program.

2.3.96 trace: To execute the component steps of a computer program, displaying the state of selected system resources after each step.

2.3.97 unit address: The component of a node name that indicates the device node's position within the address space defined by its parent node.

2.3.98 user interface: The portion of a firmware system that processes commands entered by a human. (The user interface defined by this standard consists of a Forth command interpreter plus a set of Forth words for interactively performing various Open Firmware functions. In its fully elaborated form, the Open Firmware user interface gives interactive access to all Open Firmware capabilities.)

2.3.99 value word: A Forth word created by the defining word **value**. (A **value** word, when executed by itself, places a numeric value on the data stack, much like a **constant**. The numeric value of a **value** word is changed by preceding it with the Forth word **to**.)

2.3.100 var-aligned: Alignment suitable for the storage of a Forth variable.

2.3.101 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.)

NOTE—The use of the term *virtual address* in this document does imply that Open Firmware necessarily uses mapping hardware if it is present on a particular system.

2.3.102 word: See: **Forth word**.

2.3.103 word name: A text string denoting a particular Forth word.

2.4 Forth language assumptions and conventions

This subclause contains basic rules for numeric interpretation, syntax, stack comments and so on. The following conventions and assumptions apply to all commands in this document (unless specified otherwise).

2.4.1 General conventions

- All numbers in this document are decimal numbers unless indicated otherwise. Hexadecimal (base sixteen) numbers are indicated in the text with a “0x” prefix, i.e., 0x1234, or with the suffix “(hex)”. Note that this is a documentation convention: The *command interpreter* (described in 7.2) is not required to interpret numbers with a “0x” prefix or a “(hex)” suffix. The **h#** command (i.e., **h# ffe0**) may be used to specify hex numbers when using the command interpreter.
- All numbers on the stack are signed integers, 32 bits, unless stated otherwise.
- The word “address” refers to a *virtual address* that occupies one cell, unless otherwise specified. The initial value of **base** when a Forth or *FCode program* begins execution is not defined; consequently, the program should explicitly set the desired **base** value if needed for numeric input or output.
- The actual *alignment* value for a *var-aligned* address is implementation-specific, but shall be at least doublet (2-byte) aligned.
- Additional notation conventions are described in A.1.2 of annex A.

2.4.2 Data types

The following data types represent numerical values or bit patterns:

- **byte**. An 8-bit value.
- **cell**. A value consisting of at least 32-bits, capable of representing a *virtual address*. The actual cell size for a particular ISA shall be specified in the ISA supplement for that ISA.
- **doublet**. A 16-bit value.
- **quadlet**. A 32-bit value.

2.4.3 ANS Forth compatibility

This Open Firmware standard follows the conventions and guidelines laid out in ANS Forth, with the following clarifications (items marked with an asterisk [*] are implementation-specific):

display after return of ACCEPT and EXPECT :	advance to beginning of next line
aligned address requirements:	*
behavior of EMIT for non-graphic values:	*
case sensitivity:	case-insensitive
character-aligned address requirements:	none
character set, character editing of ACCEPT and EXPECT :	see 7.2.1
character set:	ISO 8859-1 : 1987
control-flow stack:	*
console input and output device:	*
exception abort sequence:	*
input line terminator:	*
display after input terminates in ACCEPT and EXPECT :	advance to beginning of next line
maximum string length for ENVIRONMENT? , in characters:	31
method of selection of console input and output device:	see 4.2.5
methods of dictionary compilation:	*
methods of memory space management:	*
minimum search order:	*
number of bits in one address unit:	8 bits
ranges for n, +n, u:	32 bits (two's complement)
ranges for d +d ud:	64 bits (two's complement)
size of one cell in address units:	4
size of one character in address units:	1
size of the console input device's text-input buffer:	128
size of the pictured-numeric-output-string buffer:	66
size of the scratch area whose address is returned by PAD :	N/A
list of non-standard words using PAD :	N/A
treatment of control characters for space-delimited parsing:	control characters are always treated as spaces when parsing with space as the delimiter
system prompt:	ok
type of division:	floored
values returned after arithmetic overflow:	2's complement wraparound for addition/subtraction, unspecified for multiplication/division
maximum size of a parsed string:	255
size of buffer at WORD :	80
values of STATE when true:	-1

whether the current definition can be found after **DOES>**: *
source and format of display by **SEE**: *

Number conversion is case-insensitive; e.g., “a” and “A” are equivalent within hexadecimal numbers.

The *data stack* and return stack shall each be at least 64 entries deep.

2.5 Hardware assumptions

- **Processor.** Open Firmware requires at least one instruction-set processor for execution of the functions described in this document. While this model can apply to systems with more than one processor, the Open Firmware execution model assumes a single thread of execution on one processor. The processor may have associated *memory management units (MMUs)* and caches.
- **Memory.** Open Firmware requires some amount of random-access memory (RAM) for storage of data and program extensions. The firmware program itself is stored in, and executes from, either ROM, RAM, or some arbitrary combination of the two.
- **Console devices.** The optional Open Firmware interactive *command interpreter* requires a character input device and a character output device for use as the console.
- **Timer device.** Certain Open Firmware functions benefit from the presence of an optional timer device capable of interrupting the processor at periodic intervals. A timer resolution of at least 1 ms is preferred, but a resolution as coarse as 100 ms is still useful.
- **Nonvolatile memory.** Certain optional Open Firmware features require some amount (typically between 1 and 8 Kbytes) of nonvolatile memory. Examples of nonvolatile storage devices are battery-backed memory, electrically erasable read-only memory (EEPROM), and reserved space on disk drives.
- **Boot device.** The booting process requires a device capable of supplying the *client program* to Open Firmware. Examples of boot devices are disk drives, ROM, network interfaces, and serial communication links.
- **Other devices.** In addition to the devices listed above, Open Firmware may support other types of devices, but they are not generally required for basic operation.
- **Built-in vs. plug-in devices.** The devices that are used by Open Firmware may, in most cases, be either *built-in devices* or *plug-in devices*. The *firmware device drivers* for built-in devices are usually included as permanent parts of the system’s Open Firmware implementation. The drivers for plug-in devices are typically stored on the device itself, and thus are automatically installed and removed when the device is installed and removed.

3. Internal structure

The functions provided through the various external interfaces are based on, and described in terms of, the following elements:

- **Forth programming language.** Basic software execution model.
- **Forth dictionary.** Globally available functions.
- **Device tree.** System hardware description and drivers.
- **Configuration variables.** Storage for the user's configuration choices.

3.1 Forth dictionary

The Forth dictionary is a set of *Forth words* (software procedures in the Forth programming language). Most of the functions available through the user and *device interfaces* correspond directly to words in the Forth dictionary. Other words in the dictionary may serve as internal support functions to assist in the implementation of the specified interface functions.

Some or all of the words may have externally visible names, allowing them to be executed by name from any of the three external interfaces.

The Forth dictionary is extensible; a user or *device driver* may add new Forth words to the dictionary.

Generally, words within the Forth dictionary are globally available; they may be called directly, from any context, at any time. This is in contrast to *package methods* (described later), whose calling rules are more restricted.

3.2 Device tree

The *device tree* is a hierarchical data structure that describes the system hardware, describes user configuration choices, contains *firmware device drivers* for hardware devices, and contains support routines for use by those drivers.

The device tree's structure mimics the organization of the system hardware, viewed as a hierarchy of interconnected buses and their attached devices.

The device tree consists of a set of *device nodes* that are interconnected to form a tree. An individual device node represents either a hardware bus, a hardware device, or a set of interrelated software procedures.

3.2.1 Device nodes

The root of the *device tree* is a node representing the machine's main physical address bus.

Each *device node* may have *children* (other device nodes directly subordinate to it), *properties* (externally visible data structures describing the node and its associated device), *methods* (software procedures that may be used to access the device), and *data* (initial values of *private data* used by the methods).

Device nodes with children are called hierarchical nodes. A node's *parent* is the node to which it is attached in the device tree. The *root node* has no parent. Device nodes without children are called *leaf nodes*.

A node with children usually represents a bus and its associated controlling hardware. Each bus is assumed to define a physical address space; each device connected to that bus has a distinct physical address within that space, uniquely distinguishing the particular device from other devices on that bus. The form of a physical address is bus-specific. The children of a bus node are distinguished from one another with software representations of the same

physical addresses that the bus device uses to distinguish attached devices. Open Firmware uses several different representations of addresses with similar meanings but different forms:

- **text representation.** The human-readable form of a physical address. The format is bus-dependent. For example, some buses use a comma-separated list of numbers represented as ASCII text in hexadecimal notation.
- **stack representation.** Used to pass arguments to and results from *Forth words*. This form usually consists of one or more binary numbers on the *data stack*.
- **property-encoded representation.** Used to communicate with *client programs* through *property* values. This form usually consists of a sequence of binary numbers stored within an array of bytes.

The forms of these representations differ, but their meanings are the same: they represent *physical addresses* within a bus's *physical address space*.

The details of these different representations differ from bus to bus, depending on the addressing characteristics of the individual bus. Specifications of Open Firmware address representations for several standard buses are specified in supplements to this document (see the IEEE P1275.x documents in 2.1).

The device tree initially contains nodes for a computer system's *built-in devices*. Additional nodes for *plug-in devices* are added later by the probing process.

Some nodes in the device tree do not represent physical devices. These system nodes are used instead for various general *firmware* purposes. System nodes do not have physical addresses. Their *node names* have a *driver name* field but not a *unit address* field.

3.2.1.1 Node names

Each node in the *device tree* is identified by a *node name* using the following notation:

driver-name@unit-address:device-arguments

The *driver name* field is a sequence of between one and 31 letters, digits, and punctuation characters from the set “, . _ + -”. Uppercase and lowercase characters are distinct. By convention, this name includes the name of the device's manufacturer and the device's model name separated by a “, ”. (See the definition of “**name**” in annex A.) Inclusion of the manufacturer name within driver name is especially important for devices intended to plug into standard buses, as this minimizes the risk of accidental name collisions. It is somewhat less important for devices that are permanently attached to a particular system.

If the manufacturer name component is omitted (i.e., there is no “, ” within the driver name), the convention is to assume that the manufacturer name is the same as that of the nearest ancestor node (*parent node*, or grandparent node, etc.) that has an explicit manufacturer name component.

The *unit address* field is the text representation of the *physical address* of the device within the address space defined by its parent node. The form of the text representation is bus-dependent.

The *device arguments* field is a sequence of printable characters other than “/”, “:”, and “@”. Uppercase and lowercase characters are distinct. The length is arbitrary. The device arguments field is interpreted by the driver and typically represents additional device information, such as partition name or protocol. The device arguments field and its preceding “:” may be omitted when specifying a node name, as it does not serve to identify the *device node*; instead, it is passed to that node's **open** method if that driver is opened. By convention, “, ” is used to separate subfields within the device arguments field.

3.2.1.2 Path names

A particular node is uniquely identified by describing its position in the *device tree* by completely specifying the path from the *root node* through all intermediate nodes to the node in question. The textual representation of a such a path is called a *device path*. Device paths are composed as follows:

/node-name0/node-name1/ . . . /node-nameN

When Open Firmware is searching for a particular node, and either the *driver name* or *@unit-address* portion of the *node name* is not given, Open Firmware shall arbitrarily choose a node matching the portion that is present.

The complete, unambiguous device path of a *device node* lists the node names of all devices in the path from the root of the tree to the desired device. A device path is represented as a list of node names separated by “/”, e.g., “/sbus@1,f8000000/SUNW,esp@0,800000/sd@1,0”.

The root of the tree is the *root node*, which is not named explicitly but is indicated by a leading “/”. The root node itself is named by the pathname “/”.

The *user interface* and the *client interface* both use pathnames to identify particular device nodes.

3.2.1.3 Aliases

A *device alias*, or simply *alias*, is a shorthand representation of a *device path*. For example, the alias **disk** may represent the device path “/sbus@1,f8000000/esp@0,400000/sd@3,0:b”. An alias represents an entire device path, but that path need not refer to a *leaf node*. Each implementation may have a number of predefined aliases for devices commonly installed on that machine. Users may create, modify, and examine aliases with the **devalias** command. User-defined aliases are lost after a system reset or power cycle, but the effect of a persistent alias may be achieved by storing the **devalias** command in the *script*, either manually or with the **nvalias** command. The term *device-specifier* denotes a string that is either a device path, an alias, or an alias that refers to a non-*leaf node* followed by additional *node-name* components.

An *alias* name is a sequence of printable characters other than “/”, “\”, “:”, “[”, “]”, and “@”. An alias value is a device path.

3.2.2 Packages

A *package* is the combination of a device node’s properties, *methods*, and *private data*. In most cases, the terms package and *device node* may be used interchangeably. The term device node is typically used when the emphasis is on the node as a part of the *device tree*; the term package is used with emphasis on the use of the node’s driver methods. The text string that identifies a device node is called a *device path*. The numerical identifier of a device node is called a *phandle*.

The “type” of a package is the list and interface description of its externally visible methods together with its properties. Several distinct packages may implement the same interface. For example, there may be two display *device driver* packages, each implementing the standard display *device interface*, but for different types of display hardware. From a usage standpoint, the two packages are equivalent, even though their internal implementations may differ widely.

3.2.2.1 Properties

Properties describe characteristics of hardware devices, software, and user choices. Properties are externally visible; both *firmware* procedures and *client programs* may inspect and perhaps modify properties.

Each property consists of a *property name* and its associated *property value*.

3.2.2.1.1 Property names

The *property name* is a human-readable text string consisting of one to thirty-one printable characters. Property names shall not contain uppercase characters or the characters “/”, “\”, “:”, “[”, “]” and “@”.

Properties are accessed by name. Given a string, it is possible to determine whether or not there is a property with that name in a particular *device node*, and if so, its value.

This standard defines some property names and the meanings of their values; properties with names that are not defined by this standard may be used to convey other information at the developer’s discretion.

Property names beginning with the character “+” are reserved for use by future revisions of this standard.

Each device node has at least one property, whose property name is the string “**name**”, and whose value is a text string naming the device. The value of this “**name**” property is the *driver name* component of the *node name* that identifies the device in device pathnames.

3.2.2.1.2 Property values

The *property value* is an array of zero or more bytes containing the information associated with that *property*. The meaning of those bytes depends on the particular property.

This standard defines standard ways to encode various types of information in property value byte arrays, and procedures for performing the encoding and decoding process. The encoding technique is called *property encoding*. Property encoding has provisions for encoding text strings, integers, byte arrays, and various derived types that are composed by concatenation of those primary data types.

The property-encoding format is independent of hardware byte order and *alignment* characteristics. The encoded byte order is well-defined (in particular, it is *big endian*). Individual items are concatenated to form composed types without any “padding” for alignment reasons.

A property may have a null value, i.e., a byte array of length zero. Such properties usually convey true/false information. The presence of the property signifies *true*, and its absence signifies *false*.

The property encoding rules are as follows:

- **byte array.** An array of bytes is encoded in a property value byte array by storing the successive bytes of the input array into successive locations in the property value byte array.
- **32-bit integer.** A 32-bit integer is encoded into a property value byte array by storing the most significant byte at the next available address, followed (at address+1) by the high middle byte, the low middle byte, and (at address+3) the least significant byte.
- **text string.** A text string consisting of *n* printable characters is encoded into a property value byte array by storing the *n* characters at successive locations in the array, followed by a null byte (of binary value zero) denoting the end of the string.
- **composite values.** A succession of individual data items is encoded into a property value byte array by encoding the individual items in sequence. No alignment or padding is performed (i.e., the items are “packed”); each successive item immediately follows its predecessor.

This standard specifies functions for performing these and other derived encodings and for performing the inverse decoding operations. The program that creates a property and the program that uses it must agree on what information is contained in the value of a property of a particular name and on the order in which that information appears. The property encoding and decoding functions do not describe the meaning of the information; they simply allow it to be expressed in an ISA-independent manner.

3.2.2.2 Methods

Methods are named software procedures that control hardware devices or provide other services. Each *device node* may have zero or more methods.

The name space for a given device node's methods is distinct from the name space of its *properties*.

Some device node methods are externally visible in that they may be invoked by software entities outside the device node. Other methods are for internal use only and may be invoked only by other methods of the same device node.

The list of methods for a particular device node forms a Forth **wordlist**. Each method is a *Forth word*. Arguments to and results from device methods are passed on the Forth stack. The method lists for different device nodes are disjoint from one another. Although different device nodes often contain methods with identical names (for example, many nodes contain a **selftest** method), those individual methods are distinct.

In general, before a *package*'s methods can be used, the package must be *opened*, thus creating an *instance* of the package. However, it is possible to define methods that can be used without opening their packages; such methods are called *static methods*. A static method can not refer to any instance-specific data, nor can it execute any function that implicitly refers to the *current instance*. A static method can call another static method, and it can call any method in a package that has been opened.

This standard defines several methods with predefined meanings as given in this clause.

3.2.2.3 Private data

Private data is used by *package methods* to maintain internal information. Unlike *properties*, which can be accessed by external software, private data can not be directly accessed from outside the package; rather, any access to package data is mediated by the package's methods. Private data can be instance-specific or static.

3.2.2.3.1 Instance-specific data

When a *package* is *opened*, memory is allocated for its instance-specific data. The contents of that memory can be changed without affecting any other instances of that package. Instance-specific data can be either initialized or zero-filled.

A package contains initial values for its instance-specific data. When a package is opened, these initial values are copied into the corresponding locations within the memory allocated for that instance. Subsequent modifications of the instance's initialized data do not affect the initial values contained within the package.

The *Forth words* **variable**, **value**, and **defer**, in conjunction with **instance**, and their *FCode* equivalents, create initialized data items.

When a package is opened, the locations within the memory allocated for that instance corresponding to the zero-filled data are set to zero.

The Forth word **buffer:**, in conjunction with **instance**, and their *FCode* equivalents, creates zero-filled data items.

3.2.2.3.2 Static data

Static data is shared among all instances of a particular *package* and persists even when no instance exists. Static data may thus be used to communicate between different instances of the same package. This includes two instances that exist simultaneously as well as two instances that exist at different times.

3.2.2.4 Package creation

Each *package* is a *device node* in the *device tree*. Packages implementing *device drivers* are located in the device tree as children of the device node for the bus on which the device resides. Packages implementing utility functions not associated with any specific device are usually located as children of the **/packages** device node, which exists as a place to put those utility packages.

New packages are created during the probing portion of the system start-up sequence. The process of creating a new package involves the following:

- a) Creating a new “empty” device node at the appropriate location in the device tree and making it the *active package*.
- b) Interpreting an *FCode program* to define its *methods* and *properties* and allocating storage for its data.
- c) Finishing the package to “freeze” the initial values of its instance-specific data.

At any given time, if there is an active package,

- Newly created *Forth words* become methods of the active package.
- Newly created *Forth variables*, values, buffers and **defer** words may define either static or instance-specific data areas. The default is static. Instance-specific data is allocated if the defining word is preceded by the **instance** modifier.
- If there is a *current instance*, newly created properties are added to the package from which that instance was created; otherwise, newly created properties are added to the active package.
- The process of searching for *Forth words* considers first the methods of the active package, followed by globally visible words.

If there is not an active package:

- Newly created *Forth words* are globally visible (i.e., not specific to any package).
- Newly created *Forth variables*, values, buffers, and **defer** words allocate global storage (not associated with any particular package).
- If there is a current instance, newly created properties are added to the package from which that instance was created; otherwise, new properties cannot be created.
- The process of searching for *Forth words* considers only globally visible words.

There is always an active package when an *FCode program* is being evaluated. When a new package is created by probing an *FCode program*, the new package is placed in the device tree as a child of the hierarchical device that performs the probing. (In general, hierarchical devices are responsible for knowing how to probe their children.)

3.2.3 Package instances

In most cases, in order to use a *package*, the package must be *opened*, thus creating an *instance* of that package. A *package instance* consists of a copy of the package’s *private data*, copies of any arguments that were provided when the instance was created, and linkage information that allows the package instance to locate an instance of its parent device’s package. The numerical identifier of an instance is called an *ihandle*. In general, when a package is opened, all of the packages in the path from the *device tree* to that package are opened also.

Several instances of the same package may be in use simultaneously, each with its own separate copy of the package’s private data and instance arguments.

By analogy to traditional operating systems, a package is similar to a program residing in a disk file, and an instance is similar to a running “process” or “task”. Similar to the way that multiple independent processes may be created from a single program, multiple independent instances may be created from a single package.

3.2.3.1 Instance chains

A *package* is *opened* from a *device path* (or equivalent *device alias*) that specifies the path from the root of the *device tree* to the device in question. Each component of that path, beginning at the root, is opened, creating an *instance* for each device in the path. The instances are linked together so that each *package instance* may locate the instance of its parent.

When an instance is no longer needed (i.e., the program that opened a package decides that it will no longer need the function provided by that package), the instance may be *closed*, thus freeing the memory used by its *private data* and perhaps deactivating devices controlled by that package or its parents. An instance may be closed individually or the entire instance chain to which it belongs may be closed. In the latter case, the instances in the chain are closed in the order opposite to the order in which they were opened. In other words, an instance is closed, and then its parent instance is closed, and then its parent's parent, and so on.

3.2.3.2 Instance methods

Before using a *package method*, in most cases the package must first be *opened*, thus creating an *instance*. The instance is represented by an *ihandle*, a numerical value that refers to a particular instance. At any given time, one instance, at most, may be the *current instance*. The *private data* of the current instance is accessible; the private data of other instances is not.

In order to use a package method, its instance must be made current.

To call a method from another method within the same package, no special action need be taken; since the appropriate instance must already be the current instance for the calling method to run, the called method may be invoked directly, without changing the current instance.

To call a package method from the “outside” (i.e., from some execution context other than the instance containing the method in question), a different approach is used. The caller identifies both the method to be called and the instance that is to be current for the duration of that method's execution. There are various commands for calling methods, differing in the forms of method and instance identification. In general, such a command saves the *ihandle* of the current instance by pushing it on the return stack, makes the called instance the current instance, executes the method, and then pops the saved *ihandle* from the return stack, restoring the current instance to that value.

The caller must know the *ihandle* of the instance corresponding to the method to be called. There are two common ways to acquire this knowledge:

- The caller may have opened the package whose method is to be called. The process of opening a package returns the resulting *ihandle*, which the caller may save for later use in calling the *ihandle*'s associated methods.
- The package to be called may be the parent of the calling instance. As mentioned in 3.2.3.1, each instance automatically knows the *ihandle* of its parent.

A *static method* can be executed at any time, regardless of whether or not its package is open. It can be executed directly by other methods within the same package, or its execution token may be passed to **execute** or **catch** or stored in a **defer** word for later execution. There are various ways to determine its execution token in various circumstances, for example, **find-method** or **[']**.

3.3 Configuration memory

Configuration memory stores information that affects the behavior of various Open Firmware functions according to the user's preferences. The choices are stored in some form of nonvolatile memory, such as electrically erasable PROM or battery-backed RAM.

Open Firmware provides functions for setting the configuration memory through the *user interface* and the *client interface*.

The precise layout of the data stored in configuration memory is not exposed through any of the Open Firmware external interfaces. Configuration memory fields are accessed by name, as with *device-node properties*. This facilitates the addition of new fields and the deletion of fields that are no longer needed, since external software is unaware of precise storage locations and internal storage formats.

3.3.1 Configuration variables

Configuration variables are predefined configuration memory choices that affect Open Firmware in well-defined ways.

The list of configuration variables varies from system to system. Such choices often include the device from which to boot the operating system, the device to use as the console, and the amount of memory to be tested.

For each configuration variable supported by a particular implementation, Open Firmware maintains both the default value of the variable, typically stored in ROM, and the current value, stored in configuration memory. The default values are useful for restoring the settings to their factory defaults, which may be done either upon user command or automatically, if the *firmware* determines that the contents of configuration memory have been corrupted.

3.3.2 Custom start-up script

The custom start-up *script* is a portion of configuration memory that can contain an arbitrary user-supplied Forth language program. That program can be executed automatically as part of the Open Firmware start-up sequence. It can be used for a variety of purposes, including work-arounds or patches for *firmware* or *device driver* bugs, user enhancements, or customizations beyond the capabilities of the *configuration variables*.

The custom start-up script occupies the portion of configuration memory that is not dedicated to other purposes such as configuration variables. Its length can vary from zero up to the amount of available configuration memory. The custom start-up script can execute essentially any of the commands that are present in the *user interface*.

The contents of the custom start-up *script* can be set either through the *client interface* or with the text editor that is an optional part of the user interface.

3.4 Standard property names

These standard *property names* apply to all *device nodes*, regardless of type; particular types of devices have additional *properties* specified in subsequent sections. Most of the following properties are optional; a *package* includes the property to declare a particular characteristic if desired. The “**name**” property is required for all packages.

The *property value* data formats for these properties are described in their glossary entries in annex A.

“name”	Standard <i>property name</i> to define the name of the package.
“reg”	Standard <i>property name</i> to define the package’s registers.
“device_type”	Standard <i>property name</i> to specify the implemented interface.
“interrupts”	Standard <i>property name</i> to define the interrupts used.
“model”	Standard <i>property name</i> to define a manufacturer’s model number.
“address”	Standard <i>property name</i> to define large virtual address region(s).
“compatible”	Standard <i>property name</i> to define alternate “name” property values.
“status”	Standard <i>property name</i> to indicate the device’s operational status.

3.5 Standard system nodes

Standard system nodes are as follows:

/

This is the *root node*. It is the root of the *device tree*; all other nodes descend from it. Its lists describe basic machine *properties* such as model, revision, and manufacturing date. The *physical address space* defined by this node is the main physical address bus of the system. The methods of this node provide mapping services for the main system bus. On a uniprocessor machine this node may also contain properties describing the CPU. On a multiprocessor machine, each CPU would have its own node below the root node.

Property name	Encoding	Value
name	string	Name of system's manufacturer and model number, e.g., "ABC,mat750".

/aliases

The property list of this node is the devalias list. For each property in this node, the *property name* is the name of an alias, and the property value is the alias's expansion, encoded as with "**encode-string**".

Property name	Encoding	Value
name	string	"aliases"

/openprom

Describes the Open Firmware.

The standard properties of this node are as follows. Additional system-dependent properties may also be present.

Property name	Encoding	Value
name	string	"openprom"
model	string	Describes manufacturer and revision level of firmware. See model property.
relative-addressing	(none)	Presence of this property indicates that the Open Firmware supports bus-relative physical addressing (early precursors to Open Firmware used a different addressing scheme).

/options

The properties of this node are the system's *configuration variables*. The property names are the names of those configuration variables, and the *property values* are the output text representations (see 7.4.4.1) of those configuration variables. Client programs may examine and change the values of these properties with **getprop** and **setprop**, thus examining and changing the values of the corresponding configuration variables. Similarly, users may examine and change them with **printenv**, **setenv**, and **\$setenv**.

Property name	Encoding	Value
name	string	"options"

/chosen

Has properties describing parameters chosen or specified at runtime.

Property name	Encoding	Value
name	string	“chosen”
stdin	int	<i>lhandle</i> for the console input device. May be used with the <i>read client interface</i> function to read characters from that device.
stdout	int	<i>lhandle</i> for the console output device. May be used with the <i>write client interface</i> function to write characters to that device.
bootpath	string	The <i>device path</i> for the last boot device. Client programs may use this property to locate the device they were booted from. This property may be modified to select a different boot device.
bootargs	string	The arguments to the last boot command. This property may be modified in order to alter the options to the boot command, which options may be interpreted by the Open Firmware or by the client program(s).
memory	int	<i>lhandle</i> for the package that describes the allocation status of physical memory. See 3.7.6.
mmu	int	<i>lhandle</i> for the package, if any, that the firmware is currently using for memory management. This property shall not be present if there is no such package. See 3.6.5.

/packages

This node may have several children, but instead of describing a physical bus, this node serves as a parent node for *support package* nodes (both standard support packages and system-specific ones). The children of this node are general-purpose support packages not attached to any particular hardware device. The physical address space defined by this hierarchical node is the trivial one: all addresses are the same (0,0). Its children are distinguished by name alone.

Property name	Encoding	Value
name	string	“packages”

3.6 Standard packages

A standard *package* implements a defined set of *properties* and *methods*, thus providing a service that may be used by other *firmware* components. There are three kinds of standard packages: standard system *nodes*, *devices types*, and standard *support package*.

- **Standard system nodes.** The purpose of a standard system node is to contain system-level information. Most standard system nodes contain only properties, with no executable methods. The **/packages** standard system node contains the standard support packages (described later).
- **Device type packages.** The purpose of a typical package is to provide a *device driver* for a particular hardware device. A standard device type is a specification for the interface presented by packages of that type, so that other firmware components defined by this standard can use such packages in a predictable and useful fashion. Typically, some such packages are preconfigured into a boot firmware system by the manufacturer, and others are added later by evaluating *FCode programs*. The preconfigured packages correspond to *built-in devices*, and those that are added later correspond to *plug-in devices*. The particular set of properties and methods depends on the package’s device type. This standard defines several device types, their corresponding sets of properties and methods, and the ways in which packages implementing those devices types are used by other Open Firmware components. A particular firmware system may have several distinct packages implementing a particular device type. For example, a system might have several different kinds of disk controllers, each with a package implementing the disk device type.
- **Standard support packages.** The purpose of a standard support package is to assist in the implementation of packages of standard device types. Standard support packages are preconfigured into the boot firmware system residing (with the exception of the display driver support packages) as children of the **/packages** node. A standard support package is somewhat analogous to a “subroutine library.” This standard defines several standard support packages, each with its own defined set of properties and methods. Typically, a particular system

has exactly one of each such package (since support packages are located by name in the “flat” name space of the **/packages** node, there is no way to find or use more than one of each).

This clause lists several standard device types and several standard support packages and describes how they are used. The definitive specification of those device types and standard support packages is given in the glossary in annex A. This clause also describes a template for a set of methods that are commonly used by device types corresponding to particular buses. The individual methods in that set are defined by this standard (in annex I), but the definition of particular standard device types (groupings of methods) that include those methods is left to other related standards, such as bus-specific supplements.

3.6.1 Parent methods

When a *package* is *opened* with **open-dev**, all of the packages in the path from the root of the *device tree* to that package are opened also, resulting in an *instance chain*. Similarly, when that instance chain is *closed*, all the packages that it represents are closed. This implies that any *device node* with children that can be opened with **open-dev** must itself have **open** and **close** methods; otherwise, it would not be possible to open those children.

This standard explicitly defines one such node, the *root node*. Typical systems have additional such nodes, usually corresponding to standard expansion buses. The details of those nodes are the subject of related standards, or may be vendor-dependent for proprietary buses, but those nodes must have **open** and **close** methods.

The children of the **/packages** node are *support packages*, which are opened with **open-package** instead of **open-dev**. Since **open-package** does not open the entire path, the **/packages** node need not have **open** and **close** methods.

In many cases, these **open** and **close** methods can be very simple; often just returning **true** with no other action for **open** and doing nothing for **close**. Such simple implementations are appropriate for “hardwired” bus hardware where the bus adapter hardware has little or no software-visible “state”.

A very similar requirement is imposed by the semantics of pathname resolution. In order to match the *unit address* component of a *node name*, the parent’s **decode-unit** method is executed, transforming the text representation of the unit address into its numerical form. This implies the need for a **decode-unit** method in any device node that defines a *physical address space* for its children. In most systems, the nodes to which this requirement applies are the same as the ones to which the **open** and **close** requirement applies. However, it is conceivable that a system might have a node with a fixed set of children that are distinguished by name only without needing any form of physical address space. Such a node would not need a **decode-unit** method.

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
decode-unit	(addr len -- phys.lo ... phys.hi)	Convert text unit-string to physical address.
encode-unit	(phys.lo ... phys.hi -- unit-str unit-len)	Convert physical address to text unit-string.

3.6.2 Generic methods

Any *package*, regardless of its *device type*, can implement a **selftest** method so that the *user interface test* command can be used to cause its corresponding device to be tested. This standard imposes no requirement on the existence of a **selftest** method in any package, but customer support considerations often demand it.

selftest	(-- ?error?)	Perform self-test for this device.
-----------------	----------------	------------------------------------

The **reset** method can be implemented to put the device in a quiescent state. The **reset** method is not invoked by any standard Open Firmware functions, but may be explicitly executed for particular “problem” devices in particular Open Firmware implementations.

reset	(--)	Put this device into a quiescent state.
--------------	--------	---

3.6.3 Package I/O model

Most input, output, and storage devices are accessed with a byte-oriented “read/write/seek” interface. I/O operations are specified by a starting *virtual address* and a byte count, and the implementation “hides” device-specific details like block sizes and records. A particular device only implements the set of operations that make sense for it; for example, the **read method** might not be applicable to a *frame-buffer device*, and the **write method** might not be applicable to a CD-ROM.

The *client interface* entries **read**, **write**, and **seek**, described in clause 6, are essentially direct interfaces to similarly named methods of standard *packages*.

3.6.4 Expansion bus device class template

A memory-mapped bus logically extends the processor’s memory address space to include the *devices* on that bus, allowing the use of processor load and store cycles to directly address those devices. The details vary from bus to bus. This standard does not specify the adaptation of Open Firmware to any particular bus, but other related standards do so specify (see 2.1). This subclause lists a set of *methods* that deal with requirements common to most memory-mapped buses. This subclause is intended as a suggested starting point for the development of complete sets of methods for particular buses; also see related standards, such as IEEE Std 1275.2-1994. The methods provide mapping services for establishing the correspondence between processor virtual and device *physical addresses*, allocation of DMA memory, and probing to locate *plug-in devices*.

map-in	(phys.lo ... phys.hi size -- virt)	Map the specified region; return a virtual address.
map-out	(virt size --)	Destroy mapping from previous map-in .
dma-alloc	(... size -- virt)	Allocate a memory region for later use.
dma-free	(virt size --)	Free memory allocated with dma-alloc .
dma-map-in	(... virt size cacheable? -- devaddr)	Convert virtual address to device bus DMA address.
dma-map-out	(virt devaddr size --)	Free DMA mapping set up with dma-map-in .
dma-sync	(virt devaddr size --)	Synchronize (flush) DMA memory caches.
probe-self	(arg-str arg-len reg-str reg-len fcode-str fcode-len --)	Evaluate FCode as a child of this node.

The following properties are specific to this class of *device node*:

“ranges”	Standard <i>property name</i> to define a device’s physical address.
“#address-cells”	Standard <i>property name</i> to define the package’s address format.
“#size-cells”	Standard <i>property name</i> to define the package’s address size format.

3.6.5 Memory management device class template

An MMU is a *device* that performs address translation between a CPU’s *virtual addresses* and the *physical addresses* of some bus, typically the bus represented by the *root node*. In general, the details are both processor-specific and bus-specific. This standard does not specify the adaptation of Open Firmware to any particular MMU, but other related standards may so specify (see 2.1). This standard does not require the presence of an MMU.

This subclause lists a set of *methods* that deal with requirements common to most MMUs. This subclause is intended as a suggested starting point for the development of complete sets of methods for particular MMUs. The methods provide services for fine-grained control of the allocation and mapping of virtual addresses, particularly intended for use by *client programs* through the **call-method client interface** service (see also the “**mmu**” *property* of the **/chosen** node). In general, the use of these methods makes a client program system-specific; nevertheless, they are useful in some circumstances. The arguments and results shown are intended as guidelines; particular MMUs might require additional arguments or changes to the arguments shown.

The presence of an MMU node does not imply that the Open Firmware is necessarily using virtual-to-physical address translation hardware.

The following methods, defined in the glossary, are recommended for MMU *packages*:

claim	([virt] size align -- base)	Allocate (claim) addressable resource
release	(virt size --)	Free (release) addressable resource.
map	(phys.lo ... phys.hi virt size mode --)	Create address translation
unmap	(virt size --)	Invalidate existing address translation.
modify	(virt size mode --)	Modify existing address translation.
translate	(virt -- false phys.lo ... phys.hi mode true)	Translate virtual address to physical address.

Additional requirements for the **claim** and **release** methods:

- The address format, *virt*, is a single-cell *virtual address*.
- The allocation length, *size*, is a single cell.
- The allocated resource is a region of virtual address space.

The following properties are recommended for MMU packages:

“available”	The property values are as defined for the standard “ reg ” format, with single-cell virtual addresses. The regions of virtual address space denote the virtual address space that is currently unallocated by the Open Firmware and is available for use by client programs.
“existing”	The value of this property defines the regions of virtual address space managed by the MMU in whose package this property is defined without regard to whether or not these regions are currently in use. The encodings of <i>virt</i> and <i>len</i> are MMU-specific.

NOTE—Freeing virtual address space does not necessarily free any associated physical resource. The correct sequence of operations for freeing mapped memory is to first use **unmap**, thus destroying the translation. Then the physical memory and virtual address space can be freed with the **release** methods of the respective nodes.

3.7 Standard device types

The *device type* of a particular *package* identifies the set of *properties* and *methods* that the package is expected to implement. Any particular package may also implement an arbitrary number of properties and methods in addition to those implied by its device type. The device type of a *node* is given by the *property value* (of type string) of its “**device_type**” property.

It is not necessary for every node to have a “**device_type**” property. If a particular *device* is not useful for any Open Firmware function (e.g., booting, console, probing) then it need not have a device type. For example, Open Firmware has no use for a FAX modem, so such a device does not need a device type. However, there is no restriction preventing it from having a device type so long as its device type is not the same as one of the standard types (i.e., a device should not claim to be something that it is not).

Open Firmware supplies a set of standard *support packages* that assist in the implementation of methods for standard device types in terms of “lower-level” methods that more closely match the native capabilities of some common types of hardware devices. For example, the “**deblocker**” support package implements a byte-oriented **read** method in terms of record-oriented operations on a tape device. A particular implementation of a standard package for a given device type is allowed, but not required, to use the provided support packages.

The following standard device types are defined by Open Firmware. Subsequent subclauses discuss these device types more extensively.

“ device_type ” value	Example	Typical use
“ display ”	bit-mapped frame-buffer	console output
“ block ”	hard disk	booting
“ byte ”	tape	booting
“ network ”	Ethernet interface	booting
“ serial ”	asynchronous serial line	console input and output

Related standards will define additional standard device types. For example, standards specifying the application of Open Firmware to particular expansion buses will define standard device types for those buses (see 2.1).

3.7.1 “display” devices

“**display**” *devices* are user output devices that can display text, perform cursor-positioning operations controlled by embedded ANSI X3.64 escape sequences, and possibly display bit-mapped logos. Examples of “**display**” devices include bit-mapped *frame buffers*, graphics displays, and character-mapped displays. Open Firmware typically uses display devices for console output.

There are several standard *support packages* to assist in the implementation of the standard “**display**” device methods. The *terminal emulator* support package processes a stream of text with embedded ANSI X3.64 escape sequences, converting it to a sequence of calls to primitive display operations such as **draw-character**. For certain types of bit-mapped frame buffers, the “**fb8**” support package implements those primitive operations. Finally, the “**font**” support package provides a default bit-mapped font suitable for use with the “**fb8**” support package.

The “**display**” glossary entry (annex A) specifies the set of methods for this *device type*. For reference, those methods are listed as follows:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
write	(addr len -- actual)	Write memory buffer to device, return actual byte count.
draw-logo	(line# addr width height --)	Calls draw-logo routine for this device.
restore	(--)	Restore device to useable state after unexpected reset.

The following *property* is specific to this device type:

character-set	Standard <i>property name</i> to specify the character set for this device.
----------------------	---

3.7.2 “block” devices

“**block**” *devices* are nonvolatile mass storage devices whose information can be accessed in any order. Examples of “**block**” devices include hard disks, floppy disks, and CD-ROMs. Open Firmware typically uses “**block**” devices for booting.

Although standard *packages* of the “**block**” *device type* present a byte-oriented interface to the rest of the system, the associated hardware devices are usually block-oriented; i.e., the device reads and writes data in “blocks” (groups of, for example, 512 or 2048 bytes). The standard “**deblocator**” *support package* assists in the presentation of a byte-oriented interface “on top of” an underlying block-oriented interface, implementing a layer of buffering that “hides” the underlying block length.

“**block**” devices are often subdivided into several logical “partitions”, as defined by a *disk label*—a special block, usually the first one—containing information about the device. The driver is responsible for appropriately interpreting a disk label. The driver may use the standard “**disk-label**” support package if it does not implement a specialized label. The “**disk-label**” support package interprets a system-dependent label format. Since the disk-booting protocol usually depends upon the label format, the standard “**disk-label**” support package also implements a **load method** for the corresponding boot protocol.

The “**block**” glossary entry (annex A) specifies the set of methods for this device type. For reference, those methods are listed as follows:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.

read	(addr len -- actual)	Read device into memory buffer, return actual byte count.
(continued)		
write	(addr len -- actual)	Write memory buffer to device, return actual byte count.
seek	(pos.lo pos.hi -- status)	Set device position for next read or write .
load	(addr -- size)	Load a client program from device to memory.

3.7.3 “byte” devices

“**byte**” devices are sequential-access mass storage devices, typically, tape devices. Open Firmware typically uses byte devices for booting.

Although standard *packages* of the “**byte**” device type present a byte-oriented interface to the rest of the system, the associated hardware devices are usually record-oriented; i.e., the device reads and writes data in “records” containing more than one byte. The records may be either fixed length (all records must be the same length) or variable length (the record length may vary from record to record). Tapes may be subdivided into several tape files delimited by file marks.

The standard “**debloader**” support package assists in the presentation of a byte-oriented interface “on top of” an underlying record-oriented interface, implementing a layer of buffering that “hides” the underlying record structure.

The “**byte**” glossary entry (annex A) specifies the set of *methods* for this device type. For reference, those methods are listed as follows:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
read	(addr len -- actual)	Read device into memory buffer; return actual byte count.
write	(addr len -- actual)	Write memory buffer to device; return actual byte count.
seek	(pos.lo pos.hi -- status)	Set device position for next read or write .
load	(addr -- size)	Load a client program from device to memory.

3.7.4 “network” devices

“**network**” devices are packet-oriented devices capable of sending and receiving packets (frames) that are addressed according to Local Area Network (LAN) specifications for Media Access Control (MAC) addresses administered by the IEEE Registration Authority.⁸ Open Firmware typically uses “**network**” devices for booting.

The standard “**obp-tftp**” support package assists in the implementation of the “**load**” method for this device type.

The “**network**” glossary entry (annex A) specifies the set of *methods* for this device type. For reference, those methods are listed as follows:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
read	(addr len -- actual)	Read device into memory buffer; return actual byte count.
write	(addr len -- actual)	Write memory buffer to device; return actual byte count.
load	(addr -- size)	Load a client program from device to memory.

The following *properties* are specific to this device type:

⁸ For information on the use of MAC addresses and Organizationally Unique Identifiers (OUI), see IEEE Std 802-1990. To apply for an OUI or MAC address, contact the Registration Authority, IEEE Standards Dept., 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

“local-mac-address”	Standard <i>property name</i> to specify preassigned network address.
“mac-address”	Standard <i>property name</i> to specify network address last used.
“address-bits”	Standard <i>property name</i> to indicate network address length.
“max-frame-size”	Standard <i>property name</i> to indicate maximum packet size.

3.7.5 “serial” devices

“serial” *devices* are byte-oriented sequentially accessed devices such as asynchronous communication lines (often attached to “dumb” terminals). Open Firmware typically uses **“serial”** devices for console input and output.

The **“serial”** glossary entry (annex A) specifies the set of *methods* for this *device type*. For reference, those methods are listed as follows:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
read	(addr len -- actual)	Read device into memory buffer; return actual byte count.
write	(addr len -- actual)	Write memory buffer to device; return actual byte count.
install-abort	(--)	Begin polling for a console abort sequence.
remove-abort	(--)	Cease polling for a console abort sequence.
restore	(--)	Restore device to useable state after unexpected reset.
ring-bell	(--)	Ring the bell.

3.7.6 Memory

In this context, *memory* refers to traditional RAM, suitable for temporary storage of data. Typically, the aggregate amount of main memory on a system is represented by a single *device node*. The *properties* of that node describe the regions of memory that exist and those that are currently available. The *methods* provide fine-grained control over the allocation of that memory. These allocation methods are intended for use by system-specific programs that need precise control over their use of physical memory. Portable programs must use other functions (e.g., **alloc-mem** and **free-mem**).

The **“memory”** glossary entry (annex A) specifies the set of methods for this *device type*. For reference, those methods are listed as follows:

claim	([phys.lo ... phys.hi] size ... align -- base.lo...base.hi)	Allocate (claim) addressable resource.
release	(phys.lo ... phys.hi size ... --)	Free (release) addressable resource.

The **“memory”** glossary entry (annex A) defines the following *properties*. For reference, those properties are listed as follows:

“reg”	Standard property defining the <i>physical addresses</i> installed in the system without regard to whether or not that memory is currently in use by the Open Firmware or <i>client program</i> .
“available”	Standard “reg” format property defining the regions of physical address space that are currently unallocated by the Open Firmware.

3.8 Standard support packages

Support packages are used by other *packages* to implement commonly used functions. With the exception of the support packages related to display *devices*, they are located in the **/packages** *device node*, *opened* with either **open-package** or **\$open-package**, and *closed* with **close-package**.

3.8.1 “disk-label” support package

The “**disk-label**” *package* interprets the *disk label*, interpreting any “partitioning” information contained therein. It is used by “**block**” *device drivers*.

This package uses the **read** and **seek** *methods* of its parent. It defines the following methods:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
load	(addr -- size)	Load a client program from device to memory.
offset	(d.rel -- d.abs)	Convert partition-relative disk position to absolute position.

3.8.2 “obp-tftp” support package

The “**obp-tftp**” *package* implements the Internet Trivial File Transfer Protocol (TFTP) for use in network booting. It is typically used by “**network**” *device drivers*.

This package uses the **read** and **write** *methods* of its parent, and defines the following methods:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
load	(addr -- size)	Load a client program from device to memory.

3.8.3 “debloader” support package

The “**debloader**” *package* assists in the implementation of byte-oriented **read** and **write** *methods* for block-oriented or record-oriented *devices* such as disks and tapes. It provides a layer of buffering to implement a high-level byte-oriented interface “on top of” a low-level block-oriented interface. The “**debloader**” *support package* defines the following methods:

open	(-- okay?)	Prepare this device for subsequent use.
close	(--)	Close this previously opened device.
read	(addr len -- actual)	Read device into memory buffer; return actual byte count.
write	(addr len -- actual)	Write memory buffer to device; return actual byte count.
seek	(pos.lo pos.hi -- status)	Set device position for next read or write .

Any package that uses the “**debloader**” support package must define the following methods, which the debloader uses as its low-level interface to the device.

block-size	(-- block-len)	Return “granularity” for accesses to this device.
max-transfer	(-- max-len)	Return size of largest possible transfer.
read-blocks	(addr block# #blocks -- #read)	Read <i>#blocks</i> , starting at <i>block#</i> , from device into memory.
write-blocks	(addr block# #blocks -- #written)	Write <i>#blocks</i> from memory into device, starting at <i>block#</i> .

3.8.4 Terminal emulator support package

The *terminal emulator support package*, if present, shall interpret control sequences as described in annex B.

3.8.4.1 Terminal emulator interface conventions

For historical reasons, the *terminal emulator support package*’s interface conventions differ from those used by other support packages.

A standard *package* of the “**display**” *device* type that uses the terminal emulator support package does not create **open** and **close** *methods* in the usual way. Instead, it executes **is-install** and **is-remove**, which themselves create suitable **open** and **close** *methods* that, when later executed, automatically install and initialize

the terminal emulator support package. The **selftest** method for such a package can either be created in the normal fashion or by executing **is-selftest**.

is-install	(xt --)	Create open and other methods for this display device.
is-remove	(xt --)	Create close method for this display device.
is-selftest	(xt --)	Create selftest method for this display device.

Instead of acquiring its low-level services with the usual technique of calling methods of its parent, the terminal emulator uses a set of interface **defer** words and **values**. This interface is called the **defer** words interface. A **defer** word is a *Forth* word (or an equivalent *FCode* function) which, when executed, has the effect of executing another *Forth* word. It is like a variable that executes its contents. The terminal emulator uses a group of **defer** words for invoking display device driver routines. When a display device driver is *opened*, it sets the values of those **defer** words so that they later execute the driver's device-specific routines for performing various display operations.

Open Firmware functions that write text to a display device use the normal package methods interface. The **defer** words interface is used for the internal communication between the terminal emulator software and the low-level display management routines.

3.8.4.2 Terminal emulator state variables

The following **value** words are used and/or set by the *terminal emulator*. The “**fb8**” generic *frame-buffer support package* uses them to determine where and how to display characters. Display device drivers may use them as needed. A display device driver routine is permitted to change their values temporarily, but the previous values must be restored before that routine exits.

line#, **column#**, **inverse?** and **inverse-screen** are set by the terminal emulator and are used by the driver or the “**fb8**” support package. **#lines** and **#columns** are set by the driver or **fb8-install** and are used by the terminal emulator and the “**fb8**” support package.

line#	(-- line#)	Return the current cursor line number.
column#	(-- column#)	Return the current cursor column number.
inverse?	(-- white-on-black?)	Indicates how to paint characters.
inverse-screen?	(-- black?)	Indicates how to paint the background.
#lines	(-- rows)	Return number of lines of text in text window.
#columns	(-- columns)	Return number of columns of text in text window.

3.8.4.3 Display device low-level interfaces

The *terminal emulator* uses the following **defer** words to access display device driver routines. When a display device driver is *opened*, it must set the values of these **defer** words so that they will execute the corresponding device-dependent routines defined by that device driver. Many display device drivers can use the “**fb8**” generic *frame-buffer support package* to do most of the work. In that case, the following **defer** words are set as a group by executing **fb8-install**, and then selected **defer** words, for which the generic implementations are incorrect for that device, are changed as needed.

The text cursor is considered to be between two adjacent characters. For displays that indicate the cursor position by highlighting a particular character, the “true” cursor position is just before the highlighted character.

draw-character	(char --)	Draw a character at the current cursor position.
reset-screen	(--)	Perform frame-buffer device initialization.
toggle-cursor	(--)	Toggle the state of the text cursor.
erase-screen	(--)	Clear the screen.
blink-screen	(--)	Flash the screen.
invert-screen	(--)	Exchange the foreground and background colors.

insert-characters	(<i>n</i> --)	Insert <i>n</i> spaces to the right of the cursor.
delete-characters	(<i>n</i> --)	Delete <i>n</i> characters to the right of the cursor.
insert-lines	(<i>n</i> --)	Insert <i>n</i> blank lines at and below the cursor line.
delete-lines	(<i>n</i> --)	Delete <i>n</i> lines at and below the cursor line.

is-install creates a **draw-logo** *method* in the current *package* that will execute the **draw-logo defer** word. The **draw-logo** *method* is not invoked by the terminal emulator, but by **banner**.

draw-logo (*line#* *addr* *width* *height* --) Draw (at *line#*) the logo stored at location *addr*.

NOTE—These **defer** words are intended to be called *only* by the terminal emulator. Essentially, this **defer** words interface is a private communication channel between the terminal emulator package and a display device driver that happens to use it. There is no requirement that a given display device driver must use the terminal emulator support package.

3.8.4.4 Frame-buffer support routines

The *frame-buffer* support routines make it easy to implement the low-level display interfaces for certain kinds of “dumb” bit-mapped frame buffers. There are two sets of frame-buffer support routines. One set is for 1-bit-per-pixel frame buffers, and the other for 8-bits-per-pixel frame buffers. The 8-bits-per-pixel routines can sometimes be coerced into service for deeper (e.g., 24-bits-per-pixel) frame buffers as well. It can be difficult to achieve acceptable text display performance with “dumb” frame buffers if the rendering routines are written directly in *FCode*, because each display operation typically changes hundreds of individual pixels. These support routines can alleviate that problem, since their underlying implementation can be in optimized machine code.

The following discussion applies to both the obsolete 1-bit frame-buffer support routines and the 8-bit frame-buffer support routines. When referring to the obsolete 1-bit frame-buffer routines substitute the number 1 for the number 8. The frame-buffer support routines are used as follows:

The display driver *package* determines the *virtual address* of the beginning of the frame buffer (typically, with a mapping operation), sets **frame-buffer-adr** to that address with the *FCode* equivalent of the phrase “to **frame-buffer-adr**”, sets up the font with **set-font**, and then executes **fb8-install** with appropriate arguments describing the width and height of the frame buffer. This establishes behaviors for the **defer** words that comprise the low-level display *device interface*. The display driver package then replaces the behaviors of any of those **defer** words for which it has a better or more appropriate implementation than the one supplied by the frame-buffer support package just installed and corrects the centering if necessary by changing the values of **window-left** and possibly **window-top**.

set-font establishes the font for use by the frame-buffer support routines. Its arguments are typically supplied by **default-font**, thus using the font provided by the system, but a display driver can supply its own font by calling **set-font** with arguments denoting that font. **>font** is used internally by the frame-buffer support packages. It can also be used by display drivers that perform their own rendering (not using the frame-buffer support packages) using the system default font.

default-font (-- *addr* *width* *height* *advance* *min-char* *#glyphs*)
Return the font parameters for the default system font.

set-font (*addr* *width* *height* *advance* *min-char* *#glyphs* --)
Set the current font as specified.

>font (*char* -- *addr*)
Return beginning address for *char* in the current font.

The following **value** words are used internally by both the 1-bit and the 8-bit frame-buffer support routines.

— **frame-buffer-adr** is set by the display driver prior to installing the support routines.

- **screen-height**, **screen-width**, **window-top**, and **window-left** are set by the execution of **fb8-install**. The display driver can change **window-top** and **window-left** afterwards if that is necessary to correct the centering of the display on the screen.
- **char-height**, **char-width**, and **fontbytes** are set by the execution of **set-font**.

Most display driver packages do not directly use **screen-height**, **screen-width**, **window-top**, **window-left**, **char-height**, **char-width**, **font**, or **fontbytes**. For those that do, the primary use is for drivers that install their own routines in place of one or more of the support package routines; such routines often need to know the display and font geometry, and the driver can avoid keeping a duplicate copy of the information by using these values.

frame-buffer-adr	(-- addr)	Return current frame-buffer virtual address.
screen-height	(-- height)	Return total <i>height</i> of the display in pixels.
screen-width	(-- width)	Return total <i>width</i> of the display in pixels.
window-top	(-- border-height)	Return window top border in pixels.
window-left	(-- border-width)	Return window left border in pixels.
char-height	(-- height)	Return the <i>height</i> of a font character in pixels.
char-width	(-- width)	Return the <i>width</i> of a font character in pixels.
fontbytes	(-- bytes)	Return interval between entries in the font table.

3.8.4.4.1 1-bit frame-buffer support routines

The “**fb1**” generic *frame-buffer support package* implements the display device low-level interfaces for frame buffers with 1 memory bit per pixel. The “**fb1**” generic frame-buffer support package is an obsolete feature that is implemented in many existing systems. It is described in annex H. A system may, but need not, implement this support package.

3.8.4.4.2 8-bit frame-buffer support routines

The “**fb8**” generic *frame-buffer support package* implements the display device low-level interfaces for frame-buffers with 8 memory bits per pixel. It assumes that successive memory bytes correspond to successive pixels, possibly with undisplayed bytes at the end of each scan line, and that bytes at lower addresses correspond to pixels to the left. In normal (not inverse) video mode, background pixels are drawn with the value 0x00, and foreground pixels are drawn with the value 0xFF.

Execution of **fb8-install** installs the other routines as the behaviors of the corresponding low-level display device interface **defer** words, and sets the values of **screen-height**, **screen-width**, **window-top**, **window-left**, **#lines**, and **#columns**.

fb8-install	(width height #columns #lines --)	Install all built-in generic 8-bit frame-buffer routines.
fb8-draw-character	(char --)	Implement the “ fb8 ” draw-character function.
fb8-reset-screen	(--)	Implement the “ fb8 ” reset-screen function.
fb8-toggle-cursor	(--)	Implement the “ fb8 ” toggle-cursor function.
fb8-erase-screen	(--)	Implement the “ fb8 ” erase-screen function.
fb8-blink-screen	(--)	Implement the “ fb8 ” blink-screen function.
fb8-invert-screen	(--)	Implement the “ fb8 ” invert-screen function.
fb8-insert-characters	(n --)	Implement the “ fb8 ” insert-characters function.
fb8-delete-characters	(n --)	Implement the “ fb8 ” delete-characters function.
fb8-insert-lines	(n --)	Implement the “ fb8 ” insert-lines function.
fb8-delete-lines	(n --)	Implement the “ fb8 ” delete-lines function.
fb8-draw-logo	(line# addr width height --)	Implement the “ fb8 ” draw-logo function.

4. Internal procedures

Open Firmware's primary task is to control the machine from the time power is applied until the primary operating system has been *loaded* and has taken control of the machine.

In typical operation, Open Firmware performs the following sequence of operations, in the order given:

- a) Initialize and test *built-in devices*.
- b) Locate, initialize, and test *plug-in devices*.
- c) Load and execute a *client program*.
- d) Provide services requested by the client program.

4.1 Forth language environment

The underlying execution model for Open Firmware is the execution model of the Forth programming language. The Forth execution model includes the following items:

- **Dictionary.** The list of *Forth words*.
- **Data space.** The memory used by Forth words.
- **Data stack.** The stack used for parameter passing.
- **Return stack.** The stack used for procedure nesting.
- **Input buffer.** The current line of textual input.
- **Input source.** The source device for textual input.
- **Output stream.** The destination of textual output.
- **Text interpreter.** Processes textual commands.

Open Firmware uses the dialect of Forth described in ANSI X3.215-1994. That document should be consulted for more information about the Forth programming language.

4.2 Start-up sequence

The typical start-up sequence is described in more detail below. Some portions of the start-up sequence may not be present in all implementations. For example, the “probing” process may not be relevant to systems without any provision for *plug-in devices*, and the ability to manually interrupt the start-up sequence is of dubious value if no *user interface* is present.

After the system power is turned on, or the system is otherwise restarted, but before Open Firmware gains control of the machine, other system-dependent *firmware* may be executed. Typically that other firmware tests some portion of the machine before passing control to Open Firmware. The details of such other firmware, the *methods* it uses to pass control to Open Firmware, and the “division of labor” between that other firmware and Open Firmware in the initialization of *built-in devices* and other *devices* that do not use the Open Firmware methods of auto-identification, are outside the scope of this standard.

Upon entry, Open Firmware initializes its own data structures and those devices necessary for its own execution. If the *script* is present and enabled, Open Firmware executes the contents of the script. Open Firmware then initializes a system-dependent set of built-in devices and locates and initializes the system's plug-in devices.

The process of initializing a device generally includes some amount of testing to ensure that the device is functioning correctly.

After the devices are initialized, Open Firmware selects devices for console input and output and displays a start-up message on the console output device. If the preceding initialization/testing steps detected any device failure conditions, messages describing those failures may be displayed on the console output device before the start-up message.

After the console is established, additional initialization and testing operations may be performed. For example, the initialization and testing of large memory regions might be deferred until after the console has been established.

Open Firmware then selects a boot device, uses it to *load* a *client program* into memory, and executes the client program. Subsequently, the client program uses Open Firmware services via the *client interface*. The sequence may be different in certain circumstances. Some examples follow:

- If a serious hardware failure is detected during an initialization/testing step, the normal start-up sequence may be aborted at a system-dependent step.
- The user may choose to abort the start-up sequence from the keyboard, perhaps to avoid automatic loading and execution of the client program.
- The system may allow the user to dynamically control the extent of various testing steps.
- The execution of the script may modify subsequent portions of the start-up sequence.
- Automatic booting may be disabled by *configuration variables*.

4.2.1 Initial self-test

Initial self-test is whatever testing is performed in the very early stages of the start-up sequence, before Open Firmware gains control of the machine.

The details of the initial self-test sequence are outside the scope of this standard but typically include some amount of “sanity checking” of the processor and the hardware that is closely connected to it. Usually, some portion of initial self-test (perhaps all of it) is written in register-based assembly language, because it must run when the machine first begins to execute instructions and must initialize and test the computer at a very low level.

The details of how this initial self-test transfers control to Open Firmware are not specified by this standard, because the state of the machine at the time of that transfer tends to be quite system-specific. Often, the transfer can be as simple as jumping to a well-known location in ROM.

4.2.2 Firmware initialization

After the machine-dependent initial self-test, Open Firmware gains control of the machine and begins to initialize itself. The precise details of this initialization depend on the implementation and the computer system, but the following steps are often included (in no particular order):

- Determining the memory configuration.
- Selecting and preparing the memory to be used for Forth and Open Firmware data structures like stacks, memory allocation pools, and *device tree* internal structures.
- Initializing various devices (e.g., MMUs, interrupt controllers, timers) that are required for the basic functioning of Forth and Open Firmware.
- Initializing a “fallback” diagnostic output device to display error reports in case an error occurs during *firmware* initialization.
- Testing configuration memory to determine if its contents are valid and, if not, resetting the *configuration variables* contained therein to their default values.

4.2.3 Start-up script evaluation

If the *script* is present and enabled, the Forth program contained therein is evaluated. Normally, after the script is evaluated, the start-up sequence continues. However, it is possible to include in the custom start-up script commands that modify the remainder of the start-up sequence in an essentially arbitrary manner.

The normal Open Firmware start-up sequence is as follows:

- a) Power-on self-test (POST)
- b) System initialization
- c) Evaluate the script (if **use-nvramrc?** is true)
- d) **probe-all** (evaluate *FCode*)
- e) **install-console**
- f) **banner**
- g) Secondary diagnostics and other system-dependent initialization
- h) Default boot (if **auto-boot?** is true)
- i) Invoke the *command interpreter* (if the preceding step returns)

If the *user interface* script feature is implemented, users may modify the portions of the start-up sequence that follow its evaluation.

4.2.4 Plug-in device probing

Probing is the process of determining the presence and characteristics of *plug-in devices*. For a *device* that uses the Open Firmware identification *methods*, the bulk of the probing process consists of the execution of an *FCode program* associated with the device.

Some devices are always attached to a system, and other plug-in devices are optional. The *device nodes* and associated drivers for permanently attached devices usually reside in the main Open Firmware system ROM, where they are permanently installed in the *device tree*. The device nodes for plug-in devices are not permanently installed in the device tree; Open Firmware must locate those devices before using them.

Probing is the process of locating plug-in devices and installing their device nodes in the device tree. This involves selecting a bus device and using it to test for the presence of devices attached to that bus.

It is possible for a plug-in device itself to be a bus device. For example, an SBus (IEEE Std 1496-1993 [B2]) plug-in device might be an adapter for a VMEbus (IEEE Std 1014-1987 [B1]) in an external card cage. Before the children of a bus device can be probed, the device itself must already exist in the device tree. For the preceding example, the SBus would have to be probed to locate the VMEbus adapter and install its device node before the VMEbus could be probed for its children.

The device tree is thus constructed incrementally, beginning from the permanent part representing *built-in devices* and proceeding outward toward the leaves of the tree.

The system default probing sequence is automatically executed during the start-up sequence, unless overridden.

Each bus device that is capable of accepting plug-in devices defines a device method for probing its subordinate devices. In addition, bus devices may define device-dependent *user interface commands* for probing.

4.2.5 Console selection

The console is the pair of input and output *devices* that Open Firmware uses for communicating with the user (for example, a keyboard and a bit-mapped display). The console devices are selected after probing, allowing the use of *plug-in devices* for the console. Input and output may be on different devices.

After probing, an input device and an output device are selected for use as the console devices. The selection process may be either automatic, based upon the set of devices found during probing, or as configured by *configuration variables*. The drivers for the selected devices are *opened*, and console input and output is directed to those devices.

Before the console is activated, any output produced by Open Firmware must either be buffered for later display or directed to a separate diagnostic output device. Whether a diagnostic output device exists, how it is chosen, and how it is accessed, are all system-dependent.

After the console is activated, subsequent Open Firmware output is displayed on the console output device by invoking its **write method**. Characters are received from the console input device by invoking its **read method**.

4.2.6 Secondary self-test

Except for those *devices* that are so closely connected to the main processor that they must be tested very early in the start-up sequence, device testing is often deferred until after the console has been activated. This has three benefits:

- Messages showing the progress of and the results from the testing may be displayed on the console.
- The user-perceived delay from the time power is applied until the console is activated is minimized (because the testing in question occurs after console activation rather than before).
- The testing may be done via *package methods*, thus providing a rich set of tools for use in developing and controlling the tests and allowing machine-independent testing of *plug-in devices*.

4.2.7 Booting

Booting is the process of *loading* and executing a *client program*, usually the operating system. Booting usually happens automatically, requiring no user intervention. From the *command interpreter*, the user can also explicitly initiate booting.

4.2.7.1 Boot process

The boot process proceeds as follows: A *device* is selected for booting by invoking the device's **load method**, a program is read from that device into memory using a protocol that depends on the type of device, and the program is then executed. The further behavior of that program may be controlled by an argument string that is made available to the program. Often, this program is a *secondary boot program*, whose purpose is to load yet another program. The secondary boot program may be capable of using additional protocols other than the protocol that Open Firmware used to load the first program. For example, Open Firmware may use the Trivial File Transfer Protocol (TFTP) to load the secondary boot program, which then uses the Network File System (NFS) protocol to load the operating system from another file.

For disk booting, Open Firmware might load the secondary boot program by reading the first few sectors from the disk, and that secondary boot program might understand the operating system's native file system structure, loading the operating system from such a file.

Typical secondary boot programs accept arguments of the following form:

filename -flags ...

where *filename* is the name of a file containing the operating system, and *-flags* is a list of options controlling the details of the start-up phase of either the secondary boot program, the operating system or both. However, from Open Firmware's point of view the boot arguments are an opaque string that is passed uninterpreted to the boot program. The boot arguments are made available through the *client interface*.

The *device path* of the boot device is also available to client programs, so they may determine the device from which they were booted.

4.2.7.2 Boot protocol

The protocol used to *load* the first *client program* depends on the type of *device*. For example, the first stage disk boot might read a fixed number of blocks from the beginning of the disk. The first stage tape boot might read a particular tape file.

4.2.8 Interrupting start-up

The provisions for interrupting the start-up sequence by user command are implementation-dependent. The typical *methods* for doing so are as follows:

- The *device driver* for the console input device may periodically test for the presence of some event that means the user wants to interrupt the current operation. Typically, that event is a particular character or a particular combination of keys simultaneously depressed. The usual response to such a user request is to abort the current operation and invoke the *command interpreter*.

The effectiveness of this method depends on the presence of a timer-driven interrupt facility (the “alarm” function) to periodically invoke the console input driver so that it may test for the user-interrupt request regardless of what other operations may be occurring. In most implementations, it is inappropriate to interrupt certain portions of the start-up sequence. Those portions may be guarded by disabling the timer interrupt while they are executing.

- Test sequences that may take a long time to finish, such as memory tests, may choose to explicitly check for a user-interrupt request at certain times. The usual response is to skip the remainder of the test, but thence to return to the normal start-up sequence so that it may proceed.

4.3 Path resolution

This section defines the process of resolving a *device path* given by a device-specifier. There are three contexts in which this can occur:

- **find-device.** In this context, the intention is to locate the named *device node* and select it as the *active package* without any other side effects.
- **open-dev.** In this context, the intention is to *open* every node named in the path by executing its **open** method, thereby creating an instance chain, and to return the *ihandle* of the node at the tail end of the chain (the node farthest from the root node).
- **execute-device-method.** In this context, the intention is to open every node named in the path *except for the last node*. An instance chain is created, including an instance for the last node, but instead of executing that last node's **open** method, a different method, given as an argument, is executed. Then the open instances are *closed* and the instance chain is destroyed.

The overall structure of the path resolution process is the same in all three contexts. This description shows it as one process with conditional tests at places where the details are context-dependent. However, it need not be implemented that way; for example, each context could be implemented separately.

The process is described in English as a set of procedures, each consisting of steps that are generally executed in order, with the scope of conditional tests shown by indentation and looping structures shown by labels and “go to”

lines. It makes liberal use of variable names to identify intermediate data items. The scope of such variables is “global” with respect to the procedures. The use of these variable names does not imply that an implementation must or should use such variables; they are used solely for descriptive purposes. Similarly, the description of the process in terms of procedures does not imply that the implementation should be so structured; the separate procedures were used in the description so that the top-level description would not be unwieldy.

The following notation describes the parsing of pathnames into component parts:

$\text{left-split}(\text{string}, “x”) \rightarrow \text{initial}, \text{remainder}$

String, *initial*, and *remainder* are the names of string variables, and “x” is a character.

Left-split divides *string* into two disjoint substrings, setting *initial* to the portion of *string* before the first occurrence of the character “x”, and *remainder* to the portion of *string* following the first occurrence of the character “x”. Neither *initial* nor *remainder* contains that first occurrence of “x”, although *remainder* may contain other later occurrences of that character. If *string* does not contain the character “x”, *initial* is set to *string* in its entirety, and *remainder* is set to the empty string.

$\text{right-split}(\text{string}, “x”) \rightarrow \text{initial}, \text{remainder}$

Right-split is similar to left-split, except that the division of the string occurs around the last occurrence of the character “x”, rather than the first.

The use of the preceding notation does not necessarily imply the existence of functions named left-split and right-split; it is simply a notational convention. (This standard does define a function **left-parse-string** whose semantics are very similar to left-split, but the details of returning the results are somewhat different.)

In searching for a matching node, the order in which the various *child nodes* are considered is unspecified. At the implementation's discretion, if no match is found among the children, the search may be widened to include the children's children, recursively to any depth.

In the following algorithmic description (4.3.1 through 4.3.5), the text enclosed in boxes is commentary describing the intention of the algorithm. The text outside of the boxes is definitive.

4.3.1 Path resolution procedure (top level procedure)

If the pathname does not begin with “/”, and its first node name component is an alias, replace the alias with its expansion.

- a) If PATH_NAME does not begin with the “/” character,
 - 1) Left-split(PATH_NAME, “/”) -> HEAD, TAIL.
 - 2) Left-split(HEAD, “:”) -> ALIAS_NAME, ALIAS_ARGS.
 - 3) If ALIAS_NAME matches a defined alias,
 - i) Replace ALIAS_NAME with its alias value.
 - ii) If ALIAS_ARGS is not empty:
 - a) Right-split(ALIAS_NAME, “/”) -> ALIAS_HEAD, ALIAS_TAIL.

- b) Right-split(ALIAS_TAIL, “:”) -> ALIAS_TAIL, DEAD_ARGS.
 - c) If ALIAS_HEAD is not empty,
Concatenate(ALIAS_HEAD, “/”, ALIAS_TAIL) -> ALIAS_TAIL.
 - d) Concatenate(ALIAS_TAIL, “:”, ALIAS_ARGS) -> ALIAS_NAME.
- iii) If TAIL is empty, replace PATH_NAME with ALIAS_NAME.
- iv) Otherwise (when TAIL is not empty),
Concatenate(ALIAS_NAME, “/”, TAIL) -> PATH_NAME.

If the pathname, after possible alias expansion, begins with “/”, begin the search at the root node. Otherwise, begin at the active package.

- b) Otherwise (when PATH_NAME begins with the “/” character),
- 1) Remove the “/” from PATH_NAME.
 - 2) Set the *active package* to the root node.
- c) If there is no active package, exit this procedure, returning **false**.

Begin the creation of an instance chain.

*NOTE—If, at this step, the active package is not the root node and we are in **open-dev** or **execute-device-method** contexts, the instance chain that results from the path resolution process may be incomplete.*

- d) Set the temporary variable PARENT-INSTANCE to zero, ARGUMENTS and UNIT_ADDR to empty strings.

This is the beginning of a loop whose body is executed once for each node name of the pathname.

- e) If PATH_NAME is empty, go to step m).

Open the node if that is appropriate.

- f) If in **execute-device-method** or **open-dev** context,
- 1) Create a new linked instance using the procedure described in 4.3.2.
 - 2) Execute the node’s **open** method.

Parse the next node name into its driver name, unit address, and device argument components.

- g) Left-split(PATH_NAME, “/”) -> COMPONENT, PATH_NAME.
- h) Left-split(COMPONENT, “:”) -> NODE_ADDR, ARGUMENTS.
- i) Left-split(NODE_ADDR, “@”) -> NODE_NAME, UNIT_ADDR.
- j) Search for a matching child node using the procedure described in 4.3.3.
- k) If the search succeeds,

The following optional step creates well-formed instance chains (i.e., with no missing components) by opening intermediate nodes that were not explicitly named in the pathname.

- 1) (Optional.) If in **execute-device-method** context or **open-dev** context, traverse the path from the *active package* to the *child node* found in the search step; at each node in that path, exclusive of the two endpoints,
 - i) Set active package to the node in question.
 - ii) Create a new linked instance using the procedure described in 4.3.2, with ARGUMENTS and UNIT_ADDR temporarily set to the empty string.
 - iii) Execute the node's **open** method.

Move to the matching node.

- 2) Set the active package to the child node found in step j).

Go back to the beginning of the loop to handle further pathname components.

- 3) Go back to step e).

On a failing search, clean up any resources that have been allocated so far, then exit.

- l) Otherwise, (when the search fails):
 - 1) Close and destroy any instances that were created during this procedure, closing more recently created instances first.
 - 2) Restore the *current instance* to the instance that was current prior to beginning this procedure.
 - 3) Restore the active package to the package that was active prior to beginning this procedure.
 - 4) If in **open-dev** context or **execute-device-method** context, exit from this procedure, returning **false**.
 - 5) Otherwise (when in **find-device** context), exit from this procedure by **throwing** a nonzero code of unspecified value.

At this point, the final node name has been selected.

- m) If in **open-dev** context,

Open the final node, thus completing the instance chain, and return its ihandle.

- 1) Create a new linked instance using the procedure described in 4.3.2.
- 2) Execute the node's **open** method.
- 3) Restore the current instance to the instance that was current prior to beginning this procedure.
- 4) Restore the active package to the package that was active prior to beginning this procedure.
- 5) Exit from this procedure, returning the *ihandle* of the instance created in step m1).

n) Otherwise, if in **execute-device-method** context,

Complete the instance chain, execute the desired method, clean up, and exit.

- 1) Create a new linked instance using the procedure described in 4.3.2.
 - 2) Attempt to execute the method whose name was given as the *method* argument to **execute-device-method**, guarded by a **catch**.
 - 3) Destroy the current instance.
 - 4) Close and destroy any parent instances that were created during this procedure, closing more recently created instances first.
 - 5) Restore the current instance to the instance that was current prior to beginning this procedure.
 - 6) Restore the active package to the package that was active prior to beginning this procedure.
 - 7) If the method execution in step n2) succeeds (i.e., the method exists and did not **throw** an error), exit from this procedure, returning the value that resulted from the execution of the named method and **true**.
 - 8) Otherwise (when the method execution in step n2) fails), exit from this procedure, returning **false**.
- o) Otherwise (when in **find-device** context), exit from this procedure, leaving the active package set to the node that was located by this procedure.

4.3.2 Create new linked instance procedure

Create a new instance, add it to the instance chain, and set its various fields (used several places by “Path Resolution” procedure).

- a) Create a new *instance* of the *active package* and make it the *current instance*.
- b) Set the instance's **my-args** field to ARGUMENTS.
- c) Set its **my-parent** field to PARENT-INSTANCE.
- d) Set PARENT-INSTANCE to the newly created instance.
- e) Set the instance's **my-unit** field as follows:
 - 1) If UNIT_ADDR is empty,
 - i) If the *active package* has a “**reg**” property, set **my-unit** to the physical address in the first component of the “**reg**” property value.
 - ii) Otherwise, set **my-unit** to 0, ... 0.
 - 2) Otherwise, set **my-unit** to UNIT_PHYS.
- f) Exit this procedure.

4.3.3 Search for matching child node procedure

Search for a node that matches the given driver name and unit address, searching first the direct children of the active package, and then, optionally, deeper levels of the tree (used by “Path Resolution” procedure).

- a) If UNIT_ADDR is not empty,

Initialize the unit search for the active package by converting the unit address string to its canonical numerical form.

- 1) If the *active package* has a **decode-unit** method, execute the active package’s **decode-unit** method with UNIT_ADDR as the argument and set UNIT_PHYS to the result.
 - 2) Otherwise, return FAILURE.
- b) Search for a matching node among the direct children of the active package, first using the exact match criteria described in 4.3.4 and, if no exact match is found, then using the wild card match criteria described in 4.3.5.
- c) If a match is found among those direct children, return SUCCESS.

This optional step allows nodes to be located even if some intermediate nodes were omitted from the pathname.

- d) (Optional.) Otherwise, (when no matching node is found among those direct children), repeat the following steps for each of those children:
- 1) Set the *active package* to a *child node*.
 - 2) Recursively execute this “Search for Matching Child Node” procedure until either a matching node is found or all nodes in this subtree have been searched.
 - 3) Restore the active package to the node that was active at the beginning of the process.
 - 4) If a match is found, return SUCCESS.
- e) Return FAILURE.

4.3.4 Exact Match criteria

Under the Exact Match criteria, the node must match the driver name and unit address components if both are given in the pathname. Otherwise, the node must match whichever component is given (used by “Search for Matching Child Node” procedure).

- a) If NODE_NAME is not empty,
- 1) If the test node has no “**name**” property, return FAILURE.
 - 2) If the value of the “**name**” property does not match NODE_NAME, according to the criteria described in 4.3.6, return FAILURE.
- b) If UNIT_ADDR is not empty,
- 1) If the test node has no “**reg**” property, return FAILURE.

- 2) If the physical address in the first component of the value of the “**reg**” property does not match UNIT_PHYS, return FAILURE.
- c) If both NODE_NAME and UNIT_ADDR are empty, return FAILURE.
- d) Return SUCCESS.

4.3.5 Wildcard Match criteria

*Under the Wildcard Match criteria, the node must match the driver name component if it is given in the pathname, and the node must have no “**reg**” property (used by “Search for Matching Child Node” procedure).*

- a) If NODE_NAME is not empty,
 - 1) If the test node has no “**name**” property, return FAILURE.
 - 2) If the value of the “**name**” property does not match NODE_NAME, according to the criteria described in 4.3.6, return FAILURE.
- b) If the test node has a “**reg**” property, return FAILURE.
- c) If both NODE_NAME and UNIT_ADDR are empty, return FAILURE.
- d) Return SUCCESS.

4.3.6 Node Name Match criteria

The Node Name Match criteria allows the “manufacturer name” portion of the node name to be optionally omitted from the pathname.

- a) If NODE_NAME contains a “,”,
 - 1) If the NODE_NAME string is the same as the entire string value of the “**name**” property, return SUCCESS
 - 2) Otherwise return FAILURE.
- b) If NODE_NAME does not contain a “,”,
 - 1) If the NODE_NAME string is the same as the entire string value of the “**name**” property, return SUCCESS.
 - 2) If the NODE_NAME string is the same as the string value of the portion of the “**name**” property following its first “,”, return SUCCESS.
 - 3) Otherwise return FAILURE.